МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Т. О. Коротєєва

АЛГОРИТМИ
ТА СТРУКТУРИ ДАНИХ

Навчальний посібник

*Рекомендувала Науково-методична рада
Національного університету «Львівська політехніка»*

Львів
Видавництво Львівської політехніки
2014

УДК 004.4(075)
ББК 32.973.2-018.492я7
К 48

Рецензенти:

Мельник Р. А., доктор технічних наук, професор Національного університету «Львівська політехніка»;

Журавчак Л. М., доктор технічних наук, ст. наук. співр. відділу геоелектромагнітних методів Карпатського відділення Інституту геофізики ім. С.І. Субботіна НАН України;

Дзелендзяк У. Ю., кандидат технічних наук, доцент кафедри комп'ютеризованих систем автоматики Національного університету «Львівська політехніка»

Рекомендувала Науково-методична рада
Національного університету «Львівська політехніка»
як навчальний посібник для студентів спеціальності «Програмна інженерія»
(протокол № 8/2013 від 22 квітня 2014 року)

Коротєєва Т. О.

К 48 Алгоритми та структури даних : навч. посібник / Т. О. Коротєєва. – Львів : Видавництво Львівської політехніки, 2014. – 280 с.
ISBN 978-617-607-660-5

Викладено основи класифікації структур даних. Докладно розглянуто рівні організації структур даних та найпоширеніші моделі даних. Наведено способи зображення моделей структур даних у пам'яті комп'ютера. Розглянуто алгоритми сортування та пошуку даних, приклади їх реалізації та обґрунтовано алгоритмічну складність. Фрагменти програм студенти можуть використати для самостійного розроблення власних програм під час виконання лабораторних та курсових завдань. Розглянуто реалізацію структур даних на прикладі мови програмування C++ та описано основні контейнери бібліотеки STL. Розділи супроводжуються контрольними запитаннями та задачами для індивідуального виконання.

Посібник рекомендовано для студентів ВНЗ та коледжів, які навчаються за напрямами «Програмна інженерія» або «Комп'ютерні науки».

УДК 004.4(075)
ББК 32.973.2-018.492я7

© Коротєєва Т.О., 2014
© Національний університет «Львівська політехніка», 2014

ISBN 978-617-607-660-5

ЗМІСТ

1. Структури даних. основні визначення та поняття 7
1.1. Термінологія 7
1.2. Класифікація структур даних 9
1.3. Документування даних 12
Контрольні запитання та вправи 13
Приклади тестових питань 13

2. Рівні організації даних 16
2.1. Логічна організація даних 17
2.2. Представлення даних 20
2.3. Фізична організація даних 26
Контрольні запитання та вправи 32
Приклади тестових питань 33

3. Лінійні структури даних. Стек, черга, дек 34
3.1. Стек 34
3.1.1. Реалізація стека 38
3.1.2. Використання функції assert 42
3.2. Черга 44
3.2.1. Реалізація черги на основі масиву 47
3.3. Дек 48
Контрольні запитання 49
Приклади тестових питань 49

4. Масиви, множини, кортежі 51
4.1. Масиви 51
4.2. Множини і кортежі 53
4.3. Зберігання множин і масивів 56
Контрольні запитання та вправи 58
Приклади тестових питань 59

5. Лінійні списки 60
5.1. Основні визначення та поняття 60
5.2. Однонапрямлені списки 62
5.3. Двонапрямлені списки 64
5.4. Циклічні списки 65
Контрольні запитання та вправи 66
Приклади тестових питань 67

6. Нелінійні структури даних 69
6.1. Таблиці 69

6.2. Дерева ... 71
6.2.1. Бінарні дерева ... 73
6.2.2. Алгоритми обходу дерева ... 74
6.2.3. Зображення в пам'яті комп'ютера графоподібних структур ... 75
6.3. Спискові структури ... 77
6.3.1. Ієрархічні списки ... 78
6.3.2. Організація спискових структур ... 79
6.4. Сіткові структури ... 81
Контрольні запитання та вправи ... 84
Приклади тестових питань ... 85

7. Алгоритми. Складність алгоритмів ... 88
7.1. Визначення та способи зображення алгоритмів ... 88
7.2. Складність алгоритмів ... 90
7.3. Класи алгоритмів ... 93
7.4. Способи реалізації алгоритмів ... 96
7.4.1. Ітерація й рекурсія ... 97
7.4.2. Паралельна обробка ... 99
7.4.3. Співпрограми ... 99
7.5. Документація алгоритмів ... 100
Контрольні запитання ... 101
Приклади тестових питань ... 102

8. Методи сортування ... 103
8.1. Задача сортування ... 103
8.2. Метод простої вибірки ... 103
8.3. Метод бульбашки ... 105
8.4. Швидкий метод сортування ... 107
8.5. Сортування включенням ... 109
8.6. Сортування розподілом ... 111
8.7. Сортування злиттям або об'єднанням ... 112
8.8. Сортування підрахунком ... 114
Контрольні запитання та вправи ... 116
Приклади тестових питань ... 116

9. Пошук даних ... 118
9.1. Послідовний пошук ... 118
9.2. Двійковий пошук ... 120
9.3. Прямий пошук рядка ... 121
9.4. Алгоритм Кнута, Моріса і Прата пошуку в рядку ... 123
9.5. Алгоритм Бойера – Мура пошуку в рядку ... 125
9.6. Пошук у таблиці ... 128

9.6.1. Пошук у таблицях з обчислюваними адресами ... 128
Контрольні запитання ... 131
Приклади тестових питань ... 131

10. Типи даних у мові С + + ... 134
10.1. Класифікація та оголошення типів даних ... 134
10.2. Арифметичні типи даних ... 136
10.3. Типи рядків ... 137
10.3.1. Масиви символів ... 137
10.3.2. Тип рядків AnsiString ... 138
10.4. Перелічені типи ... 139
10.5. Множини ... 141
10.6. Вказівники ... 144
10.7. Посилання ... 147
10.8. Масиви в С++ ... 148
10.8.1. Одновимірні масиви в С++ ... 148
10.8.2. Багатовимірні масиви ... 151
10.8.3. Операції з масивами, використання масивів як параметрів функцій ... 151
10.9. Структури ... 154
10.9.1. Структури в мові С ... 154
10.9.2. Самоадресовані структури ... 155
10.9.3. Структури в мові С++ ... 157
10.10. Класи ... 159
10.10.1. Оголошення класу ... 159
10.10.2. Шаблони класів ... 162
Контрольні запитання та вправи ... 164
Приклади тестових питань ... 165

11. Послідовні контейнери в складі стандартної бібліотеки шаблонів ... 168
11.1. Клас vector ... 169
11.2. Клас deque ... 174
11.3. Клас list ... 180
11.3.1. Відмінності між методами списків і методами векторів та двосторонніх черг ... 184
11.4. Клас queue ... 191
11.5. Клас stack ... 194
11.6. Клас BinSearchTree ... 197
11.6.1. Алгоритми обходу двійкового дерева ... 203
11.6.2. Двійкові дерева пошуку ... 207
Контрольні запитання та вправи ... 208
Приклади тестових питань ... 211

12. Асоціативні контейнери у складі стандартної бібліотеки шаблонів 214
12.1. Клас Set 214
12.2. Клас Multiset 216
12.3. Клас Map 216
12.4. Клас Multimap 217
Контрольні запитання та вправи 218
Приклади тестових питань 219

13. Похідні контейнери бібліотеки шаблонів 220
13.1. Клас priority_queue 220
13.2. Кучі 223
13.3. Алгоритм Хаффмана 231
13.4. Збалансовані двійкові дерева пошуку 233
13.5. AVL-дерева 238
13.5.1 Застосування AVL-дерева: програма перевірки орфографії 239
Контрольні запитання та вправи 242
Приклади тестових питань 245

14. Червоно-чорні дерева 247
14.1. Висота червоно-чорного дерева 250
14.2. Клас rb_tree Hewlett-Packard 253
14.3. Метод insert класу rb_tree 256
14.4. Метод erase 262
Контрольні запитання та вправи 273
Приклади тестових питань 274

Предметний покажчик 277

Список літератури 278

1. СТРУКТУРИ ДАНИХ. ОСНОВНІ ВИЗНАЧЕННЯ ТА ПОНЯТТЯ

1.1. Термінологія

Під терміном «**дані**» розуміють інформацію: сукупність фактів, явищ і подій, що становлять інтерес і підлягають реєстрації та обробці, подану у вигляді, який дає змогу автоматизувати процес збирання, зберігання і обробки її на комп'ютері.

Отже, дані відображають і представляють реальний світ. Але, вирішуючи конкретні проблеми, маємо справу не з усім реальним світом, а тільки з деякими його об'єктами. Тобто **об'єктами** називаємо елементи реального світу, інформацію про які ми запам'ятовуємо; сукупність таких об'єктів утворює **предметну область**. Прикладами об'єктів можуть бути люди, що зазначені в будь-якій платіжній відомості; деталі, які виготовляє завод; банківські рахунки тощо.

Очевидно, що об'єкти відрізняються один від одного. Їх необхідно описати характеристиками, які є найважливішими для певної задачі. Такі характеристики називають **атрибутами**. Атрибути має кожний об'єкт. Об'єкти відрізняються один від одного значеннями атрибутів. Значення елемента даних повинно бути пов'язане з конкретним атрибутом конкретного об'єкта.

Під час зображення даних у пам'яті комп'ютера одному елементу даних виділяється певна одиниця пам'яті, яку переважно називають **полем**. Сукупність полів, у яких записано послідовність елементів даних, що розглядаються як одне ціле (наприклад, один рядок накладної), називають **записом**. Сукупність записів утворює **файл**. Під файлом здебільшого розуміють сукупність записів, організованих на зовнішній пам'яті комп'ютера.

Сама сукупність даних, що систематизована певним чином, має ім'я, запам'ятовується в пам'яті комп'ютера більш-менш постійно, її

використовують багато користувачів і вона не залежить від програм користувачів, називається **базою даних** (БД).

Щоб багато користувачів могли модифікувати та використовувати ці дані, необхідне програмне забезпечення – **система управління базами даних** (СУБД). Головне її значення полягає у можливості оперувати даними незалежно від способу їх зберігання.

Банком даних називають систему програмних, мовних, організаційних і технічних засобів, призначених для нагромадження і колективного використання даних.

Отже, між інформаційною моделлю реального світу і даними в комп'ютері існує взаємно однозначна відповідність: одній предметній області відповідає один файл; кількість об'єктів у предметній області дорівнює кількості записів у файлі; кількість атрибутів, що описує об'єкт, дорівнює кількості полів у кожному записі. БД відображає стан об'єктів та їх відношень у певний момент часу у предметній області, що розглядається.

Елементи даних зазвичай поділяють на дві основні групи: скаляри й структури. До скалярів належать прапори, коди, числа й слова; до структур – масиви, таблиці, списки, стеки, множини й записи. Структури формуються зі скалярних даних, згрупованих за певними правилами. Зовнішні до програмної системи дані також формуються зі структурованих елементів.

Тоді як деякі мови програмування забезпечені засобами для формування різних даних, інші дають змогу тільки явно задати певну структуру. Способи опису або зображення складних структур даних, прийняті в людській практиці, часто відрізняються від способів їх опису для використання в комп'ютері. Програмісти настільки звикли до роботи з масивами, що іноді забувають, що структура масиву зазвичай не відображає структури даних, які зберігаються в ньому. Використовуючи масив, мають на увазі, що, по-перше, відома кількість його елементів, по-друге, усі елементи належать тому самому типу і мають той самий розмір і, нарешті, по-третє, доступ до елементів масиву організує відповідно до їхніх положень у масиві. Реальні множини даних рідко бувають однотипними. Порядок використання даних може виявитися несуттєвим або може визначатися значеннями даних. Тобто, можна зробити висновок про те, що мови програмування накладають штучні обмеження на дані.

1.2. Класифікація структур даних

Структури даних – це сукупності різних структурованих типів даних. Останні, своєю чергою, мають власну структуру, яка відображає різноманітні відношення між їхніми компонентами. Основними структурованими типами даних є [2]:

1) масив;
2) декартовий добуток;
3) об'єднання;
4) множина;
5) послідовність;
6) рекурсивний тип.

Особливе місце серед структурованих даних займає тип вказівника або посилання, призначений для забезпечення можливості посилання на інші дані. Компонент такого типу уже має свою структуру.

Відношення, які існують у наведених структурованих типах даних, можуть існувати як серед елементів даних, так і серед їхніх сукупностей.

За аналогією з типами даних сукупність, що складається тільки з елементів даних, називається **простою**, а сукупність, що містить інші сукупності, – **складовою**. Сукупність даних може відповідати об'єкту в конкретних застосуваннях, а її елементи – властивостям цього об'єкта. Глибина вкладень сукупностей у складові може бути як завгодно великою.

Складові сукупності дають змогу будувати **ієрархічні відношення** між її членами, коли вирізняють батьківську і залежну сукупності. Якщо відношення між сукупностями не ієрархічне, кожна залежна сукупність може бути пов'язана з однією або декількома батьківськими сукупностями. Якщо ж відношення ієрархічне, кожна залежна сукуність може бути пов'язана тільки з однією батьківською сукупністю.

Зв'язки між компонентами структур можна задавати явно і неявно. Зокрема структуровані дані типу посилання дають змогу будувати зв'язки між компонентами структур у явному вигляді. Якщо ж структуру побудовано на основі відношень між іншими структурованими типами даних, такі зв'язки між її компонентами задано неявно.

Система обробки даних може передбачати більше як один тип відношення між сукупностями, тобто допускаються відношення, що мають різні правила композиції, або по-різному застосовуються для складання інших структур.

У загальному випадку всі **типи зв'язків** між компонентами структур можна поділити на три: 1) зв'язок типу *1:1*; 2) зв'язок типу *1: N*; 3) зв'язок типу *M:N* (рис. 1.1).

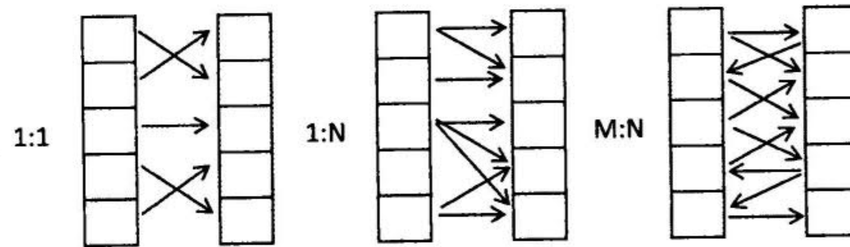


Рис. 1.1. Типи зв'язків у структурах даних

Враховуючи типи даних і зв'язки між ними, поділимо всі структури даних на три класи: найпростіші, лінійні і нелінійні. При цьому розглянемо всі можливі структури, які бувають на абстрактному та конкретному рівнях.

До **найпростіших**, або примітивних структур належать усі типи даних, над якими безпосередньо виконуються машинні операції, тобто арифметичні і рядкові елементи даних. Із рядкових елементів найпростішими є символьні і бітові. Останні є не в усіх системах обробки даних. Наприклад, у мові Фортран бітового рядка немає, а є тільки символьний, а в мові Сі є і бітовий, і символьний.

Лінійні і нелінійні структури – це сукупності структурованих типів даних. **Лінійною структурою даних** назвемо таку сукупність структурованих типів даних, яка відображає відношення сусідства між компонентами. Інші структури назвемо **нелінійними структурами даних.**

Прості сукупності даних, а також ті прості структури, в яких реалізовано зв'язок *1:1*, зараховуємо до лінійних структур даних. Двозв'язні лінійні списки, в яких реалізовано зв'язок *1:2*, як частковий випадок структур із зв'язком *1: N*, також зарахуємо до лінійних структур даних. Інші складні структури, в яких реалізовано зв'язок *1:N* та *M:N* між складовими сукупностями, зарахуємо до нелінійних структур даних.

Отже, класифікацію структур даних можна зобразити діаграмою, де ***S*** означає структури даних (рис. 1.2).

Рис. 1.2. Класифікація структур даних

В усіх типах структур даних найзагальнішими є такі чотири операції.

1. Операція «**створити**» наявна в усіх системах обробки даних, але по-різному реалізована. Наприклад, у мовах Сі, Фортран, Паскаль та інших змінні можна створити за допомогою операторів описання. Пам'ять для змінних у цих мовах виділяється під час виконання або компіляції програми залежно від того, як розподілено пам'ять – динамічно або статично. Є й інші способи створення структур даних. Але важливо те, що незалежно від мови програмування всі структури даних, що є в програмі, не можуть виникнути «із нічого», а явно чи неявно оголошуються операторами створення структур.

2. Операція «**ліквідувати**» не є необхідною, але допомагає ефективно використовувати пам'ять. Деякі мови, наприклад, такі, як Фортран, не дають програмісту можливості ліквідувати створені структури даних. У мові Сі структури даних, що є в середині блоку програми, ліквідовуються упродовж виконання програми під час виходу з цього блоку.

3. Операція «**вибрати**» дає програмісту можливість доступу до даних у середині самої структури. Форма цієї операції значною мірою залежить від типу структури даних, до якої звертаються, і є однією з найважливіших властивостей структур.

4. Операція «**поновити**» дає змогу змінювати дані в структурі. Операція присвоєння є наочним прикладом операції поновлення. Для неї існують і інші складніші форми, наприклад, передача параметрів при вході в блок або підпрограму.

Перелічені операції є загальними для всіх структур даних, але форма їх реалізації суттєво залежить від типу самих структур.

1.3. Документування даних

До документації організації даних належать схеми й діаграми, що характеризують відносини між елементами потоків даних і програмами, між елементами потоків даних і областями зберігання даних і різні рівні структурування даних [1].

Функціональні схеми використовуються для зображення потоків, зовнішніх стосовно програм даних, а ієрархічні схеми модулів – для зображення внутрішніх потоків даних. Відомості про дані, які не локалізовані в якомусь одному модулі, а беруть участь в обміні інформацією між модулями або програмами, подають у формі таблиць потоків даних, словників даних і схем організації даних. Схеми організації даних можуть мати форму графічних схем або графів відносин між даними. Опис усіх рівнів об'єднання даних в агрегати також має відобразитися в документації. Слід зазначити в документації про методи зберігання даних. Якщо в системі є файли, їх опис має містити інформацію про файлові мітки, інакше їх описують так само, як поділювані області пам'яті й параметри.

Інформацію про логічну й фізичну організацію обов'язково вводять до документації. Крім словника даних, до документації мають входити:

- схема й форма організації зовнішніх даних;
- визначення тих компонентів апаратного й програмного забезпечень системи, які є джерелами даних;
- визначення тих частин системи, які звертаються до даних;
- визначення тих частин системи, які змінюють дані;
- опис усіх рівнів організації складних структур даних;
- опис пам'яті, необхідної для розміщення даних, зокрема розмір файла, блокування, мітки тощо.

До загальної системної документації обов'язково має входити документація, у якій міститься інформація, по-перше, про потоки даних, що циркулюють між функціональними блоками й областями зберігання даних, по-друге, про дані, якими обмінюються програми й області зберігання (цю інформацію також зручно подавати у вигляді словника даних), і, по-третє, про використовувані ресурси. Програмна документація також міститиме інформацію, що стосується документації даних, а саме опис потоків даних між програмними модулями. Документація модулів, своєю чергою, міститиме аналогічну інформацію, пов'язану з елементами даних, визначеними усередині модулів.

Контрольні запитання та вправи

1. Що таке об'єкти предметної області та їхні атрибути?
2. Що таке дані?
3. Які існують інформаційні одиниці даних?
4. Які є структуровані типи даних?
5. Які бувають сукупності даних?
6. Які є типи зв'язків між компонентами структур?
7. Що передбачає ієрархічне відношення між сукупностями?
8. Які є класи структур даних?
9. Що належить до найпростіших структур даних?
10. Який тип зв'язку визначає клас лінійних структур даних?
11. Який тип зв'язку визначає клас нелінійних структур даних?
12. Назвіть основні операції над структурами даних.

Приклади тестових питань

1. Який тип зв'язку визначає клас лінійних структур даних?
   а) 1:1;
   б) 1:N;
   в) N:M.

2. Які є класи структур даних?
   а) найпростіші, лінійні і нелінійні;
   б) лінійні і нелінійні;
   в) найпростіші і лінійні.

3. Який тип зв'язку визначає клас нелінійних структур даних?
   а) 1:1;
   б) 1:N;
   в) N:M.

4. Який тип структур з наведених належить до нелінійних?
   а) рядки;
   б) масиви;

в) таблиці;
г) стеки;
д) списки.

5. Який тип структур з наведених належить до лінійних?
а) таблиці;
б) дерева;
в) сітки;
д) стеки, черги, деки.

6. Що таке об'єкти предметної області?
а) елементи реального світу;
б) змінні та функції класу;
в) змінні типу «клас».

7. Що таке дані?
а) сукупність фактів, явищ і подій, що становляють інтерес і підлягають реєстрації та обробці;
б) набір чисел;
в) літери алфавіту.

8. Які є типи зв'язків між компонентами структур?
а) 1:10;
б) 2: N;
в) 1:1, 1: N, M:N.

9. Які із зазначених є структурованими типами даних?
а) структури;
б) масив;
в) декартовий добуток;
г) таблиці;
д) об'єднання;
е) множина;
ж) послідовність;
з) записи;
к) рекурсивний тип;
л) посилання.

10. Які з перелічених є найпростішими структурами даних?
а) арифметичні і рядкові;

б) символьні і бітові;
в) цілі і дійсні.

11. Які бувають сукупності даних?
а) однотипні;
б) різнотипні;
в) прості;
г) складові.

12. Що передбачає ієрархічне відношення між сукупностями?
а) кожна залежна сукупність може бути пов'язана тільки з однією батьківською сукупністю;
б) кожна залежна сукупність може бути пов'язана з багатьма батьківськими сукупностями;
в) сукупності пов'язані між собою в межах одного рівня.

13. Яку структуру утворює сукупність числових елементів з плаваючою крапкою?
а) лінійну;
б) нелінійну.

14. Яку структуру утворює сукупність символьно-цифрових елементів?
а) лінійну;
б) нелінійну.

15. Яка операція не належать до операцій над структурами даних?
а) створити;
б) поновити;
в) ліквідувати;
г) копіювати;
д) вибрати.

2. РІВНІ ОРГАНІЗАЦІЇ ДАНИХ

Програміст, проектувальник і користувач мають свої погляди на організацію даних. Відповідно до цього можна виділити три рівні організації даних:

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

Логічна організація даних – це, по суті, представлення даних з боку користувача. Індивідуальні дані повинні бути згруповані відповідно до їхніх зв'язків, а також до зв'язків, що відображають способи їх використання. В основу логічної організації даних покладено вимоги користувача й внутрішньо властиві даним зв'язки. Це найважливіший рівень абстракції, використовуваний у представленні даних, оскільки саме вимоги користувачів визначають вигляд проектованої системи. Якщо на етапі проектування системи вдало обрано логічну організацію даних, зміни системних вимог, що не приводять до модифікації логічної структури даних, не спричинять реорганізації на нижчих рівнях представлення даних. Тільки на логічному рівні можуть застосовуватися формальні методи опису динамічнозмінюваних 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.

Інформацію про представлення даних можна поділити за окремими програмними модулями, причому можна використовувати як зовнішню, так і внутрішню форми представлення даних. **Зовнішнє**

представлення – це 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 містять номер рахунка, ім'я, адресу, дати попередніх витрат, платежі й рахунки. 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 зображаються 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 вибрати серед доволі вузького кола способів представлення даних, які мають більш-менш зрозумілу фізичну організацію. Агрегати даних, що зберігаються у швидкодіючій пам'яті, можна подати масивами або стеками. До записів файла можна звертатися послідовно або у довільній послідовності, використовуючи ключі різних типів.

2.1. Логічна організація даних

Існують поняття логічної та фізичної незалежності даних: перша означає, що загальну логічну структуру можна змінити без зміни програм; друга – що фізичне розміщення і організація даних можуть змінюватись без зміни загальної логіки структур даних прикладних програм [1].

Проблема проектування структури даних полягає, по-перше, у визначенні способу їх логічної організації й, по-друге, у пошуку шляху вираження цієї організації на рівні зовнішніх представлень, що допускаються системою й мовою програмування. Системне програмне забезпечення управляє потоком даних між програмами й зовнішніми пристроями, а мови програмування управляють обміном даними між програмами.

Ввід і вивід є нестандартними операціями, і тому системний аналітик і програміст повинні не тільки добре знати можливості мови програмування, але й розбиратися в роботі системного програмного забезпечення. Проектування й реалізацію потрібно провести так, щоб зміна вимог користувача, а також модифікація апаратного й програмного забезпечення якнайменше впливали на проектовану програмну систему. Цей вплив можна звести до мінімуму, ізолюючи операції системного інтерфейсу й вводу-виводу.

Системне програмне забезпечення дає змогу ізолювати реальні процедури вводу-виводу від програміста. Тобто, програми в принципі не змінюються при зміні зовнішніх пристроїв. Програміст, своєю чергою, повинен захистити користувача від змін у системному програмному забезпеченні. Користувач не повинен також піклуватися про структури й розміри файлів. Аналогічно в прикладних системах не повинні використовуватися внутрішні представлення даних поза окремими модулями. Програми доступу, які слугують для розміщення, пошуку й зміни даних, повинні забезпечувати контакт із даними тільки на рівні зовнішнього представлення. Це означає, що процес переходу від рівня логічної організації до рівня представлень має передбачати проектування не тільки структур даних, але й програмних модулів. Причому при проектуванні структур даних використовується принцип ізоляції інформації.

Програма користувача не повинна бути пов'язана з методом зберігання даних. Коли для зберігання інформації використовується база даних, СУБД вибирає шлях доступу відповідно до початкового визначення бази даних. Здійснюючи запит, необхідно дотримуватися правил, установлених для реалізації бази даних. Якщо інформація міститься не в базі даних, а у файловій системі, то для того, щоб можна було використати нові форми запитів, необхідно або змінити існуючі програми, або створити нові. Організація нових шляхів доступу, відмінних від передбачених при початковій організації, може потребувати додаткової обробки даних.

Ретельне визначення шляхів доступу і явний опис структур даних тільки в деяких модулях слід робити принаймні із двох причин. По-перше, користувач не повинен мати справи з не стосовними до його задачі питаннями зберігання даних. По-друге, якнайменше модулів повинні залежати від організації зберігання даних. Якщо організація даних ізольована, зміна їх представлення спричинить незначні зміни тексту програми. Тому необхідно намагатися розміщати структуру даних тільки в одному програмному модулі.

Наявність постійних файлів приводить до додаткових проблем. У системах баз даних постійні файли належать до внутрішніх структур, і доступи до них здійснюються тільки за допомогою адміністратора бази даних, тобто структура постійних файлів у базах даних ізольована. В інших системах не можна обмежитися дозволом доступу до них тільки одній програмі. Найкращим виходом є розміщення опису файла в невеликій кількості програмних модулів. Це дає змогу мінімізувати можливі зміни програм. Якщо в кожній програмі, яка працює з файлом, доступ до нього забезпечує один модуль, можна рекомендувати зберігати опис файла в бібліотеці й у всіх програмах використовувати для доступу той самий модуль або принаймні той самий опис файла.

Дані в комп'ютері можуть бути впорядковані за допомогою ключів, які є елементами даних. Для організації даних можна використовувати рівні ієрархії або багатоаспектний доступ і перехресні посилання. Для опису інформації в базах даних найчастіше використовуються ієрархічні або реляційні моделі, причому реляційна модель ближча до логічної структури даних.

За допомогою трьох процедур нормалізації можна привести будь-які дані до єдиної форми. Перша нормальна форма (1НФ) дає змогу описати дані без повторюваних груп. Це здійснюється за допомогою виділення частини структури, яка повторюється в іншій структурі. Процес перетворення до першої нормальної форми зводиться до задачі побудови двох типів структур, пов'язаних перехресними посиланнями.

Структура даних належить до другої нормальної форми (2НФ), якщо вона, по-перше, належить до першої нормальної форми й, по-друге, не ключові елементи функціонально повно залежать від ключа. Це означає, що ключ потрібен для визначення інших елементів, і якщо ключ є складовим, жоден з інших елементів не може бути визначений на підставі знання тільки частини ключа. Приведення структури до 2НФ здійснюється видаленням елементів даних, які залежать тільки від частини ключа, і формуванням з них та з часткових ключів окремої структури.

Структура даних належить до третьої нормальної форми (3НФ), якщо не існує неключових елементів, які функціонально залежать від інших неключових елементів. Якщо ж такий елемент існує, він безпосередньо пов'язаний із ключем. Переведення з 2НФ у 3НФ здійснюється видаленням функціонально залежних елементів і формуванням з них та з елементів, які їх визначають, окремої структури.

Внутрішнє представлення даних

Представлення даних у програмах залежить від застосовуваної мови програмування. Найчастіше використовуються такі типи даних, як скаляри, одновимірні масиви й записи. У більшості мов програмування передбачено можливість роботи з усіма цими типами даних.

Часто найвідповіднішими є структури даних типу множин, зв'язаних списків та таблиць. Множина являє собою сукупність елементів. На відміну від масиву або послідовного файла елементи множини не впорядковано. Необхідно так представити дані, щоб можна було завжди швидко отримувати потрібну інформацію. Через те, що мови програмування дають змогу працювати з різними типами даних, завданням програміста є вибір такої структури даних, яка є найзручнішою для представлення інформації.

Для вводу-виводу скалярні елементи даних слід розглядати або як незалежні елементи, або як елементи записів. Залежно від можливостей мови програмування дані можуть бути або рядками символів, або десятковими або двійковими числами; числа, своєю чергою, можуть бути представлені у формі з фіксованою або плаваючою крапкою. Конкретна форма представлення даних та їхні розміри залежать від конкретного комп'ютера.

У програмі або в області розміщення дані зазвичай об'єднано в масиви, структури або складніші агрегати, яких часто безпосередньо не передбачено в мовах програмування. Існуючі механізми доступу, що працюють зі структурами й масивами, ізольовані від програміста. Якщо розробляються нові складні типи даних, що розширюють можливості мови, механізми доступу до них також слід ізолювати.

Переважно для реалізації складних типів даних можна використовувати кілька способів. Наприклад, стек можна реалізувати у вигляді масиву змінної довжини, що знаходиться усередині масиву з фіксованими межами, або у вигляді зв'язаного списку, керованого вказівниками. Для того, щоб спростити внесення змін до даних таких типів, їх слід помістити в окремі програмні модулі. Якщо для внутрішнього представлення даних зручно використовувати стек або множину, програміст може створити спеціальне розширення мови, що містить тип даних, причому звертатися до нього можна лише викликом підпрограм. Інші модулі програмної системи не залежать від способу реалізації цього нового типу даних.

На рис. 2.1 наведено приклади внутрішнього представлення неорієнтованого графу.

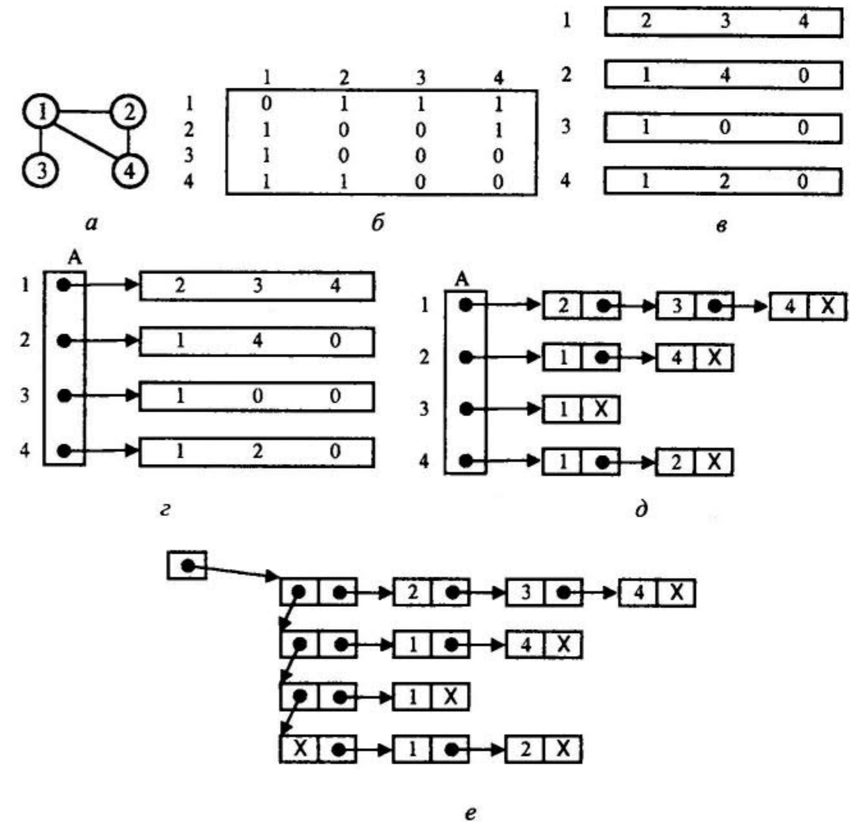


Рис. 2.1. Внутрішнє представлення неорієнтованого графу: а – неорієнтований граф; б – матриця суміжності; в – масив статичних векторів; г – масив динамічних векторів фіксованого розміру; д – масив списків; е – список списків

Дані, що є зовнішніми стосовно системи (наприклад, вхідні й вихідні дані), повинні бути подані у формі, зручній насамперед для сприйняття людиною. Як вхідні, так і вихідні дані повинні легко читатись, не потребуючи для цього спеціальних знань про формати, коди й представлення даних. Для проектування цієї частини зовнішнього представлення даних важливі як загальна організація даних, так і конкретні деталі представлення. Організація залежить від системних специфікацій і вимог користувачів. Деталі залежать від мови реалізації, але насамперед визначаються стилем і можливостями адаптації до умов, що виникають упродовж використання системи.

Формат вводу числових даних має містити десяткову крапку й пробіли ліворуч. Виведені числові дані повинні бути відредаговані й позначені спеціальними мітками так, щоб можна було легко зрозуміти, чи вони є грошовими одиницями чи результатами вимірювань фізичних величин. Припустимо, що необхідно використовувати коди. Тоді мнемонічні скорочення переважають числові коди (M и F краще, ніж 1 і 2). Якщо потрібно застосувати абревіатуру, слід віддавати перевагу стандартним скороченням (M, T, W, TH, F, SA, SU краще, ніж M, T, W, H, F, S, U). Якщо застосовується нестандартна абревіатура або код, користувачеві часто буде потрібна підказка, що пояснює зміст повідомлення. Відповідні форми повинні бути передбачені для представлення як вхідних, так і вихідних даних.

Вхідні дані слід уводити відповідно до організації, під час введення використовувати роздільники, таблиці вводити порядково. Межі рядків повинні припадати на інтервали між даними. Помилки введення, що полягають у пропуску символу або появі зайвого символу, можна легко виявити ще до введення даних до комп'ютера, якщо дані подано у формі, зручній для сприйняття. Системи введення даних, що працюють у режимі on-line, видають користувачу інформацію, що вводиться, даючи йому в такий спосіб можливість виправити допущені помилки ще до передавання даних до комп'ютера. Вхідні дані повинні бути зручні для сприйняття.

Вихідні дані зазвичай виводять у формі, яка в економіці одержала назву «звіти», або у формі, яка в діалогових системах називається відповідями на запити. Яку б кількість даних не виводили, їх треба розмістити відповідно до заданих форматів, назвати, датувати й забезпечити спеціальними мітками. Числові дані редагують й позначають спеціальними мітками. Нечислові повідомлення слід виводити разом з поясненнями, що стосуються використовуваних у них кодів і скорочень. Кожну сторінку нумерують, називають і датують. На результати, що, можливо, містять помилку, звертають увагу користувача за допомогою різноманітних прапорців.

Звіти зазвичай ділять на сторінки з полями керування елементами даних для їх групування. Зміна значення цього елемента приводить до тимчасового переривання процесу обробки, що називається керівним перериванням.

Зовнішнє представлення даних можна описати або зобразити за допомогою графічної схеми – так званої «ієрархічною схемою даних» (рис. 2.2). Введення нового керівного поля приводить до появи ще одного рівня ієрархії. Кожний рівень схеми пов'язаний з однією керівною групою.

Рис. 2.2. Ієрархічна схема звіту (логічна структура)

На рис. 2.3 показано фізичну організацію даних, що складають звіт. У ній враховано такі характеристики друкарського пристрою, як довжина рядка й розмір сторінки, і пов'язана зі змістом звіту лише в тому випадку, якщо розподіл за сторінками залежить від керівних переривань. Крім того, ширина смуг і розташування інформації залежать від характеристик як звіту, так і друкарського пристрою. Заголовки й підсумки, що належать до сторінок, повинні містити номери сторінок, дату, ідентифікатор звіту, легенди й, можливо, заголовки стовпців і підсумкові результати. Ієрархічні схеми, наведені на рис. 2.2 і 2.3, призначені для опису того самого звіту, але знаходяться у такому відношенні один до одного, що безпосередньо зіставити їх не можна. Діаграма логічних відносин між даними, графічна схема зовнішнього представлення даних і графічна схема фізичної організації є трьома спорсобами представлення даних, що належать до трьох різних точок зору на дані.

Рис. 2.3. Ієрархічна схема множини сторінок, що складають звіт (фізична структура)

2.3. Фізична організація даних

На фізичному рівні дані можуть бути впорядкованими або неупорядкованими, однорідними або неоднорідними, містити ключі або не містити. Результати фізичних вимірювань вважаються впорядкованими, якщо вони робляться в строго визначені проміжки часу і якщо час є однією з використовуваних для аналізу змінних, яка явно не входить до складу даних. Дані можна впорядкувати й за допомогою інших вимірюваних величин – таких, як відстань. Дані вважаються неупорядкованими, якщо та величина, за значеннями якої вони групуються, не є хоча б неявно частиною даних. Упорядкованість передбачає неявне завдання, достатнє для порівняння значень змінної; цей захід, якщо потрібно, може бути явним, тобто на логічному рівні дані будуть неупорядкованими, якщо порядок уводиться тільки з міркувань зручності доступу до окремих елементів даних і підрахунку загальної кількості даних. Найадекватнішою формою представлення неупорядкованих даних є множина. Зручною формою представлення впорядкованих даних може бути черга, стек, список або масив [1].

Результати фізичних вимірювань зазвичай неоднорідні, оскільки або одні величини вимірюють у декілька разів частіше, ніж інші, або одиниці вимірювань різні, або множина даних містить елементи різних типів. Записи економічної інформації також неоднорідні, оскільки вони складаються із символьних і числових полів різної довжини. Внутрішні й зовнішні форми представлення однорідних і неоднорідних даних можуть бути тими самими, якщо мовою програмування передбачено або динамічний розподіл пам'яті, або те, що пам'яттю може управляти програміст. На рівні фізичної організації сукупність даних повинна бути однорідною.

Якщо ніколи не буде потрібно ідентифікувати окремі елементи даних тільки на підставі їх порядку, вважають, що дані на логічному рівні не містять ключів. У протилежному випадку можна вважати, що дані містять ключі. Характеристика, використовувана для ідентифікації (ключ), є полем даних. Іноді доцільно мати кілька ключів для того, щоб забезпечити багатоаспектний доступ. Ключі можуть бути основою вибору внутрішнього представлення даних. Від них може залежати також фізична організація даних.

Одні множини даних є структурованими, інші неструктурованими. Якщо множина даних не має характерної структури, його елементи все-таки можна згрупувати для зручності роботи з ними, наприклад, коли блокуються записи або кілька коригувальних повідомлень уводяться разом з термінального пристрою. Якщо елементи даних пов'язані один з одним, але належать різним типам, їх можна об'єднати або в таблиці, або в структури (як у випадку з інформацією про передплатників). Такі агрегати даних, своєю чергою, поєднуються в ще більші агрегати. Файл може складатися з рядків табличної інформації або з набору структурованих записів. Аналіз потоків даних приводить як до декомпозиції потоків даних на повторювані компоненти, що складаються із окремих елементів даних, так і до об'єднання елементів даних в агрегати з метою формування на їхній основі структурованих областей зберігання даних.

Області зберігання даних містять одночасно кілька екземплярів різних агрегатів даних. Хоча агрегати даних можуть бути логічно неоднорідними, обмеження, що накладаються мовами програмування й пристроями вводу – виводу, вимагають, щоб області зберігання даних були однорідними на рівні агрегатів. У загальному випадку дані повинні бути впорядковані за допомогою або ключів, або штучно введених правил. Вибір фізичної організації даних дає змогу визначити, чи є дані впорядкованими та чи потрібні для їхньої організації ключі.

Упорядкованість / невпорядкованість

Неупорядковані дані формально можна описати за допомогою множини, у якій існує правило, що дає змогу визначити, належить елемент множині чи ні, і системи доступу, яка організує доступ до потрібного елемента. Елементи множини можуть або мати ключі, або самі бути ключами. Якщо з елементами множини необхідно виконувати певні операції, потрібна система доступу, яка переглядає елементи множини послідовно або паралельно без урахування якого-небудь попередньо визначеного порядку. Через те, що до елементів множини необхідно постійно звертатися, множини зазвичай реалізуються у вигляді одновимірних масивів або у вигляді файлів з послідовним доступом. Прямий доступ здійснюється за допомогою вибору структури файла, що припускає як прямий, так і послідовний доступ.

Файли й масиви можуть складатися з упорядкованих і неупорядкованих даних. Якщо елементи множини не містять ключів, їх можна зберігати в масиві або в послідовно організованому файлі. Якщо ключі передбачені, саме вони є основою організації системи зберігання. Дані, які необхідно впорядкувати декількома способами, використовуючи різні ключі, можуть зберігатися в базі даних, у багатовимірному масиві, у файлі із пов'язаними за допомогою ключів елементами або в системі інвертованих файлів.

Однорідність / неоднорідність

Неоднорідні елементи даних відрізняються один від одного за типом, організацією чи довжиною або складаються із частин, що відрізняються за типом, організацією чи довжиною. На фізичному рівні з неоднорідними даними можна працювати, якщо стандартизувати різні частини даних, тобто необхідно уніфікувати типи, розміри й усе те, що пов'язане з представленням даних у комп'ютері. Це можна зробити введенням єдиного для всіх даних формату, що містить порожні поля; стандартизації довжини й використання самозумовлених даних; об'єднання елементів різної довжини в рядок постійної довжини й використання декількох описів. Наприклад, для опрацювання багатоцифрових чисел за домовленістю відсутні розряди заміняються нулями (рис. 2.4).

Вхідний масив чисел	Стандартизований масив
32	0032
5714	5714
5	0005
4325	4325
678	0678

Рис. 2.4. Приведення даних до стандарту

Наявність ключів / відсутність ключів

Усі збережені дані повинні бути доступні. Елементи даних, що не мають ключів, повинні розміщуватися послідовно в масиві або в послідовному файлі, щоб до них можна було легко одержати доступ. Дані, що містять ключі, залежно від відносної частоти послідовних і випадкових звертань до них можуть розміщатися або послідовно один за одним, або за деяким методом хешування. Якщо дані розміщуються один за одним, вони можуть бути або відсортовані (по-перше, коли впорядковані, по-друге, коли доступ до них здійснюється за допомогою індивідуальних ключів), або не відсортовані. Якщо порядок не має значення, сортувати дані не обов'язково. Якщо елементи даних розміщено за певним методом хешування й, крім того, можливий випадок послідовного звертання до них, необхідно доповнити спеціальними кодами невикористовувані проміжки масиву чи файла або пов'язати інформаційні області пам'яті, запроваджуючи так впорядкованість.

Для визначення позиції елемента даних у масиві в алгоритмах хешування часто використовуються або визначені цифри, що входять

до ключа, або залишок від ділення ключа на певне число. Якщо хешування для двох ключів дає однакове значення, конфлікт можна зняти, розмістивши друге значення в пам'яті, що була виділена для першого значення. У табл. 2.1 наведено приклад визначення адрес для заданих ключів як остача від ділення значення ключа на число 10.

Таблиця 2.1

Побудова хеш-таблиці

Ключ	Адреса
10	0
15	5
23	3
37	7

Фізична організація зовнішніх даних залежить від типу використовуваних пристроїв вводу-виводу. У режимі безпосереднього доступу до системи інформація, призначена для вводу або виводу, являє собою потік даних. Кожне повідомлення, що адресоване користувачу і яке зчитується з екрана монітора, являє собою одиничний елемент даних. Останній може бути або простим елементом, або цілою структурою даних, або копією екрана монітора. Для зберігання даних у моніторі використовується буфер. Він не має засобів структуризації даних і зберігає структуру даних. Спосіб буферизації, наприклад, використання подвійної буферизації або циркулярних буферів, ізолюється системою від користувачів.

Для розв'язання більшості економічних задач доводиться мати справу з файлами. Інформація про службовців, клієнтів, товари, а також різні повідомлення зазвичай зберігаються у файлах. Крім файлів для зберігання вихідних даних і проміжних результатів в економічних додатках використовуються допоміжні файли, які по суті є послідовними й зберігаються на диску. Місце зберігання визначається вимогами, що існують у конкретній організації. Немає потреби зберігати записи про клієнтів, які більше не мають справ із фірмою. Але в медичних установах дані про пацієнтів повинні зберігатися протягом декількох років. У навчальних організаціях також необхідно достатньо довго зберігати особисті справи студентів, що вже закінчили навчання.

Файли, призначені для реєстрації змін, повинні містити інформацію про всі ті модифікації головного файла, які проводилися.

Для кожного запису, який було змінено, необхідно зберігати дані про його початковий стан, зміни й кінцевий стан. Файли, у яких реєструються зміни, крім того, можна використовувати разом з контрольними крапками для повторного запуску програми корегування файлів, виконання якої було перервано через системний збій.

В областях, не пов'язаних з економікою, для зберігання даних також використовуються файли. Щоразу, коли зібрано велику кількість даних, їх треба десь зберігати. Файли потрібні для розміщення в них експериментальних даних, що їх одержують учені й інженери, статистичних даних, що їх збирають соціологи, і текстових даних, які досліджують фахівці гуманітарних наук. Файли можна використовувати просто для зберігання зібраної інформації доти, поки вона необхідна; їх можна використовувати для зберігання даних, які обробляються комп'ютером, але яких занадто багато для того, щоб тримати їх в оперативній пам'яті. Крім того, у файлах можна зберігати дані, до яких необхідно періодично звертатися.

Операційні системи використовують файли для тимчасового зберігання вхідних і вихідних даних. Спеціальні файли призначені для розміщення компіляторів, асемблерів і завантажників. Програми користувачів можуть зберігатися у файлах або тимчасово, або постійно. Файли обліку й реєстрації містять інформацію про роботу комп'ютера. Для особливо важливих пакетів дисків створюються файли-дублікати.

Для розв'язання на комп'ютері практично всіх задач тією чи іншою мірою необхідні файли. З файлами, як і з усіма іншими структурами даних, можна виконувати різні операції. Файли можна створювати, ініціювати, модифікувати, переводити в пасивний стан і знищувати. Файл модифікується, якщо до нього додається нова інформація, або частина його інформації знищується, або деяка інформація, що міститься в ньому, змінюється. Як інформацію можна використовувати сукупність символів або рядків, повідомлення, записи й сторінки тексту. У багатьох системах дані, що групуються в блоки фіксованої довжини, називаються фізичними записами. Ці блоки можуть складатися з одного або декількох логічних записів. Залежно від можливостей операційної системи записи й файли можуть мати змінну довжину (і тоді необхідна пам'ять буде виділятися динамічно) або їхню довжину обмежено заздалегідь визначеною величиною.

У програмах, що використовують потокоорієнтований ввід-вивід, перетворення із символьної і на символьну форму здійснюються автоматично. Програміст керує вводом–виводом за допомогою форматів і оголошень.

Для даних, що безпосередньо вводяться з клавіатури, можна використовувати як потокоорієнтоване передавання, так і передавання блоками. У першому випадку за один раз передається тільки один символ і їх групують засобами операційної системи тільки в пункті призначення. Із символів формуються зображення на екрані монітора або вони групуються так, щоб зручніше було зберігати їх у масовій пам'яті.

У другому випадку символи нагромаджуються в спеціальному буфері й передаються цілими групами. Керують передаванням за допомогою ключа кінець-рядка, кінець-повідомлення або кінець-сторінки, починаючи з узгодження перших символів рядка, повідомлення або сторінки. Незалежно від того, чи складаються файли з інтерактивних блоків або із записів, операції вводу-виводу дають змогу створювати, ініціювати, вибирати, модифікувати, переводити в пасивний стан, зберігати й знищувати файли. Текстові редактори, призначені для роботи в режимі безпосереднього доступу, дають змогу маніпулювати символами, рядками, повідомленнями або сторінками. Системи керування файлами й масовою пам'яттю орієнтовані на роботу з даними, поділеними на окремі записи.

Два методи доступу до даних використовуються у разі потокоорієнтованого передавання й передавання блоками. Якщо символи, рядки, сторінки або записи файла обробляються поодинці, один за одним, використовується послідовний доступ. Якщо необхідні тільки певні частини файла, використовується прямий доступ. Прямий доступ застосовується також у системах редагування текстів, призначених для роботи в інтерактивному режимі, коли редактор визначає місцерозташування частини тексту або рядка, що починається із заданої послідовності символів. У системах, що використовують масову пам'ять, може запитуватися будь-який запис, що має ключ, причому ключ може бути частиною запису.

Процедури, які здійснюють доступ до інформації, що міститься у файлах, використовують такі елементарні операції:

- визначення початку поточного елемента даних;
- визначення позиції наступного елемента;
- визначення позиції заданого елемента;
- зчитування елемента із заздалегідь визначеної позиції файла;
- запис елемента у заздалегідь визначену позицію файла;
- перевірку доступності елемента.

Визначення позиції наступного елемента можливе тільки для файла, який містить упорядковану за якоюсь ознакою інформацію. Ця операція є основою як послідовного, так і потокоорієнтованого вводу-виводу. Перевірка доступності елемента пов'язана з можливістю визначення границь файла й з можливістю розпізнання елемента за його позицією, вмістом визначеного поля або за ключем. Визначити місцезнаходження заданого елемента можна, якщо інформація у файлі має ключі й можна автоматично шукати потрібну послідовність символів. Ключі фізичних файлів не обов'язково повинні бути логічними ключами. Дані впорядковуються й забезпечуються ключами для зручності організації доступу до файла навіть тоді, коли вони логічно не впорядковані й не мають ключів.

В операційних системах найчастіше використовують послідовну, основану на відносній нумерації, й індексну організації файлів. Існують також різні варіанти цих типів організацій. Крім того, різні способи опису структури файлів і різні методи доступу до них поширені в різних мовах програмування й у різних операційних системах.

Контрольні запитання та вправи

1. Назвіть рівні організації даних.
2. Які є форми представлення даних?
3. На якому етапі формується логічна організація даних?
4. Що являє собою фізичний рівень організації даних?
5. Які дані називають впорядкованими?
6. Які елементи даних називають неоднорідними?
7. Чим забезпечується доступність даних?
8. Наведіть приклад фізичної організації зовнішніх даних.
9. Назвіть основні операції доступу до інформації.
10. Назвіть основні способи організації файлів.
11. Визначте логічну організацію даних для системи каталогізації книг у бібліотеці.
12. Визначте логічну організацію даних для системи реєстрації заявок на користування тенісними кортами.

Приклади тестових питань

1. Яка операція є основною для послідовного вводу-виводу ?
   а) визначення початку поточного елемента даних;
   б) визначення позиції наступного елемента;
   в) визначення позиції заданого елемента.
2. Для визначення границь файла важливим є:
   а) визначення позиції наступного елемента;
   б) перевірка доступності елемента;
   в) визначення позиції заданого елемента.

3. Який метод доступу до даних використовують у системах редагування текстів, призначених для роботи в інтерактивному режимі?
   а) прямий;
   б) послідовний.

4. У якому типі вводу-виводу інформація перетворюється на символьну форму автоматично?
   а) ввід-вивід потоком;
   б) ввід-вивід блоками.

5. Чим відрізняються неоднорідні елементи даних один від одного?
   а) типом;
   б) організацією;
   в) значеннями ключів;
   г) довжиною.

6. На якому рівні розглядається логічна організація даних?
   а) проектний рівень;
   б) рівень мови реалізації;
   в) машинний рівень.

7. На якому рівні розглядається фізична організація даних?
   а) проектний рівень;
   б) рівень мови реалізації;
   в) машинний рівень.

8. На якому рівні розглядається представлення даних?
   а) проектний рівень;
   б) рівень мови реалізації;
   в) машинний рівень.

3. ЛІНІЙНІ СТРУКТУРИ ДАНИХ. СТЕК, ЧЕРГА, ДЕК

Стек, черга і дек – це однотипні лінійні структури. Їх називають динамічними лінійними структурами даних у зв'язку з тим, що вибірка елемента із цих послідовностей зводиться до операції вилучення, тобто ліквідації його у послідовностях. Відрізняються ці структури одна від однієї порядком виконання операцій введення і вилучення елементів.

До основних операцій над цими структурами належать [2]:

а) створення нової структури даних;

б) запис або введення елемента до вже наявної структури даних;

в) вилучення елемента зі структури;

г) перевірка умови існування структури.

3.1. Стек

Стек – це впорядкована лінійна динамічно змінювана послідовність елементів, у якій виконуються такі умови:

1) новий елемент приєднується завжди до того самого боку послідовності;

2) доступ до елемента здійснюється завжди з того боку послідовності, до якого приєднується елемент;

3) елемент зберігається в послідовності до моменту його виклику [2].

Прикладом стеку є стос тарілок або магазин патронів у рушниці. За аналогією з останнім прикладом стек ще часто називають магазинною пам'яттю.

Операцію введення елемента до стека називають «проштовхуванням», а операцію вилучення із послідовності – «виштовхуванням». Найбільш і найменш доступні елементи стека навивають відповідно верхом і низом стека. Операції введення і вилучення зі стека

виконують в одному і тому ж боку послідовності, але вилучаються елементи зі стека у послідовності, зворотній до тієї, в якій вони потрапили до послідовності. У кожний момент часу зі стека можна забрати лише один елемент. Дисципліну обслуговування стека часто називають дисципліною ***LIFO***: «останній прийшов, перший пішов».

Стек можна зобразити у вигляді послідовності $A=A_1,...,A_{n-1},A_n$, де A_n – елемент, що вказує на вхід–вихід у послідовність; A_1, A_n – відповідно низ і верх стека. Щоб прочитати елемент A_j, необхідно вибрати зі стека елементи $A_n,..., A_{j+1}$.

Також стек можна подати у вигляді трубки з підпруженим дном, розміщеної вертикально (рис. 3.1). Верхній кінець трубки відкритий, до нього можна вводити, або заштовхувати елементи. Новий елемент, що додається, проштовхує елементи, введені до стека раніше, на одну позицію донизу. У разі вилучення елементів зі стека вони виштовхуються нагору.

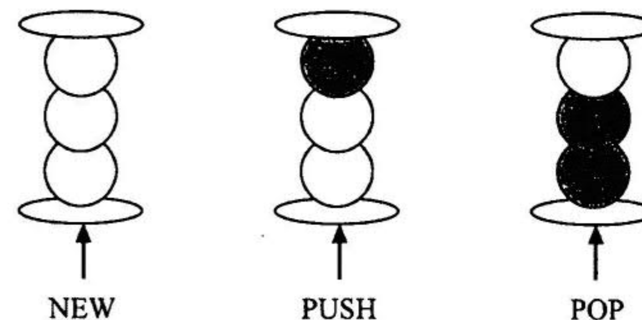


Рис. 3.1. Операції роботи зі стеком

Незалежно від того, як реалізовано стек, процедури, що працюють із ним, повинні вміти вміщувати елементи в стек, вибирати елементи зі стеку й перевіряти, чи не порожній стек. Для того, щоб можна було працювати з новим типом даних, необхідні спеціальні операції, наприклад:

- **NEW** (стек) – створення порожнього стека
- **PUSH** (стек, елемент даних) – розміщення елемента в стек
- **POP** (стек) – вибірка елемента з вершини стека
- **EMPTY** (стек) – перевірка, чи порожній стек; результатом є значення «істина», якщо стек порожній, і «хибно» у протилежному випадку.

NEW і PUSH – базові операції, використовувані під час роботи зі стеком. Перша створює стек, а друга вміщує в нього елементи. POP – процедура, характерна тільки для роботи зі стеком. Її можна визначити за допомогою двох базових проміжних операцій TOP і REMOVE, які дають змогу виділити дві основні функції, реалізовані POP, але не визначають способів організації доступу до структури з інших модулів.

Відповідно до наведених вище визначень, POP вибирає вершину стека й видаляє її зі стека. Вершина стека описується так: якщо стек не порожній, вершина являє собою останній вміщений до стека елемент; а якщо ні, то стек не має вершини. Операція REMOVE дає в результаті стек без елемента, розміщеного в нього останнім. До тільки-но створеного стека операцію REMOVE застосовувати не можна. Для визначення порожнього стека використовується поняття нового (або тільки-но створеного) стека: порожній стек не містить жодного елемента.

Будь-яка реалізація стека повинна задовольняти ці визначення. Важливо, що стек визначається як спосіб зберігання даних, які підпорядковуються певним правилам, що діють при звертанні до даних, а не як масив зі вказівниками, які пересуваються за певними правилами.

Основною перевагою стека перед іншими організаціями даних є те, що у ньому не потрібна адресація елементів [10]. Для обслуговування стека потрібні лише дві команди: PUSH – «проштовхнути» і POP – «виштовхнути». Зі стеком пов'язаний завжди один вказівник, який вказує на верхній елемент у стеку. На початку такий вказівник дорівнює нулю.

Найпростішим прикладом вдалого застосування стека може бути задача передавання вектора у зворотній послідовності. На практиці стекова структура даних найчастіше застосовується в рекурсивних алгоритмах, під час трансляції, а також обробки переривань програм.

Стек застосовується у різних ситуаціях. Поєднує їх мета: потрібно зберегти деяку роботу, яку ще не виконано, за необхідності перемикання на інше завдання. Стек використовується для тимчасового збереження стану не виконаного до кінця завдання. Після збереження стану комп'ютер перемикається на інше завдання. Після його виконання стан відкладеного завдання відновлюється зі стека, і комп'ютер продовжує перервану роботу.

Припустімо, що комп'ютер виконує завдання *A*. У процесі його виконання виникає необхідність виконати завдання *B*. Стан завдання *A* запам'ятовується, і комп'ютер переходить до виконання завдання *B*.

Але ж і під час виконання завдання *B* комп'ютер може перемкнутися на інше завдання *C*, і потрібно буде зберегти стан завдання *B*, перш ніж перейти до *C*. Пізніше, після закінчення *C*, буде спершу відновлено стан завдання *B*, потім, після закінчення *B*, – стан завдання *A*. Отже, відновлення відбувається в послідовність, зворотній до збереження, що відповідає дисципліні роботи стека.

Стек дає змогу організувати рекурсію, тобто звернення підпрограми до самої себе або безпосередньо, або через ланцюжок інших викликів. Нехай, наприклад, підпрограма *A* виконує алгоритм, що залежить від вхідного параметра *X* і, можливо, від стану глобальних даних. Для найпростіших значень *X* алгоритм реалізується безпосередньо. У випадку складніших значень *X* алгоритм реалізується як звернення до застосування того самого алгоритму для простіших значень *X*. При цьому підпрограма *A* звертається сама до себе, передаючи як параметр простіше значення *X*. Під час такого звернення попереднє значення параметра *X*, а також усі локальні змінні підпрограми *A* зберігаються в стеці. Потім створюють новий набір локальних змінних і змінну, що містить нове (простіше) значення параметра *X*. Викликана підпрограма *A* працює з новим набором змінних, не руйнуючи попереднього набору. Після закінчення виклику старий набір локальних змінних і старий стан вхідного параметра *X* відновлюються зі стека, і підпрограма продовжує роботу з того місця, де її було перервано.

Насправді навіть не доводиться спеціально зберігати значення локальних змінних підпрограми в стеці. Річ у тім, що локальні змінні підпрограми (тобто її внутрішні змінні, які створюються на початку її виконання й знищуються наприкінці) розміщуються в стеці, реалізованому апаратно на базі звичайної оперативної пам'яті. На самому початку роботи підпрограма захоплює місце в стеці під свої локальні змінні – цю ділянку пам'яті в апаратному стеці називають **блоком локальних змінних**, або англійською **frame**. У момент закінчення роботи підпрограма звільняє пам'ять, видаляючи зі стека блок своїх локальних змінних.

Крім локальних змінних, в апаратному стеці зберігаються адреси повернення при викликах підпрограм. Нехай у деякій точці програми *A* викликається підпрограма *B*. Перед викликом підпрограми *B* адреса інструкції, що слідує за інструкцією виклику *B*, зберігається в стеці. Це так звана **адреса повернення** в програму *A*. Після закінчення роботи підпрограма *B* витягає зі стека адресу повернення в програму *A* і повертає керування за цією адресою. Тобто

комп'ютер продовжує виконувати програму *A*, починаючи з інструкції, що слідує за інструкцією виклику. У більшості процесорів є спеціальні команди, що підтримують виклик підпрограми з попереднім розміщенням адреси повернення в стек і повернення з підпрограми за адресою, що зберігається в стеку.

У стеці розміщують також параметри підпрограми або функції перед її викликом. Порядок їх розміщення в стек залежить від угод, прийнятих у мовах високого рівня. Так, у мові C або C++ у вершині стека зберігається перший аргумент функції, під ним другий і так далі.

3.1.1. Реалізація стека

Реалізація стека на базі масиву є класикою програмування. Іноді навіть саме поняття стека не зовсім коректно ототожнюється із цією реалізацією.

Базою реалізації є масив розміром *N* (рис. 3.2). Так реалізується стек обмеженого розміру, максимальна глибина якого не може перевищувати *N*. Індекси елементів масиву змінюються від 0 до *N – 1*. Елементи стека зберігаються в масиві так: елемент на дні стека розташовується на початку масиву, тобто в елементі з індексом *0*. Елемент, розташований над найнижчим елементом стека, зберігається в елементі з індексом *1*, і так далі. Вершина стека зберігається десь у середині масиву. Індекс елемента на вершині стека зберігається в спеціальній змінній, яку зазвичай називають **вказівником** стека (англійською Stack Pointer, або просто SP).

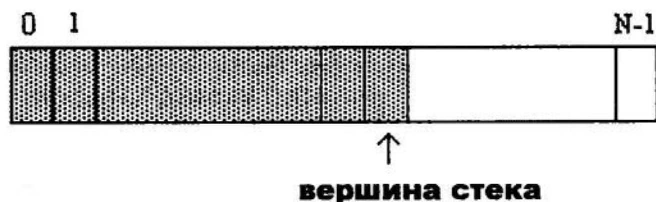


Рис. 3.2. Реалізація стека на базі масиву за збільшенням індексу

Коли стек порожній, вказівник стека містить значення мінус одиниця. Під час введення елемента вказівник стека спочатку збільшується на одиницю, потім в елемент масиву з індексом, що зберігається в вказівнику стека, записується елемент, який вводять. У разі видалення елемента зі стека спершу вміст елемента масиву з індексом, що

зберігається у вказівнику стека, запам'ятовується в тимчасовій змінній як результат операції, потім вказівник стека зменшується на одиницю.

У наведеній реалізації стек зростає у бік збільшення індексів елементів масиву. Часто використовується інший варіант реалізації стека на базі вектора, коли дно стека міститься в останньому елементі масиву, тобто в елементі з індексом *N – 1* (рис. 3.3). Елементи стека займають безперервний відрізок масиву, починаючи із елемента, індекс якого зберігається у вказівнику стека, і закінчуючи останнім елементом масиву. За цим варіантом стек росте в бік зменшення індексів. Якщо стек порожній, то вказівник стека містить значення *N*, яке на одиницю більше, ніж індекс останнього елемента масиву).

Рис. 3.3. Реалізація стека на базі масиву за зменшенням індексу

Розглянемо реалізацію стека дійсних чисел мовою C. Реалізація містить два файли: «streal.h», в якому описується інтерфейс виконавця «Стек», і «streal.cpp», що реалізує функції роботи зі стеком.

Використовується перший варіант реалізації стека на базі масиву: стек зростає у бік збільшення індексів масиву (рис. 3.2). Простір під масив елементів стека захоплюється в динамічній пам'яті в момент ініціалізації стека. Функції ініціалізації *st_init* передається розмір масиву, тобто максимально можлива кількість елементів у стеці. Для завершення роботи стека потрібно викликати функцію *st_terminate*, яка звільняє захоплену в *st_init* пам'ять. Нижче наведено вміст файла "streal.h", що описує інтерфейс стека.

```
// Файл "streal.h"
// Стек дійсних чисел, інтерфейс
//
#ifndef ST_REAL_H
#define ST_REAL_H

// Прототипи функцій, що реалізують приписи стека:

void st_init(int maxsize); // Почати роботу (вх: цілий
                           //     макс. розмір стека)
void st_terminate();       // Закінчити роботу
```

```
void st_push(double x); // Додати елемент (вх: дійс. x)
double st_pop();        // Вибрати елемент: дійс.
double st_top();        // Вершина стека: дійс.
int st_size();         // Поточний розмір стека: цілий
bool st_empty();        // Стек порожній? : лог.
int st_maxsize();       // Макс. розмір стека: цілий
bool st_freespace();     // Є вільне місце? : лог.
void st_clear();         // Вилучити всі елементи
double st_elementat(int i); // Елемент стека на
                            // глибині (вх: i): дійс.
#endif
// Кінець файла "streal.h"
```

Директиви умовної трансляції

```
#ifndef ST_REAL_H
#define ST_REAL_H
. . .
#endif
```

використовуються для запобігання повторному введенню h-файла: при першому введенні файла визначається змінна препроцесора *ST_REAL_H*, а директива *#ifndef ST_REAL_H* під'єднує текст, тільки якщо цю змінну не визначено. Такий спосіб використовується практично у всіх h-файлах. Потрібний він тому, що одні h-файли можуть під'єднувати інші, і без цього механізму уникнути повторного введення того самого файла важко.

