МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Дніпродзержинський державний технічний університет

О. І. МИХАЛЬОВ, В. В. КРАМАРЕНКО
К. М. ЯЛОВА, К. Ю. НОВІКОВА

СТРУКТУРИ ДАНИХ ТА АЛГОРИТМИ

Навчальний посібник

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів
вищих навчальних закладів

Дніпродзержинськ
«ДДТУ»
2010

УДК 004.415.2.043 + 004.421

Структури даних та алгоритми / **Михальов О. І., Крамаренко В. В., Ялова К. М., Новікова К. Ю.** Навч. посібник. — Дніпродзержинськ: ДДТУ, 2010. — 286 с.

Навчальний посібник містить інформацію про сучасні структури даних та способи їх використання для забезпечення можливості розв'язання задач як обчислювального, так й інформаційного характеру. Навчальний посібник буде корисний для викладачів та студентів денної та заочної форми навчання спеціальностей 6.050103 «Програмна інженерія», 6.080403 «Програмне забезпечення автоматизованих систем», «Інформаційні управляючі системи та технології» та «Інформаційні технології проектування».

Рецензенти:

Оксанич А. П. *– доктор технічних наук, професор, ректор Кременчуцького університету економіки, інформаційних технологій і управління;*

Скалозуб В. В. *– доктор технічних наук, професор, завідувач кафедри КІТ Дніпропетровського національного університету залізничного транспорту ім. академіка Лазаряна;*

Стухляк П. Д. *– доктор технічних наук, професор, декан факультету «Компьютерних технологій», завідувач кафедри «Компьютерно-інтегрованих технологій» Тернопільского державного технічного університету ім. І. Пулюя.*

Рекомендовано Міністерством освіти і науки України як навчальний посібник (Лист № 1/11-6269 від 12.07.10).

Рекомендовано Вченою радою Дніпродзержинського державного технічного університету (протокол № 1 від 29.09.2009 р.).

ISBN 978-966-175-014-1

© Михальов О. І.,
Крамаренко В. В.,
Ялова К. М.,
Новікова К. Ю., 2010
© ДДТУ, 2010

ЗМІСТ

ВСТУП ………………………………………………………….. 7

Розділ 1. НАЙПРОСТІШІ СТАТИЧНІ СТРУКТУРИ: ВЕКТОРИ, МАСИВИ ЗАПИСІВ І ТАБЛИЦІ … 9

1.1. Вектор ……………………………………………... 9

1.2. Фізична структура вектора ……………………….. 10

1.3. Масиви, записи, таблиці ………………………….. 11

1.4. Багатовимірні масиви …………………………….. 22

1.5. Опрацьовування помилок, пов’язаних із неправильної індексацією ……………………………… 23

1.6. Структури й об’єднання в *С*++ ………………….. 26

Контрольні запитання та завдання для самоконтролю ……… 29

Розділ 2. НАПІВСТАТИЧНІ СТРУКТУРИ: СТЕКИ, ЧЕРГИ, ДЕКИ ………………………... 31

2.1. Список ……………………………………………... 31

2.2. Стек ………………………………………………... 32

2.3. Черга, дек ………………………………………….. 41

Контрольні запитання та завдання для самоконтролю ……… 53

Розділ 3. ЛІНІЙНІ ДИНАМІЧНІ СТРУКТУРИ: ОДНОЗВ’ЯЗНІ І ДВОЗВ’ЯЗНІ СПИСКИ ……. 56

3.1. Спискові структури ……………………………….. 56

3.1.1. Лінійний однозв’язний список …………….. 63

3.1.2. Лінійний двозв’язний список ……………… 67

3.1.3. Кільцеві структури ………………………….. 70

3.2. Операції над списками …………………………… 75

3.2.1. Формування списку ……………………….. 75

3.2.2. Вставка елемента до списку ………………... 75

3.2.3. Видалення елемента зі списку ……………... 80

3.2.4. Пошук елемента в списку ………………….. 81

3.2.5. Способи прискорення пошуку в спискових структурах …………………………………… 82

3.3. Приклад організації і представлення лінійних зв'язних списків у пам'яті комп'ютера ………….. 85
3.4. Ланцюговий спосіб організації збереження спискових структур ……………………………... 87
Контрольні запитання та завдання для самоконтролю ……… 92

