МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені ВОЛОДИМИРА ДАЛЯ

Рогоза М.Є.
Рамазанов С.К.
Велігура А.В.
Танченко С.М.

ОСНОВИ ІНФОРМАТИКИ ТА ТЕХНОЛОГІЙ ПРОГРАМУВАННЯ

Навчальний посібник

Луганськ 2012

УДК 519,2,621,02.(07)
Р

Рекомендовано Міністерством освіти і науки, молоді та спорту України як навчальний посібник для студентів вищих навчальних закладів, які навчаються за освітньо-професійною програмою баралавра із спеціальності «Економічна кібернетика»
(*лист № 1/11-11673 від 16.07.2012 р)*

Відповідальний редактор :
Рамазанов С.К., доктор економіч них наук, доктор технічних наук, професор, завідувач кафедри економічної кібернетики Східноукраїнського національного університету імені Володимира Даля

Рецензенти:

Клебенова Т.С. – *д.е.н., професор, завідуюча кафедрою економічної кібернетики Харківського національного економічного університету*

Порохня В.М. - *д.е.н., професор, проректор-директор гуманітарного університету "Запорізький інститут державного муніципального управління"*

Лєві Л.І. - *д.т.н., професор, завідуючий кафедрою фізико-математичних дисциплін Луганського національного аграрного університету*

Рогоза М.Є.,
Р **Основи інформатики та технологій програмування:** навчальний посібник / Рогоза М.Є., Рамазанов С.К., Велігура А.В., Танченко С. М. - Луганськ: Вид-во СНУ ім. В.Даля, 2012. - 568 с.: табл. 57, іл. 135, бібліогр. 25 назв.

ISBN 978-966-590-936-1

Навчальний посібник містить теоретичні та практичні аспекти для опанування дисциплін «Інформатика» та «Технології програмування». На прикладі середовища розробки Microsoft Visual Basic представлені прийоми візуального створення програмних продуктів, розглянуті базові алгоритми та структури даних. Особлива увага приділена алгоритмам, що використовуються у економіко-математичному моделюванні – сортування, пошуку, обробки динамічних структур даних, оптимізаційним алгоритмам пошуку у деревах рішень.

Посібник призначений для студентів та аспірантів, що навчаються за напрямками: «Економічна кібернетика», «Маркетинг», «Менеджмент» та вивчають сучасні інформаційні технології.

УДК ББК

ISBN 978-966-590-936-1

ПЕРЕДМОВА

Навчальний посібник призначений для вивчення Навчальний посібник призначений для вивчення дисципліни «Технології програмування», яка є основною для формування практичних знань і навичок бакалаврів напрямку «економічна кібернетика». Даний спецкурс спирається на отримані теоретичні знання студентами з дисципліни «Інформатика» і призначений для формування у бакалаврів спеціальності «економічна кібернетика» науково обґрунтованого підходу до алгоритмізації економіко-математичних задач та розробки програмних продуктів з використанням сучасних технологій.

Коло питань, що розглядаються в навчальному посібнику, включає в себе основні теми, які полягають в розкритті понять про організацію і принципи функціонування комп'ютерних систем, алгоритмічні мови та системи програмування, основні принципи розробки алгоритмів і програм, технології створення програмних продуктів з використанням спеціалізованих середовищ візуального програмування. У навчальному посібнику розглянуто мову програмування Visual Basic, наведено приклади реалізації базових алгоритмів і структур даних. Виділене коло питань визначає структуру посібника, який складається з 14 розділів.

У першому розділі розглядаються особливості побудови і функціонування сучасних комп'ютерних систем, визначаються основні апаратні і програмні компоненти, наведено відомості про особливості операційних систем сучасних персональних комп'ютерів.

Другий розділ присвячений алгоритмічним мовам та системам програмування. Розглянуто способи перекладу програм з мови програмування на мову мікропроцесору, засоби створення програм, розглянуто особливості використання інтегрованих середовищ розробки програмних засобів, середовищ бистрого проектування, надано відомості про основні системи програмування.

Етапи рішення задач на ЕОМ, форми запису алгоритмів, мова блок-схем та особливості структурного підходу до розробки програм розглядаються у третьому розділі.

Четвертий розділ присвячений вивченню властивостей алгоритмів, розглядаються поняття складності алгоритмів, методики пошуку складних частин, особливості рекурсивних алгоритмів, функції оцінки порядку складності, фактори, що впливають на швидкість виконання, затримки, які викликає звернення до файлу підкачки, та інші фактори, що впливають на продуктивність роботи алгоритмів.

П'ятий розділ присвячений опису системи швидкої розробки додатків – Visual Basic. Наведено відомості про основні групи операторів, типи даних та змінні, константи, масиви. Розглянути засоби реалізації класів об’єктів, операторів оголошення, управління ходом виконання програми, процедур і функцій. Усі пояснення проілюстровано практичними прикладами.

Шостий розділ містить відомості про розробку користувацького інтерфейсу. Розглянуто переваги і недоліки усіх типів користувацьких інтерфейсів, їх основні елементи.

У сьомому розділі наведено відомості про стандартні елементи управління, які використовуються при розробці програм.

Восьмий розділ присвячений питанням роботи з файлами та організації друку результатів роботи програми. Розглянуто особливості роботи з файлами послідовного, довільного доступу і з бінарними файлами.

У де’вятому розділі розглянуто особливості реалізації динамічних структур даних. Наведено відомості про побудову упорядкованих та неупорядкованих списків, різновиди зв’язних списків (циклічні, двозв’язні, потоки).

В десятому розділі описано способи реалізації стеків, черг (циклічних, на основі зв’язних списків, пріоритетних, багато поточних).

Одинадцятий розділ містить відомості про реалізацію дерев в Visual Basic. Описано способи подання дерев (повні вузли, списки нащадків, подання зв’язків нумерацією, повні дерева), обходу дерев. Для упорядкованих дерев наведено приклади додавання, видалення елементів та обходу дерева. Для АВЛ-дерев описано алгоритми обертання, вставки і видалення вузлів.

У дванадцятому розділі наведено приклади використання дерев у задачах, що пов’язані з прийняттям рішень. Розглянуто питання пошуку у деревах рішень (метод гілок та межі, сходження на пагорб, найменшої вартості, збалансованого прибутку, послідовного наближення та ін.). Наведено вирішення задач про здійснимість, розбивку, пошуку Гамільтонового шляху, комівояжера, пожежні депо.

У тринадцятому розділі розглянуто алгоритми сортування. Зокрема, детально описано особливості реалізації сортування вибором, вставкою, бульбашкового сортування, сортування злиттям, швидкого сортування, сортування підрахунком, блокового сортування. Окремо розглядається питання рандомізації.

Чотирнадцятий розділ присвячено методам вирішення задач пошуку. Розглянуто алгоритми повного перебору, двійкового пошуку,

інтерполяційного пошуку. Окремо виділено питання пошуку строкових даних (пошук що стежить, інтероляціний пошук, що стежить).

При викладенні матеріалу велика увага приділена практичним аспектам розробки алгоритмів і програм. Всі теоретичні положення з алгоритмізації та програмування ілюструються численною кількістю програмних реалізацій. Усі приведені у посібнику програми детально прокоментовані. У додатках наведено коди програм, що реалізують алгоритми, описані у розділах навчального посібника.

У кожному розділі підручника наведені контрольні запитання для самоперевірки та задачі для самостійного виконання.

Посібник розроблений колективом авторів, так розділи 1, 4 - підготував проф. Рогоза М.Є., розділи 2-3, 12-14 Рамазанов С.К., розділи 5-7, 10-11 доц. Велігура А. В., розділ 8-9, 13 – ас. Танченко С.М.

ЗМІСТ

Розділ І

ОРГАНІЗАЦІЯ ФУНКЦІОНУВАННЯ КОМП'ЮТЕРНИХ СИСТЕМ

Компоненти комп'ютерної системи

Щоб краще зрозуміти місце й роль операційної системи в процесі обчислень, розглянемо комп'ютерну систему в цілому. Вона складається з наступних компонентів:

1. **Апаратура (hardware)** комп'ютера, основні частини якої – **центральний процесор (Central Processor Unit - CPU**), що виконує **команди (інструкції)** комп'ютера; **пам'ять (memory), що** зберігають дані й програми, і **пристрої введення- виведення, або зовнішні пристрої (input-output devices, I/O devices**), що забезпечують введення інформації в комп'ютер і виведення результатів роботи програм у формі, яка сприймається користувачем-людиною або іншими програмами. Часто на програмістському сленгі апаратуру називають "залізом".
2. **Операційна система (operating system)** – системне програмне забезпечення, що управляє використанням апаратури комп'ютера різними програмами й користувачами.
3. **Прикладне програмне забезпечення (applications software)** – програми, призначені для рішення різних класів завдань. До них відносяться, зокрема, **компілятори**, що забезпечують трансляцію програм з мов програмування, наприклад, C++, у машинний **код** (команди); **системи управління базами даних (СУБД); графічні бібліотеки**, **ігрові програми**, **офісні програми**. Прикладне програмне забезпечення утворює наступний, більш високий рівень, у порівнянні з операційною системою, і дозволяє вирішувати на комп'ютері різні прикладні й повсякденні завдання.
4. **Користувачі (users)** – люди й інші комп'ютери. Віднесення користувача-людини до компонентів комп'ютерної системи -

зовсім не жарт, а реальність: будь-який користувач фактично стає частиною обчислювальної системи в процесі своєї роботи на комп'ютері, тому що повинен підкорятися певним строгим правилам, порушення яких приведе до помилок або неможливості використання комп'ютера, і виконувати великий обсяг типових рутинних дій – майже як сам комп'ютер. Одна з важливих функцій ОС саме й полягає в тому, щоб позбавити користувача від більшої частини такої рутинної роботи (наприклад, резервного копіювання файлів) і дозволити йому зосередитися на роботі творчій. Інші комп'ютери в мережі також можуть відігравати роль користувачів (**клієнтів**) стосовно комп'ютера, який виступає в ролі **сервера**, що, наприклад, використовується для зберігання файлів або виконання великих програм.

Девізом фірми Sun Microsystems ще в 1982 р., при її створенні, став афоризм "**The network is the computer**" (Мережа – це комп'ютер). Цю істину варто пам'ятати всім користувачам комп'ютерів та операційних систем і ширше використовувати можливості комп'ютерних мереж, розподіляючи різні функції між комп'ютерами, що складають мережу (або **хостами – hosts**, як на комп'ютерному сленгі прийнято називати комп'ютери в мережі). Ізольований від мережі комп'ютер нині - це "кам'яний вік". Звідси - нерозривний зв'язок операційних систем і мереж.

Останнім часом в літературних джерелах можна зустріти поряд з терміном *управління підприємством* термін *менеджмент підприємства*. Запозичені з англійської слова *менеджмент* і *менеджер* (management – управління, керівництво; manager – керівник, управляючий) швидко набули поширення і визнання як у фахівців-економістів, так і у повсякденному спілкуванні. Тому сьогодні поняття *управління* і *менеджмент* частіше за все використовуються як тотожні.

Загальна картина функціонування комп'ютерної системи

Користувачам комп'ютера доступні верхні рівні програмного забезпечення - системні й прикладні програми (наприклад, компілятори, текстові редактори, системи керування базами даних). Ці програми взаємодіють із операційною системою, що, у свою чергу, управляє роботою комп'ютера.

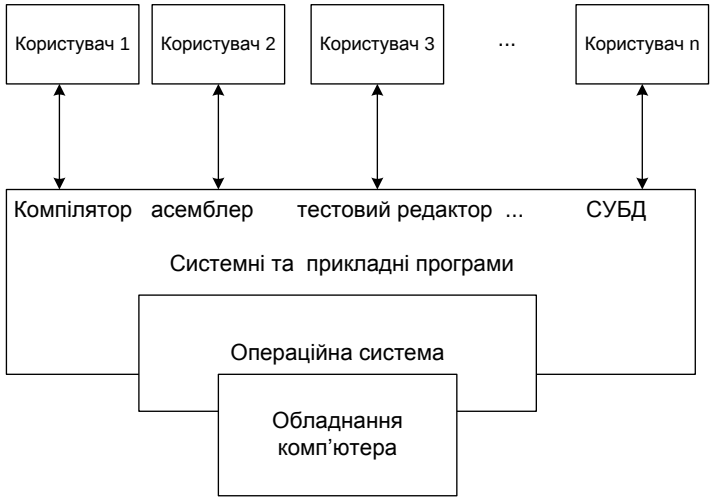


Рис. 1.1. Загальна картина функціонування комп'ютерної системи

Класифікація комп'ютерних систем

Для того, щоб уявити собі розмаїтість і масштабованість операційних систем, розглянемо насамперед класифікацію сучасних комп'ютерних систем, для яких розробляються й використовуються ОС – від **суперкомп'ютерів** до **мобільних пристроїв**, - і підсумуємо вимоги до ОС для цих класів комп'ютерів.

Суперкомп'ютери (super-computers) – потужні багатопроцесорні комп'ютери, найбільш сучасні, які мають продуктивність до декількох **petaflops** (1015 дійсних операцій у секунду; абревіатура **flops** розшифровується як **floating-point operations per second**). Наприклад, суперкомп'ютер "Ломоносов", встановлений у МДУ. Суперкомп'ютери використовуються для обчислень, що вимагають великих обчислювальних потужностей, надвисокої продуктивності й великого обсягу пам'яті. У реальній практиці це насамперед задачі моделювання - наприклад, моделювання клімату в регіоні й прогнозування на основі побудованої моделі погоди в даному регіоні на найближчі дні. Особливістю суперкомп'ютерів є їхня паралельна архітектура - як правило, всі вони є багатопроцесорні. Відповідно, ОС для суперкомп'ютерів повинні підтримувати розпаралелювання рішення

завдань і синхронізацію паралельних процесів, що одночасно вирішують підзадачи деякої програми.

Багатоцільові комп'ютери, або **комп'ютери загального призначення (mainframes)** – традиційна історична назва для комп'ютерів, розповсюджених в 1950-х – 1970-х рр., ще до епохи загального поширення персональних комп'ютерів. Саме для mainframe-комп'ютерів створювалися перші ОС. Типові приклади таких комп'ютерів: IBM 360/370; з вітчизняних – М-220, БЭСМ-6. На таких комп'ютерах вирішувалися всі необхідні завдання – від розрахунку зарплати співробітників в організації до розрахунку траєкторій космічних ракет. Подібний комп'ютер виглядав досить незграбно й громіздко й міг займати цілий великий зал. Згадаєте, наприклад, величезний комп'ютер HAL на космічному кораблі у фантастичному фільмі 1960-х рр. Стэнли Кубрика "Космічна Одисея 2001 р." Але ніякі фантасти не змогли передбачати прогресу комп'ютерної техніки XXI століття – насамперед, того, що потужний комп'ютер не буде займати цілу кімнату, а буде міститися в невеликому ящику. Параметри ранніх mainframe-комп'ютерів були досить скромними: швидкодія - кілька тисяч операцій у секунду, оперативна пам'ять – кілька тисяч комірок (слів). Недостатньо зручним був користувальницький інтерфейс (інтерактивна взаємодія з комп'ютерами була реалізована набагато пізніше, в 1960-х рр.). Проте, на таких комп'ютерах вирішувалися досить серйозні завдання оборонного й космічного призначення. З появою персональних і портативних комп'ютерів класичні mainframe-комп'ютери відійшли в минуле. Однак варто підкреслити, що в саме в операційних системах для mainframe-комп'ютерів були реалізовані всі основні методи й алгоритми, що розглянуті в даному посібнику, і які згодом були використані в ОС для персональних, кишенькових комп'ютерів і **мобільних пристроїв.**

Кластери комп'ютерів (computer clusters) – групи комп'ютерів, фізично розташовані поруч і з'єднані один з одним високошвидкісними шинами й лініями зв'язку. Кластери комп'ютерів використовуються для високопродуктивних паралельних обчислень. Найбільш відомі у світі комп'ютерні кластери, розташовані в дослідницькому центрі CERN (Швейцарія) - тому самому, де перебуває великий адронний коллайдер. Як правило, комп'ютерні кластери розташовуються в дослідницьких інститутах і в університетах. Операційна система для кластерів повинна, крім загальних можливостей, надавати засоби для конфігурування кластера, керування комп'ютерами (процесорами), що його складають, розпаралелювання рішення завдань між комп'ютерами

кластера й моніторингу кластерної комп'ютерної системи. Прикладами таких ОС є ОС фірми Microsoft - Windows 2003 for clusters; Windows 2008 High-Performance Computing (HPC).

Настільні комп'ютери (desktops) – це найпоширеніші в наш час комп'ютери, якими користуються вдома або на роботі всі люди, від школярів і студентів до хатніх господарок. Такий комп'ютер розміщається на робочому столі й складається з монітора, системного блоку, клавіатури й миші.

Параметри сучасного (2010 р.) **настільного комп'ютера**, найбільш прийнятні для використання сучасних ОС: швидкодія процесора 1 – 3 ГГц, оперативна пам'ять – 1 – 8 гігабайт і більше, обсяг жорсткого диска (hard disk drive – HDD) – 200 Гб – 1 Тб і більше (1 терабайт, Тб = 1024 Гб). Вся розмаїтість сучасних операційних систем (Windows, Linux і ін.) – до послуг користувачів **настільних комп'ютерів**.

При необхідності на **настільному комп'ютері** можна встановити дві або більше операційних системи, розділивши його дискову пам'ять на кілька розділів (partitions) і встановивши на кожний з них свою операційну систему, так що при включенні комп'ютера користувачеві надається стартове меню, з якого він вибирає потрібну операційну систему для завантаження.

Портативні комп'ютери (laptops, notebooks – дослівно "комп'ютери, що містяться на колінах"; " комп'ютери-зошити") – це мініатюрні комп'ютери, які по своїм параметрам не уступають настільним, але по своїм розмірам вільно містяться в невелику сумку або рюкзак, або, наприклад, на колінах користувача, що летить у літаку у відрядження й не бажає втрачати час даром. Ноутбуки коштують звичайно в кілька разів дорожче, ніж настільні комп'ютери з аналогічними характеристиками. На **ноутбуках** використовуються ті ж операційні системи, що й для настільних комп'ютерів (наприклад, Windows або MacOS). Характерними рисами **портативних комп'ютерів** є всілякі убудовані порти й адаптери для бездротового зв'язку:

Wi-Fi (офіційно IEEE 802.11) – вид радіозв'язку, що дозволяє працювати в бездротовій мережі із продуктивністю 10-100 мегабіт у секунду (використовується звичайно на конференціях, у готелях, на вокзалах, аеропортах – тобто в зоні радіусом у кілька сотень метрів від джерела прийому-передачі);

Bluetooth – також радіозв'язок на більш коротких відстанях (10 – 100 м для Bluetooth 3.0), використовувана для взаємодії комп'ютера з мобільним телефоном, навушниками, плеєром і ін.

Зовнішні пристрої (додаткові жорсткі диски, принтери, іноді навіть DVD-ROM) підключаються до ноутбука через порти USB. Ще років 10 назад на ноутбуках активно використовувалися **інфрачервоні порти (IrDA**), які, однак, незручні, тому що вимагають присутності "відповідного" IrDA – порту іншого пристрою на відстані 20-30 см від порту ноутбука, при відсутності між ними перешкод.

Інша характерна риса ноутбуків – це наявність кард-ридеров – портів для читання всіляких карт пам'яті, використовуваних у мобільних телефонах або цифрових фотокамерах; забезпечується також інтерфейс FireWire (офіційно – IEEE 1394) для підключення цифрової відеокамери; таким чином, ноутбуки добре пристосовані для введення, обробки й відтворення обробки мультимедійної інформації. Нині портативний комп'ютер є майже в кожного студента, він використовується для підготовки до відповіді на іспиті, або для рішення завдань практикуму, іноді прямо в університетському буфеті. Один із критичних параметрів ноутбука – час роботи його батарей без підзарядки; дуже добре, якщо цей час становить порядку 10 годин, що поки порівняно рідко зазвичай цей час становить не більше 5 годин. Популярний різновид ноутбука нині – це **нетбук** - ноутбук, призначений для роботи в мережі, звичайно менш потужний і тому більш дешевий, а також більш мініатюрний.

Кишенькові портативні комп'ютери й органайзери (КПК, handhelds, personal digital assistants – PDA) – це "іграшки для дорослих" у вигляді мініатюрного комп'ютера, що міститься на долоні або в кишені, але по своїй швидкодії іноді не уступає ноутбуку. При всій привабливості, серйозні недоліки КПК, з погляду автора, - це незручність введення інформації (доводиться користуватися паличкою-стайлусом, - адже не носити ж із собою ще й громіздку клавіатуру), а також незручність читання інформації на маленькому екрані.

Сучасні КПК мають фактично ті ж порти й адаптери, що й ноутбуки - Wi-Fi, Bluetooth, IrDA, USB. Операційні системи для КПК аналогічні ОС для ноутбуків, але все-таки враховують більше тверді обмеження КПК по обсязі оперативної пам'яті. У цей час для КПК широко використовується ОС Windows Mobile - аналог Windows для мобільних пристроїв. Донедавна була також широко поширена PalmOS для організайзерів типу PalmPilot фірми 3COM. Зрозуміло, для КПК є апаратура й програмне забезпечення для підключення до ноутбука або настільного комп'ютера з метою синхронізації даних, що забезпечує додаткову надійність.

Мобільні пристрої (mobile intelligent devices – мобільні телефони, комунікатори) – це пристрої, якими кожний з нас

користується постійно для голосового зв'язку, рідше – для запису або обробки якої-небудь інформації або для виходу в Інтернет. Найбільш важливі параметри мобільного пристрою – це якість голосового зв'язку й час автономної роботи батареї. Однак все більшого значення набувають убудовані в них цифрові фото і відеокамери.

Операційні системи для мобільних пристроїв відрізняються більшою компактністю, через більш тверді обмеження по пам'яті. Епоха домінування на ринку мобільних телефонів операційних систем типу Symbian, закінчується вони поступаються місцем більше сучасним Google Android і Microsoft Windows Mobile. Для мобільних пристроїв, як і для КПК, досить важлива характеристика ОС – це її надійність, зокрема, схоронність даних після переповнення пам'яті, що виникає, наприклад, у результаті прийому великого числа SMS-Повідомлень, інтенсивної фото- або відеозйомки. У мобільних телефонах використовується платформа ("видання") JME – **Java Micro Edition**, і будь-який мобільний телефон, що випускається от уже більше 10 років, підтримує Java. Програми на Java для мобільних телефонів називаються **мидлетами** (від абревіатури **MID – Mobile Intelligent Device**).

Комп'ютери, що носяться, (wearable computers) - для повсякденного життя досить екзотичні пристрої, однак для спеціальних застосувань (наприклад, убудовані в скафандр космонавта або в кардіостимулятор) вони життєво важливі. Зрозуміло, їхня пам'ять і швидкодія значно менше, ніж у настільних комп'ютерів, але критичним фактором є їхня надвисока надійність, а для їхніх операційних систем і іншого програмного забезпечення – мінімальний можливий **час відповіді (response time)** – інтервал, протягом якого система обробляє інформацію від датчиків, від користувача або з мережі, перевищення якого загрожує катастрофічними наслідками. Із цього погляду, ОС для **комп'ютерів, що** носяться, можна віднести **до систем реального** часу.

Розподілені системи (distributed systems) – це системи, що складаються з декількох комп'ютерів, об'єднаних у провідну або бездротову мережу. Фактично, такі нині всі комп'ютерні системи (згадаєте девіз **"Мережа – це комп'ютер"**). Всі операційні системи повинні, таким чином, підтримувати розподілений режим роботи, засоби мережевої взаємодії, високошвидкісну надійну передачу інформації через мережу.

Системи реального часу (real-time systems) – обчислювальні системи, призначені для керування різними технічними, військовими й іншими об'єктами в режимі реального часу. Характеризуються основною вимогою до апаратури й

програмного забезпечення, у тому числі до операційної системи: **неприпустимість перевищення часу відповіді** системи, тобто очікуваного часу виконання типової операції системи. Для ОС вимоги реального часу накладають досить тверді обмеження - наприклад, в основному циклі роботи системи неприпустимі переривання (тому що вони приводять до неприпустимих тимчасових витрат на їхню обробку). Системи реального часу - особлива досить серйозна й специфічна область, вивчення якої виходить за рамки даного курсу.

Наведений огляд дає деяке подання про розмаїтість комп'ютерних систем у наш час. Для кожної з них повинна бути розроблена адекватна операційна система.

Класифікація комп'ютерних архітектур

Комп'ютерні системи відрізняються між собою не тільки по своїм параметрам і своєму призначенню, але й по своїм внутрішнім архітектурним принципам. Найбільш відомі наступні підходи до архітектури комп'ютерних систем.

CISC (Complicated Instruction Set Computers – комп'ютери з ускладненою системою команд) – історично перший підхід до комп'ютерної архітектури, суть якого в тім, що в систему команд комп'ютера включаються складні по семантиці операції, що реалізують типові дії, часто використовувані при програмуванні й при реалізації мов – наприклад, виклик рекурсивних процедур і автоматичне відновлення дисплей-регістрів, групові операції пересилання рядків і масивів і ін. Типовими представниками CISC-**Комп'ютерів** були: із закордонних комп'ютерних систем – машини серії **IBM 360/370,** з вітчизняних – багатопроцесорні обчислювальні комплекси (МВК) **"Ельбрус**". В IBM 360, наприклад, була реалізована команда MVC (move characters), що виконувала пересилання масиву символів (рядка) з однієї області пам'яті в іншу, причому адреси джерела, одержувача й довжина пересилаємої стрічки задавалися в регістрах. В "Ельбрусі" був апаратно реалізований у загальному виді вхід у процедуру з передачею через стек параметрів, відновленням дисплей-регістрів, що вказують на доступні процедурі області локальних даних. Інший приклад – в "Ельбрусі" команда зчитування в стек значення по заданій адресі здійснювала автоматичний прохід "непрямого ланцюжка" заздалегідь не відомої довжини – якщо значення виявлялася також адресою, то відбувалося зчитування в стек значення по ньому й т.д., доти, поки

зчитана в стек величина не виявиться значенням, а не адресою. З одного боку, зрозуміле прагнення авторів CISC-**Архітектур** зробити апаратуру як можна більш "розумної". З іншого боку, тверде "вшиття" складних алгоритмів виконання команд в "залізо" приводило до того, що апаратура виконувала щораз деякий загальний алгоритм команди, що вимагав десятків або навіть сотень тактів процесора, але оптимізувати виконання цих команд із використанням конкретної інформації про довжину рядка, непрямого ланцюжка й т.д. можливості не було. Інший недолік CISC-**Архітектур** у тім, що подібні групові операції на час їхнього виконання фактично припиняли роботу **конвеєра (pipeline)** - реалізованої в будь-якій комп'ютерній архітектурі апаратної оптимізації, паралельного виконання декількох сусідніх команд за умови їхньої незалежності один від одного за даними.

RISC (Reduced Instruction Set Computers – комп'ютери зі спрощеною системою команд) – спрощений підхід до архітектури комп'ютерів, запропонований на початку 1980-х рр. професором Дэвидом Паттерсоном (університет Беркли, США) і його студентом Дэвидом Дитцелом (згодом – великим ученим, керівником компанії Transmeta). Приклади сімейств RISC-**Комп'ютерів**: **SPARC, MIPS, PA-RISC, PowerPC**. Принципи даного підходу: спрощення семантики команд, відсутність складних групових операцій (які можуть бути реалізовані послідовностями команд, що містять цикли); однакова довжина команд (32 біта – архітектура була розроблена розраховуючи на 32-бітові процесори); виконання арифметичних операцій тільки в регістрах і використання спеціальних команд зчитування з пам'яті в регістр і запису з регістра на згадку; відсутність спеціалізованих регістрів (наприклад, дисплей-регістрів для адресації доступних областей локальних даних у стеці); використання великого набору регістрів (**реєстрового файлу**) загального призначення – 512, 1024, 2048 регістрів і т.д., залежно від конкретної моделі процесора; передача при виклику процедур параметрів через регістри. Подібна архітектура дає широкий простір для оптимизацій, виконуваних компіляторами, що й демонструють компілятори Sun Studio розробки фірми Sun / Oracle для ОС Solaris і Linux. RISC-Архітектура дотепер використовується при розробці нових комп'ютерів.

VLIW (Very Long Instruction Word – комп'ютери із широким командним словом) – підхід до архітектури комп'ютерів, що склався в 1980-х – 1990-х рр. Основна ідея даного підходу – **статичне планування паралельних обчислень компілятором** на рівні окремих послідовностей команд і підкоманд. При даній архітектурі кожна команда є "**широкою" (long) і** містить

підкоманди, які виконуються паралельно за один машинний такт на декількох однотипних пристроях процесора – наприклад, у такому комп'ютері може бути два пристрої додавання, два логічних пристрої, два пристрої для виконання переходів і т.д. Завданням компілятора є оптимальне планування завантаження всіх цих пристроїв у кожному машинному такті й генерація таких (широких) команд, які дозволили б оптимально завантажити на кожному такті кожний із пристроїв. Достоїнством такої архітектури є можливість распаралелювання обчислень, недоліком – складність (у порівнянні з RISC-**Архітектурою**). Приклади комп'ютерів таких архітектур: із закордонних – комп'ютери Cray X/MP, Cray Y/MP і ін., розроблені комп'ютерним генієм Сіяй муром Креем (Cray) і його фірмою Cray Research; з вітчизняних – багатопроцесорний обчислювальний комплекс "Ельбрус-3".

EPIC (Explicit Parallelism Instruction Computers – комп'ютери з явним распаралелюванням) – по архітектурі аналогічні **VLIW**, але з рядом важливих удосконалень: наприклад, **спекулятивних** обчислень – паралельного виконання обох гілок умовної конструкції з обчисленням умови. Підхід зложився та використовується з 1990-х рр. Приклади процесорів даної архітектури - Intel IA-**64, AMD-64**.

Multi-core computers (багатоядерні комп'ютери) – найбільш широку популярність одержала у наш час архітектура комп'ютерів, де кожний процесор має **кілька ядер (cores)**, об'єднаних в одному кристалі й паралельно працюючих на одній і тій же загальній пам'яті, що дає широкі можливості для паралельних обчислень. Дуже відомі **багатоядерні** процесори фірми Intel (Cure 2 Duo, Dual Core і ін.), а також потужні **багатоядерні** процесори фірми Sun / Oracle: **Ultra SPARC-T1 ("Niagara")** - 16-ядерний процесор**; Ultra SPARC-T2 ("Niagara2")** – 32-ядерний процесор. Всі провідні фірми світу зайняті розробкою й випуском усе могутніших **багатоядерних** процесорів. Відповідно, творці операційних систем для таких комп'ютерів розробляють базові бібліотеки програм, що дозволяють повною мірою використовувати можливості паралельного виконання на **багатоядерних** процесорах.

Hybrid processor computers (комп'ютери з гібридними процесорами) – новий підхід до архітектури комп'ютерів, при якому процесор має **гібридну** структуру – складається з (**багатоядерного**) **центрального процесора (CPU)** і (також **багатоядерного**) **графічного процесора (GPU – Graphical Processor Unit**). Така архітектура була розроблена, у зв'язку з необхідністю паралельної обробки графічної й мультимедійної інформації, що особливо

актуально для комп'ютерних ігор, перегляду на комп'ютері високоякісного цифрового відео та ін. Гібридна архітектура є новим "інтелектуальним викликом" для розроблювачів компіляторів, яким необхідно розробити й реалізувати адекватний набір оптимизацій як для центральних, так і для графічних процесорів. Прикладами таких архітектур є нові процесори фірми AMD, а також графічні процесори серії Tesla фірми NVidia.

Основні компоненти операційної системи

Розглянемо тепер основні частини ОС.

Ядро (kernel) – низькорівнева основа будь-якої операційної системи, яка виконується апаратурою в особливому **привілейованому режимі**. Ядро завантажується у пам’ять один раз і перебуває в пам'яті **резидентно** – постійно, по одних і тих же адресах.

Підсистема управління ресурсами (resource allocator) – частина операційної системи, що управляє обчислювальними ресурсами комп'ютера - оперативною й зовнішньою пам'яттю, процесором і ін.

Управляюча програма (control program, supervisor) – підсистема ОС, що управляє виконанням інших програм і функціонуванням пристроїв вводу-виводу.

Архітектура комп'ютерної системи

Комп'ютерна система має модульну структуру. Для кожного пристрою (пам'ять, зовнішні пристрої) у системі є спеціальний пристрій управління (інакше кажучи, спеціальний процесор), який називається **контролером пристрою**. Всі модулі (центральний процесор, пам'ять і контролер пам'яті, зовнішні пристрої та їхні контролери) з'єднані між собою **системною шиною (system bus),** через яку вони обмінюються сигналами. Як ми вже знаємо роботою кожного контролера керує **драйвер -** спеціалізована низькорівнева програма, що є частиною ОС.

Ось типова структура сучасної настільної або портативної комп'ютерної системи, із вказівкою найпоширеніших типів пристроїв і їхніх характеристик.

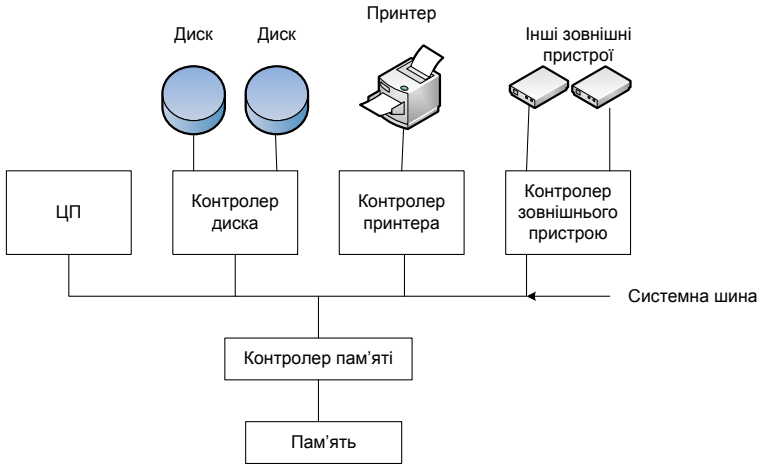


Рис. 1.2. Архітектура комп'ютерної системи

Центральний процесор – пристрій, що виконує **команди (instructions)** комп'ютерної системи. У сучасних комп'ютерах, як правило, він є **багатоядерним**, тобто має у своєму складі від 2 до 32 ядер (копій) процесора, що паралельно працюють у загальній пам'яті, або **гібридним**, що складається із центрального й графічного процесорів. Продуктивність кожного ядра – 3 – 3.2 GHz. Помітимо, що під продуктивністю розуміється в цьому випадку **тактова частота процесора (ядра)** – час виконання ним однієї найпростішої машинної команди. Однак є й інші важливі фактори, що визначають загальну продуктивність системи, - тактова частота пам'яті й системної шини. Фактично підсумкову продуктивність системи можна оцінити по самій повільній із цих частин системи (звичайно це системна шина). Ці характеристики необхідно брати до уваги при виборі й покупці комп'ютера.

Оперативна (основна) пам'ять, або просто **пам'ять** – пристрій, що зберігає оброблювані дані. Обсяг пам'яті - 1 - 16 гігабайт і більше; менший обсяг пам'яті використовувати не рекомендується, тому що це може привести до значного вповільнення системи. Тактова частота пам'яті - 667 MHz - 1.5 GHz.

Системна шина – пристрій, до якого приєднані всі модулі комп'ютера й через який вони обмінюються сигналами, наприклад, про переривання. Тактова частота шини – 1 – 1.5 GHz (це і є фактично якась сумарна продуктивність системи). Звичайно використовується шина типу **PCI (Personal Computer Interface).** До неї можуть бути

приєднані процесор, пам'ять, диски, принтер, модем і інші зовнішні пристрої.

Порти – пристрої з роз'ємами для підключення до комп'ютера зовнішніх пристроїв. Кожний порт має свій контролер (і, відповідно, свій драйвер).

Найчастіше використовується **порт USB (Universal Serial Bus),** з характерним плоским роз'ємом, розміром порядку 1 см, із зображенням тризубця. До портів USB можуть підключатися більшість видів пристроїв, причому для цього не потрібно попередньо відключати комп'ютер і підключати пристрій, що дуже зручно. Є кілька стандартів USB з різною швидкодією. Найпоширеніший нині стандарт USB 2.0, що забезпечує швидкодію порту 240 – 260 мегабіт у секунду. Для порівняння, попередній стандарт – USB 1.0 – забезпечував лише 10 – 12 мегабіт у секунду. Розпізнати тип USB-Порту на Вашім комп'ютері можна, якщо вивести інформацію про пристрій; в Windows: **Мій комп'ютер / (права кнопка миші) Властивості / Устаткування / Диспетчер пристроїв / Пристрою USB**. При цьому контролер порту USB 2.0 буде позначений як **розширений (enhanced).** Якщо це не так, Вам необхідно модернізувати порти USB або сам комп'ютер, інакше при запису на флешку Вам доведеться чекати в 20 разів довше (!). Існують також "перехідники" USB 1.0 -> USB 2.0. Новітній стандарт USB 3.0, реалізація якого тільки почалася, забезпечить швидкодію не менш 1 гигабіта в секунду. До порту USB можна підключати клавіатуру, мишу, принтери, сканери, зовнішні жорсткі диски, флешки й навіть TV-**Тюнери** - пристрої для прийому телевізійного сигналу з антени й показу телевізійного зображення на комп'ютері. Рекомендується кожний пристрій підключати завжди до того самого порту USB, інакше для деяких пристроїв (наприклад, того ж TV-Тюнера) можуть виникнути проблеми.

Порти COM (communication ports) – порти для підключення різних комунікаційних пристроїв, наприклад, **модемів** – пристроїв для виходу в Інтернет і передачі інформації по аналоговій або цифровій телефонній лінії. Більш стара назва стандарту COM-Порту – **RS-232**. У комп'ютерах 10-15 – літньої давнини до COM-Порту часто підключалася мишка (зараз вона, зрозуміло, підключається через USB). Роз'єми COM-Портів мають два формати – "великий" (з 25 контактами - **pins**) і "малий" (з 9 контактами). У сучасних комп'ютерах часто роз'єми COM-Порти відсутні, але операційна система, за традицією, імітує наявність у системі **віртуальних COM-Портів** – уявлюваних COM-Портів, які ОС як би інсталює в систему при установці, наприклад, драйверів для взаємодії через Bluetooth або

через кабель комп'ютера з мобільним пристроєм. При цьому фізично мобільний телефон або органайзер може бути підключений до порту USB (або з'єднаний з комп'ютером без дротяним зв'язком), але однаково для взаємодії з ним ОС використовує віртуальний COM-Порт, звичайно з більшим номером (наприклад, 10 або 15).COM-Порт інакше називають **послідовним портом (serial port),** тому що, з погляду ОС і драйверів, COM-Порт – це символьний пристрій послідовної дії.

Порт LPT (від line printer), або **паралельний порт** – це нині вже застарілий вид порту для підключення принтера або сканера, з товстим у перетині кабелем і великим роз'ємом. Всі нові моделі принтерів і сканерів працюють через USB-Порти. Однак іноді доводиться вирішувати завдання підключення до нового комп'ютера старого принтера. Якщо на комп'ютері немає LPT-Порту, доводиться купувати спеціальний перехідник, що підключається до USB або інших портів. Однак і тут можливий сюрприз – роз'єм LPT-Порту має трохи не сумісні один з одним модифікації. Незручність LPT-Порту в тім, що він вимагає попередньо вивантажити ОС і виключити принтер, і тільки після цього виконувати приєднання до комп'ютера, інакше можливий вихід з ладу принтера або комп'ютера. LPT-Порт може, як правило, працювати й для введення інформації, наприклад, зі сканером, але для цього потрібна низькорівнева утиліта Setup, запустивши її при завантаженні ОС, встановіть для LPT-Порту спеціальний режим роботи: **EPP – Extended Parallel Port.**

Порти SCSI і SCSI-Пристрою. **SCSI (Small Computer System Interface**; вимовляється "**скАзи",** з наголосом на першому складі) – інтерфейс, адаптери й порти для підключення широкого спектра зовнішніх пристроїв – жорстких дисків, CD-ROM / DVD-ROM, сканерів і ін. Стандарт SCSI був запропонований на початку 1980-х рр. і одержав широке поширення, завдяки фірмі Sun, що широко використовувала його у своїх робочих станціях. Характерною зручною можливістю SCSI є можливість підключення до одному SCSI-Порту **гірлянди (ланцюжка) SCSI-Пристроїв** (до 10), кожний з яких має унікальний для даного з'єднання **SCSI ID** – число від 0 до 9, установлюване звичайно на задній панелі SCSI-Пристрою. Наприклад, за традицією, SCSI ID сканера дорівнює 4. На одному з кінців ланцюжка – SCSI-Порт із контролером, на іншому – **термінатор –** перемикач на задній панелі пристрою, установлюваний у певне положення як ознака кінця SCSI-Ланцюжка. Кожний пристрій, крім останнього, з'єднано з наступним SCSI-Пристроєм спеціальним кабелем. SCSI-Роз'єм нагадує роз'єм порту LPT, однак має з боків спеціальні металеві захвати ("лапки") для більшої

надійності підключення. Перевага SCSI, крім можливості використання гірлянд пристроїв, у його швидкодії та надійності. Ранні моделі SCSI мали швидкість обміну інформацією до 10-12 мегабіт у секунду, зараз - 240-250 мегабіт у секунду. Є кілька стандартів SCSI (у тому числі - Wide SCSI, Ultra Wide SCSI), на жаль, не сумісних по роз'ємах.

Порт **VGA (Video Graphic Adapter)** використовується для підключення **монітора (дисплея),** керованого **графічним контролером (процесором).**

IEEE 1394 (FireWire) – порти для підключення цифрових відеокамер або фотоапаратів. Характерна риса – невеликий блискучий плоский роз'єм шириною 3-5 мм (є два його стандарти). Порт працює в дуплексному режимі, тобто дозволяє управляти не тільки введенням інформації з камери в комп'ютер, але й установками самої камери (наприклад, перемотуванням стрічки) за допомогою комп'ютерної програми (наприклад, Windows Movie Maker). За допомогою такого ж порту може бути підключений також телевізор, що має інтерфейс FireWire. Характерною рисою сучасних комп'ютерів є те, що FireWire-Порти монтуються прямо на **материнській платі (motherboard) –** основній друкованій платі комп'ютера, на якій змонтовані процесор і пам'ять. У таких випадках у технічних характеристиках комп'ютера звичайно вказується: **"FireWire on board (на борті)".** Читачам рекомендується не плутати **FireWire** з Wi-**Fi** – стандартом швидкого бездротового зв'язку.

HDMI (High Definition Multimedia Interface) – інтерфейс і порт, що дозволяє підключити до комп'ютера телевізор або інше відеоустаткування, що забезпечує найкращу якість відтворення (HD – High Definition). Роз'єм HDMI нагадує роз'єм USB. HDMI-Порт входить у комплектацію всіх сучасних портативних комп'ютерів.

Bluetooth – пристрій для бездротового підключення (за допомогою радіозв'язку) до комп'ютера мобільних телефонів, органайзерів, а також навушників, плейєрів і т.ін. Зручність Bluetooth у тім, що комп'ютер і телефон залишаються з'єднаними, навіть якщо відійти від комп'ютера з телефоном на деяку відстань, не більше 10-15 метрів (Bluetooth 2.0). Новий стандарт Bluetooth 3.0 забезпечує взаємодію на відстані 200-250 м. Звичайно портативні комп'ютери комплектуються убудованими адаптерами Bluetooth, або можна придбати адаптер Bluetooth, що підключається через USB. Недолік Bluetooth - відносно маленька сумарна швидкість передачі інформації. Наприклад, при пересиланні на комп'ютер через Bluetooth з мобільного телефону Nokia 3230 цифрової фотографії обсягом 500 кілобайт потрібно чекати порядку 10 - 15 секунд.

Інфрачервоний порт (IrDA) – порт для підключення ноутбука до мобільного телефону (або двох ноутбуків один до одного) через інфрачервоний зв'язок. Незручність портів IrDA – необхідність установки двох пристроїв, що з'єднуються, поруч, на відстані 20-30 см один від одного, без фізичних перешкод між ними. Швидкість передачі інформації – 10-12 мегабіт у секунду. Сучасні ноутбуки вже не комплектуються портами IrDA.

Є також **мережні пристрої – порти й адаптери –** для підключення комп'ютера до локальної мережі.

Функціонування комп'ютерної системи

Перевага описаного модульного підходу до апаратури в тім, що центральний процесор, пам'ять і зовнішні пристрої можуть функціонувати паралельно. Роботою кожного пристрою управляє спеціальний контролер. При необхідності виконання введення-виведення центральний процесор генерує переривання, у результаті якого викликається операційна система, у свою чергу, як реакція на переривання запускається драйвер пристрою, який активізує його контролер. Кожний контролер пристрою має локальний буфер – спеціалізовану пам'ять для обміну інформацією між комп'ютером і пристроєм. Для того, щоб контролер міг почати вивід на пристрій, попередньо центральний процесор (точніше, драйвер пристрою, запущений на ньому) повинен переслати інформацію із заданої ділянки оперативної пам'яті в буфер пристрою. Далі контролер пристрою вже виконує вивід інформації з буфера на сам пристрій (наприклад, записує її в задану ділянку жорсткого диска). По закінченні обміну інформацією, контролер генерує сигнал про **переривання (interrupt)** по системній шині, цим інформуючи процесор про закінчення операції. Для того, щоб уникнути повторних пересилань великих обсягів інформації, у сучасних комп'ютерах застосовують **DMA (Direct Memory Access) – контролери** – контролери із прямим доступом до оперативної пам'яті. Такі контролери використовують при обміні із пристроєм не свою спеціалізовану пам'ять, а прямо ділянку оперативної пам'яті, де і розміщається буфер обміну.

Обробка переривань

Операційну систему можна розглядати як **програму, керовану перериваннями (interrupt-driven program).** Переривання центрального процесора передає керування підпрограмі обробки даного виду переривань, яка є частиною ОС. У більшості комп'ютерів цей механізм реалізований через **вектор переривань (interrupt vector)** – резидентний масив в оперативній пам'яті, у якому зберігаються доступні по номерах переривань адреси підпрограм-оброблювачів переривань (модулів ОС). При обробці переривання апаратура й ОС зберігають **адресу перерваної команди.** При поновленні обчислень виконання перерваної команди буде знову повторене.

Очевидно, що при обробці переривання, у свою чергу, може виникнути інше переривання. У цьому випадку нове вхідне переривання **затримується (disabled),** і інформація про нього запам'ятовується в **черзі переривань** – системній структурі ОС, що забезпечує почергову обробку всіх виниклих переривань.

Крім переривань, генеруємих апаратурою неявно при обчисленнях, можливо також **програмоване переривання (trap**; дослівно – **пастка**) за допомогою спеціальної команди процесора. У випадку такого переривання працює загальний механізм запуску оброблювача переривання - частини ОС. Таким чином, зі спрощеної точки зору, ОС можна розглядати як набір оброблювачів переривань.

При перериванні ОС зберігає **стан процесора –** значення регістрів і значення **лічильника команд (program counter – PC)** – адреси перерваної команди**.** Оброблювач переривання в ОС визначає по вмісту сегмента об'єктного коду, якого виду переривання виникли і які дії по його обробці варто почати. Серед можливих видів переривань, крім фіксації різних помилок, є також **переривання по таймеру** – періодичні переривання через певний квант часу, призначені для **опитування пристроїв (polling)** – дій операційної системи по періодичній перевірці стану всіх портів і зовнішніх пристроїв, що можуть мінятися із часом: наприклад, до USB-Порту була підключена флешка; принтер закінчив печатку й звільнився, і т.д. ОС виконує реконфігурацію системи й коректує системні таблиці, що зберігають інформацію про пристрої.

На рис. 1.3 зображено тимчасову діаграму переривань процесора, який виконує введення-виведення.

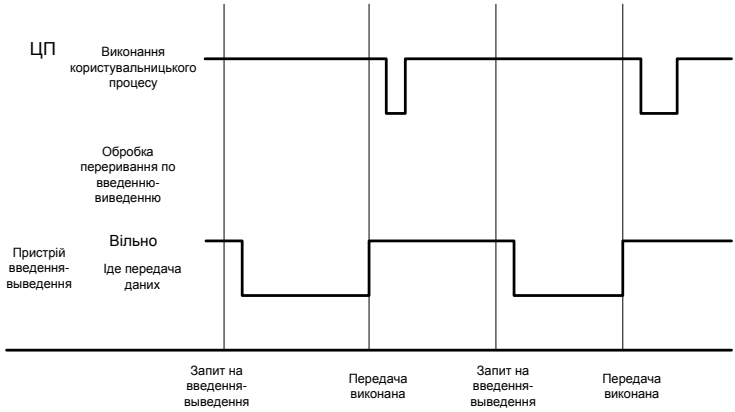


Рис. 1.3. Тимчасова діаграма переривань процесора при введенні-виведенні

На діаграмі видні моменти зміни станів процесора й пристрою введення-виведення: переривання по запиті на введення-виведення, обробка цього переривання й пересилання інформації з пам'яті в буфер пристрою, виклик драйвера й контролера, закінчення обміну й переривання контролера, продовження обчислень.

Є два різновиди режиму введення-виведення – **синхронний** і **асинхронний** .

Синхронне введення-виведення – це введення-виведення, виконання якого приводить до переходу програми в стан очікування, доти, поки операція введення-виведення не буде повністю закінчена. На апаратному рівні – команда введення-виведення переводить процесор у стан очікування (idle) до наступного переривання. При даному режимі в кожний момент виконується не більше одного запиту на введення-виведення; одночасне введення-виведення відсутньо. Синхронне виведення виконують відомі всім програмістам оператори виду **println(x).** При їхньому використанні в програмах ми не замислюємося над тим, що використовуємо досить неефективний варіант синхронного введення-виведення. Однак мислення більшості програмістів - послідовне, у тому розумінні, що про свою програму вони мислять як про послідовно виконувану, і взагалі не думають про

можливість якого-небудь распаралелювання. При налагодженні програми, або якщо розмір виведеної інформації невеликий, це цілком припустимо.

Асинхронне введення-виведення – введення-виведення, виконуване паралельно з виконанням основної програми. Після того, як починається асинхронне введення-виведення, керування вертається користувальницькій програмі, без очікування завершення введення-виведення (останнє може бути виконано спеціальною явною операцією). Таким чином, операція асинхронного обміну як би розбивається на дві: **почати введення-виведення** і **закінчити введення-виведення**. Остання виконується для того, щоб у цьому місці програма все-таки очікувала завершення обміну, якщо його результат необхідний для подальших обчислень. Такий підхід до реалізації обміну більш важкий для розуміння програмістами й може привести до помилок (наприклад, використана тільки операція початку введення-виведення, а виклик операції її закінчення забута).

Таблиця стану пристроїв

На системному рівні, при обміні відбувається наступне. Виконується **системний виклик (system call)** – запит до ОС шляхом виклику системної підпрограми, у цьому випадку – щоб дозволити користувачеві очікувати завершення введення-виведення. Операційна система зберігає **таблицю стану пристроїв**, у якій кожному пристрою відповідає елемент, що містить тип пристрою, його адресу й стан. ОС індексує таблицю пристроїв, з метою визначення стану пристрою й модифікації елемента таблиці для включення в неї інформації про переривання.

Архітектура синхронного (a) і асинхронного (b) введення-виведення ілюструється на рис. 1.4.

На схемі видно, що відмітною рисою синхронного обміну є перехід процесора в стан очікування до закінчення операції введення-виведення.

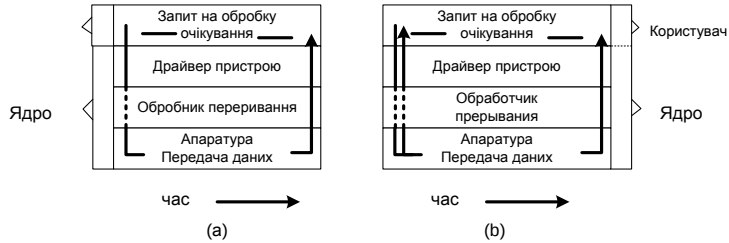


Рис. 1.4. Архітектура синхронного й асинхронного введення-виведення

На рис. 1.5 показано приклад стану таблиці пристроїв введення-виведення, яка збережена операційною системою. Для кожного пристрою зберігається інформація про його ім'я, стан, а для зайнятих пристроїв – адреса початку й довжина порції інформації, що підлягає обміну. Якщо для деякого пристрою (у прикладі – **диск 3**) є кілька запитів на введення-виведення, всі вони організуються в чергу й обслуговуються по черзі, у міру звільнення пристрою.

Рис. 1.5. Приклад стану таблиці зовнішніх пристроїв ОС

Прямий доступ до пам'яті (Direct Memory Access - DMA) - більш ефективний метод роботи контролерів пристроїв введення-виведення, використовуваний для роботи високошвидкісних пристроїв, здатних передавати інформацію зі швидкістю, близькою до швидкості роботи пам'яті.

DMA-Контролер передає блок даних з буферної пам'яті безпосередньо в основну пам'ять, без участі процесора. Перевага подібного широко застосовуваного підходу - не тільки в тім, щоб уникнути зайвого пересилання даних з однієї ділянки пам'яті в іншу, але також у тім, що переривання в цьому випадку генерується на

кожний блок даних, що пересилаються, але не на кожний пересилаємий байт, як при більш традиційному способі обміну.

Структура пам'яті

Основна (оперативна) пам'ять – єдина велика частина пам'яті, до якої процесор має безпосередній доступ. Як відомо, уміст основної пам'яті не зберігається після перезавантаження системи або після вимикання комп'ютера. **Зовнішня (вторинна) пам'ять** – розширення основної пам'яті, що забезпечує функціональність стійкої (що зберігається) пам'яті великого обсягу.

Як вторинна пам'ять найчастіше використовуються **жорсткі диски (hard disks).** Фізично вони складаються із твердих пластин з металу або скла, які покриті магнітним шаром для запису. Поверхня диска логічно ділиться на **доріжки (tracks),** які, у свою чергу, діляться на **сектори.** Контролер диска визначає логіку взаємодії між пристроєм і комп'ютером.

Пристрій жорсткого диска показано на рис. 1.6.

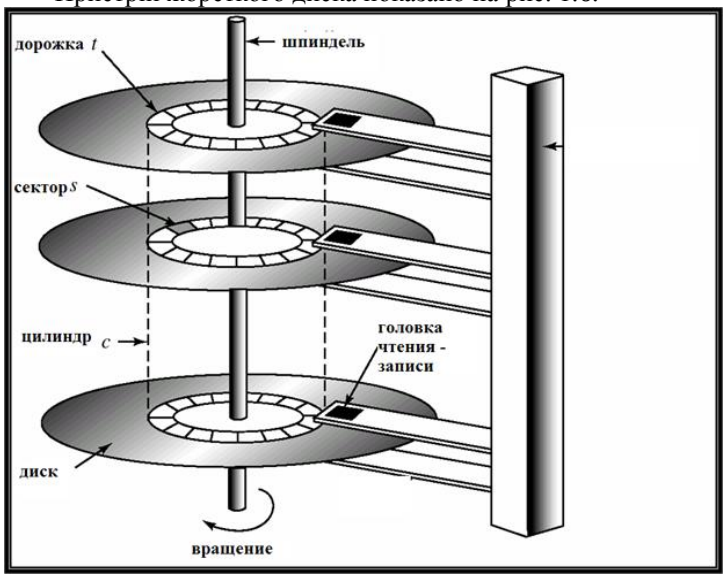


Рис. 1.6. Пристрій жорсткого диска

Як видно з рисунка, **циліндр** - це група вертикально розташованих один під одним секторів різних магнітних дисків з одним і тим же номером доріжки.

Системи пам'яті організовані в **ієрархію**, виходячи з їхньої швидкодії, вартості й можливості збереження інформації (стійкості). Для оптимізації роботи пам'яті будь-якого виду використовується **асоціативна пам'ять (кэш – cache), яка** розташована в більш швидкодіючих системах пам'яті, вона зберігає елементи більш повільної пам'яті, які частіше використовуються. Із цього погляду, оперативну пам'ять можна розглядати як кеш для зовнішньої пам'яті. Кеш-пам'ять – це, по суті справи, асоціативний список пар **(Адреса, Значення),** причому апаратний пошук у ньому відбувається за адресою як по ключу. Таким чином, перед звертанням до повільної зовнішньої пам'яті спочатку відбувається пошук по заданій адресі в кеш-пам'яті, і якщо він не привів до успіху, виконується стандартне звертання до зовнішньої пам'яті. Принцип кешування дуже важливий і дозволяє істотно прискорити роботу із зовнішньою пам'яттю. Однак він вимагає реалізації спеціальної політики керування кеш-пам'яттю, тому що кешування вводить додатковий рівень в ієрархії пам'яті й вимагає погодженості даних, збережених одночасно на різних рівнях пам'яті. Апаратура й ОС підтримують **кеш команд, кеш даних, кеш жорсткого диска** й т.д. – для всіх видів пам'яті.

Ієрархія пристроїв пам'яті (у спрощеному виді) показана на рис. 1.7

Більше швидкі види пам'яті на схемі розташовані вище, більше повільні - нижче. Схема особливих коментарів не вимагає. Деякі часто використовувані види зовнішньої пам'яті:

- **флеш-пам'ять (флешка)** – зовнішня пам'ять компактного розміру, модуль якої підключаються через USB-Порт. Параметри: обсяг - до 128 гігабайт і більше; швидкість обміну через порт USB 2.0: 240 – 260 мегабіт у секунду;
- **зовнішній жорсткий диск (ZIV drive** і інші) – обсяг до 1 терабайта; працює також через порт USB;

BluRay – диски – новий різновид компакт-дисків великого об’єму (однобічні – 25 гігабайт, двосторонні – 50 гігабайт). Для порівняння, стандартний об’єм диска DVD становить 4.7 гігабайт.

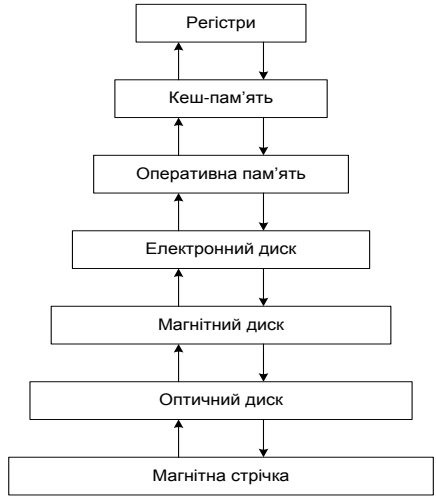


Рис. 1.7. Ієрархія пристроїв пам'яті

Апаратний захист пам'яті й процесора

З метою спільного використання системних ресурсів (пам'яті, процесора, зовнішніх пристроїв) декількома програмами, потрібно, щоб апаратура й операційна система забезпечили неможливість впливу програми, що виконується некоректно, на інші користувальницькі програми. Для цього необхідна апаратна підтримка, як мінімум, двох режимів виконання програм – **користувальницького (непривілейованого) режиму (user mode) –** для виконання програм користувачів і **системного (привілейованого, режиму ядра - system mode, monitor mode)** - для модулів операційної системи. Ідея двох режимів у тім, щоб виконувані в привілейованому режимі модулі ОС могли виконувати розподіл і виділення системних ресурсів, зокрема, формувати нові адреси, а користувальницькі програми, у результаті помилок або навмисних атак, виконуючись у звичайному режимі, не могли б звернутися до ділянки пам'яті операційної системи або іншого завдання, змінити їх і цим порушити їхню цілісність. Для визначення поточного режиму виконання команд в апаратурі вводиться **біт режиму**, рівний 0 для системного й 1 – для користувальницького режиму. При перериванні або збої апаратура автоматично

перемикається в системний режим. Деякі привілейовані команди, що змінюють системні ресурси й стан системи (наприклад, регістр стану процесора), повинні виконуватися тільки в системному режимі, це захистить системні ресурси від випадкового або навмисного псування при виконанні команд звичайної користувальницької програми.

Для **захисту введення-виведення** всі команди введення-виведення вважаються привілейованими. Необхідно гарантувати, щоб користувальницька програма ніколи не одержала керування в системному режимі й, зокрема, не могла б записати нову адресу у вектор переривань, що, як ми вже відзначали, містить адреси підпрограм обробки переривань, зокрема, зв'язаних з введенням-виведенням.

Використання системного виклику для виконання введення-виведення ілюструється на рис. 1.8.

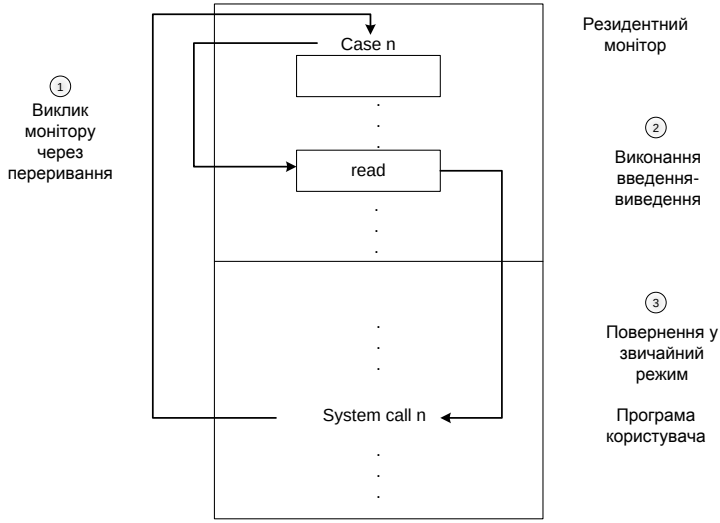


Рис. 1.8. Використання системного виклику для виконання введення-виведення

На схемі системний виклик номер **n** викликає програмуєме переривання (trap), що викликається ОС у привілейованому режимі, і по номері системного виклику визначається операція введення-виведення, що повинна бути виконана по даному перериванню. Потім у привілейованому режимі виконується операція введення-виведення, після чого відбувається переривання й повернення в користувальницьку програму, виконувану у звичайному режимі.

Для **захисту пам'яті** необхідно забезпечити захист, принаймні, для вектора переривань і підпрограм обслуговування переривань. Наприклад, неприпустимо дозволити користувальницькій програмі формувати у звичайному режимі довільні адреси й звертатися по них, тому що при цьому може бути порушена цілісність системних ділянок пам'яті. Щоб цього уникнути, в апаратурі вводяться два регістри, які відзначають межі припустимої ділянки пам'яті, яка виділена користувальницькій програмі. Це **базовий регістр (base register), що** містить початкову адресу ділянки пам'яті, що виділена користувальницькій програмі, і **регістр межі (limit register), що** містить розмір користувальницької ділянки пам'яті. Пам'ять поза відзначеним діапазоном уважається захищеної, тобто звертання до неї з користувальницької програми не допускаються (при спробі такого обігу виникає переривання).

Використання базового регістру й регістру меж ілюструється на рис. 1.9.

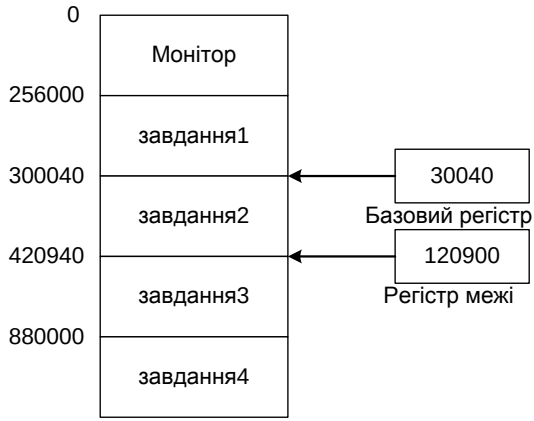


Рис. 1.9. Використання регістру бази й регістру меж для захисту пам'яті

На схемі завданню 2 виділена ділянка пам'яті, починаючи з адреси 300040 (зберігається в регістрі бази), довжиною 120900 (зберігається в регістрі межи), тобто за адресою 420939 включно. Обіг, наприклад, за адресою 420940 із програми завдання 2 приводить до переривання як неприпустиме - спрацьовує захист пам'яті.

Схема апаратного захисту адрес пам'яті ілюструється на рис.1.10.

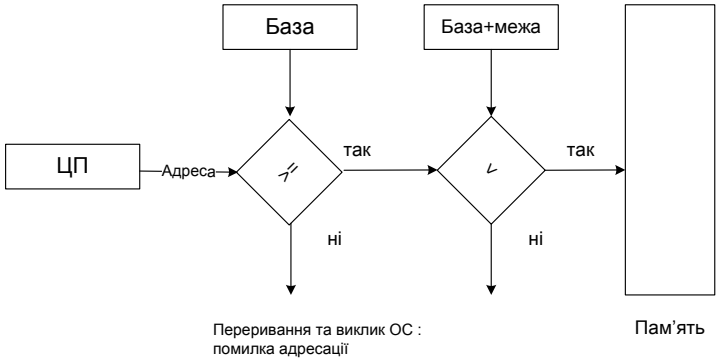


Рис. 1.10. Схема апаратного захисту адрес пам'яті

Апаратний захист адрес пам'яті в системах з теговою архітектурою

Більш радикальні міри для захисту пам'яті початі в системах з теговою архітектурою - МВК "Ельбрус", Burroughs 5000/6700/7700 та ін.

Як вже пояснювалося, у такій комп'ютерній системі кожне слово пам'яті має **тег** – інформацію про тип даних, що зберігається в даному слові. Спеціальні теги мають будь-які дані - наприклад, числа (цілі та дійсні), адреси, покажчики на процедури й ін. Апаратура при виконанні команди виконує динамічний контроль типів - перевіряє, чи відповідають теги операндів виконуваній операції. Якщо не відповідають відбувається переривання.

Адреси в системі з теговою архітектурою представлені спеціальним адресним словом - **дескриптором** (**descriptor**). Крім тегу й властиво адреси початку адресуємого масиву в пам'яті, дескриптор містить також **довжину** масиву й 4 біти захисту – **від читання, від запису, від виконання** й **від запису адресної інформації.** Формування й зміна дескриптора можливо тільки засобами ОС у привілейованому режимі. Користувальницька програма не може формувати та змінювати дескриптор, вона працює зі своєю ділянкою пам'яті як з масивом, захищеним тегом і дескриптором, утворюючи від нього підмассиви та формуючи їхні дескриптори (така дія дозволена). Припустима операція над масивом - **індексація a[i],** у якій апаратно перевіряється, що індекс **i** не виходить за межі масиву

а. Таким чином, втручання в "чужу" ділянку пам'яті в такій системі принципово неможливо. Неможлива також адресна арифметика (у стилі C / C++), тому що спроба виконання арифметичної операції над словом з тегом **дескриптор** приводить до негайного переривання.

Крім дескриптора, є також **непряме слово (indirect word)** – тегова адреса для звертання до елемента даних однією командою, безпосередньо за адресою (без індексації). Для непрямих слів фактично виконуються ті ж апаратні перевірки, що й для дескрипторів.

Подібна система захисту, з одного боку, досконала й стовідсоткова, з іншого, зрозуміло, вимагає більших накладних витрат на апаратну перевірку тегів, яку відключити неможливо, навіть у випадках, коли з коду програми очевидно, що ніяких помилок при роботі з адресною інформації немає.

Особливості ОС для персональних комп'ютерів

Персональні комп'ютери призначені, як правило, для одного користувача. Проте, ОС для персональних комп'ютерів повинна передбачати режим мультипрограмування (багатозадачності), тому що користувачам часом зручніше виконувати кілька завдань паралельно - наприклад, набирати деякий текст у редакторі, приймати електронну пошту й одночасно друкувати на принтері які-небудь документи. Крім того, при роботі в локальній мережі можливий віддалений вхід на комп'ютер інших користувачів. Тобто, ОС для персональних комп'ютерів повинна підтримувати також режим розподілу часу.

Персональні комп'ютери мають різноманітний набір пристроїв введення-виведення, роботу з якими повинна підтримувати операційна система за допомогою **драйверів –** низькорівневих системних програм для керування цими пристроями. Для користувача зручніше за все, якщо всі необхідні драйвери убудовані в операційну систему. Однак ситуація ускладнюється тим що драйвери пристроїв розробляє зазвичай фірма-розроблювач відповідного пристрою - в англомовній термінології, **Original Equipment Manufacturer (OEM),** а не фірма-розроблювач ОС. Тому при випуску й установці на комп'ютер нової ОС можуть виникнути проблеми із драйверами – який-небудь пристрій нова ОС "не розуміє". На практиці, повинно пройти не менш двох-трьох років експлуатації нової ОС, перш ніж

для неї з'являться драйвери для всіх використовуваних зовнішніх пристроїв, хоча останнім часом щодо цього ситуація значно покращилася - нові ОС стають усе більше"тямущими" і мають у своєму складі величезні набори драйверів.

Персональний комп'ютер має традиційні клавіатуру й мишу, що підключаються через USB-Порт, або бездротові клавіатуру й мишу, блок керування який також підключається через USB-Порт. Портативний комп'ютер може мати також убудований маніпулятор типу trackball (кулька для переміщення курсору миші) або touchpad (плоска пластинка для цієї ж мети). До комп'ютера підключений монітор: для настільного комп'ютера – до порту VGA, для портативного – монітор убудований у комп'ютерну систему, але додатково може підключатися через порт VGA зовнішній монітор або мультимедійний проектор. До традиційних додаткових зовнішніх пристроїв ставиться також принтер (підключається через порт USB, більше старі моделі – через так званий **паралельний порт,** або **LPT – абревіатура від Line PrinTer)**. Рідше використовується **сканер** – пристрій для оцифровки паперових зображень, наприклад, підписаних або рукописних документів. Сканер може також підключатися через порт USB, однак деякі моделі сканерів підключаються через інший інтерфейс – **SCSI**, використовуваний і для жорстких дисків. Є внутрішній жорсткий диск (hard drive) ємністю 250 GB – 1 TB і більше, що підключається через інтерфейс IDE або SATA. Можуть підключатися через порт USB також зовнішні накопичувачі - flash-**пам'ять, ZIV drives** і інші різновиди **зовнішніх жорстких дисків**, що мають ємність до 1 терабайта. Операційна система повинна забезпечувати їхнє використання як частин комп'ютерної системи (наприклад, на зовнішній ZIV-Диск може бути навіть встановлене програмне забезпечення, у тому числі - інша операційна система). Для настільного комп'ютера в комплект входить пристрій читання й запису компакт-дисків у різних форматах - CD-ROM, CD-RW (з можливістю запису на CD); DVD-ROM/DVD-RW; DVD-RAM (останнє означає пристрій з режимом безпосереднього запису на компакт-диск, як у пам’ять); BluRay - більш сучасний формат компакт-дисків ємністю до 25 або 50 GB і ін. Досить важливим зовнішнім пристроєм, особливо для портативного комп'ютера, є порт для підключення цифрової відеокамери (IEEE 1394, або FireWire), більш мініатюрний, ніж USB. Він має дуплексний режим роботи, так що, наприклад, перемотування відеострічки на відеокамері може запускатися програмним шляхом з комп'ютера.

Найбільш важливими властивостями ОС для персонального комп'ютера повинні бути, звичайно, простота й зручність у

використанні, дружність до користувача. Це досягається насамперед, зручним і сучасним апаратним і програмним користувальницьким інтерфейсом, наприклад, інтерфейсом типу multi-touch (з доступом безпосередньо до екрана), ноутбуками типу Tablet PC (з можливістю повороту екрана й введення інформації дотиком до екрана).

При розробці ОС для ПК використовуються ті ж технології, які застосовуються й в "великих" ОС (для mainframe-комп'ютерів). Однак, оскільки користувач має персональний доступ до комп'ютера, він часто не має потреби в яких-небудь системних програмах для оптимізації роботи процесора або в поліпшених засобах захисту (останньою, однак, не слід зневажати й відключати її, тому що на комп'ютер можливі мережні атаки).

На тому самому персональному комп'ютері можуть бути встановлені, при необхідності, дві або більше операційних системи - такий комп'ютер зветься **double bootable system**, і при його включенні користувачеві видається початкове меню для уточнення, яку саме ОС потрібно запустити – **boot loader** (завантажник ОС). Таке використання комп'ютера рекомендується, наприклад, для студентів, що вивчають ОС і бажають спробувати нову операційну систему, або вивчити іншу вже відому, на яку дотепер бракувало часу, - наприклад, установити на одному комп'ютері Windows і Linux. Для установки другої ОС необхідно скористатися спеціальною утилітою (наприклад, **Partition Magic**) для виділення на диску для інсталяції нової ОС окремого **розділу (partition)** – суміжної ділянки дискової пам'яті, що має певне позначення, найчастіше – у вигляді латинської букви.

Персональні комп'ютери мають **мережні адаптери (мережні карти) –** пристрої для підключення до локальної мережі. Відповідно, ОС для персональних комп'ютерів мають у своєму складі драйвери мережних адаптерів і користувальницький інтерфейс для настроювання підключення комп'ютера до локальної мережі.

Паралельні комп'ютерні системи й особливості їх ОС

Паралельні комп'ютерні системи – це мультипроцесорні системи з декількома безпосередньо взаємодіючими процесорами. Класичні приклади: із закордонних комп'ютерів - CRAY, з вітчизняних - "Ельбрус"; з більше сучасних - комп'ютери серії СКІФ. У цей час випускаються мультипроцесорні робочі станції - наприклад, купивши або одержавши в подарунок настільний комп'ютер, Ви можете виявити в його складі два або навіть чотири

процесори. Відповідно, ОС повинна забезпечувати реконфігурацію такої системи, підключення нових процесорів або видалення процесорів із системи, распаралелювання рішення завдання на декількох процесорах і синхронізацію вирішальних її паралельних процесів.

Серед паралельних комп'ютерів виділяються **тісно зв'язані (tightly coupled) системи**, у яких процесори розділяють загальну пам'ять і таймер (такти); взаємодія між ними відбувається через загальну пам'ять.

Багатоядерні (multi-core) комп'ютери – комп'ютерні системи, засновані на тісно зв'язаних один з одним процесорах (**ядрах**), що перебувають в одному кристалі, що розділяють асоціативну пам'ять (кеш) другого рівня й працюють на загальній пам'яті.

Переваги паралельної комп'ютерної системи:

1. **Поліпшена продуктивність** (**throughput**) – очевидно, що распаралелювання алгоритму рішення завдання може зменшити сумарний час її рішення;
2. **Економічність** – у паралельній системі ОС можна доручити частину роботи іншому процесору або ядру;
3. **Підвищена надійність** – при збої або відмові одного із процесорів ОС можна перемкнути обчислення на інший процесор;
4. **"Дружнє" до користувача зниження продуктивності (graceful degradation)** – якщо один із процесорів відмовив і виведений з конфігурації, користувач, при правильній організації комп'ютера й ОС, може навіть не відчути вповільнення обчислень;
5. **Стійкість до помилок (fail-soft system)** – стабільна робота багатопроцесорної системи при помилці в апаратурі або в програмі.

Симетричні й асиметричні мультипроцесорні системи

Симетрична мультипроцесорна система - symmetric multiprocessing (SMP) – це багатопроцесорна комп'ютерна система, всі процесори якої рівноправні й використовують ту саму копію ОС. Операційна система при цьому може виконуватися на **будь-якому** процесорі. У такій системі будь-якому вільному процесору може бути доручено будь-яке завдання. Всі процесори використовують загальну пам'ять і загальні дискові ресурси. Кілька процесів (або потоків) можуть виконуватися одночасно без істотного порушення продуктивності. Більшість сучасних ОС підтримують архітектуру

SMP. Після інсталяції ОС (наприклад, Linux) на симетричну мультипроцесорну систему користувач може помітити в меню boot loader, що фактично на його комп'ютер встановилася не одна, а дві версії ОС - з підтримкою SMP і без неї.

Асиметрична мультипроцесорна система (asymmetric multiprocessing) – це багатопроцесорна комп'ютерна система, у якій процесори спеціалізовані за своїми функціями. Кожному процесору дається специфічне завдання; **головний процесор (master processor)** планує роботу **підлеглих процесів (slave processors).** У такій системі ОС, як правило, виконується на одному певному, закріпленому за нею, центральному процесорі. Подібна архітектура більш типова для дуже великих систем. Приклад – система "Ельбрус", що мала у своєму складі, залежно від конфігурації, від одного до 10 центральних процесорів, від одного до чотирьох спеціалізованих **процесорів введення-виведення (ПВВ),** від одного до чотирьох **процесорів передачі даних (ППД).**

Схема організації SMP-Архітектури комп'ютерів наведена на рис. 1.11.

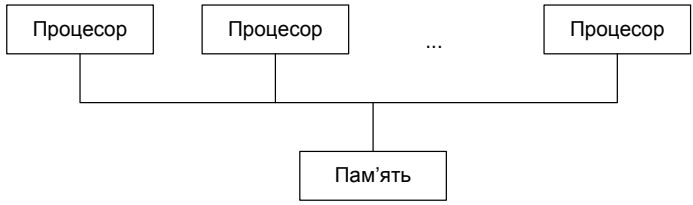


Рис. 1.11. Схема організації SMP-Архітектури комп'ютерів

Розподілені комп'ютерні системи й особливості їх ОС

У **розподіленій системі (distributed system)** обчислення розподілені між декількома фізичними процесорами (комп'ютерами), об'єднаними між собою в мережу.

Слабко зв'язана система (loosely coupled system) – розподілена комп'ютерна система, у якій кожний процесор має свою локальну пам'ять, а різні процесори взаємодіють між собою через **лінії зв'язку –** високошвидкісні шини, телефонні лінії, бездротовий зв'язок (Wi-Fi, EVDO, Wi-Max і др.).

Переваги розподілених систем:

1. **Розподіл (спільне використання) ресурсів**: у розподіленій системі різні ресурси можуть зберігатися на різних комп'ютерах. Немає необхідності дублювати програми або дані, зберігаючи їхні копії на декількох комп'ютерах.

2. **Спільне завантаження (load sharing)**: кожному комп'ютеру в розподіленій системі можуть бути доручені певні завдання, що він виконує паралельно з виконанням іншими комп'ютерами своїх завдань.

3. **Надійність**: при відмові або збої одного з комп'ютерів розподіленої системи його завдання може бути перерозподілено іншому комп'ютеру, щоб збій у мінімальному ступені вплинув або зовсім не вплинув на підсумковий результат.

4. **Зв'язок**: у розподіленій системі всі комп'ютери зв'язані один з одним, так що, наприклад, при необхідності можливий віддалений вхід з одного комп'ютера на інший з метою використання ресурсів могутнішого комп'ютера.

У розподіленій системі комп'ютери зв'язані в мережну інфраструктуру, що може бути:

1. локальною мережею (local area network - LAN);

2. глобальною або регіональною мережею (wide area network - WAN).

По своїй організації розподілені системи можуть бути клієнт-серверними (client-server) або одноранговими (peer-to-peer) системами. У клієнт-**серверній** системі певні комп'ютери відіграють роль серверів, а інші – роль клієнтів, що користуються їхніми послугами. Подібна організація розподілених систем найпоширеніша, і ми розглянемо її докладніше. В **одноранговій** розподіленій системі всі комп'ютери рівноправні.

Структура клієнт-серверної системи зображена на рис. 1.12.

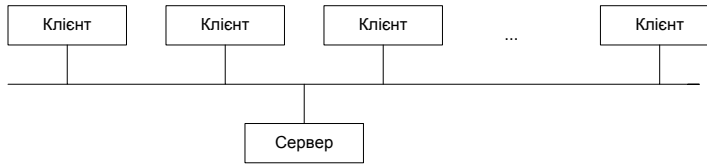


Рис. 1.12. Структура клієнт-серверної системи

Види серверів у клієнт-серверних комп'ютерних системах

Клієнт-Серверна архітектура розподілених систем досить широко поширена й підтримана операційними системами. Тому дуже важливо знати, які види й функції серверів пропонують сучасні розподілені системи.

Файл-Сервер (file server) – комп'ютер і програмне забезпечення, що надають доступ до підмножини файлових систем, розташованих на дисках комп'ютера-сервера, іншим комп'ютерам локальної мережі (LAN). Приклад – серверне програмне забезпечення **SAMBA (SMB –** скорочення від **Server Message Block)** для ОС типу UNIX (Linux, FreeBSD, Solaris і т.д.), що забезпечує доступ з Windows-Комп'ютерів локальної мережі до файлових систем UNIX-Машин. Samba також реалізована для платформи Macintosh / MacOS.

Сервер додатків (application server) – комп'ютер і програмне забезпечення, що надає обчислювальні ресурси (пам'ять і процесор) і необхідне оточення для віддаленого запуску певних класів (як правило, великих) додатків з інших комп'ютерів локальної мережі. Приклади серверів додатків - WebSphere (IBM), WebLogic (BEA) - найкращі з відомих серверів додатків, що працюють в Java Enterprise Edition (JEE).

Сервер баз даних (database server) – комп'ютер і програмне забезпечення, що надає доступ іншим комп'ютерам мережі до баз даних, розташованих на комп'ютері-сервері. Приклад: серверне програмне забезпечення для доступу до баз даних Microsoft SQL Server.

Веб-Сервер (Web server) – комп'ютер і програмне забезпечення, що надає доступ клієнтам через WWW до Web-Сторінок, розташованих на комп'ютері-сервері. Приклад: вільно розповсюджуваний Web-Сервер Apache.

Прокси-Сервер – комп'ютер і програмне забезпечення, що є частиною локальної мережі й підтримує ефективний обіг комп'ютерів локальної мережі до Інтернету, фільтрацію трафіка, захист від зовнішніх атак. Proxy-Сервер звичайно вбудований в операційну систему.

Сервер електронної пошти – комп'ютер і програмне забезпечення, що виконують відправлення, одержання й "розкладку" електронної пошти для комп'ютерів деякої локальної мережі. Можуть забезпечувати також **криптування** пошти (email encryption) – шифрування електронних листів перед відправленням адресатам з

певного мережевого домена (як правило, замовникові) і їхнє дешифрування після одержання від замовника.

Серверний бэк-энд (Server back-end) – група (пул) зв'язаних у локальну мережу серверних комп'ютерів, використовуваних замість одного сервера, з метою більшої надійності й надання більшого обсягу ресурсів. Інший термін, близький до цього, - **центр обробки даних (data center).** Ці поняття особливо актуальні у зв'язку з усе більшим поширенням хмарних обчислень, що є, із цього погляду, найбільш сучасною реалізацією клієнт-серверної схеми взаємодії.

Кластерні обчислювальні системи і їх ОС

Комп'ютерні кластери досить популярні для наукових обчислень. Комп'ютери в кластері, як правило, зв'язані між собою через швидку локальну мережу. Кластеризація дозволяє двом або більше системам використовувати загальну пам'ять. Кластеризація забезпечує високу надійність. Розрізняють комп'ютерні кластери двох видів:

- **асиметрична кластеризація (asymmetric clustering) –** організація комп'ютерного кластера, при якій один комп'ютер виконує додаток, а інші простоюють;
- **симетрична кластеризація (symmetric clustering) -** організація комп'ютерного кластера, при якій всі машини кластера виконують одночасно різні частини одного великого додатка.

Розрізняють також:

- **кластери з високошвидкісним доступом (high-availability clusters) –** комп'ютерні кластери, що забезпечують оптимальний **доступ** до ресурсів, які надані комп'ютерами кластера, наприклад, до баз даних;
- **кластери з балансованим завантаженням (load-balancing clusters)** – комп'ютерні кластери, які мають кілька вхідних комп'ютерів, що балансують запити (front - ends), та розподіляють завдання між комп'ютерами серверного back-end'a (серверної ферми).

Кластери часто використовуються в університетах, в дослідницьких центрах. Операційні системи для кластерів: Windows 2003 for clusters; Windows 2008 High-Performance Computing.

Системи й ОС реального часу

Системи реального часу часто використовуються як управляючі пристрої для спеціальних додатків, - наприклад, для наукових експериментів; у медичних системах, пов'язаних із зображеннями; системах управління в промисловості; системах відображення (display); системах управління космічними польотами, АЕС і ін. Для таких систем характерні наявність і чітке виконання певних тимчасових обмежень (час реакції - response time; час наробітку на відмову й ін.).

Розрізняються системи реального часу видів **hard real-time** і **soft real-time**.

Hard real-time – системи – системи реального часу, у яких при порушенні тимчасових обмежень може виникнути критична помилка (відмова) керованого нею об'єкта. Приклади: система керування двигуном автомобіля; система керування кардіостимулятором. У таких системах вторинна пам'ять обмежена або відсутня; дані зберігаються в оперативній пам'яті (RAM) або постійному запам'ятовувальному пристрої (ПЗУ, ROM). При використанні таких систем можливі конфлікти із системами розподілу часу, що не мають місця для ОС загального призначення. Точніше, при роботі подібних систем не допускаються переривання; всі необхідні дані для основного циклу роботи системи повинні бути попередньо завантажені у пам’ять; процес, що виконує код такої системи, не повинен піддаватися відкачці на диск. ОС для таких систем звичайно спрощені, замість віртуальної пам'яті виділяється фізична, всі інші види віртуалізації ресурсів виключені. Популярною практикою розробки ОС реального часу є практика розробки таких ОС на основі відкритих вихідних кодів ОС загального призначення шляхом "відсікання всього зайвого". Однак при цьому слід дотримуватися обережності.

Soft real-time – системи – системи реального часу, у яких порушення тимчасових обмежень не приводить до відмови керованого нею об'єкта. Звичайно це системи керування декількома взаємозалежними системами з ситуацією що постійно змінюється. Приклад - система планування рейсів на комерційних авіалініях**.** У випадку якої-небудь затримки в роботі такої системи, у найгіршому разі, пасажирам деяких рейсів прийде небагато почекати в аеропорті, але ніяких фатальних наслідків не буде. Подібні системи мають обмежену корисність для промислових систем керування. Вони також корисні в сучасних додатках (наприклад, для мультимедіа й віртуальної реальності), що вимагають розвинених можливостей ОС.

Розвиток концепцій і можливостей ОС представлене на рис.1.13.

На схемі добре видні аналогічні "хвилі" ("витки") розвитку ОС - спочатку для mainframe-комп'ютерів, потім - для мінікомп'ютерів, для персональних і для кишенькових комп'ютерів. Кожна хвиля проходить у своєму розвитку певні етапи. ОС розвиваються від резидентних моніторів до підтримки пакетного режиму (для ранніх моделей комп'ютерів), потім - режиму поділу часу, багатокористувальницьких і мережних можливостей.

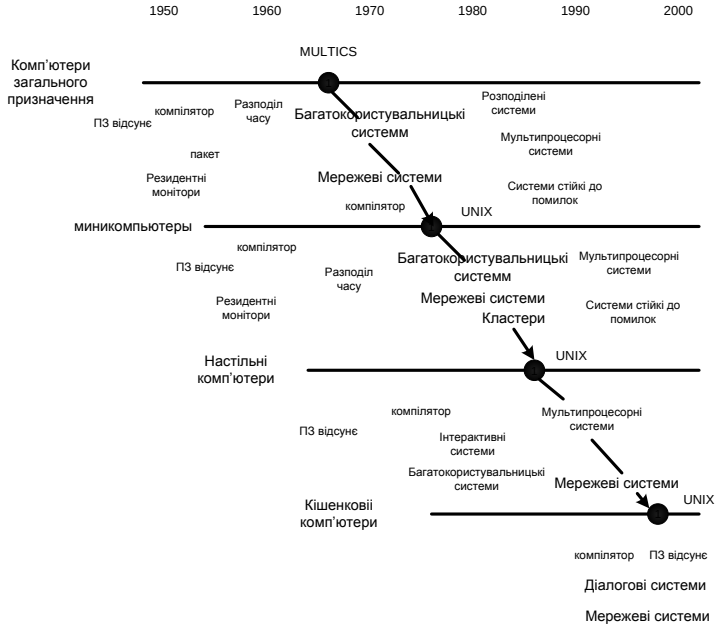


Рис. 1.13. Розвиток концепцій і можливостей ОС

Обчислювальні середовища

У сучасному світі IT має місце тенденція до інтеграції описаних вище пристроїв і їхніх локальних мереж в **обчислювальні середовища** – інтегровані розподілені комп'ютерні системи для рішення завдань у різних проблемних сферах. Обчислювальні середовища підрозділяються на наступні види:

- **традиційні обчислювальні середовища** – локальні й регіональні мережі, використовувані протягом декількох десятків років;
- **Web-Орієнтовані обчислювальні середовища –** обчислювальні середовища на основі Web-Сервісів, характерні для теперішнього часу, починаючи з 1990-х рр.; до цього класу ставляться й середовища для хмарних обчислень;
- **убудовані (embedded) обчислювальні середовища –** обчислювальні середовища для спеціалізованих пристроїв, наприклад, мережі мікропроцесорів, убудованих в елементи лінії електропередач.

Всі ці види обчислювальних середовищ повинні адекватно обслуговуватися операційними системами, у чому й складаються найближчі завдання їхньої розробки.

Хмарні обчислення й ОС для хмарних обчислень

Хмарні обчислення (**cloud computing**) є одним з найбільш популярних напрямків розвитку IT. **"Хмара" (cloud) –** це вже десятки років використовувана метафора для зображення сервісів, надаваних через Інтернет або іншу комунікаційну мережу (наприклад, через ATM-Мережу). **Хмарні обчислення** – модель обчислень, заснована на **динамічно масштабованих (scalable)** і **віртуалізованих** ресурсах (даних, додатках, ОС і ін.), які доступні й використовуються як **сервіси** через Інтернет і реалізуються за допомогою високопродуктивних **центрів обробки даних (data centers)**

Недолік хмарних обчислень у тім, що користувач виявляється повністю залежним від використовуваної ним "хмари" (у якій доступні використовувані ним дані й програми) і не може управляти не тільки роботою "хмарних" комп'ютерів, але навіть резервним копіюванням своїх даних. У зв'язку із цим виникає цілий ряд важливих питань про безпеку хмарних обчислень, збереження конфіденційності користувальницьких даних і т.д.; далеко не всі з них на даний момент вирішені.

Серйозною проблемою організації хмарних обчислень із погляду апаратури центрів обробки даних є економія електроенергії й проблема розподілу завантаження, тому що хмарні обчислення в кожному центрі обробки даних мають (або в найближчому майбутньому будуть мати) **мільйони** віддалених користувачів.

Найбільш популярна "хмарна" платформа - Microsoft Windows Azure (хмарна ОС) і Microsoft Azure Services Platform

(реалізована на основі Microsoft.NET). Windows Azure можна розглядати як "ОС у хмарі". Користувачеві немає необхідності турбуватися про її інсталяцію на його комп'ютері, який може не мати для цього необхідних ресурсів. Усе, що потрібно, це мати Web-Браузер і мінімальний пакет надбудов (plug - ins) для запуску й використання через браузер хмарних сервісів.

У цей час всі великі компанії (Microsoft, IBM, HP, Dell, Oracle і ін.) розробляють свої системи хмарних обчислень; є тенденція до інтеграції цих корпоративних систем у єдину доступну користувачеві "хмару".

Резюме

Мета навчання комп'ютерним наукам полягає в тому, щоб виробити уявлення про конкретні процеси і механізми обробки інформації та зрозуміти відносини, які існують між прикладними програмами та обчислювальними системами.

В даному розділі розглядається організація функціонування обчислювальної системи, основні її компоненти та архітектура, розкриваються механізми управління, що дозволяють ефективно координувати роботу обчислювальних ресурсів при проведенні обчислень. Розглядаються різні типи комп'ютерних систем та особливості їх операційних систем.

Контрольні запитання та завдання

1. Перелічіть складові комп'ютерних систем. Опишіть загальну картину функціонування комп'ютерної системи.
2. З якою метою організована таблиця стану пристроїв?
3. Опишіть структуру пам'яті комп'ютера.
4. Перелічіть елементи ієрархії пам'яті.
5. Як організовано апаратний захист пам'яті й процесора?
6. Як класифікуються сучасні комп'ютерні системи?
7. Що представляють собою кластери комп'ютерів?
8. Перелічіть особливості CISC-архітектури.
9. Перелічіть особливості RISC-архітектури.
10. Перелічіть особливості VLIW-архітектури.

11. Перелічіть особливості EPIC-архітектури.
12. Які основні функції операційної системи? Чи немає між ними протиріч?
13. Перелічіть основні компоненти операційної системи.
14. Що спільного й у чому відмінності між мережною і розподіленою операційними системами? Яка з них складніша в реалізації і чому?
15. Перелічіть види серверів у клієнт-серверних операційних системах.

Розділ ІІ

АЛГОРИТМІЧНІ МОВИ ТА СИСТЕМИ ПРОГРАМУВАННЯ

Алгоритми та мови програмування

Керування комп'ютером здійснюється відповідно деяким *алгоритмам.* Під алгоритмом розуміють певний опис способу рішення завдання у вигляді кінцевої (за часом) послідовності дій. Для подання алгоритмів у вигляді, який розуміє комп'ютер, використовують *мови програмування.* Спочатку розробляється алгоритм дій, потім алгоритм записується на одній з таких мов. У результаті отримують текст програми - повний, закінчений і детальний опис алгоритму мовою програмування. Текст програми переводиться в машинний код або виконується під управлінням спеціальних службових програм, які називаються *трансляторами*.

Самому написати програму в машинному коді досить складно, причому складність різко зростає зі збільшенням розміру програми й трудомісткістю рішення потрібного завдання. Умовно можна вважати, що машинний код прийнятний, якщо розмір програми не перевищує декількох десятків байтів і нема потреби в операціях ручного уведення/виведення даних. Сьогодні практично всі програми створюються за допомогою мов програмування. Теоретично програму можна написати засобами звичайної людської (природної) мови - це називається програмуванням на *метамові* (подібний підхід зазвичай використовується на етапі складання алгоритму), але автоматично перевести таку програму в машинний код поки неможливо, через високу неоднозначність природної мови.

Мови програмування - штучні мови. Від природних вони відрізняються обмеженим числом «слів», значення яких зрозуміло транслятору, і дуже суворими правилами запису команд *(операторів).* Сукупність подібних вимог утворює *синтаксис* мови програмування, а *зміст* кожної команди й інших конструкцій мови - її *семантику.*

Порушення форми запису програми приводить до того, що транслятор не зрозуміє призначення оператору й видає повідомлення про синтаксичну помилку, а правильно написане, але не відповідаюче алгоритму використання команд мови приводить до семантичних помилок (логічних помилок або помилок часу виконання). Процес пошуку помилок у програмі називається *тестуванням,* процес усунення помилок - *налагодженням.*

Компілятори й інтерпретатори

За допомогою мови програмування створюється не готова програма, а тільки її текст, що описує раніше розроблений алгоритм. Щоб одержати працюючу програму, треба цей текст або автоматично перевести в машинний код (для цього служать *програми-компілятори)* і потім використати окремо від вихідного тексту, або відразу виконувати команди мови, зазначені в тексті програми (цим займаються *програми-інтерпретатори).*

Інтерпретатор бере черговий оператор мови з тексту програми, аналізує його структуру й потім відразу виконує (звичайно після аналізу оператор транслюється в деяке проміжне подання або навіть машинний код для більш ефективного подальшого виконання). Тільки після того як поточний оператор успішно виконаний, інтерпретатор перейде до наступного. При цьому якщо той самий оператор повинен виконуватися в програмі багаторазово, інтерпретатор щораз буде виконувати його так, начебто зустрів уперше. Внаслідок цього, програми, у яких потрібно здійснювати великий обсяг повторюваних обчислень, можуть працювати повільно. Крім того, для виконання такої програми на іншому комп'ютері там також повинен бути встановлений інтерпретатор - адже без нього текст програми є просто набором символів.

По-іншому, можна сказати, що інтерпретатор моделює якусь віртуальну обчислювальну машину, для якої базовими інструкціями є не елементарні команди процесора, а оператори мови програмування.

Компілятори повністю обробляють весь текст програми (він іноді називається *вихідний код).* Вони переглядають його в пошуках синтаксичних помилок (іноді кілька разів), виконують певний значеннєвий аналіз і потім автоматично переводять *(транслюють)* на машинну мову - генерують машинний код. Нерідко при цьому виконується *оптимізація* за допомогою набору методів, що дозволяють підвищити швидкодію програми (наприклад, за

допомогою інструкцій, орієнтованих на конкретний процесор, шляхом виключення непотрібних команд, проміжних обчислень і т.д.). У результаті закінчена програма виходить більш компактною та ефективною, працює в сотні разів швидше програми, виконуваної за допомогою інтерпретатора, і може бути перенесена на інші комп'ютери із процесором, що підтримує відповідний машинний код.

Основний недолік компіляторів - трудомісткість трансляції мов програмування, орієнтованих на обробку даних складної структури, часто заздалегідь невідомої або динамічно мінливої під час роботи програми. Тоді в машинний код доводиться вставляти безліч додаткових перевірок, аналізувати наявність ресурсів операційної системи, динамічно їх захоплювати й звільняти, формувати й обробляти в пам'яті комп'ютера складні об'єкти, що на рівні жорстко заданих машинних інструкцій здійснити досить важко, а для ряду завдань практично неможливо.

За допомогою інтерпретатора, навпаки, припустимо в будь-який момент припинити роботу програми, дослідити вміст пам'яті, організувати діалог з користувачем, виконати як завгодно складні перетворення даних і при цьому постійно контролювати стан навколишнього програмно-апаратного середовища, завдяки чому досягається висока надійність роботи. Інтерпретатор при виконанні кожного оператора перевіряє безліч характеристик операційної системи й при необхідності максимально докладно інформує розроблювача про виникаючі проблеми. Крім того, інтерпретатор дуже зручний для використання в якості інструменту вивчення програмування, тому що дозволяє зрозуміти принципи роботи будь-якого окремого оператора мови.

У реальних системах програмування перемішані технології і компіляції, і інтерпретації. У процесі налагодження програма може виконуватися по кроках, а результуючий код не обов'язково буде машинним - він навіть може бути вихідним кодом, написаним на іншій мові програмування (це істотно спрощує процес трансляції, але вимагає компілятора для кінцевої мови), або проміжним машинно-незалежним кодом абстрактного процесора, що у різних комп'ютерних архітектурах стане виконуватися за допомогою інтерпретатора або компілюватися у відповідний машинний код.

Рівні мов програмування

Різні типи процесорів мають різні набори команд. Якщо мова програмування орієнтована на конкретний тип процесора й ураховує його особливості, то він називається *мовою програмування низького рівня*. У цьому випадку «низький рівень» не значить «поганий». Мається на увазі, що оператори мови близькі до машинного коду й орієнтовані на конкретні команди процесора.

Мовою найнижчого рівня є *мова асемблера,* що просто представляє кожну команду машинного коду, але не у вигляді чисел, а за допомогою символьних умовних позначок, які називають *мнемоніками.* Однозначне перетворення однієї машинної інструкції в одну команду асемблера називається *транслітерацією.* Тому що набори інструкцій для кожної моделі процесора відрізняються, конкретній комп'ютерній архітектурі відповідає своя мова асемблера, і написана на ній програма може бути використана тільки в цьому середовищі.

С допомогою мов низького рівня створюються дуже ефективні й компактні програми, тому що розроблювач одержує доступ до всіх можливостей процесора. З іншого боку, потрібно дуже добре розуміти пристрої комп'ютера, ускладнюється налагодження великих додатків, а результуюча програма не може бути перенесена на комп'ютер з іншим типом процесора. Подібні мови звичайно застосовують для написання невеликих системних додатків, драйверів пристроїв, модулів стикування з нестандартним устаткуванням, коли найважливішими вимогами стають компактність, швидкодія й можливість прямого доступу до апаратних ресурсів. У деяких областях, наприклад у машинній графіці, мовою асемблера пишуться бібліотеки, що ефективно реалізують потребуючі інтенсивних обчислень алгоритми обробки зображень.

Мови програмування високого рівня значно ближчі й зрозуміліші людині, ніж комп'ютеру. Особливості конкретних комп'ютерних архітектур в них не враховуються, тому створювані програми на рівні вихідних текстів легко переносяться на інші платформи, для яких створений транслятор цієї мови. Розробляти програми на мовах високого рівня за допомогою зрозумілих і потужних команд простіше, а помилок при створенні програм допускається набагато менше.

Покоління мов програмування

Мови програмування прийнято ділити на п'ять *поколінь*. У перше покоління входять мови, створені на початку 50-х років, коли перші комп'ютери тільки з'явилися на світ. Це була перша мова асемблера, створена за принципом «одна інструкція - один рядок».

Розквіт другого покоління мов програмування прийшовся на кінець 50-х - початок 60-х років. Тоді був розроблений символічний асемблер, у якому з'явилося поняття змінної. Він став першою повноцінною мовою програмування. Завдяки його виникненню помітно зросли швидкість розробки й надійність програм.

Появу третього покоління мов програмування прийнято відносити до 60-м років. У цей час народилися універсальні мови високого рівня, з їхньою допомогою вдається вирішувати завдання з будь-яких галузей. Такі якості нових мов, як відносна простота, незалежність від конкретного комп'ютера й можливість використання потужних синтаксичних конструкцій, дозволили різко підвищити продуктивність праці програмістів. Зрозуміла більшості користувачів структура цих мов залучила до написання невеликих програм (як правило, інженерного або економічного характеру) значне число фахівців з некомп'ютерних галузей. Переважна більшість мов цього покоління успішно застосовується й сьогодні.

З початку 70-х років по теперішній час триває період мов четвертого покоління. Ці мови призначені для реалізації великих проектів, підвищення їхньої надійності й швидкості створення. Вони звичайно орієнтовані на спеціалізовані галузі застосування, де гарних результатів можна домогтися, використовуючи не універсальні, а проблемно-орієнтовані мови, що оперують конкретними поняттями вузької предметної галузі. Як правило, у ці мови вбудовуються потужні оператори, що дозволяють одним рядком описати таку функціональність, для реалізації якої на мовах молодших поколінь потрібні були б тисячі рядків вихідного коду.

Народження мов п'ятого покоління відбулося в середині 90-х років. До них відносяться також системи автоматичного створення прикладних програм за допомогою візуальних засобів розробки, без знання програмування. Головна ідея, що закладається в ці мови, - можливість автоматичного формування результуючого тексту на універсальних мовах програмування (який потім потрібно відкомпілювати). Інструкції ж уводяться в комп'ютер у максимально наочному виді за допомогою методів, найбільш зручних для людини, не знайомої із програмуванням.

Fortran (Фортран). Це перша мова, що компілюється, створена Джимом Бекусом в 50-і роки. Програмісти, що розробляли програми винятково на асемблері, виражали серйозний сумнів у можливості появи високопродуктивної мови високого рівня, тому основним критерієм при розробці компіляторів Фортрану була ефективність коду, що виконується. Хоча у Фортрані вперше був реалізований ряд найважливіших понять програмування, зручність створення програм була принесена у жертву можливості одержання ефективного машинного коду. Однак для цієї мови була створена величезна кількість бібліотек, починаючи від статистичних комплексів і закінчуючи пакетами управління супутниками, тому Фортран продовжує активно використовуватися в багатьох організаціях, зараз навіть ведуться роботи над черговим стандартом Фортрану F2k. Є стандартна версія Фортрану HPF (High Performance Fortran) для паралельних суперкомп'ютерів з безліччю процесорів.

Cobol (Кобол) - мова, яка теж компілюється. Призначена вона для застосування в економічній галузі. Мова розроблена на початку 60-х років і відрізняється великою «багатослівністю» - її оператори іноді виглядають як звичайні англійські фрази. У Коболі були реалізовані дуже потужні засоби роботи з великими обсягами даних, що зберігаються на різних зовнішніх носіях. На цій мові створено багато додатків, які активно експлуатуються й сьогодні. Досить сказати, що найбільшу зарплату в США одержують програмісти на Коболі.

Algol (Алгол). Мова яка **к**омпілюється, створена в 1960 році. Вона була покликана замінити Фортран, але через більш складну структуру не одержала широкого поширення. В1968 році була створена версія Алгол 68, яка по своїх можливостях і сьогодні випереджає багато мов програмування, однак через відсутність досить ефективних комп'ютерів для неї не вдалося вчасно створити гарні компілятори.

Pascal (Паскаль). мова Паскаль, створена наприкінці 70-х років засновником безлічі ідей сучасного програмування Никлаусом Виртом, багато в чому нагадує Алгол, але в ній більш строгі вимоги до структури програми і є можливості, які дозволяють успішно застосовувати її при створенні великих проектів.

Basic (Бейсик). Для цієї мови є й компілятори, і інтерпретатори, а по популярності віна посідає перше місце у світі. Мова створена в 60-х роках як навчальна мова і є дуже простою у вивченні.

Си. Дана мова була створена у лабораторії Bell і спочатку не розглядалася як масова. Віна планувався для заміни асемблера, щоб мати можливість створювати настільки ж ефективні й компактні програми, і в той же час не залежати від конкретного типу процесора.

Си багато в чому схожа на Паскаль і має додаткові засоби для прямої роботи з пам'яттю *(покажчики).* На цій мові в 70-і роки написано безліч прикладних і системних програм і ряд відомих операційних систем (Unix)

C++ (Си++). Си++ - це об'єктно-орієнтоване розширення мові Си, створене Бьярном Страуструпом у 1980 році. Безліч нових потужних можливостей, що дозволили різко підвищити продуктивність програмістів, наклалися на успадковану від мови Си певну низкорівневість, у результаті чого створення складних і надійних програм зажадало від розроблювачів високого рівня професійної підготовки.

Java (Джава, Ява). Ця мова була створена компанією Sun на початку 90-х років на основі Си++. Віна покликана спростити розробку додатків на основі Си++ шляхом виключення з неї усіх низкорівневих можливостей. Але головна *особливість* цієї мови - компіляція не в машинний код, а в платформно-незалежний байт-код (кожна команда займає один байт). Цей байт-код може виконуватися за допомогою інтерпретатора - віртуальної Java-машини (Java Virtual Machine), версії якої створені сьогодні для будь-яких платформ. Завдяки наявності безлічі Java-машин програми на Java можна переносити не тільки на рівні вихідних текстів, але й на рівні двійкового байта-коду, тому по популярності мова Ява, сьогодні посідає друге місце у світі після Бейсика.

Особлива увага в розвитку цієї мови приділяється двом напрямкам: підтримці всіляких мобільних пристроїв і мікрокомп'ютерів, що вбудовують у побутову техніку (технологія Jini) і створенню платформно-незалежних програмних модулів, здатних працювати на серверах у глобальних і локальних мережах з різними операційними системами (технологія Java Beans). Поки основний недолік цієї мови - невисока швидкодія, тому що мова Ява інтерпретується.

Системи програмування

Засоби створення програм

У самому загальному випадку для створення програми обраною мовою програмування потрібно мати наступні компоненти.

1. *Текстовий редактор.* Так як текст програми записується за допомогою ключових слів, що звичайно походять від слів англійської мови, і набору стандартних символів для запису всіляких операцій, то формувати цей текст можна в будь-якому редакторі, одержуючи в результаті текстовий файл із *вихідним текстом* програми. Краще використовувати спеціалізовані редактори, які орієнтовані на конкретну мову програмування й дозволяють у процесі уведення тексту виділяти ключові слова й ідентифікатори різними кольорами й шрифтами. Подібні редактори створені для всіх популярних мов і додатково можуть автоматично перевіряти правильність синтаксису програми безпосередньо під час її уведення.
2. Вихідний текст за допомогою *програми-компілятора* переводиться в машинний код. Якщо виявлені синтаксичні помилки, то результуючий код не буде створюватися. На цьому етапі вже можливе одержання готової програми, але найчастіше в ній не вистачає деяких компонентів, тому компілятор звичайно видає проміжний *об'єктний код* (двійковий файл, зі стандартним розширенням .OBJ)
3. Похідний текст великої програми складається, як правило, з декількох *модулів* (файлів з вихідними текстами), тому що зберігати всі тексти в одному файлі незручно - у них складно орієнтуватися. Кожний модуль компілюється в окремий файл із об'єктним кодом, які потім треба об'єднати в одне ціле.

   Крім того, до них треба додати машинний код підпрограми, що реалізують різні стандартні функції (наприклад, що обчислюють математичні функції sin або ln). Такі функції утримуються в *бібліотеках* (файлах зі стандартним розширенням .LIB), які постачаються разом з компілятором. Сгенерований код модулів і підключені до нього стандартні функції треба не просто об'єднати в одне ціле, а виконати таке об'єднання з урахуванням вимог

операційної системи, тобто одержати на виході програму, що відповідає певному формату.

Об'єктний код обробляється спеціальною програмою - *редактором зв'язків* або *збирачем,* що виконує зв'язування об'єктних модулів і машинного коду стандартних функцій, Знаходячи їх у бібліотеках, він формує на виході працездатний додаток - *здійсненний код* для конкретної платформи.

Якщо з якихось причин один з об'єктних модулів або потрібна бібліотека не виявлені (наприклад, неправильно зазначений каталог з бібліотекою), то збирач повідомляє про помилку й готова програма не виходить.

4. *Здійсненний код* - це закінчена програма, яку можна запустити на будь-якому комп'ютері, де встановлена операційна система, для якої ця програма створювалася. Як правило, підсумковий файл має розширення .EXE або .COM.

Інтегровані системи програмування

Отже, для створення програми потрібні:

- текстовий редактор;
- компілятор;
- редактор зв'язків;
- бібліотеки функцій.

Як правило, у стандартну поставку входять як мінімум три останніх компоненти, але гарна *інтегрована система* містить у собі й спеціалізований текстовий редактор, причому майже всі етапи створення програми в ній автоматизовані: після того як вихідний текст уведений, його компіляція й зборка виконуються одним натисканням клавіші. Це дуже зручно, тому що не вимагає ручного настроювання безлічі параметрів запуску компілятора й редактора зв'язків, указування їм потрібних файлів вручну й т.д. Процес компіляції звичайно демонструється на екрані: показується, скільки рядків вихідного тексту вже скомпільовано, або видаються повідомлення про знайдені помилки.

У сучасних інтегрованих системах є ще один компонент - *відладчик,* що дозволяє аналізувати роботу програми під час її виконання. З його допомогою можна послідовно виконувати окремі оператори вихідного тексту *по кроках,* спостерігаючи при цьому, як міняються значення різних змінних. Без відладчика розробити великий додаток дуже складно.

Середовища швидкого проектування

В останні кілька років у програмуванні (особливо в програмуванні для операційної системи Windows) намітився так званий *візуальний підхід.* До цього серйозною перешкодою для розробки графічних додатків була складність створення різних елементів управління й контроль за їхньою роботою. Досить глянути на вікно будь-якої Windows-програми. У ньому є безліч стандартних елементів управління (кнопки, пункти меню, списки, перемикачі й т.д.). Дуже трудомістко вручну описувати процес створення цих елементів відповідно до вимог Windows, на око визначати координати, відслідковувати їхній стан за допомогою спеціальних команд. Наприклад, для простої програми, що складає два числа, буде потрібно один оператор (один рядок вихідного тексту) для виконання потрібного обчислення й сотні рядків коду для підготовки додатка до роботи в Windows, створення кнопки й пари полів уведення.

Цей процес автоматизований у *середовищах швидкого проектування* (Rapid Application Development, *RAD-середовища).* Всі необхідні елементи оформлення й управління створюються й обслуговуються не шляхом ручного програмування, а за допомогою готових візуальних *компонентів,* які за допомогою миші «перетаскуються» у проектоване вікно. Їхні властивості й поводження потім настроюються за допомогою простих редакторів, що візуально показують характеристики відповідних елементів. При цьому допоміжний вихідний текст програми, відповідальний за створення й роботу цих елементів, генерується *RAD-середовищем* автоматично, що дозволяє зосередитися тільки на логіці розв'язуваного завдання. У результаті програмування багато в чому заміняється на проектування - подібний підхід називається ще *візуальним програмуванням.*

Компоненти досить легко створювати самостійно, тому у світі сьогодні поширюються тисячі безкоштовних і платних компонентів для найбільш відомих *RAD-середовищ,* з них формуються бібліотеки компонентів – *об'єктні репозитарії.* Компоненти виступають у ролі «будівельних цеглинок», що дозволяють збирати готовий додаток з багатими можливостями, написавши всього десяток рядків вихідного коду, і такий *компонентний підхід* до створення програм уважається дуже перспективним, тому що без зайвих зусиль і на законних підставах допускає *повторне використання* чужої праці.

Основні системи програмування

З універсальних мов програмування сьогодні найбільш популярні наступні:

Бейсик (Basic) - для освоєння вимагає початкової підготовки (загальноосвітня школа);

Паскаль (Pascal) - вимагає спеціальної підготовки (школа з поглибленим вивчанням предмета або загально технічні ВНЗ);

Си++ (C++), Ява (Java) - вимагають професійної підготовки (спеціалізовані середні й вищі навчальні заклади).

Для кожної із цих мов програмування сьогодні є чимало систем програмування, що випускаються різними фірмами й орієнтовані на різні моделі ПК і операційні системи. Найбільш популярні наступні візуальні середовища швидкого проектування програм для Windows:

- Basic: Microsoft Visual Basic
- Pascal: Borland Delphi
- C++: Borland C++Bulider, Microsoft Visual Studio
- Java: Symantec Cafe

Для розробки серверних і розподілених додатків можна використати систему програмування Microsoft Visual C++, продукти фірми Inprise під маркою Borland, практично будь-які засоби програмування на Java.

Надалі будуть розглядатися можливості, характерні для Microsoft Visual Basic.

Резюме

Для подання способу рішення завдання у вигляді, який зрозумілий комп'ютеру, використовують мови програмування. Далі текст програми переводиться в машинний код, або виконується під управлінням спеціальних службових додатків, які називаються *трансляторами*.

В розділі розглянуто рівні та покоління мов програмування, зроблений огляд мов програмування високого рівня. Визначені основні засоби створення програм, охарактеризовані інтегровані системи програмування, середовища швидкого проектування, розглянуті найбільш популярні сьогодні універсальні системи програмування.

Контрольні запитання та завдання

1. Що таке мова програмування?
2. Що таке транслятор?
3. Чим відрізняється компіляція від інтерпретації?
4. Поясніть терміни „мова низького рівня” й „мова високого рівня”, чим вони відрізняються?.
5. Розкажіть про покоління мов програмування.
6. Які мови програмування активно використовуються сьогодні?
7. Укажіть основні компоненти, які необхідні для створення програми.
8. Що таке інтегроване середовище програмування?
9. Дати визначення поняттю „візуальне програмування”.
10. Охарактеризуйте основні напрямки розвитку мов програмування.
11. Розкажіть про основні системи програмування.
12. Перелічіть складові інтегрованого середовища розробки програм.

Розділ ІІІ
ОСНОВНІ ПРИНЦИПИ РОЗРОБКИ АЛГОРИТМІВ І ПРОГРАМ

Етапи рішення завдання на ЕОМ

Варто виділити такі етапи підготовки та розв'язання завдань за допомогою ЕОМ: постановка завдання, вибір методу, розробка алгоритму, складання програми, введення її в пам'ять ЕОМ, налагодження програми, її тестування та підготовка документації. Не можна ігнорувати жодного з цих етапів.

Рішення будь-якого завдання на ЕОМ складається з декількох етапів, серед яких варто виділити основні:

1. постановка завдання;
2. формалізація (математична постановка задачі);
3. вибір (або розробка) методу рішення;
4. розробка алгоритму (алгоритмізація);
5. складання програми (програмування), введення її в пам'ять ЕОМ ;
6. налагодження програми, її тестування;
7. обчислення й обробка результатів.

Послідовне виконання вказаних етапів становить повний цикл розробки, налагодження й обчислення програми. Наведений розподіл є умовним, але не варто ігнорувати жодного з цих етапів. Розглянемо найбільш загальні й необхідні етапи. Разом із зазначеними користувач ЕОМ у процесі рішення завдання може виконувати також такі етапи, як вибір мови програмування, опис структури даних, оптимізація програми, тестування, документування й ін.

Постановка завдання. При постановці завдання першорядну увагу треба приділити з'ясуванню кінцевої мети й виробленню загального підходу до досліджуваної проблеми; з'ясуванню, чи існує рішення поставленого завдання й чи єдине воно; вивченню загальних властивостей розглянутого фізичного явища або об'єкта, аналізу можливостей конкретної ЕОМ і даної системи програмування. На цьому етапі потрібне глибоке розуміння сенсу поставленого завдання. Правильно сформулювати завдання іноді не менш складно, ніж її вирішити.

Формалізація. Формалізація, як правило, полягає у побудові математичної моделі розглянутого явища, коли в результаті аналізу сутності завдання визначаються обсяг і специфіка вихідних даних, вводиться система умовних позначок, встановлюється приналежність розв'язуваного завдання до одного з відомих класів завдань і вибирається відповідний математичний апарат. При цьому потрібно вміти сформулювати мовою математики конкретні завдання фізики, механіки, економіки, технології й т. п. Для успішного подолання цього етапу потрібні не тільки солідні відомості з відповідної предметної галузі, але й гарне знання обчислювальної математики, тобто тих методів, які можуть бути використані при рішенні завдання на комп'ютері.

Повна постановка багатьох складних завдань нездійсненна засобами обчислювальної техніки. Тому ці завдання потрібно спрощувати. Грамотне спрощення завдання неможливе без гарного подання про те, які фактори й параметри найбільш важливі для досліджуваного завдання, а які - менш істотні. При цьому дуже важливо знати, яка з можливих розрахункових схем може привести до спрощення обчислювального характеру, обумовлених вибором обчислювального методу. Якщо наявних засобів недостатньо, тоді необхідно розробити новий підхід, нові методи дослідження.

Вибір методу рішення. Після того як визначено математичне формулювання завдання, треба вибрати метод його рішення. Загалом, застосування будь-якого методу приводить до побудови ряду формул і формулювання правил, що визначають зв'язки між цими формулами. Все це розбивається на окремі дії так, щоб обчислювальний процес міг бути виконаний машиною. При виборі методу треба враховувати, *по-перше,* складність формул і співвідношень, пов'язаних з тим або іншим чисельним методом, *по-друге,* необхідну точність обчислень і характеристики самого методу. На вибір методу рішення великий вплив мають смаки й знання самого користувача.

Цей етап - найважливіший у процесі рішення задачі. З ним зв'язані численні невдачі, які є результатом легковажного підходу до помилок обчислень. При рішенні завдання на ЕОМ необхідно пам'ятати, що будь-який одержуваний результат є наближеним! Якщо відомо алгоритм точного рішення, то крім випадкових помилок (збоїв у роботі ЕОМ), можливі помилки, пов'язані з обмеженою точністю подання чисел в ЕОМ. При обчисленнях, що полягають у знаходженні результату із заданим ступенем точності, виникає додаткова погрішність, яку, якщо можливо, оцінюють на

даному етапі (до виходу безпосередньо на ЕОМ). Ця погрішність визначається обраним чисельним методом рішення завдання.

Розробка алгоритму. Даний етап полягає в розкладанні обчислювального процесу на можливі складові частини, установленні порядку їхнього проходження, описі змісту кожної такої частини в тій або іншій формі й наступній перевірці, що повинна показати, чи забезпечується реалізація обраного методу. У більшості випадків не вдається відразу одержати задовільний результат, тому складання алгоритму проводиться методом «спроб і усунення помилок» і для одержання остаточного варіанту потрібно кілька кроків корекції й аналізу.

Як правило, у процесі розробки алгоритм проходить кілька етапів деталізації. Спочатку складається укрупнена схема алгоритму, у якій відбиваються найбільш важливі й істотні зв'язки між досліджуваними процесами (або частинами процесу). На наступних етапах розкриваються (деталізуються) виділені на попередніх етапах частини обчислювального процесу, що мають деяке самостійне значення. Крім того, на кожному етапі деталізації виконується багаторазова перевірка й виправлення (відпрацьовування) схеми алгоритму. Подібний підхід дозволяє уникнути можливих помилкових рішень.

Орієнтуючись на великоблочну структуру алгоритму, можна швидше й простіше розробити кілька різних його варіантів, провести їхній аналіз, оцінку й вибрати найкращий (оптимальний).

Ефект поетапної деталізації алгоритму багато в чому залежить від того, як здійснюється його структуризація: розчленовування алгоритмічного процесу на складові частини, що повинне визначатися не сваволею користувача (програміста), а внутрішньою логікою самого процесу. Кожний елемент великоблочної схеми алгоритму повинен бути максимально самостійним і логічно завершеним у такому ступені, щоб подальшу його деталізацію можна було виконувати незалежно від деталізації інших елементів. Це спрощує процес проектування алгоритму й дозволяє здійснювати його розробку врозділ одночасно кількома виконавцями.

У процесі розробки алгоритму можуть використовуватися різні способи його опису, що відрізняються за простотою, наочністю, компактністю, ступенем формалізації, орієнтацією на машинну реалізацію й іншими показниками. У практиці програмування найбільшого поширення набули:

1. словесний запис алгоритмів;
2. схеми алгоритмів;
3. псевдокод (формальні алгоритмічні мови);

4. структурограми (діаграми Нассі - Шнейдермана) .

Розробка алгоритмів є в значній мірі творчим, евристичним процесом і, як правило, вимагає великої ерудиції, винахідливості, нестандартних і нетрадиційних підходів до рішення завдання.

Складання програми. Подання алгоритму у формі, що допускає введення в машину, переклад на машинну мову є завданнями етапу складання програми (програмування). Тобто розроблений алгоритм завдання необхідно викласти мовою, що буде зрозуміла ЕОМ безпосередньо або після попереднього машинного перекладу. Від вибору мови програмування залежить процес налагодження програми, під час якого програма набуває остаточного робочого вигляду.

Налагодження програми. Складання програми являє собою трудомісткий процес, що вимагає від виконавця напруженої уваги. Практика показує, що в обчисленнях варто уникати поспішності й дотримуватися золотого правила: «краще менше, та краще». Але на попередніх етапах стільки можливостей припуститися помилки, і як би ми ретельно не діяли, спочатку складена програма звичайно містить помилки. Машина або не може дати відповіді, або наводить неправильне рішення.

Налагодження починається з того, що програма, акуратно записана на бланку, перевіряється безпосередньо особою, що здійснила підготовку й програмування завдання. З'ясовується правильність написання програми, виявляються змістовні й синтаксичні помилки й т.п. Потім програма вводиться у пам'ять ЕОМ і помилки, що залишилися непоміченими, виявляються вже безпосередньо за допомогою машини.

Досвідчений користувач ЕОМ знає, що необхідний діючий контроль над процесом обчислень, який дозволяє вчасно виявляти й запобігати помилки. Для цього використовуються різного роду інтуїтивні міркування, правдоподібні міркування й контрольні формули. Користувач - початківець часто вважає налагодження зайвим, а одержання контрольних точок - неприємною додатковою роботою. Однак дуже скоро він переконується, що пошук пропущеної помилки вимагає значно більшого часу, ніж час, витрачений на контроль.

Гарантією правильності рішення, наприклад, може служити:

a. перевірка виконання умов завдання (наприклад, для алгебраїчного рівняння знайдені коріння підставляються

у вихідне рівняння й перевіряються розходження лівої й правої частин);

б. якісний аналіз завдання;

в. перерахування (по можливості іншим методом).

Для деяких складних за структурою програм процес налагодження може зажадати значно більше машинного часу, ніж саме рішення на ЕОМ, тому що погано сплановані процеси алгоритмізації, програмування і налагодження приводять до помилок, які можуть бути виявлені лише після багаторазових перевірок.

