МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Шушура О.М., Золотухіна О.А.

ЛОГІКА ТА АЛГОРИТМИ ОБРОБКИ ДАНИХ

Навчальний посібник

Київ-2019

ЗМІСТ

Вступ 5
1 Особливості рішення задач з використанням ЕОМ 6
1.1 Етапи рішення задач з використанням ЕОМ 6
1.2 Поняття методу розв'язання задачі. Властивості, способи опису, класифікація методів 7
Контрольні питання 13
Практичні завдання 13
2 Властивості та способи представлення алгоритмів 15
2.1 Визначення та властивості алгоритмів 15
2.2 Складність алгоритму 17
1.3 Способи представлення алгоритмів 20
2.4 Поняття програми. Мови програмування та їх складові елементи 24
Контрольні питання 28
Практичні завдання 28
3 Типи даних та операції над ними 30
3.1 Поняття змінної 30
3.2 Тип цілих чисел 31
3.3 Тип дійсних чисел. 32
3.4 Логічний тип 35
3.5 Символьний тип 37
Контрольні питання 38
Практичні завдання 38
4 Базові алгоритмічні структури 40
4.1 Лінійна структура 40
4.2 Розгалужені алгоритми 40
4.2.1 Повне розгалуження. 41
4.2.2 Неповне рогалуження. 42
4.2.3 Багатоваріантний вибір. 43
4.3 Цикли 44

4.3.1 Цикли з параметром 44
4.3.2 Ітераційні цикли 46
4.4 Рекурентні обчислення 47
Контрольні питання 51
Практичні завдання 51
5 Структурне та об’єктно-орієнтоване програмування 54
5.1. Структурне програмування 54
5.1.1 Основні принципи структурного програмування 54
5.1.2 Підпрограми 56
5.1.3 Рекурсивні підпрограми 57
5.1.4 Переваги структурного програмування 58
5.2 Об’єктно-орієнтоване програмування 59
5.2.1 Причини виникнення об’єктно-орієнтованої парадигми програмування 59
5.2.2 Класи 61
5.2.3 Наслідування 62
5.2.4 Поліморфізм та перевантаження 63
Контрольні питання 65
Практичні завдання 65
6 Основні структури даних 66
6.1 Рівні організації даних 66
6.2 Класифікація структур даних 67
6.3 Базові операції зі структурами даних 69
6.4 Масиви 70
6.4.1 Поняття масиву. Способи опису 70
6.4.2 Цикл foreach 75
6.4.3 Алгоритми пошуку в одновимірних масивах 76
6.4.4 Алгоритми сортування в одновимірних масивах 77
6.4.2 Багатовимірні масиви 79
6.4.3 Динамічні масиви 81
6.4.4 Масиви масивів 83

6.4.5 Розріджені масиви 85
6.5 Структури 86
6.6 Перерахований тип 87
6.7 Списочні структури 89
6.7.1 Характеристика динамічних типів даних 89
6.7.2 Лінійні списки 90
6.7.3 Двох-зв’язний список 92
6.7.4 Операції над списками 93
6.7.5 Мультисписки 95
Контрольні питання 96
Практичні завдання 96
7 Додаткові структури даних 99
7.1 Множина 99
7.2 Об’єднання 99
7.3 Таблиці 100
7.4 Стеки 101
7.5 Черга 103
7.6 Деки 105
7.7 Рядки 106
7.8 Вектор 108
7.9 Мар 108
7.10 Графи 110
7.11 Дерева 111
7.12 Хешування даних 114
Контрольні питання 116
Практичні завдання 117
Додаток 1. Приклади роботи з C# 118
Загальна структура програми в C#. 118
Використання Microsoft Visual Studio. 119
Введення/виведення даних 119
Створення програми «Hello World» 120

ВСТУП

Системний аналіз – це наукова методологія, об’єктом аналізу якої є проблема, незалежно від сфери діяльності, де вона виникла, метою системного аналізу є проект вирішення проблеми. Системний аналіз в інформаційних технологіях орієнтований на вирішення складних проблем аналізу та створення комп’ютерних, інфокомунікаційних, експертних та інтелектуальних систем.

Для розробки програмного забезпечення в першу чергу необхідно поставити задачу, розробити алгоритм розв’язку задачі, визначити необхідні дані, а потім вже обрати підходящу мову програмування та перекласти на неї алгоритм.

Навчитися складати алгоритми вирішення нових завдань, вивчаючи лише вже розроблені алгоритми рішення «схожих» завдань, дуже складно, якщо не неможливо. Того, хто навчається, можна переконати в правильності, в логічній бездоганності знайдених рішень, але йому не завжди зрозуміло, як до цього «додумався» автор рішення, як він сам міг би здогадатися вирішити задачу.

Даний посібник містить опис основних засобів розробки алгоритмів. До кожного розділу за темою наводиться теоретичний матеріал, який супроводжується прикладами, контрольні запитання для самостійного опрацювання, а також перелік практичних завдань для самостійного вирішення.

1 ОСОБЛИВОСТІ РІШЕННЯ ЗАДАЧ З ВИКОРИСТАННЯМ ЕОМ

1.1 Етапи рішення задач з використанням ЕОМ

Поняття «рішення задачі» при використанні ЕОМ включає в себе набагато більше, ніж просто обчислення. У процесі підготовки завдань до вирішення за допомогою ЕОМ можна виділити ряд етапів:

1) формулювання завдання в термінах деякої прикладної області знання (математики, фізики, економіки і т.п.);

2) формалізація задачі, побудова математичної або інформаційної моделі;

3) вибір методу рішення формалізованої завдання (включає визначення інформаційних технологій та (або) засобів програмної обробки даних);

4) розробка проекту рішення або алгоритму;

5) тестування і налагодження програми;

6) передача програми в експлуатацію.

Зверніть увагу!

Наведена послідовність відображає ***водоспадну модель розробки****. При використанні інших моделей порядок та зміст етапів може відрізнятися!*

При постановці прикладної задачі фахівець тієї чи іншої галузі знань або діяльності формулює завдання, визначає вихідні дані і мета її вирішення.

Запис умови задачі за допомогою математичних рівнянь та нерівнянь, формулювання цілей рішення мовою математичних понять є математичною постановкою завдання або формалізацією. Опис найбільш істотних в розв'язуваній задачі властивостей досліджуваних об'єктів або явищ за допомогою математичних формул і рівнянь називається побудовою математичної моделі цього об'єкта або явища.

Наступним кроком у вирішенні завдання на ЕОМ є алгоритмізація і програмування. Для простих навчальних програм ці два етапи можуть об'єднатися в один. Для складних завдань необхідний етап проектування, що включає чітке формулювання завдання, опис структур даних і основних процесів їх обробки, розбиття вихідної задачі на підзадачі. Це можуть бути підзадачі введення вихідних даних (чи то з файлу, то чи в діалозі з користувачем програми), підзадачі обробки керуючих кодів, введених користувачем в процесі виконання програми (наприклад, натисканням спеціально обумовлених клавіш клавіатури), підзадачі виконання розрахунків, визначених цілями вихідної задачі, підзадачі виведення результатів на екран або пристрої друку тощо.

Наступним кроком у вирішенні завдань на ЕОМ є тестування і налагодження. Випробування будь-якої системи завжди є один з найбільш відповідальних етапів її розробки і часто бувають пов'язані з найбільшими труднощами-ми і найбільшими втратами часу. Налагодження і тестування – це два чітко помітних і пов'язаних між собою етапи, призначених для перевірки працездатності програми.

Тестування алгоритму (програми) – це випробування його на придатність до вирішення поставленого завдання. Тестування виконується при вихідних даних завдання (конкретних значеннях аргументів), для яких заздалегідь відома реакція програми.

Якщо отримані результати не збігаються з очікуваними хоча б в одному з тестів, то переходять до налагодження – пошуку і виправлення помилок в алгоритмі.

Зверніть увагу!

При тестуванні обов'язково використовують набори даних, що відповідають коректній роботі програми, та набори, які можуть призвести до виникнення помилок.

Переконавшись в тому, що програма працює правильно, її використовують для роботи з реальними даними. Якщо програма була комерційним продуктом і призначалася для широкого кола користувачів, то вона надходить на ринок програмних продуктів.

1.2 Поняття методу розв'язання задачі. Властивості, способи опису, класифікація методів

Метод рішення являє собою формалізований опис способу вирішення поставленого завдання.

Основними вимогами, що пред'являються до методів вирішення завдань, є наступні:

1) метод вирішення повинен забезпечувати отримання вірних результатів для будь-яких допустимих вихідних даних;

2) структури і типи даних, що використовуються в методі, повинні бути допустимими в програмуванні;

3) опис методу повинно бути точним, зрозумілим і однозначним;

4) метод повинен забезпечувати раціональне рішення задачі.

Існує кілька способів опису методів вирішення завдань:

1) математичний опис (за допомогою математичних термінів, формул, визначень і т.п.);

2) словесний опис (за допомогою коротких, але ясних формулювань);

3) графічний опис (за допомогою малюнків, графіків, геометричних позначень і т.п.).

Зверніть увагу!

Для більшої зрозумілості методу як правило, використовується комбінація способів опису. Наприклад, при вирішенні геометричної задачі використовується графічне зображення методу рішення, яке супроводжується математичними формулами і словесними коментарями.

Далі наведено приклади позначень, які можуть бути використані при описі математичної моделі методу вирішення задачі.

1. $\forall$ Кожен, будь-який, всякий (квантор спільності).

2. $\exists$ «Існує» (квантор існування).

3. параметр $= \overline{\text{поч, кін}}$

Визначає зміни параметра від значення *поч* до значення *кін* включно з кроком 1. Наприклад, показати, що величина і приймає значення від 1 до N можна за допомогою виразу

$$i = \overline{1, N}$$

Якщо необхідна зміна параметра з іншим кроком, то в цьому випадку просто вказується через кому список з декількох значень, за якими можна визначити крок, наприклад:

$$i = 1,5,9\ldots$$

Даний приклад показує, що величина і змінюється з кроком 4.

4. | - Вертикальна риска відокремлює математичний вираз або групу висловів від умов, за яких ці вирази повинні обчислюватися. Вирази пишуть зліва від межі, умови або обмеження – праворуч. Дія умов або обмежень поширюється на всі вирази, які охоплює висота риски.

Приклад 1.1. Знайти кількість нульових елементів масиву X.

Математичний запис: $k_{нач} = 0$
$k = k + 1 \left| \forall x_i = 0, i = \overline{1, n} \right.$

5. $\min\limits_{\substack{\text{обмеження} \\ \text{умови}}}$ (елемент)

Визначає мінімальний серед зазначених елементів з урахуванням обмежень та вказаних умов. Умови та обмеження можуть накладатися як на самі значення елементів, так і на їхні номери.

Приклад 1.2. Знайти мінімум серед елементів одновимірного масиву довжиною N.

Математичний запис: $\min\limits_{i=\overline{1,N}}(a_i)$

Якщо необхідно обчислити мінімум не всіх елементів з набору, а тільки тих, які задовольняють деякому умові, то дані умови описуються з використанням знаку вертикальної риски.

Приклад 1.3. Знайти мінімуми по рядках серед елементів матриці розміром MxN.

Математичний запис:

$$\min_i = \min\limits_{j=\overline{1,N}}(a_{ij}) \Big| \forall\, i = \overline{1,M}$$

Приклад 1.4. Знайти мінімум серед позитивних елементів одновимірного масиву з N елементів.

Математичний запис: $\min = \min(a_i) \Big|$ для $\forall\ a_i > 0; i = \overline{1,N}$

або $\min = \min\limits_{\substack{i=\overline{1,N} \\ \forall a_i > 0}}(a_i)$

6. $\max\limits_{\substack{\text{обмеження} \\ \text{умови}}}(\text{елемент})$

Визначає максимальний серед зазначених елементів з урахуванням обмежень та вказаних умов. Умови та обмеження можуть накладатися як на самі значення елементів, так і на їхні номери. Застосування аналогічно мінімуму.

7. $\sum\limits_{\text{поч}}^{\text{кін}} \text{елемент}$

Визначає обчислення суми елементів, розташованих праворуч від знаку підсумовування, номер елемента змінюється від *поч* до *кін*. Значення *поч* і *кін* можуть бути рівні ± ∞. У цьому випадку указуються додаткові умови, які визначають, до яких пір виробляти обчислення суми.

В якості елемента суми може бути не тільки конкретна величина, а й інше математичний вираз.

Приклад 1.5. Обчислити суму елементів одновимірного масиву з N елементів.

Математичний запис:

$$S = \sum_{i=1}^{N} a_i$$

Приклад 1.6. Обчислити суму елементів матриці розміром MxN.

Математичний запис:

$$S = \sum_{i=1}^{M} \sum_{j=1}^{N} a_{ij}$$

Приклад 1.7. Обчислити суми елементів матриці розміром MxN по рядках.

Математичний запис:

$$S_i = \sum_{i=1}^{M} a_{ij} \mid \forall i = \overline{1, M}$$

Приклад 1.8. Обчислити суми елементів матриці розміром MxN по стовпцях.

Математичний запис: $S_j = \sum_{j=1}^{N} a_{ij} \mid \forall j = \overline{1, N}$

Якщо необхідно обчислити суму не всіх елементів з набору, а тільки тих, які задовольняють деякому умові, то дані умови описуються з використанням знаку вертикальної риски.

Приклад 1.9. Обчислити суму додатних елементів одновимірного масиву з N елементів.

Математичний запис: $S = \sum_{i=1}^{N} a_i \mid \text{для} \forall\ a_i > 0$

8. $\prod_{\text{поч}}^{\text{кін}} \text{елемент}$

Визначає обчислення добуток елементів, розташованих праворуч від знаку добутку, номер елемента змінюється від *поч* до *кін*. Застосування аналогічно сумі.

9. $\text{величина} = \begin{cases} \text{знач1}, & \text{умова1} \\ \text{знач2}, & \text{умова2} \\ \dots & \\ \text{значN}, & \text{уманаN} \end{cases}$

Величина, що стоїть зліва від знаку рівності, приймає одне із значень (знач1, знач2 і т.д.), що стоять за дужкою в залежності від виконання тієї чи іншої умови. Умови повинні бути взаємовиключними. Допускається замінювати остання умова словом «інакше», якщо воно включає в себе весь діапазон значень, який залишився після виключення інших умов:

$$\text{величина} = \begin{cases} \text{знач1}, & \text{умова1} \\ \text{знач2}, & \text{умова2} \\ \dots & \\ \text{значN}, & \text{інакше} \end{cases}$$

В якості значень можуть бути інші математичні вирази (крім фігурної дужки). В умовах також допускається використання математичних виразів (крім фігурної дужки).

Методи вирішення задач можна поділити на спеціалізовані та універсальні.

До універсальних можна віднести наступні:

1. Покрокова деталізація (послідовне уточнення, розбиття завдання на підзадачі, «розділяй і володарюй»), тобто кожна підзадача вирішується окремо, а потім вони з'єднуються одна з одною.

2. Зведення однієї задачі до іншої, вже вирішеної (наприклад, задача знаходження мінімуму серед N елементів зводиться до задачі знаходження мінімуму серед двох елементів).

3. Динамічне програмування (в найбільш загальній формі) – процес покрокового розв’язку задачі, коли на кожному кроці вибирається один розв’язок з множини допустимих на цьому кроці, причому такий, який оптимізує задану цільову функцію або функцію критерію.

4. Метод сходження – полягає у тому, щоби протягом пошуку найкращого розв’язку алгоритм відшукував все кращі та кращі варіанти розв’язку. Якщо ввести деяку кількісну оцінку якості розв’язку, який шукається, то такий метод подібний на здолання все нової та нової висоти при сходженні на вершину.

5. Метод спроб та помилок. Багато задач не допускають аналітичного розв’язку, а тому їх доводиться вирішувати методом спроб та помилок, тобто перебираючи усі можливі варіанти та відкидаючи їх у випадку невдачі. У разі, якщо побудова розв’язку є складною процедурою, то фактично під час роботи будується дерево можливих кроків алгоритму, а потім – у випадку невдачі – відрізаються відповідні гілки дерева, доки не буде побудовано той шлях, який веде до успіху. Проходження вздовж гілок дерева та відхід у разі невдач є алгоритмом з поверненнями.

Розробка ***складних завдань*** здійснюється методом низхідного проектування (званого також методом покрокової деталізації або послідовного уточнення), який передбачає конкретизацію алгоритмічних рішень від загальної постановки завдання до вирішення допоміжних завдань. Метод висхідного проектування передбачає «збірку» основного алгоритму з окремих модулів, що виконують допоміжні функції і розроблені незалежно один від одного.

Слід зазначити, що універсальні методи не обмежуються цим списком. Ключовою їх особливістю є те, що їх можна застосувати для вирішення майже будь-яких задач.

Спеціалізовані методи дозволяють вирішити конкретні задачі або застосовуються до деякого класу структур даних. Приклади спеціалізованих методів:

1. Метод перебору (перебираючи елементи, значення, виконуємо необхідні дії). Цей метод використовується в складі інших методів. Застосовується перебір до структур даних, які містять декілька елементів.

Приклад 1.10. Знайти кількість нульових елементів масиву X.

$k_{поч} = 0$

$k = k + 1 \mid \forall x_i = 0, i = \overline{1, n}$

2. Метод накопичення. В загальному формулюванні метод накопичення можна сформулювати наступним чином:

- задати початкове значення результату;
- для кожного елемента, що обробляється, застосувати його значення для зміни операції по зміні результату;
- при необхідності після обробки всієї структури даних, додатково обробити результат.

Методом накопичення можна вирішувати наступні розрахункові задачі:

- сума елементів;
- кількість елементів в наборі, які відповідають певним умовам;
- добуток елементів;
- середнє арифметичне елементів;
- інші подібні задачі.

Приклад 1.11. Знайти суму елементів масиву X.

$S_{поч} = 0$

$S = S + x_i \mid i = \overline{1, n}$

3. Метод послідовних наближень (наприклад, рішення задачі знаходження суми нескінченного ряду).

Приклад 1.12. Обчислити $y = e^x$ за формулою:

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!} + ...$$

Зауваження: При заданому n задача вирішується методом накопичення.

4. Метод пошуку мінімуму/максимуму:

На першому кроці початкове значення мінімуму (максимуму) встановлюється рівним будь якому елементу з тих, серед яких шукається мінімум (максимум), або заздалегідь більше (для мінімуму) або менше (для максимуму).

Далі застосовується метод перебору: на кожному кроці поточний елемент порівнюється з мінімумом (максимумом) і, якщо він менше за мінімум (більше за максимум), то значення мінімуму (максимуму) змінюється і становить рівним поточному елементу.

5. Інші методи.

Контрольні питання

1. Наведіть етапи необхідно виконати для вирішення задачі на ЕОМ. Який їх порядок та зміст?
2. Дайте визначення методу вирішення задачі.
3. Які способи запису методів рішення задач існують? Наведіть приклади.
4. Які позначення використовуються для опису математичних моделей? Наведіть приклади.
5. Для чого застосовуються універсальні методи? Наведіть приклади універсальних методів.
6. Для чого застосовуються спеціалізовані методи? Наведіть приклади спеціалізованих методів.

Практичні завдання

Опишіть математичну модель та метод вирішення для наступних задач.

1. Дано натуральне число n. Знайти суму першої та останньої цифри цього числа. Переставити місцями першу і останню цифри цього числа.
2. Серед усіх n-значних чисел вказати ті, сума цифр яких дорівнює даному числу k.
3. Задані три натуральних числа A, B, C, які позначають число, місяць і рік. Знайти порядковий номер дати, починаючи відлік з початку року.
4. Знайти на відрізку [n; m] натуральне число, що має найбільшу кількість дільників.
5. Знайти всі дільники натурального числа n.
6. Дана послідовність цілих чисел a1, a2,. . . , an. Визначити, яке число в ній зустрічається раніше – позитивне або негативне.
7. Дана послідовність чисел a1, a2,. . . , an. Вказати найменшу довжину числової осі, що містить всі ці числа.
8. Дана послідовність з n різних цілих чисел. Знайти суму її елементів, розташованих між максимальним і мінімальним значеннями (в суму включити і обидва цих числа).
9. Дана цілочисельна квадратна матриця n-го порядку. Визначити, чи є вона магічним квадратом, тобто такий, в якій суми елементів у всіх рядках і стовпчиках однакові.
10. У кожному стовпці квадратної матриці знайти максимальний по модулю елемент і, якщо він не є діагональним, поміняти його місцями

з діагональним елементом. Підрахувати кількість таких перестановок.

11. Розглядаючи елементи рядка прямокутної матриці як координати точки в n-вимірному просторі, визначити номери точок, відстань d між якими максимальна.
12. Дана дійсна матриця A розмірністю n × m. Визначити k - кількість "особливих" елементів матриці. Елемент вважається "особливим", якщо він більше суми інших елементів його стовпчика.
13. Дано рядок. Визначити кількість слів у рядку.
14. Дано рядок. Замінити кожен знак '+' на два знака '++'.
15. Дано рядок. Видалити з нього усі однакові символи, що йдуть підряд. Наприклад: вихідний рядок 'aaabbbcdefgggabc', результат 'cdefabc'.
16. В арифметичному виразі перевірити узгодженість дужок, правильність їх розташування і видати відповідні повідомлення.
17. У двовимірному просторі задана пряма і множина точок. Знайти точки, що знаходяться на мінімальному і максимальному відстані від прямої.
18. У тривимірному просторі задано множину точок. Чи перетнуться дві прямі, одна з яких проходить через дві найближчі до початку координат із заданої в просторі множини точок, друга – через дві максимально віддалені.
19. У двовимірному просторі задано множину точок. З кожного квадранта взята точка, максимально віддалена від початку координат. Перевірити, чи буде паралелограмом чотирикутник, побудований за обраними з множини точок.
20. У двовимірному просторі задано множину точок. Знайти параметри кола мінімального радіуса проходить через три точки множини.

2 ВЛАСТИВОСТІ ТА СПОСОБИ ПРЕДСТАВЛЕННЯ АЛГОРИТМІВ

2.1 Визначення та властивості алгоритмів

Алгоритмом називається зрозуміле і точне розпорядження виконавцю виконати послідовність дій, спрямованих на досягнення зазначеної мети чи на розв'язання поставленої задачі.

В цьому означенні використовується поняття "виконавець". Що це означає? Під виконавцем алгоритму ми розуміємо будь-яку істоту (живу чи неживу), яка спроможна виконати алгоритм. Все залежить від того, якої мети ми намагаємося досягнути. Наприклад: риття ями (виконавці - людина або екскаватор), покупка деяких товарів (один із членів родини), розв'язування математичної задачі (учень або комп'ютер) тощо. Поняття алгоритму в інформатиці є фундаментальним, тобто таким, котре не визначається через інші ще більш прості поняття (для порівняння у фізиці - поняття простору і часу, у математиці - крапка).

Будь-який виконавець (і комп'ютер зокрема) може виконувати тільки обмежений набір операцій (екскаватор копає яму, вчитель вчить, комп'ютер виконує арифметичні дії). Тому алгоритми повинні мати наступні властивості.

1. Зрозумілість. Щоб виконавець міг досягти поставленої перед ним мети, використовуючи даний алгоритм, він повинен уміти виконувати кожну його вказівку, тобто розуміти кожну з команд, що входять до алгоритму.

Наприклад: Мамі потрібно купити в магазині їжу. Виконавцем цього алгоритму може бути хтось із родини: батько, син, бабуся, маленька дочка тощо. Зрозуміло, що для тата достатньо сказати, які купити продукти, а далі деталізувати алгоритм не потрібно. Дорослому сину-підлітку необхідно детальніше пояснити в яких магазинах можна придбати потрібний товар, що можна купити замість відсутнього товару і таке інше. Маленькій дочці алгоритм необхідно деталізувати ще більше: де взяти сумку, щоб принести товар, яку решту грошей необхідно повернути з магазину, як дійти до магазину і як там поводитись (якщо дитина вперше йде за покупками).

2. Визначеність (однозначність). Зрозумілий алгоритм все ж таки не повинен містити вказівки, зміст яких може сприйматися неоднозначно. Наприклад, вказівки "почисти картоплю", "посоли за смаком", "прибери в квартирі" є неоднозначними, тому що в різних випадках можуть призвести до різних результатів. Крім того, в алгоритмах неприпустимі такі ситуації, коли після виконання чергового розпорядження алгоритму виконавцю не зрозуміло, що потрібно робити на наступному кроці. Наприклад: вас послали

за яким-небудь товаром у магазин, та ще попередили "без (хліба, цукру і таке інше) не повертайся", а що робити, якщо товар відсутній?

Отож, *визначеність* - це властивість алгоритму, що полягає в тім, що алгоритм повинен бути однозначно витлумачений і на кожному кроці виконавець повинен знати, що йому робити далі.

3. Дискретність. Як було згадано вище, алгоритм задає повну послідовність дій, які необхідно виконувати для розв'язання задачі. При цьому, для виконання цих дій їх розбивають у визначеній послідовності на прості кроки. Виконати дії наступного розпорядження можна лише виконавши дії попереднього. Ця розбивка алгоритму на окремі елементарні дії (команди), що легко виконуються даним виконавцем, і називається дискретністю.

4. Масовість. Дуже важливо, щоб складений алгоритм забезпечував розв'язання не однієї окремої задачі, а міг виконувати розв'язання широкого класу задач даного типу. Наприклад, алгоритм покупки якого-небудь товару буде завжди однаковий, незалежно від товару, що купується. Або алгоритм прання не залежить від білизни, що переться, і таке інше. Отож, під масовістю алгоритму мається на увазі можливість його застосування для вирішення великої кількості однотипних завдань.

5. Результативність. Взагалі кажучи, очевидно, що виконання будь-якого алгоритму повинне завершуватися одержанням кінцевих результатів. Тобто ситуації, що в деяких випадках можуть призвести до так званого "зациклення", повинні бути виключені при написанні алгоритму.

6. Правильність. При застосуванні алгоритму до допустимих вхідних даних має бути отриманий потрібний результат. Доведення правильності алгоритму – один з найскладніших етапів його створення. Найпоширеніша процедура перевірки правильності алгоритму – обґрунтування правомірності та перевірка правильності кожного з кроків при наборі тестів, підібраних так, щоб охопити всі допустимі вхідні і вихідні дані.

7. Ефективність. Забезпечувати розв'язання задачі за мінімальний час з мінімальними затратами апаратних і програмних ресурсів. Для оцінки алгоритмів існує багато критеріїв. Найчастіше аналіз алгоритму (або, як кажуть, аналіз складності алгоритму) полягає в оцінці затрат часу на розв'язок задачі у розрахунку на одиницю вхідних даних. Фактично, ця оцінка зводиться до оцінки кількості базових елементарних операцій, на які можна розкласти даний алгоритм, оскільки кожна така операція виконується за конкретний, відомий відрізок часу. Ефективність алгоритму оцінюється також кількістю апаратурних ресурсів, зокрема обсягом пам'яті, задіяної для виконання даного алгоритму.

2.2 Складність алгоритму

При оцінці ефективності алгоритму часто використовують поняття складності. Створення та реалізація алгоритму відповідно до свого призначення визначає його складність. Проте не існує інтегрованого показника складності алгоритму, хоча навіть існує спеціальний розділ – метрична теорія алгоритмів, що займається саме проблемами складності. Інтуїтивно можна виділити такі основні складові складності алгоритму:

1. Логічна складність - кількість людино-годин, витрачених на створення алгоритму.
2. Статична складність - довжина опису алгоритмів (кількість операторів).
3. Тимчасова складність - час виконання алгоритму.
4. Ємкісна складність - кількість умовних одиниць пам'яті, необхідних для роботи алгоритму.

Головною метою теорії складності є забезпечення механізму класифікації алгоритмів за складністю. Складність алгоритму дозволяє визначитися з вибором ефективного алгоритму серед існуючих, що побудовані для розв’язання конкретної проблеми. А саме вибір серед уже існуючих алгоритмів дозволяє не розглядати логічну та статичну складність, а оцінювати ті ресурси, що знадобляться під час реалізації обраних алгоритмів.

Означення. Складність алгоритму – це кількісна характеристика, що відображує споживані алгоритмом ресурси під час свого виконання.

Основними ресурсами, що оцінюються, є час виконання і простір пам’яті.

Інтуїтивно це поняття досить зрозуміле. В алгоритму є вхід - опис завдання, яке потрібно вирішити. На його розв’язання алгоритм витрачає певний час (тобто кількість операцій). Складність - це функція від довжини входу, значення якої дорівнює максимальній (за будь-якими входами даної довжини) кількості операцій, необхідних алгоритму для отримання відповіді.

Приклад 2.1. Нехай дана послідовність з нулів та одиниць і нам потрібно з'ясувати, чи є там хоч одна одиниця. Яку складність матиме алгоритм розв’язання цієї задачі?