Розділ 4. БІНАРНІ ДЕРЕВА, *B*-ДЕРЕВА ………………... 97
4.1. Пошук по дереву в зовнішній пам'яті …………... 97
4.2. *B*-дерева …………………………………………… 98
4.2.1. Визначення і пошук ………………………… 98
4.2.2. Вставка і видалення ключів, послідовний доступ ………………………………………… 106
4.3. Різновиди *B*-дерев. *B*+-дерево ……………………. 113
4.4. Зміни структури *B*-дерева ………………………... 115
4.5. Бінарне дерево ……………………………………….. 116
4.6. Застосування бінарних дерев …………………….. 121
4.7. Приклади застосування бінарних дерев в *Pascal* 127
Контрольні запитання та завдання для самоконтролю ……… 131

Розділ 5. РЕКУРСІЯ …………………………………………... 135
5.1. Поняття рекурсії й основні визначення ………… 135
5.2. Форми рекурсивних функцій …………………….. 140
5.3. Виконання дій на рекурсивному спуску ………… 147
5.4. Виконання дій на рекурсивному поверненні …… 148
5.5. Виконання дій як на рекурсивному спуску, так і на рекурсивному поверненні …………………….. 150
5.6. Приклади використання рекурсії в *Pascal* та *C*++ 152
Контрольні запитання та завдання для самоконтролю ……… 160

Розділ 6. ЛОГІЧНІ СТРУКТУРИ ТА СПОСОБИ ОПРАЦЮВАННЯ ФАЙЛІВ …………………… 163
6.1. Введення ……………………………………………... 163
6.2. Стислі характеристики логічних структур ……… 164
6.3. Метод доступу ……………………………………….. 168
6.4. Опрацювання файлів ……………………………... 169

6.4.1. Опрацювання файлів з послідовною організацією ……………………………….. 171
6.4.2. Опрацювання індексно-послідовних файлів 173
6.4.3. Опрацювання файлів із довільною організацією ……………………………… 175
6.5. Критерії оцінювання і вибору структур файлів і методів опрацювання ………………………….. 178
Контрольні запитання та завдання для самоконтролю ……… 179
Розділ 7. ПОСЛІДОВНІ ФАЙЛИ ……………………… 180
7.1. Структура послідовних файлів ………………….. 180
7.2. Пошук у послідовних файлах…………………….. 183
7.2.1. Послідовний або лінійний пошук …………. 184
7.2.2. Двійковий пошук …………………………… 185
7.3. Основні операції над послідовними файлами ….. 190
7.3.1. Додавання запису …………………………… 190
7.3.2. Видалення запису …………………………… 191
7.3.3. Відновлення запису ………………………… 193
7.4. Реорганізація послідовних файлів ………………. 193
7.5. Сортування ……………………………………….. 194
7.6. Внутрішнє сортування …………………………… 197
7.6.1. Сортування простими вставками ………….. 200
7.6.2. Сортування простим вибором ……………... 204
7.6.3. Просте обмінне сортування («пухирцеве» сортування) ………………………………….. 208
7.7. Характеристики продуктивності елементарних методів сортуваня ……………………………… 212
7.8. Зовнішнє сортування ……………………………... 216
7.8.1. Сортування злиттям ………………………… 218
7.9. Приклади програм створення та модифікації файлу в *Pascal* та *C++* …………………………….. 223
Контрольні запитання та завдання для самоконтролю …….. 237