Обчислення й обробка результатів. Тільки після того як з'явиться повна впевненість, що програма забезпечує одержання правильних результатів, можна приступати безпосередньо до розрахунків по програмі. Безпосереднє рішення завдання на ЕОМ не вимагає обов'язкової участі користувача. Ця робота виконується оператором ЕОМ. Порядок роботи на машині при рішенні завдання докладно описується в інструкції до програми. Після завершення розрахунків наступає етап використання результатів обчислень у практичній діяльності або, як говорять, етап впровадження результатів. Інтерпретація результатів обчислень знову належить до тієї предметної галузі знань, звідки виникло завдання.

Словесна форма запису алгоритмів

Спочатку для запису алгоритмів користувалися засобами звичайної мови, але з ретельно відібраним набором слів і фраз, що не допускає повторень, синонімів, двозначностей, зайвих слів. Крім того, приймалися певні угоди про форму запису, порядок виконання дій, допускалося використання математичних символів.

Розглянемо як приклад *алгоритм Евкліда.* Завдання, розв'язуване за допомогою цього алгоритму, формулюється так: дані два цілих позитивних числа, знайти їхній найбільший загальний дільник (НЗД).

Як відомо, рішення цього завдання може бути отримане послідовним розподілом спочатку більшого числа на менше, потім меншого числа на отриманий залишок, першого залишку на другий залишок і т.д. доти, поки в залишку не вийде нуль. Останній по рахунку дільник і буде шуканим результатом.

Позначимо через *M і N* вихідні цілі числа, прийнявши в якості їхніх початкових значень задані константи. Зведемо розподіл до повторного вирахування. Тоді алгоритм може бути сформульований у такий спосіб:

- Якщо M <> N, то перейти до п. 2, інакше перейти до п. 5.
- Якщо M > N, то перейти до п. 3, інакше перейти до п. 4.
- Від M відняти N і далі цю різницю вважати значенням M. Перейти до п. 1.
- Від N відняти M и далі вважати цю різницю значенням N. Перейти до п. 1.
- Вважати, що НЗД дорівнює M. Закінчення.

Проілюструємо використання алгоритму для знаходження найбільшого загального дільника двох чисел: 95 і 60. Приймемо, наприклад, перше з них - 95 за початкове значення M, *а* друге 60 - за початкове значення N. Тоді послідовність виконаних пунктів алгоритму, а також значень M та N, що змінюються, будуть такими, як у табл. 3.1.

Таблиця 3.1

Таблиця значень алгоритму Евкліда

M	**N**	**Номера виконуваних пунктів**
95	60	1,2,3
35	60	1,2,4
35	25	1,2,3
10	25	1,2,4
10	15	1,2,4
10	5	1,2,3
5	5	1,5

Дійсно, виконання операцій по п. 1 приводить нас до п. 2. Через те що числові значення даних такі, що M > N, то після виконання п. 3 числове значення M стало дорівнювати 35, а N, як і раніше, дорівнює 60. Відповідно до вказівки п. 3 повертаємося до п. 1. Знову результатом його виконання буде перехід до п. 2. Тому що при новому значенні M співвідношення M > N вже несправедливе, то переходимо до п. 4 і т.д.

Оскільки алгоритм призначений для подальшої реалізації на ЕОМ, то обов'язково треба ввести покажчики початку й кінця, які використовуються також для виділення деяких самостійних частин алгоритму. Разом із цим запис повинен бути доповнений вказівками про введення в ЕОМ початкових значень M і *N* і виведення отриманого результату у надрукованому вигляді. Замість фраз «від *M* відняти *N* і далі цю різницю вважати значенням M – N» або «від N відняти M і далі вважати цю різницю значенням N» можна використовувати записи «M : = M – N» і « N : = N — M». Це дозволяє більш компактно записати п. 3, 4 і 5. У деяких пунктах можуть бути відсутні явні вказівки, куди перейти після їхнього виконання. У цих випадках мається на увазі перехід до пункту з порядковим номером на одиницю більшим, ніж у того, що виконується. Введені спеціальні угоди дозволяють *алгоритм Евкліда* представити більш компактним словесним записом:

Початок

1. Увести (M, N).
2. Якщо M ≠ N, то перейти до п. 3, інакше перейти до п.6.
3. Якщо M > N, то перейти до п. 4, інакше перейти до п.5.
4. M: = M — N; перейти до п. 2.
5. N : = N — M; перейти до п. 2.
6. НЗД : = M.
7. 7.Друкувати (НЗД).

Кінець

Розглянемо ще один приклад — алгоритм відшукання мінімуму й максимуму в будь-якій кінцевій послідовності з *n* дійсних чисел a_1, a_2, ..., a_i ...., a_n. При невеликій кількості чисел досить швидкого погляду, щоб указати максимум і мінімум. Однак якщо *n* велике, завдання ускладнюється. У цьому легко переконатися, якщо спробувати відшукати максимум і мінімум серед декількох сотень багато розрядних чисел. Тому необхідно дотримуватися певної системи. Наприклад, взяти як початкове значення як для максимуму, так і для мінімуму перше число, далі послідовно перебирати числа й порівнювати кожне з них із установленим на даний момент значенням максимуму. Якщо чергове число перевищує максимум, вважати його новим значенням максимуму (колишнє значення «забути»), після чого можна переходити до нового числа. Якщо аналізоване число не більше

максимуму, то зрівняти його із встановленим до даного моменту значенням мінімуму. Якщо число виявляється меншим мінімуму, уважати його новим значенням мінімуму й перейти до наступного числа; якщо число не менше мінімуму, просто переходити до аналізу наступного числа. Перебравши в такий спосіб всі числа, одержимо остаточні значення максимуму й мінімуму. Викладені правила можна представити у вигляді словесного запису:

Початок

1. Увести (ai, i= 1,2, ..., n).
2. min: = a1; max:= a1
3. i: = 2.
4. Якщо ai > max, то перейти до п. 5, інакше перейти до п. 6.
5. max: = ai, перейти до п. 8.
6. Якщо ai < min, то перейти до п. 7, інакше перейти до п. 8.
7. min : = ai
8. i: = i + 1.
9. Якщо i< п, то перейти до п. 4, інакше перейти до п. 10.
10. Друкувати (max, min).

Кінець

Схеми алгоритмів

Схема - це графічне подання алгоритму, доповнене елементами словесного запису. Кожний пункт алгоритму відображається на схемі деякою геометричною фігурою-блоком (блоковим символом), причому різним за типом виконуваних дій блокам відповідають різні геометричні фігури, зображувані за ДСТ. Правила виконання схем алгоритмів регламентує ДЕРЖСТАНДАРТ 19.002-80 (який повністю відповідає міжнародному стандарту ИСО 2636-73). Застосовувані графічні символи, що відбивають основні операції процесу обробки даних, установлює ДЕРЖСТАНДАРТ 19.003-80 (позначення символів відповідає міжнародному стандарту ИСО 1028-73). У табл.3..2 наведені найбільш часто вживані блоки й дані пояснення до них.

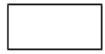
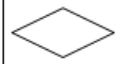
Графічні символи на схемах з'єднуються лініями потоку інформації. Основний напрямок потоку інформації йде зверху вниз і зліва направо (стрілки на лініях можуть не вказуватися). В інших випадках застосування стрілок обов'язкове. На рис. 1.1 наведені приклади умовних графічних позначень на схемах. Стосовно блоку лінії потоку можуть бути вхідними або вихідними. Кількість вхідних ліній для блоку принципово не обмежена. Вихідна лінія може бути тільки одна. Виключення становлять логічні блоки, що мають не менше двох вихідних ліній потоку, кожна з яких відповідає одному з можливих варіантів перевірки логічної умови (рис. 1.1, в, г), а також блоки модифікації.

При великій кількості пересічних ліній, великій їхній довжині й багаторазовим змінам напрямку схема стає малонаочною. У цих випадках допускається розривати лінії потоку інформації, розміщаючи на обох кінцях розриву спеціальний символ «з'єднувач» (рис. 3.1, *в).* Усередині поля з'єднувачів, що відзначають розрив однієї й тієї ж лінії, ставиться однакове маркування окремою буквою або буквено-цифровою координатою блоку, до якого підходить лінія потоку. Якщо схема розташовується на декількох аркушах, перехід ліній потоку з одного аркуша на інший позначається за допомогою символу «міжсторінковий з'єднувач» (рис. 3.1, *б).* При цьому на аркуші із блоком-джерелом з'єднувач містить номер аркуша й координати блоку-приймача, а на аркуші із блоком-приймачем - номер аркуша й координати блоку-джерела.

Усередині блоків і поруч із ними роблять записи й позначення (для уточнення виконуваних ними функцій) так, щоб їх можна було читати зліва направо і зверху вниз незалежно від напрямку потоку. Наприклад, на рис.3.1 вид 1 і вид 2 читаються ідентично. Порядкові номери блоків проставляють у верхній частині графічного символу в розриві його контуру (рис. 3.1, *б, в, г).*

Таблиця 3.2

Умовні графічні позначення, застосовувані при складанні схем алгоритмів

№ п/п	Назва символу	Символ	Відображувана функція
1.	Блок обчислень (процес)		Обчислювальна дія або послідовність обчислювальних дій
2.	Логічний блок (рішення)		Вибір напрямку виконання алгоритму залежно від деяких умов
3.	Блоки уведення - висновку		Загальне позначення уведення або висновку даних (у незалежності від фізичного носія)
			Висновок даних, носієм яких служить документ (друкувальний пристрій)
4.	Початок - кінець (вхід - вихід)		Початок або кінець програми, вхід або вихід у підпрограмах
5.	Визначений процес (підпрограма)		Обчислення по стандартній підпрограмі або підпрограмі користувача.
6.	Блок модифікацій (заголовок циклу)		Виконання дій, що змінюють пункти алгоритму
7.	З'єднувач		Вказівка зв'язку між перерваними лініями потоку інформації в межах однієї сторінки.
8.	Міжсторінковий з'єднувач		Вказівка зв'язку між частинами схеми, розташованими на різних аркушах.

При виконанні схем алгоритмів необхідно витримувати мінімальну відстань - *3 мм* між паралельними лініями потоків і *5 мм* - між іншими символами. У блоках прийняті розміри: *а= 10, 15, 20 мм; в= 1,5*. Якщо необхідно збільшити розмір схеми, то допускається збільшувати *а* на число, кратне 5.

На рис.3.1 наведена схема алгоритму пошуку максимуму й мінімуму в кінцевій послідовності чисел.

Схема є винятково наочним і простим способом подання алгоритму. При цьому не накладається ніяких обмежень на ступінь деталізації в зображенні алгоритму. Вибір її цілком залежить від програміста. Однак необхідно мати на увазі, що зайво загальний характер схеми небажаний через малу інформативність, а дуже

детальна схема програє в наочності. Тому, особливо для складних і великих алгоритмів, доцільно скласти кілька схем різних рівнів деталізації. Схема 1-го рівня відображає весь алгоритм цілком. Схеми 2-го рівня розкривають логіку окремих блоків схеми 1-го рівня. При необхідності можуть бути складені схеми наступних рівнів з ще більшим ступенем деталізації. Таке покрокове уточнення схеми алгоритму становить сутність методу спадного проектування, що, у свою чергу, є основою структурного програмування.

Структурне програмування

Практика програмування показала необхідність науково обгрунтованої методології розробки й документування алгоритмів і програм. Ця методологія повинна стосуватися аналізу вихідного завдання, поділу його на досить самостійні частини й програмування цих частин по можливості незалежно одна від одної. Такою методологією є *структурне програмування*, що зародилося на початку 70-х років, а в останній час набуло поширення. За своєю суттю воно втілює принципи системного підходу в процесі створення й експлуатації програмного забезпечення ЕОМ. В основу структурного програмування покладені досить прості положення:

1. алгоритм і програма повинні складатися поетапно (по кроках);
2. складне завдання повинне розбиватися на досить прості, частини, що легко сприймаються, кожна з яких має тільки один вхід і один вихід;
3. логіка алгоритму й програми повинна спиратися на мінімальне число досить простих *базових управляючих структур*.

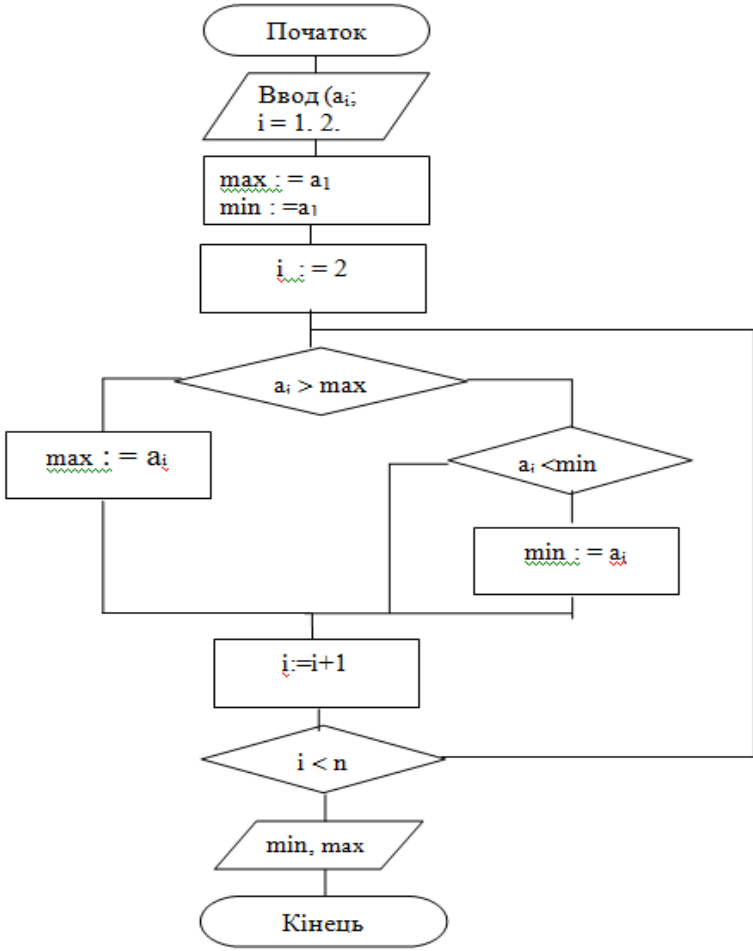


Рис.3.1. Схема алгоритму пошуку максимуму й мінімуму

Використання цих положень дозволяє внести певну систему в працю програміста й отримувати алгоритми (і програми), які зручно читати, легко вивчати й перевіряти. Фундаментом структурного програмування є *теорема про структурування***.** Ця теорема встановлює, що, яким би складним не було завдання, схема відповідної програми завжди може бути представлена з використанням досить обмеженого числа елементарних управляючих структур. Елементарні структури можуть з'єднуватися між собою,

утворюючи більш складні структури, по тих же самих елементарних схемах. Базовими елементарними структурами є структури, зображені на рис.3.2. Вони мають функціональну повноту, тобто будь-який алгоритм може бути реалізований у вигляді композиції цих трьох конструкцій. Кожна з конструкцій має свою назву. Так, перша з них (рис.3.2, а) називається структурою типу послідовність (або просто послідовністю), друга (рис.3.2, б) -—структурою вибору (розгалуженням), третя (рис.3.2, *в)* -—структурою циклу із передумовою. При словесному записі алгоритму зазначені структури мають відповідно такий зміст: «виконати `S1`; виконати `S2`»; «якщо *P*, то виконати `S1`, інакше виконати `S2`»; «доти, поки *P*, виконувати `S`», де *P* — умова; `S, S1, S2` - дії.

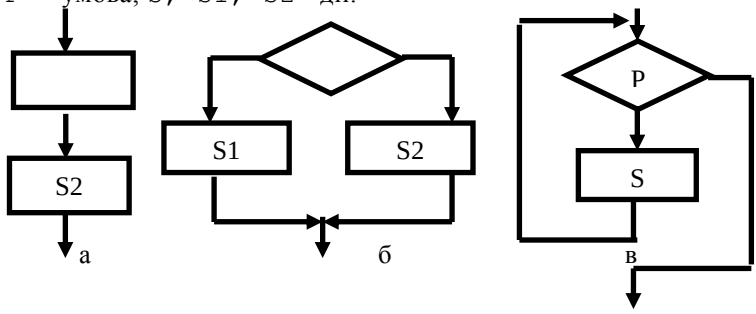


Рис.3.2. Базові елементарні управляючі структури

Розглядаючи схему програми, можна виділити в ній частини (фрагменти), досить прості й зрозумілі за структурою. Подання цих фрагментів укрупненими блоками істотно полегшує сприйняття алгоритму (а надалі й програми) у цілому. На рис.3.3 виділені елементарні структури на схемі алгоритму Евкліда (див. раніше).

Досить часто структурне програмування має на увазі використання більше трьох базисних структур. Стосовно до мови Visual Basic, у якому найбільш повно знайшли своє відбиття ідеї структурного програмування, доцільно при проектуванні алгоритмів додатково використовувати ще чотири елементарні структури: скорочений запис розгалуження (рис.3.4, а); структуру варіанту (рис.3.4, б); структуру повторення або циклу з параметром (рис.3.5, а*);* структуру циклу з передумовою (рис.3.5, б*).* Кожна з представлених структур має один вхід і один вихід. У мові Visual Basic є засоби (оператори), що дозволяють безпосередньо реалізувати в програмі кожну із цих структур, тому правильне використання типових структур у процесі розробки алгоритму забезпечує спрощення наступних етапів рішення завдання на ЕОМ.

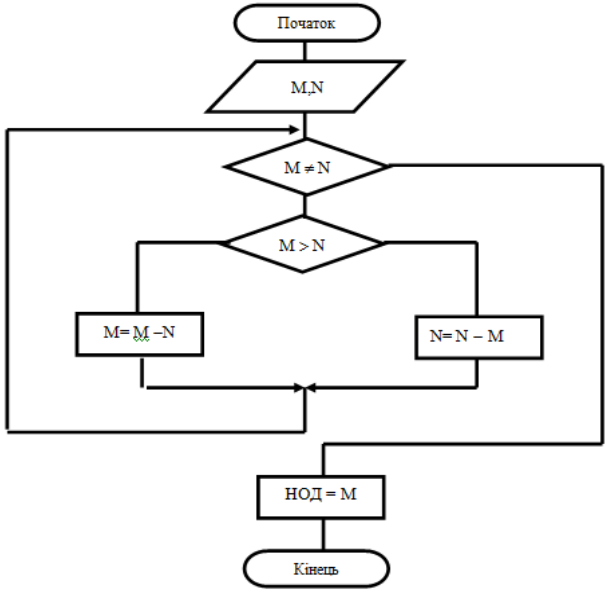
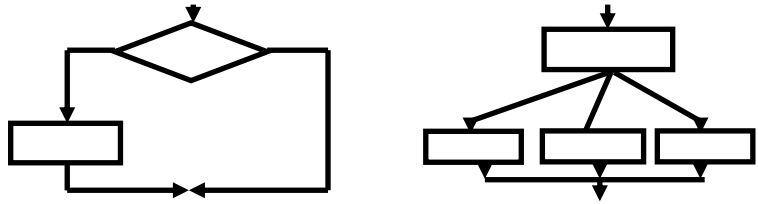
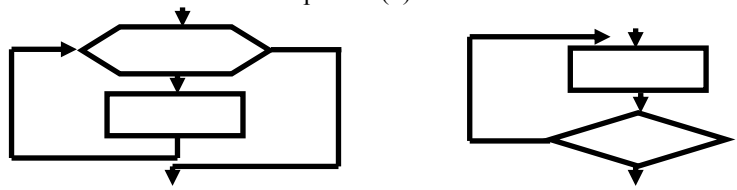


Рис.3.3. Схема алгоритму Евкліда



а б

Рис.3.4. Скорочений запис розгалуження (а); структура варіанта (б)



а б

Рис.3.5. Цикл з параметром (а), цикл з передумовою (б)

Як приклад структурного підходу до проектування схеми алгоритму розглянемо завдання табулювання функції, тобто обчислення деякої функції $z = z(x)$ однієї змінної x, що змінюється від початкового значення x_0 з постійним кроком h_x до кінцевого значення x_n — скорочено позначається $x = x_o(h_x)x_n$.

Нехай функція $z(x)$ має вигляд $z(x) = \dfrac{y^2 + x}{Ln(2 + x^2)}$,

де $y = \begin{cases} \sin(x), & якщо \ \ x \le a, \\ y1(x), & якщо \ \ x > a, \end{cases}$ $y1 = \begin{cases} x2, & если \ \ x < b \\ \sqrt{x}, & если \ \ x \ge b \end{cases}$, при

x_0=-3, h_x=1, x_n=10, a=0, b=3.85

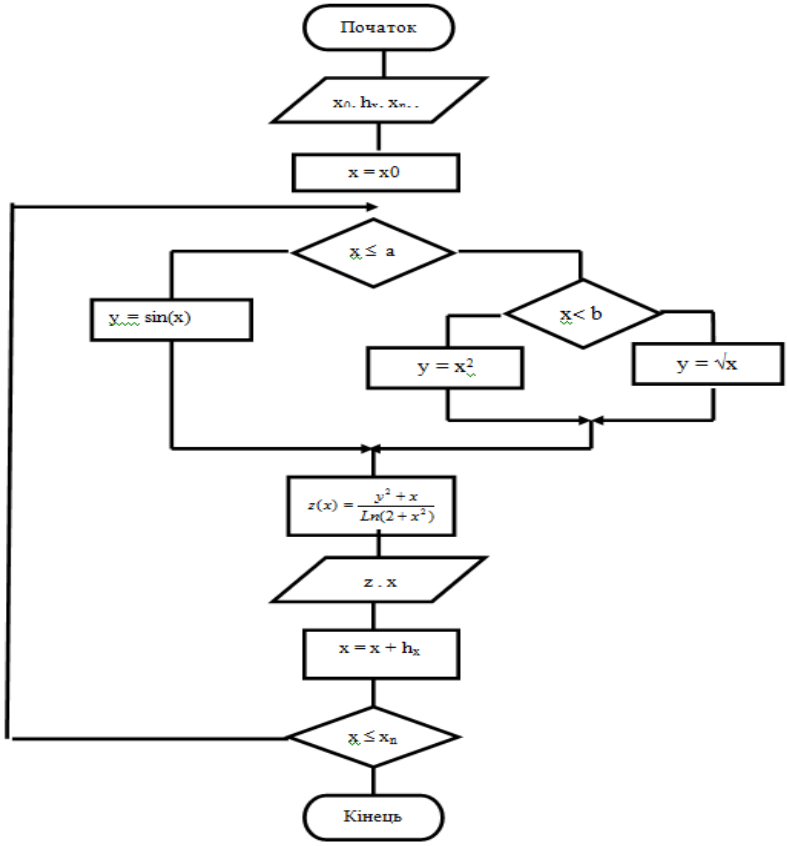


Рис.3.6. Схема алгоритму табулювання функції z = z(x)

Процес спадного проектування схеми алгоритму табулювання функції z = z*(x)* показаний на рис.3.7. На першому рівні деталізації схема відбиває процес табулювання функції в найбільш загальному вигляді. Для цього використовуються структури типів послідовність і цикл із передумовою. Далі здійснюється деталізація блоку *Обчислення функції,* що представляється у вигляді послідовності блоків другого рівня. Оскільки функція *y = y(x), що* входить як аргумент у визначення функції *z = z(x),* є складною, вона визначається структурою, що розгалужується, деталізація якої здійснена на 3-му і 4-му рівнях.

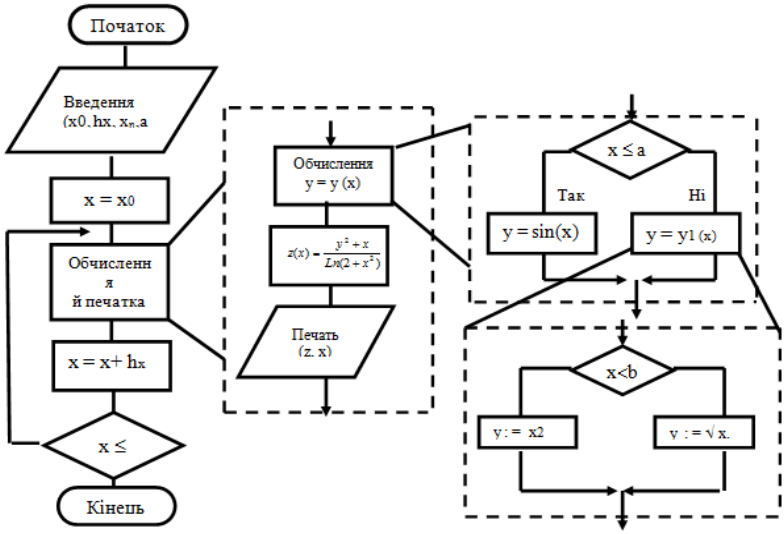


Рис.3.7. Процес спадного проектування схеми алгоритму табулювання функції

Проведену деталізацію блоків, що здійснюють визначення функції *y =y(x),* можна пояснити за допомогою залежностей:

$$y = \begin{cases} \sin(x), & если\ \ x \le a, \\ y1(x), & если\ \ x > a, \end{cases} \qquad \text{де} \quad y1 = \begin{cases} x2\ , & если\ \ x < b \\ \sqrt{x}\ , & если\ \ x \ge b, \end{cases}$$

які легко забезпечують застосування структури розгалуження.

Підставляючи замість блоків, які підлягають деталізації, відповідні структури більше низького рівня, одержуємо остаточну схему алгоритму табулювання функції z = *z(x)*, зображену на рис. 3.7.

Резюме

У цьому розділі розглянуті основні етапи підготовки та розв'язання задач за допомогою ЕОМ: постановка задачі, вибір методу, розробка алгоритму, складання програми, введення її в пам'ять ЕОМ, відладка програми, її тестування та підготовка документації. Розглянуті форми запису та схеми представлення алгоритмів. Особлива увага приділена методології структурного програмування, яка втілює принципи системного підходу в процесі створення й експлуатації програмного забезпечення.

Контрольні запитання та завдання

1. Назвіть основні етапи підготовки та розв'язання задач за допомогою ЕОМ.
2. Прокоментуйте кожний етап розробки програми. Які етапи та в яких ситуаціях можуть бути пропущені?
3. Що враховують при виборі методу вирішення задачі?
4. Назвіть основні форми запису алгоритмів. Наведіть графічні елементи, які застосовуються при складанні схем алгоритмів
5. Що покладено в основу структурного програмування?
6. В чому сенс теореми про структурування?
7. Назвіть основні базові структури структурного програмування.
8. Складіть словесний опис алгоритму знаходження найбільшого значення із чотирьох чисел.
9. Складіть блок-схему алгоритму переведення числа з будь-якої системи числення у десяткову.
10. Складіть блок-схему алгоритму визначення заробітної платні бухгалтера через п'ять років, якщо щороку вона збільшуватиметься на 10 %.

11. Побудувати блок-схему алгоритму піднесення цілого числа до цілого степеня.
12. Побудувати блок-схему алгоритму обчислення факторіала натурального числа.
13. Використовуючи покрокову деталізацію, спроектуйте блок-схему алгоритму обчислення складної функції при заданих значеннях змінних a, b, x:
14. $$y = \begin{cases} \dfrac{(a^2 + b^2)x}{\sqrt{|1 + a|}}, & якщо \ \ x < 0, \\ (a^2 + b^2)^2, & якщо \ x = 0, \\ \dfrac{1nx}{a2 + b2}, & якщо \ 0 < x < 1, \\ \sqrt{x(a2 + b2)}, & якщо \ x \geq 1. \end{cases}$$
15. Складіть блок-схему алгоритму перетворення двовимірного масиву A розміром 3*4 в одномірний масив B, так що ($b_1 = a_{11}$, $b_2 = a_{12}$, $b_3 = a_{13}$, $b_4 = a_{21}$, ...).
16. Записати алгоритм «циклічного» обміну значеннями між трьома однотипними змінними a, b, c (a ← b, b ← c, c ← a).
17. Складіть блок-схему алгоритму визначення наявності, кількості та значень коренів квадратного рівняння.
18. Побудувати блок-схему для визначення типу трикутника (рівносторонній, рівнобедрений, різносторонній) за довжинами його сторін. Якщо трикутник не можна побудувати, слід вивести повідомлення.

Розділ IV
АЛГОРИТМИ ТА АЛГОРИТМІЗАЦІЯ

Алгоритм та його властивості

Алгоритм – це повністю визначена послідовність інструкцій, виконання якої забезпечує отримання вірного результату.

Сучасне значення слова алгоритм дуже схоже зі значенням слів рецепт, процес, метод, спосіб, процедура, програма, проте в слові "алгоритм" є свій додатковий смисловий відтінок. Крім того, що алгоритм – це зведення кінцевого числа правил, які задають послідовність виконання операцій при рішенні тієї або іншої специфічної задачі, для алгоритму характерні властивості:

- ***закінчення*** (фінітність) - алгоритм завжди повинен закінчуватися після кінцевого числа кроків;
- ***визначеність*** - кожен крок алгоритму повинен бути точно визначений;
- ***введення*** - алгоритм може мати деяку кількість вхідних даних, які задані до початку виконання;
- ***вивід*** - алгоритм може мати одну або декілька вихідних величин, які деяким чином співвідносяться з вхідними даними;
- ***ефективність*** алгоритму означає, що всі операції, які необхідно провести в алгоритмі, повинні бути достатньо простими, щоб їх можна було виконати точно і за кінцевий відрізок часу за допомогою олівця і паперу.

Порівняємо поняття "алгоритм" із рецептом кулінарної книги. Рецепти мають властивість фінітності (закінчуються після кінцевого числа кроків), мають вхідні дані (яйця, мука та ін.), вихід (урочистий обід і т.п.), але відрізняються відсутністю визначеності. Візьмемо, наприклад, інструкцію "Додайте щіпку солі". "Щіпка" визначається як кількість, "менша ніж 1/8 чайної ложки". Порція солі, мабуть, достатньо добре визначена, але куди потрібно додати цю сіль (зверху, збоку і т.д.)? Інструкції на зразок: "злегка потрясіть, поки суміш не стане грудкуватою", "підігрійте коньяк в маленькій каструлі" і т.д. – є, можливо, цілком достатніми для досвідченого

кухаря, але алгоритм повинен бути визначений настільки чітко, щоб навіть новачок зміг би слідувати інструкції.

Алгоритми містять тільки покроковий опис дій, які виконуються над даними без опису самих даних. Вони можуть бути записані на мові, яку вживають, на мові програмування, а також за допомогою математичної або іншої символічної нотації. Назва алгоритму може вказувати на його призначення (наприклад, алгоритм сортування, обігу матриць, гри в «хрестики - нулі» і т.п.) або визначати метод рішення, що використовується в ньому.

Формалізацією алгоритмів займаються вже тисячі років. Ще за 300 років до н.е. Евклід написав алгоритми розподілу кутів, перевірки рівності трикутників і рішення інших геометричних задач. Він почав з невеликого словника аксіом, таких як «паралельні лінії не перетинаються» і побудував на їхній основі алгоритми для вирішення складних задач. Формалізовані алгоритми такого типу добре підходять для задач математики, де повинна бути доведена вірність будь-якого положення або можливість виконання будь-яких дій, швидкість виконання таких алгоритмів не є важливою. Для виконання реальних задач, наприклад, задачі сортування на комп'ютері записів про мільйон покупців, швидкість виконання є важливою частиною постановки задачі.

Складність алгоритмів, аналіз швидкості виконання

Залежність часу роботи програми від обсягу даних, які обробляються, визначається оцінкою складності алгоритму. Є декілька способів оцінки складності алгоритмів. Програмісти звичайно приділяють увагу швидкості виконання алгоритму, але важливі й інші вимоги, наприклад, до розміру пам'яті, вільного місця на диску або інших ресурсів. Від швидкого алгоритму може бути мало сенсу, якщо під нього потрібно більше пам'яті, ніж має комп'ютер.

Якщо алгоритм призначений для обробки рівних за обсягом даних, тривалість його роботи кожного разу залишається незмінною і складність алгоритму є постійною.

Час роботи алгоритму обробки будь-яких масивів даних залежить від розмірів цих масивів. Час роботи алгоритму, що виконує тільки операції читання і занесення даних в оперативну пам'ять, визначається формулою $an + b$, де a – час, необхідний для читання

і занесення в пам'ять однієї інформаційної одиниці, n – розмір масиву, b – час, затрачуваний на виконання допоміжних функцій. Оскільки ця формула виражає лінійну залежність від n, складність відповідного алгоритму називають *лінійною*.

Розглянемо обмінне сортування (метод бульбашки) та визначимо кількість порівнянь для нього. В обмінному сортуванні n елементів списку спочатку визначається найменший елемент всього списку і здійснюється його обмін з першим елементом, потім визначається найменший елемент списку, що залишився, і проводиться його обмін з другим елементом і т.д. При виконанні цього алгоритму проводиться $(n-1)$ порівнянь при визначенні першого найменшого, $(n-2)$ порівнянь при визначенні другого і т.д. Загальний обсяг порівнянь для такого алгоритму обчислюється за формулою:

$$(n-1)+(n-2)+(n-3)+\ldots+4+3+2+1=\left(n^2-n\right)/2 \qquad (4.1)$$

Час, затрачуваний алгоритмом, як функція розміру задачі, називається *тимчасовою складністю* алгоритму. Поведінка цієї складності в межах при збільшенні розміру задачі називається *асимптотичною тимчасовою складністю*.

Ідея просторово-часової складності

Багато алгоритмів надають вибір між швидкістю виконання і ресурсами, що використовує програма. Задача може виконуватися швидше, використовуючи більше пам'яті, або навпаки, повільніше, зайнявши менший обсяг пам'яті.

Гарним прикладом у даному випадку може служити алгоритм знаходження найкоротшого шляху. Задавши карту вулиць міста у вигляді мережі, можна написати алгоритм, що обчислює найкоротшу відстань між будь-якими двома точками в цій мережі. Замість того щоб кожного разу наново перераховувати найкоротшу відстань між двома заданими точками, можна наперед прорахувати її для всіх пар точок і зберегти результати в таблиці. Тоді, щоб знайти найкоротшу відстань для двох заданих точок, буде достатньо просто узяти готове значення з таблиці.

При цьому ми одержимо результат практично миттєво, але це вимагатиме великого обсягу пам'яті. Карта вулиць для великого міста може містити сотні тисяч точок. Для такої мережі таблиця

найкоротших відстаней містила б більше 10 мільярдів записів. В цьому випадку вибір між часом виконання і обсягом необхідної пам'яті очевидний: поставивши додаткові 10 гігабайт оперативної пам'яті, можна примусити програму виконуватися набагато швидше.

Звідси витікає ідея просторово-часової складності алгоритмів, при цьому підході передбачається оцінювати складність алгоритму в термінах часу і простору та знаходити між ними компроміс.

Оцінка з точністю до порядку

При порівнянні різних алгоритмів важливо розуміти, як складність алгоритму співвідноситься зі складністю вирішуваної задачі. При розрахунках відповідно одному алгоритму сортування тисячі чисел може зайняти 1 секунду, а сортування мільйона — 10 секунд, тоді як розрахунки по іншому алгоритму можуть зажадати 2 і 5 секунди відповідно. В цьому випадку не можна однозначно сказати, яка з двох програм краще — це залежатиме від початкових даних.

Відмінність продуктивності алгоритмів на задачах різної обчислювальної складності часто більш важлива, ніж просто швидкість алгоритму. У вищенаведеному випадку перший алгоритм швидше сортує короткі списки, а другий — довгі.

Продуктивність алгоритму можна оцінити за порядком величини. Алгоритм має складність порядку O(f(N)) (вимовляється «O велике від f від N»), якщо час виконання алгоритму росте пропорційно функції f(N) із збільшенням розмірності початкових даних N. Наприклад, розглянемо фрагмент коду, що сортує позитивні числа:

```
For I = 1 To N
  'Пошук найбільшого елемента в списку.
  MaxValue = 0
  For J = 1 to N
       If Value(J)> MaxValue Then
             MaxValue = Value(J)
             MaxJ = J
       End If
  Next J
  'Вивід найбільшого елемента на друк.
  Print Format$(MaxJ)& ":" & Str$(MaxValue)
```

```
  'Обнуління елемента для виключення його з подальшого
  пошуку.
  Value(MaxJ)= 0
Next I
```

В цьому алгоритмі змінна циклу I послідовно приймає значення від 1 до N. Для кожного приросту I змінна J у свою чергу також приймає значення від 1 до N. Таким чином, в кожному зовнішньому циклі виконується ще N внутрішніх циклів. У результаті внутрішній цикл виконується N*N або N^2 разів і, отже, складність алгоритму порядку $O(N^2)$.

При оцінці порядку складності алгоритмів використовується найбільш швидко зростаюча частина рівняння. Припустимо, час виконання алгоритму пропорційний N^3+N. Тоді складність алгоритму буде дорівнювати $O(N^3)$. Відкидання частини рівняння, що поволі росте, дозволяє оцінити поведінку алгоритму при збільшенні розмірності даних задачі N.

При великих N внесок другої частини в рівняння N^3+N стає все менш помітним. При N=100 різниця $N^3+N=1000100$ і N^3 дорівнює всього 100, або менш ніж 0,01 відсотка. Але це вірно тільки для великих N. При N=2 різниця між $N^3+N=10$ і $N^3=8$ дорівнює 2, а це вже 20 відсотків.

Постійні множники в співвідношенні також ігноруються. Це дозволяє легко оцінити зміни в обчислювальній складності задачі. Алгоритм, час виконання якого пропорційний $3*N^2$, матиме порядок $O(N^2)$. Якщо збільшити N в 2 рази, то час виконання задачі зросте приблизно в 22, тобто в 4 рази.

Ігнорування постійних множників дозволяє також спростити підрахунок числа кроків алгоритму. В попередньому прикладі внутрішній цикл виконується N^2 раз, при цьому усередині циклу виконується декілька інструкцій. Можна просто підрахувати число інструкцій `If`, можна підрахувати також інструкції, виконувані у середині циклу або, крім того, ще і інструкції в зовнішньому циклі, наприклад, оператори `Print`.

Обчислювальна складність алгоритму при цьому буде пропорційна N^2, $3*N^2$ або $3*N^2+N$. Оцінка складності алгоритму по порядку величини дасть одне і те ж значення $O(N^3)$, і відпаде необхідність в точному підрахунку кількості операторів.

Пошук складних частин алгоритму

Звичайно найскладнішим є виконання циклів і викликів процедур. В попередньому прикладі весь алгоритм укладений в двох циклах.

Якщо процедура викликає іншу процедуру, необхідно враховувати складність процедури, що викликається. Якщо в ній виконується фіксоване число інструкцій, наприклад, здійснюється виведення на друк, то при оцінці порядку складності її можна не враховувати. З іншого боку, якщо в процедурі, що викликається, виконується O(N) кроків, вона може вносити значний внесок в складність алгоритму. Якщо виклик процедури здійснюється усередині циклу, цей внесок може бути ще більше.

Наведемо як приклад програму, що містить повільну процедуру `Slow` зі складністю порядку $O(N^3)$ і швидку процедуру `Fast` зі складністю порядку $O(N^2)$. Складність всієї програми залежатиме від співвідношення між цими двома процедурами.

Якщо процедура `Slow` викликається в кожному циклі процедури `Fast`, порядки складності процедур перемножуються. В цьому випадку складність алгоритму дорівнює добутку $O(N^2)$ і $O(N^3)$ або $O(N^3*N^2)=O(N^5)$. Наведемо фрагмент коду, який ілюструє цей випадок:

```
Sub Slow()
Dim I As Integer
Dim J As Integer
Dim K As Integer
  For I = 1 To N
      For J = 1 To N
            For K = 1 To N
                        ' Виконання будь-яких дій
            Next K
      Next J
  Next I
End Sub
Sub Fast()
Dim I As Integer
Dim J As Integer
Dim K As Integer
  For I = 1 To N
      For J = 1 To N
            Slow          ' Виклик процедури Slow
      Next J
  Next I
```

```
End Sub

Sub MainProgram()
  Fast
End Sub
```

З іншого боку, якщо процедури незалежно викликаються з основної програми, їхня обчислювальна складність підсумовується. В цьому випадку повна складність буде дорівнювати $O(N^3)+O(N^2)=O(N^3)$. Таку складність, наприклад, матиме такий фрагмент коду:

```
Sub Slow()
Dim I As Integer
Dim J As Integer
Dim K As Integer

  For I = 1 To N
      For J = 1 To N
            For K = 1 To N
                  ' Виконання будь-яких дій
            Next K
      Next J
  Next I
End Sub
Sub Fast()
Dim I As Integer
Dim J As Integer
  For I = 1 To N
      For J = 1 To N
            ' Виконання будь-яких дій
      Next J
  Next I
End Sub

Sub MainProgram()
  Slow
  Fast
End Sub
```

Складність рекурсивних алгоритмів

Рекурсивними процедурами (recursive procedure) називаються процедури, що викликають самі себе. В багатьох рекурсивних алгоритмах саме ступінь вкладеності рекурсії визначає складність алгоритму, при цьому не завжди легко оцінити порядок складності. Рекурсивна процедура може виглядати простою, але при цьому робити великий внесок в складність програми, багато разів викликаючи саму себе.

Наступний фрагмент коду містить підпрограму всього з двох операторів. Проте для заданого N підпрограма виконується N разів, таким чином, обчислювальна складність фрагмента порядку O(N).

```
Sub CountDown(N As Integer)
  If N <= 0 Then Exit Sub
  CountDown N - 1
End Sub
```

Багатократна рекурсія

Рекурсивний алгоритм, що викликає себе кілька разів, є прикладом *багатократної рекурсії* (multiple recursion). Процедури з множинною рекурсією складніше аналізувати, ніж просто рекурсивні алгоритми, і вони можуть давати більший внесок в загальну складність алгоритму.

Нижче приведена підпрограма схожа на попередню підпрограму `CountDown`, тільки вона викликає саму себе двічі:

```
Sub DoubleCountDown(N As Integer)
  If N <= 0 Then Exit Sub
  DoubleCountDown N - 1
  DoubleCountDown N - 1
End Sub
```

Можна б було припустити, що час виконання цієї процедури буде в два рази більше, ніж для підпрограми `CountDown`, і оцінити її складність порядку 2*O(N)=O(N). Насправді ситуація трохи складніше.

Якщо T(N) - число разів, яке виконується процедура `DoubleCountDown` з параметром N, то легко помітити, що T(0)=1.

Якщо викликати процедуру з параметром N рівним 0, то вона просто закінчить свою роботу після першого кроку.

Для великих значень N процедура викликає себе двічі з параметром, рівним N-1, виконуючись 1+2*T(N-1) разів. В табл. 4.1 наведені деякі значення функції T(0)=1 і T(N)=1+2*T(N-1). Якщо звернути увагу на ці значення, можна побачити, що T(N)=2(N+1)-1, що дає оцінку складності процедури порядку O(2N). Хоча процедури `CountDown` та `DoubleCountDown` і схожі, друга процедура вимагає виконання набагато більшого числа кроків.

Таблиця 4.1

Значення функції часу виконання для підпрограми `DoubleCountDown`

N	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
T(N)	1	3	7	15	31	63	127	255	511	1023	2047

Вимоги рекурсивних алгоритмів до обсягу пам'яті

Для деяких рекурсивних алгоритмів важливий обсяг доступної пам'яті. Можна легко написати рекурсивний алгоритм, який потребує невеликого обсягу пам'яті при кожному своєму виклику. Обсяг зайнятої пам'яті може збільшуватися в процесі послідовних рекурсивних викликів.

Тому для рекурсивних алгоритмів необхідно хоча б приблизно оцінювати вимоги до обсягу пам'яті, щоб переконатися, що програма не вичерпає при виконанні всю доступну пам'ять.

Наведена нижче підпрограма потребує пам'ять при кожному виклику. Після 100 або 200 рекурсивних викликів, процедура вичерпає всю вільну пам'ять, і програма аварійно зупиниться з помилкою «Memory».

```
Sub GobbleMemory(N As Integer)
Dim Array() As Integer
  ReDim Array (1 To 32000)
  GobbleMemory N + 1
End Sub
```

Навіть якщо усередині процедури пам'ять не запрошується, система виділяє пам'ять із *системного стеку* (system stack) для

збереження параметрів при кожному виклику процедури. Після повернення з процедури пам'ять зі стеку звільняється для подальшого використовування.

Якщо в підпрограмі зустрічається довга послідовність рекурсивних викликів, програма може вичерпати стек, навіть якщо виділена програмі пам'ять ще не вся використана. Якщо запустити на виконання наступну підпрограму, вона швидко вичерпає всю вільну стекову пам'ять, і програма аварійно припинить роботу з повідомленням про помилку «stack Space». Після цього ви зможете отримати значення змінної Count, щоб дізнатися, скільки разів підпрограма викликала себе перед тим, як вичерпати стек.

```
Sub UseStack()
Static Count As Integer
  Count = Count + 1
  UseStack
End Sub
```

Визначення локальних змінних у середині підпрограми також може займати пам'ять зі стеку. Якщо змінити підпрограму UseStack з попереднього прикладу так, щоб вона визначала три змінних при кожному виклику, програма вичерпає стековий простір ще швидше:

```
Sub UseStack()
Static Count As Integer
Dim I As Variant
Dim J As Variant
Dim K As Variant
  Count = Count + 1
  UseStack
End Sub
```

Якнайгірший та усереднений випадок

Оцінка з точністю до порядку дає верхню межу складності алгоритму. Те, що програма має певний порядок складності, не означає, що алгоритм дійсно виконуватиметься так довго. За певних

початкових даних, багато алгоритмів виконуються набагато швидше, ніж можна припустити на підставі їхнього порядку складності. Наприклад, наступний код реалізує простий алгоритм вибору елемента зі списку:

```
Function LocateItem(target As Integer) As Integer
  For I = 1 To N
      If Value(I)= target Then Exit For
  Next I
  LocateItem = I
End Sub
```

Якщо шуканий елемент знаходиться наприкінці списку, доведеться перебрати все N елементів для того, щоб його знайти. Це займе N кроків, тобто складність алгоритму порядку O(N). В цьому так званому *якнайгіршому випадку* (worst case) час виконання алгоритму буде найбільшим.

З іншого боку, якщо шукане число на початку списку, алгоритм завершить роботу практично відразу, вчинивши всього декілька ітерацій. Це так званий *якнайкращий випадок* (best case) зі складністю порядку O(1). Звичайно і якнайкращий, і якнайгірший випадки зустрічаються відносно рідко, і інтерес представляє *оцінка усередненої або очікуваної* (expected case) поведінки.

Якщо спочатку числа в списку розподілені випадково, шуканий елемент може опинитися в будь-якому місці списку. В середньому буде потрібно перевірити N/2 елементів для того, щоб його знайти. Отже, складність цього алгоритму в усередненому випадку порядку O(N/2), або O(N), якщо прибрати постійний множник.

Для деяких алгоритмів порядок складності для найгіршого і якнайкращого варіантів розрізняється. Наприклад, складність алгоритму швидкого сортування в якнайгіршому випадку порядку $O(N^2)$, але в середньому його складність порядку O(N*log(N)), що набагато швидше. Іноді алгоритми типу швидкого сортування бувають дуже довгими, щоб якнайгірший випадок досягався украй рідко.

Функції оцінки порядку складності

В табл. 4.2 наведені деякі функції, які зустрічаються при оцінці складності алгоритмів. Функції наведені в порядку зростання обчислювальної складності зверху вниз. Це означає, що алгоритми з порядком складності функцій, які розташовані у верхній частині таблиці, виконуватимуться швидше, ніж ті, складність яких визначається функціями з нижньої частини таблиці.

Таблиця 4.2

Функції оцінки порядку складності, що часто зустрічаються

Функція	**Примітка**
f(N)=C	C - постійна
f(N)=log(log(N))	
f(N)=log(N)	
f(N)=NC	C - постійна від нуля до одиниці
f(N)=N	
f(N)=N*log(N)	
f(N)=NC	C - постійна більша одиниці
f(N)=CN	C - постійна більша одиниці
f(N)=N!	тобто 1*2* … N

Складність алгоритму, визначувана рівнянням, яке є сумою функцій з таблиці, зводитиметься до складності тієї з функцій, яка розташована в таблиці нижче. Наприклад, $O(\log(N)+N^2)$ - це те ж саме, що і $O(N^2)$.

Звичайно алгоритми зі складністю порядку N*log(N) і менш складних функцій виконуються дуже швидко. Алгоритми порядку N^C при малих C, наприклад N^2, виконуються достатньо швидко. Обчислювальна ж складність алгоритмів, порядок яких визначається функціями C^N або N!, дуже велика, і ці алгоритми придатні тільки для вирішення задач з невеликим N.

Наприклад, в табл. 4.3 показано, як довго комп'ютер, що виконує мільйон інструкцій в секунду, виконуватиме деякі повільні алгоритми. З таблиці видно, що при складності порядку $O(C^N)$ можуть бути вирішені тільки невеликі задачі, і ще менше параметр N може бути для задач зі складністю порядку O(N!). Для вирішення задачі порядку O(N!) при N=24 потрібний час більший, ніж час існування всесвіту.

Логарифми

Перед тим, як продовжити далі, слід зупинитися на логарифмах, оскільки вони відіграють важливу роль в різних алгоритмах. Логарифм числа N по підставі B - це ступінь P, в який треба звести основу, щоб одержати N, тобто $B^P=N$. Наприклад, якщо $2^3=8$, то відповідно $\log_2(8)=3$.

Таблиця 4.3

Час виконання складних алгоритмів

	N=10	**N=20**	**N=30**	**N=40**	**N=50**
N^3	0,001 с	0,008 с	0,027с	0,064 с	0,125 с
2^N	0,001 с	1, 05 с	17,9 хвилин	1,27 дня	35,7 років
3^N	0,059с	58,1 хвилин	6,53 роки	$3,86*10^5$ років	$2,28*10^{10}$ років
N!	3,63 с	$7,71*10^4$ років	$8,41*10^{18}$ років	$2,59*10^{34}$ років	$9,64*10^{50}$ років

Можна привести логарифм до іншої основи за допомогою співвідношення $\log_B(N)=\log_C(N)/\log_C(B)$. Наприклад, щоб обчислити логарифм числа по підставі 10, знаючи його логарифм по підставі 2, можна скористатися формулою $\log_{10}(N)=\log_2(N)/\log_2(10)$. При цьому $\log_2(10)$ - це таблична константа, яка приблизно дорівнює 3,32. Оскільки постійні множники при оцінці складності алгоритму можна опустити, то $O(\log_2(N))$ - це ж саме, що і $O(\log_{10}(N))$ або $O(\log_B(N))$ для будь-кого B. Оскільки підстава логарифма не має значення, часто просто пишуть, що складність алгоритму порядку O(log(N)).

В програмуванні часто зустрічаються логарифми по підставі 2, що обумовлено вживаною в комп'ютерах двійковою системою числення. Тому ми для спрощення виразів скрізь писатимемо log(N), маючи на увазі під цим $\log_2(N)$. Якщо використовується інша підстава алгоритму, це буде позначено особливо.

Реальні умови та швидкість виконання

Хоча при дослідженні складності алгоритму звичайно корисно відкинути малі члени рівняння і постійні множники, іноді їх все-таки необхідно враховувати, особливо якщо розмірність даних задачі N мала, а постійні множники достатньо великі.

Припустимо, ми розглядаємо два алгоритми рішення однієї задачі. Один виконується за час порядку O(N), а інший - порядку $O(N^2)$. Для великих N перший алгоритм, ймовірно, працюватиме швидше.

Проте, якщо узяти конкретні функції оцінки часу виконання для кожного з двох алгоритмів, наприклад, для першого f(N)=30*N+7000, а для другого $f(N)=N^2$, то в цьому випадку при N менше 100 другий алгоритм виконуватиметься швидше. Тому якщо відомо, що розмірність даних задачі не перевищуватиме 100, можливо доцільніше застосувати другий алгоритм.

З іншого боку, час виконання різних інструкцій може сильно відрізнятися. Якщо перший алгоритм використовує швидкі операції з пам'яттю, а другий - повільне звернення до диска, то перший алгоритм буде швидше у всіх випадках.

Інші чинники можуть також ускладнити проблему вибору оптимального алгоритму. Наприклад, перший алгоритм може вимагати більшого обсягу пам'яті, ніж встановлено на комп'ютері. Реалізація другого алгоритму, у свою чергу, може зажадати набагато більше часу, якщо цей алгоритм набагато складніше, а його налагоджування може перетворитися на справжній кошмар. Іноді подібні практичні міркування можуть зробити теоретичний аналіз складності алгоритму майже безглуздим.

Проте аналіз складності алгоритму корисний для розуміння особливостей алгоритму і звичайно знаходить частини програми, що займають велику частину комп'ютерного часу. Надавши увагу оптимізації коду в цих частинах, можна внести максимальний ефект в збільшення продуктивності програми в цілому.

Іноді тестування алгоритмів є найбільш відповідним способом визначення якнайкращого алгоритму. При такому тестуванні важливо, щоб тестові дані були максимально наближені до реальних даних. Якщо тестові дані сильно відрізняються від реальних, результати тестування можуть сильно відрізнятися від реальних.

Звернення до файлу підкачки

Важливим чинником при роботі в реальних умовах є частота звернення до *файлу підкачки* (page file). Операційна система Windows відводить частину дискового простору під *віртуальну пам'ять* (virtual memory). Коли вичерпується оперативна пам'ять, Windows скидає частину її вмісту на диск. Пам'ять, що звільнилася, надається програмі. Цей процес називається підкачкою, оскільки сторінки, скинені на диск, можуть бути підвантажені системою назад в пам'ять при зверненні до них.

Оскільки операції з диском набагато повільніші за операції з пам'яттю, дуже часте звернення до файлу підкачки може значно знизити продуктивність програми. Якщо програма часто звертається до великих обсягів пам'яті, система буде часто використовувати файл підкачки, а це приведе до уповільнення роботи.

Як приклад подана програма `Pager`, програма запрошує все більше і більше пам'яті під створювані масиви до тих пір, поки не почне звертатися до файлу підкачки. На запит програми введіть кількість пам'яті в мегабайтах та натисніть кнопку **Page** (Підкачка). Якщо ввести невелике значення, наприклад 1 або 2 Мбайт, програма створить масив в оперативній пам'яті і виконуватиметься швидко.

Якщо ж ви введете значення, близьке до обсягу оперативної пам'яті вашого комп'ютера, то програма почне використовувати файл підкачки. Цілком вірогідно, що вона при цьому звертатиметься до диска постійно. Ви також помітите, що програма виконується набагато повільніше. Збільшення розміру масиву на 10 відсотків може привести до 100-відсоткового збільшення часу виконання.

Програма `Pager` може використовувати пам'ять одним з двох способів. Якщо ви натиснули кнопку **Page,** програма почне послідовно звертатися до елементів масиву. По мірі переходу від однієї частини масиву до іншої системі може бути потрібно підвантажувати їх з диска. Після того як частина масиву опинилася в пам'яті, програма продовжить роботу з нею.

Якщо ж ви натиснули кнопку **Thrash** (Пробуксовка), програма буде випадково звертатися до різних ділянок пам'яті. При цьому вірогідність того, що потрібна сторінка знаходиться у цей момент на диску, набагато зростає. Це надмірне звернення до файлу підкачки називається *пробуксовкою пам'яті* (thrashing). В табл. 4.4 наведено час виконання програми `Pager` на комп'ютері з процесором Pentium з тактовою частотою 90 Мгц і 24 Мбайт оперативної пам'яті. Залежно від конфігурації вашого комп'ютера,

швидкості роботи з диском, кількості встановленої оперативної пам'яті, а також наявності інших запущених паралельно програм час виконання програми може сильно різнитися.

Спочатку час виконання тесту росте майже пропорційно розміру зайнятої пам'яті. Коли починається звернення до файлу підкачки, швидкість роботи програми різко падає. Помітьте, що до цього тести із зверненням до файлу підкачки і пробуксовкою поводяться практично однаково, тобто коли весь масив знаходиться в оперативній пам'яті, послідовне і випадкове звернення до елементів масиву займає однаковий час. При підкачці елементів масиву з диска випадковий доступ до пам'яті набагато менш ефективний.

Для зменшення числа звернень до файлу підкачки є декілька способів. Основний прийом - економне витрачання пам'яті. При цьому треба пам'ятати, що програма звичайно не може зайняти всю фізичну пам'ять, тому що частину її займають система й інші програми. Комп'ютер, на якому були одержані результати, наведені в табл. 4.4, починав інтенсивно звертатися до диска, коли програма займала 20 Мбайт з 24 Мбайт фізичної пам'яті.

Іноді можна написати код так, що програма звертатиметься до блоків пам'яті послідовно. Наприклад, алгоритм сортування злиттям, описаний в 13 розділі, маніпулює великими блоками даних. Ці блоки сортуються, а потім зливаються разом. Впорядкована робота з пам'яттю зменшує число звернень до диска.

Таблиця 4.4

Час виконання програми `Pager` в секундах

Пам'ять (Мб)	**Підкачка**	**Пробуксовка**
2	3,79	3,73
4	7,58	7,47
6	11,37	11,2
8	15,49	14,94
10	18,9	18,62
12	22,85	22,36
14	26,47	26,15
16	30,26	29,88
18	34,51	37,46
20	132,37	12,155 (оцінка)

Псевдопокажчики, посилання на об’єкти та колекції

В деяких мовах, наприклад в C, C++ або Delphi, можна визначати змінні, які є *покажчиками* (pointers) на ділянки пам'яті. В цих ділянках можуть міститися масиви, рядки або інші структури даних. Часто покажчик посилається на структуру, яка містить інший покажчик, і так далі. Використовуючи структури, що містять покажчики, можна організовувати всілякі списки, графи, мережі та дерева.

До третьої версії Visual Basic не містив засобів для прямого створення посилань. Проте оскільки покажчик всього лише посилається на будь-яку ділянку даних, то можна, створивши масив, використовувати цілочисельний індекс масиву як покажчик на його елементи. Це так званий *псевдопокажчик* (fake pointer).

Посилання

У 4-й версії Visual Basic були вперше введені класи. Змінна, яка вказує на екземпляр класу, є посиланням на об'єкт. Наприклад, в наступному фрагменті коду змінна `obj` – це посилання на об'єкт класу `MyClass`. Ця змінна не вказує, на який об'єкт, поки вона не визначається за допомогою зарезервованого слова `New`. В другому рядку оператор `New` створює новий об'єкт і записує посилання на нього в змінну `obj`.

```
Dim obj As MyClass
  Set obj = New MyClass
```

Посилання в Visual Basic - це різновид покажчиків.

Об'єкти в Visual Basic використовують *лічильник посилань* (reference counter) для спрощення роботи з об'єктами. Коли створюється нове посилання на об'єкт, лічильник посилань збільшується на одиницю. Після того як посилання перестає вказувати на об'єкт, лічильник посилань відповідно зменшується. Коли лічильник посилань дорівнює нулю, об'єкт стає недоступним програмі. У цей момент Visual Basic знищує об'єкт і повертає зайняту ним пам'ять.

Колекції

Окрім об'єктів і посилань, починаючи з 4-й версії Visual Basic також з'явилися колекції. Колекцію можна представити як різновид масиву. Вони надають в розпорядження програміста зручні можливості, наприклад, можна міняти розмір колекції, а також здійснювати пошук об'єкта по ключу.

Питання продуктивності

Псевдопокажчики, посилання і колекції згадуються в цьому розділі, тому що вони в змозі сильно впливати на продуктивність програми. Посилання і колекції можуть спрощувати програмування певних операцій, але вони можуть зажадати додаткові витрати пам'яті.

Програма `Faker (диск з прикладами – папка ProgR4)` демонструє взаємозв'язок між псевдопокажчиками, посиланнями і колекціями. Коли ви вводите число і натискуєте кнопку **Create List** (Створити список), програма створює список елементів одним з трьох способів. Спочатку вона створює об'єкти, відповідні окремим елементам, і додає посилання на об'єкти до колекції. Потім вона використовує посилання усередині самих об'єктів для створення зв'язаного списку об'єктів. І, нарешті, вона створює зв'язний список за допомогою псевдопокажчиків. Поки не зупинятимемося на тому, як працюють зв'язні списки.

Після натиснення на кнопку **Search List** (Пошук в списку), програма `Faker` виконує пошук по всіх елементах списку, а після натиснення на кнопку **Destroy List** (Знищити список) знищує всі списки і звільняє пам'ять.

В табл. 4.5 наведені значення часу, який потрібен програмі для виконання цих задач на комп'ютері з процесором Pentium з тактовою частотою 90 Мгц. З таблиці видно, що за зручність роботи з колекціями доводиться платити ціною більшого часу, затрачуваного на створення і знищення колекцій.

Колекції також містять індекс списку. Частина часу, затрачуваного при створенні колекції, йде на створення індексу. При знищенні колекції посилання, що зберігаються в ній, звільняються. При цьому система перевіряє і обновляє лічильники посилань для

всіх об'єктів. Якщо вони дорівнюють нулю, то сам об'єкт також знищується. Все це займає додатковий час.

При використовуванні псевдопокажчиків створення і знищення списку відбувається так швидко, що цим часом можна практично нехтувати. Системі при цьому не треба піклуватися про посилання, лічильники посилань і звільнення об'єктів.

З іншого боку, пошук в колекції здійснюється набагато швидше, ніж в двох останніх випадках, оскільки колекція використовує швидке хешування (hashing) побудованого індексу, тоді як список посилань і список псевдопокажчиків використовують повільний послідовний пошук.

Таблиця 4.5

Час Створення/Пошуку/Знищення списків в секундах

Розмір списку	**1000**	**2000**	**3000**
Колекція посилань	0,77/0,33/0,60	1,53/0,60/2,25	2,42/1,05/0,22
Список зв'язаних посилань	0,11/1,49/0,06	0,33/6,32/0,17	0,60/14,66/0,22
Псевдопокажчики	0,00/1,48/0,00	0,00/5,77/0,00	0,00/13,07/0,00

Хоча вживання псевдопокажчиків звичайно забезпечує кращу продуктивність, воно менш зручне, ніж використовування посилань. Якщо в програмі потрібен лише невеликий список, посилання і колекції можуть працювати достатньо швидко. При роботі з великими списками можна одержати більш високу продуктивність, використовуючи псевдопокажчики.

```
' Приклад програми, яка виділяє багато пам'яті
' у випадку файлу підкачки або пробуксовки
' ***********************************************
Option Explicit

Dim Memory() As Long

Private Sub CmdPage_Click()
    RunTest False
    MemText.SetFocus
End Sub

Private Sub CmdThrash_Click()
    RunTest True
    MemText.SetFocus
```

```
End Sub

Private Sub MemText_GotFocus()
    MemText.SelStart = 0
    MemText.SelLength = Len(MemText.Text)
End Sub

Private Sub mnuFileExit_Click()
    End
End Sub

Private Sub mnuHelpAbout_Click()
Dim txt As String

    txt = "       This program allows you to allocate
  bigger and bigger arrays until paging occurs." &
  vbCrLf
    txt = txt & "        Page: Program allocates
  indicated amount memory and traverses it in an
  orderly fashion. This will cause paging if array is
  larger than amount real memory your computer has." &
  vbCrLf
    txt = txt & "        Thrash: Program allocates
  indicated amount memory and accesses it randomly.
  This will cause LOTS paging if array is larger than
  amount real memory your computer has." & vbCrLf
    txt = txt & "        Note that times shown will
  probably be less than actual time you wait for
  program. This is because time shown does not include
  time to allocate array."

    MsgBox txt, vbOKOnly, "Help About... Pager"
End Sub
' ************************************************
' Run test.
' ************************************************
Private Sub RunTest(thrash As Boolean)
Const LONGS_PER_MEG = 1048576 / 4

Static megs As Integer

Dim new_megs As Integer
Dim max_index As Long
Dim start_time As Double
Dim i As Long
Dim index As Long

    On Error Resume Next
```

```
    new_megs = CInt(MemText.Text)
    If new_megs < 1 Then
        Beep
        MsgBox "Please enter an integer greater than
  0."
        MemText.SetFocus
        Exit Sub
    End If
    On Error GoTo 0

    TimeText.Text = ""
    MousePointer = vbHourglass
    DoEvents

    ' Allocate memory
    max_index = new_megs * LONGS_PER_MEG
    If new_megs <> megs Then
        ReDim Memory(1 To max_index)
        megs = new_megs
    End If

    ' Traverse memory to cause paging
    start_time = Timer
    If thrash Then
        ' Jump all over to causing thrashing.
        For i = 1 To max_index
            index = Int(max_index * Rnd)+ 1
            Memory(index)= i
        Next i
    Else
        ' Walk through memory to page without
        ' thrashing. Notice random assignment
        ' index so this version does about as
        ' much work as thrashing version.
        For i = 1 To max_index
            index = Int(max_index * Rnd)+ 1
            Memory(i)= i
        Next i
    End If

    TimeText.Text = _
        Format$(Timer - start_time, "0.00")
    MousePointer = vbDefault
    Beep
End Sub
```

Резюме

Аналіз продуктивності допомагає порівнювати різні алгоритми, він дозволяє передбачати, як будуть вести себе алгоритми за різних обставин. Вказуючи тільки ті частини алгоритму, на які потрібно найбільший час при виконанні програми, аналіз допомагає визначити, доробка яких ділянок коду збільшить продуктивність.

У програмуванні існує безліч компромісів, які не припустимі в реальних умовах. Один алгоритм може працювати швидше, але тільки за рахунок використання величезного обсягу додаткової пам'яті. Інший алгоритм простіше і реалізовувати і підтримувати, але працювати він буде повільніше, ніж будь-який більш складний алгоритм.

Виконавши аналіз наявних алгоритмів, з'ясувавши, як вони поводять себе в різних умовах, які вимоги висувають до ресурсів, ви зможете вибрати оптимальний варіант для вирішення поставленого завдання.

Контрольні запитання та завдання

1. Дайте визначення алгоритму, назвіть головні властивості алгоритму.
2. Чим відрізняється алгоритм від програми?
3. Що таке асимптотично тимчасова складність алгоритму?
4. Як виконується оцінка алгоритму з точністю до порядку?
5. Що означає складність O(log2n)?
6. Наведіть деякі функції, які використовують при оцінці складності алгоритмів.
7. Як виконується пошук складних частин алгоритму?
8. Як оцінити вимоги до обсягу пам’яті для рекурсивних алгоритмів?
9. Які чинники можуть ускладнити проблему вибору оптимального алгоритму?
10. Що є основним чинником при виборі методу для вирішення задач за допомогою перебору великого числа можливих варіантів?
11. Чим визначається час роботи алгоритму, що виконує тільки операції читання і занесення даних в оперативну пам'ять?
12. Що таке швидкодія алгоритму в кращому разі?

Розділ V
СИСТЕМА ШВИДКОЇ РОЗРОБКИ ДОДАТКІВ — VISUAL BASIC

Основні групи операторів програмування

Visual Basic — операторна мова, тому програма (модуль) являє собою деяку послідовність операторів. Базовий набір операторів Visual Basic дуже великий. В даному розділі ми розглянемо основні групи операторів, які найчастіше використовуються:

- групу декларативних операторів, що служать для опису об'єктів, з якими працює програма (типів, змінних, констант, об'єктів, додатків);
- групу операторів, що забезпечують привласнення і зміну значень цих об'єктів;
- групу операторів, що управляють ходом обрахувань;
- групу операторів, які забезпечують роботу з каталогами і файлами.

Типи даних і змінні

При вивченні мови програмування виникає безліч запитань: як в ньому влаштована система типів даних, які є прості типи; як створюються складні, структурні типи; чи є можливість визначення власних типів і динамічних типів; чи можна в мові визначати класи — "справжні" типи, де визначається не лише область можливих значень і структура даних, але і операції над ними.

Коли ми говоримо, що *T* — це тип даних, то розуміємо, що визначення типу *T* задає:

- область можливих значень типу;
- структуру організації даних;
- операції, визначені над даними цього типу.

Історично склалося так, що при визначенні типу опускають операції, дозволені над ним, маючи на увазі їх неявно. Наприклад, визначаючи тип `Integer`, говорять, що він задає цілі числа в деякому діапазоні. Звичайно, пізніше обов'язково буде сказано, які операції дозволені над цілими. У зв'язку з розвитком об'єктно-орієнтованого програмування визначення типу стали давати "по-справжньому" і включати в нього усі три компоненти, такі типи називаються класами.

Типи даних прийнято розділяти на *прості* і *складні* залежно від того, як влаштовані їх дані. У простих (скалярних) типах можливі значення даних єдині і неділимі. Складні типи характеризуються способом структуризації даних, — одне значення складного типу складається з декількох значень даних, організуючих складний тип.

Є й інші критерії класифікації типів. Так, типи розділяються на *вбудовані* і *визначені програмістом (користувачем)*. Вбудовані типи первісно належать мові програмування і становлять його базис. В основі системи типів будь-якої мови програмування завжди лежить базисна система типів, вбудована в мову. На основі вбудованих типів програміст може будувати власні, ним визначені типи даних. Але способи (правила) створення таких типів також вбудовані в мову.

Типи даних розділяються на *статичні* і *динамічні*. Для даних статичного типу пам'ять відводиться у момент оголошення — необхідний розмір даних відомий при їхньому оголошенні. Для даних динамічного типу розмір даних у момент оголошення невідомий, і пам'ять їм виділяється динамічно в процесі виконання програми за запитом.

Потужність мови багато в чому визначається тим, чи дозволяється програмістові визначати динамічні і власні типи даних.

Серед мов програмування виділяються два крайні випадки — мови, що строго типізуються, і безтипові мови. У першому випадку кожна змінна має строго фіксований у момент оголошення тип, і він не може змінюватися в процесі виконання програми. Такі мови є надійнішими, оскільки забезпечують суворий контроль типів і дозволяють виявляти помилки ще на стадії компіляції програми. Класичним прикладом є мова Паскаль. Безтипові мови — це мови з одним єдиним типом, в таких мовах одна і та ж змінна по ходу програми може зберігати дані фактично різних типів. Тип Variant мови VB є прикладом такого узагальнювального типу. Безтипові мови забезпечують більш швидке виконання програм і надають програмістам велику гнучкість, але досягається це за рахунок надійності програм. Однією з перших безтипових мов був `Lisp`.

Прості типи даних

Як і будь-яка мова, VB містить усі звичні вбудовані прості типи даних : логічні, арифметичні і строкові (табл. 5.1).

Таблиця 5.1

Система простих типів мови VBA

Ім'я типу	**Можливі значення**	**Необхідна пам'ять**
`Boolean`	True, False	2 байти
`Byte`	0...255	1 байт
`Integer`	-32768...+32767	2 байти
`Long`	Приблизно: — 2000 000 000.. +2000 000 000	4 байти
`Decimal`	Приблизно 30 десяткових цифр. Можна вказати число цифр після десяткової точки.	12 байтів
`Single`	--3,4E38..-1,4E- 45 — для від'ємних значень 1E- 45..3,4E38 — для додатніх значень	4 байти
`Double`	--1,7E308..-4,9E- 324 — для від'ємних значень 4,9-324.. 1,7E308 — для додатніх значень	8 байтів
`Currency`	Десяткові числа з фіксованою позицією коми. можливі 15 цифр до коми і 4 потім.	8 байтів
`String`	Є 2 види рядків : Рядки фіксованої довжини мають до 216 символів Рядки змінної довжини мають до 231 символу	8 байтів
`Date`	Дати змінюються в діапазоні від 1 січня 100 р. до 31 грудня 9999 р.	8 байтів
`Object`	Посилання на об'єкт (покажчик)	4 байти
`Variant`	Універсальний тип, значенням якого можуть бути дані будь-якого з перерахованих вище типів, об'єкти, значення NULL і значення помилок ERROR	Залежить від контексту, але не менше 16 байтів

Змінні типу `Decimal` не можна оголошувати так, як змінні інших типів, — наприклад, оператором Dim. Цей тип є одним з варіантів типу `Variant`, і для його завдання використовується функція CDec. Тип Currency використовується при грошових розрахунках.

Особливо слід сказати про тип Variant. Такий універсальний тип дозволяє перетворити мову на безтипову — усі дані можуть мати один тип (Variant). Звичайно, це зручно (думати не потрібно), а іноді і корисно, але загрожує неприємними помилками, та і пам'ять витрачається неефективно. VB провокує надмірне використання цього типу, оскільки дозволяє, оголошуючи змінну, не вказувати її тип, і тоді за умовчанням він встановлюється як Variant. Правильно завжди оголошувати тип змінної. Контроль над типами допоможе уникнути помилок при виході значення за можливі межі.

Змінні типу `Variant` можуть набувати значень будь-якого типу залежно від контексту. Крім того, вони можуть набувати і деяких спеціальних значень:

- `Empty` — змінна не ініціалізувала;
- `Null` — дані помилкові;
- `Error` — значення містить код помилки, який може бути використаний для її обробки в програмі;
- `Nothing` — змінна типу `Object` ні на що не посилається: зв'язок між нею і конкретним об'єктом перервана або не встановлена.

Оголошення змінних і констант простих типів

У своєму розвитку VB пройшов складний шлях — від безтипової мови з примітивним способом оголошення змінних до структурованої мови з добре визначеними типами даних і структурами управління. Але "рудименти" продовжують жити, і тому в мові одночасно існують різні за ідеологією способи визначення даних. Зокрема, можна оголошувати або не оголошувати змінні, і тоді тип їм буде присвоєний або за умовчанням, або за першою буквою імені, або за спеціальним символом оголошення типу, яким може закінчуватися ім'я. Явно оголосити змінну можна як на початку блоку, так і в тому місці, де виникла необхідність використовувати нову змінну. Усіма цими рудиментами користуватися не слід. Якщо в

початок модуля вставити оператор `Option Explicit` (опція Явно), то явне визначення змінних в цьому модулі стає обов'язковим.

Слід дотримуватися такої стратегії:

При оголошенні змінної визначається її тип і зона видимості — область, де ім'я змінної видиме і, значить, можливий доступ до її значення. Важливо розуміти, що змінну можна оголошувати на двох рівнях — рівні процедури і рівні модуля. Для оголошення змінних використовуються оператори `Dim`, `Public`, `Private` і `Static`. Перший можна використовувати на обох рівнях, `Public` і `Private` — на рівні модуля, `Static` — тільки на рівні процедури.

Змінні, оголошені на рівні процедури, називаються локальними по відношенню до неї. Їхньою зоною видимості є тільки та процедура, в якій вони оголошені. Змінні рівня модуля є глобальними. Вони оголошуються в розділі `Declarations`, який є у кожного модуля. Зона видимості глобальних змінних може поширюватися:

- на усі процедури одного модуля, в якому вони оголошені. Такі глобальні змінні, що звуться закритими (`Private`), мають бути оголошені на рівні модуля, або оператором `Private`, або оператором `Dim`.

- на увесь програмний проект — усі процедури усіх модулів цього проекту. Такі глобальні змінні, називані відкритими (`Public`), мають бути оголошені оператором `Public`.

Локальні змінні рівня процедури можуть бути оголошені оператором `Static`, що робить їх статичними. Звичайні локальні змінні "народжуються" при вході в процедуру, видимі тільки в ній і "помирають", виходячи з процедури. Це означає, що пам'ять під змінні відводиться при вході в процедуру, а при виході вона вивільняється. Зона видимості статичних змінних, як і раніше, — процедура, але час життя інший, оскільки у них не відбирається пам'ять при виході — вона просто стає тимчасово недоступною. Тому при повторному вході в процедуру статичні змінні відновлюють значення, які були у них при останньому виході. Статичні змінні — це зберегічі інформації між багатократними викликами однієї і тієї ж процедури. Щоб статично змінні мали сенс, потрібна первинна ініціалізація змінних, — вони повинні мати хоч якісь значення вже при входженні в процедуру. От як VB ініціалізує змінні у момент їх оголошення:

- — для чисельних значень;
- порожній рядок ("") — для рядків змінної довжини;
- рядок, що містить нулі, — для рядків фіксованої довжини;

- Empty (значення, що вказує на відсутність ініціалізації) — для типу Variant;
- для масивів і записів (типу, визначеного програмістом), кожен елемент ініціалізувався відповідно до вказаних правил.

Синтаксис оголошення простих змінних

Оголошення простих змінних має такий синтаксис:

```
{Dim | Private } Public | Static } <ім'я змінної> [As <ім'я типу>]
.....[, <ім'я змінної> [as <Ім'я типу>]]
```

Спочатку йде ім'я оператора, а потім список оголошень змінних, де роль роздільника відіграє кома. Кожне оголошення зв'язує ім'я змінної з її типом, заданим конструкцією As. Будьте уважні, VBA неприємно відрізняється в цьому питанні від інших мов програмування. Тут, як звичайно, одним оператором можна оголосити довільне число змінних, але треба в кожному оголошенні вказувати конструкцію As, інакше змінним без As буде приписаний тип Variant.

Наведемо приклад, де діють модулі Father і Mother, у кожному з яких оголошені глобальні (загальні й закриті) змінні. У кожному з модулів оголошені дві процедури, що взаємно викликають одна одну. Друкування в процедурах дозволяє простежити за зміною значень глобальних, локальних і статичних змінних. Ось тексти модулів Father і Mother — оголошення глобальних змінних і процедур:

```
'Option Explicit
Public Fx As Byte, Fz As Integer
Private Fy As Integer

Public Sub Start()
  'Ініціалізація глобальних змінних
  Fx = 10: Fy = 11: Fz = 12
  Mx = 20: My = 21: Mz = 22
  Father1
End Sub

Public Sub Father1()
  Dim Fz As Byte 'Локальна змінна
```

```
  Fx = Fx + 2
  Fy = Mx — 2
  Fz = 1
  Debug.Print "Father1: Fx=", Fx, " Fy =", Fy, "Fz =",
  Fz
'Виклик процедури іншого модуля
  Mother1
End Sub
```

Тут ми наводимо тексти модуля Mother:

```
'Option Explicit
Public Mx As Byte
Private My As Integer

Public Sub Mother1()
  'Оголошення статичної змінної
  Static Count As Byte
  Count = Count + 1
  Mx = Mx — 2: Fz = My + 2
  Debug.Print "Mother: Статична змінна Count =", Count
  'Виклик процедури Father іншого модуля або заключної
  — Finish
  If Fx < Mx Then Father1 Else Finish
End Sub

Public Sub Finish()
  'Заключна печатка
  Debug.Print "Finish: Fx = ", Fx, "Fy =", Fy, "Fz =",
  Fz
  Debug.Print "Mx =", Mx, "My =", My, "Mz =", Mz
  'Оголошення різних типів і печатка значень, отриманих
  при оголошенні
  Dim B As Byte, I As Integer, L As Long
  Dim Sng As Single, D As Double, C As Currency
  Dim SF As String * 7, SV As String, Dat As Date
  Dim O As Object, V
  Debug.Print "B =", B, "I=", I, "L=", L
  Debug.Print "Sng =", Sng, "D =", D; "C=", C
  Debug.Print "SF =", SF, "SV =", SV, "Dat=", Dat
  If O Is Nothing Then Debug.Print "Об'єкт не
  визначений"
```

```
  If V = Empty Then Debug.Print "Variant змінні не
  инициализированы"
End Sub
```

Запустивши процедуру `Start` модуля `Father`, одержимо такі результати :

```
      Father1: Fx=     12      Fy =    18      Fz =    1
Mother: Статична змінна Count =  1
Father1: Fx= 14     Fy =   16     Fz =   1
Mother: Статична змінна Count =  2
Father1: Fx= 16     Fy =   14     Fz =   1
Mother: Статична змінна Count =  3
Finish: Fx =16      Fy =   Fz =   2
Mx =   14     My =   0      Mz =
B =    0      I=     0      L=     0
Sng =  0      D =    0       C=    0
SF =   SV =   Dat=   0:00:00
Об'єкт не визначений
Variant змінні не ініціалізовані
```

Коментарі:

- Процедури `Father` і `Mother` тричі взаємно викликають одна одну. Статична змінна Count підраховує число зроблених обігів.
- Процедура `Father` друкує значення глобальних змінних `Fx`, `Fy` і локальної змінної `Fz`. Помітьте, локальне оголошення "сильніше" глобального.
- Процедура `Finish` друкує заключні значення змінних `Fx`, `Fy`, `Fz`, `Mx`, `My` і `Mz`. Тут друкується значення глобальної змінної `Fz`, а значення двох змінних — `Fy` і `Mz` — не визначені. Справа в тому, що `Fy` — закрита змінна модуля `Father`, а `Mz` взагалі не оголошувалася. І все-таки помилки не виникає, тому що діють оголошення за замовчуванням, і обидві ці змінні вважаються оголошеними в модулі `Mother` і за замовчуванням мають тип `Variant`. Саме тут криється причина можливих помилок, щоб уникнути їх, ми й радили включити оператор `Option Excplicit`. Будь це зроблено, — з'явилися б попереджуючі повідомлення. У нас ці опції написані, але спеціально закоментовані для демонстрації ефекту їхньої відсутності. Якщо їх включити, то помилка відразу зустрінеться в процедурі `Start` - адже змінні `My` і `Mz` не визначені в модулі `Father`.

- Друга частина процедури `Finish` носить самостійний характер — вона демонструє оголошення всіх простих типів і друк значень, одержуваних змінними при оголошенні. Помітьте, як перевіряється невизначеність об'єкта й порожнеча значень змінних.

Оголошення за замовчунням

У VB є старі засоби Бейсика, що дозволяють не оголошувати змінну явно, але встановлювати її тип за першим або останнім символом імені змінної. Імена змінних в VB можуть закінчуватися спеціальним символом, що дозволяє встановити тип цієї змінної :

```
% — Integer;
& — Long;
! — Single;
# — Double;
@ — Currency;
$ — String.
```

Є ще можливість визначення типу за першою буквою імені. З цією метою в мову введена серія операторів `DefType` (по одному на кожен тип — `DefBool`, `DefInt` і так далі), що визначають свій діапазон букв для кожного типу. Якщо перша буква імені неоголошеної змінної входить в той або інший діапазон, їй приписується відповідний тип. Ці оператори встановлюються на рівні модуля і діють на усі його процедури.

Кінцевий символ встановлення типу сильніший, ніж `DefType`, а той у свою чергу сильніший за стандартне "умовчання" `Variant`.

Константи

Одними змінними не обійтися — потрібні і константи. Для кожного простого типу є відповідні йому константи. Синтаксис побудови констант задається так, щоб за її значенням однозначно можна було приписати їй тип. Тому на відміну від змінних константи

можуть не оголошуватися, не мати імені і з'являтися там, де потрібно, задані своїми значеннями.

В той же час можна оголошувати іменовані константи, задаючи у момент оголошення значення константи і, можливо, її тип. Взагалі оголошення константи багато в чому нагадує оголошення змінної. Проте у цей момент задається значення, яке вже не можна змінити. Розглянемо синтаксис оператора `Const`, використовуваного для оголошення іменованих констант :

```
[Public | Private] Const <ім'я константи> [As type] = <константне вираження>
```

Ось приклад оголошення класичної константи :

```
Public Const pi As Double = 3.141593
```

Як і змінні, іменовані константи можна оголошувати на рівні процедури або модуля. У першому випадку використовується тільки ключове слово `Const`, в другому — додатково можна задати специфікатор `Public` або `Private`, що дозволяють оголосити константу загальною для усіх модулів або закритою. За умовчанням глобальні константи мають статус `Private`.

Існує також велика кількість вбудованих констант. Більшість таких констант починаються з префікса `vb`, наприклад: `vbWhite`, `VbSunday`. Вони дозволяють задавати тип календаря, дня тижня, колір і т. п.

Масиви

Простий і найпоширеніший структурний тип *(масив)* — впорядкована сукупність даних одного типу. Порядок в елементах масивів задається індексами його елементів. У VB масиви можуть бути одновимірними і багатовимірними. Число вимірів доходить до 60.

Усе сказане про глобальні і локальні змінні стосується і масивів. Є тільки одне доповнення, пов'язане з необхідністю вказувати розмірність масиву і межі зміни індексів. Тому синтаксис оголошення масивів дещо розширений — після імені змінної в круглих дужках вказується список розмірності масиву :

```
{Dim | Private | Public | Static} <ім'я змінної> (<список розмірності>)
[As <ім'я типу>]
```

Синтаксично кожен вимір в списку відділяється комою і визначається заданням нижньої і верхньої меж виміру індексів. З історичних причин в Бейсику нижня межа була фіксована і дорівнювала 0. Потім розробники ввели спеціальну опцію `OptionBase`, що дозволяє встановлювати цю межу, яка дорівнює 1 або 0. Потім дозволили задавати нижню і верхню межі, причому і та і інша можуть бути виразами при одному обмеженні — це мають бути константні вирази, що не містять змінних. Ось "класичне" оголошення одновимірного масиву і робота з ним:

```
Public Sub myArray()
    Const LowBound As Integer = — 5, Hibound As Integer
  = 5
    Dim MyArr(LowBound To Hibound) As Byte
    Dim I As Integer
    Debug.Print "Елементи масиву MyArr :"
    For I = LowBound To Hibound
      MyArr(I) = I + 6
      Debug.Print MyArr(I)
    Next I
End Sub
```

При друкуванні елементів цього масиву будуть виведені числа від 1 до 11 по числу його елементів.

При оголошенні масиву завжди слід задавати нижню межу, виходячи з того, що гранична пара, яка задає розмірність, повинна задовольняти синтаксису :

```
<гранична пара> ::= <константне вираження> To
  <константне вираження>
```

Це краще, ніж вживати `Option Base.` Ось приклад, що підтверджує це висловлювання :

```
''Option Base 0
Option Base 1
Dim Vector(10)
Dim A1(5), A2(5) As Integer
Dim Matrix(10, 20) As Double
Public Sub War()
Dim i
  For i = 0 To 5
    A1(i) = i: A2(i) = i + 5
    Vector(i) = A1(i) : Vector(i + 5) = A2(i)
  Next i
End Sub
```

В результаті роботи цієї програми видається повідомлення "Out of range" — вихід за допустимі межі.

Динамічні масиви

Динамічні масиви VB — потужний засіб, відсутній в багатьох інших мовах програмування. Масив вважається динамічним, якщо при первинному оголошенні не вказується його розмірність, але в наступному вона може бути визначена і перевизначена оператором ReDim. Розмірність визначається динамічно в тій процедурі і у той момент, коли вона стає фактично відомою. Більше того, при перевизначені розмірності масиву можна зберегти раніше отримані елементи і розширити масив, додавши нові елементи. Для цього необхідно просто задати ключове слово Preserve при перевизначенні масиву.

Розглянемо приклад. На рівні модуля оголосимо глобальний динамічний масив Vector. Його розмірність у момент роботи з ним визначається в діалозі з користувачем. По ходу справи масив розширюється, зберігаючи старі значення.

```
Public Sub DynArray()
    'Визначається фактична розмірність масиву Vector
    Dim N As Byte, I As Byte
    N = InputBox("Уведіть число елементів вектора")
    ReDim Vector(1 To N)
    For I = 1 To N
        Vector(I) = 2 * I + 1
    Next I
'   Масив розширюється зі збереженням раніше обчислених
  елементів
    ReDim Preserve Vector(1 To 2 * N + 1)
    For I = N + 1 To 2 * N + 1
        Vector(I) = 2 * I
    Next I
    'А тепер друк
    Debug.Print "Елементи Vector:" & Chr(13)
    For I = 1 To 2 * N + 1
        Debug.Print Vector(I)
    Next I
End Sub
```

При друкуванні елементів цього масиву спочатку будуть надруковані непарні числа від 3 до 21, а потім парні від 22 до 42.

Динамічні масиви з успіхом можна застосовувати там, де необхідні динамічні структури даних, наприклад, списки, стеки, черги.

Записи і тип, визначений програмістом

Разом з масивами, що являють об'єднання елементів одного типу, існує ще один спосіб створення складного типу — запис (у мові Паскаль) або структура (у мові С). Запис являє об'єднання елементів, кожен з яких може мати свій тип. Елементи запису часто називають її полями.

Визначення кожного запису можна розглядати як визначення дерева, з коренем якого пов'язаний сам запис, а з вершинами - нащадками пов'язані поля цього запису. Оскільки кожне поле може бути елементом довільного типу, у тому числі записом, такі вершини мають нащадків. Завдяки вкладеності запис може задавати структуру скільки завгодно складно побудованого дерева, у кожної вершини якого може бути довільне число нащадків. Це статична структура, повністю визначувана у момент оголошення запису.

У VB відсутній тип даних, званий записом, проте існують засоби для роботи з такими типами, а також тип даних, визначувані програмістом.

При визначенні типу програміст дає йому ім'я, що дозволяє потім визначати змінні цього типу звичайним способом. Оскільки усі елементи сукупності іменовані, можливий прямий доступ по імені до кожного з елементів.

Синтаксис визначення типу в VBA досить прозорий:

```
[Private | Public] Type <ім'я типу>
      <ім'я елементу> [([<розмірність масиву>])] As
  <тип елементу>
      <ім'я елементу> [([<розмірність масиву>])] As
  <тип елементу>
...
End Type
```

Визначення типу дається на рівні модуля, і якщо він є закритим (Private), то поширюється на один модуль, а для загальних (Public) типів — на усі. Обмежимося стандартним прикладом, в якому визначаються 2 типи (записи), : Fam і Person, де одне з полів `Person` має тип `Fam`.

```
'Визначення записів — користувальницьких типів
Type Fam
  firstName As String
  lastName As String
End Type

Type Person
  Fio As Fam
  Birthdate As Date
End Type
```

Ці оголошення ми помістили в розділ оголошень модуля `Father`. От одна з його процедур, що використовує ці типи даних:

```
Public Sub UserType()
    Dim Петров As Person
    Dim Козлов As Person
    Петров.Fio.firstName = "Петро"
    Петров.Fio.lastName = "Петров"
    Петров.Birthdate = #1/23/1961#
    Козлів.Fio.firstName = Петров.Fio.firstName
    Козлів.Fio.lastName = "Козлов"
    Козлів.Birthdate = #7/21/1966#
    MsgBox (Петров.Fio.firstName & " " &
  Петров.Fio.lastName _
     & " народився " & Петров.Birthdate)
    MsgBox (Козлов.Fio.firstName & " " &
  Козлов.Fio.lastName _
    & " народився " & Козлов.Birthdate)
End Sub
```

Ось що буде надруковано в результаті її роботи:

```
Петро Петров народився 23.01.61
Петро Козлов народився 21.07.66
```

Дії над записами

Для записів допустимі тільки операції привласнення. Наприклад, нехай певний призначений тип користувача **T** і оголошені

змінні цього типу, наприклад **X і Y**. Допустимою є операція X=Y. Можна написати:

Петров = Козлов

Але не можна написати:

If (Петров = Козлов) Then Debug.Print "Записи ідентичні"

Відсутність дозволених операцій над записами не означає, що з ними не можна працювати. Головне, що визначено прямий доступ до полів запису, і цього досить — з полями можна працювати як зі змінними. Наш приклад `(Sub UserType)` демонструє роботу з полями записів.

Поняття “класу”

Клас є узагальненням поняття “тип даних” і задає властивості і поведінку об'єктів (екземплярів) класу. Кожен об'єкт належить деякому класу. Відношення між об'єктом і його класом таке ж, як між змінною та її типом. Клас — це об'єднання даних з процедурами та функціями, які їх оброблюють. Дані називаються також змінними класу, а процедури і функції — методами класу. Змінні визначають властивості об'єкта, а сукупність їхніх значень — його стан. Разом з властивостями і методами з класом зв'язується ще одне поняття — “подія”. Кожен клас має деякий набір подій, які можуть виникати при роботі з об'єктом классу: найчастіше — за певних дій користувача, іноді — як результат дії системи. При виникненні події, яка пов'язана з тим або іншим об'єктом, система посилає повідомлення об'єкту, яке може бути оброблене методом — обробником події, спеціально створеним при конструюванні об'єкта. Події забезпечують велику гнучкість при роботі з об'єктами. Методи класу виконуються однаково для усіх об'єктів класу, а на події кожен об'єкт реагує індивідуально, оскільки має власний обробник події.

VBA дозволяє розробникові створювати власні класи. Розглянемо приклад створення класу «Особистість».

Синтаксично класи у VBA оформляються у вигляді модуля класу. Тому починати створення класу в редакторі Visual Basic потрібно з вибору в меню Insert пункту Class Module. Цей модуль має таку ж структуру, як і стандартний. Модуль складається з двох розділів — оголошень і методів. У першому з них природним чином описуються властивості класу, а в другому — його методи. І тут діють специфікатори області дії Public і Private. Public - властивості і

Public-методи складають інтерфейс класу. Тільки до цих властивостей і методів можна звертатися при роботі з об'єктами класу, оголошеними в інших модулях, де клас є видимим.

```
Option Explicit ' Розділ оголошень класу

'Клас Особистість
'Властивості класу: ім'я, по батькові, прізвище, дата
  народження
'Закриємо від прямого доступу
'Отримати і змінити їх можна тільки через методи класу
Private Імя As String
Private Побатькові As String
Private Прізвище As String
Private Датанародження As Date
Public Sub InitPerson (ByVal FN As String, ByVal LN As
  String, _
ByVal DoB As Date)
    'Ініціалізація особистості
    Ім’я = FN
    Прізвище = LN
    Датанародження = DoB
End Sub

  Public Sub PrintPerson ()
    'Друк в налагоджувальному вікні Immediate
      Dim S As String
    If WhoIs Then S = "народилася" Else S = "народився"
      Debug.Print Ім’я, Побатькові, Прізвище, S,
  Датанародження
  End Sub

  Public Sub CopyPerson (You As Особистість)
    Ім’я = You.ВашеІм’я
    Прізвище = You.ВашаПрізвище
    Датанародження = You.ВашаДатаНародження
End Sub

  Public Function WhoIs () As Boolean
    'Намагається визначити стать особистості,
  аналізуючи ім'я та прізвище     'Повертає True, якщо
```

```
  думає, що має справу з жінкою.
      Dim F1 As Boolean, F2 As Boolean
    F1 = ОстанняБуква (Ім'я) = "А" Or ОстанняБуква
  (Ім'я) = "Я"
    F2 = ОстанняБуква (Прізвище) = "А" Or ОстанняБуква
  (Прізвище) = "Я"
    If F1 And F2 Then
        'Можна вважати, що наша Особистість — жінка
        WhoIs = True
    ElseIf Not F1 And Not F2 Then
        WhoIs = False
    Else 'Є сумніви
        If Побатькові = "" Then
            Побатькові = InputBox (Ім'я & "" & Прізвище
  _
            & "!" & "Назвіть по батькові, будь ласка.")
        End If
        WhoIs = ОстанняБуква (батькові) = "А"
      End If
End Function

  Public Sub SayWhoIs ()
'Виведення повідомлення про стать і вік особистості
      If WhoIs Then
        MsgBox ("Думаю," & Ім'я & _
", Ви з прекрасної половини людства!")
    ElseIf Year (Дата народження)> 1967 Then
        MsgBox ("Думаю," & Ім'я & ", Ви — молода
  людина!")
    Else
        MsgBox ("Думаю," & Прізвище & ",Ви —
  чоловік!")
      End If
  End Sub

  Private Function ОстанняБуква (ByVal W As String) As
  String
    'Внутрішня функція: повертає у верхньому регістрі
  'Останню букву слова W
    ОстанняБуква = UCase (Right (W, 1))
End Function

  Public Property Get ВашеІм'я () As String
```

```
    ВашеІмя = Ім'я
End Property
Public Property Let ВашеІм'я (ByVal vNewValue As
  String)
      Ім'я = vNewValue
End Property
Public Property Get Побатькові () As String
    По батькові = Побатькові
End Property
Public Property Let ВашеОтчество (ByVal vNewValue As
  String)
   Побатькові = vNewValue
End Property

  Public Property Get ВашаФамілія () As String
    ВашеПрізвище = Прізвище
End Property

  Public Property Let ВашеПрізвище (ByVal NewValue As
  String)
    Прізвище = NewValue
End Property

  Public Property Get ВашаДатаНародження () As Date
    ВашаДатаНародження = Датанародження
End Property

  Public Property Let ВашаДатаНародження (ByVal
  NewValue As Date)
    Датанародження = NewValue
End Property

  Private Sub Class_Initialize ()
    Ім'я = "Адам"
    Прізвище = "Людина"
    Дата народження = # 1/1/100 #
End Sub
```

У клас «Особистість» додано 5 загальних методів: `initPerson`, `PrintPerson`, `CopyPerson`, `WhoIs`, `SayWhoIs` і по парі методів `Get` і `Let` на кожну закриту

властивість. У класі є і закрита для зовнішнього використання функція ОстанняБуква, і обробник події Initialize.

Метод Init, в тому або іншому вигляді, має бути визначений в кожному класі. Це - перший метод об'єкта, що викликається. Перш ніж почати роботу з об'єктом, його треба ініціалізувати.

Метод Print також повинен бути майже в кожному класі — треба ж роздруковувати інформацію про об'єкт!

CopyPerson — ще один загальний, необхідний метод, що дозволяє реалізувати справжнє привласнення, коли копіюється не посилання, а значення полів класу, він дозволяє мати не 2 посилання на один об'єкт, а 2 ідентичних об'єкти.

Булева функція WhoIs — метод, специфічний для нашого завдання. Це спроба визначити стать за ім'ям та прізвищем. По ходу справи потрібно було ввести допоміжну функцію ОстанняБуква.

Метод SayWhoIs викликає WhoIs, додатково визначає вік (тільки для чоловіків) і виводить відповідне повідомлення у вікно Message.

Закрита для зовнішнього використання функція Остання буква повертає у верхньому регістрі останню букву слова. Це завдання вирішується двома викликами функції роботи з рядками Right і UCase. Перша повертає "хвіст" слова, друга — перетворить результат у верхній регістр.

Зупинимося детальніше на закритих властивостях і спеціальних методах Get і Let. Взагалі кажучи, властивості можна не закривати, а спеціальні методи не вводити. Але VBA дозволяє наслідувати традиції об'єктно-орієнтованого програмування, згідно з якими вважається правильним безпосередньо не змінювати властивості, оскільки іноді це може привести до некоректного стану об'єкта. Тому доступ до них закривається (вони повинні бути оголошені з атрибутом Private), та зате вводяться спеціальні методи Get (для набуття значення властивості) і Let (для зміни значення на нове).

З кожним із об'єктів створеного нами класу пов'язуються дві події: Initialize і Terminate. Перше виникає при первинному зверненні до об'єкта, друге — після закінчення роботи з ними.

Приклад роботи з об'єктами класу, його методами і властивостями:

```
Public Sub Знайомство ()
    Dim UserOne As New Особистість
```

```
    Dim UserTwo As New Особистість
    Dim UserThree As New Особистість
    Debug.Print UserOne.ВашеІмя
    UserOne.InitPerson FN: = "Петро", LN: = "Петров",
  DoB: = # 1/23/1968 #
    UserTwo.InitPerson FN: = "Ганна", LN: = "Козлова",
  DoB: = # 7/21/1968 #
    UserOne.PrintPerson
    UserTwo.PrintPerson
    UserOne.SayWhoIs
    UserTwo.SayWhoIs
    UserTwo.ВашеПрізвище = UserOne. ВашеПрізвище & "а"
    Debug.Print UserOne. ВашеПрізвище
    Debug.Print UserTwo. ВашеПрізвище
    UserThree.InitPerson FN: = "Ганна", LN: = "Керн",
  DoB: = # 5/17/1803 #
    UserThree.PrintPerson
    UserThree.SayWhoIs
End Sub
Ось які результати  друку будуть видані у вікно
  Immediate:
Адам
Петро Петров народився 23.01.68
Ганна Козлова народилася 21.07.68
Петров Петрова
Ганна Петрівна Керн народилася 17.05.1803
```

Програмні оголошення

Оголошення можна давати на двох рівнях — модуля і процедури. На рівні модуля розділ оголошень йде першим і автоматично відділяється рискою від розділу методів. На рівні процедури оголошення і оператори можуть бути перемішані, вимагається лише, щоб оголошення змінної передувало її використанню. Вважається хорошим тоном мати в процедурах два чітко виділених розділи й усі оголошення розміщувати на початку процедури так, щоб вони передували виконуваній частині процедури — розділу операторів. Програміст повинен підтримувати структуру розділу оголошень і розділяти оголошення констант, типів і змінних. Чітко виділимо основні частини розділу оголошень: розділ опцій, розділ констант, розділ типів, розділ змінних, розділ *Declare*.

Розділ опцій

Опції є покажчиками для транслятора, вони можуть задаватися тільки на рівні модуля і повинні починати розділ оголошень. Це — синтаксична вимога. Опції задаються ключовим словом `Option`, після якого йде ім'я опції і, можливо, параметри. Повний склад опцій такий:

Explicit. При вказівці цієї опції транслятор вимагає, щоб усі змінні модуля були явно описані.

Base. Дана опція має два значення — 0 та 1, які вказують нижню межу індексів масивів за умовчанням. Правильніше не користуватися цією опцією, а завжди самостійно вказувати нижню межу індексів.

Private. Цю опцію зазвичай додають в головний модуль проекту. При її завданні проект робиться закритим і не доступним для інших проектів в системі документів.

Compare. Опція вказує транслятору, як він повинен виконувати порівняння рядків в процедурах модуля. Параметр опції може приймати одне з 3 можливих значень : {`Binary| Text | Database`}. За умовчанням VBA приймає метод `Binary`, при якому рядки порівнюються по внутрішніх кодах відповідних символів. У Windows порядок сортування визначається кодовою сторінкою. Ось частина такого типового порядку : `A<Z<a<z<A<Я<a<я`. Метод порівняння `Text` заснований на порівнянні, не чутливому до регістра, так що при порівнянні заголовні і рядкові букви не розрізняються. Для цих же символів порядок при цьому порівнянні буде таким: `((Ф=ф)<(Я=я) <(A=a)<(Я=я)`.

Розділи констант, типів і змінних

Бажано, щоб ці розділи йшли в описаному порядку. Тоді вже оголошені константи з'являтимуться при описі меж масивів, типи змінних передуватимуть опису самих змінних.

Для констант і типів ключові слова `Const` і `Type` однозначно визначають описуваний об'єкт. Для змінних ключові слова можуть бути різними (`Dim, Public, Private`).

Розділ Declare

Цей розділ з'являється в тих випадках, коли модулі проекту використовують динамічно зв'язувані бібліотеки DLL. Якщо DLL має бібліотеку типів TypeLib і вона доступна проекту, то немає необхідності описувати компоненти бібліотеки, вони будуть знайдені автоматично. Але якщо TypeLib недоступна або не визначена, то в цій ситуації кожна з функцій і процедур бібліотеки, викликувана в модулі, повинна бути попередньо описана спеціальним оператором Declare. Ось його синтаксис:

```
[Public | Private] Declare {Sub | Function} ім'я Lib
  "ім'я_бібліотеки" _[Alias "псевдонім"]
  [([параметри])] [As що повертає_тип]
```

Тут вказується ім'я бібліотеки, ім'я процедури або функції, можливий псевдонім, параметри й значення, що повертається функцією. Докладно про DLL, операторів Declare із прикладами на цю тему буде розказано в одному з розділів цієї роботи.

Правила іменування

Є декілька простих правил, яким треба слідувати, щоб мати добрий стиль програмування, вони стосуються оформлення тексту програм. Більшість із них досить прості й по ходу справи ми про них неодноразово говорили. Ось основні:

- використовуйте коментарі;
- дотримуйтесь правил іменування;
- структуруйте текст;
- будуйте програми з модулів "оптимального розміру".

Поговоримо про одне технічне, але практично важливе питання — як правильно давати імена константам, змінним і іншим об'єктам у наших програмах. Ім'я відіграє важливу роль у розумінні програми, тому в повсякденній практиці при написанні імен змінних треба слідувати правилу, яке полягає в такому: ім'я повинне відбивати сенс і складатися з одного або декількох разом написаних слів, кожне з яких починається з великої букви.

Розробники від Microsoft рекомендують дотримуватися суворіших правил. Ім'я повинне відбивати не лише сенс, але і тип змінної і її зону дії. Тому ім'я повинне складатися з префікса і власного імені. Префікс також є складеним, дві його частини

відбивають зону дії і тип змінної. Наведемо можливі значення префікса.

Таблиця 5.2

Можливі значення префікса згідно з правилами Microsoft

Префікс, який задає область дії	
Перша частина префікса	**Зона дії**
g	Global — увесь проект
m	Module — Для Private змінних модуля
відсутній	Procedure — Для локальних змінних
Префікс, який задає тип змінної	
Друга частина префікса	**Тип змінної**
str	string
int	Integer
byt	Byte
lng	Long
sng	Single
dbl	Double
cur	Currency
var	variant
obj	Object
bln	Boolean

От кілька прикладів правильно побудованих імен: `gstrOneWord`, `mintNumberOne`, `strAnswer`, `curSalary`. Також як за значенням константи можна відновити її тип, по правильно побудованих іменах можна однозначно відновити їхнє оголошення. Зробимо це в нашому прикладі:

```
Public gstrOneWord As String
Private mintNumberOne As Integer
Dim strAnswer As String
Dim curSalary As Currency
```

Згідно з цими же рекомендаціями імена констант варто будувати тільки із заголовних букв. Якщо ім'я константи складається з декількох слів, то для їхнього об'єднання використовується знак підкреслення, наприклад: `MY_DIRECTORY_PATH`. Зверніть увагу, при побудові констант Office 2000 використовується префікс, що вказує, якому з додатків належить константа.

Таблиця 5.3

Відповідність додатка та префікса констант

Додаток	**Префікс констант**
Access	ac
Excel	xl
FrontPage	fp
Office	mso
OfficeBinder	bind
Outlook	ol
Power Point	pp
Word	wd
VBA	vb

Префікси варто використовувати й при іменуванні об'єктів, для форм звичайно використовується префікс `frm`, для командних кнопок — `cmd` і так далі. Варто розуміти, що якщо вже користуватися префіксами, то загальновживаними, не слід займатися самодіяльністю в цьому питанні.

Оператори

Оператори забезпечують привласнення і зміну значень об'єктів, з якими працює програма.

Оператори та рядки

При записі тексту програми для спрощення читання, налагодження та її модифікації зручніше кожен оператор розташовувати в окремому рядку. Проте в одному рядку можна

розміщувати і декілька операторів. Символом розподілу операторів в рядку служить *двокрапка*.

Іноді виникає ситуація, коли оператор не уміщається в рядку і його слід продовжити в декількох рядках. Для продовження (переносу) оператора на наступний рядок використовується пара символів *пропуск-підкреслення*.

Перед оператором в рядку може стояти мітка — послідовність символів, що починається з букви і закінчується двокрапкою. Мітки можна розміщувати і в окремих рядках перед тими операторами, які вони повинні позначати. Мітки потрібні для операторів переходу типу `GoTo`, але їх використання вважається "поганим тоном" в структурному програмуванні.

Оператор коментарю

Коментарі на виконання програми не впливають, але потрібні як "ознака хорошого стилю". Ви можете заощадити на коментарях і написати, а потім налагодити невеликий модуль без них. Але вже через тиждень ніхто, у тому числі і ви, не зможете зрозуміти його дію і модифікувати потрібним чином.

Будь-який рядок тексту програми може закінчуватися коментарем. Коментар в VB починається апострофом (') і включає будь-який текст, розташований в рядку правіше. Зазвичай в коментарях описують завдання, що вирішуються модулями, функції, що виконуються процедурами, сенс основних змінних, алгоритми роботи процедур.

Інше застосування коментарі знаходять при налагодженні програм. Якщо потрібно виключити з програми деякі оператори, то досить перед ними поставити апостроф.

У VB є ще один спосіб виділення коментарів за допомогою ключового слова `REM`. Такий коментар (на відміну від коментарю, що починається апострофом) повинен відділятися в рядку від попереднього оператора двокрапкою:

```
weight=weight+z  :  Rem Збільшення ваги
value=weight*price      'Нова вартість
```

Привласнення

Оператори привласнення — основний засіб зміни стану програми (значень змінних і властивостей об'єктів).

Оператор Let. За його допомогою відбувається "звичайне" привласнення змінній або властивості значення виразу. Синтаксис оператора:

```
[Let] змінна = вираз
```

Ключове слово Let, як правило, опускається, *змінна* є ім'ям змінної або властивості; вираз задає значення, що привласнюється змінній. Тип виразу повинен відповідати типу змінної. Порушення цієї умови, наприклад, при спробі присвоїти числовій змінній строкове значення, призводить до того, що при компіляції з'явиться повідомлення про помилку. Змінним типу `Variant` можна привласнювати значення різних типів. Строковій змінній можна привласнювати будь-яке значення типу `Variant`, окрім `Null`. Числовій змінній значення типу `Variant` можна присвоїти, тільки це значення може бути перетворене до числа.

Оператор `Let` можна також застосовувати для привласнення значення змінної типу запис іншій змінній такого ж типу.

Оператор LSet. Цей оператор використовують для привласнення рядків з одночасним вирівнюванням ліворуч, а також для копіювання запису визначеного користувачем типу в запис іншого типу. Його синтаксис:

```
LSet СтрЗмінна = СтрВираз
LSet Змінна 1 = Змінна 2
```

У даному випадку ключове слово `LSet` обов'язкове, `СтрЗмінна` — ім'я строкової змінної, `СтрВираз` — вираз строкового типу. У другому варіанті `Змінна1` — ім'я змінної деякого визначеного користувачем типу, в яку виконується копіювання, `Змінна2` — ім'я змінної, можливо, іншого призначеного для користувача типу, значення якої копіюється.

Результатом привласнення рядків завжди є рядок тієї ж довжини, що і у `СтрЗмінна`. Якщо при цьому `СтрВираз` коротше, додаються пропуски справа; довше — зайві символи справа видаляються.

В другому варіанті оператора двійкове представлення запису з ділянки пам'яті, відведеного `Змінний2`, копіюється в ділянку пам'яті, яка відведена `Змінний1`, — при цьому дані не перетворяться відповідно до типів елементів (полів) запису, і якщо типи відповідних

елементів обох записів не співпадають, результат операції важко передбачити (часто видається повідомлення про невідповідність типів).

Приклади:

```
Public Sub Assign2()
  Dim Str1 As String, Str2 As String
  Str1 = "0123456789"
  Str2="abcd"
Debug.Print Str1, Str2

LSet Str2=Str1 'Результат — "0123"
LSet Str1 = "Вліво" 'Результат "Вліво"
Debug.Print Str1 Str2
End Sub
```

Результати налагоджувального друку :

```
00123456789              abcd
Вліво                     0123
```

Оператор RSet. Привласнює значення строкової змінної з вирівнюванням праворуч:

```
RSet   СтрЗмінна = СтрВираз
```

Де `СтрЗмінна` — ім'я строкової змінної, `СтрВираз` — вираз строкового типу. На відміну від оператора `LSet`, оператор `RSet` не можна використовувати для копіювання змінних записів. Результатом привласнення рядків завжди є рядок тієї ж довжини, що і `СтрЗмінна`. Якщо при цьому `СтрВираз` коротше, додаються пропуски ліворуч, довше — зайві символи ліворуч віддаляються.

Оператор Set. Цей оператор застосовується при роботі з об'єктами, встановлює посилання на знову створений або існуючий об'єкт. Його синтаксис:

```
Set ОбЗмінна = {[New] ОбВираз | Nothing}
```

`ОбЗмінна` — ім'я змінної або властивості, `New` — необов'язкове ключове слово, використовуване для явного виклику операції створення нового екземпляра класу (об'єкта). Якщо `ОбЗмінна` містила посилання на об'єкт, при привласненні це посилання звільниться. `ОбВираз` може бути ім'ям об'єкта (класу), іншою змінною того ж типу, функцією або методом, що повертають об'єкт відповідного типу. Виконання оператора `Set` з правою частиною `Nothing` перериває зв'язок між `ОбЗмінною` та об'єктом, на який вона посилалася. Якщо при цьому на нього не залишилося інших посилань, ресурси системи і пам'ять, виділені під цей об'єкт,

звільняються. У загальному випадку, якщо ключове слово `New` не вказане, нова копія об'єкта не створюється, а `ОбЗмінна` як значення отримує посилання на існуючий об'єкт. При цьому може виявитися, що декілька змінних посилаються на один об'єкт і зміна цього об'єкта через одну з них впливає на усі інші.

Управляючі оператори та цикли

Умовний оператор If Then Else End If

Це загальноприйнятий в мовах програмування оператор управління обчисленнями. Він дозволяє вибирати і виконувати дії залежно від істинності деякої умови. Є два варіанти синтаксису : в один рядок і у формі блоку. У першому випадку оператор має вигляд:

```
If умова Then [Оператори1] [Else Оператори2]
```

У другому випадку оператор розташований на декількох рядках:

```
If умова Then
[Оператори]
        [ElseIf  умова n Then
[Оператори  n]
[Else
[ІнакшеОператори]]
End If
```

Тут умова обов'язкова в обох операторах. Вона може бути числовим або строковим виразом зі значеннями True або False (Null трактується як False). Як умова може використовуватися і вираз вигляду ТуреОf Им'яОб'єкта Is ТипОб'єкта, де Им'яОб'єкта — посилання на об'єкт, а ТипОб'єкта — довільний коректний тип об'єкта. Оператори1 і Оператори2 — це послідовності з одного або декількох розділених двокрапкою операторів. У крайньому разі одна з цих послідовностей має бути не порожньою. Якщо умова істинна (True), виконується послідовність Оператори1, помилкова (False) — Оператори2. Форма умовного оператора визначається за наявності в рядку, що починається з If умова за then тексту, відмінного від коментаря. Якщо такий текст є, вважається, що використана форма в один рядок, немає — оператор повинен мати форму блоку.

У цьому випадку підблоки виду:

```
[ElseIf умова-n Then
[оператори-n]
```

можуть бути відсутніми або повторюватися кілька разів; підблок:

```
[Else
  [Інакшеоператори]]
```

також необов'язковий, а закриваючий оператор `EndIf` необхідний. Принаймні одна з послідовностей `оператори`, `оператори-n` … або `Інакшеоператори` повинна бути непустою. Якщо `умова` істинна, виконуються `оператори`, ні — відшукується перша істинна `умова-n`, і виконуються `оператори-n`. Якщо всі ці умови помилкові, виконуються `Інакшеоператори`. Після виконання однієї (можливо, порожньої) послідовності операторів управління передається операторові, що випливає за `EndIf`. Приклади:

```
Public Sub MinMax1(ByVal X As Integer, ByVal Y As
  Integer, _
      Min As Integer, Max As Integer)
      'Оператор If в один рядок
      If X > Y Then Max = X: Min = Y Else Max = Y: Min
  = X

End Sub

Public Sub MinMax2(ByVal X As Integer, ByVal Y As
  Integer, _
      Min As Integer, Max As Integer)
      'Оператор If у вигляді блоку
      If X > Y Then
            Max = X
            Min = Y
      Else
            Max = Y
            Min = X
      End If
End Sub

Public Sub If1()
      Dim Max As Integer, Min As Integer

      Call MinMax1(2 + 3, 2 * 3, Min, Max)
      Debug.Print Max, Min
      Call MinMax2(2 + 3, 2 * 3, Min, Max)
```

```
      Debug.Print Max, Min
End Sub
```

Оператор вибору Select Case

Цей оператор реалізує розбір випадків і залежно від значення тестового виразу вибирає і виконує одну з послідовностей операторів. Синтаксис оператора:

```
Select Case ВиразТест
[Case Список виразів  n]
                      [Операотри  n]
         [Case Else
[ІнакшеОператор]]
End Select
```

ВиразТест має бути присутній обов'язково. Він може бути довільним виразом з числовим або строковим значенням. СписокВиразів n має бути присутній в рядку, що починається ключовим словом Case (Випадок). Вирази в цьому списку відділяються комами і можуть мати одну з форм :

— вираз;

— вираз-нижня межа To вираз-верхня межа;

— Is оператор-порівняння вираз.

Перша форма задає окремі значення, друга і третя дозволяють задавати відразу цілі діапазони (області значень). Послідовність операторів Оператори n необов'язкова. Вона буде виконана, якщо відповідний Список виразів n є першим списком, порівнянним з поточним значенням ВиразТесту (він явно містить це значення або воно потрапляє в один із заданих в списку діапазонів). після виконання операторів послідовності Оператори n перевірка на відповідність іншим спискам виразів не проводиться, і управління передається на оператор, що йде за EndSelect. Необов'язкова послідовність ІнакшеОператори виконується, якщо жоден із списків виразів непорівнянний зі значенням ВиразТесту.

Приклад:

```
Public Sub Case1()
  Dim Before AsInteger
Dim CurrentYear As Integer, Str As String
' Ініціалізація змінних :
CurrentYear=2000
Before=InputBox("Скільки років тому"?, "Коли", 10)
```

```
Select Case CurrentYear — Before
       Case 1954 To 1969, 1971 To 1975, 1982
                    Str="Роки навчання"
       Case 1972 To 1989
                    Str="Роки виховання"
       Case Else
                    Str="Інші роки"
End Select
Debug.Print Str
End Sub
```

У даному випадку якщо перед виконанням вибору `Before=20` значення тестового виразу буде 1980, і працюватиме другий варіант `("Роки виховання")`. При `Before=25` значення тестового виразу 1975 і працюватиме перший варіант `("Роки навчання")`.

Діапазони значень можна задати і для рядків. При цьому їхні значення вважаються впорядкованими лексографічно. Наприклад, можливий такий список виразів :

```
Case "everything", "nuts" To "soup".
```

Безліч рядків, що задається, включає рядок `"everything"` і усі рядки від `"nuts" до "soup"` (наприклад, "onion").

Цикл For... Next

Цей оператор дозволяє повторити групу операторів задане число разів. Його синтаксис:

```
For Лічильник_циклу = Початок To Кінець [Step крок]
        тіло циклу
Next [Лічильник_циклу]
```

Тут `Лічильник_циклу` — числова змінна. Напочатку виконання циклу вона приймає значення, що задається числовим виразом `Початок` (змінна `Лічильник_циклу` не може мати тип `Boolean` або бути елементом масиву). Числовий вираз `Кінець` задає завершальне значення лічильника циклу. Воно обчислюється до початку виконання тіла циклу і не міняється, навіть якщо змінні, що в нього входять, змінюють свої значення в тілі циклу. Числовий вираз `крок` необов'язковий. Його значення також обчислюється на початку циклу і додається до лічильника циклу кожний раз, коли

завершується виконання тіла циклу і обчислення досягає рядка Next *[Лічильник_циклу]*. Якщо крок циклу явно не вказаний, за умовчанням він дорівнює 1. Тіло циклу — це послідовність операторів, яка буде виконана задане число разів. При якому значенні змінної Лічильник_циклу відбувається завершення циклу, залежить від знаку параметра крок. Якщо *крок* додатний, цикл завершиться, коли уперше виконається умова: Лічильник_циклу > Кінець

Якщо крок циклу від'ємний, умова його завершення : Лічильник_циклу < Кінець.

Ця умова перевіряється перед початком виконання циклу, а потім — після кожного збільшення кроку до лічильника циклу в операторі Next. Якщо вона виконана, управління передається на оператор, що йде за Next, ні — виконуються оператори з тіла циклу. Завершити цикл For ... Next можна і за допомогою оператора Exit For. Такі оператори можуть бути розташовані в тих місцях тіла циклу, де вимагається з нього вийти, не чекаючи виконання умови завершення.

Приклад. У цьому прикладі три вкладені цикли For ... Next використані для обчислення твору двох цілочисельних матриць, що ініціалізовані випадковими числами. Потім результуюча матриця перевіряється на наявність нульових значень:

```
Public Sub For1()
  Dim A(1 To 5, 1 To 5) As Integer
  Dim B(1 To 5, 1 To 5) As Integer
  Dim C(1 To 5, 1 To 5) As Integer
  Dim I As Integer, J As Integer, K As Integer
Dim Res As String
'  Ініціалізація матриць A і B випадковими числами в
  інтервалі [- 10, +10]
For I=1 To 5
For J=1 To 5
'  Отримання випадкового числа Rnd і перетворення його
  в ціле
  A(I, J)=Int(21*Rnd) — 10
  Next J
Next I
For I=1 To 5
  For J=1 to 5
  B(I, J)=Int(21*Rnd) — 10
  Next J
Next I
'  Обчислення добутку матриць
```

```
For I=1 To 5
  For J=1 To 5
  C(I, J)=0
       For K=1 to 5
       C(i, j)=C(I, J)+A(I, K)*B(K, J)
       Next K
  Next J
Next I
Res="No"
C(2,2)=0
'  Перевірка на нульове значення
For I=1 To 5
  For J=1 To 5
  If C(I, J)=0 Then
       Debug.Print "Индексы:", I, J
       Res="Yes"
       exit For
  End If
  Next J
Next I
Debug.Print Res
End Sub
```

Слід звернути увагу на оператор виходу `Exit For`, що припиняє виконання тільки внутрішнього циклу, так що перевірка на нуль здійснюватиметься в кожному рядку матриці, незалежно від існування нулів в попередніх рядках.

Зробимо ще декілька зауважень з приводу оператора `For... Next`:

- після закінчення циклу лічильник циклу зберігає своє значення у момент виходу, і його можна використовувати, наприклад, для аналізу передчасного виходу з циклу;
- в операторові `Next` можна не вказувати ім'я змінної, яка задає лічильник, — це ім'я мається на увазі за умовчанням. Проте цю можливість не слід використовувати, рекомендується завжди явно вказувати ім'я лічильника в операторові `Next`;
- допускається міняти значення лічильника в тілі циклу, але робити цього не слід ніколи, оскільки існує ризик зациклення програми.

```
Public Sub For2()
  Dim A(1 To 5) As Integer
  Dim i As Integer
  For i=5 to 1 step – 1
            A(i)=i
```

```
            i=i+1
  Next i
End Sub
```

Зазвичай спроба змінити значення лічильника циклу в його тілі означає, що замість циклу For .. next слід було б застосовувати цикл іншого виду.

Цикл Do...Loop

Повторює блок операторів, поки задана умова є істинною або поки воно не стане істинною.

Синтаксис:

Є чотири варіанти синтаксису цього циклу. У двох перших варіантах умова перевіряється на початку циклу:

```
Do [{While | Until} умова]
тіло циклу
Loop
```

В інших двох варіантах умова перевіряється наприкінці циклу:

```
Do
тіло циклу
Loop [{While | Until} умова]
```

Тут `умова` є числовим або строковим вираженням зі значеннями `True` або `False`. Взагалі вони необов'язкові. Значення `Null` умови трактується як `False`. `Тіло циклу` — це послідовність операторів, що буде виконуватися, поки `умова` залишається істинною, якщо перед нею йде ключове слово `While` або поки вона залишається помилковою — у варіантах циклу із ключовим словом `Until`. Таким чином, цикли виду `While умова` еквівалентні циклам виду `Until Not умова`. Крім того, у `тіло циклу` може входити оператор `Exit Do`, виконання якого відразу припиняє цикл і передає управління операторові, що безпосередньо йде за `Loop`. У випадку декількох вкладених циклів `Do … Loop` оператор `Exit Do` завершує лише самий внутрішній цикл, у тілі якого він розташований.