Файл «streal.cpp» описує загальні статичні змінні, над якими працюють функції, що відповідають приписам стека, і реалізує ці функції.

```
// Файл "streal.cpp"
// Стек дійсних чисел, реалізація
#include <stdlib.h>
#include <assert.h>
#include "streal.h" // Під'єднати опис функцій стека

// Загальні змінні для функцій, що реалізують
// приписи стека:
static double *elements = 0; // Вказівник на масив елементів
                             // стека в дин. пам'яті
static int max_size = 0;      // Розмір масиву
static int sp = (-1);        // Індекс вершини стека

// Приписи стека:
void st_init(int maxsize) { // Почати роботу (вх:
//     макс. розмір стека)
```



```
    assert(elements == 0);
    max_size = maxsize;
    elements = (double *) malloc(
        max_size * sizeof(double)
    );
    sp = (-1);
}
void st_terminate() { // Закінчити роботу
    if (elements != 0) {
        free(elements);
    }
}
void st_push(double x) { // Додати елемент (вх: дійс. x)
    assert(   // твердження:
        elements != 0 && //   стек почав роботу і
        sp < max_size-1  // є вільне місце
    );
    ++sp;
    elements[sp] = x;
}
double st_pop() { // Вибрати елемент: дійс.
    assert(sp >= 0); // твердження: стек не порожній
    --sp;            // елемент видаляється зі стека
    return elements[sp + 1];
}
double st_top() { // Вершина стека: дійс.
    assert(sp >= 0); // твердження: стек не порожній
    return elements[sp];
}
int st_size() { // Поточний розмір стека: цілий
    return (sp + 1);
}
bool st_empty() { // Стек порожній? : лог.
    return (sp < 0);
}

int st_maxsize() { // Макс. розмір стека: цілий
    return max_size;
}

bool st_freespace() { // Є вільне. місце? : лог.
    return (sp < max_size - 1);
}

void st_clear() { // Видалити всі елементи
```

```
    sp = (-1);
}
double st_elementat(int i) { // Елемент стека на
                               // глибині (вх: i): дійс.
    assert(                    // твердження:
    elements != 0 &&           // стек почав роботу й
     0 <= i && i < st_size() // 0 <= i < розмір стека
    );
    return elements[sp - i];
}
// Кінець файла "streal.cpp"
```

3.1.2. Використання функції assert

Для реалізації стека мовою С неодноразово використовували функцію ***assert***, у перекладі з англійської **«твердження»**. Фактичним аргументом функції є логічний вираз. Якщо він істинний, то нічого не відбувається; якщо хибний, то програма завершується аварійно, видаючи діагностику помилки.

Функція *assert* є реалізацією конструкції «твердження», використання якої має на меті:

1) програміст під час написання програми може явно сформулювати твердження, яке, на його думку, повинне виконуватися в певному місці програми. У цьому разі конструкція «твердження» виконує роль коментаря, полегшуючи створення й розуміння програми;

2) комп'ютер, виконуючи програму, перевіряє всі явно сформульовані твердження. Істинність твердження відповідає припущенням програміста, зробленим під час написання програми, тому виконання програми триває. Хибність твердження свідчить про помилку програміста. Видається повідомлення про помилку, й виконання програми негайно припиняється. Отже, конструкція «твердження» дає змогу комп'ютеру перевіряти коректність програми безпосередньо в процесі її виконання.

Ситуація, коли програма аварійно завершується через те, що твердження не виконується, називається **відмовою**.

Багато приписів структури даних здійсненні не у всіх ситуаціях. Наприклад, елемент можна вибрати зі стека тільки у випадку, коли стек не порожній. Тому перед початком виконання припису «Вибрати елемент» перевіряється умова «стек не порожній»:

```
double st_pop() { // Вибрати елемент: дійс.
    assert(sp >= 0); // твердження: стек не порожній
    . . .
```


Якщо це твердження неправильне, то виникає ситуація «відмова»: комп'ютер завершує роботу, видаючи діагностику помилки. Невиконання твердження завжди свідчить про помилку програміста: програма має бути написана так, щоб некоректні вихідні дані не призводили до відмови.

Використання конструкції «твердження» – потужний засіб відлагодження програми. Важливість ситуації «відмова» усвідомлено в процесі еволюції програмування. Спочатку на першому плані були вимоги ефективності програми – її компактності й швидкодії. І лише в міру проникнення комп'ютерів у всі сфери життя на перший план вийшла надійність програми. У сучасному світі від надійності програми в багатьох випадках залежить людське життя, тому будь-які способи підвищення надійності дуже важливі.

Чому у разі невиконання твердження виникає ситуація «відмова»? Альтернативою могло б бути ігнорування некоректної ситуації й таке або інше продовження програми: наприклад, при спробі вибрати елемент з порожнього стека видавався б нуль. Насправді це найгірший розв'язок із усіх, які тільки можуть бути. Будь-яку помилку треба діагностувати й виправляти якомога раніше.

Тому рекомендується використовувати конструкцію «твердження» якнайчастіше. Якщо під час розроблення програми у цьому місці має виконуватися деяке твердження, то необхідно сформулювати його явно, щоб комп'ютер під час виконання програми міг би перевірити це твердження. Так знаходять й виправляють значну частину помилок.

При частому використанні конструкції «твердження» виникає проблема зі зменшенням швидкості виконання програми. Її вирішено в мовах С й С++ так: насправді конструкція *assert* у С – це не функція, а макровизначення, яке обробляється препроцесором. Текст С-програми може транслюватися у двох режимах: відлагодження і нормальному. У нормальному режимі відповідно до стандарту ANSI визначено змінну *NDEBUG* препроцесора. Для визначення макрокоманди *assert* використовується умовна трансляція: якщо змінна *NDEBUG* визначена, то *assert* визначається як порожній вираз; якщо ні, то *assert* визначається як перевірка умови й виклик функції *_assert* (до імені доданий символ підкреслення), якщо умова неправильна. Функція *_assert* друкує діагностику помилки й завершує виконання програми, тобто реалізує ситуацію «відмова».

Отже, умова реально перевіряється лише упродовж трансляції програми під час відлагодження. Завдяки цьому швидкодія програми не знижується, однак у нормальному режимі умова не перевіряється.

3.2. Черга

Черга – це лінійна динамічно змінювана послідовність елементів, у якій виконуються такі умови:

1) новий елемент приєднується завжди з одного і того самого боку послідовності;

2) доступ до елементів або їх вилучення завжди здійснюється з іншого боку послідовності.

Наприклад, нехай послідовність $Q=Q_1... ,Q_n$ зображує чергу. Тоді елемент Q_1 називають «головою» черги, який вказує на місце для вилучення елемента; елемент Q_n називають «хвостом» черги, який вказує на місце для введення елемента до черги.

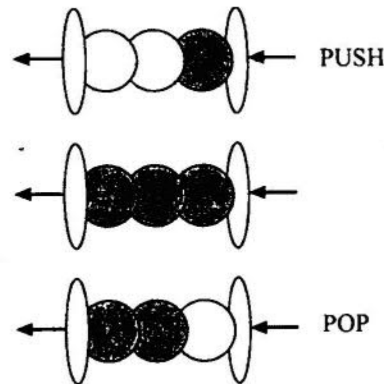


Рис. 3.4. Операції роботи з чергою

Чергу можна уявити у вигляді трубки. В один кінець трубки можна додавати кульки – елементи черги – з іншого кінця їх видаляють (рис. 3.4). Елементи в середині черги, тобто кульки усередині трубки, недоступні. Кінець трубки, у який додаються кульки, відповідає кінцю черги; кінець, з якого вони видаляються – початку черги. Отже, кінці трубки не симетричні, кульки усередині трубки рухаються тільки в одному напрямку.

Отже, в структурі черги потрібні два вказівники: один для посилання на вершину послідовності (для введення елемента), другий – для посилання на основу послідовності (для вилучення елемента). За кожного вилучення елемента із такої структури вилучається завжди найстаріший елемент. Черги обслуговуються за дисципліною ***FIFO*** – «перший прийшов, перший пішов».

Типовими операціями для черги є:

- **SIZE**(черга) – визначення кількості елементів у черзі;
- **PUSH** (черга, елемент даних) – додавання елемента в кінець черги;

- **POP** (черга) – вибірка елемента з початку черги;
- **EMPTY** (черга) – перевірка чи порожня черга; результатом є значення «істина», якщо черга порожня, і «хибно» у протилежному випадку.

Розрізняють такі типи черг:

- ➢ лінійні черги;
- ➢ черги з пріоритетом;
- ➢ циклічні черги.

Звичайну **лінійну** чергу можна зобразити масивом із двома вказівниками: перший вказує на елемент для вибірки з черги, другий – на останній елемент, записаний у чергу. Багаторазове звертання до елементів лінійної черги призводить до того, що пам'ять для зберігання такої черги використовується неефективно. У лінійній черзі після вибірки елемента пам'ять, зайнята чергою, звільняється і повторно не використовується, оскільки нові елементи приписуються з іншого краю послідовності.

Чергу, в якій є можливість вводити або вилучати елементи з певної позиції залежно від деяких її характеристик, називають **пріоритетною** чергою. Прикладом пріоритетної черги може бути порядок розв'язування потоку задач на комп'ютері у деяких операційних системах. Така черга зводиться до послідовності лінійних черг, якщо відомі пріоритети її елементів. Кожна черга з послідовності обслуговується за дисципліною *FIFO*, але елементи з другої черги обслуговуються тільки тоді, коли порожня попередня черга, а з третьої – тоді, коли порожні перша і друга черги. Під час введення елементи приєднуються до боку однієї з черг згідно з їхнім пріоритетом.

Черги, в яких елементи Q_1 ,Q_2, ...Q_n розміщуються так, що за елементом Q_n іде елемент Q_1, називаються **циклічними** (рис. 3.5). Циклічну чергу також можна зобразити лінійною послідовністю, але при записі нового елемента на відміну від лінійної черги вся послідовність зсувається на одне поле і новий елемент записується знову на її початку. Вибірка елемента буде також виконуватися за вказівником на кінець послідовності.

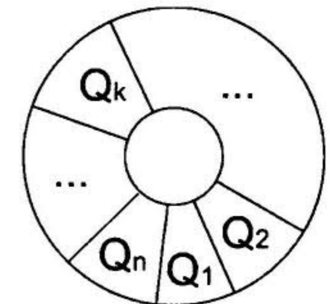


Рис. 3.5. Графічне зображення циклічної черги

Прикладом циклічної черги може бути робота обчислювальної системи з розподілом часу, з якою одночасно працює багато користувачів. Оскільки така система має переважно один блок обробки, який називають процесором, і одну пам'ять, то в кожний момент часу ці ресурси належать одному користувачеві. Для кожного користувача виділяється певний інтервал часу. Тільки на відміну від пріоритетної черги одна задача до кінця не розв'язується, а робить це «порціями» протягом виділеного їй інтервалу часу. Програми, що чекають на виконання, утворюють циклічну чергу.

Використання черги в програмуванні майже відповідає її ролі у звичайному житті. Черга практично завжди пов'язана з обслуговуванням запитів у тих випадках, коли вони не можуть бути виконані миттєво. Черга підтримує також порядок обслуговування запитів.

Будь-яка, навіть найпростіша, операційна система завжди тією чи іншою мірою багатозадачна. Це означає, що в момент натискання клавіші операційна система може бути зайнята якою-небудь іншою роботою. Проте операційна система ні в якій ситуації не має права проігнорувати натискання на клавішу. Тому переривається робота комп'ютера, він запам'ятовує свій стан і перемикається на обробку натискання на клавішу. Така обробка повинна бути дуже короткою, щоб не порушити виконання інших завдань. Команда, що віддається натисканням на клавішу, просто додається в кінець черги запитів, що чекають на своє виконання. Після цього переривання закінчується, комп'ютер відновлює свій стан і продовжує роботу, яку було перервано натисканням на клавішу. Запит, поставлений у чергу, буде виконано не відразу, а тільки коли настане його черга.

У системі Windows роботу віконних додатків основано на повідомленнях, які посилають цим додаткам. Наприклад, бувають повідомлення про натискання на клавішу миші, про закриття вікна, про необхідність перемальовування області вікна, про вибір пункту меню тощо. Кожна програма має чергу запитів. Коли програма одержує свій квант часу на виконання, вона вибирає черговий запит з початку черги й виконує його. Тобто, робота віконного додатка полягає в послідовному виконанні запитів з її черги. Черга підтримується операційною системою.

Підхід до програмування, що полягає не в прямому виклику процедур, а в посиланні повідомлень, які ставляться в чергу запитів, має багато переваг і є однією з ознак об'єктно-орієнтованого програмування. Так, наприклад, якщо віконній програмі необхідно завершити роботу з якої-небудь причини, краще не викликати відразу команду завершення, яка небезпечна, тому що вона порушує логіку роботи й може привести

до втрати даних. Замість цього програма посилає самій собі повідомлення про необхідність завершення роботи, яке буде поставлено в чергу запитів і виконане після запитів, що зробили раніше.

3.2.1. Реалізація черги на основі масиву

Усі структури даних можна реалізовувати на основі масиву. Зазвичай, така реалізація може бути багатоетапною і не завжди масив є безпосередньою базою реалізації. У випадку черги найпопулярніші дві реалізації: безперервна на основі масиву, яку називають також реалізацією на основі кільцевого буфера, і посилальна реалізація, або реалізація на основі списку.

При безперервній реалізації черги базою є масив фіксованої довжини N. Тобто черга обмежена й не може містити понад N елементів (рис. 3.6). Індекси елементів масиву змінюються в межах від 0 до $N-1$. Крім масиву, реалізація черги зберігає три прості змінні: індекс початку черги, індекс кінця черги, число елементів черги. Елементи черги зберігаються у відрізку масиву від індексу початку до індексу кінця.

Рис. 3.6. Реалізація черги на основі масиву

При додаванні нового елемента в кінець черги індекс кінця спочатку збільшується на одиницю, потім новий елемент записується в елемент масиву із цим індексом. Аналогічно, в разі видалення елемента з початку черги вміст елемента масиву з індексом початку черги запам'ятовується як результат операції, потім індекс початку черги збільшується на одиницю. Як індекс початку черги, так і індекс кінця під час роботи рухаються зліва направо. Що відбувається, коли індекс кінця черги досягає кінця масиву, тобто $N-1$?

Ключова ідея реалізації черги полягає в тому, що масив ніби зациклюється в кільце (рис. 3.7). Вважають, що за останнім елементом масиву розміщений його перший елемент (останній елемент має індекс $N-1$, а перший – індекс 0). При зрушенні індексу кінця черги праворуч, коли він вказує на останній елемент масиву, він переходить на

перший елемент. Тобто, безперервний відрізок масиву, зайнятий елементами черги, може переходити через кінець масиву на його початок.

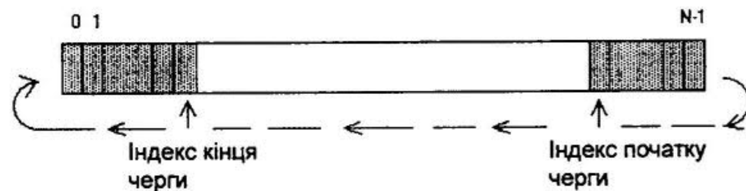


Рис. 3.7. Просування черги масивом

Циклічну чергу якнайкраще зображати циклічним або кільцевим списком, доповненим вказівниками на позицію для введення в чергу і вилючення елементів із черги.

3.3. Дек

Дек – це впорядкована лінійна динамічно змінювана послідовність елементів, у якій виконуються такі умови:

1) новий елемент може приєднуватися з обох боків послідовності;

2) вибірка елементів можлива також з обох боків послідовності.

Наприклад, якщо послідовність $D = D_1,...,D_n$ зображує дек, то її елементи D_1 *і* D_n вказують одночасно на місце введення і вилучення елементів.

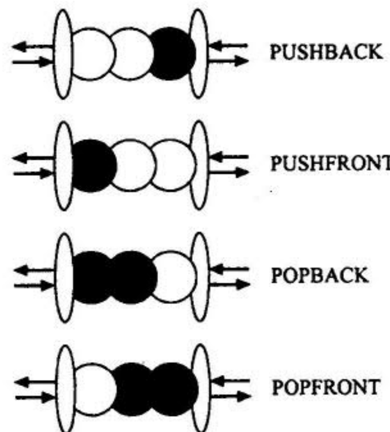


Рис. 3.8. Операції роботи з деком

Деки називають також **реверсивними**, або двосторонніми чергами, в яких введення і вилучення елемента здійснюють з двох боків послідовності.

Типовими операціями для дека є (рис. 3.8):

- **PUSHBACK** (дек, елемент даних) – додавання в кінець дека;
- **PUSHFRONT** (дек, елемент даних) – додавання на початок дека;
- **POPBACK** (дек) – вибірка з кінця дека;
- **POPFRONT** (дек) – вибірка з початку дека;
- **EMPTY** (дек) – перевірка, чи порожній дек.

Деки бувають з обмеженнями з одного боку на виконання однієї з операцій (наприклад, один бік послідовності може бути закритий для вибірки елементів і відкритий тільки для приєднування, і навпаки). Тоді вони стають звичайними чергами з дисципліною обслуговування **FIFO**.

Як і у випадку звичайних черг, для обслуговування деків також потрібні два вказівники – по одному за кожного боку послідовності. Деки є найзагальнішою лінійною динамічною структурою даних.

Задачі, що вимагають структури дека, трапляються в обчислювальній техніці і програмуванні набагато рідше, ніж задачі, що реалізовуються на структурі стека або черги.

Прикладом дека може бути певний термінал, до якого вводяться команди, кожна з яких виконується заданий час [8]. Якщо ввести наступну команду і не дочекатись закінчення виконання попередньої, то вона встане у чергу і почне виконуватися, як тільки звільниться термінал. Це черга з дисципліною обслуговування FIFO. Якщо ж додатково ввести операцію відміни останньої введеної команди, то виходить дек.

Контрольні запитання

1. Які є основні операції над лінійними структурами даних?
2. Що називається стеком?
3. Що таке верх і низ стека?
4. Що називають дисципліною *LIFO?*
5. Які є переваги стека перед іншими організаціями даних?
6. Що називається чергою?
7. Що називають дисципліною *FIFO?*
8. Дайте визначення лінійної черги.
9. Дайте визначення пріоритетної черги.
10. Дайте визначення циклічної черги.
11. Що називається деком?

Приклади тестових питань

1. Яка операція додає елемент до стека?
   а) PUSH;
   б) POP.

2. Яка операція видаляє елемент зі стека?
   а) PUSH;
   б) POP.

3. Скільки входів містить дек?
   а) 1;
   б) 2;
   в) 3.

4. Скільки вказівників необхідно для роботи з чергою?
   а) 1;
   б) 2;
   в) 3.

5. Скільки вказівників достатньо для роботи зі стеком?
   а) 1;
   б) 2;
   в) 3.

6. Скільки вказівників необхідно для роботи з деками?
   а) 1;
   б) 2;
   в) 3.

7. Яку структуру даних обслуговує дисципліна обслуговування LIFO?
   а) стек;
   б) черга;
   в) деки.

8. Яку структуру даних обслуговує дисципліна обслуговування FIFO?
   а) стек;
   б) черга;
   в) деки.

9. Яка дисципліна обслуговування черги?
   а) FIFO;
   б) LIFO.

10. Яка дисципліна обслуговує стек?
   а) FIFO;
   б) LIFO.

4. МАСИВИ, МНОЖИНИ, КОРТЕЖІ

4.1. Масиви

Масив – це набір однотипних елементів даних, з кожним з яких пов'язана впорядкована послідовність цілих чисел, які називають індексами. Індекси однозначно визначають місце цього елемента в масиві і забезпечують прямий доступ до нього. Локалізувати елемент у масиві означає задати його індекс. Кількість індексів визначає **розмірність** масиву. Кількість елементів визначають **розмір** масиву.

Розмір і розмірність масиву фіксовані. Найпростіший масив одновимірний, його називають також вектором. **Двовимірний масив** – **матриця**, де кожний елемент *b[i,j]* належить одночасно двом лінійним ортогональним послідовностям: *j*-му стовпцю та *i*-му рядку. Подібну структуру мають також масиви більшої розмірності, тобто кожний індекс масиву визначає одну ортогональну лінійну послідовність так само, як координати точки в багатовимірній системі координат.

Універсальні мови програмування (такі, як Фортран, Паскаль, С, С++) дають можливість працювати тільки з масивами ортогональної структури. Але ті самі двовимірні масиви можуть бути симетричними, діагональними, квазідіагональними, де відмінні від нуля тільки ті елементи b_{ij}, для яких $j = i$ або $j = i+1$. Трикутні матриці називають тетраедральними. Крім того, існують ще так звані розріджені матриці, в яких багато елементів є нульовими. Такі структури можуть мати масиви різних розмірностей. Кожна з цих структур має свою структуру зберігання в пам'яті комп'ютера. Переважно ортогональні багатовимірні масиви відображуються в одновимірні.

Найпоширенішими операціями над структурою масиву є:

- пошук елемента за заданим індексом;
- локалізація елемента в масиві;
- запис елемента в масив;
- злиття масивів і розбиття масиву на частини;

- сортування елементів масиву за деякими правилами;
- копіювання масивів.

Найскладнішою є операція локалізації або обчислення номера елемента відносно першого елемента. Розглянемо модифікації цієї операції для різних типів масивів і способів відображення їх у одновимірний масив.

Очевидно, що локалізація елемента вектора не є складною, оскільки індекс елемента однозначно визначає його номер у послідовності.

Номер елемента b_{ij} двовимірного масиву, що складається із n рядків та m стовпців, у разі зберігання в пам'яті комп'ютера «за рядками» обчислюється за формулою $(j-1)*n + (i-1)$, а у разі зберігання його «за стовпцями» – за формулою $(i-1)*m+(j-1)$.

У загальному випадку для n -вимірного масиву, елемент якого позначається $B\ [S_1,\ ...\ ,S_n\ ]$ і діапазон зміни індексів визначається нерівностями $1 <= S_i <= u_i$, $i = 1,n$, функція адресації цього масиву у разі запам'ятовування його «за рядками» має вигляд

$$f(S_1, ..., S_n) = u_2\ u_3 ...\ u_n (S_1-1) + u_3\ u_4 ...\ u_n (S_2-1) + ... + u_n (S_{n-1}-1) + (S_n-1).$$

i \ j	1	2	3
1	3	5	7
2	4	1	2
3	8	10	(6)
4	9	12	11

Рис. 4.1. Локалізація елемента у двовимірному масиві

Наприклад, для двовимірного масиву B[3×4] елемент, що знаходиться на перетині третього рядка та третього стовпця і має значення 6, відносно першого елемента масиву в разі зберігання «за рядками» знаходиться на восьмій позиції (рис. 4.1):
$n=3,\quad m=4,\quad i=3,\quad j=3,\quad f(b_{ij}) = (3-1)\cdot 3+(3-1) = 8,$

а в разі зберігання «за стовпцями» – на десятій позиції:

$n=3,\ m=4,\ i=3,\ j=3,\ f(b_{ij}) = (3-1)\cdot 4+(3-1) = 10.$

Оперативна пам'ять із погляду програміста – це масив елементів. Будь-який елемент масиву можна прочитати або записати відразу, за одну елементарну дію. Масив можна розглядати як базову структуру даних. Структури даних, у яких можливий безпосередній доступ до довільних їхніх елементів, називають структурами даних із **прямим**, або з **довільним доступом** (англійською random access).

З логічного погляду, масивом є також найважливіша складова комп'ютера – магнітний диск. Елементарною одиницею читання й запису для магнітного диска є блок. Розмір блока залежить від конятрукції конкретного диска, зазвичай він кратний 512. За одну елементарну операцію можна прочитати або записати один блок із заданою адресою.

Отже, найважливіші запам'ятовувальні пристрої комп'ютера – оперативна пам'ять і магнітний диск – являють собою масиви. Робота з елементами масиву відбувається винятково швидко – усі елементи масиву доступні без будь-яких попередніх дій. Проте масивів недостатньо для написання ефективних програм. Наприклад, **пошук елемента** в масиві, якщо його елементи не впорядковані, неможливо реалізувати ефективно: не можна винайти нічого кращого, крім послідовного перебору елементів. У випадку впорядкованого зберігання елементів можна використовувати ефективний **бінарний пошук**, але складнощі виникають під час додавання або видалення елементів у середині масиву й приводять до **масових операцій**, тобто операцій, час виконання яких залежить від кількості елементів структури. Від цих недоліків вдається позбутися, реалізуючи множину елементів на основі **збалансованих дерев,** або **хеш-функцій**.

Є й інші причини, за яких необхідно використовувати складніші, ніж масиви, структури даних. Логіка багатьох задач вимагає організації певного порядку доступу до даних. Наприклад, у випадку **черги** елементи можна додавати тільки в кінець, а забирати тільки з початку черги; у **стеку** доступні лише елементи у вершині стека, у **списку** – елементи до й за вказівником.

Нарешті, масив має обмежений розмір. Збільшення розміру масиву приводить до переписування його вмісту в захоплену область пам'яті більшого розміру, тобто знову ж до масової операції. Від цього недоліку позбавлені **посилальні реалізації** структур даних: реалізації на основі лінійних списків або на основі дерев.

4.2. Множини і кортежі

Множина – найпростіша структура, в якій між окремими ізольованими елементами немає ніякого внутрішнього зв'язку. Набір таких елементів являє собою множину, яка не має ніякої структури. Це сукупність даних деякого типу, елементи якої мають певну властивість. Але повинні бути чітко встановлені область визначення елементів даних і правила їх відбору у множину. Основними операціями над множинами є (рис. 4.2):

- об'єднання A ∪ B;
- перетин A ∩ B;
- різниця A \ B;
- перевірка елемента на належність цій множині.

Множину, на якій встановлено відношення порядку «<=», називають **впорядкованою**. Якщо таке відношення існує для всіх елементів множини, таку множину називають **повністю впорядкованою**, інакше це **частково впорядкована** множина.

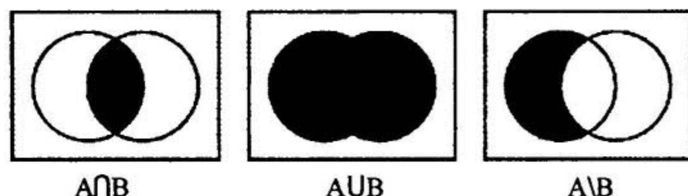


Рис. 4.2. Операції над множинами

Множину *M* називають **індексованою**, якщо задано її відображення в натуральний ряд чисел, тобто $I : M \rightarrow \{1,2,....|x|\}$, де $|x|$ – **потужність** множини *M*, тобто кількість елементів у множині.

У множину *M* можна додати елемент *x*, з множини *M* можна видалити елемент *x*. Якщо під час додавання елемента *x* він уже міститься в множині *M*, то нічого не відбувається. Аналогічно, жодних дій не відбувається в разі видалення елемента *x*, коли він не міститься в множині *M*. Нарешті, для заданого елемента *x* можна визначити, чи належить він множині *M*. Множина – це потенційно необмежена структура, яка може містити будь-яку кінцеву кількість елементів.

У деяких мовах програмування накладають обмеження на тип елементів і на максимальну кількість елементів множини. Так, іноді розглядають множину елементів дискретного типу, кількість елементів якого не може перевищувати деякої константи, що задається під час утворення множини. (Тип називається дискретним, якщо всі можливі значення цього типу можна занумерувати цілими числами). Такі множини називають *Bitset* («набір бітів»), або просто *Set*. Переважно для реалізації таких множин використовують бітову реалізацію множини на основі масиву цілих чисел. Кожне ціле число розглядається у двійковому представленні як набір бітів, що містить 32 елементи. Біти усередині одного числа нумеруються справа наліво (від молодших розрядів до старших); нумерація бітів триває від одного числа до іншого, коли ми перебираємо елементи масиву. Наприклад, масив з десяти цілих чисел містить 320 бітів, номери яких змінюються від 0 до 319. Множина у цій реалізації може містити будь-який набір цілих чисел у діапазоні від 0 до 319. Число *N* належить множині тоді й тільки тоді, коли біт з номером *N* дорівнює

одиниці. Відповідно, якщо число *N* не належить множині, то біт з номером *N* дорівнює нулю. Нехай, наприклад, множина містить елементи 0, 1, 5, 34. Тоді в першому елементі масиву встановлено біти з номерами 0, 1, 5, у другому – біт з номером 2 = 34 – 32. Відповідно, двійкове представлення першого елемента масиву дорівнює 10011 (біти нумеруються справа наліво), другого – 100, це числа 19 і 4 у десятковому представленні. Усі інші елементи масиву нульові.

У програмуванні доволі часто розглядають складнішу структуру, ніж проста множина: **навантажену множину**. Нехай кожний елемент множини міститься в ній разом з додатковою інформацією, яку називають навантаженням елемента. При додаванні елемента до множини потрібно також вказувати навантаження, яке він несе. У різних мовах програмування й у різних стандартних бібліотеках такі структури називають **Картою** (Map) або **Словником** (Dictionary). Справді, елементи множини ніби наносяться на навантаження, яке вони несуть. В інтерпретації Словника елемент множини – це іноземне слово, навантаження елемента – це переклад слова певною мовою (зрозуміло, переклад може містити кілька варіантів, але тут переклад розглядається як єдиний текст).

Усі елементи містяться в навантаженій множині в одному екземплярі, тобто різні елементи множини не можуть бути однакові. На відміну від самих елементів, їх навантаження можуть збігатися (так, різні іноземні слова можуть мати однаковий переклад). Тому елементи навантаженої множини називають ключами, їхнє навантаження – значеннями ключів. Кожний ключ унікальний. Вважають, що ключі відображаються на їхні значення. Як приклад навантаженої множини поряд зі словником можна розглянути множину банківських рахунків. Банківський рахунок – це унікальний ідентифікатор, що складається з десяткових цифр. Навантаження рахунка – це вся інформація, яка йому відповідає, що містить ім'я й адресу власника рахунка, код валюти, суму залишку, інформацію про останні транзакції тощо.

Найчастіше застосовувана операція в навантаженій множині – це визначення навантаження для заданого елемента *x* (значення ключа *x*). Реалізація цієї операції передбачає пошук елемента *x* у множині, тому ефективність будь-якої реалізації множини визначається насамперед швидкістю пошуку.

Кортеж – елемент *n*-кратного добутку множини $X : X{\cdot}X{\cdot}..{\cdot}X{=}X^n$. На відміну від скінченної впорядкованої множини, яка є підмножиною декартового добутку деяких множин X_1, X_2 , ..., X_n,

елементи кортежу можуть повторюватись. Другою відмінністю кортежів від множини є впорядкованість елементів кортежу.

Часто кортеж з n елементів визначається індуктивно через впорядковану пару, тобто n-ка (де $n > 2$) визначається як впорядкована пара її першого елемента, та кортеж з $n-1$ її останніх елементів:

$(a_1, a_2, ..., a_n) = (a_1, (a_2, ..., a_n))$

Отже:

1. 0-кортеж (тобто порожній кортеж) визначається як $\varnothing$.
2. Якщо x є n-ка, то {{a}, {a, x}} є (n + 1)-ка.

Наприклад, для трійки (1, 2, 2) це призводить до наступного визначення:

(1,(2,(2,()))) = (1,(2, {{2}, {2, ∅}})) = (1, {{2}, {2, {{2}, {2, ∅}}}}) = {{1}, {1, {{2}, {2, {{2}, {2, ∅}}}}}}.

4.3. Зберігання множин і масивів

Множини – це структури найпростішого вигляду. В пам'яті їх можна зображати двома способами:

1) для кожного елемента множини зберігати в пам'яті його опис аналогічно до математичного способу задання множини переліком її елементів;

2) визначити всі потенційно можливі елементи множини, а потім для будь-якої підмножини такої універсальної множини вказувати для кожного можливого елемента, чи належить він цій підмножині, чи ні. Цей спосіб аналогічний до предикатної форми задання множини в математиці.

У першому випадку множину зручно зберігати у вигляді вектора або лінійного списку. У другому використовуються вектори бітів. Для цього пронумеруємо всі можливі елементи множини, наприклад, від 1 до N. Відведемо вектор пам'яті із M бітів і визначимо множину, встановлюючи в i-й біт одиницю, якщо i-й можливий елемент присутній у цій множині, і нуль у протилежному випадку. Очевидно, якщо N велике, а конкретна підмножина мала, то краще користуватися першим способом. Другий спосіб має переваги, які дають змогу виконувати операції над множинами за допомогою логічних операцій машинного рівня.

Треба зазначити, що як тільки множина виявиться записаною в пам'яті, спосіб її зображення визначає на ній індексацію, і різні методи зображення дають змогу по-різному її обробляти.

Розглянемо реалізацію звичайної ненавантаженої множини. (Реалізація навантаженої множини аналогічна до реалізації звичайної, паралельно з елементами потрібно додатково зберігати навантаження елементів). У звичайній реалізації множини її елементи зберігаються в масиві, починаючи з першої комірки (рис. 4.3). Спеціальна змінна містить поточну кількість елементів множини, тобто кількість використовуваних у цей момент комірок масиву.

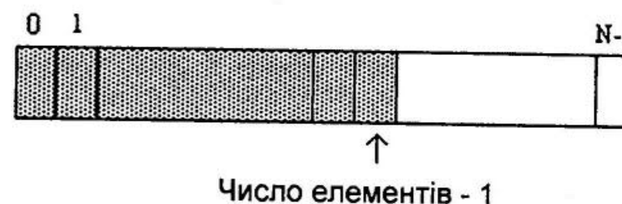


Рис. 4.3. Реалізація множини на основі масиву

Під час додавання елемента x до множини спочатку необхідно визначити, чи міститься він у множині (якщо міститься, то множина не змінюється). Для цього використовується процедура пошуку елемента, яка відповідає на запитання, чи належить елемент x множині, і якщо належить, видає індекс комірки масиву, що містить x. Та сама процедура пошуку використовується й під час видалення елемента з множини. В разі додавання елемента він дописується в кінець (у комірку масиву з індексом, що дорівнює числу елементів), і змінна числа елементів збільшується на одиницю. Для видалення елемента достатньо останній елемент множини переписати на місце, що видаляється, й зменшити змінну числа елементів на одиницю.

У звичайній реалізації елементи множини зберігаються в масиві у довільному порядку. Це означає, що при пошуку елемента x доведеться послідовно перебрати всі елементи та переконатись, що елемент x знайдено, або що його там немає. Нехай множина містить 1.00.00 елементів. Тоді, якщо x належить множині, доведеться переглянути в середньому 500.00 елементів, якщо немає – усі елементи. Тому звичайна реалізація підходить тільки для невеликих множин.

Зображення в пам'яті масивів очевидне. Найпростіший масив одновимірний, який ототожнюється з вектором. Багатовимірні масиви також зображуються векторами. Однак так звані розріджені матриці зображуються в пам'яті нелінійними багатозв'язними структурами.

Розрідженими називають такі матриці, більшість елементів яких дорівнює нулю. Такі матриці зберігаються у вигляді зв'язаних структур. Існують спеціальні алгоритми введення і множення розріджених матриць, зображених у такій формі. Одним з основних способів їх зберігання є табличний. Він полягає у запам'ятовуванні ненульових елементів матриці в одновимірному масиві та ідентифікації кожного елемента масиву індексами рядка і стовпця.

Наприклад, нехай задано розріджену матрицю (рис. 4.4).

Її можна зберігати у вигляді трьох векторів, які містять відповідно ненульові елементи матриці, а також індекси їх рядків та індекси стовпців: Z=*{6,9,2,7,8,12,3}* – значення ненульових елементів; *R = {1,1,2,2,2,4,5}* – індекси рядків; *S={3,5,1,4,5,3,4}* – індекси стовпців. Щоб не було повторень у векторі індексів рядків R, у ньому можна зберігати тільки індекс першого елемента у послідовності S (рис. 4.5).

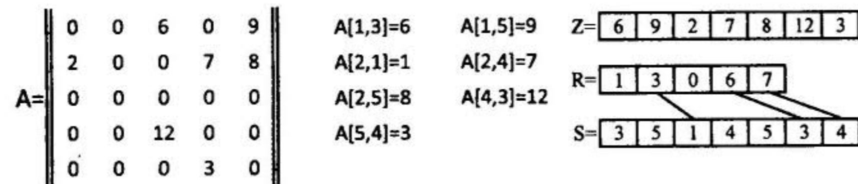


Рис. 4.4. Приклад розрідженої матриці

Рис. 4.5. Приклад зберігання розрідженої матриці

Контрольні запитання та вправи

1. Яку структуру називають масивом?
2. Які бувають масиви і чим вони характеризуються?
3. Як задається розмір масиву?
4. Що таке розмірність масиву?
5. Перелічіть найпоширеніші операції над масивами.
6. Нехай задано прямокутну матрицю A[20 × 30]. Обчисліть індекс елемента *A* (7, 13) в разі зберігання матриці «за стовпцями» і «за рядками».
7. Обчисліть індекс елемента A (5, 12, 9) тривимірного масиву A [10×20×15] в разі зберігання його «за рядками» і «за стовпцями».
8. Які ви знаєте способи зображення множин у пам'яті?

9. Яку множину доцільно зображати вектором бітів?
10. Який вигляд у пам'яті комп'ютера матиме множина з семи елементів, кожний з яких може дорівнювати квадрату числа від 1 до 10?
11. Як знайти елемент A_{ji} верхньої трикутної матриці без діагональних елементів, що зберігається у векторній пам'яті «за стовпцями»?

Приклади тестових питань

1. Прямий доступ до елементів масивів запезпечує:
   а) індекс;
   б) вказівник;
   в) ім'я масиву.

2. Розмірність масиву визначає:
   а) кількість елементів;
   б) кількість літер у назві масиву;
   в) кількість індексів

3. За якою формулою обчислюють адресу елемента двовимірного масиву в разі зберігання його «за рядками» ?
   а) $(j-1)\cdot n + (i-1)$;
   б) $(i-1)\cdot m+(j-1)$.

4. Адресу елемента двовимірного масиву в разі зберігання його «за стовпцями» обчислюють за формулою:
   а) $(j-1)\cdot n + (i-1)$;
   б) $(i-1)\cdot m+(j-1)$.

5. Розмір масиву визначає:
   а) кількість елементів у масиві;
   б) кількість індексів елемента масиву.

5. ЛІНІЙНІ СПИСКИ

5.1. Основні визначення та поняття

Раніше розглядались структури, в яких зв'язок між елементами задано неявно. Однак у класі лінійних структур даних існують значно складніші зв'язні структури, в яких функціональний зв'язок між його елементами задано явно. До таких структур належать спискові. Розглянемо найпростіші з таких структур, а саме лінійні списки.

Список, що відображає відношення сусідства між елементами, називають **лінійним.** Будь-який інший список вважають нелінійним. Тип списків визначається типом зв'язків між його елементами. За кількістю зв'язків розрізняють списки одно- і багатозв'язні, а за типом функції зв'язку – лінійні і нелінійні.

Основний принцип спискової структури даних полягає в тому, що логічний порядок слідування елементів задається сукупністю посилань або вказівників. У послідовній пам'яті дані розміщуються в послідовних одиницях пам'яті, чим визначається порядок їх слідування.

Отже, поняття «список» можна пояснити так: організацію зберігання даних, при якій логічний порядок слідування даних визначається посиланнями або вказівниками, називають **списковою організацією пам'яті**, а дані, що так зберігаються **списком**. Спискову організацію пам'яті також називають зчепленням.

Елемент списку складається як мінімум з двох полів: безпосереднього значення елемента даних і вказівника на наступний елемент списку. *У* деяких випадках значення елементів даних можуть зберігатися окремо, тоді у полі «значення» елемента списку може бути вказівник місцезташування цього елемента даних. Вказівники елементів списку називають також адресами зв'язку, або зв'язками.

Однією з основних властивостей списків є те, що елементи розміщуються в пам'яті за довільним способом, а не в суміжних одиницях пам'яті, як у векторів.

Кожний список має свій **заголовок**, у якому зберігається посилання на перший елемент списку. Заголовок списку називають також **іменем списку**. Всіх інших елементів досягають проходженням списку за допомогою вказівників.

Існує два способи зображення списків: 1) графічний (рис. 5.1, *а*); 2) дужковий (рис. 5.1, *б*).

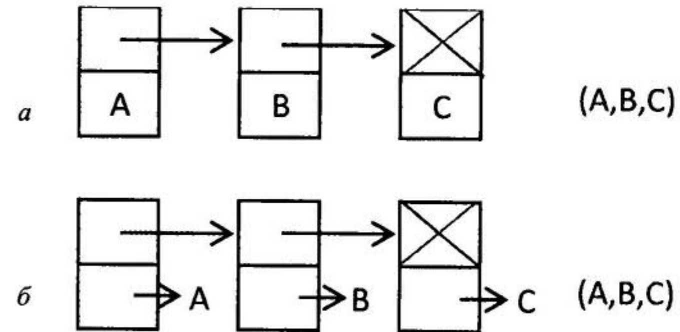


Рис. 5.1. Приклади зображення списків: а – графічний; б – дужковий

За графічним способом список зображують у вигляді ланцюга, кожна ділянка якого складається з двох полів – довідки і тіла. Дужковий вираз для зображення списку зручний тоді, коли списки складаються з різних типів даних, і зміна одного списку не впливає на інший. Наприклад, на рис. 5.1, *а* зображено список, у якого в полі тіла елемента записано безпосередньо значення елемента даних, а на рис. 5.1, *б* – вказівник місцезташування елемента даних. Дужковий вираз у обох випадках залишається однаковим і не відображає різного вмістимого поля тіла елементів списку.

Під час обробки списків найчастіше виконують такі операції [2]:

- доступ до *к*-го елемента списку з метою аналізу і заміни його полів;
- введення нового елемента безпосередньо перед заданим;
- вилучення заданого елемента;
- об'єднання декількох списків в один;
- розбиття списку на два або більшу кількість списків;
- копіювання списку;

- визначення кількості елементів у списку;
- знаходження елемента за заданими властивостями;
- пересортування або впорядкування елементів списку у висхідній або низхідній послідовності.

5.2. Однонапрямлені списки

Найприроднішим і простим типом списку є **однонапрямлений**, призначений для того, щоб проглядати його в одному напрямку – від початку до кінця. Однонапрямлений список найчастіше зображують у вигляді ланцюга, тому його називають також ланцюговим, або лінійним однозв'язним (рис. 5.2).

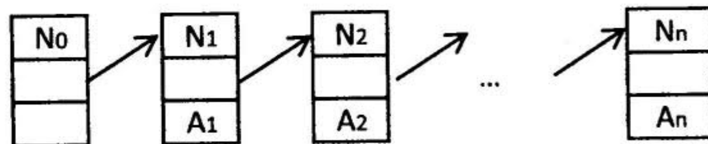


Рис. 5.2. Приклад однонапрямленого списку

Довідка кожної ділянки такого списку складається з двох значень. Першим є довжина N_i *i*-ї ділянки, яка, своєю чергою, складається з довжини запису і довжини довідки. Другим значенням довідки є посилання на початок наступної ділянки. Далі розміщується тіло ділянки або безпосередньо запис. **Заголовна** ділянка складається з трьох полів: довжини заголовної ділянки; посилання на початок першої ділянки; посилання на початок вільного місця. Перші два значення утворюють довідку заголовної ділянки, а посилання на вільне місце являє собою тіло цієї ділянки. У довідці останньої ділянки поле вказівника порожнє. У загальному випадку всі записи в однонапрямленому списку можуть мати різну довжину.

Однонапрямлений список можна відобразити в одномірний масив так. Заголовний запис займатиме елементи масиву *S[o], S[1], S[2]*. Потім розміщуються інші записи. Якщо індекс ділянки, що містить запис *X*, має значення *i*, то в елементі *S[i]* знаходяться довжина цієї ділянки, в елементі *S[i+1]* – індекс ділянки з наступним записом списку (або нуль, якщо запис *X* останній). Починаючи з елемента *S[i+2]*, знаходиться сам запис (тіло ділянки).

Схему процедури **введення** нового запису *Z* після ділянки з індексом *i* показано на рис. 5.3. Спочатку формується нова ділянка із записом *Z*, яка розміщується на вільному місці пам'яті, а потім корегується зв'язок *i* -ї ділянки.

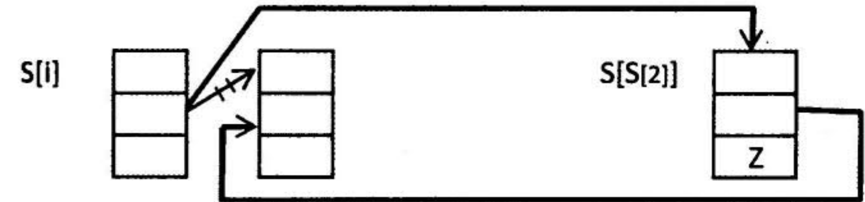


Рис. 5.3. Схема процедури введення елемента в ланцюговий список

Процедура **вилучення** елемента з такого списку також зводиться до корегування зв'язків між ділянками (рис. 5.4). Припустімо, що потрібно вилучити зі списку ділянку, яка розміщується після ділянки з індексом *i*. Отже, індекс ділянки, яку вилучають, розміщено в *S[i+1]*. Значення *S[i+1]* замінюється на значення, яке зберігалось у довідці ділянки, що вилучається.

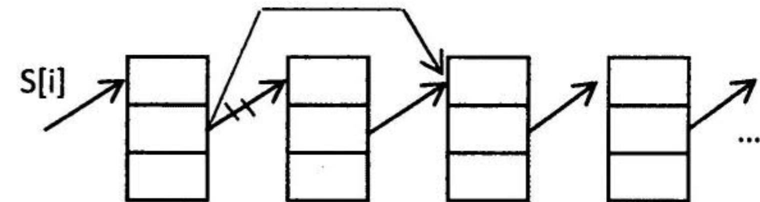


Рис. 5.4. Схема процедури вилучення елемента з однонапрямленого списку

Коли зі списку вилучаються записи, виникає питання про використання того місця пам'яті, де зберігалися вилучені записи. Якщо всі записи мають однакову довжину, нові записи можна розміщувати на цих місцях, для чого з вилучених ділянок утворюється **список вільних ділянок**. Якщо ж ділянки мають різну довжину, задача значно ускладнюється. Один із методів її розв'язання полягає у тому, щоб час від часу утворювалась копія списку, куди входили б тільки дійсні записи.

Будь-який список супроводжується списком вільної пам'яті, який готується обслуговуючими програмами. Такий список має власний вказівник, який містить адресу першого вільного елемента пам'яті, а також кількість таких елементів. Перший вільний елемент містить адресу другого елемента і т.д. Але побудова такого списку

вільної пам'яті значно ускладнюється, якщо записи елементів мають різну довжину. Тоді до довідки ділянки списку вільної пам'яті необхідно приєднувати також довжину запису. Процедуру, пов'язану з поверненням вільних квантів пам'яті до списку вільної пам'яті, називають **«збиранням сміття»**.

5.3. Двонапрямлені списки

Двонапрямлені списки – це такі списки, які дають змогу рухатися вперед і назад під час перегляду його елементів. Вони зображуються у вигляді ланцюга, кожна ділянка якого містить вказівники на наступний і попередній елементи.

На рис. 5.5 зображено структуру заголовної (рис. 5.5, *а*) і внутрішньої (рис. 5.5, *б*) ділянок, а також самого двонапрямленого списку (рис. 5.5, *в*). Кожна ділянка такого списку складається з чотирьох значень (рис. 5.5, *б*), а саме: значення самого елемента даних і довідки стандартного вигляду з трьох значень: довжини ділянки, вказівників на наступну і попередню ділянки. У загальному випадку довжина кожної ділянки може бути різною. Заголовна ділянка також складається з чотирьох значень, але її довжина відома і дорівнює чотирьом, оскільки записом її елемента є вказівник, а всі вказівники мають однакову довжину (рис. 5.5, *а*).

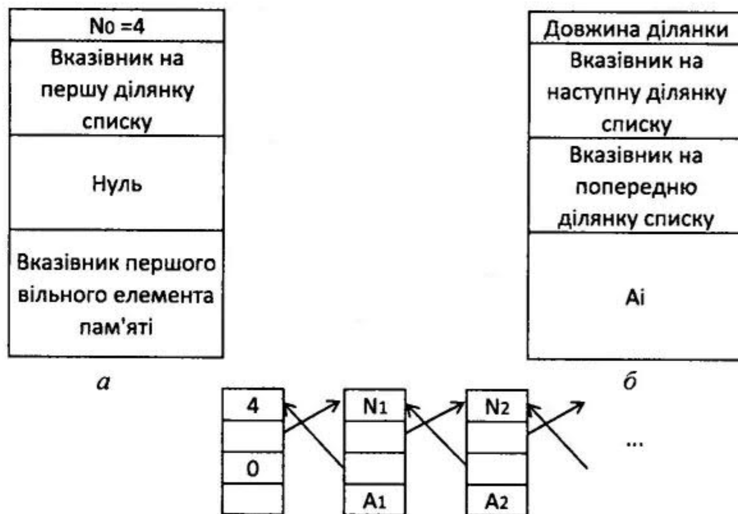


Рис. 5.5. Зображення двонапрямленого списку та його ділянок

Очевидно, що операції введення-вилучення запису в цьому випадку аналогічні до попередніх і також зводяться до корегування зв'язків між ділянками. Однак операція введення запису в двонапрямлений список буде значно складнішою від операції вилучення запису з нього, оскільки в першій операції необхідно прочитувати дві ділянки, а в другій – тільки ту, яку вилучають. Попередній запис потрібний для корегування зв'язку на наступний елемент, але цей запис не треба читати – знайдемо його у ділянці, яку вилучаємо. Тому можна записати туди нову адресу без читання самої ділянки.

Двонапрямлений список також відображується в одновимірний масив, починаючи зі стандартної заголовної ділянки, яка має в масиві індекси 0, 1, 2, 3.

5.4. Циклічні списки

Циклічні, або кільцеві, списки – найпоширеніший різновид двонапрямлених списків. У такому списку остання ділянка посилається на заголовну як на наступну, а ця, своєю чергою, на останню ділянку, як на попередню. Така організація спрощує процедуру пошуку або перебору записів з довільного місця з автоматичним переходом від кінця на початок і навпаки. Бувають також однозв'язні кільцеві списки, в яких остання ділянка посилається на заголовну, як на наступну. На рис. 5.6 зображено циклічний двозв'язаний список.

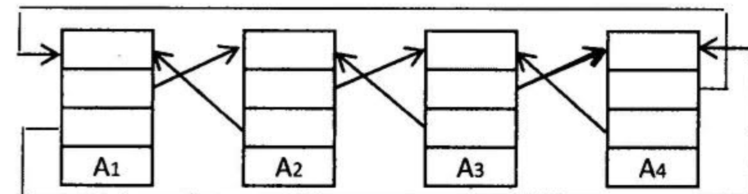


Рис. 5.6. Циклічний список

Операція **введення** до кільцевого двозв'язаного списку подібна до попередніх. Наприклад, нехай потрібно встановити ділянку з новим записом після ділянки з індексом *i*. Нова ділянка розміщується на вільному місці пам'яті і вводиться у список корегуванням вказівників у ділянці з індексом *i*, а також у тій ділянці, яка раніше була після неї.

Операція **вилучення** ділянки із кільцевого двозв'язного списку також зводиться до корегування двох посилань: у «сусіда ліворуч» змінюється вказівник на наступну ділянку, а у «сусіда праворуч» – вказівник на попередню ділянку.

Циклічний двозв'язний список можна відобразити у одномірний масив аналогічно до лінійного списку. У такому випадку процедури введення-вилучення елементів зводяться до обчислення індексів вказівників, як і в лінійному списку, і заміни вмістимого елементів масиву з цими індексами. Якщо тіло елемента списку являє собою рядок символів або бітів, а вказівники – цілі числа, то не в кожній мові програмування таке відображення можливе. Тому, наприклад, простіше відображувати списки у два масиви: один для тіла елемента, другий – для довідкової частини.

Частіше, однак, елементи списку не розташовують у якомусь масиві, а просто розміщують кожний окремо в оперативній пам'яті, виділеній для задачі. Зазвичай елементи списку розміщуються в динамічній пам'яті. Як вказівники у цьому випадку використовують адреси елементів в оперативній пам'яті.

Голова списку зберігає вказівник на перший і останній елементи списку. Оскільки список зациклений у кільце, то наступним за головою списку буде його перший елемент, а попереднім – останній елемент. Голова списку зберігає тільки вказівник й не зберігає жодного елемента. Це ніби порожня комірка, до якої не можна нічого покласти і яка використовується тільки для того, щоб зберігати адреси наступного й попереднього елементів списку, тобто першого й останнього. Коли список порожній, голова списку зациклена сама на себе.

Перевага вказівникової реалізації списку полягає в тому, що процедури додавання й видалення елементів не приводять до масових операцій.

Контрольні запитання та вправи

1. Який список називається лінійним?
2. Які бувають списки за кількістю зв'язків та за типом функції зв'язку?
3. З чого складається елемент списку?
4. Назвіть основні властивості списків.
5. Що записано в заголовку списку?

6. Які є способи зображення списків?
7. Які дії виконуються під час обробки списків?
8. Дайте визначення однонапрямленого списку.
9. Зі скількох значень складається довідка однонапрямленого списку?
10. Що записано у довідці останньої ділянки однонапрямленого списку в полі вказівника?
11. Поясніть процедуру введення нового запису в однонапрямленому списку.
12. Поясніть процедуру вилучення нового запису в одно напрямленому списку.
13. Які є варіанти використання місця пам'яті, де зберігалися вилучені записи однонапрямленого списку?
14. Що називають «збиранням сміття»?
15. Дайте визначення двонапрямленого списку.
16. Зі скількох полів складається ділянка двонапрямленого списку?
17. Дайте визначення циклічного списку.

Приклади тестових питань

1. Елемент списку складається з:
   а) цілої та дробової частин;
   б) дійсної та уявної частин;
   в) значення елемента даних і вказівника на наступний елемент списку.

2. Список називається лінійним, якщо
   а) всі його елементи розміщено в одну лінію;
   б) алгоритм його сортування має лінійну складність;
   в) він відображає відношення сусідства між елементами.

3. У заголовку списку записано:
   а) кількість елементів списку;
   б) функцію зв'язку;
   в) посилання на перший елемент списку.

4. Довідка однонапрямленого списку має таку кількість значень:

а) 1;
б) 2;
в) 3.

5. У довідці останньої ділянки однонапрямленого списку в полі вказівника записано:
а) вказівник на перший елемент;
б) вказівник на пусте місце;
в) поле є порожнім.

6. Довідка елемента двонапрямленого списку має таку кількість полів:
а) 2;
б) 3;
в) 4.

7. Довідка заголовної ділянки однонапрямленого списку має таку кількість значень:
а) 1;
б) 2;
в) 3.

8. Списки зображають:
а) діаграмами;
б) графіками;
в) графічно.

6. НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ

У попередніх структурах даних реалізовано переважно одновимірні відношення. Але існують структури даних, у яких виражені значно складніші відношення між їхніми компонентами. Всі такі структури утворюють важливий і складний клас нелінійних структур даних, основні з яких – таблиці з нелінійним відношенням сусідства елементів, дерева, спискові структури та сітки.

6.1. Таблиці

Поширеним видом діяльності є довідково-інформаційне обслуговування, яке містить нагромадження даних, прийом і видачу даних на вимогу. Типовою задачею довідково-інформаційного обслуговування є задача організації і сумісного зберігання однакових записів і видачі на вимогу довільного запису незалежно від того, які записи і в якій послідовності видавались раніше. У таких задачах використовують структуру даних, яку називають таблицею. **Таблиця** – це набір поіменованих об'єктів (або записів) довільної природи, з кожним з яких однозначно пов'язане його ім'я. Ім'я запису називають його **ключем**. Таблиці є основною структурою запам'ятовування у файлових структурах, організованих на зовнішній пам'яті комп'ютера.

Ідентифікатор	Адреса
(Ключ)	(Значення)
A	0101
B	0102
C	0103
D	0104

Рис. 6.1. Приклад таблиці

Ключ однозначно визначає місце кожного в таблиці і забезпечує прямий доступ до нього. Приклад таблиці наведено на рис. 6.1.

Кожний запис у таблиці містить один або більше ключів. З цими ключами пов'язується тип виконуваних дій. Наприклад, ключем

може бути номер службовця або його прізвище. Ключ може складатися з кількох атрибутів. Основні **властивості ключа**:

1) однозначна ідентифікація елемента;

2) відсутність надмірності.

Для одного елемента може існувати декілька наборів атрибутів, які задовольняють ці дві властивості. Ключ, який використовуватиметься для ідентифікації записів, називають **первинним**. Для нього атрибути слід вибирати так, щоб для них був заздалегідь відомий діапазон значень, а їх кількість була б якнайменшою.

Таблиці поділяють на **впорядковані** та **невпорядковані**.

У найпростіших невпорядкованих таблицях записи розміщуються один за одним без пропусків. Але невпорядковані таблиці неефективні під час пошуку елементів, тому їх застосовують у тих випадках, коли частота доступу до їхніх елементів невелика.

Частіше використовуються впорядковані таблиці. Вони можуть бути впорядковані за різними атрибутами, наприклад, за зростанням цифрового коду ключа або частотою звертання до записів. У першому випадку для пошуку записів застосовують алгоритм двійкового пошуку, у другому – алгоритм послідовного пошуку (див. розділ 9). Організація впорядкованих таблиць потребує додаткових витрат машинного часу.

Таблиці можуть мати ієрархічну структуру, наприклад, у вигляді дерева. Структура таблиці відображає насамперед шляхи доступу під час оброблення індивідуальних елементів таблиці. Для будь-якої табличної структури характерним є те, що операції введення і вилучення елементів можна виконувати в будь-якому місці. Операції з таблицями задаються відносно ключа, а виконуються також і над записом.

Основні операції з таблицями:

- введення: дано пару (ключ і запис); ввести запис до таблиці так, щоб його потім можна було знайти за ключем;
- заміна: дано пару (ключ і запис); знайти запис із заданим ключем і замінити його на задане значення;
- вилучення елемента за заданим ключем;
- пошук: за заданим ключем знайти відповідний запис;
- друк елементів таблиці.

У загальному випадку названі операції можна звести до двох основних: сортування і пошук. Ці операції залежать від методу доступу до елементів таблиці, а методи доступу, своєю чергою, від структури таблиці.

Існує три основні класи методів обробки таблиць залежно від характеристики їхніх структур:

1) використовує стратегію послідовного доступу;

2) організовує доступ за деревом;

3) робить шляхи доступу до елементів таблиці безпосередніми.

Структуру зберігання таблиць вибирають залежно від типу операцій, що виконуються над ними. Для внутрішніх таблиць можна вибрати довільну структуру, яку можна побудувати в оперативній пам'яті. Залежно від методу доступу до елементів таблиці поділяються на:

- **послідовні;**
- **деревоподібні**;
- **з обчислювальними адресами**.

Останні, своєю чергою, поділяються на таблиці з **прямим доступом** і **хеш-таблиці**. Усі, крім таблиць з прямим доступом, можна зберігати у векторній і зчепленій пам'яті.

У найпростіших послідовних таблицях для зберігання одного запису потрібна щонайменше одна ділянка пам'яті з двома полями: КЛЮЧ і значення. Таблицю можна зображати як вектор із цих ділянок або як лінійний список.

Деревоподібні таблиці або таблиці, організовані як дерева порівнянь, також можуть зображатися у векторній і зчепленій пам'яті. У першому випадку елементи даних повинні бути впорядковані за значенням ключа, у другому зображені у вигляді двозв'язного списку. Для цього до кожного елемента таблиці необхідно приєднати два вказівники; відповідно на ліве і праве піддерево. Тоді елемент таблиці перетворюється на такий:

ЛВ	КЛЮЧ	ТІЛО	ПВ

де ЛВ, ПВ – відповідно лівий і правий вказівники.

6.2. Дерева

Деревоподібні і взагалі графові структури мають широке застосування. Найчастіше **деревоподібні структури використовуються** [9]:

а) під час трансляції арифметичних виразів;

б) під час формування таблиць символів у трансляторах;

в) у задачах синтаксичного аналізу;

г) під час трансляції таблиць розв'язків.

Під час роботи з деревами часто використовуються рекурсивні алгоритми, тобто алгоритми, які можуть викликати самі себе. Під час виклику алгоритму йому передається як параметр вказівник на вершину дерева, яка розглядається як корінь піддерева, що росте із цієї вершини. Якщо вершина термінальна, тобто в неї немає синів, то алгоритм просто застосовується до цієї вершини. Якщо ж у вершини є сини, то він рекурсивно викликається для кожного із синів. Загальні графові структури застосовують під час зображення розріджених матриць, у машинній графіці, під час пошуку інформації і в інших складних задачах. Прикладом графоподібної структури є ієрархічна циклічна структура, що має рівні аналогічно до дерева або орієнтованого графу, але елементи на будь-якому рівні можуть бути циклічно зв'язані, як, наприклад, в ієрархічних списках. У структурах даних найчастіше використовується табличний або матричний спосіб задання графу.

Кореневим деревом називають орієнтований граф, у якого:

1) є одна особлива вершина, до якої не заходить жодне ребро і яку називають коренем дерева;

2) до всіх інших вершин заходить рівно одне ребро, а виходить скільки завгодно;

3) немає циклів.

Існує ще так зване рекурсивне визначення дерева, згідно з яким дерево трактують як скінченну множину вершин T, кожна з яких (крім кореня) належить до однієї з підмножин $T_1,..., T_m$; $m=>0$, $T_i \cap T_j = 0$, $i \neq j$. Підмножина T_i називається піддеревом цієї вершини. Число піддерев цієї вершини називають **степенем** цієї вершини. Вершину з нульовим степенем називають **листком**, усі решта називаються внутрішніми.

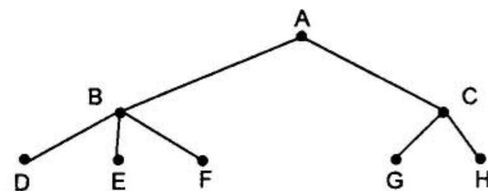


Рис. 6.2. Приклад 3-арного дерева

Рівнем, або **рангом** вершини відносно дерева називають довжину шляху від кореня до цієї вершини.

Довжина шляху – це кількість дуг, які треба пройти від кореня для досягнення вершини.

Висота дерева дорівнює кількості рівнів у дереві.

Кінцева множина непересічних дерев називається **лісом.**

Говорять, що кожний корінь є батьком коренів своїх піддерев. Останні є синами свого батька і братами.

Дерево називають ***n*** **-арним**, якщо кожний його вузол має не більше ніж *n* потомків. На рис. 6.2 зображено 3-арне дерево.

6.2.1. Бінарні дерева

Будь-яке дерево можна звести до бінарного. **Бінарним** називають таке 2-арне дерево, в якого один потомок є лівим, а другий – правим. Вершина бінарного дерева може взагалі не мати потомків або мати тільки ліве, або тільки праве піддерево, або обидва піддерева одночасно.

Бінарне дерево з *m* вершинами називають **збалансованим,** якщо різниця між рівнями будь-яких двох вершин не більша від одиниці. Наприклад, дерево на рис. 6.2 є збалансованим.

Бінарні дерева бувають позиційними і впорядкованими. У **впорядкованих** деревах між множиною вершин і натуральним рядом чисел існує взаємно однозначна відповідність. **Позиційні** дерева відрізняються від впорядкованих тим, що в перших важлива позиція вершини, а в других – тільки номер (рис. 6.3).

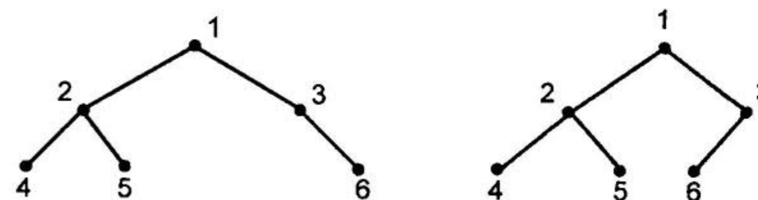


Рис. 6.3. Різні позиційні, але однаково впорядковані бінарні дерева

У позиційному бінарному дереві кожну вершину можна єдиним способом описати за допомогою рядка символів над алфавітом {0, 1}. Це дає значну зручність для зображення дерев у пам'яті комп'ютера. При цьому корінь дерева характеризується рядком «0». Будь-який син вершини Z характеризується рядком, префікс (початкова частина) якого є рядком, що характеризує Z. Рядок, приписаний будь-якій висячій вершині Z, не є префіксом для жодних рядків, що характеризують інші вершини дерева. Множина рядків, що відповідає висячим вершинам деякого дерева, утворює префіксний код цього дерева (рис. 6.4). Довжина рядка, що відповідає вершині, дорівнює рівню цієї вершини.