Розділ 8. ІНДЕКСНО-ПОСЛІДОВНА ОРГАНІЗАЦІЯ ……………………………….. 240
8.1. Основні поняття. Визначення використовуваних термінів ……………………………………… 240
8.2. Індексно-послідовні файли. Їх структура ……… 242
8.3. Поняття індексу. Одно- та багаторівневі індекси 244
8.4. Основні операції над індексно-послідовними файлами …………………………………………….. 248
8.5. Анкерні крапки ………………………………… 252
8.6. Коефіцієнт розширення індексу ………………… 253
8.7. Доступ до області переповнювання за допомогою індексу ……………………………………... 254
8.8. Реорганізація індексно-полідовних файлів та області використання …………………………….. 257
Контрольні запитання і завдання для самоперевірки ……….. 258
Розділ 9. ФАЙЛИ ПРЯМОГО ДОСТУПУ ……………… 260
9.1. Введення ……………………………………………... 260
9.2. Способи адресації блоків ………………………… 261
9.3. Основні операції над файлами прямого доступу 266
9.4. Хешування ідентифікатора ……………………… 267
9.5. Функція хешування ……………………………… 271
9.6. Методи опрацювання переповнювання ………… 275
9.6.1. Метод відкритої адресації ………………….. 275
9.6.2. Метод нелінійного пошуку ………………… 276
9.6.3. Метод роздільних ланцюжків ……………… 276
Контрольні запитання і завдання для самоперевірки ……….. 278
ЛІТЕРАТУРА ………………………………………………. 280
ПРЕДМЕТНИЙ ПОКАЖЧИК ……………………………… 282

ВСТУП

Оскільки посібник містить реалізації корисних алгоритмів і докладну інформацію про характеристики продуктивності цих алгоритмів, він може стати у нагоді і тим, хто займається самоосвітою і тим, кого цікавить розробка комп'ютерних систем та програмних додатків.

До складу посібника входять контрольні завдання і вправи, рисунки і приклади реалізації описаних алгоритмів із застосуванням мов програмування високого рівня, що дає змогу більш ефективного навчання і забезпечує можливість самооцінки отриманих знань.

Приведені лістинги програм покликані познайомити читачів з основними властивостями максимально широкого кола основних алгоритмів. Описані алгоритми знаходять широке застосування і є важливими як для професійних програмістів, так і для тих, хто вивчає комп'ютерні науки.

Посібник складається з дев'яти розділів. Перший та другий розділи посібника охоплюють найпростіші статичні структури: вектори, масиви та напівстатичні структури: стеки, черги, деки.

У розділах 3 та 4 розглядаються лінійні динамічні структури: однозв'язні і двозв'язні списки та бінарні і *B*-дерева.

У розділі 5 подається детальна інформація про рекурсію, форми рекурсивних функцій та приклади їх використання на практиці.

Розділи 6—9 присвячені файловим структурам: послідовним, індексно-послідовним та файлам прямого доступу. При цьому подано їх порівняльну характеристику та сфери максимально ефективного застосування.

По кожному розділу приведені контрольні запитання та завдання для самоперевірки.

У навчальному посібнику приводиться інформація про інструментальні засоби, щоб читачі могли усвідомлено реалізовувати, налагоджувати і надбудовувати алгоритми для рішення конкретних задач або для забезпечення заданих функціональних можливостей програмних додатків. Оскільки приклади реалізації алгоритмів наведені з використання реальних мов програмування, а не псевдокодом, програми можна швидко задіяти в практичних цілях, що дасть змогу отримати не тільки теоретичні знання, а і практичні навички і уміння. Для реалізацій наведених алгоритмів використовуються мови програмування високого рівня *C++* та *Pascal*, знання яких є базовими для вивчення *Delphy* та *C#*.

Вичерпна інформація про сучасні використовувані структури даних дає можливість набути знання, що стають незамінним фундаментом для вивчення і розробки баз даних і проектування автоматизованих інформаційних систем.