Розв’язання. Нехай n – кількість символів в послідовності. Алгоритм буде послідовно перевіряти, чи немає одиниці в поточному місці заданої послідовності, а потім рухатися далі, поки вхід не скінчиться. Оскільки одиниця дійсно може бути тільки одна, для отримання точної відповіді на це питання в гіршому випадку доведеться перевірити всі n символів входу. В результаті отримуємо складність порядку cn, де c – кількість кроків, що потрібна для перевірки поточного символу і переходу до наступного. Оскільки

такого роду константи сильно залежать від конкретної реалізації, математичного сенсу вони не мають, і їх зазвичай ховають за символом O (O-велике): в даному випадку фахівець із теорії складності визначив би, що алгоритм має складність

$$O(n);$$

іншими словами, він лінійний.

Існує і продовжує розширюватися клас варіантів поняття складності: складність знизу, зверху й у середньому, алгебраїчна складність, мультиплікативна складність, бітова складність, фундаментальні асимптотичні оцінки складності, оцінка алгоритмів за їх належністю до класів складності самих проблем, що вони розв'язують, і т. д.

Одним зі спрощених видів аналізу складності алгоритмів, що використовують при комп'ютерній їх реалізації, є асимптотичний аналіз алгоритмів. Він використовується з метою порівняння витрат часу та інших ресурсів різноманітними алгоритмами, призначеними для вирішення одного і того самого завдання. Досліджуючи зростання часу роботи алгоритму при вхідних даних досить великих розмірів, ми тим самим вивчаємо асимптотичну ефективність алгоритмів. Це означає, що нас цікавить тільки те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних, коли цей розмір збільшується до нескінченності. Зазвичай алгоритм, більш ефективний в асимптотичному сенсі, буде більш продуктивним для всіх вхідних даних, за винятком дуже маленьких. Нижче перелічені основні оцінки складності.

Позначення, що вводяться для опису асимптотичної поведінки часу роботи алгоритму, використовують функції, область визначення яких – множина невід'ємних цілих чисел $N = \{0,1,2, ...\}$. Подібні позначення зручні для опису часу роботи $T(n)$ в найгіршому випадку як функції, визначеної тільки для цілих чисел, що становлять розмір вхідних даних.

Означення. Функція складності алгоритму $f(n)$ має оцінку Θ(тета) й записується як $f(n) = \Theta(g(n))$, якщо існує невід'ємна функція $g(n)$ та додатні $n0$, $c1$, $c2$ такі, що

$$c1g(n) \le f(n) \le c2g(n),$$

при $n > n0$.

У такому разі говорять, що функція $g(n)$ є асимптотично точною оцінкою функції $f(n)$, оскільки за визначенням функція $f(n)$ не відрізняється від функції $g(n)$ з точністю до сталого множника. Важливо розуміти, що $\Theta(g(n))$ є не однією функцією, а множиною функцій для опису зростання $f(n)$ з точністю до

сталого множника. Іншими словами, функція f(n) належить множині Θ (g(n)), якщо існують додатні c1 та c2 , що дозволяють обмежити цю функцію у рамки між функціями c1g(n) та c2g(n) для достатньо великих значень n.

Наприклад, для методу сортування послідовності чисел алгоритмом heapsort оцінка трудомісткості становить $f(n) = \Theta (n\log_2 n)$, тобто в цьому разі $g(n) = n\log_2 n$.

З означення f(n) = Θ (g(n)) випливає, що g(n) = Θ (f(n)).

Інтуїтивно зрозуміло, що при асимптотичній точній оцінці асимптотично невід'ємних функцій, доданками нижчих порядків у них можна знехтувати, оскільки при великих n вони стають неістотними. Навіть невеликої частки доданка найвищого порядку достатньо для того, щоб перевершити доданки нижчих порядків. Таким чином, для виконання нерівностей у визначенні Θ, достатньо як c1 вибрати значення, яке дещо менше коефіцієнта при самому старшому доданку, а як c2– значення, яке дещо більше цього коефіцієнта.

Тому коефіцієнт при старшому доданку можна не враховувати, тому що він лише змінює зазначені константи.

Приклад 2.2. Розглянемо квадратичну функцію $f(n) = an^2 + bn + c$, де a, b, c – константи, причому a > 0. Відкидаючи доданки нижчих порядків за перший та ігноруючи коефіцієнт a, одержимо $f(n) = \Theta (n^2)$.

Загалом кажучи, для будь-якого полінома p(n) маємо $p(n) = \Theta((n^d))$, де d – степінь полінома при $a^d > 0$. Оскільки будь-яка константа – це поліном нульового степеня, то сталу функцію можна записати як $\Theta(n^0)$ або Θ(1).

Означення. Функція складності алгоритму f(n) має оцінку O(«O-велике») й записується як f(n) = O(g(n)), якщо існує невід'ємна функція g(n) та додатні n0, c такі, що

$$0 \le f(n) \le cg(n),$$

при n > n0 .

Означення. Функція складності алгоритму f(n) має оцінку Ω(«омега-велике») й записується як f(n) = Ω(g(n)), якщо існує невід'ємна функція g(n) та додатні n0 і c такі, що 0 ≤, cg(n) ≤ f(n) при n > n0 .

O– позначення застосовуються, коли необхідно вказати верхню межу функції з точністю до сталого множника. В позначеннях теорії множин Θ(g(n)) ⊂ O(g(n)). Оскільки O-позначення описують верхню межу, то в ході їх використання для обмеження часу роботи алгоритму в найгіршому випадку ми отримуємо верхню межу цієї величини для будь-яких вхідних даних. Таким чином, асимптотична оцінка $O (n^2)$ для часу роботи алгоритму у найгіршому випадку застосовна для часу виконання завдання з будь-якими вхідними даними, чого не можна сказати про Θ-позначення. Наприклад, оцінка в $\Theta(n^2)$

для часу сортування вставками в найгіршому випадку не придатна для довільних вхідних даних. Наприклад, якщо вхідні елементи, що подаються на сортування, вже відсортовані в потрібному порядку, час роботи алгоритму сортування вставкою оцінюється як Θ(n).

Оскільки Ω - позначення використовуються для визначення нижньої межі часу роботи алгоритму в найкращому випадку, то вони визначають нижню межу часу роботи алгоритму при довільних вхідних даних. Наприклад, час роботи алгоритму сортування вставками знаходиться в межах між Ω(n) та O(n2), тобто між лінійною та квадратичною функціями від n.

Асимптотичний аналіз алгоритмів має не лише практичне, а й теоретичне значення. Так, наприклад, доведено, що всі алгоритми сортування, ґрунтуються на попарному порівнянні елементів, відсортують n елементів за час, не менший $\Omega(n\log_2 n)$.

1.3 Способи представлення алгоритмів

Алгоритм може задаватися різними способами:

– словесно, словами і реченнями природної мови – української, англійської тощо, в тому числі з використанням формул;

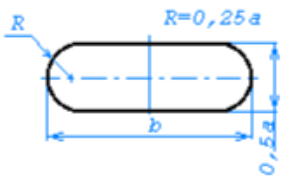
– графічно, за допомогою спеціальних схем із застосуванням умовних графічних позначень;

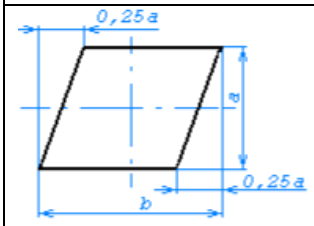
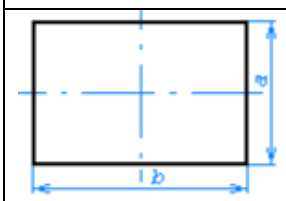
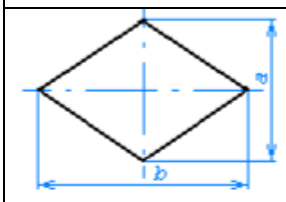
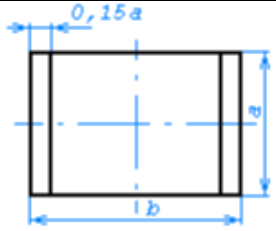
– символами спеціальної алгоритмічної мови.

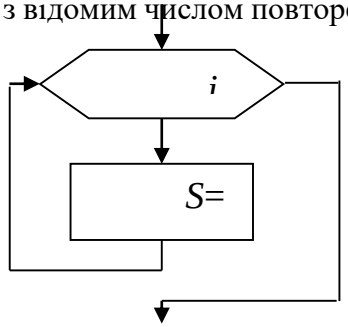
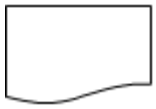
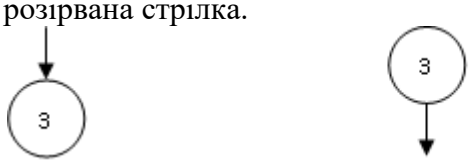
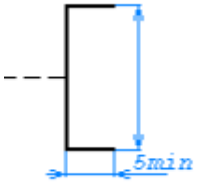
Запис алгоритму словами і реченнями природної мови найдоступніший для ширшого кола користувачів і не потребує спеціальної підготовки. Водночас така форма найменш точна та визначена, оскільки допускає різні тлумачення одного і того самого поняття внаслідок синонімічного багатства природних мов. Застосування спеціальних графічних схем для запису алгоритму дає змогу наочно бачити взаємозв'язок вхідних і вихідних величин на різних етапах виконання алгоритму. Крім того, алгоритм у такій формі доволі просто змінювати і коригувати.

В таблиці 1 наведені позначення стандарту, які використовуються для опису алгоритмів програм.

Таблиця 1.

Графічне зображення блоку	*Найменування*	*Функція*
	Термінатор (пуск-зупинка)	Позначення початку і кінця алгоритму. При описі основного алгоритму він поміщається вгорі схеми і містить слово «Початок», а також розміщується в кінці алгоритму зі словом «Кінець» всередині. Так повинен починатися і закінчуватися кожен алгоритм. Блоки початку і кінця в

Графічне зображення блоку	*Найменування*	*Функція*
		алгоритмі на схемі тільки один раз. При описі допоміжних алгоритмів (підпрограм) їх схема також обов'язково містить блок початку і кінця в єдиному екземплярі. Але замість слова «початок» в блоці вказується ім'я алгоритму і змінні, значення яких передаються з викликає алгоритму. У блоці завершення замість слова «кінець» наводяться змінні, значення яких повертаються в викликає алгоритм по закінченню роботи допоміжного алгоритму.
	Дані (введення-виведення)	Перетворення даних у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Використовується для позначення будь-якої операції введення / виведення. В такому блоці зазначаються змінні, значення яких повинен ввести користувач в даному місці виконання алгоритму.
	Процес (операторний блок)	Виконання операції або групи операцій у результаті яких змінюється значення, форма представлення або розташування даних. Типове його використання – позначення оператора присвоювання. Операторний блок - це прямокутник, в який вписується деяка дія або вираз. Цей блок може мати декілька входів і тільки один вихід, що забезпечує однозначність у визначенні послідовності виконуваних дій.
	Рішення (умовний блок)	В умовному операторі вказується умова (логічний вираз). З нього завжди виходить дві лінії, які позначені «+» і «-» (можлива позначка словами «так» і «ні»). Якщо умова істинна, то будуть виконуватися дії по стрілці, поміченої «+», інакше виконається оператор, до якого веде лінія, позначена «-».
	Зумовлений процес (підпрограма)	Відображає виконання процесу, який складається з однієї або кількох операцій, що визначені в іншому місці програми. Використовується для позначення виклику підпрограм (процедур і функцій). У блоці вказується ім'я допоміжного алгоритму (підпрограми) і значення змінних, які передаються в цей алгоритм. За типом і кількістю ці значення повинні збігатися зі змінними, вказаними в блоці

Графічне зображення блоку	*Найменування*	*Функція*
		початку допоміжного алгоритму.
	Цикл з відомим числом повторень	В такому блоці вказується змінна циклу, її початкове значення, кінцеве значення і величина кроку збільшення після кожного проходження циклу. Приклад схеми, що містить цикл з відомим числом повторень.  На схемі вказано цикл по знаходженню суми непарних елементів масиву. Стрілки позначають переходи між виконанням команд.
	Вивід на друк	В такому блоці вказується змінні або константи, значення яких виводиться на друк на даному етапі виконання алгоритму.
	З'єднувач	Відображає зв'язок між перерваними лініями потоку інформації на одній сторінці. При неможливості провести стрілку до блоку на одній сторінці (багато перетинів з іншими лініями) проводиться розрив лінії. Стрілка підводиться до даного блоку, в ньому вказується номер розриву. Потім в зручному для продовження місці вставляється даний блок з тим же номером і з нього триває розірвана стрілка. 
	Міжсторінковий з’єднувач	Застосовується аналогічно попередньому блоку, з тією відмінністю, що в ньому вказується сторінка, на яку виконується перехід, та, через крапку, номер переходу на цю сторінку. Наприклад 3.2 - другий перехід на третю сторінку.
	Коментар	Використовується для надання більш детальної інформації про кроки, процеси або групи процесів. Опис коментаря поміщається з боку квадратної дужки і охоплюється нею по всій висоті.

Графічне зображення блоку	*Найменування*	*Функція*
		Пунктирна лінія йде до описуваного елементу, або групи елементів. При цьому група виділяється замкнутою пунктирною лінією.
Розмір *a* повинен вибиратися з ряду 10, 15, 20 мм. Допускається збільшувати розмір a на число, кратне 5. Розмір b дорівнює 1,5a. При ручному виконанні схем алгоритмів допускається встановлювати b рівним 2a.		

Якщо операторний або умовний блоки мають більше одного входу, то зображення входів поєднується (рис. 1.1).

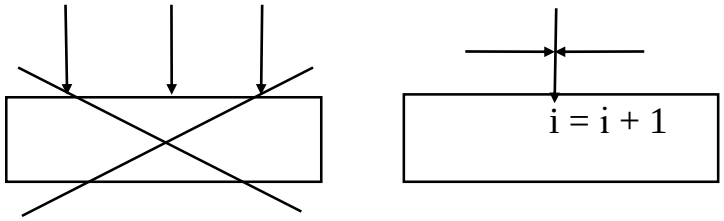
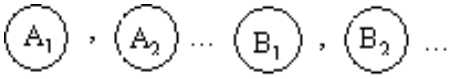


Рис.1.1 – Поєднання входів

Лінію потоку можна обривати, використовуючи на місці обриву *з'єднувачі*, якщо схему виконано на двох і більше аркушах або якщо символи, які з'єднуються, розташовано на значній відстані один від одного. Якщо схема розривається, то використовується з'єднувач, в середині якого вказують номер блоку (сторінки), до якого (-ої) або від якого (-ої) здійснюється перехід. Відповідні символи з’єднання повинні мати одне (унікальне) позначення. Нумерувати блоки можна тільки ті, які пов’язані з передачею управління та з перевіркою умов.

Якщо розв’язання задачі складається з декількох алгоритмів, що реалізують окремі процедури або функції, то в кожному алгоритмі використовується власна нумерація переходів:



На зв'язках, що визначають послідовність виконання блоків, стрілки не обов'язкові, якщо їх напрям відповідає просуванню "зверху-вниз" і "зліва-направо" і, якщо вони не мають зламів. В інших випадках їх напрямок обов'язково позначають стрілкою. Лінію потоку, як правило, підводять до середини символу. Відстань між паралельними лініями потоку має бути не меншою 3 мм, між іншими символами — не меншою 5 мм.

При складанні блок-схем необхідно керуватися наступними правилами:

– блок-схема повинна містити одну початкову, одну кінцеву й кінцеве число інших вершин;

– входи й виходи різних вершин з'єднуються дугами, спрямованими тільки від виходу до входу;

– кожен вихід будь-якої вершини з'єднується тільки з одним входом;

– для будь-якої вершини існує, принаймні, один шлях із цієї вершини до кінцевої, що проходить через інші вершини у напрямку з'єднуючих їх дуг.

Блок-схемна форма представлення алгоритму найбільш поширена, бо вона наочна і дозволяє представляти алгоритм з різним ступенем деталізації. Перевагою блок-схем є те, що за їх допомогою можна наочно продемонструвати структуру алгоритму в цілому, відобразивши його логіку (показавши розгалуження шляхів розв'язання задачі залежно від виконання певної умови, численні повторення окремих етапів обчислювального процесу).

2.4 Поняття програми. Мови програмування та їх складові елементи

Алгоритм, записаний спеціальною алгоритмічною мовою, може бути використаний на комп'ютері для розв'язку прикладних задач.

Запис алгоритму на формальній мові називається програмою. Іноді саме поняття алгоритму ототожнюється з його записом, так що слова "алгоритм" і "програма" - майже синоніми. Невелике розходження полягає в тому, що при згадці алгоритму, як правило, мають на увазі основну ідею його побудови, загальну для всіх алгоритмічних мов. Програма ж завжди пов'язана із записом алгоритму на конкретній формальній мові.

У математиці розглядаються різні види алгоритмів - програми для машин Тьюринга, алгоритми Маркова (нормальні алгоритми), частково рекурсивні функції і т.п. Знаменитий тезу Черча стверджує, що всі види алгоритмів еквівалентні один одному, тобто класи завдань, що вирішуються різними типами алгоритмів, завжди збігаються. Теза ця не може бути доведена (можна лише довести збіг для двох конкретних типів алгоритмів, наприклад, машин Тьюринга і нормальних алгоритмів), але ніхто в її вірності не сумнівається. Так що всі мови програмування еквівалентні один одному і розрізняються лише тим, наскільки вони зручні для вирішення конкретних класів задач. Наприклад, об'єктно-орієнтовані мови оптимальні для програмування в віконних середовищах, а мова Фортран успішно застосовується в наукових і інженерних розрахунках.

Програмування починалося з записи програм безпосередньо у вигляді машинних команд (в кодах, як кажуть програмісти). Пізніше для полегшення кодування була розроблена мова Асемблера, який дозволяє записувати машинні команди в символічному вигляді. Наприклад, програмісту не потрібно пам'ятати числовий код операції додавання, замість цього можна використовувати символічне позначення ADD. Мова Асемблера залежить від системи команд конкретного комп'ютера. Вона досить зручна для програмування невеликих завдань, що вимагають максимальної швидкості виконання. Однак великі проекти розробляти на мові Асемблера важко. Головна проблема полягає в тому, що програма, написана на Асемблері, прив'язана до архітектури конкретного комп'ютера і не може бути перенесена на інші машини. При удосконаленні комп'ютера всі програми на Асемблері доводиться переписувати заново.

Майже відразу з виникненням комп'ютерів були розроблені мови високого рівня, тобто мови, які не залежать від конкретної архітектури. Для виконання програми на мові високого рівня її потрібно спочатку перевести на мову машинних команд. Спеціальна програма, що виконує такий переклад, називається транслятором або компілятором. Відкомпільована програма компонується у код, а потім виконується безпосередньо комп'ютером. Існує також можливість перекладу програми на проміжну мову, що не залежить від архітектури конкретного комп'ютера, але тим не менше максимально наближена до мови машинних команд. Потім програма на проміжній мові виконується спеціальною програмою, яка називається інтерпретатором. Можливий також варіант компіляції "на льоту" (Just In Time Compilation), коли виконується фрагмент програми перекладається з проміжної мови на мову машинних команд безпосередньо перед виконанням.

Мову програмування, як і будь-яку формальну мову, визначає три її складові: алфавіт, синтаксис (граматика) і семантика, що задають зовнішній вигляд програми і дії, які виконує комп'ютер під її управлінням.

Алфавіт мови програмування (її термінальний словник) - це фіксований набір символів, які дозволяється використовувати для утворення нетермінальних конструкцій мови (її нетерміналів). Синтаксичні визначення (правила) встановлюють правила побудови допустимих конструкцій мови із символів її алфавіту. Семантика визначає зміст і правила використання мовних конструкцій, для яких були дані синтаксичні визначення. Семантичні правила пояснюють, яке смислове значення має кожна мовна конструкція і які дії повинен виконати комп'ютер під час їх виконання (виконання команди програми).

Порушення форми запису програми приводить до її нерозуміння комп’ютером і видачі повідомлення про синтаксичну помилку, а використання правильно написаних команд, але таких, що не відповідають необхідній послідовності дій, приводить до семантичних помилок (їх ще називають логічними помилками, а також помилками часу виконання).

Описом мови є опис чотирьох названих вище елементів: опис символів полягає в перерахуванні припустимих символів мови; під описом елементарних конструкцій розуміють правила їхнього утворення; опис виразів – це правила утворення будь-яких виразів, що мають зміст у даній мові; опис операторів складається з розгляду усіх типів операторів, припустимих у мові.

При описі мови програмування і при побудові алгоритмів використовуються певні поняття мови. Такий підхід, до речі, застосовується і в шкільних курсах, зокрема, геометрії. Так, щоб визначити, що таке «прямокутний трикутник», треба ввести поняття «трикутник», «кут», «прямий кут», тощо. В будь-якій мові програмування можна виділити наступні базові поняття:

- алфавіт - зазвичай, являє собою підмножину набору символів коду ASCII
- лексеми,
- елементи даних,
- операції,
- вирази,
- оператори.

Лексеми. Із символів алфавіту будуються лексеми - мінімальні значимі одиниці мови, що мають певний зміст (фактично це слова). Множина всіх допустимих лексем називається *словником* мови програмування (нетермінальний словник мови).

Розрізняють наступні основні види лексем.

Ідентифікатори. Це слова, що служать для завдання імен програмних об'єктів (імен змінних, констант, підпрограм, програм тощо).

Правила побудови ідентифікаторів для різних мов програмування можуть бути різними.

У мовах програмування має місце різниця у сприйнятті рядкових і прописних літер алфавіту. Так у мові C/C++ прописні і рядкові букви сприймаються як різні символи, тобто myvar і MyVar – різні імена, тоді як у Pascal ідентифікатори байдужі до регістра клавіатури.

У будь-якому випадку рекомендується вибирати змістовні ідентифікатори, використовуючи для наочного їх представлення прописні і рядкові літери. Наприклад, замість ідентифікатора nomerotdela, краще

написати DepartmentNumber, виділивши прописними буквами кожну із двох змістовних частин.

Зарезервовані (ключові) слова. Це слова, що мають у мові програмування певне призначення, яке не може змінюватися. Зарезервовані слова використовуються для позначення алгоритмічних конструкцій, розділів програми тощо.

Літерали (рядки символів або просто текст). Це певним чином оформлена послідовність символів коду ASCII.

Коментарі. Це частина тексту програми, що виконує чисто інформаційну функцію, тобто, служить для пояснення використання об'єктів програми і дій над ними. Коментар являє собою певним чином виділену послідовність символів (не обов'язково з алфавіту мови), яка не обробляється компілятором.

Специфічним видом лексем, що використовуються у програмах, зокрема, на мові C/C++, є управляючі послідовності (Esc-послідовності) - спеціальні символьні комбінації, які використовуються у функціях введення і виведення інформації. Вони, зазвичай, будуються на основі використання символа «\» (обов'язковий перший символ) і комбінації латинських літер і цифр. Наприклад, у мові C/C++ \n - символ нового рядка.

Дві сусідні лексеми повинні бути відокремлені одна від одної одним і декількома роздільниками, до числа яких належать пробіл, символи горизонтальної та вертикальної табуляції, символ нового рядка тощо. Виняток - рядки, які можуть включати символ пропуску.

Елементи даних. Елементи даних – це найменші неподільні одиниці даних. Елементами даних, що обробляються в програмах, є константи і змінні.

Константи - дані, значення яких в процесі роботи програми залишаються незмінними (постійними). Константи характеризуються своїм значенням і, можливо, ім'ям (типізовані константи).

У найпростішому випадку константа являє собою число (123, 2.87), символ (‘A’), літерал ("це рядок"), логічне значення (TRUE, FALSE). Числа, що використовуються в програмах, можуть бути цілими або дійсними; додатними або від’ємними. Дійсні числа можуть бути представлені у формі з фіксованою або плаваючою крапкою.

Цілі числа можуть бути представлені у різних системах числення. Зокрема, у мові Pascal вони можуть задаватися у десятковій або шістнадцятковій системах числення; у мові C/C++ - не тільки у десятковій та шістнадцятковій (починаються з префікса 0x або 0X) системах числення, а й у вісімковій (починаються з префікса нуль - 0). Наприклад, 074, 0x2B.

Змінні - це дані, значення яких в процесі роботи програми можуть змінюватися. Змінні характеризуються ім'ям (ідентифікатором) і значенням (початковим, поточним).

Для кращого розуміння будемо вважати змінною іменовану комірку в пам’яті, значення в якій може змінюватися.

Операції. Для завдання дій над даними служать операції.

Вирази. Вираз – це синтаксична конструкція мови, яка складається із даних (операндів) та знаків операцій і служить для обчислення значення невідомої величини.

Оператори (інструкції).

Оператор - це команда мови програмування, якою задається певний крок процесу обробки інформації на ЕОМ.

Контрольні питання

1. Охарактеризувати поняття алгоритму, його властивості. Навести приклади.
2. Проаналізувати можливі способи представлення алгоритмів, в тому числі такі способи як операторний, граф-схеми, HIPO-схеми.
3. Вивчити класифікацію сучасних мов програмування та навести їх приклади.

Практичні завдання

1. Опишіть за допомогою операторів блок-схем наступні дії:
   1.1 Цикл з передумовою, який виконується n раз.
   1.2 Цикл з післяумовою, який виконується n раз.
   1.3 Цикл з відомою кількістю повторів, який виконується n раз.
   1.4 Виведення усіх чисел, починаючи з одиниці, до числа, введеного користувачем.
   1.5 Оператор розгалуження, який обчислює формулу

$$f(x)=\begin{cases} x^2+8, & \text{якщо } -5<x; \\ -x^3+2, & \text{у протилежному випадку} \end{cases}$$

   1.6 Оператори, які дозволяють знайти мінімальне значення між двома дійсними числами x та y.
   1.7 Оператори, які дозволяють знайти максимальне значення серед трьох заданих чисел, що зберігаються в змінних a, b, c. Максимальне значення записується у змінну max.

2. Реалізуйте складені алгоритми за допомогою мови програмування та доведіть їх правильність.

3 ТИПИ ДАНИХ ТА ОПЕРАЦІЇ НАД НИМИ

3.1 Поняття змінної

Будь-який алгоритм, реалізований на комп'ютері, призначений для обробки певної інформації. Ця інформація має зберігатися у пам'яті комп'ютера. Для обробки інформації в пам'яті комп'ютера було введено поняття змінної.

Змінна в програмуванні – частина (комірка) пам'яті комп'ютера, що отримала від програміста певне ім'я, та інформацію в якій можна змінювати. Основними операціями зі змінною є:

– запис у змінну нової інформації;

– читання інформації, що в ній записана.

Запис нової інформації у змінну здійснюється спеціальною операцією – присвоювання.

Операція присвоювання знищує у змінній стару інформацію і записує нову. Операція присвоювання позначається := або просто = (для мов програмування, що вийшли з С). Надалі будемо операцію присвоювання будемо позначати =.

Загальний вид операції присвоювання:

ім'я змінної = значення;

В інших операціях зі змінною завжди проводиться лише зчитування з неї даних.

Наприклад:

```
x = 2;
x = x+5;
```

Першою операцією в змінну x було записано значення 2. У другій операції спочатку було зчитано значення з x (там зберігалося 2), до нього додали 5 и результат 7 знову записали в x.

Як відомо, вся інформація в комп'ютері записується в бітах та байтах за допомогою 0 та 1. Але в комп'ютері зберігаються різні числа, тексти, зображення та інша інформація. Для того, щоб знати, яка інформація записана в комірці памяті, введено типи даних, базовими з яких є цілий, дійсний, логічний та текстовий тип.

3.2 Тип цілих чисел

Тип ціле число є основним для будь-якого алгоритмічної мови. Пов'язано це з тим, що вміст комірки пам'яті або регістра процесора можна розглядати як ціле число. Адреси елементів пам'яті також є цілі числа, з їх допомогою записуються машинні команди і т.д. Символи представляються в комп'ютері цілими числами - їх кодами в деякій кодуванні. Зображення також задаються масивами цілих чисел: для кожної точки кольорового зображення зберігаються інтенсивності її червоною, зеленою і синьою складової (в більшості випадків - в діапазоні від 0 до 255). Як кажуть математики, цілі числа дано зверху, все інше сконструювала з них людина.

Загальноприйнятий в програмуванні термін ціле число або цілочисельна змінна, строго кажучи, не цілком коректний. Цілих чисел нескінченно багато, десяткова або двійковий запис цілого числа може бути як завгодно довгою і не поміщатися в області пам'яті, відведеної під одну змінну. Ціла змінна в комп'ютері може зберігати лише обмежену множину цілих чисел в деякому інтервалі. У сучасних комп'ютерах під цілу змінну відводиться 4 байта, тобто 32 двійкових розряди. Вона може зберігати числа від нуля до 2 в 32-му ступені мінус 1. Таким чином, максимальне ціле число, яке може зберігатися в цілочисельній змінній дорівнює:

$$2^{32} - 1 = 4294967295$$

Розряди (біти) цілого числа нумеруються від 0 до 31. Старший розряд з номером 31 зазвичай зберігає знак числа: 0 – позитивне число, 1 – негативне число. Якщо результат операції над цілими числами не вміщується в біти від 0 до 30, то може бути зачеплений знаковий розряд. Таким чином сума двох великих позитивних чисел може дати від'ємне число, або сума двох від’ємних чисел – позитивне число. Таку ситуацію називають **переповненням**.