Приклади. У нашому прикладі реалізовані три варіанти пошуку за зразком з перевіркою умови на початку циклу, наприкінці циклу й у середині циклу для варіанта пошуку за зразком з бар'єром:

```
Public Sub Loop1()
  Const Size = 5
```

```
Dim X() As Integer
Dim i As Integer
Dim Found As Boolean
Const pat = 7
'Ініціалізація випадковими числами в інтервалі [1 —
10]
ReDim X(1 To Size)
Randomize
For i = 1 To Size
    X(i) = Int(11 * Rnd)
Next i

'Пошук за зразком з перевіркою на початку циклу
i = 1: Found = False
Do While (i <= Size) And (Not Found)
    If X(i) = pat Then
          Found = True
    Else: i = i + 1
    End If
Loop
If Found Then
    Debug.Print "Знайдений зразок!"
Else: Debug.Print "Зразок не знайдений!"
End If

'Пошук за зразком з перевіркою наприкінці циклу
i = 1: Found = False
Do
    If X(i) = pat Then
          Found = True
    Else: i = i + 1
    End If
Loop Until Found Or (i = Size + 1)
If Found Then
    Debug.Print "Знайдений зразок!"
Else: Debug.Print "Зразок не знайдений!"
End If

'Пошук з бар'єром
ReDim Preserve X(1 To Size + 1)
```

```
  X(Size + 1) = pat
  i = 1
  Do
       If X(i) = pat Then Exit Do
       i = i + 1
  Loop
  If i = Size + 1 Then
       Debug.Print "Зразок не знайдений!"
  Else: Debug.Print "Зразок знайдений!"
  End If
End Sub
```

Цикл While...Wend

Повторює виконання послідовності операторів, поки задана умова не стане помилковою.

Синтаксис:

```
While умова
тіло циклу
Wend
```

Тут `умова` й `тіло циклу` такі ж, як і для циклу `Do…Loop...` Тільки для цього виду циклу не передбачений оператор виходу `Exit`. Фактично цикл `While…Wend`-окремий випадок циклу `Do…Loop`-залишений у мові для сумісності з попередніми версіями.

Цикл For Each...Next

Повторює задану послідовність операторів для кожного елемента масиву або набору.

Синтаксис:

```
For Each елемент In група
тіло циклу
Next [елемент]
```

Тут `елемент` — змінна, котра пробігає як значення елементи колекції або масиву. Для колекцій `елемент` може бути змінною типу `Variant`, змінною типу `Object` або змінною

(об'єктом) деякого класу. У випадку циклу по масиву елемент зобов'язаний бути змінної типу Variant. Група — це ім'я набору об'єктів (найчастіше це колекція об'єктів) або масиву, для елементів яких виконується цикл. Цикл не застосуємо для масивів, тип елементів яких визначений користувачем, тому що такі елементи не можуть бути значеннями змінної типу Variant. У таких масивах можна використати цикл виду For … Next... Тіло циклу — послідовність операторів, виконувана для кожного елемента набору або масиву, — може містити оператори Exit For, що дозволяють перервати виконання циклу й передавати управління операторові, що випливає за Next (звичайно такий вихід відбувається при виконанні деякої умови, що перевіряється в операторі If…Then…Else)... Указувати змінну елемент після ключового слова Next не обов'язково, але бажано.

Приклади. У прикладі створюється колекція, число елементів якої й самі елементи вибираються випадковим чином. Потім ця колекція копіюється в динамічний масив, розмірність якого збільшується в процесі копіювання. На останньому етапі масив роздруковується. Цикли типу For …Each прекрасно працюють у подібних ситуаціях:

```
Public Sub ForEach1()
  Dim X As New Collection
  Dim Y() As Integer
  Dim item As Variant
  Dim i As Integer, Size As Integer

  'Ініціалізація колекції
  Randomize
  Size = Int(21 * Rnd)
  For i = 1 To Size
      X.Add Int(11 * Rnd)
  Next i

  'Копіювання колекції в динамічний масив
  Size = 1
  For Each item In X
       ReDim Preserve Y(1 To Size)
       Y(Size) = item
       Size = Size + 1
  Next item

  'Друкування елементів динамічного масиву
  For Each item In Y
```

```
      Debug.Print item
  Next item
End Sub
```

Крім розглянутих керуючих операторів, VB містить оператори переходу по мітці `GoTo`, переходу по мітці з поверненням `GoSub...Return` і умовні оператори переходу по мітках `On...GoSub i On...GoTo`. Ми ніколи не користуємося цими операторами.

Визначення процедур

Чим більше ви створюєте програм і чим більшими вони стають, тим частіше доводиться стикатися з тим, що один і той же блок коду знову і знову з'являється в різних місцях вашої програми (або в декількох програмах). Для коду, що повторюється, треба використовувати процедури.

Процедура (функція) — це програмна одиниця VB, що включає оператори опису її локальних даних і виконувані оператори. Зазвичай в процедуру об'єднують регулярно виконувану послідовність дій, яка вирішує окрему задачу або підзадачу. Тобто процедура — це сегмент коду, який виконує те або інше завдання, а потім передає управління в ту частину коду, з якої він був викликаний. Це означає, що одну і ту ж процедуру можна викликати з різних місць вашої програми і при правильному використанні застосовувати в різних програмах.

Ви вже стикалися з використанням процедур, навіть не знаючи про це. Кожного разу, вводячи код, який виконується у відповідь на якусь подію елемента управління типу Button (або якогось іншого типа), ви створювали процедури, що називаються *обробниками подій*. Обробники подій автоматично викликаються програмою при винекненні тієї або іншої події. Ви можете створювати свої власні процедури і викликати їх у міру потреби. Створені вами процедури часто називаються *процедурами користувача*. І хоча в попередніх прикладах код повністю знаходився в процедурах обробки подій, в реальних програмах, велика кількість коду може знаходитися в окремих процедурах користувача.

Види процедур

Процедури можна класифікувати за кількома ознаками: за способом використання (виклику) в програмі, за способом запуску процедури на виконання, за місцем знаходження коду процедури у проекті.

Процедури VB поділяються на підпрограми і функції. Перші описуються ключовим словом *Sub,* другі — *Function.*

За способом запуску процедур на виконання можна виділити в окрему групу процедури, що запускаються автоматично при виникненні тієї чи іншої події, — ми називаємо їх процедурами обробки подій.

За місцем знаходження коду процедури у проекті розрізняються процедури, що знаходяться у спеціальних програмних одиницях — стандартних модулях, модулях класів і модулях, пов'язаних з об'єктами, що реагують на події.

Ще один спеціальний тип процедур — процедури-властивості `Property Let`, `Property Set` і `Property Get`. Вони служать для завдання й одержання значень закритих властивостей класу.

Головне призначення процедур у всіх мовах програмування полягає в тому, що при їхньому виклику змінюється стан програмного проекту — змінюються значення змінних (властивості об'єктів), описаних в модулях проекту.

Існує два способи, за допомогою яких процедура отримує і передає інформацію, змінюючи тим самим стан системи документів.

Перший і основний спосіб полягає у використанні параметрів процедури. При виклику процедури її аргументи, що відповідають вхідним параметрам, повинні отримати значення, які программа виробляє або отримує від зовнішнього середовища. В результаті роботи процедури формуються значення вихідних параметрів, що передаються програмі за посиланням.

Другий спосіб полягає у використанні процедурою глобальних змінних і об'єктів як для одержання, так і для передачі інформації.

Робота з процедурами

Основна ідея, що стоїть за процедурою будь-якого виду, полягає в розподілі програми на послідовність завдань меншого розміру. Кожне з цих завдань можна потім оформити у вигляді процедури (функції). Такий підхід має декілька переваг.

0 Кожне завдання можна тестувати окремо. Чим менше в процедурі коду, тим легше проводити налагодження і легко займатися проектом спільно з іншими розробниками.

1 Можна позбавитися від зайвого коду, викликаючи при кожному виконанні завдання відповідну процедуру, а не повторюючи в програмі один і той же код.

2 Можна створити бібліотеку процедур, щоб використовувати їх в різних програмах і тім самим економити час при розробці нових проектів.

3 Стає більш легко підтримувати програми, оскільки, по-перше, якщо код не повторюється, то редагувати блок коду потрібно один раз. Крім того, відділення один від одного ключових компонентів (наприклад, призначеного для користувача інтерфейсу і засобів роботи з базами даних) дозволяє робити значні зміни в якійсь частині програми, не переписуючи її всієї.

Синтаксис процедур і функцій

Опис процедури Sub в VBA має такий вигляд:

```
[Private | Public] [Static] Sub ім'я([список-
  аргументів])
 тіло-процедури
End Sub
```

Ключове слово `Public` в заголовку процедури використовується для того, щоб оголосити процедуру загальнодоступною, тобто дозволити викликати її зі всіх інших процедур всіх модулів будь-якого проекту. Якщо модуль, в якому описана процедура, містить закриваючий оператор `Option Private`, то процедура буде доступна лише модулям свого проекту. Альтернативний ключ `Private` використовується, щоб закрити процедуру від всіх модулів, крім того, в якому вона описана. За умовчанням процедура вважається загальнодоступною.

Ключове слово `Static` означає, що значення локальних (оголошених в тілі процедури) змінних зберігатимуться в проміжках між викликами процедури (використовувані процедурою глобальні змінні, описані поза її тілом, при цьому не зберігаються).

Параметр `ім'я` — це ім'я процедури, що задовольняє стандартним умовам VB на імена змінних.

Необов'язковий параметр `список-аргументів` — це послідовність розділених комами змінних, задаючих передавані процедури при виклику параметрів.

Аргументи, або, по-іншому, формальні параметри, що задаються при описі процедури, завжди представляють лише імена (ідентифікатори). В той же час при виклику процедури її аргументи — фактичні параметри — можуть бути не лише іменами, але і виразами.

Послідовність операторів `тіло – процедури` задає програму виконання процедури. Тіло процедури може включати як «пасивні» оператори оголошення локальних даних процедури (змінних, масивів, об'єктів і ін.), так і «активні», які змінюють стани аргументів, локальних і зовнішніх (глобальних) змінних і об'єктів. У тіло можуть входити також оператори `Exit Sub`, що призводять до негайного завершення процедури і передачі управління в зухвалу програму. Кожна процедура в VB визначається окремо від інших, тобто тіло однієї процедури не може включати опису інших процедур і функцій.

Розглянемо детальніше структуру одного аргументу зі списку аргументів.

```
[Optional] [ByVal | ByRef] [ParamArray] змінна[ ( )]
  [As тип] [= значення-за-умовчанням']
```

Відзначимо одну цікаву особливість, яку не слід використовувати, але яку слід враховувати, — VB допускає, щоб фактичне значення аргументу, передаваного за посиланням, було константою або вираженням відповідного типу. В цьому випадку даний аргумент розглядається як передаваний за значенням, і не видається жодних повідомлень про помилку. Ключове слово `Optional` означає, що заданий ним аргумент є можливим — його необов'язково задавати у момент виклику процедури. Для таких аргументів можна задати значення за умовчанням. Необов'язкові аргументи завжди поміщаються в кінці списку аргументів.

Альтернативні ключі `ByVal i ByRef` визначають спосіб передачі аргументу в процедуру.

`ByVal` означає, що аргумент передається за значенням, тобто при виклику процедури створюватиметься локальна копія змінної з

початковим передаваним значенням і зміни цій локальній змінній під час виконання процедури не відіб'ється на значенні змінної, що передала своє значення в процедуру при виклику. Передача за значенням можлива лише для вхідних параметрів, які передають інформацію в процедуру, але не є результатами. Для таких параметрів передача за значенням частенько зручніше, ніж передача по засланню, оскільки у момент виклику аргумент може бути заданий скільки завгодно складним вираженням. Відмітимо, що вхідні параметри, що є об'єктами, масивами або змінними призначеного для користувача типу, передаються по засланню, що дозволяє уникнути створення копій. Вирази над такими аргументами все одно недопустимі, тому передача за значенням втрачає сенс.

`ByRef` означає, що аргумент передається по засланню, тобто всі зміни значення передаваній змінній при виконанні процедури безпосередньо відбуватимуться із змінною — аргументом з тієї програми, що викликала дану процедуру. У VB за умовчанням аргументи передаються по засланню (`ByRef`). Це не зовсім зручно для програмістів, які звикли до інших мов (наприклад, Паскаль або С), де за умовчанням аргументи передаються за значенням. Тому при описі процедури рекомендують явно вказувати спосіб передачі кожного аргументу, навіть якщо цей аргумент зустрічається в лівій частині.

Процедура VB допускає необов'язкові аргументи, які можна опустити у момент виклику. Узагальненням такого підходу є можливість мати змінне, заздалегідь не фіксоване число аргументів. Досягається це за рахунок того, що один з параметрів (останній в списку) може задавати масив аргументів — в цьому випадку він задається з описувачем `ParamArray`. Якщо список аргументів включає масив аргументів `ParamArray`, то ключ `Optional` використовувати в списку не можна. Ключове слово `ParamArray` може з'являтися перед останнім аргументом в списку з метою вказати, що за аргумент — масив з довільним числом елементів типу `Variant`. Перед ним не можна використовувати ключі `ByVal`, `ByRef` або `Optional`.

`Змінна` — це ім'я змінної, що представляє аргумент. Якщо після імені змінної задані круглі дужки, то це означає, що відповідний параметр є масивом.

Параметр `тип` задає тип значення, передаваного в процедуру. Він може бути одним з базисних типів VBA (не допускаються лише рядки `String` з фіксованою довжиною). Обов'язкові аргументи можуть також мати тип визначеного користувачем запису або класу. Якщо тип аргументу не вказаний, то

за умовчанням йому приписується тип Variant. Тип може бути одним з типів Office 2000. Для необов'язкових (Optional) аргументів можна явно задати значення за умовчанням. Це константа або константне вираження, значення якого передається в процедуру, якщо при її виклику відповідний аргумент не заданий. Для аргументів типу об'єкт (Object) як значення за умовчанням можна задати лише Nothing.

Синтаксис визначення процедур-функцій схожий на визначення звичайних процедур:

```
[public | Private] [Static] Function ім'я [(список-
  аргументів)] [As тип-значення]
тіло-функції
End Function
```

Відмінність лише в тому, що замість ключового слова Sub для оголошення функції використовується ключове слово Function, а після списку аргументів слід вказати тип значення, що повертається функцією. У тілі функції має бути використаний оператор привласнення виду: ім'я = вираження

Тут, в лівій частині оператора, стоїть ім'я функції, а в правій — значення виразу, який задає результат обчислення функції. Якщо при виході з функції значення змінної ім'я явно не привласнене, то функція повертає значення відповідного типу, визначене за умовчанням. Для числових типів це 0, для рядків — рядок нульової довжини (""), для типу Variant функція поверне значення Empty, а для заслань на об'єкти — Nothing.

Щоб негайно завершити обчислення функції і вийти з неї, в тілі функції можна використовувати оператор: **Exit Function**

Основна відмінність процедур від функцій полягає в способі їх використання в програмі. Наступна функція cube повертає аргумент, піднесений в куб:

```
Function cube(ByVal N As Integer) As Long
cube= N*N*N
End Function
```

Виклик цієї функції може мати вигляд

```
Dim x As Integer, в As Integer
в = 2
x =cube(y+3)
```

Вже говорилося, що будь-яку функцію можна перетворити в еквівалентну їй процедуру, при цьому з'являється додатковий параметр, необхідний для завдання результату. Отже, в еквівалентної процедури cube1 два аргументи:

```
Sub cube1 (ByVal N As Integer, ByRef З As Long)
C= N*N*N      ' здобуття результату в змінній, заданій
  по засланню
End Sub
```

Її можна використовувати для такого ж піднесення в куб: `cubel y+3, x.`

Але це не означає, що не має значення, який вид процедур слід використовувати в програмі. Якби вираз, в якому бере участь функція, був складніше, наприклад:

```
 x = cube(y)+sin(cube(x))
```

то його обчислення за допомогою процедури `cube1` потребувало б виконання декількох операторів і введення додаткових змінних:

```
cube1 y,z: cube1 x,u :   x=z+sin(u)
```

Функції з побічним ефектом

У класичному варіанті всі аргументи функції є вхідними параметрами, і єдиний результат обчислення функції — це значення, яке вона повертає. Прикладом є функція cube1. Але найчастіше використовуються функції з побічним ефектом, тобто такі функції, які, окрім набуття значення функції, змінюють значення деяких результуючих параметрів, передаваних функції по засланню. Наприклад:

```
Public Function SideEffect(ByVal X As Integer, ByRef Y
  As Integer) As Integer
SIdeEffect = X + Y
Y=Y+1
 End Function
Public Sub TestSideEffectO
Dim X As Integer, Y As Integer, Z As Integer
X = 3: Y = 5
Z = X + B + SideEffect(X, Y)
Debug.Print X, Y, Z
X = 3: Y = 5
    Z = SideEffect(X, Y)+ X + Y
Debug.Print X, Y, Z
End Sub
```

Результати обрахувань:

3 6 16

3 6 17

Як бачите, поводитися з функціями, що володіють побічним ефектом, слід обережно. У прикладі результат обчислення суми трьох доданків залежить від порядку їх запису. Це заперечує основним принципам математики. Більш того, слід розуміти, що результат обчислення непередбачуваний, оскільки VBA може для збільшення ефективності змінювати порядок дій при обрахуванні арифметичних виразів. Тому краще не використовувати у вираженні виклик функції з побічним ефектом, що змінює значення тих змінних, що входять в нього.

Виклики процедур і функцій

Виклик звичайної процедури Sub з іншої процедури можна оформити по-різному. Перший спосіб:

```
Ім'я список фактичних параметрів
```

де Ім'я — ім'я процедури, що викликається, а список фактичних параметрів — список аргументів, передаваних процедурі, він повинен відповідати списку аргументів, заданому в описі процедури. Задати цей список можна різними способами. У простому випадку значення передаваних процедурі аргументів перераховуються через кому в тому ж порядку, що і в списку аргументів із заголовка процедури.

Може виявитися, що в одному проекті декілька модулів містять процедури з однаковими іменами. Для відмінності таких процедур потрібно при їх виклику вказувати ім'я процедури через крапку після імені модуля, в якому вона визначена. Наприклад, якщо кожен з двох модулів `Mod1` і `Mod2` містить визначення процедури `ReadData`, а в процедурі `Myproc` потрібно скористатися процедурою `Mod2`, то цей виклик буде мати вигляд:

```
Sub Myproc()
Mod2.ReadDate
 End Sub
```

Якщо потрібно використовувати процедури з однаковими іменами з різних проектів, додайте до імен модуля і процедури ім'я

проекту. Наприклад, якщо модуль Mod2 входить в проект MyBook, той же виклик можна уточнити так:

```
MyBook.Hod2.ReadData
```

Другий спосіб виклику процедур пов'язаний з використанням оператора Call. В цьому випадку виклик процедури виглядає так:

```
Call ім'я(список-фактичних-параметрів)
```

Зверніть увагу на те, що в цьому випадку *список фактичних параметрів* поміщено в круглі дужки, а в першому випадку — ні. Спроба викликати процедуру без оператора Call, але із завданням круглих дужок, є джерелом синтаксичних помилок, особливо для розробників з великим досвідом програмування на Паскаль або С, де списки параметрів завжди в дужках. Слід звернути увагу на одну важливу і, мабуть, неприємну особливість виклику процедур VBA. Якщо процедура VBA має лише один параметр, то вона може бути викликана без оператора Call, з використанням круглих дужок і не повідомлення про помилку виклику. Це було б не так страшно, якби повертався правильний результат. На жаль, це не так. Проілюструємо сказане прикладом:

```
Public Sub MyInc(ByRef X As Integer)
X = X + 1
End Sub
Public Sub TestInc()
Dim X As Integer X = 1
'Виклик процедури з параметром в дужках
'синтаксично допустимо, але працює некоректно'
Mylnc (X)
Debug.Print X
'Коректний виклик
Mylnc X
Debug.Print X
'Це теж коректний виклик
Call MyInc(X)
Debug.Print X
End Sub
```

Результати роботи:

```
1
2
3
```

Хоча перший раз процедура викликається нормально і збільшує значення результату, але після закінчення її роботи значення

аргумента не змінюється. У цій ситуації не діє описувач `ByRef`, виклик йде так, ніби параметр описаний з описувачем `ByVal`.

Якщо ж процедура має більш ніж один параметр, то спроба викликати її, взявши параметри в круглі дужки і без ключьового слова `Call`, призводить до синтаксичної помилки. Ось простий приклад:

```
Public Sub SUMXY(ByVal X As Integer, ByVal Y As
  Integer, ByRef Z As Integer)
Z = X + Y
End Sub
Public Sub TestSumXYO
Dim a As Integer, b As Integer, з As Integer
a = 3: b = 5
'SUMXY (a, b, з)   'Синтаксична помилка
SUMXY a, b, з
Debug.Print з
End Sub
```

В даному прикладі некоректний виклик процедури `SUMXY` буде виявлений на етапі перевірки синтаксису.

Розглянемо ще одну особливість виклику VB-процедур, пов'язану з аргументами, передаваними по засланню. Як правило, в мовах програмування для таких аргументів можливе значення фактичного параметра обмежується, — воно має бути ім'ям змінної, заслання на яку передається процедурі. VB допускає можливість завдання для таких аргументів констант і виразів у момент виклику. Всі ці допуски роблять мову менш надійною і призводять до серйозних помилок.

Виклики функцій. Оформлення виклику функції залежить від того, чи потрібно використовувати її значення в процедурі. Якщо ви хочете передати обчислюване функцією значення в змінну або застосувати його у вираженні правої частини оператора привласнення, то виклик призначеної для користувача функції має той же вигляд, що і виклик вбудованої функції, наприклад, `sin(x)`. При виклику вказується ім'я функції, а після нього йде список фактичних параметрів в круглих дужках. Наприклад, якщо заголовок функції `Myfunc`:

```
Func Myfunc(Name As String, Age As Integer, Newdate As
  Date) As Integer
```

то використовувати її значення можна за допомогою викликів:

```
val= Myfunc("Alex",25, "10/04/97")
```

або

```
x = sqrt(Myfunc("Alex",25, "10/04/97"))+ x
```

Якщо ж значення, що обчислюється функцією, вас не цікавить і потрібно скористатися лише її побічними ефектами, то виклик функції може мати ту ж форму, що і виклик процедури Sub. Наприклад:

```
Myfunc "Alex",I, "10/04/97"
або:
Call Myfunc(Myson, 25, DateOfArrival)
```

Використання іменованих аргументів

У попередніх прикладах фактичні параметри виклику процедури або функції розташовувалися в тому ж порядку, що і формальні параметри в її заголовку. Це не завжди зручно, особливо якщо деякі аргументи необов'язкові (Optional). VB дозволяє вказувати значення аргументів в довільному порядку, використовуючи їхні імена. При цьому після імені аргумента ставляться двокрапка, а також знак рівності, після якого поміщається значення аргумента (фактичний параметр). Наприклад, виклик розглянутої вище процедури — функції Myfunc може виглядати так:

```
Myfunc Age:= 25, Name:= "Alex", Newdate:= DateOfArrival;
```

Спосіб особливо зручний при виклику процедур з необов'язковими аргументами, які завжди поміщаються в кінець списку аргументів в заголовку процедури. Хай, наприклад, заголовок процедури ProcEx:

```
Sub ProcEx(Name As String, Optional Age As Integer,
  Optional City = "Москва")
```

Список аргументів даної процедури включає один обов'язковий аргумент Name і два необов'язкових: Age і City, причому для останнього задано значення за умовчанням «Москва». Якщо при виклику цієї процедури другий аргумент не потрібний, то при виклику, що не використовує іменованих параметрів, сам параметр опускається, але кома, що виділяє його, повинна залишитися:

```
ProcEx "Оля",,"Тверь"
```

Замість цього можна використовувати виклик з іменами аргументів:

```
ProcEx City:="TВерь", Name:="0ля"
```

Тут не потрібно замінювати пропущений аргумент комами і дотримувати певний порядок дотримання аргументів. Якщо деякий необов'язковий аргумент не заданий при виклику, замість нього підставляється значення, визначене користувачем за умовчанням. Якщо і таке не визначене, то підставляється значення, визначене за умовчанням для відповідного типу. Наприклад, при виклику:

```
ProcEx Name:="Оля"
```

як значення аргумента `Age` в процедуру передасться 0, а як аргумент `City` — явно задане за умовчанням значення «Москва».

Як процедура «дізнається», чи переданий їй при виклику необов'язковий аргумент? Для цього можна скористатися функцією `IsMissing`. Вона по імені аргумента повертає логічне значення `True`, коли значення аргумента не передане в процедуру, і `False` — якщо аргумент заданий. Але це все працює лише в тому випадку, якщо параметр має тип `Variant`. Для всіх інших типів даних вважається, що в процедуру завжди передано значення параметра, явно або неявно задане за умовчанням. Тому якщо така перевірка необхідна, то параметр повинен мати тип `Variant`. Відзначимо також, що для масиву аргументів `ParamArray` функція `IsMissing` завжди повертає `False`, і для встановлення його порожнечі потрібно перевіряти, що верхній кордон індексу менше нижнього. Розглянемо функцію від двох аргументів, другий з яких необов'язковий:

```
Function TwoArgs (I As Integer, Optional X As Variant)
  As Variant
If IsMissing(X) Then
' якщо 2-ий аргумент відсутній, то повернути 1-й.
TwoArgs = I
Else
' якщо 2-ий аргумент є, то повернути їх добуток
TwoArgs = I*X
End If
End Function
```

Результати декількох викликів цієї функції у вікні налагодження:

```
? TwoArgs(5,7)
35
? TwoArgs(5.5)
6
? TwoArgs(5, 5.5)
27,5
```

```
? TwoArgs(5, "6")
30
```

Аргументи масиви

Аргументи процедури можуть бути масивами. Процедурі передається ім'я масиву, а його розмірність визначається вбудованими функціями `LBound` і `UBound`. Наведемо приклад процедури, що обчислює скалярний добуток векторів:

```
public Function ScalarProduct(X() As Integer, Y() As
  Integer) As Integer •Обчислює скалярний добуток двох
  векторів. 'Передбачається, що кордони масивів
  збігаються.
Dim i As Integer, Sum As Integer Sum = 0
For i = LBound(X) To UBound(X) Sum = Sum + X(i)* Y(1)
Next i ScalarProduct = Sum
End Function
```

Обидва параметри процедури, передавані по засланню, є масивами. Робота з ними в тілі процедури не викликає утруднень завдяки тому, що функції `LBound` і `UBound` дозволяють встановити кордони масиву по будь-якому виміру. Наведемо програму, в якій викликається функція `ScalarProduct`:

```
Public Sub TestScalarProductQ Dim A(1 To 5) As Integer
  Dim B(1 To 5) As Integer Dim З As Variant Dim Res As
  Integer Dim i As Integer
З = Array(1, 2, 3, 4, 5) For i = 1 Te 5 A(i)= C(i — 1)
Next i
З = Array(5, 4, 3, 2, 1)
For i = 1 Te 5
B(i)= C(i — 1) Next i Res = ScalarProduct(A, B)
  Debug.Print Res
End Sub
```

Інколи, коли в процедуру слід передати лише один масив, можна використовувати конструкцію `РагатАrray`. Наступна процедура `PosNeg` прочитує суми надходжень `Positive` і витрат `Negative`, вказаний у масиві `Sums`:

```
Sub PosNeg(Poslt1ve As Integer, Negative As Integer,
  ParamaArray Sums() As Variant)
Dim I As Integer
Positive = 0: Negative = 0
For I = 0 To UBound(Sums())' цикл по всіх елементах
  масиву
If Sums(I)> 0 Then
Positive = Positive + Sums(I)
Else
Negative = Negative — Sums(I)
End If
  Next I
    End Sub
```

Виклик процедури PosNeg може мати вигляд:

```
Public Sub TestPosNeg()
Dim Incomes As Integer, Expences As Integer
PosNeg Incomes, Expences -20, 100, 25, -44, -23, -60,
  120
 Debug.Print Incomes, Expences
End Sub
```

В результаті змінна Incomes набуде значення 245, а змінна Expences — 147. Відмітьте, перевагою використання масиву аргументів ParamArray є можливість безпосереднього перерахування елементів масиву у момент виклику.

Проте таке використання масиву аргументів ParamArray не вичерпує всіх його можливостей. У складніших ситуаціях передавані аргументи можуть мати різні типи. Наведемо приклад.

Завдання про медіану

Для масиву M і елемента Cand обчислити різницю між числом елементів масиву M, більших і менших, ніж Cand.

Це варіація завдання про медіану — «середній» елемент масиву. Медіану можна визначити, наприклад, таким алгоритмом: упорядкувавши масив, узяти елемент, що знаходиться в середині. Є і ефективніші алгоритми. Але обмежимося простішим завданням — перевіркою на «медіанність». Відмітимо: якщо всі елементи масиву M різні і число їх непарне, то для медіани шукана в завданні різниця

дорівнює 0. У загальному випадку значення різниці є мірою наближення параметра `Cand` до медіани масиву `M`.

Але займемося аспектами програмування цього завдання. У функції, що реалізовує це завдання, на вході — масив, а на виході — скаляр. Хотілося б, щоб ця функція могла викликатися у формулах робочої сторінки, а як фактичний параметр їй могли б бути передані як offeV1 """•-'' дана IsMediana:

```
Public Function IsMediana(M As Variant, Cand As
  Variant) As Integer
•даний масив M і елемент Cand. Як результат
  повертається
•різниця між числом елементів масиву M, більших і
  менших ніж Cand. Dim i As Integer, j As Integer niffi
  Pos As Integer, Neg As Integer pos = 0: Neg = 0
•Аналіз типа параметра M
If TypeName(M)= "Range" Then For 1 = 1 To M.Rows.Count
For j = 1 To M.Columns.Count If M.Cellsd, j) > Cand
  Then
Pos = Pos + 1 Elself M.Cellsd, j) < Cand Then
Neg = Neg + 1 End If Next j Next i IsMediana = Pos —
  Neg Elself TypeName(M)= "VARIANTO" Then 'TypeName is
  "VARIANTO"
'Це масив, але не зовсім справжній, для нього не
  визначені, 'наприклад, функції кордонів: LBound,
  UBound. Dim Val As Variant For Each Val In M If Val >
  Cand Then
Pos = Pos + 1 Elself Val < Cand Then
Neg = Neg + 1 End If Next Val IsMediana = Pos — Neg
  Elself TypeName(M)= "INTEGERO" Then
'Це справжній масив цілих VBA, для якого 'визначені
  функції кордонів. For i = LBound(M) To UBound(M) If
  M(i)> Cand Then
Pos = Pos + 1 Elself M(i)< Cand Then
Neg = Neg + 1 End If Next i IsMediana = Pos — Neg Else
MsgBox ("При виклику функции:IsMediana(M,Cand)"
& "- M не є масивом або об'єктом Range!") End If
End Function
```

Рекурсивні процедури

Рекурсія відбувається, якщо функція або підпрограма викликає сама себе.

Пряма рекурсія (direct recursion) виглядає приблизно так:

```
Function Factorial(num As Long) As Long
  Factorial = num * Factorial(num — 1)
End Function
```

В разі непрямої рекурсії (indirect recursion) рекурсивна процедура викликає іншу процедуру, яка, у свою чергу, викликає першу:

```
Private Sub Ping(num As Integer)
  Pong(num — 1)
End Sub

Private Sub Pong(num As Integer)
  Ping(num / 2)
End Sub
```

Стандартний приклад рекурсивної процедури – функція — факторіал Fact(N)= N!. Її визначення в VB:

```
Function Fact(n As Integer) As Long
If n <= 1 Then      ' базис індукції.
  Fact = 1          ' 0! =1
  Else              ' рекурсивний виклик в разі N > 0.
    Fact = Fact(n — 1)* n
End If
End Function
```

Спочатку ця функція перевіряє, що число менше або дорівнює 0. Факторіал для чисел менше нуля не визначений, але ця умова перевіряється для підстраховування. Якби функція перевіряла лише умову дорівнюваності числа нулю, то для від'ємних чисел рекурсія була б нескінченною.

Якщо вхідне значення менше або дорівнює 0, функція повертає значення 1. В інших випадках значення функції дорівнює добутку вхідного значення на факторіал від вхідного значення, зменшеного на одиницю.

Те, що ця рекурсивна функція врешті-решт зупиниться, гарантується двома фактами. По-перше, при кожному подальшому виклику значення параметра num зменшується на одиницю. По-друге, значення num обмежене знизу нулем. Коли num дорівнюватиме 0, функція зупиняє рекурсію. Умова, наприклад, в даному випадку умова num<=0, називається умовою зупинки рекурсії.

При кожному виклику підпрограми система зберігає ряд параметрів в системному стеку. Оскільки цей стек відіграє важливу роль, інколи його називають просто стеком. Якщо рекурсивна функція викличе себе надто багато разів, вона може вичерпати стековий простір і аварійно завершити роботу з помилкою «Out of stack space».

Число разів, яке функція може викликати сама себе до того, як використає весь стековий простір, залежить від об'єму встановленої на комп'ютері пам'яті і кількості даних, що поміщаються програмою в стек. В одному з тестів програма вичерпала стековий простір після 452 рекурсивних викликів. Після зміни рекурсивної функції таким чином, щоб вона визначала 10 локальних змінних при кожному виклику, програма могла викликати себе лише 271 раз.

Оскільки кожен виклик процедури вимагає накладних витрат, ефективніше для факторіалу ітеративна програма:

```
Function Fact1(n As Integer) As Long
Dim Fact As Long, i As Integer
Fact = 1      ' 0! =1.
If n > 1 Then    ' цикл замість рекурсії.
    For i = 1 To n
        Fact = Fact * i
    Next i
End If
Fact1 = Fact
End Function
```

Наведемо процедуру, що оцінює час виконання рекурсивного і нерекурсивного варіантів:

```
Private Sub cmStart_Click()
' Порівняння за часом рекурсивної і не рекурсивної
  реалізації факторіалу.
Dim i As Long, Res As Long
Dim n As Integer
Dim Start As Single, Finish As Single
'Рекурсивне обчислення факторіалу
n = txInput.Text
Start = Timer
For i = 1 To 100000
 Res = Fact(n)
```

```
Next i
Finish = Timer
lbOutput.Caption = lbOutput.Caption & vbCrLf & _
"Час рекурсивних обрахувань:" & (Finish — Start)
'Не рекурсивне обчислення факторіалу
Start = Timer
For i = 1 To 100000
Res = Fact1(n)
Next i
Finish = Timer
lbOutput.Caption = lbOutput.Caption & vbCrLf & _
"Час нерекурсивних обрахувань:" & (Finish — Start)
End Sub
```

Результати обчислень рис.5.1 наведені для двох запусків тестів процедури.

Як бачите, в даному випадку нерекурсивний варіант працює в три рази швидше. Окрім проблем з часом виконань, рекурсивні процедури можуть легко вичерпати і стекову пам'ять, в якій розміщуються аргументи кожного рекурсивного виклику. Тому уникайте неконтрольованого розмножения рекурсивних викликів і замінюйте рекурсивні алгоритми на ітеративні там, де використання рекурсії по суті не потрібне.

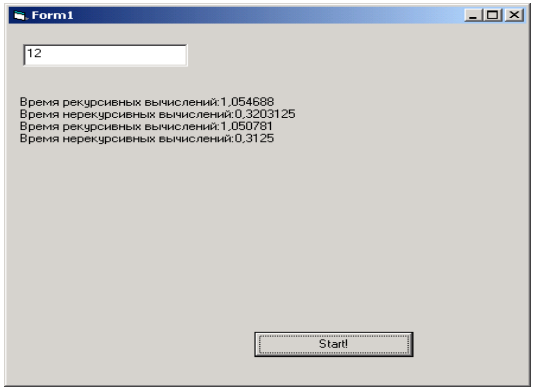


Рис.5.1. Порівняння за часом рекурсивної і нерекурсивної реалізації факторіалу

Рекурсивне обчислення найбільшого загального дільника

Найбільшим загальним дільником (greatest common divisor, GCD) двох чисел називається найбільше ціле, на яке діляться два числа без залишку. Наприклад, найбільший загальний дільник чисел 12 і 9 дорівнює 3. Два числа називаються взаємно простими (relatively prime), якщо їхній найбільший загальний дільник дорівнює 1.

Математик Ейлер, що жив у вісімнадцятому столітті, виявив цікавий факт: якщо A без остачі ділиться на B, то `GCD(A, B)= A.`

Інакше `GCD(A, B)= GCD(B Mod A, A).`

Цей факт можна використовувати для швидкого обчислення найбільшого загального дільника. Наприклад:

GCD(9, 12) = GCD(12 Mod 9, 9) = GCD(3, 9) = 3

На кожному кроці числа стають все менше, оскільки `1<=B Mod A<A`, якщо `A` не ділиться на `B` без залишку. У міру зменшення аргументів, врешті-решт, A набуде значення 1. Оскільки будь-яке число ділиться на 1 без залишку, на цьому кроці рекурсія зупиниться. Таким чином, в якій-небудь момент `B` розділиться на `A` без залишку, і робота процедури завершиться.

Відкриття Ейлера закономірним чином наводить до рекурсивного алгоритму обчислення найбільшого загального дільника:

```
public Function GCD(A As Integer, B As Integer) As
  Integer
  If B Mod A = 0 Then      ' Чи ділиться B на A без
  залишка?
       GCD = A             ' Так. Процедура завершена.
  Else
       GCD = GCD(B Mod A, A)      ' Немає. Рекурсія.
  End If
End Function
```

Щоб проаналізувати час виконання цього алгоритму, необхідно визначити, наскільки швидко зменшується змінна `A`. Оскільки функція зупиняється, коли `A` доходить до значення 1, то швидкість зменшення `A` дає верхній кордон оцінки часу виконання алгоритму. Виявляється, при кожному другому виклику функції `GCD` параметр `A` зменшується, принаймні, в 2 рази.

Припустимо, `A < B`. Ця умова завжди виконується при першому виклику функції `GCD`. Якщо `B Mod A <= A/2`, то при

наступному виклику функції GCD перший параметр зменшиться, принаймні, в 2 рази, і доказ закінчений.

Передбачимо зворотне. Допустимо, B Mod A > A / 2. Першим рекурсивним викликом функції GCD буде GCD(B Mod A, A).

Підстановка у функцію значення B Mod A і A замість A і B дає наступний рекурсивний виклик GCD(B Mod A, A).

Але ми передбачили, що B Mod A > A / 2. Тоді B Mod A розділиться на A лише один раз, із залишком A – (B Mod A). Оскільки B Mod A більше, ніж A / 2, то A – (B Mod A) повинне бути менше, ніж A / 2. Значить, перший параметр другого рекурсивного виклику функції GCD менший, ніж A / 2, що і потрібно було довести.

Передбачимо тепер, що N — це вихідне значення параметра A. Після двох викликів функції GCD значення параметра A повинне зменшитись, принаймні, до N / 2. Після чотирьох викликів це значення буде не більше, ніж (N / 2) / 2 = N / 4. Після шести викликів значення не перевершуватиме (N / 4) / 2 = N / 8. У загальному випадку після 2 * K викликів функції GCD значення параметра A буде не більше, ніж N / 2K.

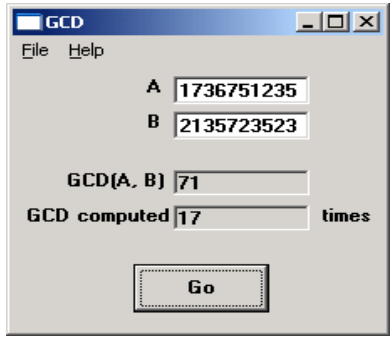


Рис.5.2. Результат обчислення найбільшого загального дільника

Оскільки алгоритм повинен зупинитися, коли значення параметра A дійде до 1, він може продовжувати працю лише до тих пір, поки не виконається рівність N/2K=1. Це відбувається, коли N=2K або коли K=log2(N). Оскільки алгоритм виконується за 2*K кроків, це означає, що алгоритм зупиниться не більш, ніж через 2*log2(N) кроки. З точністю до постійного множника це означає, що алгоритм виконується за час порядку O(log(N)). Алгоритми

порядку O(log(N)) зазвичай виконуються дуже швидко, і алгоритм знаходження найбільшого загального дільника не є виключенням з цього правила. Наприклад, щоб знайти, що найбільший загальний дільник чисел 1.736.751.235 і 2.135.723.523 дорівнює 71, функція викликається всього 17 разів. Фактично алгоритм практично миттєво обчислює значення, що не перевищують максимального значення числа у форматі long , — 2.147.483.647. Функція Visual Basic Mod не може оперувати значеннями, більшими за це, тому це практична межа для даної реалізації алгоритму.

Користь від рекурсивних процедур більшою мірою може виявитися при обробці даних, що мають рекурсивну структуру (скажімо, ієрархічну або мережеву).

Небезпеки рекурсії

Рекурсія може служити потужним методом розбиття великих завдань на частини, але вона таїть у собі кілька небезпек. У цьому розділі ми розглянемо деякі з цих небезпек і пояснимо, коли варто, а коли не варто використовувати рекурсію.

Нескінченна рекурсія

Найбільш очевидна небезпека рекурсії полягає в нескінченній рекурсії. Якщо неправильно побудувати алгоритм, то функція може пропустити умову зупинки рекурсії й буде виконуватися нескінченно. Найпростіше зробити цю помилку, якщо просто забути про перевірку умови зупинки, як це зроблено в наступній помилковій версії функції факторіала. Оскільки функція не перевіряє, чи досягнута умова зупинки рекурсії, вона буде нескінченно викликати саму себе.

```
Private Function BadFactorial(num As Integer) As
  Integer
  BadFactorial = num * BadFactorial (num — 1)
End Function
```

Функція також може викликати себе нескінченно, якщо умова зупинки не припиняє всі можливі шляхи рекурсії. У наступній

помилковій версії функції факторіала, функція буде нескінченно викликати себе, якщо вхідне значення — не ціле число, або якщо воно менше 0. Ці значення не є припустимими вхідними значеннями для функції факторіала, тому в програмі, що використовує цю функцію, може знадобитися перевірка вхідних значень. Проте буде краще, якщо функція виконає цю перевірку сама.

```
Private Function BadFactorial2(num As Double) As Double
  If num = 0 Then
       BadFactorial2 = 1
  Else
       BadFactorial2 = num * BadFactorial2( num-1)
  End If
End Function
```

Наступна версія функції `Fibonacci` є більш складним прикладом. У ній умова зупинки рекурсії припиняє виконання тільки декількох шляхів рекурсії, і виникають ті ж проблеми, що й при виконанні функції `BadFactorial2`, якщо вхідні значення від'ємні або не цілі.

```
Private Function BadFib(num As Double) As Double
  If num = 0 Then
       BadFib = 0
  Else
       BadFib = BadPib(num — 1) + BadFib (num — 2)
  End If
End Function
```

І остання проблема, пов'язана з нескінченною рекурсією, полягає в тім, що «нескінченна» насправді означає «доти, поки не буде вичерпаний стековий простір». Навіть коректно написані рекурсивні процедури будуть іноді приводити до переповнення стека й аварійне завершення роботи. Наступна функція, що обчислює суму `N + (N — 1) + … + 2 +1`, приводить до вичерпання стекового простору при великих значеннях `N`. Найбільше можливе значення `N`, при якому програма ще буде працювати, залежить від конфігурації вашого комп'ютера.

```
Private Function BigAdd(N As Double) As Double
  If N <= 1 Then
```

```
      BigAdd = 1
  Else
      BigAdd = N + BigAdd(N — 1)
  End If
End Function
```

Програма BigAdd (диск з прикладами – папка ProgR5) демонструє цей алгоритм. Перевірте, наскільки велике вхідне значення ви можете ввести в цій програмі до того, як наступить переповнення стеку на вашому комп'ютері.

Втрати пам'яті

Інша небезпека рекурсії полягає у втратах пам'яті. При кожному виклику підпрограми система виділяє пам'ять для локальних змінних нової процедури. Під час складної послідовності рекурсивних викликів значна частина часу й пам'яті комп'ютера буде йти на виділення й звільнення пам'яті для цих змінних під час рекурсії. Навіть якщо це не приведе до вичерпання стекового простору, час, витрачений на роботу зі змінними, може бути значним.

Існує кілька способів зменшення цих накладних витрат. По-перше, не слід використовувати велику кількість непотрібних змінних. Навіть якщо підпрограма не використовує їх, Visual Basic однаково буде відводити пам'ять під ці змінні. Наступна версія функції BigAdd ще швидше приводить до переповнення стеку, ніж попередня.

```
Private Function BigAdd(N As Double) As Double
Dim I1 As Integer
Dim I2 As Integer
Dim I3 As Integer
Dim I4 As Integer
Dim I5 As Integer

  If N <= 1 Then
      BigAdd = 1
  Else
      BigAdd = N + BigAdd (N — 1)
  End If
End Function
```

Якщо ви не впевнені, чи потрібна змінна, використовуйте оператор Option Explicit і закоментуйте визначення змінної.

При спробі виконати програму Visual Basic повідомить про помилку, якщо змінна використовується в програмі.

Ви також можете зменшити використання стеку за рахунок застосування глобальних змінних. Якщо ви визначите змінні в секції `Declarations` модуля замість того, щоб визначати їх у підпрограмі, то системі не знадобиться відводити пам'ять при кожному виклику підпрограми.

Кращим рішенням буде визначення змінних у процедурі за допомогою зарезервованого слова `Static`. Статичні змінні використовуються спільно всіма екземплярами процедури, і системі не потрібно відводити пам'ять під нові копії змінних при кожному виклику підпрограми.

Необґрунтоване застосування рекурсії

Менш очевидною небезпекою є необґрунтоване застосування рекурсії. При цьому використання рекурсії не є найкращим способом рішення завдання. Наведені вище функції факторіала, найбільшого загального дільника, чисел Фібоначчі й функції `BigAdd` не обов'язково повинні бути рекурсивними.

У випадку факторіала й найбільшого загального дільника непотрібна рекурсія є по більшій частині нешкідливою. Обидві ці функції виконуються досить швидко для досить великих вихідних значень. Їхнє виконання також не буде обмежене розміром стеку, якщо ви не використовували більшу частину стекового простору в інших частинах програми.

З іншого боку, застосування рекурсії погіршує алгоритм обчислення чисел Фібоначи. Для обчислення `Fib(N)`, алгоритм спочатку обраховує `Fib(N – 1)` і `Fib(N – 2)`. Але для обрахування `Fib(N – 1)` він повинен спочатку обрахувати `Fib(N – 2)` і `Fib(N – 3)`. При цьому `Fib(N – 2)` обраховується двічі.

Попередній аналіз цього алгоритму показав, що `Fib(1)` і `Fib(0)` обраховуються `Fib(N + 1)` раз під час обрахування `Fib(N)`. Тому що `Fib(30) = 832.040` те, щоб обрахувати `Fib(29)`, доводиться браховувати ті самі значення `Fib(0)` і `Fib(1)` 832.040 разів. Алгоритм обрахування чисел Фібоначи витрачає величезну кількість часу на обрахування цих проміжних значень знову й знову.

У функції `BigAdd` існує інша проблема. Хоча вона виконується швидко, але приводить до великої глибини вкладеності

рекурсії, і дуже швидко вичерпує стековий простір. Якби не переповнення стеку, ця функція могла б обраховувати результати для більших вхідних значень.

Схожа проблема існує й у функції факторіала. Для вхідного значення `N` глибина рекурсії для факторіала й функції `BigAdd` дорівнює `N`. Функція факторіала не може бути обрахована для таких більших вхідних значень, які припустимі для функції `BigAdd`. Максимальне значення факторіала, що може вміститися в змінній типу `double`, дорівнює `170! ( 7,257E+30)`, тому це найбільше значення, що може обраховувати ця функція. Хоча ця функція приводить до глибокої рекурсії, вона викликає переповнення до того, як наступить переповнення стеку.

Коли потрібно використовувати рекурсію

Ці міркування можуть змусити вас думати, що рекурсія завжди небажана. Але це звичайно не так. Багато алгоритмів є рекурсивними за своєю природою. І хоча будь-який алгоритм можна переписати так, щоб він не містив рекурсії, багато алгоритмів складніше розуміти, аналізувати, налагоджувати й підтримувати, якщо вони написані нерекурсивно.

Якщо алгоритм рекурсивний за природою, записуйте його з використанням рекурсії. У найкращому разі ви не зустрінетеся з жодною з описаних проблем. Якщо ж ви стикнетеся з деякими з них, ви зможете переписати алгоритм без використання рекурсії за допомогою методів, представлених у наступних розділах. Переписати алгоритм часто набагато простіше, ніж із самого початку написати його без застосування рекурсії.

Хвостова рекурсія

Згадаємо представлені раніше функції для обрахування факторіалів і найбільшого загального дільника, а також функцію `BigAdd`, що приводить до переповнення стеку навіть для відносно невеликих вхідних значень.

```
Private Function Factorial(num As Integer) As Integer
  If num <= 0 Then
      Factorial = 1
```

```
  Else
      Factorial = num * Factorial(num — 1)
  End If
End Function

Private Function GCD(A As Integer, B As Integer) As
  Integer
  If B Mod A = 0 Then
      GCD = A
  Else
      GCD = GCD(B Mod A, A)
  End If
End Function

Private Function BigAdd(N As Double) As Double
  If N <= 1 Then
      BigAdd = 1
  Else
      BigAdd = N + BigAdd(N — 1)
  End If
End Function
```

У всіх цих функціях остання дія перед завершенням функції — це рекурсивний крок. Цей тип рекурсії наприкінці процедури називається *хвостовою рекурсією* (tail recursion).

Через те що після рекурсії в процедурі нічого не відбувається, існує простий спосіб її усунення. Замість рекурсивного виклику функції процедура скидає свої параметри, установлюючи ті, які б вона одержала при рекурсивному виклику, і потім виконується знову.

Розглянемо загальний випадок рекурсивної процедури:

```
Private Sub Recurse(A As Integer)
  ' Виконуються які- або дії, обчислюється B, і т.д.
  Recurse B
End Sub
```

Цю процедуру можна переписати без рекурсії як:

```
Private Sub NoRecurse(A As Integer)
  Do While (not done)
      ' Виконуються які- або дії, обчислюється B, і
  т.д.
      A = B
  Loop
End Sub
```

Ця процедура називається *усуненням хвостової рекурсії* (tail recursion removal). Цей прийом не змінює час виконання програми. Рекурсивні кроки просто заміняються проходами в циклі `While`.

Усунення хвостової рекурсії, проте, усуває виклики підпрограм, і тому може збільшити швидкість роботи алгоритму. Що більш важливо, цей метод також зменшує використання стеку.

Деякі компілятори автоматично усувають хвостову рекурсію, але компілятор Visual Basic цього не робить.

Використовуючи усунення хвостової рекурсії, легко переписати функції факторіала, найбільшого загального дільника, і `BigAdd` без рекурсії. Ці версії використовують зарезервоване слово `ByVal` для збереження значень своїх параметрів для викликаючої процедури.

```
Private Function Factorial(ByVal N As Integer) As
  Double
Dim value As Double

  value = 1#              ' Це буде значенням функції.
  Do While N > 1
       value = value * N
       N = N — 1          ' Підготувати аргументи для
  "рекурсії".
  Loop
  Factorial = value
End Function

Private Function GCD(ByVal A As Double, ByVal B As
  Double) As Double
Dim B_Mod_A As Double

  B_Mod_A = B Mod A
  Do While B_Mod_A <> 0
       ' Підготувати аргументи для "рекурсії".
       B = A
       A = B_Mod_A
       B_Mod_A = B Mod A
  Loop
  GCD = A
End Function

Private Function BigAdd(ByVal N As Double) As Double
Dim value As Double

  value = 1#        ' ' Це буде значенням функції.
```

```
  Do While N > 1
      value = value + N
      N = N — 1     ' підготувати параметри для
  "рекурсії".
  Loop
  BigAdd = value
End Function
```

Для алгоритмів обрахування факторіала й найбільшого загального дільника практично не існує різниці між рекурсивною й нерекурсивною версіями. Обидві версії виконуються досить швидко, і обидві можуть оперувати завданнями великої розмірності.

Для функції `BigAdd`, проте, різниця величезна. Рекурсивна версія призводить до переповнення стеку навіть для досить невеликих вхідних значень. Оскільки нерекурсивна версія не використовує стек, вона може обраховувати результат для значень **N** аж до 10^{154}. Після цього наступить переповнення для даних типу `double`. Звичайно, виконання 10^{154} кроків алгоритму займе дуже багато часу, тому можливо ви не станете перевіряти цей факт самі. Помітимо також, що значення цієї функції збігається зі значенням більш просто обраховуваної функції, `N * N(N + 1) / 2`.

Нерекурсивне обрахування чисел Фібоначи

На жаль, нерекурсивний алгоритм обрахування чисел Фібоначи не містить тільки хвостову рекурсію. Цей алгоритм використовує два рекурсивних виклики для обрахування значення, і другий виклик йде після завершення першого. Оскільки перший виклик не перебуває в самому кінці функції, то це не хвостова рекурсія, і від її не можна позбутися, використовуючи прийом усунення хвостової рекурсії.

Це може бути зв'язане й з тим, що обмеження рекурсивного алгоритму обрахування чисел Фібоначи пов'язане з тим, що він обраховує занадто багато проміжних значень, а не глибиною вкладеності рекурсії. Усунення хвостової рекурсії зменшує глибину рекурсії, але не змінює час виконання алгоритму. Навіть якби усунення хвостової рекурсії було б застосовне до алгоритму обрахування чисел Фібоначи, цей алгоритм однаково залишився б надзвичайно повільним.

Проблема цього алгоритму в тім, що він багаторазово обраховує ті ж значення. Значення `Fib(1)` і `Fib(0)` обраховуються `Fib(N + 1)` раз, коли алгоритм обраховує `Fib(N)`. Для обрахування `Fib(29)`, алгоритм обраховує ті самі значення `Fib(0)` і `Fib(1)` 832.040 разів.

Оскільки алгоритм багаторазово обраховує ті самі значення, варто знайти спосіб уникнути повторення обрахувань. Простий і конструктивний спосіб зробити це — побудувати таблицю обрахованих значень. Коли знадобиться проміжне значення, його можна буде взяти з таблиці, замість того щоб обраховувати заново.

У цьому прикладі можна створити таблицю для зберігання значень функції Фібоначи `Fib(N)` для `N`, що не перевершують 1477. Для `N >= 1477` відбувається переповнення змінних типу `double`, використовуваних у функції. Наступний код містить змінену в такий спосіб функцію, що обраховує числа Фібоначи

```
Const MAX_FIB = 1476              ' Максимальне значення.

Dim FibValues(0 To MAX_FIB) As Double

Private Function Fib(N As Integer) As Double
  ' Обчислити значення, якщо воно не перебуває в
  таблиці.
  If FibValues(N) < 0 Then _
       FibValues(M) = Fib(N — 1) + Fib(N — 2)

  Fib = FibValues(N)
End Function
```

При запуску програми вона привласнює кожному елементу в масиві `FibValues` значення -1. Потім вона привласнює `FibValues(0)` значення 0, і `FibValues(1)` — значення 1. Це умови зупинки рекурсії.

При виконанні функції вона перевіряє, чи перебуває вже в масиві значення, що їй потрібне. Якщо його там немає, вона, як і раніше, рекурсивно обраховує це значення й зберігає його в масиві для подальшого використання.

Програма `Fibo2 (диск з прикладами – папка ProgR5)` використовує цей метод для обрахування чисел Фибоначчи. Програма може швидко обрахувати `Fib(N)` для `N` до 100 або 200. Але якщо ви спробуєте обрахувати `Fib(1476)`, то програма

виконає послідовність рекурсивних викликів глибиною 1476 рівнів, що ймовірно переповнить стек вашої системи.

Проте у міру того як програма обраховує нові значення, вона заповнює масив FibValues. Значення з масиву дозволяють функції обраховувати все більші й більші значення без глибокої рекурсії. Наприклад, якщо обрахувати послідовно Fib(100), Fib(200), Fib(300), і т.д. то, зрештою, можна буде заповнити масив значень FibValues і обрахувати максимально можливе значення Fib(1476).

Процес повільного заповнення масиву FibValues приводить до нового методу обрахування чисел Фібоначи. Коли програма ініціалізує масив FibValues, вона може заздалегідь обрахувати всі числа Фібоначи.

```
Private Sub InitializeFibValues()
Dim i As Integer

  FibValues(0) = 0        ' Ініціалізація умов зупинки.
  FibValues(1) = 1
  For i = 2 To MAX_FIB
     FibValues(i) = FibValues(i — 1) + FibValues(i — 2)
  Next i
End Sub

Private Function Fib(N As Integer) As Duble
  Fib — FibValues(N)
End Function
```

Певний час у цьому алгоритмі займає складання масиву з табличними значеннями. Але після того як масив створений, для одержання елемента з масиву потрібний всього один крок. Ні процедура ініціалізації, ні функція Fib не використовують рекурсію, тому жодна з них не приведе до вичерпання стекового простору. Програма Fibo3 (диск з прикладами – папка ProgR5) демонструє цей підхід.

Варто згадати ще один метод обрахування чисел Фібоначи. Перше рекурсивне визначення функції Фібоначи використовує підхід зверху вниз. Для одержання значення Fib(N) алгоритм рекурсивно обраховує Fib(N — 1) і Fib(N — 2) і потім складає їх.

Підпрограма InitializeFibValues, з іншого боку, працює знизу уверх. Вона починає зі значень Fib(0) і Fib(1).

Вона потім використовує менші значення для обрахування більших, доти, поки таблиця не заповниться.

Ви можете використовувати той же підхід знизу уверх для прямого обрахування значень функції Фібоначи щораз, коли вам буде потрібне значення. Цей метод вимагає більше часу, ніж вибірка значень із масиву, але не вимагає додаткової пам'яті для таблиці значень. Це приклад просторово- тимчасового компромісу. Використання більшого обсягу пам'яті для зберігання таблиці значень робить виконання алгоритму більш швидким.

```
Private Function Fib(N As Integer) As Double
Dim Fib_i_minus_1 As Double
Dim Fib_i_minus_2 As Double
Dim fib_i As Double
Dim i As Integer

  If N <= 1 Then
       Fib = N
  Else
       Fib_i_minus_2 = 0          ' Спочатку Fib(0)
       Fib_i_minus_1 = 1          ' Спочатку Fib(1)
       For i = 2 To N
              fib_i = Fib_i_minus_1 + Fib_i_minus_2
              Fib_i_minus_2 = Fib_i_minus_1
              Fib_i_minus_1 = fib_i
       Next i
       Fib = fib_i
  End If
End Function
```

Усунення рекурсії в загальному випадку

Функції факторіала, найбільшого загального дільника, і `BigAdd` можна спростити усуненням хвостової рекурсії. Функцію, що обраховує числа Фібоначи, можна спростити, використовуючи таблицю значень або переформулювавши завдання з використанням підходу знизу нагору.

Деякі рекурсивні алгоритми настільки складні, що застосування цих методів утруднене або неможливе.

Основний підхід при цьому полягає в тім, щоб розглянути порядок виконання рекурсії на комп'ютері й потім спробувати

сімітувати кроки, виконувані комп'ютером. Потім новий алгоритм буде сам здійснювати «рекурсію» замість того, щоб всю роботу виконував комп'ютер.

Оскільки новий алгоритм виконує практично ті ж кроки, що й комп'ютер, можна поцікавитися, чи зросте швидкість обрахувань. В Visual Basic це звичайно не виконується. Комп'ютер може виконувати завдання, які потрібні при рекурсії, швидше, ніж ви можете їх імітувати. Проте оперування цими деталями самостійно забезпечує кращий контроль над виділенням пам'яті під локальні змінні, і дозволяє уникнути глибокого рівня вкладеності рекурсії.

Звичайно, при виклику підпрограми система виконує три речі. По-перше, зберігає дані, які потрібні їй для продовження виконання після завершення підпрограми. По-друге, вона проводить підготовку до виклику підпрограми й передає їй керування. По-третє, коли викликувана процедура завершується, система відновлює дані, збережені на першому кроці, і передає керування назад у відповідну крапку програми. Якщо ви перетворите рекурсивну процедуру в нерекурсивну, вам доведеться виконувати ці три кроки самостійно.

Розглянемо наступну узагальнену рекурсивну процедуру:

```
Sub Subr(num)
  <1 блок коду>
  Subr(<параметри>)
  <2 блок коду>
End Sub
```

Оскільки після рекурсивного кроку є ще оператори, ви не можете використовувати усунення хвостової рекурсії для цього алгоритму.

Спочатку позначимо перші рядки в 1 і 2 блоках коду. Потім ці мітки будуть використовуватися для визначення місця, з якого потрібно продовжити виконання при поверненні з «рекурсії». Ці мітки використовуються тільки для того, щоб допомогти вам зрозуміти, що робить алгоритм — вони не є частиною коду Visual Basic. У цьому прикладі мітки будуть виглядати так:

```
  Sub Subr(num)
1      <1 блок коду>
       Subr(<параметри>)
2      <2 блок коду>
  End Sub
```

Використовуємо спеціальну мітку «0» для позначення кінця «рекурсії». Тепер можна переписати процедуру без використання рекурсії, наприклад, так:

```
Sub Subr(num)
Dim pc As Integer   ' Визначає, де потрібно продовжити
  рекурсію.

  pc = 1            ' Почати спочатку.
  Do
      Select Case pc
             Case 1
      <1 блок коду>
      If (досягнута умова зупинки) Then pc = 2
' Пропустити рекурсію й перейти до блоку 2.

                    Else
: pc = 1

  ' Зберегти змінні, потрібні після рекурсії.
  ' Зберегти pc = 2. Крапка, з якої продовжиться
  ' виконання після повернення з "рекурсії".
  ' Установити змінні, потрібні для рекурсії.
  ' Наприклад, num = num — 1.
      ' Перейти до блоку 1 для початку рекурсії.
                    End If
             Case 2       ' Виконати 2 блок коду
                    <2 блок коду>
                    pc = 0
             Case 0
                    If (це остання рекурсія) Then Exit
  Do
                    ' Інакше відновити pc і інші змінні,
                    ' збережені перед рекурсією.
      End Select
  Loop
End Sub
```

Змінна `pc`, що відповідає лічильнику програми, повідомляє процедуру, який крок вона повинна виконати наступним. Наприклад, при `pc = 1`, процедура повинна виконати 1 блок коду.

Коли процедура досягає умови зупинки, вона не виконує рекурсію. Замість цього вона привласнює `pc` значення 2 і продовжує виконання 2 блоку коду.

Якщо процедура не досягла умови зупинки, вона виконує «рекурсію». Для цього вона зберігає значення всіх локальних змінних, які їй знадобляться пізніше після завершення «рекурсії».

Вона також зберігає значення pc для ділянки коду, що буде виконувати після завершення «рекурсії». У цьому прикладі наступним виконується 2 блок коду, тому вона зберігає 2 як наступне значення pc. Найпростіший спосіб збереження значень локальних змінних і pc складається у використанні стеків.

Реальний приклад допоможе вам зрозуміти цю схему. Розглянемо злегка змінену версію функції факторіала. У ньому переписана тільки підпрограма, що повертає своє значення за допомогою змінної, а не функції (для спрощення роботи).

```
  Private Sub Factorial(num As Integer, value As
  Integer)
  Dim partial As Integer
1      If num <= 1 Then
             value = 1
       Else
             Factorial(num — 1, partial)
2            value = num * partial
       End If
  End Sub
```

Після повернення процедури з рекурсії потрібно довідатися вихідне значення змінної num, щоб виконати операцію множення value = num * partial. Оскільки процедурі потрібен доступ до значення num після повернення з рекурсії, вона повинна зберігати значення змінних pc і num до початку рекурсії.

Наступна процедура зберігає ці значення у двох стеках на основі масивів. При підготовці до рекурсії вона проштовхує значення змінних num і pc у стеки. Після завершення рекурсії вона виштовхує додані останніми значення зі стеків. Наступний код демонструє нерекурсивну версію підпрограми обрахування факторіала.

```
Private Sub Factorial(num As Integer, value As Integer)
ReDim num_stack(1 to 200) As Integer
ReDim pc_stack(1 to 200) As Integer
Dim stack_top As Integer         ' Вершина стека.
Dim pc As Integer

  pc = 1
  Do
       Select Case pc
             Case 1
                   If num <= 1 Then          ' Ця умова
  зупинки.                                   value = 1
```

```
                        pc = 0              ' Кінець
  рекурсії.
                    Else                    '
  Рекурсія.
                        ' Зберегти num і наступне
  значення pc.
                        stack_top = stack_top + 1
                        num_stack(stack_top) = num
                        pc_stack(stack_top) = 2   '
  Відновити з 2.
                        ' Почати рекурсію.
                        num = num — 1
                        ' Перенести блок керування в
  початок.
                        pc = 1
                    End If
              Case 2
                    ' value містить результат останньої
                    ' рекурсії. Помножити його на num.
                    value = value * num
                    ' "Повернення" з "рекурсії".
                    pc = 0
              Case 0
                    ' Кінець "рекурсії".
                    ' Якщо стеки порожні, вихідний
  виклик
                    ' підпрограми завершений.
                    If stack_top <= 0 Then Exit Do
                    ' Інакше відновити локальні змінні й
  pc.
                    num = num_stack(stack_top)
                    pc = pc_stack(stack_top)
                    stack_top = stac_top — 1
              End Select
        Loop
End Sub
```

Так само, як і усунення хвостової рекурсії, цей метод імітує поводження рекурсивного алгоритму. Процедура заміняє кожний рекурсивний виклик ітерацією циклу `While`. Оскільки число кроків залишається тим же самим, повний час виконання алгоритму не змінюється.

Так само, як і у випадку з усуненням хвостової рекурсії, цей метод усуває глибоку рекурсію, що може переповнити стек.

Рекурсивна побудова кривих Гільберта

Криві Гільберта (Hilbert curves) — це самоподібні (self-similar) криві, які зазвичай визначаються за допомогою рекурсії. На рис. показані криві Гільберта з 1, 2 або 3 порядку.

Крива Гільберта, як і будь-яка інша самоподібна крива, створюється розбиттям великої кривої на менші частини. Далі ви можете використовувати цю ж криву, після зміни розміру і повороту, для побудови цих частин. Ці частини можна розбити на дрібніші частини, і так далі, поки процес не досягне потрібної глибини рекурсії. Порядок кривої визначається як максимальна глибина рекурсії, яку досягає процедура.

Процедура Hilbert управляє глибиною рекурсії, використовуючи відповідний параметр. При кожному рекурсивному виклику процедура зменшує параметр глибини рекурсії на одиницю. Якщо процедура викликається з глибиною рекурсії, що дорівнює 1, вона рисує просту криву 1 порядку, показану на рис. зліва, і завершує роботу. Це умова зупинки рекурсії.

Наприклад, крива Гільберта 2 порядку складається з чотирьох кривих Гільберта 1 порядку. Аналогічно крива Гільберта 3 порядку складається з чотирьох кривих 2 порядку, кожна з яких складається з чотирьох кривих 1 порядку. На рис. показані криві Гільберта 1, 2 і 3 порядку. Менші криві, з яких побудовані криві більшого розміру, виділені напівжирними лініями.

Наступний код будує криву Гільберта 1 порядку:

```
Line -Step (Length, 0)
Line -Step (0, Length)
Line -Step (-Length, 0)
```

Передбачається, що малювання починається з верхнього лівого кута області і що Length — це задана довжина кожного відрізка ліній.

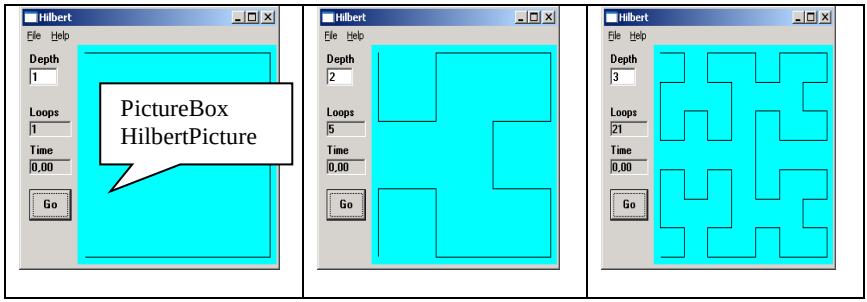


Рис.5.3. Інтерфейс програми малювання кривих Гільберта

Програма малювання кривих Гільберта

```
' ************************************************
' Hilbert.FRM
'' Приклад рекурсивного малювання кривих Гільберта
' ************************************************
' Кафедра ЕК, 2003.
' ************************************************
Option Explicit
Dim Loops As Double ' #
'******************************************************
' Починаємо малювати криву.
' ******************************************************
Private Sub CmdGo_Click()
Dim start_time As Double
Dim depth As Integer
Dim gaps As Single
Dim dist As Single
Dim margin As Single
    LoopsLabel.Caption = ""
    Loops = 0
    If Not IsNumeric(DepthText.Text) Then
        Beep
        Exit Sub
    End If
    depth = CInt(DepthText.Text)
    If depth < 1 Or depth > 10 Then
        Beep
        Exit Sub
```

```
    End If
    margin = 120
    gaps = (2 ^ depth) — 1
    dist = (HilbertPicture.ScaleWidth — 2 * margin) /
  gaps
    HilbertPicture.Cls
    HilbertPicture.CurrentX = margin
    HilbertPicture.CurrentY = margin
     start_time = Timer
    Hilbert depth, dist, 0
    TimeLabel.Caption = Format$(Timer — start_time,
  "0.00")
    LoopsLabel.Caption = Format$(Loops)
End Sub
' ****************************************************
' Робимо розмір малюнка максирисьним для нового
' розміру форми.
' ****************************************************
Private Sub Form_Resize()
Dim wid As Single
Dim hgt As Single
    '    If WindowState = 1 Then Exit Sub
    wid = ScaleWidth — HilbertPicture.Left —
  HilbertPicture.Top
    hgt = ScaleHeight — 2 * HilbertPicture.Top
    If hgt < wid Then wid = hgt
    HilbertPicture.Width = wid
    HilbertPicture.Height = wid
End Sub
' ***********************************************
' Рисуємо криву Гільбертаhe Hilbert.
'' Dx і Dy вказують напрям для першої
' частини кривої.. Іншими словами, якщо крива
' починається в крапці (x, в), вона повинна закінчитися
  в
' (X + Dx, в + Dy) після прорисьовування першої третини
' кривої.
' ***********************************************
Private Sub Hilbert(ByVal depth As Integer, ByVal Dx As
  Single, ByVal Dy As Single)
    Loops = Loops + 1
```

```
    If depth > 1 Then Hilbert depth — 1, Dy, Dx
    HilbertPicture.Line -Step(Dx, Dy)
    If depth > 1 Then Hilbert depth — 1, Dx, Dy
    HilbertPicture.Line -Step(Dy, Dx)
    If depth > 1 Then Hilbert depth — 1, Dx, Dy
    HilbertPicture.Line -Step(-Dx, -Dy)
    If depth > 1 Then Hilbert depth — 1, -Dy, -Dx
End Sub
Private Sub mnuFileExit_Click()
    End
End Sub
```

Операції та вбудовані функції

Операції

У будь-якій мові програмування припустимі вирази. Вирази будуються зі змінних, констант, убудованих функцій з використанням знаків операцій і дужок. Запис виразу задає правило (алгоритм) обрахування його значення і його типу. Тип і значення всіх змінних виразу повинні бути визначені до моменту обрахування. Мови програмування розрізняються між собою тим, до якого ступеня вони допускають автоматичне перетворення типів даних у процесі обрахування виразу. Вважається, що мова є більш надійною та охороняє від багатьох помилок програміста, якщо вона (мова) не допускає автоматичного перетворення типів. Звичайно, тут необхідний розумний компроміс. Так, всі мови допускають при обрахуванні виразу `X+Y` дійсний тип для змінної `X` і цілочисельний тип для змінної `Y`, проводячи автоматичне перетворення до дійсного типу й виконуючи, потім уже, додавання дійсних чисел. Мова VBA щодо цього займає золоту середину. Серед її убудованих функцій є велика кількість функцій, призначених для явного перетворення типів, що дозволяє програмувати в кращих традиціях надійних мов програмування, не довіряючи неявним перетворенням.

Наведемо основні операції, які можна виконувати над даними в мові VBA, класифікуючи їх за типом й пріоритетом:

Таблиця 5.4

Операції та їхній пріоритет

Пріоритет	**Арифметичні**	**Порівняння**	**Логічні**	**Опис деяких операцій**
1	Зведення в ступінь — (^)	Рівність — (=)	Заперечення — (Not)	При піднесенні до степеня основа й показник можуть бути арифметичними виразами будь-якого типу
2	Унарний мінус — (—)	Нерівність — (<>)	Кон’юнкція — (And)	
3	Множення, Ділення — (*, /)	Менше — (<)	Диз'юнкція — (Or)	
4	Ділення націло — (\)	Більше — (>)	Виняткове Або — (Xor)	Ділення націло визначено над цілочисельними даними і дає результат цілого типу.
5	Залишок від ділення націло — (mod)	Менше або дорівнює — (<=)	Еквівалентність — (Eqv)	Операція mod визначена над даними цілого типу й повертає результат цілого типу — залишок від ділення націло.
6	Додавання, віднімання — (+,-)	Більше або дорівнює — (>=)	Імплікація — (Imp)	Серед логічних операцій визначена операція імплікація, помилкова в єдиному випадку, коли посилка True, а висновок False.

Продовження табл. 5.4

7	Конкатенація рядків — (&)	Подоби — (Like), Рівність посилань — (Is)		Операція Like перевіряє відповідність рядка зразку. Операція Is, діє над об'єктами, не перевіряє рівність самих об'єктів, вона перевіряє збіг посилань. Посилання повинні задавати той же самий об'єкт.

Якщо вираз містить операції різних категорій, то першими виконуються арифметичні операції, потім операції порівняння й останніми — логічні.

Всі операції порівняння мають один пріоритет. Арифметичні й логічні операції виконуються відповідно до визначеного пріоритету.

Операція, яка записана кілька разів підряд, або операції одного пріоритету (множення й ділення, додавання й віднімання) виконуються зліва направо, — із двох операцій першою виконується та, котра йде лівіше в записі виразу.

Дужки дозволяють змінити зазначений порядок обрахувань виразу, оскільки вираз в дужках має найвищий пріоритет, то обраховується першим. Усередині дужок діє звичайний порядок обрахування.

Операція конкатенації не є арифметичною, вона виконується після всіх арифметичних операцій, але до операцій порівняння.

Робота із числовими даними

Арифметика в VBA представлена досить повно. Нагадаємо, що арифметичний тип підрозділяється на підтипи:

- `Byte`, `Integer`, `Long` — для подання цілочисельних даних.
- `Single`, `Double` — для подання дійсних даних.
- `Decimal` — для подання чисел у формі з фіксованою точкою, що важливо, зокрема, для фінансових обрахувань.
- `Currency` — спеціальний тип для подання грошових даних.

- `Variant` — узагальнений тип, що дозволяє зберігати й обробляти дані різного типу.
- Приклад роботи із числовими даними:

```
Public Sub WorkWithArithmetic()
  Dim X As Integer, Y As Integer
  Dim U As Single, V As Single
  Dim Z As Double
  U = 15.8: V = -6.5
  Z = U / V: X = CInt(U / V): Y = U \ V
  Debug.Print X, Y, Z, U, V, X \ Y, X Mod Y, U Mod V

End Sub
```

Ось результати друку у вікні налагодження:

```
-2    -2    -2,43076926011306    15,8    -6,5    1    0    4
```

Помітимо, що хоча цілочисельні операції можливі над дійсними даними, застосовувати їх не треба, оскільки це один з тих випадків, коли виконуються внутрішні перетворення, точна інтерпретація яких складна, що може, в остаточному підсумку, бути джерелом програмістських помилок.

Розглянемо основні вбудовані математичні функції.

Математичні функції

Набір математичних функцій VBA досить стандартний, назвемо їх з короткими поясненнями:

- *`Abs(число)`* — абсолютне значення числа.
- *`Atn(число)`* — арктангенс (у радіанах) аргумента, що задає тангенс кута.
- *`Cos(число)`* — косинус кута, аргумент задає кут у радіанах.
- *`Exp(число)`* — експонента, тобто результат зведення числа `e` (підстава натуральних логарифмів) у зазначений степінь.
- *`Log(число)`* — натуральний логарифм числа.
- *`Rnd[(число)]`* — результат є рівномірно розподілене випадкове число в інтервалі `[0 — 1]`. Якщо аргумент `число` не заданий або більше нуля, то породжується чергове випадкове число, якщо він дорівнює `0`, то результатом буде

попереднє випадкове число, а якщо `число` менше нуля, то всякий час породжується одне і те саме число, обумовлене аргументом. Перед тим як одержати послідовність випадкових чисел, необхідно викликати функцію `Randomize` для ініціалізації послідовності. Помітимо, для формування значення випадкових чисел використовується таймер. Щоб одержати цілочисельну послідовність рівномірно розподілених випадкових чисел в інтервалі `[Min — Max]`, варто використати наступне перетворення `Int((Max — Min +1)*Rnd)+ Min`

- *`Sgn(число)`* — знак числа (якщо `число` більше нуля — `1`, дорівнює нулю — `0`, менше нуля — `-1`).
- *`Sin(число)`* — синус кута, аргумент `число` задає кут у радіанах.
- *`Sqr(число)`* — квадратний корінь.
- *`Tan(число)`* — тангенс кута, аргумент `число` задає кут у радіанах.

У всіх цих описах під аргументом функції *`число`* розуміється числовий вираз.

У довідковій системі або в будь-якому математичному довіднику ви зможете знайти вказівки, як на основі цього базису можна обрахувати велику кількість інших математичних функцій. Наприклад, обрахування гіперболічного синуса виконується за формулою:

```
HSin(X) = (Exp(X) — Exp(-X))/2
```

а обрахування арккотангенса:

```
Arccotan(X) = Atn(X) + 2*Atn(1)
```

Робота з рядками

Працювати з текстами програмістові, як правило, доводиться значно частіше, ніж працювати із числами. Тому варто добре уявляти основні базисні операції над рядками, які в більшості випадків реалізуються за допомогою вбудованих функцій. Помітимо, що в VBA Office 2000 додані нові операції над рядками, що розширюють можливості ефективної роботи з перетворення тексту.

Порівняння рядків

Звичайні операції порівняння застосовні й до строкових даних. Інтерпретація цих операцій залежить від установки опції *Option Compare*.

- Якщо ця опція встановлена як *Text*, то порівняння на "більше - менше" представляє лексикографічне порівняння, коли рядки порівнюються за їхнім розташуванням в словнику. Помітьте, це порівняння не чутливе до регістра, так що великі й малі букви не розрізняються. Програмісти, звичайно, розуміють, що порівняння рядків означає порівняння кодів їхніх символів, тому лексикографічний порядок визначається кодуванням символів алфавіту.
- Якщо ця опція встановлена як *Binary*, то порівняння йде по бітах. У цьому випадку порівняння природно, чутливе до регістра.
- При роботі з рядками в Access за замовчуванням застосовується сортування, задане на рядках бази даних Access. Помітьте, при створенні модуля в Access за замовчуванням вставляється опція `Option Compare Database`. Звичайно, ця опція застосовна тільки при роботі в Access.

Якщо потрібно локально перевизначити вид порівняння, заданий опцією для всього модуля, то можна використати вбудовану функцію *StrComp*, що повертає результат порівняння рядків. Її синтаксис:

```
StrComp(string1, string2[, compare])
```

Аргументи `string1` і `string2` — порівнювані рядки. Необов'язковий аргумент `compare` вказує спосіб порівняння рядків: значення за замовчуванням `0` використовується, щоб виконати двійкове порівняння, `1` задає посимвольне порівняння без врахування регістра.

Якщо `string1` менше ніж `string2`, то результат дорівнює `-1`, якщо рядки рівні, то — `0`, якщо друга менше, то — `1`, якщо хоч один з рядків має значення `Null`, то результат також дорівнює `Null`.

Потужним і досить корисним засобом при роботі з текстами є операція `Like`, що задає порівняння за зразком. Необхідність знаходження в наборі всіх рядків, що задовольняють деякому шаблону (зразку), виникає в найрізноманітніших завданнях. VBA

дозволяє вирішувати її в одну операцію. Наведемо таблицю спеціальних символів, припустимих при завданні зразка.

Таблиця 5.5

Спеціальні символи, використовувані при завданні шаблона

Символи	**Інтерпретація**	**Приклади**
*	Будь-який текст — довільне число символів	Шаблону Agent* відповідають всі тексти, що починаються зі слова Agent.
?	Один будь-який символ	Шаблону Ко? задовольняють, зокрема, рядки Кок і Кук.
#	Будь-яка цифра від 0 до 9	Шаблону Agent### відповідає 1000 різних рядків, серед яких і Agent007.
[безліч_символів]	Будь-який символ, що належить безлічі	Задати безліч можна за допомогою перерахування й інтервалів. Шаблону Ко[а — у] задовольняють слова «Як», «Кок», «Кук». Чутливість до регістра залежить від установки опції Option Compare.
[!безліч_символів]	Будь-який не приналежній безлічі символ	Шаблону [!а — я] задовольняє символ, що не є буквою російського алфавіту.

Наведемо приклад роботи з операцією Like:

```
Public Sub LikeOperation()
Const pat1 = «[A-Z]»
Const pat2 = «[a-z]»
Const pat3 = «[!a-z]»
Const pat4 = «[3-5]»
Dim res As Byte
```

```
Dim Sym As String

res = «Кук» Like «ДО[аоу]до»
Debug.Print res

res = «f» Like pat1
Debug.Print res
res = «f» Like pat2
Debug.Print res
res = «f» Like pat3
Debug.Print res

res = «5» Like pat4
Debug.Print res
Sym = "3"
res = Sym Like pat1 & pat4
Debug.Print res
res = Sym Like pat1 Or Sym Like pat4
Debug.Print res

End Sub
```

Ось результати друку:

```
255
 0
 255
 0
 255
 0
 255
```

Зверніть увагу на останні два результати, що демонструють некоректний і коректний способи роботи з об'єднанням множини символів, що перевіряють.

Основні операції над рядками

У класичній математиці давно визначений набір основних операцій над числовими й булевими даними. Строковою арифметикою серйозно стали займатися з появою комп'ютерів. Твердий стандарт на ці операції ще не склався. Як правило, у всіх мовах є тільки одна операція — конкатенація рядків, позначувана

символом "&". Всі інші основні операції реалізуються за допомогою вбудованих функцій, імена яких і їхніх аргументів можуть варіюватися від мови до мови. Більше того, варіюється й сам набір цих операцій. Мінімально, крім конкатенації необхідні ще дві операції, перша з яких дозволяє виявити індекс входження одного рядка в інший, друга — виділити з рядка його підрядок. VBA має досить потужний набір операцій. Розглянемо спочатку ті операції, які й раніше існували в мові, а вже потім більш докладно поговоримо про нові можливості.

Функція ***Len(string)*** повертає довжину рядка, заданого аргументом `string`. (число символів рядка).

Функція ***InStr*** визначає позицію (індекс) першого входження одного рядка у середину іншого рядка. Синтаксис

InStr([start,]string1, string2[, compare])

Необов'язковий аргумент `start` задає позицію, з якої починається пошук (за замовчуванням — з першого символу рядка). Аргумент `string1` задає рядок, у якому виконується пошук, а `string2` — підрядок, входження якого шукається. Необов'язковий аргумент `compare` має той же зміст, що й для функції `StrComp`. Значення, що повертаються, визначені у таблиці.

Таблиця 5.6

Результат роботи функції `InStr`

Умови	**Значення функції `InStr`**
`string1` — порожній рядок	`0`
`string1` або `string2` рівні `Null`	Порожнє значення — `Null`
`string2` — порожній рядок	`start`
Входження `string2` не знайдено в `string1`	`0`
Входження `string2` знайдено в `string1`	Позиція виявленого підрядка
`start >Len(string2)`	`0`

Наприклад, два виклики цієї функції повернуть такі результати:

```
? InStr(4, "XXпXXпXXпXXп", "П", 1)
 6
? InStr(4, "XXпXXпXXпXXп", "П", 0)
 9
```

Функція ***Left(string, length)*** виділяє в рядку `string` зазначене число `length` символів ліворуч, дозволяючи виділити префікс рядка.

Функція ***Right(string, length)*** виконує аналогічну операцію, виділяючи символи праворуч, що дозволяє одержати суфікс (закінчення) рядка.

Більш універсальна функція ***Mid(string, start[, length])*** дозволяє виділити з рядка *string* підрядок довжини *length*, починаючи з позиції `start`.

Ось приклад виклику цих функцій безпосередньо з вікна налагодження:

```
? VBA.Left("рококо",3)
доля
? VBA.Right("рококо",3)
око
? VBA.Mid("рококо",3,3)
кок
```

Функції ***LTrim(string)***, ***RTrim(string)***, ***Trim(string)*** повертають копію рядка, з якого вилучені пробіли, що перебували на початку рядка `(LTrim)`, наприкінці рядка `(RTrim)` або на початку й кінці рядка `(Trim)`.

Функция ***String*** створює рядок, який містить задане число повторюваних символів.

Синтаксис:

```
String(number, character)
```

Аргумент `number` задає довжину рядка, а `character` — код символу або строковий вираз, перший символ якого використовується при створенні результуючого рядка.

Функції ***LCase(string)*** і ***UCase(string)*** повертають копію рядка, символи якого наведені до нижнього *(Low)* або верхнього регістра *(Upper)*.

Функції, що повертають рядки, існують у двох варіантах, які відрізняються іменами. Функції, імена яких ми приводили, повертають результат типу `Variant`. У другому варіанті імена функцій закінчуються знаком `$`, наприклад, `Mid$`, `UCase$`. У цьому випадку результат повертається типу `String`. Такі функції виконуються швидше, але не можуть коректно працювати з рядками, що мають значення `Null`.

Наведемо як приклад корисну функцію, що знаходить шлях до активного документа Word і робить розбір всіх компонентів цього шляху:

```
Public Function AppPath(Disk As String, Dir As String,
  FileName As String) As String
'Ця Функція повертає як результат повний шлях активного
  документа
'Її параметри містять компоненти цього шляху — ім'я
  диска, каталог на диску й ім'я файлу
  Dim MyDoc As Document
  Dim Path As String
  Dim Start As Byte, Finish As Byte

  'Визначаємо повний шлях до файлу, що задає активний
  документ Word
  Set MyDoc = ActiveDocument
  Path = MyDoc.FullName

  'Виділяємо ім'я диска — перший символ повного шляху
  Disk = VBA.Left(Path, 1)

  'Виділяємо каталог, у якому зберігається документ
  Start = VBA.InStr(1, Path, "\")
  Finish = VBA.InStrRev(Path, "\")
  Dir = VBA.Mid(Path, Start + 1, Finish — Start)

  'Виділяємо ім'я файлу
  FileName = VBA.Mid(Path, Finish + 1)

'Вертається результат — повний шлях до каталогу
  AppPath = VBA.Left(Path, Finish)
End Function

Public Sub MyPath()
Dim Path As String
Dim Dir As String
Dim Disk As String
Dim FileName As String
Path = AppPath(Disk, Dir, FileName)
Debug.Print Disk, Dir, FileName, Path

End Sub
```

Ось результати друку після запуску процедури MyPath:

```
E O2000\VBA2000\Ch8\     Ch8.doc    E:\O2000\VBA2000\Ch8\
```

Зверніть увагу, у функції AppPath ми використали нову функцію InStrRev, що полегшило рішення нашого завдання. До опису нових уведених функцій ми й переходимо.

Нові функції для роботи з рядками

В VB додані корисні функції для роботи з рядками.

Функція InStrRev — **пошук останнього входження підрядка**

Функція InStrRev симетрично доповнює функцію InStr, аналогічно тому, як функція Right доповнює функцію Left. Ця функція шукає входження підрядка в рядок, але починає свою роботу із правого кінця рядка. Її використання може істотно прискорити роботу, якщо заздалегідь відомо, що шуканий підрядок перебуває десь наприкінці рядка — джерела. Якщо входження шуканого підрядка єдино, то обидві функції дають той самий результат. При множинному входженні функція InStr повертає перше входження, у той час як InStrRev — останнє. Її синтаксис:

InstrRev(stringcheck, stringmatch[, start[, compare]])

Її параметри мають той же зміст, що й у функції InStr, але, помітьте, порядок їхнього завдання змінений. Необов'язковий параметр Start тепер задається третім за рахунком. Коли він опущений, то за замовчуванням його значення дорівнює "-1", і пошук починається з останнього символу. Якщо повернутися до останнього прикладу, то для рішення завдання було потрібно визначити перше й останнє входження символу "\" у рядок, що задає шлях до файлу. Тому обидві функції були досить доречні. Потрібно відзначити, що це типова ситуація при розборах тексту.

Функція Replace — **заміна всіх входжень підрядка.**

Це дивно, що в наборі вбудованих функцій не було дотепер функції Replace. Заміна одного підрядка на інший — це одна з основних операцій при роботі з рядками. Тому в арсеналі практично кожного із програмістів була своя версія функції Replace. Тепер можна користуватися стандартною реалізацією. Її безперечним достоїнством є те, що вона дозволяє замінити не тільки перше входження шуканого підрядка, але й всі інші входження, не вимагаючи організації циклу. З іншого боку, можна

обмежитися тільки заміною першого або декількох перших входжень. Розглянемо синтаксис цієї функції:

Replace(expression, find, replace[, start[, count[, compare]]])

Перший аргумент `expression` задає строковий вираз, результат якого визначає рядок — джерело, у якому здійснюється заміна. Аргументи `find` і `Replace` задають замінну підрядка і його нове значення. Аргумент `Count` визначає число замін. Звичайно він дорівнює `1`, коли мова йде про заміну першого входження, або опускається — у цьому випадку його значення за замовчуванням дорівнює `"-1"`, що означає заміну всіх входжень. Аргумент `compare` має звичайний сенс.

Як приклад наведемо функцію `Rep`.

```
Public Sub Rep()
'Ця процедура перетворить виділений програмний текст
'Заміняючи пробіли табуляцією й кінець абзацу м'яким
  кінцем рядка
  Dim TxtRange As String

  TxtRange = Selection.Range.Text
  'Заміна пробілів: 4-х, 3-х і 2-х символом табуляції
  TxtRange = Replace(TxtRange, "    ", vbTab)
  TxtRange = Replace(TxtRange, "   ", vbTab)
  TxtRange = Replace(TxtRange, "  ", vbTab)

  'Заміна кінців абзацу
  Selection.Range.Text = Replace(TxtRange, VBA.Chr(13),
  VBA.Chr(11))

End Sub
```

Незважаючи на всю корисність функції `Replace`, нам довелося написати ще одну власну модифікацію цієї функції. Справа в тому, що в `Replace` є одна особливість, на яку варто звернути увагу, — результат, що повертається нею, починається не з першої позиції, а з позиції, `заданої` аргументом `Start`. Так що крім своєї основної ролі вона ще обрубує голову рядка, якщо тільки `Start`, відрізняється від `1`. Оскільки заміну найчастіше потрібно робити не із самого початку, а одержувати хочеться повний рядок, то ми написали свій варіант цієї функції. Наведемо його текст:

```
Public Function MyReplace(ByVal Expr As String, ByVal
  find As String, _
      ByVal rep As String, Optional ByVal start As Long
  = 1, Optional
      ByVal count As Long = -1, _
      Optional ByVal compare As VbCompareMethod =
  vbBinaryCompare) As String

  'Виклик стандартної функції Replace
  If start = 1 Then
      MyReplace = replace(Expr, find, rep, start,
  count, compare)
  Else
      MyReplace = VBA.Left(Expr, start — 1) &
  replace(Expr, find,
      rep, start, count, compare)
  End If

End Function
```

Ось результати декількох викликів `Replace i MyReplace` у вікні налагодження:

```
? Replace("A+B *(D*B +B)","B","C",4,1)
 *(D*C +B)
? MyReplace("A+B *(D*B +B)","B","C",4,1)
A+B *(D*C +B)
? MyReplace("A+B *(D*B +B)","B","C",4)
A+B *(D*C +C)
```

Видалення підрядка**.** Видалення входжень підрядка в рядок також є однією з основних операцій над рядками. Хоча спеціальної убудованої функції `DelStr` немає, але написати її реалізацію, маючи `Replace`, зовсім просто. Видалення підрядка еквівалентне заміні її порожнім рядком. Ось текст, написаної нами функції `DelStr`:

```
Public Function DelStr(ByVal Expr As String, ByVal find
  As String, _
      Optional ByVal start As Long = 1, Optional ByVal
  count As Long = -1, _
```

```
        Optional ByVal compare As VbCompareMethod =
    vbBinaryCompare)

    'Виклик функції MyReplace з порожнім рядком для
    заміни
    DelStr = MyReplace(Expr, find, "", start, count,
    compare)
End Function
```

Результати її викликів у вікні налагодження:

```
? DelStr("T* + U** — V*","*")
T + U — V
? DelStr("T* + U** — V*","*",5)
T* + U — V
? DelStr("T* + U** — V*","*",5,1)
T* + U* — V*
```

Помітьте, при визначенні `DelStr` ми користувалися власною модифікацією `MyReplace`, оскільки, як нам здається, вона більшою мірою відповідає суті справи.

Розбір рядка. Функції Split, Join і Filter

Важливу групу нових функцій становлять функції, призначені для розбору тексту. Їхня поява дозволяє ефективно вирішувати цілий клас завдань, що часто зустрічаються при роботі з текстами. Суть справи така: текст, заданий рядком, найчастіше представляє сукупність структурованих елементів — абзаців, пропозицій, слів, дужкових виразів і так далі. При роботі з таким текстом необхідно розділити його на елементи, користуючись тим, що є спеціальні роздільники елементів, — це можуть бути пробіли, дужки, розділові знаки. Подібне завдання виникає практично постійно при роботі зі структурованими текстами. Нові функції істотно полегшують вирішення таких завдань. Функція `Split` дозволяє розділити рядок на елементи й створити масив із цих елементів. Функція `Filter` дозволяє виділити потрібні елементи в цьому масиві, а функція `Join` вирішує зворотне завдання, перетворюючі масив у рядок. Розглянемо ці функції докладніше.

Перетворення рядка в масив

Функція Split. Передбачається, що вихідний рядок складається з елементів (підрядків), розділених спеціальними символами — роздільниками. Функція `Split` повертає одномірний масив з елементів рядка. Її синтаксис:

```
Split(expression[, delimiter[, limit[, compare]]]),
```

де:

- `expression` — рядковий вираз, результат якого задає рядок — джерело, що складається з елементів і роздільників.
- `delimeter` — рядок, що задає послідовність символів, використовуваних як роздільник. Якщо цей параметр опущений, то за замовчуванням передбачається, що в ролі роздільника виступає пробіл.
- `limit` — необов'язковий параметр, що дозволяє обмежити число поворотних елементів. За замовчуванням його значення дорівнює `-1`, що означає виділення всіх елементів.
- `compare` — необов'язковий параметр, що має стандартний зміст у всіх операціях над рядками. За замовчуванням передбачається двійкове побітове порівняння.

Наведемо приклад, у якому складнопідрядна пропозиція розділяється на прості пропозиції:

```
Public Sub SplitString()
  'У цій процедурі складна пропозиція розділяється на
  прості
  'Повідомляємо динамічний масив
  Dim Simple() As String, i As Byte

  'Розмірність масиву Simple установлюється автоматично
  'відповідно до розмірності масиву, що повертає
  функцією Split
  Simple = Split("А це пшениця, що у темному прикомірку
  зберігається в будинку, " _
      & "який побудував Джек", ", ")
  For i = LBound(Simple) To UBound(Simple)
      Debug.Print Simple(i)
  Next i

End Sub
```

У результаті роботи цієї процедури створюється масив із трьох елементів:

```
А це пшениця
яка в темному прикомірку зберігається в будинку
який побудував Джек
```

Зверніть увагу, для використання результатів роботи функції `Split` створюється динамічний масив рядків. Про його розмірність немає потреби піклуватися, оскільки в момент присвоєння результату виконання `Split`, він заповнюється елементами й у нього з'являється нижня й верхня границя. Далі із цим масивом можна працювати звичайним чином.

У функції `Split`, на наш погляд, є один недолік. Хоча в ролі роздільника може виступати послідовність символів (у нашому прикладі це пари символів "кома, пробіл"), але не можна використати різні роздільники елементів. У завданнях подібного роду типовою є ситуація, коли елементи відділяються різними роздільниками, можливо, оточеними пробілами. Пізніше ми покажемо, як можна впоратися із цим завданням.

Збирання елементів масиву в рядок

Функція Join. Функція `Join` вирішує зворотне завдання — вона відновлює рядок за його елементами, що зберігаються в масиві, додаючи роздільники в момент їхнього об'єднання. Її синтаксис:

```
Join(sourcearray[, delimiter]).
```

Параметри зрозумілі без особливих пояснень. Помітимо, що якщо необов'язковий аргумент `delimiter` відсутній, то елементи розділяються пробілами. Як приклад наведемо зворотне складання складної пропозиції:

```
Public Sub SplitAndJoin()
'У цій процедурі складна пропозиція розділяється на
  прості
'А потім після обробки рядок відновлюється
  'Повідомляємо динамічний масив
  Dim Simple() As String, i As Byte
  Dim Simple1() As String, Res As String
  'Розмірність масиву Simple установлюється автоматично
```

```
  'відповідно до розмірності масиву, що повертає
  функцією Split
  Simple = Split("А це пшениця, що у темному прикомірку
  зберігається в будинку, " _
      & "який побудував Джек", ", ")
  'Створюємо новий масив
  ReDim Simple1(1 To UBound(Simple) + 2)
  Simple1(1) = "А це веселий птах — синиця"
  Simple1(2) = "яка часто краде пшеницю"
  For i = 3 To UBound(Simple1)
      Simple1(i) = Simple(i — 2)
  Next i
  'Створюємо рядок з масиву Simple1

  Res = Join(Simple1, ", ")
  Debug.Print Res
End Sub
```

Ось результат друку:

```
А це веселий птах — синиця, що часто краде пшеницю, що
  у темному
прикомірку зберігається в будинку, що побудував Джек
```

Фільтрація елементів масиву

Функція Filter. Функція `Filter` є одним з варіантів пошуку за зразком серед елементів масиву. Тут не потрібно точного збігу елемента й зразка, досить, щоб рядок - зразок утримувався в рядку, заданому елементом масиву. Оскільки збігів може бути досить багато, то результатом є масив відфільтрованих елементів. Деякі деталі будуть пояснені при описі аргументів функції `Filter`. Ось її синтаксис:

`Filter(sourcearray, match[, include[, compare]])`, де:

- `sourcearray` — одномірний масив, елементи якого є рядками. Він може бути отриманий, наприклад, як результат розщеплення рядка в масив.
- `match` — зразок пошуку (рядок, входження якого шукається в кожному елементі вихідного масиву).

- include — необов'язковий аргумент булевого типу. За замовчуванням має значення True, й означає, що елементи, які задовольняють зразок, входять у результуючий масив. Якщо задано значення False, то результуючий масив складається з елементів, що не задовольняють зразок.
- compare — має звичайний сенс.
  Як результат повертається масив відфільтрованих елементів.

Наш приклад є розширеним варіантом процедури SplitAndJoin:

```
Public Sub SplitAndJoin()
'У цій процедурі складна пропозиція розділяється на
  прості
'А потім після обробки рядок відновлюється
'Тут же демонструється фільтрація елементів масиву
  'Повідомляємо динамічний масив
  Dim Simple() As String, i As Byte
  Dim Simple1() As String, Res As String
  Dim Simple2() As String
  'Розмірність масиву Simple установлюється автоматично
  'відповідно до розмірності масиву, що повертає
  функцією Split
  Simple = Split("А це пшениця, що у темному прикомірку
  зберігається в будинку, " _
      & "який побудував Джек", ", ")
  'Створюємо новий масив
  ReDim Simple1(1 To UBound(Simple) + 2)
  Simple1(1) = "А це веселий птах — синиця"
  Simple1(2) = "яка часто краде пшеницю"
  For i = 3 To UBound(Simple1)
      Simple1(i) = Simple(i — 2)
  Next i
  'Створюємо рядок з масиву Simple1

  Res = Join(Simple1, ", ")
  Debug.Print Res
  'Фільтрація елементів масиву
  Simple2 = Filter(Simple1, "котор")
  Res = Join(Simple2, ", ")
  Debug.Print Res

  Simple2 = Filter(Simple1, "котор", False)
```

```
  Res = Join(Simple2, ", ")
  Debug.Print Res
End Sub
```

Ось результат друку:

```
А це веселий птах — синиця, що часто краде пшеницю, що
  у темному
прикомірку зберігається в будинку, що побудував Джек
яка часто краде пшеницю, що у темному прикомірку
  зберігається в будинку,
який побудував Джек
А це веселий птах – синиця
```

Робота з датами й часом

Для того щоб забезпечити програмістові можливість коректно працювати з датами й часом, VB надає спеціальний тип даних `Date`, який зберігає дату й час. Над даними цього типу можна виконувати деякі операції, але, звичайно ж, при роботі з ними найчастіше використовуються спеціальні вбудовані функції. Спробуємо коротко розглянути основні можливості роботи з датами. Насамперед помітимо, що можливий діапазон дат охоплює дати від `1.1. 100 року` до `1-го січня 9999 року`. Якщо говорити про внутрішнє подання дат, що займають 4 байти пам'яті, то ціла частина зберігає число днів від деякої початкової дати, дробова частина зберігає час від півночі. Початковою датою є `30-е грудня 1899 року`. Завдяки такому внутрішньому поданню додавання й вирахування цілого числа сприймається як додаток або вирахування днів.

Присвоювання значень

При присвоюванні значень змінним типу `Date` варто поміщати дату в спеціальні обмежники "#" або задавати її як рядкову константу. При завданні дати в обмежниках, наприклад, `#9, May, 99 #` вона автоматично перетвориться у стандартний формат

#5/9/99# (місяць/день/рік). От приклад деяких дій над датами:

```
Public Sub WorkWithDates()

   'Робота з датами
   Dim dat1 As Date, dat2 As Date, dat3 As Date

   'Присвоювання дат
   dat1 = 12
   dat2 = 9/5 / 99
   dat3 = #9/5/1999#
   Debug.Print dat1, dat2, dat3
   dat1 = "15/7/99"
   dat2 = #5/9/1999#
   dat3 = dat3 + 100
   Debug.Print dat1, dat2, dat3
   If dat3 > dat2 Then
       Debug.Print dat3 — dat2
   Else
       Debug.Print dat2 — dat3
   End If

End Sub
```

Результати виконання цієї програми:

```
11.01.1900    0:26:11             05.09.99
15.07.99            09.05.99            14.12.99
 219
```

Прокоментуємо результати, оскільки деякі з них можуть викликати здивування. Зупинимося на перших двох присвоюваннях. Щоб зрозуміти їх, потрібно згадати внутрішнє подання дат. У першому випадку до початкової дати додається 12 днів, звідси й виходить 11 січня 1900 року. У другому випадку, оскільки вираз у правій частині не укладено в обмежники, він обраховується як звичайний арифметичний вираз, отримане дробове число сприймається як час від початку доби, — воно й друкується. Зверніть увагу на третє присвоювання: перше число сприймається як номер місяця, тому його краще задавати, використовуючи назви місяців, які автоматично будуть перетворені у відповідний формат.

Далі в програмі продемонстровані деякі можливості виконання операцій над датами — додаток цілого числа днів, порівняння дат, вирахування дат. Помітьте, вирахування дат дає різницю між ними, виражену в днях.

Убудовані функції для роботи з датами

Основна робота з датами виконується з використанням убудованих функцій. Розглянемо коротко їхній опис і призначення.

Визначення поточної дати або часу

- `Date` — повертає поточну дату.
- `Time` — повертає поточний час за годинником комп'ютера.
- `Now` — повертає значення типу `Variant (Date)`, що містить поточну дату й час за системним календарем й годинником комп'ютера.

Обрахування над датами

Функція **`DateAdd`** призначена для додавання або вирахування зазначеного тимчасового інтервалу зі значення дати, за її допомогою можна задавати часовий інтервал не тільки в днях, але й у місяцях, роках і так далі.

DateAdd(interval, number, date),

де

`interval` — рядок, що вказує тип тимчасового інтервалу, що додає, `number` — число тимчасових інтервалів, на яке варто змінити дату, `date` — дата, до якої додається зазначений часовий інтервал. Припустимі значення аргументу `interval` наведені в таблиці.

Таблиця 5.7

Можливі тимчасові інтервали

Значення	**Опис**
уууу	Рік
Q	Квартал
m	Місяць
Y	День року
D	День місяця
w	День тижня
ww	Тиждень

Продовження табл. 5.7

Значення	Опис
H	Години
N	Хвилини
S	Секунди

Для прикладу, наведемо два виклики цієї функції у вікні налагодження:

```
? DateAdd("m", 1, "31-янв-95")
28.02.95
? DateAdd("m", -1, "31-янв-95")
31.12.94
```

Функція **DateDiff** призначена для визначення часу, що пройшов між двома датами. Наприклад, за допомогою цієї функції можна обрахувати число днів між двома датами або число тижнів між поточною датою й кінцем року. Синтаксис:

```
DateDiff(interval, date1, date2[, firstdayofweek[,
  firstweekofyear]])
```

де :

- `interval` задає тип тимчасового інтервалу при обрахуванні різниці між датами `date1 i date2` (його можливі значення ті ж, що й для функції `DateAdd`)
- `date1 i date2` — дві дати, різницю між якими варто обрахувати.
- `firstdayofweek` — константа, що вказує перший день тижня (за замовчуванням уважається, що тиждень починається з неділі).
- `firstweekofyear` — константа, що вказує перший тиждень року (за замовчуванням першим тижнем уважається тиждень, що містить 1 січня).

Приклад виклику цієї функції у вікні налагодження:

```
? DateDiff("m", "18.10.55", "31-янв-95")
 471
? DateDiff("Y", "18.11.97", "01.01.97")
-321
```

Функція **DatePart** призначена для визначення зазначеного компонента дати. Наприклад, за допомогою цієї функції можна визначити день тижня або поточну годину. Синтаксис:

```
DatePart(interval, date[,firstdayofweek[,
  firstweekofyear]])
```

де :

- `interval` задає тип тимчасового інтервалу, що повертає
- `date` — дата, що підлягає обробці.
- необов'язкові аргументи `Firstdayofweek` і `Firstweekofyear` мають той же зміст, що й для функції `DateDiff`.

Приклади виклику:

```
? DatePart("ww", "02.09.89")
 35
? DatePart("w", "02.09.89")
 7
```

Функція **DateSerial** дозволяє обрахувати значення дати типу `Variant (Date)` по її компонентах, — року, місяця й дня. Її синтаксис:

```
DateSerial(year, month, day)
```

Значення кожного аргументу має лежати у відповідному діапазоні: `100 - 9999` для року, `1 — 31` для днів і `1 — 12` для місяців. Можна також використати для аргументів числові вирази для опису відносної дати.

Приклади:

```
? DateSerial(1997 — 25, 10 — 2, 18 — 1)
17.08.72
? DateSerial(Year(Now) — 3,Month(Now) — 2, Day(Now) —
  1)
08.03.96
```

Функція **DateValue** переводить аргумент-рядок у дату. На відміну від функції перетворення `CDate`, функція `DateValue` правильно обробляє припустимі дати, що містять повні або короткі назви місяців.

Її синтаксис:

DateValue(date)

Аргумент date може задавати як дату, так і час. Можливі різні формати завдання дати, у тому числі й з назвами місяців.

Приклади:

```
? DateValue ("25.04.1997")
25.04.97
? DateValue ("04.25.1997")
25.04.97
? DateValue ("25 квітня 1997")
25.04.97
? DateValue ("25 — квіт.-97")
25.04.97
? DateValue ("Квітень, 25, 97")
25.04.97
? DateValue ("25/04/1997")
25.04.97
? DateValue ("25.04")
25.04.99
```

Остання рівність пов'язана з тим: що за відсутності року функція DateValue використовує поточний рік за системним календарем комп'ютера.

Функції **Day(date), Month(date), і Year(date)** є, у деякому змісті, зворотними до двох попередніх. Вони за аргументом-датою визначають номер дня, місяця й року. Функція Weekday(date, [firstdayofweek]) повертає значення типу Variant (Integer), що містить ціле число, яке представляє день тижня. Другий аргумент задає перший день тижня (за замовчуванням — неділя). Значення 2 відповідає понеділку.

Функції **Hour(час), Minute(час) і Second(час)** за аргументом, що представляє час та є числовим або рядковим виразом, повертає ціле число, що містить, відповідно, години, хвилини або секунди в значенні часу.

Функція **TimeSerial(hour, minute, second)** обчислює результат Variant (Date), що містить значення часу, яке відповідає зазначеним годині, хвилині й секунді.

Функція **TimeValue(час)** повертає значення типу Variant (Date), що містить час. Аргумент час звичайно задається рядковим виразом, що представляє час від 0:00:00 (12:00:00 A.M.) до 23:59:59 (11:59:59 P.M.) включно.

Задавати його потрібно коректно, щоб не виникало помилок, як в останніх двох прикладах:

```
? Year(Now)
 1999
? Month(now)
 5
? Day(Now)
 9
? Hour(Now)
 11
? Minute (Now)
 57
? Second(Now)
 28
? TimeSerial(Hour(Now), Minute(now), Second(Now))
11:59:00
? TimeValue("12:30 PM")
12:30:00
? TimeValue("12.30")
0:00:00
? TimeValue("12,5")
0:00:00
```

Функція Timer і хронометраж обрахувань

Функція `Timer` повертає значення типу `Single`, що представляє число секунд, що пройшли після півночі. Ця функція широко використовується при проведенні хронометражу обрахувань у програмах. Щоб підвищити ефективність виконання VB програм найчастіше доводиться вдаватися до проведення хронометражу, щоб виявити найбільш критичні за часом виконання ділянки програм, а потім уже застосовувати спеціальні заходи для прискорення обрахувань. Наведемо приклад, у якому аналізується час, затрачуваний на виконання операцій над арифметичними даними різних підтипів.

```
Public Sub Speed()
  'Ця програма виконує виміри часу обрахувань над
  даними різного типу

  Dim Start As Single, Finish As Single
  Dim i As Long, j As Long
  Dim bx As Byte, by As Byte, bz As Byte
  Dim ix As Integer, iy As Integer, iz As Integer
  Dim lx As Long, ly As Long, lz As Long
  Dim sx As Single, sy As Single, sz As Single
  Dim dx As Double, dy As Double, dz As Double

  For i = 1 To 5 'Зовнішній цикл для повторення вимірів
  часу
      Start = Timer
      For j = 1 To 100000
            bx = 99: by = 101
            bz = by * (by — bx) \ by
      Next j
      Finish = Timer
      Debug.Print "Тип Byte: Час ", i, " = ", Finish —
  Start
      Debug.Print "bz = ", bz

      Start = Timer
      For j = 1 To 100000
            ix = 99: iy = 101
            iz = iy * (iy — ix) \ iy
      Next j
      Finish = Timer
      Debug.Print "Тип Integer: Час ", i, " = ", Finish
  — Start
      Debug.Print "iz = ", iz

      Start = Timer
      For j = 1 To 100000
            lx = 99: ly = 101
            lz = ly * (ly — lx) \ ly
      Next j
      Finish = Timer
```

```
        Debug.Print "Тип Long: Час ", i, " = ", Finish —
    Start
        Debug.Print "lz = ", lz

        Start = Timer
        For j = 1 To 100000
               sx = 99: sy = 101
               sz = sy * (sy — sx) / sy
        Next j
        Finish = Timer
        Debug.Print "Тип Single: Час ", i, " = ", Finish
    — Start
        Debug.Print "sz = ", sz

        Start = Timer
        For j = 1 To 100000
               dx = 99: dy = 101
               dz = dy * (dy — dx) / dy
        Next j
        Finish = Timer
        Debug.Print "Тип Double: Час ", i, " = ", Finish
    — Start
        Debug.Print "dz = ", dz

     Next i

End Sub
```

Деякі інші вбудовані функції

Як ми вже говорили, вбудованих функцій безліч. Пам'ятати їх усі не потрібно, оскільки під рукою завжди є довідкова система. Для загального знайомства розглянемо деякі групи функцій.

Функції перевірки типів даних

До цієї групи належить функція TypeName, яка за іменем змінної повертає значення типу String, що представляє її тип.

Виклик має вид:

TypeName(им'язмінної),

де параметр им'язмінної представляє вираз типу Variant, що визначає будь-яку змінну, за винятком змінної з типом, обумовленим користувачем.

Рядок, що повертає функція TypeName, може містити кожний з таких типів: Byte, Integer, Long, Single, Double, Currency, Decimal, Date, String, Boolean, Error, Empty (якщо змінна не ініціалізована), Null, Object, Unknown (тип невідомий), Nothing (об'єктна змінна, що не утримує посилання на об'єкт).

Для перевірки типу є також набір функцій з булевими значеннями. Вони застосовуються до змінних або виразів і визначають, чи мають ті заданий тип або значення. У таблиці наведені функції цієї групи й зазначені обумовлені ними типи.

Таблиця 5.8

Функції перевірки типів

Функція	Що перевіряє
IsArray(змінна)	Чи є змінна масивом.
IsDate(вираз)	Чи може значення виразу бути перетворене в значення дати.
IsEmpty(змінна)	Чи була ініціалізована змінна.
IsError(вираз)	Чи представляє вираз значення помилки.
IsNull(вираз)	Чи є результатом виразу порожнє значення (Null).
IsNumeric(вираз)	Чи має вираз числове значення (для виразів типу Date повертає False).
IsObject(змінна)	Чи представляє змінна об'єктну змінну. Повертає True, навіть для змінної зі значенням Nothing.

Перетворення типів даних

Для приведення даних до потрібного типу в VB включений великий набір функцій: `CBool, CByte, CCur, CDate, CDbl, CDec, CInt, CLng, CSng, CStr, CVar, CVErr, Fix, Int.`

Всі вони мають такий синтаксис:

`Им'яфункції(вираз)`

Обов'язковий аргумент `вираз` є будь-яким рядковим виразом або числовим. У таблиці зазначені типи значень, що повертаються, та їхні діапазони.

Таблиця 5.9

Функції приведення до типу

Функція	**Тип**	**Діапазон значень аргумента**
`CBool`	`Boolean`	Будь-який припустимий рядок або числовий вираз. Наприклад, `Cbool(0) = Cbool("00")= False, Cbool(1) =Cbool("-6")=True.`
`CByte`	`Byte`	Від 0 до 255.
`CCur`	`Currency`	Від -922 337 203 685 477,5808 до 922 337 203 685 477,5807. Наприклад, `CCur("25,878755")= 25.8788`, а `CCur("25.878755")` — помилковий виклик, тому що в рядку відокремлювати дробову частину має кома.
`CDate`	`Date`	Будь-який припустимий вираз дати. Наприклад, `Cdate(32444.5) =28.10.88 12:00:00, Cdate("17/08/1972")=17.08.72, Cdate("6:45:28 PM")= 18:45:28.`
`CDbl`	`Double`	-4,94065645841247E-324 для від'ємних чисел; від 4,94065645841247E-324 до 1,79769313486232E308 для позитивних чисел.
`CDec`	`Decimal`	+/- 7,92281625142643375935439503 35. Мінімальне ненульове число 0,0000000000000000000000000001.

Продовження табл.5.9

Функція	Тип	Діапазон значень аргумента
CInt	Integer	Від -32 768 до 32 767 з округленням дробової частини.
CLng	Long	Від -2 147 483 648 до 2 147 483 647 з округленням дробової частини.
CSng	Single	Від -3,402823E38 до -1,401298E-45 для від'ємних чисел; від 1,401298E-45 до 3,402823E38 для позитивних чисел.
CVar	Variant	Діапазон значень Double для числових значень. Діапазон значень String для нечислових значень.
CStr	String	Значення оберненої функції CStr залежать від аргумента виразу. Наприклад, CStr(True) = "Істина", CStr(122.344) = "122.344", CStr(#10/24/47#) = 24.10.47

Якщо значення аргумента передане у функцію, перебуває поза припустимим діапазоном для відповідного типу даних, виникає помилка.

Функція CVErr повертає значення типу Variant з підтипом Error, що містить код помилки, зазначений користувачем. Вона використовується для створення обумовлених користувачем помилок.

Якщо дробова частина числа дорівнює 0,5, то функції CInt і CLng завжди округлять число до найближчого парного числа. Наприклад, CInt (0,5) = 0, а CInt (1,5) = 2.

Функції Fix і Int обраховують цілу частину числа (без округлення). Вони відрізняються на від'ємних числах: Fix (-7.6) = -7, а Int(-7.6) = -8.

Форматування даних. Функції групи Format

Числові та строкові дані, дані типу дата, дані типу час можуть бути відформатовані відповідно до визначеного в мові формату або формату, передбачуваного користувачем. Для цієї мети використовується група функцій форматування.

Функція Format

Функція повертає рядок, відформатований відповідно до вказівок, заданих при виклику. Її синтаксис:

Format(expression[, format[, firstdayofweek[, firstweekofyear]]])

Її параметри:

- `expression` — будь-який правильний вираз.
- `format` — ім'я убудованого параметра або визначення користувальницького формату. Якщо параметр опущений, то застосовується формат, що залежить від типу першого аргумента
- `firstdayofweek i firstweekofyear` — їхній зміст був описаний, коли ми розглядали роботу з датами

Наведемо приклади, у яких застосовано форматування чисел на основі форматів, обумовлених за замовчуванням і користувачем:

```
? VBA.Format(55)
55
? VBA.Format(5.5)
5,5
? VBA.Format(-52.125, "##0.#0")
-52,13
? VBA.Format(-52.125, "000.##0")
-052,125
? VBA.Format(1152.125, "#,##0.#0")
1 152,13
? VBA.Format(0.52125, "0.##0%")
52,125%
```

Кілька прикладів форматування рядків:

```
? VBA.Format("5.4")
05.04.99
? VBA.Format("5,4")
5,4
? VBA.Format("Марія", ">")
МАРІЯ
? VBA.Format("Марія", "<")
марія
? VBA.Format("Марія", "> Це ")
 Це МАРІЯ
```

Дати форматуються звичайно з використанням убудованих форматів, але можна застосовувати й власні визначення форматів. От кілька прикладів:

```
? VBA.Format(VBA.Time, "Long Time")
16:24:57
? VBA.Format(VBA.Time, "Short Time")
16:25
? VBA.Format(VBA.Time, "hh/mm/ss")
16.26.03
? VBA.Format(VBA.Date, "Long Date")
9 Травень 1999 р.
? VBA.Format(VBA.Date, "Short Date")
09.05.99
? VBA.Format(VBA.Date, "yy/mm/dd")
99.05.09
```

Функція `Format` є загальною функцією, застосованою до довільних виразів. Але є функції застосовані до виразів спеціального типу, вони мають деякі додаткові можливості. Ми не станемо їх докладно розглядати, обмежимося простим перерахуванням:

- `FormatCurrency` — повертає грошовий вираз, використовуючи грошовий знак.
- `FormatDataTime` — повертає дату або час.
- `FormatNumber` — повертає вираз, відформатований як число.
- `FormatPercent` — повертає вираз, заданий у відсотках, із вказівкою знака відсотка.

Резюме

Розглянуто основні групи операторів програмування в середовищі Visual Basic. Особлива увага приділена роботі з процедурами та функціями, організації рекурсивних процедур.

Слід мати на увазі, що при застосуванні рекурсивних алгоритмів варто уникати трьох основних небезпек:

- *нескінченної рекурсії.* Переконайтеся, що умови зупинки вашого алгоритму припиняють всі рекурсивні шляхи.
- *глибокої рекурсії.* Якщо алгоритм досягає занадто великої глибини рекурсії, він може привести до переповнення стека. Мінімізуйте використання стека завдяки зменшенню числа обумовлених у процедурі змінних, використання глобальних змінних, або визначення змінних як статичних. Якщо процедура однаково приводить до переповнення стека,

перепишіть алгоритм у нерекурсивному виді, використовуючи усунення хвостової рекурсії.

- *непотрібної рекурсії.* Звичайно це відбувається, якщо алгоритм типу рекурсивного обрахування чисел Фібоначчі багаторазово обчислює одні й ті ж проміжні значення. Якщо ви стикнетеся із цією проблемою у своїй програмі, спробуйте переписати алгоритм, використовуючи підхід знизу угору.

Застосування рекурсії не завжди неправильне. Багато завдань є рекурсивними за своєю природою. У цих випадках рекурсивний алгоритм буде простіше зрозуміти, налагодити й підтримувати, ніж його нерекурсивну версію.

Контрольні запитання та завдання

1. Пояснити призначення оператора цикл з лічильником? Які значення має крок циклу? Чи завжди він вказується?
2. Чи завжди можна замінити оператор циклу з лічильником оператором умовного циклу і навпаки?
3. У яких випадках використовуються оператори умовного циклу?
4. У чому полягає відмінність у вживанні ключових слів While і Until?
5. Чи можуть ключові слова While і Until одночасно вживатися в одному операторі циклу?
6. З якою метою використовуються масиви? Навести приклади.
7. Що означає поняття розмірність масиву? Наведіть приклади одновимірних, двовимірних і тривимірних масивів.
8. Скільки місця в пам'яті займає двовимірний масив 100x100 елементів, якщо всі елементи: а) типу Integer, б) типу Single, в) типу Currencu?
9. Які стандартні функції мови Visual Basic ви знаєте?
10. Для чого необхідний опис процедур? Коли виконуються дії, зазначені в описі процедур?
11. Порівняйте вхідні і вихідні параметри? Які вимоги висуваються до відповідності формальних і фактичних параметрів?
12. У чому відмінність функції від процедури?

13. Дані дійсні, позитивні числа a, b, c, x, y. З'ясувати, чи пройде цеглина із ребрами a, b, c в прямокутний отвір із сторонами x, y. Просовувати цеглину в отвір дозволяється тільки так, щоб кожне з її ребер було паралельним або перпендикулярним кожній із сторін отвору.
14. Рука робота має три ланки, які є відрізками прямої лінії довжиною L1,L2,L3 відповідно. Кожна ланка має три ступені свободи. Визначити, чи зможе робот кінцем своєї руки дістати до її основи ?
15. Фірма володіє *N* пакетами акцій різних емітентів. Кількість акцій в пакеті A_i . Імовірний річний прибуток на *i*-ту акцію дорівнює P_i. Обрахувати середній імовірний прибуток фірми на будь - яку акцію.
16. Вантаж замовника переміщається із пункту A (координати (x_0,y_0) в пункт B (координати (x_n,y_n) через проміжні пункти з координатами $(x_1,y_1),...,(x_2,y_2)$. Шлях між сусідніми пунктами - це відрізок прямої, що їх з'єднує. Вартість перевезення вантажу із k-го в k+1-й пункт w(k) умовних одиниць за одиницю відстані. Обрахувати загальну вартість перевезення вантажу.
17. Хтось NN вирішив закупити партію цукру на суму в G гривень. Цукор продають M фірм за ціною P_i за одиницю ваги, де i номер фірми, причому запаси цукру на i-й фірмі дорівнюють V_i. Скласти програму, яка визначає план закупки, що максимізує її об'єм (вагу).
18. Діяльність фірми характеризується N напрямками, серед яких вона розподіляє свій капітал згідно з частками q(1), q(2),…, q(N). Ризик діяльності за i-м напрямком (імовірність втрати капіталу) дорівнює r(i). Напрямки діяльності між собою незалежні. Визначити загальний ризик втрати капіталу.
19. Дано текстовий рядок, що складається з букв і цифр. Перетворити рядок, замінивши в ньому кожну групу букв одним пробілом. Надрукувати максимальне число в отриманій послідовності чисел.
20. Дано текстовий рядок. Групи символів, розділені пробілами (одним чи декількома) і не маючи пробілів усередині себе, будемо називати словами. Підрахувати скільки разів кожне слово зустрічається в тексті.
21. Скласти рекурсивну процедуру, що обчислює двійковий код позитивного цілого числа.