Необхідно зауважити, що матеріали посібника внесені до бази даних «Автоматизованої системи навчання та контролю знань», розробленою авторами, і можуть бути доступними як в локальній мережі, так і в мережі *Internet.*

Розділ 1. НАЙПРОСТІШІ СТАТИЧНІ СТРУКТУРИ: ВЕКТОРИ, МАСИВИ ЗАПИСІВ І ТАБЛИЦІ

1.1. Вектор

Вектор (одномірний масив) — скінчена упорядкована множина простих даних одного типу, які називаються елементами вектора. Вектор характеризується властивостями [1]:

- сталість структури протягом всього часу існування;
- суміжність елементів;
- безперервність області пам'яті, відведеної відразу для всіх елементів структури.

Упорядковуючи елементи векторів, можна пронумерувати їх послідовними цілими числами, які називаються індексами елементів.

Кожному елементу вектора ставиться відповідно визначені значення індексу, що дає можливість ідентифікувати відповідний елемент. Під час опису логічної структури вектора, зазвичай, йому присвоюється ім'я (ідентифікатор), указується мінімальне й максимальне значення індексу і, можливо, розмір пам'яті для кожного з елементів. Для оголошення вектора в мові програмування *Pascal*:

Vec: Array [1..9] of Byte;

На логічному рівні доступ до елемента вектора здійснюється шляхом його ідентифікації, для чого достатньо зазначити ім'я вектора і значення індексу відповідного елемента. Наприклад:

Vec[3];

Над вектором також можуть виконуватися операції:

- визначення нижньої і верхньої меж індексу по заданому імені вектора;
- одержання опису елемента вектора (наприклад, тип і довжину).

1.2. Фізична структура вектора

Фізична структура вектора представляє в машинній пам’яті послідовність однакових по довжині ділянок пам’яті, названих полями (слотами), кожна з яких призначена для збереження одного елемента вектора. Слот може мати розмір мінімальної величини, що адресується чарунці пам’яті або відповідати цілій групі послідовних чарунок пам’яті. Якщо слот складається з декількох чарунок пам'яті, то його адресою, зазвичай, вважають адресу самої лівої чарунки. Елементи вектора розташовуються в пам’яті в порядку зростання адрес відповідних їм слотів, хоча можливий і зворотній порядок.

Нерідко відповідно фізичній структурі ставиться дескриптор або заголовок, що містить загальні відомості про фізичну структуру. Дескриптор необхідний, наприклад, у тому випадку, коли граничні значення індексів елементів масиву невідомі на етапі компіляції, і, отже, виділення пам’яті для масиву може бути виконано тільки на етапі виконання програми. Дескриптор зберігається, як і сама фізична структура, у машинній пам’яті, він представляє собою структуру даних, названу записом, що буде докладно розглянута нижче. Поки відзначимо лише, що дескриптор складається з полів, характер, кількість і розміри яких залежать від тієї структури, відповідно до якої поставлено дескриптор. Відносно вектора дескриптор може містити такі поля, як ім’я вектора, адреса його першого елемента в пам’яті комп’ютера (точніше, адреса першого слота), нижня і верхня межа індексу, тип елемента і розмір слота. Крім того, дескриптор, зазвичай, містить спеціальне кодове поле, що визначає ту структуру даних, якою асоційований дескриптор.

1.3. Масиви, записи, таблиці

У логічній структурі доступ до будь-якого елемента вектора здійснюється за допомогою імені вектора й індексу необхідного елемента. На рівні фізичної структури ім'я вектора та індекс елемента перетворюється на адресу відповідного елемента.

Масив — це послідовна упорядкована сукупність елементів деякого типу, що адресуються за допомогою деякого індексу [2]. Масив є прикладом композитної структури. Це означає, що він створений з більш простих, вже існуючих у мові, типів даних. Вивчення композитної структури припускає аналіз того, яким способом відбуваються організація такої структури з більш простих структур, а також того, яким чином з композитної структури відбувається доступ до якогось компонента. Найменший індекс масиву називається нижньою межею масиву, а найбільший — верхньою межею масиву.