Детальніше про реалізацію бінарних дерев на програмному рівні буде розглянуто в розділі 11.6.

Багато алгоритмів формальних граматик і пошуку даних мають структуру бінарного дерева. Прикладом є алгоритми обробки пре-

фіксних формул у трансляторах арифметичних виразів. На рис. 6.5 зображено бінарне дерево, що відповідає арифметичному виразу:

$$a - b * (c / d + e / f).$$

При цьому знак операції міститься в корені, перший операнд – у лівому піддереві, а другий операнд – у правому. Дужки під час побудови такого дерева опускаються. У результаті всі числа й змінні розмістяться в листках, а знаки операцій – у внутрішніх вузлах.

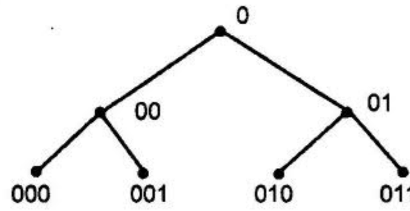


Рис. 6.4. Символьний опис дерева

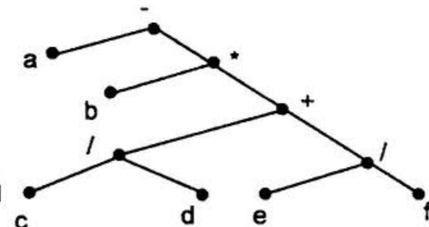


Рис. 6.5. Зображення формули у вигляді дерева

6.2.2. Алгоритми обходу дерева

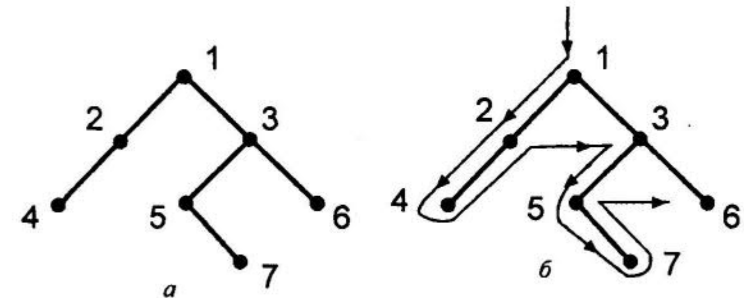
Алгоритм обходу дерева являє собою спосіб методичного дослідження вершин дерева, за якого кожна вершина проглядається тільки один раз. Повне проходження дерева дає лінійне розміщення вершин, після якого можна говорите про «наступну вершину» як таку, що розміщується або перед цією вершиною, або після неї. Розглянемо три алгоритми обходу бінарного дерева на прикладі дерева, зображеного на рис. 6.6, *а*.

1. Послідовність **низхідного обходу** (рис. 6.6, *б*): обробка кореня, низхідний обхід лівого піддерева, низхідний обхід правого піддерева. Вершини дерева проглядаються в послідовності: 1, 2, 4, 3, 5, 7, 6.

2. Послідовність **змішаного обходу** (рис. 6.6, *в*): змішаний обхід лівого піддерева, обробка кореня, змішаний обхід правого піддерева. Вершини дерева проглядають у послідовності: 4, 2, 1, 5, 7, 3, 6.

3. Послідовність **висхідного обходу** (рис. 6.6, *г*): висхідний обхід лівого піддерева, висхідний обхід правого піддерева, обробка кореня. Вершини дерева проглядаються у послідовності: 4, 2, 7, 5, 6, 3, 1.

Реалізацію алгоритмів обходу дерева наведено в розділі 11.6.2.



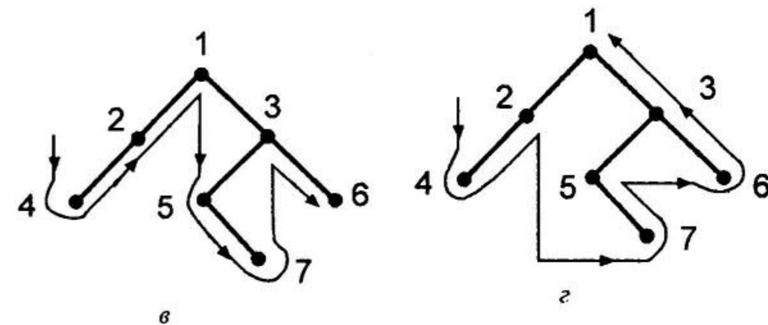


Рис. 6.6. Алгоритми обходу бінарного дерева:
а – бінарне дерево; б – низхідний обхід: згори донизу;
в – змішаний обхід: зліва направо; г – висхідний обхід: знизу догори.

6.2.3. Зображення в пам'яті комп'ютера графоподібних структур

Ознайомимось із загальною задачею зображення графу $G=(N,E)$ у вигляді спискової структури, тобто запам'ятовування його на зчепленій пам'яті. Розглянемо орієнтовані графи. Для кожної вершини n існує множина вершин $E(n)$, з якими її з'єднано ребрами. Тоді один із способів зображення такого графу полягає в тому, щоб із кожною вершиною зіставити одну ділянку списку, в якій одне поле відвести під мітку вершини, якщо така існує, і щонайменше $|E(n)|$ полів – під множину вказівників, тобто ребер. Якщо задано ваги ребер, то потрібно ще додатково $|E(n)|$ полів для зберігання цих ваг. Це один із способів зберігання графів у зчепленій пам'яті.

Але інформацію про ребра можна відділити від інформації про вершини, для чого потрібно ввести два типи ділянок (одну для вершини, другу для ребер) і зв'язати ці ділянки в один ланцюг. Крім

того, всі ребра можна зберігати окремо, по одній ділянці на кожне ребро, і зв'язувати ці ділянки в один ланцюг.

Таким способом зображення графів можна користуватися також для графів з двосторонньою орієнтацією – тоді кожний напрямок треба зображувати незалежно, що спричиняє дублювання інформації, і при зміні графу під час роботи потрібно було б змінювати його в двох місцях. Цю незручність можна ліквідувати за допомогою петель. У цьому випадку для кожного ребра *(a, b)* потрібна буде зв'язуюча ділянка, в якій зберігався б вказівник, що веде через інші вершини у вершину *a*, і вказівник, що веде через інші вершини у вершину *b* , а також додаткові поля для зберігання ваг і міток.

Отже, для зображення в пам'яті графів існує **декілька способів**:

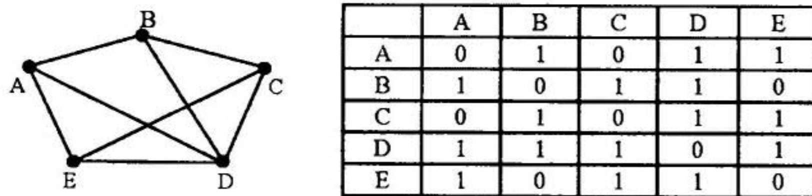
1) використовується матриця (або матриця суміжності, або матриця інцедентності), яка зберігається стандартним способом у векторній пам'яті;

2) використовується спискова структура у вигляді черги, в якій вказівники відповідають ребрам графу;

3) використовується спискова динамічна структура, де для кожної нової підмножини пам'ять виділяється в процесі побудови фрагмента графу.

Кожний з цих способів має свої позитивні та негативні сторони. Який із них більше підходить для зображення графу, залежить від кількості ребер.

На рис. 6.7 зображено граф та відповідну матрицю суміжності, в якій елемент дорівнює 1, якщо в графі існує ребро між відповідними вершинами рядка і стовпця, інакше, якщо ребро не існує, то відповідний елемент матриці дорівнює 0.



	A	B	C	D	E
A	0	1	0	1	1
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	1	0	1
E	1	0	1	1	0

Рис. 6.7. Зображення графу матрицею суміжності

Для зображення дерев як різновиду графів можна застосовувати ті самі способи, що і для зображення графів взагалі. Однак матриці

суміжності, як правило, не використовуються, оскільки вони дуже розріджені. Майже завжди **для зображення дерев застосовують списки,** в яких вказівники зображують ребра.

Дерева можна зберігати в пам'яті також у послідовному вигляді, але при цьому вибирати **певний** напрямок обходу дерева. Тоді зберігаються тільки вершини дерева, а зв'язки (ребра) опускаються, оскільки порядок розташування вузлів вказує на зв'язок між ними.

6.3. Спискові структури

Список, що містить усі підсписки, які відходять від нього, називають **списковою структурою**. Ці структури являють собою узагальнення лінійних списків.

Спискові структури описуються трьома характеристиками.

1. **Порядок**. Над елементами списку задано транзитивне відношення, що визначається послідовністю, в якій елементи з'являються в середині списку. Наприклад, у списку *(x,y,z)* елемент *x* передує *y* і *y* передує *z*. Список *(x,y,z)* не еквівалентний списку *(y,z,x)*. При графічному зображенні списків порядок визначається горизонтальними стрілками. При цьому розуміється, що елемент, з якого виходить стрілка, передує елементу, до якого заходить стрілка.

2. **Глибина**. Це максимальний рівень, який приписується елементам у середині списку або в середині будь-якого підсписку в списку. Рівень елемента визначається залежно від кількості підсписків, що містяться в середині списку, тобто числом пар дужок, якими оточується елемент.

Наприклад, на рис. 6.8 графічно зображено список *(a,(b,c,d), e,(f,g))*, у якого рівень елементів *a* і *e* дорівнює одиниці, а елементів *b,c,d,f,g* – двом. Глибина цього списку дорівнює двом.

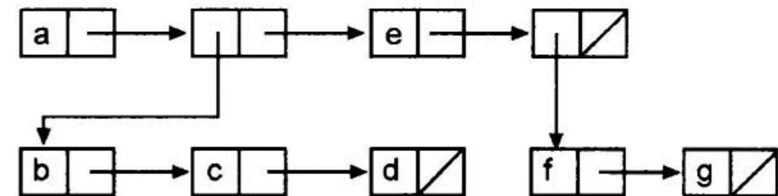


Рис. 6.8. Спискова структура

При графічному зображенні списку поняття глибини і рівня полегшуються для розуміння, якщо кожному одиночному елементу

приписати деяке число *k*. Значення *k* для елемента *x*, тобто *k(x)*, дорівнює кількості вертикальних стрілок, які необхідно пройти для того, щоб досягти даного з першого елемента списку. Наприклад, *k(a) = 0, k(b) = 1* (рис. 6.8). Тоді рівень елемента задається відношенням *k(x)+1*, а глибина списку дорівнює максимальному значенню рівнів серед усіх рівнів елементів списку.

Наприклад, максимальний рівень елементів у списку на рис. 6.8 дорівнює двом, тому глибина списку також дорівнює двом.

3. **Довжина.** Це кількість елементів списку першого рівня.

Наприклад, довжина списку *(a,(b,c), d)* дорівнює трьом, а списку, зображеного на рис. 6.8 – чотирьом, тому що він містить чотири елементи: елемент *a*, підсписок *(b, c, d)*, елемент *e* і підсписок *(f, g)*.

6.3.1. Ієрархічні списки

Спискові структури можуть бути достатньо складними. Прикладом складної спискової структури можуть бути ієрархічні списки, які об'єднують у один список записи з різним внутрішнім порядком. Така ієрархія записів може бути східчастою і достатньо глибокою. Якщо елементами списку верхнього рівня є підлеглі списки, то зручно вважати, що з погляду верхнього рівня запис, що в нього входить, є посиланням на підлеглий список, оформлений як заголовок цього підлеглого списку. При цьому в тілі заголовка підсписку може бути і деяка змістовна загальна інформація. Список верхнього рівня може містити поруч із заголовками підлеглих списків також самостійні елементи, тобто звичайні записи. Отже, список верхнього рівня зображується у вигляді ланцюга, в якого кожна ділянка складається з довідки і звичайного запису або заголовка підсписку – списку з нижнього рівня ієрархії (рис. 6.9).

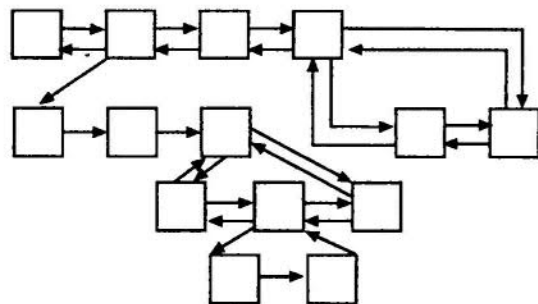


Рис. 6.9. Приклад структури ієрархічного списку

На рис. 6.9 ієрархічно пов'язані між собою списки різних типів. Двонапрямлений список найвищого рівня складається з чотирьох елементів, причому перший і третій елементи є звичайними записами; другий елемент є однонапрямленим списком, у якого третій елемент – двонапрямлений кільцевий список, що, своєю чергою, складається зі звичайного запису, однонапрямленого кільцевого списку і ще одного звичайного запису; четвертий елемент списку найвищого рівня являє собою двонапрямлений кільцевий список із двох записів.

6.3.2. Організація спискових структур

Розглянемо зображення в пам'яті спискових структур. Очевидно, що для них найзручнішим є зчеплене зображення, яке за необхідності забезпечує динамічне розміщення вершин, легкість обробки і здатність розділяти підсписки. Список у загальному випадку задається деякою варіацією бінарного зображення натуральних дерев з використанням двох полів зв'язку: одного для зазначення членства в середині списку і другого – для зазначення адреси.

Типову структуру елемента списку в комп'ютерному зображенні наведено на рис. 6.10, *а*.

Тут поле *ПС* вказує на перший елемент у підсписку; поле *НС* – на наступний елемент у цьому списку, а поле *ІЕ* містить інформацію безпосередньо про елемент даних.

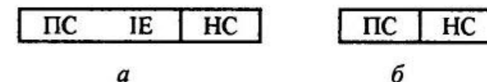


Рис. 6.10. Структура елемента списку в комп'ютерному зображенні

Але такий формат елемента зручний тільки тоді, коли інформація про нього займає невеликий обсяг пам'яті – тоді поле *ІЕ* буде невеликим.

У загальнішому випадку для елементів даних потрібен порівняно великий обсяг пам'яті, тому частіше використовуються формати, що складаються тільки з двох полів (рис. 6.10, *б*). Тут поле *НС* виконує ті самі функції, що й раніше, а поле *ПС* вказує або на перший елемент підсписку, що входить у вузол списку, або на інформацію, пов'язану з ізольованим елементом списку, який утворює ізольований вузол. Припускається, що ізольовані вузли містять якусь унікальну характеристику, наприклад, поле маркера, що дає змогу відрізняти ізольовані вузли від спискових. Такий формат дає змогу зробити всі вузли в середині спискової структури однаковими за розміром та уникнути зайвих втрат інформаційного простору у

вузлах. У наведеному нижче прикладі (рис. 6.11) елементи списку також зображено у форматі з двох полів, де *x* у полі *ПС* означає вказівник на інформацію, пов'язану з ізольованим елементом списку.

Приклади. Нехай маємо спискові структури в дужковому зображенні, де в дужки взято списки, а елементи поділяються комами (рис. 6.12):

а) *(a, (b,c, d),e, (f,g))* – довжина чотири (два ізольовані елементи списку – *a, e* і два підсписки – *(b,c,d), (f,g))*.

б) ( ) – нуль-список, довжина нуль;

в) *((a))* – довжина одиниця (ізольований елемент списку утворює підсписок).

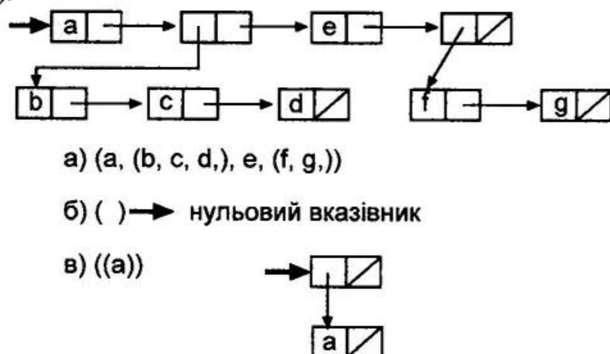


Рис. 6.11. Зображення спискових структур у пам'яті комп'ютера

Між списковими структурами і орієнтованими графами існує прямий зв'язок. Зокрема список являє собою орієнтований граф з однією початковою вершиною, що відповідає входу до списку. Кожна вершина безпосередньо пов'язана з початковою вершиною, що відповідає елементу списку: або з вершиною з півступенем виходу нуль (для ізольованих), або з вершиною, що має вихідні гілки (для елементів, які самі є списками). Ребра, що виходять із вершин, можна розглядати як впорядковані списки. Це означає, що розпізнається перше ребро, друге і т.д., які відповідають впорядкованим списковим елементам для першого, другого і т.д. елементів.

Тоді списки, зображені на рис. 6.11, можна зобразити також у вигляді графів (рис. 6.12, *а, б, в*).

Існують списки, які не можна зобразити як дерева, але кожне дерево можна зобразити у вигляді якогось списку. Списки можуть мати внутрішню рекурсивну структуру, чого не мають дерева. Такі списки відповідають безкінечним графам.

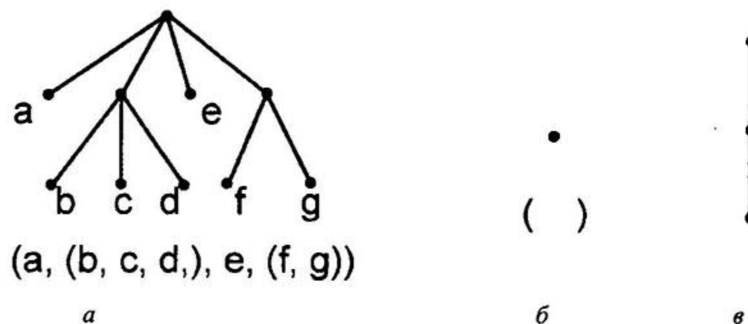


Рис. 6.12. Зображення спискових структур у вигляді графу

Отже, для зображення в пам'яті самих спискових структур існує багато способів, але всі вони мають спільні ознаки, які ґрунтуються на використанні бінарних дерев для зображення лісів. Одне з полів у записі вузла використовується для вказівника на наступний елемент списку, а друге поле можна використати для вказівника на перший елемент підсписку.

6.4. Сіткові структури

Якщо у відношенні між даними породжений елемент має більше ніж один вхідний або породжуючий елемент, то таке відношення уже не можна описати як деревоподібну або ієрархічну структуру – його описують у вигляді сіткової структури. Отже, **сіткова структура** – це орієнтований зв'язний граф, у якого будь-який породжений вузол може мати більше одного породжуючого вузла і, крім того, всі вузли розміщено так, що породжені елементи розміщуються нижче за породжувальні. У сітковій структурі будь-який елемент може бути пов'язаний з будь-яким іншим елементом (рис. 6.13).

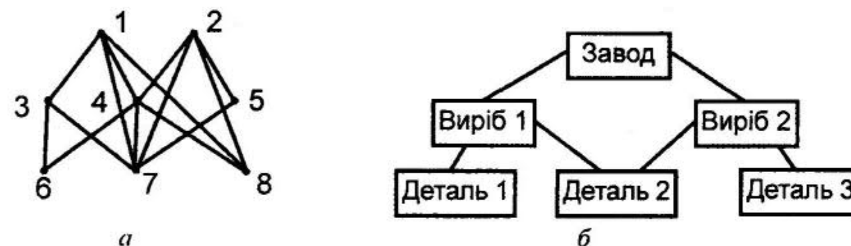
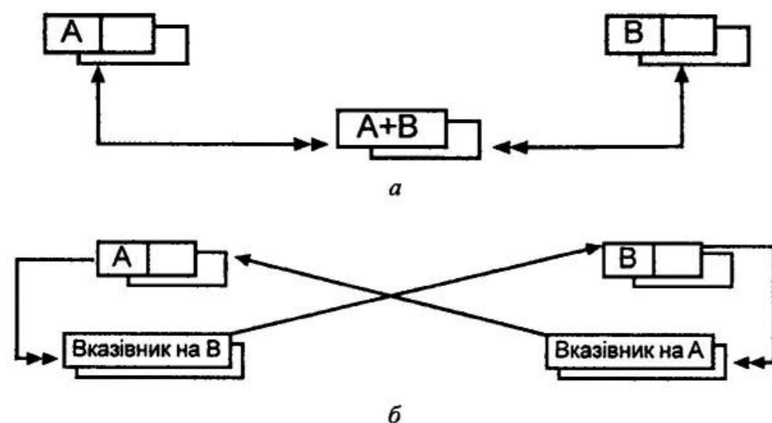


Рис. 6.13. Елементи сіткових структур: а – однорідні; б – неоднорідні

Однорідна сіткова структура, що містить однотипні елементи, вироджується в дерево або список.

Неоднорідна сіткова структура, в якій зображення відношення батько–син є складним (тобто в одного батька може бути декілька синів), а зображення відношення син–батько простим (у сина може бути тільки один батько), переважно також зводиться до дерева. Сіткову структуру з таким зображенням відношень називають простою.

Якщо зображення відношень син–батько також складне, таку структуру називають **складною**. На рис. 6.13 зображено складні сіткові структури. Існує скорочений спосіб графічного зображення складних відношень – здвоєні стрілки (рис. 6.14). Прості відношення зображуються однією стрілкою і називаються зв'язком типу *1:M*. Відношення між записами, що відображають зв'язки зі здвоєними стрілками в обох напрямках, прийнято називати зв'язком типу *M: N*. Майже завжди у складних сіткових структурах при зв'язках типу *M:N* виникають дані перетину, які породжуються з різнотипності вузлів сітки. Наприклад, у відношенні постачальник–виріб такими даними перетину є ціна.



б

Рис. 6.14. Приклади зображення даних перетину:
а – за допомогою файла; б – за допомогою вказівників

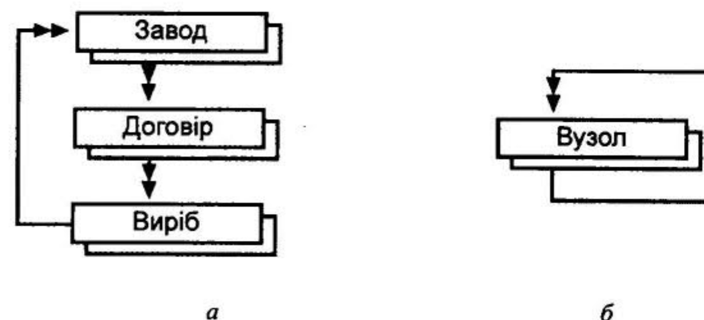
В основному процесі експлуатації бази даних виникають дві складні задачі:

1) зображення даних перетину у сіткових структурах;
2) доповнення даних перетину у базу даних.

На рис. 6.14 показано два способи підтримки зв'язків типу *M:N* між записами з ключами A і B з метою фіксації даних перетину: на рис. 6.14, *а* є файл записів *A* та файл записів B і окремо інформація про зв'язки між цими файлами; на рис. 6.14, *б* використовується ланцюг вказівників, який з'єднує кожний запис файла *A* з відповідними йому записами файла *B*. Введення в структуру даних перетину фактично руйнує зв'язок типу *M:N*.

Деякі сіткові структури містять цикли. **Циклом** у такій схемі вважають ситуацію, коли попередник вузла водночас є його послідовником. Відношення вхідний–породжений утворюють замкнутий контур. Наприклад, завод випускав різну продукцію. Деякі вироби виготовляються на інших заводах-субпідрядниках. За одним договором можуть виготовляти декілька виробів. Зображення цих відношень утворює цикл (рис. 6.15, *а*).

Деколи записи файла пов'язані з іншими записами цього самого файла. Таку ситуацію називають **петлею**. В петлі міститься тільки один тип запису, тобто тип породженого запису зливається з типом вхідного (рис. 6.15, *б*).



а *б*

Рис. 6.15. Цикли і петлі в сітках: а – цикл; б – петля

Усю сіткову структуру можна звести до простішого вигляду введенням надлишковості. Причому така надлишковість належить не до всього запису, а тільки до первинного ключа або способу його ідентифікації. Так саме перетворюють на деревоподібні структури і складні сітки – введенням надлишковості в ключах (рис. 6.16).

Перетворюють сітку на дерево дублюванням вершин, до яких заходить декілька ребер. Очевидно, що кількість повторень і рівень таких вершин у дереві залежить від кількості її вхідних ребер і вершин у сітці. Наприклад, на рис. 6.16, *а* вершину 3 записано у

дереві двічі на другому і третьому рівнях, тому що в початковій сітці до неї заходить два ребра: з вершин 1 і 2.

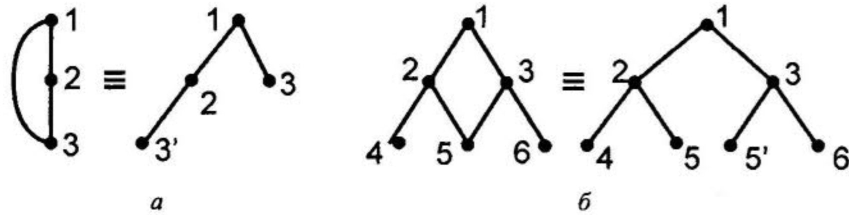
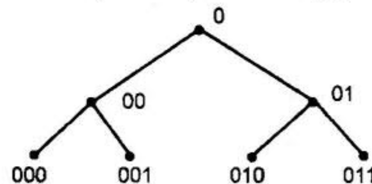


Рис. 6.16. Перетворення сіток на дерева:
а – звичайні сітки; б – решітки

Решіткою називають частковий випадок сіткової структури, в якій граф є кореневим і вхідні вузли деякого породженого вузла належать одному і тому самому рівню. На рис. 6.16, *б* зображено решітку. Всі ієрархічні структури, які мають спільні підструктури, являють собою решітки.

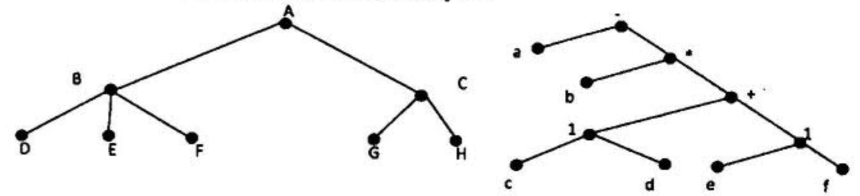
Контрольні запитання та вправи

1. Що називається таблицею?
2. Назвіть основні операції з таблицями.
3. Назвіть основні властивості ключа запису в таблиці.
4. Залежно від методу доступу до елементів таблиці поділяються на…
5. Яке дерево називається кореневим?
6. Що називають степенем вершини?
7. Чому дорівнює степінь листка?
8. Що таке довжина шляху?
9. Що таке рівень вершини дерева?
10. Який ранг має корінь зображеного дерева?



11. Що таке висота дерева?

12. Яка висота зображених дерев?



13. Як описується вершина «син» у позиційному дереві?
14. Які існують алгоритми обходу дерев?
15. Що таке бінарне дерево?
16. Яке дерево називають збалансованим?
17. Які відомі способи зображення графів у пам'яті комп'ютера?
18. Які способи зображення дерев у пам'яті комп'ютера є оптимальними?
19. Чим відрізняється лінійний список від спискової структури?
20. Назвіть основні характеристики спискових структур.
21. Вкажіть, які з наведених далі списків еквівалентні: (a,b,c,d,e,f), (a,(b,c),d,(e,f)),(a,c,b,d,e,f), (a,b,(d,f,e)), (a,(b,c),d,(e,f)), (a,b,c,d,e,f), (a,(b,c),(e,f)), (a,b,(d,f,e)), (a,b,(d,f,e)).
22. Визначить глибину і довжину всіх списків із вправи 21.
23. Які списки називають ієрархічними?
24. Визначить глибину і довжину всіх списків структури, зображеної на рис. 6.7.
25. Що називається сітковою структурою?
26. Що називається петлею у сітковій структурі?
27. Як поділяються сіткові структури за типом елементів?
28. Що називається циклом у сітковій структурі?
29. Що називається решіткою?

Приклади тестових питань

1. Первинний ключ – це:
   а) ключ для ідентифікації записів у таблиці;
   б) перший ключ з множини ключів запису;
   в) ключ першого запису таблиці.

2. Хеш-таблиця належить до лінійних структур даних:
а) так;
б) ні.

3. Первинний ключ – це ключ першого запису в таблиці:
а) так;
б) ні.

4. Місцезнаходження елемента в таблиці визначається ключем:
а) так;
б) ні.

5. Місцезнаходження елемента в таблиці визначається індексом:
а) так;
б) ні.

6. Висота дерева – це кількість:
а) рівнів у дереві;
б) листків;
в) дуг, які треба пройти від кореня для досягнення вершини.

7. Довжина шляху – це:
а) кількість дуг, які треба пройти від кореня для досягнення певної вершини;
б) кількість піддерев певної вершини;
в) ранг вершини.

8. Степінь вершини – це:
а) кількість піддерев;
б) кількість листків;
в) рівень вершини.

9. Рівень вершини дерева – це:
а) довжина шляху від кореня до цієї вершини;
б) кількість вершин, які треба пройти від кореня для досягнення цієї вершини;
в) кількість піддерев певної вершини.

10. Існують такі алгоритми обходу дерев:
а) низхідний, висхідний, змішаний;
б) паралельний;
в) бінарний.

11. Бінарне дерево – це таке:
а) 2-арне дерево, в якого один потомок є лівим, а другий – правим;
б) висота якого дорівнює 2;
в) що має два кореня.

12. Вершина син у позиційному дереві:
а) за допомогою рядка символів над алфавітом {0, 1};
б) цифрами;
в) формулою.

13. Збалансованим називають дерево:
а) різниця між рівнями всяких двох вершин не більша від одиниці;
б) яке має однакову кількість лівих і правих піддерев.

14. Оптимальний такий спосіб зображення дерев у пам'яті ЕОМ:
а) матриця;
б) список;
в) таблиця.

15. Степінь листка дерева дорівнює:
а) 0;
б) 1;
в) n;
г) висоті дерева.

16. Якщо довжина шляху від кореня до листка дорівнює 5, то рівень цього листка дорівнює
а) 4;
б) 5;
в) 6.

7. АЛГОРИТМИ. СКЛАДНІСТЬ АЛГОРИТМІВ

7.1. Визначення та способи зображення алгоритмів

Алгоритм – це формально описана обчислювальна процедура, яка отримує вихідні дані та видає результат обчислень на виході [3]. Вихідні дані також називаються входом алгоритму або його аргументом. Алгоритми будуються для розв'язання тих або інших обчислювальних задач. Формулювання задачі описує, яким вимогам повинен задовольняти розв'язок задачі. Алгоритм, що розв'язує цю задачу, знаходить об'єкт, який задовольняє ці вимоги.

До властивостей алгоритмів належать: *визначеність, скінченність, результативність, правильність, формальність, масовість* [8].

Визначеність алгоритму. Алгоритм визначений, якщо він складається з допустимих команд виконавця, які можна виконати для деяких вхідних даних.

Невизначеність виникне, якщо деяку команду буде записано неправильно, бо така команда не належатиме до набору допустимих команд, наприклад, ділення на нуль.

Скінченність алгоритму. Алгоритм повинен бути скінченним – послідовність команд, які треба виконати, має бути скінченною. Кожна команда починає виконуватися після виконання попередньої. Цю властивість ще називають **дискретністю** алгоритму.

Результативність алгоритму. Алгоритм результативний, якщо він дає результати, хоча вони можуть виявитися і неправильними.

Правильність алгоритму. Алгоритм вважається правильним, коректним, якщо для будь-якого входу він закінчує роботу та видає результат, що задовольняє вимоги задачі. Неправильний алгоритм може зовсім не закінчити роботи або дати неправильний результат.

Формальність алгоритму. Алгоритм формальний, якщо його можуть виконати не один, а декілька виконавців з однаковими резуль-

татами. Ця властивість означає, що коли алгоритм *A* застосовують до двох однакових наборів вхідних даних, то й результати мають бути однакові.

Масовість алгоритму. Алгоритм масовий, якщо він придатний для розв'язування не однієї задачі, а задач певного класу.

Прикладами масових алгоритмів є загальні правила, якими користуються для множення, додавання, ділення двох багатозначних чисел, бо вони застосовні для будь-яких пар чисел. Масовими є алгоритми розв'язування математичних задач, описаних у загальному вигляді за допомогою формул – їх можна виконати для різних вхідних даних.

Алгоритм відрізняється від системи та програми тим, що в ньому міститься тільки опис дій, що виконуються над даними, але повністю відсутні будь-які описи даних. Алгоритми можуть бути представлені за допомогою таблиць розв'язків, вербально з покроковим описом дій та псевдокодів. Алгоритми містять визначення покрокового процесу обробки даних з описом перетворень даних і функцій керування. Їх можна записати природною мовою, мовою програмування, а також за допомогою математичної або іншої символічної нотації. Назва алгоритму може вказувати на його призначення (наприклад, алгоритм сортування, обертання матриць, гри в «хрестики й нулики» тощо) або визначати використовуваний у ньому метод розв'язання.

Зображення алгоритмів найкраще пояснити на прикладі. Розглянемо алгоритм знаходження елемента вектора з найбільшим алгебраїчним значенням.

Алгоритм GT *визначає найбільший за значенням елемент вектора A, що містить n елементів, і присвоює його значення величині MAX. Символ i використовується як індекс елемента вектора A.*

GT1. Перевірка умови – чи вектор не порожній?
Якщо n <1 , то друк повідомлення; → GT6.
GT2. Початок: MAX = A[1], i=2.
GT3. Повторення кроків GT4, GT5 доти, доки i<=n.
GT4. Заміна значення MAX, якщо воно менше від значення наступного елемента: якщо MAX <A[i], то MAX =A[i]
GT5. i= i + 1.
GT6. Кінець, вихід.

Алгоритму присвоєно ім'я ***GT***. За ним іде короткий опис задачі, яку розв'язує алгоритм, і водночас описуються всі змінні, які тут

використовуються. Після цього наводиться сам алгоритм у вигляді послідовності кроків. Кожний крок описується фразою, яка коротко пояснює дію, що виконується. Символ «→» замінює оператор переходу ***go to*** в мовах програмування. Крок GT3 є аналогом заголовку циклу в мовах програмування. Коли значення індексу *i* перевершить величину *n*, відбудеться перехід на крок GT6 (наступний після кінця циклу).

Виконання будь-якого алгоритму починається з кроку *i* та продовжується послідовно доти, доки цю послідовність не порушить оператор умовного або безумовного переходу.

В алгоритмах часто трапляється слово «ініціалізація», що означає призначення деяких початкових значень змінним алгоритму. Слово «трасування» слід розуміти як запис послідовності виконуваних дій алгоритму та їхніх результатів для конкретних даних задачі.

7.2. Складність алгоритмів

Час роботи будь-якого алгоритму залежить від кількості вхідних даних. Так, під час роботи алгоритму сортування час залежить від розміру вхідного масиву. Загалом вивчають залежність часу роботи алгоритму від розміру входу. В одних задачах **розмір входу** – це кількість елементів на вході (задача сортування, перетворення Фур'є). В інших випадках розмір входу – це загальна кількість бітів, що необхідні для представлення всіх вхідних даних.

Часом роботи алгоритму, або **часовою складністю** називається число елементарних кроків, які він виконує [3]. Вважається, що один рядок псевдокоду потребує не більше ніж фіксованої кількості операцій.

Розрізняють також виклик функції, на який іде фіксована кількість операцій, та виконання функції, яке може бути довгим. Так, для алгоритму пошуку максимального за значенням числа у масиві розміром *n* необхідно виконати *n* кроків. Тоді час роботи такого алгоритму позначимо *T(n)*. Для кожного алгоритму оцінюють кожну операцію, скільки разів ця операція виконується. Всі оцінки додаються і виводиться залежність часу роботи алгоритму від розміру входу.

Якщо алгоритм призначений для обробки однакових за обсягом даних, тривалість його роботи щораз залишається незмінною й складність алгоритму є постійною. Час роботи алгоритму обробки яких-небудь масивів даних залежить від розмірів цих масивів. Час роботи алгоритму, що виконує тільки операції читання й занесення даних до оперативної пам'яті, визначається формулою $an+b$, де a – час,

необхідний для читання й занесення до пам'яті однієї інформаційної одиниці, і b – час, затрачуваний на виконання допоміжних функцій. Оскільки ця формула виражає лінійну залежність від *n*, складність відповідного алгоритму називають лінійною.

Визначення складності алгоритму переважно зводиться до аналізу циклів і рекурсивних викликів. Якщо алгоритм передбачає виконання двох незалежних циклів, перший від *1* до *k*, і другий від *1* до *t*, то загальна кількість кроків буде $k+t$, а час роботи $T(k+t)$. Якщо алгоритм передбачає виконання двох вкладених циклів, від *1* до *k* і від *1* до *t*, то загальна кількість кроків буде $k \times t$, а час роботи $T(k \times t)$. Це є лінійна функція, і вона є найсприятливішою для оцінювання швидкодії алгоритму. Що більший степінь складності алгоритму залежно від розміру входу (n^2, n^3, n^n...), то ефективність алгоритму нижча.

Час роботи алгоритму також залежить від подання вхідних даних. Так, для алгоритму сортування «бульбашкою» суттєво, як розміщено елементи вхідного масиву. Якщо їх частково відсортовано, то вихід за прапорцем відбудеться швидше. Якщо елементи стоять у різнобій, то алгоритм відпрацює всі цикли повністю.

Складність можна оцінити як стосовно задачі, так і стосовно алгоритму. Оцінками часу роботи алгоритму є максимальний, мінімальний і середній час його виконання. Залежно від використовуваних евристик ці оцінки можуть збігатися або не збігатися.

Складність задачі оцінюють за реалізаціями правильних алгоритмів. Вона являє собою верхню межу для часу роботи алгоритму, але часто можна знайти також і нижню межу. Очевидно, що для виконання якого-небудь виду обробки *n* елементів потрібен час, принаймні пропорційний *n*.

Час роботи в гіршому випадку є цікавішим для дослідження ефективності алгоритму, оскільки:

1) це гарантує, що виконання алгоритму закінчиться за деякий час, незалежно від розміру входу;
2) на практиці «погані» входи трапляються частіше;
3) час роботи в середньому є достатньо близьким до часу роботи в гіршому випадку.

Розглянемо три позначення асимптотики зростання часу роботи алгоритму [3].

Θ - позначення (тета)

Наприклад, якщо час роботи $T(n)$ деякого алгоритму становить залежність n^2 від розміру входу, то знайдуться такі константи c_1, $c_2 > 0$ і таке число n_0, що $c_1 n^2 \leq T(n) \leq c_2 n^2$ при всіх $n \geq 0$. Взагалі, якщо

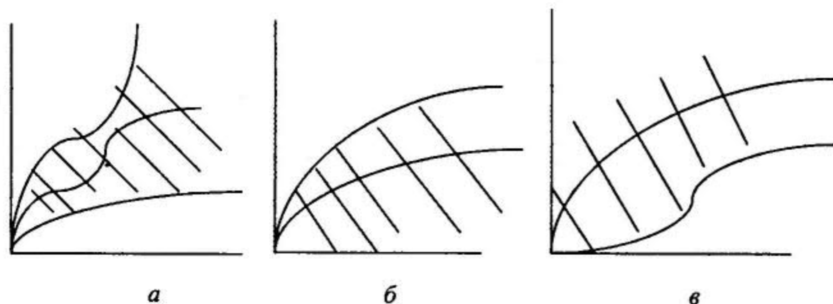
$g(n)$ – деяка функція, то запис $f(n) = \Theta(g(n))$ (тета) означає, що знайдуться такі $c_1, c_2 > 0$ і таке n_0, що $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всіх $n \ge n_0$. Тобто складність алгоритму геометрично розміщена в ≤ межах умовної смуги між функціями $c_1 g(n)$ і $c_2 g(n)$ (рис. 7.1, *а*).

***О*- позначення (о)**

Запис $f(n) = \Theta(g(n))$ містить дві оцінки: верхню і нижню. Їх можна розділити. Говорять, що $f(n) = O(g(n))$, якщо знайдеться така константа $c > 0$ і таке число n_0, що $0 < f(n) \le cg(n)$ для всіх $n \ge n_0$ – верхня оцінка, тобто складність алгоритму не перевищує деякої функції $cg(n)$ і геометрично знаходиться нижче від графіка (рис. 7.1, *б*).

***Ω* – позначення (омега)**

Говорять, що $f(n) = \Omega(g(n))$, якщо знайдеться така константа $c > 0$ і таке число n_0, що $0 \le cg(n) \le f(n)$ для всіх $n \ge n_0$ – нижня оцінка, тобто складність алгоритму не менша за деяку функцію $cg(n)$ і геометрично знаходиться вище її графіка (рис. 7.1, *в*).



а *б* *в*

Рис. 7.1. Визначення оцінок складності :
а – для $f(n) = \Theta(g(n))$; б – для $f(n) = O(g(n))$; в – для $f(n) = \Omega(g(n))$

Введені позначення мають властивості транзитивності, рефлективності та симетричності.

Транзитивність:

$f(n) = \Theta(g(n))$ *і* $g(n) = \Theta(h(n))$ *то* $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ *і* $g(n) = O(h(n))$ *то* $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ *і* $g(n) = \Omega(h(n))$ *то* $f(n) = \Omega(h(n))$

Рефлексивність:

$f(n) = \Theta(f(n)), \quad f(n) = O(f(n)), \quad f(n) = \Omega(f(n))$.

Симетричність:

$f(n) = \Theta(g(n))$ *якщо і тільки якщо* $g(n) = \Theta(f(n))$.

Наприклад, алгоритм зчитування й занесення в пам'ять множини даних має оцінку $O(n)$. Алгоритм двійкового пошуку в таблиці з упорядкованими елементами оцінюється як $O(\log_2 n)$. Просте обмінне сортування має оцінку $O(n^2)$, а множення матриць – $O(n^3)$. Коли говорять, що складність сортування дорівнює $O(n^2)$, то мають на увазі, що інформацію буде відсортовано в найкращому випадку за час, пропорційний n^2.

Складність розв'язку задачі можна зменшити, якщо знайти вигідніший метод (як, наприклад, при заміні лінійного пошуку двійковим) або використовувати оптимізаційні евристики.

У табл. 7.1 показано залежність різних видів складності алгоритмів від зростання *n*. Логарифмічна залежність прийнятніша, ніж лінійна, а лінійна залежність ніж поліноміальна й експонентна [1].

Таблиця 7.1

Складність алгоритмів

N	O (log₂n)	O(n)	O (nlog₂n)	O(n²)	O(2ⁿ)	O (n!)	O (nⁿ)
1	0	1	0	1	2	1	1
5	2,3219	5	11,6069	25	32	120	3,125
10	3,3219	10	33,2193	100	1,024	$3,6 \cdot 10^6$	10^{10}
20	4,3219	20	86,4386	400	10^6	$2,4 \cdot 10^{18}$	10^{26}
30	4,9069	30	147,2067	900	10^9	$2,6 \cdot 10^{32}$	$2 \cdot 10^{44}$
100	6,6439	100	664,3856	10000	10^{30}	10^{51}	10^{200}

7.3. Класи алгоритмів

Кожна наукова дисципліна має свої методи одержання результатів. Щодо цього програмування не є винятком. Основна відмінність між задачами полягає в тому, що для одних існують прямі методи розв'язання, а інші не можна вирішити без додаткової інформації, отриманої з відповідей на деякі питання. Варіанти відповідей заздалегідь передбачено.

Якщо задачу можна розв'язати прямим способом, говорять, що вона має ***детермінований*** метод розв'язання [1]. Детерміновані алгоритми завжди забезпечують регулярні розв'язки. У них відсутні елементи, що вносять невизначеність, крім того, для них неможлива довільність у виборі рішень, що визначають послідовність дій. Для побудови детермінованих алгоритмів неприпустимо застосовувати методи проб і помилок. До задач, що мають детермінований розв'язок, належать математичні рівняння, перевірка даних, друк звітів.

Якщо розв'язок задачі не можна отримати прямим методом, а необхідно вибирати серед заздалегідь визначеної множини варіантів, така задача має ***недетермінований*** розв'язок [1]. Алгоритм недетермінований, якщо для його реалізації використовуються методи проб і помилок, повторів або випадкового вибору. Це такі задачі, як знаходження дільників числа, пошук сторінки, а також класичні задачі про комівояжера, про вісім ферзів і задачі знаходження найкоротшого шляху через лабіринт [3].

Третій, основний тип алгоритмів, призначений не для пошуку відповіді на поставлену задачу, а для моделювання фізичних систем з використанням комп'ютера.

Багато задач можна вирішити за допомогою як детермінованих, так і недетермінованих алгоритмів. Алгоритм розв'язання задачі знаходження найбільшого спільного дільника для двох цілих чисел, побудований на основі перебору всіх можливих дільників, починаючи з найбільшого числа й закінчуючи одиницею, є недетермінованим. Інший спосіб розв'язання цієї задачі ґрунтується на використанні алгоритму Евкліда [3]. Цей алгоритм полягає в знаходженні послідовності пар чисел, кожна з яких утворюється з мінімального числа попередньої пари й різниці між числами попередньої пари. Відповідь представляється парою, що містить два рівні, але відмінні від нуля числа. За першим із цих двох методів кожне число перевіряється на відповідність умові. За другим така перевірка не потрібна. Але заздалегідь відомо, що правильне використання алгоритму дає правильну відповідь.

Відомі алгоритми розкладання числа на прості множники є недетермінованими, тому що в них використовується метод проб і помилок. Кожне передбачуване число можна перевірити на правильність діленням на нього вихідного числа, але єдиний відомий спосіб розкладання полягає у випробуванні різних варіантів множників. Було багато спроб математично визначити детермінований алгоритм розкладання числа на множники, але дотепер такий алгоритм не визначено. Деякі із задач є за природою недетермінованими, і можна показати, що для них не існує детермінованого алгоритму.

Задача про вісім ферзів є не детермінованою, оскільки під час її розв'язання використовується метод проб і помилок. Доти поки не буде знайдено правильного розміщення ферзів, розв'язок проводиться перебором місць їх розташування. Перед отриманням розв'язку можна переглянути безліч різних варіантів розташування ферзів. Недетермінований характер притаманний іграм і головоломкам, і це перетворює їх на приємну розвагу. Зміст ігор і головоломок полягає у

знаходженні стратегії, що дає змогу перемогти супротивника або розв'язати головоломку.

Недетермінований алгоритм описує систематичну процедуру пошуку потрібного розв'язку серед усіх можливих. Його існування насамперед залежить від побудови множини потенційних розв'язків, у якій міститься шуканий. Задачу про вісім ферзів можна розв'язати за допомогою певної послідовності усіх можливих спроб розміщення ферзів на шахівниці. Простий множник числа можна знайти, перевіряючи всі числа, не більші за квадратний корінь числа. Але ідеальний алгоритм для реальної гри в шахи не можна побудувати, оскільки послідовність варіантів гри занадто велика для реального використання методу перебору. Те саме можна сказати про розв'язок задачі про комівояжера.

Відмінність між задачею пошуку простих множників та іншими задачами полягає в тому, що під час її розв'язання для генерації і перевірки всіх варіантів простих множників повернення й повторних спроб не здійснюють. Отже, цей процес є ефективним.

Основним фактором під час вибору методу для задач, що розв'язуються за допомогою перебору великої кількості можливих варіантів, є сумарний час знаходження розв'язку. Методи, що використовуються для скорочення кількості варіантів під час перебору або які дають змогу вибирати найправдоподібніші варіанти, називаються **евристичними**. Іноді потрібно знайти правильний, але не обов'язково оптимальний розв'язок. Один з евристичних методів розв'язання задачі про вісім ферзів полягає в розміщенні ферзів попарно в кожному із чотирьох квадрантів шахової дошки.

Якщо для розв'язання задачі використовується метод проб і помилок, генерація й перевірка множини можливих варіантів розв'язання у відповідному недетермінованому алгоритмі повинні здійснюватися за деякими правилами, тобто систематично. Якщо для розв'язання можна використати різні недетерміновані алгоритми, перед вибором якого-небудь із них необхідно оцінити його ефективність порівняно з іншими. Ефективність застосування алгоритму залежить від методу генерації найімовірніших варіантів і від евристики, що скеровує й регулює пошук. Детермінований алгоритм вибирають, враховуючи економію часу або використану пам'ять комп'ютера. В іншому випадку можна використати найзрозуміліший алгоритм. Вибираючи алгоритм розв'язання задачі для підвищення ефективності процесу обчислень, слід віддавати перевагу детермінованим алгоритмам.

Алгоритми, час роботи яких за розміру входу *n* становить не більше ніж $O(n^k)$, називаються **поліноміальними** [3]. Клас поліноміальних задач позначається **P** і має властивість замкнутості. Композиція двох поліноміальних алгоритмів також працює за поліноміальний час. Це випливає з того, що сума, добуток та композиція багаточленів є багаточленом.

Задачі, для яких існують перевіркові алгоритми, що працюють за поліноміальний час, утворюють клас **NP**. Це означає недетермінований поліноміальний час. Інакше кажучи, клас **P** – це клас задач, які можна розв'язати швидко, а клас **NP** це клас задач, розв'язок яких можна швидко перевірити.

Якщо така перевірка неможлива за поліноміальний час, то відповідна задача називається **NP-повною**. Наприклад, задача про гамільтоновий цикл. Для розв'язання **NP-повних** задач є два варіанти:

1) відшукання експоненціального алгоритму, що працює для реальних даних за реальний час;

2) за поліноміальний час відшукати не оптимальне рішення, а деяке наближення до нього. Такі алгоритми називаються **наближеними алгоритмами**.

Прикладами **NP-повних** задач є задача комівояжера, задача про покриття множини, задача про суми підмножин та інші.

Якщо для розв'язання задачі алгоритм викликає сам себе для розв'язання підзадач цієї задачі, то такий алгоритм називається **рекурсивним**. Для оцінювання часу роботи рекурсивного алгоритму необхідно врахувати час рекурсивних викликів. Допустимо, що алгоритм розбиває задачу розміру *n* на *a* підзадач, кожна з яких має в *b* разів менший розмір. Вважаємо, що розбиття потребуєє часу $D(n)$, а з'єднання отриманих розв'язків – часу $C(n)$. Тоді час роботи всього алгоритму $T(n)= aT(n/b) + D(n) + C(n)$. Це відношення виконано для достатньо великих *n* , коли є сенс задачу розбивати на підзадачі.

7.4. Способи реалізації алгоритмів

Метою будь-якої програми є перетворення вхідних даних на вихідні. Якщо окремому вихідному елементу відповідає свій вхідний елемент, основу програми становить конструкція повторень. Якщо організація вихідного потоку відрізняється від організації вхідного, логічніше використовувати дві незалежні програми: одну для одержання вихідних даних, іншу – для їхньої організації. Як показано вище, ці програми можна реалізувати інвертуванням однієї з них, у результаті чого

утворюється модуль із декількома програмними секціями, що використовується іншою програмою. Якщо порядок проходження даних неістотний, можна використовувати паралельну обробку.

Усі види обробки можна поділити на такі класи: послідовну, що використовує повторення; структурну, що виконується паралельно за допомогою незалежних програм; довільну обробку із застосуванням паралельних обчислень [1].

7.4.1. Ітерація й рекурсія

Повторення є основною керуючою конструкцією обробки даних, за винятком випадків, коли вхідні дані складаються з одного елемента, який перетвориться на один вихідний елемент (наприклад, у випадку системи обробки запитів). Існують дві основні форми повторень: **ітерація** й **рекурсія**. Ітерація переважно використовується для тих видів обробки, які якнайкраще визначаються виразом типу «виконати для всіх *x*», а рекурсія – для одержання результуючих даних, які найлегше описати рекурсивно, тобто задати виразом вигляду «виконати те саме, що й востаннє». Поточна дія визначається за допомогою попередньої відповіді або попередніх стадій обчислень. Ітерація й рекурсія взаємозамінні.

Класичним прикладом рекурсії може слугувати визначення факторіала в такому вигляді:

$$n!=1\ n=0;\ n!=n(n-1)!\ n>0. \tag{7.1}$$

Це визначення є рекурсивним, тому що задає функцію за її допомогою. Усі рекурсивні визначення являють собою множину виразів, з яких принаймні один не є рекурсивним і забезпечує закінчення рекурсивних викликів. Аналогічно до цього для кожної ітерації необхідна наявність граничних умов. Інше, менш формальне, визначення факторіала описується виразом:

$$n! = 1 *2*3*...(n-1)*n. \tag{7.2}$$

Рекурсивну функцію, побудовану на математичній залежності (2.1), показано нижче:

```
f(n)
{ if (n==0) return 1;
 else return (n* f( n-1));
}
```

Наступна функція реалізує співвідношення (7.2) і заснована на ітераційному алгоритмі:

```
f(n)
{
```

```
m= 1;
k=n;
while (k>0)
{ m=m *k;
k= k -1;
}
 return( m );
}
```

В ітераційній функції використовується змінна *k* для підрахунку кількості повернень від *n* до 0. У рекурсивних функціях для цієї цілі призначений аргумент функції. Виконання операції множення починається зі старших чисел в ітераційній функції й з молодших – у рекурсивній. Умови закінчення виконання обох функцій аналогічні одна до одної.

Ітераційна й рекурсивна функції можуть застосовуватися для простої задачі друку списку чисел, що вводяться, яку зазвичай розв'язують ітераційним способом. Ітераційна функція містить явно заданий цикл.

Повторення рекурсивної функції утворюються за допомогою звернення її до самої себе. Умова закінчення обох функцій однакова.

Обидві функції розрахунків факторіала мають лінійну складність. У тому й в іншому випадках ітераційна форма краща, тому що для рекурсивної процедури потрібні додаткові витрати пам'яті й часу. За кожного рекурсивного виклику вміст усіх регістрів комп'ютера зберігається, а в разі повернення відновлюється так само, як і за будь-якого виклику функції. Крім цього, повинні зберігатися всі локальні змінні до моменту повернення, тому що вони можуть бути змінені функцією, що викликає. Оскільки відбувається багаторазове звернення з функції до самої себе, необхідно створити й зберігати багато копій регістрів, змінних і точок повернення. Для зберігання цієї інформації використовується стекова пам'ять. У разі повернення після відповідного виклику регістри відновлюються, й виклик завершується. Керування повертається до точки виклику стільки разів, скільки було викликано функцію. За звичайного виклику функції глибина **вкладеності** рідко перевищує 6. Глибина вкладеності рекурсивних викликів зазвичай значно більша. Рекурсію можна використовувати тільки в тих мовах програмування, у яких для реалізації викликів функції використовується стек.

7.4.2. Паралельна обробка

На прикладі задачі відновлення записів у файлах із прямим доступом можна найпростіше проілюструвати застосування паралельної обробки. Теоретично можна допустити, що кожного користувача обслуговує своя функція, що виконує одну операцію перезапису. Загальним ресурсом у цьому випадку є тільки файл. У такому разі необхідно координувати операції перезапису. Трудноці, пов'язані з паралельною обробкою, виникають під час розв'язання задачі координації доступу до файла. Одним з найпоширеніших видів процесорів для паралельної обробки є матричні процесори.

Є багато видів матричної обробки для різних математичних розрахунків, логіка яких дозволяє паралельне виконання. Якщо масив заповнюється при ініціалізації однією й тією самою величиною або якщо в кожний елемент, що ініціалізується, пересилається величина, що залежить тільки від положення елемента в масиві, послідовність, у якій заповнюються в цій ситуації елементи, не має значення. Ініціалізація може виконуватися за допомогою перерахування елементів масиву за зростанням їхніх номерів.

Можлива будь-яка інша послідовність роботи, тому операцію ініціалізації можна виконувати на *n* процесорах, що запускаються в довільному порядку, кожний з яких призначено для ініціалізації одного елемента масиву.

Паралельну обробку можна застосовувати для пошуку в неупорядкованому масиві за допомогою одночасного доступу до всіх елементів, для обробки неупорядкованого набору даних, виконання операцій на різних гілках програми й перевірки множини різних варіантів розв'язків недетермінованої задачі. Усі ці операції виконуються одночасно з доступом до всіх елементів.

7.4.3. Співпрограми

Співпрограми так само, як і паралельні процеси, виконуються одночасно. Їхня відмінність полягає в тому, що паралельні процеси стартують у той самий час і є рівнозначними за важливістю. Співпрограми складаються з головної програми й підлеглої їй підпрограми. Головна програма передає керування її співпрограмі. Після цього керування багаторазово може переходити від співпрограми до головної програми, поки остаточно не повернеться до головної програми. У разі передавання керування співпрограмі

остання, як правило, продовжує виконуватися з того місця, на якому її було перервано, на відміну від підпрограми, яка частіше починає виконуватися спочатку.

У багатопроцесорних системах співпрограми можуть виконуватися паралельно. Центральний процесор ініціалізує функціонування процесорів, призначених для обміну, після чого продовжує працювати одночасно з ними. Сигнали, передані між центральним процесором та іншими процесорами, мають координувати роботу системи. Це характерно для завдання типу «постачальник – споживач», яке являє класичний приклад взаємодії співпрограм. Один процес (*P*) постачає деякий продукт, інший (*C*) цей продукт використовує. *P* може випустити стільки одиниць продукту, скільки може розміститися в загальній області зберігання. Якщо ж область зберігання повна, то *P* повинен чекати *C*. *C* може використовувати одиниці продукту тільки в тому випадку, якщо вони є в області зберігання. Якщо ж вона порожня, то *C* повинен чекати *P*. Процес починається з того, що *P* робить першу одиницю продукту, а закінчується, коли або *C* одержав достатню для нього кількість продукту, або в *P* скінчився матеріал, потрібний для виробництва продукту, або коли *P* повідомляє *C* про припинення роботи.

Керування співпрограмами можна розглянути на прикладі друку звітів, що мають заголовки сторінок. Підпрограма-постачальник генерує вихідні рядки звіту з даних. Підпрограма-споживач друкує звіт посторінково, постачаючи кожну сторінку заголовком і закінченням. Якщо в мові програмування не передбачено взаємодії співпрограм, цього досягають інверсією програми – споживача, що забезпечує виконання її функції за допомогою звертання до окремих секцій інвертованої програми із програми-постачальника.

7.5. Документація алгоритмів

Документація алгоритмів є частиною документації програм і модулів. Якщо використовуваний алгоритм добре відомий або його опис можна знайти в спеціальному джерелі, документація повинна містити ім'я алгоритму або посилання на джерело і авторів. Так, наприклад, для розв'язання задачі про комівояжера можна скористатися алгоритмом Дейкстри. Для знаходження квадратного кореня використовують метод Ньютона–Рафсона, а для обчислення

sin(x) – поліноми Чебишова. Упорядковують списки за допомогою сортування Шелла, а також різних видів бульбашкового сортування.

Якщо в алгоритмі використано спеціальний математичний апарат (як, наприклад, у випадку біноміальних коефіцієнтів), до документації вводять математичне обґрунтування методу.

Якщо для спрощення задачі або збільшення швидкості розрахунку її на комп'ютері застосовуються які-небудь евристичні методи, їх зазначають у документації. До неї вводять загальний опис використовуваного методу й підходи до застосовуваної моделі, якщо це допоможе краще зрозуміти алгоритм. Також зазначають особливості реалізації алгоритму у випадку застосування рекурсії або паралельної обробки. Добре відомі алгоритми не мають потреби в детальному описі. Якщо ж використовуються менш відомі алгоритми або вони були розроблені програмістом, описи повинні бути докладними, можливо, із застосуванням псевдокодів.

Контрольні запитання

1. Що таке алгоритм?
2. Які відомі способи зображення алгоритмів?
3. Що називається часом роботи алгоритму?
4. Що таке **Θ** - позначення?
5. Що таке **O** - позначення?
6. Що таке **Ω** - позначення?
7. Які алгоритми називаються поліноміальними? Приклад.
8. Які задачі утворюють клас **NP**?
9. Яка задача називається **NP**-повною?
10. Який алгоритм називається рекурсивним?
11. Які алгоритми називаються детермінованими? Приклад.
12. Які алгоритми називаються недетермінованими? Приклад.
13. Дайте означення ітерації. Коли вона використовується?
14. Дайте означення рекурсії. Коли вона використовується?
15. Що таке паралельна обробка?
16. Що таке співпрограми?
17. Що входить до документації алгоритму?

Приклади тестових питань

1. Часом роботи алгоритму називають:
   а) кількість елементарних кроків алгоритму;
   б) кількість секунд (годин) роботи;
   в) кількість вхідних даних.

2. О-позначення означає:
   а) верхню оцінку складності алгоритму;
   б) нижню оцінку складності алгоритму;
   в) усереднену оцінку складності алгоритму.

3. Θ -позначення означає:
   а) верхню оцінку складності алгоритму;
   б) нижню оцінку складності алгоритму;
   в) усереднену оцінку складності алгоритму.

4. Ω -позначення означає:
   а) верхню оцінку складності алгоритму;
   б) нижню оцінку складності алгоритму;
   в) усереднену оцінку складності алгоритму.

5. До властивостей алгоритмів не належить:
   а) визначеність;
   б) скінченність;
   в) результативність;
   г) оптимальність.

6. Розмістіть за зростанням складності:
   а) $O(n^n)$;
   б) $O(\log_2 n)$;
   в) $O(n^2)$;
   г) $O(2^n)$.

7. Розмістіть за спаданням складності:
   а) $O(n)$;
   б) $O(n!)$;
   в) $O(n\log_2 n)$;
   г) $O(n^2)$.

8. МЕТОДИ СОРТУВАННЯ

8.1. Задача сортування

Загальна задача **сортування** є такою:
нехай дано множину елементів, яка є індексованою, тобто довільно пронумерованою від 1 до n . Необхідно індексувати цю множину елементів так, щоби з умови $i<j$ випливало $a_i < a_j$ – для всіх $i,j=1...n$.

Отже, процес сортування полягає у послідовних перестановках елементів доти, доки їх індексація не узгодиться з їх впорядкованістю.

Розглядатимемо ефективність алгоритмів у термінах розмірності множини щодо простоти програмування. Задачу сортування зручніше розглядати відносно одновимірних масивів (векторів).

Нехай маємо вектор з n елементів $R_1, R_2, ..., R_n$ .. Задача сортування полягає у тому, щоб знайти таку перестановку елементів вектора, після якої вони розмістилися б за зростанням їх значень.

Сортування називається **стійким**, якщо елементи з однаковими значеннями залишаються на попередньому місці. Для простоти аналізу вважатимемо, що всі елементи масиву, який сортується, займають однакові кванти пам'яті і жодні два елементи не можуть мати рівних значень, але в алгоритмах такий випадок необхідно передбачити.

Наведемо основні алгоритми у формальному аспекті і розглянемо на прикладах їх роботу.

8.2. Метод простої вибірки

Задано масив елементів $R_1, R_2, ..., R_n$. Цей алгоритм реорганізує масив у висхідному порядку, тобто для його елементів справедливе співвідношення $R_i < R_j$ – для всіх $i,j=1..n$.

Алгоритм S

S1. Цикл за індексом проходження. Повторювати кроки S2 - S 4 при i=1..n-1.

S2. Зафіксувати перший поточний елемент: встановити R0 = R_i.

S3. Пошук найменшого значення min R_j для елементів з індексом j=i+1,i+2,...,n

S4. Перестановка елементів. Якщо min R_j < R0 та j !=i , то min R_j <-> R0.

S5. Кінець. Вихід.

З алгоритму S видно, що для сортування потрібно виконати *n-1* проходження послідовності елементів. Одним проходженням називаємо пошук елемента з наступним найменшим значенням.

Проаналізуємо алгоритм. При першому проходженні, коли знаходиться елемент з найменшим значенням, порівнюється *n-1* елементів. У загальному випадку при і -му проходженні під час сортування порівнюється *n-i* елементів. Тоді загальна кількість порівнянь, які треба виконати для сортування масиву із *n* елементів, становитиме:

$$\Sigma (n-i) = (n-1) + (n-2) + ... + (n-n+1) = ½*n*(n-1), i=1..n-1$$

Отже, ефективність алгоритму пропорційна до величини n^2. Алгоритм S має ефективність ***O(n²)***.

Кількість перестановок елементів залежить від того, як спочатку було відсортовано масив [5]. Але оскільки під час одного проходження у цьому алгоритмі потрібно виконати не більш як одну перестановку, максимальна кількість перестановок за такого сортування дорівнює величині *n-1*.

Приклад програмної реалізації

Вихідний код програмної реалізації методу простої вибірки мовою програмування С має такий вигляд:

```
void sort(int *arr, int cnt)
{
 for (int i = 0; i < cnt - 1; i++)
 {
        int min = i;
        for (int j = i + 1; j < cnt; j++)
              if (arr[j] < arr[min])
                     min = j;
        int tmp = arr[i];
        arr[i] = arr[min];
        arr[min] = tmp;
 }
}
```


Покрокове сортування методом простої вибірки наведено в табл. 8.1. Для вхідного масиву з семи елементів алгоритм виконав 7–1 =6 перестановок.