22. Скласти рекурсивну функцію, що обчислює середнє геометричне n елементів одновимірного масиву.
23. Скласти рекурсивну процедуру для друку k-го елемента масиву цілих чисел.
24. Скласти рекурсивну функцію додавання двох цілих чисел, використовуючи тільки додаток одиниці.
25. Скласти рекурсивну функцію множення двох цілих чисел, використовуючи тільки операцію додавання.
26. Скласти рекурсивну процедуру без параметрів, яка підраховує кількість цифр у тексті.
27. Дана множина точок на площині: (x_1, y_1), (x_2, y_2),..., (x_n, y_n). Створити процедуру, яка визначає кількість точок, що належать перетину кругів $(x-a)^2+(y-b)^2<=r^2$ та $(x-c)^2+(y-d)^2<=s^2$. Скористатись цією процедурою.
28. Створити функцію, яка обчислює периметр n-кутника, заданого масивом координат його вершин. Скласти програму, яка використовує цю функцію.
29. Скласти процедуру циклічної перестановки елементів масиву a(1), a(2), ..., a(n), при якій a(1) переміщається в a(2), a(2) в a(3) і т.д. a(n 1) в a(n), a(n) в a(1).
30. Скласти функцію логічного типу, яка визначає належність точки x n-вимірного простору (X=(x(1),x(2),...,x(n))) області, що задається системою нерівностей:
31. a(1,1)*x(1)+a(1,2)*x(2)+...+a(1,n)*x(n)<=b(1)
32. a(2,1)*x(1)+a(2,2)*x(2)+...+a(2,n)*x(n)<=b(2)
33. .....
34. a(k,1)*x(1)+a(k,2)*x(2)+...+a(k,n)*x(n)<=b(k)

Розділ VI
РОЗРОБКА КОРИСТУВАЛЬНИЦЬКОГО ІНТЕРФЕЙСУ

Інтерфейс — це зовнішня оболонка додатка разом із програмами керування доступом та іншими схованими від користувача механізмами керування, яка дає можливість працювати з документами, даними й іншою інформацією, що зберігається в комп'ютері або за його межами. Головна мета будь-якого додатка — забезпечити максимальну зручність і ефективність роботи з інформацією: документами, базами даних, графікою або зображеннями. Тому інтерфейс є, мабуть, найважливішою частиною будь-якого додатка.

Добре розроблений інтерфейс гарантує зручність роботи із додатком і в остаточному підсумку його комерційний успіх. У цьому розділі описано види інтерфейсів і процес створення інтерфейсу з усіма його основними елементами керування: меню, контекстним меню, панелями інструментів, рядком стану.

Проектування інтерфейсу — процес циклічний. На цьому етапі розробки додатка бажано частіше спілкуватися з користувачами й замовниками додатка для вироблення найбільш прийнятних за ефективністю, зручністю й зовнішнім виглядом інтерфейсних рішень.

Вибір того або іншого типу інтерфейсу залежить від складності розроблювального додатка, оскільки кожний з них має деякі недоліки й обмеження й призначений для вирішення певних завдань. При цьому необхідно відповісти на ряд запитань: яка кількість типів документів обробляється в додатку, чи мають дані деревоподібну ієрархію, чи буде потрібна панель інструментів, яка кількість документів обробляється за певний інтервал часу (наприклад, за день) і т.д. Тільки після цього можна вибирати конкретний тип інтерфейсу.

Розглянемо можливі типи інтерфейсів та їх характерні особливості, що впливають на вибір інтерфейсного рішення додатка.

Загальні поради по розробці інтерфейсу

При розробці інтерфейсу потрібно керуватися наступними принципами:

- **Стандартизація.** Рекомендується використати стандартні, перевіреними багатьма програмістами й користувачами інтерфейсні рішення. Для Visual Basic це, зрозуміло, рішення Microsoft. Причому як стандарт (зразок для "наслідування") може служити кожний з додатків — Word, Excel або інші додатки Microsoft. Під рішеннями маються на увазі дизайн форм, розподіл елементів керування у формах, їхнє взаємне розташування, значки на кнопках керування, назви команд меню.
- **Зручність і простота роботи.** Інтерфейс повинен бути інтуїтивно зрозумілим. Бажано, щоб всі дії легко запам'ятовувалися й не вимагали стомлюючих процедур: виконання додаткових команд, зайвих натискань на кнопки, виклику проміжних діалогових вікон.
- **Зовнішній дизайн.** Не можна, щоб інтерфейс стомлював зір. Він повинен бути розрахований на тривалу роботу користувача з додатком протягом дня.
- **Не перевантаженість форм.** Форми повинні бути оптимально завантажені елементами керування. При необхідності можна використати вкладки або додаткові сторінки форм.
- **Угруповання**. Елементи керування у формі необхідно групувати за змістом, використовуючи елементи угруповання: рамки, фрейми.
- **Розрідженість об'єктів форм.** Елементи керування варто розташовувати на деякій відстані, а не ліпити один на одного. Для виділення елементів керування можна організувати порожні простори у формі.

Тут перераховані основні принципи, які варто враховувати при проектуванні інтерфейсу додатка, але вони не є догмою. Згодом у процесі роботи з користувачами й накопиченням практичного досвіду будуть вироблятися й свої оптимальні принципи побудови інтерфейсу.

Типи інтерфейсів

Для додатків, розроблювальних у середовищі Windows за допомогою Visual Basic 6, використовують три типи інтерфейсу: однодокуметний SDI (Single-Document Interface), багатодокументний MDI (Multiple-Dot Interface) й інтерфейс типу провідник (Explorer).

Під документом слід розуміти форму, яка призначена для роботи з даними, а не з конкретним документом.

У цьому розділі основна увага буде приділена першим двом типам інтерфейсу, оскільки вони найбільше часто застосовуються для розробки користувацьких додатків. Інтерфейс типу провідник використається в додатках не так часто, тому ми дамо тільки його короткий опис.

Однодокументний інтерфейс — це тип інтерфейсу, у якому надається можливість роботи тільки з одним документом в одному вікні. Прикладом може бути редактор Microsoft WordPad. Для роботи з декількома документами в такому інтерфейсі необхідно багаторазово запускати додаток. Для кожного типу даних і документів потрібна своя форма й, відповідно, свій додаток з інтерфейсом типу SDI. У принципі, це теж один з можливих варіантів, але він підходить тільки для роботи з невеликою кількістю форм документів. При завантаженні великої кількості SDI-додатків починає переповнюватися оперативна пам'ять комп'ютера й додатки працюють дуже повільно. Щораз при запуску SDI-додатка у пам'ять завантажуються одні й ті ж елементи (меню, панель, елементи керування), що виконують однакові дії, це призводить до неефективної й повільної роботи додатків.

Однак повністю відмовлятися від інтерфейсу типу SDI не потрібно, оскільки він цілком годиться для роботи з одним або двома документами (наприклад, для копіювання з одного документа в інший). Є й позитивні сторони додатків такого типу інтерфейсу — вони займають менше місця на диску й в оперативній пам'яті, та й на їхню розробку йде небагато часу, що буває важливо.

Інтерфейс типу MDI дає можливість працювати в одному додатку з будь-якою кількістю відкритих вікон.

SDI-інтерфейс

Інтерфейс типу SDI показаний на рис.6.1. Він складається з наступних елементів:

- головного меню;
- панелі інструментів з елементами керування;
- вікна додатка для розміщення елементів керування даними (у такому випадку це вікно із заголовком "Документ WordPad");
- елементів керування для роботи з даними. На рис.6.1 це одне велике поле для роботи з текстом;
- рядка стану.

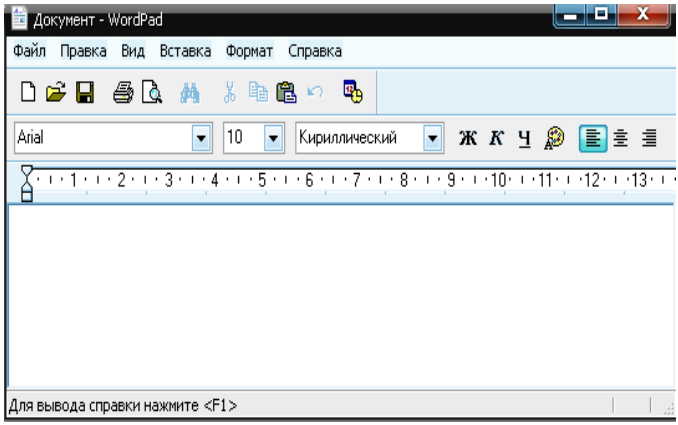


Рис. 6.1. Інтерфейс типу SDI

Цей тип інтерфейсу підходить для додатків, створених для роботи з документом одного типу з невеликою кількістю полів (інакше форма перевантажується елементами керування й інтерфейс стає складним і незручним).

MDI-інтерфейс

Головна особливість MDI полягає в тому, що для цього типу інтерфейсу можна багаторазово відкривати форму одного виду

документа для декількох різних за змістом документів. Прикладом інтерфейсу типу MDI може бути програма Microsoft Word (рис.6.2).

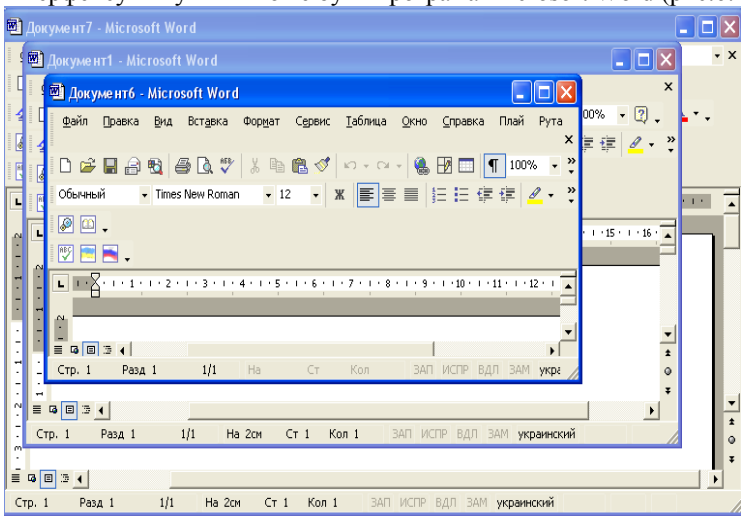


Рис. 6.2. Інтерфейс типу MDI

Для інтерфейсу такого типу характерна наявність одного головного вікна (MDI-вікно), що звичайно йменується батьківським вікном (на рис.6.2 це вікно із заголовком Microsoft Word), і необхідного для роботи кількості підпорядкованих (вкладених) вікон, називаних дочірніми (на рис.6.2 це вікна із заголовками Документ1, Документ6). Кількість відкритих дочірніх вікон обмежено лише можливостями комп'ютера.

Підкреслимо, що батьківське вікно для MDI-інтерфейсу може бути тільки одне, при цьому воно є контейнером для всіх дочірніх вікон. Це означає, що при мінімізації батьківського вікна разом з ним мінімізуються й всі дочірні вікна.

У свою чергу, дочірні вікна можуть перебувати тільки усередині батьківського, тобто при розкритті на весь екран дочірні вікна розкриваються повністю тільки в межах батьківського вікна й не можуть бути винесені або переміщені за ці межі.

До складу інтерфейсу MDI входять такі елементи:

- головне меню;
- панель інструментів з елементами керування;
- головне вікно додатка (MDI-вікно);
- дочірні вікна;

- елементи керування для роботи з даними, розташовані в дочірніх вікнах (на рис.6.2 це одне велике поле для роботи з текстом документа в кожному з вікон);
- рядок стану.

Розглянемо докладніше елементи інтерфейсу й технологію їхнього створення за допомогою інструментальних засобів Visual Basic.

Батьківське вікно MDI-інтерфейсу

Для додавання батьківської форми в проект можна виконати одну з таких дій:

1. у меню **Project** (Проект) вибрати команду **Add MDI Form** (Додати MDI-форму);
2. у вікні провідника нажати праву кнопку миші й вибрати з контекстного меню команду **Add**, а потім значення **MDI Form**. MDI-форма показана на рис.6.3.

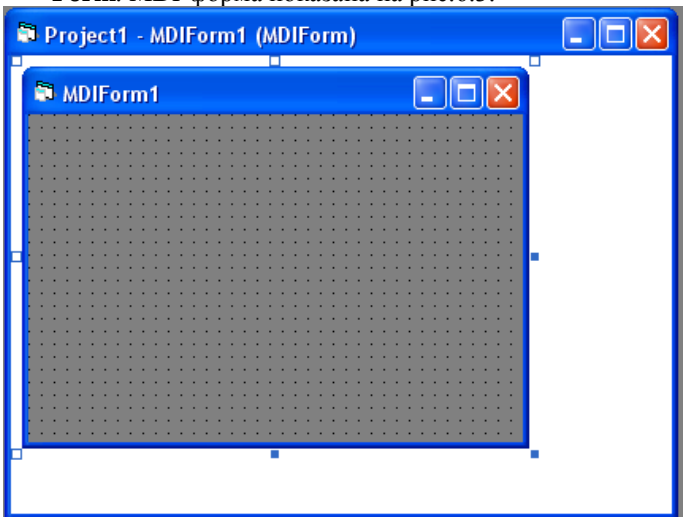


Рис. 6.3. Форма батьківського вікна інтерфейсу типу MDI

Батьківське й дочірнє вікна мають набір основних властивостей, описаний в табл. 6.1. Доступ до цих властивостей можна одержати за допомогою вікна **Properties** (Властивості).

Таблиця 6.1

Набір основних властивостей батьківських і дочірніх вікон

Властивість	**Призначення**
`BackColor`	Визначає колір поля форми. Вибирається із пропонованої палітри
`BorderStyle`	Визначає тип обрамлення, а також управляє змінами розмірів форми. Додатково визначає наявність кнопок у рядку заголовка вікна
`Caption`	Задає текст, який буде виведений у заголовку форми
`ControlBox`	Визначає наявність у лівому верхньому куті вікна кнопки у вигляді значка, що відкриває віконне меню
`Font`	Задає шрифт у вікні
`ForeColor`	Задає кольори виведеного у вікні тексту. Рекомендується використати стандартні кольори як поля, так і тексту
`Height`	Задає висоту форми. При конструюванні автоматично модифікується при зміні розмірів за допомогою миші
`Icon`	Задає значок для форми
`Left`	Задає відстань від форми до лівого краю екрана
`MaxButton`	Управляє наявністю кнопки розгортання (максимізації) вікна
`MinButton`	Управляє наявністю кнопки згортання (мінімізації) вікна
`MDIChild`	Управляє можливістю перетворення вікна типу MDI у дочірнє. Приймає значення True або False. За замовчуванням приймає значення False.
Name	Задає унікальне ім'я форми для використання в програмних модулях і тексті коду. При призначенні імені рекомендується використати префікс frm або mdi
ScaleMode	Задає одиницю виміру властивостей Width, Height, Left і Top
ShowInTaskbar	Управляє відображенням значка вікна на панелі завдань системи Windows
Width	Визначає ширину форми

Продовження табл. 6.1

Властивість	Призначення
WindowState	Управляє станом форми при запуску й може приймати наступні значення: 0-Normal — нормальний стан, 1-Minimized — мінімізований стан і 2-Maximized — максимізований стан

Властивість `BorderStyle` визначає тип обрамлення вікна й може приймати значення, наведені в табл. 6.2.

Таблиця 6.2

Тип обрамлення вікна, що задає властивістю `BorderStyle`

Значення властивості	Тип обрамлення
0-None	Відсутня рамка вікна. Зміна розмірів і переміщення вікна заборонена. Відсутні основні атрибути вікна: заголовок, меню, кнопки згортання, розгортання й закриття
1-Fixed Single	Вікно з рамкою у вигляді одинарної лінії й з фіксованими розмірами. Зміна розмірів вікна за допомогою миші заборонено. Основні атрибути заголовка форми, згадані вище, доступні для цього значення
2-Sizable	Вікно має рамку, розміри якої можна змінити за допомогою миші. Дозволене також переміщення вікна. Всі атрибути заголовка вікна присутні. Це значення встановлюється за замовчуванням при створенні форми
3-Fixed Dialog	Вікно має рамку, розміри якої змінити не можна, при цьому дозволене переміщення вікна. У заголовку розташована тільки кнопка закриття вікна. Це значення властивості BorderStyle використається, в основному, для діалогових вікон
4-Rxed ToolWindow	Аналогічно значенню 3-Fixed Dialog. При цьому заголовок уже й містить тільки кнопку закриття вікна
5-SizableToolWindow	Аналогічно значенню 2-Sizable, однак кнопки згортання й розгортання вікна при цьому недоступні

Властивість ScaleMode задає одиницю виміру властивостей Width, Height, Left і Top і може приймати значення, описані в табл. 6.3.

Таблиця 6.3

Значення, які приймає властивість ScaleMode

Значення	Одиниця виміру
0-User	Користувальницька
1-Twip	Твіп. Є стандартною одиницею виміру для Visual Basic і встановлюється за замовчуванням при створенні форми. В одному дюймі втримується 1440 твіпів
2-Point	Крапка. Необхідно мати на увазі, що в дюймі 72 крапки
3-Pixel	Піксел. Базова апаратна одиниця виміру. Визначає мінімальний припустимий розмір крапки екрана або крапки при виводі на принтер
4-Character	Символ
5-Inch	Дюйм
6-Milimeter	Міліметр
7-Centimeter	Сантиметр

Властивості вікон можна змінювати в режимі проектування. Крім того, Visual Basic дає можливість змінювати властивості вікон із програми в режимі виконання.

Тут варто помітити, що на відміну від дочірнього, для батьківського вікна доступна тільки частина з перерахованих властивостей. Наприклад, недоступні властивості MaxButton, MinButton або ControiBox, що керують кнопками в заголовку вікна.

У свою чергу, існують властивості, характерні тільки для батьківського вікна. Це властивості AutoShowChildren і ScrollBars. Якщо властивість AutoShowChildren має значення **True** (це значення використається за замовчуванням), то при завантаженні батьківського вікна автоматично завантажується дочірнє вікно. Властивість ScrollBars визначає наявність смуг прокручування в батьківському вікні для дочірніх вікон, що виходять за межі видимості. За замовчуванням ця властивість має значення **True**, що дозволяє відображення смуги прокручування.

У табл. 6.4 наведені основні події, які найчастіше використаються для форм. У процесі програмування ви познайомитеся з ними більш докладно.

Таблиця 6.4

Події, використовувані формами

Подія	Опис
Activate	Відбувається в той момент, коли форма стає активною. Пов'язане з подіями Initialize, Load, GotFocus. При відкритті форми спочатку відбувається подія Initialize, потім Load, після цього безпосередньо Activate і завершує процес відкриття подія GotFocus
Click	Відбувається при натисненні кнопки миші
DblClick	Подвійний натиснення кнопки миші
Deactivate	Подія, протилежне Activate. Відбувається, коли форма стає неактивною
GotFocus	Відбувається при установці фокуса на формі, що активується
Initialize	Ініціалізація форми
KeyUp	Відбувається при натисканні клавіші на клавіатурі
Load	Відбувається при завантаженні форми у пам’ять до її появи на екрані
MouseUp	Відбувається при натисканні кнопки миші
Resize	Відбувається при зміні розмірів форми
Unload	Протилежне події Load. Відбувається перед вивантаженням форми з пам'яті й видаленням її з екрана

При проектуванні можна переглянути список подій, пов'язаних з формою, у вікні редактора коду форми. Для цього необхідно виконати команду Code (Код) меню View (Вид) або перемістити покажчик у поле конструювання форми й двічі клацнути кнопкою миші. У верхній частині вікна редактори коду розташовані два списки. Виберіть із лівого списку об'єктів поточну форму. При цьому в правому списку будуть розташовані весь події форми.

При програмуванні подій Visual Basic 6 відразу ж при виборі події надає готову конструкцію (шаблон коду) для програмування дій по події. Ім'я процедури обробки події завжди пов'язане з його ім'ям.

Наприклад, для обробки події `Load` автоматично надається такий шаблон:

```
Private Sub Form_Load()
... код процедури обробки події
End Sub
```

Дочірнє вікно MDI-інтерфейсу

Дочірнє вікно (рис.6.4) має всі описані в табл. 6.1 власті вікон. Практично всі вони доступні при проектуванні.

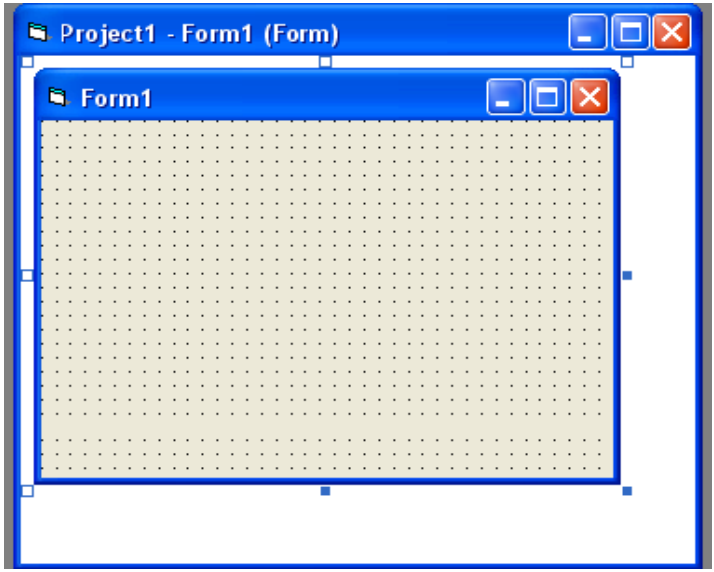


Рис. 6.4. Форма дочірнього вікна інтерфейсу типу MDI

Основні особливості дочірніх вікон:

- дочірня форма завжди розташовується в границях батьківської;
- дочірню форму не можна перемістити за межі батьківської форми;
- дочірнє вікно згортається тільки усередині батьківського;
- при розгортанні дочірнє вікно займає весь внутрішній простір батьківського вікна.

Розташуванням дочірніх вікон у батьківському вікні можна управляти за допомогою методу Arrange батьківського вікна й набору змінних, які задають способи розміщення дочірніх вікон (табл. 6.5).

Таблиця 6.5

Значення констант для методу Arrange

Константа	**Значення**	**Опис**
vbCascade	0	Каскадне розташування дочірніх форм, при цьому кожна наступна зрушена вниз і вправо приблизно на ширину заголовка форми
vbTileVertical	1	Розташування у вигляді вертикальної мозаїки, при цьому дочірні форми мають висоту батьківської форми й таку ширину, щоб розміститися по всій ширині батьківського вікна
vbTileHorizontal	2	Розташування у вигляді горизонтальної мозаїки, при цьому дочірні форми мають ширину батьківської форми й таку висоту, щоб розміститися по всій висоті батьківського вікна
vbArrangeIcon	3	При мінімізації вікна розташовуються на нижньому краї батьківського вікна

Для додавання дочірньої форми в проект необхідно виконати команду **Add Form** (Додати форму) меню **Project** . Як дочірню форму можна використати форму автоматично створювану при виборі нового проекту. При цьому властивість `MDichiid` варто встановити в значення **True.**

Для дослідження характеристик і взаємодії батьківського й дочірнього вікон створимо невеликий додаток. Для цього виконаємо наступні дії:

1. Створіть новий проект. Відкрийте вікно властивостей проекту, у поле **Project Name** уведіть ім'я проекту **MyMDlApp.**
2. Щоб автоматично використати створену форму проекту в якості дочірньої, відкрийте вікно властивостей, викликавши його командою **Properties Window** (Вікно властивостей) у

меню **View** (Вид), і встановіть для властивості `MDichild` значення **True**.

3. Використовуючи властивість `Name`, уведіть ім'я форми f**rmChildMDI**.
4. У поле властивості `Caption` уведіть заголовок вікна **Дочірнє вікно MDI**.
5. Додайте в проект MDI-форму, виконавши команду **Add MDI Form** меню **Project**.
6. Скориставшись властивістю `Name`, уведіть ім'я форми **mdiParentMDI**.
7. Використовуючи властивість `Caption`, уведіть заголовок вікна **Батьківське вікно MDI**.
8. Для демонстрації деяких дій нам знадобиться просте меню. Створимо його по крокам (редактор меню описаний у розділі "Меню" даної глави).

- Виберіть батьківське вікно.
- Викличте редактор меню командою **Menu Editor** (Редактор меню) з меню **Tools** (Сервіс).
- У поле `Name` (Ім'я) уведіть ім'я **mnuFile.**
- У поле `Caption` (Заголовок) уведіть заголовок меню **Файл**.
- Тепер створіть пункт цього меню. Натисніть кнопку `Next` і кнопку зі стрілкою вправо. У поле `Name` введіть ім'я меню **mnuFileNewFortn**, а в поле `Caption` його заголовок **Нова форма**.

Невеликий додаток готовий до роботи. Для перевірки виконайте його за допомогою команди **Start** (Запустити) меню **Run** (Запуск). Зупинка додатка виконується командою **End** (Зупинити) того ж меню.

9. Запрограмуємо виклик декількох дочірніх форм. Для цього виділіть батьківську форму в конструкторі форм і двічі клацніть лівою кнопкою миші для виклику редактора коду батьківського вікна.
10. У списку об'єктів (лівий список, що розкривається) виберіть пункт; меню `mnuFileNewForm`, У лівому списку подій виберіть подію `Click` і напишіть для нього наступний код :

```
Private' Sub mnuFileNewForm_Click()
Dim frmNewForm As New frmChildMDI
frmNewForm.Show
End Sub
```

У тексті коду ми оголосили об'єктну змінну `frmNewForm` для посилання на знову створений екземпляр вікна. Потім за допомогою методу `Show` новий екземпляр візуалізується в батьківському вікні. Для обчислення номера дочірнього вікна використовується змінна `frmcount`. Цей код виконується при виборі команди **Нова форма** з меню **Файл** батьківського вікна.

11. За допомогою цього простого додатка можна перевірити всі основні характеристики батьківського й дочірнього MDI-вікон. Для перевірки можливостей упорядкування дочірніх вікон у батьківському вікні(властивість Arrange) додамо в меню пункт **Вікно** й команду меню, що виконує впорядкування. Для цього відкрийте редактор меню й додайте пункт **Вікно** й необхідну команду. Привласніть пункту меню ім'я mnuwindow, а команді, призначеній для розташування вікон каскадом, ім'я mnuWindowTiieVerticai. Програмний код повинен виглядати в такий спосіб:

```
Private Sub mnuWindowTileVertical_Click()
mdiParentMDI.Arrange vbTileVertical
End Sub
```

Нижче наведений повний текст коду створеного нами простого додатка:

```
Dim frmCount As Integer
Private Sub MDIForm_Load()
frmChildMDI.Caption = "Дочірня форма 1"
frmCount = 1
End Sub

Private Sub mnuFileNewForm_Click ()
Dim frmNewForm As New frmChildMDI
frmCount = frmCount + 1
frmNewForm.Caption = "Дочірня форма  "  + Str(frmCount)
frmNewForm.Show
End Sub

Private Sub mnuWindowTileVertical_Click()
mdiParentMDI.Arrange vbTileVertical
End Sub
```

Працюючий додаток з упорядкованими вікнами представлений на рис.6.5.

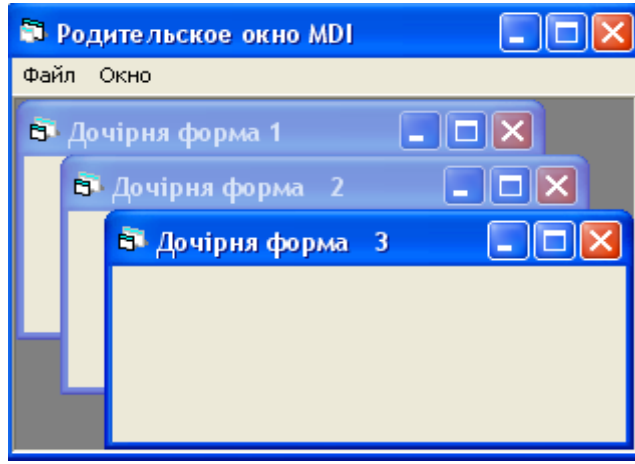


Рис.6.5. Робота простого додатка у стилі інтерфейсу MDI

Аналогічно можна запрограмувати інші способи впорядкування дочірніх вікон у батьківському.

Інтерфейс типу провідник

Інтерфейс типу провідник розробляється для доступу до ієрархічних деревоподібних структур, тобто до таким, де зустрічається вкладеність. Прикладом вкладеності можуть служити папки й файли. Файли лежать у папках, які у свою чергу лежать у вищестоящих папках і так далі. Прикладом такого інтерфейсу є провідник Windows (рис.6.6). На рисунку наочно видна структура зберігання папок і файлів, що утворить ієрархічне дерево. По своїй суті це аналог інтерфейсу SDI, розроблений спеціально для деревоподібних структур.

Інтерфейс додатка типу провідник містить наступні елементи:

- головне меню;
- вікно додатка для розміщення елементів керування даними (у нашому випадку це вікно із заголовком "Провідник — Microsoft Visual Studio");

- ієрархічний список елементів деревоподібної структури. Це можуть бути папки й файли, документи, якщо вони організовані в ієрархічну структуру;
- елементи керування для роботи з даними: кнопки, поля, прапорці й т.п.;
- рядок стану.

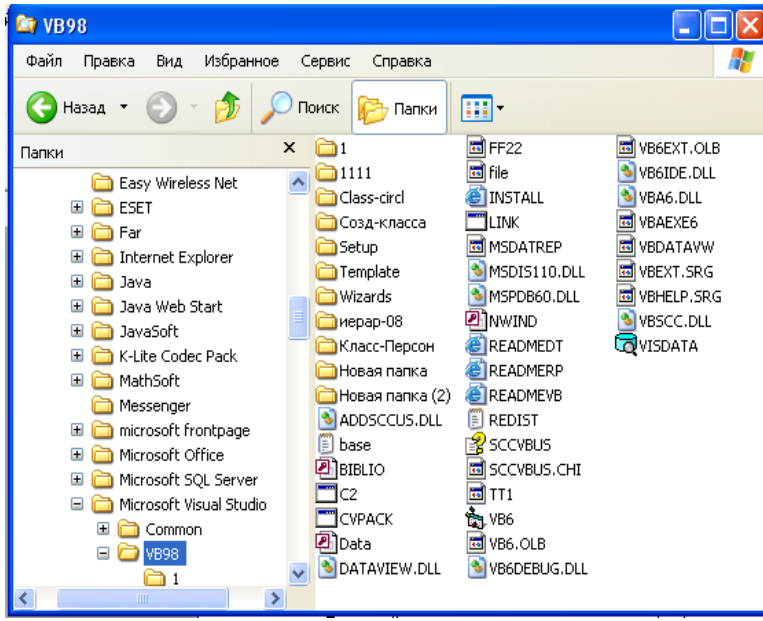


Рис. 6.6. Інтерфейс типу провідник (Explorer)

Елементи інтерфейсу

Розглянемо докладніше основні елементи інтерфейсу й можливості їхнього використання в додатках Visual Basic 6.

Меню

Будь-який додаток створюється для реалізації комплексу функцій, забезпечуючих виконання загального завдання додатка. Для швидкого доступу до всіх функцій додатка використається меню: головне меню додатка й контекстне меню окремих об'єктів додатка (форм, панелей).

При проектуванні меню варто керуватися певними принципами. Головний з них — стандарти. Рекомендується дотримуватися стандартних назв команд меню і їх розташування: наприклад, пункт меню для роботи з файлами рекомендується називати у своїх додатках **File** (Файл), а пункт меню для виклику довідкової системи додатка — **Help** (Довідка). При цьому пункт меню **File** бажано розташовувати найпершим, а пункт **Help** — останнім. На цей стандарт для всіх додатків Windows, до якого звикли багато мільйонів користувачів, можна цілком полягати. Додаток буде при цьому більше зрозумілим користувачеві.

У процесі розробки меню бажано групувати команди меню, реалізуючи функції для рішення конкретного завдання (наприклад, робота з файлами), в одне меню, що розкривається, якому буде відповідати пункт меню. Наприклад, всі команди, що реалізують функції роботи з файлами, бажано згрупувати в одне меню, що розкривається, зв'язане з пунктом меню **File.**

У додатках кожній команді меню, як правило, відповідає "гаряча" клавіша (клавіша швидкого доступу). Рекомендується по можливості використовувати стандартні клавіші, наприклад, як в Microsoft Word. У принципі, цей додаток може служити стандартним зразком для створення власних додатків при розробці не тільки меню, але й всіх інших елементів.

Як і будь-який інший об'єкт додатка, меню має набір властивостей. Властивості меню доступні для редагування у вікні Properties (Властивості) форми, який належить меню (рис.6.7). Основні властивості меню наведені в табл. 6.6.

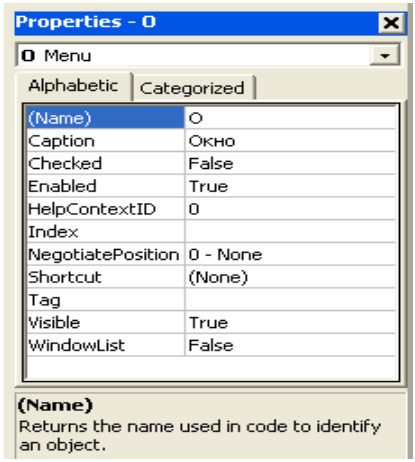


Рис. 6.7. Властивості меню

Таблиця 6.6.

Основні властивості меню

Властивість	**Призначення**
`Name`	Найменування (ім'я) меню. Повинне бути унікальним. Бажано користуватися стандартним присвоєнням імені, тобто ім'я повинне починатися з букв mnu
`Caption`	Текст, відображуваний у пункті меню. Якщо в цьому тексті перед однією з букв помістити символ "&", то буква в пункті меню буде підкреслена й клавіша цієї букви буде призначена "гарячою" клавішею для швидкого доступу до даного пункту меню
`Checked`	Якщо ця властивість має значення True, при роботі додатка ліворуч від найменування обраного пункту меню з'являється галочка
`Enabled`	Властивість, що визначає можливість виконання команди (пункту) меню. Залежно від контексту об'єкта команди забороняються або дозволяються
`HelpContexti`	Ідентифікатор довідкової системи, що відповідає довідці про це меню

Продовження табл. 6.6

Властивість	Призначення
`Index`	Ідентифікатор пункту меню в масиві елементів керування додатка
`NegotiatePosition`	Визначає положення меню на екрані
`Shortcut`	Комбінація клавіш для швидкого виконання пункту меню
`Visible`	Визначає видимість на екрані пункту меню. При роботі додатка за допомогою цієї властивості пункти меню можна динамічно ховати або показувати
`WindowList`	Призначає властивість формування динамічного списку вікон. При установці цієї властивості в меню буде додаватися список вікон у міру їхнього запуску при роботі додатка. Це властивість використається для пункту меню самого верхнього рівня й для батьківського вікна додатків з інтерфейсом типу MDI

Редактор меню Menu Editor

Для проектування меню всіх видів використається редактор меню **Menu Editor** (Редактор меню) середовища проектування IDE (рис.6.8). Редактор меню викликається одним з наступних способів:

- командою **Menu Editor** (Редактор меню) меню **Tools** (Інструменти);
- натисканням кнопки Menu Editor  на стандартній панелі інструментів;
- натисканням комбінації клавіш <Ctrl>+<E>.

Редактор створює меню для активного в цей момент вікна, тобто, якщо активно MDI-вікно, проектується меню для нього, якщо активна дочірня форма, проектується меню для дочірньої форми. На рис.6.8 показаний редактор меню батьківського вікна, створеного автоматично майстром додатків **Application Wizard**.

Редактор меню складається із двох груп: елементів керування властивостями й елементів конструювання структури меню.

Управляти основними властивостями меню, про які було сказано вище, можна за допомогою наступних елементів редактори меню:

- поле **Caption** (Заголовок) — найменування пункту меню, тобто текст, що з'являється в меню;
- поле **Name** (Ім'я) — ім'я меню. Використається для ідентифікації об'єкта при написанні програмних кодів;
- список, що розкривається, **Shortcut** (Оперативна клавіша) — призначає комбінації клавіш для швидкого виклику команди меню;
- поле **HelpContextID** (Ідентифікатор довідки) — посилання на тему в довідниковій системі;
- прапорець **Enabled** (Доступно) — доступ до пункту меню;
- прапорець **Visible** (Видимість) — визначає, чи буде видний на екрані елемент меню;
- прапорець **WindowList** (Список вікон) — визначає наявність списку відкритих вікон.

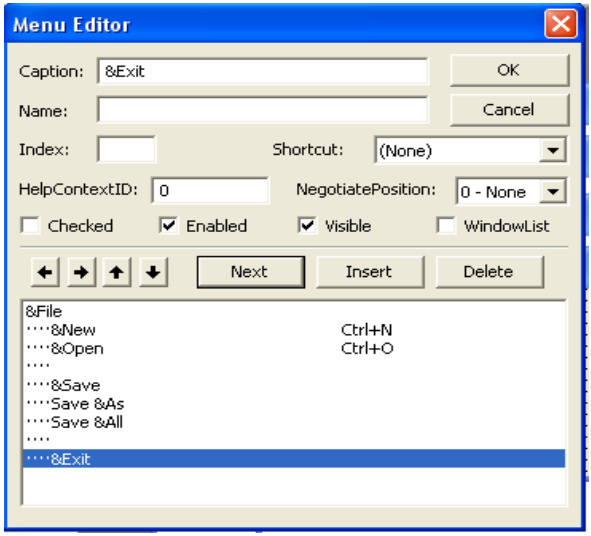


Рис. 6.8. Вікно редактора меню Menu Editor

Елементи групи конструювання структури меню дозволяють додавати й видаляти нові пункти меню, переміщати пункти по вертикалі, міняючи порядок їхнього проходження, і по горизонталі, міняючи розташування пунктів в ієрархії системи меню:

- кнопки зі спрямованими вправо й уліво стрілками переміщають пункти або команди меню в ієрархії меню;
- кнопки зі спрямованими нагору й униз стрілками переміщають пункти або команди меню за структурою меню;
- **Next** (Наступний) — переміщає покажчик до наступного пункту меню. Якщо покажчик перебуває на останньому пункті меню, то створюється новий пункт меню або нова команда меню такого ж рівня ієрархії;
- **Insert** (Вставити) — додає пункт меню або команду в пункт меню;
- **Delete** (Видалити) — видаляє пункт меню або команду з пункту меню.

Для додавання пункту меню виконайте наступні кроки:

1. Виберіть місце в наявній структурі меню.
2. Додайте пункт меню, нажавши кнопку **Insert** (Вставити).
3. У поле **Caption** (Заголовок) уведіть назву пункту меню, що буде відображатися в рядку меню при запуску додатка на виконання.
4. У поле **Name** (Ім'я) уведіть ім'я пункту меню, по якому він ідентифікується в програмному коді.

Контекстне меню

Для додатка будь-якого типу можна використати зручний засіб швидкого доступу до функцій — контекстне меню. Контекстне меню пов'язана з деякою дією (звичайно це щиглик правої кнопки миші на об'єкті) і викликається в будь-якім місці додатка. У вихідному стані контекстне меню невидимо й візуалізується поруч із покажчиком миші після виклику. Контекстним таке меню називається тому, що воно появляється поруч із обраним об'єктом, і його склад залежить від змісту (контексту) цього об'єкта. Після вибору команди з контекстного меню вона зникає.

Проектується контекстне меню як один з пунктів верхнього рівня рядка меню. Оскільки меню повинне бути приховане й викликатися певною дією, властивість **Visible** проектованого пункту меню необхідно встановити в значення **False**.

Для візуалізації контекстного меню на екрані необхідно використовувати метод **PopUpMenu** форми. Наприклад, для виклику меню **File** (Файл) при натисканні правої кнопки миші досить написати таку умовну конструкцію для події **MouseUp** форми:

```
If Button = vbRightButton
Then Forml.PopUpMenu mnuFile
End If
```

У цьому прикладі при виконанні у формі події MouseUp (Натискання кнопки миші) перевіряється, яка кнопка натиснута. У цьому випадку, при допомозі внутрішньої константи **Visual Basic 6 vbRightButton** (Права кнопка) перевіряється натискання на праву кнопку миші й запускається метод **PopUpMenu**. Як параметр задається ім'я необхідного контекстного меню.

Панелі інструментів

У доповнення до рядка стану й контекстному меню, сьогодні великою популярністю користуються панелі інструментів, що дозволяють прискорити доступ до функцій додатка. Звичайно панель інструментів містить найбільш часто використовувані команди рядка меню або контекстних меню. При розробці додатків можна застосовувати панелі двох видів: звичайну панель TooBar (рис.6.9) і поліпшену панель CoolBar (рис.6.10).

Рис. 6.9. Звичайна панель інструментів

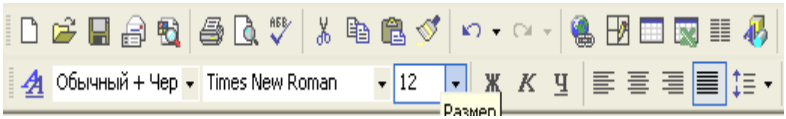


Рис. 6.10. Поліпшена панель інструментів

Для проектування панелей інструментів в Visual Basic 6 використаються:

- майстер панелей інструментів, що входить до складу майстра додатків;
- елемент керування TooBar для створення звичайної панелі інструментів;
- елемент керування CoolBar для створення поліпшеної панелі інструментів.

Майстер панелей інструментів Toolbar Wizard

Для проектування панелей інструментів у програмі Visual Basic можна використати майстер панелей інструментів, що працює в складі майстра додатків VB Application Wizard.

Діалогове вікно майстра показане на рис.6.11. У верхній частині вікна майстра розташована пропонована за замовчуванням панель інструментів, нижче розташовані два списки. Лівий список містить набір кнопок, що можуть бути додані на панель інструментів. Правий список відображає набір кнопок, уже розміщених на панелі інструментів.

Для додавання кнопки на панель інструментів необхідно вибрати в лівому списку необхідну кнопку й перенести її в правий список, виконавши одне з наступних дій:

- двічі клацнути мишею;
- нажати розташовану між списками кнопку із зображенням направленої вправо стрілки;
- використати механізм перенести-й-залишити, тобто нажати кнопку миші й, утримуючи її натиснутою, перенести кнопку в правий список, після чого відпустити кнопку миші.

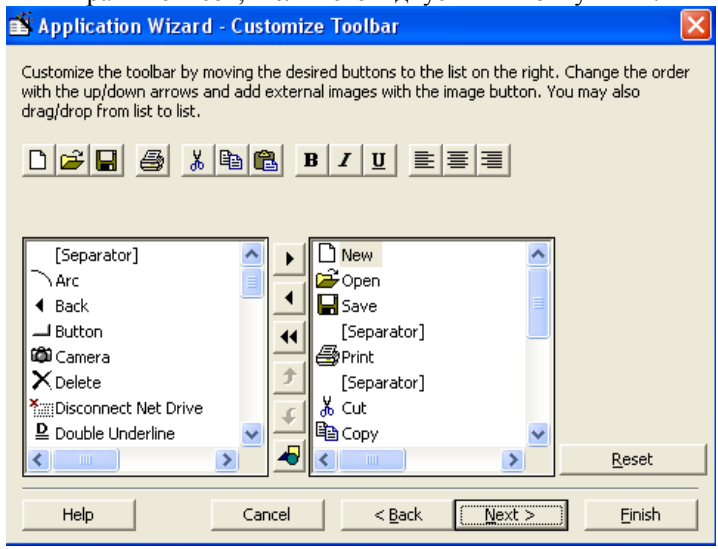


Рис. 6.11. Вікно майстра проектування панелей інструментів

При виконанні кожної із цих дій необхідна кнопка переміщається в правий список і одночасно додається на панель інструментів.

Щоб відредагувати найменування або змінити значок кнопки, розміщених на панелі інструментів у верхній частині вікна майстри, натисніть дану кнопку. Відкриється діалогове вікно **Button Attributes** (Атрибути кнопки) (рис.6.12), у якому можна змінити необхідні параметри. Для зміни значка натисніть кнопку **Change Bitmap** і за допомогою діалогового вікна, що відкрилося, виберіть графічний файл для зображення, розташовуваного на кнопці.

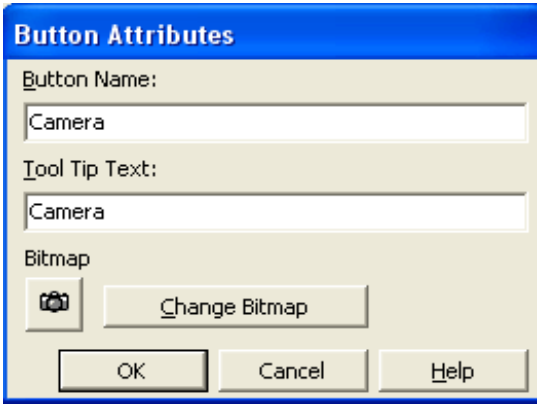


Рис. 6.12. Діалогове вікно Button Attributes

Кнопка **Reset** (Скидання) повертає панель інструментів у початковий стан, який встановлюється майстром за замовчуванням. Змінити послідовність кнопок на панелі інструментів можна кнопками переміщення позицій у списку. Варто мати на увазі, що кнопки переміщення позицій управляють тільки правим списком.

Використовуючи кнопку **Load an External Bitmap or Icon**  , можна додати на панель інструментів свою кнопку. Користувальницька кнопка розміщається в правому списку й на панелі інструментів.

Елемент управління ToolBar

Елемент управління ToolBar надає можливість застосовувати в додатках звичайні панелі інструментів. Для використання цього елемента управління в додатку необхідно підключити до проекту бібліотеку Microsoft Windows Common Controls 6.0, вибравши команду **Components** (Компоненти) меню **Project** (Проект). У вікні **Components**, що відкрилося при виконанні цієї команди, необхідно знайти в списку зазначену бібліотеку, установити розташовуваний з лівої сторони прапорець і нажати кнопку **ОК**. Після підключення цієї бібліотеки елемент керування ToolBar з'явиться на панелі елементів керування.

Панель інструментів **ToolBar** є контейнером і може містити наступні елементи керування:

- кнопку **PushButton**, що працює як звичайна кнопка форми;
- кнопка-прапорець **CheckButton**, що працює як прапорець і може перебувати в станах натиснуте/віджате;
- групу кнопок-перемикачів **ButtonGroup**, що працюють як аналог перемикача у формі, тобто натиснута може бути тільки одна кнопка із групи;
- кнопка-роздільник **Separator**, використовувана для створення інтервалів між кнопками;
- кнопка-замінник **Placeholder**, що служить для резервування місця на панелі для інших типів елементів керування, таких як, список ComboBox, список ListBox, перемикач OptionBox, текстове поле TextBox;
- кнопку списку, який розкривається, **DropDown**, що працює аналогічно.

Управляти панеллю можна за допомогою основних властивостей, описаних у табл. 6.7.

Таблиця 6.7

Основні властивості панелі інструментів

Властивість	**Призначення**
AllowCustomize	Дозволяє настроювати панель інструментів при роботі додатка, тобто додавати, видаляти, переміщати кнопки панелі
BorderStyle	Задає тип обрамлення панелі

Продовження табл. 6.7

Властивість	Призначення
ButtonHeight	Задає висоту кнопок панелі, при цьому автоматично обчислюється висота самої панелі
ButtonWidth	Задає ширину кнопок панелі
Enabled	Управляє доступом до виконання дій на панелі інструментів
ButtonHeight	Управляє доступом до виконання дій на панелі інструментів
ShowTips	Установлює режим відображення підказки для кнопок панелі інструментів
visible	Задає видимість панелі інструментів
Wrappable	Установлює режим автоматичного переносу кнопок на інший ряд при недоліку місця

Можливості настроювання панелі інструментів розглянемо на прикладі. Для цього сконструюємо невелику панель інструментів у додатку **MyMDIApp**, створеному для перевірки властивостей інтерфейсу в стилі **MDI**. Виконаємо для цього наступні дії:

1. Підключить до проекту бібліотеку **Microsoft Windows Common Controls 6.0**, у якій утримується елемент керування ToolBar, скориставшись діалоговим вікном **Components** (Компоненти).
2. Додайте в батьківську форму об'єкт **ToolBar**, двічі клацнувши мишею кнопку **ToolBar** на панелі елементів керування.
3. Відкрийте вікно властивостей, вибравши команду **Properties Window** (Вікно властивостей) у меню V**iew** (Вид) при виділеній у формі панелі інструментів.
4. Використовуючи властивість **Name**, привласніть панелі ім'я **tbrTools**.
5. Панель інструментів створена, тепер необхідно розмістити на ній кнопки. Для цього викличемо вікно проектування панелі, нажавши на ній праву кнопку миші й вибравши в контекстному меню, що з'явився, команду **Properties** (Властивості).
6. В діалоговому вікні, що відкрилося, **Property Pages** (рис.6.13) за замовчуванням обрана вкладка **General** (Основна), що дозволяє встановити основні властивості

панелі. Залишимо параметри у початковому стані. Виберіть вкладку **Buttons** (Кнопки), за допомогою якої створюються кнопки панелі інструментів.

7. Натисніть кнопку **Insert Button** (Вставити кнопку) для додавання нової кнопки на панель інструментів. Цю кнопку будемо використати для виклику нового дочірнього вікна форми, тобто як аналог команди **Нова форма** меню **Файл** батьківського вікна.
8. У полях, що управляють властивостями кнопок, задайте наступні значення:

- у поле **Caption** уведіть **Нова**;
- у поле **Key** уведіть ідентифікатор кнопки для використання в програмному коді **ToolsNewForm;**
- у поле **ToolTipText** (Підказка) уведіть текст **Відкрити нове дочірнє вікно**, що буде з'являтися у вигляді підказки під курсором при його установці на кнопку;
- у поле **Style** залишіть значення **0-tbrDefault**, що визначає звичайну кнопку. Розмір кнопки буде адекватний уведеному тексту в поле **Caption** і зміні недоступний.

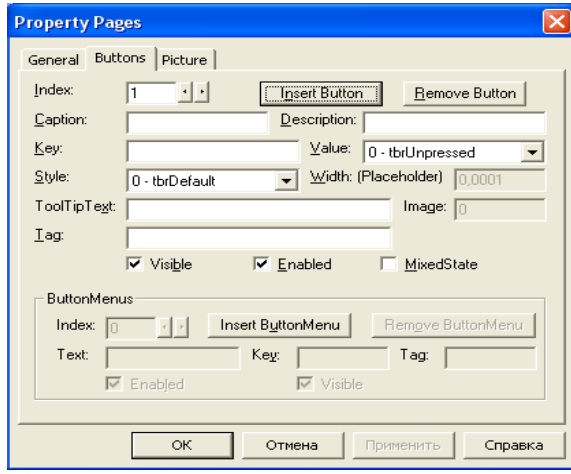


Рис. 6.13. Діалогове вікно Property Pages

9. Запрограмуємо дію, виконувану при натисканні нової кнопки. Для виклику редактора коду двічі клацніть мишею на панелі **tbrTools.** При цьому буде автоматично використаний шаблон для дії, виконуваних при натисканні на кнопку панелі інструментів. Уведіть програмний код, що слідує:

```
Private Sub tbrTools_ButtonClick(ByVal Button As
  MSComctlLib.Button)
Select Case Button.Key
Case Is = "ToolsNewForm"
Dim frmNewForm As New frmChildMDI
frmCount = frmCount + 1
frmNewForm.Caption = "Дочірня форма " +
Str(frmCount)
frmNewForm.Show
End Select
End Sub
```

Незважаючи на те, що кнопка на панелі інструментів усього одна, у коді використана конструкція Select case для відпрацьовування дій інших кнопок, що додають на панель.

На рис.6.14 показаний додаток, що містить панель інструментів з однією кнопкою.

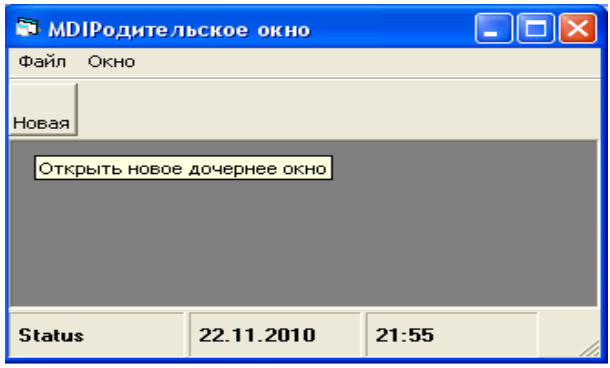


Рис. 6.14. Приклад додатка з використанням панелі інструментів

Замість тексту на кнопці можна розмістити графічне зображення, як це звичайно й робиться для економії місця. Для цього необхідно спочатку створити файл графічного зображення, а потім підключити його до кнопки панелі. Як об'єкт для зберігання списку графічних зображень можна використати елемент керування ImageList. Створимо такий список у нашому додатку. Даний елемент керування перебуває в тій же бібліотеці, що й панель інструментів, тому додаткового підключення бібліотек не потрібно. Для розміщення у формі елемента управління **ImageList** виконайте наступні дії:

1. Розмістіть в батьківській формі об'єкт **ImageList**, двічі клацнувши мишею кнопку ImageList  на панелі елементів управління.
2. У вікні властивостей задайте ім'я створеного об'єкта **imlImageListToois.**
3. Виділіть об'єкт, натисніть праву кнопку миші й виберіть у контекстному меню, що з'явилося, команду **Properties** (Властивості). Відкриється діалогове вікно Property **Pages** (Сторінки властивостей), за допомогою якого створимо список графічних зображень (рис.6.15).
4. Виберіть вкладку **Images** (Зображення).

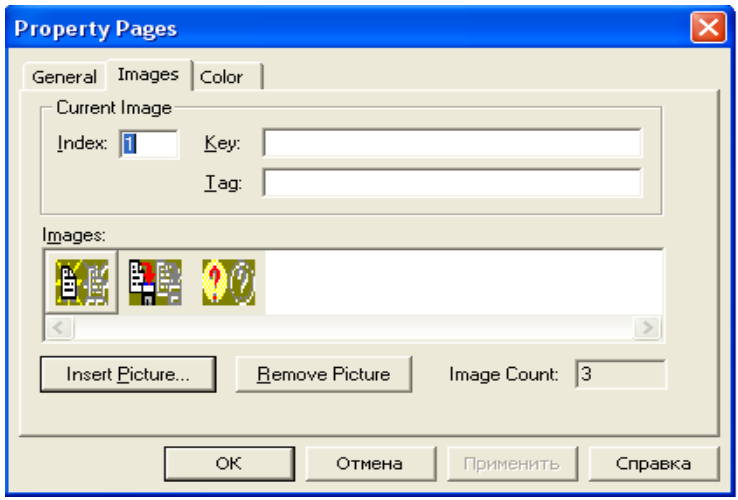


Рис. 6.15. Діалогове вікно Property Pages об'єкта ImageList

5. Для ідентифікації списку зображень уведіть у поле Key (Ключ) ім'я списку imlTools.
6. Щоб додати в список графічне зображення, натисніть кнопку **Insert Picture** (Вставити зображення). Відкриється діалогове вікно пошуку графічних файлів. Виберіть необхідний файл і натисніть кнопку **Відкрити**. Графічне зображення додається в список. Для додавання елемента, що слідує у списку знову натисніть кнопку **Insert Picture**. Завершивши формування всього списку графічних зображень, натисніть кнопку **ОК** для закриття діалогового вікна **Property Pages.**

   Для створення панелі інструментів у своєму додатку ми скористаємося графічними файлами New.bmp і Open.bmp з

каталогу Wisual Basic \Common\Grafics\Bitmaps\OffCtrlBr\Small\Color.

7. Список зображень готовий. Для підключення створеного списку зображень до панелі необхідно викликати вікно властивостей панелі на вкладці **General** (Загальні) і вибрати найменування створеного графічного списку зображень в списку, що відкривається, **ImageList** (Список зображень).

Для розміщення графічного зображення на кнопці панелі інструментів виконайте наступні дії:

1. Відкрийте діалогове вікно **Property Pages.**
2. На вкладці **General** виберіть зі списку ImageList найменування створеного нами списку зображень imlImageListTools.
3. Перейдіть на вкладку **Buttons**. Установіть в поле **Index** значення 1, указуючи, що вибирається перша кнопка.
4. Установіть в поле **Image** значення 1, указуючи, що вибирається перше зображення зі списку. На кнопці після цього з'являється необхідне зображення. Розміри кнопки автоматично встановлюються відповідно до розміру зображення.

Додаток, що містить на панелі інструментів графічну кнопку, показано на рис.6.16.

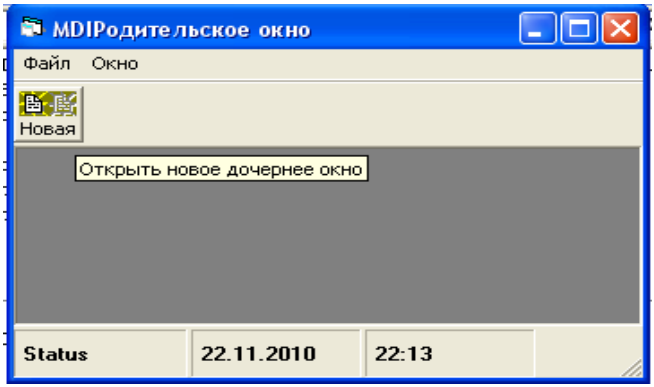


Рис. 6.16. Панель інструментів додатка містить графічну кнопку

Для створення інтервалів між кнопками використаються кнопки, для яких властивість `Style` установлена в значення `tbrSeparator`.

Елемент управління CoolBar

За допомогою елемента управління CoolBar можна створювати в додатку поліпшені панелі інструментів. Для використання цього елемента управління в додатку необхідно підключити до проекту бібліотеку Microsoft Windows Common Controls-3 6.0, скориставшись діалоговим вікном **Components** (Компоненти), що відкривають при виборі команди **Components** (Компоненти) меню **Project** (Проект).

На відміну від об'єкта ToolBar, елемент керування **CoolBar** більш універсальний і крім кнопок може містити інші елементи керування, у тому числі й панелі типу **ToolBar.** Удосконалена панель являє собою контейнер, причому він складається з маленьких контейнерів Band (Смуга), які є об'єктами й безпосередньо містять у собі всі елементи керування, що вводять в **CoolBar**.

Для створення в проекті панелі інструментів типу **CoolBar** виконаєте такі дії:

1. Підключіть до проекту бібліотеку Microsoft Windows Common Controls-3 6.0, яка містить елемент управління CoolBar.
2. Додайте в батьківську форму об'єкт CoolBar, двічі клацнувши мишею кнопку CoolBar  на панелі елементів управління.
3. Привласніть нової панелі інструментів ім'я cbrTools.
4. Виділіть об'єкт CoolBar, натисніть праву кнопку миші й виберіть у контекстному меню, що з’явилося, команду Properties (Властивості). Відкриється діалогове вікно Property Pages (рис.6.17), призначене для створення панелі типу CoolBar.
5. Використовуючи кнопку Insert Band (Вставити смугу), додайте на панель інструментів ще кілька об'єктів Band.

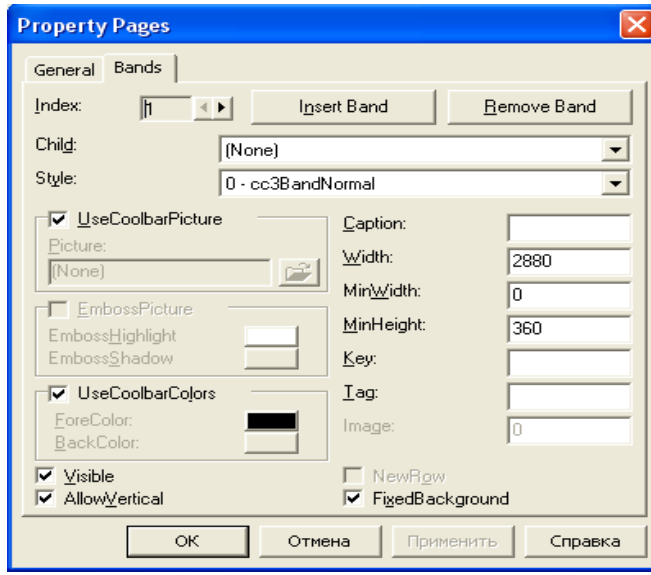


Рис. 6.17. Діалогове вікно Property Pages для об'єкта CoolBar

Тепер нову панель можна настроювати, додаючи на неї необхідні об'єкти або видаляючи їх. Додамо, наприклад, кнопку керування для виклику дочірнього вікна. Для цього виконаємо наступні дії:

1. Використовуючи кнопку **CommandButton** на панелі елементів керування, розмістіть на створюваній панелі інструментів `CoolBar` кнопку керування.
2. Скорегуйте у вікні **Properties** (Властивості) для створеної командної кнопки наступні властивості:
   - у правий стовпець властивості `Name` уведіть найменування об'єкта **cbNewCoolBar**;
   - у властивість Caption уведіть текст **Нова**;
   - у властивість `Height` кнопки введіть значення 300;
   - для властивості `Top` задайте значення 25.
3. Для створення коду, виконуваного при натисканні нової кнопки на панелі інструментів, у вікні редактора коду введіть наступні команди, що здійснюють виклик дочірньої форми:

```
Private Sub cbNewCoolBar_Click() •
Dim. frmNewForm As New frmChildMDI
frmCount = frmCount + 1
```

```
frmNewForm.Caption = "Дочірня форма+ Str(frmCount)
frmNewForm.Show
End Sub
```

4. Запустіть додаток на виконання. Працюючий додаток з панеллю інструментів типу CoolBar подано на рис.6.18.

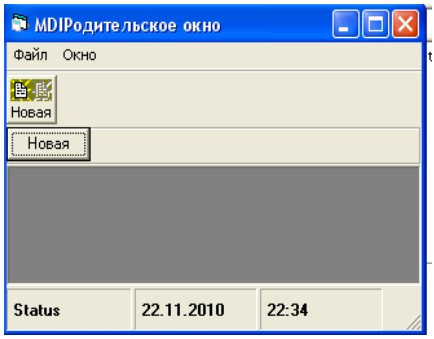


Рис. 6.18. Додаток з панеллю інструментів типу CoolBar

Діалогові вікна

В Visual Basic 6 існує спеціальний вид вікон — діалогові. У розпорядженні розроблювача є добре розвинутий інструментарій для їхнього створення. Діалогові вікна бувають двох типів — модальні й немодальні. *Модальне діалогове вікно* — це вікно, з якого не можна перейти в інше вікно, не закривши поточне. Даний вид діалогових вікон використовується для посилання повідомлень про хід роботи додатка, його настроювання або уведення яких-небудь даних, необхідних для роботи. Прикладом такого діалогового вікна в програмі Visual Basic є вікно About. Модальне діалогове вікно змушує користувача робити деякі дії або відповідати на запит додатка уведенням інформації або виконанням якої-небудь дії.

Немодальне діалогове вікно — це вікно, що дозволяє переміщати фокус на інше вікно або форму без закриття поточного вікна. Даний тип діалогових вікон використається рідко. Прикладом немодального діалогового вікна в Visual Basic є вікно Find (Пошук), що дає можливість здійснювати пошук потрібної інформації.

Найпростіші з діалогових вікон — це вікна повідомлень і вікна, які призначені для введення інформації. На додаток до них в Visual Basic 6 існує набір більш складних стандартних діалогових вікон для додатків:

- **Open** (Відкрити) — діалогове вікно для пошуку у файловій структурі потрібного файлу;
- **Save As** (Зберегти як) — для пошуку місця зберігання файлу та введення його імені;
- **Font** (Шрифт) — для вибору та установки шрифту;
- **Color** (Кольори) — для вибору колірної палітри;
- **Print** (Печатка) — для настроювання режиму печатки;
- **Help** (Довідка) — для роботи з довідковою системою додатка.

Розглянемо діалогові вікна більш докладно.

Процес виведення інформації є важливою частиною будь-якої прикладної програми. Ви завжди повинні тримати своїх користувачів в курсі того, що відбувається в програмі і в якому стані вона знаходиться. Для цього варто використовувати спеціально призначені засоби — вікна повідомлень.

Вікно повідомлень (message box) — найпростіший вигляд форми з однією або декількома стандартними кнопками, такими як `Ok` або `Cancel`, призначеними для відображення повідомлень.

Вікно повідомлень (MsgBox)

Діалогове вікно повідомлення (рис.6.19) не вимагає проектування й викликається із програми командою MsgBox або за допомогою аналогічної функції MsgBox (), що має наступний синтаксис:

```
MsgBox (prompt[, buttons] [, title] [,
  helpfile, context])
```

де:

- `prompt` – текст повідомлення в діалоговому вікні. Максимальна довжина тексту 1024 символа. У цей текст можна вставити як роздільники рядків перевод каретки Chr(13), перевод рядка Chr(l) або їхню комбінацію;
- `buttons` – числовий вираз, що задає параметри для кнопок керування й значків у діалоговому вікні, він складений з констант, що надані в табл. 6.8 і 6.9. Якщо

значення не зазначене, то за замовчуванням привласнюється значення 0;

- `title` — текст заголовка діалогового вікна;
- `helpfile` – посилання на файл довідкової системи;
- `context` – посилання на зміст у файлі довідкової системи.

Необхідно мати на увазі, що для завдання декількох параметрів кнопок і значків одночасно, варто просто скласти відповідні константи.

Для приклада введіть у командному вікні середовища проектування **Immediate** наступну команду й натисніть клавішу < Enter>:

```
MsgBox "Вітаємо Вас!", vbYesNo + vbExclamation,"Вікно
  повідомлень"
```

У відповідь одержите діалогове вікно, показане на рис.6.19

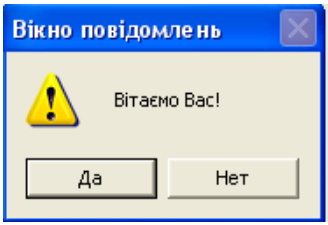


Рис. 6.19. Діалогове вікно повідомлення

Таблиця 6.8

Константи параметрів для значків діалогового вікна повідомлення

Значок	**Константа**	**Значення**	**Тип повідомлення**
	`vbExclamation`	48	Попередження
	`vbQuestion`	32	Запит
	`vbinformation`	64	Інформація
	`vbCritical`	16	Помилка

У діалогових вікнах повідомлень можна управляти відображуваними у вікні повідомлення кнопками й установкою фокуса на одній із кнопок при відкритті вікна. Для цього можна використати константи, наведені в табл. 6.9.

Таблиця 6.9

Константи параметрів наявності кнопок у вікні повідомлення й установки фокуса на кнопку

Константа	**Значення**	**Набір кнопок у діалоговому вікні**
vbOkOnly	0	ОК
vbOkCancel	1	ОК, Скасування
vbAbortRetrylgnore	2	Стоп, Повтор, Пропустити
vbYesNoCancel	3	Так, Ні, Скасування
vbYesNo	4	Так, Ні
vbRetryCancel	5	Повтор, Скасування
vbDefaultButtonl	0	Встановлює фокус на першій кнопці
vbDefaultButton2	256	Встановлює фокус на другій кнопці
vbDefaultButton3	512	Встановлює фокус на третій кнопці
vbDefaultButton4	768	Встановлює фокус на четвертій кнопці
vbApplicationModal	0	Призначає модальність додатка. Діалогове вікно буде модальним, тобто потребуючим обов'язкового закриття для переходу в інші вікна

Продовження табл. 6.9

Константа	Значення	Набір кнопок у діалоговому вікні
vbSystemModal	4096	Призначає модальність системи. Діалогове вікно буде модальним на рівні системи, тобто поки діалогове вікно не закрите, у будь-який інший додаток перейти не можна
vbMsgBoxHelpButton	16384	Додає в діалогове вікно кнопку Довідка
VbMsgBoxSetForeground	65536	Повідомляє діалогове вікно фоновим вікном
vbMsgBoxRight	524288	Вирівнює по правому краї текст у діалоговому вікні
vbMsgBoxRtlReading	1048576	Перевертає текст для читання з права уліво

Залежно від вибору кнопки діалогове вікно MsgBox повертає одне зі значень, заданих системними константами. Це необхідно для аналізу натиснутої кнопки й виконання відповідних дій у програмі. У вихідному коді для цього можна використати константи, зазначені в табл. 6.10.

Таблиця 6.10

Значення констант, що повертаються кнопками вікна повідомлення

Кнопка	Константа	Значення при натисканні на кнопку
OK	vbOk	1
Скасування	vbCancel	2
Стоп	vbAbort	3
Повтор	vbRetry	4
Пропустити	vbIgnore	5
Так	vbYes	6
Ні	vbNo	7

Діалогове вікно введення інформації (InputBox)

Досить часто в діалоговому вікні необхідно не тільки нажати кнопки вибору дії, але й ввести певну інформацію, що потім аналізується програмою. Для виконання такого роду дій в Visual Basic можна використати діалогове вікно введення інформації InputBox (рис.6.20). Функція InputBox має наступний синтаксис:

```
InputBox (prompt [, title] [, default] [,
  xpos] [, ypos] [, helpfile, context])
```

де:

`prompt` — текст повідомлення в діалоговому вікні. Максимальна довжина тексту 1024 символу. У цей текст можна вставити як роздільники рядків перевод каретки Chr(13), перевод рядка Chr(10) або їхню комбінацію;

`title` — текст заголовка діалогового вікна;

`default` — значення текстового поля введення за замовчуванням. Якщо параметр відсутній, рядок залишається порожній;

`xpos` — позиція по горизонталі лівого верхнього кута діалогового вікна щодо лівого верхнього кута екрана. За замовчуванням привласнюється значення, що відповідає середині екрана;

`ypos` — позиція по вертикалі лівого верхнього кута діалогового вікна відносно лівого верхнього кута екрана. За замовчуванням привласнюється значення, що відповідає середині екрана;

`helpfile` — посилання на файл довідкової системи;

`context` — посилання на зміст у файлі довідкової системи.

Для приклада введіть у командному вікні середовища проектування `Immediate` наступну команду:

```
strUserTest = InputBox ("Уведіть пароль", "Запуск
  додатка", "****")
```

У результаті одержите діалогове вікно, показане на рис.6.20.

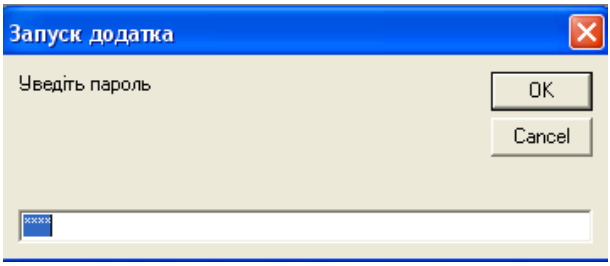


Рис. 6.20. Діалогове вікно введення

На відміну від діалогового вікна MsgBox, у вікні InputBox завжди є тільки дві кнопки керування: **ОК** і **Cancel**. Кнопка **ОК** підтверджує введення даних, кнопка **Cancel** — закриває діалогове вікно без введення даних.

Використання елемента керування CommonDialog для створення діалогових вікон

Всі діалогові вікна, перераховані на початку розділу "Діалогові вікна", можна створити за допомогою елемента керування CommonDialog. Перш ніж його використати, необхідно підключити до проекту бібліотеку Microsoft Common Dialogs Control 6.0 через діалогове вікно **Components** (Компоненти) середовища проектування.

Для виклику діалогових вікон необхідно використати відповідні кожному із цих вікон методи елемента керування CommonDialog, зазначені в табл. 6.11.

Таблиця 6.11

Методи елемента керування CommonDialog

Метод	**Опис**
ShowOpen	Викликає діалогове вікно, використовуване для відкриття файлу
ShowSave	Викликає діалогове вікно, використовуване для збереження файлу
ShowColor	Викликає діалогове вікно настроювання колірної палітри

Продовження табл. 6.11

Метод	Опис
ShowFont	Викликає діалогове вікно настроювання шрифтів тексту
ShowPrinter	Викликає діалогове вікно настроювання печатки й діалогове вікно печатки
ShowHelp	Підключає довідкову систему в стилі Windows

Для вивчення діалогових вікон, утворених за допомогою елемента керування CommonDialog, створимо невеликий проект. Виконайте наступні дії:

1. Створіть новий стандартний проект.
2. Привласніть проекту ім'я **MySmallProject.** Після перейменування проекту ця команда прийме вид **MySmallProject Properties** (Властивості MySmallProject).
3. Підключіть до проекту бібліотеку Microsoft Common Dialog Control 6.0. Для цього в меню **Project** (Проект) виберіть команду **Components,** у діалоговому вікні, що відкрилися, встановіть прапорець, розташований поруч із назвою цієї бібліотеки.
4. Задайте найменування форми **Проекту FormForControlCommDialog**.
5. Використовуючи властивість **Caption** форми, уведіть заголовок вікна **Форма для перевірки діалогів CommonDialog**.
6. Додайте у форму елемент керування CommonDialog, двічі клацнувши мишею кнопку **CommonDialog**  на панелі елементів керування. Якщо ця панель відсутня на екрані, то в меню **View** (Вид) виберіть команду **ToolBox** (Панель інструментів).
7. Привласніть елементу керування **CommonDialog** найменування **cdMyDialog.** I
8. Додайте у форму кнопку управління типу CommandButton і назвіть її. cbControl. Ця кнопка буде використатися нами для виклику діалогового вікна по події Click.
9. Використовуючи властивість Caption, уведіть назву кнопки **Перевірка діалогів CommonDialog.**

Отриманий додаток представлений на рис.6.21. Цей проект, який містить елемент керування CommonDialog, будемо використати для створення діалогових вікон і знайомства з ними, розміщаючи в

події Click кнопки cbControl код настроювання параметрів необхідного діалогового вікна й виклику відповідного йому методу.

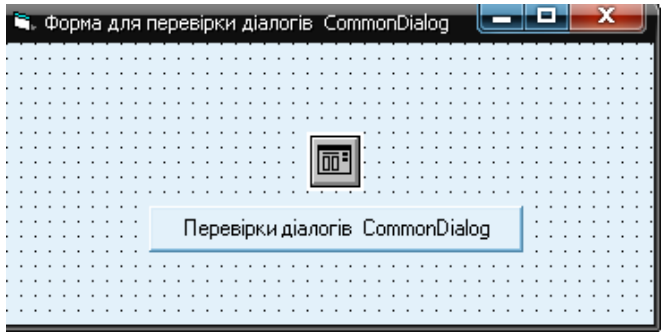


Рис.6.21. Проект MySmallProject для перевірки діалогових вікон, створюваних за допомогою елемента керування CommDialog

Діалогове вікно відкриття файлу

Діалогове вікно відкриття файлу призначено для пошуку файлів, використовуваних у програмі. Ім'я обраного файлу або списку файлів вертається у властивості FileName об'єкта CommonDialog. Для виклику діалогового вікна відкриття файлу (рис. 6.22) необхідно в події Click кнопки cbControl додатка MySmallProject ввести наступний код:

```
Private Sub cbControl_Click()
cdlMyDialog.ShowOpen
End Sub
```

Для введення коду досить двічі клацнути на об'єкті cbControl лівою кнопкою миші. При цьому відкривається редактор коду із шаблоном коду для події Click цієї кнопки.

Значення для властивості cdiMyDiaiog. Flags зазначені в табл. 6.12.

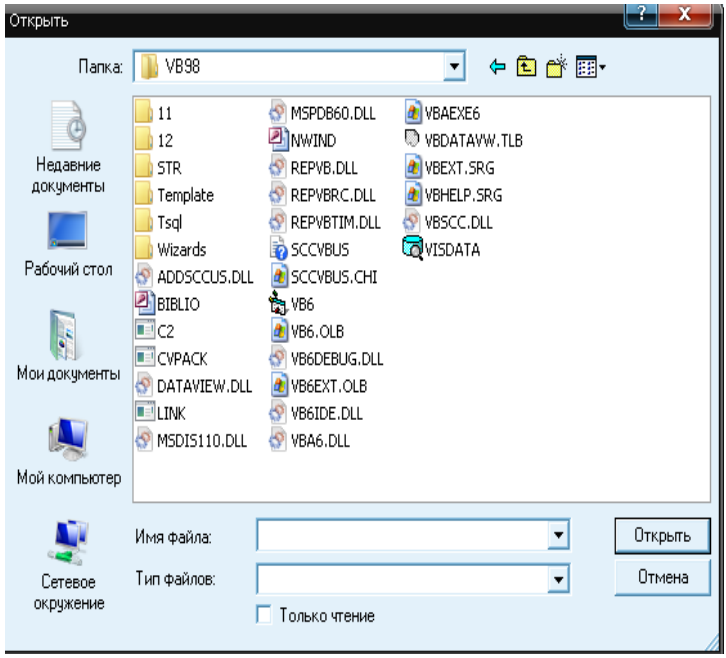


Рис. 6.22. Діалогове вікно відкриття файлу, створене за допомогою об'єкта CommonDialog

Таблиця 6.12

Значення для методу Flags діалогового вікна відкриття файлу

Константа	Значення	Опис
`cdlOFNAllowMultiselect`	&H200	Встановлює можливість множинного вибору файлів у діалоговому вікні. Імена файлів, що повертаються, перебувають у властивості `FileName` елемента керування `CommonDialog` і розділені пробілами

Продовження табл. 6.12

Константа	Значення	Опис
cdlOFNCreatePrompt	&H2000	Установлює для діалогового вікна запит на підтвердження створення нового файлу, якщо він не існує
cdlOFNExplorer	&H80000	Призначає діалоговому вікну стиль провідника
cdlOFNExtensionDifferent	&H400	Повідомляє, що розширення файлу відрізняється від установленого за замовчуванням у властивості DefaultExt
cdlOFNFileMustExist	&H1000	Встановлює можливість введення в діалоговому вікні тільки імен існуючих файлів. При введенні неіснуючого файлу видається повідомлення про помилку
cdlOFNHelpButton	&H10	Вказує на необхідність розміщення в діалоговому вікні кнопки Довідка
cdlOFNHideReadOnly	&H4	Файли тільки для читання не відображаються
cdlOFNLongNames	&H200000	Дозволяє використання довгих імен файлів
cdlOFNNoChangeDir	&H8	Призначає папку, що відкриває за замовчуванням при запуску діалогового вікна
cdlOFNNoLongNames	&H40000	Забороняє використання довгих імен файлів

Продовження табл. 6.12

Константа	Значення	Опис
cdlOFNNoReadOnlyReturn	&H8000	Указує, що виведені в діалоговому вікні файли не повинні бути призначені тільки для читання й не перебувають у каталозі, захищеному від запису (Write-protected)
cdlOFNNoValidate	&H100	Повідомляє про уведення неприпустимих символів в імені файлу
cdlOFNOverwritePrompt	&H2	Встановлює для діалогового вікна запит на підтвердження перезапису існуючого файлу
cdlOFNPathMustExist	&H800	Встановлює вимогу вказівки повного шляху до файлу
cdlOFNReadOnly	&H1	Задає перевірку прапорця Тільки читання
CdlOFNShareAware	&H4000	Задає ігнорування помилки типу блокування, зайнятості файлу (Sharing violation errors)

Діалогове вікно збереження файлу

Для пошуку файлу, у якому будуть збережені дані із програми, використовується діалогове вікно збереження файлу (рис.6.23). Це діалогове вікно викликається так само, як вікно відкриття файлу. Для його створення в події Click кнопки cbControl додатка **MySmaHProject** необхідно замінити код на наступний:

```
Private Sub cbControl_Click()
cdlMyDialog.ShowSave
End Sub
```

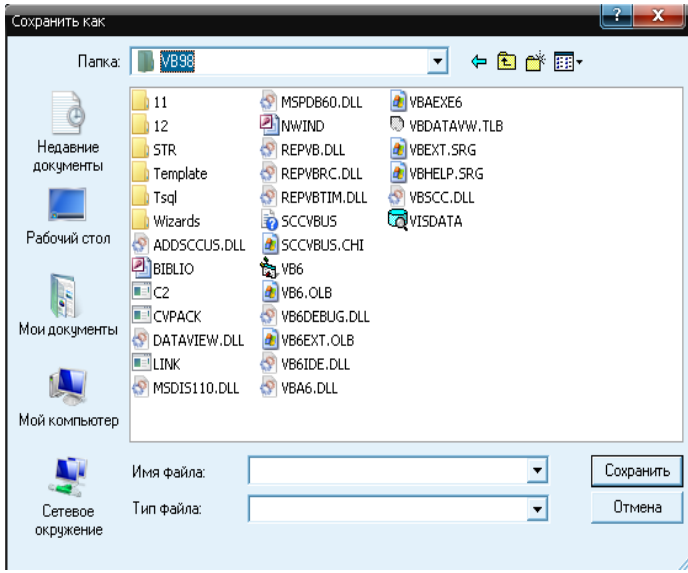


Рис. 6.23. Діалогове вікно збереження файлу, створене за допомогою об'єкта CommonDialog

Як видно з рисунка, це діалогове вікно аналогічно вікну, призначеному для відкриття файлу. Прапори цього діалогового вікна перераховані в табл. 6.12.

Діалогове вікно настроювання колірної палітри

Для настроювання кольорів поля форми й розташованих у формі елементів можна використати діалогове вікно настроювання колірної палітри (рис.6.24).

Для виклику цього діалогового вікна необхідно замінити код у події Click кнопки cbControl додатка MySmallProject на наступний:

```
Private Sub cbControl_Click()
cdlMyDialog.ShowColor
End Sub
```

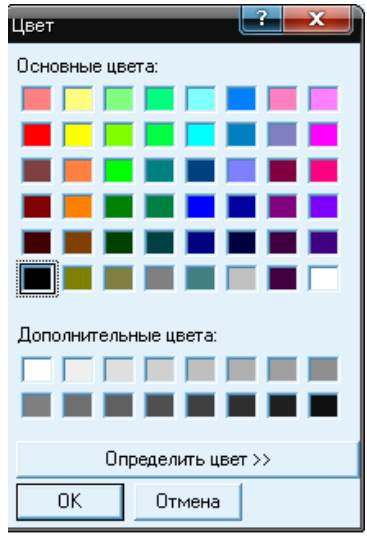


Рис.6.24. Діалогове вікно настроювання колірної палітри

Константи, використовувані для настроювання діалогового вікна вибору колірної палітри, перераховані в табл. 6.13. Розширене діалогове вікно настроювання колірної палітри можна створити за допомогою об'єкта CommonDialog (рис.6.25).

Таблиця 6.13

Значення для методу Flags діалогового вікна колірної палітри

Константа	**Значення**	**Опис**
`cdlCCFullOpen`	&H2	Поміщає в діалогове вікно додаткову кнопку Додати в набір і колірне поле з маркером і лінійкою призначення відтінку для додавання квітів у поля Додаткові кольори діалогового вікна колірної палітри (рис.6.25)

Продовження табл. 6.13

Константа	Значення	Опис
cdlCCShowHelp	&H8	Додає в діалогове вікно кнопку Довідка
cdlCCPreventFullOpen	&H4	Приховує кнопку Додати в набір
cdlCCRGBInit	&H1	Відновлює в діалоговому вікні вихідний набір кольорів

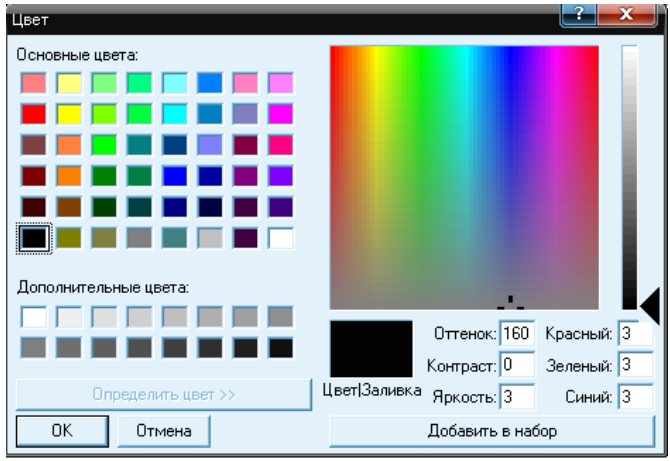


Рис. 6.25. Розширене діалогове вікно настроювання колірної палітри, створене за допомогою об'єкта CoitimonDialog із прапором cdlCCFullOpen

Діалогове вікно настроювання шрифтів тексту

Для виклику діалогового вікна настроювання шрифтів (рис.6.26) необхідно в подію Click кнопки cbControl увести наступний код:

```
Private Sub cbControl_Click()
cdlMyDialog.Flags = cdlCFBoth + cdlCFEffects
cdlMyDialog.ShowFont
End Sub
```

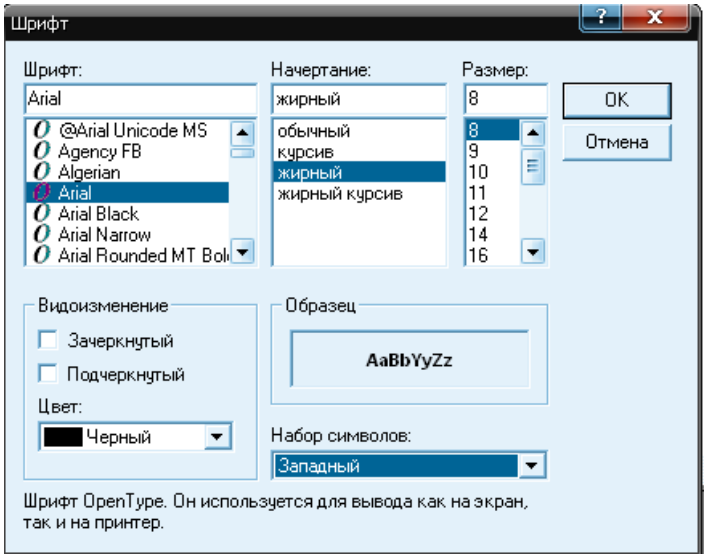


Рис. 6.26. Діалогове вікно настроювання шрифту, відкрите з використанням об'єкта CommonDialog

Прапори діалогового вікна настроювання шрифтів перераховані в табл. 6.14.

Таблиця 6.14

Значення для методу Flags діалогового вікна настроювання шрифтів

Константа	**Значення**	**Опис**
`cdlCFANSIOnly`	&H400	Призначає вибір тільки із системних наборів шрифтів Windows
`cdlCFApply`	&H200	Додає в діалогове вікно кнопку **Застосувати**
`cdlCFBoth`	&H3	Призначає використання екранних шрифтів принтера, зазначеного у властивості hDC

Продовження табл. 6.14

Константа	**Значення**	**Опис**
CdlCFEffects	&H100	Вказує на необхідність розміщення в діалоговому вікні елементів керування, що задають ефект підкреслення, закреслювання та кольори
cdlCFForceFontExist	&H10000	Призначає вивід попередження про помилку при виборі неіснуючого шрифту
cdlCFHelpButton	&H4	Додає в діалогове вікно кнопку **Довідка**
cdlCFLimitSize	&H2000	Призначає вибір розмірів шрифтів в інтервалі, зазначеному у властивостях Min і Max
cdlCFNoFaceSel	&H80000	Відключає вибір найменування шрифту
cdlCFNoSizeSel	&H1000	Відключає вибір розміру шрифту
cdlCFNoSimulations	&H200000	Відключає в діалоговому вікні графічні шрифти
cdlCFNoStyleSel	&H100000	Відключає вибір стилю шрифту
cdlCFNoVectorFonts	&H800	Відключає вибір векторних шрифтів
cdlCFPrinterFonts	&H2	Призначає вибір тільки шрифтів принтера, що зазначений у властивості hDC
cdlCFScreenFonts	&H1	Призначає вибір тільки екранних шрифтів системи
cdlCFTTOnly	&H40000	Призначає тільки вибір шрифтів типу True Typi
CdlCFWYSIWYG	&H8000	Призначає вибір тільки тих шрифтів, які підходять одночасно й для принтера і для екрана. При цьому повинні додатково використатися прапори cdlCFBoth і й cdlCFScalableOnly

Діалогове вікно печатки

Діалогове вікно печатки можна викликати за допомогою елемента керування CommonDialog, замінивши код у події Click кнопки з найменуванням cbControl на наступний:

```
Private Sub cbControl_Click()
cdlMyDialog.ShowPrinter
End Sub
```

Відкрите діалогове вікно показане на рис.6.27.

Якщо потрібно попередньо настроювати печатку, то необхідно встановити для властивості Flags значення **cdlPDPrintSetup**. У цьому випадку код буде виглядати в такий спосіб:

```
Private Sub cbControl_Click()
cdlMyDialog.Flags = cdlPDPrintSetup
cdlMyDialog.ShowPrinter
End Sub
```

Запустивши додаток **MySmallProject,** ви одержите діалогове вікно, показане на рис.6.28.

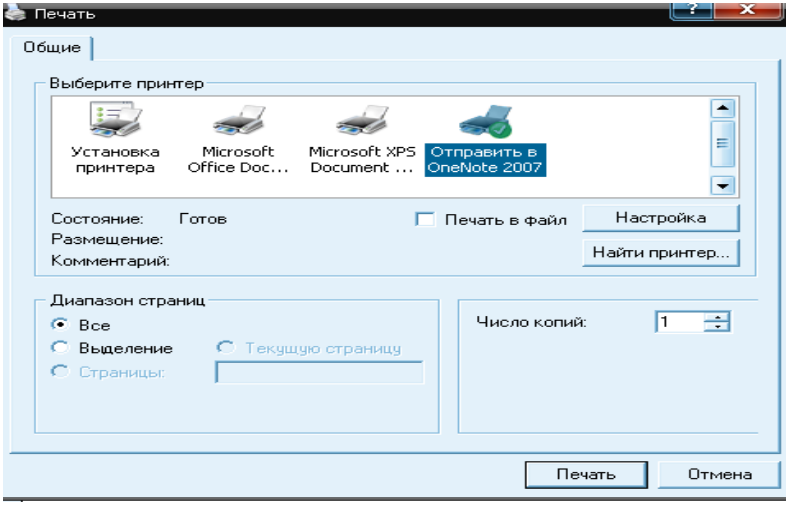


Рис. 6.27. Діалогове вікно печатки, відкрите з використанням обьекта CommonDialog

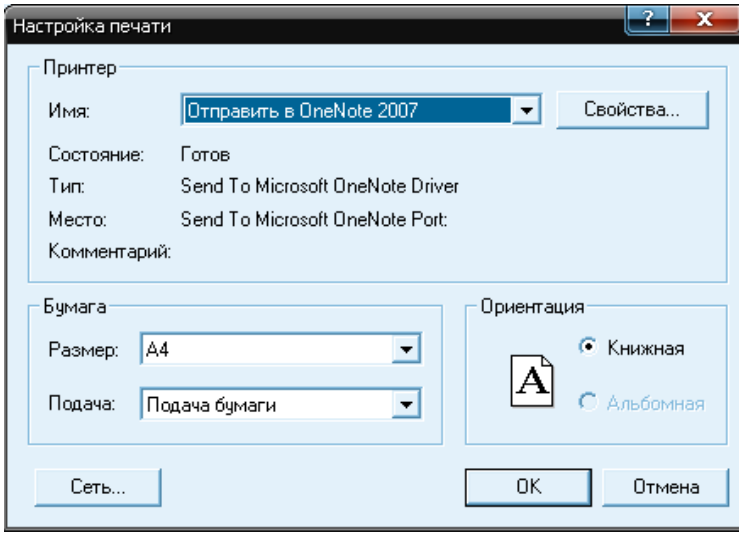


Рис. 6.28. Діалогове вікно настроювання печатки, створене за допомогою об'єкта CommonDialog

Припустимі прапори діалогових вікон печатки й настроювання печатки перераховані в табл. 6.15.

Таблиця 6.15

Значення для методу Flags діалогового вікна печатки

Константа	**Значення**	**Опис**
`cdlPDAHPages`	&H0	Повертає або встановлює стан перемикача печатки всіх сторінок
`cdlPDDisablePrintToFile`	&H80000	Робить недоступним прапорець діалогового вікна **Печатка у файл**
`cdlPDHelpButton`	&H800	Додає в діалогове вікно кнопку **Довідка**

Продовження табл. 6.15

Константа	Значення	Опис
cdlPDHidePrintToFile	&H100000	Приховує прапорець діалогового вікна **Печатка у файл**
cdlPDNoPageNums	&H8	Приховує елементи керування в групі **Друкувати**
cdlPDNoWarning	&H80	Скасовує попередження про відсутність принтера за замовчуванням
cdlPDPageNums	&H2	Робить недоступними елементи керування, розташовані в області **Друкувати**
cdlPDPrintSetup	&H40	Виводить діалогове вікно настроювання печатки
cdlPDPrintToFile	&H20	Повертає або встановлює стан прапорця **Печатка у файл**
cdlPDReturnDefault	&H400	Повертає ім'я принтера, використовуване за замовчуванням
cdlPDUseDevModeCopies	&H40000	Якщо драйвер принтера не підтримує створення копій, робить недоступним лічильник **Число копій**. Якщо копіювання підтримується, вказує, що номер копії зберігається у властивості **Copies**

Довідкова система в стилі Windows

Прапори діалогового вікна довідкової системи перераховані в табл. 6.16. Для використання цього діалогового вікна необхідно створити довідкову систему для додатка (файл довідки).

Таблиця 6.16

Значення для методу Flags діалогового вікна довідкової системи

Константа	Значення	Опис
`cdlHelpCommadHelp`	&H102	Викликає довідкову систему окремою командою
`cdlHelpContents`	&H3	Викликає зміст довідкової системи
`cdlHelpContext`	&H1	Викликає окрему тему довідкової системи
`cdlHelpContextPopup`	&H8	Викликає тему довідки по індексу
`cdlHelpIndex`	&H3	Викликає покажчик довідкової системи
`cdlHelpKey`	&H101	Викликає довідкову систему по ключовому слову

Рядок стану

Рядок стану — це спеціальний елемент вікна, що складається з кількох панелей для відображення поточної інформації про стан і режим роботи додатка. На рис.6.29 показаний рядок стану, що відображає стан додатка, дату й поточний системний час. Цей елемент інтерфейсу звичайно розміщається в нижній частині батьківського вікна додатка, якщо не потрібно спеціально встановити його в іншім місці вікна. Таке положення рядка стану є стандартним.

Рис. 6.29. Рядок стану додатка

Для додавання рядка стану у форму використовується елемент управління StatusBar. Щоб цей об'єкт можна було використати в додатку, необхідно у вікні Components (Компоненти) підключити до обраного проекту бібліотеку Microsoft Window Common Control 6.0. Після підключення бібліотеки елемент керування StatusBar з'являється на панелі елементів управління середовища проектування і його можна додати у форму стандартним способом, як і всі інші елементи керування.

Рядок стану складається з набору панелей, кожна з яких є об'єктом і має наступні основні властивості:

- Alignment — задає вирівнювання тексту на панелі рядка стану;
- Bevel — встановлює затінення для додання об'ємності панелям;
- Minwidth — визначає мінімальний розмір панелей рядка стану;
- Picture — задає графічне зображення, що буде розміщено на панелі;
- Style — визначає тип панелі. Може приймати значення, зазначені в табл. 6.17;
- ext — задає текст, розташовуваний на панелі. Як правило, цей текст формується програмно;
- TooiTip — задає текст підказки для панелі. Виводиться при затримці покажчика миші на панелі.

Таблиця 6.17

Значення властивості `Style` панелей рядка стану

Стиль	**Значення**	**Опис**
`sbrText`	0	Дає можливість відображати текст або зображення у властивостях панелі `Text` і `Picture`, відповідно
`sbrCaps`	1	Відображає стан клавіші `<Caps Lock>`. Якщо ця клавіша натиснута, текст `CAPS` на панелі яскравий, якщо віджато — затінений

Продовження табл. 6.17

Стиль	Значення	Опис
sbrNum	2	Відображає стан клавіші <Num Lock>
sbrins	3	Відображає стан клавіші <Ins>
sbrScrl	4	Відображає стан клавіші <Scroll Lock>
sbrTime	5	Виводить поточний час
sbrDate	6	Виводить поточну дату

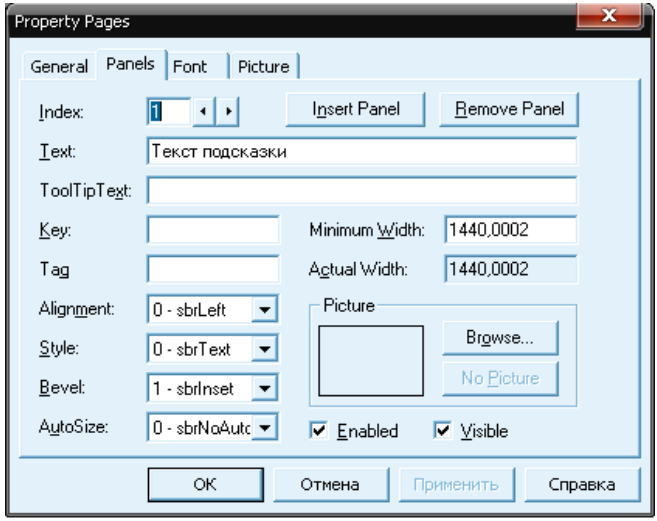


Рис. 6.30. Діалогове вікно Property Pages для рядка стану

Настроювання рядка стану й керування ним виконується за допомогою вікна властивостей Property Pages (Сторінка властивостей) цього елемента керування, що відкривається при виборі команди Properties (Властивості) контекстного меню об'єкта (рис.6.30).

Розглянемо створення рядка стану на прикладі нашого додатка MyMDIApp. Відкрийте додаток і виконаєте наступні дії:

1. Додайте у форму елемент керування **StatusBar**, двічі клацнувши мишею кнопку **StatusBar**  на панелі елементів керування.

2. Після появи у формі рядка стану привласніть йому ім'я **sbStatusBa**r.
3. Встановіть курсор на рядок стану, натисніть праву кнопку миші й виберіть із контекстного меню, що з'явився, **команду** Properties (Властивості). Відкриється діалогове вікно **Property Pages** для настроювання рядка стану.
4. С допомогою кнопки **Insert Panel** (Вставити панель), додайте в рядок стану ще дві панелі.
5. Використовуючи лічильник **Index** (Індекс), перейдіть до настроювання панелі 1. Панелі рядка стану проектуються незалежно один від одного, переключення з однієї на іншу виконується за допомогою лічильника **Index.**
6. Першу панель будемо використати для відображення текстової інформації. Встановіть для неї властивість **Style** у значення **0 — sbrText**, потім у властивість Text цієї панелі введіть значення ***Текст підказки.***
7. Перейдіть до настроювання другої панелі, установивши для лічильника **Index** значення 2.
8. Друга панель буде використатися для відображення поточної дати. Встановіть для властивості **Style** значення **0 — sbrDate.**
9. Перейдіть до настроювання третьої панелі, встановивши для лічильника Index значення 3.
10. Встановіть для властивості Style значення **0 -sbrCaps**. Це буде панель, що відображає стан клавіші <Caps Lock>.
11. Запустіть отриманий додаток на виконання командою Start меню Run. Створений додаток представлений на рис.6.31.

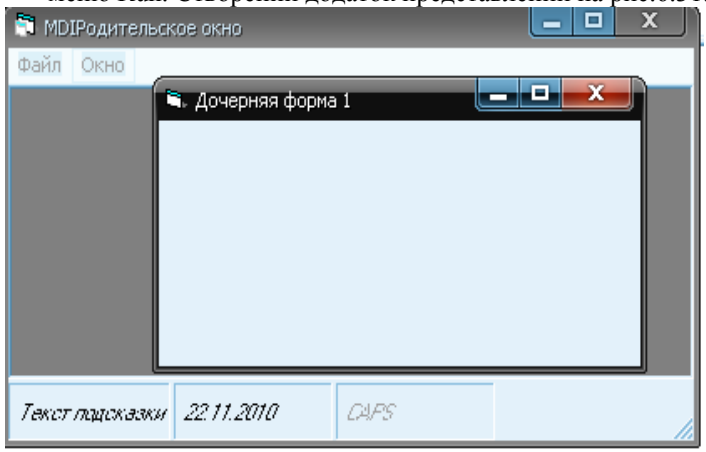


Рис. 6.31. Робота додатка, що містить рядок стану

Резюме

В розділі розглянуто типи інтерфейсів та основні конструкційні елементи, подано технологію їхнього створення за допомогою інструментальних засобів Visual Basic, Особливо підкреслено основні принципи, які слід враховувати при проектуванні інтерфейсу додатка, але вони не є догмою, тому що згодом, у процесі накопичення практичного досвіду, виробляються свої оптимальні принципи побудови інтерфейсу.

Контрольні запитання та завдання

1. Якими принципами слід керуватися при розробці інтерфейсу ?
2. Перелічіть та охарактеризуйте основні типи інтерфейсу.
3. Які елементи входять до складу MDI-інтерфейсу ?
4. Охарактеризуйте основні властивості батьківського вікна інтерфейсу типу MDI.
5. Перелічіть основні події пов’язані з формою.
6. Перелічіть особливості дочірніх вікон.
7. Які елементи містить інтерфейс додатка типу провідник?
8. Які інструментальні засоби використовуються для проектування панелей інструментів в Visual Basic?
9. Які типи діалогових вікон існують у Visual Basic ?
10. Дати визначення модального та немодального діалогового вікна.
11. Напишіть програму для стандартного шаблону меню File, а також програмний код, який здійснює виконання команд Open, Save, Save as та інших.
12. Розробіть панель інструментів з трьома кнопками, яку користувачі зможуть пересувати до верхньої, нижньої, правої або лівої межі вікна.
13. Розробіть програмний код, який поточну дату та час буде відображати у рядок стану.
14. Розробіть програму, яка за допомогою смуг прокручування буде управляти типом контуру і стилем обрамлення елементу управління Shape.
15. Розробіть програмний код, який при введенні користувачем паролю буде контролювати проміжок часу.

Розділ VII
СТАНДАРТНІ ЄЛЕМЕНТИ УПРАВЛІННЯ

Використання стандартних елементів управління

Більшість додатків, створених в Visual Basic, працюють в інтерактивному режимі. На екран виводиться інформація, призначена для користувача програми, і очікується його відповідна реакція у вигляді уведення даних або команд. Інтерактивний додаток в Visual Basic створюється на базі форми, що є, як правило, основним вікном інтерфейсу. Елементи управління форми забезпечують взаємодію з користувачами. Добре знання найбільш часто використовуваних елементів управління Visual Basic, їхніх основних властивостей необхідно розробнику програмного забезпечення. В цьому розділі ми познайомимся з основними елементами управління.

У постачання Visual Basic включений досить великий набір елементів управління, за допомогою яких можна вирішити більшість типових задач. Деякі з елементів управління з'являються на екрані відразу після запуску Visual Basic. Вони розташовані в блоці инструметів і використовуються частіше за все при створенні програм. Такі елементи управління називаються *внутрішніми.* Перелік стандартних елементи управління поданий у табл.7.1.

Таблиця 7.1

Стандартні елементи управління

Елемент	Назва	Призначення	Опис
	`Label`	Напис	Служить для відображення тексту, який користувач не може змінити
	`TextBox`	Текстове поле	Служить для відображення, введення і редагування тексту

Продовження табл. 7.1

Елемент	Назва	Призначення	Опис
	CommandButton	Кнопка	Служить для ініціації деяких дій програми після клацання на ній кнопкою миші
	CheckBox	Прапорець	Служить для установки і відображення логічних (типуТак/Ні) параметрів програми
	OptionBox	Перемикач	Служить для вибору одного параметра з кількох представлених в групі
	ComboBox	Поле із списком	Дозволяє користувачеві вибрати одне значення із списку, а також вводити нове значення в текстове поле
	ListBox	Список	Служить для вибору елемента із списку
	Timer	Таймер	Дозволяє програмам виконувати різні дії з таймером
	Image	Малюнок	Служить для відображення малюнка у формі
	Frame	Група	Служить контейнером для інших елементів управління. Крім того, він дозволяє згрупувати декілька елементів управління
	PictureBox	Графічний фрейм	Фрейм для відображення графічних зображень. Крім того, він може служити контейнером для інших елементів управління

Відображення тексту в полях типа Label

Основним призначенням елементів управління типу Label є відображення тексту на екрані. Найчастіше вони використовуються для підпису розташованих поряд з ними текстових полів. Для використання даного елементу управління необхідно розташувати його у формі, дати назву і ввести текст напису у властивість Caption.

І хоча користувач не може змінити текст, що знаходиться в полях типа Label, проте, з точки зору програміста, вони є повноцінними елементами управління, оскільки для них передбачений певний набір властивостей і подій. Все це дозволяє прямо з програми управляти зовнішнім виглядом і вмістом полів типа Label.

Розглянемо більш докладно його властивості. Текст мітки задається властивістю Caption. Він може бути встановлений у вікні Properties або програмно. Шрифт текстової інформації визначається властивістю Font (Шрифт). Для вибору шрифту у вікні властивостей встановіть курсор у дану властивість і натисніть кнопку із трьома крапками в правому стовпці властивості. Відкривається діалогове вікно **Вибір шрифту** (рис. 7.1), що містить три списки, що дозволяють указати найменування, накреслення й розмір шрифту.

Використовуючи властивості ForeColor і Backcolor, можна задати кольори текстової інформації й кольори тла елемента керування. Властивість BorderStyle (Стиль рамки) визначає тип обрамлення навколо об'єкта Label, дозволяючи оформити напис у вигляді текстового поля. Для цього замість використовуваного за замовчуванням значення **None** необхідно вибрати для властивості значення **Fixed Single.**

Властивість Appearance дозволяє додати тексту деяку об'ємність.

Властивість Alignment (Вирівнювання) визначає вирівнювання тексту в елементі керування по правому, лівому краї або по ширині. Якщо інформація в об'єкті розміщена на декількох рядках, то вирівнювання здійснюється на кожному рядку.

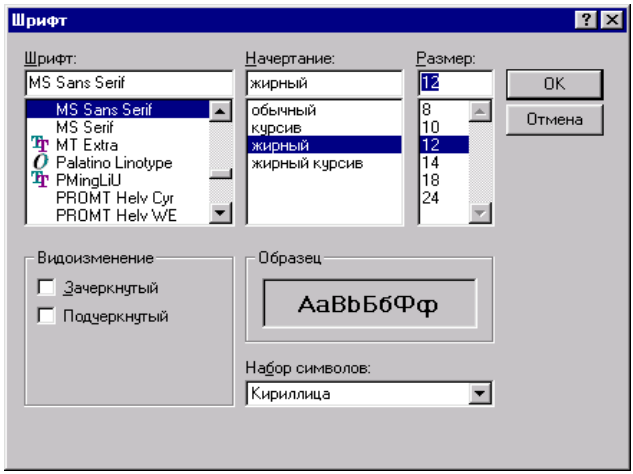


Рис. 7.1. Діалогове вікно Вибір шрифту

Оформлення написів в елементі керування Label міняється залежно від значення властивостей Alignment, Appearance И BorderStyle (рис. 7.2).

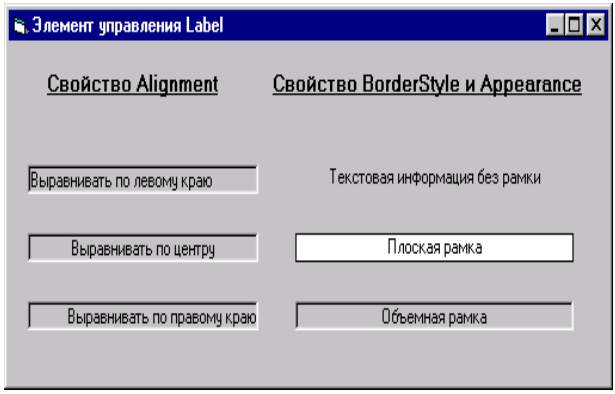


Рис. 7.2. Оформлення написів

Текст, що задається об'єктом `Label`, може мати досить великий розмір і займати кілька рядків. Максимальна кількість його

символів — 65528. Задати розміри напису можна за допомогою наступних засобів:

- миші;
- клавіші-стрільці при натиснутій клавіші <Shift>;
- властивості Height (Висота) і Width (Ширина);
- властивості AutoSize і Wordwrap.

Перші три способи завдання розміру текстового об'єкта зручні в тих випадках, коли він має невеликий конкретно заданий розмір. Але хоча користувач додатка не може змінювати текст, відображуваний за допомогою об'єкта Label, його можна змінювати програмно. У цьому випадку точний розмір текстового об'єкта заздалегідь не відомий, і для завдання його розміру зручно використати властивості Autosize і Wordwrap. Розглянемо їх більш докладно.

Властивість *AutoSize*

Властивість Autosize може мати два значення. При установці значення False розмір об'єкта залишається постійним і не залежить від довжини заданого властивістю Caption тексту. Якщо довжина тексту перевищує довжину об'єкта Label, частина інформації, що не помістилася в об'єкт, буде не видна. При установці для властивості Autosize значення **True** довжина об'єкта встановлюється таким чином, щоб у ньому помістилася текстова інформація задає, що властивістю Caption.

Властивість *Wordwrap*

Властивість Wordwrap аналогічно властивості Autosize, але в цьому випадку змінюється висота об'єкта, а ширина залишається незмінної. При установці для даної властивості значення **False** розмір об'єкта Label може змінюватися тільки в горизонтальному напрямку. Якщо значення властивості Wordwrap дорівнює **True,** розміри об'єкта Label будуть збільшуватися у вертикальному напрямку вниз, поки не поміститься весь текст. При цьому здійснюється автоматичний перенос слів. У табл. 7.2 зазначені

розміри текстового об'єкта залежно від значення властивостей `Wordwrap` і `Autosize`.

Таблиця 7.2

Значення властивостей `Wordwrap` і `Autosize`

Значення властивостей	**Розмір об'єкта Label**
`WordWrap`= False `AutoSize`= False	Об'єкт Label має спочатку задані для нього розміри
`WordWrap`= True `AutoSize`= False	Об'єкт Label має спочатку задані для нього розміри
`Wordwraps`= False `AutoSize`= True	Якщо інформація, задана властивістю Caption, не міститься в об'єкті, змінюється розмір поля в горизонтальному напрямку при незмінній висоті
`Wordwraps`= True `AutoSize`= True	Якщо інформація, задана властивістю Caption, не міститься в об'єкті, змінюється розмір поля у вертикальному напрямку при незмінній ширині

Властивості `AutoSize` і `Wordwrap` зручно використати для написів, розміри яких можуть змінюватися в процесі виконання форми. Щоб при зміні розміру об'єкта у вертикальному напрямку не змінилася його ширина, необхідно спочатку встановити значення **True** для властивості `Wordwrap`, і лише потім аналогічне значення для властивості `AutoSize`.

Властивість *UseMnemonic*

Властивість UseMnemonic елемента керування Label дозволяє визначити, як буде інтерпретуватися символ амперсанда (&), розміщений у властивості Caption. Якщо встановлено значення **True,** то амперсанд із тексту віддаляється, а символ, перед яким він розташований, підкреслюється. Ця можливість застосовується для визначення клавіш швидкого доступу. При використанні мітки як клавіши швидкого доступу користувач може комбінацією клавіш <Alt>+<підкреслена клавіша в мітці> установлювати фокус на елемент керування, що слідує по порядку для клавіш <Tab> за міткою.

Розглянемо такий приклад. У формі розташовано кілька елементів керування. Серед них є текстове поле, у яке ви хочете перейти натисканням клавіші швидкого доступу. У цьому випадку виконаєте наступні дії:

1. Створіть мітку для текстового поля.
2. Відкрийте вікно властивостей **Properties** мітки.
3. Виділите властивість `Caption` і в текст мітки перед символом, що ви хочете використати як клавіша швидкого доступу, помістите символ амперсанда.
4. За допомогою властивості TabIndex мітки й текстового поля, встановіть для мітки значення на одиницю меньше, ніж у текстового поля.

Якщо елемент керування має властивість `Caption` (наприклад, елемент `CheckBox`), то для завдання клавіші швидкого доступу необхідно використати його власну властивість.

Щоб поекспериментувати з властивість `Caption`, створіть новий стандартний проект Visual Basic і помістить у форму елемент управління типа `Label`. Двічі клацніть на ньому і введіть у вікно кода текст приведеної нижче процедури обробки події `Click`.

```
Private Sub Label1_Click()
Label1.Caption = "Current time: " & Time$
End Sub
```

Кожного разу, як тільки ви клацнете мишею на елементі управління типа `Label`, у ньому з'явиться поточний час.

Привласнюючи текст властивості `Caption`, можна змусити перейти його на новий рядок. Для цього потрібно включити в текст спеціальні символи управління — так звані символи повернення каретки і переходу на новий рядок. У Visual Basic для переходу на новий рядок потрібно помістити в текст спеціальні ASCII символи — 13 (переклад рядка) і 10 (повернення каретки). Для цієї мети передбачена спеціальна константа vbCrLf. Нижче наведений приклад установки властивості `Caption`, коли текст на екрані відображуватиметься в два рядки:

```
Labell.Caption = "Перший рядок" & vbCrLf &
  "Другий рядок"
```

Введення тексту в текстові поля

Елемент управління TextBox, розміщений у формі, служить для уведення користувачем інформації під час роботи додатка або відображення інформації, що задається властивістю Text програмно або при розробці. Об'єкт TextBox, так само як і мітка, відрізняється більшим набором властивостей. Розглянемо їх більш докладно.

Властивості, що визначають оформлення тексту

Для завдання стилю рамки текстового поля застосовується властивість `Borderstyle` (Стиль рамки). Вона містить два значення. За замовчуванням використається значення **Fixed Single,** при якому поле виділене рамкою. При установці значення **None** рамка навколо поля відсутня. Властивість Appearance дозволяє додати об'ємність текстовому полю, що має рамку.

Властивості `Backcolor i ForeColor` дозволяють відповідно задати кольори поля й кольори тексту, розташовуваного в елементі управління `TextBox`.

За допомогою властивості `Alignment` (Вирівнювання) можна задати варіант вирівнювання інформації в полі: по центрі, по лівому або правому краю.

Для завдання найменування, розміру й накреслення шрифту, відображуваного в текстовому полі, використається властивість `Font` (Шрифт).

Багатострочні текстові поля

За замовчуванням передбачається, що текстове поле служить для уведення одного рядка тексту. Властивості MultiLine і ScrollBar елемента управління TextBox дозволяють настроїти об'єкт таким чином, що він буде використатися для введення декількох рядків або навіть великого блоку текстової інформації (табл. 7.3).

Таблиця 7.3

Призначення властивостей MultiLine і ScrollBar

Властивість	**Призначення**
`MultiLine`	Визначає спосіб відображення поля. При встановленому значенні `True` текст автоматично переноситься за словами на кілька рядків. При уведенні інформації в поле для переходу на новий рядок необхідно використати клавішу <Enter>
`ScrollBar`	Дана властивість використається, якщо властивість `MultiLine` має значення `True` і розмір поля не дозволяє відобразити заданий у властивості `Text` текст. Властивість може приймати одне з наступних значень: `0-None` — смуга прокручування відсутній `1-Horizontal` — горизонтальна смуга прокручування `2-Vertical` — вертикальна смуга прокручування `3-Both` — горизонтальна й вертикальна смуги прокручування

Вид текстового поля міняється залежно від значення властивостей MultiLine і ScrollBar (рис. 7.3).

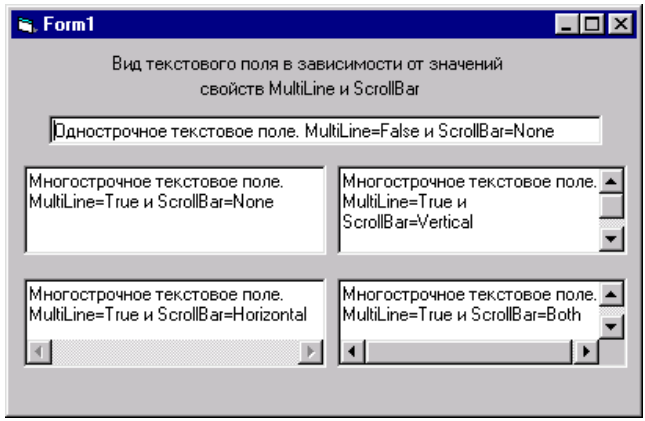


Рис. 7.3. Вид текстового поля залежно від значення властивостей MultiLine і ScrollBar

Керування текстом в об'єкті TextBox

Програма Visual Basic дозволяє під час виконання програми управляти текстом, відображуваним у текстовому полі, за допомогою властивостей SelStart, SelLength та SelText.

У випадку, коли фокус уперше переходить на текстове поле, курсор за замовчуванням установлюється ліворуч від тексту, що перебуває в поле. У результаті уведення або перегляду інформації курсор може переміщатися в межах поля. При наступному поверненні фокуса на поле курсор установлюється в те місце, куди він був установлений востаннє.

Використовуючи властивість `SelStart` об'єкта `TextBox`, можна вказати місце розміщення курсору в поле при установці фокуса. Значення 0 відповідає крайній лівій позиції. Властивість `SelLength` задає ширину крапки уведення. За замовчуванням вона дорівнює 0, тобто в тім місці, де курсор установлений, можна починати уведення символів, не видаляючи розташованої в ньому інформації. Розглянемо наступний приклад. Необхідно, щоб у формі, яка призначена для уведення даних, розміщені дані заміщалися новою інформацією. Для цього при установці фокуса на поле, символи повинні виділятися інверсними кольорами (рис. 7.4) і при уведенні даних віддалятися. Властивість `SelText` дозволяє задати текст, що замінить під час виконання програми виділений фрагмент.

Розглянемо приклад створення невеликого додатка, що містить форму, представлену на рис. 7.4. Форма містить заголовок і текстове поле, у якому при установці фокуса виділяється розміщений у ньому за замовчуванням текст. При створенні додатка будемо використати розглянуті нами елементи керування Label і TextBox.

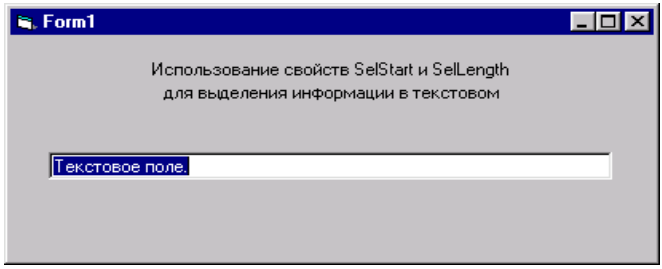


Рис. 7.4. Програмне виділення інформації, розміщеної в текстовому полі

1. Для створення нового додатка в меню **File** виберіть команду **New Project**. Відкривається вікно **Project** з розташованої в ньому новою формою.
2. Щоб розмістити у формі заголовок, натисніть кнопку `Label` на панелі елементів управління, встановіть курсор у верхню частину форми й намалюйте рамку необхідного розміру.
3. Відкрийте вікно властивостей, виділите властивість `Caption`, введіть у правому стовпці текст заголовка.
4. Щоб зберегти первісну ширину заголовка й розташувати текст у два рядки, встановіть для властивості `WordWrap` значення **True.**
5. Для завдання точної ширини заголовка привласніть властивості Autosize значення **True.**
6. За замовчуванням текст заголовка вирівняний по лівому краї. Щоб його отцентрувати, виділіть властивість `Alignment` (Вирівнювання). Потім у правому стовпці натисніть кнопку розкриття списку й виберіть із нього значення **Center.**
7. Для задання використовуваного в оформленні заголовка найменування шрифту, його розміру й накреслення скористайтеся властивістю `Font` (Шрифт).
8. Розмістимо тепер у формі текстове поле. Для задання тексту, відображуваного у текстовому полі при виконанні форми у властивість `Text` введіть, наприклад, ***Текстове поле.***
9. Ми хочемо, щоб при установці фокуса на текстове поле в ньому виділявся розмішений текст, тому створимо процедуру обробки події. Для відкриття вікна редактори коду двічі клацніть мишею на об'єкті `TextBox`.
10. У вікні редактора коду зі списку **Object** обране значення **Text1,** що вказує найменування текстового поля. Виберіть із правого списку **Procedure** значення **GotFocus, що** дозволяє задати процедуру обробки події одержання текстовим полем фокуса.
11. В області створення процедури між операторами `Private Sub Text1_Got Focus () i End Sub` розташуєте наступні команди:

```
Text1.SelStart = 0
Text1.SelLength = Len(Text1.Text)
```

Перша команда задає початкове положення виділюваного в поls тексту. Друга команда процедури задає довжину виділюваного фрагмента тексту. Використовувана в ній функція `Len` обчислює

довжину тексту, розміщеного в текстовому полі й заданого властивістю `Text`. На рис.7.5 показане вікно редактора коду з уведеними командами.

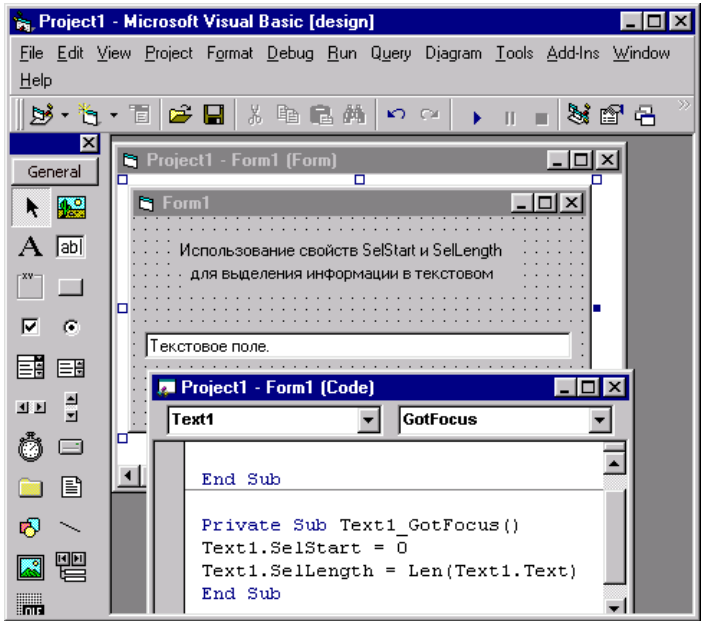


Рис. 7.5. Вікно редактора коду

Якщо ви хочете під час виконання програми замінити виділену в текстовому полі інформацію іншим текстом, то можете використати властивість `SelText` і вставити в процедуру наступну команду: `Text1.SelText` = "Інформація, що вводить за допомогою властивості SelText". У лапках розташовується розташовуваний у текстовому полі новий текст.

12. Після уведення тексту процедури закрийте вікно редактора коду.
13. Запустіть програму на виконання. Результат представлений на рис. 7.4. Інформація в текстовому полі виділена, і ви можете відразу ж уводити на її місце нову.

Текстові поля, що не редагують

Як ми вже говорили, текстові поля, на відміну від міток, призначені для уведення інформації користувачами додатка. Але в тому випадку, якщо ви хочете, щоб дані проглядалися, але не редагувалися, то можете скористатися властивістю `Locked`. При установці значення **True** користувач зможе переглядати дані без можливості їхнього редагування. У цьому випадку зміна інформації в текстовому полі може виконуватися тільки програмно.