Масив характеризується такими властивостями:

- сталість структури протягом всього часу існування;
- суміжність елементів;
- безперервність області пам'яті, відведеної одразу для всіх елементів структури.

Масивом називається такий вектор, кожний елемент якого — вектор. У свою чергу елементи вектора, що є елементами масиву, можуть бути векторами. Якщо масив складається з векторів, елементами яких є прості дані, то логічна структура такого масиву може бути подана прямокутною матрицею, у котрій будь-який її елемент може бути однозначно ідентифікований вказівкою пари індексів (перший задає номер рядка, другий — номер стовпчика) на перетині яких розташовано елемент.

Найпростіша форма масиву — одновимірний масив. Абстрактно він може бути визначений як скінчений, упорядкований набір елементів. Під «скінченим» ми розуміємо наявність у масиві конкретної кількості елементів [3]. Ця кількість може бути великою або

маленькою, проте вона обов'язково повинна існувати. Під «упорядкованим» розуміється той факт, що всі елементи масиву — упорядковані таким чином, що є перший елемент, другий, третій. «Однорідний» означає, що всі елементи масиву належать до одного типу даних, наприклад: масив може містити цілі числа або символьні рядки, проте він не може одночасно містити і ті й інші.

Над одновимірним масивом можна виконувати дві базові операції. Першою з них є витягання із масиву деякого заданого елемента. Вихідними параметрами для виконання такої операції є сам масив і покажчик того, до якого з його елементів проводиться доступ. Цей покажчик подається у вигляді цілого числа, названого індексом. Друга операція – додавання елементів у масив.

Для оголошення одновимірного масиву використовується наступна форма запису в мові програмування *C*++[4]:

тип ім'я_ масиву [розмір];

Наприклад: Тут за допомогою елемента запису оголошується базовий тип масиву. Базовий тип визначає тип даних кожного елемента, що складає масив. Кількість елементів, які будуть зберігатися в масиві, визначається елементом розмір. Наприклад, при виконанні наведеної нижче інструкції оголошується масив, що складається з 10 елементів, з іменем *sample* [4].

int sample[10];

В *C*++ перший елемент масиву має нульовий індекс. Оскільки масив *sample* містить 10 елементів, його індекси змінюються від 0 до 9. Щоб одержати доступ до елемента масиву по індексу, досить указати потрібний номер елемента в квадратних дужках. Так, першим елементом масиву *sample* є *sample*[0], а останнім — *sample*[9].

Для оголошення одновимірного масиву в мові програмування *Pascal* використовується наступна форма [5]:

ім'я_ масиву Array [розмір] тип даних;

Наприклад:

Mas: Array[1..10] of integer;

Однією з важливих особливостей масиву в мові *Pascal* є те, що створений масив є статичним. Це означає, що його верхня межа (а отже, і його розмір) не може бути змінена. Спроби змінити розмір масиву призведуть до помилки. Отже, на протязі всього часу свого існування масив у мові *Pascal* містить фіксовану кількість елементів. Розмір масиву повинен бути встановлений до записування до нього будь-яких значень.

Наприклад наступна програма, створена за допомогою мови програмування *C++*, поміщає в масив *sample* числа від 0 до 9 [6].

```
#include <iostream>
using namespace std;
int main ()
{  int sample[10];          // Ця інструкція резервує область
                            // пам'яті для 10 елементів типа int.
  int t;
// Розміщаємо в масив значення
for(t=0; t<10; ++t) sample[t] = t;
//Зверніть увагу на те, як //індексується масив sample.
// Відображаємо вміст масиву.
  for(t=0; t<10; ++t)
       cout <<"Це елемент sample[" << t << "]: "<<
sample[t] << "\n";
   return 0;}
```

Елементи масиву в пам'яті розташовані послідовно один за одним. Гніздо з найменшою адресою ставиться до першого елемента масиву, а з найбільшим — до останнього. Масиви часто використовуються в програмуванні, оскільки дозволяють легко обробляти велику кількість зв'язаних змінних. Наприклад, у наступній програмі, створеної за допомогою мови програмування *C++*, створюється масив з десяти елементів, кожному елементу привласню-