Таблиця 8.1

Покрокове сортування методом простої вибірки

Вхідний масив	6 15 4 20 7 13 0
Крок 1	**0** 15 4 20 7 13 **6**
Крок 2	0 **4** **15** 20 7 13 6
Крок 3	0 4 **6** 20 7 13 **15**
Крок 4	0 4 6 **7** **20** 13 15
Крок 5	0 4 6 **7** **13** **20** 15
Крок 6	0 4 6 **7** 13 **15** **20**
Вихідний масив	0 4 6 **7** 13 15 20

8.3. Метод бульбашки

Другим добре відомим методом із класу вибірки є метод, що ґрунтується на ідеї спливаючої бульбашки [5]. На відміну від попереднього, в цьому алгоритмі два елементи обмінюються місцями, як тільки виявлено, що між ними порушено порядок.

АЛГОРИТМ B

Задано масив елементів R_1 ,R_2 ,...,R_n.

Цей алгоритм реорганізує масив у висхідному порядку, тобто для його елементів існуватиме співвідношення R_i < R_j – для всіх i,j=1..n.

B1. Цикл за індексом проходження. Повторювати кроки B2 і B3 при i=1..n-1.

B2. Ініціалізація прапорця перестановки: встановити Fl=0.,

B3. Виконання проходження. Повторювати при j=1,2,...,n-i : якщо R_{j+1} <R_j , то встановити Fl=1 та переставити місцями елементи R_j<->R_{j+1} ; якщо Fl=0 , то завершити виконання алгоритму.

B4. Кінець. Вихід.

Робота алгоритму B очевидна. Перед кожним проходженням змінна *Fl* набуває значення нуль. Її значення аналізується в кінці кожного кроку. Якщо воно не змінилося, сортування виконане повністю. Характеристика сортування бульбашкою в гіршому випадку становить *1/2n(n-1)* порівнянь і *1/2n(n-1)* перестановок. Середня кіль-

кість проходжень приблизно дорівнює величині $1,25n\sqrt{n}$. Наприклад, якщо $n=10$, потрібно виконати шість проходжень послідовності.

Отже, середня кількість порівнянь і перестановок пропорційна величині n^2. Складність алгоритму сортування бульбашкою становить $O(n^2)$. Цей метод неефективний для масивів великого розміру. Існує багато модифікацій цього алгоритму.

Приклад програмної реалізації

Вихідний код варіанта програмної реалізації алгоритму сортування методом бульбашки з прапорцем мовою програмування С має вигляд:

```
void sort(int *arr, int cnt)
{int Fl=0;
 for (int i = 0; i < cnt - 1; i++)
       for (int j = 0; j < cnt - i - 1; j++)
             if (arr[j] > arr[j + 1])
             {
                   int tmp = arr[j];
                   arr[j] = arr[j + 1];
                   arr[j + 1] = tmp;
                   Fl=1;
             }
             if (Fl==1) break;  }
```

Покрокове сортування методом бульбашки наведено в табл. 8.2. Для цього вхідного масиву із семи елементів алгоритм виконав 12 перестановок.

Таблиця 8.2

Покрокове виконання сортування методом бульбашки

Вхідний масив	6 15 4 20 7 13 0
Крок 1	6 **4 15** 20 7 13 0
Крок 2	6 4 15 **7 20** 13 0
Крок 3	6 4 15 7 **13 20** 0
Крок 4	6 4 15 7 13 **0 20**
Крок 5	**4 6** 15 7 13 0 20
Крок 6	4 6 **7 15** 13 0 20
Крок 7	4 6 7 **13 15** 0 20
Крок 8	4 6 7 13 **0 15** 20
...	...
Крок 12	**0 4 6 7** 13 15 20
Вихідний масив	0 4 6 **7** 13 15 20

8.4. Швидкий метод сортування

У класі алгоритмів вибірки слід відзначити так зване швидке сортування, в якому виконується така схема обмінів [5].

Є два вказівники i та j, причому на початку $i = 1, j = n.$, де n – кількість елементів масиву. Довільно вибирається за базовий будь-який елемент з масиву (перший, середній або останній). Нехай, наприклад, це буде перший елемент $X= R_1$. Встановлюємо $i=1$ (від першого), $j=n$ (від останнього), якщо знайдено $R_i >=X$ та $R_j < X$, то потрібно провести обмін $R_i <-> R_j$ за умови, що $i<j$. Після першого обміну збільшуємо i на одиницю та шукаємо $R_i >=X$. Якщо такий елемент знайдено, то j зменшуємо на одиницю і шукаємо $R_j < X$. Проводимо наступний обмін за умови, що $i<j$. Якщо $R_i >=X$ не знайдено, а $i>=j$, то першу ітерацію закінчено. Отже, алгоритм працює за принципом «спалювання свічки з обох кінців». Масив буде розділено так: $R_1, R_2, ..., R_{i-1}, R_i, R_{i+1}, ...,$Rn, причому $R_l < X, l=1,...,i-1$; $R_m <= X, m=i+1,...,n$. У лівій частині масиву стоятимуть усі елементи, що є меншими від базового, а в правій – що є більшими та сам базовий елемент. Потім до кожної з цих підмножин рекурсивно застосовується цей метод. Рекурсія закінчується, коли всі підмножини складатимуться з одного елемента або весь масив буде впорядковано.

Приклад програмної реалізації

Вихідний код програмної реалізації методу швидкого сортування мовою програмування С має вигляд:

```
int n, a[n]; //n - кількість елементів
void qs(int* a, int first, int last)
{
    int i = first, j = last, x = a[(first + last) / 2];

    do {
        while (a[i] < x) i++;
        while (a[j] > x) j--;
         if(i <= j) {
            if (i < j) swap(a[i], a[j]);
            i++;
            j--;
        }
    } while (i <= j);
     if (i < last)
        qs(a, i, last);
```

```
    if (first < j)
        qs(a, first,j);
}
```

Функція `swap(a[i], a[j])` обмінює місцями задані елементи.

Приклад:

Нехай масив містить такі елементи: 5 3 2 6 4 1 3 7

Встановлюємо i=1, j=8, X=5.

Для i=1 виконується умова 5>=5, для j=7 виконується умова 3<5. Оскільки *i<j*, то обмінюємо місцями знайдені елементи 5<->3. Повторюємо кроки для новоутвореного масиву 3 3 2 6 4 1 5 7 :для i=4, 6>5, для j=6, 1<5, *i<j*, обмінюємо місцями елементи 6<->1. Під час наступної ітерації умова *i<j* не виконається:

3 3 2 1 4 6 5 7, i=8, j=5 i>j,

Отже, перший прохід розділить масив на дві частини. Рекурсивно відсортовуємо по черзі обидві частини (табл. 8.3).

Таблиця 8.3

Покрокове швидке сортування

Вхідний масив x=5	5 3 2 6 4 1 3 7
Крок 1	**3** 3 2 6 4 1 5 7
Крок 2	3 3 2 **1** 4 **6** 5 7
Крок 3. Рекурсія: x=3	3 3 2 1 4 **1** 3 2 **3** 4
Крок 4	1 **2 3** 3 4
Крок 5. Рекурсія: x=6	6 5 7 **5 6** 7
Вихідний масив	1 2 3 3 4 5 6 7

Час роботи алгоритму швидкого сортування залежить від збалансованості, що характеризує розбиття. Збалансованість, своєю чергою, залежить від того, який елемент вибрано базовим (відносно якого елемента виконується розбиття).

- Найгірше розбиття. Найгіршою є поведінка у тому випадку, коли процедура, що виконує розбиття, породжує одну підзадачу з (n – 1) елементом, а другу – з 0 елементами. Нехай таке незбалансоване розбиття виникає під час кожного рекурсивного виклику. Для самого розбиття потрібен час $O(n)$. Тоді рекурентне співвідношення для часу роботи можна записати так:

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n).$$

Розв'язком такого співвідношення є: $T(n) = O(n^2)$.

- Найкраще розбиття. У найкращому випадку процедура поділу ділить задачу на дві підзадачі, розмір кожної з яких не перевищує (n / 2). Час роботи описується нерівністю:

$T(n) \leq 2 \cdot T(n / 2) + \Omega(n)$. Тоді: $T(n) = \Omega(n \cdot \log(n))$ – асимптотично найкращий час.

- Середній випадок. Математичне очікування часу роботи алгоритму на всіх можливих вхідних масивах є $\Theta(n \cdot \log(n))$, тобто середній випадок ближчий до найкращого.

Тобто складність методу швидкого сортування $O(n \cdot \log(n))$.

8.5. Сортування включенням

Під час сортування включенням до відсортованої множини *R* щоразу приєднується один елемент [5], а саме: із невідсортованої вхідної множини *M* вибирається довільний елемент і розміщується у вихідну множину R.

<u>Алгоритм V</u>

Нехай задано множину елементів M, |M|=n.

V1. k =1; повторювати кроки V2, V3, доки k<n.

V2. Вибір довільного елемента x вхідної множини M : x=M(k).

V3. Розміщення x у вихідній множині R так, щоб вона залишалась відсортованою.

V4. Кінець. Вихід.

Вихідну множину R при кожному включенні можна відсортовувати відомим методом сортування, наприклад, методом простої вибірки. Майже всі методи сортування включенням у найгіршому випадку вимагають порядку n^2 порівнянь, тому їх застосування пов'язане з деяким ризиком. Є багато варіантів цього методу сортування.

Розглянемо **метод Шелла** [5]. Його суть полягає в тому, що на кожному кроці групуються та сортуються елементи, що стоять один від одного на певній відстані h. Потім ця відстань зменшується на крок, що дорівнює степеню два. На останньому кроці йде звичайне одинарне сортування. Перша відстань вибирається відносно кількості елементів у масиві, поділеної на 2.

Приклад програмної реалізації

Вихідний код програмної реалізації методу Шелла мовою програмування С має вигляд:

```
int n, a[n];
void shellSort(int *a, int n)
 {
  int j;
  int step = n / 2;
  while (step > 0)
  {
    for (int i = 0; i < (n - step); i++)
    {
     j = i;
     while ((j >= 0) && (a[j] > a[j + step]))
      {
       int tmp = a[j];
       a[j] = a[j + step];
       a[j + step] = tmp;
       j--;
      }
    }
    step = step / 2;
  }
 }
```

Покрокове сортування методом Шелла наведено в табл. 8.4. Для цього вхідного масиву з восьми елементів алгоритм виконав 3 зміни кроку.

Таблиця 8.4

Покрокове сортування методом Шелла

Вхідний масив	44 55 12 42 94 18 06 47
Крок 1 h = 8/2 = 4	44 **18** 12 42 94 **55** 06 47
	44 18 **06** 42 94 55 **12** 47
Крок 2 h = 4/2 = 2	**06** 18 **12** 42 **44** 55 **94** 47
	06 *18* 12 *42* 44 *47* 94 *55*
Крок 3. h = 2/2 = 1	06 18 12 42 44 47 94 55
	06 **12 18** 42 44 47 94 55
	06 12 18 42 44 47 **55 94**
Вихідний масив	06 12 18 42 44 47 55 94

Час роботи залежить від вибору значень h. Існує декілька підходів до вибору цих значень:

- При виборі $h_1 = \left\lfloor \frac{n}{2} \right\rfloor$, $h_2 = \left\lfloor \frac{h_1}{2} \right\rfloor$, $h_3 = \left\lfloor \frac{h_2}{2} \right\rfloor$, ..., $h_m = 1$ час роботи алгоритму, в найгіршому випадку, становить $O(n^2)$.
- Якщо h – впорядкований за спаданням набір чисел вигляду $(2^i - 1) < n, j \in n$, то час роботи є $O(n^{1.5})$.
- Якщо h – впорядкований за спаданням набір чисел вигляду $2^i * 3^j < n/2,\ i, j \in n$, то час роботи є $O(n \cdot \log^2 n)$.

8.6. Сортування розподілом

Методи сортування розподілом являють собою природне узагальнення ручних методів сортування. Вони передбачають розбиття вхідної множини елементів M на p підмножин $M_1, ..., M_p$, для яких виконуються умови $M_1 < M_2 < ... < M_p$.

Алгоритм R *має рекурсивну структуру (позначено квадратними дужками)*

Нехай задано множину M *елементів, яку розбито на* p *підмножин* M_i

R1. Якщо $|M|=0$ *або* $|M|=1$*, кінець.*

R2. Інакше [Розбиття M *на* $M_1 ... M_p$ *так, щоб* $M_1 < M_p$*;* $i=1$*, виконувати, доки* $i<p$ *[Сортування розподілом підмножини* M_i*;* $i=i+1$*]].*

R3. Кінець. Вихід.

Одну з версій цього алгоритму називають **цифровим,** або порозрядним сортуванням за аналогією з системою числення [5]. На практиці механічне сортування починається з молодшого розряду і просувається в напрямку до старшого, при цьому відсортовані підмножини послідовно об'єднуються і перерозподіляються. Приклад цифрового сортування наведено в табл. 8.5.

Цифрове сортування не використовує порівняння елементів, тому не можна оцінювати його в звичайних термінах. Якщо елементи містять m цифр, то потрібно виконати m проходів, і якщо на кожному проході розподіляється n елементів, то основні кроки виконуються $m*n$ разів. Цифрове сортування зручно використовувати для рядків символів.

Таблиця 8.5

Покрокове цифрове сортування

Вхідний масив	Сортування «одиниць»	Сортування «десятків»	Сортування «сотень»
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

8.7. Сортування злиттям або об'єднанням

Методи цього класу працюють за таким принципом: невідсортовану вхідну множину довільно розбивають на підмножини $M_1,..., M_p$. Потім ці підмножини сортують окремо цим самим методом і об'єднують в одну відсортовану множину [5]. В цьому методі діє відомий принцип «розділяй та пануй». В літературі цей метод називається MergeSort.

Продемонструємо цю схему на двох множинах, оскільки багаторазове злиття можна здійснювати багаторазовим попарним злиттям.

Нехай маємо дві відсортовані множини X і Y; потрібно об'єднати їх у множину Z, яка також повинна бути відсортованою. У ролі Z_1 приймаємо $min(x_1, y_1)$: якщо $Z_1 = x_1 \in X$, тоді $Z_2 = min(x_2, y_1)$ і т.д.

Для запису алгоритму використовуємо таке позначення: i – індекс для множини X, j – індекс для множини Y, k – індекс для множини Z, $i=1..n; j=1..m; k=1..(n+m)$.

<u>Алгоритм С</u>

Дано $X=\{ x_1 ,..., x_n \}; Y=\{ y_1 ,..., y_m \}$.
С1. Ініціалізація індексів $i=1, j=1, k=1$;
С2. Виконувати С3, С4 доки $k<n+m$.
С3 [Якщо $x_i < y_i$ *то* $[z_k = x_i; i=i+1]$ *інакше* $[z_k=y_j, j=j+1]$
С4. $k=k+1]$
С5. Кінець. Вихід.

У квадратних дужках записано рекурсивні дії.

Кількість порівнянь, яку необхідно виконати в алгоритмі **С** для злиття двох множин, дорівнює $|x|+|y|=n+m$.

Уся процедура злиття разом потребує не більше ніж n порівнянь для n елементів і потрібно буде виконати $log_2 n$ переходів із однієї множини у другу. Тобто алгоритм C потребує $n \ log_2 \ n$ порівнянь.

Час роботи алгоритму злиття $T(n)$ для n елементів задовольняє рекурентне співвідношення: $T(n) = 2 \cdot T(\cdot n/2) + O(n)$, де $T(\cdot n/2)$ – час на впорядкування половини масиву, $O(n)$ – час на злиття цих половинок. Враховуючи, що $T(1) = O(1)$, розв'язком співвідношення є: $T(n) = \mathbf{O(n \cdot log(n))}$.

Приклад програмної реалізації

Вихідний код програмної реалізації етапу злиття мовою програмування С виглядає так:

```
int n = 0, l1, l2;
int m1[l1], m2[l2], ret[l1+l2];
// зливаемо масиви, поки один не закінчиться
while (l1 && l2)
{
    if (*m1 < *m2)
    {
        ret[n] = *m1;
        m1++;
        l1--;
    }
    else
    {
        ret[n] = *m2;
        m2++;
        l2--;
    }
    n++;
}
// Якщо закінчився перший масив
if (l1 == 0)
{
    for (int i=0; i<l2; i++)
    {
        ret[n++] = *m2++;
    }
}
// Якщо закінчився другий масив
else
{
    for (int i=0; i<l1; i++)
    {
        ret[n++] = *m1++;
    }
}
```

У табл. 8.6 наведено приклад сортування масиву методом злиття.

Таблиця 8.6

Покрокове сортування методом злиття

Вхідний масив	5 3 2 6 4 1 3 7
Крок 1: сортування пар	**3 5** 2 6 **1 4** 3 7
Крок 2: злиття пар у четвірки	**3 5** **2 6** **2 3 5 6** *1 4* *3 7* *1 3 4 7*
Крок 3. Злиття четвірок	**2 3 5 6** *1 3 4 7* 1 2 3 3 4 5 6 7
Вихідний масив	1 2 3 3 4 5 6 7

8.8. Сортування підрахунком

Сортування підрахунком (англійською «Counting Sort») – алгоритм впорядкування, що застосовується за малої кількості різних елементів у масиві даних. Час його роботи лінійно залежить як від загальної кількості елементів у масиві, так і від кількості різних елементів.

Ідея алгоритму є такою: спочатку підрахувати, скільки разів кожен елемент трапляється в вихідному масиві. За цими даними можна одразу з'ясувати на якому місці має стояти кожен елемент, а потім за один прохід поставити всі елементи на свої місця.

В алгоритмі наявні тільки прості цикли довжини n (довжина масиву) та один цикл довжини k (величина діапазону). Отже, обчислювальна складність роботи алгоритму становить $\boldsymbol{O(n + k)}$.

В алгоритмі використовується додатковий масив. Тому алгоритм потребує $E(k)$ додаткової пам'яті.

За такою реалізації алгоритм є стабільним. Саме ця його властивість дає змогу використовувати його як частину інших алгоритмів сортування (наприклад, сортування за розрядами). Використовувати цей алгоритм доцільно тільки у випадку малих k.

Приклад програмної реалізації

Вихідний код програмної реалізації сортування методом підрахунку мовою програмування С має вигляд:


```
void sort(short int *a, int n)
{
      int min = a[0],
            max = a[0];
// визначення діапазону значень у масиві a
      for (int i = 1; i < n; i++)
            if (a[i] < min)
                  min = a[i];
            else if (a[i] > max)
                  max = a[i];
// формування додаткового масиву b
      int *b = (int*)calloc(max - min + 1,
sizeof(int));
      for (int i = 0; i < n; i++)
            b[a[i] - min]++;
// формування вихідного відсортованого масиву a
      for (int i = min, j = 0; i <= max; i++)
            while (b[i - min]-- > 0)
                  a[j++] = i;
      free(b);
}
```

Приклад

Нехай масив А містить такі елементи: 3 3 4 6 2 2 4 3 1 1 6 1 1 . Тоді n=13, k=5. Сформуємо додатковий масив В розміром k, у якому зафіксуємо кількість однакових елементів: 4 2 3 2 0 2 (4 – це кількість одиниць у вхідному масиві, 2 – кількість двійок, п'ятірки не траплялись взагалі, тому їх кількість дорівнює 0 і т.д.).

За цими даними формуємо вихідний відсортований масив: оскільки одиниць у вхідному масиві було чотири, то у вихідний масив А записуємо чотири одиниці, потім дві двійки, три трійки, дві четвірки і дві шестірки: 1 1 1 1 2 2 3 3 3 4 4 6 6.

Таблиця 8.7

Покрокове сортування методом підрахунку

Вхідний масив А	3 3 4 6 2 2 4 3 1 1 6 1 1
Додатковий масив В	4 2 3 2 0 2
Вихідний масив	1 1 1 1 2 2 3 3 3 4 4 6 6

Контрольні запитання та вправи

1. Які з наведених алгоритмів вимагають не менше n^2 порівнянь?
2. Чи є стійким сортування методом простої вибірки?
3. Для яких вхідних даних алгоритм бульбашки не є ефективним?
4. Який елемент масиву можна вибрати як базовий в алгоритмі швидкого сортування?
5. Чому дорівнює відстань для вибору елементів за методом Шелла?
6. Для яких вхідних даних цифровий алгоритм є ефективним?
7. Який з наведених алгоритмів сортування має лінійну складність? У чому його суть?
8. Відсортуйте швидким методом послідовність символів: F, A, C, B, G, M, E, K, I.
9. Побудуйте функцію сортування методом злиття списку студентів групи, в якій є однофамільці.
10. Виконайте трасування алгоритму цифрового сортування для послідовності:

BABA, AAAB, BAAB, BBAA, AAAA, ABAA, BBBB, BAAA.

Приклади тестових питань

1. Як проглядаються елементи масиву в методі швидкого сортування?

а) з початку;
б) з кінця;
в) з обох боків одночасно;
г) з середини.

2. За методом бульбашки порівнюють такі елементи масиву:

а) сусідні;
б) відділені один від одного з певним кроком;
в) крайні.

3. За методом сортування Шелла порівнюються такі елементи масиву:

а) сусідні;
б) відділені один від одного з певним кроком;
в) крайні.

4. В алгоритмі сортування злиттям вхідний масив має бути:

а) розбитий на частини;
б) закодований.

5. Для реалізації методу швидкого сортування використовується такий засіб програмування:

а) сканування масиву;
б) пошук максимального значення;
в) рекурсію.

6. Складність методу сортування підрахунком становить:

а) O(n+m);
б) O(n);
в) $O(n^2)$.

7. Складність алгоритму сортування методом простої вибірки вища за складність методу бульбашки:

а) так;
б) ні.

9. ПОШУК ДАНИХ

Задача пошуку виникла пізніше, ніж задача сортування. Поява її спричинена переважно появою автоматизованих інформаційних систем *АІС*, побудованих на основі комп'ютерів.

Розглянемо звичайну пошукову задачу: як знаходити дані, що зберігаються в пам'яті комп'ютера за певною ідентифікацією.

Наприклад, в обчислювальній задачі потрібно знайти функцію $f(x)$, маючи значення x і таблицю значень функції $f(x)$; у лінгвістичних задачах може цікавити англійський еквівалент відповідного слова.

Припускаємо, що зберігається множина з n записів і необхідно визначити місцезнаходження відповідного запису. Розглянемо ряд алгоритмів пошуку і наведемо їхні короткі характеристики, а також рекомендації застосування.

9.1. Послідовний пошук

Найпростішим методом пошуку певного запису, що знаходиться у невпорядкованій таблиці, є послідовний перегляд кожного запису, доки не буде виявлено збігу з шуканим. Такий метод називають послідовним, або лінійним пошуком.

<u>Алгоритм L</u>

Таблиця містить n записів R_1 ,..., R_n з ключами k_1 ,..., k_n . Необхідно знайти запис із заданим ключем k.

L1. Ініціалізація індексу проходження таблиці: i=1.

L2. Якщо $k=k_i$ – кінець успішний, перехід на крок L5; якщо ні, перехід на L3.

L3. Зміна індексу i =i+1.

L4. Перевірка умови i<n? Так: перехід на L2. Ні – кінець неуспішний.

L5. Вихід

Ефективність алгоритму можна оцінити, підрахувавши кількість виконаних порівнянь тих значень ключів, які беруть участь у пошуку. Середня кількість таких порівнянь дорівнює величині n. Тобто асимптоматична складність алгоритму $O(n)$. Через малу ефективність порівняно з іншими алгоритмами лінійний пошук зазвичай використовують лише тоді, коли відрізок пошукової системи містить дуже мало елементів, однак лінійний пошук не потребує додаткової пам'яті або аналізу функції, так що може працювати в потоковому режимі під час безпосереднього отримання даних з будь-якого джерела. Також лінійний пошук часто використовується у вигляді лінійних алгоритмів пошуку максимуму (мінімуму).

Приклад програмної реалізації

Вихідний код програмної реалізації алгоритму лінійного пошуку мовою програмування С виглядає так:

```
int* search(int *arr, int cnt, int req)
{
 for (int i = 0; i < cnt; i++)
       if (arr[i] == req)
             return i;
 return NULL;
}
```

Результатом роботи послідовного алгоритму пошуку є значення індексу шуканого елемента в заданому масиві або повідомлення про його відсутність у заданому масиві. У табл. 9.1 наведено приклад роботи послідовного алгоритму пошуку.

Таблиця 9.1

Покрокове виконання послідовного алгоритму пошуку

Вхідний масив А: Шуканий елемент:	1 3 5 7 9 11 13 15 7
Крок 1:	**<u>1</u>** 3 5 7 9 11 13 15
Крок 2:	1 **<u>3</u>** 5 7 9 11 13 15
Крок 3.	1 3 **<u>5</u>** 7 9 11 13 15
Крок 4	1 3 5 **<u>7</u>** 9 11 13 15
Індекс знайденого елемента	4

9.2. Двійковий пошук

Іншим відносно простим методом доступу до таблиць є метод бінарного пошуку. Двійковий пошук можливий у таблицях, організованих як дерева порівнянь. Елементи в таких таблицях зберігаються за лексикографічною (тобто алфавітною) послідовністю, або за зростанням числових значень ключів. Отже, **необхідною умовою** роботи алгоритму двійкового пошуку є впорядкованість вхідних даних.

Загальна стратегія організації дерева полягає у поділі таблиці на підтаблиці або сукупності елементів на підмножини елементів. Тоді процедура пошуку елемента в дереві порівнянь нагадує пошук імені в телефонному довіднику. Для відшукання елемента потрібно спочатку вирішити, в якій підмножині знаходиться елемент, знайти цю підмножину і після цього продовжувати пошук.

Розглянемо спочатку, як можна записати дерево до векторної пам'яті.

Нехай ключі зберігаються у векторі *КЛЮЧ*(*i;j*). Елементи у векторній пам'яті повинні бути впорядкованими за значенням ключів.

Під час пошуку певного елемента зробимо коренем дерева вмістиме *КЛЮЧ*(*m*), де $m=[(i+j)/2]$ – найбільше ціле, яке менше або дорівнює $(i+j)/2$. Тоді ліве піддерево розміщується у векторі *КЛЮЧ*(*i;m*-1), а праве – у векторі *КЛЮЧ*(*m*+1;*j*). Цей процес повторюється, доки не буде знайдено потрібний елемент, тобто доки процес не зійдеться у одній вершині, для якої обидва індекси у векторі *КЛЮЧ* матимуть однакові значення.

Тоді алгоритм *D* пошуку ключа *K* у векторі *КЛЮЧ*(*i;j*) можна записати такою послідовністю кроків:

Алгоритм D

На вхід надходить відсортований вектор КЛЮЧ(i;j)

D0. Ініціалізація індексів i, j .

D1. Повторювати кроки D2 – D5 доти, доки i<j.

D2. Обчислення індекса кореня дерева: m=[(i+j)/2].

D3. Якщо [КЛЮЧ (m)]= K, то REZ= m, перехід на крок D6.

D4. Інакше: якщо [КЛЮЧ (m)] <K,

то i=m +1 (пошук праворуч); перехід на D2.

D5. Інакше j=m-1 (пошук ліворуч); перехід на D2;

D6. Вихід.

Основне завдання полягає у вилученні на кожному кроці з подальшого пошуку якомога більшої кількості елементів. Оптимальним рішенням буде вибір середнього елемента, оскільки в такому разі буде вилучено половину елементів. Максимальна кількість порівнянь дорівнює $log_2\ n$. Отже, складність алгоритму двійкового пошуку $\boldsymbol{O(log_2\ n)}$, де *n* – кількість елементів даних.

Приклад програмної реалізації

Вихідний код програмної реалізації алгоритму двійкового пошуку мовою програмування C має вигляд:

```
int Binsearch (int *A, int N, int x)
{
int L = 0, R = N, m, flag = 0;
while ( L < R ) {
   m = (L + R) / 2;
   if ( A[m] == x ) {
      flag = 1;
   break;
      }
if ( x < A[m] ) R = m;
   else  L = m + 1;
}
if ( flag )
     return m;
else return NULL;
}
```

У табл. 9.2 наведено приклад пошуку значення 13 у заданому відсортованому масиві методом двійкового пошуку.

Таблиця 9.2

Покрокове виконання методу двійкового пошуку

Вхідний масив А: Шуканий елемент:	1 3 5 7 9 11 13 15 13
Крок 1: A[m]= 7, 13>7	9 11 13 15
Крок 2: A[m]= 11, 11< 13	13 15
Крок 3. A[m]=13	
Індекс знайденого елемента	8

9.3. Прямий пошук рядка

Нехай задано масив *S* з *n* елементів та масив *P* з *m* елементів, *m*<=*n*. Необхідно знайти перше входження масиву *P* у масив *S* . Алгоритм зводиться до повтору порівнянь окремих елементів.

Зазвичай масив *S* вважають деяким текстом, а масив *P* словом, і необхідно знайти перше входження цього слова у вказаному тексті.

<u>Алгоритм R</u>

R1. Встановити i=1.. n-m, j=1..m.

R2. Якщо S[i] = P[j], то зафіксувати перший збіг k=i та перевірити збіг усього масиву P у масиві S. За першого незбігу відмінити значення k та продовжити пошук, i++.

R3. Кінець.

Кількість порівнянь дорівнює $n \times m$. Цей алгоритм працює достатньо ефективно, якщо припустити, що незбіг пари символів відбувається достатньо часто. Можна припустити, що для текстів, складених зі 128 символів, незбіг виявлятиметься після однієї або двох перевірок. Проте, у гіршому випадку, продуктивність може виявитися набагато нижчою.

Приклад програмної реалізації

Вихідний код програмної реалізації алгоритму прямого пошуку рядка мовою програмування C має вигляд:

```
int straightFind(S, n, P, m)
{
  int i, j;
  i = -1;
  do
   {
    j = 0;
    i++;
    while( (j < m) && (S(i + j) == P(j)) ) j++;
    } while( j<m && i<n-m );
   if (j == m)
{
     return i;
    }
    else return -1;
}
```

У табл. 9.3 наведений приклад пошуку рядка P у тексті S. Символи, які збігалися, виділено жирним шрифтом. Крок зсуву P за S дорівнює 1.

Таблиця 9.3

Покрокове виконання алгоритму прямого пошуку рядка

Вхідні рядка S: P:	a c a a b c a b c
Крок 1	**a** c a a b c **a** b c
Крок 2	a c a a b c a b c
Крок 3	a c **a** a b c **a** b c
Крок 4	a c a **a b c** **a b c**
Перше входження масиву P в масив S	4

9.4. Алгоритм Кнута, Моріса і Прата пошуку в рядку

На вхід надходять два масиви символів: *S* розміром *n* (текст) та *P* розміром *m* (слово). Необхідно знайти перше входження слова у тексті. Схема алгоритму полягає у поступовому порівнянні слова з текстом та у разі знайденого незбігу зсуву по тексту на крок, що дорівнює різниці кількості збіжних символів та значення відповідної префікс-функції. Алгоритм використовує просте спостереження, що коли є незбіг тексту і слова, то слово містить достатньо інформації для того, щоб визначити, де наступне входження може початися, пропускаючи кількаразову перевірку попередньо порівняних символів. Попередньо досліджується слово та визначається префікс-функція.

Префікс-функція рядка – це довжина найбільшого префікса рядка S[1..i], який не збігається з цим рядком і одночасно є її суфіксом. Простіше кажучи, це довжина найдовшого початку рядка, що одночасно є її кінцем. Позначимо таку функцію D(S,i).

Наприклад, для рядка «ababcaba» визначення префікс-функції наведено в табл. 9.4.

Таблиця 9.4

Визначення префікс-функції

Суфікс	**Префікс**	**D(S,i).**
a	a	0
ab	ab	0
ab**a**	**a**ba	1
ab**ab**	**ab**ab	2
ababc	ababc	0
ababc**a**	**a**babca	1
ababc**ab**	**ab**abcab	2
ababc**aba**	**aba**bcaba	3

За відсутності однакових префіксів і суфіксів D(S,i)=m.

Алгоритм КМП

КМП 1. Для заданого слова визначити префікс-функцію D

КМП 2. Встановити i=0, j=0, d=1.

КМП 3. Поки j<m, i<n

Перевірка: якщо S[i]=P[j], то d++, i++,j++ поки d != m.

КМП 4. Інакше встановити зсув слова i=i+d-D(d). Перейти на крок КМП 3.

КМП 5. Кінець.

Через те, що дві складові алгоритму мають складності, відповідно, ***O(n)*** і ***O(m)***, складність всього алгоритму становить ***O(n + m)***.

Приклад програмної реалізації

Вихідний код програмної реалізації алгоритму Кнута, Моріса і Прата пошуку рядка мовою програмування С має вигляд:

```
int Find_KMP (char s[], char p[])
{
        int i, j, N, M;
        N = strlen(s);
        M = strlen(p);
        int *d =(int*)malloc(M*sizeof(int)); /*
динамічний масив довжиною M для результатів префікс-
функції   */
        /* Обчислення префікс-функції */
        d[0]=0;
        for(i=1,j=0;i<M;i++)
        {
                while(j>0 && p[j]!=p[i])
                        j = d[j-1];
                if(p[j]==p[i])
                        j++;
                d[i]=j;
        }
        /* пошук */
            i=0;j=0;
            while ( i<N-j)
              {
                while(j>0 && p[j]!=s[i])
                               i=i+j-d[j-1]-1;
                if(p[j]==s[i])
                        j++;
                if (j==M)
                {
                        return i-j+1;
                }
            }
        return -1;
}
```


У табл. 9.5 наведено приклад пошуку рядка P у тексті S за алгоритмом КМП. Символи, які збіглися, виділено жирним шрифтом. Крок зсуву P за S визначається за префікс-функцією h=d-D(d), де d – кількість символів тексту, які збіглися, і слова при знайденому незбігу.

Таблиця 9.5

Покрокове виконання алгоритму КМП

Вхідні рядки S: P: Префікс-функція D:	a b a b a a b a b a c a a b a b a c a 0 0 1 2 3 0 1
Крок 1 d=5, D(5)=3, h=5-3=2	**a b a b a** a b a b a c a **a b a b a** c a
Крок 2 d=3, D(3)=1, h=3-1=2	a b **a b a** a b a b a c a **a b a** b a c a
Крок 3 d=1, D(1)=0, h=1-0=1	a b a b **a** a b a b a c a **a** b a b a c a
Крок 4 d=7, d=m,	a b a b a **a b a b a c a** **a b a b a c a**
Перше входження масиву P у масив S	6

9.5. Алгоритм Бойера – Мура пошуку в рядку

На жаль, збіги відбуваються значно рідше, ніж незбіги. Тому виграш від використання алгоритму КМП переважно незначний. Інший алгоритм – Бойера–Мура – оснований на схемі: порівняння символів починається з кінця слова, а не з початку. Нехай для кожного символу x слова d_x – відстань від першого правого у слові

входження x до кінця слова. Припустімо, знайдено незбіг між словом та текстом на символі x у тексті. Тоді слово можна посунути праворуч на d_x позицій, що є більше або дорівнює 1. Якщо x у слові взагалі не зустрічається, то посунути слово можна зразу на всю його довжину m.

У цьому алгоритмі символ x розглядається як **стоп-символ** – це є символ у тексті, який є першим незбігом тексту і слова під час порівняння праворуч (з кінця слова). Розглянемо три можливі ситуації:

1. Стоп-символа у слові взагалі не має, тоді зсув дорівнює довжині слова m.

2. Крайня права позиція k входження стоп-символа у слові є меншою від його позиції j у тексті. Тоді слово можна зсунути праворуч на k-j позицій так, щоб стоп-символ у слові і тексті опинились один під одним.

3. Крайня права позиція k входження стоп-символа у слові є більшою від його позиції j у тексті. Тоді, якщо у слові є ще один такий самий символ, то необхідно зсунути слово до збігу цього символу з символом у тексті, інакше зсув дорівнює 1.

Для оптимізації пошуку пропонується попередньо сформувати таблицю стоп-символів, розмір якої відповідає кількості літер алфавіту, що використаний у вхідному тексті. Початково всім елементам таблиці надається значення m – кількість символів шуканого слова, для реалізації першої з можливих ситуацій зі стоп-символом.

У найгіршому випадку алгоритм Бойера–Мура потребує n порівнянь, де n – кількість символів у тексті. У найкращих обставинах, коли останній символ слова завжди не збігається з символом тексту, кількість порівнянь дорівнює $\boldsymbol{n / m}$, де m – кількість символів слова. Отже, складність алгоритму Бойера–Мура $\boldsymbol{O(n / m)}$.

Приклад програмної реалізації

Вихідний код програмної реалізації алгоритму Бойера–Мура пошуку рядка мовою програмування С має вигляд:

```
// Формування таблиці стоп-символів
int w, j, m;
for (w = 1; w<[illegible]; w++)
   skip[w] = m;
for (j = 1; j<m; j++)
   skip[x[j]] = m - j;
```



```
// Алгоритм Бойера-Мура
int BM (*S, *P, n, m, *skip)
{
i = m;
j = m;
while (j > 0 && i<=n)
{
   if (S[i] == P[j])
      i = i - 1;
      j = j - 1;
   else
// незбіг
      i = i + skip[S[i]];
      j = m;
}
if (j < 1)
   return (i+1);
else
   return (0);

}
```

У табл. 9.6 наведено приклад пошуку рядка P в тексті S за алгоритмом Бойера–Мура. Жирним шрифтом виділено стоп-символ.

Таблиця 9.6

Покрокове виконання алгоритму Бойера–Мура

Вхідні рядки S: P:	there they are they
Крок 1: Стоп-символ - **r** ситуація 1	the**r**e they are they
Крок 2 Стоп-символ - **h** ситуація 2	there t**h**ey are they
Крок 3	there they are they
Перше входження масиву P до масиву S	7

9.6. Пошук у таблиці

Пошук в масиві іноді називають пошуком у таблиці, особливо якщо ключ сам є складовим об'єктом, таким як масив чисел або символів. Для того, щоб встановити факт збігу, необхідно переконатись, що всі символи рядків, які порівнюються, збігаються. Порівняння зводиться до пошуку їх незбігу, тобто пошуку на нерівність. Якщо нерівних частин не існує, то можна говорити про рівність. Якщо розмір шуканого взірця невеликий, то доцільно використати алгоритм лінійного пошуку. У протилежному випадку необхідно впорядкувати таблицю та скористатись алгоритмом пошуку діленням навпіл. Від вибраного алгоритму пошуку залежитиме алгоритмічна складність.

9.6.1. Пошук у таблицях з обчислюваними адресами

Розглянемо метод, за яким тривалість пошуку не залежатиме від кількості записів у таблиці. Для цього таблиці треба організовувати так, щоб місцезнаходження її елементів визначалося за допомогою обчислюваної адреси. Це так звані таблиці з обчислюваними адресами. Загальна ідея організації таких таблиць полягає у тому, щоб використовувати ключ безпосередньо як адресу або визначати адресу за допомогою функції від ключа і запам'ятовувати запис за цією адресою.

Функції, що відображають ім'я або ключ у деяке ціле число, називають функціями **хешування** *h(k)* для ключа *k*. Їх також називають функціями перемішування або рандомізації.

Ситуацію, за якої $h(k_i)= h(k_j)$ при $k_i != k_j$, називають **колізією**.

Таблиці з обчислюваними адресами поділяються на таблиці з прямим або безпосереднім доступом і перемішані, або хеш-таблиці. Задача колізії існує лише в хеш-таблицях.

1. Пошук у таблицях з прямим доступом

Таблицю називають таблицею з **прямим**, або безпосереднім **доступом**, якщо для визначення місцезнаходження кожного запису використовується його ключ. Функція хешування визначається як відображення *K->A* , де *K* – множина ключів, які можуть ідентифікувати записи в таблиці з прямим доступом; *A* – множина адрес. Доступ до запису за ключем *k* здійснюється в такому випадку за значенням функції ***h(k)***.

Мірою використання пам'яті в таблицях з прямим доступом є коефіцієнт заповнення *q*, що визначається як відношення кількості записів *n* до кількості місць *m* у таблиці: ***q=n/m***.

Вибір функції адресації, що забезпечує взаємну однозначність перетворення ключа на адресу її зберігання, взагалі є доволі складним завданням. На практиці його можна вирішити тільки для постійних таблиць із заздалегідь відомим набором значень ключа. Такі таблиці застосовуються в трансляторах (наприклад, таблиця символів вхідної мови є таблицею з прямим доступом).

Наведемо **приклад** таблиці з прямим доступом. Якщо маємо *n* ключів, то кожному заданому ключу *k* може відповідати адреса запису в діапазоні 0... *n*-1.

Нехай ключами записів є такі шифри автомобільних номерів (табл. 9.7).

Таблиця 9.7

Побудова таблиці з прямим доступом

Ключ	Коди букв	Адреса
AN	0, 13	13
AX	0, 23	23
BC	1, 2	28
BI	1, 8	34
BJ	1, 19	35
CM	2, 12	64
JR	9, 17	251
MO	12, 14	326

Тут використано одну з можливих хеш-функцій, що ґрунтується на алгебраїчному кодуванні. Функція адресації обчислюється залежно від коду ключа. Наприклад, тіло ключа MO запам'ятовується за адресою 12 × 26 + 14 = 326; де 12 – код літери M, 14 – код літери O. Тіло ключа CM за адресою 2 × 26 + 12 = 64. За такою функцією адресації загалом буде 26^2 можливих адрес (26 – кількість букв латинського алфавіту). З цього прикладу видно, що пошук у таких таблицях найшвидший, оскільки безпосередньо потрапляємо в потрібне місце без порівняння з іншими ключами. Недоліком є неефективне використання пам'яті.

2. Пошук у хеш-таблицях

Природним розвитком застосування таблиць з безпосереднім доступом є обчислення адреси не в повному діапазоні 0...*n*-1, а в деякому обмеженому. Цей метод відомий як метод перемішаних адрес, або метод змістовної адресації.

У простій таблиці з безпосереднім доступом з *n* можливих ключів можна згенерувати унікальну адресу в діапазоні 0... *n*-1. Але

якщо з'являється тільки $l<<n$ ключів, то втрачається великий обсяг пам'яті. Отже, доцільно обчислювати адресу в діапазоні $l < m<<n$, однак це збільшить ймовірність появи колізії.

Розглянемо приклад з табл. 9.7. Для восьми елементів відведемо лише 10 одиниць пам'яті, тобто $l = 8$, $m = 10$.

Наприклад, для побудови хеш-таблиці обчислюємо хеш-адресу від ключа в діапазоні 0...9 так: складемо всі десяткові коди обох символів, що утворюють ключ, а потім візьмемо залишок від ділення цієї суми на 10 (ділення за модулем 10). Значення такої хеш-функції наведено в табл. 9.8, а в табл. 9.9 – побудова хеш-таблиці за цією функцією. Для ліквідації ситуації колізії застосовувалося повторне хешування.

Таблиця 9.8

Обчислення хеш-адреси

№	Ключ	Коди букв	Адреса
1	AN	0 13	3
2	AX	0 23	3
3	BC	1 2	3
4	BI	1 8	9
5	BJ	1 19	0
6	CM	2 12	4
7	JR	9 17	6
8	MO	12 14	6

Таблиця 9.9

Побудова хеш-таблиці

Адреса	Ключ
0	BJ
1	
2	
3	AX
4	CM
5	AN
6	MO
7	JR
8	BC
9	BI

Для фіксованих таблиць можна знайти достатньо ефективну функцію адресації. Якщо ж таблиці не фіксовані і всі можливі ключі вибираються з деякої достатньо великої множини потенційно можливих ключів, то ідеальної функції адресації не існує.

Довжина пошуку в хеш-таблицях залежить від коефіцієнта заповнення адресного простору, тобто відношення $q=n/m$, де n – кількість записів; m – розмірність таблиці, а також методу розв'язання задачі колізії.

Контрольні запитання

1. У чому суть задачі пошуку?
2. Які алгоритми з наведених мають лінійну складність?
3. Якою має бути вхідна множина для алгоритму двійкового пошуку?
4. У чому перевага алгоритму КМП над алгоритмом прямого пошуку рядка?
5. У чому полягає суть попереднього дослідження взірця в алгоритмі КМП?
6. Які розглядаються варіанти стоп-символа в алгоритмі Бойера–Мура?
7. Що називається функцією хешування?
8. Що називається колізією?
9. Як визначається адреса запису в таблиці з прямим доступом?
10. Як визначається адреса запису в хеш-таблиці?

Приклади тестових питань

1. Взірець порівнюють з текстом за пошуковим методом КМП:
   а) з кінця взірця;
   б) з початку взірця;
   в) з середини тексту.

2. Взірець порівнюють з текстом за пошуковим методом Бойера–Мура:

а) з кінця взірця;

б) з початку взірця;

в) з середини тексту.

3. В алгоритмі бінарного пошуку вхідний масив має бути:

а) розбитий на частини;

б) відсортований.

4. Алгоритм пошуку КМП передбачає пошук збігів чи незбігів символів тексту і взірця:

а) збігів;

б) незбігів.

5. Алгоритм пошуку Бойера–Мура передбачає пошук збігів чи незбігів символів тексту і взірця:

а) збігів;

б) незбігів.

6. Необхідно визначити стоп-символ в алгоритмі пошуку:

а) КМП;

б) Бойера–Мура.

7. Попередньо необхідно визначити максимальний зсув в алгоритмі пошуку:

а) КМП;

б) Бойера–Мура.

8. Складність алгоритму пошуку КМП O(n+m):

а) так;

б) ні.

9. Яка складність алгоритму пошуку Бойера–Мура?

а) $O(n/m)$;

б) $O(n)$;

в) $O(n^2)$.

10. Складність алгоритму послідовного пошуку квадратично залежить від кількості вхідних даних:

а) так;

б) ні.

11. Алгоритм Боейра–Мура починає пошук з початку взірця:

а) так;

б) ні.

12. Алгоритм КМП починає пошук з початку взірця:

а) так;

б) ні.

13. Функція від значення ключа запису в таблиці називається функцією хешування:

а) так;

б) ні.

14. Колізія – це ситуація рівності значень функції хешування для різних значень ключів:

а) так;

б) ні.

15. Поняття «колізія» пов'язано зі структурою даних:

а) список;

б) черга;

в) таблиця.

16. Поняття «функція хешування» пов'язано зі структурою даних:

а) список;

б) черга;

в) таблиця.

10. ТИПИ ДАНИХ У МОВІ С + +

10.1. Класифікація та оголошення типів даних

Усі типи, які використані в С + +, можна поділити на чотири групи:

Таблиця 10.1

Групи типів даних у С++

Aggregate		Структури даних
	Array	масиви
	Struct	структури
	Union	об'єднання
	Class	класи
Function		функції
Scalar		скалярні
	Arithmetic	арифметичні
	Enumeration	перелічені
	Pointer	показники
	Reference	посилання
Void		відсутність значень

Інший спосіб класифікації типів пов'язаний з їх розбиттям на основні і похідні типи. До основних належать:

void, char, int, float, double,

а також їхні варіанти з модифікаторами

short (короткий),

long (довгий),

signed (зі знаком),

unsigned (без знака).

Наприклад, **unsigned char**, **unsigned int**, **signed int** (модифікатор **signed** мається на увазі за замовчуванням і тому зазвичай не вказується).

Похідні типи містять вказівники та посилання на типи та масиви будь-яких типів, типи функцій, класи, структури, об'єднання. Типи вважаються похідними, оскільки, наприклад, класи, структури, об'єднання можуть містити об'єкти різних типів.

Можна виділити ще одну категорію типів – порядкові, в яких значення впорядковані і для кожного з них можна вказати попередній та наступний. До них належать цілі, символи, перелічувані типи.

Таблиця 10.2

Основні типи даних у С + +

Тип	Розмір у байтах	Діапазон значень
char	1	від -128 до 126
unsigned char	1	від 0 до 255
short	2	від -32 768 до 32 767
unsigned short	2	від 0 до 65 535
enum	2	від -2 147 483 648 до 2 147 483 647
long	4	від -2 147 483 648 до 2 147 483 647
unsigned long	4	від 0 до 4 294 967 295
int	4	як у long
unsigned int	4	як у unsigned long
float	4	від $3.4 \times 10\text{-}^{38}$ до $3.4 * 10^{38}$
double	8	від $1.7. \times 10\text{-}^{308}$ до $1.7 * 10^{308}$
long double	10	від $3.4 \times 10\text{-}^{4932}$ до $1.1 * 10^{4932}$
bool	1	true або false

Типи даних вказуються при оголошенні будь-яких змінних і функцій. Наприклад:

```
double a = 5.4, b = 2;
int c;
void F1 (double A);
```

Користувач може вводити в програму свої власні типи. Оголошення типів можна зробити в різних місцях коду. Місце оголошення впливає на область видимості або область дії так само, як і у випадку оголошення змінних.

Синтаксис оголошення типу:

typedef *визначення_типу ідентифікатор;*

Ідентифікатор вводиться користувачем як ім'я нового типу, а *визначення_типу* описує цей тип. Наприклад, оператор

```
typedef double Ar [10];
```

оголошує тип користувача з ім'ям **Ar** як масив з 10 дійсних чисел. Надалі на цей тип можна посилатися при оголошенні змінних. Наприклад:

```
Ar  A = {1,2,3,4,5,6,7,8,9,10};
```

10.2. Арифметичні типи даних

Арифметичні типи даних – це цілі та дійсні типи. До *цілих типів* належать:

char, short, int та **long**

разом з їх варіантами:

signed – зі знаком,

unsigned – без знака.

З цих ключових слів можна сформувати безліч цілих типів даних.

Специфікатори **signed** і **unsigned** можна застосовувати тільки до **char, shot, int, long**. Якщо тип позначено просто як **signed** або **unsigned**, то маються на увазі відповідно **signed int** і **unsigned int**.

За відсутності у вказівці типу специфікатора **unsigned** для цілих типів розуміємо **signed**. Винятком з цього правила є тип **char**. C++BUILDER дає змогу вам встановити за замовчуванням для **char** **signed** або **unsigned**. У цьому випадку, якщо записано оголошення

```
char  ch;
```

воно сприймається як

```
signed char ch;
```

Якщо ж необхідно оголосити змінну типу **char** без знака, то треба це зробити явно:

```
unsigned char ch;
```

Специфікатори **long** і **short** можна використовувати тільки з **int.** Якщо тип позначено просто як **long** або **short**, то мається на увазі відповідно **long int** і **short int.**

Обсяг пам'яті, зайнятий різними цілими типами, не лімітований стандартом ANSI C. Зазначено лише, що **short, int** і **long** повинні утворювати неспадну послідовність, тобто **short <= int <= long**. Тому не виключається, що всі три типи вимагають однакового обсягу пам'яті. Отже, обсяг пам'яті може змінюватися від однієї платформи до іншої і це треба враховувати під час створення переносних програм.

Типи зі знаком використовують старший біт для зберігання знака: 0 – додатний, 1–від'ємний.

Основними типами даних для подання *дійсних чисел* з плаваючою комою є типи

float,

double.

Перший з них розташовується в 32 бітах, другий – у 64. До типу **double** може застосовуватися специфікатор **long**, який збільшує розмір пам'яті до 80 біт.

Стандарт ANSI C не накладає жодних обмежень на спосіб задання дійсних чисел. Рекомендується використовувати для визначення розміру пам'яті для дійсних типів операцію **sizeof**.

10.3. Типи рядків

10.3.1. Масиви символів

У C++ відсутній спеціальний тип рядків. Рядки розглядаються як масиви символів, що закінчуються нульовим символом **('\ 0').** Рядок доступний через вказівник на перший символ у рядку. Значенням рядка є адреса його першого символу. У цьому випадку рядки подібні до масивів, тому що масив теж є вказівником на свій перший елемент.

Рядок може бути оголошений як масив символів, або як змінна типу **char *.** Кожне з двох наведених нижче еквівалентних оголошень присвоює рядковій змінній початкове значення «рядок».

```
char S [] = "рядок";
char * Sp = "рядок";
```

Перше оголошення створює масив із 7 елементів **S,** що містить символи 'р', 'я', 'д', 'о', 'к' та \ 0. Друге оголошення створює змінну вказівник **Sp,** який вказує на рядок з текстом «рядок», що знаходиться десь у пам'яті. Але в обох випадках число збережених символів на 1 більше за кількість значущих символів завдяки кінцевому нульовому символу.

Доступ до окремих символів рядка здійснюється за індексами, які починаються з нуля. Наприклад, **S [0]** і **Sp [0]** – перші символи оголошених вище рядків, S [1] і Sp [1] – другі.

У наведених оголошеннях довжина рядків визначалася автоматично компілятором. Можна оголошувати рядкові змінні заданої довжини. Наприклад, оператор

```
char buff [100];
```

визначає змінну *buff*, яка може містити рядок до 99 значущих символів плюс завершальний нульовий символ.

Для обробки рядків існують бібліотечні функції. Основні з них:

strcat – конкатенація (склеювання) двох рядків;

strcmp – порівняння двох рядків;

strcpy – копіювання одного рядка в інший;

strstr – пошук у рядку заданого підрядка;

strlen – визначення довжини рядка;

strupr – перетворення символів рядка до верхнього регістру;

sprintf – побудова рядка за заданим рядком форматування і списку аргументів.

10.3.2. Тип рядків AnsiString

У C++ Builder тип рядків **AnsiString** реалізований як клас, що оголошений у файлі **vcl/dstring.h**. Це рядки з нульовим символом на кінці. Під час оголошення змінні типу **AnsiString** ініціалізуються порожніми рядками.

Для **AnsiString** визначено операції відношень = =, ! =, > =, <=. Порівнюють, враховуючи регістр. Порівнюються коди символів, починаючи з першого, і якщо чергові символи не однакові, рядок, що містить символ з меншим кодом, вважається меншим. Якщо всі символи збіглися, але один рядок довший і в ньому є ще символи, то він вважається більшим.

Для **AnsiString** визначено операції присвоєння =, += і операцію склеювання рядків (конкатенації). Визначено також операцію індексації []. Індекси починаються з 1. Наприклад, якщо S1 = «Привіт», то Sl [l] поверне 'П', S1 [2] поверне 'р' і т.д.

Тип **AnsiString** використовується для ряду властивостей компонент C++Builder. Наприклад, для таких, як властивості **Text** вікон редагування, властивості **Caption** міток і розділів меню і т.д. Цей же тип використовується для відображення окремих рядків у списках рядків типу **TStrings**.

Розглянемо деякі приклади роботи з **AnsiString**. Наступний оператор демонструє конкатенацію (склеювання) двох рядків:

```
Label1-> Caption = Edit1-> Text + ' ' + Edit2-
> Text;
```

У цьому випадку у властивості **Label1 -> Caption** відображається текст, введений користувачем у вікні редагування **Edit1**,

потім записується символ пробілу, а пізніше текст, введений у вікні редагування **Edit2**. Склеювання рядків типу **AnsiString** легко здійснюється перевантаженою операцією додавання «+».

Розглянемо метод, за яким можна переходити від типу **AnsiString** до типу (**char** *). Незважаючи на те, що **AnsiString** практично завжди зручніше (**char** *), такі переході доводиться робити під час передавання параметрів у деякі функції, що вимагають тип параметрів (**char** *). Найчастіше це пов'язано з викликом функцій **API Windows** або функцій **C++Builder**, що інкапсулюють такі функції. Наприклад, функція **Application-> MessageBox** вимагає як два свої перші параметри (повідомлення і заголовок вікна) тип (**char** *).

Перетворення рядка **AnsiString** на рядок (**char** *) здійснюється функцією **c_str** () без параметрів, що повертає рядок з нульовим символом в кінці та містить текст того рядка **AnsiString**, до якого її застосовано. Наприклад, якщо необхідно рядки **S1** і **S2** типу **AnsiString** передати у функцію **Application-> MessageBox** як повідомлення і заголовок вікна, то виклик **Application-> MessageBox** може мати вигляд:

```
Application-> MessageBox (S1.c_str (), S2.c_str (),
MB_OK).
```

Можливе і зворотне перетворення рядка (**char** *) на рядок **AnsiString**. Для цього використовується функція

```
AnsiString (char * S),
```

яка повертає рядок типу **AnsiString**, що містить текст, записаний у рядку **S**, що є аргументом функції.

10.4. Перелічені типи

Перелічені типи визначають впорядковану множину ідентифікаторів, що являють собою можливі значення змінних цього типу. Багато типів C++Builder є переліченими, що спрощує роботу з ними, оскільки дає можливість працювати не з абстрактними числами, а з осмисленими значеннями.

Нехай, наприклад, у програмі повинна бути змінна **Rezim**, у якій зафіксовано один з можливих режимів роботи програми: читання даних, їх редагування, запис даних. Можна подати змінній **Rezim** тип **int** і надавати цій змінній у потрібні моменти часу одне з трьох умовних чисел: 0 – режим читання; 1 – режим редагування; 2 – режим запису. Тоді програма міститиме оператор

```
if (Rezim == 1) ...
```

Через деякий час вже забудеться, що означає значення **Rezim** 1, і розібратися в такому коді буде дуже складно. А можна вчинити інакше: визначити змінну **Rezim** як змінну переліченого типу і позначити її можливі значення як **mRead**, **mEdit**, **mWrite**. Тоді наведений вище оператор зміниться так:

```
if ( Rezim  == mEdit) ...
```

Звичайно, такий оператор зрозуміліший, ніж попередній.

Змінні переліченого типу можна оголошувати так:

enum *{<константа 1>,. .., <константа n>} <імена змінних>;*

Наприклад

```
enum {mRead, mEdit, mWrite} Rezim;
```

Цей оператор вводить іменовані константи **mRead, mEdit, mWrite** і змінну **Rezim**, яка може набувати значення цих констант. У момент оголошення змінна ініціалізується значенням першої константи, в нашому прикладі – **mRead**. Надалі можна надавати їй будь-яких допустимих значень. Наприклад:

```
Rezim = mEdit;
```

Значення змінної переліченого типу можна перевіряти, порівнюючи її величину з можливими значеннями. Крім того, потрібно враховувати, що перелічувані належать до цілих порядкових типів, і до них застосовуються будь-які операції порівняння. Наприклад, можна написати оператори:

```
if (Rezim > mRead) ...;
if (Rezim <mWrite) ...;
if (Rezim == mEdit)…;
```

Можна також використовувати **Rezim** у структурі **switch**;

```
switch (Rezim)
{
  case mRead: ...
    break;
  case mEdit:…
    break;
  case mWrite: ...
}
```

За замовчуванням перелічувані значення, вказані в оголошенні **enum**, інтерпретуються як цілі числа, причому перше значення еквівалентне 0, друге – 1 і т.д. Саме ці значення розглядаються в операціях відношень >, < та ін. Значення за замовчуванням можна змінити, якщо після імені константи вказати знак рівності (=) і задати ціле значення – як додатне, так і від'ємне. Наприклад:


```
enum {mRead = -1, mEdit, mWrite = 2) Rezim;
```

Якщо після якихось констант не задано їхнього цілого значення, його вважаються на 1 більшим за попереднє. Тому в наведеному прикладі **mRead** еквівалентно -1, **mEdit** еквівалентно 0, **mWrite** еквівалентно 2.

Після ключового слова **enum** може слідувати тег – ім'я типу, що оголошується. Наприклад:

```
enum rr {mRead = -1, mEdit, mWrite = 2}
Rezim, Rezim1;
```

Цей оператор оголошує дві змінні **Rezim** і **Rezim1** переліченого типу, і крім цього визначає тип **rr**. Надалі можна скористатися ім'ям **rr** для оголошення якихось нових змінних, наприклад:

```
rr Rezim3;
```

10.5. Множини

Множина – це група елементів, яка асоціюється з її ім'ям і з якою можна порівнювати інші величини, щоб визначити, чи належать вони цій множині. Як окремий випадок, множина може бути порожньою. Множину реалізовано в C++Builder як шаблон класу, визначений у файлі заголовка **vcl / sysdefs.h.**

Оголошується множина оператором:

Set *<type, minval, maxval> змінні;*

Параметр **type** визначає тип елементів множини. Зазвичай це вбудовані типи **int**, **char** або перелічені. Параметри **minval** і **maxval** типу **unsignedchar** визначають мінімальне та максимальне значення елементів множини. Мінімальне значення повинно бути не меншим за 0, максимальне – не більшим за 255.

Приклади оголошення множин

Оголошення змінної **lat** як множини всіх заголовних латинських букв має вигляд:

```
Set <char,'A','Z'> lat;
```

Наступний оператор оголошує множину **Sym**, що містить усі символи:

```
Set <char, 0, 255> Sym;
```

Наступні оператори оголошують тип **UPPERCASESet** множині всіх заголовних латинських букв і оголошують змінні **lat1** і **lat2** цього типу:

```
typedef Set <char,'A','Z'> UPPERCASESet;
UPPERCASESet lat1, lat2;
```

Наступні оператори визначають множину **M**, елементами якої є дані переліченого типу **Col: red, yellow, green**:

```
enum Col {white, red, yellow, green};
Set <Col, red, green> M;
```

Оголошення змінної типу множина **Set** не ініціалізує її якимись значеннями. Ініціалізацію можна робити за допомогою описаної нижче операції << - додавання елемента до множини.

Для множини в табл. 10.3 визначено такі операції (в описі операцій словами «дано множину» позначається лівий операнд):

Таблиця 10.3

Операції над множинами

Операція	Визначення	Опис
1	2	3
-	**Set ___fastcall operator -(const Set& rhs) const;**	ця множина дорівнює різниці двох множин: цієї та **rhs**
-=	**Set& ___ fastcall operator -=(const Set& rhs);**	створення нової множини як різниці двох множин: цієї і **rhs**
	Set& ___ fastcall operator *(const Set& rhs);	створення нової множини як перетин двох множин: даної і **rhs**
*=	**Set ___fastcall operator *=(const Set& rhs) const;**	ця множина дорівнює перетину двох множин: цієї і **rhs**
+	**Set ___fast call operator +(const Set& rhs) const;**	створення нової множини як об'єднання двох множин: цієї і **rhs**
+=	**Set& __fastcall operator +=(const Set& rhs);**	множина дорівнює об'єднанню двох множин: цієї і **rhs**
<<	**Set& __fastcall operator <<(const T el);**	додавання елемента **el** до цієї множини
<<	**friend ostream& operator <<(ostream& os, const Set& arg);**	помістити множину **arg** у потік **ostream** (виводиться 0 або 1 для кожного елемента залежно від його наявності у множині)
>>	**Set& __fastcall operator >>(const T el);**	Видалення елемента **el** з цієї множини
>>	**friend istream& operator >>(istream& is, Set& arg);**	витягти множину **arg** з потоку **istream** (вводиться 0 або 1 для кожного елемента залежно від його наявності в множині)
=	**Set& __fastcall operator =(const Set& rhs);**	присвоювання множині вмісту множин **rhs**
==	**bool___fastcall operator ==(const Set& rhs) const;**	еквівалентність двох множин: цієї і **rhs** (збіг усіх елементів)
!=	**bool___fastcall operator !=(const Set& rhs) const ;**	нееквівалентність двох множин: цієї і **rhs**

Усі операції можна застосовувати тільки до множин одного типу, тобто до таких, під час оголошення яких усі аргументи оголошення (**type**, **minval** і **maxval**) збігаються. В операціях, що створюють нову множину (операції +, -, *), змінна, до якої заноситься результат, повинна бути того самого типу, що і операнди. Операція еквівалентності повертає **true** у випадку, коли обидва операнди містять тільки елементи, які збігаються. Відповідно тільки в цьому випадку операція нееквівалентності повертає **false**.

Для множин **Set** у табл. 10.4 наведено два методи.

Таблиця 10.4

Методи для роботи з множинами

Метод	Визначення	Опис
Clear	**Set& ___ fastcall Clear();**	Очищення множини
Contains	**bool ___ fastcall Contains(const T el) const;**	Перевірка наявності в множині елемента **el**

Розглянемо приклади роботи з множинами.

1. Нехай користувачеві в програмі задаються деякі запитання, що припускають відповідь типу **«Yes / No»**. Тоді можливі символи, які ввів користувач як відповіді, утворюють множини, що містять символи **«y»**, **«Y»**, **«n»** і **«N»**. Сформувати таку множину можна операторами:

```
Set <char, 0, 255> TrueKey;
TrueKey <<'y' <<'Y' <<'n'<< 'N';
```

Тоді перевірити, чи належить введений користувачем символ **Key** множині допустимих відповідей, можна за методом **Contains**:

```
if (! TrueKey.Contains (Key))
ShowMessge ("Ви ввели помилкову відповідь");
else. . .
```

2. Нехай необхідно, щоб у вікні редагування **Edit1** користувач міг вводити тільки число, тобто тільки цифри від 0 до 9. Це можна зробити, включивши в обробник події **OnKeyPress** цього вікна оператори:

```
Set <char, '0', '9'> Dig;
 Dig <<'0' <<'!'<<'2 '<<'3'<< '4 '<<'5' <<'6' <<'7'<< '8 '<<'9';
 if (! Dig.Contains (Key))
{Key = 0; Beep ();}
```

За спроби користувача ввести символ, відмінний від цифри, пролунає звук (його забезпечить функція **Beep**), і символ не з'явиться у вікні.