Перевірка правильності уведення даних

При роботі з текстовими полями виникає необхідність перевірки даних, що вводяться користувачем. В Visual Basic для цих цілей призначена подія `Validate`. Наприклад, розміщене у формі поле служить для уведення дати й ви хочете, щоб при уведенні інформації в іншому форматі з'являлося відповідне попередження. Для цього виконаєте наступні дії:

1. Розмістіть у формі текстове поле.
2. Для завдання процедури обробки події `Validate` відкрийте вікно редактора коду, двічі клацнувши кнопку миші на текстовому полі.
3. Зі списку `Object` за замовчуванням обране значення **Text1,** що вказує найменування елемента керування `TextBox`. Із правого списку `Procedure` виберіть значення `Validate`, що дозволяє задати процедуру перевірки. даних, що вводять у поле.
4. Створіть наступну процедуру:

```
Private Sub Text1_Validate
If Not IsDate(Textl) Then MsgBox "Уводять данные, що,
  повинні бути датою"
End Sub
```

1. Закрийте вікно редактора коду. Розмістіть у формі ще одне текстове поле, що буде використано як об'єкт, на який можна перевести фокус після уведення інформації в перше поле.
2. Запустіть додаток на виконання.

3. Уведіть у створене поле інформацію в якому-небудь довільному форматі, наприклад текстову, і натисніть клавішу <Tab> для переміщення фокуса на друге поле. На екрані з'явиться попередження, аналогічне представленому на рис. 7.6.

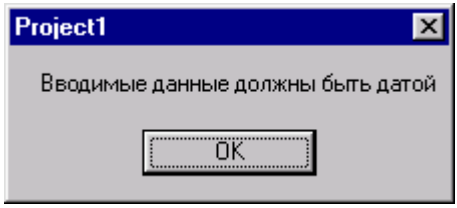


Рис. 7.6. Попередження, що з'являється при уведенні даних, що не відповідають заданим умовам

Використання текстового поля для уведення пароля

Текстове поле в Visual Basic характеризується двома властивостями, що дозволяють використати їх при створенні полів, що призначають для уведення пароля:

- `passwordchar` — задає символ, відображуваний у поле замість символів, що вводять;
- `MaxLength` — максимальне число символів, що вводять у поле.

На рис. 7.7 показана форма, у якій як символ, відображуваний в текстовому полі при уведенні пароля, використається символ зірочки.

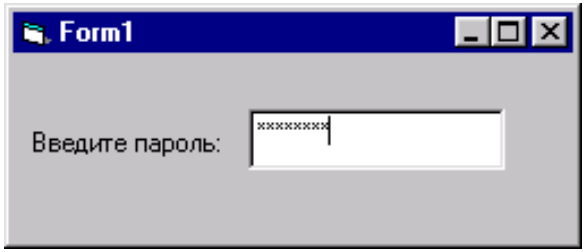


Рис. 7.7. Форма, призначена для уведення пароля

Підказка

Visual Basic дозволяє за допомогою властивості ToolTipText (Текст підказки) створювати текст короткого пояснення, що з'являється нижче курсору, коли він встановлюється на поле (рис. 7.8). Щоб задати текст пояснення до текстового поля, уведіть текст підказки в правий стовпець властивості.

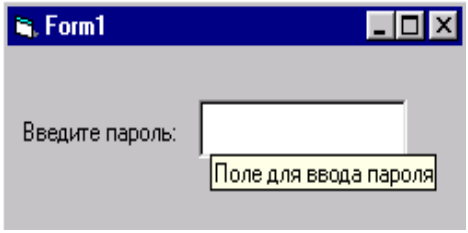


Рис. 7.8. Властивість ToolTipText дозволяє задати підказку, що з'являється під курсором

Елементи управління для ухвалення рішення

Ми розглянули способи ввода в програму чисел або рядків тексту. Але як бути, якщо потрібно з'ясувати у користувача просте питання, типа чи "Є у нього машина?" або "Який його сімейний стан?". Йдеться про такий тип питань, на які можна дати однозначну відповідь "Так/Ні"' або "Одружений/Неодружений". Відповіді на поставлені запитання можно одержати скориставшись спеціальною групою елементів управління, призначених для ухвалення рішення: `CommandButton, Check Box, Option Button, List Box, Combo Box.`

Вказані елементи управління призначені для введення і відображення на екрані логічних даних. Детальніше вони будуть розглянуті в подальших розділах цієї глави.

Кнопка (елемент CommandButton)

Мабуть, одним з найважливіших елементів управління для будь-якого застосування є кнопка або елемент типа **CommandButton**. Він призначений для ініціації користувачем в програмі деяких дій, які починають відбуватися після клацання на ньому кнопкою миші. Використовувати елемент управління типа `CommandButton` дуже просто — спочатку його потрібно помістити у форму, а потім привласнити його властивості `Caption` текст, який має відображуватися на кнопці. Помістить програмний код, що виконує потрібні дії, в процедуру обробки події `Click`. У даній процедурі, як і в будь-якій іншій, можна використовувати будь-які допустимі в Visual Basic оператори.

Хоча більшість користувачів вважають за краще клацати на кнопках мишею, деякі все ж люблять користуватися клавіатурою і вводити команди виключно з її допомогою. Дуже часто так поступають користувачі, які вводять велику кількість інформації в комп'ютер, наприклад заповнюють поля форми і відправляють інформацію в базу даних. Для задоволення потреб останньої категорії користувачів, потрібно зробити так, щоб натиснення на певну клавішу або комбінацію клавіш призводило до виникнення події, аналогічної клацанню мишею на кнопці. Це легко зробити, призначивши кнопці клавішу швидкого доступу. Тоді, якщо користувач натискуватиме клавішу `<Alt>` і, не відпускаючи її, натискуватиме вказану на кнопці клавішу швидкого доступу, виникне подія `Click` для даної кнопки форми.

Клавіша швидкого доступу призначається у момент привласнення значення властивості `Caption` елементу управління типа `CommandButton`. Помістить в рядку, який буде привласнений властивості `Caption`, символ амперсанда (&) перед буквою, відповідній бажаній клавіші швидкого доступу. Наприклад, якщо ви хочете, щоб після натиснення комбінації клавіш `<Alt+P>` спрацьовувала кнопка, на якій написане `Print`, привласніть властивості `Caption` цієї кнопки рядок `&Print`. Сам символ амперсанда не відображуватиметься на кнопці, замість цього наступний за ним символ буде підкреслений, в результаті на кнопці з'явиться напис `Print`. Це означає, що для даної кнопки вибрана комбінація клавіш швидкого доступу `<AIt+P>`.

При створенні форми, одну з її кнопок можна за умовчанням помістити у фокус. Така кнопка називається стандартною (default button). Тоді, якщо після відкриття форми користувач натискуватиме клавішу <Enter>, станеться подія Click для стандартної кнопки. Вказана дія аналогічна клацанню мишею на стандартній кнопці. Аби визначити стандартну кнопку форми, потрібно привласнити її властивості Default значення True. Зверніть увагу, що у формі може бути лише одна стандартна кнопка.

У формі можна також призначити кнопку відміни операції (Cancel button), яка аналогічна описаній в попередньому абзаці стандартній кнопці, за винятком того, що подія Click виникає після натиснення клавіші <Esc>. Для цього привласніть властивості Cancel потрібної кнопки значення True. Як і у попередньому випадку, у формі може бути лише одна кнопка відміни операції. Якщо властивостям Default і Cancel деяких кнопок привласнити значення True, то аналогічним властивостям інших кнопок автоматично будуть привласнені значення False.

Для управління зовнішнім виглядом кнопки використається властивість Style (Стиль). Воно містить два значення. За замовчуванням установлене значення **Standard,** що припускає, що кнопка буде містити текст, що задає властивістю Caption. Ви можете використати у формі графічні кнопки. У цьому випадку замість текстового напису на кнопці розміщається графічне зображення, що задає властивістю picture, а властивість Style повинне мати значення **Graphical.**

Для завдання графічного зображення, що поміщає на кнопці, виділите властивість picture і натисніть кнопку, розташовану в правому стовпці. У результаті відкриється діалогове вікно **Load Picture** (рис. 7.9), використовуючи яке ви можете вибрати на диску файл, що містить зображення.

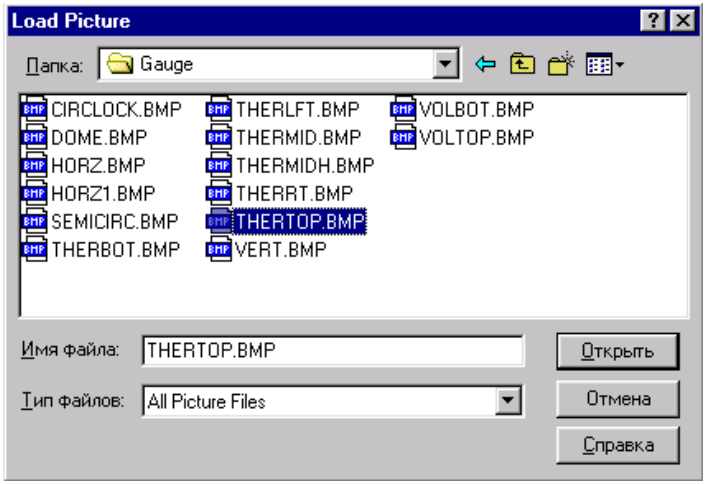


Рис.7.9. Діалогове вікно Load Picture, призначене для вибору зображення, що поміщає на кнопці

Крім перерахованих вище властивостей, для керування видом графічної кнопки використаються також властивості `Disabledpicture` і `DownPicture`. Властивість `DisabledPicture` дозволяє задати графічне зображення неактивної кнопки, а `DownPicture` — графічне зображення, відображуване на кнопці при її натисканні.

Прапорець (елемент Check Boxes)

Прапорці використовуються для відображення у формі логічних даних, тобто даних, які можуть набувати лише два значення, — `True` або `False`. Іншими словами, за допомогою прапорців у користувача можна запитати відповіді на поставлені програмою питання у формі "Так/Ні". Робота даного елементу управління чимось нагадує звичайний електричний вимикач. Він може знаходитися лише в двох станах — вимкненому або включеному; жодних проміжних станів бути не може! Коли прапорець знаходиться "у включеному стані", на ньому змальована галочка (v). Це означає, що користувач позитивно відповів на поставлене питання. Якщо ж прапорець

знаходиться "у вимкненому стані", квадратик буде порожнім. Це означає, що користувач негативно відповів на поставлене питання.

Проте, на відміну від електричного вимикача, прапорець може знаходитися в так званому *третьому стані*. В цьому випадку галочка зображається на фоні сірого квадратика. Такий режим часто використовується при написанні програм установки, коли під одним прапорцем "захований" цілий набір інших прапорців, причому деякі з них (але не всі!) знаходяться у включеному стані.

Зовнішній вигляд включеного і вимкненого прапорця залежить від значення його властивості, `Style`. Якщо встановити значення даної властивості рівним 1 — `Graphical`, то замість звичайного квадратика з галочкою, прапорець буде мати вигляд кнопки, стан якої (втоплений або підведений) залежить від значення відповідного логічного параметра. Крім того, на кнопку можна навіть помістити який-небудь малюнок.

Для вибору зображення, яке потрібно помістити на кнопку використовуються дві властивості — `Picture` і `DownPicture`. Перша властивість визначає картинку, яка з'явиться на кнопці, коли вона знаходиться у віджатому, або підведеному стані. Відповідно, друга властивість визначає картинку, яка з'явиться на кнопці, коли та знаходиться в натиснутому або втопленому стані..

Перемикач (елемент option Button)

Інколи перемикачі називають радіокнопками (radio button), оскільки їх функції дуже схожі на функції, виконувані кнопками перемикання діапазонів в автомобільних радіоприймачах. Перемикачі завжди повинні знаходитися в групах, причому лише один з перемикачів групи може бути "натиснутий" або бути активізований. Описуваний елемент управління служить для вибору одного параметра з представленої групи взаємовиключних параметрів. Давайте поекспериментуємо з перемикачами. Помістить у форму три перемикачі. При створенні перемикача, його властивості **Value** за умовчанням привласнюється значення `False`. Таким чином, у вихідному стані перемикач вимкнено. Зверніть увагу, що лише один з перемикачів групи може знаходитися у включеному стані. Іншими словами, як тільки ви встановите значення властивості `Value` одного з перемикачів рівним True, аналогічні властивості всіх інших перемикачів даної групи скидаються в `False`.

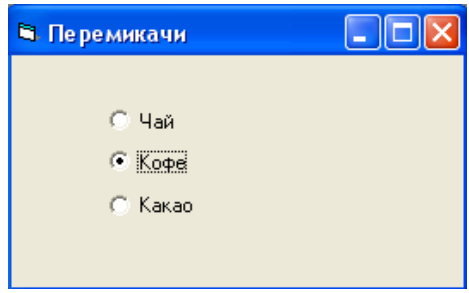


Рис.7.10 Використання перемикачів
для вибору елементу із списку

У програмі перемикачі можна використовувати двояко. Якщо потрібно виконати які-небудь дії після того, як користувач встановив один з перемикачів, помістить відповідний код в процедуру обробки події `Click`. Даний метод дуже добре працює, якщо перемикачі є частиною масиву елементів управління, як показано на наступному прикладі:

```
Private Sub optDrink_Click(Index As Integer) Select
  Case Index Case 0
MsgBox "Ви вибрали чай" Case 1
MsgBox "Ви вибрали каву" Case 2
MsgBox "Ви вибрали какао" End Select End Sub
```

Не поміщайте код в процедури обробки подій, що поступають від окремих елементів перемикача. Якщо потрібно визначити стан перемикача, скористайтеся оператором If, як показано нижче.

```
Private Sub cmdStartDrink_Click() If optSugar — True
  Then
DoWithSugar Else
DoWithoutSugar End If End Sub
```

Другий спосіб використовується в тому випадку, якщо потрібно дати можливість користувачеві зробити вибір, тобто після установки потрібного перемикача програма не повинна негайно виконувати які-небудь дії.

З рис.7.10. виходить, що лише один з перемикачів, розташованих у формі, може знаходитися у вибраному стані. Проте інколи треба помістити у форму декілька наборів перемикачів і

встановити по одному перемикачу в кожному наборі. Для вирішення подібного завдання набір логічно зв'язаних один з одним перемикачів потрібно помістити в об'єкт-контейнер, яким зазвичай є група. Використання об'єктів-контейнерів, таких як елементи управління типа Frame, буде розглянуто нижче. А доки погляньте на рис.7.11., на якому показано, як можна об'єднати два набори перемикачів в різні групи.

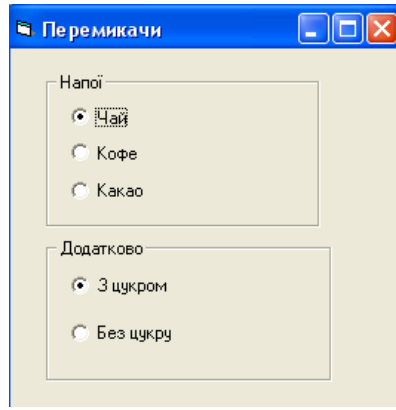


Рис.7.11. Набори перемикачів об’єднані в групи

Список (елемент ListBox)

Списки або елементи управління типа **ListBox** використовуються в програмах для наочного представлення інформації, а також для надання користувачеві можливості вибору одного елементу з деякої кількості представлених в списку. Значення в списку можуть розміщатися в одну або декілька колонок. Кількість стовпчиків задається властивістю `Columns` (Стовпчика). У тому випадку, якщо елементи списку не містяться у виділену для них у формі область, з'являються смуги прокручування, що дозволяють переглянути весь список.

Елемент управління типа **ListBox** характеризується такими параметрами як

- *Список елементів.* Як випливає з назви, це список тих елементів, з яких користувач повинен зробити свій вибір..

- *Вибраний елемент.* Це той елемент, на якому зупинив свій вибір користувач. Залежно від типа списку, вибраний елемент буде або підсвічується в списку, або поряд з ним буде встановлений прапорець.
- *Смуга прокрутки.* Поява смуги пркрутки свідчить про те, що в списку знаходиться більше елементів, чим показано на екрані. Крім того, смуга прокрутки дозволяє користувачам "прокрутити" список вниз або вгору у пошуках необхідного елементу. Для цього потрібно або перетягнути бігунок в потрібному напрямі, або клацнути на одній із стрілок смуги прокрутки.

Формуванням елементів списку займається програміст на етапі розробки програми або сама програма під час свого виконання. Користувач може лише зробити вибір із запропонованого списку.

При поміщенні першого елементу управління типа **ListBox** у форму ви побачите на екрані порожній прямокутник, усередині якого знаходитиметься текст `List1` (стандартне ім'я списку, призначене Visual Basic). У новому списку не буде ні смуги прокрутки ні елементів, які зможе вибрати користувач. Вертикальна смуга прокрутки з'являється в списку автоматично, як тільки новий елемент, що додається до списку, вже не вміщатиметься у виділеній для списку області на екрані. Зверніть увагу, що в списку немає горизонтальної смуги прокрутки. Тому ви повинні попіклуватися про те, щоб його ширина була достатньою для відображення щонайдовшого елементу списку.

Використання списків

Найпростішим способом наповнення списку новими елементами є використання методу **AddItem.** При виклику цього методу йому потрібно передати лише один параметр — текстовий рядок, який потрібно помістити в список. А тепер давайте розглянемо нескладний приклад з елементом управління типа **ListBox.**

Почніть новий стандартний проект Visual Basic, помістить у форму елемент управління типа `ListBox`, після чого додайте приведений нижче фрагмент коду в процедуру обробки події Load для форми.

```
Private  Sub  Form_Load()
Dim i As   Integer
For   i   =   1   To   100
```

```
Listl.Addltem "Елемент номер " & i
Next i
End Sub
```

В наведеному вище прикладі наповнення списку елементами сталося у момент виконання програми, але існує можливість створити готовий список ще під час розробки програми. Для цього потрібно помістити елементи списку у властивість `List`. Кожен рядок властивості `List` відповідає одному елементу списку. Після введення поточного елементу для переходу на новий рядок натискуйте клавіші `<Ctrl+Enter>`.

Запустить проект на виконання. На екрані ви побачите список, в якому знаходитиметься 100 елементів. Для очищення списку від його вмісту існує метод **Clear:** `Listl.Clear`

Елементи списку зберігаються у вигляді масиву. До будь-якого з них можна звернутися за допомогою властивості **List** елементу управління типа **ListBox.** Перший індекс масиву починається з нуля, тому, щоб вивести на екран перший елемент списку, скористайтеся наступним оператором: `Print Listl.List(0)`

Щоб протестувати роботу цього оператора, перервіть роботу програми, натиснувши клавіші `<Ctrl+Break>`, після чого наберіть цей оператор у вікні Immediate.

Існують ще дві корисні властивості, які впливають на роботу елементу управління типа ListBox, — **ListCount** і **Listlndex**. За допомогою властивості `ListCount` можна визначити, скільки елементів знаходиться в списку; властивість `Listlndex` містить номер вибраного елементу списку.

Найчастіше значення властивості `ListCount` використовується при обробці вмісту всіх елементів списку в циклі. Проте при цьому не варто забувати, що індекс першого елементу списку дорівнює нулю, а не одиниці. Тому максимальне значення змінної циклу, яка використовується як індекс масиву, має бути на одиницю менше значення властивості `ListCount`. Нижче наведений приклад програми, яка оброблює елементи списку в циклі.

```
For i = 0 To Listl.ListCount-1
 ' Обробка елементу Listl.List(i)
Next i
```

Якщо жоден елемент не вибраний в списку, значення властивості `Listlndex` дорівнює -1.

Щоб поекспериментувати з властивістю Listlndex, додайте в обробник події Click наступний рядок коду: Msgbox Listl.List(Listl.Listlndex)

Знову запустить програму і клацніть на одному з елементів списку. В результаті на екрані повинно з'явиться діалогове вікно, в якому знаходиться вибраний елемент списку.

Існує ще один спосіб дізнатися, який елемент списку вибрав користувач. Для цього потрібно скористатися значенням властивості Text елементу управління типа ListBox. Річ у тому, що значенням даної властивості є рядок списку, що знаходиться у фокусі. Таким чином, в разі простого списку рядок, на якому клацнув користувач, автоматично поміщається у властивість Text. Якщо жоден елемент в списку ще не вибраний, значення властивості Text дорівнює порожньому рядку (" ").

```
Private Sub Listl_KeyDown(KeyCode As Integer, Shift As
  Integer)
If KeyCode = vbKeyDelete And Listl.Listlndex <> -1 Then
Listl.Removeltem Listl.Listlndex
End If
End Sub
```

Запустить програму і переконаєтеся, що при натисненні клавіші <Delete> дійсно відбувається видалення із списку вибраного елементу.

Додавання елементів у список

Елементи в список можуть додаватися під час розробки й програмно з використанням методу Additem. При формуванні списку під час розробки у властивості List вручну задається весь необхідний список. Дані не обов'язково вводити за абеткою, тому що їх можна впорядкувати, установивши для властивості Sorted (Сортування) значення **True.** У цьому випадку елементи, що вводять знову у список також будуть розташовуватися за абеткою.

Описувана можливість сортування списків є дуже зручною і дозволяє швидко упорядкувати дані. Проте змінити значення властивості Sorted для конкретного списку можна лише при розробці програми. Тому, якщо під час виконання програми вам потрібно періодично відключати сортування, слід скористатися двома

списками, один з яких буде відсортовано, а другий — ні. Змінюючи властивість Visible, ви легко зможете відображувати на екрані потрібний список.

Небажано використовувати сортування, якщо дані в список будуть додаватися методом Additem з використанням параметра Index.

Для додавання елементів у список програмним способом призначений метод AddItem, який має наступний синтаксис:

```
NameList.AddItem вираження [, Index]
```

де:

- NameList — найменування списку, що задає властивістю Name;
- вираження — елемент списку. Якщо це символьна величина, то вона повинна бути поміщена в лапки;
- Index — порядковий номер елемента в списку. Якщо цей параметр відсутній, елемент додається в кінець списку.

При використанні параметра Index необхідно враховувати, що нумерація елементів починається з 0.

Наприклад, для програмного формування вмісту списку вам необхідно задати наступну процедуру:

```
Private Sub Form Load()
List1.AddItem "Москва"
List1.AddItem "С.Петербург"
List1.AddItem "Псков"
List1.AddItem "Новгород"
List1.AddItem "Чебоксари"
End Sub
```

На рис. 7.12 показане вікно редактора коду, що містить процедуру формування списку міст.

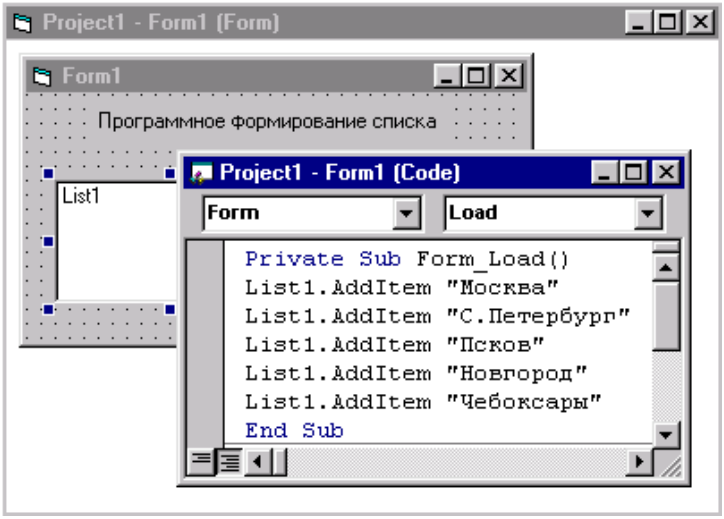


Рис.7.12. Вікно редактора коду із процедурою, що формує список

Міста в списку розташовуються в тім порядку, у якому вони задані в процедурі. Наприклад, якщо рядок коду List1.Additem "Чебоксари" замінити рядком

```
List1.Additem "Чебоксари",0
```

то в списку міст **Чебоксари** будуть поміщені в першу позицію (рис. 7.13).

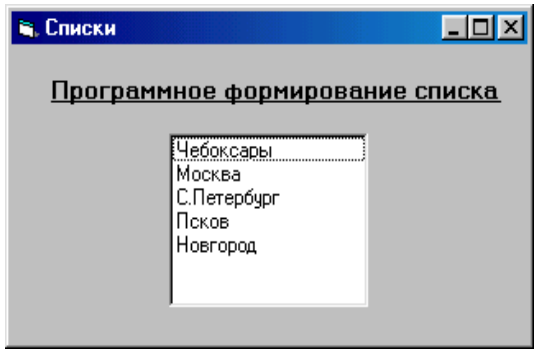


Рис. 7.13. Використання параметра Index дозволяє задати порядок елементів у списку

Видалення елементів зі списку

Visual Basic дозволяє програмно видаляти елементи зі списку за допомогою методу Removeitem, що має наступний синтаксис:

```
NameList.RenioveItem Index
```

де `NameList` — найменування списку, що задається властивістю `Name`,

`Index` — порядковий номер елемента, що видаляється.

Наприклад, щоб видалити зі створеного нами в попередньому прикладі списку `List1` другий елемент, необхідний наступний програмний код:

```
List1.Remove Item 1
```

Щоб видалити всі елементи зі списку, можна використати метод `Clear` (Очистити). У цьому випадку програмний код виглядає так: `List1.Clear`.

Стиль оформлення списку

Для керування зовнішнім виглядом списку використається властивість `Style` (Стиль). Вона містить два значення: **Standard і CheckBox.** За замовчуванням використається значення **Standard,** що ми вже розглядали раніше. При установці для властивості `Style` значення **CheckBox** до елементів списку будуть додані прапорці. На рис. 7.14 показані два види списків залежно від значення властивості Style.

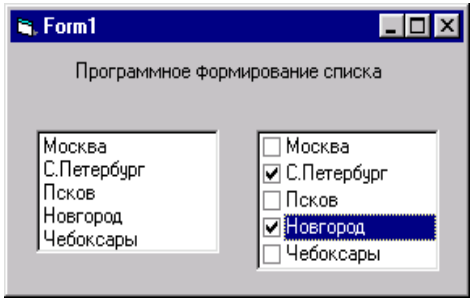


Рис. 7.14. Два стилі оформлення списків залежно від значення властивості `Style`

Для зміни значення властивості Style під час виконання додатка можна використати константи Visual Basic **vbListBoxStandard** і **vbListBoxCheckBox.**

Вибір декількох елементів зі списку

Програма Visual Basic дозволяє використати списки, що надають користувачеві можливість вибирати декілька елементів. Для створення таких списків призначена властивість **MultiSelect** (Множинний вибір). Вона може приймати значення, представлені в табл. 7.4.

Таблиця 7.4

Значення властивості MultiSelect

Значення властивості	**Опис**
0-Non (Немає)	Стандартний список, використовуваний за замовчуванням. Дозволяє вибирати одне значення
1-Simple (Простий множинний вибір)	Дозволено вибір декількох елементів клацанням миші або натисканням клавіші <Spacebar> (Пробіл). Для скасування вибору необхідно клацнути мишею або нажати клавішу <Spacebar> ще раз
2-Extended (Розширений множинний вибір)	Дозволено вибір декількох елементів за допомогою способу, використовуваного в Windows. Для вибору елементів, розташованих підряд, необхідно при натиснутій клавіші <Shift> вибрати перший елемент із обираних, а потім останній. При цьому будуть обрані всі розміщені між ними елементи. Для вибору елементів, розташованих не один по одному, необхідно нажати клавішу <Ctrl> і, утримуючи її натиснутої, виділити необхідні елементи списку

Для обробки множинного вибору не можна використати властивість Listindex, застосовувану при роботі зі звичайними списками. Вибір декількох елементів списку фіксується у властивості Selected, що є масивом, розмірність якого дорівнює кількості елементів у списку (визначається властивістю ListCount).

Обраному елементу списку відповідає значення **True** відповідного елемента властивості `Selected`, а іншим — **False.**

Розглянемо приклад використання властивості `Selected`. Розмістимо у формі два списки. У перший список уведемо елементи з використанням властивості `List`. Список елементів у другому списку буде формуватися при переміщенні на нього фокуса після вибору елементів першого списку. Таким чином, другий список буде містити обрані в першому списку елементи. Для створення додатка виконаєте наступні дії:

1. Розмістіть у формі два списки за допомогою кнопки **ListBox** на панелі елементів керування.
2. Використовуючи властивість `List` першого списку, уведіть елементи списку.
3. Щоб дозволити вибір з першого списку декількох елементів, установите для властивості `Multiselect` значення **1-Simple.**
4. Тепер необхідно задати процедуру формування елементів другого списку при переміщенні на нього фокуса. Для цього двічі клацніть мишею на другому списку.
5. У вікні, що відкрилося, редактора коду `Procedure` зі списку виберіть подію `GotFocus` (Одержання фокуса) і введіть наступний код:

```
Private Sub List2_GotFocus()
List2.Clear
For i = 0 To List1.ListCount – 1
If List1.Selected(i) Then List2.Addltem (List1.List
   (i))
Next
End Sub
```

На рис. 7.15 представлений отриманий результат.

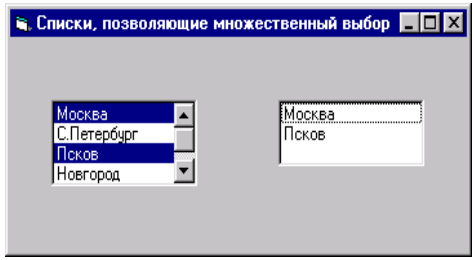


Рис. 7.15. Відображення обраних значень у другому списку

Асоціація даних з елементами списка

Дуже часто виникає завдання представити список яких-небудь елементів, з якими оперує програма, в наочному вигляді. Наведемо простий приклад. Припустимо, що ви створюєте базу даних співробітників якого-небудь відділу. Кожному із співробітників привласнений табельний номер, по якому програма "впізнає" цю людину. Користувачеві вашої програми табельний номер співробітника ні про що не говорить. Для нього бажано, щоб програма виводила список співробітників в звичному вигляді, тобто з іменами і прізвищами. Таким чином, потрібно сформувати список співробітників і асоціювати з кожним елементом цього списку табельний номер. Для цієї мети в Visual Basic передбачена спеціальна властивість елементу управління типа `ListBox`, яке називається **ItemData**. Дана властивість є масивом цілих чисел подвійної точності (`Long Integer`), розмірність якого відповідає загальному числу елементів списка. Однією з важливих особливостей описуваної властивості є те, що порядок елементів в списку не впливає на дані, які знаходяться в масиві `ItemData`. Іншими словами, якщо ви асоціювали з яким-небудь елементом списку деяке число і помістили його у відповідне місто властивості `ItemData`, то це число як би "прилипає" до даного елементу списку, навіть якщо згодом порядок елементів в списку буде змінений, наприклад внаслідок сортування. Наведемо приклад. Щоб отримати число, що асоціюється з першим елементом списку `List1`, треба скористатися наступним оператором:

```
Listl.ItemData(0)
```

А якщо потрібно отримати число, що асоціюється з поточним виділеним елементом списку, скористайтеся таким оператором:

```
Listl.ItemData(Listl.Listlndex)
```

Нагадаємо, що у властивості `ListIndex` зберігається індекс поточного вибраного елементу в списку.

Як тільки в список додається новий елемент, для нього тут же створюється відповідне поле в масиві `ItemData`. Зрозуміло, що ваше завдання помістити в правильне поле цього масиву асоційоване число. Але як визначити його номер, особливо якщо після додавання нового елементу, список буде відсортовано? На щастя, в Visual Basic це завдання вирішується дуже просто. Потрібно скористатися значенням властивості **Newlndex** елементу управління `ListBox`. У ньому знаходитиметься індекс останнього доданого в список елементу. У наведеному нижче прикладі до відсортованого списку

співробітників додається новий елемент, після чого табельний номер нового співробітника записується в правильне поле масиву асоційованих даних ItemData.

```
lstCustomers.Addltem   "Іванов,   Петро"
lstCustomers.ItemData(lstCustomers.Newlndex)   =
  21472301
```

Створивши масив асоційованих даних, ви відразу "вбиваєте двох зайців". По-перше, користувач може легко вибрати із списку потрібного співробітника по його прізвищу і імені. По-друге, програма відразу ж визначає асоційований з цим співробітником табельний номер і виконує з ним які-небудь дії. Відпадає необхідність у виконанні проміжної операції — пошуку табельного номера по імені співробітника. Покажемо це на прикладі процедури обробки події `Click`, що відбувається у момент вибору потрібного співробітника в списку.

```
Private  Sub  lstCustomers_Click()
Dim  lgThisCust  As   Long
lgThisCust
  =lstCustomers.ItemData(lstCustomers.Listlndex)
Call   LookUpAccount(lgThisCust)
 End  Sub
```

Поле із списком (елемент ComboBox)

Поле із списком є ще одним елементом управління, що дозволяє представляти у вигляді списка великі обсяги інформації. Існує декілька варіантів використання полів із списком.

14. *Поле із списком, що розкривається.* Даний тип елементу управління є звичайним текстовим полем, об'єднаним із списком, що розкривається. Користувач може або вибрати готовий елемент із списку, або ввести новий елемент в текстове поле.
15. *Поле із звичайним списком.* У цьому типі елементу управління інформація відображується в звичайному (що не розкривається) списку. Як і у попередньому випадку, користувач може вибрати готовий елемент із списку або ввести новий елемент в текстове поле.

16. *Список, що розкривається.* Як випливає з його назви, спочатку список знаходиться в згорнутому (закритому) стані. Щоб зробити вибір, користувач повинен розкрити список, клацнувши на направленій вниз стрілці. Основною відмінністю даного елементу управління від двох попередніх є те, що користувач не може ввести новий елемент в поле. Він може вибрати лише те, що представлене в списку.

З точки зору програмування, робота з полями із списком нічим не відрізняється від роботи із звичайними списками. Для зміни вмісту списка використовуються методи **Addltem, Removeltem** і **Clear.** Елементи списка можуть бути як відсортовані, так і ні. У поля із списками також передбачені властивості `Itemdata` і `Newlndex`. Єдина відмінність описуваних списків — наявність текстового поля для введення інформації.

Стилем оформлення списку типу ComboBox управляє властивість `Style`. Вона може приймати значення, представлені в табл. 7.5.

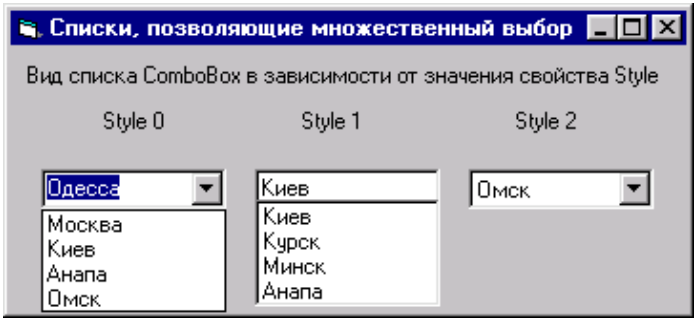


Рис. 8.16. Стилі оформлення списку типу ComboBox

Для встановлення значення `Style` можна використати також внутрішні константи Visual Basic **vbComboDropDown, vbComboSimple**, та **vbComboDropDownLis**; відповідно.

Таблиця 7.5

Значення властивості Style

Значення властивості	Опис
`0-Dropdown Combo`	Стиль, використовуваний за замовчуванням. Даний список показаний на рис. 7.15 ліворуч. Користувач може ввести значення в текстове поле, розташоване у верхній частині списку, або відкрити список, нажавши кнопку зі спрямованою вниз стрілкою із правої сторони поля, і вибрати необхідне значення. Обране значення переноситься в текстове поле
`1-Simple Combo`	Вид даного списку показаний на рис. 7.15 у центрі. При даному значенні список відображається у формі у відкритому стані. Якщо всі елементи в ньому не містяться, з'являється вертикальна смуга прокручування. Користувач може ввести значення в текстове поле, що розташовується у верхній частині списку, або вибрати необхідне значення, що буде перенесено в текстове поле
`2-Dropdown List`	Вид даного списку показаний на рис. 7.16 праворуч

Додавання елементів у список типу *ComboBox*

Елементи в список типу **ComboBox** можуть додаватися під час розробки за допомогою властивості `List` і програмно з використанням методу `Additem` так само, як у список типу `ListBox`. При формуванні списку під час розробки у властивості `List` вручну задається весь необхідний список. Дані, відображувані списком, можна впорядкувати, установивши для властивості `Sorted` (Сортування) значення **True.**

Для додавання елементів у список програмно використається метод Additem, що має наведений нижче синтаксис:

```
NameList.AddItem вираження [, index]
```

де:

`NameList` — найменування списку, що задає властивістю Name;

`вираження` — елемент списку. Якщо' це символьна величина, то вона повинна бути поміщена в лапки;

Index — порядковий номер елемента в списку.

Якщо при додаванні елементів у список параметр index відсутній, то елемент додається в кінець списку.

Звичайно для програмного формування списку використається процедура Form_Load обробки події. Наприклад, щоб програмно сформувати елементи для списку типу Combovox, що містить список міст, вам досить задати наступну процедуру:

```
Private Sub Form Load()
Combo1.Additem "Москва"
Combo1.Additem "С.Петербург"
Combo1.Additem "Псков"
Combo1.Additem "Новгород"
Combo1. Additem "Чебоксари"
End Sub
```

Якщо при формуванні списку ви не використаєте параметр Index, елементи в списку розташовуються в тім порядку, у якому вони задані в процедурі. Наприклад, якщо до наведеного нижче процедурі додати рядок коду

```
Combo1.AddItem "Київ",0
```

то в списку міст **Київ** буде поміщений у першу позицію, а всі інші елементи зміщені на одну позицію вниз.

Видалення елементів зі списку типу *ComboBox*

Видалення елементів зі списку типу Combobox здійснюється за допомогою методу RemoveItem, що має наступний синтаксис:

```
NameList.Removeltem index
```

де NameList — найменування списку, що задається властивістю Name, а Index — порядковий номер елемента, що видаляє, у списку.

Наприклад, для видалення зі списку, що має найменування combol, першого елемента, можна використати наступний код:

```
Combo 1. RemoveItem 0
```

Для видалення всіх елементів зі списку можна використати метод Clear (Очистити). У цьому випадку програмний код виглядає так: Combol. Clear.

Доступ до елементів списку

Для одержання доступу до обраного елемента списку типу ComboBox можна використати властивість Text. Значенням цієї властивості є уведене в текстове поле списку (для списків, у яких значення Style дорівнює **0-Dropdown Combo** або **1-Simple Combo)** або обране зі списку значення.

Щоб одержати доступ до елементів списку, використайте властивість List. Значення цієї властивості є масивом, розмірність якого дорівнює значенню властивості ListCount, тобто кількості елементів у списку. Наприклад, значення першого елемента списку буде дорівнює Combol. List (0), другого — Combo1.List (1) і т.д.

Для визначення положення елемента в списку типу ComboBox можна використати властивість ListIndex. Наприклад, при виборі першого елемента списку значення властивості ListIndex буде дорівнює 0.

Значення властивості ListIndex дорівнює -1, якщо зі списку не обраний елемент, а також якщо елемент у текстове поле списку уведений користувачем вручну.

Для одержання індексу останнього доданого в список елемента можна використати властивість Newindex (Новий індекс).

Елементи управління спеціального призначення

Останню групу внутрішніх елементів управління Visual Basic ми назвали "елементами спеціального призначення", оскільки їх не можна віднести до жодного з описаних вище типів. Проте це зовсім не означає, що вони рідко використовуються або абсолютно даремні. Нижче приведений список спеціальних елементів управління, що розглядаються в даному розділі.

- Смуги прокрутки (елементи типа VScrollBar і HScrollBar).
- Таймер (елемент типа Timer).
- Група (елемент типа Frame).

Смуги прокрутки

Принцип роботи смуг прокрутки чимось нагадує повзунок регулювальника гучності стереосистеми. Вони використовуються для встановлення параметра, значення якого може мінятися в деякому діапазоні — від мінімуму до максимуму. З точки зору програмування, описуваний елемент управління повертає числове значення, яке залежить від положення бігунка, а також від вибраного програмістом діапазону чисел. Те, як це число використовувати в програмі, далі вирішує програміст.

У Visual Basic передбачено два типи смуг прокрутки: вертикальна і горизонтальна. У документації вони називаються як елементи управління типа **VScrollBc HScrollBar** відповідно.

Описувані два типи смуг прокрутки відрізняються лише зовнішнім виглядом, або напрямом руху бігунка. У прикладах, що розглядаються нижче, ми користуємося горизонтальною смугою прокрутки, хоча все інформація відноситься також і до вертикальної смуги прокрутки. Ці два елементи абсолютні рівноправні.

Створення смуги прокрутки

З точки зору програмування, смуги прокрутки є одними з найпростіших елементів управління. Перед початком роботи вам необхідно визначити діапазон чисел, що вводяться, встановивши значення властивостей Min і Max. Подальша робота з описуваним елементом управління полягає в читанні і установці властивості **Value**. Як правило, вибирається діапазон чисел від 0 до 100. оскільки при читанні значення властивості Value величини повинні виражатися у відсотках.

Значення властивостей Min, Max і Value елементів управління типа vScrollBar і HScrollBar є цілими числами, тобто діапазон їх змінення від -32768 до +32767. Значення властивості Value безпосередньо залежить від мінімального і максимального значення діапазону, встановленого у властивостях Min і мах. Не забувайте про це при програмуванні, оскільки використання неправильного типа даних або установка неправильного значення цих властивостей приведе до помилки.

При зміні значення властивості Value автоматично виникає подія **Change** описуваного елементу управління. Давайте поекспериментуємо з горизонтальною смугою прокрутки. Створіть

новий стандартний проект Visual Basic і розташуйте у формі горизонтальну смугу прокрутки і текстове поле. Помістить приведений нижче текст програми у вікно коду.

```
Private Sub Form_Load()
HScroll1.Min = Me.Left
HScroll1.Max = Me.ScaleWidth
End Sub
Private Sub HScroll1_Change()
text1.Left = HScroll1.Value
End Sub
```

Запустить програму. Зверніть увагу, що, як тільки ви почнете перетягувати бігунок смуги прокрутки, положення на екрані текстового поля синхронно змінюватиметься. Для переміщення бігунка можете також скористатися клавішами управління курсором, а також <Home> і <End>.

Зміна величини переміщення

Якщо вам доводилося працювати з однією з Windows-програм, наприклад, ви, напевно, звертали увагу, що за допомогою смуги прокрутки можна швидко переміщатися по документу, а також виконувати його плавну прокрутку. Наприклад, якщо клацнути на одній із стрілок, розташованих на початку або кінці смуги прокрутки, документ буде плавно переміщений на невелику відстань у відповідному напрямі, а якщо клацнути прямо на смузі прокрутки (не на бігунку!), то буде виконана прокрутка документа на цілу сторінку.

Величина зміни значення властивості Value після клацання на одній із стрілок смуги прокрутки залежить від значення властивості **SmallChange.** Як випливає з назви, властивості SmallChange потрібно привласнити невелике значення, аби отримати плавне переміщення бігунка. За умовчанням йому привласнено значення 1, яке може застосовуватися в більшості випадків.

Якщо клацнути на вільній частині смуги прокрутки (між повзунком і одній із стрілок), значення властивості Value зміниться на більшу величину, ніж після клацання на одній із стрілок. Воно визначається значенням властивості **LargeChange**. За умовчанням значення цієї властивості, як і властивості SmallChange, дорівнює 1. Проте, залежно від виконуваних функцій вашої програми, ви повинні

привласнити властивості LargeChange відповідне значення. Наприклад, якщо положення бігунка відлічуватиметься у відсотках (тобто значення Min = 0, а Max = 100), то найбільш відповідним значенням для властивості LargeChange буде значення 10.

Відображення значення властивості Value на екрані

Хоча візуальне представлення чисел дуже зручно, інколи у користувача виникає потреба визначити точне значення параметра, якому відповідає певне положення бігунка. Крім того, інколи смуги прокрутки використовуються як елемент інтерфейсу, що дозволяє встановити деяке значення представленого ним параметра. Наприклад, смугу прокрутки можна використовувати для вибору однієї з букв алфавіту, що визначає критерій пошуку в базі даних. Таким чином, при переміщенні бігунка на екрані одне за іншим з'являтимуться нові значення змінного параметра Як тільки користувач відпустить кнопку миші, буде виконаний пошук в базі даних.

Основна хитрість описуваного способу полягає у тому куди помістити програмний код. Щоб значення властивості Value завжди відображувалося на екрані, потрібно обробити три події, вказані нижче.

1. **Form_Load**. Ця подія використовується для відображення початкового значення властивості Value після призначення діапазону змінення цієї властивості.
2. **Change**. Дана подія відбувається у момент відпуску кнопки миші після перетягання смугою прокрутки бігунка в нове положення. Крім того, ця ж подія відбувається також після клацання мишею на одній із стрілок або на порожній області смуги прокрутки.
3. **Scroll.** Ця подія виникає після переміщення бігунка смуги прокрутки. Вона дозволяє відображувати на екрані значення властивості Value або виконати які-небудь інші дії до того, як станеться подія Change.

Щоб реалізувати описаний вище приклад з вибором букв алфавіту, створіть новий стандартний проект Visual Basic. Помістить у форму смугу прокрутки і елемент управління типа Label. У вікно коду введіть текст програми, приведений нижче

```
'  Приклад використання смуги прокрутки
Private  Sub  Form_Load()
'   Встановимо числові   значення   властивостей  Max
  i Min
Hscroll1.Max  = Asc("Z")
Hscroll1.Min  =  Asc("A")
'   Відображуватимемо початкове  значення  властивості
  Value
Label1.Caption  =  Chr$(KScrolll.Value)
End  Sub

Private  Sub HScrolll_Change()
Labell.Caption  =   "Пошук   "   &
  Chr$(HScrolll.Value)
'   Тут  потрібно  помістити   код для пошуку  значення
  в  базі даних
End  Sub

Private   Sub  HScrolll_Scroll ()
Labell.Caption  =   "Відпустите  кнопку,   аби вибрати
  "   & Chr$(HScrolll.Value)
End  Sub
```

Запустить програму і зверніть увагу, як змінюватиметься текст напису у міру того як ви маніпулюватимете з різними частинами смуги прокрутки. Помістивши код в потрібні процедури обробки подій (як показано в лістингу), ви забезпечили правильне функціонування програми. Іншими словами, програма завжди відображуватиме на екрані правильне значення властивості Value для смуги прокрутки, а також виконуватиме задумані дії (в даному випадку пошук в базі даних) лише в потрібні моменти часу.

Таймер

В Visual Basic існує елемент керування, що обробляє дані системнного часу. Цей об'єкт називається таймером. Його можна використати для виконання певних дій через заданий інтервал часу.

На відміну від реального годинника, елемент управління типа **Timer** не видає жодних сигналів по досягненню заданого моменту часу. Замість цього він запускає відповідну процедуру обробки події **Timer**.

Елемент управління типа **Timer** може багато разів виконувати відлік заданого інтервалу часу, якщо значення властивості `Enabled` рівне True.

Описуваний елемент управління призначений для відліку невеликих проміжків часу (не більше 1 хвилини).

У прикладних програмах таймер використовується для різних цілей, як, наприклад, виконання певної дії через заданий інтервал часу або виконання якихось дій із складеного заздалегідь розкладу.

Для розміщення у формі таймера використається кнопка **Timer** (Таймер)  на панелі елементів управління форми. Об'єкт даного типу має наступні властивості (табл. 7.6)

Таблиця 7.6

Властивості об'єкту Timer

Властивість	**Призначення**
`Interval` (Інтервал)	Інтервал активізації об'єкта в милісекундах. Може приймати значення від 0 до 64767 (від 0 до 64,8 секунди)
`Enabled` (Доступно)	Установлює режим роботи таймера. Якщо значення властивості дорівнює True (Істина), то таймер починає відраховувати час відразу ж після запуску форми. У противному випадку ви повинні запустити таймер по якій-небудь зовнішній події (наприклад, при натисканні на кнопку). Установка для властивості значення False припиняє операції таймера

Подія `Timer` (Таймер) об'єкта-таймера наступає через кожний встановлений у властивості interval проміжок часу. У процедурі обробки даної події необхідно визначити дії, виконувані із заданою частотою.

Для запуску таймера можна використати метод `Reset` (Установити). Цей метод не пов'язаний з якою-небудь подією, тому ви повинні виконати його при настанні яких-небудь інших подій, наприклад при натисканні на кнопку запуску таймера.

Використання об'єкта-таймера розглянемо на прикладі форми, у якій через заданий інтервал часу на екран буде виводитися системний час комп'ютера. Для створення даної форми виконайте наступні дії:

1. Відкрийте вікно для створення нового проекту.

2. Помістіть у форму мітку для відображення поточного системного часу. Створіть пояснювальний напис до мітки.
3. Для створення об'єкта-таймера натисніть кнопку **Timer** (Таймер) на панелі елементів управління й розташуєте його у формі.

Розміщений у формі елемент управління **Timer** зображується у вигляді значка, показаного на рис. 7.17. При запуску форми на виконання він стає невидимим користувачеві додатка.

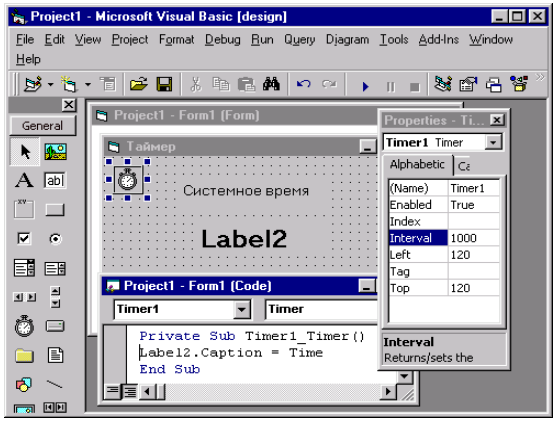


Рис. 7.17. Форма з розміщеним у ній об'єктом Timer

4. Визначите інтервал часу, через який необхідно робити відновлення часу у формі. Для цього скористайтеся властивістю `Interval`, значення якого задається в миллисекундах. Для відновлення часу щосекунди введіть значення **1000.**
5. Відкрийте вікно редактора коду й створіть просту процедуру, що привласнює властивості Caption мітки поточний час:

```
Private Sub Timer1 Timer()
Label1.Caption = Time
End Sub
```

Форма з розміщеними елементами управління показана на рис. 7.18.

6. Збережіть створену форму й запустіть її на виконання. Ви побачите час, обновлюваний щосекунди (рис. 7.18).

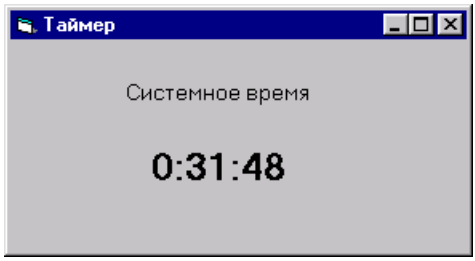


Рис. 7.18. Використання таймера

Резюме

Інтерактивний додаток в Visual Basic створюється на базі форми, яка є основним вікном інтерфейсу. Елементи управління форми забезпечують взаємодію з користувачами. Добре знання найбільш часто використовуваних елементів управління Visual Basic, їх основних властивостей необхідно розробнику програмного забезпечення. В даному розділі ви познайомилися з основними стандартними елементами управління.

Контрольні запитання та завдання

1. Дайте перелік основних стандартних елементів управління, які входять в постачання Visual Basic.
2. Які елементи управління використовуються для відображення тексту ?
3. Перелічіть властивості, елемента управління TextBox, що визначають оформлення тексту.
4. Як у Visual Basic організується перевірки даних, що вводяться користувачем ?
5. Дайте перелік елементів управління, які призначені для введення і відображення на екрані логічних даних.
6. Якими параметрами характеризуеться елемент управління ListBox?

7. Які елементи управління відносяться до елементів управління спеціального призначення ?
8. Для чого використовується смуга прокрутки ?
9. Які типи смуг прокрутки існує у Visual Basic ?
10. Як у Visual Basic організується виконання певної дії через заданий інтервал часу?
11. Розробіть додаток для сортування текстового файлу із використанням елементу ListBox.
12. Напишіть програму, яка у списку ListBox або у комбінованому списку ComboBox міняє порядок розміщення елементів на зворотній.
13. Розробіть універсальне діалогове вікно вводу пароля.
14. Розробіть процедуру, яка буде відображати поточний час.

Розділ VIII
РОБОТА З ФАЙЛАМИ ТА ОРГАНІЗАЦІЯ ДРУКУ

При проектуванні додатка досить часто виникає необхідність працювати безпосередньо з файлами. Це потрібно, наприклад, для додавання, видалення файлів або каталогів (папок), запису даних у файли або читання з них як програмно, так і в інтерактивному режимі. Необхідність роботи з файлами виникає також при створенні програми інсталяції розробленого додатка на користувацькі комп'ютери, читання даних з файлів при ініціалізації додатка з використанням файлів настроювання, організації виведення файлів на друк. З цією метою Visual Basic має повний набір функцій, що працюють із файлами, папками й пристроями.

У Visual Basic існує поняття типу файлу, що визначається організаційною структурою зберігання інформації у файлі й способом доступу до цієї інформації. Прийнято виділяти такі типи файлів.

Файли послідовного доступу. Як правило, це текстові файли або аналогічні ним. Такі файли являють собою послідовність символів. При цьому дані можуть бути з роздільниками або без роздільників, тобто зміст файлу може мати якусь структуру. Структурною одиницею вмісту в подібних файлах, як правило, є рядок. Прикладами таких файлів є текстові файли й файли ініціалізації програм.

Файли довільного доступу. Це структуровані файли, які містять інформацію у вигляді записів. Прикладом можуть бути файли баз даних.

Двійкові (бінарні) файли. Файли з побайтовим доступом. У принципі, це ті ж файли з послідовним доступом, але інформація в них не організована в рядки. Особливість даних файлів — робота з байтами або блоками байтів. До таких файлів можна віднести виконувані програми, файли динамічних бібліотек, файли документів Word.

Подібний розподіл файлів на типи досить умовний й визначається особливостями організації файлів і доступу до даних. Наприклад, файл із послідовним доступом можна відкрити й у режимі двійкового доступу. Якщо цей файл має роздільників, то для роботи з ним доведеться написати спеціальну процедуру обробки роздільників і розбору даних, тому що двійковий доступ забезпечує по байтовий запис/читання з файлу. Очевидно, що це незручно. Саме тому й уведено умовний розподіл файлів на типи залежно від формату файлу

й доступу до даних. Відповідно згруповані й функції Visual Basic для запису/читання даних.

Тип файлу задає оптимальний набір функцій запису й читання даних з файлу. Тому при роботі з файлами для написання ефективної програми завжди необхідно мати інформацію про типи файлів, з якими буде працювати програма, і про організації зберігання даних у цих файлах. Це дає можливість забезпечити оптимальний доступ і використати відповідні цьому доступу функції.

Традиційний підхід при роботі з файлами

Традиційний підхід при роботі з файлами залишається незмінним практично з перших версій Visual Basic і полягає у використанні функцій і операторів, що забезпечують прямий доступ до інформації у файлах. Функції й оператори, використовувані при роботі з файлами, наведені в табл. 8.1. У стовпці **Тип файлу** таблиці прийняті такі скорочення типів файлів:

- П — файл послідовного доступу;
- Д — файл довільного доступу;
- Б — бінарний файл.

Ми розглянемо тільки основні функції й оператори, необхідні для одержання навичок роботи з файлами.

Таблиця 8.1

Функції й оператори для роботи з файлами

Функція, оператор	**Опис**	**Тип файлу**
`Open`	Відкриває файл	П, Д, Б
`Close`	Закриває всі файли	П, Д, Б
`Close #`	Закриває файл по ідентифікатору (дескриптору)	П, Д, Б
`Reset`	Закриває всі відкриті файли, записує вміст буферів	П, Д, Б
`Print tt`	Записує дані у файл	П
`FileCopy`	Копіює файл	П, Д, Б
`EOF`	Визначає мітку кінця файлу	П, Д, Б
`FileAttr`	Повертає режим доступу відкритого файлу	П, Д, Б

Продовження табл. 8.1

Функція, оператор	**Опис**	**Тип файлу**
`FileDateTime`	Повертає дату й час створення файлу	П, Д, Б
`FileLen`	Повертає розмір файлу в байтах	П, Д, Б
`FreeFile`	Повертає номер вільного ідентифікатора файлу (дескриптора)	П, Д, Б
`GetAttr`	Одержує атрибути файлу	П, Д, Б
`SetAttr`	Встановлює атрибути файлу	П, Д, Б
`Loc`	Повертає номер поточної позиції у файлі	Д, Б
`LOF`	Повертає розмір відкритого файлу в байтах	П, Д, Б
`Seek`	Встановлює на задану номером позицію або запис у файлі	П, Д, Б
`Dir`	Повертає вміст поточної папки	П, Д, Б
`Kill`	Видаляє файл	П, Д, Б
`Lock`	Блокує файл при роботі в багатокористувацькому середовищі	П, Д, Б
`Unlock`	Знімає блокування файлу в багатокористувацькому середовищі	П, Д, Б
`Name`	Задає (перейменовує) ім'я файлу	П, Д, Б
`Get #`	Читає дані з файлу	Д, Б
`Input`	Читає дані з файлу	П, Б
`Input #`	Читає дані з файлу	П
`Line Input #`	Читає рядок з файлу	П
`Put #`	Записує дані у файл	Д, Б
`Write #`	Записує дані у файл	П

Для зручності згрупуємо функції й оператори за виконуваною дією, як це прийнято у Visual Basic. Таке об'єднання зручно при виборі функції або оператора для виконання необхідних дій з файлами (табл. 8.2).

Таблиця 8.2

Функції й оператори для роботи з файлами по групах

Виконувана дія	Функції, оператори
Відкрити або створити файл	`Open`
Закрити файл	`Close, Reset`
Визначення параметрів виведення даних	`Format, Spc, Tab, Width #`
Скопіювати файл	`FileCopy`
Одержати інформацію про файл	`EOF, FileAttr, FlleDateTime, FileLen, FreeFile, GetAttr, Loc, LOF`
Організувати управління файлами	`Dir, Kill, Lock, Unlock, Name`
Прочитати дані з файлу	Get #, Input, Input #, Line Input #
Одержати інформацію про розмір файлу	FileLen
Встановити атрибути файлу	SetAttr
Знайти позиції у файлі	Seek
Записати дані у файл	Print #, Put #, Write #

Відкриття файлів

Як було зазначено вище, робота з кожним з типів файлів має свої особливості. Однак є дві дії, загальні для всіх типів файлів — їхнє відкриття й закриття.

Зрозуміло, що перед тим як записати дані у файл або прочитати дані з файлу, необхідно спочатку відкрити файл. Відкриття файлу виконується оператором `Open`:

```
Open pathName For mode [access] [lock] As [#]fileNumber
  [Len=recLength],
```

де

- `pathName` — повне ім'я файлу;
- `mode` — режим доступу до файлу. Може набувати значення: **Append, Binary, Input, Output або Random;**
- `access` — тип доступу до файлу. Визначає характер дій з файлом — читання або запис даних. Може набувати

значення: **Read** (Читання), **Write** (Запис) або **Read/Write** (Читання/Запис);

- `lock` — тип дозволу доступу до файлу іншим процесам. Визначає можливість одночасної роботи з файлом декількох додатків або декількох користувачів. Може набувати значення: **Shared** (Загальний), **Lock Read** (Блокування читання), **Lock Write** (Блокування запису) або **Lock Read Write** (Блокування читання й запису);
- `fileNumber` — цілочисловий вираз, що задає ідентифікатор файлу (дескриптор). Може мати значення від 1 до 511 включно;
- `recLength` — число, що визначає розмір буфера даних для запису/читання у файлах прямого доступу. Для файлів довільного доступу це число задає довжину одного запису файлу. Може мати значення до 32,767 (байтів).

При роботі оператора `Open` створюється спеціальний лічильник номерів (ідентифікаторів) відкритих файлів (в операторі це параметр `fileNumber`) для однозначного визначення файлу, з яким програма працює в даний момент.

Якщо зазначений в операторі `Open` файл не знайдений за заданим шляхом або не існує, він буде створений цим оператором для режимів доступу `Append, Binary, Output або Random`. Для режиму доступу Input новий файл не створюється.

Важливою особливістю режимів доступу є можливість багаторазового відкриття файлу для режимів `Binary, Input і Random`, тобто файл можна відкрити кілька разів з різними номерами. Але для режимів доступу `Output і Append,` використовуваних для запису даних, це неприпустимо.

Якщо номер файлу, що відкриває, спеціально не контролюється й не задається програмою, його можна довідатися за допомогою функції `FreeFile,` що повертає останній вільний номер файлу, що відкриває.

Закриття файлів

Закриття файлів виконується дуже просто. Для цього необхідно використати оператор `Close`, що має синтаксис:

```
Close [fileNumberLlist],
```

де `fileNumberLlist` — список файлів, що закривають, представляють номерами файлів і перераховують через кому: `[#fileNumber] [,#fileNumber] . . . [,#fileNumber]`. При цьому номер файлу аналогічний номеру файлу у функції `Open`.

Необхідно мати на увазі, якщо список файлів не зазначений, оператор Close закриває всі відкриті файли.

Робота з файлами послідовного доступу

Файли послідовного доступу — це, як правило, текстові файли, тобто послідовності ASCII-символів, організовані в рядки. Прикладом може бути файл ініціалізації Windows NT (рис. 8.1).

При відкритті файлів послідовного доступу можливі три режими доступу:

- `Input` — відкритий для послідовного читання даних;
- `Output` — відкритий для послідовного запису даних, при цьому інформація записується завжди з початку файлу (попередня затирається, якщо у файлі вже щось записане);
- `Append` — відкритий для додавання даних до вже наявних у файлі.

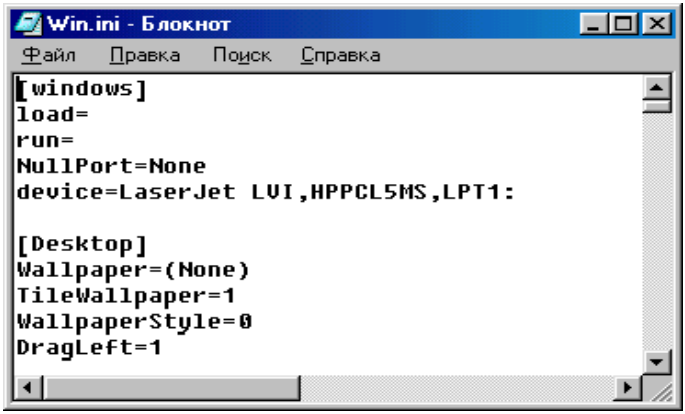


Рис. 8.1. Приклад файлу послідовного доступу

Читання даних

Читання даних з файлу послідовного доступу виконується за допомогою функції `Input` і операторів `Input #` і `Line Input #`. Розглянемо ці функції й оператори. Функція `Input` має такий синтаксис:

```
Input(number, #fileNumber),
```

де

- `number` — ціле число, що задає кількість зчитуваних з файлу символів;
- `fileNumber` — номер файлу, аналогічний номеру файлу в операторі `Open`.

Функція `Input` зчитує з файлу задану кількість символів і зазвичай використається для читання даних, записаних у файл оператором `Print #`.

Варто мати на увазі, що функція Input вимагає знання кількості зчитуваних символів. Тому для читання даних з файлу необхідно попередньо обчислити його довжину за допомогою функції `FileLen`.

Якщо в програмі потрібно прочитати дані з файлу, у якому інформація в рядках має структуру з роздільниками (як роздільники використовуються коми), необхідно застосовувати оператор `Input #`:

```
Input #fileNumber, varlist,
```

де

- `fileNurnber` — номер файлу, аналогічний номеру файлу в операторі `Open`;
- `varlist` — список змінних.

При роботі цього оператора спочатку зчитується рядок повністю, а потім підрядки, відділені роздільниками (комами), поміщаються у відповідні змінні списку. Для коректної роботи оператора рядки файлу повинні мати задану структуру з роздільниками. Зазвичай цей оператор використовується в парі з оператором запису Write #.

Для читання всього вмісту файлу за допомогою оператора Input # необхідно організувати циклічне зчитування даних з файлу, оскільки дані зчитуються цим оператором по рядках.

Для порядкового читання даних з послідовного файлу застосовується оператор `Line Input #`. Синтаксис цього оператора такий:

```
Line Input # fileNurnber, varName,
```

де

- `fileNurnber` — номер файлу, аналогічний номеру файлу в операторі `Open`;
- `varName` — ім'я змінної.

Оператор `Line Input #` посимвольно зчитує весь рядок даних з файлу й поміщає його в строкову змінну. При цьому роздільником рядків у файлі є стандартний роздільник рядків — символ повернення каретки CHR(13) або послідовність символів повернення каретки й переведення рядка CHR(13) + CHR(10), причому в змінну `varName` ці роздільники не вставляються. Зазвичай оператор `Line Input #` використовується в парі з оператором `Print #`.

Для того щоб прочитати всі дані з файлу за допомогою оператора `Line Input #`, необхідно організувати цикл читання даних.

Для вивчення функцій і операторів, призначених для роботи з файлами, створимо невеликий додаток. Інтерфейс додатка для вивчення функцій роботи з файлами показаний на рис. 8.2.

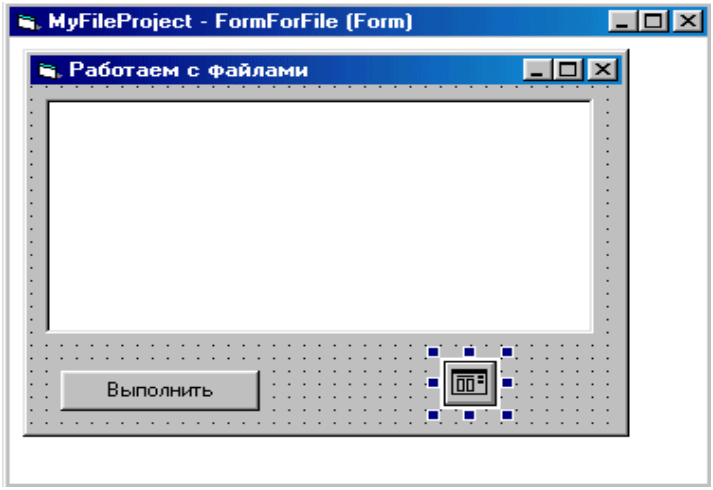


Рис. 8.2. Вид додатка для вивчення функцій роботи з файлами

Отриманий додаток можна тепер використати для вивчення роботи з файлами. Запрограмуємо додаток на читання даних з файлу послідовного доступу. Для цього відкриємо вікно редактора коду й введемо такий код:

```
Dim strFileName As String
Dim strFileContent As String
Dim nFreeFile As Integer
Dim nFileLenght As Integer
Private Sub cbStart_Click()
cdCommonDialog.ShowOpen
strFileName = cdCommonDialog.FileName
nFreeFile = FreeFile
If strFileName <> "" Then
Open strFileName For Input As nFreeFile
nFileLenght = FileLen(strFileName)
strFileContent = Input(nFileLenght, #nFreeFile)
txtFile.Text = strFileContent
Close
End If
End Sub
```

Працюючий додаток показаний на рис. 8.3. У цьому випадку при натисканні кнопки **Виконати** відкривається діалогове вікно пошуку файлу. Шлях і ім'я файлу, що повертається діалоговим вікном, зберігаються у змінній `StrFileName`. Знайдений файл (він повинен бути текстовим) відкривається за допомогою оператора `Open` для читання даних. Функція Input зчитує відразу весь файл у змінну `StrFileContent`, зміст якої потім поміщається в текстове поле `txtFile`. Після виконання всіх дій файл закривається оператором `Close`.

Щоб прочитати дані за допомогою оператора `Line Input #`, необхідно замінити текст коду, розташований відразу після оператора `Open`, на такий:

```
Do While Not EOF(nFreeFile)
Line Input #nFreeFile, strFileContent
txtFile.Text = txtFile.Text + strFileContent +Chr$(13)
  + Chr$(10) Loop Close
```

У цьому циклі дані з файлу по рядках зчитуються в змінну `Strfilecontent` і заносяться в текстове поле `txtFile`. Цикл працює до досягнення кінця файлу, що контролюється функцією `EOF`.

Застосування циклу не вимагає знання довжини файлу, що іноді важливо.

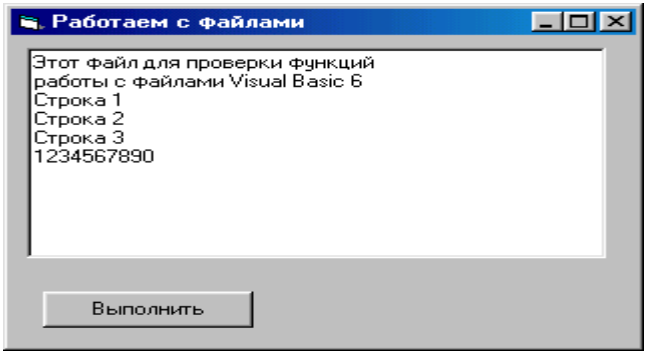


Рис. 8.3. Зчитування даних з файлу

Для вивчення роботи оператора `Input #` додайте у форму ще одне текстове поле. Назвіть його `txtNumber` і у властивість поля `DataFormat` занесіть значення **Number.** Текст у додатку після оператора Open необхідно замінити на такий:

```
Dim nFileContent As Integer
Input #nFreeFile, strFileContent, nFileContent
txtFile.Text = strFileContent
txtNumber.Text = nFileContent
Close
```

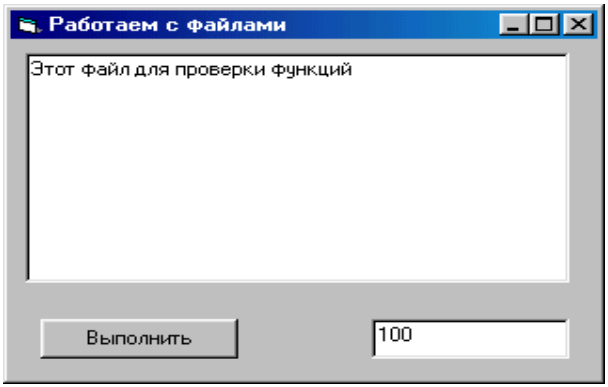


Рис. 8.4. Зчитування даних з файлу в змінні

Працюючий додаток показано на рис. 8.4. При цьому рядки файлу, що відкривається, мають структуру, аналогічну такому рядку:

```
Це файл для перевірки функцій, 100,
```
тобто текст і число розташовані через кому.

Перехід на задану позицію у файлі

Перехід на задану позицію у файлі можна організувати за допомогою оператора `Seek`, що має наступний синтаксис:

```
Seek #fileNumber, position,
```

де

- `fileNumber` — номер файлу, аналогічний номеру файлу в операторі `Open`;
- `position` — цілочисловний вираз, що задає позицію покажчика у файлі.

Позиціонування при цьому виконується посимвольно. Оператор `Seek` встановлює покажчик на необхідну позицію. Якщо після цього виконати функції читання або запису, то дія цих функцій буде починатися з позиції покажчика, знайденого оператором `Seek`.

Можна в код додатка безпосередньо перед функцією Input вставити вираз, що задає позиціонування:

```
Seek #nFreeFile, 77
nFileLenght = nFileLenght – 76
```

У цьому випадку дані з файлу будуть зчитані, починаючи з 77 числом символу, від початку й до кінця файлу. При цьому необхідно зменшити кількість символів, які зчитуються, оскільки зчитується весь файл.

Запис даних

Дані у файл послідовного доступу записуються за допомогою операторів `Print #` і `Write #`. Оператор `Write #` додатково автоматично вставляє у файл роздільники, але не дає гнучкості при управлінні форматуванням даних. Як можна було помітити при вивченні функцій читання даних, кожна з функцій запису працює в парі з певною функцією читання. Для оператора `Print #` це функції

`Input` або `Line Input #`, а для оператора `Write #` — це `Input #`. Рекомендується при застосуванні операцій запису/читання даних з файлів використати саме такі пари функцій і операторів запису/читання.

Розглянемо оператор `Print #`. Він має такий синтаксис:

```
Print # fileNumber, [outputlist],
```

де

- `fileNumber` — номер файлу, аналогічний номеру файлу в операторі `Open`;
- `outputlist` — список виразів або змінних для запису.

Оператор `Write #` має такий синтаксис:

```
Write # fileNumber, [outputlist],
```

де

- `fileNumber` — номер файлу, аналогічний номеру в операторі `Open`;
- `outputlist` — список виразів або змінних для запису.

При виконанні запису даних у файл оператор `Write #` після кожного рядка автоматично вставляє символ переведення каретки CHR(13) і символ нового рядка CHR(10).

Замінимо код у нашому додатку **MyFileProject** на такий:

```
Dim strFileName As String
Dim strFileContent As String
Dim nFreeFile As Integer
Private Sub cbStart_Click()
nFreeFile = FreeFile
cdCommonDialog.ShowSave
strFileName = cdCommonDialog.FileName
If strFileName <> "" Then
Open strFileName For Output As nFreeFile
strFileContent = txtFile.Text
Print #nFreeFile, strFileContent
Close
End If
End Sub
```

В отриманому додатку дані, уведені в текстове поле, будуть записуватися оператором `Print #` у файл, знайдений за допомогою діалогового вікна збереження файлів.

Робота з файлами довільного доступу

Файл із довільним доступом має заздалегідь задану структуру й складається із записів. Кожний запис у файлі — це деяка порція даних, що має строго визначений розмір і свій конкретний номер у файлі. Доступ до даних у файлі довільного доступу здійснюється саме за номером запису. Дані з файлу такого типу читаються й записуються записами. Прикладами файлу довільного доступу є бази даних, що завжди мають строго визначену структуру.

При відкритті файлів довільного доступу можливий тільки один режим доступу — `Random`. До речі, цей режим є режимом за умовчанням для функції `Open`.

Використовуючи можливості Visual Basic, можна створити файл довільного доступу користувацької, тобто своєї власної структури. Продемонструємо це на невеликому прикладі. За допомогою оператора `Type` оголосимо тип змінної, що має задану структуру запису:

```
Type PhisFace
PhisFaseID As Integer
FIO As String * 50
End Type
```

У даному прикладі оголошена структура у вигляді запису із двох полів. Першим полем є ідентифікатор, а другим — прізвище, ім'я та по батькові. Оголошення користувацького типу даних необхідно здійснювати в програмному модулі.

Відкриття файлу довільного доступу

Файл довільного доступу відкривається трохи інакше, ніж файл послідовного доступу. Синтаксис оператора `Open` при цьому виглядає в такий спосіб:

```
Open pathName [For Random] As fileNumber Len =
  recLength,
```

де

- `pathName` — повне ім'я файлу;
- `fileNumber` — номер файлу;

- `recLength` — довжина запису в байтах.

При використанні оператора `Open` для відкриття файлу довільного доступу атрибут `For` не обов'язковий, тому що у Visual Basic цей параметр установлюється за умовчанням. Як видно із синтаксису, на відміну від файлу з послідовним доступом, при відкритті файлу з довільним доступом необхідно обов'язково вказувати довжину запису. При цьому, якщо довжина запису невідома, її можна обчислити з використанням функції `Len`.

Читання даних з файлу довільного доступу

Дані з файлу довільного доступу, як правило, зчитуються записами. Для цього використається оператор `Get #`, що має такий синтаксис:

```
Get #fileNumber, [recNumber], varName,
```

де

- `fileNumber` — номер файлу;
- `recNumber` — номер запису у файлі;
- `varName` — змінна.

Якщо параметр `recNumber` у функції `Get` не зазначений, зчитується поточний запис, на якому позиціонований покажчик.

Для позиціонування покажчика можна використати функцію `Seek`. Синтаксис цього оператора такий же, як для файлів послідовного доступу, але має інший зміст. Якщо для послідовних файлів позиціонування виконується по символах, то для файлів довільного доступу — за номером запису:

```
Seek #fileNumber, position,
```

де

- `fileNumber` — номер файлу;
- `position` — цілочисловний вираз, що задає номер запису у файлі.

Запис у файл довільного доступу

Для запису даних у файл довільного доступу використовується оператор `Put #`, що має такий синтаксис:

```
Put #fileNumber, [recNumber], varName,
```

де

- `fileNumber` — номер файлу, аналогічний номеру в операторі `Open`;
- `recNumber` — цілочисловий вираз, що задає номер запису у файлі;
- `varName` — змінна, що вказує джерело записуваних даних.

Цей оператор використовується тільки для файлів довільного доступу й бінарних. Якщо номер запису не зазначений, то за умовчанням береться поточна позиція покажчика запису.

При використанні оператора `Put` необхідно мати на увазі, що дані в записі із зазначеним в операторі номером будуть замінені на ті, які ми записуємо у файл. Додавання записів виконується за допомогою цього ж оператора, але з деякими особливостями.

Зміна даних у файлі довільного доступу

Для зміни даних у записах файлу (редагування, додавання, видалення записів) застосовується оператор `Put #`. При його використанні необхідно мати на увазі, що дані в записі будуть замінені на ті, які ми передаємо у файл. Підкреслимо, що новий запис із даними не створюється.

Для додавання записів у файл необхідно вказувати номер запису на одиницю більший номера останнього запису. У цьому випадку запис буде доданий у файл, а не змінений. Наприклад:

```
Put #  FileNum, LastRecord + 1, ForFileRecords
```

Для обчислення поточного номера останнього запису `LastRecord` можна використати довжину запису й розмір файлу, що повертає функцією `LOF`.

Перейдемо до опису процесу видалення даних з файлу довільного доступу. Існують два способи. Можна просто очистити відповідні поля зазначених записів, тобто записати в них порожні значення. Однак у цьому випадку у файлі залишаються порожні

записи. Зрозуміло, що при такому підході ресурси (дисковий простір) використовуються нераціонально.

Для остаточного видалення записів рекомендується перезаписувати дані в новий файл, пропускаючи порожні записи. Алгоритм цих дій такий:

1. Створіть новий файл за допомогою оператора `Open`.
2. Перепишіть всі непусті записи в новий файл, використовуючи оператор
3. `Put #`.
4. Закрийте вихідний файл і видаліть його за допомогою оператора `Kill`.
5. Перейменуйте новий файл у вихідний оператором `Name`.

Одержуємо той же файл, але вже без порожніх записів. При цьому заощаджується простір диска й час пошуку даних у такому файлі.

Робота із двійковими файлами

Двійковий файл у корені відрізняється від файлів послідовного й довільного доступу. Подивіться на рис. 8.5. Тут як приклад двійкового файлу за допомогою редактора **Notepad** системи Windows відкритий виконуваний файл Visual Basic.

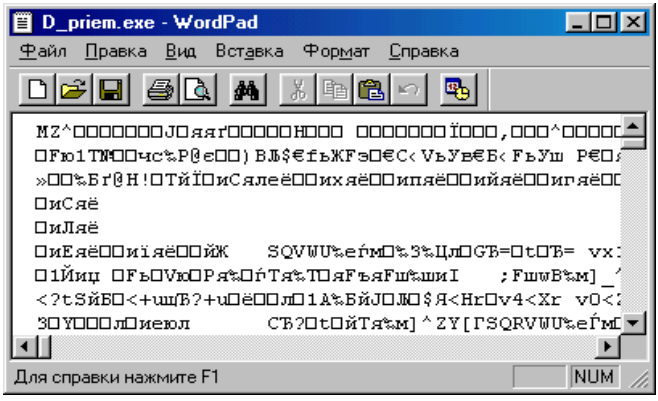


Рис. 8.5. Приклад двійкового файлу

З рис. 8.5 видно, що нема рації переглядати файл у чистому виді, його розуміє тільки спеціальна програма. Двійковий файл не організований у рядки, як файл послідовного доступу, представлений раніше на рис. 8.1. Тут не можна виділити рядок, запис або іншу інформаційну структуру, крім байтів або блоків байтів.

Двійковий файл відкривається тільки у двох режимах:

- `Binary` — двійковий доступ за номером байта;
- `Random` — довільний доступ.

Для режиму доступу `Random` до двійкових файлів, на відміну від файлів довільного доступу, запис не має строго заданого розміру й звичайно обчислюється за відповідним алгоритмом або зберігається в записах файлу.

Робота із двійковими файлами відрізняється більшою свободою виконання різних дій, ніж з файлами послідовного доступу. На відміну від файлів послідовного доступу, двійковий файл відкривається відразу й для читання, і для запису.

Відкриття двійкового файлу

Двійковий файл відкривається по-іншому, ніж файл послідовного доступу. Синтаксис оператора `Open` для двійкового файлу виглядає в такий спосіб:

```
Open pathName For Binary As fileNumber,
```

де

- `pathName` — повне ім'я файлу;
- `fileNumber` — номер файлу.

Читання даних із двійкових файлів

Читання даних із двійкових файлів виконується за допомогою оператора `Get #`, що має такий синтаксис:

```
Get #fileNumber, [recNumber], varName,
```

де

- `fileNumber` — номер файлу;
- `recNuinber` — розмір запису в байтах;
- `varName` — змінна, у яку читаються дані з файлу.

Запис даних у двійкові файли

Для запису даних у двійкові файли використовується оператор `Put #`, що має такий синтаксис:

```
Put [#]fileNumber, [recNumber], varName,
```

де

- `fileNumber` — номер файлу;
- `recNumber` — розмір запису в байтах;
- `varName` — змінна, у якій зберігаються дані, записувані у файл.

Позиціонування у двійковому файлі виконується так само, як у файлі послідовного доступу.

Робота з атрибутами файлів

Для роботи з атрибутами файлів використовуються функції `GetAttr i SetAttr`. Синтаксис функції `GetAttr`, що повертає атрибути файлів, такий:

```
GetAttr (pathName),
```

де `pathName` — шлях, що включає ім'я файлу, папку, пристрій. За умовчанням застосовується поточна папка й пристрій.

Функція `GetAttr` повертає число, що визначає атрибути файлу. Інтерпретувати це число можна за допомогою параметрів, описаних у табл. 8.3.

Таблиця 8.3

Константи атрибутів файлів

Константа	**Опис**	**Значення**
vbNormal	Звичайний файл	0
vbReadOnly	Файл тільки для читання	1
vbHidden	Схований файл	2
vbSystem	Системний файл	4
vbDirectory	Каталог(папка)	16
vbArhive	Архівний файл	32

Аналогічно працює функція Setattr, що виконує установлення атрибутів файлів. Синтаксис цієї функції такий:

```
SetAttr pathName, attributes,
```

де

- `pathName` — шлях, що включає ім'я файлу, папку, пристрій. За умовчанням використовується поточна папка й пристрій;
- `attributes` — число або вираз, що задає суму атрибутів.

У функції `SetAttr` використовуються всі константи, описані в табл. 8.3, крім константи `vbDirectory`.

Для того щоб задати кілька атрибутів, можна просто додати зазначені константи. Наприклад, для установлення атрибутів файлу `Hidden i Read—only` необхідно використати функцію `Setattr` такого виду:

```
SetAttr "MyFile", vbHidden + vbReadOnly
```

При цьому для файлу **MyFile** будуть установлені атрибути `Hidden i Read—only`.

Робота з папками й пристроями

Файл перебуває на самому нижньому рівні зберігання інформації у файловій системі комп'ютера. Вище в ієрархії файлової системи розташовані папки й пристрої. Під пристроєм розуміється не тільки твердий диск, але й, наприклад, пристрій для читання CD-ROM.

Для операцій з папками й пристроями у Visual Basic існує набір функцій і операторів (табл. 8.4), які дозволяють створювати й видаляти папки, перейменовувати їх, розкривати їхній зміст.

Таблиця 8.4

Функції й оператори для роботи з папками й пристроями

Функція, оператор	**Виконувана дія**
ChDir	Змінює поточну папку
ChDrive	Змінює поточний пристрій
MkDir	Створює папку
RmDir	Видаляє папку

Продлвження табл. 8.4

Функція, оператор	Виконувана дія
Name	Надає папці ім'я або перейменовує папку
CurDir	Повертає поточний каталог
Dir	Повертає список файлів папки

Із усього списку функцій для роботи з папками однією із найбільш корисних є функція `Dir`. Дія цієї функції нагадує аналогічну команду DOS. За допомогою цієї функції можна одержати список файлів зазначеної папки. При цьому можна застосовувати шаблон імені файлу, використовуючи типові позначення "*" — множинна підміна, "?" — підміна одного символу.

Функція `Dir` має такий синтаксис:

```
Dir(pathName[, attributes]),
```

де

- `pathName` — шлях, що включає ім'я файлу, папку, пристрій. За умовчанням застосовується поточна папка й пристрій. Для позначення імені можна застосувати шаблон за аналогією з DOS. Якщо файл не знайдений, то вертається порожній рядок;
- `attributes` — атрибути файлів. Мають значення, зазначені в табл. 8.5. Якщо атрибути не призначені, то за умовчанням вертаються файли без атрибутів.

Указуючи атрибути для функції `Dir`, ми призначаємо фільтр для списку виведених файлів. Для того щоб призначити одночасно кілька атрибутів, їх можна просто арифметично додати.

Указуючи конкретне значення файлу, функцію `Dir` можна використати для підтвердження існування зазначеного файлу на диску. Якщо зазначений файл не існує, то вертається порожній рядок.

Таблиця 8.5

Константи атрибутів файлів функції Dir

Константа	Значення	Опис
vbNormal	0	Задає файли без атрибутів
vbReadOnly	1	Задає файли тільки для читання
vbHidden	2	Задає сховані файли
vbSystem	4	Задає системні файли
vbVolume	8	Задає мітку тому
vbDirectory	16	Задає папку (каталог)

Коротко опишемо синтаксис ще декількох корисних функцій, використовуваних при роботі з папками.

Для створення папки застосовується оператор `MkDir`, синтаксис якого виглядає в такий спосіб:

```
MkDir path
```

де `path` — шлях, що включає ім'я файлу, папку, пристрій. За умовчанням використається поточна папка й пристрій. Якщо параметр `path` не містить ім'я пристрою, папка буде створена на поточному пристрої.

Для видалення папок використається оператор `RmDir`, що має наступний синтаксис:

```
RmDir path
```

де `path` — шлях, що включає ім'я файлу, папку, пристрій. За умовчанням використовується поточна папка й пристрій. При використанні оператора `RmDir` необхідно мати на увазі, що якщо видаляється папка, що, містить файли, буде видаватися системне повідомлення про помилку. Щоб уникнути цього, рекомендується спочатку видалити всі файли з папки за допомогою оператора `Kill` і лише потім видалити зазначену папку.

Для вибору іншого пристрою використовується оператор `chDrive`, що має такий синтаксис:

```
ChDrive drive,
```

де `drive` — рядковий вираз, що призначає новий пристрій.

Параметром цієї функції є літерне позначення пристрою вашої системи. При використанні цієї функції необхідно мати на увазі, що пристрій повинен існувати й бути доступним.

Організація друку

Друк тексту, тобто виведення даних на принтер, можна організувати за допомогою файлового оператора `Print #`. Дійсно, з погляду виведення даних, у принципі однаково, куди пересилати інформацію. Необхідно тільки правильно вказати її одержувача.

Для пересилання даних на принтер використовується пряме призначення порту принтера (LPT1, LPT2) як одержувача даних. Це можна зробити за допомогою оператора відкриття файлу `Open`:

```
Open "LPT 1" For Output As #nPrinterHandle
```

Після виконання цього оператора для адресації даних на принтер необхідно використати дескриптор (ідентифікатор файлу) `#nprinterHandle`. Якщо в додатку не підтримується обчислення ідентифікатора файлів, то варто використати функцію `FreeFile` для його обчислення. Аналогічно можна направити дані в будь-який інший порт комп'ютера або мережі.

Після того як порт відкритий для прийому даних, можна використати оператор `Print #`. Такий вираз посилає на принтер дані для друку:

```
Print #nPrinterHandle, strExpression
```

У цьому виразі `Strexpression` задає текст, який друкується.

Використовуючи функції й оператори читання даних з файлу, можна організувати циклічне порядкове виведення файлу на друк.

Однак цей метод організації виведення даних на друк має свої тонкості. Всі операції роботи принтера (Позиціонування друківної голівки, переведення рядка, переведення сторінки та ін.) тепер буде потрібно програмувати за допомогою спеціальних операторів, які розуміє принтер. Такі оператори називаються Esc-кодами (Esc-послідовностями). Їхній опис додається до кожного принтера й тут ми не будемо їх розглядати.

Після завершення друку даних порт закривається оператором `Close`, що також використається для закриття файлу:

```
Close #nPrinterHandle
```

або

```
Close
```

При використанні оператора `Close` без дескриптора одночасно з портом закриваються й файли, з яких виводилася інформація.

Резюме

Файли використовують для тривалого зберігання даних, при інсталяції розроблених додатків, при ініціалізації додатків з використанням файлів настроювання. Файл зберігається на диску, кількість даних в файлі при його описуванні не вказується, елементи файлів не мають індексів, тому робота з файлами - важливий навик, який потрібний при розробці програмного забезпечення. Традиційний підхід при роботі з файлами у Visual Basic полягає у використанні функцій і операторів, що забезпечують доступ до інформації у файлах. При роботі з файлами необхідно мати інформацію про тип

файлів і організацію зберігання даних у файлах - це забезпечує використання відповідних функцій обробки файлів.

Контрольні запитання та завдання

1. Означити поняття файл. Чим схожі файли і масиви? Чим вони відрізняються?
2. Які дані можуть бути елементами файлу?
3. Чим визначається поняття тип файлу у Visual Basic?
4. Які типи файлів існують?
5. Чим відрізняються файли з довільним і з послідовним доступом?
6. У чому полягають традиційні підходи при роботі з файлами?
7. Чим зумовлена потреба у відкритті та закритті файлів?
8. Чим текстовий файл відрізняється від типізованого?
9. Створіть типізований файл з інформацією про книжки з програмування.
10. Поясніть різницю між функціями `FileLen` і `Lof`?
11. Як працюють функції `Close, Close #, Reset.`
12. Як одержати інформацію про файл?
13. У чому особливість двійкових (бінарних) файлів, коли їх використовують?
14. Як організується зміна даних у файлах довільного доступу?
15. Як пересилаються дані із файлу на принтер?
16. Створити два файли цілих чисел. Відсортувати їх вміст. Відсортовані файли злити в один впорядкований файл.
17. Задано два текстових файли. Видалити з цих файлів рядки, що мають однакові номери, але самі не є однаковими. Результати записати в нові файли.
18. Записати в перевернутому вигляді рядки файлу f в файл g. Порядок проходження рядків у файлі повинен бути зворотним по відношенню до порядку рядків вихідного файлу.
19. Переписати компоненти файлу f в файл g, вставляючи в початок кожного рядка порядковий номер рядка. Порядок проходження компонент зберегти.

20. Отримати всі рядки файлу f, що містять як фрагмент рядок S.
21. Виключити з кожного рядка файлу d символи, відмінні від букв і цифр. Результат записати в файл g.
22. Обчислити для кожного рядка текстового файлу кількість відкритих і закритих дужок і дописати обчислені значення в кінець кожного рядка. Результати записати у новий файл.
23. Вилучити коментарі з написаної на Visual Basic програми.
24. У файлі зберігаються відомості про поставки товарів: найменування товару, вартість, одиниця вимірювання, кількість, країна поставки. Розробити програму, яка дозволяє одержувати список країн, що імпортують вказаний товар у вказаному обсязі.
25. Створити файл, що має такі поля: дисципліна, викладач, код групи, день тижня, номер пари, аудиторія. Розробити програму, яка дозволить друкувати розклад занять групи в заданий день тижня.
26. У файлі зберігатися інформація про пасажирів і багаж. Багаж пасажира характеризується кількістю речей і загальною вагою речей. Про пасажира відомо: ПІБ, номер рейсу, місто. Чи є пасажир, багаж якого перевищує багаж кожного з решти пасажирів за кількістю речей і за вагою?
27. Створити файл записів, що має такі поля: прізвище, ім'я, ім'я по батькові, рік вступу на роботу, домашня адреса, телефон. Видалити із файлу всі записи, в яких прізвище починається з заданої користувачем літери.
28. Створити файл дійсних чисел і переписати його компоненти у новий файл у зворотному порядку.

Розділ IX
ДИНАМІЧНІ СТРУКТУРИ У VISUAL BASIC

Існує чотири основні способи розподілу пам'яті у Visual Basic: оголошення змінних стандартних типів (цілі, із плаваючою крапкою і т.д.); оголошення змінних типів, які визначені користувачем; створення екземплярів класів за допомогою оператора `New` та зміна розміру масивів. Крім того, існує ще декілька способів, наприклад, створення нового екземпляра форми або елемента керування, але всі ці способи не дають великих можливостей при створенні складних структур даних.

Використовуючи ці методи, можна легко будувати статичні структури даних, такі як великі масиви, тип яких визначений користувачем. Можна змінювати розмір цих масивів за допомогою оператора `Redim`. Проте перерозподіл даних може бути досить складним. Наприклад, для того щоб перенести елемент із одного кінця масиву на іншій, потрібно буде переупорядкувати увесь масив, зрушивши всі елементи на одну позицію, щоб заповнити простір, який звільнився. І тільки потім можна помістити елемент на його нове місце.

Динамічні ж структури даних дозволяють швидко й легко виконувати такого роду зміни. Усього за кілька кроків можна перемістити будь-який елемент у структурі даних у будь-яке інше положення.

Цей розділ присвячений описанню методів створення динамічних списків у Visual Basic. Різні типи списків мають різні властивості. Деякі з них прості й мають обмежену функціональність, інші ж, такі як циклічні списки, одно - або двозв’язні списки є більш складними й підтримують більш розвинені засоби керування даними. Методи, які будуть описані, використовуються для побудови стеків, черг, масивів, дерев, хеш-таблиць і мереж. Вам необхідно засвоїти матеріал цього розділу перед тим, як продовжити читання інших.

Застосування динамічних структур даних

Динамічні структури характеризуються відсутністю фізичної суміжності елементів структури в пам'яті, непостійністю і непередбачуваністю розміру структури в процесі її обробки. Оскільки елементи динамічної структури розташовуються за непередбачуваними адресами пам'яті, адреса елемента такої структури не може бути обчислена з адреси початкового або попереднього елемента. Для встановлення зв'язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв'язки між елементами. Таке представлення даних в пам'яті називається зв'язним.

Переваги зв'язного подання даних – в можливості забезпечення значної змінності структур: розмір структури обмежується тільки доступним обсягом машинної пам'яті, при зміні логічної послідовності елементів структури потрібне не переміщення даних в пам'яті, а тільки корекція покажчиків, велика гнучкість структури.

Основні недоліки зв'язного подання: для організації зв'язку витрачається додаткова пам'ять, доступ до елементів зв'язної структури може бути менш ефективним за часом.

Зв'язне подання практично ніколи не застосовується в задачах, де логічна структура даних має вид вектора або масиву – з доступом за номером елемента, але часто застосовується в задачах, де логічна структура вимагає іншої початкової інформації доступу (списки, таблиці, дерева і т.д.).

Знайомство зі списками

Найпростіша форма списку - це група об'єктів. Вона містить у собі об'єкти й дозволяє програмі звертатися до них. Якщо це все, що вам потрібно від списку, ви можете використовувати масив як список, відслідковуючи за допомогою змінної `Numinlist` кількість елементів у списку. Визначивши за допомогою цієї змінної кількість наявних елементів, програма може потім по черзі звертатися до них у циклі `For` і виконувати необхідні дії.

Якщо ви у своїй програмі можете обійтися цим підходом, використовуйте його. Цей ефективний метод, який легко

підтримувати завдяки його простоті. Проте більшість програм не настільки проста і в них потрібні більш складні конструкції навіть для таких простих об'єктів, як списки. Тому в наступних підрозділах цього розділу обговорюються деякі шляхи створення списків з більшою функціональністю.

У даному розділі описуються шляхи створення списків, які можуть рости й зменшуватися згодом. Іноді заздалегідь не можна визначити, наскільки великий список знадобиться, у такій ситуації допоможе список, який при необхідності може змінювати свій розмір.

Можна організувати *неупорядковані списки* (`unordered list`), що дозволяють видаляти елементи з будь-якої частини списку. Неупорядковані списки дають більший контроль над вмістом списку, ніж прості. Вони є більш динамічними, тому що дозволяють змінювати вміст у довільний момент часу.

Для створення більш гнучких структур даних застосовують *зв'язні списки* (`linked list`), які використовують *покажчики*. Ви можете додавати або видаляти елементи з будь-якої частини зв'язного списку з мінімальними зусиллями. У даному розділі описані також інші різновиди зв'язних списків, такі як циклічні, двозв'язні або списки з посиланнями.

Прості списки

Якщо у вашій програмі необхідний список постійного розміру, ви можете створити його, просто використовуючи масив. У цьому випадку можна при необхідності опитувати його елементи в циклі `For`.

Багато програм використовують списки, які ростуть або зменшуються згодом. Можна створити масив, що відповідає максимально можливому розміру списку, але таке рішення не завжди буде оптимальним. Не завжди можна заздалегідь знати, наскільки великим може стати список, крім того, імовірність, що список стане дуже великим, може бути невелика й створений масив гігантських розмірів може більшу частину часу лише даремно займати пам'ять.

Колекції

Програма може використовувати колекції Visual Basic для зберігання списку змінного розміру. Метод `Add Item` додає елемент у колекцію. Метод `Remove` видаляє елемент. Наступний фрагмент коду демонструє програму, яка додає три елементи до колекції й потім видаляє другий елемент.

```
Dim list As New Collection
Dim obj As Myclass
Dim I As Integer
   ' Створити й додати 1 елемент.
   Set obj = New Myclass
   list.Add obj
   ' Додати ціле число.
   i = 13
   list.Add I
   ' Додати рядок.
   list.Add "Робота з колекціями"
   ' Вилучити 2 елемент (ціле число).
   list.Remove 2
```

Колекції намагаються забезпечити підтримку будь-яких додатків і виконують чудову роботу. Їх легко використовувати, вони дозволяють витягати елементи, які проіндексовані за ключем, і дають прийнятну продуктивність, якщо не містять занадто багато елементів.

Проте колекціям властиві й певні недоліки. Для великих списків колекції можуть працювати повільніше, ніж масиви. Якщо у вашій програмі не потрібні всі властивості, надавані колекцією, більш швидким може бути використання простого масиву.

Схема хешування, яку колекції використовують для управління ключами, також накладає ряд обмежень. По-перше, колекції не дозволяють дублювати ключі. По-друге, для колекції можна визначити, який елемент має заданий ключ, але не можна довідатися, який ключ відповідає даному елементу. І, нарешті, колекції не підтримують множинних ключів. Наприклад, може бути, що вам потрібно, щоб програма могла робити пошук за списком службовців, використовуючи ім'я співробітника або його ідентифікаційний номер у системі соціального страхування. Колекція не зможе підтримувати обидва методи пошуку, тому що вона здатна оперувати тільки одним ключем.

Далі описуються методи побудови списків, вільні від цих обмежень.

Список змінного розміру

Оператор Visual Basic `Redim` дозволяє змінювати розмір масиву. Ви можете використовувати цю властивість для побудови простого списку змінного розміру. Почніть із оголошення масиву без розміру для зберігання елементів списку. Також визначте змінну `Numinlist` для відстеження кількості елементів у списку. При додаванні елементів до списку використовуйте оператор `Redim` для збільшення розміру масиву, щоб новий елемент зміг вміститися в ньому. При видаленні елемента також використовуйте оператор `Redim` для зменшення масиву й вивільнення непотрібної більше пам'яті.

```
Dim List() As String              ' Список елементів.
Dim Numinlist As Integer  ' Число елементів у списку.
Sub Addtolist(value As String)
  ' Збільшити розмір масиву.
  Numinlist = Numinlist + 1
  Redim Preserve List (1 To Numinlist)
  ' Додати новий елемент до кінця списку.
  List(Numinlist) = value
End Sub
Sub Removefromlist()
  ' Зменшити розмір масиву, звільняючи пам'ять.
  Numinlist = Numinlist - 1
  Redim Preserve List (1 To Numinlist)
End Sub
```

Ця проста схема непогано працює для невеликих списків, але в неї є пара недоліків. По-перше, доводиться часто змінювати розмір масиву. Для створення списку з 1000 елементів доведеться 1000 разів змінювати розмір масиву. Гірше того, при збільшенні розміру списку на зміну його розміру буде потрібно більше часу, оскільки прийдеться щораз копіювати зростаючий список у пам'яті.

Для зменшення частоти змін розміру масиву можна додавати додаткові елементи до масиву при збільшенні його розміру, наприклад, по 10 елементів замість одного. При цьому, коли ви будете додавати нові елементи до списку в майбутньому, масив вже буде містити невикористовуванні комірки, у які ви зможете занести нові елементи без збільшення розміру масиву. Нове збільшення

розміру масиву буде потрібно тільки тоді, коли порожні комірки закінчаться.

Подібним чином можна уникнути зміни розміру масиву при кожному видаленні елемента зі списку. Можна почекати, доки в масиві буде 20 невикористовуваних комірок, перш ніж зменшувати його розмір. При цьому потрібно залишити 10 вільних комірок для того, щоб можна було додавати нові елементи без необхідності знову збільшувати розмір масиву.

Зазначимо, що максимальна кількість не використовуваних комірок (20) повинна бути більшою, ніж мінімальна кількість (10). Це зменшує кількість змін розміру масиву при видаленні або додаванні його елементів.

При такій схемі в списку звичайно є кілька вільних комірок, проте їх кількість достатньо мала, і зайві витрати пам'яті невеликі. Вільні комірки гарантують можливість додавання або видалення елементів без зміни розміру масиву. Фактично, якщо ви неодноразово додаєте до списку, а потім видаляєте з нього один або два елементи, вам може ніколи не знадобитися змінювати розмір масиву.

```
Dim List() As String              ' Список елементів.
Dim Arraysize As Integer   ' Розмір масиву.
Dim Numinlist As Integer   ' Число використовуваних
  елементів.
' Якщо масив заповнений, збільшити його розмір, додавши
  10 комірок.
' Потім додати новий елемент у кінець списку.
Sub Addtolist(value As String)
  Numinlist = Numinlist + 1
  If Numinlist > Arraysize Then
       Arraysize = Arraysize + 10
       Redim Preserve List(1 To Arraysize)
  End If
  List(Numinlist) = value
End Sub
' Вилучити останній елемент зі списку. Якщо залишилося
  більше
' 20 порожніх комірок, зменшити список, звільняючи
  пам'ять.
Sub Removefromlist()
  Numinlist = Numinlist - 1
  If Arraysize - Numinlist > 20 Then
       Arraysize = Arraysize -10
       Redim Preserve List(1 To Arraysize)
  End If
End Sub
```

Для дуже великих масивів це рішення може також виявитися не найкращим. Якщо вам потрібний список, що містить 1000 елементів, до якого звичайно додається по 100 елементів, то усе ще занадто багато часу буде витрачатися на зміну розміру масиву. Очевидною стратегією в цьому випадку було б збільшення розміру масиву з 10 до 100 або більше комірок. Тоді можна було б додавати по 100 елементів одночасно без частої зміни розміру списку.

Більш гнучким рішенням буде зміна збільшення залежно від розміру масиву. Для невеликих списків це збільшення було б також невеликим. Хоча зміни розміру масиву відбувалися б частіше, вони вимагали б відносно небагато часу для невеликих масивів. Для великих списків збільшення розміру буде більшим, тому їхній розмір буде змінюватися рідше.

Наступна програма намагається підтримувати приблизно 10 відсотків списку вільними. Коли масив заповнюється, його розмір збільшується на 10 відсотків. Якщо вільний простір становить більше ніж 20 відсотків від розміру масиву, програма зменшує його.

При збільшенні розміру масиву додається не менше 10 елементів, навіть якщо 10 відсотків від розміру масиву становлять меншу величину. Це зменшує кількість необхідних змін розміру масиву, якщо список дуже малий.

```
Const WANT_FREE_PERCENT = .1      ' 10% вільного місця.
Const MIN_FREE = 10               ' Мінімальне число
  порожніх комірок.
Global List() As String           ' Масив елементів
  списку.
Global Arraysize As Integer       ' Розмір масиву.
Global Numitems As Integer        ' Число елементів у
  списку.
Global Shrinkwhen As Integer      ' Зменшити розмір, якщо
  Numitems < Shrinkwhen.
' Якщо масив заповнений, збільшити його розмір.
' Потім додати новий елемент у кінець списку.
Sub Add(value As String)
  Numitems = Numitems + 1
  If Numitems > Arraysize Then Resizelist
  List(Numitems) = value
End Sub
' Вилучити останній елемент зі списку.
' Якщо в масиві багато порожніх комірок, зменшити його
  розмір.
```

```
Sub Removelast()
  Numitems = Numitems - 1
  If Numitems < Shrinkwhen Then Resizelist
End Sub
' Збільшити розмір масиву, щоб 10% комірок були вільні.
Sub Resizelist()
Dim want_free As Integer
  want_free = WANT_FREE_PERCENT * Numitems
  If want_free < MIN_FREE Then want_free = MIN_FREE
  Arraysize = Numitems + want_free
  Redim Preserve List(1 To Arraysize)
  ' Зменшити розмір масиву, якщо Numitems < Shrinkwhen.
  Shrinkwhen = Numitems - want_free
End Sub
```

Клас Simplelist

Щоб використати цей простий підхід, програмі необхідно знати всі параметри списку, при цьому потрібно стежити за розміром масиву, кількістю використовуваних елементів і т.і. Якщо необхідно створити більше одного списку, буде потрібно багато копій змінних і код, який управляє різними списками, буде дублюватися.

Класи Visual Basic можуть сильно полегшити виконання цього завдання. Клас `Simplelist` інкапсулює структуру списку та спрощує управління списками. У цьому класі присутні методи `Add` і `Remove` для використання в основній програмі. У ньому також є процедури витягу властивостей `Numitems` і `Arraysize`, за допомогою яких програма може визначити кількість елементів у списку й обсяг займаної ним пам'яті.

Процедура `Resizelist` оголошена як приватна усередині класу `Simplelist`. Це приховує зміну розміру списку від основної програми, оскільки цей код повинен використовуватися тільки усередині класу.

Використовуючи клас `Simplelist`, легко створити в додатку кілька списків. Для того щоб створити новий об'єкт для кожного списку, просто використовується оператор `New`. Кожний з об'єктів має свої змінні, тому кожний з них може управляти окремим списком:

```
Dim List1 As New Simplelist
Dim List2 As New Simplelist
```

Програма Simlist (диск з прикладами – папка ProgR9) демонструє використання класу Simplelist. Введіть значення в поле введення тексту й натисніть кнопку **Add** (Додати). Програма додасть елемент до списку. При необхідності об'єкт Simplelist збільшить розмір масиву. Ви зможете вилучити останній елемент за допомогою кнопки **Remove** (Вилучити), якщо список ще не порожній.

Коли об'єкт Simplelist збільшує масив, він виводить вікно повідомлення, де показано розмір масиву, кількість не використовуваних елементів у ньому і значення змінної Shrinkwhen. Коли кількість використаних комірок у масиві стає меншою, ніж значення Shrinkwhen, програма зменшує розмір масиву. Зазначимо, що коли масив практично порожній, змінна Shrinkwhen іноді дорівнює нулю або стає негативною. У цьому випадку розмір масиву не буде зменшуватися, навіть якщо ви вилучите всі елементи зі списку.

Програма Simlist додає до масиву ще 50 відсотків порожніх комірок, якщо необхідно збільшити його розмір, і завжди залишає при цьому не менше однієї порожньої комірки. Ці значення були обрані для зручності роботи із програмою. У реальному додатку відсоток вільної пам'яті повинен бути меншим, а кількість вільних комірок більшою. Більш розумним у такому випадку було б вибрати значення порядку 10 відсотків від поточного розміру списку й мінімум 10 вільних комірок.

Неупорядковані списки

У деяких додатках може знадобитися видаляти елементи із середини списку, додаючи при цьому елементи в кінець списку. У цьому випадку порядок розташування елементів може бути не важливий, але при цьому може бути необхідно видаляти певні елементи зі списку. Списки такого типу називаються *неупорядкованими списками* (unordered lists). Вони також іноді називаються «множиною елементів».

Неупорядкований список повинен підтримувати такі операції:

- додавання елемента до списку;
- видалення елемента зі списку;
- визначення наявності елемента в списку;
- виконання якихось операцій (наприклад, друку на дисплей або принтер) для всіх елементів списку.

Просту структуру, представлену в попередньому підрозділі, можна легко змінити для того, щоб обробляти такі списки. Коли видаляється елемент із середини списку, інші елементи зрушуються на одну позицію, заповнюючи проміжок, що утворився. Це показано на рис.9.1, де другий елемент видаляється зі списку, а третій, четвертий, і п'ятий елементи зрушуються вліво, заповнюючи вільну ділянку.

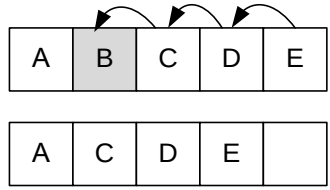


Рис.9.1. Видалення елемента з середини масиву

Видалення з масиву елемента при такому підході може зайняти досить багато часу, особливо якщо видаляється елемент на початку списку. Щоб вилучити перший елемент в масиві з 1000 елементів, буде потрібно зрушити вліво на одну позицію 999 елементів. Набагато швидше можна видаляти елементи за допомогою простої схеми *збирання сміття* (garbage collection).

Замість видалення елементів зі списку, позначте їх як невикористовуванні. Якщо елементи списку - дані простих типів, наприклад цілі, можна позначати елементи, використовуючи певне, так зване *«сміттєве» значення* (garbage value).

Для цілих чисел можна використовувати для цього значення -32767. Для змінної типу `Variant` можна використовувати значення `NULL`. Це значення надається кожному невикористовуваному елементу. Наступний фрагмент коду демонструє видалення елемента з подібного цілочислового списку:

```
Const GARBAGE_VALUE = -32767
' Позначити елемент як невикористовуваний.
Sub Removefromlist(position As Long)
```

```
  List(position) = GARBAGE_VALUE
End Sub
```

Якщо елементи списку - це структури, визначені оператором `Type`, ви можете додати до такої структури нове поле `Isgarbage`. Коли елемент видаляється зі списку, значення поля `Isgarbage` встановлюється в `True`.

```
Type Mydata
  Name As Sring                          ' Дані.
  Isgarbage As Integer           ' Цей елемент не
  використовується?
End Type
' Позначити елемент, що як не використовується.
Sub Removefromlist (position As Long)
  List(position).Isgarbage = True
End Sub
```

Для простоти, далі в цьому розділі, передбачається, що елементи є даними універсального типу і їх можна позначати значенням `NULL`.

Тепер можна змінити інші процедури, які використовують список, щоб вони пропускали позначені елементи. Наприклад, так можна модифікувати процедуру, яка друкує список:

```
' Друк елементів списку.
Sub Printitems()
Dim I As Long
  For I = 1 To Arraysize
      If Not Isnull(List(I)) Then      ' Якщо елемент
  не позначений
            Print Str$(List(I))        ' надрукувати
  його.
      End If
  Next I
End Sub
```

Після використання протягом деякого часу схеми позначки «сміття» список може виявитися повністю ним заповнений. Зрештою, підпрограми, подібні цій процедурі, більше часу будуть витрачати на пропуск непотрібних елементів, ніж на обробку справжніх даних.

Для того щоб уникнути цього, можна періодично запускати процедуру збору сміття (garbage collection routine). Ця процедура переміщає всі непомічені записи в початок масиву. Після цього їх можна додати до вільних елементів наприкінці масиву. Коли буде потрібно додати до масиву додаткові елементи, їх також можна буде використовувати без зміни розміру масиву.

Після додавання позначених елементів до інших вільних комірок масиву повний обсяг вільного простору може стати досить великим, і в цьому випадку можна зменшити розмір масиву, звільняючи пам'ять:

```
Private Sub Collectgarbage()
Dim i As Long
Dim good As Long
  good = 1          ' Перший використовуваний елемент.
  For i = 1 To m_Numitems
       ' Якщо він не позначений, перемістити його на
  нове місце.
       If Not Isnull(m_List(i)) Then
             m_List(good) = m_list(i)
             good = good + 1
       End If
  Next i
  ' Останній використовуваний елемент.
  m_Numitems(good) = good - 1
       ' чи Необхідно зменшувати розмір списку?
  If m_Numitems < m_Shrinkwhen Then Resizelist
End Sub
```

При виконанні збору сміття використовувані елементи переміщаються ближче до початку списку, заповнюючи простір, який займали позначені елементи. Виходить, положення елементів у списку може змінитися під час цієї операції. Якщо інші частини програми звертаються до елементів списку за їхнім положенням в ньому, необхідно модифікувати процедуру чищення пам'яті для того, щоб вона також обновляла посилання на положення елементів у списку. У загальному випадку це може виявитися досить складним, призводячи до проблем при супроводі програм.

Можна вибирати різні моменти для запуску процедури збору сміття. Один з них - коли масив досягає певного розміру, наприклад, коли список містить 30000 елементів.

Цьому методу властиві певні недоліки. *По-перше,* він використовує великий обсяг пам'яті. Якщо ви часто додаєте або видаляєте елементи, «сміття» буде займати досить велику частину масиву. При такій неощадливій витраті пам'яті програма може витрачати час на свопінг, хоча список міг би цілком міститися в пам'яті при більш частому переупорядкуванні.

По-друге, якщо список починає заповнюватися непотрібними даними, процедури, які його використовують, можуть стати надзвичайно неефективними. Якщо в масиві з 30000 елементів 25000 не використовуються, підпрограма типу описаної вище `Printitems`, може виконуватися дуже повільно.

І, нарешті, збирання сміття для дуже великого масиву може потребувати значного часу, особливо якщо при обході елементів масиву програмі доводиться звертатися до сторінок, вивантажених на диск. Це може призводити до «підвисання» вашої програми на кілька секунд під час чищення пам'яті.

Щоб розв'язати цю проблему, можна створити нову змінну `Garbagecount`, у якій буде перебувати кількість непотрібних елементів у списку. Коли значна частина пам'яті, займаної списком, містить непотрібні елементи, ви може почати процедуру «збору сміття».

```
Dim Garbagecount As Long   ' Число непотрібних
  елементів.
Dim Maxgarbage As Long           ' Це значення
  визначається в Resizelist.
' Позначити елемент як непотрібний.
' Якщо «сміття» занадто багато, почати чищення пам'яті.
Public Sub Remove(position As Long)
  m_List(position) = Null
  m_Garbagecount = m_Garbagecount + 1
  ' Якщо «сміття» занадто багато, почати чищення
  пам'яті.
  If m_Garbagecount > m_Maxgarbage Then Collectgarbage
End Sub
```

Програма `Garbage (диск з прикладами – папка ProgR9)` демонструє цей метод чищення пам'яті. Вона пише поруч із невикористовуваними елементами списку слово «`unused`», а поруч із позначеними як непотрібні - слово «`garbage`». Програма використовує клас `Garbagelist` приблизно так само, як програма

Simlist використовувала клас Simplelist, але при цьому вона ще здійснює «збирання сміття».

Щоб додати елемент до списку, введіть його значення й натисніть на кнопку **Add** (Додати). Для видалення елемента виділіть його, а потім натисніть на кнопку **Remove** (Вилучити). Якщо список містить занадто багато «сміття», програма почне виконувати чищення пам'яті.

При кожній зміні розміру списку об'єкта Garbagelist програма виводить вікно повідомлення, у якому наводиться число використовуваних і вільних елементів у списку, а також значення змінних Maxgarbage і Shrinkwhen. Якщо вилучити достатню кількість елементів, так що більше, ніж Maxgarbage елементів будуть позначені як непотрібні, програма почне виконувати чищення пам'яті. Після її закінчення програма зменшить розмір масиву, якщо він містить менше, ніж Shrinkwhen, зайнятих елементів.

Якщо розмір масиву повинен бути збільшений, програма Garbage додасть до масиву ще 50 відсотків порожніх комірок і завжди залишить хоча б одну порожню комірку при будь-якій зміні розміру масиву. Ці значення були обрані для спрощення роботи користувача зі списком. У реальній програмі відсоток вільної пам'яті повинен бути меншим, а кількість вільних комірок - більшою. Оптимальними виглядають значення порядку 10 відсотків і 10 вільних комірок.

Зв'язні списки

Інша стратегія використовується при управлінні зв'язаними списками. Зв'язаний список зберігає елементи в структурах даних або об'єктах, які називаються комірками (cells). Кожна комірка містить покажчик на наступну комірку в списку. Тому що єдиний тип покажчиків, які підтримує Visual Basic — це посилання на об'єкти, то комірки у зв'язному списку повинні бути об'єктами.

У класі, що задає комірку, повинна бути визначена змінна Nextcell, яка вказує на наступну комірку в списку. У ньому також повинні бути визначені змінні, які містять дані, з якими буде працювати програма. Ці змінні можуть бути оголошені як відкриті (public) усередині класу, або клас може містити процедури для

читання й запису значень цих змінних. Наприклад, у зв'язному списку із записами про співробітників у цих полях можуть перебувати ім'я співробітника, номер соціального страхування, назва посади і т.д. Визначення для класу `Empcell` можуть виглядати приблизно так:

```
Public Empname As String
Public SSN As String
Public Jobtitle As String
Public Nextcell As Empcell
```

Програма створює нові комірки за допомогою оператора `New`, задає їхні значення й з'єднує їх, використовуючи змінну `Nextcell`.

Програма завжди повинна зберігати посилання на вершину списку. Для того, щоб визначити, де закінчується список, програма повинна встановити значення `Nextcell` для останнього елемента списку таким, що дорівнює `Nothing` (нічого). Наведений нижче фрагмент коду створює список, що представляє трьох співробітників:

```
Dim top_cell As Empcell
Dim cell1 As Empcell
Dim cell2 As Empcell
Dim cell3 As Empcell
  ' Створення гнізд.
  Set cell1 = New Empcell
  cell1.Empname = "Стівенс"
  cell1.SSN = "123-45-6789"
  cell1.Jobtitle = "Автор"
  Set cell2 = New Empcell
  cell2.Empname = "Кетс"
  cell2.SSN = "123-45-6789"
  cell2.Jobtitle = "Юрист"
  Set cell3 = New Empcell
  cell3.Empname = "Тулі"
  cell3.SSN = "123-45-6789"
  cell3.Jobtitle = "Менеджер"
  ' З'єднати комірки, утворюючи зв'язний список.
  Set cell1.Nextcell = cell2
  Set cell2.Nextcell = cell3
  Set cell3.Nextcell = Nothing
  ' Зберегти посилання на вершину списку.
```

```
  Set top_cell = cell1
```

На рис. 9.2 показаний схематичний вигляд цього зв'язного списку.

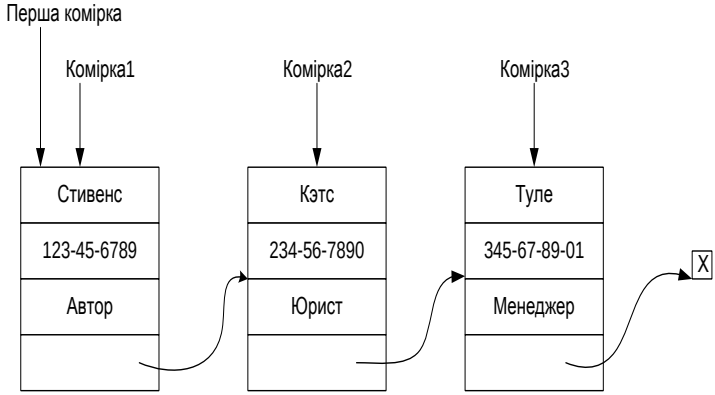


Рис.9.2. Схематичне представлення простого зв’язного списку

Прямокутники представляють комірки, а стрілки - посилання на об'єкти. Маленький перекреслений прямокутник представляє значення `Nothing`, яке позначає кінець списку. Майте на увазі, що `top_cell`, `cell1` і `cell2` – це не справжні об'єкти, а тільки посилання, які вказують на них.

Наступний код використовує зв'язний список, побудований за допомогою попереднього прикладу для друку імен співробітників зі списку. Змінна `ptr` використовується як покажчик на елементи списку. Вона спочатку вказує на вершину списку. У коді використовується цикл `Do` для переміщення `ptr` за списком доти, поки покажчик не дійде до кінця списку. Під час кожного циклу процедура друкує поле `Empname` комірки, на яке вказує `ptr`. Потім вона збільшує `ptr`, вказуючи на наступну комірку в списку. Зрештою, `ptr` досягає кінця списку й одержує значення `Nothing`, і цикл `Do` зупиняється.

```
Dim ptr As Empcell
  Set ptr = top_cell       ' Почати з вершини списку.
  Do While Not (ptr Is Nothing)
       ' Вивести поле Empname цієї комірки.
       Debug.Print ptr.Empname
       ' Перейти до наступної комірки в списку.
       Set ptr = ptr.Nextcell
```

```
  Loop
```

Після виконання коду ви одержите такий результат:

```
Стівенс
Кетс
Тулі
```

Використання покажчика на інший об'єкт називається *непрямою адресацією* (`indirection`), оскільки ви використовуєте покажчик для непрямого маніпулювання даними. Непряма адресація може бути дуже заплутаною. Навіть для простого розташування елементів, такого як зв'язний список, іноді важко запам'ятати, на який об'єкт указує кожне посилання. У більш складних структурах даних покажчик може посилатися на об'єкт, що містить інший покажчик. Якщо є кілька покажчиків і кілька рівнів непрямої адресації, ви легко можете заплутатися в них.

Для того щоб полегшити розуміння, у книзі використовують ілюстрації, такі як рис. 9.2, щоб допомогти вам наочно уявити ситуацію там, де це можливо. Багато з алгоритмів, які використовують покажчики, можна легко проілюструвати подібними рисунками.

Додавання елементів

Простий зв'язний список, показаний на рис. 9.2, має декілька важливих властивостей. *По-перше,* можна дуже легко додати нову комірку в початок списку. Встановимо покажчик нової комірки `Nextcell` на поточну вершину списку. Потім встановимо покажчик `top_cell` на нову комірку. Рис. 9.3 відповідає цій операції. Код мовою Visual Basic для цієї операції дуже простий:

```
Set new_cell.Nextcell = top_cell
Set top_cell = new_cell
```

Порівняйте розмір цього коду й коду, який довелося б написати для додавання нового елемента в початок списку, заснованого на масиві, у якому треба було б перемістити всі елементи масиву на одну позицію, щоб звільнити місце для нового елемента. Ця операція зі складністю порядку O(N) може потребувати багато часу, якщо список досить довгий. Використовуючи зв'язний список, можна додати новий елемент у початок списку всього за пару кроків.