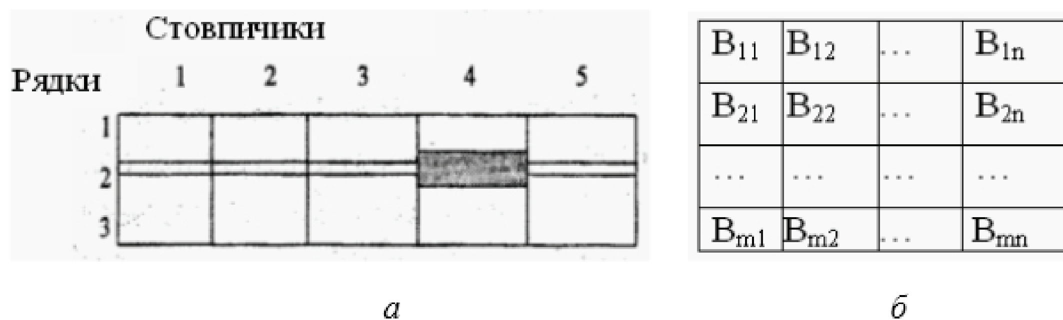
ється деяке число, а потім обчислюється й відображається середнє від цих значень, а також мінімальне й максимальне [6].

```
/ * Обчислення середнього і визначення мінімального і максимального з набору значень* /
#include <iostream>
using namespace std;
int main()
{
  int i, avg, min_val, max_val;
  int nums[10];
  nums[0] = 10;
  nums[1] = 18;
  nums[2] = 75;
  nums[3] = 0;
  nums[4] = 1;
  nums[5] = 56;
  nums[6] = 100;
  nums[7] = 12;
  nums[8] = –19;
  nums[9] = 88;
  nums[0] = 10;
  // Обчислення середнього значення.
  avg =0;
  for(i=0;  i<10;  i++)
  avg += nums[i];       // Підсумовуємо значення масиву nums.
  avg /= 10;               // Обчислюємо середнє значення
  cout << "Середнє значення дорівнює " << avg <<  '\n';
  // Визначення мінімального і максимального значення.
  min_val = max_val = nurns[0];
  for(i=l;  i<10;  i++)  {
   if(nums[i] < min_val)  min_val = nums[i];          //мінімум
   if (nums [i], > max_val) max_val – nums[i]          //максимум
  }
```

```
    cout << "Мінімальне значення: " << min_val << '\n';
    cout << "Максимальне значення: " <<max_val <<  '\n';
    return 0;
}
```

Зверніть увагу на те, як у програмі виконується доступ до елементів масиву *nums*. Той факт, що оброблювані тут значення збережені в масиві, значно спрощує процес визначення середнього, мінімального й максимального значень. Керуюча змінна циклу *for* використовується в якості індексу масиву при доступі до чергового його елемента. Подібні цикли дуже часто застосовуються при роботі з масивами.

У різних мовах програмування можна використовувати багатовимірні масиви. Двовимірний масив, по суті, є списком одновимірних масивів (рис. 1.1, *а*). Двовимірний масив є таким набором даних, у якому доступ до будь-якого з елементів здійснюється по двох індексах: номеру рядка й номеру стовпчика (рис. 1.1, *б*). Із цього випливає, що, якщо доступ до елементів масиву надати в порядку, у якім вони реально зберігаються в пам'яті, то правий індекс буде змінюватися швидше за лівий.