10.6. Вказівники

Вказівник – це змінна, значення якої дорівнює значенню адреси пам'яті, за якою зберігається значення деякої іншої змінної. У цьому сенсі ім'я цієї іншої змінної відсилає до її значення безпосередньо, а вказівник – опосередковано. Посилання на значення за допомогою вказівника називається непрямою адресацією.

Вказівник перед своїм використанням необхідно оголосити. Оголошення вказівника має вигляд:

```
type * ptr;
```

де **type** – один із визначених користувачем типів, а **ptr** – вказівник. Читається це оголошення так: «**ptr** є вказівником на значення типу **type**».

Наприклад,

```
int * countPtr, count;
```

оголошує змінну **countPtr** типу **int** * (тобто вказівник на ціле число) і змінну **count** цілого типу. Символ * в оголошенні належить тільки до **countPtr**. Кожна змінна, що оголошується як вказівник, повинна мати перед собою знак зірочки (*). Якщо в наведеному прикладі бажано, щоб і змінна **count** була вказівником, потрібно записати:

```
int * countPtr, * count;
```

Символ * у цих записах позначає операцію непрямої адресації. Можно оголосити і вказівник на **void:**

```
void * Pv;
```

Це універсальний вказівник на будь-який тип даних. Але перш ніж його використовувати, йому треба під час роботи присвоїти значення вказівника на якийсь конкретний тип даних. Наприклад:

```
Pv = countPtr;
```

У C++Builder вказівники використовуються дуже широко. Зокрема всі компоненти, форми тощо оголошуються саме як вказівники на відповідний об'єкт. Переглянувшись заголовний файл будь-якого проекту, можна побачити в ньому оголошення вигляду:

```
TForm1 * Form1;
TButton * Button1;
```

Вказівники повинні бути ініціалізовані або під час оголошення, або за допомогою оператора присвоєння. Вказівник може отримати як перше значення 0, **NULL** або адресу. Вказівник з початковим значенням 0 або **NULL** ні на що не вказує. **NULL** – це символічна константа, визначена спеціально з метою показати, що цей вказівник ні на що не вказує. Приклад оголошення вказівника із його ініціалізацією:


```
int * countPtr = NULL;
```

Для присвоєння вказівнику адреси деякої змінної використовується операція адресації **&**, яка повертає адресу свого операнда. Наприклад, якщо є оголошення

```
int y = 5;
int * yPtr, x;
```

то оператор

```
yPtr = & y;
```

присвоює адресу змінної **y** вказівнику **yPtr**.

Для того, щоб отримати значення, на яке вказує вказівник, виконують операцію * - *операцію непрямої адресації*. Вона повертає значення об'єкта, на який вказує її операнд (тобто вказівник). Наприклад, якщо продовжити наведений вище приклад, то оператор

```
x = * yPtr
```

присвоїть змінній **x** значення 5, тобто значення змінної **y**, на яку вказує **yPtr**.

Масиви та вказівники в C++ тісно пов'язані і можуть бути використані майже еквівалентно. Ім'я масиву можна розуміти як константний вказівник на перший елемент масиву. Його відмінність від звичайного вказівника тільки в тому, що його не можна модифікувати.

Вказівники можна використовувати для виконання будь-якої операції, зокрема індексування масиву. Нехай зроблено таке оголошення:

```
int b [5] = {1,2,3,4,5}, * Pt;
```

Тим самим оголошено масив цілих чисел **b[5]** і вказівник на ціле **Pt**. Оскільки ім'я масиву є вказівником на перший елемент масиву, то можна задати вказівнику **Pt** адресу першого елемента масиву **b** за допомогою оператора

```
Pt = b;
```

Це еквівалентно присвоюванню адреси першого елемента масиву таким чином

```
Pt = & b [0];
```

Тепер можна послатися на елемент масиву **b [3]** за допомогою виразу * **(Pt 3).**

Вказівник можна індексувати точно так само, як і масиви. Наприклад, вираз **Pt [3]** посилається на елемент масиву **b [3].** Тобто маніпуляції, визначені для масивів, визначені і для вказівників на масиви.

Вказівники можуть застосовуватися як операнди в арифметичних виразах, виразах присвоювання і виразах порівняння. Однак не всі операції, які зазвичай використовуються в цих висловах, дозволені стосовно змінних вказівників.

Із вказівниками можна виконувати обмежену кількість арифметичних операцій:

1. Вказівник можна збільшувати (++);
2. Вказівник можна зменшувати (--);
3. Складати із вказівником цілі числа (+ або +=);
4. Віднімати від вказівника цілі числа (- або -=);
5. Віднімати один вказівник від іншого.

Додавання вказівників з цілими числами відрізняється від звичайної арифметики. Додати до вказівника 1 означає перемістити його на кількість байтів, що містяться у змінній, на яку він вказував. Зазвичай такі операції застосовуються до вказівників на масиви. Якщо продовжити наведений вище приклад, у якому вказівнику **Pt** присвоєно значення b – вказівник на перший елемент масива, то після виконання оператора

```
Pt = 2;
```

Pt вказуватиме на третій елемент масиву **b**. Істинне ж значення вказівника зміниться на кількість байтів, займаних одним елементом масиву, помножену на 2. Наприклад, якщо кожен елемент масиву **b** займає 2 байти, то значення **Pt** (тобто адреса в пам'яті, на яку вказує **Pt**) збільшиться на 4. Аналогічні правила діють і під час віднімання від вказівника цілого значення.

Вказівники можна віднімати один від іншого. Наприклад, якщо **Pt** вказує на перший елемент масиву **b**, а вказівник **Pt1** – на третій, то результат виразу **Pt1** – **Pt** дорівнюватиме 2 – різниці індексів елементів, на які вказують ці вказівники. І так буде, незважаючи на те, що адреси, які містяться в цих вказівниках, відрізняються на 4 (якщо елемент масиву займає 2 байти).

Арифметика вказівників втрачає будь-який сенс, якщо її виконують не над вказівниками на масив. Не можна вважати, що дві змінні однакового типу зберігаються в пам'яті впритул одна до одного, якщо тільки вони не є сусідами в масиві.

Порівняння вказівників операціями

>, <, >=, <=

також мають сенс тільки для вказівників на один і той самий масив. Однак операції відношення == та != мають сенс для будь-яких вказівників. При цьому вказівники мають однакові значення, якщо вони вказують на ту саму адресу в пам'яті.

Вказівник можна присвоювати іншому вказівнику, якщо обидва вказівники мають однаковий тип. В іншому разі потрібно використовувати операцію приведення типу, щоб перетворити значення вказівника в правій частині присвоювання на тип вказівника в лівій частині присвоєння. Винятком з цього правила є вказівник на **void** (тобто **void ***), який є спільним вказівником, здатним представляти вказівники будь-якого типу. Вказівнику на **void** можна присвоювати всі типи вказівників без приведення типу. Однак вказівник на **void** не можна присвоїти безпосередньо вказівнику іншого типу – вказівник на **void** спочатку повинен бути приведений до типу відповідного вказівника.

Також можна використовувати масиви вказівників. Такі структури часто використовують у масивах рядків або в масивах вказівників на різні об'єкти. Наприклад, можна зробити таке оголошення:

```
char * Sa [2] = {"Це перший рядок", "Другий"};
```

Оголошено масив розміром 2 елементи типу (**char** *). Кожен елемент такого масиву – рядок. Але в C++ рядок є вказівником на його перший символ. Отже, кожен елемент у масиві рядків насправді є вказівником на перший символ рядка. Кожен рядок зберігається в пам'яті як рядок і завершується нульовим символом. Кількість символів у кожному з рядків може бути різною. Отже, масив вказівників на рядки дає змогу забезпечити доступ до рядків символів будь-якої довжини.

10.7. Посилання

Посилання – це спеціальний тип вказівника, який дає змогу працювати із вказівником як з об'єктом. Оголошення посилання робиться за допомогою операції посилання, що позначається амперсандом (**&**) – тим самим символом, який використовується для адресації. Якщо в програмі є вказівник на об'єкт якогось типу **MyObject**:

```
MyObject * P = new MyObject;
```

то можна створити посилання на цей об'єкт оператором:

```
MyObject & Ref = * P;
```

Оголошену так змінну **Ref** є посиланням на об'єкт **MyObject**. Її можна розглядати як псевдонім об'єкта. Ця змінна реально є вказівником, а не самим об'єктом, але робота з нею проводиться як з об'єктом. Наприклад, якщо необхідно отримати доступ до деякої властивості об'єкта **x**, то через вказівник на об'єкт забезпечується

доступ виразом **P-> x**, тобто через операцію стрілка. А через посилання забезпечується доступ до властивості **x** виразом **Ref.x**, тобто через операцію крапка. Аналогічно можна отримати доступ за посиланням і до будь-яких компонентів. Наприклад, якщо в програмі є позначка **Label1**, то можна звертатися до її властивості **Caption** оператором

```
Label1-> Caption = "Це звернення за вказівником";
```

Але можна ввести відповідне посилання і звертатися через нього:

```
TLabel & ref = * Label1;
ref.Caption = "Це звернення з посиланням";
```

10.8. Масиви в C++

10.8.1. Одновимірні масиви в C++

Масив являє собою структуру даних, що дає змогу зберігати під одним ім'ям сукупність даних однакового типу. Масив характеризується своїм ім'ям, типом елементів, розміром (кількістю елементів), нумерацією елементів і розмірністю.

Оголошення змінної як одновимірного масиву має вигляд:

тип змінна [константний_вираз]

Наприклад, оператор

```
int A [10];
```

оголошує масив з ім'ям **A**, що містить 10 цілих чисел. Доступ до елементів цього масиву здійснюється виразом **A [i]**, де **i** – індекс, що є цілим числом у діапазоні 0–9. Наприклад, **A [0]** – значення першого елемента; **A [1]** – другого; **A [9]** – останнього. Індекс останнього елемента на 1 менший за розмір масиву. Це пов'язано з тим, що індекси в C++ починаються з 0.

Наведемо приклади використання цього масиву.

```
A [0] = 1;
A [1] = 1;
for (int i = 2; i <10; i++) A [i] = A [i-2]+ A [i-1];
```

Цей код заповнює масив так званими числами Фібоначчі, перші 2 з яких дорівнюють 1, а кожне наступне дорівнює сумі двох попередніх.

Елементи масиву можуть бути будь-якого типу. Наприклад,

```
char S [10];
```

оголошує масив символів.

Масив символів – це фактично рядок, і з ним можна працювати як з рядком і як з масивом. У разі використання масиву символів як рядка треба пам'ятати, що це рядок фіксованої допустимої довжини. Кількість символів, які розміщені в рядку, не повинна перевищувати оголошеного розміру масиву **n – 1**, оскільки рядок закінчується нульовим символом.

Оголошення масиву можна поєднувати із заданням масиву початкових значень. Ці значення перераховуються у списку ініціалізації після знаку рівності, розділяються комами і беруться у фігурні дужки. Наприклад:

```
int A [10] = {1,2,3,4,5,6,7,8,9,10};
char S [10] = {"abcdefghi \ 0"};
```

Якщо початкових значень менше ніж елементів у масиві, то елементи, що залишилися, автоматично отримують нульові початкові значення. Наприклад, оператор

```
int A [10] = {1,2,3};
```

задає значення першим трьом елементам, а інші дорівнюватимуть 0. Оператор

```
int A [10] = {0};
```

присвоює нульові значення всім елементам масиву.

Якщо масив при його оголошенні не ініціалізований, то його елементи мають випадкові значення. Елементи такого масиву не можна використовувати у виразах, поки їм не нададуть яких-небудь значень.

У масивах символів надання нулів елементам, не зазначених у списку ініціалізації, рівносильно наданню нульових символів, що означають кінець рядка. Тому наведене вище оголошення масиву **S** з його ініціалізацією надлишкове. Нульовий символ у кінці можна не вказувати. Наприклад, нормально будуть сприйняті такі оголошення:

```
char S [10] = {"abedefghi"};
char S1 [10] = {"abc"};
```

Останнє оголошення виділяє місце під масив з 10 елементів, але ініціалізує його рядком із трьох елементів.

В оголошенні зі списком ініціалізації розмір масиву можна не вказувати. Тоді кількість елементів масиву дорівнюватиме кількості елементів у списку початкових значень. Наприклад, оголошення

```
int A [ ] = {1, 2, 3, 4, 5};
```

створює масив із п'яти елементів. Оголошення

```
char S1 [ ] = {"abc"};
```

створює масив із чотирьох елементів: три значущі символи плюс нульовий символ.

При оголошенні масиву як розмір краще завжди використовувати іменовані константи. Наприклад, нижче наведено оголошення масиву і оператор, що підраховує суму його елементів:

```
int A [10];
// Підрахунок суми
int Sum = A [0];
for (int i = 1; i <10; i++) Sum += A [i];
```

Якщо надалі потрібен буде масив **A** не з 10 елементів, а, наприклад, зі 100, необхідно буде змінити розмір масиву і в оголошенні **A**, і у всіх операторах, що працюють з цим масивом (у цьому випадку в операторі **for**). А таких операторів у різних частинах програми може бути дуже багато. Про таку програму кажуть, що вона погано масштабується. Грамотніше реалізувати цей приклад так:

```
const Amax = 10;
int A [Amax];
// Підрахунок суми
 int Sum = A [0];
 for (int i = 1; i <Amax; i++)
 Sum += A [i];
```

У цьому випадку вводиться іменована константа **Amax**, яку використовують у всіх операторах, у яких потрібен розмір масиву. Тоді за необхідності змінити розмір масиву достатньо змінити його тільки в одному операторі, що оголошує **Amax**. Програма одразу стає масштабованою. А оголошення **Amax** як константи гарантує, що оголошене значення не буде випадково змінено десь у програмі.

Аналогічний результат можна отримати, якщо замінити оголошення константити директивою компілятора **# define**.

```
# define Amax 10
```

Усі розміри масивів у програмі слід визначати іменованими константами, або макросами. Це робить програму зрозумілішою, істотно полегшує її налагодження та супровід.

Іноді потрібні константні масиви, дані з яких програма може тільки читати. Такі масиви обов'язково повинні ініціюватися в момент оголошення. Наприклад:

```
const AnsiString Day[] = {"понеділок","вівторок",
"середа", "четвер", "п'ятниця", "субота", "неділя"};
```


10.8.2. Багатовимірні масиви

У С++ можна оголошувати і багатовимірні масиви, тобто масиви, елементами яких є масиви. Наприклад, двовимірний масив можна оголосити так:

```
int A2 [10] [3];
```

Цей оператор описує двовимірний масив, який можна уявити собі як таблицю, що складається з 10 рядків і 3 стовпців. Доступ до значень елементів багатовимірного масиву забезпечується через індекси, кожен з яких береться у квадратні дужки. Наприклад,
A2 [3] [2] – значення елемента, що знаходиться на перетині четвертого рядка і третього стовпця (пам'ятайте, що індекси в С++ починаються з 0).

Якщо багатовимірний масив ініціалізується при його оголошенні, список значень за кожного розмірністю беруть у фігурні дужки. Наведений нижче оператор оголошує тривимірний масив **A3** розмірністю 4 на 3 на 2.

```
int A3 [4] [3] [2] = {{{0,1}, {2,3}, {4,5}},
                      {{6,7}, {8,9}, {10,11 }},
                      {{12,13}, {14,15}, {16,17}},
                      {{18,19}, {20,21}, {22,23}}};
```

Цей оператор створює масив **A3**, чотири рядки якого є матрицями вигляду

0	1
2	3
4	5

6	7
8	9
10	11

12	13
14	15
16	17

18	19
20	21
22	23

Наприклад, елемент **A3 [0] [1] [0]** дорівнює 2, елемент **A3 [3] [0] [1]** дорівнює 19 і т.д. Якщо в списку ініціалізації в якійсь із розмірностей не вистачає даних, то вважають, що всі подальші не перераховані елементи дорівнюють нулю.

10.8.3. Операції з масивами, використання масивів як параметрів функцій

Ім'я масиву є константним вказівником на перший елемент масиву. Взаємозв'язок масивів та вказівників докладно розглянуто в

підрозділі 10.6. Оскільки ім'я масиву – константний вказівник, його не можна модифікувати і до нього не застосовні всі операції привласнення.

До імені масиву можна застосовувати операцію **sizeof**, яка в цьому випадку повертає значення, що дорівнює загальному обсягу пам'яті, відведеному під всі елементи масиву. Отже, кількість елементів масиву **A** можна визначити виразом

```
sizeof (A) / sizeof (A [0])
```

оскільки під кожен елемент масиву відведено однаковий обсяг пам'яті. Подібне обчислення розміру масиву виконує макрос **ARRAYSIZE**. Наведений вище вираз еквівалентний виразу

```
ARRAYSIZE (A)
```

Коли масиви передаються у функцію як параметри заголовок тримає тип та ім'я масиву з подальшими порожніми квадратними дужками. Наприклад, якщо функція **F** повинна сприймати масив як параметр, її прототип може мати вигляд:

```
void F (int Ar []);
```

Звернення до такої функції можна записати так:

```
const Amax = 10;
int A [Amax];
F (A);
```

Як видно, у виклику функції вказується просто ім'я масиву. Усередині функції до елементів цього масиву можна звертатися звичайним чином, наприклад **Ar [5]**.

C++ передає ім'я масиву у функцію за посиланням. Це означає, що якщо функція змінює значення елементів масиву, то змінюються елементи вихідного масиву, який передавався у функцію.

Здебільшого тільки імені масиву мало, щоб у функції обробити його елементи. Усередині функції потрібно знати розмір масиву, щоб можна було організувати його циклічну обробку. Тому зазвичай у функцію передається не тільки масив, але і його розмір. Заголовок функції може мати вигляд:

```
void F (int Ar [], int N);
```

а виклик функції:

```
F (A, Amax);
```

Найчастіше бібліотечні функції вимагають як другий параметр не розмір масиву, а значення його останнього індексу, яке на одиницю менше за розмір. У цьому випадку виклик функції може мати вигляд:

```
F (A, Arnax - 1);
```


Передавання масиву за посиланням не гарантує захисту від несанкціонованої зміни програмою значень елементів масиву. Якщо необхідно захистити масив від подібних змін, його треба передати у функцію як константу:

```
void F (const int Ar [], int N);
```

Нехай, наприклад, необхідно написати функцію, що підраховує суму елементів масиву цілих. Тоді можна оформити її так:

```
int Sum (const int A [], int N)
{
// N - розмір масиву
  int S = A [0];
  for (int i = 1; i <N; i++) S+ = A [i];
  return S;
  }
```

Нижче наведено приклад тестування цієї функції.

```
# define Bmax 10.
int B [10] = {1,2,3,4,5,6,7,8,9,10};
ShowMessage ("Сума дорівнює" IntToStr (Sum (B,
Bmax)));
```

При виклику функції не обов'язково передавати весь масив. Можна передати тільки якусь його частину. Наприклад, можна передати у функцію параметр розміру масиву, менший за істинний. Якщо в наведеному прикладі як другий параметр передати в функцію не **Bmax**, а **Bmax - 2**, то функція оброблятиме тільки вісім перших елементів з індексами від 0 до 7. Можна і як початок масиву передати у функцію вказівник на якийсь елемент масиву. Наприклад, якщо звертатись до функції так:

```
Sum (B+2, Bmax - 2),
```

то передається в неї вказівник не на перший, а на третій елемент. Тому, коли функція звертатиметься до елементів масиву від 0 до 7, насправді вона працюватиме з елементами, індекси яких від 2 до 9. Тобто суму буде підраховано для елементів, починаючи з третього.

Якщо у функцію передається багатовимірний масив, то в заголовку тільки квадратні дужки першої розмірності залишаються порожніми, а в дужках наступних розмірностей повинні вказуватися константами їхні розміри. Наприклад, якщо функція **F2** повинна приймати двовимірний масив розміром 3 на 3, то її заголовок може мати вигляд:

```
void F (const int Ar [] [3]);
```

Викликають цю функцію звичайним передаванням у неї імені масиву. Наприклад, **F (A)**.

10.9. Структури

10.9.1. Структури в мові С

Структури – це складові типи даних, побудовані з використанням інших типів. Вони являють собою об'єднаний загальним іменем набір різних типів. Саме тим, що в них можуть зберігатися дані різних типів, вони і відрізняються від масивів, що зберігають дані одного типу. Окремі дані в структурі називаються елементами, або полями.

Найпростіший варіант оголошення структури виглядає так:

```
struct TPers
{
    AnsiString Fam, Nam, Par;
    unsigned Year;
    bool Stat;
    AnsiString Dep;
};
```

Ключове слово **struct** починає визначення структури. Ідентифікатор **TPers** – ім'я структури. Ім'я структури використовується при оголошенні змінних структур цього типу. У цьому прикладі ім'я нового типу – **TPers**. Імена, оголошені в фігурних дужках опису структури – це елементи структури. Елементи однієї і тієї самої структури повинні мати унікальні імена, але дві різні структури можуть містити неконфліктуючі елементи з однаковими іменами. Кожне визначення структури має закінчуватися крапкою з комою.

Визначення **TPers** містить шість елементів. Передбачається, що така структура може зберігати дані про працівника якоїсь установи. Типи даних різні: елементи **Fam**, **Nam**, **Par** і **Dep** – рядки, що зберігають відповідно прізвище, ім'я, по батькові працівника і назву відділу, в якому він працює. Елемент **Year** всього типу зберігає рік народження, елемент **Stat** булевого типу зберігає відомості про стать. Елементи структури можуть бути будь-якого типу, але структура не може містити екземпляри самої себе. Наприклад, елемент типу **TPers** не може бути оголошений у визначенні структури **TPers**. Однак можна ввести вказівник на іншу структуру типу **TPers**. Структура, що містить елемент, який є вказівником на такий самий структурний тип, називається структурою із самоадресацією. Такі структури зручно використовувати для формування різних списків.

Саме по собі оголошення структури не резервує жодного простору в пам'яті, воно тільки створює новий тип даних, який може

використовувати оголошення змінних. Змінні структури оголошуються так само, як змінні інших типів. Оголошення

```
TPers Pers, PersArray [10], * Ppers;
```

оголошує змінну **Pers** типу **TPers**, масив **PersArray** – з 10 елементами типу **TPers** і вказівник **Ppers** на об'єкт типу **TPers**.

Змінні структури можна оголошувати і безпосередньо в оголошенні самої структури після закриваючої фігурної дужки. У цьому випадку називати ім'я не обов'язково:

```
struct {
        AnsiString Fam, Nam, Par;
        unsigned Year;
        bool Stat;
        AnsiString Dep;
        } Pers, PersArray [10], * Ppers;
```

Для доступу до елементів структури використовуються операції доступу до елементів: операція крапка (.) і операція стрілка (->). Операція крапка звертається до елемента структури за іменем об'єкта або за посиланням на об'єкт. Наприклад:

```
Pers.Fam = "Іванов";
Pers.Nam = "Іван";
Pers.Par = "Іванович";
Pers.Year = 1960;
Pers.Stat = true;
Pers.Dep = "Бухгалтерія";
```

Операція стрілка забезпечує доступ до елемента структури через вказівник на об'єкт. Припустимо, що виконаний оператор

```
Ppers = &Pers;
```

який привласнив вказівнику **Ppers** адресу об'єкта **Pers**. Тоді зазначені вище присвоєння елементам структури можна виконати так:

```
Ppers -> Fam = "Іванов";
Ppers -> Nam = "Іван";
Ppers -> Par = "Іванович";
Ppers--> Year = = I960;
Ppers -> Stat = true;
Ppers -> Dep = "Бухгалтерія
```

10.9.2. Самоадресовані структури

Нерідко в пам'яті треба динамічно розміщувати послідовності структур, ніби формуючи якийсь фрагмент бази даних, що призначений для оперативного аналізу та обробки. Оскільки динамічне розміщення

проводиться в непередбачуваних місцях пам'яті, то такі структури треба наповнити елементами, що містять вказівники на наступну аналогічну структуру. Такі структури з посиланнями на аналогічні структури і називаються самоадресованими. На рис. 10.1 наведено схему зв'язку таких структур у послідовність. Полю вказівника в останній структурі зазвичай присвоюється значення **NULL**, що є ознакою останньої структури при організації пошуку в списку.

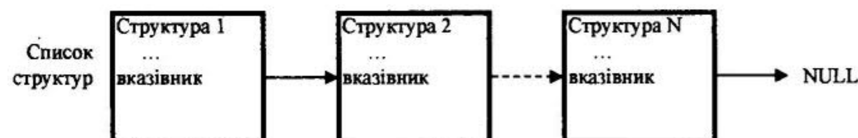


Рис. 10.1. Схема зв'язку структур у послідовність

Для прикладу з попереднього підрозділу оголошення структури як самоадресованої виглядатиме так:

```
    struct TPers {
            AnsiString Fam, Nam, Par;
        unsigned Year;
        bool Stat,
        AnsiString Dei:; TPers * pr;   };
```

Наведемо приклад формування в пам'яті списку таких структур. Для цього треба визначити три змінні, що є вказівниками на структури:

```
   TPers * P0 = NULL, * Pnew, * Pold;
```

Перша з цих змінних завжди вказуватиме на першу структуру у списку. Дві інші змінні – допоміжні. Якщо в деякий момент виникла необхідність динамічно розмістити в пам'яті чергову структуру і вставити її в кінець списку, це можна зробити таким кодом:

```
   // Виділення пам'яті під нову структуру
   Pnew = new TPers;
   // Заповнення елементів структури
  Pnew-> Fam = "Іванов";
  Pnew-> Nam = "Іван";
  Pnew-> Par = "Іванович";
  Pnew-> Year = 1960;
  Pnew-> Stat = true;
  Pnew-> Dep = "Бухгалтерія";
  Pnew-> pr = NULL;
  if (P0 == NULL) P0 = Pnew; // P0 - вказівник на
першу структуру
  else Pold-> pr = Pnew; // вказівник на чергову структуру
  Pold = Pnew;
```


Якщо список ще не розпочато (**P0 = NULL**), то вказівнику **P0** присвоюється посилання на розміщену структуру (**Pnew**). В іншому випадку посилання на нову структуру присвоюється полю **pr** попередньої структури у списку (**Pold**). Тобто нову структуру вводять до загального списку. Полю **pr** цієї структури присвоюється значення **NULL**. Це є ознакою того, що ця структура є останньою в списку.

Для перегляду списку необхідно проходити в циклі за вказівниками. Наприклад:

```
   Pnew = P0;
   while (Pnew ! = NULL)
   {
      ShowMessage (Pnew-> Fam+" " + Pnew-> Nam+" " +
Pnew -> Par );
      Pnew  =  Pnew->  pr;      //  перехід  до  нової
структури
   }
```

Також можна в списку переставляти структури, їх видаляти тощо. Для всіх цих операцій не треба нічого переміщувати в пам'яті. Достатньо тільки змінювати відповідні посилання в полях **pr**. Раніше подібні списки використовували для створення в пам'яті стеків, черг та інших упорядкованих списків. Однак у C++Builder введено спеціальні типи даних **TList** і **TStringList**, які обробляють подібні списки і мають зручні методи для управління ними.

10.9.3. Структури в мові C++

Все, що розглянуто в попередніх розділах, стосується як до мови С, так і C++. Але в C++ поняття структури істотно розширене і наближене до поняття класу. Зокрема в структурах, крім розглянутих раніше даних-елементів, дозволяється описувати функції-елементи. Розглянемо це на прикладі раніше оголошеної структури **TPers**. Введемо до цієї структури функцію-елемент **Show**, яка буде показувати інформацію, що зберігається в структурі:

```
   struct TPers {
       AnsiString Fam, Nam, Par;
       unsigned Year;
       bool Stat;
       AnsiString Dep;
       TPers * pr;
       void Show ()
  {
```

```
ShowMessage ("Співробітник відділу \"" "+ Dep+" \ "
"+ Fam + " "+ Nam +" " + Par +","+ IntToStr (Year)+
"р.н., стать" +
      (Stat? "чоловічий": "жіночий"));}
   };
```

Функція **Show** відображає інформацію вигляду: «Співробітник відділу «Бухгалтерія» Іванов Іван Іванович, 1960 р.н., стать чоловіча». Звертаються до цієї функції-елемента через змінну структури операцією крапка або через вказівник на змінну операцією стрілка. Наприклад:

```
    Pers. Show ();
   Pnew -> Show ();
```

З використанням введеної функції **Show** наведений у попередньому підрозділі приклад перегляду списку можна спростити:

```
   Pnew = P0;
   while (Pnew! = NULL)
   {
   Pnew-> Show ();
   Pnew = Pnew-> pr;}
```

У С++ можна вводити специфікатори доступу до даних-елементів і функцій-елементів так само, як це робиться в класі. Дозволяються специфікатори **public** (відкритий) і **private** (закритий). Закриті елементи структури можуть бути доступні тільки для функцій-елементів цієї структури. Ані через об'єкт, ані через вказівник на об'єкт доступ до них неможливий. Закритими оголошуються якісь допоміжні дані-елементи, які не цікаві для користувача, а також допоміжні функції (утиліти), необхідні для роботи основних функцій-елементів структури.

Відкриті елементи структури можуть бути доступні для будь-яких функцій у програмі. Основне завдання відкритих елементів полягає – надати клієнтам структури уявлення про можливості, які вона має. Це відкритий інтерфейс структури.

За замовчуванням доступ до елементів структури **public** – відкритий. Якщо треба сховати від користувача якісь елементи, необхідно вказати специфікатор **private,** що завершується двокрапкою, і розмістити після нього оголошення закритих елементів. Все, що оголошено після специфікатора **private** до кінця структури або до специфікатора **public**, буде приховано від користувача. Наприклад, в оголошенні структури:

```
   struct MyStr {
    int x, y;
```



```
    int Get ();
   private:
    int a, b;
    void F ();
};
```

дані **x, y** і функція **Get** – відкриті і можуть використовуватися під час роботи зі структурою, а дані **a, b** і функція **F** – закриті і ними може користуватися тільки функція **Get**.

10.10. Класи

10.10.1. Оголошення класу

Клас – це тип даних, визначений користувачем. Поняття клас, структура та об'єднання в С++ доволі близькі. Тому майже все, що говоритимемо про класи, застосовується також до структур і об'єднань.

Клас повинен бути оголошений до того, як буде оголошена хоча б одна змінна цього класу. Тобто клас не може оголошуватися всередині оголошення змінної. Синтаксис оголошення класу такий:

```
   class <ім'я класу>: <список класів - батьків>
   {
   public:                    // доступно всім
   <дані, методи, властивості, події>
  protected:               // доступно тільки нащадкам
  <дані, методи, властивості, події>
 private:                  // доступно тільки в класі
   <дані, методи, властивості, події>
   } <список змінних>;
```

Наприклад:

```
  class MyClass: public Class1, Class2
 {
public:
    MyClass (int = 0);
    void SetA (int);
    int GetA (void);
private:
    int FA;
   double B, C;
protected:
   int F (int);
};
```

Ім'я класу може бути будь-яким допустимим ідентифікатором. Ідентифікатори класів, які успадковують класи бібліотеки компонентів C++Builder, прийнято починати із символу «**T**». Клас може успадковувати поля (вони називаються дані-елементи), методи (вони називаються функції-елементи), властивості, події від інших класів – своїх предків, може скасовувати якісь з цих елементів класу або вводити нові. Якщо передбачаються такі класи-предки, то в оголошенні класу після його імені ставлять двокрапку і потім подають список батьків. У наведеному вище прикладі передбачено множинне успадкування класів **Class1** і **Class2**. Якщо серед класів-предків трапляються класи бібліотеки компонентів C++Builder чи класи, що успадковують їх, то множинне успадкування заборонено.

Якщо оголошений клас не має попередників, то список класів-родичів разом з двокрапкою опускається. Наприклад:

```
class MyClass1
{
....
};
```

Доступ до декларованих елементів класу визначається тим, у якому розділі їх оголошено.

Розділ **public** (відкритий) призначений для оголошень, які доступні для зовнішнього використання. Це відкритий інтерфейс класу.

Розділ **private** (закритий) містить оголошення полів і функцій, що використовуються тільки всередині цього класу.

Розділ **protected** (захищений) містить оголошення, доступні тільки для нащадків декларованого класу.

Як і у випадку закритих елементів, можна приховати деталі реалізації захищених елементів від кінцевого користувача. Однак на відміну від закритих, захищені елементи залишаються доступні для програмістів, які захочуть робити від цього класу похідні класи, причому не потрібно, щоб похідні класи оголошувались у цьому самому модулі.

У наведеному вище прикладі через об'єкт цього класу можна отримати доступ тільки до функцій **MyClass**, **SetA** і **GetA**. Поля **FA**, **B**, **C** і функція **F** – закриті елементи. Це допоміжні дані і функція, які використовують у своїй роботі відкриті функції. Відкрита функція **MyClass** з ім'ям, що збігається з ім'ям класу – це так званий конструктор класу, який повинен ініціалізувати дані у момент створення об'єкта класу. Присутність конструктора в оголошенні класу не обов'язкова. За відсутності конструктора користувач повинен сам подбати про задання початкових значень даних-елементів класу.

Перед іменами класів-батьків в оголошенні класу також може бути вказаний специфікатор доступу (у прикладі **public**). Сенс цього специфікатора той самий, що і для елементів класу у разі успадкування:

– **public** (відкрите успадкування) можна звертатися через об'єкт даного класу до методів і властивостей класів-предків;

– **private** подібне звернення неможливе. За замовчуванням у класах (на відміну від структур) передбачається специфікатор **private**. Тому можна вводити до оголошення класу дані і функції, не викликаючи специфікатора доступу.

Все, що введено до опису до першого специфікатора доступу, вважають захищеним. Аналогічно, якщо не вказано специфікатора перед списком класів-батьків, передбачено захищене успадкування.

Оголошення даних-елементів (полів) виглядають так само, як оголошення змінних або оголошення полів у структурах:

<Тип> <імена полів>;

В оголошенні класу поля забороняється ініціалізувати. Для ініціалізації даних слугують конструктори. Оголошення функцій-елементів у найпростішому випадку не відрізняються від звичайних оголошень функцій.

Після того, як оголошено клас, можна створювати об'єкти цього класу. Якщо клас не успадковує класів бібліотеки компонентів C++Builder, то об'єкт класу створюється як будь-яка змінна іншого типу простим оголошенням.

Наприклад, оператор

```
MyClass MC, MC10 [10], * Pmc;
```

створює об'єкт **MC** оголошеного вище класу **MyClass**, масив **MC10** з десяти об'єктів цього класу і вказівник **Pmc** на об'єкт цього класу.

У момент створення об'єкта класу, що містить конструктор, можна ініціювати його дані, перераховуючи в дужках після імені об'єкта значення даних.

Наприклад, оператор

```
MyClass MC (3);
```

не тільки створює об'єкт **MC**, але й задає його полю **FA** значення 3. Якщо цього не зробити, то в момент створення об'єкта поле отримає значення за замовчуванням, що міститься в оголошенні класу прототипу конструктора.

Створення змінних, що використовують клас, можна поєднати з оголошенням самого класу, розміщуючи їх список після закриваючої клас фігурної дужки, і завершальною крапкою з комою.

Наприклад:

```
class MyClass: public Class1, Class2
{
 ...
} MC, MC10[10], * Prac;
```

Якщо створюється динамічно розташований об'єкт класу, то використовують операцією **new**.

Наприклад:

```
MyClass * PMC = new MyClass;
```

або

```
MyClass * PMC1 = new MyClass (3);
```

Ці оператори створюють десь у динамічно-розподіленій області пам'яті самі об'єкти і створюють вказівники на них – змінні **PMC** і **PMC1**.

Створити об'єкти класу простим оголошенням змінних можна лише у випадку, якщо серед предків вашого класу немає класів бібліотеки компонентів C++ Builder. Якщо ж такі предки є, то створювати вказівник на об'єкт цього класу можна лише операцією **new**. Наприклад, якщо клас оголошений так:

```
class MyClass2: public TObject
{
 ...
};
```

то створювати вказівник на об'єкт цього класу можна оператором

```
MyClass2 * P2 = new MyClass2;
```

10.10.2. Шаблони класів

У C++ можна визначати **шаблони класів**, названі також родовими (**generic**) класами, або генераторами класів. Іноді їх називають параметризованими типами, оскільки вони мають один або більше параметрів типу, що визначають налаштування шаблону класу на специфічний тип даних під час створення об'єкта класу. Для того, щоб використовувати шаблонні класи, програмісту достатньо один раз описати шаблон класу. Щоразу, коли потрібно реалізувати клас для нового типу даних, програміст повідомляє про це компілятору, який і створює вихідний код для необхідного класу.

Опис шаблону відрізняється від опису класу першим рядком

template *<**class** ідентифікатор>* ***class*** *ім'я класу*

У цьому рядку ідентифікатор є довільним ім'ям формального типу, який використовується надалі в описі шаблону.

Наприклад:

```
template <class T> class Matrix
{
    ...
};
```

Цей заголовок оголошує про створення шаблону класу **Matrix** і задає ідентифікатор **T** для формального типу даних. Цей ідентифікатор слід використовувати в описі класу замість зазначення типу відповідних даних.

Наведемо як приклад шаблон класу матриць:

```
/ / Шаблон класу матриць
 template <class T> class Matrix
{
      T * data;
      int N;        // Кількість рядків
      int M;        // Кількість стовпців
public:
      Matrix (int, int);
      ~Matrix () {delete [] data;)
      _property T Item [int i] [int j] =
      read = GetItem, write = SetItem}
private:
      T_ fastcall GetItem (int i, int j);
      void _ fastcall SetItem (int i, int j, T value);
};
// Конструктор
template <class T> Matrix <T>: Matrix (int n, int m)
{
      data = new T [n * m];
      for (int i = 0, i<n*m; i++)
       data [i] = 0,
      N = n;
      M = m;
}
template <class T> void fastcall
     Matrix <T>:: SetItem (int i, int j, T value)
{
// Запис значення value в елемент (i, j)
if ((i <1) || (i > N) || (j <1) || (j> M))
   ShowMessage ("Неприпустимі індекси (" +IntToStr
(i)+ "," +IntToStr (j) +")");
else data [(i-1) * M+ j-1] = value;
}
```

```
template <class T> T __fastcall
     Matrix <T>:: GetItern (int i, int j)
// Читання значення елемента (i, j)
 if ((i <1) || (i> N) || (j <1) || (j> M))
ShowMessage  ("Неприпустимі  індекси  ("  +IntToStr
(i)+"," +IntToStr (j) +")");
else return data [(i -1) * M+ j - 1];}
```

У самому шаблоні не вказують тип даних, що зберігаються. Створюючи конкретний екземпляр класу можна задавати будь-який тип: цілий, дійсний, комплексний тощо. Створюючи екземпляр матриці конкретного типу в програмі можна, наприклад, оператором:

```
Matrix <float> X (4,5);
```

Цей оператор створює матрицю **X** дійсних чисел, яка складається з 4-х рядків та 5-ти стовпчиків. Після імені класу в кутових дужках вказують тип, для якого створюється екземпляр класу. Компілятор замінить на цей тип (у цьому разі **float**) формальний тип **T**, використаний в описі шаблону.

Записуються і читаються елементи матриці у програмі за допомогою властивості **Items**. Наприклад:

```
X.Items [2] [3] = 1.5;
float y = X.Items [2] [3];
```

Перший з цих операторів заносить значення 1,5 до 3-го елемента 2-го рядка, а другий оператор читає це значення.

Контрольні запитання та вправи

1. Які типи належать до основних типів у С + +?
2. Який діапазон значень типу **unsigned char**?
3. Які типи належать до арифметичних типів даних?
4. У якому біті зберігається знак змінної?
5. Які існують способи оголошення рядка символів?
6. За допомогою якої функції відбувається пошук у рядку заданого підрядка?
7. Які операції визначено для типу **AnsiString**?
8. Наведіть приклад використання змінної перелічувального типу.
9. Як додати елемент до заданої множини? Наведіть приклад.

10. Яким має бути тип двох множин, щоб для них можна було застосувати певні операції?
11. Якого типу є змінна, що оголошена вказівником на певний тип?
12. Що є результатом операції непрямої адресації?
13. Що є вказівником на перший елемент масиву?
14. Що означає запис ***void**** *Pv;*?
15. Які операції можна виконувати з вказівниками?
16. Чим посилання відрізняється від вказівника?
17. Наведіть приклади задання символьному масиву початкових значень.
18. Що є елементом багатовимірного масиву?
19. Який результат поверне функція ***sizeof*** *(A)*, якщо *A* оголошено як масив десяти цілих?
20. Як захистити від несанкціонованої зміни програмою значень елементів масиву під час передавання масиву у функцію за посиланням?
21. Яка структура називається структурою зі самоадресацією?
22. Як доступитись до елементів структури?
23. Чим структури в мові С++ відрізняються від структур у мові С?
24. Яким за замовчуванням є доступ до елементів структури у мові С++?
25. Яким за замовчуванням є доступ до елементів класу в мові С++?
26. Кому доступні закриті елементи класу?
27. Кому доступні захищені елементи класу?
28. Яка мета створення шаблонних класів?

Приклади тестових питань

1. Якщо клас має назву Cat, оголосити об'єкт правильно як:
   а) int Cat;
   б) Cat Frisky;
   в) Cat :: Frisky() {…};
   г) Frisky Cat;
   д) Frisky

2. Клас – це:
а) набір змінних;
б) новий тип даних;
в) набір об'єктів;
г) набір функцій;
д) прототип функції.

3. За замовчуванням члени класу в С++ є:
а) загальнодоступними;
б) пасивними;
в) активними;
г) закритими;
д) віртуальними.

4. Які методи можуть звертатись до закритих членів класу:
а) тільки методи цього класу;
б) усі функції програми;
в) закриті методи цього класу;
г) вбудовані функції класу;
д) перевантажені функції.

5. Відкриті члени класу доступні:
а) всім функціям програми;
б) закритим методам класу;
в) захищеним методам похідного класу;
г) віртуальним функціям;
д) перевантаженим функціям.

6. Правильне оголошення класу:

а)
```
class Cat
{ int Age;
int Weight;
void Meow(); } ;
```
б)
```
Cat class
{ int Age;
int Weight;
void Meow(); } ;
```


в)
```
Cat :: class Cat()
{ int Age;
int Weight;
void Meow(); } ;
```
г)
```
Public class Cat
{ int Age;
int Weight;
void Meow(); } ;
```
д)
```
class public Cat
{ int Age;
int Weight;
void Meow(); } ;
```

7. **Frisky** є об'єктом класу Cat, **Meow()** – метод класу Cat. Із запропонованих варіантів виклику методу **Meow()** правильним є:
а) Meow();
б) Frisky.Meow();
в) Cat::Meow();
г) Cat.Meow();
д) Frisky::Meow();

8. У мові С++ використовують для виділення ділянки у динамічній пам'яті оператор:
а) malloc;
б) new;
в) inline;
г) virtual;
д) delete.

11. ПОСЛІДОВНІ КОНТЕЙНЕРИ В СКЛАДІ СТАНДАРТНОЇ БІБЛІОТЕКИ ШАБЛОНІВ

Однією з основних цілей за об'єктно-орієнтованого підходу є можливість багаторазового використання коду, наприклад, шляхом наслідування. Краще використовувати вже розроблені класи, ніж починати кожний проект із нуля. Наявність бібліотеки перевірених, ефективно працюючих класів-контейнерів може значно скоротити час розроблення проекту. Трьома основними компонентами стандартної бібліотеки шаблонів є:

1) Колекція шаблонних класів-контейнерів;
2) Колекція узагальнених алгоритмів, тобто шаблонних функцій;
3) Колекція класів-ітераторів.

Користувач класу-контейнера може:

1) створювати екземпляр цього классу;
2) викликати відкриті (**public**) методи цього класу.

У стандартній бібліотеці шаблонів узагальнені алгоритми оперують контейнерами за допомогою ітераторів. Ітератор одержує зі свого контейнера необхідну для доступу до всіх елементів цього контейнера інформацію.

Стандартна бібліотека шаблонів (STL) є складовою офіційної мови C++, схваленої Національним інститутом стандартизації США (**ANSI**). Це означає, що інтерфейси методів і функцій визначено, а визначеної реалізації немає: розроблювачі вільні реалізовувати класи й узагальнені алгоритми будь-яким способом, дотримуючи при цьому встановлені інтерфейси. Початкова реалізація, розроблена в дослідницькій лабораторії компанії **Hewlett-Packard,** є основою для інших, існуючих сьогодні реалізацій: Microsoft Visual C++, Inprise C++ Builder, Metroworks Code Warrior та інших.

Клас-Контейнер – це клас, у якому кожний об'єкт, який є екземпляром цього класу, складається з набору елементів.

Об'єкт-Контейнер – екземпляр класу-контейнера – може зберігатися в неперервній області пам'яті у вигляді масиву або у вигляді зв'язаної структури. У зв'язаній структурі кожний елемент зберігається в блоці типу struct, що називається вузлом, який також містить вказівник на інший вузол.

Практично з кожним класом-контейнером асоціюється клас-ітератор. **Ітератори** – це об'єкти, які дають можливість користувачеві працювати з об'єктом-контейнером, не порушуючи принцип абстракції даних. Більшість класів-ітераторів мають такі оператори:

operator!= // для порівняння ітератора з іншим ітератором;
operator++ // для переходу ітератора на наступну позицію в контейнері;
operator* // для повернення елемента, на який вказує ітератор.

Більшість класів-контейнерів мають також метод **begin()**, який повертає ітератор, встановлений на початок контейнера, і метод **end()**, який повертає ітератор, встановлений на елемент, що знаходиться за останнім елементом у контейнері.

Значна частина стандартної бібліотеки шаблонів являє собою інтерфейси методів для різних класів-контейнерів.

11.1. Клас vector

Вектор являє собою кінцеву послідовність елементів – таку, що:

1. Будь-який елемент у послідовності має індекс; за цим індексом здійснюються доступ до елемента або його модифікація. Час доступу є постійною величиною.

2. Операція вставки **push_back** у кінець послідовності в середньому забирає фіксований час, але **worstTime(n)** – максимальна кількість інструкцій, що виконується під час трасування операції вставки **push_back** є **O(n)**, де **n** – кількість елементів у послідовності.

3. Для операції **pop_back** видалення з кінця послідовності показника **worstTime(n)** є постійним.

4. Для довільних операцій вставки й видалення **worstTime(n)** є **O(n).**

Клас **vector**, як і всі класи в стандартній бібліотеці шаблонів, є шаблонним. Елементи можуть мати такі типи, як **int** або **double**, або можуть бути класом – таким, як клас **string** . Наприклад, можна визначити порожній вектор рядків так:

```
vector<string> fruits;
```

У векторі допускаються повторення елементів, тому якщо ми помістимо елементи «апельсини», «яблука», «виноград» і «яблука» у вектор fruits, вектор міститиме чотири елементи. Елемент «апельсини» матиме індекс 0, «яблука» –індекс 1, «виноград» –індекс 2, а «яблука» – індекс 3. Ці елементи не будуть розташовуватися за абеткою. Насправді елементи у векторі не обов'язково є порівнянними. Наприклад, нехай є вектор text, що являє собою послідовність рядків. У цьому випадку безглуздо говорити, що один рядок «менший», ніж інший рядок. Можна порівнювати індекси рядків і стверджувати, наприклад, що індекс поточного рядка менший, ніж індекс якого-небудь іншого рядка.

Клас **vector** має два шаблонні параметри:

```
template <class T,  class Allocator = allocator>
```

Шаблонний параметр **T** задає тип елементів. Параметр **Allocator** ставиться до моделі виділення пам'яті. Гнучкість, необхідна для підтримки множини моделей виділення пам'яті, є основною причиною складності реалізацій стандартної бібліотеки шаблонів. Для простоти ми припускаємо використання моделі виділення пам'яті, прийнятої за замовчуванням, що задається класом allocator і визначеної у файлі **<defalloc>**.

У класі **vector** є понад 50 методів. Інтерфейси методів не містять конкретних вимог до колекції полів або визначення методів. Такий поділ природний для принципу абстракції даних.

У табл. 11.1 наведено інтерфейси найпоширеніших методів класу.

Таблиця 11.1

Методи класу vector

Метод	Дія
vector<double> V;	V – новоутворений пустий вектор
V.push_back(5.4);	У кінець вектора V поміщається число 5.4
V.insert(itr, 3.5);	У позицію, на яку вказує itr вставляється число 3.5
V.pop_back();	З вектора V видаляється останній елемент
V.erase(itr);	Елемент, на який вказує itr видаляється
V.size();	Визначається кількість елементів у векторі
V.empty();	Повертається true, якщо вектор V не містить елементів, інакше false.
V.[3]=10.3;	Елемент з індексом 3 заміняється на значення 10.3
itr=V.begin()	itr встановлюється на перший елемент вектора V.
itr==V.end();	Повертається true, якщо itr вказує на елемент, що стоїть за останнім елементом у векторі, інакше false.
V.front()=7.7;	Елемент з індексом 0 заміняється на значення 7.7

Приклад. Вивести вмістиме вектора V.

```
for  (int i = 0;  i < V.size  (); i++)
cout << V [i]  << endl;
```

Цей приклад демонструє, як можна використовувати індекс для просування по вектору.

Переваги векторів

1. Основна перевага векторів над масивами полягає в тому, що методи для класу **vector** уже розроблено, тоді як користувач масиву повинен сам створити код, необхідний для роботи з масивом. Наприклад, щоб вставити або вилучити елемент масиву з довільної позиції, необхідно написати код для розширення або стискання використовуваного простору пам'яті. Методи **push_back** та **insert** класу **vector** роблять це автоматично. Крім того, методи **push_back** та **insert** автоматично змінюють розмір вектора, якщо поточний зайнятий вектором блок недостатньо великий.

2. Як для векторів, так і для масивів у стандартній бібліотеці шаблонів передбачено узагальнені алгоритми. І вектори, і масиви допускають доступ або модифікацію з використанням оператора індексу *operator[ ]*. Єдина перевага масиву над вектором полягає в його швидшій ініціалізації. Наприклад, можна визначити масив і відразу ж його ініціалізувати:

```
string [ ]  words = ("так",  "ні",  "можливо"};
```

3. Для векторів передбачено методи для вставки й видалення елемента, що займає будь-яку позицію в контейнері. І нарешті, ітератори векторів являють собою ітератори довільного доступу, тобто для ітератора вектора час доступу до будь-якого елемента вектора постійний.

Поля класу vector

Для реалізації класу **vector** необхідно знати, де зберігатимуться елементи. Щоб дозволити довільний доступ, необхідно мати безперервну структуру зберігання, тобто масив. Тому потрібні

вказівник **start**, що відповідає першій позиції в масиві;

вказівники **finish**, який вказує на комірку, що знаходиться за останнім елементом у векторі;

вказівник **end_of_storage**, який вказує на комірку, що стоїть за останньою коміркою, виділеною для масиву.

Визначення методів

Конструктор за замовчуванням не виділяє блок пам'яті під масив. Замість цього три поля вказівників ініціалізуються в NULL, тобто мають значення 0.

Метод **begin()** повертає **start**, метод **end()** повертає **finish**, а метод **size()** повертає різницю **finish - start**. Тому наступний код видає результат 0:

```
vector<double> V;
cout << V.size ();
```

Коли перший елемент вводять до вектора (за допомогою методу **push_back** або методу **insert**), виділяється блок у динамічно розподіленій пам'яті. Розмір цього блоку може бути різним для різних компіляторів. Вважаємо, що виділяється 1024 байти і що число типу **double** займає 8 байтів. Якщо першою операцією вставки є

```
V.push_back (7.3);
```

то виділяється масив з 126 елементів типу **double**, як показано на рис. 11.1.

Поля **start** і **finish** вказують на перший й останній елемент у масиві, а поле **end_of_storage** вказує на комірку, що стоїть відразу за останньою виділеною під масив коміркою пам'яті.

Інструкція

```
V.size ();
```

поверне значення 1, а саме, **finish - start**.

Інструкції

```
V.begin ();
V.end();
```

повернуть ітератори, встановлені на комірки з індексами 0 та 1.

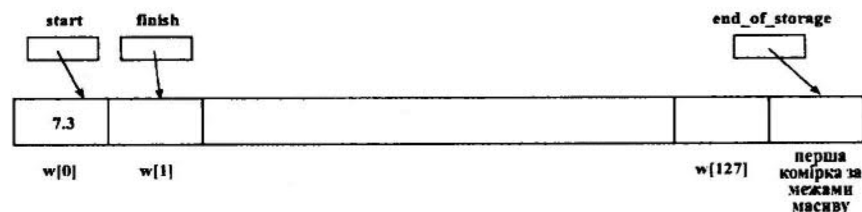


*Рис. 11.1. Вектор V після вставки в нього числа 7.3 з використанням методу **push_back***

Нехай у вектор **V** вставляється ще 127 елементів типу **double**. Розподіл пам'яті при цьому показано на рис. 11.2.

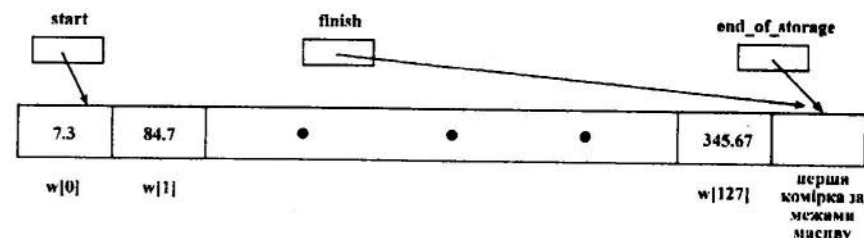


Рис. 11.2. Вектор V зі 128 елементами

Припустимо, що є така інструкція:

```
V.push_back (15.5);
```

Ознака наявності проблеми: рівність finish = end_of_storage. Елемент 15.5 не може бути збережений у поточному блоці, виділеному під вектор **V**. Не можна зберегти цей елемент у комірці, на яку вказує **end_of_storage**, оскільки ця комірка може містити змінну з якої-небудь іншої програми. Розмір базового масиву повинен бути збільшений, щоб умістити новий елемент, а це припускає виділення нового блоку динамічної пам'яті. Тоді старі елементи копіюються в новий масив, старий масив звільняється, а його елементи знищуються. Потім новий елемент вставляється в новий масив. Щоб уникнути частого перекопіювання, виділяється блок, що удвічі перевищує за своїм розміром поточний. На рис. 11.3 показано частину пам'яті, новий елемент, що вставляється і старий блок пам'яті, що звільняється.

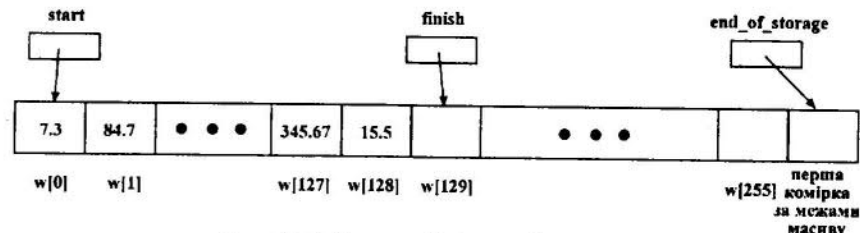


Рис. 11.3. Вектор V після зміни розміру

Якщо базовий масив об'єкта-вектора повний і здійснюється спроба вставки, ємність базового масиву подвоюється.

Реалізація методу **pop_back** потребує декрементувати вказівник **finish** і викликати деструктор для елемента, що вилучається.

В узагальненому методі **erase** здійснюється переміщення елементів для заповнення простору, що звільняється через вилучення елемента.

11.2. Клас deque

Клас **deque** описує кінцеву послідовність елементів у вигляді черги із двостороннім доступом (дек), яка має такі властивості:

1. Час доступу або модифікації для будь-якого елемента у послідовності, що має заданий індекс, є постійним.

2. Вставка в початок або в кінець послідовності в середньому забирає фіксований час, однак **worstTime (n)** є **O(n),** де **n** – кількість елементів у послідовності.

3. Для видалення з початку або з кінця послідовності показник **worstTime (n)** є постійним.

4. Для довільної вставки або видалення **worstTime (n)** є **O(n).**

Для черги із двостороннім доступом швидше здійснюються операції введення або видалення як у початок, так і в кінець послідовності, тоді як для вектора швидше здійснюються тільки операції вставки або видалення в кінець послідовності.

Єдина відмінність між вектором і чергою з двостороннім доступом полягає в такому: об'єкт типу черга із двостороннім доступом може швидко вставляти й видаляти елементи в початок і в кінець свого контейнера, тоді як об'єкт типу вектор швидко проводить лише вставку й видалення в кінець свого контейнера.

Клас **deque** не містить методів **capacity** і **reverse,** властивих класу **vector.** В іншому випадку клас **deque** і асоційований з ним клас **iterator** містить усі ті інтерфейси методів, які містить клас **vector** і його клас **iterator,** а також два додаткові методи:

```
void push_front    (const T& x);
```

Копія x вставляється в початок двосторонньої черги.

```
void pop_front ();
```

Елемент на початку двосторонньої черги видаляється.

Оскільки жодних оцінок часу виконання в умові для методу **pop_front** не наведено, вважаємо, що **worstTime(n)** постійне. Очевидно, що для черги із двостороннім доступом операції вставки і видалення в початок контейнера виконуються набагато швидше, ніж для векторів. Розглянемо деякі подібні ознаки цих двох видів контейнерів:

1. І для векторів, і для черг із двостороннім доступом будь-який елемент можна витягти або замінити за його індексом або ітератором, причому **worstTime (n)** для цих операцій постійне.

2. І для векторів, і для черг із двостороннім доступом при вставці елемента в кінець послідовності **worstTime(n) є O(n), але для n** послідовних операцій вставки в кінець послідовності **worstTime(n)** також **є O(n).**

3. І для векторів, і для черг із двостороннім доступом для операції видалення елемента з кінця послідовності **worstTime(n)** постійне.

Програма, що зазначена нижче, ілюструє застосування черги із двостороннім доступом:

```
#include <deque>
 #include <iostream>
 #include <istring>
using namespace std;
int main()
{
const string C=" Виберіть правильну відповідь";
deque<string> W;
deque<string>::iterator itr;
W.push_back      ("yes");
W.push_back      ("no");
W.push_back      ("maybe");
W.push_back      ("wow");
cout <<endl <<    "Черга після 4 вставок:" << endl;
for      (unsigned    i=0;   i<W.size();   i++)
cout << W [i]  << endl;
W.pop_front();
 W.pop_back ();
cout << endl << "Черга після видалення першого
і останнього елементів" << endl;
 for  (itr = W.begin();  itr!  = W.end(); itr++)
cout<<  (*itr)  << endl;
W.front()  = "now";
 W.back() = "but";
cout << endl << "Черга після заміни \"maybe\" на
\"now\""  <<"  i  \"yes\"  на  \"but\""  << endl  <<
*( W.begin())  <<endl << *( W.end()  - 1);
cout << endl << endl <<C; cin.get ();
```

```
return 0;
}
```

Доступ до першого елемента в **W** здійснювався трьома різними способами:

```
W [0]
W.front ()
* W.begin ()
```

Аналогічно доступ до останнього елемента також здійснювався трьома різними способами. Програма видає такий результат:

Черга після 4 вставок:

wow
maybe
yes
no

Черга після видалення першого й останнього елементів:

maybe
yes

Черга після заміни "maybe" на "now" і "yes" на "but":

now
but

Поля класу deque

У версії класу **deque** від Hewlett - Packard основне поле являє собою масив вказівників на суміжні блоки пам'яті, що зберігають елементи черги. Усі блоки мають однаковий розмір. Кількість елементів, яку може зберігати блок, становить 1 Кбайт/розмір_елемента. Масив вказівників **map**, що називається **масивом відображень,** початково має невикористані комірки на початку й в кінці масиву. Середні комірки позначатимуть блоки, які в цей момент містять елементи із двосторонньої черги.

Поля **start** і **finish** являють собою ітератори, які вказують на перший і перший після останнього елементи в черзі відповідно.

Визначення методів

Приклад. Нехай кожний блок містить п'ять елементів, а об'єкт **P** типу **deque** містить 11 елементів у такій послідовності: «яблуко», «груша», «слива», «персик», «абрикос», «вишня», « апельсин», «лимон», «ківі», «кавун», «гранат». На рис. 11.4 показано, як представляються елементи в об'єкті **P** типу **deque.** Знаками питання позначено невикористані комірки.

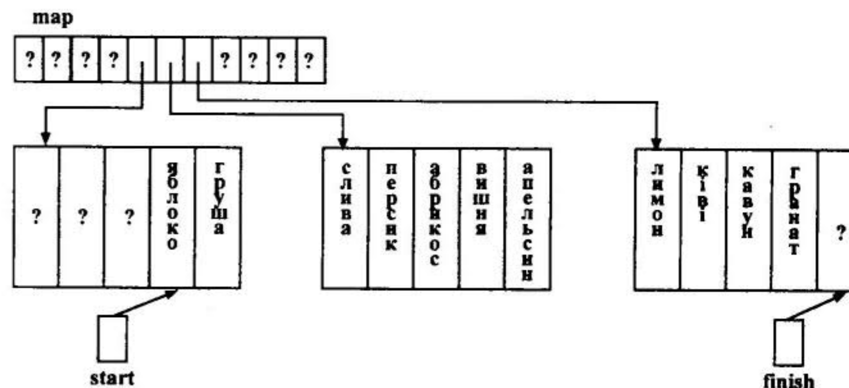


Рис. 11.4. Двостороння черга з 11 елементами

Після виконання інструкції

```
P.pop_front();
```

start вказуватиме на елемент «груша».

Після виконання інструкцій

```
P.push_back ("черешня");
P.push_back ("малина");
```

елемент «черешня» додається в кінець третього блоку, а ітератор **finish** вказуватиме на комірку, що стоїть після кінця третього блоку. За спроби додати елемент «малина» ітератор **finish** став би вказувати на комірку за кінцем блоку, що призвело б до виділення нового блоку (рис. 11.5).

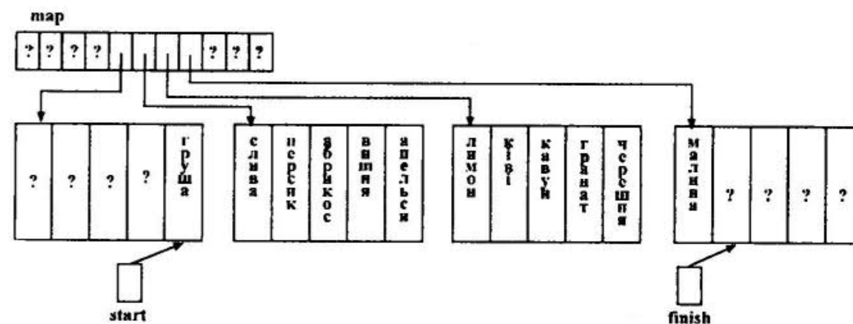


Рис. 11.5. Черга після виклику pop_front і двох викликів push_back

Поля класу **iterator** вбудовано в клас **deque.** Кожний ітератор має чотири поля. Коли ітератор встановлено на елемент **x**, то ці поля відносно елемента **x** мають такі значення:

1. **first** – вказівник на перший елемент у блоці, який містить елемент **x**;
2. **current** – вказівник на елемент **x**;
3. **last** – вказівник на комірку, що стоїть за останнім елементом у блоці, який містить елемент **x**;
4. **node** – вказівник на комірку в масиві відображень **map,** яка вказує на початок блоку, що містить елемент **x**.

Якщо для об'єкта **P** типу **deque** виконується виклик **pop_front,** перший блок використовуватися не буде, тому він звільняється, а ітератор **start** відповідно змінюється. Нижче наведено код для методу **pop_front( ):**

```
void pop_front()
{
destroy  (start.current); // виклик деструктора для
 // елемента,  на який
 // вказує start.current
++start.current;
--length;
if (empty( ) || begin (). current == begin (). last)
deallocate_at_begin(); // звільнення цього блоку
 }
```

Метод **push_front** спочатку визначає, чи потрібно виділяти новий блок на початку двосторонньої черги. Після цього **start.current** декрементується для позначення наступної доступної комірки на початку нового блоку. Нижче наведено відповідний код:

```
void push_front(const T& x)
{
if  (empty() || begin().current ==
begin().first) allocate_at_begin();
--start.current;
construct  (start.current,  x); // копіювання
об'єкта x
   // у комірку,  на яку
  // вказує start.current
++length;
}
```


Якщо всі вказівники у відображенні **map** установлені на використані блоки, а потрібен ще один блок, розмір масиву **map** буде подвоєний, а старі вказівники встановлено на середину нового масиву відображень **map.** Отже, оцінка складності для часу виконання операцій вставки у двосторонню чергу такі самі, що й для вектора: **worstTime (n)** лінійно залежить від **n.**

Зміна розміру виділеної області пам'яті стосується тільки масиву відображень **map:** жодного з використовуваних блоків не буде змінено. Для класу **vector** збільшення розміру порівняно з поточним розміром викликає копіювання всіх елементів вектора в новий масив. Яка-небудь реорганізація елементів може знадобитися тільки в тому випадку, якщо здійснюється операція вставки або видалення в комірку, відмінну від початкової або кінцевої комірки черги. Переміщення елементів сполучне з витратами, особливо якщо таких елементів багато або якщо кожний елемент має доволі великий розмір. Якщо елемент, що видаляється, розташований ближче до початку черги, то попередні елементи переміщуються донизу. Якщо елемент, що видаляється, розташований ближче до кінця черги, то наступні елементи переміщуються догори. Отже, при видаленні (або вставці) у двосторонню чергу в середньому переміщається лише четверта частина всіх елементів.