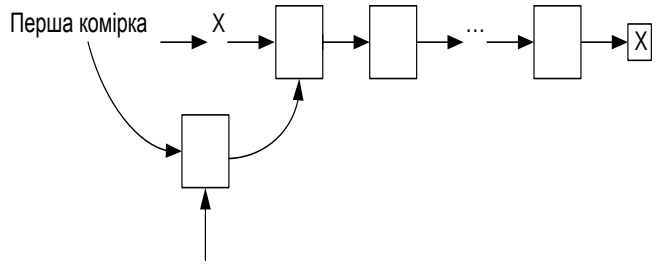


Рис.9.3. Додавання елемента у початок зв'язного списку

Так само легко додати новий елемент і в середину зв'язного списку. Припустимо, ви прагнете вставити новий елемент після комірки, на яку вказує змінна `after_me`. Встановимо значення `Nextcell` нової комірки таким, що дорівнює `after_me.Nextcell`. Тепер встановимо покажчик `after_me.Nextcell` на нову комірку. Ця операція показана на рис.9.4. Код на Visual Basic знову дуже простий:

```
Set new_cell.Nextcell = after_me.Nextcell
Set after_me.Nextcell = new_cell
```

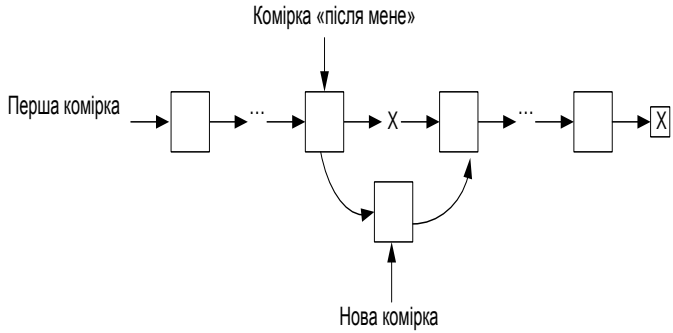


Рис.9.4. Додавання елемента у початок зв'язного списку

Видалення елементів

Видалити елемент із вершини зв'язного списку так само просто, як і додати його. Просто встановіть покажчик `top_cell` на наступну комірку в списку. Рис. 9.5 відповідає цій операції. Вихідний код для цієї операції ще простіший, ніж код для додавання елемента.

```
Set top_cell = top_cell.Nextcell
```

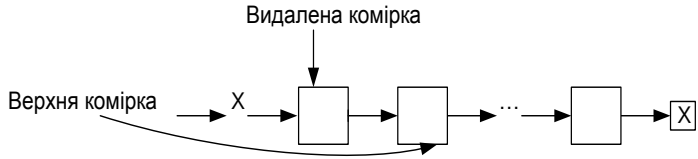


Рис. 9.5. Видалення елемента з початку зв'язного списку

Коли покажчик `top_cell` переміщається на другий елемент у списку, у програмі більше не залишиться змінних, що вказують на перший об'єкт. У цьому випадку лічильник посилань на цей об'єкт дорівнюватиме нулю, і система автоматично знищить його.

Так само просто вилучити елемент із середини списку. Припустимо, ви прагнете вилучити елемент, що знаходиться після комірки `after_me`. Просто встановіть покажчик `Nextcell` цієї комірки на наступну комірку. Ця операція показана на рис. 9.6. Код на Visual Basic простий і зрозумілий:

```
after_me.Nextcell = after_me.Nextcell.Nextcell
```

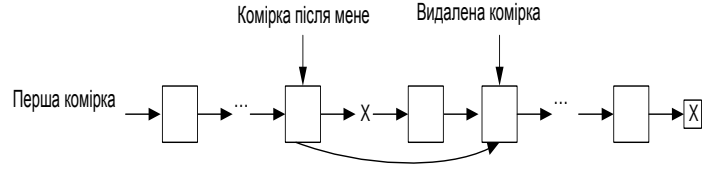


Рис. 9.6. Видалення елемента з середини зв'язного списку

Знову порівняємо цей код з кодом, який знадобився б для виконання тієї ж операції, при використанні списку на основі масиву. Можна швидко позначити вилучений елемент як невикористовуваний, але це залишає в списку непотрібні значення. Процедури, що обробляють список, повинні це враховувати і відповідно бути більш складними. Присутність надмірної кількості

«сміття» також уповільнює роботу процедури, і, зрештою, доведеться проводити чищення пам'яті.

При видаленні елемента зі зв'язного списку у ньому не залишається порожніх проміжків. Процедури, які обробляють список, усі так само обходять список від початку до кінця і не мають потреби в модифікації.

Знищення зв'язного списку

Можна припустити, що для знищення зв'язного списку необхідно обійти весь список, установлюючи значення `Nextcell` для всіх комірок таким, що дорівнює `Nothing`. Насправді процес набагато простіший: тільки `top_cell` набуває значення `Nothing`.

Коли програма встановлює значення `top_cell` таким, що дорівнює `Nothing`, лічильник посилань для першої комірки дорівнює нулю, і Visual Basic знищує цю комірку.

Під час знищення комірки система визначає, що в полі `Nextcell` цієї комірки міститься посилання на іншу комірку. Оскільки перший об'єкт знищується, то кількість посилань на другий об'єкт зменшується. При цьому лічильник посилань на другий об'єкт списку дорівнює нулю, тому система знищує і його.

Під час знищення другого об'єкта система зменшує кількість посилань на третій об'єкт, і так далі доти, доки всі об'єкти в списку не будуть знищені. Коли в програмі вже не буде посилань на об'єкти списку, можна знищити й увесь список за допомогою єдиного оператора `Set top_cell = Nothing`.

Сигнальні позначки

Для додавання або видалення елементів з початку або середини списку використовуються різні процедури. Можна звести обидва ці випадки до одного й позбутися надлишкового коду, якщо ввести спеціальну *сигнальну позначку* (sentinel) на самому початку списку. Сигнальну позначку не можна вилучити. Вона не містить даних і використовується тільки для позначення початку списку.

Тепер замість того, щоб обробляти особливий випадок додавання елемента в початок списку, можна поміщати елемент після

позначки. У такий же спосіб, замість особливого випадку видалення першого елемента зі списку, просто видаляється елемент, що випливає за позначкою.

Використання сигнальних позначок поки не вносить особливих відмінностей. Сигнальні позначки відіграють важливу роль у набагато більш складних алгоритмах. Вони дозволяють обробляти особливі випадки, такі як початок списку, як звичайні. При цьому потрібно написати й налагодити менше коду, а алгоритми стають більш погодженими й більш простими для розуміння.

У табл. 9.1 порівнюється складність виконання деяких типових операцій з використанням списків на основі масивів зі «збиранням сміття» або зв'язних списків.

Таблиця 9.1

Порівняння списків на базі масивів і зв'язних списків

Операція	**Список на основі масиву**	**Зв'язний список**
Додавання елемента в початок списку	Просто	Просто
Додавання елемента в кінець списку	Важко	Просто
Додавання елемента в середину списку	Важко	Просто
Видалення елемента з початку списку	Просто	Просто
Видалення елемента із середини списку	Просто	Просто
Перегляд значимих елементів	Середньої складності	Просто

Списки на основі масивів мають одну перевагу: вони використовують менше пам'яті. Для зв'язних списків необхідно додати поле `Nextcell` до кожного елемента даних. Кожне посилання на об'єкт займає чотири додаткові байти пам'яті. Для дуже великих масивів це може потребувати більших витрат пам'яті.

Програма `Lnklist1 (диск з прикладами – папка ProgR9)` демонструє простий зв'язний список із сигнальною позначкою. Уведіть значення в текстове поле введення, і натисніть на елемент у списку або на позначку. Потім натисніть на кнопку **Add After** (Додати після), і програма додасть новий елемент після зазначеного. Для видалення елемента зі списку натисніть на елемент, а потім на кнопку **Remove After** (Вилучити після).

Інкапсуляція зв'язних списків

Програма Lnklist1 (диск з прикладами – папка ProgR9) управляє списком явно. Наприклад, наступний код показує, як програма видаляє елемент зі списку. Коли підпрограма починає роботу, глобальна змінна Selectedindex дає положення елемента, що передує елементу, який видаляється зі списку. Змінна Sentinel містить посилання на сигнальну позначку списку.

```
Private Sub Cmdremoveafter_Click()
Dim ptr As Listcell
Dim position As Integer
  If Selectedindex < 0 Then Exit Sub
  ' Знайти елемент.
  Set ptr = Sentinel
  position = Selectedindex
  Do While position > 0
       position = position - 1
       Set ptr = ptr.nextcell
  Loop
  ' Вилучити наступний елемент.
  Set ptr.Nextcell = ptr.Nextcell.Nextcell
  Numitems = Numitems - 1
  Selectitem Selectedindex' Знову вибрати елемент.
  Displaylist
  Newitem.Setfocus
End Sub
```

Щоб спростити використання зв'язного списку, можна інкапсулювати його функції в класі. Це реалізоване в програмі Lnklist2 на (диск з прикладами – папка ProgR9). Ця програма аналогічна програмі Lnklist1, але використовує для управління списком клас Linkedlist.

Клас Linekedlist управляє внутрішньою організацією зв'язного списку. У ньому перебувають процедури для додавання й видалення елементів, повернення значення елемента за його індексом, кількості елементів у списку, і очищення списку. Цей клас дозволяє поводитися зі зв'язним списком майже як з масивом.

Це набагато спрощує основну програму. Наприклад, наступний код показує, як програма Lnklist2 (диск з

прикладами – папка ProgR9) видаляє елемент зі списку. Тільки один рядок у програмі в дійсності відповідає за видалення елемента. Інші відображають новий список. Порівняйте цей код з попередньою процедурою:

```
Private sub Cmdremoveafter_Click()
  Llist.Removeafter Selectedindex
  Selecteditem Selectedlist      ' Знову вибрати
  елемент.
  Displaylist
  Newitem.Setfocus
  Cmdclearlist.Enabled
End Sub
```

Доступ до комірок

Клас Linkedlist, використовуваний програмою Lnklst2, дозволяє основній програмі використовувати список майже як масив. Наприклад, підпрограма Item, наведена в наступному коді, повертає значення елемента за його положенням:

```
Function Item(Byval position As Long) As Variant
Dim ptr As Listcell
  If position < 1 Or position > m_Numitems Then
       ' Вихід за межі. Повернути NULL.
       Item = Null
       Exit Function
  End If
  ' Знайти елемент.
  Set ptr = m_Sentinel
  Do While position > 0
       position = position - 1
       Set ptr = ptr.Nextcell
  Loop
  Item = ptr.Value
End Function
```

Ця процедура досить проста, але вона не використовує переваги зв'язної структури списку. Наприклад, припустимо, що програмі потрібно послідовно перебрати всі об'єкти в списку. Вона

могла б використовувати підпрограму Item для почергового доступу до них, як показано в коді:

```
Dim i As Integer
  For i = 1 To Llist.Numitems
       ' Виконати які- або дії з Llist.Item(i).
             :
  Next i
```

При кожному виклику процедури Item вона переглядає список у пошуку наступного елемента. Щоб знайти елемент I, програма повинна пропустити I - 1 елементів. Щоб перевірити всі елементи в списку з N елементів, процедура пропустить 0+1+2+3+…+N-1 =N*(N-1)/2 елемента. При великих N програма втратить багато часу на пропуск елементів.

Клас Linkedlist може прискорити цю операцію, використовуючи інший метод доступу. Можна використовувати частну змінну m_Currentcell для відстеження поточної позиції в списку. Для повернення значення поточного положення використовується підпрограма Currentitem. Процедури Movefirst, Movenext і Endoflist дозволяють основній програмі управляти поточною позицією в списку.

Наприклад, код містить підпрограму Movenext:

```
Public Sub Movenext()
  ' Якщо поточна комірка не обрана, нічого не робити.
  If Not (m_Currentcell Is Nothing) Then _
       Set m_Currentcell = m_Currentcell.Nextcell
End Sub
```

За допомогою цих процедур основна програма може звернутися до всіх елементів списку, використовуючи наступний код. Ця версія трохи складніша, ніж попередня, але вона набагато ефективніша. Замість того, щоб пропускати N*(N-1)/2 елементів і опитувати по черзі всі N елементів списку, вона не пропускає жодного. Якщо список складається з 1000 елементів, це заощаджує майже півмільйона кроків.

```
Llist.Movefirst
Do While Not Llist.Endoflist
  ' Виконати  дії над елементом Llist.Item(i).
```

```
      :
  Llist.Movenext
Loop
```

Програма `Lnklist3` (диск з прикладами – папка `ProgR9`) використовує ці нові методи для управління зв'язним списком. Вона аналогічна програмі `Lnklist2`, але більш ефективно звертається до елементів. Для невеликих списків, використовуваних у програмі, ця різниця непомітна. Для програми, яка звертається до всіх елементів великого списку, ця версія класу `Linkedlist` більш ефективна.

Різновиди зв'язних списків

Зв'язні списки відіграють важливу роль у багатьох алгоритмах, і ви будете зустрічатися з ними протягом усього посібника. У наступних підрозділах розглядаються кілька спеціальних різновидів зв'язних списків.

Циклічні зв'язні списки

Замість того, щоб установлювати покажчик `Nextcell` такими, що дорівнюють `Nothing`, можна встановити його на перший елемент списку, утворюючи *циклічний список* (`circular list`), як показано на рис. 9.7.

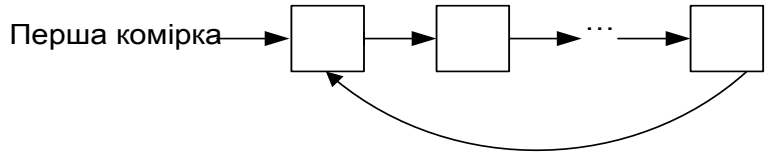


Рис.9.7. Циклічний зв'язний список

Циклічні списки корисні, якщо потрібно обходити ряд елементів у нескінченному циклі. При кожному кроці циклу програма просто переміщає покажчик на наступну комірку в списку. Припустимо, є циклічний список елементів, що містить назви днів

тижня. Тоді програма могла б перераховувати дні місяця, використовуючи такий код:

```
 ' Тут перебуває код для створення й настроювання
  списку і т.д.
  :
' Надрукувати календар на місяць.
'
' first_day — це індекс структури, що містить день
  тижня для
' першого дня місяця. Наприклад, місяць може починатися
' у понеділок.
'
' num_days — число днів у місяці.
Private Sub Listmonth(first_day As Integer, num_days As
  Integer)
Dim ptr As Listcell
Dim i As Integer
  Set ptr = top_cell
  For i = 1 to num_days
       Print Format$(i) & ": " & ptr.Value
       Set ptr = ptr.Nextcell
  Next I
End Sub
```

Циклічні списки також дозволяють досягти будь-якого місця в списку, почавши з деякого положення в ньому. Це вносить до списку привабливу симетрію. Програма може звертатися до усіх елементів списку майже однаковим чином:

```
Private Sub Printlist(start_cell As Integer)
Dim ptr As Integer
Set ptr = start_cell
  Do
       Print ptr.Value
       Set ptr = ptr.Nextcell
  Loop While Not (ptr Is start_cell)
End Sub
```

Проблема циклічних посилань

Знищення циклічного списку вимагає трохи більше уваги, ніж видалення звичайного списку. Якщо ви просто встановите значення змінної `top_cell` таким, що дорівнює `Nothing`, то програма не зможе більше звернутися до списку. Проте, оскільки лічильник посилань першої комірки не дорівнює нулю, вона не буде знищена. На кожний елемент списку вказує який-небудь інший елемент, тому жоден з них не буде знищений.

Це *проблема циклічних посилань* (`circular referencing problem`). Тому що комірки вказують на інші комірки, жодна з них не буде знищена. Програма не може одержати доступ ні до однієї з них, тому займана ними пам'ять буде витрачатися дарма до завершення роботи програми.

Проблема циклічних посилань може зустрітися не тільки в цьому випадку. Багато мереж містять циклічні посилання - навіть одинична комірка, поле `Nextcell` якої вказує на саме цю комірку, може викликати проблему.

Рішення її полягає в тому, щоб розбити ланцюг посилань. Наприклад, ви можете використовувати у своїй програмі такий код для знищення циклічного зв'язного списку:

```
Set top_cell.Nextcell = Nothing
Set top_cell = Nothing
```

Перший рядок розбиває цикл посилань. У цей момент на другу комірку списку не вказує жодна змінна, тому система зменшує лічильник посилань комірок до нуля й знищує її. Це зменшує лічильник посилань на третій елемент до нуля, і відповідно, він також знищується. Цей процес триває доти, поки не будуть знищені всі елементи списку, крім першого. Установка значення `top_cell` елемента в `Nothing` зменшує його лічильник посилань до нуля і останню комірку також знищує.

Двох зв'язні списки

Під час обговорення зв'язних списків ви могли помітити, що більшість операцій визначалася в термінах виконання чого-небудь *після* певної комірки в списку. Якщо задана певна комірка, легко додати або вилучити комірку після неї або перелічити комірки, що

йдуть за нею. Вилучити саму комірку, вставити нову комірку перед нею або перелічити комірки, що йдуть перед нею вже не так легко. Проте невелика зміна дозволить полегшити й ці операції.

Додамо нове поле покажчика до кожної комірки, яка вказує на *попередню* комірку в списку. Використовуючи це нове поле, можна легко створити двозв'язний *список* (`doubly linked list`), який дозволяє переміщатися в початок або в кінець списку (рис.9.8.). Тепер можна легко вилучити комірку, вставити її перед іншою коміркою, перелічити комірки в будь-якому напрямку.

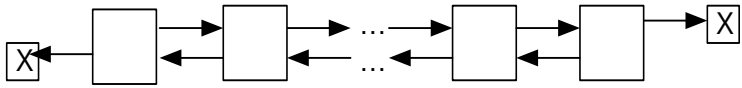


Рис.9.8. Двозв'язний список

Клас `Doublelistcell`, який використовується для списків таких типів, може оголосити змінні так:

```
Public Value As Variant
Public Nextcell As Doublelistcell
Public Prevcell As Doublelistcell
```

Часто буває корисно зберігати покажчики на початок та на кінець двозв'язного списку. Тоді ви зможете легко додавати елементи до кожного з кінців списку. Іноді також буває корисно розміщати сигнальні позначки на початку та наприкінці списку. Тоді в міру роботи зі списком вам не потрібно буде опікуватися про те, чи працюєте ви з початком, із серединою або з кінцем списку.

На рис. 9.9 показаний двозв'язний список із сигнальними позначками. На цьому рисунку не використовувані покажчики позначок `Nextcell` і `Prevcell`, встановлені в `Nothing`. Оскільки програма пізнає кінці списку, порівнюючи значення покажчиків комірок із сигнальними позначками (тобто не перевіряє, чи дорівнюють значення `Nothing`), установлення цих значень такими, що дорівнюють `Nothing`, не є абсолютно необхідним. Проте це ознака гарного стилю.

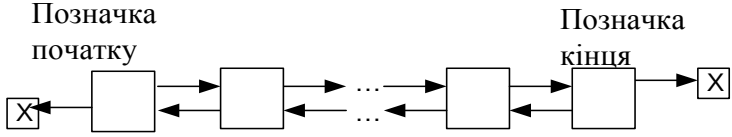


Рис.9.8. Двозв'язний список з позначками

Код для вставлення й видалення елементів із двозв’язного списку подібний наведеному раніше коду для однозв'язного списку. Процедури потребують лише незначних змін для роботи з покажчиками `Prevcell`.

Тепер ви можете написати нові процедури для вставлення нового елемента до або після даного елемента і процедуру видалення заданого елемента. Наприклад, підпрограми, що випливають, додають і видаляють комірки із двозв’язного списку. Зверніть увагу на те, що ці процедури не мають потреби в доступі ні до однієї із сигнальних позначок списку. Їм потрібні тільки покажчики на вузол, який повинен бути вилучений або доданий, і вузол, сусідній із точкою вставлення.

```
Public Sub Removeitem(Byval target As Doublelistcell)
Dim after_target As Doublelistcell
Dim before_target As Doublelistcell
  Set after_target = target.Nextcell
  Set before_target = target.Prevcell
  Set after_target.Nextcell = after_target
  Set after_target.Prevcell = before_target
End Sub
Sub Addafter (new_Cell As Doublelistcell, after_me As
  Doublelistcell)
Dim before_me As Doublelistcell
  Set before_me = after_me.Nextcell
  Set after_me.Nextcell = new_cell
  Set new_cell.Nextcell = before_me
  Set before_me.Prevcell = new_cell
  Set new_cell.Prevcell = after_me
End Sub
Sub Addbefore(new_cell As Doublelistcell, before_me As
  Doublelistcell)
Dim after_me As Doublelistcell

  Set after_me = before_me.Prevcell
  Set after_me.Nextcell = new_cell
  Set new_cell.Nextcell = before_me
  Set before_me.Prevcell = new_cell
  Set new_cell.Prevcell = after_me
End Sub
```

Якщо знову глянути на рис.9.9, ви побачите, що кожна пара сусідніх комірок утворює циклічне посилання. Це робить знищення двозв'язного списку небагато більш складним завданням, ніж знищення однозв'язних або циклічних списків. Наступний код приводить один зі способів очищення двозв'язного списку. Спочатку покажчики `Prevcell` усіх комірок установлюються такими, що дорівнюють `Nothing`, щоб розірвати циклічні посилання. Це, по суті, перетворює список в однозв'язний. Коли посилання сигнальних позначок установлюються в `Nothing`, усі елементи звільняються автоматично, так само як і в однозв'язному списку.

```
Dim ptr As Doublelistcell
  ' Очистити покажчики Prevcell, щоб розірвати циклічні
  посилання.
  Set ptr = Topsentinel.Nextcell
  Do While Not (ptr Is Bottomsentinel)
       Set ptr.Prevcell = Nothing
       Set ptr = ptr.Nextcell
  Loop
  Set Topsentinel.Nextcell = Nothing
  Set Bottomsentinel.Prevcell = Nothing
```

Якщо створити клас, який інкапсулює двозв'язний список, то його оброблювач події `Terminate` зможе знищувати список. Коли основна програма встановить значення посилання на список таким, що дорівнює `Nothing`, список автоматично звільнить займану пам'ять.

Програма `Dbllink (диск з прикладами – папка ProgR9)` працює із двозв'язним списком. Вона дозволяє додавати елементи до або після обраного елемента, а також видаляти обраний елемент.

Потоки

У деяких програмах буває зручно обходити зв'язний список не тільки в одному напрямку. У різних частинах програми вам може знадобитися виводити список співробітників за їхніми прізвищами, заробітною платою, ідентифікаційним кодом системи соціального страхування або спеціальністю.

Звичайний зв'язний список дозволяє переглядати елементи тільки в одному напрямку. Використовуючи покажчик `Prevcell`, можна створити двозв'язний список, який дозволить переміщатися за списком вперед та назад. Цей підхід можна розвити й далі, додавши більше покажчиків на структуру даних, які дозволять виводити список в іншому порядку.

Набір посилань, який задає який-небудь порядок перегляду, називається *потоком* (`thread`), а сам отриманий список *багатопотоковим списком* (`threaded list`). Не плутайте ці потоки з потоками, які надає система Windows NT.

Список може містити будь-яку кількість потоків, хоча, починаючи з якогось моменту, не варта справа заходу. Застосування потоку, що впорядковує список співробітників за прізвищем, буде обґрунтовано, якщо ваша програма часто використовує цей порядок, на відміну від розташування по-батькові, який чи навряд коли буде використовуватися.

Деякі розташування не варто організовувати у вигляді потоків. Наприклад, потік, що впорядковує співробітників за статтю, чи навряд доцільний тому, що таке впорядкування легке одержати й без нього. Для того щоб скласти список співробітників за статтю, досить просто обійти список за будь-яким іншим потоком, друкуючи прізвища жінок, а потім повторити обхід ще раз, друкуючи прізвища чоловіків. Для одержання такого розташування досить усього двох проходів списку.

Порівняйте цей випадок з тим, коли ви прагнете впорядкувати список співробітників за прізвищем. Якщо список не включає потік прізвищ, вам доведеться знайти прізвище, яке буде першим у списку, далі те, що потім випливає, і т.д. Це процес зі складністю порядку $O(N^2)$, який набагато менш ефективний, ніж сортування за статтю зі складністю порядку $O(N)$.

У загальному випадку введення потоку може бути доцільне, якщо його потрібно часто використовувати і якщо при необхідності одержати той же порядок досить складно. Потік не потрібний, якщо його завжди легко створити заново.

Програма `Treads (диск з прикладами – папка ProgR9)` демонструє простий багатопотоковий список співробітників. Заповніть поля прізвища, спеціальності, статі й номера соціального страхування для нового співробітника. Потім натисніть на кнопку **Add** (Додати), щоб додати співробітника до списку.

Програма містить потоки, які впорядковують список за прізвищем за алфавітом і у зворотному порядку, за номером

соціального страхування й спеціальності в прямому й зворотному порядку. Ви можете використовувати додаткові кнопки для вибору потоку, у порядку якого програма виводить список. На рис.9.10 показане вікно програми `Threads` зі списком співробітників, упорядкованим за прізвищем.

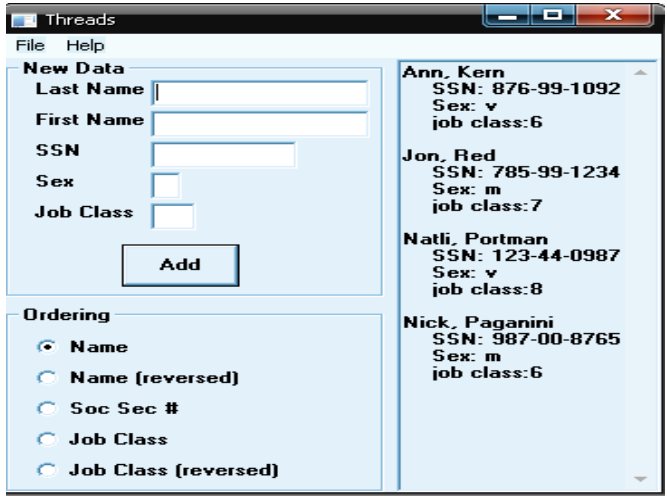


Рис.9.10. Вікно програми `Threads`

Клас `Threadedcell`, використовуваний програмою `Threads`, визначає такі змінні:

```
Public Lastname As String
Public Firstname As String
Public SSN As String
Public Sex As String
Public Jobclass As Integer
Public Nextname As Treadedcell    ' На прізвище в прямому
  порядку.
Public Prevname As Treadedcell    ' На прізвище у
  зворотному порядку.
Public Nextssn As Treadedcell     ' По номеру в прямому
  порядку.
Public Nextjobclass As Treadedcell      ' За фахом у
  прямому порядку.
Public Prevjobclass As Treadedcell      ' За фахом у
  зворотному порядку.
```

Клас Threadedlist інкапсулює багатопотоковий список. Коли програма викликає метод Additem, список обновляє свої потоки. Для кожного потоку програма повинна вставити елемент у правильному порядку. Наприклад, для того щоб вставити запис із прізвищем «Сміт», програма обходить список, використовуючи потік Nextname, доти, поки не знайде елемент із прізвищем, яке повинне іти за «Сміт». Потім вона вставляє в потік Nextname новий запис перед цим елементом.

При визначенні місця розташування нових записів у потоці важливу роль відіграють сигнальні позначки. Оброблювач подій Class_Initialize класу Threadedlist створює сигнальні позначки на вершині й наприкінці списку й ініціалізує їх покажчики так, щоб вони вказували один на одного. Потім значення позначки на початку списку встановлюється таким чином, щоб воно завжди перебувало в будь-якому значенні реальних даних для всіх потоків.

Наприклад, змінна Lastname може містити строкові значення. Порожній рядок "" іде за алфавітом перед будь-якими дійсними значеннями рядків, тому програма встановлює значення сигнальної позначки Lastname на початку списку таким, що дорівнює порожньому рядку.

У такий же спосіб Class_Initialize установлює значення даних для позначки наприкінці списку, що перевершує будь-які реальні значення у всіх потоках. Оскільки "~" іде за алфавітом після всіх видимих символів ASCII, програма встановлює значення поля Lastname для позначки наприкінці списку рівним "~".

Надаючи полю Lastname сигнальних позначок значення "" і ""~"", програма позбувається необхідності перевіряти особливі випадки, коли потрібно вставити новий елемент у початок або кінець списку. Будь-які нові дійсні значення будуть перебувати між значеннями Lastvalue сигнальних позначок, тому програма завжди зможе визначити правильне положення для нового елемента, не опікуючись тим, щоб не зайти за кінцеву позначку й не вийти за межі списку.

Наступний код показує, як клас Threadedlist вставляє новий елемент у потоки Nextname і Prevname. Тому що ці потоки використовують той самий ключ - прізвища, програма може обновляти їх одночасно.

```
Dim ptr As Threadedcell
Dim nxt As Threadedcell
```

```
Dim new_cell As New Threadedcell
Dim new_name As String
Dim next_name As String
  ' Записати значення нової комірки.
  With new_cell
       .Lastname = Lastname
       .Firstname = Firstname
       .SSN = SSN
       •sex = Sex
       .Jobclass = Jobclass
  End With
  ' Визначити місце нової комірки в потоці Nextthread.
  new_name = Lastname & ", " & Firstname
  Set ptr = m_Topsentinel
  Do
       Set nxt = ptr.Nextname
       next_name = nxt.Lastname & ", " & nxt.Firstname
       If next_name >= new_name Then Exit Do
       Set ptr = nxt
  Loop
  ' Вставити нову комірку в потоки Nextname і prevname.
  Set new_cell.Nextname = nxt
  Set new_cell.Prevname = ptr
  Set ptr.Nextname = new_cell
  Set nxt.Prevname = new_cell
```

Щоб такий підхід працював, програма повинна гарантувати, що значення нової комірки лежать між значеннями позначок. Наприклад, якщо користувач уведе як прізвище "∘∘∘", цикл вийде за позначку кінця списку, тому що "∘∘∘" іде після "∘". Потім програма аварійно завершить роботу при спробі доступу до значення `nxt.Lastname`, якщо `nxt` було встановлено таким, що дорівнює `Nothing`.

Інші зв'язні структури

Використовуючи покажчики, можна побудувати безліч інших корисних різновидів зв'язних структур, таких як дерева, нерегулярні масиви, розріджені масиви, графи й мережі. Комірка може містити будь-яку кількість покажчиків на інші комірки. Наприклад, для створення двійкового дерева можна використовувати комірку, що

містить два покажчики, один на лівого нащадка, і другий - на правого. Клас `Binarycell` може складатися з таких визначень:

```
Public Leftchild As Binarycell
Public Rightchild As Binarycell
```

На рис.9.11 показане дерево, побудоване із комірок такого типу. В 11 розділі дерева будуть обговорюватися більш докладно.

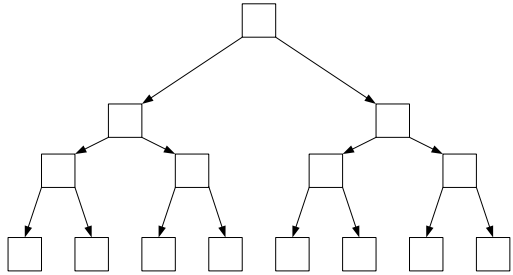


Рис.9.11 Двійкове дерево

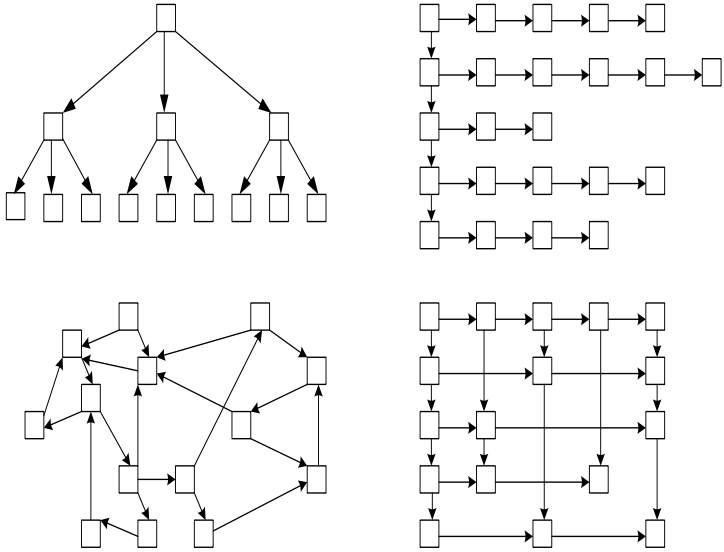


Рис.9.12. Зв’язні структури

Комірка може навіть містити колекцію або зв'язний список з покажчиками на інші комірки. Це дозволяє програмі зв'язати комірку

з будь-яким числом інших об'єктів. На рис.9.12 наведено приклади інших зв'язних структур даних.

Псевдопокажчики

За допомогою посилань у Visual Basic можна легко створювати зв'язні структури, такі як списки, дерева й мережі, але посилання вимагають додаткових ресурсів. Лічильники посилань і проблеми з розподілом пам'яті сповільнюють роботу структур даних, побудованих з використанням посилань.

Іншою стратегією, яка часто забезпечує кращу продуктивність, є застосування псевдопокажчика (`fake pointers`). При цьому програма створює масив структур даних. Замість використання посилань для зв'язування структур програма використовує індекси масиву. Знаходження елемента в масиві здійснюється в Visual Basic швидше, ніж вибірка його за посиланням на об'єкт. Це дає кращу продуктивність при застосуванні псевдопокажчика у порівнянні з відповідними методами посилань на об'єкти.

З іншого боку, застосування псевдопокажчика не настільки інтуїтивно, як застосування посилань. Це може ускладнити розробку й налагодження складних алгоритмів, таких як алгоритми мереж або збалансованих дерев.

Програма `Fakelist (диск з прикладами – папка ProgR9)` управляє зв'язним списком, використовуючи псевдопокажчики. Вона створює масив простих структур даних для зберігання ланок списку. Програма аналогічна програмі `Lnklist1`, але використовує псевдопокажчики.

Наступний код демонструє, як програма `Fakelist` створює масив кліткових структур:

```
' Структура даних гнізда.
Type Fakecell
  Value As String
  Nextcell As Integer
End Type
' Масив гнізд зв'язного списку.
Global Cells(0 To 100) As Fakecell
```

```
' Сигнальна мітка списку.
Global Sentinel As Integer
```

Оскільки псевдопокажчики - це не посилання, а просто цілі числа, програма не може використовувати значення `Nothing` для маркування кінця списку. Програма `Fakelist` використовує постійну `END_OF_LIST`, значення якої дорівнює -32767 для позначення порожнього покажчика.

Для полегшення виявлення невикористовуваних ланок програма `Fakelist` також використовує спеціальний «сміттєвий» список, що містить невикористовувані ланки. Наступний код демонструє ініціалізацію порожнього зв'язного списку. У ньому сигнальна позначка `Nextcell` набуває значення `END_OF_LIST`. Потім вона переміщає невикористовувані комірки в «сміттєвий» список.

```
' Зв'язний список невикористовуваних комірок.
Global Topgarbage As Integer

Public Sub Initializelist()
Dim i As Integer

  Sentinel = 0
  Cells(Sentinel).Nextcell = END_OF_LIST

  ' Перенести всі інші комірки в «сміттєвий» список.
  For i = 1 To Ubound (Cells) - 1
      Cells(i).Nextcell = i + 1
  Next i
  Cells(Ubound(Cells)).Nextcell = END_OF_LIST
  Topgarbage = 1
End Sub
```

При додаванні елемента до зв'язного списку програма використовує першу доступну комірку зі «сміттєвого» списку, ініціалізує поле комірки `Value` і вставляє її в список. Наступний код показує, як програма додає елемент після обраного:

```
Private Sub Cmdaddafter_Click()
Dim ptr As Integer
```

```
Dim position As Integer
Dim new_cell As Integer
  ' Знайти місце вставки.
  ptr = Sentinel
  position = Selectedlndex
  Do While position > 0
       position = position - 1
       ptr = Cells(ptr).Nextcell
  Loop
  ' Вибрати нову комірку з «сміттєвого» списку.
  new_cell = Topgarbage
  Topgarbage = Cells(Topgarbage).Nextcell
  ' Вставити елемент.
  Cells (new_cell).Value = Newitem.Text
  Cells(new_cell).Nextcell = Cells(ptr).Nextcell
  Cells(ptr).Nextcell = new_cell
  Numitems = Numitems + 1
  Displaylist
  Selectitem Selectedindex + 1          ' Вибрати новий
  елемент.
  Newitem.Text = ""
  Newitem.Setfocus
  Cmdclearlist.Enabled = True
End Sub
```

Після видалення комірки зі списку програма `Fakelist` поміщає вилучену комірку в «сміттєвий» список, щоб її потім можна було легко використовувати:

```
Private Sub Cmdremoveafter_Click()
Dim ptr As Integer
Dim target As Integer
Dim position As Integer
  If Selectedindex < 0 Then Exit Sub
  ' Знайти елемент.
  ptr = Sentinel
  position = Selectedindex
  Do While position > 0
       position = position - 1
       ptr = Cells(ptr).Nextcell
```

```
  Loop
  ' Пропустити наступний елемент.
  target = Cells(ptr).Nextcell
  Cells(ptr).Nextcell = Cells(target).Nextcell
  Numitems = Numitems - 1
  ' Додати вилучена комірkав «сміттєвий» список.
  Cells(target).Nextcell = Topgarbage
  Topgarbage = target
  Selectitem Selectedlndex' Снову вибрати елемент.
  Displaylist
  Cmdclearlist.Enabled = Numitems > 0
  Newitem.Setfocus
End Sub
```

Застосування псевдопокажчиків звичайно забезпечує кращу продуктивність, але є більш складним. Тому має сенс спочатку створити додаток, використовуючи посилання на об'єкти. Потім, якщо ви виявите, що програма значну частину часу витрачає на маніпулювання посиланнями, ви можете, якщо необхідно, перетворити її з використанням псевдопокажчиків.

Резюме

Використовуючи посилання на об'єкти, можна створювати гнучкі структури даних, такі як зв'язні списки, циклічні зв'язні списки й двозв’язні списки. Списки дозволяють легко додавати й видаляти елементи. Додаючи додаткові посилання до класу комірок, можна перетворити двозв’язний список у багатопотоковий.

Розбудовуючи й далі ці ідеї, можна створювати екзотичні структури даних, включаючи розріджені масиви, дерева, хеш-таблиці й мережі.

Контрольні запитання та завдання

1. Чим відрізняються статичні змінні від динамічних?
2. Чим характеризуються динамічні структури даних, коли і де виділяється пам’ять під динамічні змінні?

3. В чому переваги зв'язного представлення даних?
4. Як встановлюється зв'язок між елементами динамічної структури?
5. Що таке вказівник, коли і де він набуває значення?
6. В чому переваги і недоліки списків у порівнянні з масивами?
7. Який список називається порожнім?
8. Чим відрізняється початковий елемент списку від решти елементів списку?
9. Яким чином додати та видалити перший, останній та внутрішній елементи лінійного списку?
10. За якими критеріями можуть бути організовані елементи в багатозв'язних списках?
11. Що таке кільцевий список?
12. Як називається послідовний список зі змінною довжиною, включення і виключення елементів з якого виконуються тільки з одного боку?
13. Як називається послідовний список зі змінною довжиною, в якому включення елементів виконується з одного боку списку, а виключення – з іншого боку?
14. В чому переваги і недоліки двозв'язних списків у порівнянні з однозв'язними?
15. Створити однозв'язний лінійний список, елементами якого є натуральні числа. Надрукувати значення елементів, розташованих між найбільшим і найменшим елементами списку.
16. Створити однозв'язний лінійний список зі слів деякого рядка, розташувавши їх у списку за алфавітом. Визначити кількість повторень кожного слова у списку(Словом вважається обмежена пробілами послідовність символів).
17. Описати процедуру, що об'єднує два лінійних списки L1 і L2 в один за правилом L = L1 U L2.
18. Описати процедуру, що об'єднує два лінійних списки L1 і L2 в один за правилом L = L1 ∩ L2.
19. Описати процедуру, яка за лінійним списком L будує два нових списки: L1 - з позитивних елементів, L2 - з негативних елементів.
20. Описати процедуру, що об'єднує два лінійних списки L1 і L2 за правилом: елемент L1, елемент L2, елемент L1, елемент L2

т.д. Якщо один список коротший за інший, то елементи, що залишилися, записуються в звичайному порядку.

21. Описати процедуру, яка в лінійний список L вставляє новий елемент E за кожним входженням елемента B.
22. Створити двозв'язний лінійний список, елементами якого є слова тексту. Вивести слова, що знаходяться на парних позиціях під час перегляду списку у напрямку від голови до хвоста, та слова, розташовані на непарних позиціях під час перегляду списку у зворотному напрямку.
23. Створити двозв'язний лінійний список цілих чисел. Знайти елемент із введеним із клавіатури значенням. Вивести порядковий номер шуканого елемента, рухаючись з початку та з кінця списку.
24. Дано покажчик на перший елемент непорожнього двозв'язного списку. Продублювати в списку усі елементи з непарними номерами.
25. Дано покажчик на перший елемент непорожнього двозв'язного списку. Перетворити список у циклічний.

Розділ X
СТЕКИ ТА ЧЕРГИ

У цьому розділі триває обговорення списків, розпочате в 9 розділі, і описуються два особливі різновиди списків: стеки й черги. *Стек* - це список, у якому додавання й видалення елементів здійснюється з одного й того ж кінця списку. *Черга* - це список, у якому елементи додаються в один кінець списку, а віддаляються із протилежного кінця. Багато алгоритмів, включаючи деякі із представлених у наступних розділах посібника, використовують стеки й черги.

Стеки

Стек (`stack`) — це впорядкований список, у якому додавання й видалення елементів завжди відбувається на одному кінці списку. Можна уявити стек як купу предметів на підлозі. Ви можете додавати елементи на вершину й видаляти їх звідти, але не можете додавати або видаляти елементи із середини стопки.

Стеки часто називають *списками типу перший увійшов — останній вийшов* (`Last-In-First-Out list`). З історичних причин додавання елемента в стек називається *проштовхуванням* (`pushing`) елемента в стек, а видалення елемента зі стеку — *виштовхуванням* (`popping`) елемента зі стеку.

Перша реалізація простого списку на основі масиву, описана на початку 9 розділу, є стеком. Для відстеження вершини списку використовується лічильник. Потім цей лічильник використовується для вставлення або видалення елемента з вершини списку. Невелика зміна - це нова процедура `Pop`, яка видаляє елемент зі списку, одночасно повертаючи його значення. При цьому інші процедури можуть витягати елемент і видаляти його зі списку за один крок. Крім цієї зміни коду, код програми збігається з кодом, наведеним у 9 розділі.

```
Dim Stack() As Variant
Dim Stacksize As Variant
```

```
Sub Push(value As Variant)
  Stacksize = Stacksize + 1
  Redim Preserve Stack(1 To Stacksize)
  Stack(Stacksize) = value
End Sub
Sub Pop(value As Variant)
  value = Stack(Stacksize)
  Stacksize = Stacksize - 1
  Redim Preserve Stack(1 To Stacksize)
End Sub
```

Усі попередні міркування про списки також належать до цього виду реалізації стеків. Зокрема, можна заощадити час, якщо не змінювати розмір при кожному додаванні або виштовхуванні елемента. Програма `Simlist` на диску із прикладами, описана в 9 розділі, демонструє цей вид простої реалізації списків.

Програми часто використовують стеки для зберігання послідовності елементів, з якими програма буде працювати доти, поки стек не спорожніє. Дії з одним з елементів можуть приводити до того, що інші будуть проштовхуватися в стек, але, зрештою, вони всі будуть вилучені зі стеку. Як простий приклад можна навести алгоритм обігу порядку елементів масиву. При цьому всі елементи послідовно проштовхуються в стек. Потім усі елементи виштовхуються зі стеку у зворотному порядку й записуються назад у масив.

```
Dim List() As Variant
Dim Numitems As Integer
' Ініціалізація масиву.
  ' Проштовхнути елементи в стек.
For I = 1 To Numitems
  Push List(I)
Next I
' Виштовхнути елементи зі стеку назад у масив.
For I = 1 To Numitems
  Pop List(I)
Next I
```

У цьому прикладі довжина стеку може багаторазово змінюватися до того, як, зрештою, він спорожніє. Якщо відомо заздалегідь, наскільки великим повинен бути масив, можна відразу

створити досить великий стек. Замість зміни розміру стеку в міру того, як він росте та зменшується, можна відвести під нього пам'ять на початку роботи й знищити його після її завершення.

Наступний код дозволяє створити стек, якщо заздалегідь відомий його максимальний розмір. Процедура `Pop` не змінює розмір масиву. Коли програма закінчує роботу зі стеком, вона повинна викликати процедуру `Emptystack` для звільнення зайнятої під стек пам'яті.

```
Const WANT_FREE_PERCENT = .1     ' 10% вільного
  простору.
Const MIN_FREE = 10              ' Мінімальний розмір.
Global Stack() As Integer        ' Стековий масив.
Global Stacksize As Integer      ' Розмір стекового
  масиву.
Global Lastltem As Integer       ' Індекс останнього
  елемента.
Sub Preallocatestack(entries As Integer)
  Stacksize = entries
  Redim Stack(1 To Stacksize)
End Sub
Sub Emptystack()
  Stacksize = 0
  Lastitem = 0
  Erase Stack       ' Звільнити пам'ять, зайняту
  масивом.
End Sub
Sub Push(value As Integer)
  Lastitem = Lastitem + 1
  If Lastitem > Stacksize Then Resizestack
  Stack(Lastitem) = value
End Sub
Sub Pop(value As Integer)
  value = Stack(Lastitem)
  Lastitem = Lastitem - 1
End Sub
Sub Resizestack()
Dim want_free As Integer
  want_free = WANT_FREE_PERCENT * Lastitem
  If want_free < MIN_FREE Then want_free = MIN_FREE
  Stacksize = Lastitem + want_free
  Redim Preserve Stack(1 To Stacksize)
End Sub
```

Цей вид реалізації стеків досить ефективний у Visual Basic. Стек не витрачає дарма пам'ять і не занадто часто змінює свій розмір, особливо якщо відразу відомо, наскільки великим він повинен бути.

Множинні стеки

В одному масиві можна створити два стеки, помістивши один на початку масиву, а інший - наприкінці. Для двох стеків використовуються окремі лічильники довжини стеку `Top`, і стеки ростуть назустріч один одному, як показано на рис.10.1.

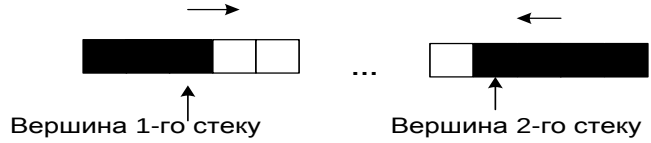


Рис.10.1. Два стеки в одному масиві

Цей метод дозволяє двом стекам рости, займаючи ту саму область пам'яті, доти, доки вони не зіштовхнуться, коли масив заповниться.

На жаль, міняти розмір цих стеків непросто. При збільшенні масиву необхідно зрушувати всі елементи у верхньому стеці, щоб виділяти пам'ять під нові елементи в середині. При зменшенні масиву необхідно спочатку зрушити елементи верхнього стеку перед тим, як міняти розмір масиву. Цей метод також складно масштабувати для оперування більш ніж двома стеками.

Зв'язні списки надають більш гнучкий метод побудови множинних стеків. Для проштовхування елемента в стек він міститься в початок зв'язного списку. Для виштовхування елемента зі стеку, видаляється перший елемент зі зв'язного списку. Тому що елементи додаються й видаляються тільки на початку списку, для реалізації стеків такого типу не потрібно застосування сигнальних позначок або двозв’язних списків.

Основний недолік застосування стеків на основі зв'язних списків полягає в тому, що вони вимагають додаткової пам'яті для зберігання покажчиків `Nextcell`. Для стеку на основі масиву, що містить `N` елементів, потрібно всього `2*N` байтів пам'яті (по 2 байти на ціле число). Той же стек, реалізований на основі зв'язного списку, вимагає додатково `4*N` байтів пам'яті для покажчиків `Nextcell`, збільшуючи розмір необхідної пам'яті втроє.

Програма Stack (диск з прикладами - папка ProgR10) використовує декілька стеків, реалізованих у вигляді зв'язних списків. Використовуючи програму, можна вставляти й виштовхувати елементи з кожного із цих списків. Програма Stack2 (диск з прикладами - папка ProgR10) аналогічна цій програмі, але вона використовує клас Linkedliststack для роботи зі стеками.

Черги

Упорядкований список, у якому елементи додаються до одного кінця списку, а віддаляються з іншого, називається *чергою* (queue). Група людей, що очікують обслуговування в магазині, утворюють чергу. Знову прибулі становляться позаду. Коли покупець доходить до початку черги, касир його обслуговує. Через їхню природу черги іноді називають *списками типу перший увійшов - перший вийшов* (First- In-First-Out list).

У Visual Basic черги можна реалізувати використовуючи методи для організації простих стеків. Створимо масив і за допомогою лічильників будемо визначати положення початку й кінця черги. Значення змінної Queuefront дає індекс елемента на початку черги. Змінна Queueback визначає, куди повинен бути доданий черговий елемент черги. У міру того як нові елементи додаються в чергу й залишають її, розмір масиву, що містить чергу, змінюється так, що він росте на одному кінці й зменшується на іншому.

```
Global Queue() As String         ' Масив черги.
Global Queuepront As Integer     ' Початок черги.
Global Queueback As Integer      ' Кінець черги.
Sub Enterqueue(value As String)
  Redim Preserve Queue(Queuefront To Queueback)
  Queue(Queueback) = value
  Queueback = Queueback + 1
End Sub
Sub Leavequeue(value As String)
  value = Queue(Queuefront)
  Queuefront = Queuefront + 1
  Redim Preserve Queue (Queuefront To Queueback - 1)
End Sub
```

На жаль, Visual Basic не дозволяє використовувати ключове слово `Preserve` в операторові `Redim`, якщо змінюється нижня границя масиву. Навіть якби Visual Basic дозволяв виконання такої операції, черга при цьому «рухалася» б по пам'яті. При кожному додаванні або видаленні елемента із черги, границі масиву збільшувалися б. Після пропущення досить великої кількості елементів через чергу, її границі могли б в остаточному підсумку стати занадто великі.

Тому, коли потрібно збільшити розмір масиву, спочатку необхідно перемістити дані в початок масиву. При цьому може утворюватися достатня кількість вільних комірок наприкінці масиву, так що збільшення розміру масиву може вже не знадобитися. А якщо ні, то можна скористатися оператором `Redim` для збільшення або зменшення розміру масиву.

Як і у випадку зі списками, можна підвищити продуктивність, додаючи відразу кілька елементів при збільшенні розміру масиву. Також можна заощадити час, зменшуючи розмір масиву, тільки коли він містить занадто багато невикористовуваних комірок.

У випадку простого списку або стеку, елементи додаються й віддаляються на одному його кінці. Якщо розмір списку залишається майже постійним, його не доведеться змінювати занадто часто. З іншого боку, тому що елементи додаються на одному кінці черги, а віддаляються з іншого кінця, може знадобитися час від часу переупорядковувати чергу, навіть якщо її розмір залишається незмінним.

```
Const WANT_FREE_PERCENT = .1     ' 10% вільного
  простору.
Const MIN_FREE = 10              ' Мінімум вільних
  гнізд.
Global Queue() As String         ' Масив черги.
Global Queuemax As Integer       ' Найбільший індекс
  масиву.
Global Queuefront As Integer     ' Початок черги.
Global Queueback As Integer      ' Кінець черги.
Global Resizewhen As Integer     ' Коли збільшити розмір
  масиву.
' При ініціалізації програма повинна встановити
  Queuemax = -1
' показуючи, що під масив ще не виділена пам'ять.
Sub Enterqueue(value As String)
```

```
  If Queueback > Queuemax Then Resizequeue
  Queue(Queueback) = value
  Queueback = Queueback + 1
End Sub
Sub Leavequeue(value As String)
  value = Queue(Queuefront)
  Queuefront = Queuefront + 1
  If Queuefront > Resizewhen Then Resizeoueue
End Sub
Sub Resizequeue()
Dim want_free As Integer
Dim i As Integer
  ' Перемістити записи в початок масиву.
  For i = Queuefront To Queueback - 1
      Queue(i - Queuefront) = Queue(i)
  Next i
  Queueback = Queueback - Queuepront
  Queuefront = 0
  ' Змінити розмір масиву.
  want_free = WANT_FREE_PERCENT * (Queueback -
  Queuefront)
  If want_free < MIN_FREE Then want_free = MIN_FREE
  Max = Queueback + want_free - 1
  Redim Preserve Queue(0 To Max)
  ' Якщо Queuefront > Resizewhen, змінити розмір
  масиву.
  Resizewhen = want_free
End Sub
```

Програма `Arrayq` (диск з прикладами - папка ProgR10) використовує цей метод для створення простої черги. Уведіть рядок і натисніть на кнопку **Enter** (Уведення) для додавання нового елемента в кінець черги. Натисніть кнопку **Leave** (Покинути) для видалення верхнього елемента із черги.

При роботі із програмою помітьте, що коли ви додаєте й видаляєте елементи, потрібна зміна розміру черги, навіть якщо розмір черги майже не змінюється. Фактично, навіть при кількаразовому додаванні й видаленні одного елемента розмір черги буде змінюватися.

Майте на увазі, що при кожній зміні розміру черги спочатку всі використовувані елементи переміщаються в початок масиву. При

цьому на зміну розміру черг на основі масиву йде більше часу, ніж на зміну розміру описаних вище зв'язних списків і стеків.

Програма `Arrayq` (диск з прикладами - папка ProgR10) аналогічна програмі `Arrayq`, але вона використовує для керування чергою клас `Arrayqueue`.

Циклічні черги

Черги, описані в попередньому підрозділі, потрібно переупорядковувати час від часу, навіть якщо розмір черги майже не змінюється. Навіть при кількаразовому додаванні й видаленні одного елемента необхідно переупорядковувати чергу.

Якщо заздалегідь відомо, наскільки велика може бути черга, переупорядкування можна уникнути, створивши *циклічну чергу* (`circular queue`). Ідея полягає в тому, щоб розглядати масив черги начебто він загортається, утворюючи коло. При цьому останній елемент масиву як би йде перед першим. На рис.10.2 зображена циклічна черга.

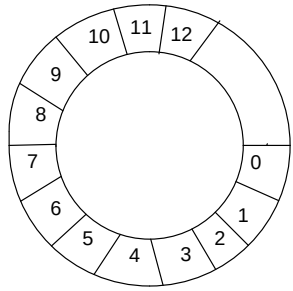


Рис.10.2. Циклічна черга

Програма може зберігати в змінній `Queuefront` індекс елемента, який найдовше перебуває в черзі. Змінна `Queueback` може містити кінець черги, у який додається новий елемент.

На відміну від попередньої реалізації, при відновленні значень змінних `Queuefront` і `Queueback` необхідно використовувати оператор `Mod` для того, щоб індекси залишалися в границях масиву. Наприклад код, що йде нижче, додає елемент до черги:

```
Queue(Queueback) = value
Queueback = (Queueback + 1) Mod Queuesize
```

На рис.10.3 показаний процес додавання нового елемента до циклічної черги, яка може містити чотири записи. Елемент C додається в кінець черги. Потім кінець черги зрушується, вказуючи на наступний запис у масиві.

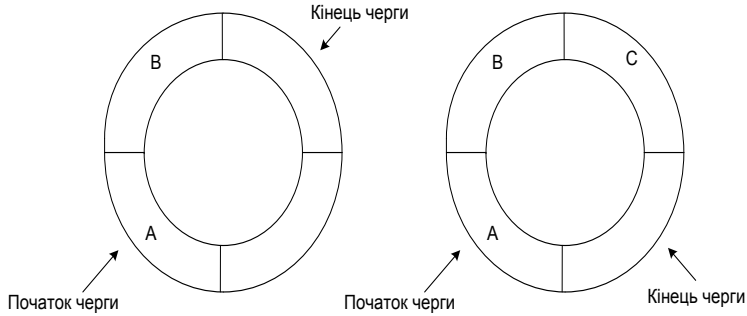


Рис.10.3. Додавання елемента до циклічної черги

У такий же спосіб, коли програма видаляє елемент із черги, необхідно обновляти покажчик на початок черги за допомогою такого коду:

```
value = Queue(Queuefront)
Queuefront = (Queuefront + 1) Mod Queuesize
```

На рис.10.4 показаний процес видалення елемента із циклічної черги. Перший елемент, у цьому випадку елемент `A`, віддаляється з початку черги, і покажчик на початок черги обновлюється, вказуючи на наступний елемент масиву.

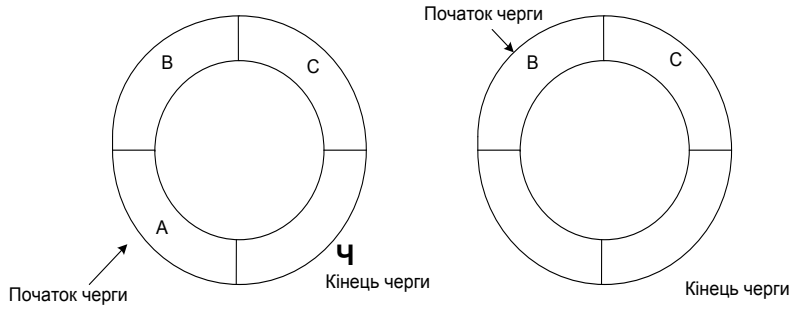


Рис.10.4. Видалення елемента із циклічної черги

Для циклічних черг іноді буває складно відрізнити порожню чергу від повної. В обох випадках значення змінних `Queuebottom` і `Queuetop` будуть рівні. На рис.10.5 показано дві циклічні черги, порожня й повна.

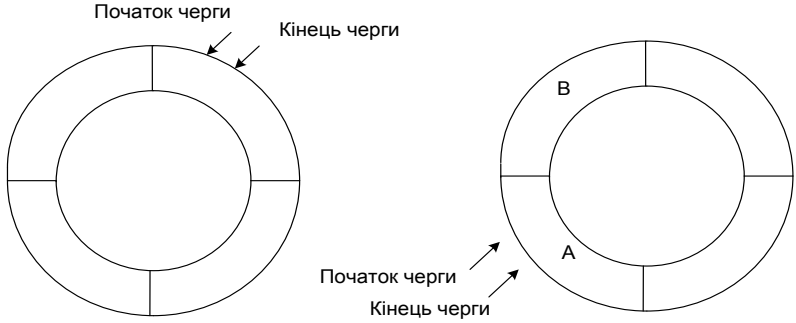


Рис.10.5. Порожня й повна циклічні черги

Простий варіант вирішення цієї проблеми - зберігати число елементів у черзі в окремій змінній `Numinqueue`. За допомогою цього лічильника можна довідатися, чи залишилися ще в черзі елементи і чи залишилося в черзі місце для нових елементів.

Наведений код використовує всі ці методи для керування циклічною чергою:

```
Queue() As String                    ' Масив черги.
Queuesize As Integer                 ' Найбільший індекс у
  черзі.
Queuefront As Integer                ' Початок черги.
```

```
Queueback As Integer              ' Кінець черги.
Numinqueue As Integer             ' Число елементів у
  черзі.
Sub Newcircularqueue(num_items As Integer)
  Queuesize = num_items
  Redim Queue(0 To Queuesize - 1)
End Sub
Sub Enterqueue(value As String)
  ' Якщо черга заповнена, вийти із процедури.
  ' У справжньому додатку буде потрібно більш складний
  код.
  If Numinqueue >= Queuesize Then Exit Sub
  Queue(Queueback) = value
  Queueback = (Queueback + 1) Mod Queuesize
  Numinqueue = Numinqueue + 1
End Sub
Sub Leavequeue (value As String)
  ' Якщо черга порожня, вийти із процедури.
  ' У справжньому додатку буде потрібно більш складний
  код.
  If Numinqueue <= 0 Then Exit Sub
  value = Queue (Queuefront)
  Queuefront = (Queuefront + 1) Mod Queuesize
  Numinqueue = Numinqueue - 1
End Sub
```

Так само як і у випадку зі списками на основі масивів, можна змінювати розмір масиву, коли черга повністю заповниться або якщо в масиві буде занадто багато невикористовуваного простору. Проте зміна розміру циклічної черги складніша, ніж зміна розміру стеку або списку, заснованого на масиві.

Коли змінюється розмір масиву, кінець черги може не збігатися з кінцем масиву. Якщо просто збільшити масив, то елементи, що вставляються, будуть перебувати наприкінці масиву, тому вони потраплять у середину списку. На рис.10.6 показано, що може відбутися при такому збільшенні масиву.

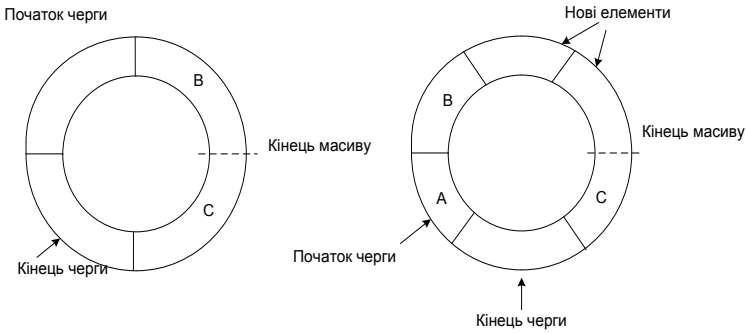


Рис.10.6. Неправильне збільшення розміру циклічної черги

При зменшенні розміру масиву виникають схожі проблеми. Якщо елементи огинають кінець масиву, то елементи наприкінці масиву, які будуть перебувати на початку черги, будуть загублені.

Для того щоб уникнути цих утруднень, необхідно переупорядкувати масив перед тим, як змінювати його розмір. Простіше всього це зробити, використовуючи тимчасовий масив. Скопіюємо елементи черги в тимчасовий масив у правильному порядку, поміняємо розмір масиву черги і потім скопіюємо елементи з тимчасового масиву назад у масив черги.

```
Private Sub Enterqueue(value As String)
  If Numinqueue >= Queuesize Then Resizequeue
  Queue(Queueback) = value
  Queueback = (Queueback + 1) Mod Queuesize
  Numinqueue = Numinqueue + 1
End Sub
Private Sub Leavequeue(value As String)
  If Numinqueue <= 0 Then Exit Sub
  value = Queue (Queuefront)
  Queuefront = (Queuefront + 1) Mod Queuesize
  Numinqueue = Numinqueue - 1
  If Numinqueue < Shrinkwhen Then Resizequeue
End Sub
Sub Resizequeue()
Dim temp() As String
Dim want_free As Integer
Dim i As Integer
```

```
    ' Скопіювати елементи в тимчасовий масив.
    Redim temp(0 To Numinqueue - 1)
    For i = 0 To Numinqueue - 1
        temp(i) = Queue((i + Queuefront) Mod Queuesize)
    Next i
    ' Змінити розмір масиву.
    want_free = WANT_FREE_PERCENT * Numinqueue
    If want_free < MIN_PREE Then want_free = MIN_FREE
    Queuesize = Numinqueue + want_free
    Redim Queue(0 To Queuesize - 1)
    For i = 0 To Numinqueue - 1
        Queue(i) = temp(i)
    Next i
    Queuefront = 0
    Queueback = Numinqueue
    ' Зменшити розмір масиву, якщо Nuninqueue <
    Shrinkwhen.
    Shrinkwhen = Queuesize - 2 * want_free
    ' Не міняти розмір невеликих черг. Це може викликати
    ' проблеми з "Redim temp(0 To Numinqueue - 1)" вище й
    ' просто нерозумно!
    If Shrinkwhen < 3 Then Shrinkwhen = 0
End Sub
```

Програма `Circleq` (диск з прикладами - папка ProgR10) демонструє цей підхід до реалізації циклічної черги. Уведіть рядок і натисніть кнопку **Enter** (Увести) для додавання нового елемента в чергу. Натисніть на кнопку **Leave** (Покинути) для видалення верхнього елемента із черги. Програма буде при необхідності змінювати розмір черги.

Програма `Circleq2` аналогічна програмі `Circleq`, але вона використовує для роботи із чергою клас `Circlequeue`.

Пам’ятайте, що при кожній зміні розміру черги в програмі вона копіює елементи в тимчасовий масив, змінює розмір черги, а потім копіює елементи назад. Ці додаткові кроки роблять зміну розміру циклічних черг більш повільним, ніж зміна розміру зв'язних списків і стеків. Навіть черги на основі масивів, у яких також потрібні додаткові дії для зміни розміру, не вимагають такого обсягу роботи.

З іншого боку, якщо кількість елементів у черзі не дуже змінюється і якщо правильно задати параметри зміни розміру, може ніколи не знадобитися міняти розмір масиву. Навіть якщо іноді це

все-таки доведеться робити, зменшення частоти цих змін вимагає додаткових зусиль на програмування.

Черги на основі зв'язних списків

Зовсім інший підхід до реалізації черг полягає у використанні двозв’язних списків. Для відстеження початку й кінця списку можна використовувати сигнальні позначки. Нові елементи додаються в чергу перед позначкою наприкінці черги, а елементи, що випливають за позначкою початку черги, видаляються. На рис.10.7 показаний двозв’язний список, який використовується як черга.

Додавати й видаляти елементи із двозв’язного списку легко, тому що в цьому випадку не буде потрібно застосовувати складних алгоритмів для зміни розміру. Перевага цього методу також у тому, що він інтуїтивно зрозуміліший у порівнянні із циклічною чергою на основі масиву. Недолік його в тому, що для покажчиків зв'язного списку `Nextcell` і `Prevcell` потрібна додаткова пам'ять. Відносно займаної пам'яті черги на основі зв'язних списків не менш ефективні, ніж циклічні списки.

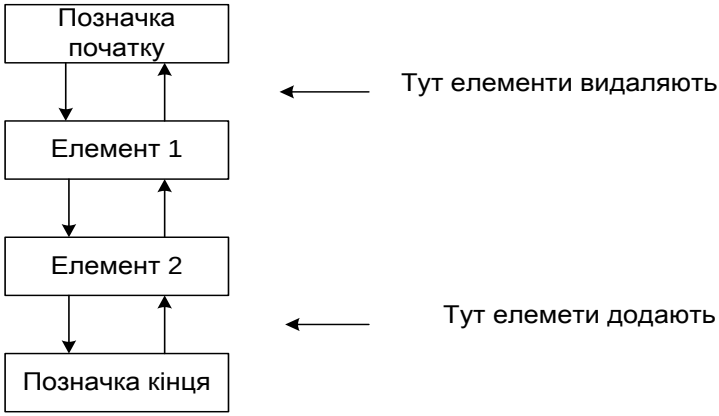


Рис.10.7. Черга на основі зв'язного списку

Програма `Linkedq` (диск з прикладами - папка ProgR10) на диску із прикладами працює із чергою за допомогою двозв’язного списку. Уведіть рядок, натисніть на кнопку **Enter**, щоб додати елемент у кінець черги. Натисніть на кнопку **Leave** для видалення елемента із черги.

Програма `Linkedq2` (диск з прикладами - папка ProgR10) аналогічна програмі `Linkedq`, але вона використовує для керування чергою клас `Linkedlistqueue`.

Застосування колекцій як черг

Колекції Visual Basic являють собою дуже просту форму черги. Програма може використовувати метод `Add` колекції для додавання елемента в кінець черги і метод `Remove` з параметром `1` для видалення першого елемента із черги. Наступний код управляє чергою на основі колекцій:

```
Dim Queue As New Collection
Private Sub Enterqueue(value As String)
  Queue.Add value
End Sub
Private Function Leavequeue() As String
  Leavequeue = Queue.Item(1)
  Queue.Remove 1
End Function
```

Незважаючи на те, що цей код дуже простий, колекції в дійсності не призначені для використання як черги. Вони надають додаткові можливості, такі як ключі елементів, і підтримка цих додаткових можливостей робить колекції більш повільними, ніж інші реалізації черг. Проте черги на основі колекцій настільки прості, що вони можуть бути прийнятним вибором для додатків, у яких продуктивність не є проблемою.

Програма `Collectq` (диск з прикладами - папка ProgR10) демонструє чергу на основі колекцій.

Пріоритетні черги

Кожний елемент у *пріоритетній черзі* (`priority queue`) має пов'язаний з ним пріоритет. Якщо програмі потрібно вилучити елемент із черги, вона вибирає елемент із найвищим пріоритетом. Як зберігаються елементи в пріоритетній черзі, не має значення, якщо програма завжди може знайти елемент із найвищим пріоритетом.

Деякі операційні системи використовують пріоритетні черги для планування завдань. В операційній системі UNIX усі процеси мають різні пріоритети. Коли процесор звільняється, вибирається готовий до виконання процес із найвищим пріоритетом. Процеси з більш низьким пріоритетом повинні чекати завершення або блокування (наприклад, при очікуванні зовнішньої події, такої як читання даних з диска) процесів з більш високими пріоритетами.

Концепція пріоритетних черг також використовується при управлінні авіаперевезеннями. Найвищий пріоритет мають літаки, у яких закінчується паливо під час посадки. Другий пріоритет мають літаки, що заходять на посадку. Третій пріоритет мають літаки, що перебувають на землі, тому що вони перебувають у більш безпечному становищі, ніж літаки в повітрі. Пріоритети змінюються згодом, тому що у літаків, які намагаються приземлитися, зрештою, закінчиться паливо.

Простий спосіб організації пріоритетної черги - помістити всі елементи в список. Якщо потрібно вилучити елемент із черги, можна знайти в списку елемент із найвищим пріоритетом. Щоб додати елемент у чергу, його заносять в початок списку. При використанні цього методу для додавання нового елемента в чергу потрібно тільки один крок. Щоб знайти й вилучити елемент із найвищим пріоритетом, потрібно O(N) кроків, якщо черга містить N елементів.

Дещо кращою була б схема з використанням зв'язного списку, у якому елементи були б упорядковані в прямому або зворотному порядку. Використовуваний у списку клас Prioritycell міг би повідомляти змінні в такий спосіб:

```
Public Priority As Integer        ' Пріоритет елемента.
Public Nextcell As Prioritycell   ' Покажчик на наступний
  елемент.
Public Value As String            ' Дані, потрібні
  програмі.
```

Щоб додати елемент у чергу, потрібно знайти його правильне положення в списку й занести його туди. Щоб спростити пошук положення елемента, можна використовувати сигнальні позначки на початку й кінці списку, надавши їм відповідні пріоритети. Наприклад, якщо елементи мають пріоритети від 0 до 100, можна надати позначці початку пріоритет 101 і позначці кінця – пріоритет 1. Пріоритети всіх реальних елементів будуть перебувати між цими значеннями.

На рис.10.8 показана пріоритетна черга, реалізована на основі зв'язного списку.

Наступний фрагмент коду показує ядро цієї процедури пошуку:

```
Dim cell As Prioritycell
Dim nxt As Prioritycell
  ' Знайти місце елемента в списку.
  cell = Topsentinel
  nxt = cell.Nextcell
  Do While cell.Priority > new_priority
       cell = nxt
       nxt = cell.Nextcell
  Loop
  ' Вставити елемент після гнізда в списку.
```

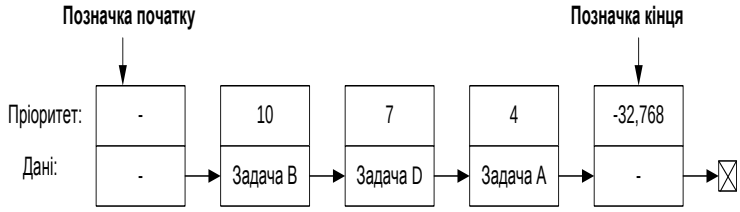


Рис.10.8. Черга з пріоритетами на основі зв'язного списку

Для видалення із списку елемента з найвищим пріоритетом, просто видаляють елемент після сигнальної позначки початку. Тому що список відсортований у порядку пріоритетів, перший елемент завжди має найвищий пріоритет.

Додавання нового елемента в цю чергу займає в середньому `N/2` кроків. Іноді новий елемент буде виявлятися на початку списку, іноді ближче до кінця, але в середньому він буде виявлятися десь в середині. Проста черга на основі списку вимагає `O(1)` кроків для додавання нового елемента й `O(N)` кроків для видалення елементів з найвищим пріоритетом із черги. Версія на основі впорядкованого зв'язного списку вимагає `O(N)` кроків для додавання елемента й `O(1)` кроків для видалення верхнього елемента. Обидві версії вимагають `O(N)` кроків для однієї із цих операцій, але у випадку впорядкованого зв'язного списку в середньому потрібно тільки `(N/2)` кроків.

Програма `Prilist` (диск з прикладами - папка ProgR10) використовує впорядкований зв'язний список для роботи із пріоритетною чергою. Ви можете задати пріоритет і значення елемента даних і натиснути кнопку **Enter** для додавання його в пріоритетну чергу. Натисніть на кнопку **Leave** для видалення із черги елемента з найвищим пріоритетом.

Програма `Prilist2` (диск з прикладами - папка ProgR10) аналогічна програмі `Prilist`, але вона використовує для керування чергою клас `Linkedpriorityqueue`.

Доклавши ще небагато зусиль, можна побудувати пріоритетну чергу, у якій додавання й видалення елемента потребують порядку O(log(N)) кроків. Для дуже великих черг, прискорення роботи може коштувати цих зусиль. Цей тип пріоритетних черг використовує структури даних у виді *піраміди*, які також застосовуються в алгоритмі пірамідального сортування. Піраміди й пріоритетні черги на їхній основі обговорюються більш докладно далі.

Багатопотокові черги

Цікавим різновидом черг є *багатопотокові черги* (`multi-headed queues`). Елементи, як звичайно, додаються в кінець черги, але черга має кілька *потоків* (`front end`) або *голів* (`heads`). Програма може видаляти елементи з будь-якого потоку.

Прикладом багатопотокової черги у звичайному житті є черга клієнтів у банку. Усі клієнти перебувають в одній черзі, але їх обслуговує декілька службовців. Банківський працівник, що звільнився, обслуговує клієнта, який перебуває в черзі першим. Такий порядок обслуговування видається слушним, оскільки клієнти обслуговуються в порядку прибуття. Він також ефективний, тому що всі службовці залишаються зайнятими доти, поки клієнти чекають у черзі.

Порівняйте цей тип черги з декількома однопотоковими чергами в супермаркеті, у яких покупці не обов'язково обслуговуються в порядку прибуття. Покупець у черзі, що повільно рухається, може прождати довше, ніж той, який підійшов пізніше, але опинився в черзі, яка просувається швидше. Касири також можуть бути не завжди зайняті, тому що якась черга може виявитися порожньою, тоді як в інших ще будуть перебувати покупці.

У загальному випадку багатопотокові черги більш ефективні, ніж кілька однопотокових черг. Останній варіант використовується в супермаркетах тому, що візки для покупок займають багато місця. При використанні багатопотокової черги всім покупцям довелося б об'єднатися в одну чергу. Коли касир звільнився б, покупцеві довелося би переміститися із громіздким візком до касира. З іншого боку, у банку відвідувачам не потрібно рухати великі візки для покупок, тому вони легко можуть уміститися в одній черзі.

Черги на реєстрацію в аеропорті іноді являють собою комбінацію цих двох ситуацій. Хоча пасажири мають із собою велику кількість багажу, в аеропорті все-таки використовуються багатопотокові черги, при цьому доводиться відводити додаткове місце, щоб пасажири могли вишикуватися по черзі.

Багатопотокову чергу просто побудувати, використовуючи звичайну однопотокову чергу. Елементи, що представляють клієнтів, зберігаються у звичайній однопотоковій черзі. Коли агент (касир, банківський службовець і т.д.) звільняється, перший елемент на початку черги видаляється й передається цьому агентові.

Модель черги

Припустимо, що ви відповідаєте за розробку лічильника реєстрації для нового термінала в аеропорті й прагнете зрівняти можливості однієї багатопотокової черги або декількох однопотокових. Вам буде потрібна якась модель поведінки пасажирів. Для цього прикладу можна зробити такі припущення:

- реєстрація кожного пасажира займає від двох до п'яти хвилин;
- при використанні декількох однопотокових черг пасажири, що прибувають, встають у найбільш коротку чергу;
- швидкість прибуття пасажирів приблизно незмінна.
- Програма `Headedq` (диск з прикладами - папка ProgR10) моделює цю ситуацію. Ви можете міняти деякі параметри моделі, включаючи такі:
- кількість пасажирів, що прибувають протягом години;
- мінімальний і максимальний затрачуваний час;
- кількість вільних службовців;
- паузу між кроками програми в мілісекундах.

При виконанні програми модель показує минулий час, середній і максимальний час очікування пасажирами обслуговування і відсоток часу, протягом якого службовці зайняті.

При експериментуванні з різними значеннями параметрів, ви помітите кілька цікавих моментів.

По-перше, для багатопотокової черги середній і максимальний час очікування буде менший. При цьому службовці також виявляються трохи більше завантаженими, ніж у випадку однопотокової черги. Для обох типів черги є поріг, при якому час очікування пасажирів значно зростає. Припустимо, що на обслуговування одного пасажира потрібно від 2 до 10 хвилин, або в середньому 6 хвилин. Якщо потік пасажирів становить 60 осіб у годину, тоді персонал витратить близько 6*60=360 хвилин у годину на обслуговування всіх пасажирів. Розділивши це значення на 60 хвилин у годині, одержимо, що для обслуговування клієнтів у цьому випадку буде потрібно 6 клерків.

Якщо запустити програму `Headedq` із цими параметрами, ви побачите, що черги рухаються досить швидко. Для багатопотокової черги час очікування становитиме всього кілька хвилин. Якщо додати ще одного службовця, щоб усього було 7 службовців, середній і максимальний час очікування значно зменшаться. Середній час очікування впаде приблизно до однієї десятої хвилини.

З іншого боку, якщо зменшити кількість службовців до 5, це приведе до великого збільшення середнього й максимального часу очікування. Ці показники також будуть рости згодом. Чим довше буде працювати програма, тим довше будуть затримки.

Таблиця 10.1

Час очікування у хвилинах та секундах для одно - і багатопотоковних черг

	Багатопоточна черга		**Однопоточна черга**	
Кількість службовців	Середній час	Максимальний час	Середній час	Максимальний час
5	11,37	20	12,62	20
6	1,58	5	6,93	13
7	0,11	2	0,54	6

У табл. 10.1 наведено середній і максимальний час очікування для 2 різних типів черг. Програма моделює роботу протягом 3 годин і припускає, що прибуває 60 пасажирів у годину й на обслуговування кожного з них іде від 2 до 10 хвилин.

Багатопотокова черга також здасться більш слушною, тому що пасажири обслуговуються в порядку прибуття. На рис.10.9 показана програма `Headedq` після моделювання ледве більш, ніж двох годин роботи термінала. У багатопотоковій черзі першим стоїть

пасажир з номером 104. Усі пасажири, що прибули до нього, уже обслужені або обслуговуються в даний момент. В однопотоковій черзі обслуговується пасажир з номером 106. Пасажири з номерами 100, 102, 103 і 105 усе ще чекають своєї черги, хоча вони й прибули раніше, ніж пасажир з номером 106.

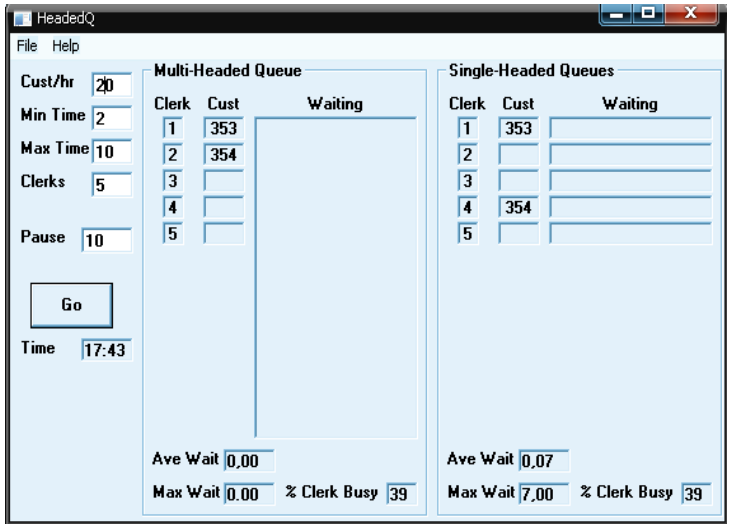


Рис.10.9. Вікно програми `Headedq`

Резюме

У розділі розглядаються два особливі різновиди списків: стеки й черги. Багато алгоритмів, включаючи деякі із представлених у наступних розділах посібника, використовують стеки й черги. Важливо знати, що різні реалізації стеків і черг мають неоднакові властивості. Стеки і циклічні черги на основі масивів прості й ефективні, особливо якщо заздалегідь відомий їхній потенційний розмір. Зв'язні списки забезпечують більшу гнучкість, якщо необхідно часто змінювати розмір списку. Володіючи знаннями можна вибрати структуру - стек або чергу, яка більше відповідає потребам алгоритму.

Контрольні запитання та завдання

1. Які основні операції характерні для стеку? В чому переваги і недоліки реалізації стеку за допомогою масиву і за допомогою списку?
2. Які операції можна виконувати зі стеком і чергою? Які покажчики для цього потрібні ?
3. Коли використовують циклічні черги? Як виконується додавання та видалення елемента до циклічної черги?
4. Як організують черги на основі зв'язних списків?
5. У чому перевага організації черг на основі колекцій?
6. У чому полягає концепція пріоритетних черг, де їх використовують?
7. У чому зміст багатопотокової черги? Дати приклади використання багатопотокових черг.
8. Файл містить текст з однаковою кількістю дужок, що відкриваються та закриваються. Побудувати стек або чергу, елементами яких є частини тексту, розташовані між парами відповідних одна одній дужок. Надрукувати номера позицій в тексті кожної пари дужок, що відкриваються та закриваються, наприклад: 8,10,12, 16 і т.д.
9. Реалізувати операцію додавання двох великих чисел, зображених у формі стеків. Значенням елемента стеку буде значення певної цифри числа. Під час обчислення суми її розряди також слід записувати у стек, а потім – виводити значення елементів цього стеку.
10. Використовуючи стек, надрукувати вміст текстового файлу T, друкуючи літери кожного його рядка у зворотному порядку.
11. Використовуючи стек, перевірити баланс відкриваючих та закриваючих дужок у заданому тексті.
12. Використовуючи чергу, за один перегляд файлу TT і без використання допоміжних файлів, надрукувати елементи файлу TT у такому порядку: спочатку – усі числа менші **a**, далі - усі числа з відрізку **[a, b],** на кінець – ті, що

залишилися, зберігаючи початковий взаємний порядок у кожній із цих трьох груп чисел.

13. Дано дві не порожні черги. Перемістити всі елементи першої черги в кінець другої.
14. Дана не порожня черга. Витягувати з черги елементи, поки чергове значення елемента черги не стане парним.
15. Використовуючи чергу, вирішити таку задачу: вміст текстового файлу F, розділений на рядки, переписати в текстовий файл G, переносячи при цьому в кінець кожного рядка всі цифри, які в нього входять (зі збереженням початкового взаємного порядку як серед цифр, так і серед решти символів рядка).
16. Мажоруючим елементом у масиві A [1..N] будемо називати елемент, що зустрічається в масиві більше ніж N/2 разів. Легко помітити, що в масиві може бути не більше одного мажоруючого елемента. Наприклад, у масиві {3, 3, 4, 2, 4, 4, 2, 4, 4} мажоруючий елемент 4, тоді як у масиві {3, 3, 4, 2, 4, 4, 2, 4} мажоруючого елемента немає. Визначити, чи є в масиві мажоруючий елемент, і якщо є, то надрукувати його (використати стек).

Розділ XI
ДЕРЕВА У VISUAL BASIC

Масиви і зв'язані списки визначають колекції об'єктів, доступ до яких здійснюється послідовно. Такі структури даних називають лінійними списками, оскільки вони мають унікальні перший і останній елементи і у кожного внутрішнього елемента є тільки один послідовник. Лінійний список є абстракцією, що дозволяє маніпулювати даними, які представляються різним чином – масивами, стеками, чергами і зв'язаними списками.

В багатьох додатках виявляється нелінійний порядок об'єктів, де елементи можуть мати декількох послідовників, наприклад, у генеалогічному дереві. Таке впорядкування називають ієрархічним. До ієрархічних впорядкувань відносять дерева.

Цей розділ присвячений поданню дерев у Visual Basic, розгляду алгоритмів обробки упорядкованих та збалансованих дерев.

Основні визначення

Дерево *– це нелінійна структура, що складається з вузлів і гілок та має напрям від кореня до зовнішніх вузлів(листів).*

Деревоподібна структура характеризується сукупністю вузлів, що походять від єдиного початкового вузла, названого коренем. Кожний вузол дерева є коренем піддерева, яке визначається даним вузлом і всіма його нащадками.

Рекурсивно дерево визначають як: порожню структуру або вузол, називаний *коренем* (`node`) дерева, який пов'язаний з нулем або більш піддеревами (`subtrees`).

На рис. 11.1 показано дерево, де кореневий вузол A пов'язаний із двома *піддеревами*, що починаються у вузлах B і C. Ці вузли пов'язані з піддеревами з коренями D, E, F і G і т.д.

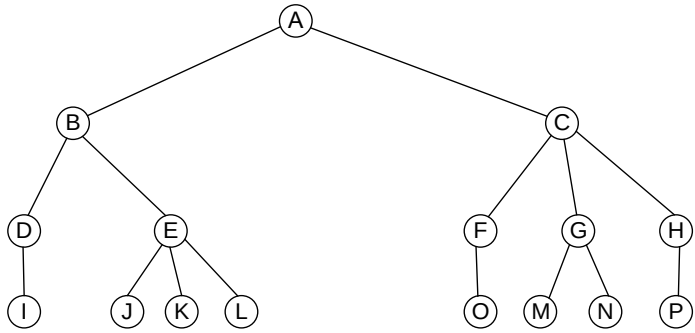


Рис. 11.1. Приклад дерева

Термінологія дерев являє собою суміш термінів, запозичених з ботаніки й генеалогії. З ботаніки прийшли такі терміни, як *вузол* (`node`), обумовлений як точка, у якій може починатися розгалуження, *гілка* (`branch`), обумовлена як зв'язок між двома вузлами, і *лист* (`leaf`) - вузол, з якого не виходять інші *гілки*.

З генеалогії прийшли терміни, які описують споріднення. Якщо один вузол перебуває безпосередньо над іншим, верхній вузол називається *батьком* (parent), а нижній *дочірнім вузлом* (`child`). Вузли на шляху нагору від вузла до кореня називаються *предками* (`ancestors`) вузла. Наприклад, на рис.11.1 вузли E, B і A - це всі предки вузла J.

Вузли, які перебувають нижче якого-небудь вузла дерева, називаються *нащадками* (`descendants`) цього вузла. Вузли D, E, I, J, K, L на рис.11.1 - це всі нащадки вузла B.

Іноді вузли, що мають одного батька, називаються *вузлами - братами* або *вузлами - сестрами* (`sibling nodes`).

Існує ще кілька термінів. *Внутрішнім вузлом* (`internal node`) називається вузол, який не є листом. *Порядком* вузла (`node degree`) називається число його дочірніх вузлів. Порядок дерева - це найбільший порядок його вузлів. Дерево на рис.11.1 - третього порядку, тому що вузли з найбільшим порядком C і E мають по 3 дочірніх вузла.

Глибина (`depth`) дерева дорівнює числу його предків плюс 1. На рис.11.1 глибина вузла E дорівнює 3. *Глибиною* (`depth`) або *висотою* (`height`) дерева називається найбільша глибина його вузлів. Глибина дерева на рис.11.1 дорівнює 4.

Дерево другого порядку називається *двійковим* деревом (`binary tree`). Дерева третього порядку іноді називаються *трійковими* (`ternary`) деревами. Більше того, дерева порядку N

іноді називаються N - арними (N-ary) деревами. Деякі уникають застосування зайвих термінів і просто говорять «дерева N-го порядку».

Подання дерев

Один зі способів подання дерев - створити окремий клас для кожного типу вузлів дерева. Для побудови дерева, поданого на рис.11.1, ви можете визначити структури даних для вузлів, які мають нуль, один, два або три дочірніх вузли. Цей підхід був би досить незручним. Крім того, якщо потрібно було б управляти чотирма різними класами, у класах потрібні були б які-небудь прапори, що вказували б тип дочірніх вузлів. Алгоритми, які оперували б цими деревами, мали б уміти працювати з усіма різними типами дерев.

Повні вузли

У якості простого розв'язання можна визначити один тип вузлів, який містить достатнє число покажчиків на нащадків для подання всіх потрібних вузлів. Звичайно такий підхід називають методом повних вузлів, тому що деякі вузли можуть бути більшого розміру, ніж необхідно насправді.

Дерево, зображене на рис 11.1, має 3-тій порядок. Для побудови цього дерева з використанням методу повних вузлів (fat nodes) потрібно визначити єдиний клас, який містить покажчики на три дочірні вузли. Наступний код демонструє, як ці покажчики можуть бути визначені в класі Ternarynode.

```
Public Leftchild As Ternarynode
Public Middlechild As Ternarynode
Public Rightchild As Ternarynode
```

За допомогою цього класу можна побудувати дерево, використовуючи записи Child вузлів, для зв'язку їх один з одним. Наступний фрагмент коду будує дерево, показане на рис.11.2.

```
Dim A As New Ternarynode
Dim B As New Ternarynode
Dim C As New Ternarynode
Dim D As New Ternarynode
  :
  Set A.Leftchild = B
  Set A.Middlechild = C
  Set A.Rightchild = D
```

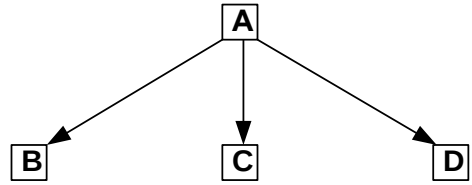


Рис. 11.2. Дерево, побудоване за допомогою класу ~~Child~~

Списки нащадків

Якщо порядки вузлів у дереві помітно різняться, метод повних вузлів приводить до даремної витрати великої кількості пам'яті.

Деякі програми додають і видаляють вузли, змінюючи порядок вузлів у процесі виконання. У цьому випадку метод повних вузлів не буде працювати. Такі динамічно змінювані дерева можна подати, помістивши дочірні вузли в списки. Є кілька підходів, що можна використовувати для створення списків дочірніх вузлів. Найбільш очевидний підхід полягає в створенні в класі вузла відкритого (`public`) масиву дочірніх вузлів, як показано в наступному коді. Тоді для оперування дочірніми вузлами можна використовувати методи роботи зі списками на основі масивів.

```
Public Children() As Treenode
Public Numchildren As Integer
```

На жаль, Visual Basic не дозволяє визначати відкриті масиви в класах. Це обмеження можна обійти, визначивши масив як закритий

(private) і оперуючи елементами масиву за допомогою процедур властивостей.

```
Private m_Chirdren() As Treenode
Private m_Numchildren As Integer
Property Get Children(Index As Integer) As Treenode
  Set Children = m_Children(Index)
End Property
Property Get Numchildren() As Integer
  Numchildren = m_Numchildren()
End Property
```

Другий підхід полягає в тому, щоб зберігати посилання на дочірні вузли у зв'язних списках. Кожний вузол містить посилання на першого нащадка. Він також містить посилання на наступного нащадка на тому ж рівні дерева. Ці зв'язки утворюють зв'язний список вузлів одного рівня, тому звичайно цей метод називають поданням у виді *зв'язного списку вузлів одного рівня* (linked sibling).

Третій підхід полягає в тому, щоб визначити в класі вузла відкриту колекцію, яка буде містити дочірні вузли:

```
Public Children As New Collection
```

Це розв’язання дозволяє використовувати всі переваги колекцій. Програма може при цьому легко додавати й видаляти елементи з колекції, присвоювати дочірнім вузлам ключі та використовувати оператор **For Each** для виконання циклів з посиланнями на дочірні вузли.

Програма Nary (диск з прикладами – папка ProgR11), показана на рис.11.3, використовує колекцію дочірніх вузлів для роботи з деревами порядку N. У цій програмі можна додавати до кожного вузла будь-яку кількість нащадків.

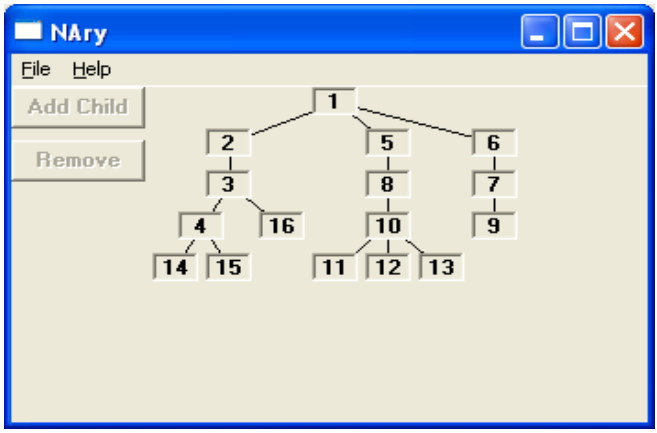


Рис. 11.3. Демонстраційна програма Nery

Для того щоб уникнути надмірного ускладнення користувацького інтерфейсу, програма `Nary` завжди додає нові вузли в кінець колекції дочірніх вузлів батька. Ви можете модифікувати цю програму, реалізувавши вставку дочірніх вузлів у середину колекції, але користувацький інтерфейс при цьому ускладниться.

Подання зв'язків нумерацією

Подання зв'язків нумерацією (`forward star`) дозволяє компактно представляти дерева, графи й мережі за допомогою масиву. Для подання дерева нумерацією зв'язків у масиві **`Firstlink`** записується індекс для перших гілок, що виходять із кожного вузла. В інший масив – **~~`Tonode`~~** заносяться вузли, до яких веде гілка.

Сигнальна мітка наприкінці масиву **`Firstlink`** указує на точку відразу після останнього елемента масиву **~~`Tonode`~~`.`** Це дозволяє легко визначити, які гілки виходять із кожного вузла. Гілки, що виходять із вузла **~~`I`~~**, перебувають під номерами від **`Firstlink(I)`** до **`Firstlink(I+1)-1`**. Для подання зв'язків, що виходять із вузла **~~`I`~~**, можна використовувати такий код:

```
For link = Firstlink(I) To Firstlink(I + 1) - 1
  Print Format$(I) & " -> " & Format$(Tonode(link))
Next link
```

Повні дерева

Повне дерево (complete tree) містить максимально можливе число вузлів на кожному рівні, крім нижнього. Усі вузли на нижньому рівні зрушуються вліво. Наприклад, кожний рівень трійкового дерева містить у точності три дочірні вузли, за винятком листів і, можливо, одного вузла на один рівень вище листів. На рис.11.4 показане повне трійкове дерево.

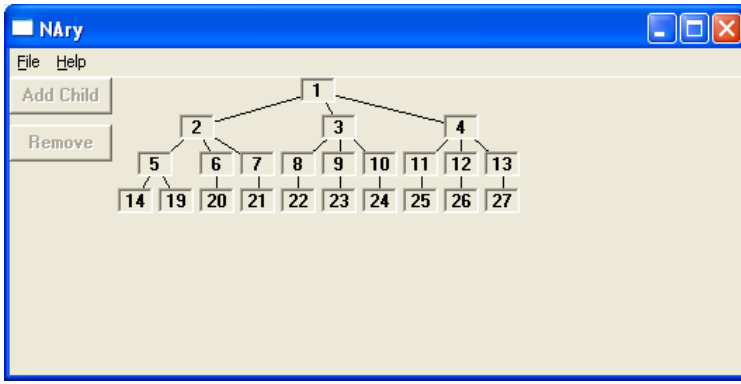


Рис. 11.4. Побудова трійкового дерева

Повні дерева володіють рядом важливих властивостей. По-перше, це найкоротші дерева, які можуть містити задане число вузлів.

По-друге, якщо повне дерево порядку `D` складається з `N` вузлів, воно матиме висоту порядку $O(\log_D(N))$ і $O(N)$ листів. Ці факти мають велике значення, оскільки багато алгоритмів обходять дерева зверху вниз або в протилежному напрямку. Час виконання алгоритму, що виконує одну із цих дій, буде порядку $O(N)$.

Надзвичайно корисна властивість повних дерев полягає в тому, що вони можуть бути дуже компактно записані в масивах. Якщо пронумерувати вузли в «природному» порядку, зверху вниз і зліва направо, то можна помістити елементи дерева в масив у цьому порядку.

При використанні цього методу записи дерева в масиві легко й просто одержати доступ до нащадків вузла. При цьому не потрібно додаткової пам'яті для колекцій дочірніх вузлів або міток у випадку подання нумерацією зв'язків. Читання й запис дерева у файл зводиться просто до збереження або читання масиву. Тому це

безсумнівно краще подання дерева для програм, які зберігають дані в повних деревах.

Обхід дерева

Послідовний обіг усіх вузлів називається *обходом* (`traversing`) дерева. Існує кілька послідовностей обходу вузлів двійкового дерева. Три найпростіші з них - прямий (`preorder`), симетричний (`inorder`) і зворотний (`postorder`) обходи описуються простими рекурсивними алгоритмами. Для кожного заданого вузла алгоритми виконують такі дії:

Прямий обхід:

1. Звертання до вузла.
2. Рекурсивний прямий обхід лівого піддерева.
3. Рекурсивний прямий обхід правого піддерева.

4. Симетричний обхід:

5. Рекурсивний симетричний обхід лівого піддерева.
6. Звертання до вузла.
7. Рекурсивний симетричний обхід лівого піддерева.

Зворотний обхід:

1. Рекурсивний зворотний обхід лівого піддерева.
2. Рекурсивний зворотний обхід правого піддерева.
3. Звертання до вузла.

Усі три порядки обходу є прикладами *обходу в глибину* (`depth - first traversal`). Обхід починається із проходу в глибину дерева доти, доки алгоритм не досягне листів. При поверненні з рекурсивного виклику підпрограми, алгоритм переміщається по дереву у зворотному напрямку, переглядаючи шляхи, які він пропустив при русі вниз.

Обхід у глибину зручно використовувати в алгоритмах, які мають спочатку обійти листи.

Четвертий метод перебору вузлів дерева - це *обхід завширшки* (`breadth - first traversal`). Цей метод застосовується до всіх вузлів на заданому рівні дерева, перед тим як перейти до більш глибоких рівнів. Алгоритми, які проводять повний пошук по дереву, часто використовують обхід завширшки.

Для дерев більше, ніж 2 порядку, має сенс визначати прямий, зворотний обхід, і обхід завширшки. Симетричний обхід визначається

неоднозначно, тому що звертання до кожного вузла може відбуватися після звертання до одного, двох, або трьох його нащадків. Наприклад, у трійковому дереві звертання до вузла може відбуватися після звертання до його першого нащадка або після звернення до другого нащадка.

Деталі реалізації обходу залежать від того, як описано дерево. Для обходу дерева на основі колекцій дочірніх вузлів програма має використовувати трохи інший алгоритм, ніж для обходу дерева, описаного за допомогою нумерації зв'язків.

Можна використовувати чергу для зберігання вузлів, які ще не були обійдені. Спочатку помістимо в чергу кореневий вузол. Після звертання до вузла, він видаляється з початку черги, а його нащадки поміщаються в її кінець. Процес повторюється доти, доки черга не спорожніє. Наступний код демонструє процедуру обходу завширшки для дерева, яке використовує вузли з колекціями нащадків:

```
Dim Root As Treenode
' Ініціалізація дерева.
  :
Private Sub Breadthfirstprint(}
Dim queue As New Collection      ' Черга на основі
  колекцій.
Dim node As Treenode
Dim child As Treenode
  ' Почати з кореня дерева в черзі.
  queue.Add Root
  ' Багаторазова обробка першого елемента
  ' у черзі, поки черга не спорожніє.
  Do While queue.Count > 0
       node = queue.Item(1)
       queue.Remove 1
       ' Звертання до вузла.
       Print Nodelabel(node)
       ' Помістити в чергу нащадків вузла.
       For Each child In node.Children
             queue.Add child
       Next child
  Loop
End Sub
```

Програма `Trav2` (диск з прикладами – папка `ProgR11`) демонструє обхід дерев, що використовують колекції дочірніх вузлів. Програма є об'єднанням програм ~~`Nary`~~, яка оперує деревами порядку `N`, і програми ~~`Trav1`~~, яка демонструє обходи дерев.

Оберіть вузол і натисніть на кнопку **Add Child** (Додати дочірній вузол), щоб додати до вузла нащадка. Натисніть на кнопки **Preorder**, **Inorder**, **Postorder** або **Breadth First**, щоб побачити приклади відповідних обходів. На рис.11.5 показана програма `Trav2`, яка відображає зворотний обхід.

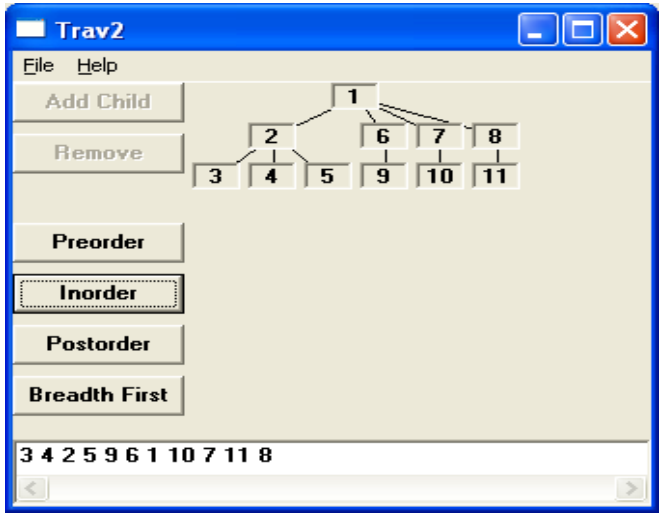


Рис. 11.5. Програма `Trav2`, яка відображає зворотний обхід

Упорядковані дерева

Двійкові дерева часто є природним засобом подання й обробки даних у комп'ютерних програмах. Оскільки багато комп'ютерних операцій є двійковими, вони природно перетворяться в операції із двійковими деревами. Наприклад, можна перетворити двійкове відношення «менше» у двійкове дерево. Якщо використовувати внутрішні вузли дерева для позначення того, що «лівий нащадок менше правого», ви можете використовувати двійкове дерево для запису впорядкованого списку. На рис.11.6 показане двійкове дерево, що містить упорядкований список із числами 1, 2, 4, 6, 7, 9.

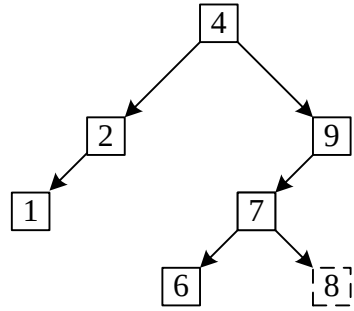


Рис. 11.6. Двійкове відношення «менше», перетворене у двійкове дерево

Додавання елементів

Алгоритм вставки нового елемента в дерево такого типу досить простий. Почнемо з кореневого вузла. По черзі порівняємо значення всіх вузлів зі значенням нового елемента. Якщо значення нового елемента менше або дорівнює значенню вузла, перейдемо вниз по лівій гілці дерева. Якщо нове значення більше, ніж значення вузла, перейдемо вниз по правій гілці. Коли цей процес дійде до листа, елемент поміститься у відповідне місце.

Щоб помістити значення 8 у дерево, показане на рис.11.6, ми починаємо з кореня, який має значення 4. Оскільки 8 більше, ніж 4, переходимо по правій гілці до вузла 9. Оскільки 8 менше 9, переходимо потім по лівій гілці до вузла 7. Оскільки 8 більше 7, знову намагаємося піти по правій гілці, але в цього вузла немає правого нащадка. Тому новий елемент вставляється в цій точці, і виходить дерево, показане на рис.11.6 (з пунктиром).

Наступний код додає нове значення нижче вузла в упорядкованому дереві. Програма починає вставку з кореня, викликаючи процедуру `Insertitem Root, new_value`.

```
Private Sub Insertitem(node As Sortnode, new_value As
  Integer)
Dim child As Sortnode
  If node Is Nothing Then
       ' Ми дійшли до листу.
       ' Вставити елемент тут.
```

```
        Set node = New Sortnode
        node.Value = new_value
        Maxbox = Maxbox + 1
        Load Nodelabel(Maxbox)
        Set node.Box = Nodelabel(Maxbox)
        With Nodelabel(Maxbox)
               .Caption = Format$(new_value)
               .Visible = True
        End With
   Elseif new_value <= node.Value Then
        ' Перейти по лівій гілці.
        Set child = node.Leftchild
        Insertitem child, new_value
        Set node.Leftchild = child
   Else
        ' Перейти по правій гілці.
        Set child = node.Rightchild
        Insertitem child, new_value
        Set node.Rightchild = child
   End If
End Sub
```

Коли ця процедура досягає кінця дерева, відбувається щось зовсім неочевидне. В Visual Basic, коли ви передаєте параметр підпрограмі, цей параметр *передається за посиланням*, якщо ви не використовуєте зарезервоване слово `Byval`. Це означає, що підпрограма працює з тією ж копією параметра, яку використовує викликаюча процедура. Якщо підпрограма змінює значення параметра, значення у викликаючій процедурі також змінюється.

Коли процедура `Insertitem` рекурсивно викликає сама себе, вона передає покажчик на дочірній вузол у дереві. Наприклад, у наступних операторах процедура передає покажчик на правого нащадка вузла як параметр вузла процедури `Insertitem`. Якщо викликувана процедура змінює значення параметра вузла, покажчик на нащадка також автоматично обновляється у викликаючій процедурі. Потім в останньому рядку коду значення правого нащадка встановлюється рівним новому значенню так, що створений новий вузол додається до дерева.

```
Set child = node.Rightchild
Insertltem child, new_value
Set node.Rightchild = child
```

Видалення елементів

Видалення елемента з упорядкованого дерева не набагато складніше, ніж його вставлення. Після видалення елемента програмі може знадобитися переупорядкувати інші вузли, щоб співвідношення «менше» продовжувало виконуватися для всього дерева. При цьому потрібно розглянути кілька випадків.

По-перше, якщо у вузла, що видаляється, немає нащадків, ви можете просто вилучити його з дерева, тому що порядок вузлів, що залишилися, при цьому не зміниться.

По-друге, якщо у вузла лише один дочірній вузол, ви можете помістити його на місце вилученого. Порядок інших нащадків вилученого вузла залишиться незмінним, оскільки вони є також нащадками й дочірнього вузла.

Якщо вузол, що видаляється, має два дочірні, то не обов'язково один з них займе місце вилученого. Якщо нащадки вузла також мають по два дочірніх вузли, то всі нащадки не зможуть зайняти місце вилученого вузла. Вилучений вузол має одного зайвого нащадка, і дочірній вузол, який ви прагнули б помістити на його місце, також має два нащадки, так що на вузол припало б три нащадки.

Щоб розв'язати цю проблему, вилучений вузол заміняється найправішим вузлом з лівої гілки. Інакше кажучи, потрібно зрушити на один крок униз по лівій гілці, що виходила з вилученого вузла. Потім потрібно рухатися по правих гілках униз доти, доки не знайдеться вузол, який не має правої гілки. Це найправіший вузол на гілці ліворуч від вузла, що видаляється.

Залишається останній варіант: вузол, що заміняється, має лівого нащадка. У нашому випадку можна перемістити цього нащадка на місце, що звільнилося в результаті переміщення вузла, що заміщається, і дерево знову буде розташовано в потрібному порядку. Уже відомо, що найправіший вузол не має правого нащадка, інакше він не був би таким. Це означає, що не потрібно турбуватися, чи не має вузол, що заміщується, двох нащадків.

На рис.11.7. наведене вихідне дерево (а), з якого послідовно видаляються вершини із ключами 13, 15, 5 і 10.

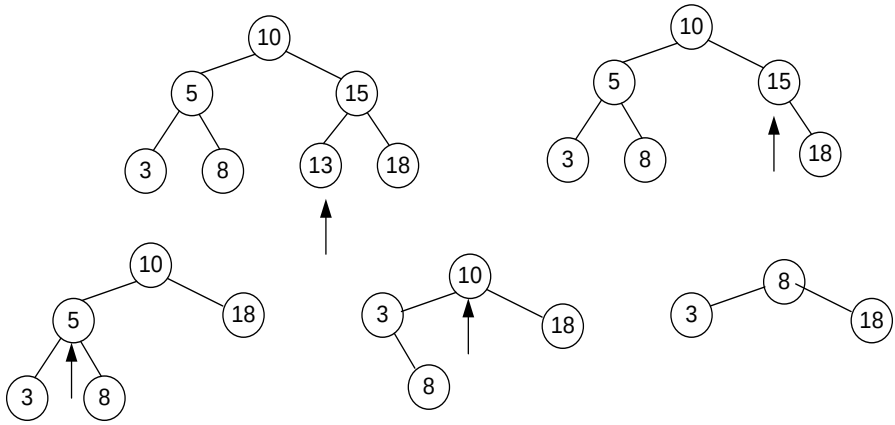


Рис. 11.7. Вихідне дерево

```
Private Sub Deleteitem(node As Sortnode, target_value
  As Integer)
Dim target As Sortnode
Dim child As Sortnode

  ' Якщо вузол не знайдено, вивести повідомлення.
  If node Is Nothing Then
       Beep
       Msgbox "Item " & Format$(target_value) & _
              " не знайдений у дереві."
       Exit Sub
  End If

  If target_value < node.Value Then
       ' Продовжити для лівого піддерева.
       Set child = node.Leftchild
       Deleteitem child, target_value
       Set node.Leftchild = child
  Elseif target_value > node.Value Then
       ' Продовжити для правого піддерева.
       Set child = node.Rightchild
       Deleteitem child, target_value
       Set node.Rightchild = child
  Else
       ' Шуканий вузол знайдено.
       Set target = node
       If target.Leftchild Is Nothing Then
              ' Замінити шуканий вузол його правим
  нащадком.
```

```
            Set node = node.Rightchild
        Elseif target.Rightchild Is Nothing Then
            ' Замінити шуканий вузол його лівим
  нащадком.
            Set node = node.Leftchild
        Else
            ' Виклик підпрограми Replacerightmost для
  заміни
            ' шуканого вузла найправішим вузлом
            ' у його лівій гілці.
            Set child = node.Leftchild
            Replacerightmost node, child
            Set node.Leftchild = child
        End If
  End If
End Sub

Private Sub Replacerightmost(target As Sortnode, repl
  As Sortnode)
Dim old_repl As Sortnode
Dim child As Sortnode

  If Not (repl.Rightchild Is Nothing) Then
      ' Продовжити рух вправо й униз.
      Set child = repl.Rightchild
      Replacerightmost target, child
      Set repl.Rightchild = child
  Else
      ' Досягли дна.
      ' Запам'ятати вузол, що заміняє, repl.
      Set old_repl = repl

      ' Замінити вузол repl його лівим нащадком.
      Set repl = repl.Leftchild

      ' Замінити шуканий вузол target with repl.
      Set old_repl.Leftchild = target.Leftchild
      Set old_repl.Rightchild = target.Rightchild
      Set target = old_repl
  End If
End Sub
```

Алгоритм використовує у двох місцях, приймання, передачу параметрів у рекурсивні підпрограми за посиланням. По-перше, підпрограма `Deleteitem` використовує це приймання, для того щоб батько шуканого вузла вказував на вузол, що заміняє. Наступні оператори показують, як викликається підпрограма `Deleteitem`:

```
Set child = node.Leftchild
Deleteitem child, target_value
Set node.Leftchild = child
```

Коли процедура виявляє шуканий вузол, вона одержує в якості параметра вузла покажчик батька на шуканий вузол. Встановлюючи параметр, що заміщає на вузол, підпрограма `Deleteitem` задає дочірній вузол для батька так, щоб він указував на новий вузол.

Наступні оператори показують, як процедура `Replacerightmost` рекурсивно викликає себе:

```
Set child = repl.Rightchild
Replacerightmost target, child
Set repl.Rightchild = child
```

Коли процедура знаходить найправіший вузол у лівій від вузла, що видаляється, гілці, у **параметрі** `repl` перебуває покажчик батька на найправіший вузол. Коли процедура встановлює значення **`repl`** рівним **`repl.Leftchild`**, вона автоматично з'єднує батька найправішого вузла з лівим дочірнім вузлом найправішого вузла.

Програма `Treesort (диск з прикладами – папка ProgR11)` використовує ці процедури для роботи з упорядкованими двійковими деревами. Введіть ціле число і натисніть на кнопку **Add**, щоб додати елемент до дерева. Введіть ціле число і натисніть на кнопку **Remove**, щоб вилучити цей елемент із дерева. Після видалення вузла дерево автоматично переупорядковується для збереження порядку «менше».

Обхід упорядкованих дерев

Корисна властивість упорядкованих дерев полягає в тому, що їхній порядок збігається з порядком симетричного обходу.

Ця властивість симетричного обходу впорядкованих дерев приводить до простого алгоритму сортування:

1. Додати елемент до впорядкованого дерева.
2. Вивести елементи, використовуючи симетричний обхід.

Цей алгоритм звичайно працює досить добре. Проте, якщо додавати елементи до дерева в певному порядку, то дерево може стати високим і тонким. На рис.11.9 показано впорядковане дерево, яке виходить при додаванні до нього елементів у порядку 1, 6, 5, 2, 3, 4. Інші послідовності також можуть приводити до появи високих і тонких дерев.

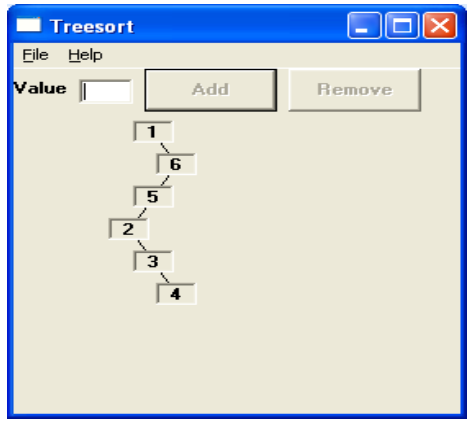


Рис. 11.9. Впорядковане дерево після додавання до нього елементів

Чим вище стає впорядковане дерево, тим більше часу потрібно для додавання нових елементів у нижню частину дерева. У найгіршому випадку після додавання `N` елементів дерево матиме висоту порядку `O(N)`. Повний час вставки всіх елементів у дерево буде за такого порядку $O(N^2)$. Оскільки для обходу дерева потрібен час порядку `O(N)`, повний час сортування чисел з використанням дерева дорівнюватиме $O(N^2)+O(N)=O(N^2)$.

Якщо дерево залишається досить коротким, воно має висоту порядку `O(log(N))`. У цьому випадку для вставки елемента в дерево буде потрібно лише порядку `O(log(N))` кроків. Вставка всіх `N` елементів у дерево зажадає порядку `O(N * log(N))` кроків. Тоді сортування елементів за допомогою дерева зажадає часу порядку `O(N * log(N)) + O(N) = O(N * log(N))`.

Час виконання порядку `O(N * log(N))` набагато менший, ніж $O(N^2)$. Наприклад, побудова високого й тонкого дерева, що містить 1000 елементів, зажадає виконання близько мільйона кроків. Побудова короткого дерева з висотою порядку `O(log(N))` займе лише близько 10.000 кроків.

Якщо елементи спочатку розташовані у випадковому порядку, форма дерева буде чимось середнім між цими двома крайніми випадками. Хоча його висота може виявитися трохи більшою, ніж `log(N)`, воно, швидше за все, не буде занадто тонким і високим, тому алгоритм сортування виконуватиметься досить швидко.

Збалансовані дерева

Форма впорядкованого дерева залежить від порядку вставки в нього нових вузлів. Високі й тонкі дерева можуть мати глибину порядку `O(N)`. Вставка або пошук елемента в такому незбалансованому дереві може займати порядку `O(N)` кроків. Навіть якщо нові елементи вставляються у дерево у випадковому порядку, у середньому вони дадуть дерево із глибиною `N / 2`, що також порядку `O(N)`.

Припустимо: будується впорядковане двійкове дерево, що містить `1000` вузлів. Якщо дерево збалансоване, то висота дерева буде порядку $\log_2(1000)$, або приблизно дорівнюватиме `10`. Вставка нового елемента в дерево займе лише `10` кроків. Якщо дерево високе й тонке, воно може мати висоту `1000`. У цьому випадку, вставка елемента в кінець дерева займе `1000` кроків.

Припустимо тепер, що ми хочемо додати до дерева ще `1000` вузлів. Якщо дерево залишається збалансованим, то всі `1000` вузлів помістяться на наступному рівні дерева. При цьому для вставки нових елементів буде потрібно близько `10 * 1000 = 10.000` кроків. Якщо дерево було не збалансоване й залишається таким у процесі росту, то при вставці кожного нового елемента воно буде ставати усе вище. Вставка елементів при цьому зажадає порядку `1000 + 1001 + … +2000 = 1,5` мільйона кроків.

Хоча не можна бути впевненим, що елементи будуть додаватися й видалятися з дерева в потрібному порядку, можна використовувати методи, які будуть підтримувати збалансованість дерева, незалежно від порядку вставки або видалення елементів.

АВЛ- дерева

АВЛ - дерева (AVL trees) були названі на честь російських математиків Адельсона - Вельського й Лендіса, які їх винайшли. Для кожного вузла АВЛ-дерева, висота лівого й правого піддерев відрізняється не більше, ніж на одиницю. На рис. 11.10 показано декілька АВЛ-дерев.

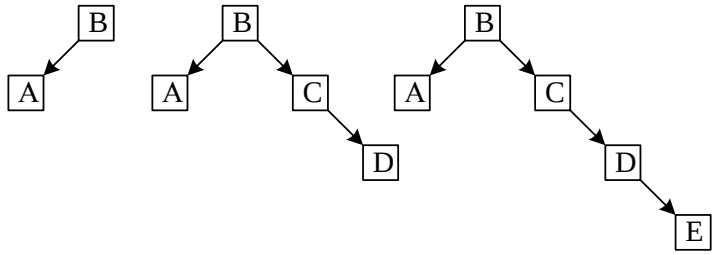


Рис. 11.10. Збалансовані дерева висотою 2, 3, 4

Хоча АВЛ-дерево може бути трохи вищим, ніж повне дерево з тим самим числом вузлів, воно також має висоту порядку `O(log(N))`. Це означає, що пошук вузла в АВЛ-дереві забирає час порядку `O(log(N))`, що досить швидко. Не настільки очевидно, як можна вставити або видалити елемент із АВЛ-дерева за час порядку `O(log(N))`, зберігаючи при цьому порядок дерева.

Процедура, що вставляє в дерево новий вузол, рекурсивно спускається донизу по дереву, щоб знайти місце розташування вузла.

Після вставки елемента відбуваються повернення з рекурсивних викликів процедури й зворотний прохід угору по дереву. При кожному поверненні із процедури перевіряється, чи зберігається усе ще властивість АВЛ-дерев на верхньому рівні. Цей тип зворотної рекурсії, коли процедура виконує важливі дії при виході з ланцюжка рекурсивних викликів, називається *висхідною*(`bottom - up`) рекурсією.

При зворотному проході угору по дереву процедура також перевіряє, чи не змінилася висота піддерева, з яким вона працює. Якщо процедура доходить до точки, у якій висота піддерева не змінилася, то висота наступних піддерев також не могла змінитися. У цьому випадку знову потрібне балансування дерева, і процедура може закінчити перевірку.

Розглянемо, що може відбутися при включенні в збалансоване дерево нової вершини. Якщо в нас є корінь r і ліве (`L`) і

праве (R) піддерева, то необхідно розрізняти три можливих випадки. Припустимо, включення в L нової вершини приведе до збільшення на 1 його висоти, тоді можливі три випадки:

1. $h_L=h_R$ – L і R стануть різної висоти, але критерій збалансованості не буде порушений.
2. $h_L<h_R$ – L і R стануть однієї висоти, тобто збалансованість навіть покращиться.
3. $h_L>h_R$ – критерій збалансованості порушиться, і дерево необхідно перебудовувати.

Візьмемо дерево, представлене на рис. 11.11. Вершини із ключами 9 і 11 можна включити, не порушуючи збалансованості дерева, дерево з коренем 10 стає однобічним (випадок 1), а з коренем 8 - лише краще збалансованим (випадок 2). Однак включення ключів 1, 3, 5 або 7 вимагає наступного балансування.

При уважному вивченні цієї ситуації виявляється, що існують лише дві по суті різні можливості, що вимагають індивідуального підходу. Усі інші можуть бути виведені із цих двох на основі симетрії. Перший випадок виникає при включенні в дерево на рис. 11.11 ключів 1 або 3; ситуація, характерна для другого випадку, виникає при включенні ключів 5 або 7.

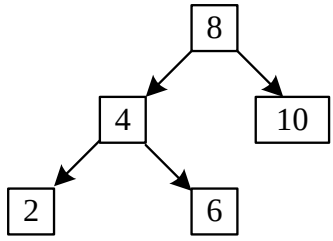


Рис. 11.11. Збалансоване дерево

Схематично ці випадки представлені на рис. 11.12, де прямокутниками позначені піддерева, причому "додана" при включенні висота відзначена підкресленням. Прості перетворення відразу ж відновлюють бажану збалансованість. Їхній результат наведений на рис. 11.13. Зверніть увагу, що допускаються лише переміщення у вертикальному напрямку, тоді як відносне горизонтальне розташування показаних вершин і піддерев має залишатися без зміни.

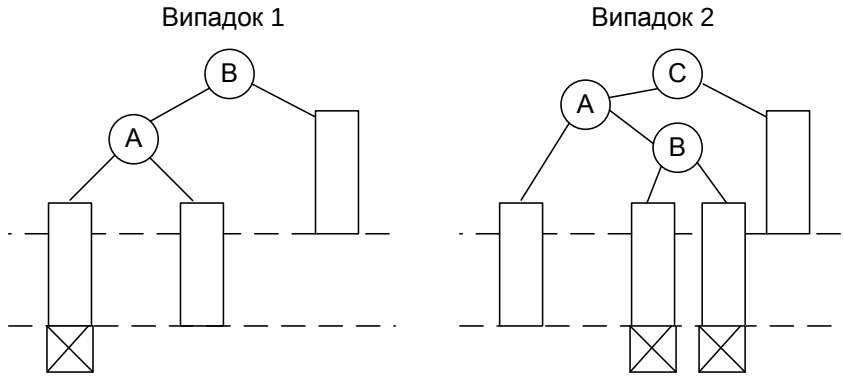


Рис. 11.12. Незбалансованість, що виникла через включення

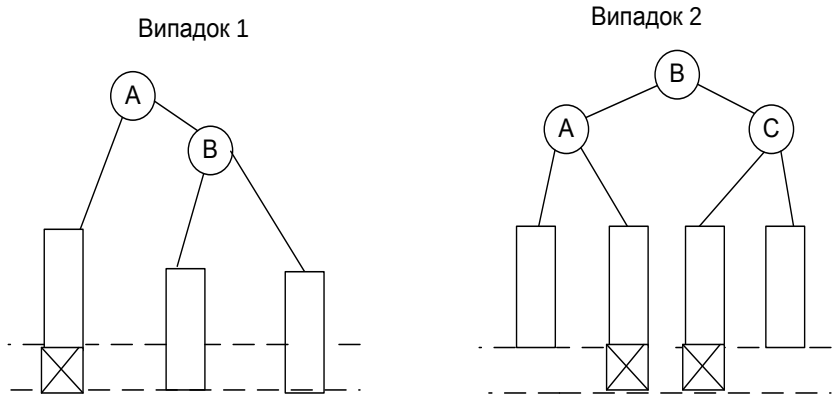


Рис. 11.13. Відновлення збалансованості

Обертання АВЛ-дерев

При вставці вузла в АВЛ-дерево, залежно від того, у яку частину дерева додається вузол, існує чотири варіанти балансування. Ці способи називаються правим і лівим обертанням, і обертанням вліво-вправо й вправо-вліво позначаються R, L, LR і RL.