а *б*

Рис. 1.1. Приклад логічної структури двовимірного масиву А: *а* — фізичне розташування рядків і стовпчиків; *б* — індексування двовимірного масиву

Очевидно, формально можна визначити масив довільної кінцевої розмірності, у якому кожний елемент визначається кінцевою

упорядкованою послідовністю індексів, причому кількість таких індексів дорівнює розмірності масиву. Усі елементи масиву належать одному типу даних. Зверненням до заштрихованого елемента (рис. 1.1, *а*) є *A*[2,4], оскільки він розташований у рядку 2 і в стовпчику 4. Як і для випадку з одновимірним масивом, нижня межа для кожного виміру є по визначенню 1 або 0.

Кількість рядків або стовпчиків дорівнює значенню верхньої межі мінус значення нижньої межі плюс одиниця. Ця кількість називається розміром по даному виміру. У вищенаведеному масиві *A* цей розмір є 3–1+1 (вважаючи значення нижньої межі рівним одиниці), що дорівнює 3, а по іншому виміру є 5–1+1, що дорівнює 5. Таким чином, масив *A* має три рядки і п'ять стовпчиків. Кількість елементів у двовимірному масиві дорівнює добутку кількості рядків на кількість стовпчиків. Отже, масив А містить 3×5=15 елементів.

Двовимірний масив добре ілюструє розходження між фізичним і логічним представленням даних. Двовимірний масив є логічною структурою даних, що зручна для програмування й розв'язання задач. Наприклад: такий масив може виявитися корисним при описанні об'єкта, що є фізично двовимірним, наприклад карта або шахівниця. Він також корисний при організації набору значень, що залежать від двох параметрів.

Щоб оголосити двовимірний масив цілочислових значень розміром 10×20 з іменем *twod*, в мові програмування *C++* достатньо записати наступне [4]:

int twod[10] [20];

У мові програмуванні *Pascal* [7]:

twod: Array [1…10, 1…15] of integer;

У масивах перетворення логічної структури у фізичну, виконується шляхом процесу лінеаризації, у ході якої логічна структура масиву відображається в одномірну фізичну структуру, що представляє собою лінійно-впорядковану послідовність елементів

масиву. Наприклад: елементи двовимірного масиву *Y* можна лінійно упорядкувати у пам'яті так:

B11 , B12 ,..., B1n, B21, B22,..., B2n,..., Bm1 , Bm2 ,..., Bmn.

Наведемо ще один приклад: у програмі для торгової організації, у якій є 20 відділень і кожне займається продажем 30 різноманітних видів товарних одиниць, може бути використаний двовимірний масив вигляду (*Pascal*) [7]:

```
Var
Dim: array[1..20,1..30] of byte ;
```

Кожний елемент *Dim*(*i,j*) є кількістю товару типу *j*, проданою відділенням *i*.

Проте, хоча для програміста зручно розглядати елементи такого масиву, організовані у вигляді двовимірної таблиці, і мови програмування містять у собі засоби роботи з ними, проте, апаратна частина більшості комп'ютерів не підтримує такі можливості. Масив повинен зберігатися в пам'яті комп'ютера, а ця пам'ять, зазвичай, має лінійну організацію. Під лінійною організацією в даному випадку ми розуміємо, що пам'ять комп'ютера є одновимірним масивом. Для того, щоб витягти будь-який елемент, використовується одна адреса (яка може розглядатися як індекс одновимірного масиву). Для реалізації двовимірного масиву необхідно розробити метод розташування його елементів в одновимірному масиві і метод перетворення двокоординатних посилань до лінійних. Одним із способів представлення двовимірного масиву в пам'яті є відображення по стовпчиках. При такому представленні перший стовпчик масиву займає першу відведену під масив групу чарунок пам'яті, другий стовпчик займає другу групу і т. д. Декілька чарунок на початку масиву можуть бути відведені під заголовок, що містить верхні межі обох вимірів (не варто плутати цей заголовок із заголовками окремих елементів, що були об'явлені раніше масиву). Альтернативний спосіб припускає збереження заголовка окремо від масиву. При цьому він повинен містити адресу пер-

шого елемента масиву. Крім того, якщо елементи двовимірного масиву є об'єктами змінної довжини, то елементи суцільної області можуть містити адреси цих об'єктів.