Основні операції над значеннями цілого типу:

- додавання;
- віднімання;
- перемноження;
- ділення (результат – ціла частина);
- залишок від ділення.

Зверніть увагу!

Позначення операцій залежить від мови програмування. У мовах програмування сімейства С це відповідно +,-,,/,%.*

3.3 Тип дійсних чисел.

До дійсних належать числа з дробовою частиною. Дійсні константи записуються в двох формах - з фіксованою десятковою крапкою або в експоненційному вигляді. У першому випадку точка використовується для поділу цілої та дробової частин константи. Як ціла, так і дрібна частини можуть бути відсутні.

Приклади: 1.2, 0.725, 1., .35, 0.

У трьох останніх випадках відсутня або дробова, або ціла частина. Десяткова точка повинна обов'язково бути присутньою, інакше константа вважається цілою. Відзначимо, що в програмуванні саме точка, а не кома, використовується для відділенні дробової частини.

Експоненціальна форма запису речової константи містить знак, мантису і десятковий порядок (експоненту).

Мантиса - це будь-яка позитивна матеріальна константа в формі з фіксованою точкою або ціла константа. Порядок вказує ступінь числа 10, на яку домножується мантиса. Порядок відділяється від мантиси буквою "e" (від слова exponent), вона може бути великою або малою. Порядок може мати знак плюс або мінус, в разі позитивного порядку знак плюс можна опускати.

Приклади:

1.5e + 6 константа еквівалентна 1500000.0

1e-4 константа еквівалентна 0.0001

-.75E3 константа еквівалентна -750.0

Дійсні числа представляються в пам’яті комп'ютеру в так званій експоненційній, або плаваючій, формі. Дійсне число r має вигляд

$$r = \pm 2^p * m$$

Представлення дійсного числа складається з трьох елементів:

1. Знак числа - плюс або мінус. Під знак числа відводиться один біт в двійковому поданні, він розташовується в старшому, тобто знаковому розряді. Одиниця відповідає знаку мінус, тобто негативного числа, нуль - знаку плюс. У нуля знаковий розряд також нульовий;

2. Показник ступеня *p*, його називають порядком. Порядок вказує ступінь двійки, на яку домножається число. Порядок може бути як позитивним, так і негативним (для чисел, менших одиниці). Під порядок відводиться фіксоване число двійкових розрядів, зазвичай вісім або одинадцять, розташованих в старшій частині двійкового представлення числа, відразу слідом за знаковим розрядом;

3. Мантиса m являє собою фіксовану кількість розрядів двійкового запису дійсного числа в діапазоні від 1 до 2: $1 \leq m < 2$

Зверніть увагу!

Ліва нерівність нечітка - мантиса може дорівнювати одиниці, а права - сувора, мантиса завжди менше двох. Розряди мантиси включають один розряд цілої частини, який з огляду на наведену нерівність завжди дорівнює 1, і фіксована кількість розрядів дробової частини. Оскільки старший двійковий розряд мантиси завжди дорівнює одиниці, зберігати його необов'язково, і в двійковому коді він відсутній. Фактично двійковий код зберігає тільки розряди дробової частини мантиси.

У сімействі мови програмування С дійсним числам відповідають типи float і double. Елемент типу float займає 4 байта, в яких один біт відводиться під знак, вісім - під порядок, інші 23 - під мантису (насправді, в мантисі 24 розряду, але старший розряд завжди дорівнює одиниці, тому зберігати його не потрібно). Тип double займає 8 байтів, в них один розряд відводиться під знак, 11 - під порядок, інші 52 - під мантиссу. Насправді в мантисі 53 розряду, але старший завжди дорівнює одиниці і тому не зберігається. Оскільки порядок може бути позитивним і негативним, в двійковому коді він зберігається в зміщеному вигляді: до нього додається константа, рівна абсолютній величині максимального по модулю негативного порядку. У разі типу float вона дорівнює 127, в разі double - 1023. Таким чином, максимальний по модулю негативний порядок представляється нульовим кодом.

Основним типом є тип double, саме він найбільш природний для комп'ютера. У програмуванні слід по можливості уникати типу float, так як його точність недостатня, а процесор все одно при виконанні операцій перетворює його в тип double.

Операції над дійсними значеннями:

- додавання +
- віднімання -
- перемноження *
- ділення /

Зверніть увагу!

Є певні особливості виконання арифметичних операцій з дійсними числами.

При виконанні додавання двох позитивних плаваючих чисел відбуваються такі дії:

1. Вирівнювання порядків. Визначається число з меншим порядком. Потім послідовно його порядок збільшується на одиницю, а мантиса ділиться на 2, поки порядки двох чисел не зрівняються. Апаратно розподіл на 2 відповідає зрушенню двійкового коду мантиси вправо, так що ця операція виконується швидко. При зрушеннях праві розряди губляться, через це може відбутися втрата точності (в разі, коли праві розряди ненульові).

2. Складання мантис.

3. Нормалізація: якщо мантиса результату дорівнює або перевищила двійку, то порядок збільшується на одиницю, а мантиса ділиться на 2. В результаті цього мантиса потрапляє в інтервал від 1 до 2. При цьому можлива втрата точності, а також переповнення, коли порядок перевищує максимально можливу величину.

Віднімання проводиться аналогічним чином. При множенні порядки складаються, а мантиси перемножуються як цілі числа, після чого у результату праві розряди відкидаються.

Дії з плаваючими числами через помилки округлення лише наближено відображають арифметику справжніх дійсних чисел. Так, якщо до великого дійсного числа додати дуже маленьке, то воно не зміниться. Дійсно, при вирівнюванні порядків все значущі біти мантиси меншого числа можуть вийти за межі розрядної сітки, в результаті чого воно стане рівним нулю. Наприклад, для типу double значення $1.0+10^{-16}$ буде дорівнювати 1.0. Це необхідно враховувати при організації процесу обчислень.

Крім втрати точності, при операціях з дійсними числами можуть відбуватися й інші неприємності:

1. Переповнення - коли порядок результату більше максимально можливого значення. Ця помилка часто виникає при множенні великих чисел.

2. Зникнення порядку - коли порядок результату негативний і дуже великий за абсолютною величиною, тобто порядок менше мінімально допустимого значення. Ця помилка може виникнути при діленні маленького числа на дуже велике або при множенні двох дуже маленьких за абсолютною величиною чисел.

Крім того, некоректною операцією є поділ на нуль. На відміну від операцій з цілими числами, переповнення і зникнення порядку вважаються помилковими ситуаціями і призводять до апаратного переривання роботи процесора. Програміст може задати реакцію на переривання - або аварійне завершення програми, або, наприклад, при переповненні привласнювати результату спеціальне значення плюс або мінус нескінченність, а при зникненні порядку - нуль.

Є спеціальна константа Not a Number, або NaN - двійковий код, який не є коректним поданням будь-якого дійсного числа.

Будь-які операції з константою NaN призводять до переривання, тому вона зручна при налагодженні програми - нею перед початком роботи програми ініціюються значення всіх дійсних змінних. Якщо в результаті помилки програміста при обчисленні виразу використовується змінна, якій не було присвоєно ніякого значення, то відбувається переривання через операцію зі значенням NaN і помилка швидко відстежується. На жаль, в разі цілих чисел такої константи немає: будь-який двійковий код представляє деяке ціле число.

3.4 Логічний тип

Логічний тип пов'язаний з результатами виконання операцій порівняння (відношень), до яких відносять:

- більше >
- більше або дорівнює >=
- менше <
- менше або дорівнює <=
- дорівнює ==
- не дорівнює !=

Зверніть увагу!

Вище наведені позначення прийняті в мовах програмування сімейства C, в інших мовах вони можуть бути відмінними.

Результат порівняння може бути істинним або хибним, відповідно true або false, які у пам’яті комп’ютера задаються значеннями 1 або 0.

Зі значеннями логічного типу можливе виконання специфічних логічних операцій, до яких зазвичай відносять:

- заперечення (ні)
- логічне «і» (кон’юнкція)
- логічне «або» (диз’юнкція)

В алгебрі логіки ці операції зазвичай позначається відповідно $\overline{A}$, A·B , A $\vee$ B.

Їх таблиці значень мають вид:

A	$\overline{A}$
0	1
1	0

A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	AvB
0	0	0
0	1	1
1	0	1
1	1	1

Всі інші логічні операції можуть бути реалізовані за допомогою наведених.

При розробці програм доцільно пам’ятати наступні логічні закони:

1. $p \vee 1 = 1,\ p \vee 0 = p$
2. $p \cdot 1 = p,\ p \cdot 0 = 0$
3. $p \cdot p = p,\ p \vee p = p$
4. $p \vee \bar{p} = 1,\ p \cdot \bar{p} = 0$
5. $p \vee q = q \vee p,\ p \cdot q = q \cdot p$
6. $p \cdot (q \vee r) = p \cdot q \vee p \cdot r$

   $p \vee q \cdot r = (p \vee q) \cdot (p \vee r)$
7. Закони де Моргана: $\overline{(p \vee q)} = \bar{p} \cdot \bar{q},\ \overline{(p \cdot q)} = \bar{p} \vee \bar{q}$
8. $p \Rightarrow q = \bar{p} \vee q$

   $p \Leftrightarrow q = (p \Rightarrow q)(q \Rightarrow p)$

   $p \oplus q = \overline{p \Leftrightarrow q}$

   $p \downarrow q = \overline{(p \vee q)}$

   $p \mid q = \overline{(p \cdot q)}$
9. Закони поглинання:

   $p \cdot (p \vee q) = p$

   $p \vee p \cdot q = p$

   $p \vee \bar{p} \cdot q = p \vee q$
10) Закони склеювання:

$p \cdot q \vee \bar{p} \cdot q = q,\ (p \vee q)(\bar{p} \vee q) = q$

11) Закон подвійного заперечення: $\bar{\bar{p}} = p$.

В мовах програмування сімейства С

– заперечення позначається !

– логічне «і» позначається &&

– логічне «або» позначається ||

Логічні змінні та операції часто використовуються в умовних операторах та операторах циклів.

3.5 Символьний тип

Значенням символьної змінної є один символ з фіксованого набору. Такий набір зазвичай включає букви, цифри, розділові знаки, знаки математичних операцій і різні спеціальні символи (відсоток, амперсанд, зірочка, коса риска та ін.). Підкреслимо, що, на відміну від строкової змінної, символьна завжди містить рівно один символ (строкова містить рядок з декількох символів.)

Зверніть увагу!

В пам'яті комп'ютера ніяких символів не міститься. Символи представляються їх цілочисельними кодами в деякому фіксованому коді.

Кодування визначається трьома параметрами:

1. Діапазоном значень кодів. Наприклад, найпоширеніша в світі модель кодування ascii (від слів american standard code of information interchange - американський стандартний код обміну інформацією) має діапазон значень кодів від 0 до 127, тобто вимагає сім біт на символ. Більшість сучасних кодувань мають діапазон кодів від 0 до 255, тобто один байт на символ. Нарешті, зараз у всьому світі здійснюється перехід на кодування unicode, яка використовує коди в діапазоні від 0 до 65535, тобто 2 байта на символ.

2. Множиною зображуваних символів. Наприклад, кодування ASCII містить букви латинського алфавіту, в західноєвропейському кодуванні до символів ASCII додані літери з Умлаут і акцентами, додаткові знаки пунктуації, зокрема, іспанські перевернуті знаки запитання й оклику, і інші символи європейських мов, заснованих на латинській графіці. Будь-яка з російських кодувань містить кирилицю.

3. Відображенням множини кодів на множину символів. Наприклад, російські кодування КОИ-8 (Код обміну інформацією восьмибитовий) і "Windows CP-1251", які традиційно використовуються в операційних системах Unix і MS Windows, мають один і той же діапазон кодів і один і той же набір символів, але відображення їх різні (одні і ті ж символи мають різні коди в кодуваннях ЯКІ-8 і Windows).

Існування різних кодувань букв кирилиці сильно ускладнює життя як програмістам, так і звичайним користувачам: файли при перенесенні з однієї системи в іншу доводиться перекодувати, періодично виникають труднощі при читанні листів, перегляді гіпертекстових сторінок і т.п. Відзначимо, що нічого подібного немає ні в одній європейській країні.

З повсюдним переходом на кодування Unicode всі проблеми такого роду повинні зникнути. Кодування Unicode включає символи алфавітів всіх європейських країн і кирилицю. На жаль, більшість існуючих комп'ютерних програм пристосоване до подання одного символу у вигляді одного байта. Тому в даний час часто використовується проміжне рішення: комп'ютерні програми працюють з внутрішнім поданням символів в кодуванні Unicode (таке рішення прийнято в мовах Java і C #). При записи в файл символи Unicode приводяться до однобайтового коду відповідно до поточної мовної установки. При цьому, звичайно, частина символів втрачається - наприклад, в кодуванні Windows неможливо одночасно записати російські букви і німецькі Умлаут, оскільки Умлаут в західно-європейської кодуванні мають ті ж коди, що і російські літери в російському кодуванні.

Ще однією особливістю обробки символів є те, що символи, які мають однакове зображення, мають різні коди (наприклад, англійська A не дорівнює А в кирилиці), що часто призводить до різноманітних проблем.

Контрольні питання

1. Охарактеризувати поняття змінної у програмуванні, навести відмінності від математичної змінної.
2. Розглянути поняття знакових та без знакових цілих типів, операції зсуву.
3. Вивчити завдання та області застосування побітових логічних операцій.

Практичні завдання

1. Реалізуйте вказані задачі мовою програмування. Проаналізуйте використані типи даних та дії над ними:
   1.1 Оголосити змінні за допомогою яких можна буде порахувати загальну суму покупки декількох товарів. Наприклад плитки шоколаду, кави і пакети молока;
   1.2 Є змінна x дійсного типу з початковим значенням 10 і змінна y дійсного типу, яка вводиться з клавіатури. Якщо змінна y більше x, то подвоїти y, інакше зменшити y на величину x;

1.3 Оголосити три змінні типу int і присвоїти першій числове значення, друга змінна дорівнює першій змінній збільшеній на 3, а третя змінна дорівнює сумі перших двох.

2. Визначте, які типи даних треба використати для представлення наступних значень:
   - 20;
   - 20.0;
   - «20.0»;
   - «20/3»;
   - вартість товару;
   - дата народження;
   - колір автомобіля;
   - колір екранного пікселя;
   - стать людини;
   - форма навчання студента.

Обґрунтуйте свій вибір. Запропонуйте альтернативні варіанти представлення там, де це можливо.

4 БАЗОВІ АЛГОРИТМІЧНІ СТРУКТУРИ

Алгоритми можна представляти як деякі структури, що складаються із окремих базових (тобто основних) елементів – *базових алгоритмічних структур*. Кожен такий елемент відповідає певному типу алгоритмічного процесу.

Базові алгоритмічні структури:

- лінійна (слідування),
- розгалуження,
- циклу.

4.1 Лінійна структура

Лінійний алгоритм складається з однієї чи декількох дій (розпоряджень виконавцю), що повинні бути виконані у строгій послідовності, без всяких умов і в строгій відповідності з тим порядком, у якому записані оператори програми.

Типовим прикладом лінійного алгоритму є процедура обчислення за певними формулами.

Слід зауважити, що обчислення виразів є досить складним процесом, який потребує багато часу та спеціальної додаткової пам’яті для збереження проміжних результатів. Порядок обчислень у кожній мові визначається за пріоритетом операцій та скобками, використаними у записі виразу. Ці вирази можуть бути настільки складними, що можуть викликати збої в роботі програми. Крім того, довгі вирази погано читаються у коді програми. Тому доцільно розбивати складний вираз на декілька більш простих, пам’ятаючи, що запис виразу будь-якої складності повинен бути лінійним. Кожен вираз бажано розміщати тільки в одному рядку програми.

У якості прикладу лінійного алгоритму розглянемо пошук площі стін прямокутної кімнати:

1. Ввести довжину кімнати A.
2. Ввести ширину кімнати B.
3. Ввести висоту кімнати H.
4. Розрахувати площу стін S по формулі: S= 2*(A + B)*H.
5. Вивести значення змінної S як результат.

4.2 Розгалужені алгоритми

У випадках, коли перетворення інформації може здійснюватись за різними схемами, залежно від властивостей вхідних даних або проміжних

результатів, використовуються *розгалужені алгоритми*. В таких алгоритмах передбачаються всі можливі варіанти обробки інформації, кожний з яких розробляється як окрема гілка алгоритму, а вибір однієї з них для виконання здійснюється за допомогою перевірки деякої умови, що відображає властивості інформації, яка використовується у процесі перетворення.

Для представлення таких алгоритмів використовуються алгоритмічні конструкції *вибору* (*розгалуження*). Дана алгоритмічна структура в залежності від результату перевірки певної умови здійснює вибір одного з альтернативних шляхів роботи алгоритму. Кожен із шляхів веде до спільного виходу, так що робота алгоритму триватиме незалежно від того, який шлях буде обраний.

Структура розгалуження існує в чотирьох основних варіантах:

- «якщо-то-інакше» (повне розгалуження);
- «якщо-то» (неповне розгалуження);
- «вибір-інакше» (повний багатоваріантний вибір);
- «вибір» (неповний багатоваріантний вибір).

4.2.1 Повне розгалуження.

Дана алгоритмічна структура передбачає виконання дій і у разі виконання, і у разі невиконання заданої умови. Графічне представленням такої структури у блок-схемах має вид, наведений на рисунку 4.1.

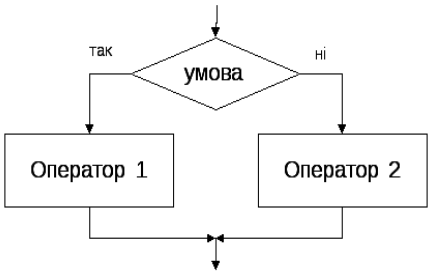


Рис. 4.1 - Загальний вид оператора розгалуження

При цьому умова формулюється таким чином, щоб відповідь перевірки була «так» чи «ні». Як правило, при запису умови використовуються операції порівняння та логічні операції. Для спрощення запису умов доцільно використовувати формули алгебри логіки.

Зазначимо, що у якості оператору 1 чи оператору 2 може виступати не одна дія, а кілька, як представлено на рисунку 4.2.

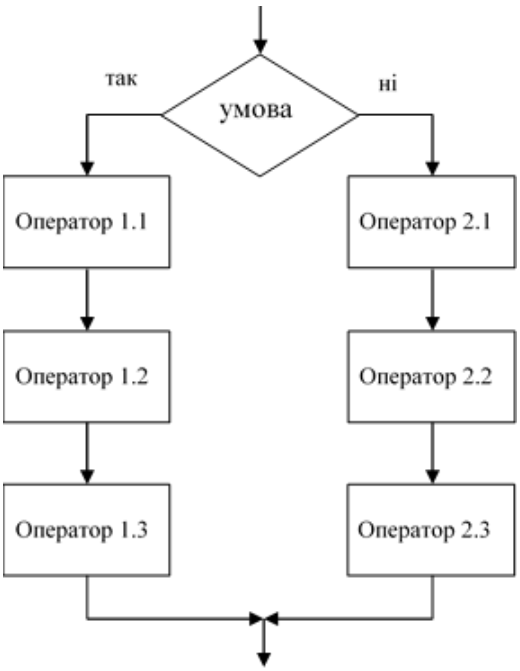


Рис. 4.2. – Блок-схема повного розгалуження

4.2.2 Неповне рогалуження.

Дана алгоритмічна структура передбачає виконання дій тільки у разі виконання, або у разі невиконання заданої умови. Тобто, одна із її гілок взагалі не передбачає ніяких дій. Графічне представленням таких структур у блок-схемах має наступний вид, представлений на рисунку 4.3.

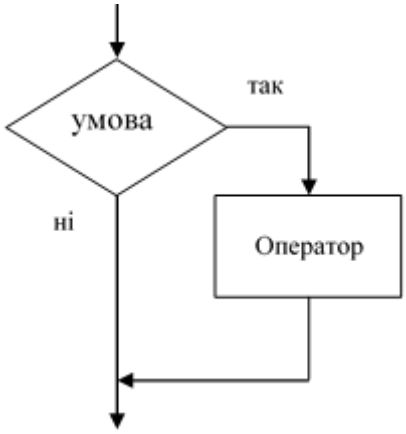


Рис. 4.3 – Блок-схема неповного розгалуження

Зверніть увагу!

У випадку, оператор має бути виконаний у разі невиконання умови, використовується логічна операція «ні» в умові.

Залежно від того, на скільки гілок розгалужується алгоритм, він може бути простим або складним. Для простого розгалуженого процесу

перевіряється одна умова, для складного — дві чи більше умов, кожна з яких відокремлює одну гілку (див. рис. 4.4).

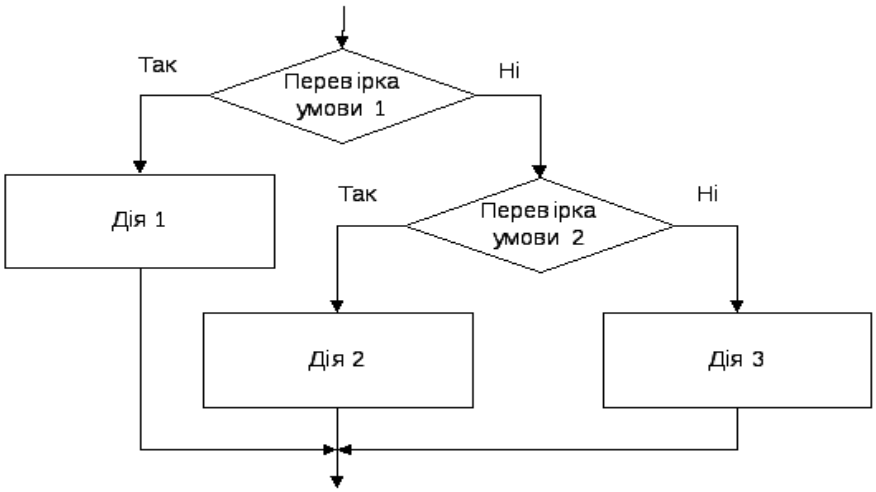


Рис. 4.4 – Блок-схема складного розгалуження

Складне розгалуження реалізується в мовах програмування вкладенням умовних операторів.

4.2.3 Багатоваріантний вибір.

Збільшення кількості умов робить алгоритм більш заплутаним, він втрачає наочність, перевірити його правильність досить складно. У таких випадках необхідно перехід до будь-якої гілки розгалуженого алгоритму пов’язати з деякою змінною, кожне значення якої відповідатиме одній із гілок розгалуження, тобто одному з варіантів обробки інформації. Тоді всі логічні блоки алгоритму об’єднуються в один блок аналізу цієї змінної, який матиме не два виходи, а стільки, скільки існує варіантів обробки. Таке розгалуження називають *багатоваріантним* (див. рис 4.5). В програмуванні такій конструкції зазвичай відповідає оператор switch.

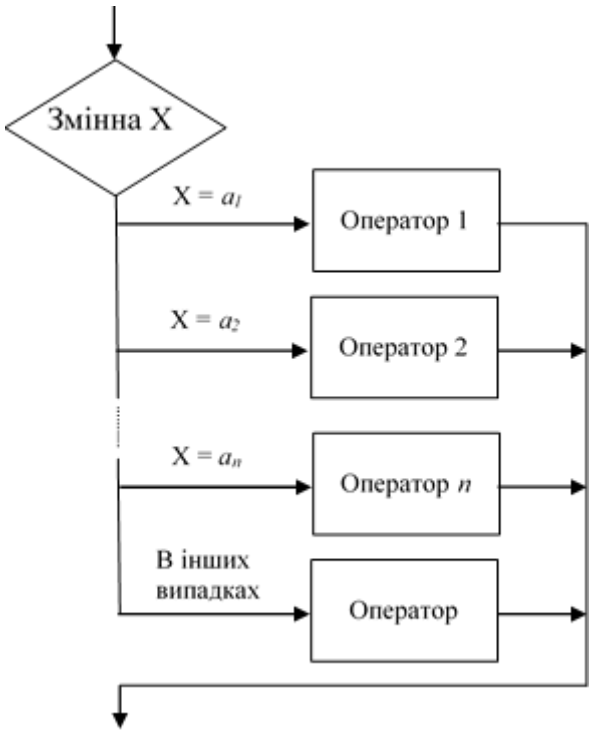


Рис. 4.5 – Блок-схема багатоваріантного вибору

4.3 Цикли

У деяких алгоритмах передбачається можливість багаторазового виконання деякої сукупності дій. Такі алгоритми називають *циклічними* (циклом), а їх повторювану частину - *тілом циклу*.

Для побудови циклічного алгоритму необхідно:

- визначити всі дії, які необхідно виконати до входу в цикл, тобто провести підготовку циклу;
- визначити всі операції, які ввійдуть до циклу;
- скласти умову повторення виконання операцій циклу або виходу з циклу.

Для представлення циклічних алгоритмів використовуються алгоритмічні конструкції повторення, які реалізуються одним із трьох наведених нижче способів.

4.3.1 Цикли з параметром

Якщо в процесі перетворення інформації є змінна, значення якої змінюється за відомим правилом, або відомі межі її зміни, то можна визначити кількість повторень ітерацій циклу та організувати вихід із циклічного процесу. Такі цикли називають *циклами з параметром* (арифметичним

циклом), а відповідну змінну - *параметром циклу*. Ще одна назва – *цикл з відомою кількістю повторювань.*

Графічне представленням циклу з параметром подане на рис. 4.6.

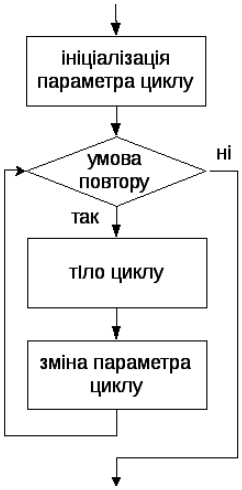
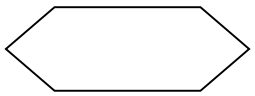


Рис. 4.6 – Блок-схема циклу з параметром

Параметр циклу – змінна, що визначає кількість повторювань циклу.

Для скорочення схеми можна використовувати спеціальний символ циклу з відомою кількістю повторювань:



В цьому елементі вказуються:

- параметр циклу
- початкове значення параметру
- кінцеве значення параметру
- крок (якщо він відмінний від 1).

Вважається, що при першому вході в цей блок параметр отримує своє початкове значення і перевіряється умова, що значення параметру не перевищує кінцевого значення. Якщо це так, виконується тіло циклу, після чого значення параметру збільшується на заданий крок та знову перевіряється умова (значення параметру не перевищує кінцевого значення. Коли значення параметру стане більшим від кінцевого значення, цикл завершується і наступним буде виконуватися оператор після циклу.

На рисунку 4.7 у прикладі показано знаходження суми значень функції sin від непарних значень аргументу в діапазоні від 1 до 100.

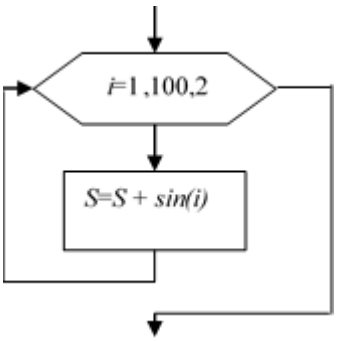


Рис. 4.7 - Приклад циклу з відомою кількістю повторювань

В мовах програмування циклу з відомою кількістю повторювань відповідає конструкція **for**.

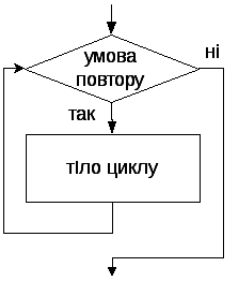
Розв’язання систем лінійних алгебраїчних рівнянь з десятками та сотнями невідомих, пошук коренів алгебраїчних рівнянь високих степенів та коренів трансцендентних рівнянь, розв’язання систем диференційних рівнянь, інтерполяція та екстраполяція функцій, обчислення значень функції за допомогою рядів, інтегрування тощо — усі ці задачі розв’язуються за допомогою циклічних алгоритмів, що реалізують циклічні ітераційні процеси, для яких заздалегідь неможливо визначити кількість повторень циклу. Тому необхідно сформулювати умову виходу з циклу з використанням особливостей самої задачі.

4.3.2 Ітераційні цикли

Розрізняють два типи ітераційних циклів (циклів з невідомою кількістю повторювань) – з *передумовою* і з *постумовою*.

У циклі *з передумовою* така умова перевіряється перед кожною ітерацією циклу (рис.4.8 а), у циклі *з постумовою* – після ітерації циклу (рис. 4.8б).

а)



б)

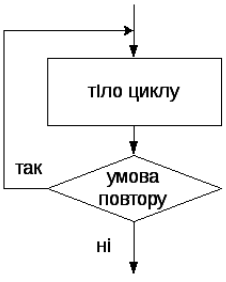


Рис.4.8 Блок-схеми ітераційних циклів

Зверніть увагу!