Переваги черги із двостороннім доступом:

1. За зміни розміру переміщень елементів не відбувається. Якщо для вставки в початок або в кінець двосторонньої черги потрібен додатковий блок, елементи в черзі переміщувати не потрібно. У тих рідких випадках, коли масив відображень **(map)** сам по собі занадто малий, створюється нова таблиця відображень, а стара копіюється в середину нової таблиці, але елементи при цьому не переміщаються. Якщо використовується вектор, зміна розміру спричиняє переміщення всіх елементів.

2. Для методу **pop_front** класу **deque worstTime (n)** постійне. У класі **vector** метод **pop_front** не передбачено, оскільки для нього **worstTme**(n) лінійно залежало б від **n**. Щоб вилучити перший елемент із вектора **V,** слід виконати таку дію:

```
V.erase  (V.begin());
```

При цьому **worstTime (n)** лінійно залежить від **n.**

3. Для черги із двостороннім доступом час виконання методу **push_front** у середньому постійне. У класі **vector** метод **push_front** не передбачено. У разі використання двосторонньої черги існує можливість, що розмір масиву відображень **(map)** буде змінено у

результаті виконання методу **push_front,** і в цьому випадку **worstTime (n)** є **O(n)**, оскільки розмір масиву відображень пропорційний кількості елементів у черзі.

4. При використанні двосторонньої черги невикористані блоки звільняються. Якщо черга стискається так, що один із зовнішніх блоків більше не використовується, цей блок звільняється. Нагадаємо, що вектори лише розширюються, але ніколи не стискаються.

5. При вставці й видаленні в чергу із двостороннім доступом потрібно менше переміщень даних, ніж для вектора. Наприклад, якщо елемент, що видаляється, розташований ближче до початку черги, переміщуються тільки елементи, розташовані до точки видалення. А якщо ні, то переміщаються тільки елементи, розташовані після точки видалення. У середньому буде переміщено тільки 25 відсотків усіх елементів проти 50 відсотків для вектора.

Великий недолік черг із двостороннім доступом полягає в необхідності застосування арифметичних дій за модулем для перетворення індексу на адресу блоку. Фактично, якщо тільки значна частина операцій не виконується над елементами, розташованими на початку контейнера, вектор матиме більшу швидкодію, ніж двостороння черга.

11.3. Клас list

У розумінні комп'ютерної обробки **список**, або **зв'язаний список** являє собою кінцеву послідовність елементів, таку що:

1) час доступу або модифікації для довільного елемента лінійно залежить від позиції цього елемента;

2) якщо ітератор встановлено на певну позицію в послідовності, то час, що витрачається на вставку або видалення елемента в цій позиції, буде постійним.

У C++ список реалізований у контейнері **list.** [7] Припустімо, **L** є екземпляром класу **list**. Необхідно здійснити доступ до елемента, що стоїть у списку **L** на **k** позиції від початку. Послідовно просуваємося або з початку списку **L** догори (тобто за збільшенням номерів позицій) до позиції **k** або з кінця списку **L** донизу (тобто за зменшенням номерів позицій) до позиції **k** залежно від того, який шлях коротший:

```
if  (k < L.size()/2)
{
// просування вперед з початку списку L
```



```
  itr = L.begin();
  for  (int i=0;  i < k; i++)
   itr++;
}
else
{
// просування назад з кінця списку L
  itr = L.end();
  for (int i = lis.size ();  i > k; i--)
    itr-;
}
```

Час такого доступу пропорційний **k**. Нехай **n** – кількість елементів у списку. У найгіршому випадку, коли **k** = **n**, кількість ітерацій у циклі становить **n/2**, тобто спостерігаємо лінійну залежність від **n**. Середня відстань від **k** до початку або до кінця списку дорівнює **n/4**, тобто також лінійно залежить від **n**.

Час доступу для списків не є постійним, як для векторів і двосторонніх черг, оскільки ітератори списків не є ітераторами довільного доступу, а являють собою двонапрямлені ітератори. Це означає, що із заданої позиції в списку ітератор може переходити на одну позицію вперед або на одну позицію назад. На противагу цьому ітератор довільного доступу може переміщуватися вперед або назад на будь-яку кількість позицій.

Однак після того, як ітератор встановлено на потрібну позицію, операції вставки або видалення в цю позицію забирають постійний час, тоді як для вектора і двосторонньої черги цей час лінійно залежить від номера позиції. Через це віддають перевагу списку над вектором (або двосторонньої чергою) у випадку, коли в програмі потрібно здійснювати багато операцій вставки й/або видалення в позиції, що не є кінцевими позиціями в контейнері.

Списки можна об'єднувати, причому час виконання такого об'єднання буде постійним. Як приклад розглянемо список **L1**, що містить елементи «телевізор», «радіоприймач», «стереосистема», «CD-плеєр». Нехай ітератор **itr** встановлено на елементі «радіоприймач». Якщо список **L2** містить елементи «магнітола», «відеомагнітофон», «лазерний програвач», то в результаті виконання інструкції

```
L1.splice (itr. L2);
```

елементи, що належать списку **L2**, буде вилучено з **L2** і вставлено в список **L1** перед елементом «радіоприймач». Отже, **L1** міститиме «телевізор», «магнітола», «відеомагнітофон», «лазерний програвач»,

«радіоприймач», «стереосистема», «CD-плеєр», а список **L2** буде порожній.

Інтерфейси методів класу **list** і асоційованого з ним класу **iterator** схожі з інтерфейсами методів для векторів і двосторонніх черг. Перелік цих методів та їх стислі характеристики наведено в табл. 11.2 [7]. Клас **list** є шаблонним класом з шаблонним параметром **T**, що позначає тип елементів у списку.

Таблиця 11.2

Методи класу list

Метод	Дія
list<double> x	Створюється пустий список **x**.
list<double> W(x)	Об'єкт **W** типу **list** містить копію об'єкта **x** типу **list**.
W.push_front (5.3)	5.3 поміщається з початку списку **W**.
W.push_back (10.5)	10.5 поміщається в кінець списку **W**.
W.insert (itr, 112.0)	112.0 поміщається в позицію ітератора **itr** , всі наступні елементи переміщуються в кінець списку **W**.
W.pop_front ()	Перший елемент у списку **W** видаляється.
W.pop_back ()	Останній елемент у списку **W** видаляється.
W.erase (itr)	Елемент, на який вказує **itr**, видаляється.
W.erase (itr1, itr2)	Видаляються елементи зі списку **W**, починаючи з позиції **itr1** до позиції **itr2,** не рахуючи останній.
W.size ()	Визначається кількість елементів у списку **W.**
W.empty ()	Якщо список **W** пустий, то повертається **true**, інакше – **false**.
itr = W.begin ()	Ітератор **itr** встановлюється на перший елемент у списку **W**.
itr = W.end ()	Якщо ітератор **itr** встановлений на позицію за останнім елементом списку **W**, то повертається **true**, інакше – **false.**
New_W = W	Попередньо визначений об'єкт **New_W** типу **list** містить копію списку **W.**
W.splice (itr, Old_W)	Всі елементи списку **Old_W** розміщуються в списку **W** перед позицією, на яку вказує ітератор **itr**.
W.sort ()	Елементи в списку **W** розміщуються в послідовності, що задає **operator<.**

Крім того, є також методи **front** і **back**, інтерфейси яких такі самі, що й для версій цих методів у класі **vector**.

Клас **list** підтримує двонапрямлені ітератори, про що свідчить відсутність оператора operator+.

Інтерфейси ітератора наведено нижче:

1. Ітератор встановлено на наступну позицію у списку, і повертається посилання на заданий ітератор.

iterator& **operator**++();

2. Ітератор встановлено на наступну позицію у списку, і повертається копія попереднього значення цього ітератора.

iterator& **operator**++ (int);

3. Ітератор встановлено на попередню позицію в списку, і повертається посилання на цей ітератор.

iterator& **operator--();**

4. Ітератор встановлено на попередню позицію в списку, і повертається копія колишнього значення цього ітератора.

iterator& **operator--();**

5. Ітератор встановлено на елемент у списку. Повертається посилання на елемент, на який встановлено цей ітератор.

T& **operator***();

6. Повертається **true**, якщо цей ітератор встановлено на ту саму позицію в списку, що й ітератор x. У протилежному випадку повертається **false**.

bool operator== (**const** iterator& x);

7. Також існує оператор operator!=, який порівнює поточне значення ітератора із заданим числом.

Нижче наведено програму, яка ілюструє декілька методів та ітераторів для роботи зі списками:

```
#include <list>
#include <iostream>
#include <string>
using namespace std;
int main ()
{
list<string> words;
list<string>::iterator itr;
words.push-back ("Так");
words.push-back ("Hi");
words.push-front ("Можливо");
words.push-front  ("Як?");
```

```
cout << "розмір дорівнює" << words.size << endl;
cout << endl << "Список після 4 вставок:" << endl;
for (itr = words.begin();  itr! = words.end(); itr++)
     cout <<  (*itr)  << endl;
words.pop_front();
words.pop_back();
cout << endl << "Список після 2 видалень:" << endl;
for (itr = words.begin(); itr! = words.end(); itr++)
     cout <<  (*itr)  << endl;
cin.get ( ) ;
return 0;
 }
```

Програма видає такий результат:
розмір = 4
Список після 4 вставок:
Як?
Можливо
Так
Ні
Список після 2 видалень:
Можливо
Так

11.3.1. Відмінності між методами списків і методами векторів та двосторонніх черг

Найзначнішою відмінністю між методами класу **list** і методами класу **vector** або **deque** є відсутність у класі **list** оператора індексу, **operator**[]. Його відсутність пояснюється неможливістю довільного доступу для елементів списків. Дію оператора індексу імітують послідовним просуванням від початку або від кінця списку, але час виконання при цьому лінійно залежатиме від кількості елементів між початковим (або кінцевим) елементом і елементом із заданим індексом.

Необхідно вибирати вектор або двосторонню чергу, якщо в програмі здійснюється доступ або модифікація елементів, позиції яких у послідовному контейнері істотно варіюються, оскільки вектори й двосторонні черги мають вищу швидкодію, ніж списки, тому їх використовувати краще.

З іншого боку, час виконання операції вставки або видалення елемента, на якому встановлено ітератор, постійний для списку, але лінійно залежить від кількості елементів для вектора або черги із двостороннім доступом.

Необхідно вибирати список, якщо в програмі здійснюється обхід послідовного контейнера і при цьому виконуються операції вставки або видалення – тоді швидкодія програми буде вищою, ніж з використанням вектора або двосторонньої черги.

Інша відмінність між класом **list** і класом **vector** та **deque** полягає в границях визнання ітератора недійсним у результаті виконання операцій вставки або видалення. У разі вставки видалення в списку визнаються недійсними тільки очевидні ітератори.

Наприклад, припустимо, що **lis** – це об'єкт списку, а **itr1** і **itr2** – його ітератори. Якщо **itr1** установлено на **item1**, а **itr2** установлено на який-небудь елемент, розташований нижче, після виконання інструкції

```
lis.erase  (itrl);
```

буде визнаний недійсним ітератор **itr1**. Це означає, що **itr1** більше не залежатиме від вказівника на **item1**. Але **itr2** буде, як і раніше, вказувати на той самий елемент, на який він вказував перед виконанням методу **erase**.

Припустимо, що є деякий сценарій для вектора **vec**, і виконується інструкція

```
vec.erase  (itrl);
```

Після цього **itr2** більше не вказуватиме на елемент, на який він вказував до виклику методу **erase**. Причина в тому, що під час операції видалення у векторі звільнюються елементи, розташовані за точкою видалення. Схожа проблема виникає під час видалення елемента із черги із двостороннім доступом. Ітератор **itr1** також вважатиметься недійсним, оскільки метод **erase** викликає деструктор для елемента, на якому встановлений **itr1**.

Подібний статус ітератори мають і після операції вставки. Для списку переміщень елементів не відбувається, тому ітератори, як і раніше, вказують на те, на що вони вказували до виконання вставки. Для векторів під час вставки може знадобитися переміщення елементів, тому ітератори можуть виявитися недійсними. Зокрема, якщо новий розмір перевищує колишній обсяг вектора, відбувається розширення, і всі ітератори й посилання визнаються недійсними, а якщо – ні, то визнаються недійсними тільки ітератори й посилання, що вказують на точку вставки або за нею. Для двосторонніх черг ситуація схожа, за ви-

нятком того, що недійсними визнаються ітератори й посилання або від точки вставки до початку черги, або від точки вставки до кінця черги, залежно від того, чи ближче точка вставки до початку або до кінця.

Розглянемо одну з реалізацій класу **list** зі стандартної бібліотеки шаблонів, а саме, версію цього класу Hewlett-Packard [7].

Поля класу list

Найважливішими є поля **length** і **node**, визначені так:

```
template<class T>
class list {
            protected:
            unsigned length;
        struct list_node
         {
          list_node* next;
          list_node* prev;
          T data;  //
         };
list_node* node;
```

На рис. 11.6 показано, що вузли списку пов'язані між собою вказівниками.

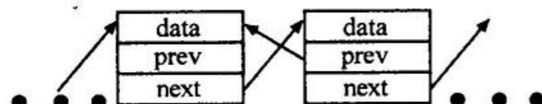
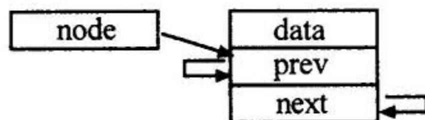


Рис. 11.6. Двонапрямлений список

list_node вказує на вузол, який називається ***вузлом заголовка.*** У вузлі заголовка поле **data** не використовується, а поля **prev** і **back** спочатку вказують на сам вузол заголовка. Тому, конструктор за замовчуванням містить код:

```
(*node).next = node;
(*node).prev = node;
```

Після виклику конструктора за замовчуванням матимемо:



Для непорожнього списку поле **next** вузла заголовка вказує на перший елемент у списку, а поле **prev** – на останній елемент у списку. Отже, список зберігається у вигляді закільцьованого списку з подвійними зв'язками.

Визначення методів

Перш ніж розглядати визначення деяких методів класу **list**, розглянемо вбудований клас **iterator**. Цей клас містить два захищені (**protected**) члени: поле й конструктор

```
protected:
list_node* node;
iterator  (list_node*x):node(x) {}
```

У цьому заголовку конструктора є ***розділ ініціалізації конструктора:*** двокрапка, за якою йде будь-яке число записів ініціалізації полів, розділених комами. Кожний запис ***ініціалізації поля*** складається з ідентифікатора поля й початкового значення, що подане у фігурних дужках. У цьому випадку поле **node** ініціалізується як **x**.

Причина оголошення конструктора як **protected** полягає в тому, що звичайним користувачам не буде нічого відомо про **list_node**, тому в них не буде підстав викликати цей конструктор. Є також відкритий (**public**) конструктор за замовчуванням.

Наприклад,

```
public:
iterator(){ }
T& operator*()  const { return  (*node) data;}
 iterator& operator++ ()
{
  node =  (*node).next;
  return *this;
}
iterator operator++ (int)
{
 iterator tmp = this;
 ++*this;
 return tmp;
}
```

Розглянемо методи класу **list**. Визначимо вісім методів: **begin, end, insert, push_front, push_back, erase** (з одним параметром), **pop_front** і **pop_back**.

Метод **end**

На вузол заголовка вказує поле **next** останнього вузла типу **list_node**. Отже, вузол заголовка перебуває на одній позиції за останнім вузлом типу **list_node** у списку. Тоді метод **end** класу **list** має

повертати ітератор, встановлений на вузол заголовка. Це відповідає визначенню:

```
iterator end() {return node;}
```

Значенням, що повертається, є **node**, тобто вказівник на вузол заголовка. Але типом результату для методу **end** є **iterator**. Ітератор має поле вказівника (яке також має ім'я **node**), але ітератор є об'єктом, а не вказівником. У C++, якщо потрапляється вираз невідповідного типу, компілятор виконуватиме ***автоматичне перетворення типу*** до відповідного типу. У цьому випадку тип **list_node*** повинен бути перетворений до типу **iterator**. Автоматичне перетворення типу може бути виконане завдяки захищеному (**protected**) конструктору:

```
iterator  (list_node* x):node(x) {}
```

Тобто, метод **end** реально повертає не копію поля **node** класу **list**, а ітератор, сформований з поля **node** класу **list**.

Метод **begin**

Метод повинен повертати ітератор, встановлений на той самий вузол типу **list_node**, на який указує поле **next** вузла заголовка, а саме на вузол типу **list_node**, який містить перший елемент у списку. Визначення методу **begin**, у якому також застосовується автоматичне перетворення типу, має такий вигляд:

```
iterator begin() {return (*node).next;}
```

Метод **insert**

```
iterator insert (iterator position, const T& x);
```

Цей метод зберігає елемент **x** у вузлі типу **list_node**, змінює вказівники **next** і **prev** так, щоб вузол **list_node** розташовувався «перед» вузлом списку, на який указує ітератор **position**, і повертає ітератор, встановлений на вставлений вузол.

При вставці в контейнер типу **list** жодні елементи не переміщуються. Отже, для реалізації методу **insert** необхідно виконати такі дії:

1. Виділити простір для вузла типу **list_node**, на який вказує **tmp**.
2. Зберегти елемент **x** у поле даних вузла **tmp**.
3. Присвоїти полю **next** вузла **tmp** значення **position.node**.
4. Присвоїти полю **prev** вузла **tmp** значення поля **prev** вузла типу **link_node**, на який вказує **position.node**.
5. Присвоїти значення **tmp** полю **next** вузла типу **link_node**, на який вказує поле **prev** вузла типу **link_node**.
6. Присвоїти значення **tmp** полю **prev** вузла типу **link_node**, на який вказує **position.node**. Це присвоєння повинне бути виконане після присвоєнь 4 і 5, а якщо – ні, то вузол, який передує вузлу, на який вказує **position**, буде недоступний.

7. Інкрементувати довжину **length**.
8. Повернути **tmp**.

Визначення методів **push_front** і **push_back** займають один рядок:

```
void push_front (const T&, x) {insert (begin (), x) ; }
void push_ back (const T& x) {insert (end (),  x) ; }
```

Стислість цих двох визначень ілюструє переваги, які надає наявність вузла заголовка: метод **insert** також управляє вставками в початок і в кінець. Метод **insert** завжди вставляє новий елемент між двома вузлами списку. Для методу **push_front** елемент вставляється між вузлом заголовка й першим вузлом. Для методу **push_back** елемент вставляється між останнім вузлом й вузлом заголовка.

Завдяки наявності вузла заголовку, кожний вузол списку має вказівник на попередній вузол списку і на наступний вузол. Це спрощує вставки і видалення.

Метод **erase** відключає елемент від списку.

Метод **pop_front** видаляє елемент, на який указує ітератор **begin()**.

Метод **pop_back** видаляє елемент, на який указує ітератор **end()**.

```
void pop_front()    {erase (begin()};
void pop_back()
{
iterator tmp = end();
erase(--tmp);
}
```

При першій вставці в список виділяється блок пам'яті – зазвичай 1 Кбайт. Цей блок називається ***буфером.*** Він використовується для наступних вставок доти, поки буфер не буде заповнений. Для визначення моменту заповнення буфера в класі **list** передбачене поле **next_avail**, що вказує на вузол, який використовуватиметься під час наступної операції вставки, і поле **last**, яке вказує на позицію за останнім вузлом у буфері:

```
list_node* next_avail;
list_node* last;
```

Поле **next_avail** використовується при виділенні нового вузла, яке відбувається в результаті викликів **insert, push_front** або **push_back**.

Коли буфер виявиться заповненим, тобто коли **next_avail = last**, виділяється новий буфер такого самого розміру. Для зберігання

інформації про всі буфери списку в класі **list** передбачене поле **buffer_list**. Це поле містить вказівник на список з одинарним зв'язком, вузли якого типу **list_node_buffer**. У кожному вузлі типу **list_node_buffer** є два поля: вказівник на буфер і вказівник на наступний вузол типу **list_node_buffer**:

```
struct list_node_buffer
{
list_node_buffer* next_buffer;
 list node* buffer;
};
list_node_buffer* buffer_list;
```

Вилучені вузли організовано у вигляді списку з одинарним зв'язком. Поле **free_list** у класі **list** містить вказівник на останній вилучений вузол:

```
list_node* free_list;
```

Поле **prev** цього вузла ігнорується, а поле **next** вказує на вузол, який вилучено передостаннім. Поле **next** вузла, вилученого передостаннім, вказує на третій з кінця вилучений вузол. Вилучений вузол додається до списку вільних вузлів за методом **put_node**:

```
void put_node (link_type p)
{
 p -> next = free list;
 free_list = p;
 };
```

Під час виконання операції вставки – за допомогою методів **push_front** або **push_back** – перевіряється список вільних вузлів. Якщо він не порожній, його перший вузол використовується для вставки до списку вільних вузлів або видалення з нього. Якщо ж список вільних вузлів порожній, виділяється новий буфер і зв'язується з початком списку буферів, після чого виділяється перший вузол у цьому новому буфері.

Простір, що зайнятий різними буферами й списками, не звільняється, поки список не буде знищено. Тому, якщо у програмі створюється великий список, а потім з нього видаляються майже всі елементи, весь зайнятий списком простір, як і раніше, залишиться виділеним.

Застосування вузла заголовка дає змогу не розрізняти операції вставки в початок і в кінець списку або операції видалення з початку

й з кінця. Для кожного конкретного вузла завжди є вузол перед ним і вузол після нього. Зокрема вузол заголовка завжди розташовується перед першим вузлом у списку, але вузол заголовка також завжди розташовується за останнім вузлом у списку.

11.4. Клас queue

Нагадаємо, що **черга** – це кінцева послідовність елементів, для якої:

1. Вставки допускаються тільки в кінець послідовності.
2. Видалення, витяг й зміни допускаються тільки з початку послідовності (див. розділ 3.2).

Елементи в черзі зберігаються в хронологічному порядку їх введення: першим зайшов, першим вийшов FIFO (first in, first out).

Елементи в черзі зберігаються в послідовності їх введення. Перший введений елемент – у кінець черги – буде у результаті першим елементом для виконання операцій видалення, витягу або зміни – з початку черги. Елемент, вставлений другим, буде другим для виконання операції видалення, витягу або зміни і т.д.

Прикладами черг можуть бути машини перед світлофором, покупці до каси в супермаркеті, літаки, що очікують дозволу на зліт.

Як і інші класи-контейнери в стандартній бібліотеці шаблонів, клас **queue** є шаблонним класом:

```
template <class T,  class container = deque<T>>
```

На додаток до типу елемента **T**, для кожного екземпляра черги також є шаблон контейнера зі значенням за замовчуванням **deque<T>**. Ідея полягає в тому, що клас **deque** може управляти визначенням атрибутів черги: вставка в кінець, видалення з початку, доступ до початку – за постійного, в середньому, часу виконання.

Методи класу deque

Розглянемо інтерфейси методів класу **deque**:

1. Ця черга ініціалізована копією цього контейнера.

```
explicit queue (const containers = Container ());
```

Зарезервоване слово **explicit** використовується тільки конструкторами. Воно вказує компілятору, що автоматичне перетворення типу для аргументу конструктора виконувати не слід. Аргумент конструктора повинен являти собою об'єкт-контейнер у класі

Container, який задається другим шаблонним аргументом. Приклади допустимих визначень черги:

a. `queue<string> ql;`

b. `queue<string>, deque <string> > q2(ql);`

c. `queue<string> q3(ql);`

d. `queue<double, list <double> > q4;`

e. `queue<double, list <double> > q5(q4);`

Визначення ql, q2 і q3 еквівалентні, як і визначення q4 і q5. З іншого боку, наступні два визначення є неприпустимими:

f. `queue<string>, list <string> > q6(ql);`

Тут аргументом конструктора повинен бути список (**list**)

g. `queue<double> > q7(q4);`

А тут аргументом конструктора повинна бути двостороння черга (**deque**).

2. Метод **empty**() повертає значення **true**, якщо ця черга порожня, а якщо ні, то повертається значення **false.**

```
bool empty() const;
```

3. Метод **size** () повертає кількість елементів у черзі.

```
unsigned size () const;
```

4. Метод **push** вставляє елемент **x** у кінець цієї черги

```
void push (const value_type& x);
```

Цей метод часто називають «поставити в чергу». Відповідно до оголошення **typedef**, **value_type** є синонімом **T**.

5. Для цієї не порожньої черги метод **front**() повертає посилання на елемент на початку черги.

```
T& front();
```

Оскільки значення, що повертається, являє собою посилання, цей метод можна використати для зміни початкового елемента в черзі. Наприклад, якщо **my_queue** є непуста строкова черга, то

```
my_queue.front() = "Група";
```

приведе до заміни початкового елемента в my_queue на «Група».

6. Для цієї не порожньої черги метод **front**() повертає посилання-константу на елемент на початку черги.

```
const T& front();
```

Оскільки значення, що повертається, є посиланням-константою, цей метод не можна використати для зміни елемента в черзі. Однак можна використовувати цей метод для витягу початкового елемента. Наприклад, якщо **my_queue** являє собою непусту чергу, то в результаті виконання інструкції

```
cout << my_queue.front();
```


буде виведено початковий елемент у черзі **my_queue**.

7. Для непорожньої черги елемент, який перед цим викликом перебував на початку черги, видаляється із черги

```
void pop();
```

Метод **pop** не повертає видаленого елемента. Щоб отримати цей елемент, слід викликати функцію **front**( ) перед викликом **pop**( ). Цей метод часто називають «вивести із черги».

8. Для непорожньої черги метод **back**() повертає посилання на елемент наприкінці черги.

```
T& back();
```

Відповідно до визначення черги, метод **back** не потрібен, але цей метод є в стандартній бібліотеці шаблонів. Цей метод може використовуватися для зміни останнього поміщеного в чергу елемента.

9. Для непорожньої черги метод **back**() повертає посилання-константу на елемент наприкінці черги

```
const T& back();
```

Для всіх методів, за винятком **push**, **worstTime(n)** постійне. Для методу **push worstTime(n) є O(n)**.

Клас **queue** не має асоційованого з ним класу ітератора. Річ у тім, що, згідно з визначенням черги, єдиним доступним елементом в об'єкті типу **queue** є елемент, розташований на початку черги.

Дизайн і реалізація класу **queue** є складовими стандартної бібліотеки шаблонів і не можуть вибиратися розробником компілятора за його розсудом.

Клас **queue** являє приклад адаптера контейнера. Адаптер контейнера C++ трансформує деякий базовий контейнер у контейнер класу C++. Адаптери контейнерів – **queue**, **stack** і **priority_queue** – обслуговуються не так, як інші компоненти стандартної бібліотеки шаблонів. Їхні визначення методів повинні викликати методи базового контейнера.

Адаптер контейнера C++ використовує деякий базовий об'єкт-контейнер для визначення методів класу C++.

Для класу **queue** усе, що потрібно від базового класу **Container**, це підтримати методи **empty, size, front, push_back** і **pop_ front**, а також методу **back**. Частину визначень класу **queue** зі стандартної бібліотеки шаблонів наведено нижче **[7]:**

```
template <class T,  class container = deque<T>>
class queue
 {
```

```
protected:
Container c;
 public:
void pop()    {c.pop_front();}
 const T& front ()  const {return c.front();
}
```

Як базовий можна розглядати клас **list**, який містить методи **size, empty, push_back, pop_front, front** і **back**.

Базовим може також бути клас **queue**; насправді саме цей клас і використовується за замовчуванням. У реалізації класу **queue** Hewlett-packard час виконання методів **size, empty, pop_front, front** і **back** постійний.

Клас **vector** не можна використати як базовий: у ньому відсутній метод **pop_front**. Такий навмисний пропуск попереджає користувач, що для операції видалення початкового елемента з вектора показник **worstTime(n)** лінійно залежить від **n**.

Інша можливість полягає у використанні масивів. За безпосереднього маніпулювання полями масиву час виконання для викликів **push_back** і **pop_front** у середньому постійний. Такий базовий класс – **queue-array** – матиме п'ять полів:

```
T[] data;
unsigned size;
int head, tail, max_size;
```

Масив **data** зберігає елементи, змінна **size** зберігає кількість елементів, а **max_size** максимальну кількість елементів. Початковий елемент завжди має індекс **head**, але **head** не завжди має значення **0**. По суті, черга «сковзає» у масиві **data** при викликах **push_back** і **pop_front**. Поле **head** містить індекс у масиві **data** для початкового елемента черги, а поле **tail** містить індекс для кінцевого елемента в черзі. **tail** ініціалізується як **-1**, щоб зазначити, що черга порожня.

11.5. Клас stack

Нагадаємо, що **стек** – це така, кінцева послідовність елементів, що єдиним елементом, який може бути видалений, вилучений або змінений, є елемент, поміщений у послідовність останнім. Цей елемент називається «верхнім» елементом стека, або вершиною стека.

Елементи в стеці зберігаються в порядку, зворотному до хронологічної послідовності їх включення: «останнім зайшов, першим вийшов».

Для стеків операція включення називається «розміщенням» або «проштовхуванням» («push»), а операція видалення – «вилученням» або «виштовхуванням» («pop»).

У класі **stack** ще менше інтерфейсів, ніж у класі **queue**, оскільки для стека може бути вставлений, вилучений або змінений тільки елемент в одній позиції (верхній).

Клас **stack** є шаблонним, а як контейнерний клас за замовчуванням використовується **deque**:

```
template < class T,  class Container = deque <T>>
class stack { …};
```

Інтерфейси методів класу stack

1. Цей стек порожній, тобто не містить елементів.

```
explicit stack  (const Containers = Container ());
```

2. Повертається **true**, якщо стек порожній. А якщо ні, то повертається **false**.

```
bool empty();
```

3. Повертається кількість елементів у стеці.

```
unsigned size ();
```

4. Елемент **x** розміщується у вершині стека.

```
void push  (const T& x);
```

5. Стек не порожній. Повертається посилання на верхній елемент у стеці.

```
T& top();
```

6. Стек не порожній. Вертається посилання-константа на верхній елемент у стеці.

```
const T& top()const;
```

7. Стек не порожній. Елемент, який до виклику цього методу перебував у вершині стека, видаляється.

```
void pop();
```

Цей метод не повертає вилученого зі стека елемента. Якщо вам потрібний доступ до вилученого елемента, то викличте метод **top**() перед викликом **pop**().

Для стеків не передбачено ітератора, оскільки єдиним доступним елементом є елемент, що перебуває у вершині стека. Це означає, що вміст стека, наприклад, не можна вивести на друк. Однак під час виведення на друк не можна використовувати які-небудь властивості стека, крім тих, які описані в інтерфейсах методу. Нижче наведено приклад програми, яка генерує стек:

```
# include <iostream>
# include <vector>
# include <string>
# include <stack>
using namespace std;
void printstack   (stack<int,   vector<int> >
ages)
{
cout << endl <<endl <<"Поточний вміст стека:" <<
endl;
while  (!ages.emplty())
{
cout << ages.top()  << endl;
ages.pop();
} }

int main()
{
const string CLOSE_WINDOW =
" Будь ласка,  натисніть Enter, щоб закрити це
вікно виводу";
 stack<int,  vector<int> > ages;
ages.push(15);
ages.push(07) ;
ages.push(21);
printstack  (ages);
ages.pop() ;
printstack (ages);
ages.pop();
printstack  (ages);
ages.push (25) ;
printstack  (ages);
cout <<endl <<endl <<CLOSE_WINDOW;
cin.get();
return 0;
}
```

Як і клас **queue**, клас **stack** є адаптером контейнера, тобто методи класу **stack** визначено в термінах методів деякого базового класу-контейнера. Клас адаптера контейнера повинен мати такі методи: **empty, size, push_back, pop_back** і **back**. Це означає, що вершина стека має розташовуватись наприкінці базового контейнера.

Для векторів, списків і двосторонніх черг час виконання методів **push_back** і **pop_back** постійний у середньому. Отже, класом

stack можуть адаптуватися і клас **vector**, і клас **deque**, і клас **list**. За замовчуванням використовується клас **deque**.

Усі адаптери контейнерів у стандартній бібліотеці шаблонів містить єдине поле

```
Container c;
```

Як і для реалізації класу **queue**, усі визначення методів класу **stack** складаються із одного рядка, у якому **c** викликає відповідний метод. От два приклади визначень методів класу **stack**:

```
void pop()
{
c.pop_back ();
}
T& top
{
return  c.back ();
}
```

11.6. Клас BinSearchTree

Клас **BinSearchTree** не входить до стандартної бібліотеки шаблонів. Він є спрощеною версією класів **AVLTree** та **rb_tree**, які розглядаються в розділах 13.5 та 14.2 відповідно.

Як вже зазначено в підрозділі 6.2.1, **двійкове** або бінарне **дерево f** є або порожнім, або складається з елемента, що називається кореневим елементом, і двох складових двійкових дерев, що називаються лівим піддеревом і правим піддеревом **t**.

Позначимо ці складові піддерева як leftTree(t) (ліве) і rightTree(t) (праве). Використання нотації у вигляді функції, наприклад, leftTree(t) замість об'єктної нотації, наприклад, t.leftTree(t), правильніше, оскільки двійкове дерево не є структурою даних. Річ у тім, що для різних типів двійкових дерев для таких операцій, як вставка й видалення, використовують методи, що значно різняться, і навіть використовують різні списки параметрів. Визначення двійкового дерева є рекурсивним.

Під час зображення двійкового дерева відповідно до прийнятої угоди кореневий елемент розташовується вгорі. Для дерев застосовується «ботанічна» термінологія: корінь, гілка, листок.

Лінія від кореневого елемента до піддерева називається **гілкою**. Елемент, для якого асоційовані з ним ліве й праве піддерева є порожніми, називається **листком**. Листок не має гілок, що відходять. Кількість листків у двійковому дереві можна визначити рекурсивно.

Нехай **t** – двійкове дерево. Кількість листків у дереві **t**, позначену **як** **leaves(t)**, можна визначити рекурсивно так:

якщо (if) t порожньо
leaves(t) =0
інакше якщо (else if) t складається тільки з кореневого елемента
leaves(t)= 1
інакше (else)
leaves(t)= leaves(leftTree(t)) + leaves(rightTree(t))

Це математичне визначення. Відповідно до останнього рядка визначення, кількість листків у дереві **t** дорівнює кількості листків у лівому піддереві плюс кількість листків у правому піддереві.

Кожний елемент у двійковому дереві унікально визначається за його місце- розташуванням у дереві. Нехай **t** – це наступне двійкове дерево (рис. 11.7):

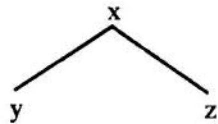


Рис. 11.7. Двійкове дерево

Ми говоримо, що **x** є батьківським елементом для **y** і що **y** є лівим дочірнім елементом елемента **x**. Аналогічно ми говоримо, що **x** є батьківським елементом для елемента **z**, а **z** є правим дочірнім елементом елемента **x**. У двійковому дереві кожний елемент має нуль, один або два дочірні елементи.

Для будь-якого елемента **w** у дереві **parent(w)** позначає батьківський елемент елемента **w**, **left(w)** – лівий дочірній елемент елемента **w**, а **right(w)** – правий дочірній елемент елемента **w**. Для дерев застосовується також термінологія, властива родовим відношенням: батьківський, дочірній, братній.

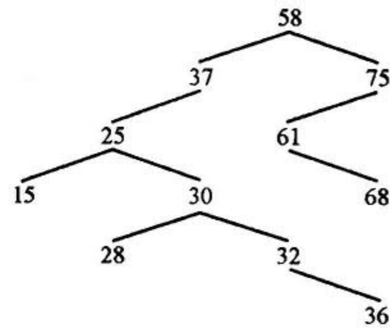


Рис 11.8. Двійкове дерево з висотою, що дорівнює 5

У двійковому дереві кореневий елемент не має батьківського, а всі інші елементи мають одного й тільки одного батька. Користуючись термінологією, прийняте в генеалогії, можна визначити елементи, які є братами, дідусями, онуками, двоюрідними братами, предками й нащадками. Наприклад, елемент A є предком елемента B, якщо B являє собою елемент у піддереві, кореневим елементом якого є елемент A.

Говорячи мовою рекурсії, A є предком B, якщо parent(A) = B, або якщо A є предком parent(B).

Якщо A є предком B, то шлях від A до B являє собою послідовність елементів, що починається з A і закінчується на B, у якій кожний елемент (за винятком останнього) є батьківським для наступного елемента. Наприклад, на рис. 11.8 послідовність 37, 25, 30.32 являє собою шлях від елемента 37 до елемента 32.

Висота двійкового дерева являє собою кількість гілок між коренем і найвіддаленішим листком, тобто листком, що має найбільшу кількість предків. Дерево на рис. 11.8 має висоту 5.

Припустимо, що для деякого двійкового дерева висота лівого піддерева дорівнює 8, а висота правого піддерева – 12. Яка буде висота всього двійкового дерева? Відповідь: 13. Взагалі висота двійкового дерева на одиницю більша за максимальну висоту серед лівого й правого піддерев. Це підводить до рекурсивного визначення висоти двійкового дерева. Але спочатку необхідно усвідомити, що таке базовий випадок, а саме – якою буде висота порожнього дерева. Хотілося, щоб висота дерева з одного елемента дорівнювала 0: у ньому немає гілок, що відходять від кореневого елемента, тобто ліве й праве піддерева є порожніми. Але якщо 0 на одиницю більше, ніж висота порожнього піддерева, то висота порожнього дерева має дорівнювати -1. Отже, висота порожнього двійкового дерева дорівнює -1.

Нехай **t** – двійкове дерево. Визначимо його висоту **height(t)** так:

якщо (if) t порожньо
height(t)=-1
інакше (else)
height(t) = 1 + максимум з {height(leftTree(t), rightTree(t)}

Як випливає з цього визначення, двійкове дерево з єдиним елементом має висоту 0, оскільки кожне зі складових його порожніх піддерев має висоту -1. Висота є властивістю всього двійкового дерева. Для кожного елемента у двійковому дереві

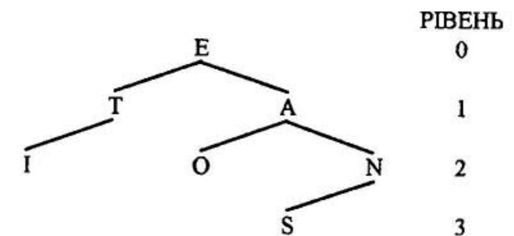


Рис 11.9. Двійкове дерево із зазначеними рівнями його елементів

можна визначити подібну ознаку, яку називають **рівнем** елемента. Рівень цього елемента в дереві являє собою кількість гілок між

кореневим елементом і цим елементом. На рис. 11.9 наведено двійкове дерево із зазначенням рівнів його елементів.

Кореневий елемент має рівень 0, а висота дерева дорівнює найбільшому рівню, який є в дереві. Формальне визначення таке. Нехай **t** – непусте двійкове дерево; тоді для будь-якого елемента **x** у **t** рівень елемента x, **level(x)** визначається так:

якщо (if) x є кореневий елемент,
level(x) = 0
інакше (else)
level(x) = 1 + level(parent(x))

Рівень елемента також називають **глибиною** елемента. Висота непустого двійкового дерева дорівнює глибині його найвіддаленішого листка.

Дерево, що **роздвоюється** (two-tree) – це двійкове дерево, яке або є порожнім, або таким, що для кожного його елемента, що не є листком, є дві гілки, що відходять. Наприклад, дерево, зображене на рис. 11.10, *а*, є таким, що роздвоюється, а дерево на рис. 11.10 *б* не є таким, що роздвоюється. Рекурсивне визначення таке: двійкове дерево **t** є деревом, що роздвоюється, якщо **t** порожнє або обоє піддерева дерева **t** є порожніми, або обоє піддерева дерева **t** є непустими деревами, що роздвоюються.

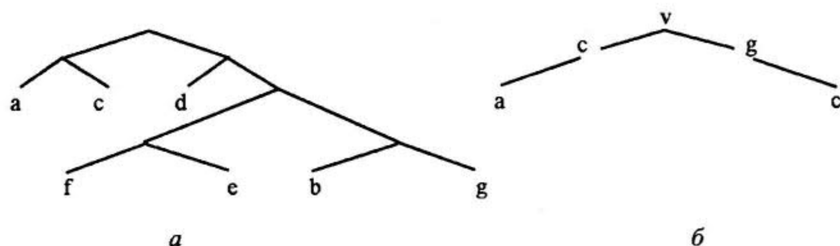


Рис. 11.10. Двійкове дерево, що роздвоюється (а); двійкове дерево, що не роздвоюється (б)

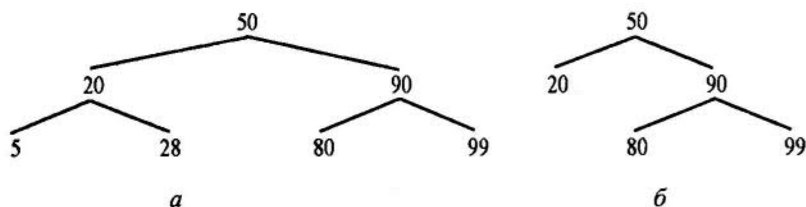


Рис. 11.11. Повне двійкове дерево (а); двійкове дерево, що не є повним (б)

Двійкове дерево t є **повним**, якщо **t** являє собою дерево, що роздвоюється, усі листки якого розташовані на тому самому рівні. Наприклад, дерево, зображене на рис. 11.11, *а*, є повним, а дерево, зображене на рис. 11.11, *б*, не є повним. Рекурсивне визначення таке: двійкове дерево **t** є повним, якщо **t** порожнє або ліве й праве піддерева дерева **t** мають однакову висоту, і обоє є повними.

Зрозуміло, будь-яке повне дерево є деревом, що роздвоюється, але зворотне не обов'язково правильно. Для повних двійкових дерев є певне співвідношення між висотою й кількістю елементів у дереві. Наприклад, якщо повне двійкове дерево має висоту 2, воно повинне мати точно 7 елементів.

Крім повноти, для дерева також використовується така характеристика, як закінченість. Двійкове дерево **t** є **закінченим**, якщо воно є повним до рівня height(t) - 1, а всі листки на найменшому рівні розташовуються якнайближче до лівого краю дерева. Під «найменшим рівнем» мають на увазі найвіддаленіший від кореня рівень.

Будь-яке повне дерево є закінченим, але не кожне закінчене двійкове дерево є повним. На рис. 11.12 показано кілька двійкових дерев. Наприклад, дерево, зображене на рис. 11.12, *а*, є закінченим двійковим деревом, але не є повним. Дерево на рис. 11.12, *б* не є закінченим, оскільки ***C*** має тільки один дочірній елемент. Дерево на рис. 11.12, *в* не є закінченим, оскільки листки ***I*** та ***J*** не є максимально наближеними до лівого краю дерева.

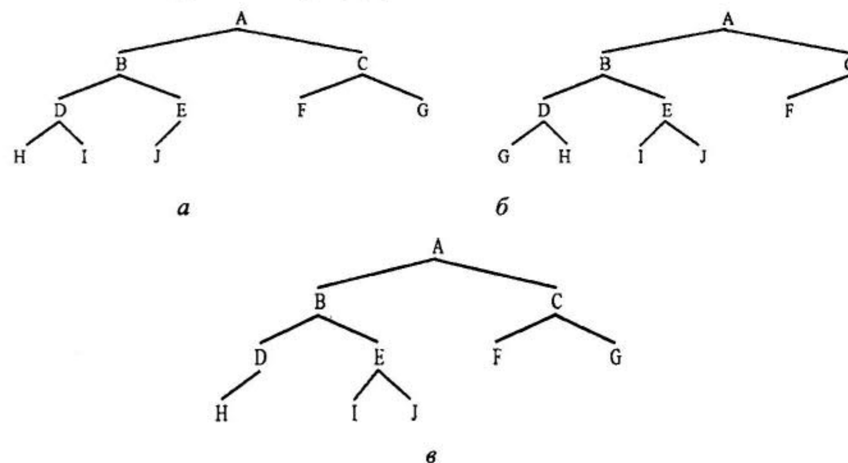


Рис 11.12. Неповне закінчене двійкове дерево (а); незакінчені двійкові дерева (б, в)

У закінченому двійковому дереві з кожним елементом можна асоціювати «індекс». Кореневому елементу призначається індекс 0. Для будь-якого позитивного цілого числа *i*, якщо елемент із індексом *i* має дочірні елементи, індекс його лівого дочірнього елемента дорівнює $2i + 1$, а індекс його правого дочірнього елемента – $2i + 2$. Наприклад, якщо двійкове дерево складається з 10 елементів, ці елементи мають такі індекси (рис. 11.13):

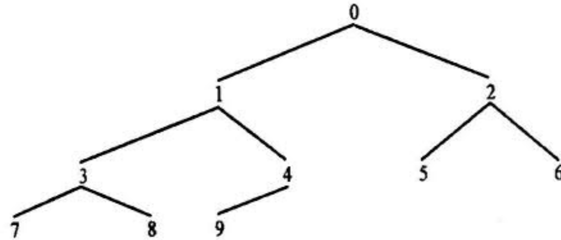


Рис 11.13. Двійкове дерево з десяти елементів

Батько елемента з індексом 8 має індекс 3, а батько елемента з індексом 5 має індекс 2. Взагалі, якщо *i* позитивне ціле число, батько елемента з індексом *i* має індекс $(i - 1)/2$.

Індекс елемента має важливе значення, тому що він дає змогу зберігати елементи закінченого двійкового дерева у вигляді масиву. Конкретно елемент, що має в дереві індекс *i*, зберігатиметься в комірці масиву з індексом *i*. Нижче наведено масив елементів для дерева, зображеного на рис. 11.12, *а*

A B C D E F G H I J

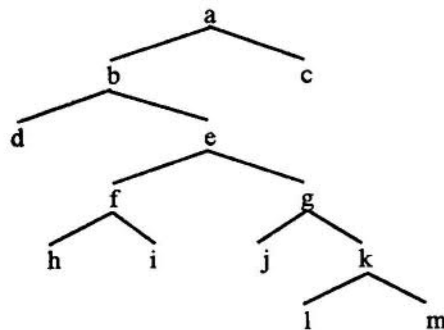


Рис. 11.14. Двійкове дерево з сумою глибини листків, що дорівнює 25

Фактично ми зберігатимемо ці елементи в масиві, а потім здійснюватимемо до них доступ, ніби вони перебувають у закінченому двійковому дереві. Отже, закінчене двійкове дерево являє собою абстракцію, яка може бути реалізована у вигляді масиву. У більшості випадків здійснюватиметься доступ до індексу дочірнього елемента через індекс його батьківського елемента або ж доступ до індексу батьківського елемента через індекс його дочірнього елемента. Ці індекси можна швидко обчислити, а відповідні елементи – швидко витягнути завдяки простоті операції довільного доступу до елементів масиву.

Рекурсивно обчислимо кількість елементів ***n(t)*** у дереві ***t***:

```
якщо (if) t порожньо
n(t) = 0
 інакше (else)
n(t) = 1 + n(leftTree(t)) + n(rightTreet(t))
```

Нехай t – непусте двійкове дерево. Тоді E(t), **довжина зовнішнього шляху** для дерева t, дорівнює сумі глибин усіх листків дерева. Наприклад, для дерева, зображеного на рис. 11.14, сума глибини листків дорівнює 2 + 4 + 4 + 4 + 5 + 5 + 1 = 25.

11.6.1. Алгоритми обходу двійкового дерева

Як вже розглядалось у підрозділі 6.2.2, обхід двійкового дерева **t** – це алгоритм, який дає змогу здійснити доступ до всіх елементів дерева **t** один і тільки один раз. Окремого класу BinaryTree не передбачено, оскільки підтримка методів вставок і видалень для двійкових дерев вже є в структурах даних стандартної бібліотеки шаблонів. Тобто наступні алгоритми не є методами.

Розглянемо чотири різні алгоритми обходу бінарних дерев.

Обхід 1. Обхід **inOrder**: ліворуч – до кореня – праворуч

Алгоритм:

```
inOrder(t)
 {
if (t не порожньо)
{
inOrder(leftTree(t));
доступ до кореневого елемента дерева t;
inOrder(rightTree(t));
} }
```

У процесі кожного рекурсивного виклику здійснюється доступ до одного елемента. Якщо **n** – кількість елементів у дереві, то worstTime(n) лінійно залежить від **n**. Можна скористатися цим рекурсивним описом для виводу списку елементів при обході способом inOrder наступного двійкового дерева **t** (рис. 11.15):

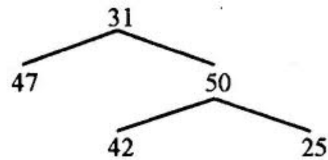


Рис. 11.15. Приклад двійкового дерева

Дерево **t** не є порожнім, тому починаємо з виконання обходу in0rder лівого піддерева leftTree(t), а саме

47

Це дерево складається з одного елемента. Потім здійснюємо доступ до кореневого елемента дерева **t**, а саме

31

Після цього виконуємо обхід inOrder правого піддерева rightTree(t) і починаємо з виконання обходу inOrder лівого піддерева leftTree(t), а саме

42

Потім здійснюємо доступ до кореневого елемента цієї версії дерева **t**, а саме

50

Нарешті, виконуємо обхід inOrder правого піддерева rightTree(t), а саме

25

Обхід дерева закінчений. Повний список елементів буде таким:

47 31 42 50 25

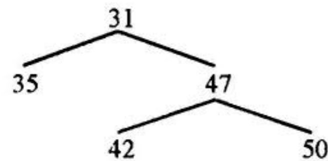


Рис. 11.16. Приклад двійкового дерева

Обхід inOrder названо так тому, що в особливого різновиду двійкового дерева – двійковому дереві пошуку – за такого способу обходу доступ до елементів здійснюватиметься за чергою їх проходження (наприклад, від меншого до більшого). Так, для двійкового дерева пошуку на рис. 11.16 доступ до елементів здійснюватиметься так:

25 31 42 47 50.

У двійковому дереві пошуку всі елементи в лівому піддереві менші або дорівнюють кореневому елементу, який, своєю чергою, менший або дорівнює всім елементам у правому піддереві.

Обхід 2. Обхід postOrder: ліворуч – праворуч – до кореня

Алгоритм:

postOrder(t)
{

if (t не порожньо)
{
postOrder(leftTree(t));
postOrder(rightTree(t);
доступ до кореневого елемента дерева t;
} }

worstTime(n) лінійно залежить від **n**, оскільки за кожного рекурсивного виклику здійснюється доступ до одного елемента.

Наприклад, виконуємо обхід способом postOrder наступного дерева (рис. 11.17):

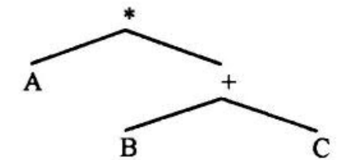


Рис. 11.17. Приклад двійкового дерева

Доступ до елементів здійснюватиметься за такою послідовністю:

A B C + *

Це двійкове дерево можна розглядати як «дерево-вираз»: кожний елемент, що не є листком, являє собою бінарний оператор, операндами якого є його ліве й праве піддерева. За такої інтерпретації обхід postOrder відповідає постфіксній нотації.

Обхід 3. Обхід preOrder: від кореня – ліворуч – праворуч

Алгоритм:

preOrder(t)
{
if (t не порожньо)
{
доступ до кореневого елемента дерева t;
preOrder(leftTree(t));
preOrder(rightTree(t));
} }

Як і для алгоритмів inOrder і postOrder, worstTime(n) лінійно залежить від **n**. У разі обходу способом preOrder дерева на рис. 11.17 доступ до елементів здійснюється так:

* A + B C.

Для дерева-виразу обхід preOrder відповідає префіксній нотації.

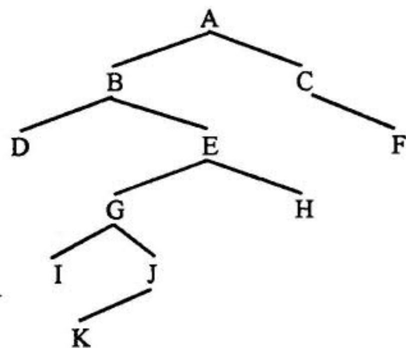


Рис. 11.18. Приклад двійкового дерева

Пошук у двійковому дереві, який реалізує обхід preOrder, називається пошуком у глибину, оскільки пошук заходить ліворуч так глибоко, наскільки можливо, перш ніж перейти до пошуку праворуч. Наприклад, виконуємо пошук у глибину в дереві з рис. 11.18.

Якщо метою є елемент H, то послідовність доступу до елементів буде такою:

A B D E G I J K H

Стратегія пошуку з поверненням заснована на пошуку в глибину, але при цьому може бути більше ніж два варіанти вибору. Наприклад, при пошуку виходу з лабіринту варіантами вибору є рух на північ, на схід, на південь або на захід. Оскільки рух на північ є першим розглянутим варіантом вибору, цей варіант буде послідовно застосовуватися, поки не буде досягнуто мети, або поки рух на північ не стане неможливим. Потім буде почато рух на схід, якщо таке можливо, а потім буде здійснено стільки переміщень на північ, скільки можливо або необхідно.

Обхід 4. Обхід breadthFirst: за рівнями (в ширину)

Щоб виконати обхід в ширину непустого двійкового дерева **t**, спочатку здійснюється доступ до кореневого елемента; потім – доступ до дочірніх елементів кореневого елемента зліва направо; потім – до дочірніх елементів дочірніх кореневого елемента зліва направо і т.д.

Наприклад, виконуємо обхід в ширину двійкового дерева на рис. 11.18. Порядок доступу до елементів таким:

A B C D F G H I J K.

Один зі способів виконання такого обходу полягає в тому, щоб генерувати, рівень за рівнем, список непустих піддерев. Потрібно витягати ці піддерева за тією самою послідовністю, у якій їх було згенеровано, щоб можна було здійснити доступ до елементів рівень за рівнем. Таку послідовність витягування елементів забезпечує контейнер черга.

Алгоритм:

```
breadthFirst(t)
{
```



```
// my_queue – це черга із двійкових дерев
// tree – двійкове дерево
 if (t не порожньо)
{
my_queue.push(t);
 while (my_queue не порожня)
{
tree = my_queue.front();
my_queue.pop();
// доступ до кореня дерева;
 if (leftTree(tree) не порожньо)
my_queue.push (leftTree(tree));
if (rightTree(tree) не порожньо)
my_queue.push (rightTree(tree));
} } }
```

На кожній ітерації циклу здійснюється доступ до одного елемента, тому worstTime(n) лінійно залежить від **n**.

Для обходу в ширину використовується черга, оскільки необхідно, щоб піддерева витягались у тій самій послідовності, якій їх було збережено (першим зайшов, першим вийшов). При обходах inOrder, postOrder і preOrder піддерева витягуються за послідовністю, зворотною до тієї, у якій їх було збережено (останнім зайшов, першим вийшов). Для кожного із цих трьох способів обходу застосовуємо рекурсію, яка еквівалентна ітеративному алгоритму, основаному на використанні стека.

11.6.2. Двійкові дерева пошуку

Почнемо з рекурсивного визначення дерева двійкового пошуку:

Двійкове дерево пошуку t – це таке двійкове дерево, що **t** або порожньо, або:

1. Кожний елемент у дереві leftTree(t) менший або дорівнює кореневому елементу дерева **t**.
2. Кожний елемент у дереві rightTree(t) більший або дорівнює кореневому елементу дерева **t**.
3. І дерево leftTree(t), і дерево rightTree(t) є двійковими деревами пошуку.

На рис. 11.19 зображено двійкове дерево пошуку.

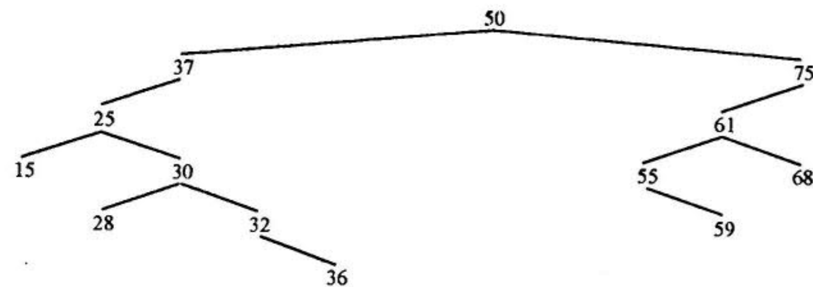


Рис. 11.19. Двійкове дерево пошуку

При обході двійкового дерева пошуку способом inOrder доступ до елементів відбувається за зростанням значень вершин. Наприклад, для двійкового дерева пошуку, зображеного на рис. 11.18, послідовність доступу при обході inOrder буде такою:

1) 25 28 30 32 36 37 50 55 59 61 68 75

Двійкове дерево пошуку являє приклад **асоціативного контейнера** (див. розділ 12). В асоціативному контейнері положення елемента визначається порівнянням елемента з іншими елементами в контейнері. Кожний елемент **має ключ:** складову елемента, яка використовується для порівнянь.

Контрольні запитання та вправи

1. Назвіть основні методи класу vector.

2. Припустимо, що під час розроблення класу **very_long_int** вирішено, що вектор **digits** міститиме ціле число зі *зворотним* порядком розрядів. Наприклад, якщо ввід містить «386X», то після зчитування цифри 3 її буде записано в позицію 0. Після зчитування цифри 8 її буде записано в позицію 0, «виштовхнувши» 3 у позицію 1. Нарешті, після зчитування цифри 6 її буде записано в позицію 0. Отже, у позиціях з 0 до 2 відповідно міститимуться цифри 6, 8, 3. Необхідно перевизначити відповідним чином перевантажені оператори >>, << , +.

3. Назвіть основні методи класу list.

4. Порівняйте списки з векторами і двосторонніми чергами відносно оцінок складності для операцій доступу, вставки й заміни.

5. Яка реалізація класу very_long_int матиме найбільшу швидкодію: у вигляді вектора, у вигляді черги із двостороннім доступом або у вигляді списку? Чому?

6. Припустимо, mylist являє собою об'єкт типу list, елементи якого мають тип double. Напишіть код для друку елементів у зворотній послідовності.

7. Припустимо, у порожній стек було поміщено елементи **а, б, в, г, д** (саме у такій послідовності). Потім дані зі стека вилучаються чотири рази й після кожного вилучення поміщаються в порожню чергу. Якщо після цього вилучити один елемент із черги, яким буде наступний доступний для видалення елемент у черзі?

8. Чи може клас **queue** слугувати базовим класом для класу **stack**? Поясніть, чому.

9. Переведіть наступні вирази в постфіксну нотацію:

a. x + y*z

b. (x + y)*z

c. x – в – z* (a + Ь)

d. (a + b)*c-(d + e*f/((g/h + i – j)*k))/r

10. Переведіть кожний з виразів у вправі 9 у префіксну нотацію.

11. Вираз у постфіксній нотації можна обчислити під час виконання програми з використанням стека. Припустимо, що постфіксний вираз містить тільки цілочислове значення й бінарні оператори. Наприклад, постфіксний вираз може бути таким:

8 5 4 + * 7 –

Обчислюють його так. Коли зустрічається значення, його розміщають у стек. Коли зустрічається оператор, зі стека вилучається перший і другий його елементи, оператор застосовується (другий елемент є лівим операндом, а перший елемент – правим операндом), а результат розміщається в стек. Після обробки постфіксного виразу значення цього виразу знаходиться у верхньому (і єдиному) елементі стека.

Наприклад, для цього виразу вміст стека буде таким:

```
        4
      5 5
    8 8 8
9       7
8   72 72 65
```

Перетворіть наступний вираз у постфіксну нотацію й використайте стек для обчислення виразу:

5 + 2 × (30 – 10/5)

12. Ні в класі **queue**, ні в класі **stack** не визначений деструктор. Чи може такий «недолік» привести до втрат пам'яті? Поясніть, чому.

13. a. Побудуйте двійкове дерево висотою 3 з восьма елементами.

b. Чи можна побудувати двійкове дерево висотою 2 з вісьма елементами?

c. Для **n** у діапазоні від 1 до 20 визначить мінімальну можливу висоту двійкового дерева з **n** елементами.

d. На основі обчислень, виконаних у пункті *c*, складіть формулу для визначення мінімально можливої висоти двійкового дерева з **n** елементами (**n** може бути будь-яким додатним цілим числом).

14. a. Яка максимально можлива кількість листків у двійковому дереві з 10 елементами? Побудуйте таке дерево.

b. Яка мінімально можлива кількість листків у двійковому дереві з 10 елементами? Побудуйте таке дерево.

15. a. Побудуйте дерево, що роздвоюється та не є закінченим.

b. Побудуйте закінчене дерево, що не роздвоюється.

c. Побудуйте закінчене дерево, що не є повним.

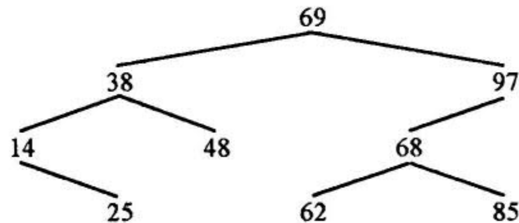
d. Скільки листків у дереві, що роздвоюється та містить 17 елементів?

e. Скільки листків у дереві, що роздвоюється та містить 731 елемент?

f. Скільки буде елементів у повному двійковому дереві висотою 4?

Н Скільки буде елементів у повному двійковому дереві висотою 12?

16. Для наведеного нижче двійкового дерева вкажіть порядок, у якому здійснюватиметься доступ у елементів при обходах inOrder, postOrder, preOrder і breadthFind.



17. Покажіть, що двійкове дерево з **n** елементами має **2n + 1** піддерев (із самим деревом). Скільки серед таких піддерев буде порожніх?

18. Поясніть, як вилучити кожний з наведених нижче елементів із двійкового дерева пошуку:

a. Елемент, що не має дочірніх елементів.

b. Елемент із одним дочірнім елементом.

c. Елемент із двома дочірніми елементами.

Покажіть, що в будь-якому закінченому двійковому дереві **t** щонайменше половина елементів є листками.

Приклади тестових питань

2) Є визначення:

```
vector<char> L;
vector<char>::iterator itr;
```

Яке слово буде виведено в результаті виконання наступної послідовності інструкцій:

```
L.push_back      ('f ');
L.push_back      ('i');
L.push_back      ('e');
L.push_back      ('r');
L.push_back      ('c');
L.push_back      ('e');
itr = L.begin();
cout <<*itr;
itr++;
cout <<*itr;
cout<< L [3];
itr+ = 4;
cout <<*itr;
```

а) fierce

б) fier

в) ecre

3) Є визначення:

```
list<char> letters;
list<char>: : iterator itr;
```

У якій послідовності розмістяться букви в списку після виконання таких інструкцій:

```
itr = letters.begin();
letters.insert  (itr,   'f');
letters.insert  (itr, 'e');
itr++;
letters.insert  (itr, 'r');
itr++;
itr++;
letters.insert  (itr,   't');
letters.insert  (itr, 'e');
letters.erase  (letters.begin  ());
itr--;
itr--;
letters.insert  (itr, 'p');
itr++;
letters.insert (itr, 'e');
itr = letters.end ();
itr--;
letters.insert  (itr, 'c');
```

а) ferret
б) tepper
в) perfect

4) Є визначення:

```
queue <int> x;
```

Як виглядатиме черга після виконання таких інструкцій:

```
a.      x.push  (200) ;
b.      x.push  (121);
c.      x.push  (103) ;
d.      x.push  (211);
e.      x . pop () ;
f.      x.push (197) ;
g.      x. Pop () ;
```

а) 200 121 103 211 197
б) 103 211 197
в) 197 103 200

5) Є визначення:

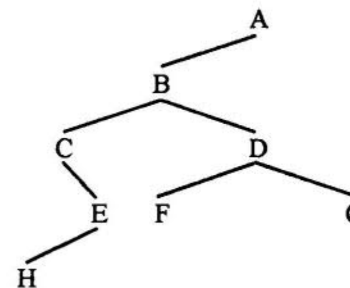
```
stack <int> x;
```

Як виглядатиме стек після виконання таких інструкцій:


```
a.      x.push  (200) ;
b.      x.push  (121);
c.      x.push  (103) ;
d.      x.push  (211);
e.      x . pop () ;
f.      x.push (197) ;
g.      x. Pop () ;
```

а) 200 121 103 211 197
б) 103 211 197
в) 103 121 200

5. Дайте відповідь на питання стосовно зображеного нижче двійкового дерева:



6) який елемент є кореневим? А) С; б) F; в) А.
7) скільки елементів містить дерево? А) 8; б) 7; в) 3.
8) скільки листків є в дереві? А) 4; б) 3; в) 8.
9) яка висота дерева? А) 3; б) 7; в) 4.
10) який рівень елемента *F?* А) 3; б) 0; в) 4.
6) який елемент є батьком для елемента *D?* А) F; б) A; в) B.
7) які елементи є нащадками елемента *B?* а) C, D; б) C, D, E, F, G, H; в) H, F, G.