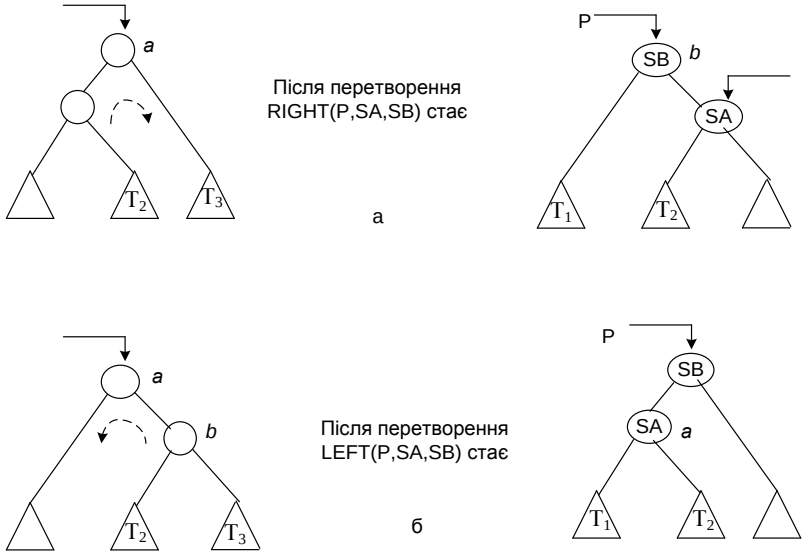


Рис. 11.14. Правий і лівий повороти

Правий і лівий повороти зображені відповідно на рис.11.14, а й б.

Вставка вузлів мовою Visual Basic

Крім звичайних полів `LeftChild` і `RightChild` клас `AVLNode` містить також поле `Balance`, що вказує, що з піддерев вузла вище. Його значення дорівнює -1, якщо ліве піддерево вище, 1 — якщо вище праве, 0 — якщо обидва піддерева мають однакову висоту.

```
Public LeftChild As AVLNode
Public RightChild As AVLNode
Public Balance As Integer
```

Щоб зробити код простішим для читання, можна використовувати постійні `LEFT_HEAVY`, `RIGHT_HEAVY`, і `BALANCED` для подання цих значень.

```
Global Const LEFT_HEAVY = -1
Global Const BALANCED = 0
Global Const RIGHT_HEAVY = 1
```

Процедура `InsertItem`, представлена нижче, рекурсивно спускається униз по дереву в пошуку нового місця розташування елемента. Коли вона доходить до нижнього рівня дерева, то створює новий вузол і вставляє його в дерево.

Потім процедура `InsertItem` використовує висхідну рекурсію для балансування дерева. При виході з рекурсивних викликів процедури вона рухається назад по дереву. При кожному поверненні із процедури, воно встановлює параметр `has_grown`, щоб визначити, чи збільшилася висота піддерева, яке вона залишає. В екземплярі процедури `InsertItem`, що зробив цей рекурсивний виклик, процедура використовує цей параметр для визначення того, чи є дерево, що перевіряється, незбалансованим. Якщо це так, то процедура застосовує для балансування дерева відповідне обертання.

Припустимо, що процедура в цей момент звертається до вузла `X`. Припустимо, що вона перед цим зверталася до правого піддерева знизу від вузла `X`, а параметр `has_grown` дорівнює `true`, означаючи, що праве піддерево збільшилося. Якщо піддерева вузла `X` до цього мали однакову висоту, тоді праве піддерево стане тепер вище лівого. У цій точці дерево збалансоване, але піддерево з коренем у вузлі `X` виросло, тому що виросло його праве піддерево.

Якщо ліве піддерево вузла `X` спочатку було вищим, ніж праве, то ліве й праве піддерева тепер матимуть однакову висоту. Висота піддерева з коренем у вузлі `X` не змінилася — вона, як і раніше, дорівнює висоті лівого піддерева плюс 1. У цьому випадку процедура `InsertItem` установить значення змінної `has_grown` рівним `false`, показуючи, що дерево збалансоване.

Зрештою, якщо праве піддерево вузла `X` було спочатку вище лівого, то вставка нового вузла робить дерево незбалансованим у вузлі `X`. Процедура `InsertItem` викликає підпрограму `RebalanceRigthGrew` для балансування дерева. Процедура `RebalanceRigthGrew` виконує ліве обертання або обертання вправо-уліво, залежно від ситуації.

Якщо новий елемент вставляється в ліве піддерево, то підпрограма `InsertItem` виконує аналогічну процедуру.

Видалення вузла з АВЛ-дерева

Раніше було показано, що видалити елемент із упорядкованого дерева складніше, ніж вставити його. Якщо видаляється елемент, що має лише одного нащадка, то можна замінити його цим нащадком, зберігши при цьому порядок дерева. Якщо в дерева два дочірніх вузли, то він заміняється на найправіший вузол у лівій гільці дерева. Якщо в цього вузла існує лівий нащадок, то цей лівий нащадок також займає його місце.

Оскільки АВЛ- дерева є особливим типом упорядкованих дерев, то для них потрібно виконати ті ж самі кроки. Проте після їхнього завершення необхідно повернутися назад по дереву, щоб переконатися у тому, що воно залишилося збалансованим. Якщо знайдеться вузол, для якого не виконується властивість АВЛ- дерев, то потрібно виконати для балансування дерева відповідне обертання. Хоча це ті ж самі обертання, які використовувалися раніше для вставки вузла в дерево, вони застосовуються в інших випадках.

Реалізація видалення вузлів мовою Visual Basic

Підпрограма `DeleteItem` видаляє елементи з дерева. Вона рекурсивно спускається по дереву в пошуку елемента, що видаляється, і коли вона знаходить шуканий вузол, то видаляє його. Якщо в цього вузла немає нащадків, то процедура завершується. Якщо є тільки один нащадок, то процедура заміняє вузол його нащадком.

Якщо вузол має двох нащадків, процедура `DeleteItem` викликає процедуру `ReplaceRightMost` для заміни шуканого вузла найправішим вузлом у його лівій галузі. Процедура `ReplaceRightMost` виконується приблизно так само, як і процедура, що видаляє елементи зі звичайного (неупорядкованого) дерева. Основна відмінність виникає при поверненні із процедури й рекурсивного проходу нагору по дереву. При цьому процедура `ReplaceRightMost` використовує висхідну рекурсію, щоб переконатися, що дерево залишається збалансованим для всіх вузлів.

```
Public Sub InsertItem(node As AVLNode, parent As
  AVLNode, _
  txt As String, has_grown As Boolean)
Dim child As AVLNode
```

```
' Якщо це нижній рівень дерева, помістити
' у батька покажчик на новий вузол.
If parent Is Nothing Then
    Set parent = node
    parent.Balance = BALANCED
    has_grown = True
    Exit Sub
End If

' Продовжити з лівим і правим піддеревами.
If txt <= parent.Box.Caption Then
    ' Вставити нащадка в ліве піддерево.
    Set child = parent.LeftChild
    InsertItem node, child, txt, has_grown
    Set parent.LeftChild = child

    ' Перевірити, чи потрібне балансування. Воно буде
    ' не потрібне, якщо вставка вузла не порушила
    ' балансування дерева або воно вже було
збалансоване
    ' на більш глибокому рівні рекурсії. У кожному
разі
    ' значення змінної has_grown дорівнюватиме False.
    If Not has_grown Then Exit Sub

    If parent.Balance = RIGHT_HEAVY Then
          ' Переважала права галузка, тепер баланс
          ' відновлений. Це піддерево не виросло,
          ' тому дерево збалансоване.
          parent.Balance = BALANCED
          has_grown = False
    ElseIf parent.Balance = BALANCED Then
          ' Було збалансовано, тепер переважує ліва
галузь.
          ' Поддерево усе ще збалансовано, але воно
виросло,
          ' тому необхідно продовжити перевірку
дерева.
          parent.Balance = LEFT_HEAVY
    Else
          ' Переважала ліва галузка, залишилося
незбалансоване.
          ' Виконати обертання для балансування на
рівні
          ' цього вузла.
          RebalanceLeftGrew parent
          has_grown = False
```

```
        End If       ' Закінчити перевірку балансування
    цього вузла.
    Else
        ' Вставити нащадка в праве піддерево.
        Set child = parent.RightChild
        InsertItem node, child, txt, has_grown
        Set parent.RightChild = child

        ' Перевірити, чи потрібне балансування. Воно буде
        ' не потрібне, якщо вставка вузла не порушила
        ' балансування дерева або воно вже було
    збалансоване
        ' на більш глибокому рівні рекурсії. У кожному
    разі
        ' значення змінної has_grown дорівнюватиме False.
        If Not has_grown Then Exit Sub

        If parent.Balance = LEFT_HEAVY Then
               ' Переважала ліва галузка, тепер баланс
               ' відновлений. Це піддерево не виросло,
               ' тому дерево збалансоване.
              parent.Balance = BALANCED
              has_grown = False
        ElseIf parent.Balance = BALANCED Then
               ' Було збалансоване, тепер переважає права
               ' галузка. Піддерево усе ще збалансоване,
               ' але воно виросло, тому необхідно
    продовжити
               ' перевірку дерева.
              parent.Balance = RIGHT_HEAVY
        Else
               ' Переважала права галузка, залишилося
    незбалансоване.
               ' Виконати обертання для балансування на
    рівні
               ' цього вузла.
              RebalanceRightGrew parent
              has_grown = False
        End If       ' Закінчити перевірку балансування
    цього вузла.
    End If           ' End if для лівого піддерева else
    праве піддерево.
End Sub

Private Sub RebalanceRightGrew(parent As AVLNode)
Dim child As AVLNode
Dim grandchild As AVLNode
```

```
  Set child = parent.RightChild

  If child.Balance = RIGHT_HEAVY Then
       ' Виконати ліве обертання.
       Set parent.RightChild = child.LeftChild
       Set child.LeftChild = parent
       parent.Balance = BALANCED
       Set parent = child
  Else
       ' Виконати обертання вправо- вліво.
       Set grandchild = child.LeftChild
       Set child.LeftChild = grandchild.RightChild
       Set grandchild.RightChild = child
       Set parent.RightChild = grandchild.LeftChild
       Set grandchild.LeftChild = parent
       If grandchild.Balance = RIGHT_HEAVY Then
             parent.Balance = LEFT_HEAVY
       Else
             parent.Balance = BALANCED
       End If
       If grandchild.Balance = LEFT_HEAVY Then
             child.Balance = RIGHT_HEAVY
       Else
             child.Balance = BALANCED
       End If
       Set parent = grandchild
  End If            ' End if для правого обертання else
  подвійне праве
                    ' обертання.
  parent.Balance = BALANCED
End Sub
```

При кожному поверненні із процедури, екземпляр процедури `ReplaceRightMost` викликає підпрограму `RebalanceRightShrunk`, щоб переконатися, що дерево в цій точці збалансоване. Оскільки процедура `ReplaceRightMost` опускається по правій галузці, то вона завжди використовує для виконання балансування підпрограму `RebalanceRightShrunk`, а не `RebalanceLeftShrunk`.

При першому виклику підпрограми `ReplaceRightMost` процедура `DeleteItem` направляє її по лівій від вузла галузки, що видаляється. При поверненні з першого виклику підпрограми `ReplaceRightMost` процедура `DeleteItem` використовує підпрограму `RebalanceLeftShrunk`, щоб переконатися, що дерево збалансоване в цій точці.

Після цього один за іншим відбуваються рекурсивні повернення із процедури `DeleteItem` при проході дерева у зворотному напрямку. Так само, як і процедура `ReplaceRightmost`, процедура `DeleteItem` викликає підпрограми `RebalanceRightShrunk` або `RebalanceLeftShrunk` залежно від того, по якому шляху відбувається спуск по дереву.

Підпрограма `RebalanceLeftShrunk` аналогічна підпрограмі `RebalanceRightShrunk`, тому вона не показана в наступному коді.

Програма `AVL (диск з прикладами – папка ProgR11)` оперує АВЛ- деревом. Уведіть текст і натисніть на кнопку **Add**, щоб додати елемент до дерева. Уведіть значення і натисніть на кнопку **Remove**, щоб видалити цей елемент із дерева. На рис. 11.15. показана програма `AVL`.

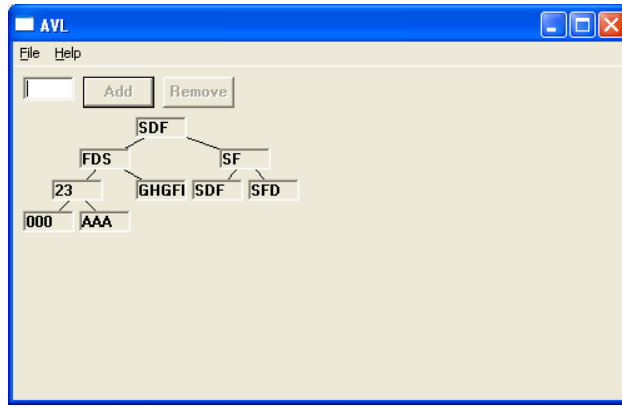


Рис. 11.15. Програма AVL

```
Public Sub DeleteItem(node As AVLNode, txt As String,
  shrunk As Boolean)
Dim child As AVLNode
Dim target As AVLNode

  If node Is Nothing Then
       Beep
       MsgBox "Елемент " & txt & " не втримується в
  дереві."
       shrunk = False
       Exit Sub
```

```
  End If

  If txt < node.Box.Caption Then
      Set child = node.LeftChild
      DeleteItem child, txt, shrunk
      Set node.LeftChild = child
      If shrunk Then RebalanceLeftShrunk node, shrunk
  ElseIf txt > node.Box.Caption Then
      Set child = node.RightChild
      DeleteItem child, txt, shrunk
      Set node.RightChild = child
      If shrunk Then RebalanceRightShrunk node, shrunk
  Else
      Set target = node
      If target.RightChild Is Nothing Then
            ' Нащадків немає або є тільки правий.
            Set node = target.LeftChild
            shrunk = True
      ElseIf target.LeftChild Is Nothing Then
            ' Є тільки правий нащадок.
            Set node = target.RightChild
            shrunk = True
      Else
            ' Є два нащадки.
            Set child = target.LeftChild
            ReplaceRightmost child, shrunk, target
            Set target.LeftChild = child
            If shrunk Then RebalanceLeftShrunk node,
  shrunk
      End If
  End If
End Sub

Private Sub ReplaceRightmost(repl As AVLNode, shrunk As
  Boolean, target As AVLNode)
Dim child As AVLNode

  If repl.RightChild Is Nothing Then
      target.Box.Caption = repl.Box.Caption
      Set target = repl
      Set repl = repl.LeftChild
      shrunk = True
  Else
      Set child = repl.RightChild
      ReplaceRightmost child, shrunk, target
      Set repl.RightChild = child
      If shrunk Then RebalanceRightShrunk repl, shrunk
  End If
```

```
End Sub

Private Sub RebalanceRightShrunk(node As AVLNode,
  shrunk As Boolean)
Dim child As AVLNode
Dim child_bal As Integer
Dim grandchild As AVLNode
Dim grandchild_bal As Integer

  If node.Balance = RIGHT_HEAVY Then
      ' Права частина переважувала, тепер баланс
  відновлений.
      node.Balance = BALANCED
  ElseIf node.Balance = BALANCED Then
      ' Було збалансовано, тепер переважає ліва
  частина.
      node.Balance = LEFT_HEAVY
      shrunk = False
  Else
      ' Ліва частина переважала, тепер не збалансована.
      Set child = node.LeftChild
      child_bal = child.Balance
      If child_bal <= 0 Then
            ' Праве обертання.
            Set node.LeftChild = child.RightChild
            Set child.RightChild = node
            If child_bal = BALANCED Then
                  node.Balance = LEFT_HEAVY
                  child.Balance = RIGHT_HEAVY
                  shrunk = False
            Else
                  node.Balance = BALANCED
                  child.Balance = BALANCED
            End If
            Set node = child
      Else
            ' Обертання вліво-вправо.
            Set grandchild = child.RightChild
            grandchild_bal = grandchild.Balance
            Set child.RightChild =
  grandchild.LeftChild
            Set grandchild.LeftChild = child
            Set node.LeftChild = grandchild.RightChild
            Set grandchild.RightChild = node
            If grandchild_bal = LEFT_HEAVY Then
                  node.Balance = RIGHT_HEAVY
            Else
                  node.Balance = BALANCED
```

```
                End If
                If grandchild_bal = RIGHT_HEAVY Then
                      child.Balance = LEFT_HEAVY
                Else
                      child.Balance = BALANCED
                End If
                Set node = grandchild
                grandchild.Balance = BALANCED
          End If
    End If
End Sub
```

Резюме

В розділі наведені визначення дерева та основні терміни, описані способи реалізації дерев в Visual Basic. Підкреслено, що способів представлення дерев багато і кожен з них надає оброблювачу ті чи інші можливості. Наприклад, формат нумерації зв'язків дозволяє швидко виконувати обхід дерева і витрачає менше пам'яті, ніж набір нащадків, але алгоритм, в такому разі складно модифікувати. Проаналізувавши операції з деревами, можна вибрати представлення, яке дозволить досягти кращого компромісу між гнучкістю і простотою використання.

Контрольні запитання та завдання

1. Що таке дерево? Чим відрізняється дерево від лінійного списку?
2. Дати означення листка, кореня, глибини дерева?
3. У чому полягає особливість бінарних дерев?
4. Як включають та виключають вузли бінарного дерева?
5. Як називається процедура, при виконанні якої кожний вузол дерева обробляється один раз деяким єдиним чином?
6. Які бувають види обходу дерев?Що таке низхідний обхід дерева?
7. Що таке АВЛ-дерева? Де вони використовуються?
8. Коли виникає необхідність балансування АВЛ-дерева?

9. Які випадки балансування АВЛ-дерев ви знаєте?
10. Яка частина АВЛ-дерева змінюється при балансуванні?
11. Створити бінарне дерево цілих чисел, визначити максимальне значення вузла дерева.
12. Створити бінарне дерево, визначити кількість вузлів на шляху від кореня до вузла, значення якого введене із клавіатури.
13. Побудувати бінарне дерево, поміняти місцями найбільший та найменший його елементи. Відобразити початкове та отримане дерево.
14. Створити бінарне дерево, підрахувати кількість його листків.
15. Побудувати бінарне дерево, створити його копію, вивести дерева.
16. Побудувати бінарне дерево, знайти в ньому елемент із заданим значенням, визначити рівень, на якому розташовано цей елемент.
17. Побудувати та вивести дерево, степінь всіх вершин якого, крім листків, дорівнює введеному натуральному числу n.
18. Скласти процедуру, яка одержує покажчик кореня бінарного дерева і створює нове бінарне дерево, що є дзеркальним відображенням першого (тобто всі ліві піддерева стають правими і навпаки).
19. Скласти процедуру, яка визначає, чи є задане бінарне дерево строго бінарним.
20. Два бінарні дерева подібні, якщо вони обидва порожні, або обидва не порожні і їх ліві піддерева подібні. Скласти процедуру, що визначає подібність двох бінарних дерев.
21. Скласти процедуру створення бінарного дерева Фібоначчі порядку n.
22. Скласти процедуру, що визначає кількість листя в бінарному дереві Фібоначчі порядку n.
23. Написати процедуру, що визначає чи є дерево Фібоначчі строго бінарним деревом.
24. Для двох далеких родичів знайти найближчого загального предка.
25. Написати процедуру, яка визначає глибину бінарного дерева Фібоначчі порядку n.

Розділ XII
ДЕРЕВА РІШЕНЬ

Багато складних реальних завдань можна змоделювати за допомогою *дерев рішень* (decision trees). Кожний вузол дерева представляє один крок рішення завдання. Кожна гілка у дереві представляє рішення, що веде до більш повного рішення. Листи являють собою остаточні рішення. Мета полягає у тому, щоб знайти «найкращий» шлях від кореня до листа при виконанні певних умов. Ці умови й значення поняття «найкращий» для шляху залежить від завдання.

Дерева рішень звичайно мають величезний розмір. Дерево рішень для гри в хрестики - нулики містить більше півмільйона вузлів. Ця гра досить проста, а реальні завдання набагато складніші. Відповідні ним дерева рішень могли б містити більше вузлів, ніж число атомів у всесвіті.

У цьому розділі обговорюються методи, які можна використовувати для пошуку в таких величезних деревах. Спочатку розглядаються *дерева гри* (`game trees`). На прикладі гри в хрестики - нулики обговорюються способи пошуку в деревах гри для знаходження найкращого можливого ходу.

Далі описуються способи пошуку в більш загальних деревах рішень. Для найменших дерев можна використовувати *метод повного перебору* (`exhaustive searching`) всіх можливих рішень. Для дерев більшого розміру можна використовувати *метод гілок та меж* (`branch - and - bound technique`), який дозволяє знайти найкраще рішення без виконання пошуку по всьому дереву.

Для дуже великих дерев потрібно використовувати *евристичний метод* або *евристику* (`heuristic` **Error! Bookmark not defined.**). При цьому отримане рішення може бути не найкращим з можливих, але воно, проте, досить близьке до найкращого. Використовуючи евристики, можна проводити пошук практично в будь-яких деревах рішень.

Наприкінці цієї глави обговорюються деякі дуже складні завдання, які ви можете спробувати вирішити за допомогою методу гілок та границь або евристичного методу. Більшість з цих завдань мають важливі застосування і знаходження для них гарних рішень конче потрібно.

Пошук у деревах гри

Стратегію настільних ігор, таких як шахи, шашки, або хрестики – нулики, можна змоделювати за допомогою *дерев гри*. Якщо в якийсь момент гри існує 30 можливих ходів, то відповідний вузол у дереві гри буде мати 30 гілок.

Наприклад, для гри в хрестики - нулики кореневий вузол відповідає початковій позиції, при якій дошка порожня. Перший гравець може помістити хрестик у кожну з дев'яти клітин дошки. Кожному із цих дев'яти можливих ходів відповідає гілка, яка виходить з кореня. Дев'ять вузлів на кінцях цих гілок відповідають дев'яти різним позиціям після першого ходу гравця.

Після того, як перший гравець зробив хід, другий може поставити нулик у кожну з восьми клітин, що залишилися. Кожному із цих ходів відповідає гілка, що виходить із вузла, який відповідає поточній позиції гри. На рис.12.1 показано невеликий фрагмент дерева гри в хрестики - нулики.

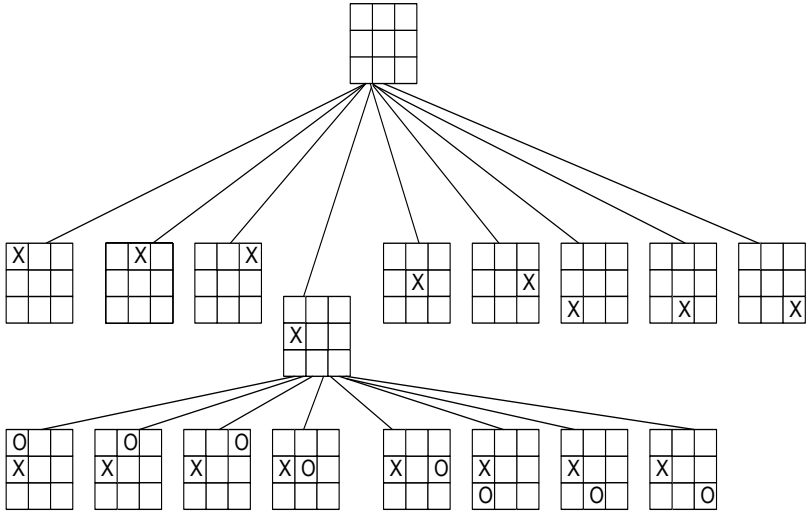


Рис.12.1. Фрагмент дерева гри в хрестики - нулики

Як можна побачити на рис. 12.1, дерево гри в хрестики - нулики росте дуже швидко. Якщо воно продовжить рости таким чином, що кожний наступний вузол у дереві буде мати на одну гілку менше, ніж його батько, то дерево цілком буде мати 9 * 8 * 7 ... * 1 =

362880 листів. У дереві буде 362880 можливих шляхів, що відповідають 362800 можливим іграм.

У дійсності багато які з вузлів дерева будуть відсутні, тому що відповідні їм ходи заборонені правилами гри. Якщо гравець, що ходив першим, за три своїх ходи поставить хрестики у верхній лівій, верхній середній і верхній правій клітинах, то він виграє й гра закінчиться. Вузол, що відповідає цій позиції, не буде мати нащадків, тому що гра завершиться на цьому кроці. Ця гра показана на рис.12.2.

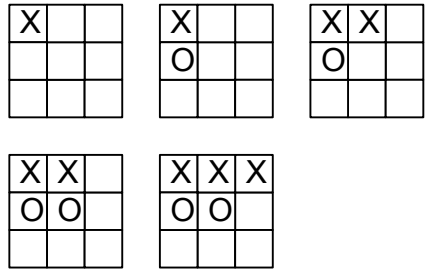


Рис.12.2. Швидке завершення гри

Після видалення всіх неможливих вузлів у дереві залишається біля чверті мільйона листів. Це усе ще дуже велике дерево, і пошук його методом повного перебору займає досить багато часу. Для більш складних ігор, таких як шашки або шахи дерева гри мають величезний розмір. Якби під час кожного ходу в шахах гравець мав 16 можливих варіантів, то дерево гри мало б більше трильйона вузлів після п'яти ходів кожного із гравців. Наприкінці цього розділу обговорюється пошук у таких величезних деревах гри.

Мінімаксний перебір

Для виконання пошуку в дереві гри потрібно мати можливість визначення *ваги* позиції на дошці. Для гри в хрестики - нулики для першого гравця більшу вагу мають позиції, у яких три хрестики розташовані в ряд, тому що при цьому перший гравець виграє. Вага тих же позицій для другого гравця мала, тому що в цьому випадку він програє.

Кожний гравець може надати позиції одну із чотирьох ваг. Якщо вага позиції дорівнює 4 - це значить, що гравець у цій позиції виграє. Якщо вага дорівнює 3, то з поточного положення на дошці незрозуміло, хто із гравців виграє. Вага позиції зі значенням 2

приводить до нічиєї. І, нарешті, вага, зі значенням 1 означає, що виграє супротивник.

Для обробки дерева методом повного перебору можна використовувати мінімаксну (`minimax`) стратегію, в якій робиться спроба *мінімізувати максимальну* вагу, яку може мати позиція для супротивника після наступного ходу. Це можна зробити, визначивши максимально можливу вагу позиції для супротивника після кожного зі своїх можливих ходів і потім, вибравши хід, що дає позицію з мінімальною вагою для супротивника.

Підпрограма `BoardValue`, наведена нижче, обчислює вагу позиції на дошці, перевіряючи всі можливі ходи. Для кожного ходу вона рекурсивно викликає себе, щоб знайти вагу, яку буде мати нова позиція для супротивника. Потім вона вибирає хід, при якому вага отриманої позиції для супротивника буде найменшою.

Для визначення ваги позиції на дошці процедура `BoardValue` рекурсивно викликає себе доти, доки не відбудеться одна із трьох подій. *По-перше,* вона може дійти до позиції, у якій гравець виграє. У цьому випадку процедура надасть позиції вагу 4, що вказує на виграш гравця, що зробив останній хід.

По - друге, процедура `BoardValue` може знайти позицію, у якій жоден із гравців не може зробити наступний хід. Гра при цьому закінчується нічиєю, тому процедура надає цій позиції вагу 2.

І нарешті, процедура може досягти заданої максимальної глибини рекурсії. У цьому випадку процедура `BoardValue` надає позиції вагу 3, що вказує, що вона не може визначити переможця. Завдання максимальної глибини рекурсії обмежує час пошуку в дереві гри. Це особливо важливо для більш складних ігор, таких як шахи, де пошук у дереві гри може тривати практично вічно. Максимальна глибина пошуку також може задавати рівень майстерності програми. Чим далі вперед програма зможе аналізувати ходи, тим краще вона буде грати.

На рис.12.3 показано дерево гри в хрестики - нулики наприкінці партії. Ходить гравець, що грає хрестиками, і в нього є три можливих ходи. Щоб вибрати найкращий хід, процедура `BoardValue` рекурсивно перевіряє кожен із трьох можливих ходів. Перший і третій можливі ходи (ліва й права галузі дерева) приводять до виграшу супротивника, тому їхня вага для супротивника дорівнює 4. Другий можливий хід приводить до нічиєї, і його вага для супротивника дорівнює 2. Процедура `BoardValue` вибирає цей хід, тому що він має найменшу вагу для супротивника.

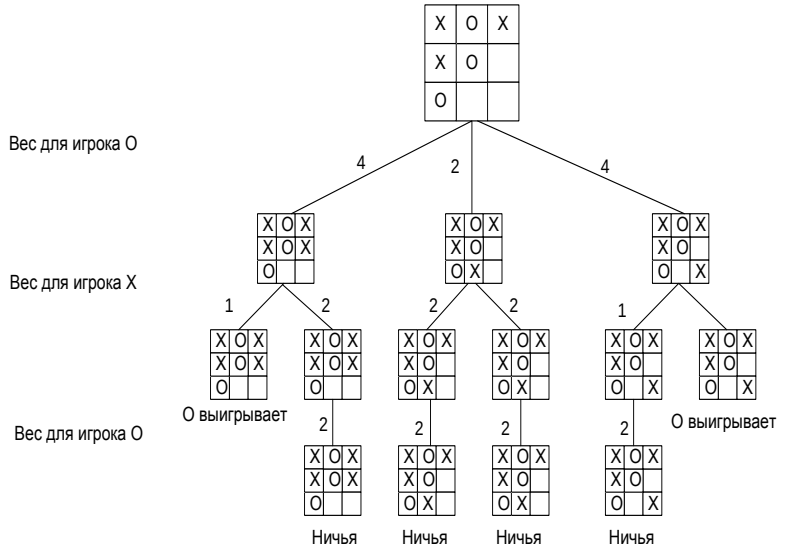


Рис.12.3. Дерево гри в хрестики - нулики наприкінці партії

```
Private Sub BoardValue(best_move As Integer, best_value
  As Integer, pl1 As Integer, pl2 As Integer, Depth As
  Integer)
Dim pl As Integer
Dim i As Integer
Dim good_i As Integer
Dim good_value As Integer
Dim enemy_i As Integer
Dim enemy_value As Integer

  DoEvents          ' Не займати 100% процесорного часу.

  ' Якщо глибина рекурсії занадто велика, результат
  невідомий.
  If Depth >= SkillLevel Then
      best_value = VALUE_UNKNOWN
      Exit Sub
  End If

  ' Якщо гра завершується, то результат відомий.
  pl = Winner()
  If pl <> PLAYER_NONE Then
      ' Перетворити вагу для переможця pl у вагу для
  гравця pl1.
      If pl = pl1 Then
```

```
            best_value = VALUE_WIN
        ElseIf pl = pl2 Then
            best_value = VALUE_LOSE
        Else
            best_value = VALUE_DRAW
        End If
        Exit Sub
    End If

    ' Перевірити всі припустимі ходи.
    good_i = -1
    good_value = VALUE_HIGH
    For i = 1 To NUM_SQUARES
        ' Перевірити хід, якщо він дозволений правилами.
        If Board(i) = PLAYER_NONE Then
            ' Знайти вагу отриманого положення для
    супротивника.
            If ShowTrials Then _
                MoveLabel.Caption = _
                    MoveLabel.Caption &
    Format$(i)
            ' Зробити хід.
            Board(i) = pl1
            BoardValue enemy_i, enemy_value, pl2, pl1,
    Depth + 1
            ' Скасувати хід.
            Board(i) = PLAYER_NONE
            If ShowTrials Then _
                MoveLabel.Caption = _
                    Left$(MoveLabel.Caption,
    Depth)

            ' чи Менше ця вага, ніж попередня.
            If enemy_value < good_value Then
                good_i = i
                good_value = enemy_value
                ' Якщо ми виграємо, то кращого
    рішення немає,
                ' тому вибирається цей хід.
                If good_value <= VALUE_LOSE Then
    Exit For
            End If
        End If       ' End if Board(i) = PLAYER_NONE ...
    Next i

    ' Перетворити вагу позиції для супротивника у вагу
    для гравця.
    If good_value = VALUE_WIN Then
```

```
        ' Супротивник виграє, ми програли.
        best_value = VALUE_LOSE
    ElseIf enemy_value = VALUE_LOSE Then
        ' Супротивник програв, ми виграли.
        best_value = VALUE_WIN
    Else
        ' Вага нічиєї або невизначеної позиції
        ' однакова для обох гравців.
        best_value = good_value
    End If
    best_move = good_i
End Sub
```

Програма TicTac (диск з прикладами – папка ProgR12) ~~на диску із прикладами~~ використовує процедуру BoardValue. Основна частина коду програми забезпечує взаємодію з користувачем, малює дошку, дозволяє користувачеві вибирати хід, задавати опції й так далі.

Якщо не обрана команда **Show Test Moves** (Показувати ходи, що перевіряються) з меню **Options** (Опції), то продуктивність програми буде набагато вищою. Якщо обрано цю опцію, то програма виводить кожний хід, що аналізується. Постійне відновлення екрана займає набагато більше часу, ніж дійсний пошук у дереві.

Інші команди в меню **Options** дозволяють вибрати рівень майстерності програми (максимальну глибину рекурсії) і вибрати гру хрестиками або нуликами. При високому рівні майстерності перший хід займає набагато більше часу.

Оптимізація пошуку в деревах рішень

Якби для пошуку в дереві гри ми розподіляли б тільки мінімаксну стратегію, то виконання пошуку у великих деревах було б дуже складним. Такі ігри, як шахи, настільки складні, що програма може провести пошук лише на декількох рівнях дерева. На щастя, існують кілька прийомів, які можна використовувати для пошуку в великих деревах гри.

Попереднє обчислення початкових ходів

По-перше, у програмі можуть бути записані початкові ходи, обрані експертами. Можна вирішити, що програма гри в хрестики - нулики має робити перший хід у центральну клітину. Це визначає першу гілку дерева гри, тому програма може ігнорувати всі шляхи, що не проходять через першу гілку. Це зменшує дерево гри в хрестики - нулики в 9 разів.

Фактично, програмі не потрібно виконувати пошук у дереві до того, доки супротивник не зробить свій хід. У цей момент і комп'ютер і супротивник вибрали кожний свою гілку, тому дерево, що залишилося, стане набагато меншим, і буде містити менше ніж 7! = 5040 шляхів. Прорахувавши заздалегідь один хід, можна зменшити розмір дерева гри від чверті мільйона до менш ніж 5040 шляхів.

Аналогічно можна записати відповіді на перші ходи, якщо супротивник ходить першим. Є дев'ять варіантів першого ходу, отже потрібно записати дев'ять відповідних ходів. При цьому програмі не потрібно вести пошук по дереву, доки супротивник не зробить два ходи, а комп'ютер - один. Тоді дерево гри міститиме менш ніж 6! = 720 шляхів. Записано лише дев'ять ходів, а розмір дерева при цьому зменшується дуже пімітно. Це ще один приклад просторово-тимчасового компромісу. Використання більшої кількості пам'яті зменшує час, необхідний для пошуку в дереві гри.

Програма `TicTac2` ~~на диску із прикладами~~ (диск з прикладами – папка ProgR12) використовує 10 записаних ходів. Задайте 9-й рівень майстерності і нехай програма робить перший хід. Потім задайте ті ж опції в програмі `TicTac`. Ви побачите величезну різницю у швидкості роботи цих двох програм.

Комерційні програми гри в шахи також починають із записаних ходів і відповідей, рекомендованих гросмейстерами. Такі програми можуть робити перші ходи дуже швидко. Після того як програма вичерпає всі записані заздалегідь ходи, вона почне робити ходи набагато повільніше.

Визначення важливих позицій

Інший спосіб поліпшення пошуку в дереві гри полягає в тому, щоб визначати важливі позиції. Якщо програма розпізнає одну із цих позицій, вона може виконати певні дії або змінити спосіб пошуку в дереві гри.

Під час гри в шахи гравці часто розташовують фігури так, щоб вони захищали інші фігури. Якщо супротивник бере фігуру, то гравець бере фігуру супротивника замість. Часто таке узяття дозволяє супротивникові у свою чергу взяти іншу фігуру, що приводить до серії обмінів.

Деякі програми знаходять можливі послідовності обмінів. Якщо програма розпізнає можливість обміну, вона тимчасово змінює максимальну глибину, на яку переглядає дерево, щоб простежити до кінця ланцюжок обмінів. Це дозволяє програмі вирішити, чи варто йти на обмін. Після обміну фігур їхня кількість також зменшується, тому пошук у дереві гри стає в майбутньому простішим.

Деякі шахові програми також відслідковують рокіровки, ходи, при яких під загрозою виявляється відразу кілька фігур, шах або напад на ферзя й так далі.

Евристики

В іграх, більш складних, ніж хрестики - нулики, практично неможливо провести пошук навіть у невеликому фрагменті дерева гри. У цих випадках можна використовувати різні *евристики*.

Евристикою називається алгоритм або емпіричне правило, що імовірно, але не обов'язково дасть гарний результат.

Наприклад, у шахах звичайною евристикою є «посилення переваги». Якщо в супротивника менше сильних фігур і однакове число інших, то варто йти на розмін при кожній можливості. Наприклад, якщо ви берете коня супротивника, втрачаючи при цьому свого, то такий обмін варто виконати. Зменшення числа фігур, що залишилися, робить дерево рішень коротшим й може збільшити відносну перевагу. Ця стратегія не гарантує виграшу, але підвищує його ймовірність.

Інша часто використовувана евристика полягає в присвоєнні різних ваг різним частинам дошки. У шахах вага клітин у центрі дошки вище, оскільки фігури, які перебувають на цих позиціях, можуть атакувати більшу частину дошки. Коли процедура `BoardValue` обчислює вагу поточної позиції на дошці, вона може надавати більшої ваги фігурам, які займають клітини в центрі дошки.

Пошук нестандартних рішень

Деякі методи пошуку в деревах гри незастосовні до узагальнених дерев рішень. Багато цих дерев не включають почергових ходів гравців, тому мінімаксний метод і обчислені заздалегідь ходи в цьому випадку безглузді. У наступних підрозділах описані методи, які можна використовувати для пошуку в деревах рішень такого типу.

Гілки та межі

Метод гілок та меж (`branch and bound`) є одним з методів *відсікання* (`pruning`) гілок у дереві рішень, щоб не розглядати всі гілки дерева. Загальний підхід при цьому полягає в тому, щоб відслідковувати межі вже виявлених і можливих рішень. Якщо в якійсь точці найкраще із вже знайдених рішень ефективніше, ніж найкраще можливе рішення на нижніх гілках, то можна ігнорувати всі шляхи вниз від вузла.

Наприклад, припустимо, що є 100 мільйонів доларів, які потрібно вкласти в кілька можливих інвестицій. Кожне із цих вкладень має різну вартість і дає різний прибуток. Необхідно вирішити, як вкласти гроші щонайкраще, щоб сумарний прибуток був максимальним.

Завдання такого типу називаються завданням *формування портфеля* (`knapsack problem`). У вас є кілька позицій (інвестицій), які мають вміститися в портфель фіксованого розміру (100 мільйонів доларів). Кожна з позицій має вартість (гроші) і ціну (теж гроші). Необхідно знайти набір позицій, що вмістяться в портфель і будуть мати максимально можливу ціну.

Це завдання можна змоделювати за допомогою дерева рішень. Кожний вузол дерева відповідає певній комбінації позицій у портфелі. Кожна гілка відповідає ухваленню рішення про те, щоб видалити позицію з портфеля або додати її в нього. Ліва гілка першого вузла відповідає першому вкладенню. На рис.12.4 показане дерево рішень для чотирьох можливих інвестицій.

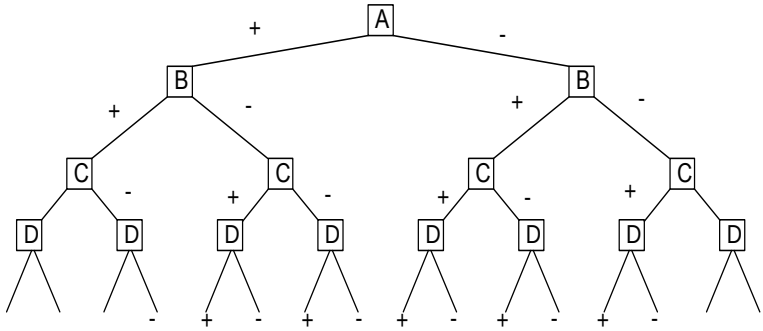


Рис.12.4. Дерево рішень для можливих інвестицій

Дерево рішень для цього завдання являє собою повне двійкове дерево, глибина якого дорівнює числу інвестицій. Кожна вершина відповідає повному набору інвестицій.

Розмір цього дерева дуже швидко росте зі збільшенням числа інвестицій. Для 10 можливих інвестицій, у дереві перебуватиме $2^{10} = 1024$ листа. Для 20 інвестицій у дереві буде вже більше мільйона листів. Можна провести повний пошук по такому дереву, але при подальшому збільшенні числа можливих інвестицій розмір дерева стане дуже великим.

Щоб використовувати метод гілок та меж, створимо масив, що буде містити позиції з найкращого знайденого дотепер рішення. При ініціалізації масив має бути порожній. Можна також використовувати змінну для відстеження ціни цього рішення. Спочатку ця змінна може мати невелике значення, щоб перше ж знайдене реальне рішення було краще вхідного.

При пошуку в дереві рішень, якщо в якійсь точці аналізоване рішення не може бути кращим, ніж існуюче, то можна припинити подальший пошук цим шляхом. Також, якщо в якійсь точці обрані позиції коштують більше 100 мільйонів, то можна також припинити пошук.

Як конкретний приклад припустимо, що є інвестиції, наведені в табл. 8.1. На рис.12.4 показане відповідне дерево рішень. Деякі із цих інвестиційних пакетів порушують граничні умови завдання. Наприклад, найлівіший шлях привів би до вкладення 178 мільйонів доларів в усі чотири можливих інвестиції.

Таблиця 12.1

Можливі інвестиції

Інвестиція	Вартість (млн)	Прибуток (млн)
A	45	10
B	52	13
C	46	8
D	35	4

Припустимо, що ми почали пошук у дереві, зображеному на рис.12.4, і виявили, що можна витратити 97 мільйонів доларів на позиції A і B, одержавши 23 мільйони прибутку. Це відповідає четвертому листу ліворуч на рис.12.4.

При продовженні пошуку в дереві можна дійти до другого ліворуч вузла B на рис.12.4. Це відповідає інвестиційному пакету, якмй включає позицію A, не включає позицію B і може включати або не включати позиції C і D. У цій точці пакет уже коштує 45 мільйонів доларів завдяки позиції A і приносить 10 мільйонів прибутку.

Позиції, що залишилися, C і D разом узяті можуть підвищити прибуток ще на 12 мільйонів. Поточне рішення приносить 10 мільйонів прибутку, тому найкраще можливе рішення нижче цього вузла принесе не більше 11 мільйонів прибутку. Це менше, ніж дохід в 23 мільйони для вже знайденого рішення, тому нема рації продовжувати пошук униз цим шляхом.

У міру просування програми по дереву їй не потрібно постійно перевіряти, чи буде часткове рішення, яке вона розглядає, краще, ніж найкраще знайдене дотепер рішення. Якщо часткове рішення краще, то краще буде й найправіший вузол унизу від цього часткового рішення. Цей вузол представляє той самий набір позицій, як і часткове рішення, тому що всі інші позиції при цьому виключені. Це означає, що програмі необхідно шукати краще рішення тільки тоді, коли вона досягає листа.

Фактично будь-який лист, до якого доходить програма *завжди* є кращим рішенням. Якби це було не так, то гілка, на якій перебуває цей лист, була б відсічена, коли програма розглядала батьківський вузол. У цій точці переміщення до листа зменшить ціну не вибраних позицій до нуля. Якщо ціна рішення не більше, ніж найкраще знайдене дотепер рішення, то перевірка нижньої границі зупинить просування програми до листа. Використовуючи цей факт, програма може обновляти найкраще рішення при досягненні листа.

Наступний код використовує перевірку верхньої й нижньої межі для реалізації алгоритму гілок і меж:

```
' Повний нерозподілений прибуток.
Private unassigned_profit As Integer

Public NumItems As Integer
Public MaxItem As Integer

Global Const OPTION_EXHAUSTIVE_SEARCH = 0
Global Const OPTION_BRANCH_AND_BOUND = 1

Type Item
  Cost As Integer
  Profit As Integer
End Type

Global Items() As Item
Global NodesVisited As Long
Global ToSpend As Integer
Global best_cost As Integer
Global best_profit As Integer

' Дорівнює True для позицій у поточному найкращому
  рішенні.
Public best_solution() As Boolean

' Рішення, що ми перевіряємо.
Private test_solution() As Boolean
Private test_cost As Integer
Private test_profit As Integer

' Ініціалізація змінних і початок пошуку.
Public Sub Search(search_type As Integer)
Dim i As Integer

  ' Завдання розміру масивів рішення.
  ReDim best_solution(0 To MaxItem)
  ReDim test_solution(0 To MaxItem)

  ' Ініціалізація - порожній список інвестицій.
  NodesVisited = 0
```

```
  best_profit = 0
  best_cost = 0
  unassigned_profit = 0
  For i = 0 To MaxItem
      unassigned_profit = unassigned_profit +
  Items(i).Profit
  Next i
  test_profit = 0
  test_cost = 0

  ' Почнемо пошук з першої позиції.
  BranchAndBound 0
End Sub

' Виконати пошук методом галузок і границь починаючи із
  цієї позиції.
Public Sub BranchAndBound(item_num As Integer)
Dim i As Integer

  NodesVisited = NodesVisited + 1

  ' Якщо це аркуш, то це краще рішення, ніж
  ' те, що ми мали раніше, інакше він був би
  ' відсічений під час пошуку раніше.
  If item_num > MaxItem Then
      For i = 0 To MaxItem
            best_solution(i) = test_solution(i)
            best_profit = test_profit
            best_cost = test_cost
      Next i
      Exit Sub
  End If

  ' Інакше перейти по галузі униз по галузях нащадка.
  ' Спочатку спробувати додати цю позицію.
  Переконатися,
  ' що вона не перевищує обмеження за ціною.
  If test_cost + Items(item_num).Cost <= ToSpend Then
      ' Додати позицію до тестового рішення.
      test_solution(item_num) = True
```

```
        test_cost = test_cost + Items(item_num).Cost
        test_profit = test_profit +
    Items(item_num).Profit
        unassigned_profit = unassigned_profit -
    Items(item_num).Profit

        ' Рекурсивна перевірка можливого результату.
        BranchAndBound item_num + 1

        ' Видалити позицію з тестового рішення.
        test_solution(item_num) = False
        test_cost = test_cost - Items(item_num).Cost
        test_profit = test_profit -
    Items(item_num).Profit
        unassigned_profit = unassigned_profit +
    Items(item_num).Profit
    End If

    ' Спробувати виключити позицію. З'ясувати, чи
    принесуть
    ' позиції, що залишилися, достатній дохід, щоб
    ' шлях униз по цій галузі перевищив нижню межу.
    unassigned_profit = unassigned_profit -
    Items(item_num).Profit
    If test_profit + unassigned_profit > best_profit Then
    BranchAndBound item_num + 1
    unassigned_profit = unassigned_profit +
    Items(item_num).Profit
End Sub
```

Програма BandB ~~на диску із прикладами~~ (диск з прикладами – папка ProgR12) використовує метод повного перебору й метод гілок та меж для вирішення завдання про формування портфеля. Уведіть максимальну й мінімальну вартість і ціну, які ви хочете надати позиціям, а також число позицій, що потрібно створити. Потім натисніть на кнопку **Randomize** (Рандомізувати), щоб створити список позицій.

Потім за допомогою перемикача знизу форми виберіть або **Exhaustive Search** (Повний перебір), або **Branch and Bound** (Метод галузей і границь). Коли ви натиснете на кнопку **Go** (Почати), то програма знайде найкраще рішення за допомогою обраного методу. Потім вона виведе на екран це рішення, а також число вузлів у

повному дереві рішень і число вузлів, які програма в дійсності перевірила. На рис.12.5. показане вікно програми ~~BindB~~ після рішення завдання формування портфеля для 20 позицій. Перед тим, як виконати повний перебір для 20 позицій, спробуйте спочатку запустити приклади меншого розміру.

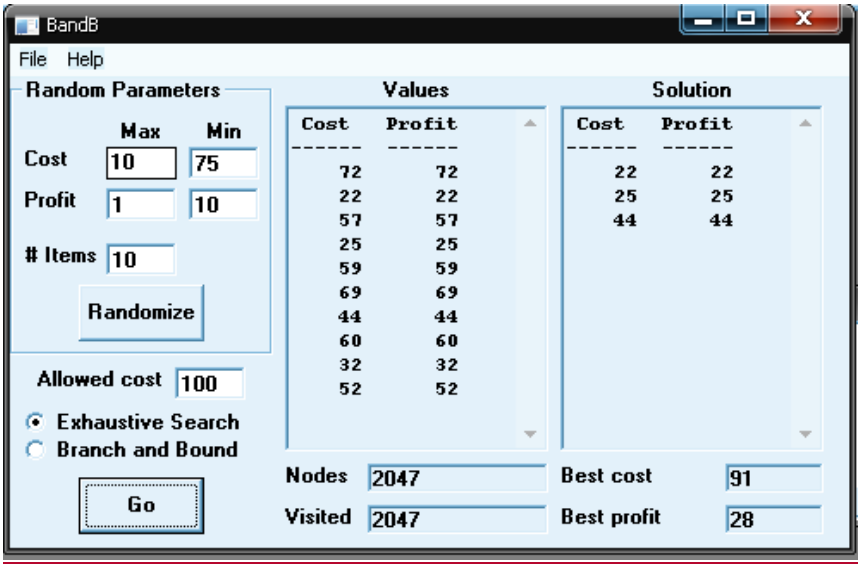


Рис.12.5. Вікно програми BandB

При пошуку методом гілок та меж число вузлів, що перевіряються, набагато менше, ніж при повному переборі. Дерево рішень для завдання портфеля з 20 позиціями містить 2.097.151 вузол. При повному переборі доведеться перевірити їх всі, при пошуку методом гілок та меж знадобиться перевірити тільки приблизно 1.500 з них.

Число вузлів, які перевіряє програма при використанні методу гілок та меж, залежить від точних значень даних. Якщо ціна позицій висока, то в правильне рішення буде входити небагато елементів. Після занесення декількох позицій у пробне рішення, позиції, що залишилися, занадто дорого коштують, щоб потрапити в портфелі, тому більша частина дерева буде відсічена.

З іншого боку, якщо елементи мають низьку вартість, то в правильне рішення увійде велике їхнє число, тому програмі доведеться досліджувати безліч комбінацій. У табл. 12.2 наведено число вузлів, перевірене програмою Bind у серії тестів за різної

вартості позицій. Програма створювала 20 випадкових позицій, і повна вартість рішення дорівнює 100.

Таблиця 12.2

Результати роботи програми `Bind` у серії тестів~~Таблиця 8.2.~~

~~Число вузлів, перевірених в методах повного перебору й галузей і границь~~**Середня вартість позиції**	**Повний перебір**	**Метод гілок та меж**
60	2.097.151	203
50	2.097.151	520
40	2.097.151	1.322
30	2.097.151	4.269
20	2.097.151	13.286
10	2.097.151	40.589
60	2.097.151	203
50	2.097.151	520

~~0~~	~~2.097.151~~	~~203~~
~~0~~	~~2.097.151~~	~~520~~
~~0~~	~~2.097.151~~	~~1.322~~
~~0~~	~~2.097.151~~	~~4.269~~
~~0~~	~~2.097.151~~	~~13.286~~
~~0~~	~~2.097.151~~	~~40.589~~

Евристика

Іноді навіть алгоритм гілок та меж не може провести повний пошук у дереві. Дерево рішень для завдання портфеля з 65 позиціями містить більше $7 * 10^{19}$ вузлів. Якщо алгоритм гілок та меж перевіряє

тільки одну десяту відсотка цих вузлів і якщо комп'ютер перевіряє мільйон вузлів у секунду, то для рішення цього завдання треба було б більше 2 мільйонів років. У завданнях, для яких алгоритм гілок та меж виконується надто повільно, можна використовувати евристичний підхід.

Якщо якість рішення не є важливою, то прийнятним може бути результат, отриманий за допомогою евристики. У деяких випадках точність вхідних даних може бути недостатньою. Тоді гарне евристичне рішення може бути таким же правильним, як і теоретичне «найкраще» рішення.

У попередньому прикладі метод гілок та меж використовувався для вибору інвестиційних можливостей. Проте вкладення можуть бути ризикованими, і точні результати часто заздалегідь невідомі. Може бути, що заздалегідь буде невідомим точний дохід або навіть вартість деяких інвестицій. У цьому випадку ефективне евристичне рішення може бути таким же надійним, як і найкраще рішення, яке ви може обчислити точно.

У цьому підрозділі обговорюються евристики, які корисні при рішенні багатьох складних завдань.. Програма `Heur` ~~на диску із прикладами~~ `(диск з прикладами – папка ProgR12)` демонструє кожну з евристик. Вона також дозволяє порівнювати результати, отримані за допомогою евристик і методів повного перебору й гілок та меж. Введіть значення мінімальної й максимальної вартості й доходу, а також число позицій і повну вартість портфеля у відповідних полях області **Parameters** (Параметри), щоб задати параметри створюваних даних. Потім виберіть алгоритм, який ви хочете протестувати, і натисніть на кнопку **Go**. Програма виведе повну вартість і дохід для найкращого рішення, знайденого за допомогою обраного алгоритму. Вона також сортує рішення за максимальним отриманим доходом й виводить час виконання для кожного з алгоритмів. Використовуйте метод гілок та меж тільки для невеликих завдань, а метод повного перебору тільки для завдань ще меншого обсягу.

На рис.12.6 показане вікно програми ~~`Heur`~~ після рішення завдання формування портфеля для 20 позицій. Евристики `Fixed1`, `Fixed2` і `No Changes 1`, які будуть незабаром описані, дали найкращі евристичні рішення. Помітьте, що ці рішення небагато гірші, ніж точні рішення, отримані при використанні методу гілок та меж.

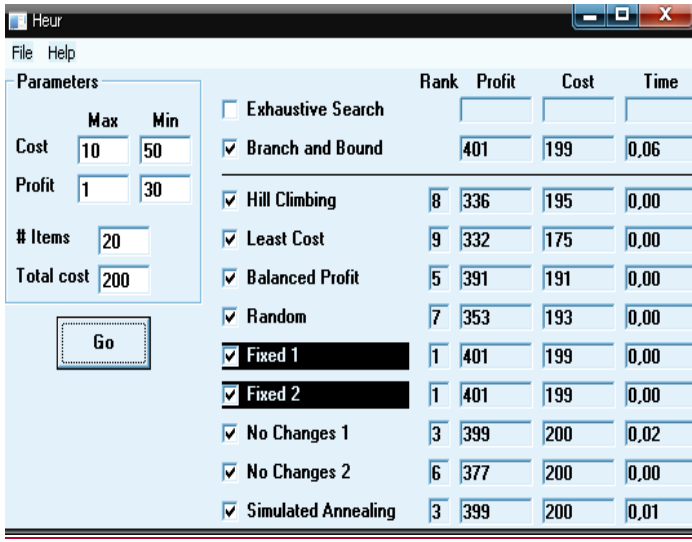


Рис.12.6. Вікно програми `Heur`

Сходження на пагорб

Евристика *сходження на пагорб* (`hill - climbing`) вносить зміни в поточне рішення, щоб максимально наблизити його до мети. Цей процес називається сходженням на пагорб, тому що він схожий на те, як мандрівник, що заблудився, намагається вночі добратися до вершини гори. Навіть якщо вже досить темно, щоб можна було щось розглянути вдалині, мандрівник може спробувати добратися до вершини гори, постійно рухаючись угору.

Звичайно, існує ймовірність, що мандрівник застрягне на вершині меншого пагорба й не добереться до піка. Ця проблема завжди може виникати при використанні цієї евристики. Алгоритм може знайти рішення і воно може виявитися локально прийнятним, але це не обов'язково найкраще можливе рішення.

У задачі про формування портфеля мета полягає у тому, щоб підібрати набір позицій, повна вартість яких не перевищує заданої межі, а загальна ціна максимальна. На кожному кроці евристика сходження на пагорб буде вибирати позицію, що приносить найбільший прибуток. При цьому рішення буде усе краще відповідати меті - одержанню максимального прибутку.

Спочатку програма додає до рішення позицію з максимальним прибутком. Потім вона додає наступну позицію з максимальним прибутком, якщо при цьому повна ціна ще залишається в припустимих межах. Вона продовжує додавати позиції з максимальним прибутком доти, доки не залишиться позицій, що задовольняють умовам.

Для списку інвестицій з табл. 12.3 програма спочатку вибирає позицію A, тому що вона дає максимальний прибуток - 9 мільйонів доларів. Потім програма вибирає наступну позицію C, що дає прибуток 8 мільйонів. У цей момент витрачені вже 93 мільйони з 100, і програма не може одержати більше позицій. Рішення, отримане за допомогою евристики, включає позиції A і C, має вартість 93 мільйони, і приносить 17 мільйонів прибутку.

Евристика сходження на пагорб заповнює портфель дуже швидко. Якщо позиції початково були відсортовані в порядку убування принесеного прибутку, то складність цього алгоритму порядку O(N). Програма просто переміщається за списком, додаючи кожну позицію, якщо під неї є місце. Навіть якщо список не впорядкований, то це алгоритм зі складністю порядку $O(N^2)$. Це набагато краще, ніж $O(2^N)$ кроків, які потрібні для повного перебору всіх вузлів у дереві. Для 20 позицій ця евристика вимагає лише близько 400 кроків, метод гілок та меж - кілька тисяч, а повний перебір - більш ніж 2 мільйони.

Таблиця 12.3

Можливі інвестиції

Інвестиція	**Вартість**	**Віддача**	**Прибуток**
A	63	72	9
B	35	42	7
C	30	38	8
D	27	34	7
E	23	26	3

```
Public Sub HillClimbing()
Dim i As Integer
Dim j As Integer
Dim big_value As Integer
Dim big_j As Integer

  ' Багаторазовий обхід списку й пошук наступної
  ' позиції, що приносить найбільший прибуток,
  ' вартість якої не перевищує верхньої границі.
  For i = 1 To NumItems
      big_value = 0
```

```
        big_j = -1
        For j = 1 To NumItems
              ' Перевірити, чи не перебуває він уже
              ' у рішенні.
              If (Not test_solution(j)) And _
                    (test_cost + Items(j).Cost <=
  ToSpend) And _
                    (big_value < Items(j).Profit)
              Then
                    big_value = Items(j).Profit
                    big_j = j
              End If
        Next j

        ' Зупинитися, якщо не знайдена позиція,
        ' що задовольняє умовам.
        If big_j < 0 Then Exit For

        test_cost = test_cost + Items(big_j).Cost
        test_solution(big_j) = True
        test_profit = test_profit + Items(big_j).Profit
  Next i
End Sub
```

Метод найменшої вартості

Стратегія, що у якомусь змісті протилежна стратегії сходження на пагорб, називається стратегією *найменшої вартості* (`least - cost`). Замість того, щоб на кожному кроці намагатися максимально наблизити рішення до мети, можна спробувати зменшити *вартість* рішення, наскільки це можливо. У прикладі з формуванням портфеля на кожному кроці до рішення додається позиція з мінімальною вартістю.

Ця стратегія намагається помістити в рішення максимально можливе число позицій. Це буде непоганим рішенням, якщо всі позиції мають приблизно однакову вартість. Якщо дорога позиція приносить більший прибуток, то ця стратегія може упустити цю можливість, даючи не кращий з можливих результатів.

Для інвестицій, показаних у табл. 12.3, алгоритм найменшої вартості починає з додавання до рішення позиції E з вартістю 23 мільйони доларів. Потім він вибирає позицію D, що коштує 27 мільйонів, і потім позицію C з вартістю 30 мільйонів. У цій точці

алгоритм уже витратив 80 мільйонів з 100 можливих, тому більше він не може вибрати ні однієї позиції.

Це рішення має вартість 80 мільйонів і дає 18 мільйонів прибутку. Це на мільйон краще, ніж рішення для евристики сходження на пагорб, але стратегія найменшої вартості не завжди дає краще рішення, ніж сходження на пагорб. Яка з евристик дає кращі результати, залежить від значень вхідних даних.

Структура програми, що реалізує евристику найменшої вартості, майже ідентична структурі програми для евристики сходження на пагорб. Єдине розходження між ними полягає у виборі наступної позиції для додавання до рішення. Евристика найменшої вартості вибирає позицію з мінімальною ціною; метод сходження на пагорб вибирає позицію з максимальним прибутком. Оскільки ці два методи дуже схожі, вони виконуються за однаковий час. Якщо позиції впорядковані відповідним чином, то обидва алгоритми виконуються за час порядку O(N). Якщо позиції розташовані випадковим чином, то вони виконуються за час порядку $O(N^2)$.

Оскільки код мовою Visual Basic для цих двох евристик дуже схожий, то ми приводимо тільки рядки, у яких відбувається вибір чергової позиції.

```
If (Not test_solution(j)) And _
   (test_cost + Items(j).Cost <= ToSpend) And _
   (small_cost > Items(j).Cost)
Then
   small_cost = Items(j).Cost
   small_j = j
End If
```

Збалансований прибуток

Стратегія сходження на пагорб не враховує вартість позицій, що додаються. Вона вибирає позиції з максимальним прибутком, навіть якщо їхня вартість велика. Стратегія найменшої вартості не враховує принесений позицією прибуток. Вона вибирає позиції з низькою вартістю, навіть якщо вони приносять мало прибутку.

Евристика *збалансованого прибутку* (`balanced profit`) порівнює при виборі вартість позицій і принесений ними прибуток. На кожному кроці евристика вибирає позицію з найбільшим відношенням прибуток - вартість.

У табл. 12.4 наведені ті ж дані, що й у табл. 12.3, але в ній доданий ще один стовпчик з відношенням прибуток - вартість. При цьому підході спочатку вибирається позиція С, тому що вона має максимальне співвідношення прибуток – вартість - 0,27. Потім до рішення додається позиція D з відношенням 0,26 і позиція В з відношенням 0,20. У цій точці буде витрачено 92 мільйони з 100 можливих, і в рішення не можна буде додати більше ні однієї позиції.

Рішення буде мати вартість 92 мільйони й давати 22 мільйони прибутку. Це на 4 мільйони краще, ніж рішення з найменшою вартістю й на 5 мільйонів краще, ніж рішення методом сходження на пагорб. У цьому випадку це буде також найкращим можливим рішенням і його також можна знайти повним перебором або методом гілок та меж. Метод збалансованого прибутку, проте, є евристичним, тому він не обов'язково знаходить найкраще можливе рішення. Він часто знаходить краще рішення, ніж методи найменшої вартості й сходження на пагорб, але це не обов'язково так.

Таблиця 12.4

Можливі інвестиції зі співвідношенням прибуток/вартість

Інвестиція	Вартість	Віддача	Прибуток	Прибуток/ Вартість
A	63	72	9	0.14
B	35	42	7	0.2
C	30	38	8	0.27
D	27	34	7	0.26
E	23	26	3	0.13

Структура програми, що реалізує евристику збалансованого прибутку, майже ідентична структурі програм для сходження на пагорб і найменшої вартості. Єдина відмінність полягає в методі вибору позиції, що додається до рішення:

```
If (Not test_solution(j)) And _
   (test_cost + Items(j).Cost <= ToSpend) And _
   (good_ratio < Items(j).Profit / CDbl(Items(j).Cost))
   _
Then
   good_ratio = Items(j).Profit / CDbl(Items(j).Cost)
   good_j = j
End If
```

Випадковий пошук

Випадковий пошук (`random search`) виконується у відповідності до своєї назви. На кожному кроці алгоритм додає випадкову позицію, що задовольняє верхньому обмеженню на сумарну вартість позицій у портфелі. Цей метод пошуку також називається *методом Монте-Карло* (`Monte Carlo search` або `Monte Carlo simulation`).

Малоймовірно, що випадково обране рішення виявиться найкращим, отже необхідно багаторазово повторювати цей пошук, щоб одержати прийнятний результат. Хоча може здатися, що ймовірність знаходження гарного рішення при цьому мала, цей метод іноді дає дивно гарні результати. Залежно від значень даних і числа перевірених випадкових рішень результат, отриманий за допомогою

цієї евристики, часто виявляється кращим, ніж у випадку застосування методів сходження на пагорб або найменшої вартості.

Перевага випадкового пошуку полягає також і в тому, що цей метод легкий у розумінні й реалізації. Іноді складно уявити, як реалізувати рішення завдання за допомогою евристик сходження на пагорб, найменшої вартості, або збалансованого доходу, але завжди просто вибирати рішення випадковим образом. Навіть для дуже складних проблем, випадковий пошук є простим евристичним методом.

Підпрограма `RandomSearch` у програмі `Heur` використовує функцію `AddToSolution` для додавання до рішення випадкової позиції. Ця функція повертає значення `True`, якщо вона не може знайти позицію, що задовольняє умовам, і `False` в іншому випадку. Підпрограма `RandomSearch` викликає функцію `AddToSolution` доти, поки більше не можна додати ні однієї позиції.

```
Public Sub RandomSearch()
Dim num_trials As Integer
Dim trial As Integer
Dim i As Integer

  ' Зробити кілька спроб і вибрати найкращий результат.
  num_trials = NumItems          ' Використовувати N
  спроб.
  For trial = 1 To num_trials
       ' Випадковий вибір позицій, поки це можливо.
       Do While AddToSolution()
              ' Всю роботу виконує функція
  AddToSolution.
       Loop

       ' Визначити, чи краще це рішення, від
  попереднього.
       If test_profit > best_profit Then
              best_profit = test_profit
              best_cost = test_cost
              For i = 1 To NumItems
                     best_solution(i) = test_solution(i)
              Next i
       End If
```

```
        ' Скинути пробне рішення й зробити ще одну
    спробу.
        test_profit = 0
        test_cost = 0
        For i = 1 To NumItems
              test_solution(i) = False
        Next i
    Next trial
End Sub

Private Function AddToSolution() As Boolean
Dim num_left As Integer
Dim j As Integer
Dim selection As Integer

    ' Визначити, скільки залишилося позицій, які
    ' задовольняють обмеженню максимальної вартості.
    num_left = 0
    For j = 1 To NumItems
        If (Not test_solution(j)) And _
              (test_cost + Items(j).Cost <= ToSpend) _
                    Then num_left = num_left + 1
    Next j

    ' Зупинитися, якщо не можна знайти нову позицію.
    If num_left < 1 Then
        AddToSolution = False
        Exit Function
    End If

    ' Вибрати випадкову позицію.
    selection = Int((num_left) * Rnd + 1)

    ' Знайти випадково обрану позицію.
    For j = 1 To NumItems
        If (Not test_solution(j)) And _
              (test_cost + Items(j).Cost <= ToSpend) _
        Then
              selection = selection - 1
              If selection < 1 Then Exit For
```

```
      End If
  Next j

  test_profit = test_profit + Items(j).Profit
  test_cost = test_cost + Items(j).Cost
  test_solution(j) = True

  AddToSolution = True
End Function
```

Послідовне наближення

Ще одна стратегія полягає в тім, щоб почати з випадкового рішення й потім робити *послідовні наближення* (`incremental improvements`). Почавши з випадково обраного рішення, програма робить випадковий вибір. Якщо нове рішення краще від попереднього, програма закріплює зміни й продовжує перевірку інших випадкових змін. Якщо зміна не поліпшує рішення, програма відкидає його й робить нову спробу.

Для завдання формування портфеля особливо просто породжувати випадкові зміни. Програма просто вибирає випадкову позицію із пробного рішення і видаляє її з поточного рішення. Вона потім знову додає випадкові позиції в рішення доти, поки вони є. Якщо вилучена позиція мала дуже високу вартість, то на її місце програма може помістити кілька позицій.

Момент зупинки

Є кілька гарних способів визначити момент, коли варто припинити випадкові зміни. Для проблеми з N позиціями можна виконати N або N^2 випадкових змін перед тим, як зупинитися.

У програмі `Heur` цей підхід реалізований у процедурі `MakeChangesFixed`. Вона виконує певне число випадкових змін з рядом випадкових пробних рішень:

```
Public Sub MakeChangesFixed(K As Integer, num_trials As
  Integer, num_changes As Integer)
Dim trial As Integer
Dim change As Integer
```

```
Dim i As Integer
Dim removal As Integer

  For trial = 1 To num_trials
      ' Знайти випадкове пробне рішення й
  використовувати його
      ' як початкову точку.
      Do While AddToSolution()
            ' All the work is done by AddToSolution.
      Loop

      ' Почати із цього пробного рішення.
      trial_profit = test_profit
      trial_cost = test_cost
      For i = 1 To NumItems
            trial_solution(i) = test_solution(i)
      Next i

      For change = 1 To num_changes
            ' Видалити K випадкових позицій.
            For removal = 1 To K
                  RemoveFromSolution
            Next removal

            ' Додати максимально можливе
            ' число позицій.
            Do While AddToSolution()
                  ' All the work is done by
  AddToSolution.
            Loop

            ' Якщо це поліпшує пробне рішення,
  зберегти його.
            ' Інакше повернути колишнє значення
  пробного рішення.
            If test_profit > trial_profit Then
                  ' Зберегти зміни.
                  trial_profit = test_profit
                  trial_cost = test_cost
                  For i = 1 To NumItems
                        trial_solution(i) =
  test_solution(i)
                  Next i
            Else
                  ' Скинути пробне рішення.
                  test_profit = trial_profit
                  test_cost = trial_cost
                  For i = 1 To NumItems
```

```
                        test_solution(i) =
  trial_solution(i)
                  Next i
            End If
      Next change

      ' Якщо пробне рішення краще попереднього
      ' найкращого рішення, зберегти його.
      If trial_profit > best_profit Then
            best_profit = trial_profit
            best_cost = trial_cost
            For i = 1 To NumItems
                  best_solution(i) = trial_solution(i)
            Next i
      End If

      ' Скинути пробне рішення для
      ' наступної спроби.
      test_profit = 0
      test_cost = 0
      For i = 1 To NumItems
            test_solution(i) = False
      Next i
  Next trial
End Sub

Private Sub RemoveFromSolution()
Dim num_in_solution As Integer
Dim j As Integer
Dim selection As Integer

  ' Визначити число позицій у рішенні.
  num_in_solution = 0
  For j = 1 To NumItems
      If test_solution(j) Then num_in_solution =
  num_in_solution + 1
  Next j
  If num_in_solution < 1 Then Exit Sub

  ' Вибрати випадкову позицію.
  selection = Int((num_in_solution) * Rnd + 1)

  ' Знайти випадково обрану позицію.
  For j = 1 To NumItems
      If test_solution(j) Then
            selection = selection - 1
            If selection < 1 Then Exit For
      End If
```

```
  Next j

  ' Видалити позицію з рішення.
  test_profit = test_profit - Items(j).Profit
  test_cost = test_cost - Items(j).Cost
  test_solution(j) = False
End Sub
```

Інша стратегія полягає в тім, щоб вносити зміни доти, поки кілька послідовних змін не приносять поліпшення. Для завдання з N позиціями програма може вносити зміни доти, поки протягом N змін підряд поліпшення не буде.

Ця стратегія реалізована в підпрограмі `MakeChangesNoChange` програми `Heur`. Вона повторює спроби доти, поки певне число послідовних спроб не дасть ніяких поліпшень. Для кожної спроби вона вносить випадкові зміни в пробне рішення доти, поки після певного числа змін не наступить ніяких поліпшень.

```
Public Sub MakeChangesNoChange(K As Integer, _
  max_bad_trials As Integer, max_non_changes As
  Integer)
Dim i As Integer
Dim removal As Integer
Dim bad_trials As Integer        ' Неефективних спроб
  підряд.
Dim non_changes As Integer       ' Неефективних змін
  підряд.

  ' Повторювати спроби, поки не зустрінеться
  max_bad_trials
  ' спроб підряд без поліпшень.
  bad_trials = 0
  Do
      ' Вибрати випадкове пробне рішення для
      ' використання як початкову точку.
      Do While AddToSolution()
            ' All the work is done by AddToSolution.
      Loop

      ' Почати із цього пробного рішення.
      trial_profit = test_profit
      trial_cost = test_cost
      For i = 1 To NumItems
            trial_solution(i) = test_solution(i)
      Next i
```

```
    ' Повторювати, поки max_non_changes змін
    ' підряд не дасть поліпшень.
    non_changes = 0
    Do While non_changes < max_non_changes
          ' Видалити K випадкових позицій.
          For removal = 1 To K
                RemoveFromSolution
          Next removal

          ' Повернути максимально можливе число
позицій.
          Do While AddToSolution()
                ' All the work is done by
                ' AddToSolution.
          Loop

          ' Якщо це поліпшує пробне значення,
зберегти його.
          ' Інакше повернути колишнє значення
пробного рішення.
          If test_profit > trial_profit Then
                ' Зберегти поліпшення.
                trial_profit = test_profit
                trial_cost = test_cost
                For i = 1 To NumItems
                      trial_solution(i) =
test_solution(i)
                Next i
                non_changes = 0 ' This was a good
change.
          Else
                ' Reset the trial.
                test_profit = trial_profit
                test_cost = trial_cost
                For i = 1 To NumItems
                      test_solution(i) =
trial_solution(i)
                Next i
                non_changes = non_changes + 1    '
Погана зміна.
          End If
    Loop   ' Продовжити перевірку випадкових змін.

    ' Якщо ця спроба краще, ніж попереднє найкраще
    ' рішення, зберегти його.
    If trial_profit > best_profit Then
          best_profit = trial_profit
          best_cost = trial_cost
```

```
            For i = 1 To NumItems
                  best_solution(i) = trial_solution(i)
            Next i
            bad_trials = 0            ' Гарна спроба.
      Else
            bad_trials = bad_trials + 1            '
  Погана спроба.
      End If

      ' Скинути тестове рішення для наступної спроби.
      test_profit = 0
      test_cost = 0
      For i = 1 To NumItems
            test_solution(i) = False
      Next i
  Loop While bad_trials < max_bad_trials
End Sub
```

Локальні оптимуми

Якщо програма заміняє випадково обрану позицію в пробному рішенні, то може зустрітися рішення, яке вона не може поліпшити, але яке при цьому не буде найкращим з можливих рішень. Наприклад, розглянемо список інвестицій, наведений у табл. 12.5.

Таблиця 12.5

Можливі інвестиції

Інвестиція	**Вартість**	**Віддача**	**Прибуток**
A	47	56	9
B	43	51	8
C	35	40	5
D	32	39	7
E	31	37	6

Припустимо, що алгоритм випадково вибрав позиції A і B як початкове рішення. Його вартість буде дорівнювати 90 мільйонам доларів і воно принесе 17 мільйонів прибутку.

Якщо програма видалить позиції A і B, то вартість рішення буде усе ще настільки великою, що програма зможе додати лише одну позицію до рішення. Оскільки найбільший прибуток приносять позиції A і B, заміна їх іншими позиціями зменшить сумарний

прибуток. Випадкове видалення однієї позиції із цього рішення ніколи не приведе до поліпшення рішення.

Найкраще рішення містять позиції C, D і E. Його повна вартість дорівнює 98 мільйонам доларів і сумарний прибуток становить 18 мільйонів доларів. Щоб знайти це рішення, алгоритму знадобилося б видалити з рішення відразу обидві позиції A і B і потім додати на їхнє місце нові позиції.

Рішення такого типу, для яких невеликі зміни рішення не можуть поліпшити його, називаються *локальним оптимумом* (`local optimum`). Можна використовувати два способи, для того щоб програма не застрягала в локальному оптимумі, а могла б знайти *глобальний оптимум* (`global optimum`).

По-перше, можна змінити програму так, щоб вона видаляла більше однієї позиції під час випадкових змін. У цьому прикладі програма могла б знайти правильне рішення, якби вона одночасно видаляла б дві випадково обрані позиції. Проте для завдань більшого розміру видалення двох позицій може бути недостатнім. Програмі може знадобитися видаляти три, чотири або більше позицій.

Другий, більш простий спосіб, полягає в тому, щоб робити більше спроб, починаючи з різних початкових рішень. Деякі з початкових рішень будуть приводити до локальних оптимумів, але одне з них дозволить досягти глобального оптимуму.

Програма `Heur` ~~на диску із прикладами~~`(диск з прикладами – папка ProgR12)` демонструє три стратегії послідовних наближень. При виборі методу **Fixed 1** (Фіксований 1) робиться N спроб. Під час кожної спроби вибирається випадково рішення, що програма потім намагається поліпшити за 2 * N спроб, випадково видаляючи по одній позиції.

При виборі евристики **Fixed 2** (Фіксований 2) робиться лише одна спроба. При цьому програма вибирає випадкове рішення й намагається поліпшити його, випадковим чином видаляючи по одній позиції доти, поки протягом N послідовних змін не буде ніяких поліпшень.

При виборі евристики **No Changes 1** (Без змін 1) програма виконує спроби доти, поки після N послідовних спроб не буде ніяких поліпшень. Під час кожної спроби програма вибирає випадкове рішення й потім намагається поліпшити його, випадковим чином видаляючи по одній позиції доти, поки протягом N послідовних змін не буде ніяких поліпшень.

При виборі евристики **No Changes 2** (Без змін 2) робиться одна спроба. При цьому програма вибирає випадкове рішення й намагається поліпшити його, випадковим чином видаляючи по дві

позиції доти, поки протягом N послідовних змін не буде ніяких поліпшень. Назви евристик і їхніх описів наведені в табл. 12.6.

Таблиця 12.6

Стратегії послідовних наближень

Назва	**Кількість випробувань**	**Кількість змін**	**Кількість елементів, що видаляються**
Fixed 1	N	2* N	1
Fixed 2	1	10* N	2
No Changes 1	Доки не буде поліпшення за N змін	Доки не буде поліпшення за N змін	1
No Changes 1	N	Доки не буде поліпшення за N змін	2

Метод «відпалу»

Метод *відпалу* (`simulated annealing`) веде свій початок з термодинаміки. При відпалі металу він нагрівається до високої температури. Молекули в нагрітому металі роблять швидкі коливання, а при повільному остиганні вони починають розташовуватися впорядковано, утворюючи кристали. При цьому молекули поступово переходять у стан з мінімальною енергією.

При повільному остиганні металу сусідні кристали зливаються один з одним. Молекули в одному із кристалів залишають стан з мінімальною енергією й набувають порядок молекул в іншому кристалі. Енергія кристала більшого розміру, що вийшов, буде меншою, ніж сума енергій двох вихідних кристалів. Якщо охолодження відбувається досить повільно, то кристали стають дуже великими. Остаточний розподіл молекул представляє стан з дуже низькою енергією, і метал при цьому буде дуже твердим.

Починаючи зі стану з високою енергією, молекули зрештою досягають стану з дуже низькою енергією. На шляху до кінцевого положення вони проходять безліч локальних мінімумів енергії. Кожне сполучення кристалів утворить локальний мінімум. Кристали можуть поєднуватися один з одним тільки завдяки тимчасовому підвищенню енергії системи, щоб потім перейти у стан з меншою енергією.

Метод відпалу використовує аналогічний підхід для пошуку найкращого рішення задачі. Під час пошуку рішення програмою, вона може застрягти в локальному оптимумі. Щоб уникнути цього, програма час від часу вносить у рішення випадкові зміни, навіть якщо чергова зміна й не приводить до миттєвого поліпшення результату. Це може допомогти програмі вийти з локального оптимуму й відшукати краще рішення. Якщо ця зміна не веде до кращого рішення, то, ймовірно, через якийсь час програма його відкине.

Щоб ці зміни не виникали постійно, алгоритм змінює ймовірність виникнення випадкових змін згодом. Імовірність P виникнення одного з подібних змін визначається формулою P = 1 / Exp(E / (k * T)), де E — збільшення «енергії» системи, k — деяка постійна, і T — змінна, яка відповідає «температурі».

Спочатку температура має бути високою, тому й імовірність змін P = 1 / Exp(E / (k * T)) також досить велика. Інакше випадкові зміни могли б ніколи не виникнути. Із часом значення змінної T поступово знижується і ймовірність випадкових змін також зменшується. Після того, як модель дійде до точки, у якій ніякі зміни не зможуть поліпшити рішення, і температура T стане досить низкою, щоб імовірність випадкових змін була малою, алгоритм закінчує роботу.

Для задачі формування портфеля інвестицій енергія E – це значення, на яке зменшується прибуток у результаті змін. Наприклад, при видаленні позиції, що дає прибуток 10 мільйонів, і заміні її на позицію, що приносить 7 мільйонів прибутку, енергія, додана до системи, дорівнюватиме 3.

Помітьте, що якщо енергія велика, то ймовірність змін P = 1 / Exp(E / (k * T)) мала, тому ймовірність більших змін нижче.

Алгоритм відпалу в програмі `Heur` встановлює значення постійної k рівним різниці між найбільшим і найменшим прибутком можливих інвестицій. Початкова температура T задається рівною 0,75, помножене на різницю між максимальним та мінімальним прибутком від можливих варіантів інвестицій. Після виконання певного числа випадкових змін, температура T зменшується множенням на постійну 0,95.

```
Public Sub AnnealTrial(K As Integer, max_non_changes As
  Integer, _
  max_back_slips As Integer)
Const TFACTOR = 0.95

Dim i As Integer
```

```
Dim non_changes As Integer
Dim t As Double
Dim max_profit As Integer
Dim min_profit As Integer
Dim doit As Boolean
Dim back_slips As Integer

  ' Знайти позицію з мінімальним і максимальним
  прибутком.
  max_profit = Items(1).Profit
  min_profit = max_profit
  For i = 2 To NumItems
       If max_profit < Items(i).Profit Then max_profit =
  Items(i).Profit
       If min_profit > Items(i).Profit Then min_profit =
  Items(i).Profit
  Next i

  t = 0.75 * (max_profit - min_profit)
  back_slips = 0

  ' Вибрати випадкове пробне рішення
  ' як початкову точку.
  Do While AddToSolution()
       ' Вся робота виконується в процедурі
  AddToSolution.
  Loop

  ' Використовувати як пробне рішення.
  best_profit = test_profit
  best_cost = test_cost
  For i = 1 To NumItems
       best_solution(i) = test_solution(i)
  Next i

  ' Повторювати, поки протягом max_non_changes змін
  ' підряд не буде поліпшень.
  non_changes = 0
  Do While non_changes < max_non_changes
       ' Видалити випадкову позицію.
       For i = 1 To K
              RemoveFromSolution
       Next i

       ' Додати максимально можливе число позицій.
       Do While AddToSolution()
              ' Вся робота виконується в процедурі
  AddToSolution.
```

```
        Loop

        ' Якщо зміна поліпшує пробне рішення, зберегти
    його.
        ' Інакше повернути колишнє значення рішення.
        If test_profit > best_profit Then
              doit = True
        ElseIf test_profit < best_profit Then
              doit = (Rnd < Exp((test_profit -
    best_profit) / t))
              back_slips = back_slips + 1
              If back_slips > max_back_slips Then
                    back_slips = 0
                    t = t * TFACTOR
              End If
        Else
              doit = False
        End If
        If doit Then
              ' Зберегти поліпшення.
              best_profit = test_profit
              best_cost = test_cost
              For i = 1 To NumItems
                    best_solution(i) = test_solution(i)
              Next i
              non_changes = 0           ' Гарна зміна.
        Else
              ' Reset the trial.
              test_profit = best_profit
              test_cost = best_cost
              For i = 1 To NumItems
                    test_solution(i) = best_solution(i)
              Next i
              non_changes = non_changes + 1    ' Погана
    зміна.
        End If
    Loop       ' Продовжити перевірку випадкових змін.
End Sub
```

Порівняння евристик

Різні евристики по-різному поводяться в різних завданнях. Для завдання про формування портфеля евристика збалансованого прибутку працює досить добре з огляду на її простоту. Стратегії

послідовного наближення звичайно дають порівнянні результати, але для великих завдань їхнє виконання займає більше часу. Для інших завдань найкращою може бути будь-яка інша евристика, у тому числі з тих, які не обговорювалися в цьому розділі.

Евристичні методи звичайно виконуються швидше, ніж метод гілок та меж. Деякі з них, наприклад, методи сходження на пагорб, найменшої вартості й збалансованого прибутку виконуються дуже швидко, тому що вони розглядають тільки одне можливе рішення. Вони виконуються настільки швидко, що є сенс виконати усі їх по черзі і потім вибрати найкраще із трьох отриманих рішень. Проте це не гарантує того, що це рішення буде найкращим, але дає деяку впевненість, що воно виявиться досить гарним.

Інші складні завдання

Існує безліч дуже складних завдань, більшість із яких не має рішень із поліноміальною обчислювальною складністю. Інакше кажучи, не існує алгоритмів, які вирішували б ці завдання за час порядку $O(N^C)$ для будь-яких постійних C, навіть за $O(N^{1000})$.

Далі коротко описані деякі із цих завдань. В них також показано, чому вони є складними в загальному випадку й наскільки великим може виявитися дерево рішень завдання. Ви можете спробувати застосувати метод гілок та меж або евристики для рішення деяких із цих завдань.

Задача про здійснимість

Якщо є логічне твердження, наприклад, "`(A And Not B) Or C`", то чи існують значення змінних A, B і C, при яких це твердження істинне? У нашому прикладі легко побачити, що твердження істинне, якщо `A = true, B = false і C = false`. Для більш складних тверджень, що містять сотні змінних, буває досить складно визначити, чи може бути твердження істинним.

За допомогою методу, схожого на той, котрий використовувався при рішенні завдання про формування портфеля, можна побудувати дерево рішень для завдання про здійснимість

(satisfiability problem). Кожна гілка дерева буде відповідати рішенню про присвоєння змінній значення true або false. Наприклад, ліва гілка, що виходить із кореня, відповідає значенню першої змінної true.

Якщо в логічному виразі N змінних, то дерево рішень являє собою двійкове дерево висотою N + 1. Це дерево має 2^N листів, кожний з яких відповідає різній комбінації значень змінних.

У завданні про формування портфеля можна було використовувати метод гілок та меж, для того щоб уникнути пошуку в більшій частині дерева. У завданні про здійсненість вираз або правильний, або неправильний. При цьому не можна одержати часткового рішення, яке можна використовувати для відсікання шляхів у дереві.

Не можна також використовувати евристики для пошуку приблизного рішення для завдання про здійсненість. Будь-яке значення змінних, отримане за допомогою евристики, буде робити вираз правильний або помилковим. У математичній логіці не існує такого поняття, як наближене рішення.

Через незастосовність евристик і меншу ефективність методу гілок та меж завдання про здійсненість звичайно є дуже складним й вирішується тільки у випадку невеликого розміру завдання.

Задача про розбивку

Якщо задано безліч елементів зі значеннями $X_1, X_2, \ldots, X_N$, то чи існує спосіб розбити його на дві підмножини так, щоб сума значень всіх елементів у кожній з підмножин була однаковою? Наприклад, якщо елементи мають значення 3, 4, 5 і 6, то їх можна розбити на дві підмножини {3, 6} і {4, 5}, сума значень елементів у кожному з яких дорівнює 9.

Щоб змоделювати це завдання за допомогою дерева, припустимо, що гілкам відповідає переміщення елемента в одну із двох підмножин. Ліва гілка, що виходить із кореневого вузла, відповідає переміщенню першого елемента в першу підмножину, а права гілка — у другу підмножину.

Якщо існує лише N елементів, то дерево рішення буде являти собою двійкове дерево висотою N + 1. Воно буде містити 2^N листів і 2^{N+1} вузлів. Кожний лист відповідає одному з варіантів розміщення елементів у двох підмножинах.

При рішенні цього завдання можна застосувати метод гілок та меж. При розгляді часткових рішень завдання треба відслідковувати, наскільки різняться сумарні значення елементів у двох підмножинах. Якщо в якийсь момент сумарне значення елементів для однієї з підмножин небагато відрізняється від сумарного значення для іншої, а додавання всіх елементів, що залишилися, не дозволяє змінити це співвідношення, то немає сенсу продовжувати рух вниз по цій галузці.

Так само, як і у випадку із завданням про здійсниміcть, для завдання про розбивку (`partition problem`) не можна одержати наближене рішення. У результаті завжди має вийти дві підмножини, сумарне значення елементів у яких буде або не буде однаковим. Це означає, що для рішення цього завдання не застосовані евристики, які використовувалися для рішення завдання про формування портфеля.

Завдання про розбивку можна узагальнити в такий спосіб. Є множина елементів зі значеннями X_1, X_2, … , X_N, потрібно знайти такий спосіб розподілу елементів на дві підмножини, щоб різниця сум значень елементів у двох підмножинах була мінімальною.

Одержати точне рішення цього завдання складніше, ніж для вихідного завдання про розбивку. Якби існував простий спосіб рішення завдання в загальному випадку, то його можна було б використовувати для рішення вихідного завдання. У цьому випадку можна було б просто знайти дві підмножини, що задовольняють умовам, а потім перевірити, чи збігаються суми значень елементів у них.

Для рішення загального випадку завдання можна використовувати метод гілок та меж, приблизно так само, як він використовувався для рішення ~~частного~~ окремого випадку завдання, щоб уникнути пошуку по всьому дереву. Можна також використовувати при цьому евристичний підхід. Наприклад, можна перевіряти елементи в порядку убування їхнього значення, поміщаючи черговий елемент у підмножину з меншою сумою значень елементів. Також можна було б легко використовувати випадковий пошук, метод послідовних наближень, або метод відпалу для пошуку наближеного рішення цього завдання в загальному випадку.

Завдання пошуку Гамільтонового шляху

Якщо задано мережу, то *Гамільтоновим шляхом* (`Hamiltonian path`) для неї називається шлях, що обходить всі вузли в мережі тільки один раз і потім повертається в початкову точку.

На рис.12.7 показана невелика мережа й Гамільтоновий шлях для неї, нарисований жирною лінією.

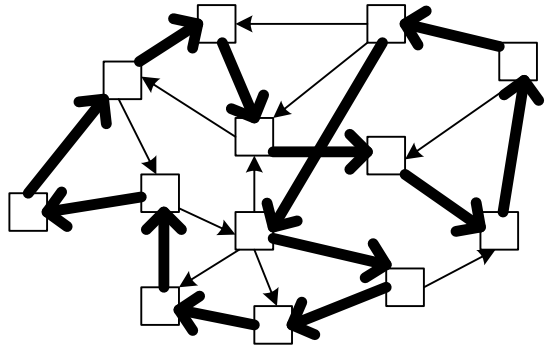


Рис.12.7. Гамільтоновий шлях

Задача пошуку Гамільтонового шляху формулюється так: якщо задано мережу, чи існує для неї Гамільтоновий шлях?

Оскільки Гамільтоновий шлях обходить всі вузли в мережі, то не потрібно визначати, які з вузлів потрапляють у нього, а які ні. Необхідно встановити тільки порядок, у якому їх потрібно обійти для створення Гамільтонового шляху.

Для моделювання цієї задачі за допомогою дерева припустимо, що гілки відповідають вибору наступного вузла в шляху. Кореневий вузол тоді буде містити `N` гілок, що відповідають початку шляху в кожному з `N` вузлів. Кожний з вузлів першого рівня буде мати `N - 1` гілок, по одній гільці для кожного із `N - 1` вузлів, що залишилися. Вузли на наступному рівні дерева будуть мати `N - 2` гілок і так далі. Нижній рівень дерева буде містити `N!` листів, що відповідають `N!` можливих шляхів. В дереві перебуватиме порядку `O(N!)` вузлів.

Кожний лист відповідає Гамільтоновому шляху, але число листів може бути різним для різних мереж. Якщо два вузли в мережі не зв'язані один з одним, то в дереві будуть відсутні галузі, які

відповідають переходам між цими двома вузлами. Це зменшує число шляхів у дереві й, відповідно, число листів.

Так само, як і в задачах про здійсненність й про розбивку, для завдання пошуку Гамільтонового шляху не можна одержати наближене рішення. Шлях може або бути Гамільтоновим, або ні. Це означає, що евристичний підхід і метод гілок та меж не допоможуть при пошуку Гамільтонового шляху. Що ще гірше, дерево рішень для задачі пошуку Гамільтонового шляху містить порядку O(N!) вузлів. Це набагато більше, ніж порядку $O(2^N)$ вузлів, які містять дерева рішень для задачі про здійсненність й розбивку. Наприклад, 2^{20} приблизно дорівнює $1 * 10^6$, тоді як 20! становить близько $2,4 * 10^{18}$ - у мільйон разів більше. Через дуже великий розмір дерева рішень задача знаходження Гамільтонового шляху, пошук у ньому можна виконати тільки для задач дуже невеликого розміру.

Задача комівояжера

Задача комівояжера (traveling salesman problem) тісно пов'язана із задачею пошуку Гамільтонового шляху. Вона формулюється так: знайти найкоротший Гамільтоновий шлях для мережі.

Ця задача має приблизно таке ж відношення до задачі пошуку Гамільтонового шляху, як узагальнений випадок завдання про розбивку для простої задачі про розбивку. У першому випадку виникає питання про існування рішення. У другому - яке наближене рішення буде найкращим. Якби існувало просте рішення другої задачі, то його можна було б використовувати для рішення першого варіанта задачі.

Звичайно задача комівояжера виникає тільки в мережах, що містять велике число Гамільтонових шляхів. У типовому прикладі комівояжерові потрібно відвідати декілька клієнтів, використовуючи найкоротший маршрут. У випадку звичайної мережі вулиць, будь-які дві точки в мережі зв'язані між собою, тому будь-який маршрут являє собою Гамільтоновий шлях. Задача полягає в тому, щоб знайти найкоротший з них.

Так само, як і у випадку пошуку Гамільтонового шляху, дерево рішень для цього завдання містить порядку O(N!) вузлів. Так само, як і в узагальненій задачі про розбивку, для відсікання гілок

дерева й прискорення пошуку рішення задачі середніх розмірів можна використовувати метод гілок та меж.

Існує також кілька гарних евристичних методів послідовних наближень для задачі комівояжера. Наприклад, використання стратегії пари шляхів, при якій перебираються пари відрізків маршруту. Програма перевіряє, чи стане маршрут коротше, якщо видалити пари відрізків і замінити їх двома новими, так щоб маршрут при цьому залишався замкнутим. На рис.12.8 показано, як змінюється маршрут, якщо відрізки X_1 і X_2 замінити відрізками Y_1 і Y_2. Аналогічні стратегії послідовних наближень розглядають заміну трьох або більше відрізків шляху одночасно.

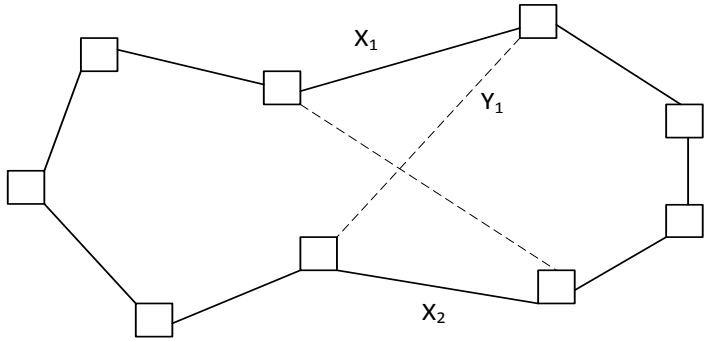


Рис.12.8.Поліпшення Гамільтонового шляху за допомогою двох оптимумів

Звичайно такі кроки послідовного наближення повторюються багаторазово або доти, поки не будуть перевірені всі можливі пари відрізків шляху. Після того, як подальші кроки не приводять до поліпшень, можна зберегти результат і почати роботу знову, випадковим чином вибравши інший вихідний маршрут. Після перевірки досить великої кількості різних випадкових вихідних маршрутів ймовірно буде знайдений досить короткий шлях.

Задача про пожежні депо

Задача про пожежні депо (`firehouse problem`) формулюється так: якщо задано мережу, деяке число `F` і відстань `D`, то чи існує спосіб розмісити `F` пожежних депо таким чином, щоб всі вузли мережі перебували не далі, ніж на відстані `D` від найближчого пожежного депо?

Цю задачу можна змоделювати за допомогою дерева рішень, у якому кожна гілка визначає місце розташування відповідного пожежного депо в мережі. Кореневий вузол буде мати `N` гілок, що відповідають розміщенню першого пожежного депо в одному з `N` вузлів мережі. Вузли на наступному рівні дерева будуть мати `N - 1` гілок, що відповідають розміщенню другого пожежного депо в одному із `N - 1`, вузлів що залишилися. Якщо існує лише `F` пожежних депо, то висота дерева рішень буде `F`, воно буде містити порядку $O(N^F)$ вузлів. У дереві буде `N * (N - 1) * ... * (N - F)` листів, що відповідають різним варіантам розміщення пожежних депо в мережі.

Так само, як і в задачах про здійсн**имість**~~енність~~, розбивку і пошук Гамільтонового шляху, у цій задачі потрібно дати позитивну або негативну відповідь на питання. Це означає, що при перевірці дерева рішень не можна використовувати часткові або наближені рішення.

Можна, проте, використовувати різновид методу гілок та меж, якщо на ранніх етапах рішення визначити, які з варіантів розміщення пожежних депо не приводять до рішення. Наприклад, безглуздо поміщати чергове депо між двома іншими, розташованими поруч. Якщо всі вузли на відстані `D` від нового пожежного депо вже перебувають у межах цієї відстані від іншого депо, виходить, нове депо потрібно помістити в будь-яке інше місце. Проте такого роду обчислення також віднімають досить багато часу, і завдання усе ще залишається дуже складним.

Так само, як і для задач про розбивку й пошук Гамільтонового шляху, існує узагальнений випадок задачі про пожежні депо. В узагальненому випадку задача формулюється так: якщо задано мережу й деяке число `F`, у яких вузлах мережі потрібно помістити `F` пожежних депо, щоб найбільша відстань від будь-якого вузла до пожежного депо була мінімальною?

Так само, як і узагальнених випадках інших задач, для пошуку часткового й наближеного рішень цієї задачі можна використовувати метод гілок та меж і евристичний підхід. Це трохи спрощує перевірку дерева рішень. Хоча дерево рішень усе ще залишається величезним, можна, принаймні, знайти приблизні рішення, навіть якщо вони й не є найкращими.

Резюме

Ви можете використати дерева рішень для моделювання складних завдань. Знаходження кращого рішення відповідає знаходженню кращого шляху через дерево. На жаль, для багатьох цікавих завдань дерева рішень мають величезний розмір, тому вирішувати такі завдання методом повного перебору складно.

За допомогою методу гілок та меж можна скорочувати кількість гілок деяких дерев, що дозволяє точно вирішувати завдання великої складності.

Однак у рішенні найбільших завдань не допоможе навіть застосування цього методу. У таких випадках треба використати евристику, щоб одержати наближені рішення. Використовуючи методи типу випадкового пошуку або послідовних наближень, можна знайти прийнятне рішення, навіть якщо не відомо, чи буде воно найкращим.

Контрольні запитання та завдання

1. Коли використовують дерева рішень?
2. Перелічіть способи пошуку для малих та великих дерев.
3. Як за допомогою дерев гри моделюється гра в хрестики - нулики?
4. У чому зміст мінімаксної стратегії?
5. Що таке евристика? Що спільне у евристичних алгоритмів?
6. Поясніть сенс метода гілок та меж.
7. Що працює швидше евристичні методи чи метод гілок та меж?
8. Порівняйте стратегії сходження на пагорб та найменшої вартості.
9. Поясніть алгоритм методу відпалу.
10. У чому ідея методу збалансованого прибутку?

~~Чим характеризуються динамічні структури даних?~~

11. Назвіть складні завдання, в рішенні яких застосовують евристики.
12. Як формулюється задача про розбивку?
13. Як моделюється задача про пожежні депо?

14. Скільки листів може мати дерево рішень для завдання про здійсниміcть?
15. Евристики або метод гілок та меж використовують для пошуку приблизного рішення в завданні про здійсниміcть?

Що таке дерево рішень в задачі комівояжера, яка вирішується методом гілок і меж?

Розділ XIII
МЕТОДИ СОРТУВАННЯ

Сортування *– це процес розміщення елементів послідовності в деякому числовому або лексикографічному порядку.*

Сортування – одна з найбільш активно досліджуваних тем у комп'ютерних алгоритмах з ряду причин.

По-перше, сортування – це завдання, яке часто зустрічається в багатьох програмних додатках. Будь-який список даних буде нести більше змісту, якщо його впорядкувати якимось чином. Часто необхідно сортувати дані декількома різними способами.

По-друге, багато алгоритмів сортування є цікавими прикладами програмування. Вони демонструють важливі методи, такі як часткове впорядкування, рекурсія, злиття списків і зберігання двійкових дерев у масиві.

Нарешті, сортування є завданням із точними теоретичними обмеженнями продуктивності. Можна показати, що час виконання будь - якого алгоритму сортування, що використовує порівняння порядку O(N * log(N)). Деякі алгоритми досягають теоретичної межі, тобто вони є оптимальними в цьому змісті. Є навіть ряд алгоритмів сортування, які використовують інші методи замість порівнянь, вони виконуються швидше, ніж за час порядку O(N * log(N)).

Загальні принципи

У цьому розділі розповідається про деякі алгоритми сортування, які поводяться по-різному в різних обставинах. Наприклад, бульбашкове сортування випереджає швидке за швидкістю виконання, якщо елементи, що сортуються, якоюсь мірою вже впорядковані, але виконується повільніше при хаотичному розташуванні елементів.

Кожен підрозділ присвячений якомусь алгоритму. Але спочатку обговорюються питання, що стосуються всіх алгоритмів сортування в цілому.

Таблиці покажчиків

При сортуванні елементів даних програма організує з них деяку подібність структури даних. Цей процес може бути швидким або повільним залежно від типу елементів. Переміщення цілого числа на нове положення в масиві може бути набагато швидшим, ніж переміщення визначеної користувачем структури даних. Якщо структура являє собою список даних про співробітників, що містить тисячі байт інформації, копіювання одного елемента може зайняти досить багато часу.

Для підвищення продуктивності сортування великих об'ємів даних можна заносити у таблицю індексів ключові поля даних, які використовуються для сортування. У цій таблиці перебувають ключі записів і індекси елементів іншого масиву, у якому і перебувають записи даних. Наприклад, припустимо, що ви збираєтесь відсортувати список записів про співробітників, обумовлений такою структурою:

```
Type Emloyee
  ID As Integer
  Lastname As String
  Firstname As String
  < і т.д.>
End Type
' Виділити пам'ять під записи.
Dim Emloyeedata(1 To 10000)
```

Щоб відсортувати співробітників за ідентифікаційним кодом, потрібно створити таблицю індексів, яка містить індекси й значення `ID values` із записів. Індекс елемента показує, який запис у масиві `Employeedata` містить відповідні дані.

```
Type Idindex
  ID As Integer
  Index As Integer
End Type

' Таблиця індексів.
Dim Idindexdata(1 To 10000)
```

Проініціалізуємо таблицю індексів так, щоб перший індекс указував на перший запис даних, другий – на другий і т.д.

```
For i = 1 To 10000
  Idindexdata(i).ID = Employeedata(i).ID
  Idindexdata(i).Index = i
Next i
```

Потім відсортуємо таблицю індексів за ідентифікаційним номером `ID`. Після цього поле `Index` у кожному елементі `Idindexdata` указує на відповідний запис даних. Наприклад, перший запис у відсортованому списку – це `Employeedata(Idindexdata(1).Index)`.

Для того щоб сортувати дані в різному порядку, можна створити кілька різних таблиць індексів і управляти ними окремо. У наведеному прикладі можна було б створити ще одну таблицю індексів, яка впорядковує співробітників за прізвищем. При додаванні або видаленні записів необхідно обновляти кожну таблицю індексів незалежно.

Об'єднання та стискання ключів

Іноді можна зберігати ключі списку в комбінованій або стислій формі. Наприклад, можна було б *об'єднати* (`combine`) у програмі два поля, що відповідають імені й прізвищу, в один ключ. Це дозволило б спростити й прискорити порівняння. Зверніть увагу на відмінності між двома наступними фрагментами коду, які порівнюють два записи про співробітників:

```
' Використовуючи різні ключі.
If emp1.Lastname > emp2.Lastname Or _
  (emp1.Lastname = emp2.Lastname And _
      And emp1.Firstname > emp2.Firstname) Then
  Dosomething

 ' Використовуючи об'єднаний ключ.
If emp1.Cominedname > emp2.Combinedname Then
  Dosomething
```

Також іноді можна *стискати* (compress) ключі. Стиснуті ключі займають менше місця та зменшують розмір таблиць індексів. Це дозволяє сортувати списки більшого розміру без перевитрати пам'яті, швидше переміщати елементи в списку, і часто також прискорює порівняння елементів.

Один з методів стискання рядків – кодування їх цілими числами або даними іншого числового формату. Числові дані займають менше місця, ніж рядки й порівняння двох чисельних значень також відбувається набагато швидше, ніж порівняння двох рядків. Звичайно строкові операції незастосовні для рядків, представлених числами.

Наприклад, припустимо, що ми прагнемо закодувати рядок, що складається із великих латинських букв. Можна вважати, що кожний символ – це число, взяте на основі 27. Необхідно використовувати основу 27, щоб представити 26 букв і ще одну цифру для позначення кінця слова. Без відмітки кінця слова закодований рядок **AA** йшов би після рядка **B**, тому що в рядку **AA** дві цифри, а в рядку **B** – одна.

Код на основі 27 для рядка із трьох символів дає формула 27^2 * (перша буква - A + 1) + 27 * (друга буква - A + 1) + 27 * (третя буква - A + 1). Якщо в рядку менше трьох символів, замість значення (третя буква - A + 1) підставляється 0. Наприклад, рядок **FOX** кодується так:

```
272 * (F - A + 1) + 27 * (O - A + 1) + (X - A +1) =
  4803
```

Рядок **NO** кодується в такий спосіб:

```
272 * (N - A + 1) + 27 * (O - A + 1) + (0) = 10.611
```

Бачимо, що 10.611 більше 4803, оскільки **NO > FOX**.

У такий же спосіб можна закодувати рядок з 6 великих букв у виді числа у форматі **long** і рядок з 10 букв – як число у форматі **double.** Дві наступні процедури конвертують рядки в числа у форматі **double** і назад:

```
Const STRING_BASE = 27
Const ASC_A = 65                ' ASCII код для символу
  "A".
' Перетворення рядка із числа у форматі double.
'
' full_len — повна довжина, яку повинен мати рядок.
```

```
' Потрібна, якщо рядок занадто короткий (наприклад,
  "AX" —
' це рядок із трьох символів).
Function Stringtodbl (txt As String, full_len As
  Integer) As Double
Dim strlen As Integer
Dim i As Integer
Dim value As Double
Dim ch As String * 1

  strlen = Len(txt)
  If strlen > full_len Then strlen = full_len

  value = 0#
  For i = 1 To strlen
       ch = Mid$(txt, i, 1)
       value = value * STRING_BASE + Asc(ch) - ASC_A + 1
  Next i

  For i = strlen + 1 To full_len
       value = value * STRING_BASE
  Next i
End Function

' Зворотне декодування рядка з формату double.
Function Dbltostring (Byval value As Double) As String
Dim strlen As Integer
Dim i As Integer
Dim txt As String
Dim Power As Integer
Dim ch As Integer
Dim new_value As Double

  txt = ""
  Do While value > 0
       new_value = Int(value / STRING_BASE)
       ch = value - new_value * STRING_BASE
       If ch <> 0 Then txt = Chr$(ch + ASC_A - 1) + txt
       value = new_value
  Loop

  Dbltostring = txt
End Function
```

Можна також кодувати рядки, що складаються не тільки із великих букв. Рядок із великих букв і цифр можна закодувати на основі 37 замість 27. Код букви A дорівнюватиме 1, B – 2, …, Z – 26,

код 0 буде 27, …, і 9 – 36. Рядок **AH7** буде кодуватися як $37^2 * 1 + 37 * 8 + 35 = 1700$.

Звичайно при використанні більшої основи довжина рядка, який можна закодувати числом типу **integer, long**– або **double** буде відповідно коротшою. При основі, що дорівнює 37, можна закодувати рядок з 2 символів у числі формату **integer,** з 5 символів у числі формату **long,** і 10 символів у числі формату **double.**

Сортування вибором

Сортування вибором (selectionsort) – простий алгоритм зі складністю порядку $O(N^2)$. Ідея полягає в пошуку найменшого елемента в списку, який потім міняється місцями з елементом на вершині списку. Потім шукається найменший елемент із тих, що залишилися, і міняється місцями із другим елементом. Процес триває доти, поки всі елементи не займуть своє кінцеве положення.

При сортуванні масиву a[1], a[2], ..., a[n] методом простого вибору серед усіх елементів знаходиться елемент із найменшим значенням a[i], і a[1] і a[i] обмінюються значеннями. Потім цей процес повторюється для одержуваних підмасивів a[2], a[3], ..., a[n], ... a[j], a[j+1], ..., a[n] доти, поки ми не дійдемо до підмасиву a[n], що містить до цього моменту найбільше значення. Робота алгоритму ілюструється прикладом у табл. 13.1.

Таблиця 13.1

Приклад сортування простим вибором

Початковий стан масиву	**8 23 5 65 44 33 1 6**
Крок 1	1 23 5 65 44 33 8 6
Крок 2	1 5 23 65 44 33 8 6
Крок 3	1 5 6 65 44 33 8 23
Крок 4	1 5 6 8 44 33 65 23
Крок 5	1 5 6 8 33 44 65 23
Крок 6	1 5 6 8 23 44 65 33
Крок 7	1 5 6 8 23 33 65 44
Крок 8	1 5 6 8 23 33 44 65

Для методу сортування простим вибором необхідне число порівнянь – n*(n-1)/2. Порядок необхідного числа пересилань (включаючи ті, які потрібні для вибору мінімального елемента) у найгіршому разі становить $O(n^2)$. Однак порядок середнього числа пересилань є $O(n^2*\ln n)$, що в ряді випадків робить цей метод кращим.

```
Public Sub Selectionsort(List() As Long, min As Long,
  max As Long)
Dim i As Long
Dim j As Long
Dim best_value As Long
Dim best_j As Long

  For i = min To max - 1
       ' Знайти найменший елемент із тих, що залишилися.
       best_value = List(i)
       best_j = i
       For j = i + 1 To max
              If List(j) < best_value Then
                     best_value = List(j)
                     best_j = j
              End If
       Next j

       ' Помістити елемент на місце.
       List(best_j) = List(i)
  List(i) = best_value
  Next i
End Sub
```

При пошуку I-го найменшого елемента, алгоритму доводиться перебрати N-I елементів, які ще не зайняли своє кінцеве положення. Час виконання алгоритму пропорційний N + (N - 1) + (N - 2) + … + 1, або порядку $O(N^2)$.

Сортування вибором непогано працює зі списками, елементи в яких розташовані випадково або в прямому порядку, але трохи гірше, якщо список спочатку відсортований у зворотному порядку. Для пошуку найменшого елемента в списку, сортування вибором виконує такий код:

```
If list(j) < best_value Then
  best_value = list(j)
  best_j = j
End If
```

Якщо спочатку список відсортований у зворотному порядку, умова **`list(j) < best_value`** виконується більшу частину часу. Наприклад, при першому проході вона буде істинна для всіх елементів, оскільки кожний елемент менше попереднього. Алгоритм буде багаторазово виконувати рядок з оператором **`If,`** що приведе до деякого вповільнення роботи алгоритму.

Це не найшвидший алгоритм, але він надзвичайно простий. Це не тільки полегшує його розробку й відладку, але й робить сортування вибором досить швидким для невеликих завдань. Багато інших алгоритмів настільки складні, що сортують дуже маленькі списки повільніше.

Рандомізація

У деяких програмах потрібне виконання операції, зворотної сортуванню. Одержавши список елементів, програма має розташувати їх у випадковому порядку. *Рандомізацію* (`unsorting`) списку нескладно виконати, використовуючи алгоритм, подібний до сортування вибором.

Для кожного положення в списку алгоритм випадковим чином вибирає елемент, який повинен його зайняти з тих, які ще не були переміщені на своє місце. Потім цей елемент міняється місцями з елементом, який перебуває на цій позиції.

```
Public Sub Unsort(List() As Long, min As Long, max As
  Long)
Dim i As Long
Dim Pos As Long
Dim tmp As Long

  For i - min To max - 1
      pos = Int((max - i + 1) * Rnd + i)
      tmp = List(pos)
      List(pos) = List(i)
      List(i) = tmp
  Next i
End Sub
```

Так як алгоритм заповнює кожну позицію тільки один раз, його складність порядку `O(N)`.

Нескладно показати, що ймовірність того, що елемент виявиться в якій-небудь позиції, дорівнює `1/N`. Оскільки елемент

може виявитися в будь-якому положенні з рівною ймовірністю, цей алгоритм дійсно приводить до випадкового розміщення елементів.

Результат залежить від того, наскільки гарним є генератор випадкових чисел. Функція **Rnd** в Visual Basic дає прийнятний результат для більшості випадків. Слід переконатися, що програма використовує оператор **Randomize** для ініціалізації функції **Rnd**, інакше при кожному запуску програми функція **Rnd** буде видавати ту ж саму послідовність «випадкових» значень.

Помітимо, що для алгоритму не важливий первісний порядок розташування елементів. Якщо вам необхідно неодноразово рандомувати список елементів, немає необхідності його попередньо сортувати.

Програма **Unsort** показує використання цього алгоритму для рандомізації відсортованого списку. Уведіть число елементів, які ви прагнете рандомізувати, і натисніть кнопку Go (Почати). Програма показує вихідний відсортований список чисел і результат рандомізації.

Сортування вставкою

Сортування вставкою (insertionsort) – ще один алгоритм зі складністю порядку $O(N^2)$. Ідея полягає в тому, щоб створити новий відсортований список, переглядаючи по черзі всі елементи у похідному списку. При цьому, вибираючи черговий елемент, алгоритм переглядає зростаючий відсортований список, знаходить необхідне положення елемента в ньому і поміщає елемент на своє місце в новий список (табл. 13.2)

Таблиця 13.2

Приклад сортування методом простого включення

Початковий стан масиву	**8 23 5 65 44 33 1 6**
Крок 1	8 23 5 65 44 33 1 6
Крок 2	8 5 23 65 44 33 1 6 5 8 23 65 44 33 1 6
Крок 3	5 8 23 65 44 33 1 6
Крок 4	5 8 23 44 65 33 1 6

Продовження табл. 13.2

Початковий стан масиву	**8 23 5 65 44 33 1 6**
Крок 5	5 8 23 44 33 65 1 6 5 8 23 33 44 65 1 6
Крок 6	5 8 23 33 44 1 65 6 5 8 23 33 1 44 65 6 5 8 23 1 33 44 65 6 5 8 1 23 33 44 65 6 5 1 8 23 33 44 65 6 1 5 8 23 33 44 65 6
Крок 7	1 5 8 23 33 44 6 65 1 5 8 23 33 6 44 65 1 5 8 23 6 33 44 65 1 5 8 6 23 33 44 65 1 5 6 8 23 33 44 65

```
Public Sub Insertionsort(List() As Long, min As Long,
  max As Long)
Dim i As Long
Dim j As Long
Dim k As Long
Dim max_sorted As Long
Dim next_num As Long
  max_sorted = min -1
  For i = min To max
       ' Це число, що вставляється.
       Next_num = List(i)

       ' Пошук його позиції в списку.
       For j = min To max_sorted
              If List(j) >= next_num Then Exit For
       Next j

       ' Перемістити більші елементи вниз, щоб
       ' звільнити місце для нового числа.
       For k = max_sorted To j Step -1
              List(k + 1) = List(k)
       Next k
       ' Помістити новий елемент.
       List(j) = next_num

       ' Збільшити лічильник відсортованих елементів.
       max_sorted = max_sorted + 1
  Next i
End Sub
```

Може виявитися, що для кожного з елементів у похідному списку, алгоритму потрібно перевіряти всі вже відсортовані елементи. Це відбувається, наприклад, якщо у похідному списку елементи були вже відсортовані. У цьому випадку алгоритм поміщає кожний новий елемент у кінець зростаючого відсортованого списку.

Повне число кроків, які потрібно буде виконати, становить `1 + 2 + 3 + … + (N - 1)`, тобто $O(N^2)$. Це не так ефективно, якщо порівнювати з теоретичною межею `O(N * log(N))` для алгоритмів на основі операцій порівняння. Фактично цей алгоритм не надто швидкий навіть у порівнянні з іншими алгоритмами порядку $O(N^2)$, такими як сортування вибором.

Вставка у зв'язних списках

Можна використовувати варіант сортування вставкою для впорядкування елементів не в масиві, а у зв'язному списку. Цей алгоритм шукає необхідне положення елемента в зростаючому зв'язному списку, і потім поміщає туди новий елемент, використовуючи операції роботи зі зв'язними списками.

```
Public Sub Linkinsertionsort(Listtop As Listcell)
Dim new_top As New Listcell
Dim old_top As Listcell
Dim cell As Listcell
Dim after_me As Listcell
Dim nxt As Listcell

  Set old_top = Listtop.Nextcell
  Do While Not (old_top Is Nothing)
       Set cell = old_top
       Set old_top = old_top.Nextcell

       ' Знайти, куди необхідно помістити елемент.
       Set after_me = new_top
       Do
              Set nxt = after_me.Nextcell
              If nxt Is Nothing Then Exit Do
              If nxt.Value >= cell.Value Then Exit Do
              Set after_me = nxt
       Loop

       ' Вставити елемент після позиції after_me.
```

```
      Set after_me.Nextcll = cell
      Set cell.Nextcell = nx
  Loop
  Set Listtop.Nextcell = new_top.Nextcell
End Sub
```

Оскільки цей алгоритм перебирає всі елементи, може знадобитися порівняння кожного елемента з усіма елементами у відсортованому списку. У цьому найгіршому випадку обчислювальна складність алгоритму порядку $O(N^2)$.

Найкращий випадок для цього алгоритму досягається, коли вихідний список спочатку відсортований у зворотному порядку. При цьому кожний наступний елемент менший, ніж попередній, тому алгоритм поміщає його в початок відсортованого списку. При цьому потрібно виконати тільки одну операцію порівняння елементів і в найкращому випадку час виконання алгоритму буде порядку $O(N)$.

В усередненому випадку, алгоритму доведеться провести пошук приблизно у половині відсортованого списку, для того щоб знайти місце розташування елемента. При цьому алгоритм виконується приблизно за $1 + 1 + 2 + 2 + \ldots + N/2$, або порядку $O(N^2)$ кроків.

Перевага використання зв'язних списків для вставки в тому, що при цьому переміщаються тільки покажчики, а не самі записи даних. Передача покажчиків може бути швидше, ніж копіювання записів цілком, якщо елементи являють собою великі структури даних.

Бульбашкове сортування

Бульбашкове *сортування* (bubblesort) – це алгоритм, призначений для сортування списків, які вже перебувають у майже впорядкованому стані. Якщо на початку процедури список повністю відсортований, алгоритм виконується дуже швидко за час порядку $O(N)$. Якщо частина елементів перебувають не на своїх місцях, алгоритм виконується повільніше. Якщо спочатку елементи розташовані у випадковому порядку, алгоритм виконується за час порядку $O(N^2)$. Тому перед застосуванням бульбашкового сортування важливо переконатися, що елементи в основному розташовані по порядку.

При бульбашковому сортуванні список проглядається доти, поки не знайдуться два сусідні елементи, розташовані не по порядку. Тоді вони міняються місцями, і процедура триває далі. Алгоритм повторює цей процес доти, поки всі елементи не займуть свої місця.

Під час кожного проходу елемент переміщається на одну позицію ближче до свого кінцевого положення. Він рухається до вершини списку подібно до бульбашки газу, яка спливе до поверхні в склянці води. Цей ефект і дав назву алгоритму бульбашкового сортування.

Просте обмінне сортування (у просторіччі називане "методом бульбашки") для масиву `a[1], a[2], ..., a[n]` працює в такий спосіб. Починаючи з кінця масиву порівнюються два сусідні елементи `(a[n] і a[n-1])`. Якщо виконується умова `a[n-1] > a[n]`, то значення елементів міняються місцями. Процес триває для `a[n-1] і a[n-2]` і т.д., поки не буде зроблене порівняння `a[2] і a[1]`. Зрозуміло, що після цього на місці `a[1]` виявиться елемент масиву з найменшим значенням. На другому кроці процес повторюється, але останніми порівняються `a[3] і a[2]`. І так далі. На останньому кроці будуть порівнюватися тільки поточні значення `a[n] і a[n-1]`. Зрозуміла аналогія з бульбашкою, оскільки найменші елементи ("найлегші") поступово "спливуть" до верхньої межі масиву. Приклад сортування методом бульбашки показаний у табл. 13.3)

Таблиця 13.3

Приклад сортування методом бульбашки

Початковий стан масиву	**8 23 5 65 44 33 1 6**
Крок 1	8 23 5 65 44 33 1 6 8 23 5 65 44 1 33 6 8 23 5 65 1 44 33 6 8 23 5 1 65 44 33 6 8 23 1 5 65 44 33 6 8 1 23 5 65 44 33 6 1 8 23 5 65 44 33 6

Продовження табл. 13.3

Початковий стан масиву	**8 23 5 65 44 33 1 6**
Крок 2	1 8 23 5 65 44 6 33 1 8 23 5 65 6 44 33 1 8 23 5 6 65 44 33 1 8 23 5 6 65 44 33 1 8 5 23 6 65 44 33 1 5 8 23 6 65 44 33
Крок 3	1 5 8 23 6 65 33 44 1 5 8 23 6 33 65 44 1 5 8 23 6 33 65 44 1 5 8 6 23 33 65 44 1 5 6 8 23 33 65 44
Крок 4	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Крок 5	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Крок 6	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Крок 7	1 5 6 8 23 33 44 65

Для методу простого обмінного сортування потрібне число порівнянь `n*(n-1)/2`, мінімальне число пересилань `0`, а середнє й максимальне число пересилань - $O(n^2)$. Можна внести в алгоритм кілька покращень.

При перегляді масиву зверху вниз елементи, які переміщаються угору, зрушуються лише на одну позицію. Ті ж елементи, які переміщаються вниз, зрушуються на кілька позицій за один прохід. Цей факт можна використовувати для прискорення роботи алгоритму бульбашкового сортування. Якщо чергувати перегляд масиву зверху вниз і знизу угору, то переміщення елементів у прямому й зворотному напрямках буде однаково швидким.

Під час проходів зверху вниз найбільший елемент списку переміщається на місце, а під час проходів знизу угору – найменший. Якщо `M` елементів списку розташовані не на своїх місцях, алгоритму буде потрібно не більш ніж `M` проходів для того, щоб розташувати елементи впорядковано. Якщо в списку `N` елементів, алгоритму буде потрібно `N` кроків для кожного проходу. Таким чином, повний час виконання для цього алгоритму буде порядку `O(M * N)`.

Останнє поліпшення – обмеження проходів масиву. Після перегляду масиву, останні переставлені елементи позначають частину масиву, яка містить неупорядковані елементи. При проході зверху вниз, наприклад, найбільший елемент переміщається в кінцеве положення. Оскільки немає більших елементів, які потрібно було б помістити за ним, то можна почати черговий прохід знизу угору із цієї точки й на ній же закінчувати наступні проходи зверху вниз.