Незалежно від типу циклу, перед кожним циклом, як правило, має бути блок підготовки до циклу. В цьому блоці задаються початкові значення змінних, що використовуються в заголовку циклу.

З алгоритмічних структур зазначених типів можна будувати складні циклічні процеси із ***вкладеними циклами***. У цьому випадку виділяються *внутрішній* і *зовнішній* цикли. Для кожної зміни значення параметра у зовнішньому циклі відбувається багаторазове виконання дій у внутрішньому циклі, який називається *вкладеним*. Кількість вкладених циклів не обмежується.

Треба зауважити, що внутрішній і зовнішній цикли можуть бути різних типів, кожен, у свою чергу, бути складним. Але вони повинні цілком вкладатися один в один. При цьому важливо пам’ятати про правильну підготовку до початку кожного з циклів.

Комбінуючи базові алгоритмічні структури між собою, можна будувати алгоритми будь-якої складності. Цих структур достатньо для створення найскладнішого алгоритму.

4.4 Рекурентні обчислення

Рекурентні обчислення отримали широке розповсюдження в програмуванні, особливо при організації циклів. Будемо розуміти під рекурентним обчисленням ситуацію, коли наступне значення змінної розраховується з використанням її попереднього значення.

Наприклад, розрахунок суми значень функції, що наведено на рисунку 6. Для розрахунку нового значення S використовується її попереднє значення.

Для програмування рекурентних обчислень спочатку необхідно розробити їх математичну модель у вигляді рекурентної формули.

У загальному випадку Рекурентним співвідношенням називається формула виду:

$$a_{n+1}=F(a_n, a_{n-1},...,a_{n-k+1}),$$

де F - деяка функція від k аргументів, яка дозволяє обчислити наступні члени числової послідовності через значення попередніх членів.

Приклади рекурентних формул:

1. Рекурентне співвідношення арифметичної прогресії:
   $a_{n+1}= a_n + d.$
2. Рекурентне співвідношення геометричної прогресії:
   $a_{n+1}= a_n \cdot q.$

3. Послідовність Фібоначчі:

$a_{n+1}= a_n + a_{n-1}$, $a_1=1$, $a_2=1$.

У найбільшій кількості випадків рекурентна формула має вигляд:

$$a_{n+1} = a_n + r(n) \text{ або } a_{n+1} = a_n * r(n)$$

де n – номер значення змінної, що розраховується, $r(n)$- вираз, що показує зв'язок між теперішнім та майбутнім значеннями змінної.

Для початку обчислень має бути задане початкове значення a_0.

Основна задача при розробці рекурентної формули – це пошук виразу $r(n)$. Якщо відомий математичний вираз для розрахунку елемента послідовності $a_n = g(n)$, то для пошуку $r(n)$ можна використати (в залежності від типу формули) наступний підхід:

$$r(n) = a_{n+1} - a_n = g(n+1) - g(n) \text{ або } r(n) = a_{n+1}/a_n = g(n+1)/g(n)$$

Наприклад, розглянемо пошук рекурентної формули для розрахунку n!.

Як відомо, $n!= 1 * 2 * 3 * \dots * n$, 0!=1.

$$r(n) = \frac{a_{n+1}}{a_n} = \frac{(n+1)!}{n!} = \frac{1 * 2 * 3 * \dots * n * (n+1)}{1 * 2 * 3 * \dots * n} = n + 1$$

Таким чином, для розрахунку $n!$ можна використовувати рекурентну формулу:

$$a_{n+1} = a_n * (n+1), \quad a_0 = 1$$

Наголосимо, що рекурентні обчислення використовуються лише там, де це необхідно. Як правило, вони застосовуються в задачах пошуку сум або добутків елементів послідовностей, в розрахунках формул зі ступенями або факторіалами.

У складних випадках в тілі циклу може бути використано кілька рекурентних формул, які необхідно правильно узгодити між собою.

Приклад 4.1. Знайти суму значень 1!,2!,3!,4!,5!,6!,7!,8!,9!,10!

Рішення. Формально цю задачу можна представити, як пошук суми значень елементів послідовності $a_n = n!$ для $n = \overrightarrow{1,10}$.

Цю задачу можна також записати у вигляді:

$$S = \sum_{n=1}^{10} n!$$

Очевидно, що для розрахунку значення суми необхідно використати рекурентну формулу:

$$S = S + n!$$

Однак для розрахунку $n!$ доцільно також використати рекурентну формулу. Введемо змінну *nf* для позначення $n!$. Для її розрахунку будемо використовувати раніше знайдену рекурентну формулу факторіалу:

$$nf = nf * (n + 1).$$

Алгоритм розрахунку має вид, представлений на рис. 4.9.

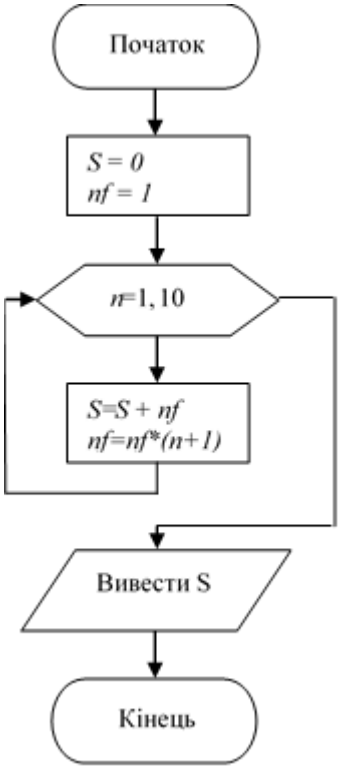


Рис. 4.9 – Алгоритм розрахунку суми факторіалів

Як бачимо, на початку алгоритму в блоці підготовки до циклу змінні отримують свої початкові значення. Оскільки кількість елементів для складення чітко визначена, використано цикл з відомою кількістю повторювань. В тілі циклу спочатку відбувається чергове накопичення суми, а потім розраховується нове значення факторіалу, яке буде використане на наступній ітерації циклу.

Однією з поширених задач, в якій використовуються рекурентні формули, є розрахунок значень функцій за допомогою їх розкладення у ступеневі ряди. Відомо, що будь-яку функцію можна наближено представити у вигляді деякого ряду. Підставивши значення аргументу, можна знайти

часткову суму ряду і вважати її значенням функції від того самого аргументу з певною точністю наближення. Чим більша точність потрібна, тим більший відрізок ряду треба буде обчислювати.

Наприклад, треба *о*бчислити з точністю ε значення функції y(x)=e^x, використовуючи її розкладання у ряд:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots = \sum_{n=1}^{\infty} \frac{x^n}{n!}$$

Обчислення суми ряду з заданою точністю ε означає, що процес обчислення доданків ряду та їх складання продовжується до тих пір, поки доданок за модулем більше ε. Оскільки факторіал зростає швидше ступеня, доданки ряду поступово зменшуються і тому такий момент настане.

Позначимо поточний елемент ряду *an*. Його формула має вид:

$$an(n) = \frac{x^n}{n!}.$$

Оскільки в цій формулі використані ступінь та факторіал, для її розрахунку доцільно застосувати рекурентні обчислення. Для пошуку коригуючого множника використаємо формулу:

$$\frac{an(n+1)}{an(n)} = \frac{x^{n+1} * n!}{(n+1)! * x^n} = \frac{x}{(n+1)}$$

Таким чином, для розрахунку складової ряду отримана рекурентна формула:

$$an = an * x/(n+1)$$

Алгоритм розрахунку наведено на рисунку 4.10.

Як бачимо, на початку алгоритму відбувається введення користувачем значення аргументу та точності, з якою треба знайти значення функції. Далі змінні циклу отримують початкові значення. Потім перевіряється умова досягнення точності. Якщо точність не досягнута (доданок ряду більше заданого значення точності), відбувається розрахунок нового значення суми ряду та його елемента за допомогою рекурентних формул. Через те, що кількість доданків ряду, які будуть відповідати критерію точності, завчасно не відомі, використано цикл з невідомою кількістю повторювань. Цикл з передумовою обрано тому, що додавати до суми ряду слід тільки ті значення, які більше по модулю заданої точності.

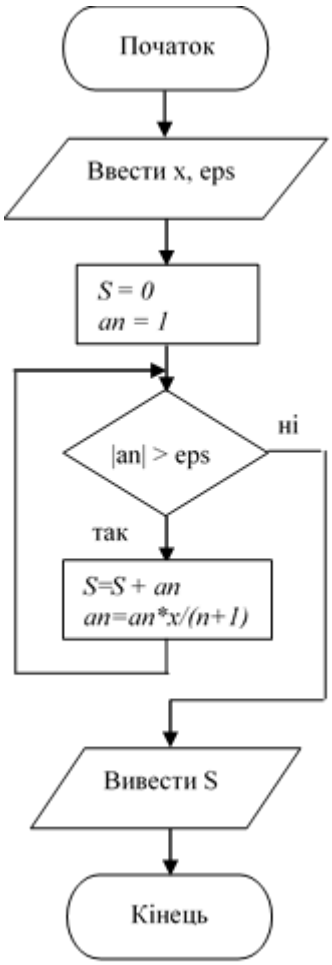


Рис. 4.10 – Алгоритм розрахунку суми ряду
з використанням коригуючого коефіцієнту

Контрольні питання

1. Охарактеризувати основні типи розгалужень в алгоритмах.
2. Наведіть основні типи циклічних процесів, приклади.
3. Наведіть приклади вкладених циклів.
4. Зазначте сфери застосування рекурентних обчислень.

Практичні завдання

Скласти блок-схему алгоритму розв’язання задачі та реалізувати рішення програмно використовуючи середовище розробки Microsoft Visual Studio та мову C#.

20. Обчислити дійсні корені квадратного рівняння.

21. Задано три дійсні числа a, b, c; обчислити периметр та площу трикутника зі сторонами a, b, c, якщо він існує.

22. Знайти мінімум серед чисел a, b, c.

23. Визначити, чи належить точка з координатами (x;y) області $x^2 + y^2 \le 25$

24. Обчислити факторіал числа N.

25. Дана послідовність із N елементів. Розрахувати суму елементів послідовності.

26. Дана послідовність із N елементів. Розрахувати кількість від'ємних елементів послідовності.

27. Дана послідовність із N елементів. Розрахувати мінімум послідовності.

28. Дана послідовність із N елементів. Розрахувати максимум послідовності.

29. Дана послідовність із N елементів. Розрахувати середнє арифметичне послідовності.

30. Дана послідовність із N елементів. Розрахувати середнє арифметичне елементів в діапазоні [a;b].

31. Надрукувати таблицю функції y=sinx+cosx.

32. Дана послідовність, що закінчується нулем. Розрахувати кількість та суму елементів послідовності.

33. Обчислити із заданою точністю ε суму нескінченого ряду $\sum_{n=1}^{\infty} \frac{x^n \cdot \sin(nx)}{n!}$, (при складанні алгоритму та програми використовувати рекурентні формули для обчислення поточного елементу ряду).

34. Надрукувати таблицю чисел виду:

```
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
```

35. Надрукувати таблицю чисел виду:

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
```

36. Серед натуральних чисел, які були введені, знайти найбільше за сумою чисел. Вивести на екран це число і суму його цифр.

37. Порахувати, скільки разів зустрічається певна цифра у введеній послідовності чисел. Кількість чисел, що вводяться, і цифра, яку необхідно порахувати, задаються введенням з клавіатури.

38. Довести гіпотезу Сіракуз на діапазоні чисел. Гіпотеза Сіракуз стверджує, що будь-яке натуральне число зводиться до одиниці в результаті

повторення таких дій над самим числом і результатами цих дій.

- якщо число парне слід розділити його на 2;
- якщо непарне, то помножити його на 3, додати 1 і розділити на 2.

39. Вводяться десять натуральних чисел більше 2. Порахувати, скільки серед них простих чисел.

5 СТРУКТУРНЕ ТА ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

5.1. Структурне програмування

5.1.1 Основні принципи структурного програмування

Становлення і розвиток структурного програмування пов'язане з ім'ям Едсгер Дейкстри. Основні принципи структурного програмування.

Принцип 1. Слід відмовитися від використання оператора безумовного переходу goto.

Принцип 2. Будь-яка програма будується з трьох базових керуючих конструкцій: послідовність, розгалуження, цикл.

Послідовність - одноразове виконання операцій в тому порядку, в якому вони записані в тексті програми.

Галуження - одноразове виконання однієї з двох або більше операцій, в залежності від виконання заданої умови.

Цикл - багаторазове виконання однієї і тієї ж операції до тих пір, поки виконується задана умова (умова продовження циклу).

Принцип 3. У програмі базові керуючі конструкції можуть бути вкладені одна в одну довільним чином. Ніяких інших засобів управління послідовністю виконання операцій не передбачається.

Принцип 4. Фрагменти програми, що повторюються, можна оформити у вигляді підпрограм (процедур і функцій). Таким же чином (у вигляді підпрограм) можна оформити логічно цілісні фрагменти програми, навіть якщо вони не повторюються. У цьому випадку в тексті основної програми, замість поміщеного в підпрограму фрагмента, вставляється інструкція «Виклик підпрограми». При виконанні такої інструкції працює викликана підпрограма. Після цього триває виконання основної програми, починаючи з інструкції, наступної за командою «Виклик підпрограми».

Принцип 5. Кожну логічно завершену групу інструкцій слід оформити як блок (block). Блоки є основою структурного програмування.

Блок - це логічно згрупована частина вихідного коду, наприклад, набір інструкцій, записаних підряд в вихідному коді програми. Поняття блок означає, що до блоку інструкцій слід звертатися як до єдиної інструкції. Блоки служать для обмеження області видимості змінних і функцій. Блоки можуть бути порожніми або вкладеними один в іншій. Межі блоку строго визначені. Наприклад, в if-інструкції блок обмежений кодом BEGIN..END (в мові Паскаль) або фігурними дужками {...} (в мові С) або відступами (в мові Пітон).

Принцип 6. Всі перераховані конструкції повинні мати один вхід і один вихід. Довільні керуючі конструкції (такі, як в страві спагетті) можуть мати довільну кількість входів і виходів. Обмеживши себе керуючими

конструкціями з одним входом і одним виходом, ми отримуємо можливість побудови довільних алгоритмів будь-якої складності за допомогою простих і надійних механізмів.

Принцип 7. Розробка програми ведеться покроково, методом «зверху вниз» (top-down method).

Спочатку пишеться текст основної програми, в якому, замість кожного зв'язкового логічного фрагмента тексту, вставляється виклик підпрограми, яка буде виконувати цей фрагмент. Замість справжніх, які працюють підпрограм, в програму вставляються фіктивні частини - заглушки, які, кажучи спрощено, нічого не роблять.

Якщо говорити точніше, заглушка задовольняє вимогам інтерфейсу замінного фрагмента (модуля), але не виконує його функцій або виконує їх частково. Потім заглушки замінюються або доопрацьовуються до справжніх повнофункціональних фрагментів (модулів) відповідно до плану програмування. На кожній стадії процесу реалізації вже створена програма повинна правильно працювати по відношенню до більш низького рівня. Отримана програма перевіряється та налагоджували.

Після того, як програміст переконається, що підпрограми викликаються в правильній послідовності (тобто загальна структура програми вірна), підпрограми-заглушки послідовно замінюються на реально працюють, причому розробка кожної підпрограми ведеться тим же методом, що і основний програми. Розробка закінчується тоді, коли не залишиться жодної заглушки.

Така послідовність гарантує, що на кожному етапі розробки програміст одночасно має справу з доступним для огляду і зрозумілим йому безліччю фрагментів, і може бути впевнений, що загальна структура всіх вищих рівнів програми вірна.

При супроводі та внесення змін до програми з'ясовується, в які саме процедури потрібно внести зміни. Вони вносяться, не зачіпаючи частини програми, які безпосередньо не пов'язані з ними. Це дозволяє гарантувати, що при внесенні змін і виправлення помилок не вийде з ладу якась частина програми, яка перебуває в даний момент поза зоною уваги.

Слід також врахувати, що деякі автори зазначають, що принципи структурного програмування в рівній мірі можуть застосовуватися при розробці програм як «зверху вниз», так і «знизу вгору».

5.1.2 Підпрограми

Підпрограма є важливим елементом структурного програмування. Спочатку підпрограми з'явилися як засіб оптимізації програм за обсягом займаної пам'яті - вони дозволили не повторювати в програмі ідентичні блоки коду, а описувати їх одноразово і викликати в міру необхідності. До теперішнього часу дана функція підпрограм стала допоміжної, головне їх призначення - структуризація програми з метою зручності її розуміння і супроводу.

Виділення набору дій в підпрограму і виклик її в міру необхідності дозволяє логічно виділити цілісну підзадачу, що має типове рішення. Така дія має ще одну (крім економії пам'яті) перевага перед повторенням однотипних дій. Будь-яка зміна (виправлення помилки, оптимізація, розширення функціональності), зроблене в підпрограмі, автоматично відбивається на всіх її виклики, в той час як при дублюванні кожну зміну необхідно вносити в кожне входження змінюваного коду. Навіть в тих випадках, коли в підпрограму виділяється одноразово вироблений набір дій, це виправдано, оскільки дозволяє скоротити розміри цілісних блоків коду, що складають програму, тобто зробити програму більш зрозумілою і доступній для огляду.

Підпрограми, як правило, поділяються на процедури та функції. Процедури виконують певні дії, об’єднані в одну структуру їх логікою, а функція обов’язково повертає значення через своє ім’я. В деяких мовах програмування (наприклад, C) всі підпрограми мають вид функцій, однак на випадок відсутності необхідності повернення значень використовують спеціальний тип результату (наприклад, void).

Опис підпрограми має такий узагальнений вигляд:

Тип <Им’я алгоритму> [(<список параметрів з типами >)]
Початок блоку
    Оператори підпрограми
Кінець блоку

Список параметрів у загальному випадку складається зі списку вхідних і вихідних даних. Заголовок підпрограми називають прототипом:

Тип <Им’я алгоритму> [(<список типів параметрів >)]

В деяких мовах програмування прототипи підпрограм розміщуються в спеціальному розділі програми для організації роботи компілятора.

Виклик підпрограми здійснюється наступним чином:

<Им’я алгоритму> [(<список значень параметрів>)]

Передача інформації в підпрограму здійснюється наступними способами:

– передача копії інформації до параметру підпрограми. В цьому випадку зміна значення параметру ніяк не впливає на значення змінних у блоці, в якому підпрограма викликана;

– передача адреси змінної або іншого об’єкта параметру підпрограми. В цьому випадку окрема пам'ять для параметру не виділяється, що корисно при обробці великих обсягів даних. Але будь-які зміни значення параметру призводять до зміни об’єкту в блоці виклику, оскільки вони використовують спільну пам'ять. Ця властивість дозволяє також передавати таким чином дані з підпрограми до блоку виклику;

– глобальні змінні (що описані у блоці, зовнішньому до блоку виклику та блоку опису підпрограми). Однак такий канал обміну інформацією дуже важко контролювати, тому використання глобальних змінних у сучасних підходах до програмування не вітається.

У випадку конкретної мови програмування способи передачі параметрів можуть уточнюватися.

Передача інформації з підпрограми до блоку виклику здійснюється наступними способами:

– через ім’я підпрограми

A = <Им’я алгоритму> [(<список типів параметрів >)] ;

– через зміну значення параметру, якому було передано адресу об’єкту.

– через глобальні змінні (вище було детально описано).

5.1.3 Рекурсивні підпрограми

Визначення підпрограм не можуть бути вкладеними, тобто не можна всередині тіла однієї функції визначити тіло іншої. Однак, можна викликати одну функцію з іншої. У тому числі функція може викликати сама себе (але не в усіх мовах програмування).

Розглянемо функцію обчислення факторіала цілого числа на мові програмування С. Її можна реалізувати двома способами. Перший спосіб використовує ітерацію:

```
int fact(int n)
{
```

```
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
    return result;
}
```

Другий спосіб (рекурсія):