12. АСОЦІАТИВНІ КОНТЕЙНЕРИ У СКЛАДІ СТАНДАРТНОЇ БІБЛІОТЕКИ ШАБЛОНІВ

Асоціативний контейнер – це контейнер, у якому положення елемента визначається порівнянням ключа цього елемента із ключами інших елементів у контейнері.

Чотири класи асоціативних контейнерів у складі стандартної бібліотеки шаблонів відрізняються варіантом відповідей на такі два питання: 1. Чи складається елемент цілком із ключа? 2. Чи допускає контейнер наявність декількох елементів з однаковим ключем?

Відповідно розрізняють контейнери **Set** (множина), **Multiset** (мультимножина), **Map** (карта), **Multimap** (мультикарта).

12.1. Клас Set

Клас Set – це асоціативний контейнер, в якому кожний елемент складається цілком із ключа, а повторювання елементів не допускаються. Для операцій вставки, видалення й пошуку worstTime(n) логарифмічно залежить від **n**.

У Standart C++ оголошення класу **Set** починається так:

```
template< class Key, class Compare = less <Key>,
        class Allocator = allocator <Key> >
class Set
{...}
```

У класі **Set** ключем є весь елемент цілком, і ключі є унікальними. Наприклад, нижче наведено визначення об'єкта типу **Set** зі стрічковими елементами:

```
set <string> names;
```

Оскільки множина являє собою асоціативний контейнер, місце зберігання елемента залежить від результату його порівняння з

іншими елементами. Під час порівняння використовується шаблонний аргумент, що відповідає шаблонному параметру Compare. Цим шаблонним аргументом за замовчуванням є клас-функція **less**, визначений у файлі <function> стандартної бібліотеки шаблонів так:

```
template< class T >
struct less:binary_function <T, T, bool> {
 bool operator() (const T& x, const T& y) const
  {return x<y;}
} ;
```

Отже, якщо об'єкт-функція **comp** є екземпляром класу-функції **less**, то **comp (x, y)** повертає значення виразу **x < y**.

Клас **Set** має звичайний асортимент контейнерних методів: конструктори, деструктор, **begin, end, size, empty, find, insert** і **erase**. Більшість визначень методів містить виклик об'єктом **t** відповідного методу. Наприклад, визначення методів **find** та **erase**:

```
     iterator find (const key_type& x) const
    {return t.find(x);}
void erase(iterator position) {t.erase ((rep_type::
iterators) position);}
```

Метод **insert** не зовсім звичайний, оскільки елемент не буде вставлено, якщо він збігається з елементом, уже наявним у контейнері типу **Set**. Із цієї причини метод **insert** повинен повертати пари: ітератор і булеве (bool) значення. Якщо елемент **x** уже є в множині, то повертається пара, що включає ітератор, установлений на раніше вставлений елемент, і булеве значення **false**. А якщо ні, то вертається пара, що включає ітератор, установлений на новий вставлений елемент, і булеве значення **true**.

Визначення методу:

```
    pair <iterator, bool> insert (const value_type&
x)
     { pair <iterator, bool> p = t. insert(x);
    return   pair   <iterator,   bool>   (p.first,
p.second); }
```

12.2. Клас Multiset

У контейнері типу **Multiset** кожний елемент складається лише із ключа, й допускаються повторювання елементів. Відповідно метод **insert** повертає ітератор, установлений на новий вставлений елемент. Оскільки може бути кілька копій елемента, у класі **Multiset** передбачено метод **lower_bound**, що повертає ітератор, установлений на першу позицію, у яку може бути вставлений елемент без порушення послідовності проходження елементів у мультимножині:

```
iterator lower_bound (const T& x) const;
```

Аналогічно метод **upper_bound** повертає ітератор, установлений на останній позиції, у яку може бути вставлений елемент без порушення послідовність проходження елементів у мультимножині. Метод **equal_range** повертає пари ітераторів, установлених на першій і останній таких позиціях для цього елемента.

12.3. Клас Map

У Standard C++ оголошення класу **Map** починається так:

```
template< class Key, class T, class Compare=less<Key>,
 <Allocator = allocator<T> >
class Map  { … }
```

У контейнері типу **Map** кожне значення є парою pair<Key, T>, а повторення ключів не припускається. Наприклад, якщо при визначенні контейнера типу **Map** з іменем **students**, в якому ключем є ім'я студента (елемент типу **string**), **T** представлятиме тип для середнього бала поточної успішності студента, а імена зберігатимуться за абеткою, слід записати:

```
Map<string, float, less<string>> students;
```

Оскільки цей студент може мати тільки один середній бал успішності в певний час, контейнер **students** типу **Map** визначає унікальну асоціацію між студентом і середнім балом успішності цього студента. Фактично, **students** являє собою карту відповідностей між студентами та їхніми балами успішності. У складі контейнера **students** типу **Map** не буде пар з однаковими ключами, тобто значень із тим самим іменем студента.

Як і для інших асоціативних класів, у набір методів для класу **Map** входять конструктори **size**, **empty**, **insert**, **erase**, **find**, **begin**, **end**. У методі **insert** використано як аргумент пару значень, для

ініціалізації якої використовується конструктор класу **pair**. Так, можна записати:

```
pair < string, float > student ("Петренко, Іван",
7.96);
students.insert (student);
```

Клас **Map** має одну особливість: асоціативний масив. У звичайному масиві індекс є цілим числом. В асоціативному масиві індекс являє собою ключ будь-якого типу: **string**, **double**, **int** або навіть клас, визначений користувачем. Операцією присвоєння

```
a[x] = m;
```

вводять до карти **a** пару **<x, m >**. Якщо у карті вже є пара із ключем **x**, то другий компонент у цій парі буде замінено на **m**. Наприклад, наведену вище вставку в контейнер **students** типу **Map** можна замінити на

```
students ["Іванейко, Петро "] = 9.6;
```

12.4. Клас Multimap

Контейнер типу **Multimap** допускає наявність декількох елементів з ідентичними значеннями ключів, а кожний елемент являє собою пару **<Key, T>**. Асоціативні масиви використовувати не дозволяється, щоб уникнути можливих непорозумінь. Так, якщо контейнер **myMulti** типу **Multimap** містить пари <"Іван", 3> і <"Іван", 5>, то вираз

```
myMulti ["Іван"]
```

можна зарахувати до другого компонента як першої, так і другої пари.

Приклад визначення об'єкта типу **Multimap**:

```
Multimap <int, string, greater <int>> most_wins;
```

Контейнер **most_wins** типу **Multimap** міститиме показники найбільші кількості перемог за один сезон у вищій лізі КВК за останні 20 років й імена капітанів команд. Тут потрібний саме контейнер типу **Multimap**, оскільки можуть бути записи з однаковою кількістю перемог. Елементи в контейнері будуть розміщені в порядку спадання числа перемог. Приклад операції вставки:

```
most_wins.insert (pair <int, string> (31, "Пилипенко
Максим"));
```

Метод **insert** класу **Multimap** містить ітератор з рекомендацією:

```
iterator insert (iterator position, const value_type& x);
```

Як і для інших версій методу **insert**, здійснюватиметься пошук у дереві з метою визначення місцярозташування елемента **x**. Ітератор **position** пропонує, з якого місця почати пошук. Нехай, наприклад, ми хочемо вставити в контейнер **multi n** елементів з масиву **data**. Припустимо, що ці елемени в **data** уже мають правильний порядок – за зростанням. Це можна зробити так:

```
itr = multi.begin() ;
for (int i = 0; i < n; i++)
itr = multi.insert (itr, data[i]);
```

На кожній ітерації циклу **for** викликається метод **insert**, для якого **itr** вказує на початкову точку пошуку. Оскільки елементи вставляються за порядком, **itr** завжди вказуватиме на найбільший елемент. Час вставки нового елемента буде постійним. Загальний час вставки **n** елементів дорівнюватиме **O(n)**, а не **O(n log n)**, що мало б місце, якби ітератор з рекомендацією не використовувався.

Контрольні запитання та вправи

1. Які контейнери належать до асоціативних?
2. Назвіть основні методи класу **Set**.
3. Поясніть як працює метод **insert** класу **Set**.
4. Чим відрізняється контейнер **Multiset** від контейнера **Set**?
5. Чи допускається повторення ключів у контейнері **Map**?
6. Якого типу індекс в асоціативному масиві?
7. Чим відрізняється контейнер **Multimap** від контейнера **Map**?
8. Який ітератор передбачає метод **insert** класу **Multimap**?
9. Є визначення

```
map < string, int > my_map;
```

Поясніть різницю між двома записами:

```
my_map.insert ("Петренко", 200);
my_map ["Петренко"] = 200;
```

10. Для кожного з поданих типів складіть відповідний об'єкт цього типу й поясніть, чому цей тип підходить для цього об'єкта.

```
set < double, greater < double >>
multiset < string >
map < double, string >
multimap <int, double, greater < int >>
```

Приклади тестових питань

1. Відомі імена й номери відділів для кожного співробітника компанії. Імена не повторюються. Необхідно зберігати цю інформацію за іменами за абеткою.

Для цього необхідно використовувати контейнер:
а) Set;
б) Multiset;
в) Map;
г) Multimap.

2 Яким є час вставки **n** елементів в контейнер класу **Multimap** :
а) O(n);
б) O(n log n);
в) O(n!).

3. Елементи контейнера класу **Set** складаються з:
а) ключа;
б) індекса;
в) вказівника.

4. Елементи контейнера класу **Set** можуть повторюватись:
а) так;
б) ні.

5. Елементи контейнера класу **Multiset** можуть повторюватись:
а) так;
б) ні.

6. У класі **Multiset** ітератор, установлений на першу позицію, у яку може бути вставлений елемент без порушення порядка проходження елементів у мультимножині, повертається методом:
а) **lower_bound**;
б) **upper_bound**;
в) **equal_range**.

7. У контейнері типу **Map** допускається повторення ключів:
а) так;
б) ні.

8. У контейнері типу **Multimap** допускається повторення ключів:
а) так;
б) ні.

13. ПОХІДНІ КОНТЕЙНЕРИ БІБЛІОТЕКИ ШАБЛОНІВ

13.1. Клас priority_queue

Доволі поширена структура даних, що являє собою різновид черги: черга із пріоритетом. Основна ідея полягає в тому, що елементи, які очікують на обслуговування у черзі, вибирають не строго за принципом «перший зайшов, перший вийшов». Приклад ресурсу, для якого доречно використовувати чергу із пріоритетом, – мережевий принтер. Зазвичай завдання роздруковуються за чергою їх надходження, але під час друку одного завдання в чергу може надійти декілька інших завдань. Найвищий пріоритет може бути привласнений завданню з найменшою кількістю сторінок. Це дасть змогу оптимізувати середній час виконання завдань. Таку саму ідею обслуговування із пріоритетом можна застосувати для будь-яких розділювальних ресурсів: центрального процесора, навчальних курсів на наступний семестр тощо.

Черга із пріоритетом – це контейнер, у якому операції доступу або видалення здійснюються стосовно елемента, що має найвищий пріоритет. Пріоритети елементам призначаються за певним правилом.

Що відбудеться, якщо найвищий пріоритет мають два або більше елементів? Перевагу в цьому випадку слід віддавати елементу, який перебуває в черзі якнайдовше. Черга із пріоритетом є справедливою, якщо для будь-яких двох елементів з однаковим пріоритетом першим видаляється із черги той елемент, який перебував у черзі довше.

Відповідно до Standard C++, визначення класу **priority_queue** розміщається у файлі <queue>. Водночас у реалізації Hewlett-Packard це визначення розміщається у файлі <stack>.

Поля класу priority_queue

Два поля в складі класу **priority_queue** являють собою об'єкт-контейнер **c** і об'єкт-функцію **comp**:

```
protected:
Container c;
Compare comp;
```

Екземпляр контейнера **c** буде, за замовчуванням, реалізацією класу **vector**. Екземпляр об'єкта-функції **comp** буде реалізацією деякого вбудованого або визначеного користувачем класу-функції (за замовчуванням використовується клас less).

Оголошення класу priority_queue починається так:

```
template <class T,  class Container = vector<T>,
class Compare = less<Container::value_type> >
class priority_queue
{  ...}
```

Клас **priority_queue** є адаптером контейнера. Перший шаблонний параметр, **T**, належить до типу елементів, що зберігаються в черзі із пріоритетом. Шаблонний параметр Container належить до базового класу-контейнеру, чиї інтерфейси методів адаптує клас **priority_queue**. До класу, що відповідає шаблонному аргументу, ставлять вимоги, щоб він містив методи **front, push_back, pop_back** і підтримував ітератори із довільним доступом. Цим вимогам задовольняє як клас **vector**, так і клас **deque**. За замовчуванням використовується клас **vector**. Шаблонний аргумент, що відповідає параметру Compare, повинен являти собою клас-функцію, для якого оператор operator() порівнює значення value_type, тип яких майже завжди збігається з типом T. За замовчуванням використовується вбудований клас-функція less, тобто елемент із найбільшим значенням перебуватиме на початку об'єкта типу priority_queue. За замовчуванням найвищий пріоритет має елемент із найбільшим значенням.

Визначення методів

У класі **priority_queue** є сім методів:

1. Ця черга із пріоритетом ініціалізується **x** та копією об'єкта типу Container.

```
explicit priority_queue (const Compare& x = Compare(),
const Container& = Container ());
```

Приклад. Визначимо дві черги із пріоритетом так:

```
pq<employee> e_pq1 ();
pq<employee> e_pq2  (e_pq1); // e_pq2 містить копію e_pq1
```

2. Ця черга із пріоритетом ініціалізується копіями елементів у позиціях від first (включаючи) до last (не включаючи), об'єктом **x** типу Compare і базовим контейнером (Container) **y**.

```
template < class InputIterator >
priority_queue (InputIterator first, InputIterator last,
  const Compare& x = Compare(),
  const Container& y = Container ());
```

3. Повертається кількість елементів у черзі із пріоритетом.

```
size type size() const;
```

4. Якщо ця черга із пріоритетом не містить елементів, повертається true, інакше повертається false.

```
bool empty() const;
```

5. **x** вставляється в чергу з пріоритетом.

```
void push  (const value_type& x);
```

6. Ця черга із пріоритетом не порожня. Елемент у цій черзі із пріоритетом, який перед відправленням цього повідомлення мав найвищий пріоритет, видаляється.

```
void pop();
```

7. Ця черга із пріоритетом не порожня. Повертається константне посилання на елемент з найвищим пріоритетом у черзі.

```
const value_type& top() const;
```

Оскільки **const** є частиною специфікації типу, що повертається, то верхній елемент в об'єкті типу priority_queue не можна модифікувати. Така модифікація може змінювати пріоритет верхнього елемента в черзі.

Метод **top()** повертає посилання, яке повертається після виклику **c.front()**. Метод **pop()** викликає метод **c.pop_back()** для видалення елемента із найвищим пріоритетом. Визначення методу **pop()**:

```
void pop ()
 {
pop_heap  (c.begin (),  c.end(), comp);
c.pop_back();
}
```

Визначення методу **push()**:

```
void push  (const value_type& x)
{
 c.push_back(x);
 push_heap  (c.begin(),  c.end(),  comp) ;
}
```


Узагальнені алгоритми **push_heap** та **pop_heap** використовують структуру типу «куча» (heap).

13.2. Кучі

Куча t – це закінчене двійкове дерево, таке що **t** або порожньо, або:

1. Кореневим елементом є найбільший елемент в **t**, що визначається відповідно до деякого методу порівняння елементів.

2. Ліве піддерево leftTree(t) і праве піддерево rightTree(t) є кучами.

На рис. 13.1 показано кучу з 10 елементами типу int.

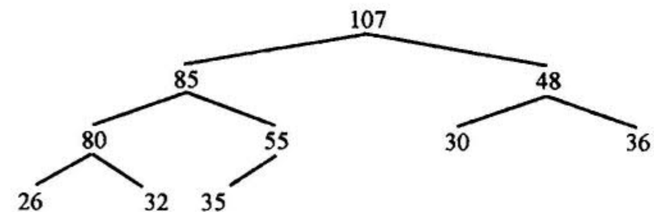


Рис. 13.1. Куча з 10 елементами типу int

Елементи в кучі впорядковано згори донизу, а не зліва направо. Кожний кореневий елемент більші за кожен зі своїх дочірніх елементів або дорівнює їм, але деякі ліві братні елементи можуть бути більшими, ніж їхні праві «брати», а деякі можуть бути меншими. Наприклад, на рис. 13.1 елемент 85 більший за його правий братній елемент 48, але елемент 30 менший, ніж його правий братній елемент 36. У кучі елементи розташовуються згори донизу, а не зліва направо.

Куча може виявитися доволі корисною в багатьох застосуваннях. Одне з них –реалізація класу **priority_queue**. Дві головні операції з кучею – це **push_heap** для включення елемента у кучу і **pop_heap** для видалення елемента з кучі. Розглянемо ще одну операцію, **make_heap**, яка створює кучу «з нуля» та активізується під час виклику конструктора **priority_queue**.

Куча є закінченим двійковим деревом. З кожним елементом у закінченому двійковому дереві **t** можна асоціювати позицію в діапазоні 0 ... n(t) - 1. Якщо розглядати ці позиції як індекси, то можна помітити, що елементи кучі можуть зберігатися у вигляді масиву. Наприклад, куча, яку зображено на рис. 10.1, у вигляді масиву має вигляд:

107 85 48 80 55 30 36 26 32 35

Довільний доступ до елементів масиву достатньо зручний для обробки куч. За індексом елемента можна швидко доступитись до його дочірніх елементів. Наприклад, дочірні елементи елемента з індексом i мають індекси $2i+1$ та $2i+2$. Батько елемента з індексом j має індекс $(j-1)/2$. Можливість швидко переміщувати батька на місце його дочірнього елемента і навпаки робить кучу ефективною структурою для реалізації черги з пріоритетом.

Будь-який контейнер – такий як масив, вектор або двостороння черга – який підтримує ітератори з довільним доступом, можна використати як кучу.

Елемент, що вставляється, на який встановлений ітератор last - 1, зберігається у тимчасовій змінній value, а звільнена позиція, в якій міститься елемент, що підлягає вставці, називається діркою (hole). Технічно використовується змінна holeIndex, яка зберігає індекс у контейнері для поточної дірки. Наприклад, на рис. 13.2 показано кучу, зображену на рис. 13.1, при виклику методу **push_heap** для вставки елемента 90.

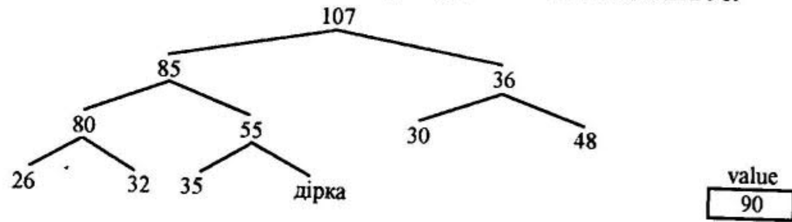


Рис. 13.2. Куча з рис. 13.1 перед початком виконання методу push_heap

Щоб визначити, чи залишиться структура кучею у випадку розміщення елемента 90 у дірку, порівнюється 90 з елементом, що зберігається в батькові дірки, 55, індекс якого дорівнює (hole_index - 1)/2. Оскільки 90 > 55, переміщаємо 55 у дірку й пересуваємо вказівник holeIndex на позицію, у якій перебував елемент 55 (рис. 13.3).

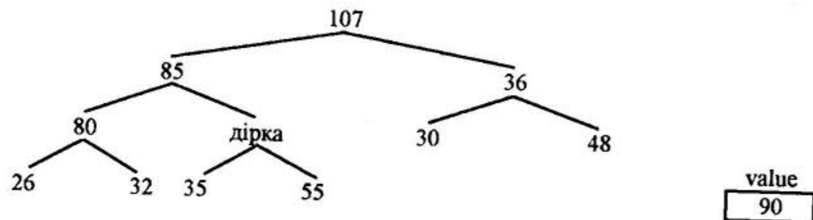


Рис. 13.3. Проміжний вигляд кучі

Продовжуємо просування, поки не досягнемо вершини кучі або поки не знайдемо позицію, для якої 90 <= елемента, що є батьком для дірки. Тоді 90 вставляється в позицію, індекс якої відповідає індексу дірки (holeIndex). Остаточний стан кучі показано на рис. 13.4.

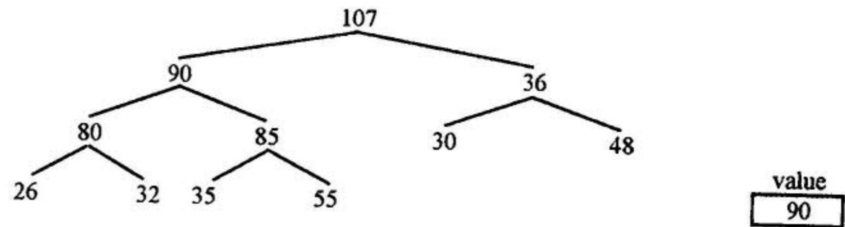


Рис. 13.4. Куча з новим доданим елементом 90

Нижче наведено частину коду з визначення методу **push_heap**.

```
    int parent = (holeIndex - 1)/2;
    while (holeIndex > topIndex && comp(*(first +
parent), value))
     {
    *(first + holeIndex)  = * (first + parent);
    holeIndex = parent;
    parent =  (holeIndex - 1)/2;
     }
    * (first + holeIndex)  = value;
```

Найгіршим є випадок, коли елемент, що вставляється, більший за елемент, на який указує first. Тоді кількість ітерацій циклу while буде пропорційна висоті дерева. Висота закінченого двійкового дерева логарифмічно залежить від **n**. Тобто, worstTime(n) логарифмічно залежить від **n**.

У середньому випадку близько половини елементів у кучі матимуть значення менші, ніж значення елемента, що вставляється, а близько половини матимуть більші значення. Але кучі являють собою доволі «густі» дерева: мінімум половина всіх елементів є листками. А відповідно до властивостей кучі, більша частина елементів зі значеннями меншими, ніж те, що вставляється, розташовуються поблизу рівня листків. Фактично середня кількість ітерацій циклу менша ніж три.

Проаналізуємо метод **push**. Найгіршим є випадок, коли куча є повною. Тоді все залежить від базового контейнерного класу – наприклад, **vector** або **deque**. У кожному разі буде здійснюватися копіювання з лінійною залежністю від кількості елементів **n**, тому worstTime(n) лінійно залежить від **n**. Однак таке копіювання відбувається не часто – одного разу для **n** операцій вставки. Ітератор last встановлено безпосередньо за останнім елементом кучі. Алгоритм **pop_heap** є протилежністю алгоритму **push_heap**: він працює, починаючи з вершини кучі й донизу. Елемент у вершині є елементом, що видаляється; він буде збережений у позиції, індекс якої вказує ітератор last - 1. Тому тимчасово переміщуємо елемент у позиції last - 1 у змінну value. Дірка спочатку перебуває в позиції на вершині кучі. На рис. 13.5 показано початкові параметри для методу **pop_heap**, що застосовується до кучі, зображеної на рис. 13.4.

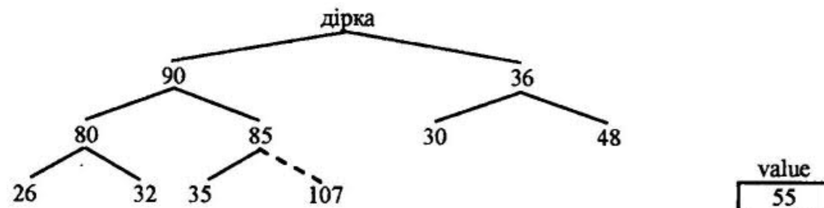


Рис. 13.5. Куча з рис. 10.4 перед викликом методу pop_heap

Елемент 107 було збережено за нижньою границею кучі, але треба знайти, куди помістити елемент 55, який зараз зберігається в змінній value. Замість того, щоб порівнювати 55 з іншими елементами, спочатку пересуваємо дірку донизу, порівнюючи дочірні елементи дірки один з одним. Більший дочірній елемент переміщується на місце дірки, а дірка переміщується в позицію, у якій зберігався більший дочірній елемент. На рис. 13.6 показано кучу після того, як дірку було поміщено в позицію, яку раніше займав елемент 90. Після ще двох ітерацій циклу дірка виявляється на рівні листка, як показано на рис. 13.7. Зверніть увагу, що елемент 107 не бере участі у порівняннях, оскільки його більше не вважають частиною кучі. Тепер можна викликати метод push_heap для вставки елемента 55 на місце дірки. На рис. 13.8 показано остаточний вид дерева, але при цьому елемент 107 не є частиною кучі.

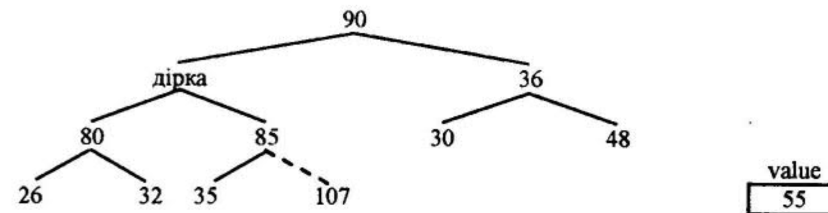


Рис. 13.6. Куча після переміщення дірки в позицію елемента 90

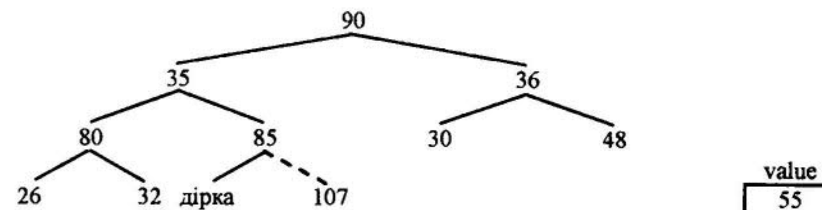


Рис. 13.7. Куча після переміщення дірки в позицію листка 35

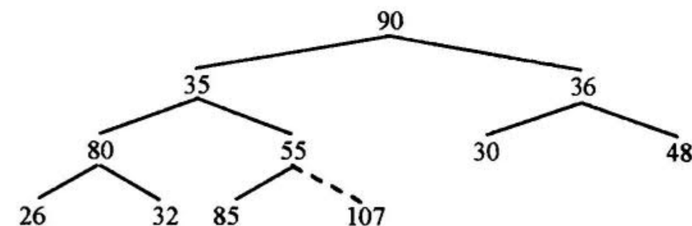


Рис. 13.8. Остаточний вигляд кучі

Основну роботу в методі **pop_heap** виконує допоміжний метод **adjust_heap**.

```
template <class RandomAccessIterator, class int,
class T,  class Compare>
void       adjust heap   (RandomAccessIterator
first, int holeIndex,  int len,
 T value, Compare comp)
{
 int topIndex = holeIndex;
 int secondChild = 2*holeIndex + 2;
while  (secondChild < len)
 {
```

```
    if  (comp(*(first + secondChild),
    *(first +  (secondChild - 1))))
    secondChild--;
    *(first + holeIndex) = * (first + secondChild);
     holeIndex = secondChild;
     secondChild = 2*(secondChild + 1);
   }
   if  (secondChild == len)
    {
    *(first + holeIndex)  = *(first +  (secondChild
- 1));
     holeIndex = secondChild - 1;
   }
    push_heap  (first, holeIndex,  topIndex, value,
comp);
   }
```

Час виконання циклу для переміщення дірки донизу на рівень листків пропорційний висоті дерева. Куча є закінченим двійковим деревом, тому висота її завжди логарифмічно залежить від **n**. Додатковий виклик методу push_heap забирає час від постійного до логарифмічно залежного від **n**. Отже, для методу **pop_heap** worstTime(n) логарифмічно залежить від **n**.

Потрібно мати можливість легко переміщуватися між батьківськими й дочірніми елементами. Цієї легкості досягають за допомогою ітераторів довільного доступу (еквівалентів індексів у масиві).

Нарешті розглянемо алгоритм **make_heap**, який створює кучу з контейнера, що підтримує ітератори з довільним доступом.

Перший елемент (first) сам по собі є кучею, тому можна здійснити обхід контейнера, що залишився, викликаючи на кожній ітерації метод **push_heap**:

```
itr = first++;
while  (itr  != last)
push_heap (first, itr++, comp);
```

Але в кучі половина всіх елементів є листками, а листок автоматично є кучею. Тобто, половина викликів **push_heap** виявляються зайвими.

Почнемо побудову кучі з елемента з найбільшим індексом, що не є листком. Наприклад, для дерева, зображеного на рис. 13.9, починаємо з індексу елемента 30. Коригування схожі на ті, які використовували для методу **pop_heap**, за винятком того, що елемент у позиції з індексом дірки, а саме, елемент 30, буде тимчасово збережений у змінній value. Після переміщення на позицію дірки елемента 35 дірка зміщується донизу на позицію, яку займав елемент 35, викликається метод push_heap для вставки елемента 30 у позицію дірки. На рис. 13.10 показано, як виглядає дерево після успішної вставки елемента 30. Переміщуючись до менших індексів у контейнері, коригуємо піддерево з коренем в елементі 40, а потім коригуємо піддерево з коренем в елементі 70. Після цього одержимо дерево, показане на рис. 13.11.

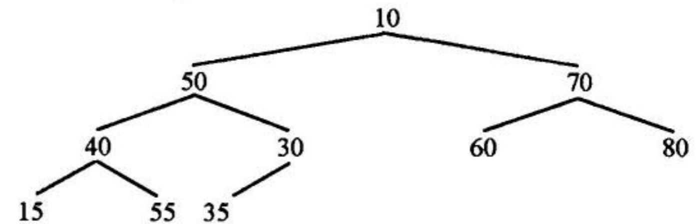


Рис. 13.9. Закінчене двійкове дерево

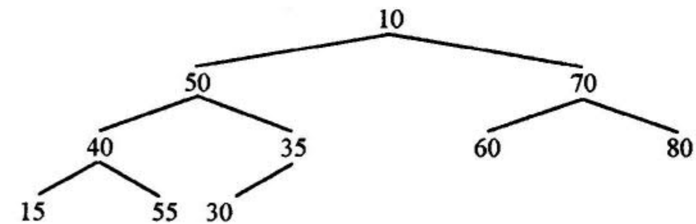


Рис. 13.10. Проміжний вигляд кучі для елемента 30

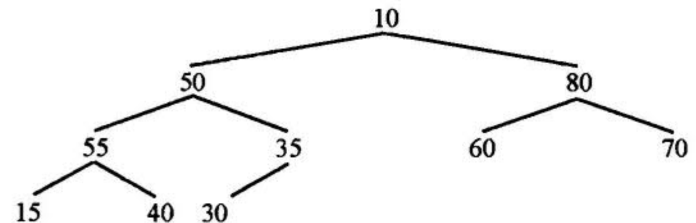


Рис. 13.11. Проміжний вигляд кучі для елементів 40 та 70

Щоб перетворити на кучу піддерево з коренем в елементі 50, дірці призначається індекс позиції, що займає елемент 50, а 50 зберігається в змінній **value**. Після переміщення в дірку елемента 55 дірка переміщується в позицію, яку займав елемент 55. На наступній ітерації в цю позицію міститься елемент 40. Після застосування методу **push_heap** для вставки елемента 50 у шлях, що починається в цій новій дірці, відбувається остаточне приведення дерева до кучі, починаючи з кореня всього дерева. Після коригування одержуємо кучу, зображену на рис. 13.12.

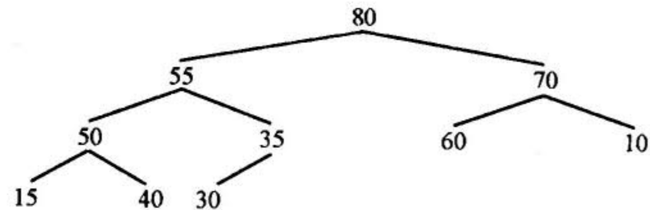


Рис. 13.12. Остаточний вигляд кучі

Наведемо найважливішу частину коду (метод **adjust_heap** викликається з методу **pop_heap**; другим аргументом методу **adjust_heap** є початковий індекс дірки):

```
if  (last_first < 2) return;
int len = last_first;
 int parent =  (len - 2)/2;
while (true)
{
 adjust_heap ( first, parent, len, T (*(first + parent)), comp);
  if  (parent == 0) return;
parent--;
}
```

Аналіз методу **make_heap** доволі складний, але результат полягає в тому, що worstTime(n) завжди лінійно залежить від **n** незалежно від того, як елементи були спочатку організовано в контейнері.

13.3. Алгоритм Хаффмана

Одним з варіантів застосування черги з пріоритетом є **система кодування Хаффмана**, в якій використовується частота появи кожного символу в повідомленні. Пара «символ–частота» вставляється в чергу з пріоритетом за зростанням частот. Тобто верхня пара міститиме символ, який трапляється в повідомленні найрідше. На цій основі створюється **двійкове дерево Хаффмана**. Для цього спочатку беруться дві перші пари з найменшою частотою. Перша пара утворює лівий листок дерева, а друга пара – правий листок. Сума їх частот утворює корінь поточного фрагмента дерева і записується в чергу з пріоритетом у відповідне місце. Ребро від кореня до лівого листка кодується нулем, а ребро від правого листка до кореня – одиницею. Процедура повторюється до моменту, коли в черзі залишиться один елемент, що утворює корінь остаточного дерева Хаффмана. Шлях від кореня до листка з кодами ребер становить відповідний код символа, який розміщений у цьому листку.

Кодування за алгоритмом Хаффмана призводить до економії ресурсів пам'яті та часу передавання повідомлень. До того ж повідомлення, що закодовані нефіксованою довжиною коду символу, довше декодуються.

Розглянемо приклад. Нехай деяке повідомлення **Т** містить символи від **а** до **з**, які трапляються в повідомленні з такою частотою:

а – 50; б – 20; в – 100; г – 80; д – 220; ж – 490; з – 40.

Кожну пару «символ–частота» вставляємо в чергу з пріоритетом, яка впорядкована за зростанням частоти:

(б, 20) (з, 40) (а, 50) (г, 80) (в, 100) (д, 220) (ж, 490)

На основі цієї черги будуємо дерево Хаффмана. Спочатку з черги вилучаємо перші два елементи. Символ б стає лівим листком двійкового дерева, символ з відповідно правим листком. Сума їх частот утворює корінь фрагмента майбутнього дерева Хаффмана (рис. 13.13) і вставляється в чергу у відповідне місце згідно із отриманою сумою. Черга набуде вигляду:

(а, 50) (, 60) (г, 80) (в, 100) (д, 220) (ж, 490)

Повторюємо попередню дію для наступної пари елементів із черги. У результаті фрагмент дерева набуде вигляду, як показано на рис. 13.14, а черга перетвориться на таку:

(г, 80) (в, 100) (, 110) (д, 220) (ж, 490)

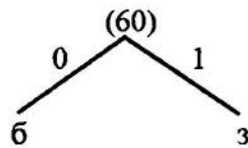


Рис. 13.13. Фрагмент майбутнього дерева Хаффмана для символів б та з

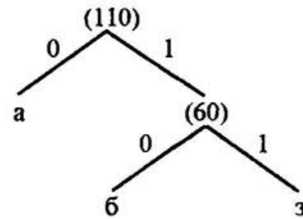


Рис. 13.14. Фрагмент майбутнього дерева Хаффмана для символів а, б та з

Наступні два елементи із черги не можуть бути зв'язані з основним деревом, тому вони утворять ліву і праву гілки піддерева дерева Хаффмана (рис. 13.15), а сума їх частот додасться до черги:

(, 110) (, 180) (д, 220) (ж, 490)

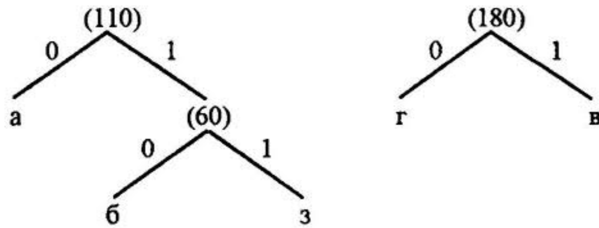


Рис. 13.15. Фрагмент майбутнього дерева Хаффмана для символів а, б,з та г,в

При виконанні чергової ітерації утворені піддерева об'єднуються (рис. 13.16), а в черзі залишається три елементи: (д, 220) (, 290) (ж, 490)

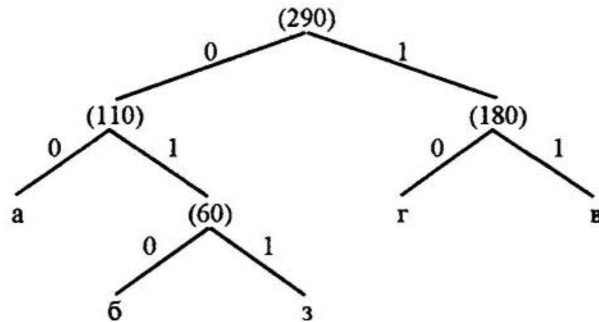


Рис. 13.16. Фрагмент майбутнього дерева Хаффмана з об'єднаними піддеревами

Остаточний вигляд дерева Хаффмана показано на рис. 13.17. Шлях від кореня до кожного листка дерева сформує відповідний код Хаффмана для символа, що розміщений у листку:

а – 1100; б – 11010; в – 1111; г – 1110; д – 10; ж – 0; з – 11011.

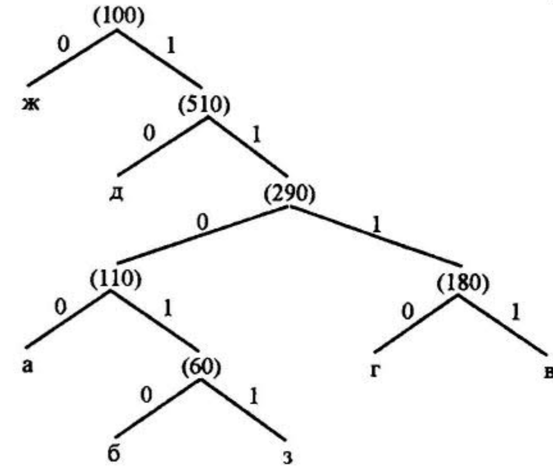


Рис. 13.17. Дерево Хаффмана

Маючи відповідні коди символів, легко перекласти повідомлення **Т** у закодоване повідомлення **К**.

Так, для повідомлення «багаж» відповідне закодоване повідомлення має вигляд:

1101011001110110000

Для дешифрування необхідно просуватись по дереву від кореня по тій гілці, яка відповідає символу кодування (0 – ліва гілка, 1 – права гілка), поки не дотягнеться листок. Для цього закодованого повідомлення перший прохід по гілках дерева буде таким:

права, права, ліва, права, ліва – символ **б**

Наступний прохід приведе до символа **а**:

права, права, ліва, ліва – символ **а**

Так повторюємо до отримання початкового повідомлення «багаж».

13.4. Збалансовані двійкові дерева пошуку

Відомо, що двійкові дерева пошуку є ефективними в середньому, а в гіршому випадку вони не дають будь-яких переваг перед векторами, двосторонніми чергами або списками. Існує декілька структур даних, що ґрунтуються на двійкових деревах пошуку, але при цьому завжди є

збалансованими. Двійкове дерево пошуку є **збалансованим,** якщо його висота логарифмічно залежить від **n** – кількості елементів у дереві.

Три найпоширеніші структури даних цієї категорії двійкових дерев пошуку: AVL – дерева, червоно-чорні дерева і косі дерева. Жодна з цих структур даних не входить до стандартної бібліотеки шаблонів. Але в реалізації стандартної бібліотеки шаблонів Hewlett - Packard клас **rb-tree** (для червоно-чорного дерева) використовується у приватному полі у визначенні чотирьох класів асоціативних контейнерів, що входять до стандартної бібліотеки шаблонів.

Основний механізм для перетворення двійкового дерева пошуку на збалансоване дерево є **поворот**: реструктуризація дерева навколо елемента, який визначає бажаний порядок елементів.

Є два основні види повороту.

1. Лівий поворот: елемент займає місце лівого дочірнього елемента, а правий дочірній елемент – місце батьківського елемента. Наприклад, на рис. 13.18 продемонстровано лівий поворот навколо елемента 50. До і після повороту дерево є двійковим деревом пошуку.

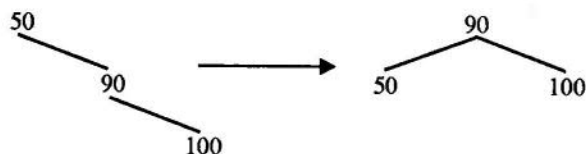


Рис. 13.18. Лівий поворот навколо елемента 50

На рис 13.19 наведено інший приклад лівого повороту навколо елемента 80, за якого висота дерева зменшується від 3 до 2. Цікавою особливістю є те, що елемент 85, який до повороту мав лівий дочірній елемент 90, у результаті стає правим дочірнім елементом елемента 80. Цей феномен є загальним для усіх лівих поворотів навколо елемента **x**: ліве піддерево правого дочірнього елемента **x** стає правим піддеревом елемента **x**. Це справедливо і для дерева на рис. 13.18, але тут ліве піддерево правого дочірнього елемента 50 порожнє.

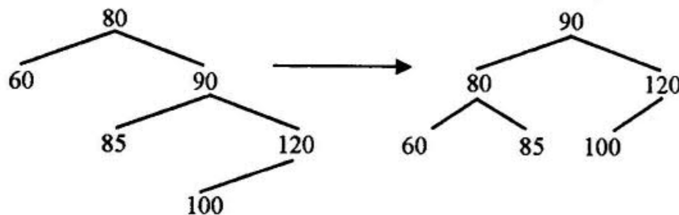


Рис. 13.19. Лівий поворот навколо елемента 80

На рис. 13.20 показано процес повороту дерева з рис 13.19 у ширшому контексті: елемент, навколо якого відбувається поворот, не є кореневим елементом дерева. Рис. 13.20 також ілюструє ще один аспект, властивий усім операціям повороту: всі елементи, які не належать до піддерева елементу, навколо якого відбувається поворот, не зачіпаються. В нашому прикладі це елементи 20, 30, 40, 50.

Код для операції повороту не передбачає будь-яких переміщень елементів; маніпуляції відбуваються тільки із вказівниками на вузли. Нехай **x** – це вказівник на вузол, **y** – вказівник на правий дочірній вузол вузла **x**. Лівий поворот навколо **x** здійснюють у два етапи:

`x -> right = y - > left;` // наприклад, елемент 85 на рис 13.19.

`y -> left = x;`

Нижче наведено повне визначення методу **rotate_left** класу **BinSearchTree**.

```
void rotateLeft (tree node* x)
{
tree_node* y = x -> right;
 x -> right = y -> left;
 if (y ->left != NULL)
y -> left -> parent = x;
y -> parent = x -> parent;
 if (x == header -> parent) // якщо x -корінь
header -> parent = y;
 else if (x == x -> parent -> left) // якщо x -
лівий
// дочірній елемент
x -> parent -> left = y;
else
x -> parent -> right = y;
y -> left - x;
x -> parent = y;
}
```

Позитивний момент: елементи не пересуваються, а час виконання залишається незмінним.

Більша частина коду, що реалізує поворот, вносить зміни до батьківського вузла елемента, навколо якого потрібно зробити поворот.

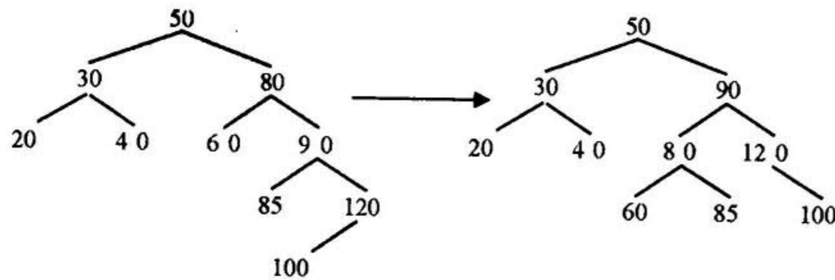


Рис. 13.20. Лівий поворот навколо елемента 80 для дерева з рис. 13.19, але 80 не є кореневим елементом

2. Правий поворот. На рис. 13.21 показано повтор навколо елемента 100. Нехай **x** – це вказівник на вузол дерева, **y** – вказівник на його дочірній вузол . Правий поворот навколо **x** можна здійснити у два кроки:

```
x -> left = y - > right;
y -> right = x;
```

Через необхідність внесення змін до батьківських вузлів метод стає набагато довшим, але час його виконання залишається постійним. Якщо змінити місцями left і right у визначенні методу **rotateLeft**, то отримаємо визначення методу **rotateRight**. Як і для лівого повороту, після правого повороту дерево залишатиметься двійковим деревом пошуку.

Рис. 13.21. Правий поворот навколо елемента 100

У розглянутих способах повороту висота дерева зменшувалась на 1. Зменшити висоту – головна мотивація для повороту. Але поворот не обов'язково призводить до зменшення висоти дерева. Наприклад, на рис. 13.22 показано лівий поворот навколо вузла 50, який не впливає на висоту дерева.

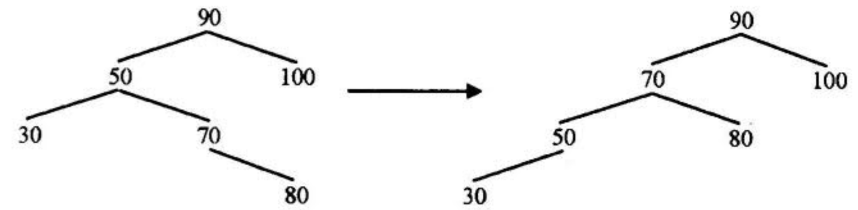


Рис. 13.22. Лівий поворот навколо елемента 50. Висота дерева після повороту залишилась 3

Повороти показані на рис. 13.22 і 13.23, треба розглядати в сукупності: лівий поворот навколо лівого дочірнього елемента 90 і потім правий поворот навколо елемента 90. Цей тип повороту називається **подвійним поворотом**.

На рис. 13.24 показано ще один вид подвійного повороту: правий поворот навколо правого дочірнього елемента 50 з подальшим лівим поворотом навколо елемента 50.

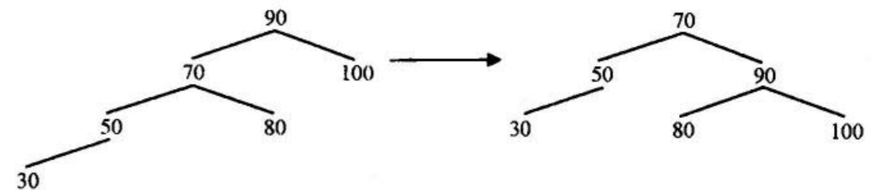


Рис. 13.23. Правий поворот навколо елемента 90. Висота дерева зменшилась з 3 до 2

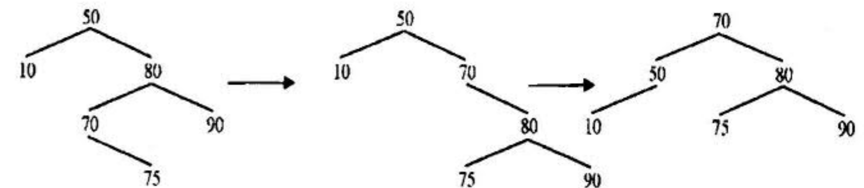


Рис. 13.24. Подвійний поворот: правий поворот навколо правого дочірнього елемента 50 з подальшим лівим поворотом навколо елемента 50

Основні властивості поворотів

1. Є чотири види поворотів:

а) лівий поворот;

б) правий поворот;

в) лівий поворот навколо лівого дочірнього елемента цього елемента з подальшим правим поворотом навколо самого цього елемента;

г) правий поворот навколо правого дочірнього елемента цього елемента з подальшим лівим поворотом навколо самого цього елемента.

2. Вузли в піддереві елемента, навколо якого здійснюється поворот, зазнають зміни під час повороту.

3. Час виконання повороту постійний.

4. До та після повороту дерево, як і раніше, залишається двійковим деревом пошуку.

5. Код для лівого повороту аналогічний до коду для правого повороту
(і навпаки): достатньо просто поміняти місцями слова left і right.

13.5. AVL-дерева

AVL-дерево – це двійкове дерево пошуку, яке або є пустим, або має такі дві властивості:

1. Висоти лівого й правого піддерев відрізняють не більше ніж на 1.

2. Ліве й праве піддерева є AVL-дерева.

Винахідниками AVL-дерев є два російські математики: Адельсон-Вельський (Adelson-Velsky) і Ландис (Landis), на честь яких AVL-дерева одержали свою назву. На рис. 13.25 показано два AVL-дерева, а на рис. 13.26 – три двійкові дерева пошуку, що не є AVL-деревами.

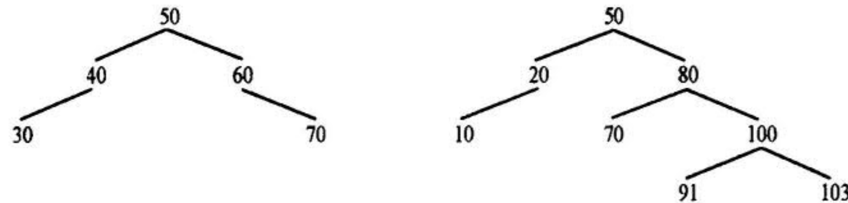


Рис. 13.25. Приклади AVL- дерев

Перше дерево, зображене на рис. 13.26, не є AVL-деревом, оскільки ні його ліве піддерево, ні його праве піддерево не є AVL-деревами. Друге дерево не є AVL- деревом, оскільки висота його лівого піддерева дорівнює 1, а висота його правого піддерева –а 3.

AVL-дерево є збалансованим двійковим деревом пошуку. Його висота завжди логарифмічно залежить від кількості елементів *n*. Це є перевагою AVL-дерев порівняно із узагальненими двійковими деревами пошуку, для яких висота лінійно залежить від *n* у найгіршому випадку.

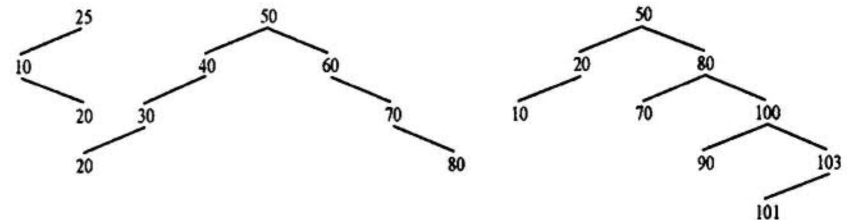


Рис. 13.26. Три двійкові дерева пошуку, що не є AVL-деревами

13.5.1 Застосування AVL-дерева: програма перевірки орфографії

Однією з найкорисніших функцій сучасних текстових процесорів є програма перевірки орфографії, яка сканує документ та шукає можливі орфографічні помилки. «Можливі», оскільки документ може містити цілком припустимі слова, але не знайдені в словнику.

Загальна постановка задачі така: маючи заданий словник **dictionaryFile** і документ, ім'я файла якого надається користувачем, вивести на друк всі слова документа, які не було знайдено в словнику [7].

Приймемо кілька спрощень:

1. У словнику присутні тільки слова з малих літер.

2. Кожне слово в документі складається тільки з букв.

3. За кожним словом у документі стоїть нуль або більше розділових знаків, за якими іде будь-яка кількість пробілів і символів кінця рядка.

4. Файл словника впорядкований за алфавітом і буде повністю розміщений у пам'яті. Файл документа не обов'язково впорядкований за алфавітом, буде повністю розміщений у пам'яті після вилучення дублікатів.

Нижче наведено файл словника, файл документа й слова документа, що не входять до словника.

```
// файл словника:
 a
all
and
be
done is
more
said
than
when
where
// файл документа:
When all is sed and done,
more is said then done.
// слова із імовірними помилками:
sed
```

Для розв'язання задачі створимо клас **Spellcheck** з такими методами:

// Зчитуються слова з файла словника; **n** – кількість слів у словнику

```
void readDictionaryFile();
```

// Зчитуються слова з файла документа; **k** – кількість слів у документі

```
void readDocumentFile();
```

// Виводяться всі слова, які є у документі, але відсутні в словнику.

```
void compare();
```

Єдиними полями є поле **dictionary**, яке зберігає слова, наявні у файлі словника, і поле **words**, яке зберігає всі унікальні слова у файлі документа. Обидва поля є об'єктами типу **AVLTree**, у яких кожний елемент являє собою рядок **(string)**, а рядки порівнюються за буквами:

```
protected:
 AVLTree<string, less<string> > dictionary, words;
```

Визначення методу **readDictionaryFile()** наведено нижче:

```
void readDictionaryFile() {
const string DICTIONARY_FILE = "dictfile.dat";
fstream dictionaryfile;
string word;
```



```
    dictionaryFile.open      (DICTIONARY_FILE.c_str(),
ios::in);
    while (dictionaryFile >> word)
    dictionary.insert (word);
    }
```

Припустимо, у файлі словника **dictionaryFile** є ***n*** слів. Тоді цикл **while** буде виконано ***n*** разів. На кожній ітерації здійснюється одна вставка в словник **dictionary.** Для кожної вставки в об'єкт **dictionary** типу **AVLTree** worstTime(n) логарифмічно залежить від ***n***, тому для методу **readDictionaryFile** worstTime(n) є **O(*n* log *n*)**, і це є найменшою верхньою межею.

Визначення методу **readDocumentFile** є складнішим. Прочитується ім'я файла документа, а потім зчитується файл. Букви кожного прочитаного з файла слова перетворяться на рядкові; якщо наприкінці слова є розділові знаки, вони відкидаються. Отримані слова вставляються в об'єкт **words,** якщо тільки таке слово вже не міститься в **words.** Повне визначення методу є таким:

```
void readDocumentFile() {
const string DOCUMENT_FILE_PROMPT =
" Будь ласка, введіть ім'я файла документа: ";
fstream documentFile;
string documentFileName;
word;
cout << endl << DOCUMENT_FILE_PROMPT;
cin >> documentFileName;
documentFile.open (documentFileName.c_str(), ios::in);
while (documentFile >> word)
{
// Перетворення букв на рядкові:
string temp;
for (unsigned i = 0; i < word.length(); i++)
temp += (char) tolower (word [i]);
word = temp;
// Усунення розділових знаків наприкінці слова:
while (!isalpha (word [word.length() - 1]))
word.erase (word.length() -1);
// вставка слова в об'єкт words,
// якщо такого слова  там ще немає:
if (words.find (word) == words.end())
words.insert (word);
} // поки в документі documentFile ще є слова
}
```

Для читання ***k*** слів з файла показник worstTime(k) лінійно залежить від ***k***. Для вставки кожного слова в об'єкт **words** типу **AVLTree** worstTime(k) логарифмічно залежить від ***k***. Отже, для методу **readDocumentFile** worstTime(k) є **O(k log k)**, і це є найменшою верхньою межею.

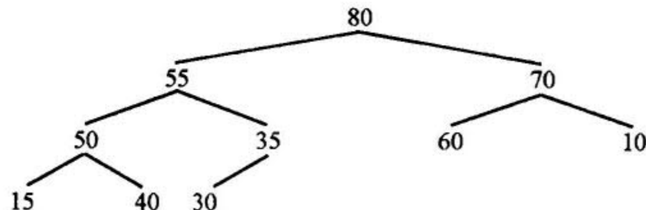
Визначення методу **compare**:

```
void compare() {
const string MISSPELLED =
"Слова з можливими помилками: ";
AVLTree < string, less <string> >::Iterator itr;
cout << endl << MISSPELLED << endl;
for (itr = words.begin(); itr != words.end();
itr++)
if (dictionary.find (*itr) == dictionary.end())
cout << *itr << endl;}
```

Для обробки ***k*** слів у **words** worstTime(k) лінійно залежить від ***k***, а для кожної операції пошуку ***n*** слів у **dictionary** за допомогою методу **find** worstTime(n) логарифмічно залежить від ***n***. Тобто, для методу **compare** worstTime(*k, n*) є **O(k log n)**, і це є найменшою верхньою межею.

Контрольні запитання та вправи

1. Які методи містить клас **priority_queue**?
2. Дайте означення кучі. Наведіть приклад кучі.
3. Які контейнери можна використати як кучу?
4. Який вигляд матиме дерево після послідовного виконання перерахованих нижче дій для кучі:



1) push (83) ;
2) push (61);
3) pop( ) ;

5. Припустимо, контейнер **c1** типу **vector** містить таку послідовність елементів:

10, 20, 30, 40, 50, 60, 70, 80, 90, 100. Покажіть, що відбудеться, якщо буде здійснено виклик:

make_heap (c1. begin (), c1.end(), less**<int>);**

6. Припустимо, контейнер **c2** типу **vector** містить ті самі елементи, що й вектор **c1** із вправи 5, але розташовані у зворотному порядку. Покажіть, що відбудеться, якщо буде здійснено такий виклик:

make_heap (c2.begin (), c2.end(), less**<int>);**

7. Для заданих частот появи символів створіть чергу з пріоритетами з парами «символ – частота появи» (найвищий пріоритет = найменша частота):

а - 10
б - 25
в - 15
г - 80
и - 22
к - 49
н – 34

8. Скористайтесь утвореною чергою з вправи 7 для побудови дерева Хаффмана.

9. На основі дерева Хаффмана з вправи 8 закодуйте слово «книга».

10. Скористайтесь деревом Хаффмана для перекладу бітової послідовності 1110101111100111010 назад у літери від «a» до «*g*» (рис. 13.27).

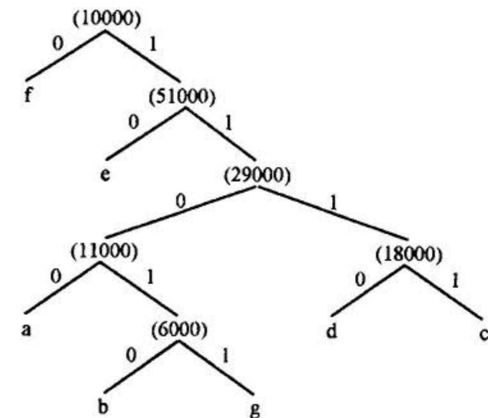
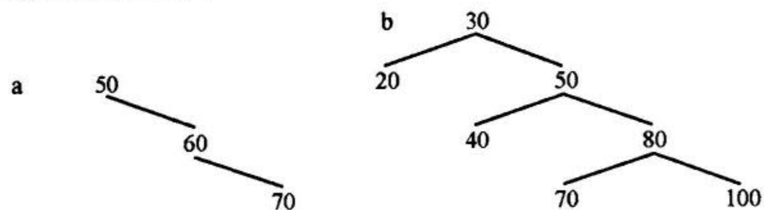
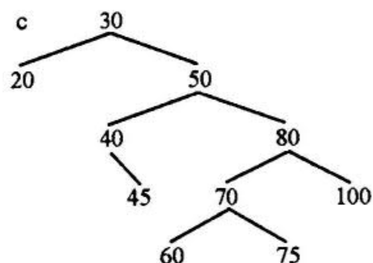


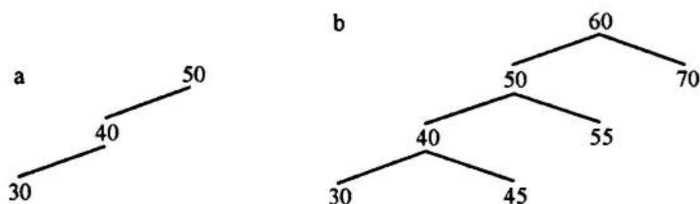
Рис. 13.27. Дерево Хаффмана

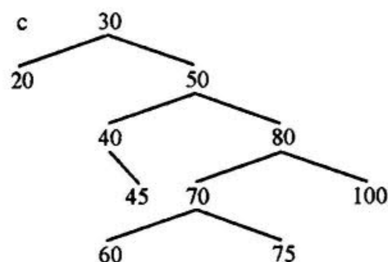
11. Для кожного з двійкових дерев пошуку виконайте лівий поворот навколо елемента 50.



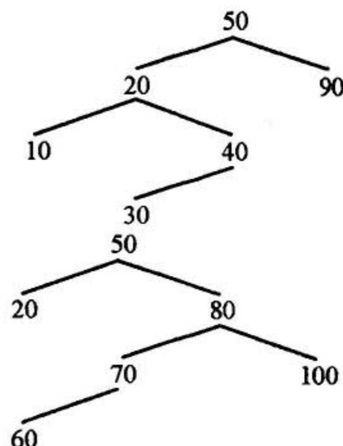


12. Для кожного з двійкових дерев пошуку виконайте правий поворот навколо елемента 50.





13. Для двійкового дерева пошуку виконайте подвійний поворот (лівий поворот навколо елемента 20, а потім правий поворот навколо елемента 50), щоб зменшити висоту дерева до 2:



14. Для двійкового дерева пошуку виконайте подвійний поворот, щоб зменшити висоту дерева до 2:

Приклади тестових питань

1. Для реалізації кучі можна використати контейнер:
   а) вектор;
   б) двостороння черга;
   в) мультикарта.

2. У системі кодування Хаффмана використовують:
   а) частоту появи символа в тексті;
   б) відстань місцезнаходження символу від початку тексту;
   в) відстань місцезнаходження символу від кінця тексту.

3. «Дірка» в кучі – це:
   а) тимчасова змінна;
   б) звільнена позиція;
   в) видалений елемент.

4. В алгоритмі кодування Хаффмана використовують:
   а) префіксне кодування;
   б) суфіксне кодування;
   в) безпрефіксне кодування.

5. До двійкових дерев пошуку належать дерева:

а) червоно-чорні;

б) AVL-дерева;

в) косі дерева;

г) триарні дерева.

6. Для перетворення двійкового дерева пошуку на збалансоване є така кількість видів повороту:

а) 3;

б) 4;

в) 5.

7. Висота лівого та правого піддерев у AVL-дереві відрізняється на:

а) 1;

б) 2;

в) кількість листків у кожному піддереві.

14. ЧЕРВОНО-ЧОРНІ ДЕРЕВА

Червоно-чорне дерево являє собою двійкове дерево пошуку, елементи якого виділяються певним кольором відповідно до прийнятої угоди. Зокрема одні елементи виділяються червоним, а інші чорним кольором. За правилом, відповідно до якого елементи виділяються або чорним, або червоним, використовується поняття шляху. Нагадаємо: якщо елемент **A** є попередником елемента **B,** то ***шлях*** від **A** до **B** являє собою послідовність елементів, що починається з **A** і закінчується на **B,** у якій кожний елемент (за винятком останнього) є батьком наступного елемента.

Розглянемо шляхи від кореня до елементів, що не мають дочірніх елементів або які мають один дочірній елемент. Наприклад, у дереві на рис. 14.1 є п'ять шляхів від кореня до елементів (зображені в рамці) без дочірніх або з одним дочірнім елементом.

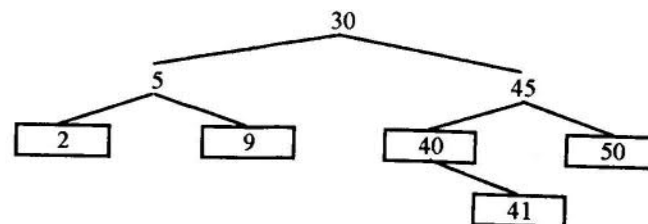


Рис. 14.1. Приклад червоно-чорного дерева

Один зі шляхів веде до елемента 40, який має один дочірній елемент. Тобто, шляхи не обов'язково ведуть до листків.

Червоно-чорне дерево – це двійкове дерево пошуку, яке є порожнім або в якому кореневий елемент має чорний колір – усі інші елементи червоного або чорного кольору, і виконуються такі правила:

– ***правило червоного:*** якщо елемент червоний, його батько повинен бути чорним;

– ***правило шляху:*** кількість чорних елементів повинна бути однаковою для всіх шляхів від кореневого елемента до елемента, що не має дочірніх або має один дочірній елемент.

Кожне червоно-чорне дерево має задовольняти правило червоного й правило шляху. Зображаючи червоно-чорні дерева, чорні елементи зображатимемо звичайним шрифтом, а червоні – ***напівжирним курсивом.***

Наприклад, на рис. 14.2 наведено червоно-чорне дерево, в якому елементи є цілими числами. Це є двійкове дерево пошуку із чорним коренем. Оскільки в ньому немає червоних елементів із червоними батьками, то правила червоного дотримано. Крім того, у кожному з п'яти шляхів від кореня до елементів, що не мають дочірніх або мають один дочірній елемент, є два чорні елементи, правило шляху дотримане. Отже, дерево є червоно-чорним.

Дерево, зображене на рис. 14.3, не є червоно-чорним деревом, незважаючи на те, що воно задовольняє правило червоного, і кожний шлях від кореня до листка містить однакову кількість чорних елементів. Правило шляху не дотримується, оскільки шлях від елемента 50 до елемента 80 містить тільки один чорний елемент, тоді як шлях від елемента 50 до елемента 20 містить два чорні елементи, як і шлях від елемента 50 до елемента 100.