У такий же спосіб, після проходу знизу угору, можна зрушити позицію, з якої почнеться черговий прохід зверху вниз і будуть закінчуватися наступні проходи знизу угору.

Реалізація алгоритму бульбашкового сортування мовою Visual Basic використовує змінні ~~**min**~~ і ~~**max**~~ для позначення першого й останнього елементів списку, які перебувають не на своїх місцях. У міру того, як алгоритм повторює проходи за списком, ці змінні обновляються, вказуючи положення останньої перестановки.

```
Public Sub Bubblesort(List() As Long, Byval min As
  Long, Byval max As Long)
Dim last_swap As Long
Dim i As Long
Dim j As Long
Dim tmp As Long

  ' Повторювати до завершення.
  Do While min < max
      ' «Спливання».
      last_swap = min - 1
      ' Тобто For i = min + 1 To max.
      i = min + 1
      Do While i <= max
            ' Знайти «бульбашку».
            If List(i - 1) > List(i) Then
                  ' Знайти, куди її помістити.
                  tmp = List(i - 1)
                  j = i
                  Do
                        List(j - 1) = List(j)
                        j = j + 1
                        If j > max Then Exit Do
                  Loop While List(j) < tmp
                  List(j - 1) = tmp
                  last_swap = j - 1
                  i = j + 1
            Else
                  i = i + 1
            End If
```

```
        Loop
        ' Обновити змінну max.
        max = last_swap - 1

        ' «Занурення».
        last_swap = max + 1
        ' Тобто For i = max -1 To min Step -1
        i = max - 1
        Do While i >= min
                ' Знайти «бульбашку».
                If List(i + 1) < List(i) Then
                        ' Знайти, куди її помістити.
                        tmp = List(i + 1)
                        j = i
                        Do
                                List(j + 1) = List(j)
                                j = j - 1
                                If j < min Then Exit Do
                        Loop While List(j) > tmp
                        List(j + 1) = tmp
                        last_swap = j + 1
                        i = j - 1
                Else
                        i = i - 1
                End If
        Loop
        ' Обновити змінну min.
        Min = last_swap + 1
    Loop
End Sub
```

Приклад – в архіві PRIMER.ZIP (диск з прикладами, папка ProgR13). Для того щоб протестувати алгоритм бульбашкового сортування за допомогою програми `Sort`, поставте галочку в полі `Sorted` (Відсортовані) в області `Initial Ordering` (Первісний порядок). Уведіть число елементів у поле `#unsorted` (Число несортованих). Після натискання на кнопку `Go` (Почати), програма створює й сортує список, а потім переставляє випадково обрані пари елементів. Наприклад, якщо ви введете число 10 у поле `#unsorted`, програма переставить 5 пар чисел, тобто 10 елементів виявляться не на своїх місцях.

Для другого варіанта первісного алгоритму програма зберігає елемент у тимчасовій змінній при переміщенні на кілька кроків. Це відбувається ще швидше, якщо використовувати функцію API **`Memcopy`.** Алгоритм бульбашкового сортування в програмі

Fastsort, використовуючи функцію **Memcopy,** сортує елементи в 50 або 75 раз швидше, ніж первісна версія, реалізована в програмі **Sort.**

Незважаючи на те, що бульбашкове сортування повільніше, ніж інші алгоритми, у нього є свої застосування. Бульбашкове сортування часто дає найкращі результати, якщо список спочатку вже майже впорядкований. Якщо програма управляє списком, який сортується при створенні, а потім до нього додаються нові елементи, бульбашкове сортування може бути кращим вибором.

Швидке сортування

Швидке сортування (`quicksort`) – рекурсивний алгоритм, який використовує підхід «розділяй і пануй». Якщо список, що сортується, більше заданого мінімального розміру, процедура швидкого сортування розбиває його на два, а потім рекурсивно викликає себе для сортування кожного із підсписків.

Якщо алгоритм викликається для підсписку, що містить не більш одного елемента, то підсписок вже відсортований, і підпрограма завершує роботу.

Інакше процедура вибирає який-небудь елемент зі списку й використовує його для розбивки списку на два підсписки. Вона поміщає елементи, які менше, ніж обраний елемент, в перший підсписок, а інші – у другий і потім рекурсивно викликає себе для сортування двох підсписків.

Основна ідея алгоритму полягає в тому, що випадковим чином вибирається деякий елемент масиву `x`, після чого масив проглядається ліворуч, поки не зустрінеться елемент `a[i]` такий, що `a[i] > x,` а потім масив проглядається праворуч, поки не зустрінеться елемент `a[j]` такий, що `a[j] < x.` Ці два елементи міняються місцями, і процес перегляду, порівняння й обміну триває, поки ми не дійдемо до елемента `x.` У результаті масив виявиться розбитим на дві частини - ліву, у якій значення ключів будуть менше x, і праву зі значеннями ключів, більшими x. Далі процес рекурсивно триває для лівої й правої частин масиву доти, поки кожна частина не буде містити в точності один елемент. Зрозуміло, що як зазвичай рекурсію можна замінити ітераціями, якщо запам'ятовувати

відповідні індекси масиву. Простежимо цей процес на прикладі нашого стандартного масиву (табл.13.4).

Таблиця 13.4

Приклад швидкого сортування

Початковий стан масиву	**8 23 5 65 44 33 1 6**

Крок 1 (у якості x вибирається a[5])	`        |--------|`
	`8 23 5 6 44 33 1 65`
	`          |---|`
	`8 23 5 6 1 33 44 65`

Крок 2 (у підмасиві a[1], a[5] у якості x вибирається a[3])	`8 23 |5| 6 1 33 44 65`
	`|--------|`
	`1 23 5 6 8 33 44 65`
	`  |--|`
	`1 5 23 6 8 33 44 65`

Крок 3 (у підмасиві a[3], a[5] у якості x вибирається a[4])	`1 5 23 |6| 8 33 44 65`
	`    |----|`
	`1 5 8 6 23 33 44 65`

Крок 4 (у підмасиві a[3], a[4] вибирається a[4])	`1 5 8 |6| 23 33 44 65`
	`    |--|`
	`1 5 6 8 23 33 44 65`

Алгоритм недарма називається швидким сортуванням, оскільки для нього оцінкою числа порівнянь і обмінів є `O(n*log n)`. Насправді у більшості утиліт, що виконують сортування масивів, використовується саме цей алгоритм.

```
Public Sub Quicksort(List() As Long, Byval min as
  Integer, _
  Byval max As Integer)
Dim med_value As Long
Dim hi As Integer
Dim lo As Integer

    ' Якщо залишилося менше 1 елемента, підсписок
відсортований.
  If min >= max Then Exit Sub

  ' Вибрати значення для розподілу списку.
  med_value = list(min)
  lo = min
```

```
  hi = max
  Do
       Перегляд від hi до значення < med_value.
      Do While list(hi) >= med_value
            hi = hi - 1
            If hi <= lo Then Exit Do
      Loop
      If hi <= lo Then
            list(lo) = med_value
            Exit Do
      End If
      ' Поміняти місцями значення lo і hi.
      list(lo) = list(hi)

      ' Перегляд від lo до значення >= med_value.
      lo = lo + 1
      Do While list(lo) < med_values
            lo = lo + 1
            If lo >= hi Then Exit Do
      Loop
      If lo >= hi Then
            lo = hi
            list(hi) = med_value
            Exit Do
      End If
      ' Поміняти місцями значення lo і hi.
      list(hi) = list(lo)
  Loop

  ' Рекурсивне сортування двох підлистів.
  Quicksort list(), min, lo - 1
  Quicksort list(), lo + 1, max
End Sub
```

Є кілька важливих моментів у цій версії алгоритму, які варто згадати. *По - перше,* значення **med_value** для розподілу списку не входить ні в один підсписок. Це означає, що у двох підсписках утримується на один елемент менше, ніж у похідному списку. Оскільки число розглянутих елементів зменшується, то в остаточному підсумку алгоритм завершить роботу.

Ця версія алгоритму використовує як роздільник перший елемент у списку. В ідеалі це значення мало б перебувати десь в середині списку, так щоб два підсписки були приблизно рівного розміру. Проте, якщо елементи спочатку майже відсортовані, то перший елемент – найменший у списку. При цьому алгоритм не

помістить жодного елемента в перший підсписок, а всі елементи в другий.

У цьому випадку кожний виклик підпрограми вимагає порядку O(N) кроків для переміщення всіх елементів у другий підсписок. Оскільки алгоритм рекурсивно викликає себе N – 1 раз, час його виконання буде порядку $O(N^2)$, що не краще, ніж у раніш розглянутих алгоритмах. Ситуацію ще більше погіршує те, що рівень вкладеності рекурсії алгоритму N – 1. Для великих списків величезна глибина рекурсії приведе до переповнення стека й збою в роботі програми.

Існує багато стратегій вибору розділового елемента. Можна використовувати елемент із середини списку. Це може виявитися непоганим вибором, проте може виявитися й так, що це буде найменший або найбільший елемент списку. При цьому один підсписок буде набагато більше, ніж інший, що призведе до зниження продуктивності до порядку $O(N^2)$ і глибокого рівня рекурсії.

Інша стратегія може полягати в тому, щоб переглянути весь список, обчислити середнє арифметичне всіх значень і використовувати його як розділове значення. Цей підхід буде давати непогані результати, але зажадає додаткових зусиль. Додатковий прохід зі складністю порядку O(N) не змінить теоретичний час виконання алгоритму, але знизить загальну продуктивність.

Третя стратегія – вибрати середній з елементів на початку, кінці й середині списку. Перевага цього підходу у швидкості, тому що буде потрібно вибрати лише три елементи. Гарантується, що цей елемент не є найбільшим або найменшим у списку і, ймовірно, виявиться десь в середині списку.

Нарешті, *остання стратегія,* яка використовується в програмі **Sort,** полягає у випадковому виборі елемента зі списку. Можливо, це буде непоганим вибором. Навіть якщо це не так, можливо, на наступному кроці алгоритм зробить кращий вибір. Імовірність постійного випадання найгіршого випадку дуже мала.

При використанні інших методів вибору точки поділу існує невелика ймовірність того, що при певній організації списку час сортування буде порядку $O(N^2)$, хоча малоймовірно, що подібна організація списку на початку сортування зустрінеться насправді, проте, час виконання при цьому буде $O(N^2)$, неважливо чому. Це те, що можна назвати «невеликою ймовірністю того, що завжди буде погана продуктивність».

Сортування злиттям

Як і швидке сортування, *сортування злиттям* (mergesort) – це рекурсивний алгоритм. Він також розділяє список на два підсписки і рекурсивно сортує підсписки.

Сортування злиттям ділить список навпіл, формуючи два підсписки однакового розміру. Потім підсписки рекурсивно сортуються і уже відсортовані зливаються, утворюючи повністю відсортований список.

Один з популярних алгоритмів внутрішнього сортування зі злиттями заснований на таких ідеях (для простоти будемо вважати, що число елементів у масиві, як і в нашому прикладі, є ступенем числа 2). Спочатку пояснимо, що таке злиття. Нехай є два відсортовані в порядку зростання масиви p[1], p[2], ..., p[n] і q[1], q[2], ..., q[n] і порожній масив r[1], r[2], ..., r[2*n], який ми прагнемо заповнити значеннями масивів p і q у порядку зростання. Для злиття виконуються такі дії: порівнюються p[1] і q[1], і менше зі значень записується в r[1]. Припустимо, що це значення p[1]. Тоді p[2] порівняється з q[1] і менше зі значень заноситься в r[2]. Припустимо, що це значення q[1]. Тоді на наступному кроці порівнюються значення p[2] і q[2] і т.д., поки ми не досягнемо межі одного з масивів. Тоді залишок іншого масиву просто дописується в "хвіст" масиву r. Приклад злиття двох масивів показаний на рис.13.1.

Для сортування зі злиттям масиву a[1], a[2], ..., a[n] заводиться парний масив b[1], b[2], ..., b[n]. На першому кроці проводиться злиття a[1] і a[n] з розміщенням результату в b[1], b[2], злиття a[2] і a[n-1] з розміщенням результату в b[3], b[4], ..., злиття a[n/2] і a[n/2+1] із розміщенням результату в b[n-1], b[n]. На другому кроці проводиться злиття пар b[1], b[2] і b[n-1], b[n] із розміщенням результату в a[1], a[2], a[3], a[4], злиття пар b[3], b[4] і b[n-3], b[n-2] із розміщенням результату в a[5], a[6], a[7], a[8], ..., злиття пар b[n/2-1], b[n/2] і b[n/2+1], b[n/2+2] із розміщенням результату в a[n-3], a[n-2], a[n-1], a[n]. і т.д. На останньому кроці, наприклад (залежно від значення n), проводиться злиття послідовностей елементів масиву довжиною n/2 a[1], a[2], ..., a[n/2] і a[n/2+1], a[n/2+2], ..., a[n] із розміщенням результату

в b[1], b[2], ..., b[n]. Для випадку масиву, використовуваного в наших прикладах, послідовність кроків показана в табл. 13.5.

Таблиця 13.5

Послідовність кроків сортування масиву

Початковий стан масиву	**8 23 5 65 44 33 1 6**
Крок 1	6 8 1 23 5 33 44 65
Крок 2	6 8 44 65 1 5 23 33
Крок 3	1 5 6 8 23 33 44 65

При застосуванні сортування злиттям число порівнянь ключів і число пересилань оцінюється як O(n*log n). Але слід ураховувати, що для виконання алгоритму для сортування масиву розміру n потрібно 2*n елементів пам'яті. Хоча етап злиття легко зрозуміти, це найцікавіша частина алгоритму. Підписки зливаються в тимчасовий масив, і результат копіюється в первісний список. Створення тимчасового масиву може бути недоліком, особливо якщо розмір елементів великий. Якщо розмір тимчасового розміру дуже великий, він може приводити до звертання до файлу підкачування й значно знижувати продуктивність. Робота з тимчасовим масивом також приводить до того, що більша частина часу йде на копіювання елементів між масивами.

Так само, як і у випадку зі швидким сортуванням, можна прискорити виконання сортування злиттям, зупинивши рекурсію, коли підписки досягають певного мінімального розміру. Потім можна використовувати сортування вибором для завершення роботи.

```
Public Sub Mergesort(List() As Long, Scratch() As Long,
  Byval min As Long, Byval max As Long)
Dim middle As Long
Dim i1 As Long
Dim i2 As Long
Dim i3 As Long
```

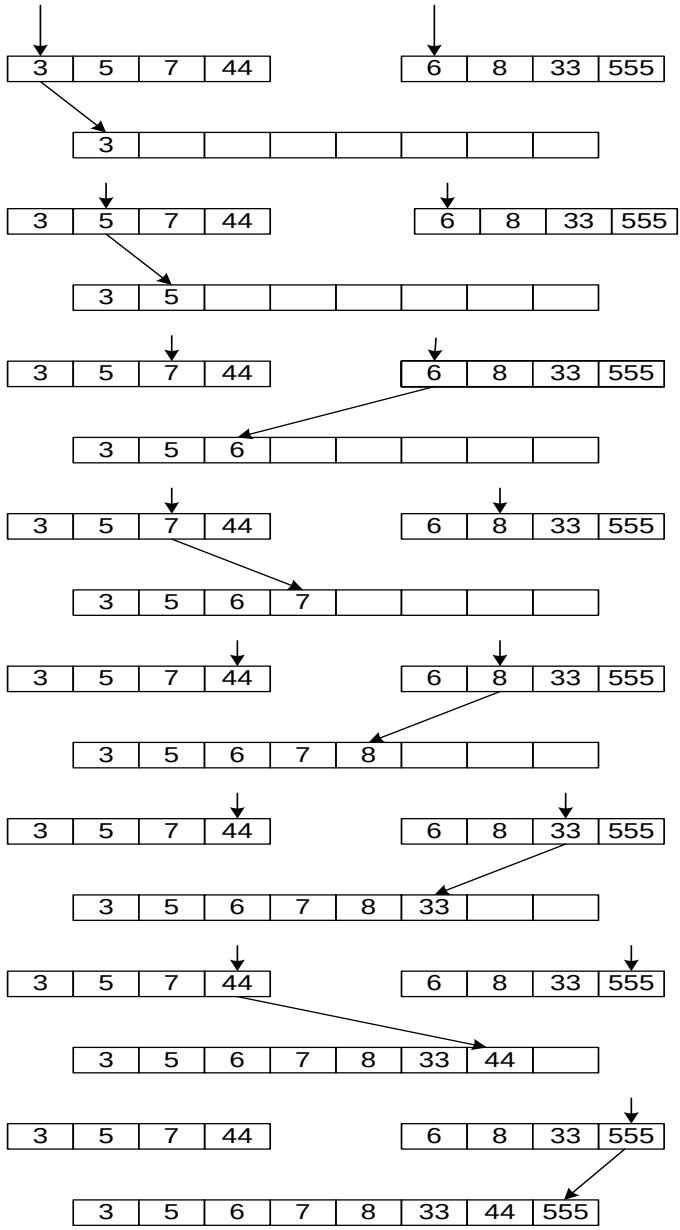


Рис.13.1. Приклад сортування злиттям

```
  ' Якщо в списку більше, ніж Cutoff елементів,
  ' завершити його сортування процедурою Selectionsort.
  If max - min < Cutoff Then
       Selectionsort List(), min, max
       Exit Sub
  End If

  ' Рекурсивне сортування підсписків.
  middle = max \ 2 + min \ 2
  Mergesort List(), Scratch(), min, middle
  Mergesort List(), Scratch(), middle + 1, max

  ' Злити відсортовані списки.
  i1 = min                ' Індекс списку 1.
  i2 = middle + 1   ' Індекс списку 2.
  i3 = min                ' Індекс об'єднаного списку.
  Do While i1 <= middle And i2 <= max
       If List(i1) <= List(i2) Then
              Scratch(i3) = List(i1)
              i1 = i1 + 1
       Else
              Scratch(i3) = List(i2)
              i2 = i2 + 1
       end If
       i3 = i3 + 1
  Loop

  ' Очищення непустого списку.
  Do While i1 <= middle
       Scratch(i3) = List(i1)
       i1 = i1 + 1
       i3 = i3 + 1
  Loop
  Do While i2 <= max
       Scratch(i3) = List(i2)
       i2 = i2 + 1
       i3 = i3 + 1
  Loop

  ' Помістити відсортований список на місце вихідного.
  For i3 = min To max
       List(i3) = Scratch(i3)
  Next i3
End Sub
```

Перевага сортування злиттям у тому, що час його виконання залишається однаковим незалежно від різних розподілів і

початкового розташування даних. Швидке ж сортування дає продуктивність порядку $O(N^2)$ і досягає глибокого рівня вкладеності рекурсії, якщо список містить багато однакових значень. Якщо список великий, швидке сортування може переповнити стек і привести до аварійного завершення роботи програми. Сортування злиттям ніколи не досягає занадто глибокого рівня вкладеності рекурсії, тому що завжди ділить список на рівні частини. Для списку з N елементів глибина вкладеності рекурсії для сортування злиттям становить лише $\log(N)$.

Пірамідальне сортування

Почнемо із простого методу сортування за допомогою дерева, при використанні якого явно будується двійкове дерево порівняння ключів. Побудова дерева починається з листів, які містять усі елементи масиву. З кожної сусідньої пари вибирається найменший елемент, і ці елементи утворюють наступний (ближчий до кореня рівень дерева). З кожної сусідньої пари вибирається найменший елемент і т.д., поки не буде побудований корінь, що містить найменший елемент масиву. Двійкове дерево порівняння для масиву, використаного в наших прикладах, показано на рис.13.2. Отже, ми вже маємо найменше значення елементів масиву. Для того щоб одержати наступний за величиною елемент, спустимося від кореня по шляху, що веде до листа з найменшим значенням. У цій листовій вершині проставляється фіктивний ключ із "нескінченно великим" значенням, а в усі проміжні вузли, що займалися найменшим елементом, заноситься найменше значення з вузлів - безпосередніх нащадків (рис.13.3.). Процес триває доти, поки всі вузли дерева не будуть заповнені фіктивними ключами (рис.13.4 - рис.13.9).

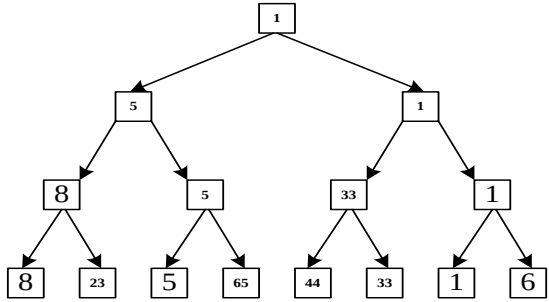


Рис. 13.2. Двійкове дерево порівняння похідного масиву

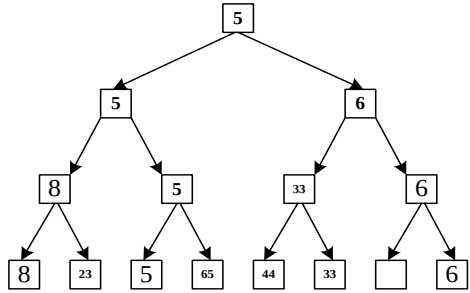


Рис.13.3. Другий крок

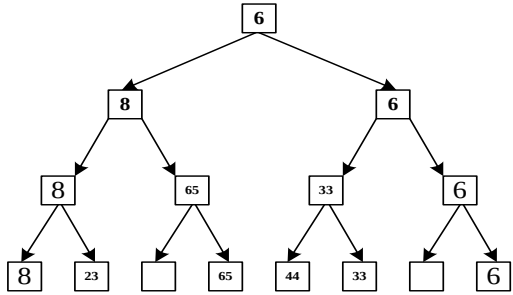


Рис.13.4. Третій крок

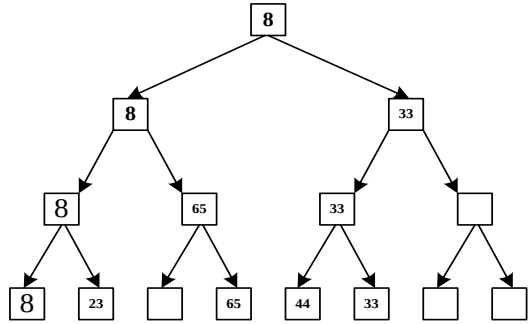


Рис. 13.5. Четвертий крок

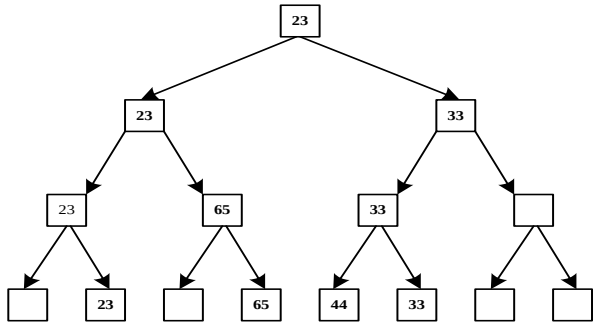


Рис. 13.6. П'ятий крок

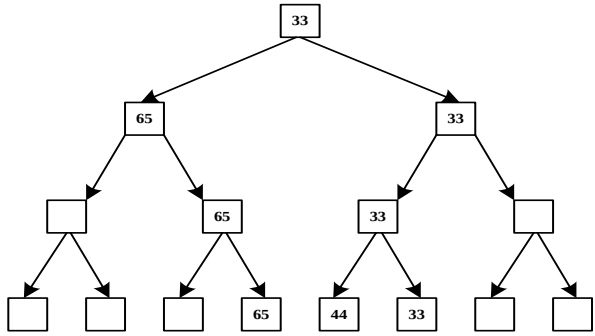


Рис. 13.7. Шостий крок

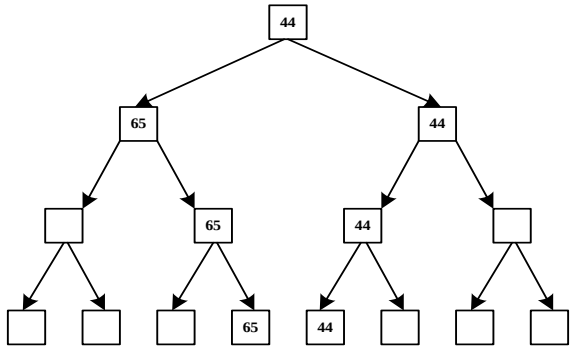


Рис. 13.8. Сьомий крок

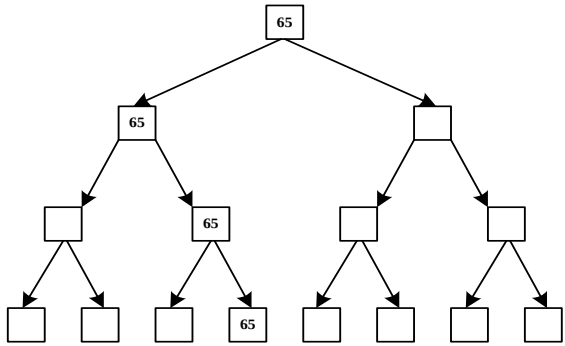


Рис. 13.9. Восьмий крок

На кожному з `n` кроків, необхідних для сортування масиву, потрібно `log n` (двійковий) порівнянь. Отже, потрібно буде `n*log n` порівнянь, але для зображення дерева знадобиться `2*n - 1` додаткових одиниць пам'яті.

Є більш досконалий алгоритм, який прийнято називати пірамідальним сортуванням (`Heapsort`). Його ідея полягає в тому, що замість повного дерева порівняння похідний масив `a[1], a[2], ..., a[n]` перетворюється в піраміду, що має таку властивість - для кожного `a[i]` виконуються умови `a[i] <= a[2*i]` і `a[i] <= a[2*i+1]`. Потім піраміда використовується для сортування.

Найбільш наочно метод побудови піраміди виглядає при деревоподібному зображенні масиву, показаному на рис.13.10. Масив

зображується у виді двійкового дерева, корінь якого відповідає елементу масиву a[1]. На другому ярусі перебувають елементи a[2] і a[3]. На третьому - a[4], a[5], a[6], a[7] і т.д. Як бачимо, для масиву з непарною кількістю елементів відповідне дерево буде збалансованим, а для масиву з парною кількістю елементів n елемент a[n] буде єдиним (найлівішим) листом "майже" збалансованого дерева.

Очевидно, що при побудові піраміди нас будуть цікавити елементи a[n/2], a[n/2-1], ..., a[1] для масивів з парним числом елементів і елементи a[(n-1)/2], a[(n-1)/2-1], ..., a[1] для масивів з непарним числом

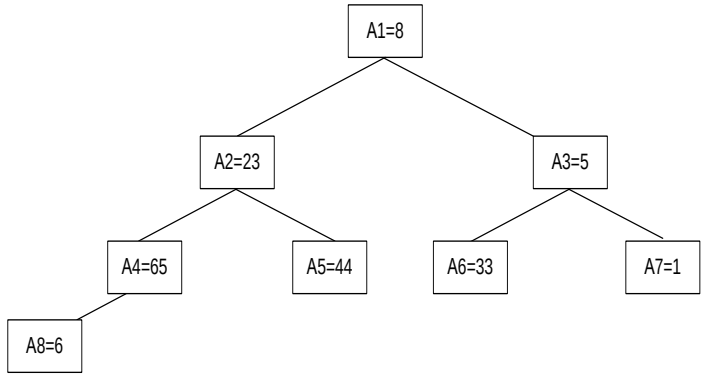


Рис. 13.10. Деревоподібне зображення похідного масиву

елементів (оскільки тільки для таких елементів істотні обмеження піраміди). Нехай i - найбільший індекс із числа індексів елементів, для яких істотні обмеження піраміди. Тоді береться елемент a[i] у побудованому дереві й для нього виконується процедура просівання, що полягає в тому, що вибирається гілка дерева, що відповідає min(a[2*i], a[2*i+1]), і значення a[i] міняється місцями зі значенням відповідного елемента. Якщо цей елемент не є листом дерева, для нього виконується аналогічна процедура і т.д. Такі дії виконуються послідовно для a[i], a[i-1], ..., a[1]. Легко бачити, що в результаті ми одержимо деревоподібний вигляд піраміди для вихідного масиву (послідовність кроків для використовуваного в наших прикладах масиву показано на рис.13.11-рис.13.14).

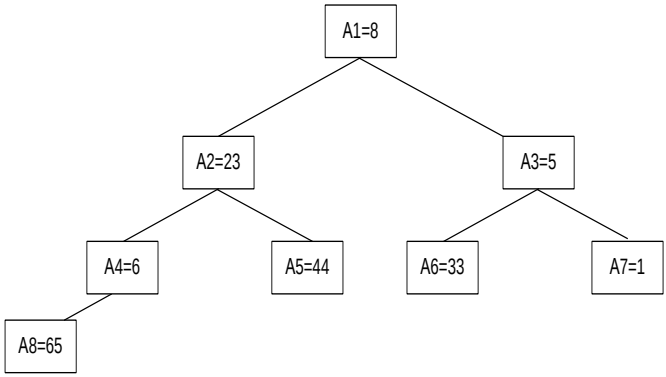


Рис.13.11.

Крок 1

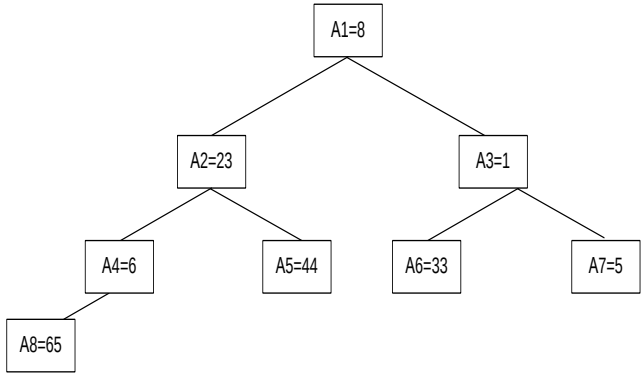


Рис. 13.12. Крок 2

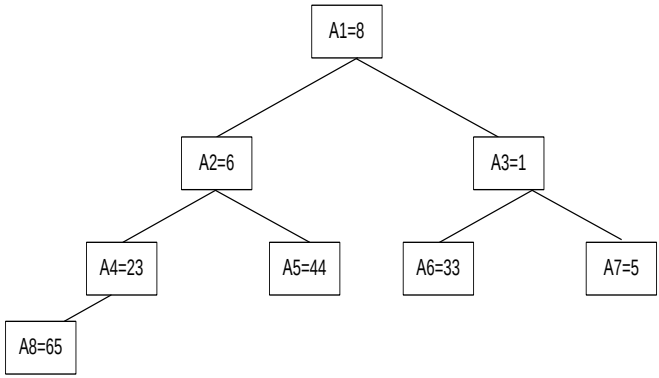


Рис. 13.13. Крок 3

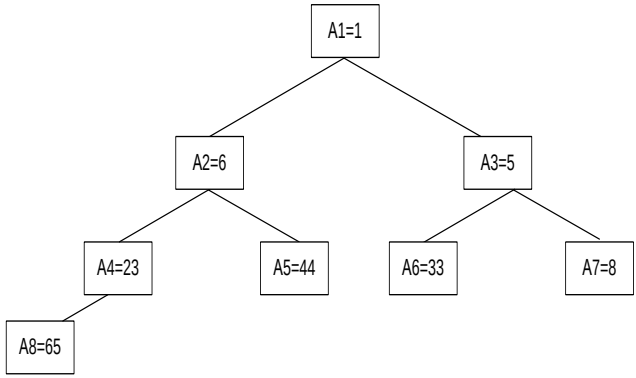


Рис. 13.14. Крок 4

Сортування підрахунком

Сортування підрахунком (countingsort) – спеціалізований алгоритм, який дуже добре працює, якщо елементи даних – цілі числа, значення яких перебувають у відносно вузькому діапазоні. Цей алгоритм працює досить швидко, наприклад, якщо значення перебувають між 1 і 1000.

Сортування підрахунком починається зі створення масиву для підрахунку числа елементів, що мають певне значення. Якщо значення перебувають у діапазоні між **min_value** і **max_value**, алгоритм створює масив **Counts** з нижньою межею **min_value** і верхньою межею **max_value**. Якщо використовується масив з попереднього проходу, необхідно значення його елементів прирівняти до нуля . Якщо існує M значень елементів, масив містить M записів, і час виконання цього кроку порядку O(M) .

```
For i = min To max
  Counts(List(i)) = Counts(List(i)) + 1
Next i
```

Зрештою алгоритм обходить масив **Counts**, поміщаючи відповідне число елементів у відсортований масив. Для кожного значення **i** між **min_value** і **max_value**, він поміщає **Counts(i)** елементів зі значенням **i** у масив. Оскільки цей крок поміщає по

одному запису в кожну позицію в масиві, він вимагає порядку O(N) кроків.

```
new_index = min
For i = min_value To max_value
  For j = 1 To Counts(i)
       sorted_list(new_index) = i
       new_index = new_index + 1
  Next j
Next i
```

Алгоритм цілком вимагає порядку O(M)+O(N)+O(N)=O(M+N) кроків. Якщо M мале в порівнянні з N, він виконується дуже швидко. Наприклад, якщо M<N, то O(M+N)=O(N), що досить швидко. Якщо N=100.000 і M=1000, то M+N=101.000, тоді як N*log(N)=1,6 мільйона. Кроки, виконувані алгоритмом сортування підрахунком, також досить прості в порівнянні із кроками швидкого сортування. Усі ці факти поєднуються, забезпечуючи разом неймовірно високу швидкість виконання сортування підрахунком.

З іншого боку, якщо M більше, ніж O(N*log(N)), то O(M+N) буде більше, ніж O(N*log(N)). У цьому випадку сортування підрахунком може виявитися повільнішим, ніж алгоритми зі складністю порядку O(N*log(N)), такі як швидке сортування. В одному з тестів швидке сортування 1000 елементів зі значеннями від 1 до 500.000 зажадало 0,054 сек., у той час як сортування підрахунком зажадало 1,76 секунд.

Сортування підрахунком спирається на той факт, що значення даних – цілі числа, тому цей алгоритм не може просто сортувати дані інших типів. В Visual Basic не можна створити масив із межами від AAA до ZZZ.

Блокове сортування

Як і сортування підрахунком, *блокове сортування* (bucketsort) не використовує операцій порівняння елементів. Цей алгоритм використовує значення елементів для розбивки їх на блоки і потім рекурсивно сортує отримані блоки. Коли блоки стають досить

малими, алгоритм зупиняється й використовує більш простий алгоритм типу сортування вибором для завершення процесу.

За змістом цей алгоритм подібний до швидкого сортування. Швидке сортування розділяє елементи на два підсписки й рекурсивно сортує їх. Блокове сортування робить те ж саме, але ділить список на безліч блоків, а не на лише два підсписки.

Для розподілу списку на блоки алгоритм припускає, що значення даних розподілені рівномірно, і розподіляє елементи по блоках рівномірно. Наприклад, припустимо, що дані мають значення в діапазоні від 1 до 100 і алгоритм використовує 10 блоків. Алгоритм поміщає елементи зі значеннями 1- 10 у перший блок, зі значеннями 11- 20 – у другий і т.д.

Якщо елементи розподілені рівномірно, у кожний блок потрапляє приблизно однакове число елементів. Якщо в списку N елементів, і алгоритм використовує N блоків, у кожний блок потрапляє лише один або два елементи. Програма може відсортувати їх за кінцеве число кроків, тому час виконання алгоритму в цілому порядку O(N) .

На практиці розподіл даних звичайно не є рівномірним. У деякі блоки потрапляє більше елементів, в інші - менше. Проте якщо розподіл у цілому близький до рівномірного, то в кожному із блоків виявиться лише невелике число елементів.

Проблеми можуть виникати, тільки якщо список містить невелике число різних значень. Наприклад, якщо всі елементи мають одне й те ж значення, вони всі будуть поміщені в один блок. Якщо алгоритм не виявить це, він знову й знову буде поміщати всі елементи в той самий блок, викликавши нескінченну рекурсію й вичерпавши весь стековий простір.

Блокове сортування із застосуванням зв'язного списку

Реалізувати алгоритм блокового сортування на Visual Basic можна різними способами. *По-перше,* можна використовувати в якості блоків зв'язні списки. Це полегшує переміщення елементів між блоками в процесі роботи алгоритму.

Цей метод може бути більш складним, якщо елементи з самого початку розташовані в масиві. У цьому випадку необхідно переміщати елементи з масиву у зв'язний список і назад у масив після завершення сортування. Для створення зв'язного списку також

потрібна додаткова пам'ять. Наступний код демонструє алгоритм блокового сортування із застосуванням зв'язних списків:

```
Public Sub Linkbucketsort(Listtop As Listcell)
Dim count As Long
Dim min_value As Long
Dim max_value As Long
Dim Value As Long
Dim item As Listcell
Dim nxt As Listcell
Dim bucket() As New Listcell
Dim value_scale As Double
Dim bucket.num As Long
Dim i As Long

  Set item = Listtop.Nextcell
  If item Is Nothing Then Exit Sub

  ' Підрахувати елементи й знайти значення min і max.
  count = 1
  min_value = item.Value
  max_value = min_value
  Set item = item.Nextcell
  Do While Not (item Is Nothing)
       count = count + 1
       Value = item.Value
       If min_value > Value Then min_value = Value
       If max_value < Value Then max_value = Value
       Set item = item.Nextcell
  Loop

  ' Якщо min_value = max_value, виходить, є єдине
  ' значення, і список відсортований.
  If min_value = max_value Then Exit Sub

  ' Якщо в списку не більш, ніж Cutoff елементів,
  ' завершити сортування процедурою Linkinsertionsort.
  If count <= Cutoff Then
       Linkinsertionsort Listtop
       Exit Sub
  End If

  ' Створити порожні блоки.
  Redim bucket(1 To count)

  value_scale = _
  Cdbl(count - 1) / _
```

```
  Cdbl(max_value - min_value)

  ' Розмістити елементи в блоках.
  Set item = Listtop.Nextcell
  Do While Not (item Is Nothing)
       Set nxt = item.Nextcell
       Value = item.Value
       If Value = max_value Then
             bucket_num = count
       Else
             bucket_num = _
                   Int((Value - min_value) * _
                         value_scale) + 1
       End If
       Set item.Nextcell = bucket (bucket_num).Nextcell
       Set bucket(bucket_num).Nextcell = item
       Set item = nxt
  Loop

  ' Рекурсивне сортування блоків, що містять
  ' більш одного елемента.
  For i = 1 To count
       If Not (bucket(i).Nextcell Is Nothing) Then _
             Linkbucketsort bucket(i)
  Next i

  ' Об'єднати відсортовані списки.
  Set Listtop.Nextcell = bucket(count).Nextcell
  For i = count - 1 To 1 Step -1
       Set item = bucket(i).Nextcell
       If Not (item Is Nothing) Then
             Do While Not (item.Nextcell Is Nothing)
                   Set item = item.Nextcell
             Loop
             Set item.Nextcell = Listtop.Nextcell
             Set Listtop.Nextcell= bucket(i).Nextcell
       End If
  Next i
End Sub
```

Ця версія блокового сортування набагато швидша, ніж сортування вставкою з використанням зв'язних списків. Для більш довгих списків різниця буде ще більшою, тому що продуктивність сортування вставкою порядку $O(N^2)$.

Блокове сортування на основі масиву

Блокове сортування також можна реалізувати в масиві, використовуючи ідеї подібні тим, які використовуються при сортуванні підрахунком. При кожному виклику алгоритму спочатку підраховується число елементів, які відносяться до кожного блоку. Потім на основі цих даних розраховуються зсуви в тимчасовому масиві, які потім використовуються для правильного розташування елементів у масиві. Зрештою, блоки рекурсивно сортуються, і відсортовані дані переміщаються назад у вихідний масив.

```
Public Sub Arraybucketsort(List() As Long, Scratch() As
  Long, _
  min As Long, max As Long, Numbuckets As Long)
Dim counts() As Long
Dim offsets() As Long

Dim i As Long
Dim Value As Long
Dim min_value As Long
Dim max_value As Long
Dim value_scale As Double
Dim bucket_num As Long
Dim next_spot As Long
Dim num_in_bucket As Long

  ' Якщо в списку не більш ніж Cutoff елементів,
  ' закінчити сортування процедурою Selectionsort.
  If max - min + 1 < Cutoff Then
      Selectionsort List(), min, max
      Exit Sub
  End If

  ' Знайти значення min і max.
  min_value = List(min)
  max_value = min_value
  For i = min + 1 To max
      Value = List(i)
      If min_value > Value Then min_value = Value
      If max_value < Value Then max_value = Value
  Next i

  ' Якщо min_value = max_value, виходить, є єдине
  ' значення, і список відсортований.
  If min_value = max_value Then Exit Sub
```

```
' Створити порожній масив з відрахунками блоків.
Redim counts(l To Numbuckets)

value_scale = _
    Cdbl (Numbuckets - 1) / _
    Cdbl (max_value - min_value)

' Створити відрахунки блоків.
For i = min To max
    If List(i) = max_value Then
          bucket_num = Numbuckets
    Else
          bucket_num = _
                Int((List(i) - min_value) * _
                      value_scale) + 1
    End If
    counts(bucket_num) = counts(bucket_num) + 1
Next i

' Перетворити відрахунки в зсув у масиві.
Redim offsets(l To Numbuckets)
next_spot = min
For i = 1 To Numbuckets
    offsets(i) = next_spot
    next_spot = next_spot + counts(i)
Next i

' Розмістити значення у відповідних блоках.
For i = min To max
    If List(i) = max_value Then
          bucket_num = Numbuckets
    Else
          bucket_num = _
                Int((List(i) - min_value) * _
                      value_scale) + 1
    End If
    Scratch (offsets (bucket_num)) = List(i)
    offsets(bucket_num) = offsets(bucket_num) + 1
Next i

' Рекурсивне сортування блоків, що містять
' більш одного елемента.
next_spot = min
For i = 1 To Numbuckets
    If counts(i) > 1 Then Arraybucketsort _
          Scratch(), List(), next_spot, _
          next_spot + counts(i) - 1, counts(i)
    next_spot = next_spot + counts(i)
```

```
  Next i

  ' Скопіювати тимчасовий масив назад у вихідний
  список.
  For i = min To max
      List(i) = Scratch(i)
  Next i
End Sub
```

Через накладні витрати, необхідні для роботи зі зв'язними списками, ця версія блокового сортування працює набагато швидше, ніж версія з використанням зв'язних списків. Проте використовуючи методи роботи із псевдовказівниками, можна поліпшити продуктивність версії з використанням зв'язних списків, так що обидві версії стануть практично еквівалентними по швидкості.

Нову версію також можна зробити ще швидшою, використовуючи функцію `API` **`Memcopy`** для копіювання елементів з тимчасового масиву назад у похідний список. Цю вдосконалену версію алгоритму демонструє програма `Fastsort` (диск з прикладами - папка ProgR13).

Переваги й недоліки розглянутих алгоритмів сортування, подані у таблиці.

Таблиця 13.6

Переваги й недоліки алгоритмів сортування

Алгоритм	**Переваги**	**Недоліки**
Сортування вставкою	Дуже простий Швидко сортує невеликі списки	Дуже повільно працює з великими списками
Сортування вставкою на основі зв'язного списку	Простий Швидко сортує невеликі списки Переміщає не дані, а покажчики	Повільно працює з великими списками
Сортування вибором	Дуже простий Швидко сортує невеликі списки	Повільно працює з великими списками
Бульбашкове сортування	Швидко працює для майже відсортованих списків	Повільно працює у всіх інших випадках

Продовження табл. 13.6

Алгоритм	Переваги	Недоліки
Швидке сортування	Швидко сортує великі списки	Приводить до проблем при великій кількості однакових значень
Сортування злиттям	Швидко сортує великі списки	Вимагає простору під тимчасові значення Працює повільніше, ніж швидке сортування
Пірамідальне сортування	Швидко сортує великі списки Не вимагає простору для тимчасових значень	Працює повільніше, ніж сортування злиттям
Сортування підрахунком	Дуже швидко працює, якщо розкид вхідних значень невеликий	Працює повільно, якщо діапазон значень > log(N) Вимагає додаткової пам'яті Працює тільки з даними цілого типу
Блокове сортування	Дуже швидко працює, якщо дані розподілені рівномірно Працює з даними, діапазон значень яких великий Працює з даними будь-якого типу	Повільніше, ніж сортування підрахунком

Резюме

Можна вивести декілька правил, які допоможуть вибрати алгоритм сортування з максимальною продуктивністю:

- якщо вам потрібно швидко реалізувати алгоритм сортування, використовуйте швидке сортування, а потім при необхідності поміняйте алгоритм;
- якщо більш 99 відсотків списку вже відсортовано, використовуйте бульбашкове сортування;
- якщо список дуже малий (100 або менш елементів), використовуйте сортування вибором;

- якщо значення перебувають у зв'язному списку, використовуйте блокове сортування на основі зв'язного списку;
- якщо елементи в списку - цілі числа, розкид значень яких невеликий (до декількох тисяч), використовуйте сортування підрахунком;
- якщо значення лежать у широкому діапазоні й не є цілими числами, використовуйте блокове сортування на основі масиву;
- якщо ви не можете витрачати додаткову пам'ять, необхідну для блокового сортування, використовуйте швидку сортування.

Якщо ви знаєте структуру даних і різні алгоритми сортування, ви можете вибрати алгоритм, що найбільше задовольняє ваші потреби.

Контрольні запитання та завдання

1. Дати визначення процесу сортування.
2. Наведіть найбільш відомі елементарні методи сортування.
3. Поясніть, чому сортування є завданням із точними теоретичними обмеженнями продуктивності.
4. Яку оцінку ефективності мають прості методи сортування?
5. Для чого використовують таблиці індексів?
6. У чому перевага методу сортування вибором?
7. Яку складність має алгоритм сортування вставкою?
8. Порівняйте бульбашкове сортування та швидке сортування.
9. Дати визначення поняттю піраміда у пірамідальному сортуванні.
10. Назвіть головні етапи алгоритму пірамідального сортування?
11. Як виконується побудова піраміди в алгоритмі пірамідального сортування?
12. Яку оцінку ефективності має алгоритм пірамідального сортування?
13. Що таке сортування підрахунком?
14. Чим відрізняються блокове сортування та блокове сортування із застосування зв’язного списку?
15. В яких випадках прості алгоритми сортування можуть бути більш прийнятними, ніж вдосконалені?

16. Відсортувати файл цілих чисел методом вставки, не використовуючи масив.
17. Реалізувати програмно-пірамідальне сортування.
18. До підготовленої статті автор додав список використаної літератури, але розташував видання в порядку появи посилань на них в тексті. Редактор зажадав розташувати джерела за абеткою. Упорядкувати список літератури на вимогу редактора (кожне видання - з нового абзацу: номер п/п, автор, назва роботи і т.д.).
19. Декілька арифметичних прогресій задано своїми параметрами: $a^{(i)}$ –перший член ; $d^{(i)}$ - різниця; $n^{(i)}$ –кількість членів ; $d^{(i)}>0$, i=1., m. Не знаходячи самих прогресій, провести їхнє злиття в один масив, що не убуває.
20. Мажоруючим елементом у масиві A [1..N] будемо називати елемент, що зустрічається в масиві більш ніж N/2 разів. Легко помітити, що в масиві може бути не більше одного мажоруючого елемента. Наприклад, у масиві {3, 3, 4, 2, 4, 4, 2, 4, 4} мажоруючий елемент 4, тоді як у масиві {3, 3, 4, 2, 4, 4, 2, 4} мажоруючого елемента немає. Визначити, чи є в упорядкованому масиві мажоруючий елемент, і якщо є, то надрукувати його.

Розділ XIV
ПОШУК

Після того як список елементів відсортований, може знадобитися знайти певний елемент у списку. В цьому розділі ми розглянемо декілька алгоритмів для *пошуку* (`search`) елементів у впорядкованих списках.

Приклади програм

Програма **Search**(диск з прикладами – папка ProgR14) в архіві із прикладами демонструє всі описані в розділі алгоритми. Введіть значення елементів, які має містити список, і потім натисніть на кнопку Make List (Створити список), програма створить список на основі масиву, у якому кожний елемент більше попереднього на число від 0 до 5. Програма виводить значення найбільшого елемента в списку, щоб ви уявляли діапазон значень елементів.

Після створення списку, установивши відповідні прапорці, виберіть алгоритми, які ви хочете використовувати. Потім уведіть значення, яке потрібно знайти, й натисніть на кнопку Search (Пошук).Програма виконає пошук елемента за допомогою обраного вами алгоритму. Оскільки список містить не всі можливі елементи в заданому діапазоні значень, вам може знадобитися ввести кілька різних значень, перш ніж одне з них найдеться в списку.

Програма дозволяє також задати число повторень для кожного з алгоритмів пошуку. Деякі алгоритми виконуються дуже швидко, тому для того, щоб порівняти їхню швидкість, може знадобиться задати для них велику кількість повторень.

На рис. 14.1 показано вікно програми **Search** після пошуку елемента зі значенням 250.000. Цей елемент перебував у позиції 99.802 списку із 100000 елементів.

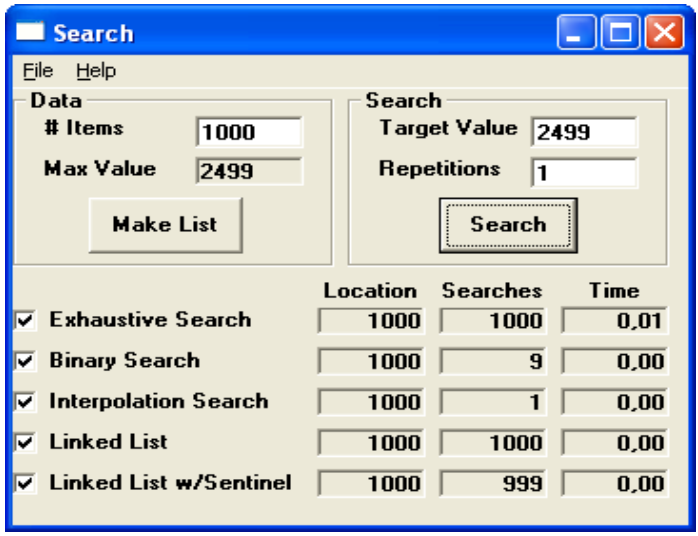


Рис. 14.1. Вікно програми Search

Пошук методом повного перебору

При виконанні *лінійного* (`linear`) пошуку або пошуку методом *повного перебору* (`exhaustive search`) пошук ведеться з початку списку, і елементи перебираються послідовно, доки серед них не буде знайдено шуканий.

```
Public Function LinearSearch(target As Long) As Long
Dim i As Long

  For i = 1 To NumItems
      If List(i) >= target Then Exit For
  Next i

  If i > NumItems Then
      Search = 0                ' Елемент не знайдено.
  Else
      Search = i                ' Елемент знайдено.
  End If
End Function
```

Оскільки цей алгоритм перевіряє елементи послідовно, то він знаходить елементи на початку списку швидше, ніж елементи, розташовані наприкінці. Найгірший випадок для цього алгоритму виникає, коли елемент перебуває наприкінці списку або взагалі не присутній у ньому. У цих випадках алгоритм перевіряє всі елементи в списку, тому час його виконання у найгіршому випадку порядку `O(N)`.

Якщо елемент перебуває в списку, то в середньому алгоритм перевіряє `N/2` елементів до того, як виявить шуканий. Таким чином, в усередненому випадку час виконання алгоритму також порядку `O(N)`.

Хоча алгоритми, які виконуються за час порядку `O(N)`, не є дуже швидкими, цей алгоритм досить простий та дає на практиці непогані результати. Для невеликих списків цей алгоритм має прийнятну продуктивність.

Пошук в упорядкованих списках

Якщо список упорядкований, то можна злегка модифікувати алгоритм повного перебору, щоб підвищити його продуктивність. У цьому випадку, якщо під час виконання пошуку алгоритм знаходить елемент зі значенням, більшим, ніж значення шуканого елемента, то він завершує свою роботу. При цьому шуканий елемент не перебуває в списку, тому що інакше він би зустрівся раніше.

Наприклад, припустимо, що ми шукаємо значення 12 і дійшли до значення 17. При цьому ми вже пройшли ту ділянку списку, у якій міг би перебуває елемент зі значенням 12, виходить, елемент 12 у списку відсутній. Наступний код демонструє дороблену версію алгоритму пошуку повним перебором:

```
Public Function LinearSearch(target As Long) As Long
Dim i As Long

  NumSearches = 0

  For i = 1 To NumItems
       NumSearches = NumSearches + 1
       If List(i) >= target Then Exit For
  Next i

  If i > NumItems Then
```

```
      LinearSearch = 0          ' Елемент не знайдено.
  ElseIf List(i) <> target Then
      LinearSearch = 0          ' Елемент не знайдено.
  Else
      LinearSearch = i          ' Елемент знайдено.
  End If
End Function
```

Ця модифікація зменшує час виконання алгоритму, якщо елемент відсутній у списку. Попередній версії пошуку було потрібно перевірити весь список до кінця, але шуканого елемента в ньому не буде. Нова версія зупиниться, як тільки виявить елемент більший, ніж шуканий.

Якщо шуканий елемент розташований випадково між найбільшим і найменшим елементами в списку, то в середньому алгоритму знадобиться порядку `O(N)` кроків, щоб визначити, що шуканий елемент відсутній у списку. Час виконання при цьому має той же порядок, але на практиці його продуктивність буде трохи вищою. Програма **`Search`**`(диск з прикладами – папка ProgR14)` на диску із прикладами використовує цю версію алгоритму.

Пошук у зв'язних списках

Пошук методом повного перебору — це єдиний спосіб пошуку у зв'язних списках. Оскільки доступ до елементів можливий тільки за допомогою покажчиків `NextCell` на наступний елемент, то необхідно перевірити по черзі всі елементи з початку списку, щоб знайти шуканий.

Так само, як і у випадку пошуку, повним перебором у масиві, якщо список упорядкований, можна припинити пошук, якщо найдеться елемент зі значенням, більшим, ніж значення шуканого елемента.

```
Public Function LListSearch(target As Long) As
  SearchCell
Dim cell As SearchCell

  NumSearches = 0

  Set cell = ListTop.NextCell
  Do While Not (cell Is Nothing)
```

```
      NumSearches = NumSearches + 1

      If cell.Value >= target Then Exit Do
      Set cell = cell.NextCell
  Loop

  If Not (cell Is Nothing) Then
      If cell.Value = target Then
            Set LListSearch = cell     ' Елемент
  знайдений.
      End If
  End If
End Function
```

Програма **Search** в архіві із прикладами використовує цей алгоритм для пошуку елементів у зв'язному списку. Цей алгоритм виконується небагато повільніше, ніж алгоритм повного перебору в масиві через додаткові накладні витрати, пов'язані з управлінням покажчиками на об'єкти. Помітьте, що програма **Search** будує зв'язні списки, тільки якщо список містить не більше 10.000 елементів.

Щоб алгоритм виконувався небагато швидше, у нього можна внести ще одну зміну. Якщо зберігати покажчик на кінець списку, то можна додати в кінець списку комірку, що буде містити шуканий елемент. Цей елемент називається *сигнальною міткою* (`sentinel`). Це дозволяє обробляти особливий випадок кінця списку так само, як і всі інші.

У нашому випадку додавання мітки в кінець списку гарантує, що зрештою шуканий елемент буде знайдено. При цьому програма не може вийти за кінець списку, і немає необхідності перевіряти умови `Not(cell Is Nothing)` у кожному циклі `While`.

```
Public Function SentinelSearch(target As Long) As
  SearchCell
Dim cell As SearchCell
Dim sentinel As New SearchCell

  NumSearches = 0

  ' Установити сигнальну мітку.
  sentinel.Value = target
  Set ListBottom.NextCell = sentinel

  ' Знайти шуканий елемент.
  Set cell = ListTop.NextCell
```

```
  Do While cell.Value < target
       NumSearches = NumSearches + 1
       Set cell = cell.NextCell
  Loop

  ' чи Визначити знайдений шуканий елемент.
  If Not ((cell Is sentinel) Or _
       (cell.Value <> target)) _
  Then
       Set SentinelSearch = cell        ' Елемент
  знайдено.
  End If

  ' Видалити сигнальну мітку.
  Set ListBottom.NextCell = Nothing
End Function
```

Хоча може здатися, що ця зміна незначна, перевірка `Not (cell Is Nothing)` виконується в циклі, що викликається дуже часто. Для великих списків цей цикл викликається безліч разів, і виграш часу підсумовується. В Visual Basic ця версія алгоритму пошуку у зв'язних списках виконується на 20 відсотків швидше, ніж попередня. У програмі **Search** наведені обидві версії цього алгоритму і ви можете порівняти їх.

Деякі алгоритми використовують потоки для прискорення пошуку у зв'язних списках. Наприклад, за допомогою покажчиків в комірках списку можна організувати список у виді двійкового дерева. Пошук елемента з використанням цього дерева забере час порядку `O(log(N))`, якщо дерево збалансоване. Такі структури даних не є просто списками, тому ми їх не розглядаємо.

Двійковий пошук

Як уже згадувалося раніше, пошук повним перебором виконується дуже швидко для малих списків, але для великих набагато швидше виконується двійковий пошук. Алгоритм двійкового пошуку (`binary search`) порівнює елемент у середині списку із шуканим. Якщо шуканий елемент менше, ніж знайдений, то алгоритм продовжує пошук у першій половині списку, якщо більше — у другій половині. Приклад пошуку зображений на рис.14. 2.

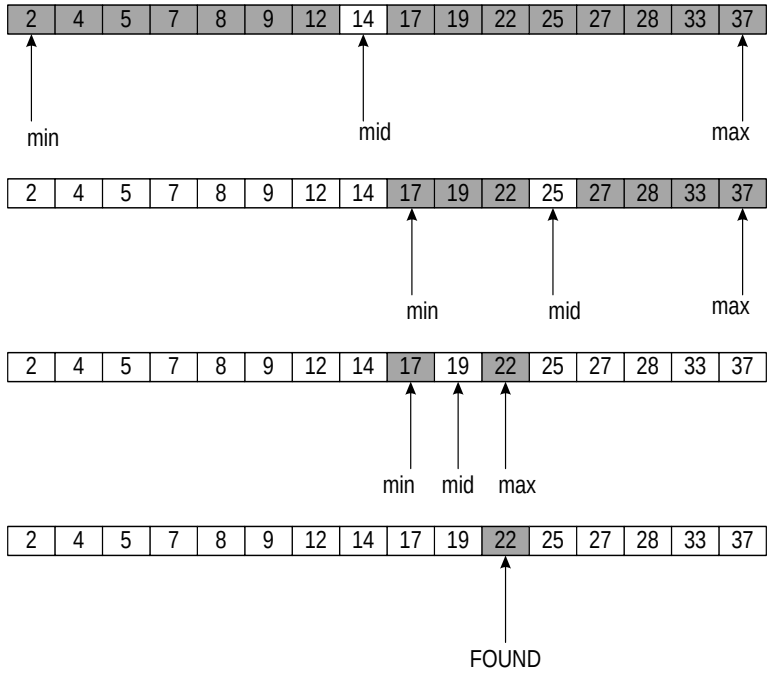


Рис. 14.2. Приклад бінарного пошуку

Хоча за своєю природою цей алгоритм є рекурсивним, його досить просто записати й без застосування рекурсії. Оскільки цей алгоритм простий для розуміння в будь-якому варіанті (з рекурсією або без), то ми наводимо тут його нерекурсивну версію, що містить менше викликів функцій.

Основна ідея, укладена в цьому алгоритмі, проста, але деталі її реалізації досить складні. Програмі доводиться акуратно відслідковувати частину масиву, що може містити шуканий елемент, інакше вона може його пропустити.

Алгоритм використовує дві змінні: `min` і `max`, у яких перебувають мінімальний і максимальний індекси комірок масиву, які можуть містити шуканий елемент. Під час виконання алгоритму індекс шуканої комірки завжди лежить між `min` і `max`. Інакше кажучи, `min <= target index <= max`.

Під час кожного проходу алгоритм виконує присвоєння `middle = (min + max) / 2` і перевіряє комірку, індекс якої дорівнює `middle`. Якщо його значення дорівнює шуканому, то мета знайдена і алгоритм завершує свою роботу.

Якщо значення шуканого елемента менше, ніж значення середнього, то алгоритм установлює значення змінної `max` рівним `middle – 1` і продовжує пошук. Оскільки індекси елементів, які можуть містити шуканий елемент, тепер перебувають у діапазоні від `min` до (`middle – 1`), програма виконує пошук у першій половині списку.

Зрештою програма або знайде шуканий елемент, або наступить момент, коли значення змінної `min` стане більше, ніж значення `max`. Оскільки індекс шуканого елемента має перебувати між мінімальним й максимальним можливими індексами, це означає, що шуканий елемент відсутній у списку.

Наступний код демонструє виконання двійкового пошуку в програмі **`Search`**:

```
Public Function BinarySearch(target As Long) As Long
Dim min As Long
Dim max As Long
Dim middle As Long

  NumSearches = 0

  ' Під час пошуку індекс шуканого елемента
  перебуватиме
  ' між Min і Max: Min <= target index <= Max
  min = 1
  max = NumItems
  Do While min <= max
       NumSearches = NumSearches + 1

       middle = (max + min) / 2
       If target = List(middle) Then    ' Ми знайшли
  шуканий елемент!
             BinarySearch = middle
             Exit Function
       ElseIf target < List(middle) Then ' Пошук у лівій
  половині.
             max = middle - 1
       Else                ' Пошук у правій половині.
             min = middle + 1
       End If
  Loop

  ' Якщо ми виявилися тут, то шуканого елемента немає в
  списку.
  BinarySearch = 0
End Function
```

На кожному кроці число елементів, які ще можуть мати шукане значення, зменшується вдвічі. Для списку розміру `N` алгоритму може знадобитися максимум `O(log(N))` кроків, щоб знайти будь-який елемент або визначити, що його немає в списку. Це набагато швидше, ніж у випадку застосування алгоритму повного перебору. Повний перебір списку з мільйона елементів зажадав би в середньому 500.000 кроків. Алгоритму двійкового пошуку буде потрібно не більше, ніж `log(1.000.000)` або 20 кроків.

Інтерполяційний пошук

Двійковий пошук забезпечує значне збільшення швидкості пошуку в порівнянні з повним перебором, оскільки він виключає великі частини списку, не перевіряючи при цьому значення елементів, що виключаються. Якщо, крім того, відомо, що значення елементів розподілені досить рівномірно, то можна виключати на кожному кроці ще більше елементів, використовуючи інтерполяційний пошук (`interpolation search`).

Інтерполяцією називається процес прогнозування невідомих значень на основі наявних. У цьому випадку індекси відомих значень у списку використовуються для визначення можливого положення шуканого елемента в списку.

Наприклад, припустимо, що є список значень, показаний на рис.14.2. Цей список містить 16 елементів зі значеннями між 2 і 37. Припустимо тепер, що потрібно знайти елемент у списку, що має значення 22. Значення 22 становить 63 відсотки відстані між 2 і 37 на шкалі чисел. Якщо вважати, що значення елементів розподілені рівномірно, то можна припустити, що шуканий елемент розташований приблизно в точці, що становить 63 відсотки від розміру списку, і займає позицію 10.

Якщо позиція, обрана за допомогою інтерполяції, виявляється неправильною, то алгоритм порівнює шукане значення зі значенням елемента в обраній позиції. Якщо шукане значення менше, то пошук триває в першій частині списку, якщо більше — у другій частині.

При двійковому пошуку список послідовно розбивається посередині на дві частини. Інтерполяційний пошук щораз розбиває список, намагаючись знайти найближчий до шуканого елемент у списку, при цьому точка розбивки визначається таким кодом:

```
middle = min + (target - List(min)) * _
     ((max - min) / (List(max) - List(min)))
```

Цей оператор поміщає значення `middle` між `min` і `max` у такому ж співвідношенні, у якому шукане значення перебуває між `List(min)` і `List(max)`. Якщо шуканий елемент перебуває поряд із `List(min)`, то різниця `target - List(min)` майже дорівнює нулю. Тоді все співвідношення цілком виглядає майже як `middle = min + 0`, тому значення змінної `middle` майже дорівнює `min`. Зміст цього полягає в тому, що якщо індекс елемента майже дорівнює `min`, то його значення майже дорівнює `List(min)`.

Аналогічно, якщо шуканий елемент перебуває поряд із `List(max)`, то різниця `target - List(min)` майже дорівнює різниці `List(max) - List(min)`. Їхня частка майже дорівнює одиниці, і співвідношення виглядає майже як `middle = min + (max - min)`, або `middle = max`, якщо спростити вираз. Зміст цього співвідношення полягає в тому, що якщо значення елемента близьке до `List(max)`, то його індекс майже дорівнює `max`.

Після того, як програма обчислить значення `middle`, вона порівнює значення елемента в цій позиції із шуканим так само, як і в алгоритмі двійкового пошуку. Якщо ці значення збігаються, то шуканий елемент знайдено і процес закінчено. Якщо значення шуканого елемента менше, ніж значення знайденого, то програма встановлює значення `max` рівним `middle - 1` і продовжує пошук елементів списку з меншими значеннями. Якщо значення шуканого елемента більше, ніж значення знайденого, то програма встановлює значення `min` рівним `middle + 1` і продовжує пошук елементів списку з більшими значеннями.

Помітьте, що в знаменнику співвідношення, що знаходить нове значення змінної `middle`, перебуває різниця (`List(max) - Lsit(min)`). Якщо значення `List(max)` і `List(min)` однакові, то відбудеться розподіл на нуль і програма аварійно завершить роботу. Таке може відбутися, якщо два елементи в списку мають однакові значення. Оскільки алгоритм підтримує співвідношення `min <= target index <= max`, то ця проблема може також виникнути, якщо `min` буде рости, а `max` зменшуватися доти, поки їхні значення не зрівняються.

Щоб упоратися із цією проблемою, програма перед виконанням операції розподілу перевіряє, чи `не співпадають List(max) і List(min)`. Якщо це так, значить, залишилося

перевірити тільки одне значення. При цьому програма просто перевіряє, чи збігається воно із шуканим.

Ще одна тонкість полягає в тому, що обчислене значення `middle` не завжди лежить між `min` і `max`. У найпростішому випадку це може бути так, якщо значення шуканого елемента виходить за межі діапазону значень елементів у списку. Припустимо, що ми намагаємося знайти значення 300 у списку з елементів 100, 150 і 200. На першому кроці обчислень `min = 1` і `max = 3`. Тоді `middle = 1 + (300 – List(1)) * (3 – 1) / (List(3) – List(1)) = 1 + (300 – 100) * 2 / (200 – 100) = 5`. Індекс 5 не тільки не перебуває в діапазоні між `min` і `max`, він також виходить за межі масиву. Якщо програма спробує звернутися до елемента масиву `List(5)`, то вона аварійно завершить роботу з повідомленням про помилку "`Subscript out of range`".

Подібна проблема виникає, якщо значення елементів розподілені між `min` і `max` дуже нерівномірно. Припустимо, що ми хочемо знайти значення 100 у списку 0, 1, 2, 199, 200. При першому обчисленні значення змінної `middle` ми одержимо в програмі `middle = 1 + (100 – 0) * (5 – 1) / (200 – 0) = 3`. Потім програма порівнює значення елемента `List(3)` із шуканим значенням 100. Оскільки `List(3) = 2`, що менше 100, вона задає `min = middle + 1`, тобто `min = 4`.

При подальшому обчисленні значення змінної `middle` програма знаходить `middle = 4 + (100 – 199) * (5 – 4) / (200 – 199) = –98`. Значення –98 не потрапляє в діапазон `min <= target index <= max` і також далеко виходить за межі масиву.

Якщо розглянути процес обчислення змінної `middle`, то можна побачити, що існують два варіанти, при яких нове значення може виявитися меншим, ніж `min` або більшим, ніж `max`. Спочатку припустимо, що `middle` менше, ніж `min`.

```
min + (target - List(min)) * ((max - min) / (List(max)
  - List(min))) < min
```

Після вирахування `min` з обох частин рівняння, одержимо:

```
(target - List(min)) * ((max - min) / (List(max) -
  List(min))) < 0
```

Оскільки `max >= min`, різниця `(max – min)` має бути більше нуля. Оскільки `List(max) >= List(min)`, різниця `(List(max) – List(min))` також має бути більше нуля. Тоді обидва значення можуть бути менше нуля, тільки якщо `(target – List(min))` менше нуля. Це означає, що шукане значення менше, ніж значення елемента `List(min)`. У цьому випадку шуканий елемент не може перебувати в списку, тому що всі елементи списку зі значенням меншим, ніж `List(min)`, вже були виключені.

Тепер припустимо, що `middle` більше, ніж `max`.

```
min + (target - List(min)) * ((max - min) / (List(max)
  - List(min))) > max
```

Після вирахування `min` з обох частин рівняння, одержимо:

```
(target - List(min)) * ((max - min) / (List(max) -
  List(min))) > 0
```

Множення обох частин на `(List(max) – List(min)) / (max – min)` приводить співвідношення до виду:

```
target - List(min) > List(max) - List(min).
```

І, нарешті, додавши до обох частин `List(min)`, одержимо:

```
target > List(max).
```

Це означає, що шукане значення більше, ніж значення елемента `List(max)`. У цьому випадку шукане значення не може перебувати в списку, тому що всі елементи списку зі значеннями більшими, ніж `List(max)` вже були виключені.

З огляду на всі ці результати одержуємо, що нове значення змінної `middle` може вийти з діапазону між `min` і `max` тільки в тому випадку, якщо шукане значення виходить за межі діапазону від `List(min)` до `List(max)`. Алгоритм може використовувати цей факт при обчисленні нового значення змінної `middle`. Він спочатку перевіряє, чи перебуває нове значення між `min` і `max`. Якщо ні, то шуканого елемента немає в списку й робота алгоритму завершена.

Наступний код демонструє реалізацію інтерполяційного пошуку в програмі **`Search`**:

```
Public Function InterpSearch(target As Long) As Long
Dim min As Long
Dim max As Long
Dim middle As Long

  min = 1
  max = NumItems
  Do While min <= max
       ' Уникаємо розподілу на нуль.
       If List(min) = List(max) Then
              ' Це шуканий елемент (якщо він є в
  списку).
              If List(min) = target Then
                     InterpSearch = min
              Else
                     InterpSearch = 0
              End If
              Exit Function
       End If

       ' Знайти точку розбивки списку.
       middle = min + (target - List(min)) * _
              ((max - min) / (List(max) - List(min)))

       ' Перевірити, чи не вийшли ми за межі.
       If middle < min Or middle > max Then
              ' Шуканого елемента немає в списку.
              InterpSearch = 0
              Exit Function
       End If

       NumSearches = NumSearches + 1
       If target = List(middle) Then    ' Шуканий
  елемент знайдений.
              InterpSearch = middle
              Exit Function
       ElseIf target < List(middle) Then ' Пошук у лівій
  частині.
              max = middle - 1
       Else                 ' Пошук у правій частині.
              min = middle + 1
       End If
  Loop

  ' Якщо ми дійшли до цієї точки, то елемента немає в
  списку.
  InterpSearch = 0
End Function
```

Двійковий пошук виконується дуже швидко, а інтерполяційний ще швидше. В одному з тестів двійковий пошук зажадав в 7 разів більше часу для пошуку значень у списку з 100.000 елементів. Ця різниця могла б бути ще більше, якби дані перебували на диску або якому-небудь іншому повільному пристрої. Хоча при інтерполяційному пошуку на обчислення йде більше часу, ніж у випадку двійкового пошуку, завдяки меншому числу звертань до диска ми заощадили б набагато більше часу.

Строкові дані

Якщо дані в списку є рядками, можна застосувати два різних підходи. Більш простий полягає в застосуванні двійкового пошуку. При двійковому пошуку значення елементів порівнюються безпосередньо, тому цей метод може легко працювати з рядковими даними.

З іншого боку, інтерполяційний пошук використовує чисельні значення елементів даних для обчислення можливого положення шуканого елемента в списку. Якщо елементи являють собою рядки, то цей алгоритм не може безпосередньо використовувати значення даних для обчислення передбачуваного положення шуканого елемента.

Якщо рядки досить короткі, то можна закодувати їх за допомогою цілих чисел або чисел формату `long` або `double`. Після цього можна використовувати для знаходження елементів у списку інтерполяційний пошук.

Якщо рядки занадто довгі і їх не можна закодувати навіть числами у форматі `double`, то усе ще можна використовувати для інтерполяції значення рядків. Спочатку знайдемо перший символ, що відрізняється, для `рядків List(min) і List(max)`. Потім можна використовувати ці значення для виконання інтерполяційного пошуку.

Наприклад, припустимо, що ми шукаємо рядок `TARGET` у списку `TABULATE`, `TANTRUM`, `TARGET`, `TATTERED`, `TAXATION`. Якщо `min = 1` і `max = 5`, то перевіряються значення `TABULATE` і `THEATER`. Ці рядки відрізняються в другому символі, тому потрібно розглядати три символи, що починаються із другого. Це будуть символи `ABU` для `List(1)`, `AXA` для `List(5)` і `ARG` для шуканого рядка.

Ці значення кодуються числами 804, 1378 і 1222 відповідно. Підставляючи ці значення у формулу для змінної middle, одержимо:

```
middle = min + (target - List(min)) * ((max - min) /
  (List(max) - List(min)))= 1 + (1222 - 804) * ((5 - 1)
  / (1378 - 804))   = 2,91
```

Це приблизно дорівнює 3, тому наступне значення змінної middle дорівнює 3. Це положення рядка TARGET у списку, тому пошук при цьому закінчується.

Пошук, що стежить

Щоб почати двійковий пошук, що стежить (binary hunt and search) порівняємо шукане значення з попереднього пошуку з новим шуканим значенням. Якщо нове значення менше, почнемо спостереження вліво, якщо більше — вправо.

Для виконання спостереження вліво, встановимо значення змінних min і max,що дорівнює індексу, отриманому під час попереднього пошуку. Потім зменшимо значення min на одиницю та порівняємо шукане значення зі значенням елемента List(min). Якщо шукане значення менше, ніж значення List(min), встановимо max = min і min = min −2 і зробимо ще одну перевірку. Якщо шукане значення усе ще менше, установимо max = min і min = min −4, якщо це не допоможе, установимо max = min і min = min −8 і так далі. Продовжимо встановлювати значення змінної max рівним значенню змінної min і віднімати чергові ступені двійки зі значення змінної min доти, поки не найдеться значення min, для якого значення елемента List(min) буде менше шуканого значення.

Необхідно стежити за тим, щоб не вийти за межі масиву, якщо min менше, ніж нижня границя масиву. Якщо в якийсь момент це виявиться так, то min потрібно надати значення нижньої границі масиву. Якщо при цьому значення елемента List(min) усе ще більше шуканого, значить шуканого елемента немає в списку. Спостереження вправо виконується аналогічно. Спочатку значення змінних min і max встановлюються рівними значенню індексу, отриманого під час попереднього пошуку. Потім послідовно

встановлюється `min = max` і `max = max + 1`, `min = max` і `max = max + 2`, `min = max` і `max = max + 4` і так далі доти, поки в якійсь точці значення елемента масиву `List(max)` не стане більше шуканого. І знову необхідно стежити за тим, щоб не вийти за межу масиву.

Після завершення фази спостереження відомо, що індекс шуканого елемента перебуває між `min` і `max`. Після цього можна використовувати звичайний двійковий пошук для знаходження точного положення шуканого елемента.

Якщо новий шуканий елемент перебуває недалеко від попереднього, то алгоритм пошуку, що стежить, дуже швидко знайде значення `max` і `min`. Якщо новий й старий шукані елементи відстоять один від одного на `P` позицій, то буде потрібно порядку `log(P)` кроків для пошуку нових значень змінних `min` і `max`.

Припустимо, що ми почали звичайний двійковий пошук без фази спостереження. Тоді буде потрібно порядку `log(NumItems) - log(P)` кроків для того, щоб значення `min` і `max` були на відстані не більше, ніж `P` позицій один від одного. Це означає, що слідку вальний пошук буде швидше звичайного двійкового пошуку, якщо `log(P) < log(NumItems) - log(P)`. Додавши до обох частин рівняння `log(P)`, одержимо `2 * log(P) > log(NumItems)`. Якщо звести обидві частини рівняння в ступінь двійки, одержимо $2^{2*\log(P)} < 2^{\log(NumItems)}$ або $(2^{\log(P)})^2 <$ `NumItems`, або після спрощення $P^2 <$ `NumItems`.

Із цього співвідношення видно, що пошук, що стежить, буде виконуватися швидше, якщо відстань між послідовними шуканими елементами буде менше, ніж квадратний корінь із числа елементів у списку. Якщо ідучи один за одним шукані елементи розташовані далеко, то краще використовувати звичайний двійковий пошук.

Інтерполяційний пошук, що стежить

Використовуючи методи з попередніх розділів, можна виконати інтерполяційний пошук, що стежить (`interpolative hunt and search`). Спочатку, як і раніше, порівняємо шукане значення з попереднього пошуку з новим. Якщо нове шукане значення менше, почнемо спостереження вліво, якщо більше — вправо.

Для спостереження вліво будемо тепер використовувати інтерполяцію, щоб припустити, де може перебувати шукане значення

в діапазоні між попереднім значенням і значенням елемента `List(1)`. Але це буде просто інтерполяційний пошук, у якому `min = 1` і `max` дорівнює індексу, отриманому під час попереднього пошуку. Після першого кроку фаза спостереження закінчується й далі можна продовжити звичайний інтерполяційний пошук.

Аналогічно виконується спостереження вправо. Просто надамо `max = Numitems` і встановимо `min` рівним індексу, отриманому під час попереднього пошуку. Потім продовжимо звичайний інтерполяційний пошук.

Якщо значення даних розташовані майже рівномірно, то інтерполяційний пошук завжди вибирає значення, що перебуває поряд із шуканим на першому або наступному кроці. Це означає, що, починаючи з попереднього знайденого значення, не можна значно поліпшити цей алгоритм. На першому кроці, навіть без використання результату попереднього пошуку, інтерполяційний пошук, імовірно, вибере індекс, що перебуває досить близько від індексу шуканого елемента.

З іншого боку, використання попереднього значення може допомогти у випадку, якщо дані розподілені нерівномірно. Якщо відомо, що нове шукане значення перебуває близько до старого, інтерполяційний пошук, що починається з попереднього значення, обов'язково знайде елемент, що перебуває поряд із попередньо знайденим. Це означає, що використання як стартової точки попереднього знайденого значення може давати певну перевагу.

Результат попереднього пошуку також сильніше обмежує діапазон можливих положень нового елемента у порівнянні з діапазоном від 1 до NumItems, тому алгоритм може заощадити при цьому один або два кроки. Це особливо важливо, якщо список перебуває на диску або якомусь іншому повільному пристрої. Якщо зберігати результат попереднього пошуку в пам'яті, то можна, принаймні, зрівняти нове шукане значення з попереднім без звертання до диска.

Резюме

Якщо елементи перебувають у зв’язному списку, використовуйте пошук методом повного перебору. Якщо можна, використовуйте сигнальну мітку наприкінці списку для прискорення пошуку.

Якщо вам потрібно час від часу проводити пошук у списку, що містить десятки елементів, також використовуйте пошук методом

повного перебору. Алгоритм у цьому випадку буде простіше відгладжувати й підтримувати, ніж більш складні методи пошуку, і він буде давати прийнятні результати.

Для виконання пошуку у великих списках використовуйте інтерполяційний пошук. Якщо значення даних розподілені досить рівномірно, то інтерполяційний пошук забезпечить найкращу продуктивність. Якщо список перебуває на диску або якомусь іншому повільному пристрої, різниця у швидкості між інтерполяційним пошуком та іншими методами пошуку може бути досить великою.

При роботі з рядковими даними можна спробувати закодувати їх числами у форматі `integer, long` або `double`, при цьому для їхнього пошуку можна буде використовувати інтерполяційний метод. Якщо рядки занадто довгі й не поміщаються навіть у числа формату `double`, то простіше використовувати двійковий пошук. У абл.. 14.1 наведені переваги й недоліки для різних методів пошуку.

Використовуючи двійковий або інтерполяційний пошук, можна дуже швидко знаходити елементи навіть у дуже великих списках. Якщо значення даних розподілені рівномірно, то інтерполяційний пошук дозволяє лише за кілька кроків знайти елемент у списку, що містить мільйон елементів.

Проте у такий великий список важко вносити зміни. Вставка або видалення елемента з упорядкованого списку забере час порядку `O(N)`. Якщо елемент перебуває на початку списку, виконання цих операцій може зажадати дуже великої кількості часу, особливо якщо список перебуває на якомусь повільному пристрої.

Таблиця 14.1

Переваги й недоліки різних методів пошуку

Метод	**Переваги**	**Недоліки**
Повний перебір	Простота Висока швидкість для невеликих списків	Низька швидкість для великих списків
Двійковий пошук	Висока швидкість для великих списків Не залежить від розподілу даних Легко працює з рядковими даними	

Продовження таблиця 14.1

Метод	Переваги	Недоліки
Інтерполяційний	Дуже висока швидкість для великих списків	Дуже складний Дані мають бути розподілені рівномірно. Складно працювати з рядковими даними

Контрольні запитання та завдання

1. Поясніть алгоритм методу повного перебору. Як модифікувати алгоритм повного перебору, щоб підвищити його продуктивність?
2. Чим відрізняються пошуки в упорядкованих та зв'язних списках ?
3. В чому сенс двійкового пошуку? Назвіть переваги двійкового пошуку.
4. Що таке бінарне дерево пошуку?
5. Поясніть сенс інтерполяційного пошуку, коли доцільне його використання?
6. Коли переважніше використання дерева пошуку ніж сортованого масиву?
7. Що таке вироджене дерево пошуку і як воно може виникнути?
8. Елементами масиву a [1..n] є неспадними масивами [1..m] цілих чисел. Відомо, що існує число, яке входить у всі масиви a [i] (існує таке x, що для всякого i з [1 .. n] знайдеться j з [1 .. m], для якого a [i] [j] = x). Знайти одне з таких чисел x.
9. Написати підпрограму, яка в двовимірному масиві A(N, M) цілих чисел, такому, що для всіх I від 1 до N, J від 1 до M-1 виконується A (I, J)> A (I, J+1) і для всіх I від 1 до N-1 виконується A (I, M)> A (I +1, M), знаходить всі елементи A (I, J), рівні J + I, або встановлює, що таких елементів немає.

10. На прямій своїми кінцями задані N відрізків і точка X. Визначити, чи належить точка міжвідрізочному інтервалу. Якщо так, то вказати кінцеві точки цього інтервалу. Якщо ні, то знайти, якій кількості відрізків належить точка, яким саме відрізкам належить точка.
11. На прямій своїми кінцями задані N відрізків. Знайти точку, що належить максимальному числу відрізків.
12. Вводиться послідовність n натуральних чисел. Необхідно визначити найменше натуральне число, якого немає у послідовності.
13. На довгій перфострічці записані N попарно різних позитивних цілих чисел. Ваша ЕОМ може перемотувати стрічку на початок і зчитувати числа одне за одним. Внутрішня пам'ять машини може зберігати тільки кілька цілих чисел. Потрібно знайти найменше додатне ціле число, якого немає на стрічці. Опишіть алгоритм, який зробить це за невелику кількість перемоток стрічки.
14. На столі у двох стовпчиках лежать 64 золотих і 64 срібних монет відповідно. Як срібні, так і золоті монети складені у порядку убування мас. Маси всіх монет різні. Яку найменшу кількість зважувань необхідно провести для визначення шістдесят четвертої монети в порядку убування мас серед всіх 128 монет? За один раз можна зважувати дві монети і визначити, яка з них важче. Відповідь обґрунтувати.
15. Задано N наборів монет з різних країн. Набори впорядковані за зростанням маси монет. У і-му наборі a_i монет. Необхідно за мінімальне число зважувань на чашкових вагах визначити к-ту за масою монету серед усіх монет.

Ключові терміни

Boot loader - завантажувач однієї з декількох ОС, встановлених на деякому комп'ютері, керований спеціальним меню при вмиканні комп'ютера.

Bluetooth – інтерфейс для бездротового підключення (за допомогою радіозв'язку) до комп'ютера мобільних телефонів, органайзерів, навушників, плеєрів та інших видів пристроїв.

BluRay – диск – різновид компакт-дисків великої ємності (25 – 50 ГБайт).

CISC (Complicated Instruction Set Computer – комп'ютер з ускладненою системою команд) – історично перший підхід до комп'ютерної архітектури, сенс якого полягає в ускладненості в системі команд внаслідок реалізації в них складних за семантикою операцій, що реалізують типові дії, часто використовувані при програмуванні й при реалізації мов (наприклад, групове пересилання рядків).

COM (communication port, serial port, послідовний порт) – порт для підключення до комп'ютера різних комунікаційних пристроїв, наприклад, модемів.

DMA (Direct Memory Access) – контролери із прямим доступом до оперативної пам'яті, минаючи використання спеціалізованої пам'яті пристрою.

Double bootable system - комп'ютер, на якому встановлені дві (або більше) операційні системи, при включенні якого користувачеві видається початкове меню для уточнення, яку саме ОС потрібно запустити.

EPIC (Explicit Parallelism Instruction Computers – комп'ютери з явним розпаралелюванням команд) – підхід до архітектури комп'ютера, аналогічний **VLIW**, але з додаванням ряду вдосконалень, наприклад, спекулятивних обчислень – паралельного виконання обох гілок умовної конструкції з обчисленням умови.

EPP (Extended Parallel Port) – двонаправлений режим роботи порту **LPT**, у якому він може працювати не тільки для виведення, але й для введення інформації.

Hard real-time – система реального часу, у який при порушенні тимчасових обмежень може виникнути критична помилка (відмова) керованого нею об'єкта.

HDMI (High Definition Multimedia Interface) – інтерфейс і порт, що дозволяє підключити до комп'ютера телевізор або інше

відеоустаткування, що забезпечує найкращу якість відтворення (HD – High Definition).

IEEE 1394 (FireWire) – порт для підключення до комп'ютера цифрової відеокамери або фотоапарата.

LPT (від line printer), або **паралельний порт -** застарілий вид порта для підключення принтера або сканера з товстим у перерізі кабелем і великим рознімачем, що вимагає попереднього вимикання комп'ютера й пристрою для їхнього безпечного з'єднання.

Original Equipment Manufacturer (OEM) - фірма-розроблювач якогось зовнішнього пристрою, що звичайно розробляє й **драйвер** до нього.

PCI (Personal Computer Interface) – найбільш поширений тип системної шини, до якої приєднані процесор, пам'ять, диски, принтер, модем та інші зовнішні пристрої комп'ютерної системи.

RISC (Reduced Instruction Set Computer – комп'ютер зі спрощеною системою команд) – спрощений підхід до архітектури комп'ютерів, що характеризується такими принципами: спрощення семантики команд; відсутність складних групових операцій; однакова довжина команд (32 або 64 біта, за розміром машинного слова); виконання арифметичних операцій тільки в регістрах і використання спеціальних команд запису й зчитування регістрів пам'яті; відсутність спеціалізованих регістрів; використання великого набору регістрів загального призначення (реєстрового файлу); передача при виклику процедур параметрів через регістри.

RS-232 - інша (більш застаріла) назва порту **COM**.

SCSI (Small Computer System Interface) – інтерфейс, адаптери й порти для підключення широкого спектра зовнішніх пристроїв – жорстких дисків, сканерів і ін. з можливістю обслуговування **гірлянди пристроїв**, що мають різні **(SCSI** IDs) номери та підключені до одного SCSI-Порту.

SCSI ID – номер пристрою (від 0 до 9), що є частиною **гірлянди SCSI-Пристроїв,** підключених до одного SCSI-Порту.

Soft real-time – система реального часу, порушення тимчасових обмежень якої не приводить до відмови керованого нею об'єкта.

TV-Тюнер - пристрій для прийому телевізійного сигналу з антени й показу телевізійного зображення на комп'ютері.

USB (Universal Serial Bus) – найпоширеніший універсальний порт комп'ютера (з характерним плоским рознімачем, розміром порядку 1 см, із зображенням тризубця), до якого можуть підключатися клавіатура, миша, зовнішній диск, принтер, сканер та інші зовнішні пристрої.

VLIW (Very Long Instruction Word – комп'ютери із широким командним словом) – підхід до архітектури комп'ютерів, заснований на таких принципах: статичне планування паралельних обчислень компілятором на рівні окремих послідовностей команд і підкоманд; подання команди як "широкої" - утримуючої декілька підкоманд, виконуваних паралельно за той самий машинний такт на декількох однотипних пристроях процесора, наприклад, двох пристроях додавання й двох логічних пристроях.

Автоматизація управління *(Computer Aided Controlling)* — комплекс технічних, організаційних, економічних дій і заходів, реалізація яких забезпечує зниження або виключення особистої участі людини у здійсненні процесу управління. По суті це впровадження нової технології обробки інформації із застосуванням сучасних засобів обчислювальної техніки, створенням автоматизованої системи управління.

Автоматизована система (в інформаційних технологіях) *(Automated System (In Information Techology)* — система, що реалізує інформаційну технологію виконання встановлених функцій за допомогою персоналу та комплексу засобів автоматизації.

Адреса *(address)* — об'єкт, що визначає розміщення даних.

Асиметрична кластеризація (asymmetric clustering) – організація комп'ютерного кластера, коли один комп'ютер виконує додаток, а інші простоюють.

Асиметрична мультипроцесорна система (asymmetric multiprocessing) – багатопроцесорна комп'ютерна система, у якій процесори спеціалізовані за своїми функціями; є головний процесор, що планує роботу підлеглих процесорів.

Асинхронне введення-виведення – введення-виведення, виконуване паралельно з виконанням програми користувача.

Асоціативна пам'ять (кеш – cache) – ділянка пам'яті, розташована в більш швидкодійній системі пам'яті, яка зберігає елементи більш повільної пам'яті разом з їхніми адресами, що найбільш часто використовуються, з метою оптимізації звертань до них.

Багатоядерний комп'ютер (multi-core computer) – найпоширеніша в наш час (2010 р.) архітектура комп'ютерів, при якій кожний процесор має кілька ядер (cores), об'єднаних в одному кристалі й такі, що працюють на одній і тій же загальній пам'яті, що дає широкі можливості для паралельних обчислень.

Багатоцільові комп'ютери (комп'ютери загального призначення, mainframes) – традиційна історична назва для

комп'ютерів, розповсюджених в 1950-х – 1970-х рр., що використовувалися для вирішення будь-яких завдань.

Базовий регістр (base register) – системний регістр, використовуваний для захисту пам'яті та такий, що утримує початкову адресу ділянки пам'яті, виділену користувальницькій програмі.

Біт режиму – біт, що зберігається в системному регістрі й задає поточний режим виконання команд: дорівнює 0 для системного режиму й 1 – для користувальницького режиму.

Браузер *(browser)* — вікно перегляду — програма (в системах програмування з багатовіконним доступом), що дає змогу переглядати в групі виділених вікон текстові уявлення програм і даних.

Буфер (buffer) — область оперативної пам'яті, що тимчасово містить дані у процесі їхнього одержання, передавання, читання або запису. Використовується для згладжування швидкісних або часових характеристик пристроїв, застосовується у терміналах, периферійних пристроях, зовнішніх запам'ятовуючих пристроях і центральному процесорі.

Алгоритм *(algorithm)* — формальна процедура, що гарантує одержання оптимального або коректного розв'язку задачі.

Аналіз вимог *(requirementi analyses)* — дослідження вимог користувача для специфікації системи.

Аналіз даних *(data analyses)* — дослідження даних у реальній або проектованій системі.

Атрибут *(attribute)* — найменування властивостей одного або кількох об'єктів. Атрибут іменує властивість, а його значення визначає конкретну властивість.

Веб-Сервер (Web server) – комп'ютер і програмне забезпечення, що надає доступ клієнтам через WWW до Web-Сторінок, розташованих на комп'ютері-сервері.

Вектор переривань (interrupt vector) – резидентний масив в оперативній пам'яті, у якому зберігаються доступні по номерах переривань адреси підпрограм-оброблювачів переривань (модулів ОС).

Віртуальний *(virtual)* — концептуальний чи можливий, але не той, що реально існує. Вказує на те, що елементи даних, структури або обладнання подають програмісту або користувачеві у виді, який відрізняється від реального. Перетворення елементів із реальних на віртуальні виконується програмним забезпеченням.

Визначення задачі *(problem definition)* — формулювання задачі, яке може містити опис даних, а також метод, процедури й алгоритм її розв'язання.

Внутрішня схема *(internal schema)* — фізична структура даних.

Вторинна пам'ять *(secondary storage)* — пам'ять, що не є складовою частиною обчислювальної машини, але безпосередньо з нею пов'язана і керована нею (вінчестер, гнучкі диски, CD—ROM та ін.).

Віртуальний COM-Порт – уявлюваний COM-Порт (такий, щоу дійсності не існує і не має рознімача), що ОС наче інсталює в систему при установці, наприклад, драйвера для взаємодії через Bluetooth або через кабель комп'ютера з мобільним пристроєм. Зазвичай має великий номер, наприклад, 18.

Гібридний процесор – новий підхід до архітектури комп'ютерів, який дуже поширюється, при цьому підході процесор має гібридну структуру – складається з (багатоядерного) центрального процесора **(CPU)** і (також багатоядерного) графічного процесора (GPU – Graphical Processor Unit).

Гірлянда SCSI-Пристроїв – ланцюжок пристроїв, підключених до одного SCSI-**Порту, які** мають різні **SCSI IDs** (номери).

Дані *(data)* — інформація, подана у формалізованому виді, придатному для пересування, інтерпретації або оброблення за участі людини чи автоматичними засобами.

Двійковий пошук *(binary search)* — метод пошуку в упорядкованій таблиці або файлі. Процедура полягає у поділі розглядуваної ділянки навпіл та виборі її верхньої або нижньої частини. Вибір ґрунтується на аналізі значення ключа всередині ділянки. Вибрана частина потім знову поділяється навпіл і так триває доти, поки не буде знайдено потрібний елемент.

Дескриптор (**descriptor**) – адресне слово в системах з теговою архітектурою; містить тегдескриптор, **адресу** початку адресованого масиву в пам'яті, **довжину** масиву й 4 біти зашиті – **від читання, від запису, від виконання** й **від запису адресної інформації.**

Дерево цілей *(aim tree)* — модель у виді зв'язного графа, вершини якого інтерпретуються як елементи (цілі, підцілі, ресурси) дерева, а ребра графа — як зв'язки між ними. Особливістю такого дерева цілей є те, що одна й та сама вершина нижчого рівня ієрархії може бути одночасно підпорядкована двом або кільком вершинам вищого рівня. Дерево може бути побудоване у двох варіантах. У

першому — елементи дерева цілей розбиваються на цільові елементи такої самої природи (ціль — підцілі першого рівня, підцілі другого рівня і т. д.). У другому - дерево цілей будується за принципом "цілі — заходи — ресурси", включає як цілі, так і заходи щодо їхнього досягнення та потрібні ресурси.

Деревоподібна (ієрархічна) структура ***(tree (hierarchical) structure)*** — структура даних, які є множиною, частково впорядкованою так, що існує лише один елемент цієї множини, який не має попереднього елемента, а решта елементів мають лише один попередній елемент.

Діалоговий режим ***(interactive mode)*** — режим оперативної взаємодії користувача з обчислювальною системою.

Динамічний розподіл ***(dynamic resource allocation)*** — розподіл, за якого ресурси, призначені для виконання програм, визначаються критеріями, що застосовуються у потрібний момент.

Динамічний розподіл пам'яті ***(dynamic storage allocation)*** — надання пам'яті процедури згідно з оперативним запитом на противагу наданню пам'яті процедури на підставі наперед передбачуваного запиту.

Доріжка (track) – частина **жорсткого диска**, розташована між двома концентричними колами на одному із магнітних дисків, що його становлять.

Драйвер – низькорівнева системна програма для керування якимось зовнішнім пристроєм (наприклад, жорстким диском).

Жорсткий диск (hard disk) - різновид **зовнішньої пам'яті, що** фізично складається із твердих пластин з металу або скла, покритих магнітним шаром для запису, шпинделя й головок зчитування – запису.

Елемент ***(element)*** — об'єкт (матеріальний, енергетичний, інформаційний), яким слід оперувати в дослідженні системи, але внутрішня будова (зміст) якого безвідносна до мети розгляду. Елементами можуть бути книга, верстат, працівник, документ, підприємство, сила, маса, енергія, файл, матриця, запис файлу тощо.

Елемент даних ***(data item, data element)*** — одиниця даних, що в певних контекстах розглядається як неподільна.

Запит ***(record)*** — звернення до програми з метою одержання інформації, що міститься в базі даних, яке формулюється у виді пропозиції або команди.

Зовнішні пристрої - див. **Пристрої введення-виведення**

Зовнішня (вторинна) пам'ять – розширення основної пам'яті, що забезпечує функціональність стійкої (що зберігається) пам'яті великого обсягу.

Ідентифікатор об'єкта або типу сутності *(entity identifier)* — атрибут або сукупність атрибутів, які однозначно ідентифікують об'єкт або тип сутності проблемної сфери.

Ієрархія *(hierarchy)* — різновид структури системи, елементи якої наділяються властивістю підпорядкування. Ієрархія називається сильною (ієрархія типу "дерево"), якщо між рівнями ієрархічної структури існують взаємовідносини суворої підпорядкованості компонентів нижчого рівня одному з компонентів вищого рівня. Ієрархія називається слабкою, якщо один і той самий вузол нижчого рівня ієрархії може бути підпорядкований кільком вузлам вищого рівня. Основне призначення ієрархічної організації у створюваній системі — розподіл функцій оброблення інформації та здійснення процедур вибору між окремими елементами системи.

Ініціатива щодо надійних і безпечних обчислень (trustworthy computing initiative) – ініціатива корпорації Microsoft (2002), метою якої є підвищення надійності й безпеки програмного забезпечення, насамперед – операційних систем.

Індекс (index) — таблиця, що використовується для визначення місця знаходження запису.

Інтерпретатор *(interpretive routine)* — програма, що розшифровує команди у псевдокодах і негайно виконує їх на відміну від компілятора, який розшифровує псевдокоди та формує програму машинною мовою для подальшого виконання.

Інтерфейс, дружній до користувача *(friendly user interface)* — властивість, що забезпечує комфортне робоче середовище при взаємодії користувача з інформаційною системою.

Інтерфейс користувача *(user interface)* — частина програми, що відповідає за діалог з користувачем і може мати форму природно-мовної системи (для інтелектуальних баз даних та експертних систем).

Інфрачервоний порт (IrDA) – порт для підключення ноутбука до мобільного телефона (або двох ноутбуків один до одного) через інфрачервоний зв'язок.

Інформаційна система *(information system)* — система, що організує пам'ять і маніпулювання інформацією про проблемну сферу.

Інформація *(information)* — знання про предмети, факти, поняття тощо проблемної сфери, якими обмінюються користувачі системи оброблення даних.

Кластери з балансуванням завантаження (load-balancing clusters) – комп'ютерні кластери, які мають кілька вхідних комп'ютерів, що балансують запити (front- ends) та розподіляють завдання між комп'ютерами серверного бек - енда.

Кластери з високошвидкісним доступом (high-availability clusters, HAC) – комп'ютерні кластери, що забезпечують оптимальний доступ до ресурсів, наданих комп'ютерами кластера, наприклад, до баз даних.

Кластери комп'ютерів – групи комп'ютерів, фізично розташовані поряд і з'єднані один з одним високошвидкісними шинами й лініями зв'язку.

Клієнт-Серверна система – розподілена комп'ютерна система, у якій певні комп'ютери відіграють роль спеціалізованих серверів, а інші – роль клієнтів, що користуються їхніми послугами.

Кишеньковий портативний комп'ютер (КПК, органайзер) - мініатюрний комп'ютер, що поміщається на долоні або в кишені, за своїми параметрами майже порівнюваний з ноутбуком, він призначений для повсякденного використання з метою запису, зберігання й читання інформації, у тому числі – мультимедійної, і комунікації через Інтернет.

Кеш-пам'ять *(cache memory)* — надоперативна пам'ять, кеш. Запам'ятовуючий пристрій з високою швидкістю доступу (в кілька разів більшою, ніж швидкість доступу до основної оперативної пам'яті), що застосовується для тимчасового зберігання проміжних результатів і вмісту комірок пам'яті, які часто використовуються.

Кільцева структура *(ring structure)* — організація даних у ланцюжки, в яких кінець ланцюжка вказує на його початок, утворюючи таким чином кільце.

Класифікація *(classification, від лат.classis — розряд, fasio — роблю)* — один із найпростіших різновидів моделювання, розподіл предметів певного роду на взаємозв'язані класи згідно з найсуттєвішими ознаками, властивими предметам одного роду, які відрізняють їх від предметів інших родів.

Ключ *(key)* — ідентифікатор, що міститься в середині набору елементів даних.

Ключ вторинний *(secondary key)* — ключ, що може ідентифікувати щонайменше один запис; ключ, який містить значення атрибуту (елемента даних), відмінного від унікального ідентифікатора (пошуковий ключ).

Ключ зовнішній *(foreign key)* — атрибут підпорядкованого логічного запису, що є первинним ключем породжувального логічного запису і слугує для з'єднання записів за цим атрибутом.

Комп'ютер переносний - надмініатюрний комп'ютер, убудований в одяг або імплантований у тіло людини, призначений для обробки інформації від датчиків, управління спеціалізованими

пристроями (наприклад, кардіостимулятором), або видачі рекомендацій з навігації й виконання інших типових дій людини.

Контролер пристрою – спеціалізований процесор (пристрій керування) для якогось пристрою комп'ютерної системи - основної пам'яті або зовнішнього пристрою.

Контрольна точка/рестарт *(check point /restarti)* — спосіб повторного пуску програми з точки, що відрізняється від початкової. Використовується після виникнення аварійної ситуації чи переривання. Контрольні точки можуть бути створені через певні проміжки часу роботи прикладної програми. В них запам'ятовуються записи, що містять достатню кількість інформації про стан програми і дають змогу здійснити її рестарт із зазначеної точки.

Контрольний приклад *(check problem)* — приклад із відомим розв'язком, що використовується для перевірки правильності роботи функціонального блока.

Користувальницький (непривілейований) режим (user mode) – стандартний режим виконання програм, у якому виконуються програми користувачів. У даному режимі заборонені деякі привілейовані операції (наприклад, зміна системних ділянок пам'яті й регістрів).

Лічильник команд – адреса поточної виконуваної або перерваної команди процесора.

Локальна мережа *(local area network(lan))* — мережа передачі даних, що об'єднує кілька цифрових пристроїв, розташованих на обмеженій площі, яка не перевищує десятки квадратних кілометрів. Як пристрої можуть бути робочі місця, міні - та мікрокомп’ютери, інтелектуальна інструментальна апаратура тощо.

Математичне забезпечення *(mathematical support)* — сукупність економіко-математичних методів, моделей та алгоритмів оброблення інформації в автоматизованій інформаційній системі.

Материнська плата (motherboard) – основна друкована плата комп'ютера, на якій змонтовані процесор і пам'ять.

Мережний адаптер (мережна карта) – пристрій для підключення комп'ютера до локальної мережі.

Мобільний пристрій (мобільний телефон, комунікатор) – кишеньковий пристрій, призначений для голосового зв'язку, обміну короткими повідомленнями, а також для читання, запису й відтворення мультимедійної інформації й комунікації через Інтернет.

Модем (абревіатура від **модулятор – демодулятор**) – пристрій для виходу в Інтернет і передачі інформації по аналоговій або цифровій телефонній лінії.

Мова опису даних *(data description language)* — штучна мова для опису даних та їхніх взаємозв'язків у базі даних.

Модель *(model, від лат. "зразок")* — штучно створений об'єкт (реальний або ідеальний), який, будучи аналогічним (подібним) досліджуваному об'єкту, відображає і відтворює у простішому вигляді структуру, зв'язки та взаємозв'язки між елементами досліджуваного об'єкта, безпосереднє вивчення якого пов'язано з певними труднощами, тим самим полегшує процес одержання інформації про нього.

Моделювання *(modeling)* — вид творчої діяльності людини, що містить такі процедури: конструювання моделі на підставі попереднього вивчення об'єкта і виділення його основних характеристик; експериментальний та теоретичний аналіз моделі; зіставлення результатів з даними про об'єкт; корекція моделі; її використання в пізнавальному, творчому або виробничому процесах.

Модель математична *(mathematical model)* — абстрактна або знакова модель, побудована засобами математики (у виді системи рівнянь, графа, формули логіки).

Модуль *(module)* — цілісна група елементів системи, описана тільки своїми входами і виходами. Модуль як частина системи відображує зв'язок між елементами, тобто обмін речовиною, енергією, інформацією. Модульний підхід дає змогу значно спростити опис системи і зробити видимими та доступними для розуміння найскладніші системи.

Мультипрограмування *(multiprogrammihg)* — режим роботи, що передбачає почергове виконання двох або більше програм одним процесором.

Настільний комп'ютер – персональний комп'ютер, розташовуваний на робочому столі й використовуваний на роботі або дома.

Обчислювальне середовище – інтегрована розподілена комп'ютерна система для рішення завдань у якихось проблемних сферах.

Операційна система – базове системне програмне забезпечення, що управляє роботою комп'ютера і є посередником (інтерфейсом) між апаратурою, прикладним програмним забезпеченням і користувачем комп'ютера.

Опитування пристроїв (polling) – дії операційної системи по періодичній перевірці стану всіх портів і зовнішніх пристроїв, що можуть мінятися із часом.

Основна (оперативна) пам'ять – швидкодіюча пам'ять, до якої процесор має безпосередній доступ під час виконання програми,

що зберігає програми й дані, інформація в якій не зберігається після вимикання комп'ютера або перезапуску системи.

Пам'ять – частина комп'ютера, що зберігає дані й програми.

Паралельна комп'ютерна система – мультипроцесорна система, що складається з декількох безпосередньо взаємодіючих процесорів.

Паралельний порт, або **LPT (абревіатура від Line PrinTer)** – порт для підключення застарілих моделей принтерів. Для підключення принтера через даний порт потрібно попередньо відключити й принтер, і комп'ютер.

Переривання за таймером – періодичні переривання через певний квант часу, призначені для **опитування пристроїв** і інших необхідних періодичних дій ОС.

Переповнення (*overflow*) — ситуація, коли запис (або сегмент) не може бути розміщений за його власною адресою, тобто адресою пам'яті, яку логічно було присвоєно йому в процесі завантаження.

Підсистема *(sybsystem)* — частина системи, що вивчається або досліджується самостійно. Є компонентом більшим, ніж. окремий елемент системи, але меншим, ніж система загалом. Повинна мати властивості системи, але для неї має бути сформульована своя локальна мета функціонування.

Підсистема управління ресурсами – компонент операційної системи, що управляє обчислювальними ресурсами комп'ютера.

Покажчик *(pointer)* — адреса запису (або іншого групування даних), що міститься в іншому записі. Прочитавши один запис, програма, використовуючи покажчик, може одержати доступ до іншого запису. Адреса може бути абсолютною, відносною або символічною.

Поле даних *(date field)* — складова частина запису, що відповідає певному атрибуту.

Порт – пристрій з рознімачем і контролером для підключення до комп'ютера зовнішніх пристроїв.

Портативний комп'ютер (ноутбук, лаптоп) – мініатюрний комп'ютер, за своїми параметрами не уступає настільному, але за своїми розмірами вільно поміщається в невелику сумку й призначений для використання в поїздці, дома, на дачі.

Прикладна задача *(application problem)* — задача, ініційована користувачем, що потребує для її розв'язання оброблення інформації.

Проблема *(problem)* — ситуація, яка має наявний і бажаний стани. Кожен з них містить набір об'єктів, властивостей та зв'язків, об'єднаних у процес. Проблеми поділяють на кількісні, якісні, слабоструктуровані та мішані.

Прикладне програмне забезпечення – програми, призначені для рішення різних класів завдань.

Пристрої введення-виведення – пристрої комп'ютера, що забезпечують введення інформації в комп'ютер і виведення результатів роботи програм у формі, сприйманої користувачем або іншими програмами.

Програма, керована перериваннями (interrupt-driven program) – програма, що запускається автоматично при виникненні переривання центрального процесора (наприклад, операційна система).

Програмувальне переривання (trap; дослівно – пастка) – переривання, яке явно генерується за допомогою спеціальної команди процесора (зазвичай для обробки помилки в програмі).

Прокси-Сервер – комп'ютер і програмне забезпечення, що є частиною локальної мережі й підтримує ефективне звернення комп'ютерів локальної мережі до Інтернету, фільтрацію трафіка, захист від зовнішніх атак.

Процес *(process)* — послідовність станів системи, що відповідає впорядкованій зміні конкретного параметра, який визначає характерні властивості системи.

Реальний час *(real time)* — час, що збігається з фактичним часом перебігу фізичного процесу. Стосовно режиму оброблення даних це означає, що потрібні розрахунки виконуються під час фактичного перебігу відповідного фізичного процесу так, що результати обчислень можна використовувати для керування цим процесом. Стосовно прикладних програм це значить, що в них відповідь на інформацію, яка вводиться, формується досить швидко, так що вона може впливати на подальше введення даних (діалоговий режим).

Регістр межі (limit register) – системний регістр, використовуваний для захисту пам'яті й такий, що утримує довжину ділянки пам'яті, виділеної користувальницькій програмі.

Рекурсія *(recursion)* — властивість структури даних або правила, що полягає в можливості звернення до самої себе один чи більше разів.

Робоча пам'ять *(working storag)* — частина оперативної пам'яті ЕОМ, що резервується для розміщення тимчасових результатів операцій.

Робоча станція *(workstation)* — підключений до мережі персональний комп'ютер користувача, якому доступні її ресурси.

Розв'язування задачі *(problem solve)* — процес одержання підсумкового показника (документа, відеокадра), що містить

інформацію для прийняття рішень під час управління діяльністю суб'єкта господарювання.

Розподілена система (distributed system) – комп'ютерна система, у якій обчислення розподілені між декількома фізичними процесорами (комп'ютерами), об'єднаними між собою в мережу проводну або безпроводну мережу.

Сектор – частина **жорсткого диска**, обмежена **доріжкою** й двома радіусами.

Сервер баз даних (database server) – комп'ютер і програмне забезпечення, що надає доступ іншим комп'ютерам мережі до баз даних, розташованих на комп'ютері-сервері локальної мережі.

Серверний бек-енд (Server back-end) – група (пул) зв'язаних у локальну мережу серверних комп'ютерів, використовуваних замість одного сервера, з метою більшої надійності й надання більшого обсягу ресурсів.

Сервер додатків (application server) – комп'ютер і програмне забезпечення, що надає обчислювальні ресурси (пам'ять і процесор) і необхідне оточення для віддаленого запуску певних класів (як правило, більших) додатків з інших комп'ютерів локальної мережі.

Сервер електронної пошти – комп'ютер і програмне забезпечення, що виконують відправлення, одержання й "розкладку" електронної пошти для комп'ютерів деякої локальної мережі. Може забезпечувати також криптування пошти (email encryption).

Синхронне введення-виведення – операція введення-виведення, виконання якої приводить до переходу програми в стан очікування, доти, поки операція введення-виведення не буде повністю закінчена.

Системний виклик (system call) – явний запит користувальницької програми до ОС шляхом виклику системної підпрограми.

Система реального часу – обчислювальна система, призначена для управління технічним, військовим або іншим об'єктом у режимі реального часу.

Системний (привілейований) режим (system mode, kernel mode, monitor mode) – особливий режим виконання команд, у якому виконуються модулі ядра ОС, що допускають виконання ряду привілейованих операцій, наприклад, зміну системних ділянок пам'яті й регістрів

Системна шина (system bus) – комунікаційний пристрій, що з'єднує між собою всі модулі комп'ютерної системи - центральний

процесор, пам'ять і контролер пам'яті, зовнішні пристрої і їхні контролери, які через системну шину обмінюються сигналами.

Слабо зв'язана система (loosely coupled system) – розподілена комп'ютерна система, у якій кожний процесор має свою локальну пам'ять, а різні процесори взаємодіють між собою через лінії зв'язку.

Сканер – пристрій для оцифрування паперових зображень, наприклад, підписаних або рукописних документів.

Симетрична кластеризація (symmetric clustering) - організація комп'ютерного кластера, при якій всі машини кластера виконують одночасно різні частини одного великого додатка.

Симетрична мультипроцесорна система (symmetric multiprocessing - SMP) - багатопроцесорна комп'ютерна система, всі процесори якої рівноправні й використовують ту саму копію ОС; операційна система при цьому може виконуватися на будь-якому процесорі.

Сервер мережі *(server)* — комп'ютер, підключений до мережі, що надає певні послуги користувачам.

Система *(system)* — сукупність елементів, зв'язки між якими з'єднують їх у структуру; об'єктивно існуючий комплекс процесів і явищ, а також зв'язків. Основні особливості систем: цілісність, відносна відокремленість від навколишнього середовища, наявність зв'язків з ним, наявність частин і зв'язків між ними (структурованість), підпорядкованість всієї системи конкретній меті.

Система збирання даних *(data acquisition (collection)* — найпростіша система телеоброблення, призначена вона для передачі інформації від абонентських систем до процесора. Наприклад, диспетчерська служба, що збирає інформацію від абонентських систем, обробляє її та передає на центральний диспетчерський пункт.

Система колективного користування *(multiaccess system)* — універсальна система, орієнтована на інтерактивний режим роботи віддалених користувачів.

Система оброблення даних *(data processiong system)* — система, що складається з сукупності технічних і програмних засобів, а також обслуговуючого персоналу, які забезпечують оброблення даних.

Система підтримки прийняття рішень *(decision support system)* — інтерактивна прикладна система, що забезпечує кінцевим користувачам, які приймають рішення, легкий та зручний доступ до даних і моделей з метою прийняття рішень у погано структурованих та неструктурованих ситуаціях різних сфер людської діяльності.

Система реального часу *(real-time system)* — інформаційно-керуюча система, що забезпечує передачу та оброблення даних зі швидкістю, яка відповідає швидкості перебігу керованого або контрольованого процесу.

Система розподіленого оброблення даних *(data distributed processing system)* — система оброблення даних, у якій база даних є централізованою (корпоративною), а обчислювальні потужності та програмне забезпечення розподілені між взаємозв'язаними ЕОМ.

Складність *(complexity)* — властивість явища (об'єкта, процесу, системи), що проявляється в несподіваності, непередбачуваності, незрозумілості, випадковості, "антиінтуїтивності" його поведінки. Виникає внаслідок непростої взаємодії великої кількості об'єктів, поведінка одного або кількох об'єктів впливає на поведінку інших.

Сортування *(sort)* — упорядкування файлу згідно з заданим ключем.

Список *(list)* — упорядкована множина елементів даних.

Стан процесора – значення регістрів і значення **лічильника команд .**

Структура *(structure, від лат. "будова")* — суттєві взаємозв'язки між елементами та їхніми групами (підсистемами), які мало змінюються при змінах у системі й забезпечують її існування та основні властивості. Зображають її у виді схеми з комірками, що відповідають елементам або групам елементів, і лініями, які з'єднують їх.

Суперкомп'ютер – потужний багатопроцесорний комп'ютер з продуктивністю до декількох петафлопс (1015 дійсних операцій у секунду), призначений для рішення завдань, що вимагають великих обчислювальних потужностей, наприклад, моделювання, прогнозування погоди.

Сутність *(entity)* — будь-який конкретний або абстрактний об'єкт зі зв'язками між іншими об'єктами.

Таблиця стану пристроїв – таблиця, яку зберігає та використовує операційна система, у якій кожному пристрою відповідає елемент, що містить тип пристрою, його адресу й стан, а для зайнятого пристрою – посилання на чергу оброблюваних запитів до нього.

Таймер – системний регістр, що містить деяке встановлене спеціальною командою значення часу, що зменшується через кожний квант (такт) процесорного часу. Коли значення таймера дорівнює нулю, відбувається переривання.

Тісно зв'язана (tightly coupled) система – паралельна комп'ютерна система, у якій процесори розділяють загальну пам'ять і таймер (такти); взаємодія між ними відбувається через загальну пам'ять.

Технічне забезпечення *(hardware)* — комплекс технічних засобів, що забезпечують роботу автоматизованої інформаційної системи.

Технологічне забезпечення *(technology support)* — сукупність організаційних, методичних і технологічних документів, що регламентують процес людино-машинного оброблення інформації в автоматизованій інформаційній системі.

Технологія електронної пошти *(e-maii technology)* — технологія комп'ютерного способу пересилання та оброблення інформаційних повідомлень, що забезпечує оперативний зв'язок між різноманітними користувачами.

Транзакція *(transaction)* — вхідне повідомлення, що належить до наявного файлу. Описує подію, яка зумовлює або формування нового запису файлу, або зміну чи вилучення наявного запису. Транзакція в сітьовому середовищі обчислень (розподілена транзакція) — логічна одиниця роботи, а також одиниця відновлення, паралелізму і цілісності.

Трафік *(traffic)* — потік інформаційного обміну, робоче навантаження лінії зв'язку.

Управління *(direction)* — процес організації цілеспрямованого впливу на об'єкт (систему), внаслідок якого він переходить у потрібний цільовий стан.

Управляюча програма – компонент операційної системи, управляючої виконанням інших програм і функціонуванням пристроїв вводу-виводу.

Файл *(file)* — множина записів одного типу.

Файл-Сервер (file server) – комп'ютер і програмне забезпечення, що надають доступ до підмножини файлових систем, розташованих на дисках комп'ютера-сервера, іншим комп'ютерам локальної мережі.

Флеш-Пам'ять (флешка) – модуль зовнішньої пам'яті компактного розміру, що підключається через USB-Порт і має ємність до 128 ГБайт.

Хмарні обчислення – модель обчислень, заснована на **динамічно масштабованих (scalable)** і **віртуалізованих** ресурсах (даних, додатках, ОС і ін.), які доступні й використовуються як **сервіси** через Інтернет і реалізуються за допомогою високопродуктивних **центрів обробки даних (data centers).**

Центральний процесор – центральна частина комп'ютера, що виконує його команди (інструкції).

Циліндр – частина **жорсткого диска**, що представляє собою сукупність **доріжок** одного діаметру, що перебувають на всіх його паралельно розташованих магнітних дисках.

Черга переривань – системна структура ОС, що забезпечує почергову обробку всіх викликаючих переривань.

Ядро – основний низькорівневий компонент будь-якої операційної системи, виконуваний апаратурою в привілейованому режимі, що завантажується при запуску ОС і резидентно перебуває в пам'яті.

ЛІТЕРАТУРА

1. Абрамов С.А., Гнездилова Г.Г., Капустина Е.Н., Селюн М.И. Задачи по программированию – М.: Наука, 1988 – 280 с.
2. Ананьев А.И., Федоров А.Ф. Самоучитель Visual Basic 6.0 - – СПб.: БХВ - Питер, 2003. – 624 с.
3. Браун С. Visual Basic 6; Учебный курс - СПб: ЗАО «Издательство «Питер». 1999. – 562 с.
4. Буч Г. Обьектно-ориентированное проектированиє с примерами применения.-К.: Диалектика, 1992.- 519с.
5. Бэкон Дж., ХаррисТ. Операционные системы – К.: Изд. груп. BHV; СПб.: Питер, 2004. – 800 с.
6. Воеводин В. В., Воеводин Вл. В., Параллельные вычисления. – СПб.: БХВ - Петербург, 2004. – 608 с.
7. Зарецька Т.І., Колодяжний Б.Г. та ін. Інформатика. Навч. посіб. .К., Навчальна книга, 2002. – 425 – 438 с.
8. Збірник задач з курсу „Інформатика та програмування” для студентів механіко-математичного факультету.: Вакал Є.С., Личман В.В., Обвінцев О.В., Бублик В.В., Попов В.В. - К.: ВПЦ „Київський університет”, 2004.
9. Кузьменко В.Г. VBA 2000 – М.: ЗАО «Издательство БИНОМ», 2000. – 408 с.: ил.
10. Лабораторний практикум з інформатики та комп’ютерних технологій / В.В. Браткевич та ін./ За ред. О.І. Пушкаря: Навч. посібник. – Х.: Видавничій дім «ІНЖЕК», 2003. – 424 с .
11. Інформатика: Комп'ютерна техніка. Комп'ютерні технологи / За ред. О. І. Пушкаря, - К.: Видав. центр «Академія», 2002. – 704 с.
12. Кнут Д. Искусство программирования. – Т.3: Сортировка и поиск. – М.: Вильямс, 2004. – 703с.
13. Ковалюк Т.В. Основи програмування. – К.: Видавнича група BHV, 2005. – 384 с.
14. Microsoft Visual Basic 6.0 для профессионалов. Шаг за шагом: Практ. пособ./ Пер. с англ. – М.: Издательство ЭКОМ, 2001. – 720 с.: ил.
15. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. - СПб.: Питер, 2001. – 544 с.

16. Пол Сана и др. Visual Basic для приложений (версия 5) в подлиннике: пер. с англ. - СПб.: BHV - Санкт-Петербург. 1999. - 704 с., ил.
17. Пильщиков В.Н. Язык Паскаль: Упражнения и задачи. – М.: Науч. мир, 2003. – 224 с.
18. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы: Теория и практика. – М.: Мир, 1980, - 4376 с.
19. Стивенс Р. Delphi. Готовые алгоритмы / Род Стивенс; Пер. с англ. Мерешука П. А. – 2-е изд., стер. – М.: ДМК Пресс: СПб.: Питер, 2004. – 384 с.: ил.
20. Таненбаум Э. Операционные системы – СПб.: Питер, 2002. – 1040 с.
21. Таненбаум Э., Ван Стеен М. Распределенные системы. Принципы и парадигмы. – СПб.: Питер, 2003. – 880 с.
22. Техника программирования: Учеб. пособие/ В.С. Проценко, П.И. Чаленко, Р.А. Сорока. – К.: Віща шк., 1990. – 183 с.: ил.
23. Экономическая информатика: Учебник для вузов / Под ред. В. В. Евдокимова -СПб.: Питер, 1997
24. Шелест В.Д. Программирование. – СПб.: БХВ Петербург, 2001. – 592 с.
25. Юркин А.Г. Задачник по программированию. – СПб.: Питер, 2002. – 192 с.

Учбове видання

Рамазанов С. К., Велігура А. В., Танченко С. М.

Основи інформатики та технологій програмування

Навчальний посібник

УКРАЇНСЬКОЮ МОВОЮ

Редактор З. І. Андронова
Техн. редактор Т. М. Дроговоз
Оригінал-макет І. О. Сидорова

Підписано до друку 23.12.07.
Формат 60x84 1/16. Папір типогр. Гарнітура Times.
Друк офсетний. Умов. друк. арк. 26,85. Обл. вид. арк. 25,13
Тираж 1000 екз. Вид. № 348. Замов. № Ціна договірна.

Видавництво
Східноукраїнського національного університету
ім. В.Даля
91034, м. Луганськ, кв. Молодіжний, 20а

Адреса видавництва: 91034, м. Луганськ, кв. Молодіжний, 20а
Телефон: 8 (0642) 46-13-04. Факс: 8 (0642) 46-13-64
E-mail: uni@vugu.lugansk.ua
http://vugu.luugansk.ua