```
int fact(int n)
{
  if (n = = 1)    // факторіал 1 дорівнює 1
        return 1;
   else         // факторіал числа n дорівнює факторіалу n-1  помноженому на n
        return n * fact(n -1);
}
```

5.1.4 Переваги структурного програмування

Дотримання принципів структурного програмування зробило тексти програм, навіть досить великих, такими, що нормально читаються. Серйозно полегшилось розуміння програм, з'явилася можливість розробки програм в нормальному промисловому режимі, коли програму може без особливих труднощів зрозуміти не тільки її автор, а й інші програмісти. Це дозволило розробляти досить великі для того часу програмні комплекси силами колективів розробників, і супроводжувати ці комплекси протягом багатьох років, навіть в умовах неминучих змін в складі персоналу.

1. Структурне програмування дозволяє значно скоротити число варіантів побудови програми по одній і тій же специфікації, що значно знижує складність програми і, що ще важливіше, полегшує розуміння її іншими розробниками.

2. У структурованих програмах логічно пов'язані оператори перебувають візуально ближче, а слабо пов'язані - далі, що дозволяє обходитися без блок-схем та інших графічних форм зображення алгоритмів (по суті, сама програма є власною блок-схемою).

3. Сильно спрощується процес тестування і налагодження структурованих програм.

Поліпшення читабельності структурних програм пояснюється тим, що відсутність оператора goto дозволяє читати програму від верху до низу без розривів, викликаних передачами управління. У підсумку можна відразу

(одним поглядом) виявити умови, необхідні для модифікації того чи іншого фрагмента програми.

5.2 Об’єктно-орієнтоване програмування

5.2.1 Причини виникнення об’єктно-орієнтованої парадигми програмування

C, Pascal, FORTRAN та інші схожі з ними мови програмування відносяться до категорії процедурних мов. Кожен оператор такої мови є вказівкою комп’ютеру виконати певну дію. Програміст створює перелік інструкцій, а комп’ютер виконує дії, що відповідають цим інструкціям.

Як би ефективно не використовувався структурний підхід, він не дозволяє в достатній мірі спростити великі складні проекти. Коли розмір програми великий, список команд стає доволі громіздким. Небагато програмістів здатні утримувати в голові більш ніж 500 рядків програмного коду.

Крім того, у процедурній програмі існує ще й проблема неконтрольованого доступу до даних. В чому суть цієї проблеми? В процедурній програмі, що написана, наприклад, мовою C, існує два типи даних. Локальні дані знаходяться всередині функції й призначені для використання виключно цією функцією. Якщо існує необхідність спільного використання одних і тих самих даних кількома функціями, то дані повинні бути оголошені як глобальні. А це, як правило, стосується тих даних програми, які є найбільш важливими. Великі програми зазвичай містять велику кількість функцій та глобальних змінних. Проблема процедурного підходу полягає в тому, що число можливих зв’язків між глобальними змінними та функціями може бути дуже великим (рис. 5.1).

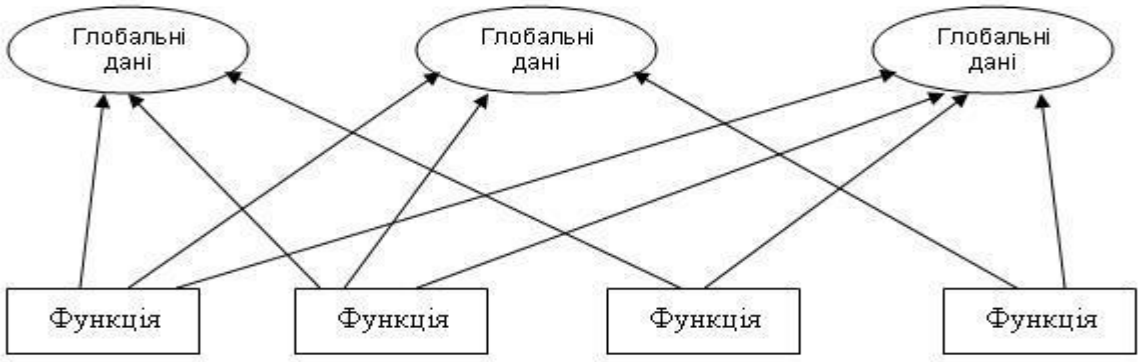


Рис. 5.1 – Приклади зв’язків між глобальними даними та функціями

Велика кількість зв’язків між функціями та даними, в свою чергу, також породжує кілька проблем. По-перше, ускладнюється структура програми. По-друге, в програму стає важко вносити зміни. Зміна структури глобальних даних може потребувати переписування всіх функцій, працюючих з цими даними. Коли зміни вносяться в глобальні дані великих програм, буває складно швидко визначитися, які саме функції необхідно скорегувати. Навіть у тому випадку, коли це вдається зробити, через велику кількість зв’язків між функціями та даними виправлені функції починають некоректно працювати з іншими глобальними даними.

І, нарешті, третя, більш важлива, проблема процедурного програмування полягає в тому, що відокремлення даних від функцій виявилось малопридатним для відображення картини реального світу.

В реальному світі нам доводиться мати справу з фізичними об’єктами, такими, наприклад, як люди, машини чи комп’ютери. Кожен об’єкт має специфічні властивості та виконує специфічні задачі (включаючи сидіння й байдикування). Та ці об’єкти не можна віднести а ні до даних, а ні до функцій, оскільки реальні речі являють собою сукупність властивостей і дій (поведінки).

Властивості. Прикладами властивостей для людей можуть бути колір очей, вік чи місце роботи; для машин - потужність двигуна, марка машини чи рік виробництва. Таким чином, властивості об’єктів рівносильні даним в програмі, бо вони мають певне значення: наприклад, 20 для віку людини чи 2005 для року випуску автомобіля.

Поведінка. Поведінка - це реакція об’єкта у відповідь на зовнішній вплив. Наприклад, ваш викладач у відповідь на прохання поставити вам залік може дати відповідь “так” чи “ні”. Якщо ви натиснете на гальмо автомобіля, це призведе до його зупинки. Відповідь викладача та зупинка є прикладами поведінки (дії) об’єкта.

Об’єктно-орієнтоване програмування є способом організації програми. Цей підхід має кілька ключових концепцій ***: об’єкти і класи, інкапсуляція, наслідування та поліморфізм***.

Основною ідеєю об’єктно-орієнтованого підходу є об’єднання даних і дій, виконуваних над цими даними, в єдине ціле, яке називається об’єктом. Головним компонентом об’єктно-орієнтованої програми є об’єкт.

5.2.2 Класи

Клас - абстракція множини однотипних об'єктів у вигляді спеціального типу, в якому для об'єктів цієї множини відображаються їх дані (атрибути) і реалізовані операції (методи). Для кожної множини об'єктів може створюватися свій клас. Визначення класу включає в себе опис, з яких складових частин або атрибутів він складається і які операції визначені для класу.

Одним з достоїнств класів є те, що вони дають користувачеві можливість створювати свої власні типи даних. Наприклад, вам необхідно працювати з об’єктами, що мають дві координати, наприклад x та y. Вам хотілося б виконувати звичайні арифметичні дії над такими об’єктами, наприклад: pos3 = pos1 + pos2 де змінні pos1, pos2 та pos3 являють собою набори з двох координат. Якщо ми опишемо клас Position, що включає в себе пару координат, то цим ми фактично створимо новий тип даних. А для роботи з подібними даними достатньо оголосити об’єкти цього класу з іменами pos1, pos2 та pos3.

Об'єкт: *окремий екземпляр класу, фактично змінна нового типу даних – класу.* Сукупність значень атрибутів окремого об'єкту називається станом.

Інкапсуляція - властивість мови програмування, що дозволяє користувачеві не замислюватися про складність реалізації використовуваного програмного компонента (що у нього всередині?), а взаємодіяти з ним за допомогою наданого інтерфейсу (публічних методів), а також об'єднати і захистити життєво важливі для компонента дані. При цьому користувачеві надається тільки специфікація (інтерфейс) об'єкта. Користувач може взаємодіяти з об'єктом тільки через цей інтерфейс. Реалізується за допомогою ключового слова: public. Користувач не може використовувати закриті дані і методи. Реалізується за допомогою ключових слів: private, protected, internal.

Рівні доступу до членів класу встановлюються при оголошенні класу.

Таким самим чином в класах закритий доступ до внутрішніх деталей, щоб користувач міг не турбуватися про те, що відбувається „під капотом”. Внутрішня робота класу закрита, тоді як інтерфейс користувача є відкритим.

Захищений рівень доступу пояснити дещо складніше. До захищених членів класу, як і до закритих, користувач звертатися не може. Проте ці члени можуть бути доступні для класів, які є похідними даного класу.

У класу є спеціальні методи.

Конструктор (constructor) – це функція, яка автоматично викликається при створенні об’єкта класу для його ініціалізації. Конструктор використовується для ініціалізації змінних-членів об’єкту, виділення необхідної пам’яті та виконання інших дій, необхідних перед початком

використання об’єкту. Класи можуть не містити явно визначеного конструктора. В таких випадках компілятор створює *конструктор за замовчуванням (default constructor)*. Для простих класів це цілком допустимо, але зазвичай конструктори вставляють у класи будь-якої значущості.

Деструктор – це спеціальна функція, яка автоматично викликається перед знищенням об’єкта. Деструктор можна розглядати як протилежність конструктору. Зазвичай він використовується для вивільнення пам’яті, виділеної класу, чи виконує інші задачі по наведенню порядку після роботи класу. Клас не може мати більше одного деструктора. Наявність деструктора не є обов’язковою. Деструктор не повертає ніякого значення й не приймає аргументів. Деструктор викликається безпосередньо перед руйнуванням класу. Клас може бути зруйнований при виході з області видимості (у випадку розміщення у стеку), чи в результаті застосування оператора delete (у випадку динамічного розміщення). В будь-якому випадку, виклик деструктора буде останньою дією перед остаточним зникненням класу.

5.2.3 Наслідування

Наслідування дозволяє описати новий клас на основі вже існуючого (батьківського), при цьому властивості і функціональність батьківського класу запозичуються новим класом. Іншими словами, клас-спадкоємець реалізує специфікацію вже існуючого класу (базовий клас). Це дозволяє працювати з об'єктами класу-спадкоємця точно так само, як з об'єктами базового класу. Клас, від якого відбулося спадкування, називається базовим або батьківським (англ. base class). Класи, які походять від базового, називаються нащадками, спадкоємцями або похідними класами (англ. derived class). У деяких мовах використовуються абстрактні класи.

Абстрактний клас – це клас, який містить хоча б один абстрактний метод, він описаний у програмі, має поля, методи і не може використовуватися для безпосереднього створення об'єкта. Тобто від абстрактного класу можна тільки наслідувати. Об'єкти створюються тільки на основі похідних класів, успадкованих від абстрактного. Наприклад, абстрактним класом може бути базовий клас «співробітник вузу», від якого успадковуються класи «аспірант», «професор» і т. д. Так як похідні класи мають спільні поля і функції (наприклад, поле «рік народження»), то ці члени класу можуть бути описані в базовому класі. У програмі створюються об'єкти на основі класів «аспірант», «професор», але немає сенсу створювати об'єкт на основі класу «співробітник вузу».

Крім тих властивостей, які є загальними для класу та підкласу, підклас може володіти й власними властивостями.

В програмуванні клас також може породити безліч підкласів. Кожен похідний клас може стати базовим (рис. 5.2)).

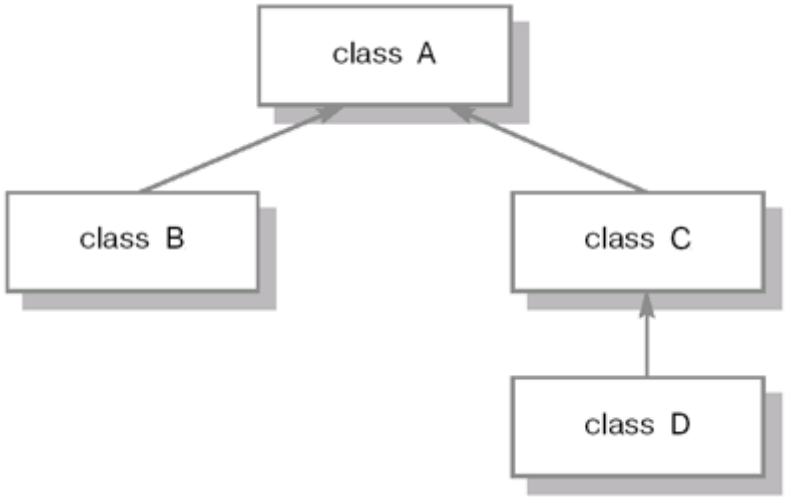


Рис. 5.2 – Приклад наслідування класів

Роль наслідування в ООП – скоротити розмір коду та спростити зв'язки між елементами програми.

В ООП концепція наслідування відкриває нові можливості програмування. А саме можливості повторного використання коду. Мова йде про те, що розроблений клас може бути використаний в інших програмах. Ця властивість називається можливістю повторного використання коду. Аналогічною властивістю в процедурному програмуванні володіють бібліотеки функцій, які можна включати в різноманітні програмні проекти.

Програміст може взяти існуючий клас, та, нічого не змінюючи, додати до нього свої елементи. Всі похідні класи унаслідують ці зміни, й в той же час кожен з похідних класів можна модифікувати окремо. Наприклад, ви (чи хтось інший) розробили клас, що представляє систему меню, аналогічну графічному інтерфейсу Windows. Ви не хочете змінювати цей клас, але вам необхідно додати ще один пункт меню. В цьому випадку ви створюєте новий клас, що наслідує всі властивості вихідного класу, й додаєте в нього необхідний код.

5.2.4 Поліморфізм та перевантаження

Зверніть увагу на те, що операції присвоювання (=) та додавання (+) для типу даних Position повинні виконувати дії, що відрізняються від тих, які ці операції виконують для об'єктів стандартних типів. Чи, наприклад, вам необхідно використати операцію додавання (+) для додавання інтервалів чи

об’єднання рядків. Як же оператори = та + розпізнають, які саме дії необхідно здійснити над операндами? Відповідь полягає в тому, що ми самі можемо задавати ці дії, зробивши потрібні оператори методами відповідного класу (наприклад, класу Position).

Використання операцій та функцій з однаковою назвою різним чином в залежності від того, з якими типами даних вони працюють, називається ***поліморфізмом***.

Коли існуюча операція, наприклад = чи +, наділяється можливістю здійснювати дії над операндами нового користувацького типу, говорять, що така операція є ***перевантаженою***. Перевантаженими можуть бути будь-які унарні чи бінарні операції. Перевантаження являє собою частковий випадок поліморфізму і є важливим інструментом ООП.

У деяких мовах також реалізовано поліморфізм періоду виконання, який базується на використанні похідних класів і віртуальних функцій.

Віртуальна функція – це така функція, яка оголошується в базовому класі з використанням ключового слова virtual і перевизначається в одному або декількох похідних класах. Таким чином, кожен похідний клас може мати власну версію віртуальної функції.

Для успішного застосування поліморфізму необхідно розуміти, що базовий і похідний класи утворюють ієрархію, розвиток якої спрямований від більшого до меншого ступеня узагальнення (тобто від базового класу до похідного). У разі коректної реалізації базовий клас забезпечує всі елементи, які похідний клас може використовувати безпосередньо. Він також визначає функції, які похідний клас повинен реалізувати самостійно. Це дає похідному класу гнучкість у визначенні власних методів, але водночас зобов'язує використовувати загальний інтерфейс. Іншими словами, оскільки формат інтерфейсу визначається базовим класом, то будь-який похідний клас повинен розділяти цей загальний інтерфейс. Таким чином, застосування віртуальних функцій дає змогу базовому класу визначати узагальнений інтерфейс, який використовуватиметься всіма похідними класами.

Віртуальні методи дозволяють програмувати дії, загальні для всіх похідних класів, в термінах базового класу. Динамічно, під час виконання програми, буде викликатися метод потрібного класу. Властивість віртуальності проявляється тільки тоді, коли звернення до методу йде через покажчик або посилання на об'єкт. Покажчик або посилання можуть вказувати як на об'єкт базового класу, так і на об'єкт похідного класу. Якщо ж в програмі є сам об'єкт, то вже під час компіляції відомо, якого він типу і, відповідно, віртуальність не використовується.

Контрольні питання

1. Наведіть основні принципи структурного програмування.
2. Які основні способи обміну інформацією з підпрограмами?
3. Що таке рекурсія?
4. Наведіть основні визначення об’єктно-орієнтованого програмування.
5. Які переваги має об’єктно-орієнтоване програмування?

Практичні завдання

1. Опишіть структуру, яка представляє точку в двовимірному просторі.
2. Опишіть множину точок в двовимірному просторі, як масив структур із пункту 1.
3. Опишіть функцію, яка дозволяє задати з клавіатури довільну множину точок у двовимірному просторі.
4. Опишіть функцію, яка дозволяє обчислити довжину ломаної лінії, яку можна утворити, поєднавши послідовно точки деякої множини.
5. Реалізуйте задачі 1-4 із використанням класів.

6 ОСНОВНІ СТРУКТУРИ ДАНИХ

6.1 Рівні організації даних

Програміст, проектувальник і користувач мають свої власні погляди на організацію даних. Відповідно до цього можуть бути виділено три рівні організації даних:

- логічна організація даних: проектний рівень.
- представлення даних: рівень мови реалізації.
- фізична організація даних: машинний рівень.

Логічна організація даних відображає погляд користувача на дані. У її основі лежать вимоги користувача й внутрішньо властиві даним зв'язки. Це найбільш важливий рівень абстракції, використовуваний представленні даних, оскільки саме вимоги користувачів визначають вигляд проектованої системи. Якщо на етапі проектування системи вдало обрана логічна організація даних, зміни системних вимог, що не приводять до модифікації логічної структури даних, не спричинять реорганізації на більш низьких рівнях представлення даних. Тільки на логічному рівні можуть застосовуватися формальні методи опису динамічно-змінюваних структур.

Логічна організація структур даних - це моделі структур, які не залежать від способу їх зберігання у комп‘ютерній пам'яті. Логічну модель даних називають абстрактною моделлю.

Опис даних мовою програмування належить до рівня представлення даних. Відношення між даними задаються у вигляді, характерному для конкретної мови. На цьому рівні оперують масивами й вказівниками.

Інформація про представлення даних може бути розподілена по окремих програмних модулях, причому можна використовувати як зовнішню, так і внутрішню форми представлення даних. Під зовнішнім представленням розуміється погляд на дані з боку інших програм, тобто представлення на рівні потоків даних. При зовнішньому представленні головним є визначення можливих шляхів доступу.

Внутрішнє представлення - це представлення у вигляді внутрішніх областей зберігання даних, тобто структура даних може бути в зовнішньому представленні стеком, а у внутрішньому - масивом або зв'язаним списком. Припустимо, що дані про покупця включають номер рахунку, ім'я, адресу, дати попередніх витрат, платежі й рахунку. Зовнішньою формою їх представлення є окремі агрегати даних про особу покупця й про його платежі. Внутрішнє представлення може складатися з таблиць, полів ознак і вказівників.

Фізична організація даних вказує на те, у якому вигляді дані зображуються в пам'яті комп‘ютера. Фізична організація даних суттєво залежить від типу пам'яті, на якій вони записуються. Фізичну модель даних називають конкретною моделлю.

Рівень фізичної організації пов'язаний із системним програмним забезпеченням. На цьому рівні доводиться оперувати із межами слів, розмірами полів, двійковими кодами й фізичними записами. Більшість засобів системного програмного забезпечення дають можливість програмісту здійснити вибір з досить вузького кола способів представлення даних, які мають більш-менш ясну фізичну організацію. Агрегати даних, що зберігаються у швидкодіючій пам'яті, можуть бути представлені масивами або стеками. До записів файлу можна звертатися в послідовному або довільному порядку, використовуючи ключі різних типів.

Входом і одним виходом, ми отримуємо можливість побудови довільних алгоритмів будь-якої складності за допомогою простих і надійних механізмів.

6.2 Класифікація структур даних

Класифікація структур даних виконується за декількома ознаками.

1) **За способом представлення**: фізична та логічна.

Поняття "фізична структура даних" має відношення до способу фізичного представлення даних у пам'яті машини і називається ще структурою збереження, внутрішньою структурою або структурою пам'яті.

Логічна чи абстрактна структура – це розгляд структури даних без врахування її представлення в машинній пам'яті. У загальному випадку між логічною і відповідною їй фізичною структурами існує розходження, ступінь якого залежить від самої структури й особливостей того середовища, у якому вона повинна бути відображеною. Внаслідок цього розходження існують процедури, що здійснюють відображення логічної структури у фізичну, і, навпаки, фізичної структури в логічну. Ці процедури забезпечують, крім того, доступ до фізичних структур і виконання над ними різних операцій, причому кожна операція розглядається стосовно до логічної чи фізичної структури даних.

2) **За складністю**: прості й інтегровані.

Прості (базові, примітивні) структури – це такі, які не можуть бути розподілені на складові частини. З погляду фізичної структури важливою є та обставина, що в даній машинній архітектурі, у даній системі програмування відомо, який буде розмір даного простого типу і яка структура його розміщення в пам'яті. З логічної точки зору прості дані є неподільними одиницями.

Інтегровані (структуровані, композитні, складні) – такі структури даних, складовими частинами яких є інші структури даних – прості чи, у свою чергу, інтегровані. Інтегровані структури даних конструюються програмістом з використанням засобів інтеграції даних, наданих мовами програмування.

3) **За наявністю зв'язків**між елементами даних: незв'язні та зв'язні.

Незв'язні структури характеризуються відсутністю зв'язків між елементами структури, зв'язні – наявністю такого зв'язку. Прикладами незв'язних структур є вектори, масиви, рядки, стеки, черги; приклади зв'язних структур – зв'язні списки.

4) **За мінливістю**: статичні, напівстатичні, динамічні.

Дуже важлива ознака структури даних - її мінливість, тобто зміна числа елементів і (чи) зв'язків між елементами структури. У визначенні мінливості структури не відбитий факт зміни значень елементів даних, оскільки в цьому випадку всі структури даних мали б властивість мінливості.

Статичні – до цієї групи відносять масиви, множини, записи, таблиці.

Напівстатичні – це стеки, черги, деки, дерева.

Динамічні – лінійні та розгалужені зв'язні списки, графи, дерева.

5) **За характером упорядкованості елементів** у структурі: лінійні та нелінійні.

Лінійні структури в залежності від характеру взаємного розташування елементів у пам'яті поділяють на структури з послідовним розподілом елементів у пам'яті (вектори, рядки, масиви, стеки, черги) і структури з довільним зв'язним розподілом елементів у пам'яті (однозв'язні і двозв¢язні лінійні списки).

Нелінійні структури – багатозв¢язні списки, дерева, графи.

6) **За видом пам'яті**, використовуваної для збереження даних: структури даних для оперативної і для зовнішньої пам'яті.

Структури даних для оперативної пам'яті – це дані, розміщені в статичній і динамічній пам'яті комп'ютера. Всі вищенаведені структури даних – це структури для оперативної пам'яті.

Структури даних для зовнішньої пам'яті називають файловими структурами чи файлами. Прикладами файлових структур є послідовні файли, файли, організовані розділами, B-дерева.

У мовах програмування поняття "структури даних" тісно пов'язано з поняттям "типи даних". Будь-які дані, тобто константи, змінні, значення функції чи виразу, характеризуються своїми типами. Інформація для кожного типу однозначно визначає:

а) структуру збереження даних зазначеного типу, тобто розподіл пам'яті та представлення даних у ній, з одного боку, та інтерпретування двійкового представлення, з іншого;

б) припустимі значення, що може мати об'єкт описуваного типу;

в) припустимі операції, що можуть бути застосовні до об'єкта описуваного типу.

6.3 Базові операції зі структурами даних

Над усіма структурами даних можуть виконуватися чотири базові операції фізичного рівня: створення, видалення, вибір (доступ), відновлення.

Операція **створення** полягає у виділенні пам'яті для структури даних. Пам'ять може виділятися в процесі виконання програми з першою появою імені змінної у вихідній програмі або на етапі компіляції, чи при активізації процедурного блока, у якому з'являються відповідні змінні. Програміст може і сам виділяти пам'ять для структур даних, використовуючи наявні в системі програмування процедури і функції для виділення. У ряді мов (наприклад, у С) для структурованих даних, що сконструйовані програмістом, операція створення містить у собі також установку початкових значень параметрів створюваної структури.

Операція **видалення** структур даних протилежна за своєю дією операції створення. Деякі мови, такі як BASIC, FORTRAN, не надають можливості програмісту видаляти створені структури даних. У мовах С, PASCAL структури даних, наявні усередині блока, знищуються в процесі виконання програми при виході з цього блока. Програміст може і сам звільняти пам'ять для структур даних, використовуючи наявні в системі програмування процедури і функції для цього. Операція видалення допомагає ефективно використовувати пам'ять.

Операція **вибору** використовується для доступу до даних усередині самої структури. Форма операції доступу залежить від типу структури даних, до якої здійснюється звертання. Метод доступу – одна з найбільш важливих властивостей структур даних, особливо в зв'язку з тим, що ця властивість має безпосереднє відношення до вибору конкретної структури даних.

Операція **відновлення** дозволяє змінити значення даних у структурі даних. Прикладом операції відновлення є операція присвоювання або більш складна форма – передача параметрів.

Вищевказані чотири операції обов'язкові для всіх структур і типів даних. Крім цих загальних операцій для кожної структури даних можуть бути

визначені специфічні операції, що працюють з даними тільки зазначеного типу (даної структури).

6.4 Масиви

6.4.1 Поняття масиву. Способи опису

Масив – це впорядкований іменований набір із фіксованої кількості однотипних даних. Доступ до будь-якого елементу масиву здійснюється за його номером. У масиви можна об’єднувати результати експериментів, списки прізвищ співробітників, різні складні структури даних. Наприклад, список учнів у класному журналі є масивом. У масиві дані розрізняються за своїми порядковими номерами (індексами). Якщо кожний елемент масиву визначається за допомогою одного номера, то такий масив називається одновимірним, якщо за двома – то двовимірним. Двовимірний масив – це таблиця з рядків і стовпців. У таблицях перший номер вказує на рядок, а другий – на положення елемента в рядку. Усі рядки таблиці мають однакову довжину.

Одновимірний масив може бути набором чисел, сукупністю символьних даних чи елементів іншої природи (навіть масив масивів).

Фізична структура масиву – це спосіб розміщення елементів масиву в пам’яті комп’ютера. Під елемент масиву виділяється кількість байт пам’яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам’яті для зберігання масиву. Елементи масиву розташовуються у пам’яті комп’ютера підряд, один за одним.

Сама найважливіша операція фізичного рівня над масивом – доступ до заданого елемента. Як тільки реалізовано доступ до елемента, над ним може бути виконана будь-яка операція, що має сенс для того типу даних, якому відповідає елемент. Перетворення логічної структури масиву у фізичну називається процесом лінеаризації, в ході якого багатовимірна логічна структура масиву перетвориться в одновимірну фізичну структуру.

Адресою масиву є адреса першого байту початкового компоненту масиву. Індексація масивів мовах сімейства С обов’язково починається з нуля.

До операцій логічного рівня над масивами необхідно віднести такі як сортування масиву, пошук елемента за ключем.

В мовах сімейства С оголошення масиву має наступний синтаксис:

<специфікація типу> <ім'я> [<константний вираз>];
<специфікація типу> <ім'я> [];

Тут квадратні дужки є елементом синтаксису, а не ознакою необов'язковості конструкції. Квадратні дужки можуть також розміщуватися перед ідентифікатором (синтаксис C#).

Оголошення масиву можуть мати одну з двох синтаксичних форм, зазначених вище. Квадратні дужки, які слідують за *ім'ям* чи перед ним, - ознака того, що змінна є масивом. Константний вираз, укладений в квадратні дужки визначає число елементів у масиві. Оскільки індексація елементів масиву в мовах сімейства C починається з нуля, останній елемент масиву має індекс на одиницю менше, ніж число елементів масиву.

У другій синтаксичній формі константний вираз у квадратних дужках опущено. Ця форма може бути використана, якщо в оголошенні масиву присутній ініціалізатор, або масив оголошується як формальний параметр функції, або дане оголошення є посиланням на оголошення масиву десь в іншому місці програми. Однак для багатовимірного масиву може бути опущена тільки перша розмірність.

Масив може складатися з елементів будь-якого типу, крім типу *void* і функцій, тобто елементи масиву можуть мати базовий, перерахований, структурний тип, бути об'єднанням, покажчиком або масивом.

Приклади оголошень масивів:

int x [10]; // Одновимірний масив з 10 цілих чисел. Індекси змінюються від 0 до 9.

double y [2][10]; // Двовимірний масив дійсних чисел з 2 рядків і 10 стовпців.

Як і прості змінні, масиви можуть бути ініціалізовані при оголошенні. Ініціалізатор для об'єктів складових типів (яким є масив) складається зі списку ініціалізаторов, розділених комами і укладених у фігурні дужки. Кожен ініціалізатор в списку являє собою або константу відповідного типу, або, у свою чергу, список ініціалізаторов. Ця конструкція використовується для ініціалізації багатовимірних масивів.

Наявність списку ініціалізаторов в оголошенні масиву дозволяє не вказувати число елементів по його першій розмірності. У цьому випадку кількість елементів у списку ініціалізаторов і визначає число елементів по першій розмірності масиву. Тим самим визначається розмір пам'яті, яка необхідна для зберігання масиву. Число елементів по решті розмірностям масиву, окрім першої, вказувати обов'язково.

Якщо в списку ініціалізаторов менше елементів, ніж у масиві, то залишилися елементи неявно ініціалізуються нульовими значеннями. Якщо ж

число ініціалізаторов більше, ніж потрібно, то видається повідомлення про помилку.

Приклади ініціалізації масивів

int a[3] = {0, 1, 2}; // Число ініціалізаторов дорівнює числу елементів

double b[5] = {0.1, 0.2, 0.3}; // Число ініціалізаторов менше числа елементів

int c [] = {1, 2, 4, 8, 16}; // Число елементів масиву визначається за кількістю ініціалізаторов

int d [2][3] = {{0, 1, 2}, {3, 4, 5}}; // Ініціалізація двовимірного масиву. Масив складається з двох рядків, в кожному з яких по 3 елементи. Елементи першого рядка отримують значення 0, 1 і 2, а другий - значення 3, 4 і 5.

int e[3] = {0, 1, 2, 3}; // Помилка - число ініціалізаторов більше числа елементів

Зверніть увагу, що не існує присвоювання масиву, відповідного описаному вище способу ініціалізації.

int a[3] = {0, 1, 2}; // Оголошення і ініціалізація

a = {0, 1, 2}; // Помилка

У мові C# Оголошення одномірного масиву виглядає в такий спосіб:

<тип>[] <ім'я масиву>;

Квадратні дужки приписані не до імені змінної, а до типу. Вони є невід'ємною частиною визначення класу, так що, наприклад, запис int [ ] варто розуміти як клас одномірний масив з елементами типу int.

Що ж стосується границь зміни індексів, то ця характеристика до класу не ставиться, вона є характеристикою змінних.

Як й у випадку оголошення простих змінних, при оголошенні масиву одночасно може бути проведена й ініціалізація. Потрібно розуміти, що при оголошенні з відкладеною ініціалізацією сам масив не формується, а створюється тільки посилання на масив, що має невизначене значення Null. Тому поки масив не буде реально створений і його елементи ініціалізовані, використати його в обчисленнях не можна. Приклад оголошення трьох масивів з відкладеною ініціалізацією:

int[] a, b, c;

Найчастіше при оголошенні масиву використовується ім'я з ініціалізацією. І знов-таки, як й у випадку простих змінних, можуть бути два варіанти ініціалізації. У першому випадку ініціалізація є явною й задається константним масивом:

double[] x= {5.5, 6.6, 7.7};

Дотримуючись синтаксису, елементи константного масиву варто брати у фігурні дужки.

У другому випадку створення й ініціалізація масиву виконується в об'єктному стилі з викликом конструктора масиву. І це найпоширеніша практика оголошення масивів:

int[] d= new int[5];

Якщо масив оголошується без ініціалізації, то створюється тільки посилання на нього. Якщо ініціалізація виконується конструктором, то в динамічній пам'яті створюється сам масив, елементи якого ініціалізуються константами відповідного типу (нуль для арифметики, порожній рядок для строкових масивів), і посилання зв'язується із цим масивом. Якщо масив ініціалізується константним масивом, то в пам'яті створюється константний масив, з яким і зв'язується посилання.

Розглянемо ще кілька прикладів оголошення масивів:

```
...
int[] k;                //k - масив
k=new int [3];          //Визначаємо масив з 3-х цілих
k[0]=-5; k[1]=4; k[2]=55; //Задаємо елементи масиву
                        //Виводимо третій елемент масиву
Console.WriteLine(k[2].ToString());
...
```

Зміст наведеного фрагменту ясний з коментарів. Зверніть увагу на деякі особливості. По-перше, масив визначається саме як

int[] k;

а не як один з наступних варіантів:

int k[]; //Невірно!
int k[3]; //Невірно!
int[3] k; //Невірно!

По-друге, тому що масив представляє посилальний об'єкт, то для створення масиву необхідний рядок

k=new int [3];

Саме в ній ми й визначаємо розмір масиву. Хоча, загалом кажучи, можливі конструкції виду

int[] k = new int [3];

Елементи масиву можна задавати відразу при оголошенні. От приклад:

int[] k = {-5, 4, 55};

Зрозуміло, наведені конструкції застосовані не тільки до типу int і не тільки до масиву розміру 3.

В C# нумерація елементів масиву починається з нуля. Таким чином, у прикладі початковий елемент масиву - це k[0], а останній - k[2]. Елемента k[3] немає.

Для ***доступу до конкретного елементу масиву*** використовуються так звані індексні вирази:

<ім'я масиву> [<цілочисельний вираз>]

Тут квадратні дужки є вимогою синтаксису мови, а не ознакою необов'язковості конструкції.

Індекс масиву може бути не тільки константою, а й виразом, який має цілочисельний тип, наприклад, *a[i + 1]* (тут *a* повинно бути ім'ям раніше оголошеного масиву, а *i* - змінної цілого типу).

Для обробки елементів масиву зазвичай використовується оператор покрокового циклу for:

for (i = 0; // Привласнюємо лічильнику циклу значення індексу першого елемента

i<n; // Умова продовження циклу - поки значення лічильника менше кількості елементів масиву

i++) // Збільшуємо лічильник циклу на 1 для переходу до наступного елементу масиву

<тіло циклу> // У тілі циклу відбувається обробка одного елемента масиву

Для обробки багатовимірного масиву використовується відповідна кількість циклів.

У мові C++ немає можливості вводити і виводити весь масив одним оператором вводу/виводу. Можна вводити і виводити тільки один елемент масиву. Отже, для того щоб ввести весь масив, треба використовувати цикл.

6.4.2 Цикл foreach

Новим видом циклу у C#, що часто використовується й достатньо зручний при роботі з масивами, є цикл foreach. Його синтаксис:

foreach(тип ідентифікатор in контейнер) оператор

Тіло циклу виконується для кожного елементу масиву й закінчується, коли повністю перебрані всі елементи. Тип ідентифікатора повинен бути погоджений з типом елементів, що зберігаються в масиві даних. Передбачається також, що елементи масиву впорядковані. На кожному кроці циклу ідентифікатор, що задає поточний елемент масиву, одержує значення чергового елемента відповідно до порядку, установленим на елементах масиву. Із цим поточним елементом і виконується тіло циклу - виконується стільки разів, скільки елементів перебуває в масиві.

Недоліком циклів foreach у мові C# є те, що цикл працює тільки на читання, але не на запис елементів. Так що наповнювати масив елементами доводиться за допомогою інших операторів циклу.

Приклади використання циклу foreach:

```
class Program
   {
      static void Main(string[] args)
      {
        int[] array1 = {0, 2, 4, 6, 8, 10};

foreach (int n in array1)
{
 System.Console.WriteLine(n.ToString());
}

string[] array2 = {"hello", "world"};

foreach (string s in array2)
{
 System.Console.WriteLine(s);
}
      }
   }
```

У наведеному прикладі наш цикл перебирає всі елементи масиву array1. На це вказує рядок

foreach (int n in array1)

яка інтерпретується так: для кожного цілого числа з масиву array1 робимо щось там. Якби елементами масиву були б не цілі, а, скажемо, речовинні числа, то запис виглядав би так:

foreach(float n in array1)

Отже ми пишемо саме тип елементів масиву. Цикл foreach використається не тільки для масивів, але й для інших об'єктів.

6.4.3 Алгоритми пошуку в одновимірних масивах

Алгоритми пошуку застосовуються для знаходження, наприклад, у масиві елемента з потрібними властивостями. Звичайно розрізняють постановки завдання пошуку для першого й останнього входження елемента. В усіх нижче викладених алгоритмах будемо вважати, що виробляється пошук у масиві A з N цілих чисел елемента, рівного X.

Лінійний пошук.

Лінійний пошук здійснюється циклом з подвійною умовою. Перша умова контролює індекс на приналежність масиву, наприклад, (i<=N). Друга умова - це умова пошуку. Фактично перевіряються всі елементи масиву з початку, поки не буде знайдено підходящий елемент.

Після виходу із циклу необхідно перевірити, по якій з умов ми вийшли. В операторі **if** звичайно повторюють перша умова циклу. Можна говорити про успішний пошук із циклом while при виконанні цієї умови.

Пошук бар’єром.

Ідея пошуку з бар'єром полягає в тому, щоб не перевіряти щораз у циклі умову, пов'язану із границями масиву. Це можна забезпечити, установивши в масив так званий бар'єр: елемент, що задовольняє умові пошуку. Тим самим буде обмежена зміна індексу.

Вихід із циклу, у якому тепер залишається тільки умова пошуку, може відбутися або на знайденому елементі, або на бар'єрі. Таким чином, після виходу із циклу перевіряється, чи не бар'єр ми знайшли? Обчислювальна складність пошуку з бар'єром менше, ніж у лінійного пошуку, але також є величиною того ж порядку, що й N - кількість елементів масиву.

Існує два способи установки бар'єра: додатковим елементом або замість крайнього елемента масиву.

Двійковий (БІНАРНИЙ) пошук.

Алгоритм двійкового пошуку можна використати для пошуку елемента із заданою властивістю тільки в масивах, упорядкованих по цій властивості. Так при пошуку числа із заданим значенням необхідно мати масив, упорядкований по зростанню або по убуванню значень елементів. А, наприклад, при пошуку числа із заданою сумою цифр масив повинен бути впорядкований по зростанню або по убуванню сум цифр елементів.

Ідея алгоритму полягає в тому, що масив щораз ділиться навпіл і вибирається та частина, де може перебувати потрібний елемент. Розподіл триває поки частина масиву для пошуку більше одного елемента, після чого залишається перевірити цей елемент, що залишився, на виконання умови пошуку.

Існують дві модифікації цього алгоритму: для пошуку першого й останнього входження. Все залежить від того, як вибирається середній елемент: округленням у меншу або більшу сторону. У першому випадку середній елемент ставиться до лівої частини масиву, а в другому - до правого.

У процесі роботи алгоритму двійкового пошуку розмір фрагмента, де цей пошук повинен тривати, щораз зменшується приблизно у два рази. Це забезпечує обчислювальну складність алгоритму порядку логарифма N по основі 2, де N - кількість елементів масиву.

6.4.4 Алгоритми сортування в одновимірних масивах

Найпростіше завдання сортування полягає в упорядкуванні елементів масиву по зростанню або убуванню. Іншим завданням є впорядкування елементів масиву відповідно до деякого критерію. Звичайно як такий критерій виступають значення певної функції, аргументами якої виступають елементи масиву. Цю функцію прийнято називати функцією, що впорядковує.

Існують різні методи сортування. Будемо розглядати кожний з методів на прикладі завдання сортування по зростанню масиву з N цілих чисел.

Сортування вибором.

Ідея методу полягає в тім, що перебуває максимальний елемент масиву й міняється місцями з останнім елементом (з номером N). Потім, максимум шукається серед елементів з першого до передостаннього й ставиться на N-1 місце, і так далі. Необхідно знайти N-1 максимум. Можна шукати не максимум, а мінімум і ставити його на перше, друге й так далі місце. Також застосовують модифікацію цього методу з одночасним пошуком максимуму й мінімуму. У цьому випадку кількість кроків зовнішнього циклу N div 2.

Обчислювальна складність сортування вибором - величина порядку N*N, що звичайно записують як O(N*N). Це порозуміється тим, що кількість порівнянь при пошуку першого максимуму дорівнює N-1. Потім N-2, N-3, і так далі до 1, разом: N*(N-1)/2.

Сортування обміном (методом "пухирця").

Ідея методу полягає в тім, що послідовно рівняються пари сусідніх елементів масиву. Якщо вони розташовуються не в тім порядку, то робимо перестановку, міняючи місцями пари сусідніх елементів. Після одного такого проходу на останнім місці номер N виявиться максимальний елемент ("сплив" перший "пухирець"). Наступний прохід повинен розглядати елементи до передостаннього й так далі. Усього потрібно N-1 прохід. Обчислювальна складність сортування обміном O(N*N).

Можна помітити, що якщо при виконанні чергового проходу в сортуванні обміном не зроблений ні однієї перестановки, те це означає, що масив уже впорядкований. Таким чином, можна модифікувати алгоритм, щоб наступний прохід робився тільки при наявності перестановок у попередньому.

Сортування вставками.

Сортування вставками – третій і останній з простих алгоритмів впорядкування одновимірних масивів. Основна ідея даного методу полягає в тому, що на першому кроці порівнюються другий та перший елемент вихідного масиву. Якщо порядок між ними, в залежності від типу сортування (за зростанням чи за спаданням) порушений, то перший елемент пересувається на одну позицію вправо. Тепер відсортований масив складається з двох елементів.

Продовжуючи ітераційний процес далі, беремо наступний (третій, четвертий і так далі) елемент і по черзі порівнюємо його, починаючи з кінця, з іншими елементами в уже відсортованому масиві. Якщо порядок між порівнюваними елементами порушений, то міняємо їх місцями, якщо ні, то вставка нового елемента закінчена і переходимо до наступної ітерації (рис.6.1).

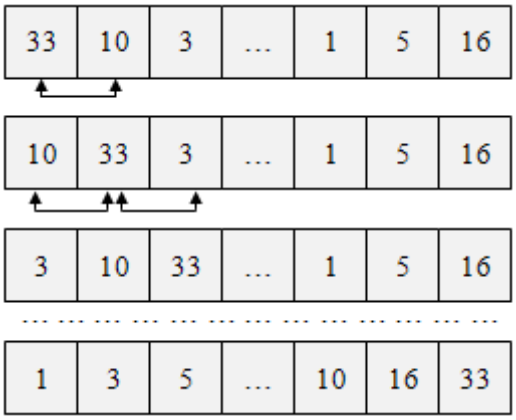


Рис.6.1 – Сортування вставками

Однак відмітимо, що розглянутий алгоритм, як і алгоритм сортування вибором та сортування обміном, володіє певним недоліком. Для знаходження позиції на яку необхідно вставити кожен наступний елемент, да,ний алгоритм **сортування** порівнює його з кожним більшим від нього значенням елементом у відсортованій частини. Для великих масивів такий ланцюжок порівнянь може бути достатньо громіздким, що може призвести до значного збільшення часу виконання програми. Для того, щоб прискорити даний процес, зазвичай використовують так званий методом половинного ділення.

Тобто, на кожній ітерації елемент, який необхідно вставити порівнюється з елементом, що міститься в середині відсортованої частини. Якщо його значення більше від даного елемента, то наступним аналізується елемент із середнім індексом в меншій частині, якщо менше — то в більшій і так далі, поки не залишиться один елемент. Після цього, всі елементи значення яких являються більшими від значення елемента, який необхідно вставити, зсуваються в циклі, і на це місце робиться вставка.

Обчислювальна складність сортування вставками - величина порядку N*N.

6.4.2 Багатовимірні масиви

Поділ масивів на одномірні і багатомірні носить історичний характер. Ніякої принципової різниці між ними немає. Одномірні масиви - це окремий випадок багатомірних. Можна говорити й по-іншому: багатомірні масиви є природним узагальненням одномірних. Одномірні масиви дозволяють задавати такі математичні структури як вектори, двовимірні - матриці, тривимірні - куби даних, масиви більшої розмірності - багатомірні куби даних.

При роботі з базами даних багатомірні куби, так названі куби OLAP, зустрічаються дуже часто.

У чому особливість оголошення багатомірного масиву? Як у типі вказати розмірність масиву? Це робиться досить просто, за рахунок використання ком. От як виглядає оголошення багатомірного масиву в загальному випадку:

<тип>[, ... ,] <об'явники>;

Приклади.

Двовимірний масив:

int[,] k = new int [2,3];

Зверніть увагу, що пари квадратних дужок тільки одна. У нашому прикладі в масиву 6 (=2*3) елементів (k[0,0] - перший, k[1,2] - останній).

Аналогічно ми можемо задавати багатомірні масиви. Приклад тривимірного масиву:

int[,,] k = new int [10,10,10];

Варіант ініціалізації багатомірного масиву:

int[,] k = {{2,-2},{3,-22},{0,4}};

Число ком, збільшене на одиницю, і задає розмірність масиву. Що стосується об'явників, те все, що сказано для одномірних масивів, справедливо й для багатомірних. Можна лише відзначити, що хоча явна ініціалізація з використанням багатомірних константних масивів можлива, але застосовується рідко через громіздкість такої структури. Простіше ініціалізацію реалізувати програмно, але іноді вона все-таки застосовується. От приклад:

```
public void TestMultiArr()
{
  int[,]matrix = {{1,2},{3,4}};
  Arrs.PrintAr2("matrix", matrix);
}//TestMultiArr
```

Не зважаючи , що поділ на одновимірні та багатовимірні масиви часто умовний з точки зору структури даних, алгоритми роботи з двомірними масивами мають свою особливість. Ця особливість в тому, що, частіше, для перебору елементів масиву використовується два цикли – зовнішній і вкладений. При цьому кожна змінна циклу відповідає за номери стовпця і рядка.

Для доступу до елементу масиву вказується його ім'я й у квадратних дужках - індекси потрібного елемента. З елементом масиву можна працювати як зі звичайної змінної, тобто можна прочитати його значення або записати в нього нове значення.

Приклади.

a[1,1] := 0; //елементу масиву a з індексом 1.1 привласнюється значення 0;

*a[1,0] := a[1,0]*2; //елемент масиву a з індексом 1, 0 подвоюється.*

В якості індексів елементів масиву можуть виступати цілі константи та цілі змінні. Використання цілих змінних дає можливість працювати з елементами масиву використовуючи цикли.

Приклад двовимірного масиву:

A0,0	A0,1	A0,2	A0,3
A1,0	A1,1	A1,2	A1,3
A2,0	A2,1	A2,2	A2,3
A3,0	A3,1	A3,2	A3,3

Перший індекс елемента масиву визначає номер рядка, другий номер – стовпця. Для квадратних масивів розрізняють поняття головної діагоналі. Це елементи у яких індекси рядка і стовпця співпадають:

A0,0	A1,1	A2,2	A3,3

та побічної діагоналі

A0,3	A1,2	A2,1	A3,0

6.4.3 Динамічні масиви

У всіх вищенаведених прикладах оголошувалися статичні масиви, оскільки нижня границя дорівнює нулю по визначенню, а верхня завжди задавалася в цих прикладах константою. Динамічний розподіл пам'яті використовується, перш за все, тоді, коли заздалегідь невідомо, скільки об'єктів знадобиться в програмі і чи знадобляться вони взагалі. За допомогою динамічного розподілу пам'яті можна гнучко управляти часом життя об'єктів, наприклад виділити пам'ять не на самому початку програми (як для

глобальних змінних), та звільняти тоді, коли потреба в зберіганні інформації минає.

Динамічні масиви організовуються за допомогою вказівників.

За допомогою спеціальних функцій виділяється блок пам’яті необхідного розміру, адреса початку якого присвоюється вказівнику.

Для доступу до елемента масиву використовується зміщення.

Наприклад для одномірного масиву a[i] == *(a+i).

В цьому випадку індекс елементу являє собою зміщення відносно адреси початку масиву *a*.

Для двомірного масиву A[N][M] його елемент A[i][j] буде мати зміщення S:

$$S = j + i*M.$$

Тобто, A[i][j] == *(A+ j+i*M).

У мові C++ дозволяється при роботі з динамічними масивами використовувати запис елементів як для статичних масивів, тобто компілятор бере на себе роботу по формуванню зміщення.

В C++ для створення нового масиву використовується оператор new, а для видалення delete.

Наприклад,

```
a= new int[100]
delete [] a;
```

Однак при роботі з динамічною пам’яттю потрібно бути обережним.

Наведемо кілька прикладів.

Використання неправильної адреси в операції delete. Результат такої операції непередбачуваний. Цілком можливо, що сама операція пройде успішно, проте внутрішня структура пам'яті буде зіпсована, що призведе або до помилки в наступній операції new, або до псування який-небудь інформації.

Пропущене звільнення пам'яті, тобто програма багаторазово виділяє пам'ять під дані, але "забуває" її звільняти. Такі помилки називають витоками пам'яті. По-перше, програма використовує непотрібну їй пам'ять, тим самим знижуючи продуктивність. Крім того, цілком можливо, що в 99 випадках з 100 програма буде успішно виконана. Однак якщо втрата пам'яті виявиться занадто великою, програмі не вистачить пам'яті під якісь дані і, відповідно, відбудеться збій.

Запис за невірною адресою. Швидше за все, будуть зіпсовані будь-які дані. Як проявиться така помилка - невірним результатом, збоєм програми або іншим чином - передбачити важко.

Приклади помилок можна наводити нескінченно. Загальні їх риси, що обумовлюють складність виявлення, це, по-перше, непередбачуваність результату і, по-друге, прояв не в момент здійснення помилки, а пізніше, можливо, в тому місці програми, яка сама по собі не містить помилки (неправильна операція delete - збій в наступної операції new, запис по невірному адресою - використання зіпсованих даних в іншій частині програми і т.п.).

Зверніть увагу!

В C# всі масиви**, незалежно від того, яким вираженням описується границя, **розглядаються як динамічні об'єкти.

Приклад, у якому описана робота з динамічним масивом:

```
public void TestDynAr()
{
  //оголошення динамічного масиву A1
  Console.WriteLine("Введіть число елементів масиву A1");
  int size = int.Parse(Console.ReadLine());
  int[] A1 = new int[size];
}
```

У даній процедурі верхня границя масиву визначається користувачем.

У C # є так званий ***"збирач сміття",*** який робить всю чорнову роботу по звільненню пам'яті самостійно.

6.4.4 Масиви масивів

Ще одним видом масивів C# є масиви масивів, названі також зубчастими масивами (jagged arrays). Такий масив масивів можна розглядати як одномірний масив, елементи якого є масивами, елементи яких, у свою чергу, знову можуть бути масивами, і так може тривати до деякого рівня вкладеності.

У яких ситуаціях може виникати необхідність у таких структурах даних? Ці масиви можуть застосовуватися для подання дерев, у яких вузли можуть мати довільне число нащадків. Таким може бути, наприклад, генеалогічне дерево.

Є деякі особливості в оголошенні й ініціалізації таких масивів. Якщо при оголошенні типу багатомірних масивів для вказівки розмірності використовувалися коми, то для зубчастих масивів застосовується більш ясна символіка - сукупності пари квадратних дужок; наприклад, int [ ][ ] задає масив, елементи якого - одномірні масиви елементів типу int.

Складніше зі створенням самих масивів й їхньою ініціалізацією. Тут не можна викликати конструктор new int[3][5], оскільки він не задає зубчастий масив. Фактично потрібно викликати конструктор для кожного масиву на самому нижньому рівні. У цьому й складається складність оголошення таких масивів.

Приклад.

```
//масив масивів
//оголошення й ініціалізація
int[][] jagger = new int[3][]
{
  new int[] {5,7,9,11},
  new int[] {2,8},
  new int[] {6,12,4}
};
```

Масив jagger має всього два рівні. Можна вважати, що в нього три елементи, кожний з яких є масивом. Для кожного такого масиву необхідно викликати конструктор new, щоб створити внутрішній масив. У даному прикладі елементи внутрішніх масивів одержують значення, будучи явно ініціалізовані константними масивами. Звичайно, припустиме й таке оголошення:

```
int[][] jagger1 = new int[3][]
{
  new int[4],
  new int[2],
  new int[3]
};
```

У цьому випадку елементи масиву одержать при ініціалізації нульові значення. Реальну ініціалізацію потрібно буде виконувати програмним шляхом. Варто помітити, що в конструкторі верхнього рівня константу 3 можна опустити й писати просто new int[ ][ ]. Виклик цього конструктора можна взагалі опустити - він буде матися на увазі:

```
int[][] jagger2 =
{
  new int[4],
  new int[2],
  new int[3]
};
```

А от конструктори нижнього рівня необхідні. Ще одне важливе зауваження - динамічні масиви можливі й тут. У загальному випадку, границі

на будь-якому рівні можуть бути вираженнями, що залежать від змінних. Більше того, припустимо, щоб масиви на нижньому рівні були багатомірними.

Ще кілька прикладів:

```
//Об'являємо 2-мірний східчастий масив
int[][] k = new int [2][];
// Об'являємо 0-й елемент нашого східчастого масиву
//Це знову масив й у ньому 3 елементи
k[0]=new int[3];
// Об'являємо 1-й елемент нашого східчастого масиву
//Це знову масив й у ньому 4 елементи
k[1]=new int[4];
k[1][3]=22; //записуємо 22 в останній елемент масиву
...
```

Зверніть увагу, що в зубчастих масивів задається пара квадратних дужок (стільки, яка розмірність масиву).

6.4.5 Розріджені масиви

На практиці зустрічаються масиви, які через певні причини можуть займати пам’ять не повністю, а частково. Це особливо актуально для масивів великих розмірів, таких що для їхнього зберігання в повному об’ємі пам’яті може бути недостатньо. Розріджений масив – це масив, більшість елементів якого рівні між собою, так що зберігати в пам’яті достатньо лише невелику кількість значень відмінних від основного (фонового) значення інших елементів. При роботи з розрідженими масивами питання розташування їх в пам’яті реалізуються на логічному рівні з врахуванням їхнього типу.

Розрізняють два типи розріджених масивів:

- масиви, в яких розташування елементів із значеннями відмінними від фонового, можуть бути описані математично;
- масиви з випадковим розташуванням елементів.

До **масивів з математичним описом розташування елементів** відносяться масиви, в яких існує закономірності в розташуванні елементів із значеннями відмінними від фонового.

Елементи, значення яких є фоновими, називають нульовими; елементи, значення яких відмінні від фонового, – ненульовими. Фонове значення не завжди рівне нулю. Ненульові значення зберігаються, як правило, в одновимірному масиві, а зв’язок між розташуванням у розрідженому масиві і в новому, одновимірному, описується математично за допомогою формули, що перетворює індекси масиву в індекси вектора.

На практиці для роботи з розрідженим масивом розробляються функції:

- для перетворення індексів масиву в індекс вектора;
- для отримання значення елемента масиву з його упакованого представлення за індексами;
- для запису значення елемента масиву в її упаковане представлення за індексами.

До **масивів з випадковим розташуванням елементів** відносяться масиви, в яких не існує закономірностей у розташуванні елементів із значеннями відмінними від фонового.

Один з основних способів зберігання подібних розріджених матриць полягає в запам'ятовуванні ненульових елементів в одновимірному масиві і ідентифікації кожного елемента масиву індексами.

Дане представлення масиву скорочує вимоги до об'єму пам'яті більш ніж в 2 рази. Спосіб послідовного розподілу має також ту перевагу, що операції над матрицями можуть бути виконані швидше, ніж це можливо при представленні у вигляді послідовного масиву, особливо якщо розмір матриці великий.

Методи послідовного розміщення для представлення розріджених матриць звичайно дозволяють швидше виконувати операції над матрицями і більш ефективно використати пам'ять, ніж методи із зв'язаними структурами. Проте послідовне представлення матриць має певні недоліки. Так включення і виключення нових елементів матриці викликає необхідність переміщення великої кількості інших елементів. Якщо включення нових елементів і їхнє виключення здійснюється часто, то повинен бути вибраний метод зв'язаних структур.

6.5 Структури

На відміну від масивів чи множин, усі елементи яких однотипні, структура може містити елементи різних типів.

Елементи структури називаються полями структури і можуть мати довільний тип, крім типу цієї ж структури, але можуть бути покажчиками на неї. Якщо при описі структури відсутній тип структури, обов'язково повинен бути вказаний список змінних, покажчиків або масивів визначеної структури.

Звернення до окремих полів структури замінюються на їхні адреси ще на етапі компіляції.

Самою найважливішою операцією для структури є операція доступу до вибраного поля структури – операція кваліфікації.

Над вибраним полем структури можливі будь-які операції, які допустимі для типів цього поля.

Більшість мов програмування підтримує деякі операції, які працюють із структурою, як з єдиним цілим, а не з окремими її полями. Це операція присвоєння значення одного запису іншому однотипному запису, при цьому відбувається по елементне копіювання.

В мові С++ визначення структури складається з двох кроків:

1) оголошення структури (завдання нового типу даних певного користувачем), структура складається з полів;

```
struct student
{
char fio[30];                    // визначено поле fio
char group[8];                   // визначено поле group
int year;
int informatika, math, fizika, history;
}
```

2) визначення змінних типу структура;

```
student Vasya, ES[50];
```

Для звернення до полів структури треба вказати ім'я змінної і через крапку ім'я поля:

Structura.Pole

Наприклад,

Vasya.Year

6.6 Перерахований тип

При написанні програм часто виникає потреба мати змінні, які приймають значення з визначеного списку констант. Для цього зручно використовувати перерахований тип.

Перерахований тип представляє собою впорядкований тип даних, який визначається програмістом, тобто програміст перераховує всі значення, які може приймати змінна цього типу.

В мові С++ для створення перерахування використовується ключове слово *enum*. Загальна форма перерахування має наступний вигляд:

enum ім'я_перерахування {список_імен} список_змінних

Тут ім'я_перерахування - ім'я типу даного перерахування. У списку імен наводяться назви констант. Вони, як і в списку змінних, відокремлюються один від одного комами.

При відсутності ініціалізації перша константа приймає нульове значення, а кожній наступній присвоюється на одиницю більше значення від попереднього.

На фізичному рівні над змінними перерахованого типу визначені операції створення, знищення, вибору, поновлення. При цьому виконується визначення порядкового номера ідентифікатора за його значенням і, навпаки, за номером ідентифікатора його значення.

При виконанні арифметичних операцій перечислення перетворюються у ціле. Оскільки перечислення є типом, який визначається користувачем, для нього можна вводити власні операції.

Наприклад, в наступному фрагменті програми спочатку визначаються перерахування міст, іменоване *cities*, і змінна *c* типу *cities*, а потім змінній *c* присвоюється значення Houston:

```
enum cities {Houston, Austin, Amarillo} c;
c = Houston;
```

У будь-якому перерахуванні значення першого (крайнього зліва) по імені за замовчуванням дорівнює 0, значення другого імені дорівнює 1 і т.д. Взагалі кожному імені присвоюється значення, на одиницю більше від значення попереднього імені. Додавши ініціалізатор, можна надати імені деяке конкретне значення. Наприклад, в наступному перерахуванні ім'я Austin матиме значення 10:

```
enum cities {Houston, Austin = 10, Amarillo}
```

У цьому прикладі ім'я Amarillo матиме значення 11.

Приклад програми:

```
enum Weekdays {SA, SU,      MO, TU, WE, TH, FR};
enum Weekdays {SA, SU = 0, MO, TU, WE, TH, FR};

void main ()
    {Enum Weekdays d1 = SA, d2 = SU, d3 = WE, d4;

     d4 = 2;                              // Помилка!
     d4 = d1 + d2;                        // Помилка!
     d4 = (enum Weekdays) (d1 + d2);      // Можна, але результат
     d4 = (enum Weekdays) (d1 - d2);      // може не потрапити
     d4 = (enum Weekdays) (TH * FR);      // в область визначення
```