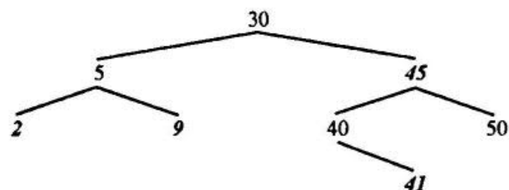


Рис. 14.2. Червоно-чорне дерево з вісьма елементами

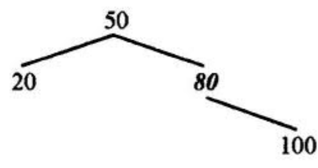


Рис. 14.3. Двійкове дерево пошуку, яке не є червоно-чорним деревом

Навіть, якщо кожний шлях від кореня до листка містить однакову кількість чорних елементів, правила шляху можна не дотримуватися.

Аналогічно дерево, зображене на рис. 14.4, не є червоно-чорним деревом. Правила шляху не дотримуються, оскільки шлях від елемента 70 до елемента 40 містить три чорні елементи, але шлях від елемента 70 до елемента 110 містить чотири чорні елементи. Це дерево не збалансоване: висота будь-якого дерева з двома листками лінійно залежатиме від *n*.

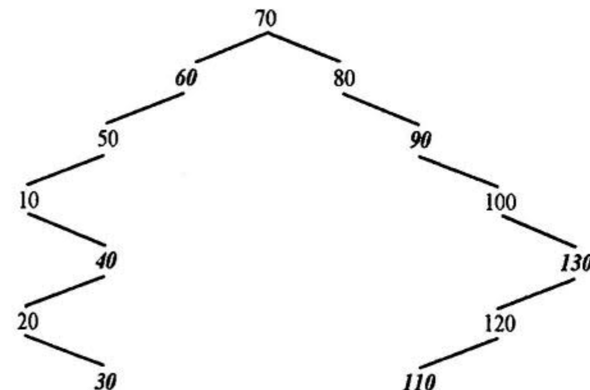


Рис. 14.4. Двійкове дерево пошуку, яке не є червоно-чорним деревом

Червоно-чорне дерево, зображене на рис. 14.2, дуже добре збалансоване, але не кожне червоно-чорне дерево має таку характеристику. Наприклад, на рис. 14.5 показано дерево, яке «завалюється» ліворуч. Це є двійкове дерево пошуку із чорним коренем, і правило червоного дотримано. Щодо правила шляху, то будь-який шлях від кореня до елемента, що не має дочірніх або має один дочірній елемент, містить рівно два чорні елементи. Тобто, дерево є червоно-чорним деревом. Однак існують певні межі розбалансу червоно-чорного дерева. Наприклад, не можна «підвісити» ще один елемент під елементом 10 на рис. 14.5. Якщо додати червоний елемент, не буде дотримано правило червоного, а якщо додати чорний елемент, буде порушено правило шляху.

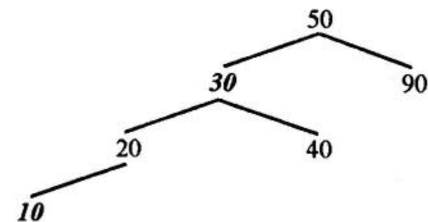


Рис. 14.5. Червоно-чорне дерево, яке погано збалансоване

Якщо червоно-чорне дерево є закінченим деревом, всі елементи якого є чорними, за винятком червоних листків на найнижчому рівні, висота цього дерева буде мінімальною й дорівнюватиме приблизно $log_2\ n$. Щоб висота дерева для заданої кількості елементів була максимальною, дерево повинне мати якнайбільше червоних елементів на одному шляху, а всі інші елементи повинні бути чорними.

Наприклад, на рис. 14.5 наведено одне таке дерево, а на рис. 14.6 – інше таке дерево. Шлях із усіма червоними елементами буде майже удвічі довшим, ніж шлях, що не містить червоних елементів.

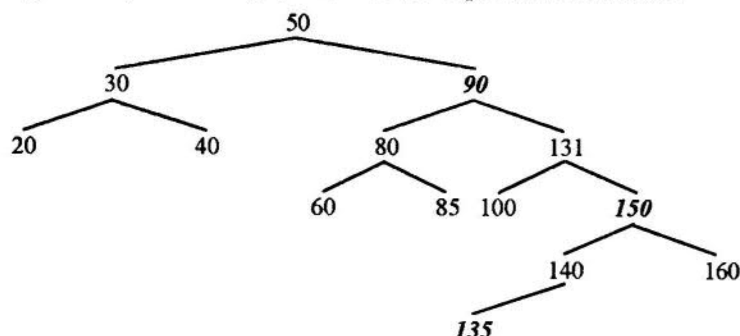


Рис. 14.6. Червоно-чорне дерево з 14 елементами й з максимальною висотою, що дорівнює 5

14.1. Висота червоно-чорного дерева

Червоно-чорні дерева є досить «густими» у тому розумінні, що майже всі елементи, що не є листками, мають по два дочірні елементи. Якщо елемент має тільки один дочірній, то цей елемент повинен бути чорним, а його дочірній елемент повинен бути червоним листком. Отже, червоно-чорне дерево є збалансованим, тобто має **висоту**, яка логарифмічно залежить від *n* навіть у найгіршому випадку. Для двійкового дерева пошуку в найгіршому випадку висота лінійно залежить від *n*. Щоб остаточно зробити висновок, що висота червоно-чорних дерев завжди логарифмічно залежить від *n*, розглянемо кілька проміжних тверджень.

Твердження 1

Нехай *y* – корінь піддерева червоно-чорного дерева. Кількість чорних елементів на будь-якому шляху від *y* до кожного з його нащадків, що не має дочірніх або має один дочірній елемент, є однаковою.

Щоб довести це твердження, припустімо, що *x* є коренем червоно-чорного дерева, а *y* – коренем піддерева. Нехай *b0* – кількість чорних елементів на шляху від *x* (включаючи) до *y* (не включаючи), а *b1* – кількість чорних елементів на шляху від *y* (включаючи) до кожного з його нащадків (включаючи), що не має дочірніх або має один дочірній елемент. Нехай *b2* – кількість чорних елементів на шляху від *y* (включаючи) до будь-якого іншого з його нащадків (включаючи), що не має дочірніх або має один дочірній елемент. Цю ситуацію проілюстровано на рис. 14.7.

Повернувшись до дерева, зображеного на рис. 14.6, припустімо, що *x* є 50, *y* є 131, а двома нащадками *y* є 100 і 135. Тоді кількість чорних елементів на шляху від 50 до 90 дорівнює *b0* = 1; елемент 131 не включається; *b1*=*b2*=2.

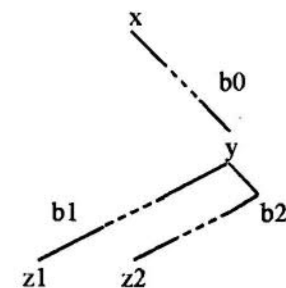


Рис. 14.7. Частина червоно-чорного дерева з коренем x

Відповідно до правила шляху, що стосується всього дерева, повинна дотримуватися рівність *b0*+*b1*= *b0*+*b2*. Із цього випливає, що *b1*=*b2*. Отже, кількість чорних елементів однакова для будь-якого шляху від *y* до кожного з його нащадків, що не має дочірніх або має один дочірній елемент.

Тепер, коли твердження 1 доведено, можна дати таке визначення.

Нехай *y* – елемент у червоно-чорному дереві; тоді ***чорну висоту*** *y*, *bh(y)* можна визначити так:

bh(y) дорівнює кількості чорних елементів у будь-якому шляху від *y* до будь-якого його нащадка, що не має дочірніх або має один дочірній елемент.

Відповідно до твердження 1, кількість чорних елементів повинна бути однаковою в будь-якому шляху від елемента до будь-якого його нащадка, що не має дочірніх або має один дочірній елемент. Тобто чорна висота є цілком визначеною характеристикою. Наприклад, для дерева, зображеного на рис. 14.5, чорна висота елемента 50 дорівнює 2; чорна висота елементів 20, 30 і 40, а також елемента 90 дорівнює 1; чорна висота елемента 10 дорівнює 0. Для червоно-чорного дерева, зображеного на рис. 14.8, чорна висота елемента 60 дорівнює 3, а чорна висота елемента 85 дорівнює 2.

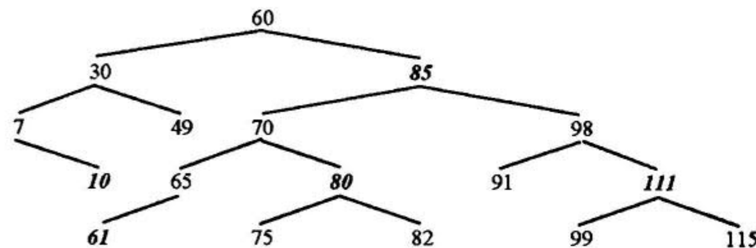


Рис. 14.8. Червоно-чорне дерево, корінь якого має чорну висоту, що дорівнює 3

Твердження 2

Для будь-якого непустого піддерева *t* червоно-чорного дерева $n(t) > 2^{bh(root(t))} - 1$,

де $n(t)$ – кількість елементів у піддереві *t*, а $root(t)$ – кореневий елемент піддерева *t*. Доведення цього твердження виконаємо методом математичної індукції стосовно висоти піддерева *t*.

Базовий випадок

Припустімо, що $height(t) = 0$. Тоді $n(t) = 1$, а $bh(root(t)) = 1$, якщо корінь є чорним, і $bh(root(t)) = 0$, якщо корінь піддерева *t* є червоним.

У кожному разі $bh(root(t)) <= 1$. Тобто, $n(t) = 1 = 2^1 - 1 >= 2^{bh(root(t))} - 1$

Це доводить твердження 2 для базового випадку.

Індуктивний випадок

Нехай *k* – будь-яке невід'ємне ціле число; припустімо, що твердження 2 істинно для будь-якого піддерева, висота якого менша або дорівнює *k*. Нехай *t* – піддерево заввишки *k+1*. Якщо корінь *t* має один дочірній елемент, то повинна виконуватися рівність $bh(root(t)) = 1$, тому

$$n(t) = 1 = 2^1 - 1 >= 2^{bh(root(t))} - 1.$$

Це доводить індуктивний випадок, якщо корінь *t* має тільки один дочірній елемент. В іншому випадку корінь *t* повинен мати лівий дочірній елемент, *v1*, і правий дочірній елемент, *v2*. Якщо корінь *t* є червоним, то $bh(root(t)) = bh(vl) = bh(v2)$. Якщо корінь *t* є чорним, то $bh(root(t)) = bh(v1) + 1 = bh(v2) + 1$. У кожному разі $bh(v1) >= bh(root(t)) - 1$ і $bh(v2) >= bh(root(t)) - 1$.

Ліве та праве піддерева мають висоту, що є меншою або дорівнює *k*, отже, можна застосувати індуктивне припущення:

$$n(leftTree(t)) >= 2^{bh(vl)} - 1 \text{ та } n(rightTree(t)) >= 2^{bh(v2)} - 1.$$

Кількість елементів в *t* на один більша від кількості елементів в *leftTree(t)* та *rightTree(t)*. Враховуючи це, отримуємо:

$$\begin{aligned} n(t) &= n(leftTree(t)) + n(rightTree(t)) + 1 \\ &>= 2^{bh(vl)} - 1 + 2^{bh(v2)} - 1 + 1 \\ &>= 2^{bh(root(t))-1} - 1 + 2^{bh(root(t)-1} - 1 + 1 \\ &>= 2 * 2^{bh(root(t))-1} - 1 >= 2^{bh(root(t))} - 1 \end{aligned}$$

Це доводить індуктивний випадок, якщо корінь *t* має два дочірні елементи. Отже, згідно із принципом математичної індукції, твердження 2 істинне для всіх непустих піддерев червоно-чорних дерев.

Твердження 3

Для будь-якого червоно-чорного дерева *t* з *n* елементами висота *height(t)* логарифмічно залежить від *n*.

Висота червоно-чорного дерева логарифмічно залежить від *n* навіть у найгіршому випадку. Щоб довести твердження 3, припустімо, що *t* – червоно-чорне дерево. Відповідно до правила червоного не більше половини елементів на шляху від кореня до найвіддаленішого листка можуть бути червоними, тому не менше половини цих елементів повинні бути чорними. Отже:

$$bh(root(t)) >= height(t)/2$$

За твердженням 2:

$n(t) >= 2^{bh(root(t))} - 1 >= 2^{height(t)/2} - 1.$

Тоді

$$height(t) <= 2 \log_2(n(t) + 1).$$

Це означає, що *height(t)* є $O(\log n)$. Оскільки $height(t) >= \log_2(n(t) + 1) - 1$, то $O(\log n)$ є найменшою верхньою межею *height(t)*. Висота дерева *height(t)* логарифмічно залежить від *n*.

Згідно із Твердженням 3, червоно-чорні дерева ніколи не втрачають збалансованості. З іншого боку, для двійкових дерев пошуку висота може лінійно залежати від *n* у найгіршому випадку, наприклад, якщо дерево є ланцюжком.

14.2. Клас rb_tree Hewlett-Packard

Розглянемо клас **rb_tree** Hewlett - Packard. Перевага його в тому, що час виконання методів find, insert і erase дорівнює O(log *n*) навіть у найгіршому випадку. В стандартній бібліотеці шаблонів немає класу, що відповідає червоно-чорним деревам. Будь-яка реалізація цієї бібліотеки радше міститиме екземпляр класу червоно-чорного дерева, сконструйованого за зразком класу **rb_tree**, який є полем у визначеннях класів **set, multiset, map, multimap**. Основною концепцією, необхідною для розуміння класу **rb_tree**, є ***ключ.*** Нагадаємо, що ключ елемента являє собою частину цього елемента, яку використовують для порівняння з іншими елементами. Наприклад, номер посвідчення соціального страхування можна використати як ключ для елемента, що описує працівника, а ідентифікаційний номер студента – як ключ для елемента, що описує студента.

Початок визначення класу **rb_tree**:

```
template <class Key, class Value, class
KeyOfValue,class Compare>
   class rb_tree {
```

class Key представляє тип ключів. При реалізації об'єкта класу **rb_tree** фактичний клас заміняє псевдотип Key. Наприклад, якщо ключі мають тип string, реалізація екземпляра починається так:

```
rb_tree< string, ... > my_tree;
```

Шаблонний аргумент, відповідний до шаблонного параметра **Value**, являє собою тип елементів, що містяться в дереві. Часто ключі й значення збігаються.

Наприклад, якщо елементами є виробники автомобілів (такі як «Ford»), то ключ може містити сам елемент. Тобто елемент «Ford» зберігатиметься в дереві, у разі порівняння цього елемента з іншими ключове слово «Ford» порівнюватиметьсь з іншими ключами.

Наступний шаблонний параметр **KeyOfValue** є типом класу-функції. Параметр **KeyOfValue** заміняється на клас-функцію, у якому оператор **operator**() повертає ключ зі значення. Припустімо, що **key** – це об'єкт-функція в цьому класі-функції, а елемент **v** типу **Value**. Розглянемо два випадки:

1. Ключ і значення збігаються. У цьому випадку **key (v)** просто повертає **v**. Цей випадок трапляєтьсядля класів **set** і **multiset** зі стандартної бібліотеки шаблонів.

2. v є парою, перший компонент якої являє собою ключ. У цьому випадку **key (v)** повертає перший компонент v. Це зустрічається у класах **map** і **multimap** зі стандартної бібліотеки шаблонів.

Наприклад, кожне значення складається з пари: виробник автомобілів і обсяг продажів. Тоді ключем є назва виробника, а другий компонент у кожній парі містить обсяг продажів для цього виробника. Наприклад, двома такими парами можуть бути «Ford», 14; «Honda», 22

Останній шаблонний параметр **Compare** являє собою ще один тип класу-функції. Об'єкт класу **rb_tree** часто реалізується із вбудованим класом-функцією **less** як четвертий шаблонний аргумент. Ключі зазвичай порівнюються за допомогою оператора <. Клас-функція **less**, визначений у бібліотеці <function>, перевантажує **operator()** у такий спосіб:

```
bool operator() (const T& x, const T& y) const
{return x < y;}
```

Отже, якщо **less** є четвертим шаблонним аргументом у реалізації об'єкта класу **rb_tree**, ключі порівнюються згідно з визначенням **operator<** (порівняння на «менше»). Для виконання порівнянь усередині об'єкта класу **rb_tree** визначається об'єкт-функція:

```
Compare key_compare;
```


Отже, якщо **less** – четвертий шаблонний аргумент у реалізації об'єкта класу **rb_tree**, інструкцію усередині цього об'єкта, таку як `key_compare (x, y)`, можна інтерпретувати як `x < y`.

Поля в класі **rb_tree** аналогічні полям у класі **list**. Базова організація класу **rb_tree** подібна до організації класу **list.** Тут також є вузли і поля header, free_list, buffer_list, next_avail, last. Крім того, клас **rb_tree** має власні засоби управління пам'яттю на основі методів get_node, put_node. Структура кожного вузла відображає суть червоно-чорного дерева.

Визначеня структури **rb_tree** є таким:

```
enum  color_type = {red, black};
struct rb_tree_node {
 color_type color_field;
 rb_tree_node* parent_link;
 rb_tree_node* left_link;
 rb_tree_node* right_link;
 Value value_field;   }
```

Прийнято дві угоди щодо кольору вузлів:

1. Вузол заголовка **header** має червоний колір (red).

2. При початковій вставці вузла він має червоний колір (red); зміна кольору може знадобитися, щоб не порушувалося правило червоного.

Здебільшого доступ до полів здійснюється за допомогою функцій **parent**, **color** і **left**. У цьому зв'язку для повернення посилання на поле **parent_link** вузла **header** використовується запис **parent (header)**, а не **(*header).parent_link**. Одна із причин використання таких функцій доступу полягає в тому, що їх визначення інкапсулюють усі деталі, пов'язані з моделлю виділення пам'яті.

Ще одне спрощення: вказівник **NULL** можна інтерпретувати як вказівник на звичайний вузол **rb_tree_node**: є особливий вузол, **NIL** чорного кольору (black), поля **left** і **right** якого є **NULL**, а поле **parent** є **NULL**, принаймні початково. Це демонструє ще одну перевагу застосування функцій доступу: можна перевірити, чи повертає **color (y)** колір black, не передбачаючи особливого випадку для значення **NULL**.

Порівняно з методом **find** класу **AVLTree** визначення методу **find** класу **rb_tree** абстрактніше (для одержання ключа зі значення використовується об'єкт-функція) і складніше для розуміння (через наявність оператора «кома» й умовного оператора). Однак базовий

алгоритм залишається таким самим, а worstTime(n) так само логарифмічно залежить від *n:*

```
Iterator find (const key& k)
{
rb_tree_node* y = header; rb_tree_node* x =
root();
 while (x != NIL)
if (!key_compare (key(x), k))
{  y = x; x = left (x);}
else
x = right (x);
     iterator j = iterator(y);
return (j == end() || key_compare (k, key (j.node))) ?
 end () : j ;     }
```

Визначення для методів **insert** і **erase** трохи коротше, але складніше, ніж для відповідних методів класу **AVLTree**. Визначення для методів **insert** і **erase** не є очевидними. Червоно-чорні дерева вперше було визначено в 1972 р. R. Bayer, «Symmetric binary B-trees: Data Itructures and maintenance algorithms». Алгоритми вставки й видалення для цих дерев, які були названі «2–3–4 деревами», були доволі довгими, але загальні стратегії вставки й видалення були прості для розуміння. У роботі «A diochromatic fram® work for balanced trees» L. Guibas, R. Sedgewick у 1978 р. запропонували коротші, але водночас важчі для розуміння методу у разі використання червоно-чорного виділення елементів у таких структурах.

14.3. Метод insert класу rb_tree

Заголовок методу **insert** має такий вигляд:

```
pair <iterator, bool> insert (const value_type& v)
```

У реалізації стандартної бібліотеки шаблонів Hewlett-Packard клас **rb_tree** слугує основою для чотирьох класів асоціативних контейнерів: **set**, **multiset**, **map** і **multimap**. У класах **set** і **map** неприпустима вставка значення, ключ якого збігається із ключем деякого значення, що вже є в контейнері. Але в класах **multiset** і **multimap** ключі, що повторюються, допускаються. Щоб мати можливість розрізняти ці ситуації, в класі **rb_tree** передбачено поле: **bool** insert_always; Кожний конструктор **rb_tree** має параметр

always типу **bool**, який ініціалізує поле **insert_always**. Отже, для класів **set** і **map** у поле типу **rb_tree** для аргументу, відповідного до параметра **always**, встановлюється значення **false**, що забороняє дублювання ключів. Для класів **multiset** і **multimap** аргумент має значення **true**, а ключі, що повторюються, дозволяються.

При виклику методу **insert** класу **rb_tree** у випадку, якщо значенням **insert_always** є **true** або якщо ключ **v** не є дублікатом вже наявного в дереві ключа, елемент **v** вставляється в дерево, пара, що повертається, складається з ітератора, встановленого на значення, що вставляється, і булевого значення **true**. Але якщо значенням поля **insert_always** є **false** і ключ **v** є дублікатом ключа, уже наявного в дереві, елемент **v** не вставляється, а пара, що повертається, складається з ітератора, встановленого на початковому значенні із заданим ключем, і булевого значення **false**.

Вставка елемента **v** передбачає п'ять кроків:

1. Створення вузла, на який указує **x**.
2. Збереження елемента **v** у поле **value_field** вузла **x**.
3. Вставка цього вузла як листка за допомогою операції вставки, прийнятої у класі **BinSearchTree**, і установка для **x** кольору **red** (червоний).
4. Корегування дерева зміною кольору елементів та його структури, якщо це необхідно.
5. Установка для кореневого елемента кольору **black** (чорний).

Метод **insert** фактично виконує тільки дії 1 і 2, після чого звертається до методу **_insert**, що приймає як аргументи **x**, батька **x** та елемент, що вставляється. Єдиною дією, яка потребує деяких міркувань, є дія 4. Спочатку з'ясуємо, чому дія 4 необхідна. Припустімо, був вставлений елемент 20 у червоно-чорне дерево і після виконання дії 3 дерево виглядало так, як на рис. 14.9:

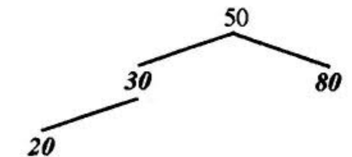


Рис. 14.9. Червоно-чорне дерево після вставки елемента 20

Правило червоного виявилося порушене, тому слід змінити колір вузлів 30 і 80 із червоного на чорний. У результаті буде отримано наступне дерево (рис. 14.10):

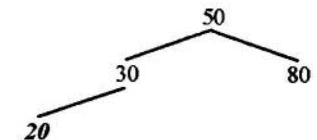


Рис. 14.10. Червоно-чорне дерево після зміни кольорів

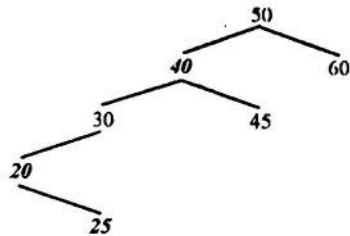


Рис. 14.11. Червоно-чорне дерево після вставки елемента 25

Іноді простої зміни кольору буде недостатньо. Припустімо, що елемент 25 вставляється в червоно-чорне дерево, яке після цього набуває вигляду, як на рис. 14.11:

Для цього дерева порушене правило червоного, тому воно потребує корегування. Це дерево неможливо знову зробити червоно-чорним лише однією зміною кольору. Оскільки чорна висота цього дерева дорівнює 2, шлях, що проходить через елементи 50, 40, 30, 20 і 25, має містити рівно 2 чорні вузли. Але корінь, елемент 50, повинен бути чорним, тому шлях, що проходить через елементи 40, 30, 20 і 25, повинен містити рівно один чорний елемент. Правило червоного вимагає, щоб шлях із чотирьох елементів містив не більш двох червоних вузлів. Але шлях із чотирьох елементів не може мати тільки один чорний елемент і при цьому не більше ніж два червоні елементи.

Перебудова, яку потрібно виконати в таких випадках, полягає в повороті частини дерева. Розробимо загальний підхід до такої перебудови.

Припустімо, встановлено вузол **x**. Чи потрібно виконувати поворот? Відповіддю буде «ні», якщо **x** є коренем, оскільки буде встановлено чорний колір кореня на кроці 5 операції вставки. Аналогічно жодної перебудови не потрібно, якщо батько **x** має чорний колір, оскільки в цьому випадку Правило червоного не порушується. Тобто, продовжується цикл, поки не буде задоволено одну з таких умов:

```
while (x!= root() && color (parent (x)) == red)
```

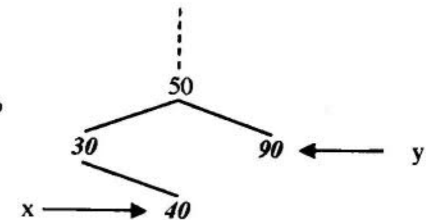
Критичним показником є колір братнього вузла батька **x**. Нижче розглядається, що відбувається, якщо батько **x** є лівим дочірнім елементом. Замінивши «лівий» на «правий» (left на righe) і навпаки, отримаємо план дій на випадок, якщо батько **x** є правим дочірнім елементом. Нехай **y** вказує на (правий) братній вузол батька **x**. Слід врахувати три випадки.

ВИПАДОК 1

`color (y) = red`

Припустімо, що елемент 40 вставлено у червоно-чорне дерево, фрагмент якого показано на рис. 14.12:

Рис. 14.12. Фрагмент червоно-чорного дерева після вставки елемента 40

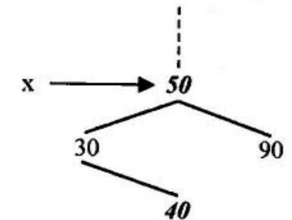


У цьому випадку необхідно виконати такі дії:

```
color (y) = black;
color (parent (x)) = black;
color (parent (parent (x))) = red;
x = parent (parent (x));
```

У результаті фрагмент дерева набуде вигляду, як на рис. 14.13:

Рис. 14.13. Фрагмент червоно-чорного дерева після зміни кольорів



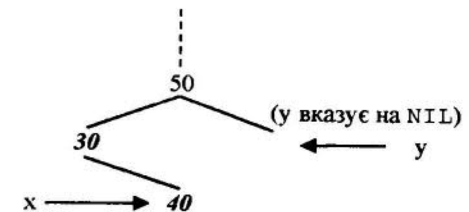
Після цього випадку і тільки після нього продовжується виконання циклу. Але тепер x перебуває в дереві на два рівні далі, тому максимальна кількість ітерацій циклу становить одну другу від висоти дерева. Саме із цієї причини для методу **insert** класу **rb_tree** worstTime(n) логарифмічно залежить від кількості елементів у дереві n.

ВИПАДОК 2

`color (y) = black , x - правий дочірній елемент`

Припустимо, що елемент 40 вставлено у червоно-чорне дерево, яке після цього набуло такого вигляду, як на рис. 14.14:

Рис. 14.14. Фрагмент червоно-чорного дерева після вставки елемента 40



Зверніть увагу, що коли **y** є ***NULL***, **y** стає вказівником на вузол ***NIL***, колір якого чорний. У цьому випадку здійснюють такі дії:

```
x = parent (x);
rotate_left (x);
```

Після цього лівого повороту дерево набуде такого вигляду, як на рис. 14.15:

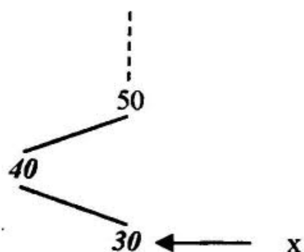


Рис. 14.15. Фрагмент червоно-чорного дерева після лівого повороту

Оскільки x та його батько обоє є червоними, це приводить до випадку 3.

ВИПАДОК 3

color (y) = black, x - лівий дочірній елемент

Припустімо, вставлено елемент 30 у червоно-чорне дерево, яке після цього набуло вигляду, як на рис. 14.16:

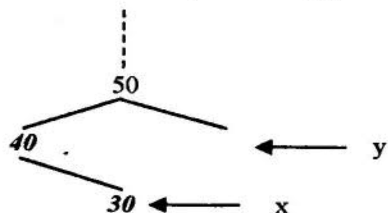


Рис. 14.16. Фрагмент червоно-чорного дерева після вставки елемента 30

У цьому випадку здійснюють такі дії:

```
color (parent (x)) = black;
color (parent (parent (x))) = red;
rotate right (parent (parent (x)));
```

Фрагмент дерева після цього правого повороту показано на рис. 14.17.

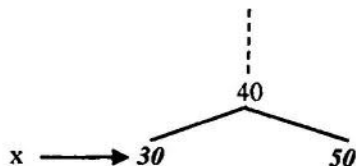


Рис. 14.17. Фрагмент червоно-чорного дерева після правого повороту

Після цих дій для випадку 3 колір батька x буде завжди чорним, а це є умовою завершення циклу **while**.

Основна ідея полягає в тому, що якщо випадок 1 не застосовується, то спочатку перевіряється, чи можна застосувати випадок 2, а потім, незалежно від того, застосований випадок 2 чи ні, застосовується випадок 3. Випадок 3 завжди застосовується після

випадку 2. Отже, загальна структура матиме вигляд (коли батько x є лівим дочірнім елементом):

```
if (y червоний) { // Випадок 1
...
}
else { // y повинен бути чорним
if (x є правим дочірнім елементом) { // Випадок 2
...
 }
// Випадок 3
...
}
```

Якщо під час будь-якої ітерації циклу випадок 1 не застосовується, повинен бути застосований випадок 3, і для батька x буде встановлено чорний колір. Після цієї ітерації виконання цикла буде завершено.

```
 while (x! = root() && color (parent(x) ==
red)
     if (parent (x) == left (parent
(parent(x))))
// якщо parent(x) - лівий дочірній елемент
      {
      y = right (parent(parent(x)));
      if (color(y) == red) // Випадок 1
      {
      color(parent(x)) = black;
      color(y) = black;
      color(parent (parent(x))) = red;
      x = parent (parent(x));
      }
      else   // колір y повинен бути чорним
      {
      if (x == right(parent(x))) // Випадок 2 {
      x = parent(x);
      rotate_left(x);
      }
      // Випадок 3
      color(parent(x)) = black;
      color (parent (parent(x))) = red;
       rotate_right(parent(x)));
      }
}
else   // parent(x) -правий дочірній елемент
{
```

```
y = left (parent(parent(x)));
if (color(y)) == red // Випадок 1
 {
      color(parent(x)) = black;
      color(y) = black;
      color(parent (parent(x))) = red;
      x = parent (parent(x));
}
else    // колір y повинен бути чорним
{
      if (x == left (parent(x))) // Випадок 2
      {
      x = parent(x);
      rotate_right(x);
      }
      // Випадок 3
      color(parent(x)) = black;
      color (parent (parent(x))) = red;
      rotate_left (parent (parent(x)));
      }
}
```

Після циклу слід виконати інструкцію

```
color (root()) = black;
```

Отже, для методу **insert** загалом worstTime(n) логарифмічно залежить від n.

14.4. Метод erase

Заголовок методу **erase**:

void erase (iterator position)

За цим методом видаляється з дерева вузол, на якому встановлено ітератор **position**. Наприклад, якщо вузол, що видаляється, має тільки один дочірній елемент, то цей дочірній елемент заміняє вузол, що видаляється. Якщо вузол, що видаляється, має два дочірні елементи, то безпосередній послідовник вузла, що видаляється, заміняє в дереві цей вузол. Висота червоно-чорного дерева є O(log ***n***), тому час виконання цієї частини методу **erase** класу **rb_tree** у найгіршому випадку логарифмічно залежить від кількості елементів. Однак видалення – це ще не все, оскільки після операції видалення можуть виявитися порушеними Правила червоного або шляху.

Якщо вузол, що видаляється, має тільки один дочірній елемент, то зробити залишається зовсім небагато. Вузол, що видаляється,

повинен бути чорним (black), а вузол, що заміняє, повинен бути червоним (red). Тому ми встановлюємо для вузла, що заміняє, чорний колір, і дерево при цьому залишається червоно-чорним.

Якщо вузол, що видаляється, є листком

Припустімо, що вузол, що видаляється, є листком. Якщо цей листок знаходиться в корені або має червоний колір (red), нічого робити не потрібно. У протилежному випадку приймемо, що **x** – вузол **NIL** (вказівник на нього), який заміняє чорний (black) лист, а **w** – братній вузол (вказівник на вузол) вузла **x**. Наприклад, необхідно видалити елемент 50 з червоно-чорного дерева, зображеного на рис. 14.18:

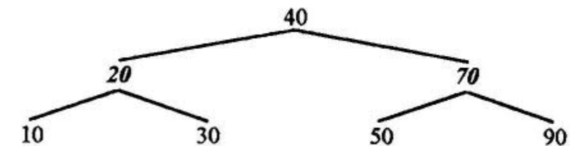


Рис. 14.18. Приклад червоно-чорного дерева

Тоді 50 заміняється вузлом **NIL** (рис. 14.19):

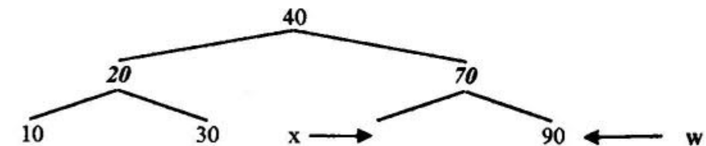


Рис. 14.19. Червоно-чорне дерево після видалення елемента 50

Відповідно до Правила червоного, оскільки вузол, що видаляється, не є **NIL**, **w** не може бути **NIL** (вказівником на **NIL**). Продовжуємо цикл переглядом чотирьох випадків, поки вузол **x**, який початково був чорним, не стане коренем або не стане червоного кольору.

Якщо вузол, що видаляється, має два дочірні елементи

Розглянемо ситуацію коли вузол, що видаляється, має два дочірні елементи. Якщо вузол, що видаляється, є червоним, а вузол, що заміняє його, також є червоним, то дерево залишається червоно-чорним. Наприклад, якщо елемент 80 видаляється із червоно-чорного дерева, зображеного на рис. 14.20, ми отримаємо червоно-чорне дерево, зображене на рис. 14.21.

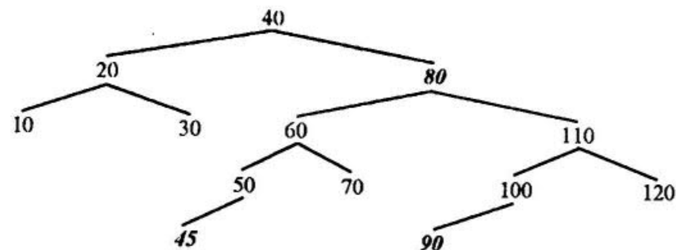


Рис. 14.20. Червоно-чорне дерево, з якого видалено елемент 80

Якщо вузол, що видаляється, був чорним, а вузол, що заміняє його, є червоним, то змінюємо колір вузла, що заміняє, на чорний, і дерево залишається червоно-чорним деревом. Наприклад, якщо видаляється елемент 40 з червоно-чорного дерева, зображеного на рис. 14.21, то отримаємо червоно-черне дерево, зображене на рис. 14.22.

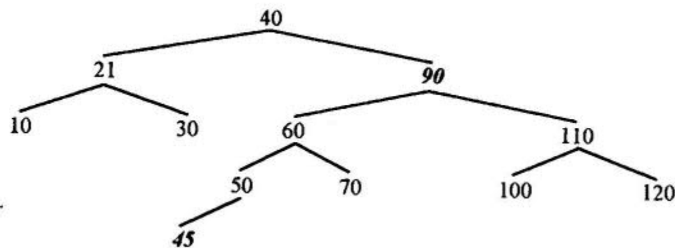


Рис. 14.21. Червоно-чорне дерево, зображене на рис. 14.20, після видалення елемента 80

Припустімо, вузол, що видаляється, має два дочірні елементи, а вузол, що заміняє його, є чорним. Починаємо з того, що призначаємо вузлу, що заміняє, колір вузла, що видаляється. Якщо вузол, що заміняє, має непустий правий дочірній елемент, то цей дочірній елемент повинен бути червоним, оскільки, вузол, що заміняє, не може мати лівий дочірній елемент. При видаленні правий дочірній елемент елемента, що заміняє, заміняє свого батька. Далі призначаємо цьому правому дочірньому елементу чорний колір і одержуємо червоно-чорне дерево (рис. 14.22).

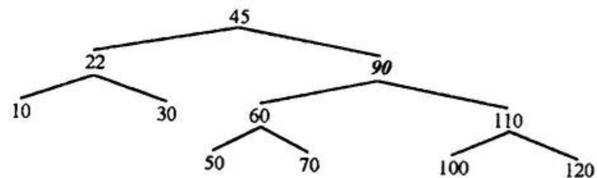


Рис. 14.22. Червоно-чорне дерево, зображене на рис. 14.21, після видалення елемента 40

Припустімо, вузол, що заміняє, має порожній правий дочірній елемент. Нехай **x** – вузол **NIL** (вказівник на вузол), який заміщує вузол, що заміняє. Нехай **w** – братній вузол (вказівник на вузол) вузла **x**, тобто **w** є братнім вузлом вузла, що заміняє. Наприклад, після видалення елемента 40 з червоно-чорного дерева, зображеного на рис. 14.23, отримаємо дерево, зображене на рис. 14.24. Оскільки вузол, що заміняє, був чорним, згідно із правилом шляху **w** не може бути **NIL**.

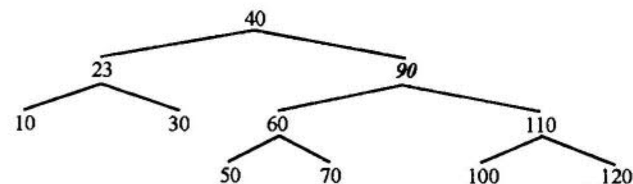


Рис. 14.23. Червоно-чорне дерево, з якого видаляється елемент 40, а елемент 50 заміняє його

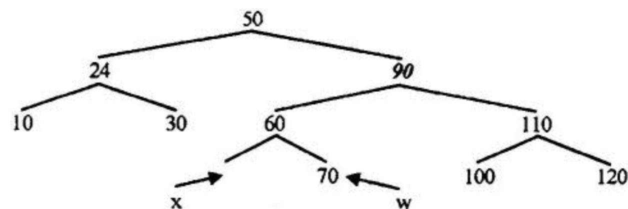


Рис. 14.24. Дерево, зображене на рис. 14.23, після видалення елемента 40

Якщо вузол **x** червоний, то потрібно зробити його чорним, і обидва Правила будуть дотримані. Аналогічно, якщо **x** є коренем, робимо його чорним. Єдина ситуація, у якій потрібні додаткові дії, це коли **x** є чорним некореневим вузлом. Тут цикл продовжується доти, поки **x** не стане червоним, або поки **x** не стане кореневим вузлом.

Якщо вузол, що видаляється, є листком або має два дочірні елементи

Якщо вузол, що видаляється, є листком, **x** являє собою вузол **NIL** (вказівник на нього), який заміняє вузол, що видаляється. Якщо вузол, що видаляється, має два дочірні елементи, **x** є правий дочірній елемент (вказівник на нього) вузла, що заміняє. В обох ситуаціях **w** являє собою братній вузол вузла **x** після того, як потрібний вузол видалено. Продовжуємо виконувати цикл, поки **x** не стане червоним або поки **x** не стане коренем. Для ситуації, коли **x** є лівим дочірнім

вузлом, є чотири можливі випадки; чотири симетричні випадки («лівий» і «правий» міняються місцями) виникають, коли **x** є правим дочірнім елементом. Як і для операції вставки, розглянемо випадки, у яких **x** є лівим дочірнім вузлом.

Усі наведені нижче випадки застосовуються, якщо вузол, що видаляється, є листком або має два дочірні вузли.

ВИПАДОК 1

color (w) = red

У цьому випадку виконуються такі дії:

```
color(w) = black;
color(parent(x) ) = red;
rotate_left (parent (x));
w = right (parent(x));
```

Припустімо, видаляється елемент 65 з такого дерева, рис. 14.25:

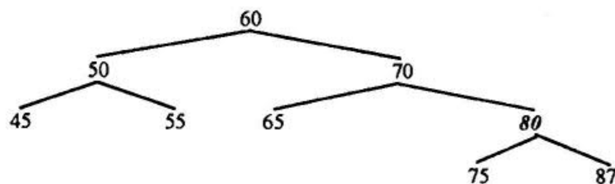


Рис. 14.25. Червоно-чорне дерево

Після видалення елемента 65 вузли **x** та **w** займуть такі положення (рис. 14.26):

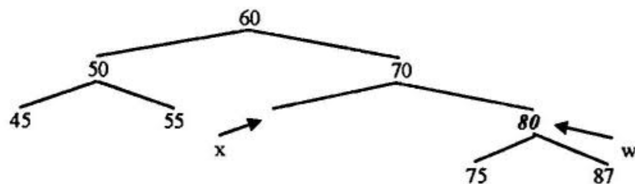


Рис. 14.26. Червоно-чорне дерево після видалення елемента 65

У цьому дереві вузол **x** є **NIL** (вказівник на **NIL). ** Оскільки вузол **w** червоний (red), застосуємо випадок **1**, коли **x** є лівим дочірнім елементом. Тому встановлюємо для **w** чорний колір (black), для батька **x** – червоний колір (red), здійснюємо лівий поворот навколо батька **x** і встановлюємо як **w** правий дочірній елемент батька **x**. Правило шляху поки ще не дотримано, але новий вузол **w** є чорним (рис. 14.27).

Отже, випадок **1** зводиться до одного із трьох інших випадків.

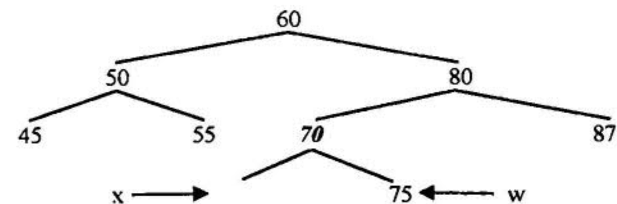


Рис. 14.27. Червоно-чорне дерево з чорним вузлом **w**

ВИПАДОК 2

Дочірні вузли вузла w є чорними

Нагадаємо, що **w** не може бути **NIL**. Один або обидва дочірні вузли вузла **w** можуть бути **NIL**, але кольором вузла **NIL** є чорний (black), тому не потрібно передбачати особливий випадок для ситуації, коли дочірні вузли вузла **w** є **NIL**. Дії для випадку **2** – незалежно від того, чи є **x** правим або лівим дочірнім вузлом – будуть такими:

```
color(w) = red;
x = parent(x) ;
```

Наприклад, можна застосувати випадок **2** до двійкового дерева пошуку на рис. 14.27. Оскільки обидва дочірні вузли вузла **w** є чорними (вузли **NIL** є чорними), то отримаємо дерево на рис. 14.28:

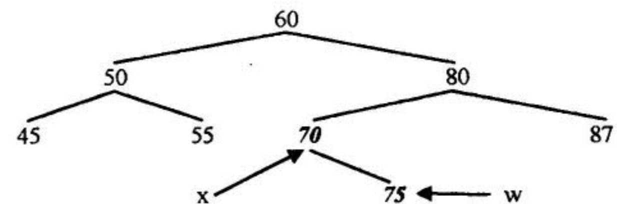


Рис. 14.28. Червоно-чорне дерево після дій випадку 2

Тепер вузол **x** є червоним, тому проходимо через цикл, встановлюємо для вузла **x** чорний колір і в підсумку одержуємо наступне червоно-чорне дерево (рис. 14.29):

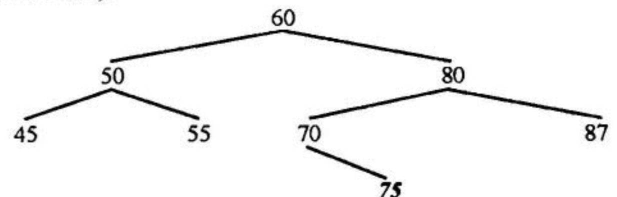


Рис. 14.29. Остаточний варіант червоно-чорного дерева після дій у випадку 2

Структура циклу в складі методу **erase** така:

```
while (x не є коренем і має чорний колір)
{
      if (x == лівий вузол батька x)
      {
      w = правий вузол батька x;
      if (вузол w має червоний колір)
{ . . .  Випадок 1  . . .  }
if (обоє дочірніх вузлів вузла w є чорними)
      { . . .  Випадок 2  . . . }
else
{ . . .  Випадки 3 і 4 . . . }
else // x є правим дочірнім елементом
{      ....      }
}
color (x) = black;
```

Цикл завжди завершується або після випадку **3**, або після випадку **4**. Цикл завжди завершується після однієї ітерації, якщо є випадок **1**, і після обробки випадку **1** немає випадку **2**. Випадок **2** – єдиний, у якому допускаються додаткові ітерації циклу, і коли це виникає, **x** стає ближчим до кореня, тому цикл **while** виконуватиметься максимум O(log *n*) разів.

ВИПАДОК 3

Правий дочірній вузол вузла w є чорним

З наведеної вище структури методу **erase** видно: якщо виник цей випадок, обидва дочірні вузли вузла **w** не можуть бути чорними. Тобто лівий дочірній вузол вузла **w** повинен бути червоним. У цьому випадку виконують такі дії:

```
color (left(w)) = black;
color(w) = red;
rotate_right(w);
w = right (parent (x));
```

Припустімо, видаляється елемент 45 з червоно-чорного дерева (рис. 14.30):

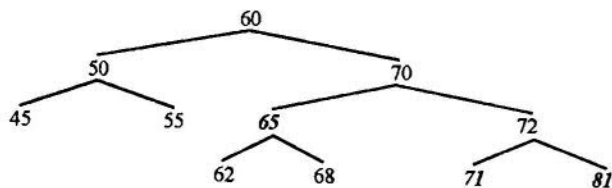


Рис. 14.30. Червоно-чорне дерево, видаляється елемент 45

На початку циклу **while** дерево має вигляд, як на рис. 14.31:

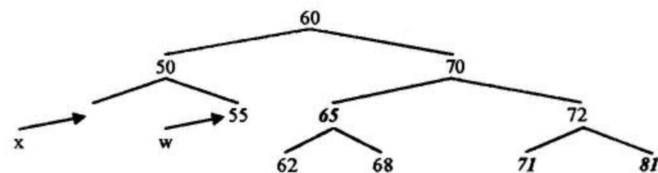


Рис. 14.31. Червоно-чорне дерево на початку циклу

Оскільки вузол **w** чорний і обидва його дочірні вузли також чорні, застосовується випадок **2**. Після застосування випадку **2** у наступній ітерації циклу одержуємо дерево, як на рис. 14.32:

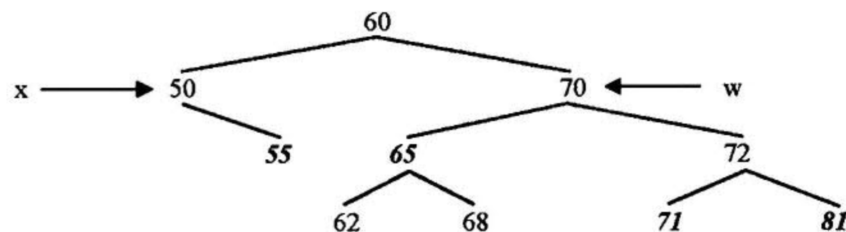


Рис. 14.32. Червоно-чорне дерево після застосування випадку 2

Випадок **3** застосовується, оскільки вузол **w** є чорним, його лівий дочірній вузол є червоним, а його правий дочірній вузол є чорним. Після зміни кольору, правого повороту й переустановки **w** одержуємо дерево на рис. 14.33:

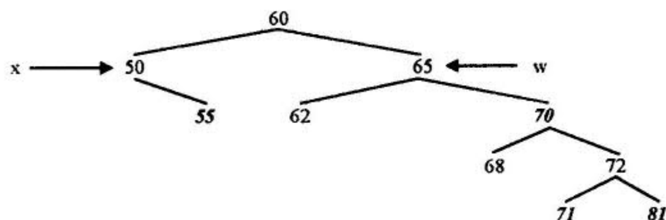


Рис. 14.33. Червоно-чорне дерево після застосування випадку 3

ВИПАДОК 4

Правий дочірній вузол вузла w є червоним

У цьому випадку виконують такі дії:

```
color(w) = color(parent (x));
color(parent(x)) = black; color(right(w)) =
black; left_rotate (parent (x));
break;
```

Наприклад, починаємо з наступного червоно-чорного дерева (рис. 14.34):

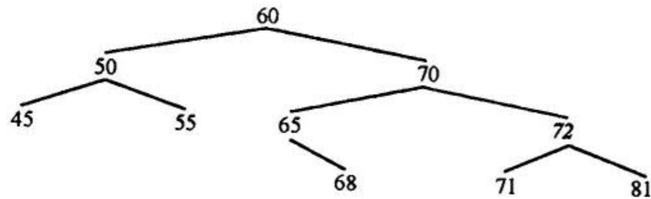


Рис. 14.34. Червоно-чорне дерево для ілюстрацій випадку 4

Після видалення елемента 50 одержуємо дерево (рис. 14.35):

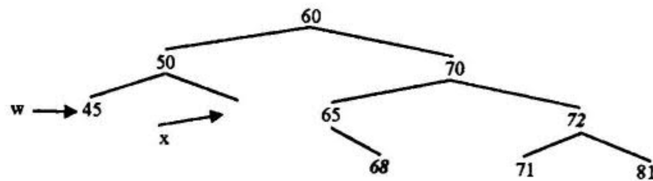


Рис. 14.35. Червоно-чорне дерево після видалення елемента 50

Застосовується випадок **2** і здійснюються однакові дії незалежно від того, чи є **x** правим дочірнім вузлом, або лівим дочірнім вузлом. Встановлюється для вузла **w** червоний колір (color(w) = red), а як вузол **x** встановлюється його батько (parent(x)). Під час наступної ітерації циклу отримаємо дерево (рис. 14.36):

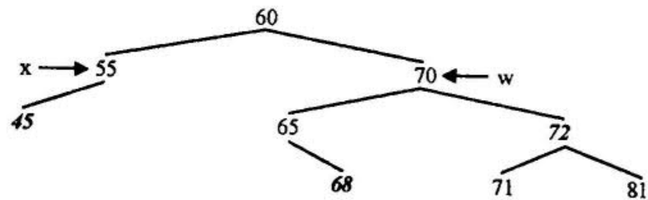


Рис. 14.36. Червоно-чорне дерево під час проходження циклу

Тепер виникає випадок **4**, тому змінюємо колір і здійснюємо лівий поворот навколо батька вузла **x** (рис. 14.37):

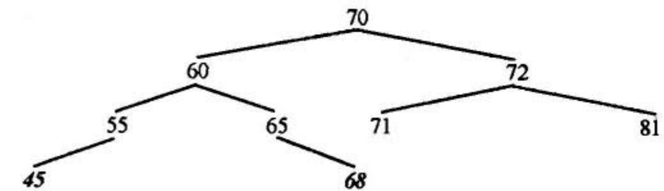


Рис. 14.37. Червоно-чорне дерево після зміни кольору і лівого повороту

Це дерево задовольняє й Правило червоного, і Правило шляху, тому можна вважати, що мета досягнута.

Як приклад розглянемо ситуацію, коли випадок **3** передує випадку **4**. Припустимо, починаємо з дерева на рис. 14.33, яке одержали наприкінці прикладу для випадку **3**. Правий дочірній вузол вузла **w** є червоним, тому застосуємо випадок **4**. Після зміни кольору та лівого повороту навколо батька **x** отримаємо дерево на рис. 14.38:

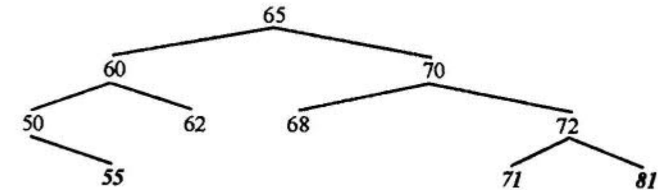


Рис. 14.38. Червоно-чорне дерево після зміни кольору і лівого повороту

Правила червоного і Правила шляху тепер дотримано.

Якщо застосуємо випадок **3**, то правий дочірній вузол вузла **w** стає червоним, тому випадок **4** завжди буде застосовано після випадку **3**. Замість того, щоб повторювати цей самий фрагмент коду для випадку **4**, розміщуємо його після коду для випадку **3**. Структура для випадків **3** і **4**, коли **x** є лівим дочірнім вузлом, має вигляд:

```
if (правий дочірній вузол вузла w є чорним)
{
... Випадок 3 ...
   }
... Випадок 4 ...
```

На початку цього розділу зазначалося, що час виконання першої частини методу **erase** логарифмічно залежить від *n* у найгіршому випадку. Що стосується циклу **while**, який складає другу частину методу **erase**, то цикл триває тільки тоді, коли можна застосувати випадок **2**, і при цьому вузол **x** переміщується ближче до кореня, на що витрачує максимум O(log *n)* ітерацій.

Нижче наведено закінчений цикл **while** із чотирма випадками для ситуації, коли **x** є лівим дочірнім вузлом, і чотирма симетричними випадками, коли **x** є правим дочірнім вузлом.

```
      while (x!= root() && color (x) == black)
if (x == left (parent(x)))
{
link_type w = right (parent(x));
if (color(w) = red) // Випадок 1
{
color(w) = black;
color(parent(x)) = red;
rotate_left(parent(x));
w = right (parent(x));
}
if (color (left(w)) = black &&
(color(right(w)) == black) // Випадок 2
{
color (w) = red;
x = parent(x); }
else
{
                    if (color(right(w)) == black
// Випадок 3
{
color(left(w)) = black;
color(w) = red;
rotate_right (w) ;
w = right (parent(x));
}
// Випадок 4
color(w) = color(parent(x));
color(parent(x)) = black;
color(right(w)) = black;
rotate_left(parent(x));
break;
}
}
      else
{
// симетрично до наведеного вище;
// "left" і "right" міняються місцями
link_type w = left(parent(x));
      if (color(w) = red) // Випадок 1
      {
```



```
      color(w) = black;
color (parent(x)) = red;
rotate_right(parent(x));
w = left(parent(x));
 }
      if (color(right(w)) = black &&
 (color(left(w)) == black) // Випадок 2
      {
    color(w) = red;
    x = parent(x);
 }
    else
      {
                    if (color(left(w)) == black //
Випадок 3
      {
    color (right(w)) = black;
    color(w) = red;
    rotate_left(w);
    w = left(parent(x));
 }
     // Випадок 4
     color(w) = color(parent(x));
     color(parent(x)) = black;
color (left(w)) = black;
rotate_right(parent(x));
break;
}
  }
  while color(x) = black;
```

Контрольні запитання та вправи

1. Дайте означення червоно-чорного дерева та наведіть приклад.
2. У чому полягає суть Правила червоного та Правила шляху?
3. Як залежить висота червоно-чорного дерева від n – кількості вершин?
4. Чи є червоно-чорне дерево збалансованим?
5. Дайте означення чорної висоти для червоно-чорного дерева.

6. Для формування яких асоціативних контейнерів використовується клас червоно-чорного дерева **rb_tree**?

7. Які дії передбачає вставка нового елемента в червоно-чорне дерево?

8. Які є варіанти видалення елемента з червоно-чорного дерева?

9. Продемонструйте ефект від вставки наступних елементів у пусте червоно-чорне дерево: 30, 40, 20, 90, 10, 50, 70, 60, 80.

10. Видаліть елементи 20 та 40 з червоно-чорного дерева, побудованого в попередній вправі.

11. Поясніть, чому неможливо побудувати червоно-чорне дерево розміром 20, яке не містить червоних елементів.

12. Нехай **v** – елемент червоно-чорного дерева та має один листок. Поясніть, чому елемент **v** повинен бути чорним, а його дочірній елемент червоним листком.

13. Побудуйте червоно-чорне дерево, яке не є AVL-деревом.

Приклади тестових питань

1. Клас червоно-чорного дерева використовується для формування контейнерів **rb_tree**:

а) вектор;

б) черга;

в) множина;

г) карта.

2. Якщо певний елемент у червоно-чорному дереві є червоного кольору, то його батьківський елемент має бути:

а) червоний:

б) чорний.

3. Висота червоно-чорного дерева має дорівнювати:

а) n;

б) $\log_2 n$;

в) n^2.

4. Який колір є визначним в Правилі шляху?

а) червоний;

б) чорний.

5. Правильним є одне з тверджень:

а) чорна висота $bh(y)$ дорівнює кількості чорних елементів у будь-якому шляху від y до будь-якого його нащадка, що не має дочірніх або має один дочірній елемент;

б) чорна висота $bh(y)$ дорівнює кількості червоних елементів у будь-якому шляху від y до будь-якого його нащадка, що не має дочірніх або має один дочірній елемент.

СПИСОК ЛІТЕРАТУРИ

1. Зиглер К. Методы проектирования программных систем: пер. с англ. – М.: Мир, 1985. – 328 с.

2. Кравець В.Л. Вступ до структур даних: навч. посіб. – К.: УМК ВО, 1989. – 88с.

3. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 2001. – 960с.

4. Вирт Н. Алгоритми и структуры даных.– СПб.: Невский діалект, 2001. – 352 с.

5. Кнут Д. Искусство программирования, Т. 3. Сортировка и поиск, 2-е изд.: пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 832 с.

6. Архангельский А.Я. Программирование в C++Builder 6. – М.: ЗАО «Издательство БИНОМ», 2002. – 1152 с.

7. Коллинз У.Дж. Структуры данных и стандартная библиотека шаблонов. – М.: ООО «Бином-Пресс», 2004. – 624 с.

8. Шаховська Н.Б., Голощук Р.О. Алгоритми та структури даних: навч. посіб. – Львів: Магнолія 2006, 2010. – 216 с.

9. Цветкова А.В. Информатика и информационные технологии: конспект лекций. – М.: Эксмо, 2007. – 192 с.

10. Борисенко В.В. Основы программирования [Електронний ресурс] – Режим доступу: http://www.intuit.ru/goods_store/books/32

11. Куликов А. Алгоритмы и структуры данных. Видеокурс. [Електронний ресурс] – Режим доступу: http://www.lektorium.tv/course/?id=22823

12. Романова Ю.Д. Информатика и информационные технологии: учеб. пособие. [Електронний ресурс] – Режим доступу: http://vuzer.info/load/tehnika/informatika_i_informacionnye_tekhnologii_konspekt_lekcij_romanova_ju_d_i_dr/27-1-0-12234

13. http://acm.mipt.ru/twiki/bin/view/Algorithms/WebHome

14. Jon Bentley Programming Pearls [Електронний ресурс] – Режим доступу: http://www.cs.bell-labs.com/cm/cs/pearls/

15. http://algolist.manual.ru/ds/

16. http://cs.mipt.ru/wiki/index.php/

ПРЕДМЕТНИЙ ПОКАЖЧИК

Алгоритм 73, 87, 88, 91, 92, 93, 102, 103, 107, 109, 110, 116, 118, 119, 120, 121, 122, 123, 124, 130, 131, 199, 200, 202, 221, 225
Алгоритми обходу дерева 73
Алгоритми пошуку
  послідовний 26, 30, 32, 116
  двійковий
  прямий пошук рядка
  Кнута, Моріса і Прата 121, 122
  Бойера – Мура 123
Алгоритми сортування
  простої вибірки 101, 102, 103, 107, 114, 115
  бульбашки 103, 104, 114, 115
  швидкий 84
  Шелла 99, 107, 108, 114
  Цифровий 114
  злиттям 110, 115
  підрахунком 112, 115
Алгоритм Хаффмана 225
Висота дерева 71, 85, 232, 248
Висота чорна 245
Вказівник 39, 141, 142, 143, 144
Дек 47
Дерево 71, 194, 196, 197, 199, 228, 238, 243, 260
  AVL 229, 233, 234, 241, 270
  бінарне 73, 74, 84, 193
  збалансоване 229, 241, 244
  позиційне
  Хаффмана 225, 226, 227, 228, 238, 240, 241
  червоно-чорне 243, 244, 245, 248, 252, 253, 254, 255, 258, 259, 263, 270
Дисципліна обслуговування
  LIFO 34, 48, 49
Дисципліна обслуговування
  FIFO 43, 44, 47, 48, 49, 187
Довжина шляху в дереві
Ітератор 164, 178, 181, 184, 212, 221
Клас 94, 156, 162, 164, 165, 166, 170, 176, 177, 178, 187, 189, 190, 191, 192, 193, 209, 210, 211, 212, 215, 216, 248, 249, 270
Ключ 28, 68, 69, 127, 128, 249
  первинний 84
Колізія 27, 131
Контейнер 164, 165, 212
Кортеж 54
Куча 218, 219, 220, 221, 222, 223
Масив 50, 51, 105, 145, 146, 172, 190
  одновимірний 51, 64
  багатовимірний 148, 150
Матриця 41
  Розріджена 50, 56, 57, 71
  суміжності 22, 75
Множина 21, 52, 53, 72, 139

Матриця
    Розріджена 51, 57, 58, 72
    суміжності 23, 76, 77
Множина 22, 53, 54, 73, 141
Петля 85
Поворот 234
Посилання 144, 147
Потужність множини 53
Правило червоного 257, 258, 271
Правило шляху 248, 266, 271
Рекурсія 107, 108
Решітка 84
Рівень вершини дерева 86
Розмір масиву 40, 59
Розмірність масиву 59
Сіткова структура 81, 82
Список 60, 67, 77, 78, 79, 184
    двонапрямлений 65, 79
    ієрархічний 78
    лінійний 71, 85, 119
    однонапрямлений 62
    циклічний 65
Спискова структура 77
Стандартна бібліотека шаблонів 168
Стек 34, 35, 36, 37, 39, 40, 41, 195
Степінь вершини дерева
Стоп-символ 126, 127
Структури 8, 9, 52, 134, 154, 157
    Самоадресовані 155
Таблиця 29, 69, 93, 105, 106, 108, 110, 111, 114, 115, 118, 119, 121, 123, 125, 127, 129, 130, 134, 135, 142, 143, 170, 182
Функція хешування 128
Часова складність алгоритму 90
Черга 44, 46, 175, 176, 177, 220, 231
    лінійна 34, 44, 48, 91, 93
    з пріоритетом 45, 222, 224, 231
    реверсна
    циклічна 72
Шаблон класу 163

Книги для навчання і роботи!

Мельник Р. А., Тушницький Р. Б.

ПРОГРАМУВАННЯ ІНТЕРНЕТ-ЗАСТОСУВАНЬ

Навчальний посібник. – 2013. – 256 с.
ISBN 978-617-607-491-5

Викладено матеріал для проектування web-сторінок. Розділи розкривають мови кодування HTML, HTML5, CSS та JQUERY, мови програмування JavaScript та PHP, технологію Ajax, зберігання даних у масивах та базах даних MySQL-сервера. Наведено приклади доступу до них з Web-сторінок. Подано основи проектування Web-сторінок у технологіях .NET та JAVA, зокрема за допомогою класів з бібліотек ASP.NET, класів побудови сервлетів javax.servlet та JSP засобів проектування динамічних web-сторінок. Рисунками проілюстровано схеми взаємодії програмних об'єктів Socket, ServerSocket, TcpListener, TcpClient тощо у мережі. Усі розділи завершуються контрольними запитаннями. Матеріал містить фрагменти програм, що можуть бути корисними студентам під час самостійного розроблення власних програм до лабораторних та курсових завдань.

Для студентів ВНЗ, технікумів та коледжів, які вивчають сучасні технології створення програмного забезпечення, зокрема за ефективною архітектурою "клієнт-сервер".

Федорчук Є. Н.

ПРОГРАМУВАННЯ СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ. ЕКСПЕРТНІ СИСТЕМИ

Навчальний посібник. – 2012. – 168 с.
ISBN 978-617-607-257-7

Для студентів вищих навчальних закладів, які навчаються за спеціальністю "Комп'ютерні науки" та "Програмна інженерія". Розглядаються основні моделі знань з чіткими та нечіткими даними, алгоритми пошуку розв'язку експертних задач, технології і засоби програмування баз знань та експертних систем. Подано приклади програмування експертних систем мовою Visual Basic у середовищі Excel.

Видавництво Львівської політехніки
вул. Ф. Колесси, 2, корп. 23 А, м. Львів, 79000
тел. +380 32 2582146, факс +380 32 2582136, http://vlp.com.ua, vmr@vlp.com.ua

НАВЧАЛЬНЕ ВИДАННЯ

Коротєєва Тетяна Олександрівна

АЛГОРИТМИ
ТА СТРУКТУРИ ДАНИХ

Редактор *Ольга Дорошенко*
Коректор *Олеся Пастущак*
Технічний редактор *Лілія Саламін*
Комп'ютерне верстання *Марти Гарасимів*
Художник дизайнер *Маріанна Рубель-Кадирова*

Здано у видавництво 15.07.2014. Підписано до друку 05.11.2014.
Формат 60×90 $^{1}/_{16}$. Папір офсетний. Друк офсетний.
Умовн. друк. арк. 17,25. Обл.-вид. арк. 14,80.
Наклад 150 прим. Зам. 140635.

Видавець і виготівник: Видавництво Львівської політехніки
Свідоцтво суб'єкта видавничої справи ДК № 4459 від 27.12.2012 р.

вул. Ф. Колесси, 2, Львів, 79013
тел. +380 32 2582146, факс +380 32 2582136
vlp.com.ua, ел. пошта: vmr@vlp.com.ua