```
d4 = (enum Weekdays) (WE / TU);          // перерахування
}
```

6.7 Списочні структури

6.7.1 Характеристика динамічних типів даних

Дуже часто в задачах програмування заздалегідь невідомо, який обсяг даних необхідно буде обробляти. Застосування динамічних масивів дозволяє тільки частково вирішити цю проблему, оскільки при виділенні пам'яті під масив необхідно вже знати його розмірність, а також тому, що пам'ять, виділена для масиву, може бути довго порожньою в очікуванні введення всіх даних. Крім того, під масив пам'ять виділяється єдиним блоком і не завжди в такому вигляді в достатньому обсязі комп'ютер може її знайти. Тому були запропоновані динамічні структури даних, в яких для кожного елемента пам'ять можна виділяти в міру необхідності.

До динамічних структур даних відносяться різні види списків. Зазвичай елемент списку містить смислове частина (власне збережені дані) і посилання на інший або інші елементи списку. Для програмування списків використовують структури або класи. Розрізняють однозв'язні списки, двозв'язні списки і дерева.

У однозв'язного списку кожен елемент зберігає адресу розташування наступного елемента. Останній елемент як адресу наступного елемента зберігає NULL. Виділяється спеціальна адресна константа, що зберігає адресу першого елемента списку (прийнято називати її head). Пересування по списку починається завжди з першого елемента, а потім відбуваються переходи по ланцюжку до кожного наступного елементу. Рухатися в такому списку можна тільки від голови до хвоста. Додавання елементів можливо в будь-яке місце списку. Воно реалізується за принципом вставки ланки в ланцюг. Ланцюг розмикається і нова ланка зв'язується з попереднім і наступним елементом.

Для полегшення пересування по списку ввели двозв'язні списки. В такому списку в елементі зберігається не тільки адреса наступного елемента, але і адреса попереднього елемента (адреси сусідів по ланцюжку). Завдяки цьому рухатися в такому списку можна не тільки вперед, але і назад. На додаток до покажчика на голову списку, вводиться покажчик на його кінець (хвіст або tail).

У дереві кожен елемент (вузол) зберігає посилання на адреси своїх нащадків. Найбільшого поширення набули бінарні дерева, в яких у кожного вузла не більше двох нащадків (лівий і правий), хоча можна створювати дерева довільної структури. Робота з деревами схожа на роботу з однозв'язного

списками (рух починається з початку, від кореня), але ускладнюється наявністю декількох дочірніх вузлів. Переваги зв'язного представлення даних:

- можливість забезпечення значної змінності структур;
- розмір структури обмежується тільки доступним об'ємом машинної пам'яті;
- при зміні логічної послідовності елементів структури потрібно виконати не переміщення даних в пам'яті, а тільки корекцію покажчиків.

Разом з тим зв'язне представлення не позбавлене й недоліків, основні з яких:

- робота з покажчиками вимагає більш високої кваліфікації від програміста;
- на поля зв'язку витрачається додаткова пам'ять;
- доступ до елементів зв'язної структури може бути менш ефективним за часом.

Останній недолік є найбільш серйозним і саме ним обмежується застосування зв'язного представлення даних. Якщо в суміжному представленні даних для обчислення адреси будь-якого елемента у всіх випадках достатньо номера елемента і інформації, яка міститься в описі структури, то для зв'язного представлення адреса елемента не може бути обчислена з початкових даних. Опис зв'язної структури містить один або декілька покажчиків, які дозволяють увійти до структури, далі пошук необхідного елемента виконується проходженням ланцюжком покажчиків від елемента до елемента. Тому зв'язне представлення практично ніколи не застосовується в задачах, де логічна структура даних має вигляд вектора або масиву – з доступом за номером елемента, але часто застосовується в задачах, де логічна структура вимагає іншої початкової інформації доступу (таблиці, списки, дерева і т.д.).

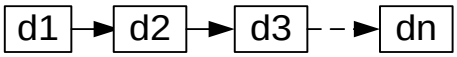
Списки набули значного поширення при роботі з даними. Розглянемо їх більш детально.

6.7.2 Лінійні списки

Списки є досить гнучкою структурою даних, так як їх легко зробити більшими або меншими, і їх елементи доступні для вставки або вилучення в будь-якій позиції списку. Списки також можна об'єднувати або розділяти на менші списки.

Лінійний список – це скінчена послідовність однотипних елементів (вузлів), можливо, з повторенням. Кількість елементів у послідовності

називається довжиною списку. Вона в процесі роботи програми може змінюватися. Лінійний список L, що складається з елементів $d_1, d_2, \ldots, d_n$, які мають однаковий тип, записують у вигляді $L = \langle d_1, d_2, \ldots, d_n \rangle$, або зображають графічно.



Важливою властивістю лінійного списку є те, що його елементи можна лінійно впорядкувати у відповідності з їх позицією в списку.

Для формування абстрактного типу даних на основі математичного визначення списку потрібно задати множину операторів, які виконуються над об'єктами типу список. Проте не існує однієї множини операторів, які виконуються над списками, які задовольняють відразу всі можливі застосування.

Найчастіше зі списками доводиться виконувати такі операції:

1) Знайти елемент із заданою властивістю;
2) Визначити і-й елемент у лінійному списку;
3) Внести додатковий елемент до або після вказаного вузла;
4) Вилучити певний елемент списку;
5) Впорядкувати вузли лінійного списку в певному порядку.

У реальних мовах програмування не існує якої-небудь структури даних для зображення лінійного списку так, щоб усі операції над ним виконувалися в однаковій мірі ефективно. Тому при роботі з лінійними списками важливе значення має подання лінійних списків, які використовуються в програмі, таким чином, щоб була забезпечена максимальна ефективність і за часом виконання програми, і за обсягом потрібної їй пам'яті.

Лінійний список є послідовність об'єктів. Позиція елемента в списку має інший тип даних, відмінний від типу даних елемента списку, і цей тип залежить від конкретної фізичної реалізації.

Над лінійним списком допустимі наступні операції:

Операція вставки – вставляє елемент в конкретну позицію в списку, переміщуючи елементи від цієї позиції і далі в наступну, більш вищу позицію.

Операція локалізації – повертає позицію об'єкта в списку. Якщо в списку об'єкт зустрічається декілька разів, то повертається позиція першого від початку списку об'єкта. Якщо об'єкта немає в списку, то повертається значення, яке рівне довжині списку, збільшене на одиницю.

Операція вибірки елемента з списку – повертає елемент, який знаходиться в конкретній позиції списку. Результат не визначений, якщо в списку немає такої позиції.

Операція вилучення – вилучає елемент в конкретній позиції зі списку. Результат невизначений, якщо в списку немає вказаної позиції.

Операції вибірки попереднього і наступного елемента – повертають відповідно наступній і попередній елемент списку відносно конкретної позиці в списку.

Функція очистки списку робить список пустим.

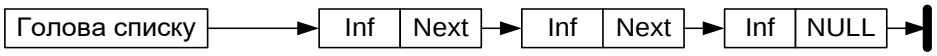
Основні ***методи зберігання лінійних списків*** поділяються на методи послідовного і зв'язного зберігання. При виборі способу зберігання в конкретній програмі слід враховувати, які операції і з якою частотою будуть виконуватися над лінійними списками, вартість їх виконання та обсяг потрібної пам'яті для зберігання списку.

Найпростіша форма представлення лінійного списку — це вектор. Визначивши таким чином список можна почергово звертатися до них в циклі і виконувати необхідні дії. Однак при такому представленні лінійного списку не вдасться уникнути фізичного переміщення елементів, якщо потрібно добавляти нові елементи, або вилучати існуючі. Набагато швидше вилучати елементи можна за допомогою простої схеми чистки пам'яті. Замість вилучення елементів із списку, їх помічають як невикористані.

Більш складною організацією при роботі зі списками є розміщення в масиві декількох списків або розміщення списку без прив'язки його початку до першого елемента масиву.

При зв'язному представленні лінійного списку кожен його елемент складається із значення і покажчика, який вказує на наступний елемент у списку.

На наступному рисунку приведена структура однозв'язного списку. Кожний список повинен мати особливий елемент, який називається покажчиком на початок списку, або головою списку.

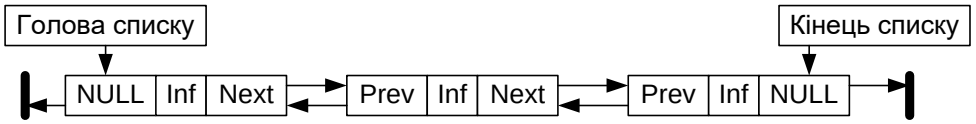


Програмно кожен елемент списку представляється типом структури, який містить тип інформаційного поля (inf) та вказівник на наступний елемент (Next).

6.7.3 Двох-зв'язний список

Обробка однозв'язного списку не завжди зручна, оскільки відсутня можливість просування в протилежну сторону. Таку можливість забезпечує

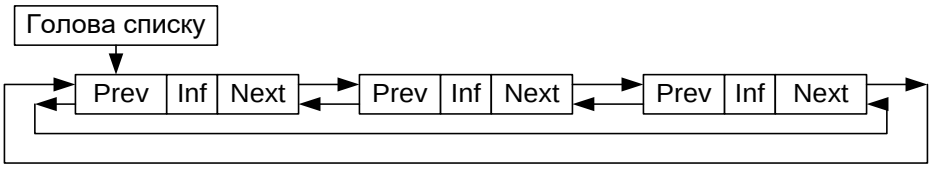
двох-зв'язний список, кожний елемент якого містить два покажчики: на наступний і попередній елементи списку.



Програмно кожен елемент списку представляється типом структури.

Для зручності обробки списку додають ще один особливий елемент – покажчик кінця списку. Наявність двох покажчиків в кожному елементі ускладнює список і приводить до додаткових витрат пам'яті, але в той же час забезпечує більш ефективне виконання деяких операцій над списком.

Різновидом розглянутих видів лінійних списків є кільцевий список, який може бути організований на основі як однозв'язного, так і двох-зв'язного списків. При цьому в однозв'язному списку покажчик останнього елемента повинен вказувати на перший елемент; в двох-зв'язному списку в першому і останньому елементах відповідні покажчики змінюються.



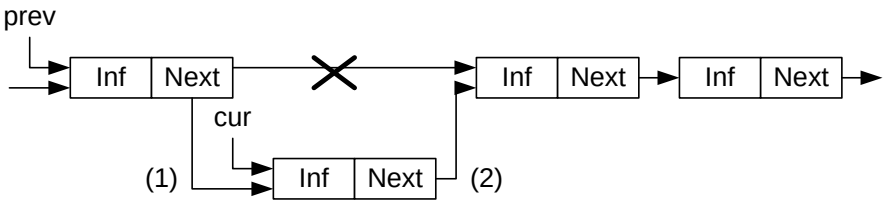
При роботі з такими списками дещо спрощуються деякі процедури, проте, при перегляді такого списку слід приймати деякі запобіжні засоби, щоб не потрапити в нескінченний цикл.

В пам'яті список є сукупністю опису однакових за розміром і форматом структур, які розміщені довільно в деякій ділянці пам'яті і пов'язані одна з одною в лінійно впорядкований ланцюжок за допомогою покажчиків. Структура містить інформаційні поля і поля покажчиків на сусідні елементи списку, причому деякими полями інформаційної частини можуть бути покажчики на блоки пам'яті з додатковою інформацією, що відноситься до елемента списку.

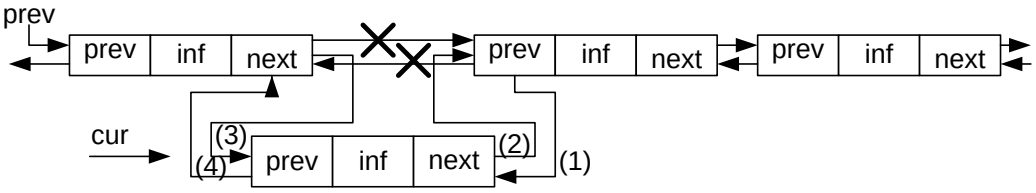
6.7.4 Операції над списками

Розглянемо деякі прості операції над списками.

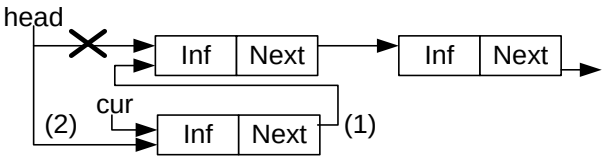
Вставка елемента в середину однозв'язного списку:



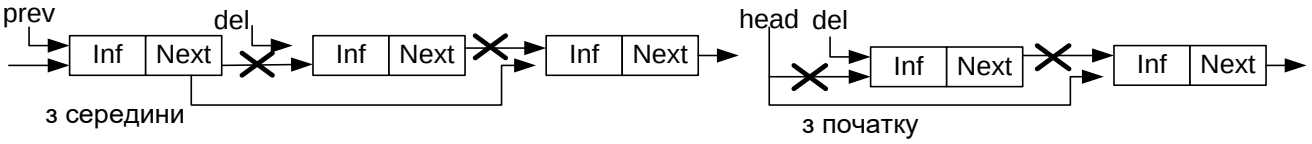
Вставка елемента в двох-зв’язний список:



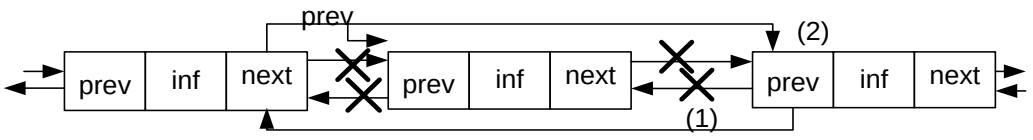
Наведені приклади забезпечують вставку в середину списку, але не можуть бути застосовані для вставки на початок списку. При такій операції повинен модифікуватися покажчик на початок списку:



Видалення елемента з однозв’язного списку для двох варіантів – з середини і з голови:

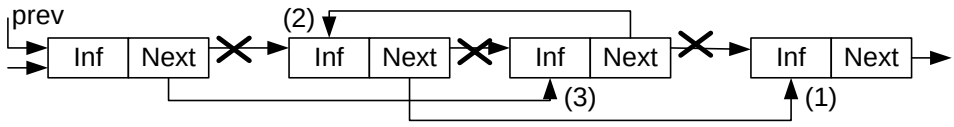


Видалення елемента з двох-зв’язного списку вимагає корекції більшої кількості покажчиків:



Процедура видалення елемента з двох-зв’язного списку виявиться навіть простішою, ніж для однозв’язного, оскільки в ній не потрібно шукати попередній елемент, він вибирається за покажчиком назад.

Змінність динамічних структур даних допускає не тільки зміни розміру структури, але і зміни зв’язків між елементами. Для зв’язних структур зміна зв’язків не вимагає пересилки даних в пам’яті, а тільки зміни покажчиків в елементах зв’язної структури. В якості прикладу приведена перестановка двох сусідніх елементів списку. В алгоритмі перестановки в однозв’язному списку виходили з того, що відома адреса елемента, який передує парі, в якій проводиться перестановка. В приведеному алгоритмі також не враховується випадок перестановки початкових елементів списку.



У процедурі перестановки для двох-зв’язного списку неважко врахувати і перестановку на початку списку.

6.7.5 Мультисписки

В програмних системах, які обробляють об’єкти складної структури, можуть вирішуватися різні підзадачі, кожна з яких вимагає обробки можливо не всієї множини об’єктів, а лише якоїсь його підмножини.

Для того, щоб при вибірці кожної підмножини не виконувати повний перегляд з відсіванням записів, які до необхідної підмножини не відносяться, в кожний запис включаються додаткові поля посилань, кожне з яких зв’язує в лінійний список елементи відповідної підмножини. В результаті виходить багато-зв’язковий список або мультисписок, кожний елемент якого може входити одночасно в декілька однозв’язних списків.

До переваг мультисписків крім економії пам’яті (при множині списків інформаційна частина існує в єдиному екземплярі) слід віднести також цілісність даних – в тому сенсі, що всі підзадачі працюють з однією і тією ж версією інформаційної частини і зміни в даних, зроблені одній підзадачей негайно стають доступними для іншої підзадачі.

Кожна підзадача працює з своєю підмножиною як з лінійним списком, використовуючи для цього певне поле зв’язку. Специфіка мультисписку виявляється тільки в операції виключення елемента із списку. Виключення елемента з якого-небудь одного списку ще не означає необхідності видалення елемента з пам’яті, оскільки елемент може залишатися у складі інших списків. Пам’ять повинна звільнятися тільки у тому випадку, коли елемент вже не входить ні в один з приватних списків мультисписку. Звичайно задача

видалення спрощується тим, що один з приватних списків є головним – в нього обов'язково входять всі наявні елементи. Тоді виключення елемента з будь-якого неголовного списку полягає тільки в зміні покажчиків, але не в звільненні пам'яті. Виключення ж з головного списку вимагає не тільки звільнення пам'яті, але і зміні покажчиків як в головному списку, так і у всіх неголовних списках, в які елемент, що видаляється, входив.

Контрольні питання

1. Наведіть визначення масиву.
2. Як описується масив?
3. Поясніть основні алгоритми пошуку інформації в масиві.
4. Наведіть відомі алгоритми сортування одновимірних масивів.
5. Самостійно проаналізуйте більш ефективні алгоритми сортування – сортування злиттям, швидке сортування та інші.
6. Дайте характеристику динамічним типам даних.
7. Наведіть визначення лінійного списку.
8. Яка структура однозв'язного списку?
9. Яка структура двозв'язного списку?
10. Напишіть алгоритми виконання основних операцій над однозв'язними та двозв'язними списками.

Практичні завдання

Скласти блок-схему алгоритму розв'язання задачі та реалізувати рішення програмно використовуючи середовище розробки Microsoft Visual Studio та мову C#.

Задачі на обробку одновимірних масивів (пошук, сортування, екстремуми, складні операції).

1 Знайти в одновимірному масиві A задане число x, використовуючи метод лінійного пошуку.
2 Знайти в одновимірному впорядкованому за зростанням масиві A число x, використовуючи метод дихотомічного (бінарного) пошуку.
3 Визначити, чи є одновимірний масив впорядкованим за зростанням або зменшенням елементів, чи ніяк не впорядкований.
4 Знайти мінімальний елемент одновимірного масиву.
5 Поміняти місцями мінімум та максимум в одновимірному масиві.
6 Серед студентів першого курсу проводиться набір гравців в університетську баскетбольну команду. Студенти, які грають у

баскетбол, повинні бути не нижче 185 см, але і не вище 205 см. Визначте, який зріст буде у найнижчого гравця в баскетбольній команді.

7 Відсортувати одновимірний масив за зростанням елементів, використовуючи метод бульбашки.
8 Відсортувати одновимірний масив за зростанням елементів, використовуючи метод Шелла.
9 Відсортувати одновимірний масив за зменшенням елементів, використовуючи метод вибору.
10 Відсортувати одновимірний масив за зменшенням елементів, використовуючи метод злиття.
11 Дано одновимірний масив із N елементів. Розширити його, продублювавши кожен елемент масиву.
12 Дало одновимірний масив. Створити новий масив, видаливши з вхідного масиву позитивні елементи.

Задачі на обробку рядків

13 Знайти у рядку S заданий символ ch.
14 Замінити у рядку всі точки на пробіли.
15 Підрахувати кількість знаків пунктуації в рядку. Перелік знаків визначити самостійно, але не меншe6 символів.
16 Визначити кількість слів у рядку, вважаючи словом будь-яку непусту послідовність символів, яка не містить пробілів.
17 Вивести слова рядка в стовпчик. Словом вважати будь-яку непусту послідовність символів, яка не містить пробілів.
18 Визначити, чи є рядок паліндромом (вважати, що рядок є підготовленим і містить тільки значущі символи).
19 Знайти, скільки разів в рядку S1 зустрічається рядок S2.

Задачі на обробку двовимірних масивів

20 Дана прямокутна матриця. Обчислити суму її елементів.
21 Дана квадратна матриця, обчислити суми елементів на головній та побічній діагоналях.
22 Дана прямокутна матриця. Обчислити суми елементів по рядках.
23 Дана прямокутна матриця. Обчислити в кожному стовпчику кількість позитивних елементів.
24 Дана прямокутна матриця. Поміняти місцями перший та останній рядок матриці.
25 Знайти мінімум матриці.
26 Знайти в матриці позицію числа x (пошук виконувати зліва направо згори донизу). Якщо число x відсутнє, повернути індекси (-1; -1).
27 Дано одновимірний масив A невід'ємних чисел. Створити зубчастий масив B, кількість елементів кожного «зубця» визначається значенням в відповідного елементу масиву A. заповнити масив B випадковими

значеннями та обчислити його суму.

Задачі на обробку списків

28 Написати функції роботи з лінійним списком (елемент списку містить 2 інформаційні поля: цілого та рядкового типів):
- створення пустого списку;
- видалення списку;
- додавання елементу в кінець списку;
- додавання елементу в голову списку;
- видалення елементу з кінця списку;
- видалення елементу з початку списку;
- видалення елементів, що мають вказане значення у заданому інформаційному полі;
- пошук елементу із вказаним значенням в заданому інформаційному полі;
- розрахунок суми елементів списку (числове інформаційне поле);
- пошук елементу із рядком максимальної довжини в рядковому інформаційному полі.

29 Написати функції роботи із двозв’язним списком (елемент списку містить 2 інформаційні поля: цілого та рядкового типів):
- створення пустого списку;
- видалення списку;
- додавання елементу в кінець списку;
- додавання елементу в голову списку;
- видалення елементу з кінця списку;
- видалення елементу з початку списку;
- видалення елементів, що мають вказане значення у заданому інформаційному полі;
- пошук елементу із вказаним значенням в заданому інформаційному полі;
- розрахунок суми елементів списку (числове інформаційне поле);
- пошук елементу із рядком максимальної довжини в рядковому інформаційному полі.

30 Реалізувати двозв’язний список, який містить таблицю навчальної групи (П.І.Б., середній бал за сесію) та обчислити середній бал в групі.

7 ДОДАТКОВІ СТРУКТУРИ ДАНИХ

7.1 Множина

Тип даних „множина” реалізований як стандартний тип не у всіх мовах програмування (є в Pascal, відсутній у C/C++), але дуже часто використовується в програмуванні і реалізовується засобами визначення типів користувача, тому дамо йому короткий опис.

Множина – така структура, яка є набором даних одного і того ж типу, що не повторюються (кожен елемент множини є унікальним). Порядок слідування елементів множини не має принципового значення.

До множин застосовується стандартний принцип виключення. Це означає, що конкретний елемент або є членом множини, або ні. Множина може бути пустою, таку множину називають нульовою.

Множина є підмножиною іншої множини, якщо в цій другій множині можна знайти усі елементи, які є в першій множині. Відповідно, множина вважається надмножиною іншої множини, якщо вона містить усі елементи цієї другої множини.

Кожен окремий елемент є членом множини, якщо він входить до складу елементів множини.

Над множинами визначені наступні специфічні операції:

1. Об’єднання множин. Результатом є множина, що містить елементи початкових множин.
2. Перетин множин. Результатом є множина, що містить спільні елементи початкових множин.
3. Різниця множин. Результатом є множина, яка містить елементи першої множини, які не входять в другу множину.
4. Симетрична різниця. Результатом є множина, яка містить елементи, які входять до складу однієї або другої множини (але не обох).
5. Перевірка на входження елемента в множину. Результатом цієї операції є значення логічного типу, що вказує чи входить елемент в множину.

7.2 Об’єднання

Об’єднання представляють собою частковий випадок структури, усі поля якої розміщуються за однією ж і тою ж адресою. Формат опису такий же, як і в структури. Довжина об’єднання рівна найбільшій із довжин його полів. У кожен момент часу в змінній типу об’єднання зберігається тільки одне значення, і відповідальність за його правильне використання лягає на програміста.

Об’єднання застосовуються для економії пам’яті в тих випадках, коли відомо, що більше одного поля одночасно не потрібно, а також для різної інтерпретації одного і того ж бітового представлення.

Дуже часто деякі об’єкти програми відносяться до одного й того ж класу, відрізняючись лише деякими деталями. У цьому випадку застосовують комбінацію структурного типу і об’єднання. Об’єднання використовують як поля структури, при цьому в структурі включають поле, яке визначає, який саме елемент об’єднання використовується в кожний момент.

У загальному випадку змінна структура буде складатися з трьох частин: набір спільних компонентів, мітки активного компоненту і частини зі змінними компонентами.

В мові С++ загальна форма оголошення об'єднання виглядає наступним чином.

union ім'я_типу{
список_членів
}Список_змінних;

Доступ до членів змінних відбувається аналогічно структурам.

7.3 Таблиці

Елементами векторів і масивів можуть бути інтегровані структури. Одна з таких складних структур – таблиця. З фізичної точки зору таблиця є вектором, елементами якого є структури. Характерною логічною особливістю таблиць є те, що доступ до елементів таблиці проводиться не за номером (індексом), а за ключем – значення однієї з властивостей об’єкту, який описується структурою-елементом таблиці. Ключ – це властивість, що ідентифікує дану структуру в множині однотипних структур і є, як правило, унікальним в даній таблиці. Ключ може включатися до складу структури і бути одним з його полів, але може і не включатися в структуру, а обчислюватися за деякими її властивостями. Таблиця може мати один або декілька ключів.

Основною операцією при роботі з таблицями є операція доступу до структури за ключем. Вона реалізовується процедурою пошуку. Оскільки пошук може бути значно більш ефективним в таблицях, впорядкованих за значеннями ключів, досить часто над таблицями необхідно виконувати операції сортування.

Іноді розрізняють таблиці з фіксованою і із змінною довжиною структури. Очевидно, що таблиці, які об’єднують структури ідентичних типів, будуть мати фіксовані довжини структур. Необхідність в змінній довжині

може виникнути в задачах, подібних до тих, які розглядалися для об'єднань. Як правило таблиці для таких задач і складаються із структур, до складу яких входять об'єднання, тобто зводяться до фіксованої (максимальній) довжини структури. Значно рідше зустрічаються таблиці з дійсно змінною довжиною структури. Хоча в таких таблицях і економиться пам'ять, але можливості роботи з такими таблицями обмежені, оскільки за номером структури неможливо визначити її адресу. Таблиці із структурами змінної довжини обробляються тільки послідовно – в порядку зростання номерів структур. Доступ до елемента такої таблиці звичайно здійснюється в два кроки. На першому кроці вибирається постійна частина структури, в якій міститься, – в явному чи неявному вигляді – довжина структури. На другому кроці вибирається змінна частина структури у відповідності з її довжиною. Додавши до адреси поточної структури її довжину, одержують адресу наступної структури.

7.4 Стеки

Для кращого розуміння організації стеку потрібно уявити, наприклад, магазин з патронами автомату: патрони вставляються зверху, один за одним, і так само видаляються. Стек організований дуже схоже.

Стеком називається множина деякої змінної кількості даних, над якою виконуються наступні операції:

- поповнення стеку новими даними;
- перевірка, яка визначає чи стек пустий;
- перегляд останніх добавлених даних;
- знищення останніх добавлених даних.

На основі такого функціонального опису, можна сформувати логічний опис. Стек – це такий послідовний список із змінній довжиною, включення і виключення елементів з якого виконуються тільки з одного боку списку. Застосовуються і інші назви стеку – магазин, пам'ять що функціонує за принципом LIFO (Last – In – First – Out – „останнім прийшов – першим вийшов").

Самий „верхній" елемент стеку, тобто останній добавлений і ще не знищений, відіграє особливу роль: саме його можна модифікувати й знищувати. Цей елемент називають вершиною стеку. Іншу частину стеку називають тілом стеку. Тіло стеку, само собою, є стеком: якщо виключити зі стеку його вершину, то тіло перетворюється в стек.

Основні операції над стеком – включення нового елемента (**push** – заштовхувати) і виключення елемента зі стеку (**pop** – вискакувати).

Корисними можуть бути також допоміжні операції:

- визначення поточної кількості елементів в стеку;
- очищення стеку;
- „неруйнуюче” читання елемента з вершини стека, яке може бути реалізоване, як комбінація основних операцій – виключити елемент зі стеку та включити його знову в стек.

Стек може бути реалізовано за допомогою масиву або списку.

При представленні стеку в статичній пам’яті для стеку виділяється пам’ять, як для вектора. В описі цього вектора окрім звичайних для вектора параметрів повинен знаходитися також покажчик стеку – адреса вершини стека. Обмеження даного представлення полягає в тому, що розмір стеку обмежений розмірами вектора.

Покажчик стеку може вказувати або на перший вільний елемент стеку, або на останній записаний в стек елемент. Однаково, який з цих двох варіантів вибрати, важливо надалі строго дотримуватися його при обробці стеку.

При занесенні елементу в стек елемент записується на місце, яке визначається покажчиком стеку, потім покажчик модифікується так, щоб він вказував на наступний вільний елемент (якщо покажчик вказує на останній записаний елемент, то спочатку модифікується покажчик, а потім проводиться запис елемента). Модифікація покажчика полягає в надбавці до нього або у відніманні від нього одиниці (стек росте у бік збільшення адреси).

Операція виключення елемента полягає в модифікації покажчика стеку (в напрямку, зворотному модифікації при включенні) і вибірці значення, на яке вказує покажчик стеку. Після вибірки комірка, в якій розміщувався вибраний елемент, вважається вільною.

Операція очищення стеку зводиться до запису в покажчик стеку початкового значення – адреси початку виділеної ділянки пам’яті.

Визначення розміру стека зводиться до обчислення різниці покажчиків: покажчика стеку й адреси початку ділянки.

При зв’язному представленні стеку кожен елемент стеку складається із значення і покажчика, який вказує на попередньо занесений у стек елемент. Зв’язне представлення викликає втрату пам’яті, що викликане наявністю покажчика в кожному елементі стеку, і представляє інтерес тільки у випадку, коли важко визначити максимальний розмір стеку.

Отже, для зв’язного представлення стеку потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Для виконання операцій над стеком потрібен один покажчик на вершину стеку. Створення пустого стеку полягатиме у присвоєнні покажчику на вершину нульового значення, що означатиме, що стек пустий.

Послідовність кроків для добавлення елементу в стек складається з декількох кроків:

1. Виділити пам’ять під новий елемент стеку;
1. Занесення значення в інформаційне поле;
2. Встановлення зв’язку між ним і „старою” вершиною стеку;
3. Переміщення вершини стеку на новий елемент.

Вилучення елементу зі стеку також проводять за кілька кроків:

4. Зчитування інформації з інформаційного поля вершини стеку;
5. Встановлення на вершину стеку допоміжного покажчика;
6. Переміщення покажчика вершини стеку на наступний елемент;
7. Звільнення пам’яті, яку займає „стара” вершина стеку.

7.5 Черга

Назва «черга» цілком розкриває цей тип даних. Уявіть собі чергу до каси супермаркету.

Чергою називається множина змінної кількості даних, над якою можна виконувати наступні операції:

- поповнення черги новими даними;
- перевірка, яка визначає чи пуста черга;
- перегляд перших добавлених даних;
- знищення самих перших добавлених даних.

На основі такого функціонального опису, можна сформувати логічний опис. Чергою FIFO (First – In – First – Out – „першим прийшов – першим вийшов”). називається такий послідовний список із змінній довжиною, в якому включення елементів виконується тільки з одного боку списку (хвіст черги), а виключення – з другого боку (голова черги).

Основні операції над чергою – ті ж, що і над стеком – включення, виключення, визначення розміру, очищення, „неруйнуюче” читання.

При представленні черги вектором в статичній пам’яті на додаток до звичайних для опису вектора параметрів в ньому повинні знаходитися два покажчики: на голову і на хвіст черги. При включенні елемента в чергу елемент записується за адресою, яка визначається покажчиком на хвіст, після чого цей покажчик збільшується на одиницю. При виключенні елемента з черги вибирається елемент, що адресується покажчиком на голову, після чого цей покажчик зменшується на одиницю.

Очевидно, що з часом покажчик на хвіст при черговому включенні елемента досягне верхньої межі тієї ділянки пам’яті, яка виділена для черги.

Проте, якщо операції включення чергувати з операціями виключення елементів, то в початковій частині відведеної під чергу пам’яті є вільне місце. Для того, щоб місця, займані виключеними елементами, могли бути повторно використані, черга замикається в кільце: покажчики (на початок і на кінець), досягнувши кінця виділеної області пам’яті, перемикаються на її початок. Така організація черги в пам’яті називається кільцевою чергою. Можливі, звичайно, і інші варіанти організації: наприклад, всякий раз, коли покажчик кінця досягне верхньої межі пам’яті, зсовувати всі не порожні елементи черги до початку ділянки пам’яті, але як цей, так і інші варіанти вимагають переміщення в пам’яті елементів черги і менш ефективні, ніж кільцева черга.

У початковому стані покажчики на голову і хвіст вказують на початок ділянки пам’яті. Рівність цих двох покажчиків є ознакою порожньої черги. Якщо в процесі роботи з кільцевою чергою кількість операцій включення перевищує кількість операцій виключення, то може виникнути ситуація, в якій покажчик кінця „наздожене” покажчик початку. Це ситуація заповненої черги, але якщо в цій ситуації покажчики порівняються, ця ситуація буде така ж як при порожній черзі. Для розрізнення цих двох ситуацій до кільцевої черги пред’являється вимога, щоб між покажчиком кінця і покажчиком початку залишався „проміжок” з вільних елементів. Коли цей „проміжок” скорочується до одного елемента, черга вважається заповненою і подальші спроби запису в неї блокуються. Очищення черги зводиться до запису одного і того ж (не обов’язково початкового) значення в обидва покажчики. Визначення розміру полягає в обчисленні різниці покажчиків з урахуванням кільцевої природи черги.

При зв’язному представленні черги кожен елемент черги складається із значення і покажчика, який вказує на попередньо занесений у чергу елемент.

Зв’язне представлення викликає втрату пам’яті, що викликане наявністю покажчика в кожному елементі черги, і представляє інтерес тільки у випадку, коли важко визначити максимальний розмір черги. Для зв’язного представлення черги потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Для виконання операцій над чергою потрібно два покажчики: на голову і хвіст черги. Створення пустої черги полягатиме у присвоєнні покажчикам на голову і хвіст черги нульових значень, що означатиме, що черга пуста.

Послідовність дій для добавлення елемента в кінець черги складається з декількох кроків:

1) виділити пам’ять під новий елемент черги;
2) занесення значення в інформаційне поле;
3) занесення нульового значення в покажчик;

4) встановлення зв’язку між ним і останнім елементом черги і новим, враховуючи випадок пустої черги;
5) переміщення покажчика кінця черги на новий елемент.
6) вилучення елементу з черги також проводять за кілька кроків:
7) зчитування інформації з інформаційного поля голови черги;
8) встановлення на голову черги допоміжного покажчика;
9) переміщення покажчика початку черги на наступний елемент;
10) звільнення пам’яті, яку займав перший елемент черги.

В реальних задачах іноді виникає необхідність у формуванні черг, відмінних від наведених структур. Порядок вибірки елементів з таких черг визначається пріоритетами елементів. Пріоритет в загальному випадку може бути представлений числовим значенням, яке обчислюється або на підставі значень яких-небудь полів елемента, або на підставі зовнішніх чинників. Так попередньо наведені структури стек і черги можна трактувати як пріоритетні черги, в яких пріоритет елемента залежить від часу його включення в структуру. При вибірці елемента всякий раз вибирається елемент з щонайбільшим пріоритетом.

Черги з пріоритетами можуть бути реалізовані на лінійних структурах – в суміжному або зв’язному представленні. Можливі черги з пріоритетним включенням – в яких послідовність елементів черги весь час підтримується впорядкованою, тобто кожний новий елемент включається на те місце в послідовності, яке визначається його пріоритетом, а при виключенні завжди вибирається елемент з голови. Можливі і черги з пріоритетним виключенням – новий елемент включається завжди в кінець черги, а при виключенні в черзі шукається (цей пошук може бути тільки лінійним) елемент з максимальним пріоритетом і після вибірки вилучається з послідовності. І в тому, і в іншому варіанті потрібний пошук, а якщо черга розміщується в статичній пам’яті – ще і переміщення елементів.

7.6 Деки

Дек – особливий вид черги. Дек (deq – double ended queue, тобто черга з двома кінцями) – це такий послідовний список, в якому як включення, так і виключення елементів, може здійснюватися з будь-якого з двох кінців списку. Так само можна сформулювати поняття деку, як стек, в якому включення і виключення елементів може здійснюватися з обох кінців.

Деки рідко зустрічаються у своєму первісному визначенні. Окремий випадок деку – дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури деку аналогічні логічній і фізичній структурі кільцевої

черги. Проте, стосовно деку доцільно говорити не про голову і хвіст, а про лівий і правий кінець.

Над деком доступні наступні операції:

- включення елемента праворуч;
- включення елемента ліворуч;
- виключення елемента з права;
- виключення елемента з ліва;
- визначення розміру;
- очищення.

Фізична структура деку в статичній пам’яті ідентична структурі кільцевої черги.

7.7 Рядки

Рядки як структура даних призначена для обробки текстової інформації. Рядок – це лінійно впорядкована послідовність символів, які належать до скінченої множини символів, яка називається алфавітом. В сімействі мов С їм відповідає тип string.

Рядки мають наступні важливі властивості:

- їхня довжина, як правило, змінна, хоч алфавіт фіксований;
- звичайне звернення до символів рядку йде з будь-якого одного боку послідовності (важлива впорядкованість послідовності, а не її індексація);
- метою доступу до рядки є на окремий її елемент, а ланцюжок символів.

Кажучи про рядки, звичайно мають на увазі текстові рядки – рядки, що складаються з символів, які входять в алфавіт якої-небудь вибраної мови, цифр, розділових знаків і інших службових символів. Текстовий рядок є найбільш універсальною формою представлення будь-якої інформації.

Базовими операціями над рядками є:

- визначення довжини рядку;
- присвоєння рядку;
- конкатенація (зчеплення) рядків;
- виділення підрядку;
- пошук входження.

Операція визначення довжини рядка має вид функції, яка повертає значення – ціле число – поточна кількість символів в рядку. Операція присвоєння має такий же сенс, що і для інших типів даних.

Операція порівняння рядків має такий же сенс, що і для інших типів даних. Порівняння рядків проводиться за наступними правилами. Порівнюються перші символи двох рядків. Якщо символи не рівні, то рядок,

що містить символ, місце якого в алфавіті ближче до початку, вважається меншою. Якщо символи рівні, порівнюються другі, треті і т.д. символи. При досягненні кінця в одному з рядків рядок меншої довжини вважається меншим. При рівності довжин рядків і попарній рівності всіх символів в них рядки вважаються рівними.

Результатом операції зчеплення двох рядків є рядок, довжина якого дорівнює сумарній довжині рядків-операндів, а значення відповідає значенню першого операнда, за яким безпосередньо слідує значення другого операнда.

Операція виділення підрядка виділяє з початкового рядка послідовність символів, починаючи із заданої позиції, із заданою довжиною.

Операція пошуку входження знаходить місце першого входження підрядку еталону в початковий рядок. Результатом операції може бути номер позиції в початковому рядку, з яким починається входження еталону або покажчик на початок входження. У разі відсутності входження результатом операції повинне бути деяке спеціальне значення, наприклад, від'ємний номер позиції або порожній покажчик.

Найпростішим способом є представлення рядку у вигляді вектора постійної довжини. При цьому в пам'яті відводиться фіксована кількість байт, в які записуються символи рядка. Якщо рядок менша відведеного під нього вектора, то зайві місця заповнюються пропусками, а якщо рядок виходить за межі вектора, то зайві (праві) символи повинні бути відкинуті.

Можливе представлення рядка вектором змінної довжини з ознакою завершення. Цей і всі подальші за ним методи враховують змінну довжину рядків. Ознака завершення – це особливий символ, який належить до алфавіту (таким чином, корисний алфавіт виявляється меншим на один символ), і займає ту ж кількість розрядів, що і всі інші символи. Витрати пам'яті при цьому способі складають 1 символ на рядок.

Окрім ознаки завершення можна використати лічильник символів – це ціле число, і для нього відводиться достатня кількість бітів, щоб їх з надлишком вистачало для представлення довжини найдовшого рядку, який можна представити. При використовуванні лічильника символів можливий довільний доступ до символів в межах рядку.

Представлення рядків списком у пам'яті забезпечує гнучкість у виконанні різноманітних операцій над ними (зокрема, операцій включення і виключення окремих символів і цілих ланцюжків) і використовування системних засобів управління пам'яттю при виділенні необхідного об'єму пам'яті для рядку. Проте, при цьому виникають додаткові затрати пам'яті. Іншим недоліком такого представлення рядків є те, що логічно сусідні

елементи рядки не є фізично сусідніми в пам’яті. Це ускладнює доступ до груп елементів рядку в порівнянні з доступом у векторному представленні.

При представленні рядку однозв’язним лінійним списком кожний символ рядка представляється у вигляді елемента зв’язного списку; елемент містить код символу і покажчик на наступний елемент. Одностороннє зчеплення представляє доступ тільки в одному напрямі уздовж рядку.

При використанні двох-зв’язних лінійних списків у кожний елемент списку додається також покажчик на попередній елемент. Двостороннє зчеплення допускає двосторонній рух уздовж списку, що може значно підвищити ефективність виконання деяких рядкових операцій.

Блочно-зв’язне представлення рядків дозволяє в більшості операцій уникнути витрат, які пов’язані з управлінням динамічною пам’яттю, але в той же час забезпечує достатньо ефективне використовування пам’яті при роботі з рядками змінної довжини.

7.8 Вектор

Вектор являється контейнером, який містить послідовності елементів, і який може динамічно змінювати свій розмір. На подобі масивів, клас vector використовує послідовне скупчення елементів у пам’яті, що дозволяє виконувати арифметичні дії над вказівниками для доступу до його елементів. Також, vector має власний вбудований клас ітераторів вільного доступу, над якими користувач може виконувати операції і які за функціоналом подібні до звичайних вказівників.

Для оптимізації використання пам’яті, std::vector містить механізм резервного надвиділення пам’яті, який дозволяє скоротити витрати на повторне виділення пам’яті для нових об’єктів. Тобто, клас vector, самостійно виділяє більше пам’яті ніж потрібно, щоб забезпечити швидку вставку елементів у кінець послідовності. Об’єм резервної пам’яті можна контролювати за допомогою спеціальної функції reserve, яка також дозволяє заздалегідь виділити необхідну пам’ять для усіх об’єктів вектора.

По суті діла, vector – це клас, подібний на масив, який самостійно управляє пам’яттю, що звільняє програміста від рутинної роботи над виділенням пам’яті, її звільненням, переміщенням елементів і т.д.

7.9 Map

Інша назва map – асоціативний масив. В ньому зберігаються дані у вигляді пар [ключ, дані]. При цьому ключами можуть бути будь-які об'єкти, які задовільняють умову унікальності ключів. В подібному

контейнері multimap можуть зберігатися елементи, що адресуються ключами? серед яких допускаються повтори.

Асоціативні масиви, в тому числі map, особливо ефективні для доступу до елементів за їх ключем, на відміну від послідовних масивів, які більш ефективні для отримання доступу до елементів по їх порядковому індексу.

Асоціативні масиви гарантовано виконують операції вставки, видалення і пошуку елементу за логарифмічний час - O(log n). Як правило вони реалізується за допомогою одного з різновидів бінарного збалансованого дерева і підтримують двосторонню ітерацію.

Одна з реалізацій map наведена у бібліотеці STL (Standard Template Library – стандартна бібліотека шаблонів) C++.

Найкраще зрозуміти застосування асоціативних масивів на прикладі:

```
#include <iostream>
#include <map>// Заголовок в якому знаходиться map
#include <string>

using namespace std;

int main()
{
   map<string,int>  months;//Оголошуємо асоціативний масив з ключами - рядками, і цілими даними.
   months["січень"]=1;
   months["лютий"]=2;

   //  ...              // Заради економії місця пропущено тепліші місяці

   months["грудень"]=12;

   map<string,int>::iterator it; // Ітератор по контейнеру
   for(it=months.begin();it!=months.end();it++) //Виводимо всі
   {
      cout << it->first << " " << it->second << endl;// Ітератор це пара: ключ, значення
      // Ключ в полі first, а значення у полі second
   }
   it=months.find("березень");
   cout << "Виберемо місяць " << it->first << endl;
   it++;
```

```
    cout << "Наступний за ним: " << it->first;

    return 0;
}
```

7.10 Графи

Граф – це складна нелінійна багато-зв'язна динамічна структура, що відображає властивості і зв'язки складного об'єкту.

Ця багато-зв'язна структура має наступні властивості:

- на кожний елемент (вузол, вершину) може бути довільна кількість посилань;
- кожний елемент може мати зв'язок з будь-якою кількістю інших елементів;
- кожний зв'язок (ребро, дуга) може мати напрям і вагу.

У вузлах графа міститься інформація про елементи об'єкту. Зв'язки між вузлами задаються ребрами графа. Ребра графа можуть мати спрямованість, тоді вони називаються орієнтованими, в іншому випадку – неорієнтовані. Граф, усі зв'язки якого орієнтовані, називається орієнтованим графом; граф зі всіма неорієнтованими зв'язками – неорієнтованим графом; граф із зв'язками обох типів – змішаним графом.

Існує два основні методи представлення графів в пам'яті комп'ютера: матричний і зв'язними нелінійними списками. Вибір методу представлення залежить від природи даних і операцій, що виконуються над ними. Якщо задача вимагає великої кількості включень і виключень вузлів, то доцільно представляти граф зв'язними списками; інакше можна застосувати і матричне представлення.

При використанні матриць суміжності їхні елементи представляються в пам'яті комп'ютера елементами масиву. При цьому, для простого графа матриця складається з нулів і одиниць, для мультиграфа – з нулів і цілих чисел, які вказують кратність відповідних ребер, для зваженого графа – з нулів і дійсних чисел, які задають вагу кожного ребра.

Орієнтований граф представляється зв'язним нелінійним списком, якщо він часто змінюється або якщо півміри входу і виходу його вузлів великі.

Багато-зв'язна структура – граф – знаходить широке застосування при організації банків даних, управлінні базами даних, в системах програмного імітаційного моделювання складних комплексів, в системах штучного інтелекту, в задачах планування і в інших сферах.

7.11 Дерева

Дерево – це граф, який характеризується наступними властивостями:

- існує єдиний елемент (вузол або вершина), на який не посилається ніякий інший елемент, – він називається коренем.
- починаючи з кореня і слідуючи по певному ланцюжку покажчиків, що містяться в елементах, можна здійснити доступ до будь-якого елемента структури.
- на кожний елемент, крім кореня, є єдине посилання, тобто кожний елемент адресується єдиним покажчиком.

Назва „дерево” виникла з логічної еквівалентності деревовидної структури абстрактному дереву з теорії графів. Лінія зв’язку між парою вузлів дерева називається гілкою. Ті вузли, які не посилаються ні на які інші вузли дерева, називаються листям. Вузол, що не є листком або коренем, вважається проміжним або вузлом галуження.

В багатьох застосування відносний порядок проходження вершин на кожному окремому ярусі має певне значення. При представленні дерева в пам’яті комп’ютера такий порядок вводиться автоматично, навіть якщо він сам по собі довільний. Порядок проходження вершин на деякому ярусі можна легко ввести, позначаючи одну вершину як першу, іншу – як другу і т.д. Замість впорядковування вершин можна задавати порядок на ребрах. Якщо в орієнтованому дереві на кожному ярусі заданий порядок проходження вершин, то таке дерево називається впорядкованим деревом.

Введемо ще деякі поняття, пов’язані з деревами. Вузол X називається предком, або батьком, а вузли Y і Z називаються нащадками, або синами, їх, відповідно, між собою називають братами (рис. 7.1). Причому, лівий син є старшим сином, а правий – молодшим. Кількість піддерев даної вершини називається мірою цієї вершини.

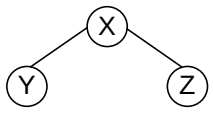


Рис.7.1 – Приклад вершин дерева

Якщо з дерева прибрати коріння і ребра, що сполучають коріння з вершинами першого ярусу, то вийде деяка множина незв’язаних дерев. Множина незв’язаних дерев називається лісом.

Є ряд способів графічного зображення дерев. Перший спосіб полягає у використовуванні для зображення піддерев відомого методу діаграм Венна, другий – методу дужок, що вкладаються одна в одну, третій спосіб – це спосіб,

який використовується при складанні змісту книг. Останній спосіб, що базується на форматі з нумерацією рівнів, схожий з методами, які використовуються в мовах програмування. При застосуванні цього формату кожній вершині приписується числовий номер, який повинен бути менший номерів, приписаних кореневим вершинам приєднаних до неї піддерев.

Повне дерево містить максимально можливу кількість вузлів на кожному рівні, крім нижнього. Повні дерева мають ряд важливих властивостей.

По-перше, це найкоротші дерева, які можуть містити задану кількість вузлів. Досить 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 ж порядку.

Існують *m*-арні дерева, тобто такі дерева, в кількість вихідних ребер з кожної вершини менша або рівна *m*. Якщо кількість вихідних ребер з кожної вершини в точності рівна або *m*, або нулю, то таке дерево називається повним m-арним деревом. При *m=2* такі дерева називаються відповідно бінарними, або повними бінарними.

Представити *m*-арне дерево в пам'яті комп'ютера складно, так як кожен елемент дерева повинен містити стільки покажчиків, скільки ребер, виходить з вузла. Це приведе до підвищеної витрати пам'яті, різноманітності початкових елементів і ускладнить алгоритми обробки дерева. Тому *m*-арні дерева, ліс необхідно привести до бінарних для економії пам'яті і спрощенню алгоритмів. Усі вузли бінарного дерева представляються в пам'яті однотипними елементами з двома покажчиками, крім того, операції над бінарними деревами виконуються просто і ефективно.

Правило побудови бінарного дерева з будь-якого дерева:

1. В кожному вузлі залишити тільки гілку до старшого сина;
2. З'єднати горизонтальними ребрами всіх братів одного батька;
3. Таким чином перебудувати дерево за правилом:
   - лівий син – вершина, розташована під даною;
   - правий син – вершина, розташована праворуч від даної (тобто на одному ярусі з нею).
4. Розвернути дерево так, щоб усі вертикальні гілки відображали лівих синів, а горизонтальні – правих.

У результаті перетворення будь-якого дерева, в бінарне, виходить дерево у вигляді лівого піддерева, підвішеного до кореня.

У процесі перетворення правий покажчик кожного вузла бінарного дерева буде вказувати на сусіда по рівню. Якщо такого немає, то правий

покажчик – **NULL**. Лівий покажчик буде вказувати на вершину наступного рівня. Якщо такої немає, то покажчик встановлюється на **NULL**.

Описаний вище метод представлення довільних впорядкованих дерев за допомогою бінарних дерев можна узагальнити на представлення довільного впорядкованого лісу.

Правило побудови бінарного дерева з лісу: корені всіх піддерев лісу з'єднати горизонтальними зв'язками. В отриманому дереві вузли в даному прикладі будуть розташовуватися на трьох рівнях. Далі перебудовувати по раніше розглянутому плану. В результаті перетворення впорядкованого лісу в бінарне дерево виходить повне бінарне дерево з лівим і правим піддеревом.

Дерева можна представляти за допомогою зв'язних списків і масивів (або послідовних списків).

Частіше всього використовується зв'язне представлення дерев, так як воно дуже сильно нагадує логічне. Зв'язне зберігання полягає в тому, що задається зв'язок від батька до синів. В бінарному дереві є два покажчики, тому зручно вузол представити у вигляді структури в якій *left* – покажчик на ліве піддерево, *right* – покажчик на праве піддерево, *inf* – містить інформацію, яка зв'язана з вершиною і має наперед визначений тип – *data*.

Над деревами визначені наступні основні операції:

1) пошук вузла із заданим ключем.
2) додавання нового вузла.
3) видалення вузла (піддерева).
4) обхід дерева в певному порядку:
   - низхідний обхід;
   - змішаний обхід;
   - висхідний обхід.

В багатьох задачах, пов'язаних з деревами, вимагається здійснити систематичний перегляд всіх його вузлів в певному порядку. Такий перегляд називається проходженням або обходом дерева.

Бінарне дерево можна обходити трьома основними способами: низхідним, змішаним і висхідним (можливі також зворотний низхідний, зворотний змішаний і зворотний висхідний обходи). Прийняті назви методів обходу зв'язані з часом обробки кореневої вершини: до того як оброблено обидва його піддерева, після того, як оброблено ліве піддерево, але до того як оброблено праве, після того, як оброблено обидва піддерева. Використовувані назви методів відображають напрям обходу в дереві: від кореневої вершини вниз до листя – низхідний обхід; від листя вгору до кореня – висхідний обхід, і змішаний обхід – від найлівішого листка дерева через корінь до найправішого листка.

Схемно алгоритм обходу бінарного дерева відповідно до низхідного способу може виглядати таким чином:

1. В якості чергової вершини взяти корінь дерева. Перейти до пункту 2.

2. Провести обробку чергової вершини відповідно до вимог задачі. Перейти до пункту 3.

3.а) Якщо чергова вершина має обидві гілки, то в якості нової вершини вибрати ту вершину, на яку посилається ліва гілка, а вершину, на яку посилається права гілка, занести в стек; перейти до пункту 2;

3.б) якщо чергова вершина є кінцевою, то вибрати в якості нової чергової вершини вершину із стека, якщо він не порожній, і перейти до пункту 2; якщо ж стек порожній, то це означає, що обхід всього дерева закінчений, перейти до пункту 4;

3.в) якщо чергова вершина має тільки одну гілку, то в якості чергової вершини вибрати ту вершину, на яку ця гілка вказує, перейти до пункту 2.

4. Кінець алгоритму.

Алгоритм істотно спрощується при використовуванні рекурсії. Так, низхідний обхід можна описати таким чином:

1). Обробка кореневої вершини;

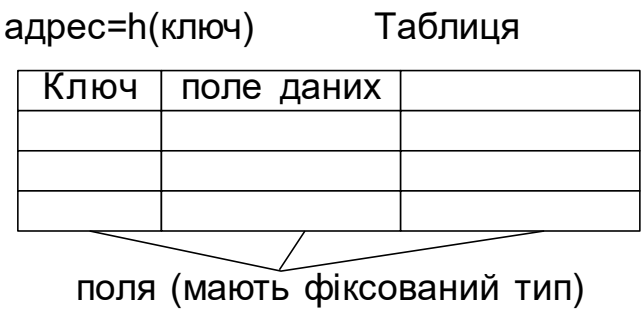
2). Низхідний обхід лівого піддерева;

3). Низхідний обхід правого піддерева.

Інші методи поступаються даному підходу.

7.12 Хешування даних

Для прискорення доступу до даних можна використовувати попереднє їх впорядкування у відповідності зі значеннями ключів. При цьому можуть використовуватися методи пошуку в упорядкованих структурах даних, наприклад, метод бінарного пошуку, що суттєво скорочує час пошуку даних за значенням ключа. Проте при добавленні нового запису потрібно дані знову впорядкувати. Втрати часу на повторне впорядкування можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних використовується так зване випадкове впорядкування або хешування. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції h, яка використовується для „обчислення” адреси за значенням ключа.



Хеш-таблицею називається структура даних, що дозволяє зберігати пари виду «ключ» - «хеш-код» і підтримує операції пошуку, вставки і видалення елемента. Хеш-таблиці застосовуються з метою прискорення пошуку, наприклад, під час запису текстових полів в базі даних може розраховуватися їх хеш-код, і дані можуть міститися в розділ, що відповідає цьому хеш-коду. Тоді при пошуку даних треба буде спочатку обчислити хеш-код тексту, і відразу стане відомо, в якому розділі їх треба шукати. Тобто, шукати треба буде не по всій базі, а тільки по одному її розділу, а це прискорює пошук.

Алгоритм хешування використовує деяку функцію, яка визначає імовірне розміщення елемента в таблиці на основі значення шуканого елементу.

Ідеальною хеш-функцією є така хеш-функція, яка для будь-яких двох неоднакових ключів дає неоднакові адреси. Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів відомі наперед. Така організація даних носить назву „досконале хешування”.

Наприклад, потрібно запам’ятати декілька записів, кожен з яких має унікальний ключ зі значенням від 1 до 100. Для цього можна створити масив з 100 комірками і присвоїти кожній комірці нульовий ключ. Щоб добавити в масив новий запис, дані з нього просто копіюються у відповідну комірку масиву. Щоб добавити запис з ключем 37, дані з нього копіюються в 37 позицію в масиві. Щоб знайти запис з певним ключем – вибирається відповідна комірка масиву. Для вилучення запису ключу відповідної комірки масиву просто присвоюється нульове значення. Використовуючи цю схему, можна добавити, знайти і вилучити елемент із масиву за один крок.

У випадку наперед невизначеної множини значень ключів і обмеженні розміру таблиці підбір досконалої функції складний. Тому часто використовують хеш-функції, які не гарантують виконанні умови.

Наприклад, база даних співробітників може використовувати в якості ключа ідентифікаційний номер. Теоретично можна було б створити масив, кожна комірка якого відповідала б одному з можливих чисел; але на практиці для цього не вистарчить пам’яті або дискового простору. Якщо для зберігання

одного запису потрібно 1 КБ пам’яті, то такий масив зайняв би 1 ТБ (мільйон МБ) пам’яті. Навіть якщо можна було б виділити такий об’єм пам’яті, така схема була б дуже неекономною. Якщо штат фірми менше 10 мільйонів співробітників, то більше 99 процентів масиву буде пустою.

Щоб вирішити цю проблему, схеми хешування відображають потенційно велику кількість можливих ключів на достатньо компактну хеш-таблицю. Якщо на фірмі працює 700 співробітників, можна створити хеш-таблицю з 1000 комірок. Схема хешування встановлює відповідність між 700 записами про співробітників і 1000 позиціями в таблиці. Наприклад, можна розміщати записи в таблиці у відповідності з трьома першими цифрами ідентифікаційного номеру. При цьому запис про співробітника з номером 123456789 буде знаходитися в 123 комірці таблиці.

Очевидно, що оскільки існує більше можливих значень ключа, ніж комірок в таблиці, то деякі значення ключів можуть відповідати одним і тим коміркам таблиці. Даний випадок носить назву „колізія”, а такі ключі називаються „ключі-синоніми”.

Щоб уникнути цієї потенційної проблеми, схема хешування повинна включати в себе алгоритм вирішення конфліктів, який визначає послідовність дій у випадку, якщо ключ відповідає позиції в таблиці, яка вже зайнята іншим записом.

Усі методи використовують для вирішення конфліктів приблизно однаковий підхід. Вони спочатку встановлюють відповідність між ключем запису і розміщенням в хеш-таблиці. Якщо ця комірка вже зайнята, вони відображають ключ на іншу комірку таблиці. Якщо вона також вже зайнята, то процес повторюється знову до тих пір, поки нарешті алгоритм не знайде пусту комірку в таблиці. Послідовність позицій, які перевіряються при пошуку або вставці елемента в хеш-таблицю, називається тестовою послідовністю.

В результаті, для реалізації хешування необхідні три речі:

– структура даних (хеш-таблиця) для зберігання даних;

– функція хешування, яка встановлює відповідність між значеннями ключа і розміщенням в таблиці;

– алгоритм вирішення конфліктів, який визначає послідовність дій, якщо декілька ключів відповідають одній комірці таблиці.

Контрольні питання

1. Дайте характеристику множині як структурі даних.
2. Коли потрібно використовувати об’єднання?
3. Які особливості таблиці як структури даних?
4. Порівняйте структури даних стек, черга, та дек.

5. Наведіть призначення типу даних «рядок» та основні операції з ним.
6. Поясніть відмінності між масивами, векторами та map.
7. В яких випадках слід використовувати структури даних графи та дерева?
8. Які алгоритми обходу бінарних дерев існують?
9. В чому полягає хешування даних, де його використовують?

Практичні завдання

1. Реалізувати чергу на основі однозв’язного списку, створеного в практичному завданні розділу 6, та продемонструвати основні операції з чергою.
2. Реалізувати стек на основі однозв’язного списку, створеного в практичному завданні розділу 6, та продемонструвати основні операції зі стеком.
3. Реалізувати чергу із пріоритетами на основі двозв’язного списку, створеного в практичному завданні розділу 6, та продемонструвати основні операції зі стеком.
4. Реалізувати структуру «дерево» на основі даних про групу (П.І.Б., середній бал за сесію), та збалансувати дерево за значенням середнього балу.

ДОДАТОК 1. ПРИКЛАДИ РОБОТИ З C#

Загальна структура програми в C#.

Програма на мові C # може складатися з одного або декількох файлів. Кожен файл може містити нуль або більше просторів імен. Простір імен може включати такі елементи, як класи, структури, інтерфейси, перерахування та делегати, а також інші простори імен. Нижче приведена скелетна структура програми C #, яка містить всі зазначені елементи.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

Використання Microsoft Visual Studio.

Жоден сучасний програміст, для забезпечення високої ефективності своєї праці, не може обійтися без використання спеціальних середовищ програмування. Однією з таких програм є Microsoft Visual Studio. Якщо середовище Visual Studio уже інстальоване на вашій машині і ви його запустили то для створення нового проекту C# необхідно скористатися кнопкою Create Project на стартовій сторінці. Ліворуч, у вікні New Project, вибираємо Visual C#, а праворуч тип додатку - Console Application. Слід вказати ім'я проекту, наприклад MyConsoleProgect та натиснути на кнопку ОК. Діалогове вікно закриється. Нова програма готова. Вибравши у меню Debug пункт StartWithoutDebugging або натиснувши Ctrl+F5 можна запускати код на виконання.

Введення/виведення даних

Метод ReadLine() ще називають оператором введення і працює він наступним чином.

1) програма зупиняється;
2) програма чекає доки користувач введе данні з клавіатури і натисне Enter;
3) після натискання Enter програма продовжує своє виконання;
4) після натискання Enter дані введені з клавіатури присвоюються відповідній змінній, якщо не було помилок перетворення (потрапляють у ту комірку пам'яті, що зберігає значення відповідної змінної).

Метод WriteLine() дозволяє вивести дані.

Приклад 1:

Console.Write ("Введіть ім'я та натисніть Enter:"); //виведення на екран запрошення

string name = Console.ReadLine(); //оголошення рядкової величини на надання їй значення новим методом

Console.WriteLine ("Привіт, " + name + "!") //виведення тексту "Привіт, " та значення, що містить змінна "name"

ReadLine() завжди повертає дані лише строкового типу, і з цим весь час потрібно рахуватись. Щоб отримати з рядкових даних числові значення, треба їх попередньо конвертувати, наприклад, використовувати методи класу Convert.

Приклад 2:

Console.Write ("Введіть свій вік і натисніть Enter:"); //виведення на екран тексту

int age = Convert.ToInt32 (Console.ReadLine ()); //оголошення цілочисельної змінної "age". Присвоєння їй значення конвертованого у чилове з введної рядкової величини.

Console.Write ("Введіть свій зріст і натисніть Enter (можна використовувати дробове значення):"); //знову виведення повідомлення на екран

double height = Convert.ToDouble (Console.ReadLine ()); //оголошення дійсної змінної "height". Присвоєння їй значення конвертованого у чилове з введної рядкової величини.

Console.WriteLine ("Отже, вам {0} років, а зріст {1}.", age, height); //виведення текстового повідомлення до якого включено аргументи {0} та {1}. Значення аргументів перераховані після текстової величини.

Ще, рядок Console.ReadLine(); дуже часто ставлять в кінці консольних програм без присвоєння можливих результатів введення даних жодним аргументам. ReadLine () зупинить виконання програми допоки користувач не натисне ENTER. Коли ENTER натиснуто, програма збирає всю інформацію, що ввів користувач з клавіатури, і нічого з нею не робить, тому що в цьому рядку не вказано присвоєння жодній змінній.

Створення програми «Hello World»

У наступній процедурі створюється версія для C # традиційної програми "Hello World". Програма відображає рядок Hello World!

1) Запустіть Visual Studio.
2) У меню Файл виберіть Створити, Проект.
3) Відкриється діалогове вікно Новий проект.
4) Розгорніть вузол Встановлені, розгорніть Шаблони, розгорніть Visual C #, а потім виберіть Консольне додаток.
5) У полі Ім'я введіть ім'я для проекту і натисніть кнопку ОК.
6) У браузері рішень з'явиться новий проект.
7) Якщо файл Program.cs не відкритий в редакторі коду, відкрийте контекстне меню Program.cs в браузері рішень, а потім натисніть кнопку Переглянути код.
8) Замініть вміст файлу Program.cs на наступний код.

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open in debug mode.
```

```
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

9) Натисніть клавішу F5, щоб запустити проект. З'являється вікно командного рядка, що містить рядок Hello World!