У наступному прикладі у двовимірний масив розміщують послідовність чисел від 1 до 12 з використанням мови програмування *C++* [8].

```
#include <iostream>
using namespace std;
int main{)
{
int t,i,  nums[3][4];
for(t=0;  t < 3; ++t)   {
for (i=0;  i < 4;  + + i)   {
 nums[t][i] = (t*4)+i+l;           //Для  індексації  масиву  nums
необхідно два індекси
 cout << nums[t][i]  << ' ';
   }
   cout <<  '\n';
  }
  return 0;
}
```

У цьому прикладі елемент *nums*[0][0] одержить значення 1, елемент *nums*[0][1] — значення 2, елемент *nums*[0][2] — значення 3 і т.д. Значення елемента *nums*[2][3] буде дорівнювати числу 12.

Необхідно пам'ятати, що місце зберігання для всіх елементів масиву визначається під час компіляції. Крім того, пам'ять, виділена для зберігання масиву використовується протягом усього часу існування масиву. Для визначення кількості байтів пам'яті, займаної двовимірним масивом, використовуйте наступну формулу.

$$\text{кількість байтів} = \text{кількість рядків} \times \text{кількість стовпців} \times$$
$$\times \text{розмір типу в байтах}$$

Отже, двовимірний цілочисловий масив розмірністю 10×5 займає в пам'яті 10×5×4, тобто 200 байт (якщо цілочисловий тип має розмір 4 байт).

Запис — скінчена упорядкована множина елементів, що характеризується в загальному випадку різними типами даних. Елементи запису називаються полями. Запис — це таке загальне поняття вектора, при якому не потрібна однотипність або однорідність елементів. Доступ до будь-якого елементу запису здійснюється за допомогою імені, що повинно задаватися для кожного елемента на етапі опису логічної структури запису. Наприклад в мові програмування *Pascal* [5]:

```
Var
Student = Record
Num: Integer;
Name: String [15];
Fuc: String [3];
Group: String [5];
End;
```

У цьому описі слово *Record* означає тип логічної структури — запис, слово *Student* — загальне ім'я всієї структури, а *Num*, *Name, Fuc, Group* — імена полів запису. Це простий запис. Для простого запису характерно, що кожний її елемент є простим даним або ланцюжком простих даних одного типу.

Загальним виглядом запису є запис, у якого кожний його елемент може бути в свою чергу записом більш низького рівня і т. д. Така ієрархічна структура запису може вмістити довільну кількість рівнів, і тому її можна назвати багаторівневим записом. Під час опису багаторівневого запису прийнято виділяти рівні його елементів, причому передбачається, що найменування всього запису (і тільки воно) належить до рівня 1.

Припустимо, наприклад, що розрахункова відомість містить такі відомості про кожного співробітника деякої організації. Оде-

ржимо запис, логічна структура якого описується послідовністю пронумерованих ідентифікаторів:

1. Рядок відомості
 2. Табельний номер
 2. Ім'я
  3. Прізвище
  3. Ініціали
 2. Кваліфікація
  3. Посада
  3. Розряд
 2. Години
  3. Штатні
  3. Понаднормові
 2. Гроші
  3. Загальна сума
  3. Доплата
  3. Відрахування
  3. Видача

Тут цифри — це номера рівнів.

Незалежно від кількості рівнів у структурі запису корисна інформація зберігається лише в тих елементах, що відповідають листам (висячим вершинам дерева, що зберігають запис). Елементи запису з корисною інформацією називаються змістовними елементами.

Таблиця — скінчена кількість записів, що мають одну і ту ж організацію. Кожний запис, що входить у таблицю називається елементом таблиці. Зазвичай, елемент таблиці — це запис, якій складається з упорядкованої послідовності полів, що мають у загальному випадку різноманітний розмір і відповідно різноманітні типи простих даних. При цьому логічна структура таблиці, відображається у вигляді послідовності розташованих один під одним рядків однакової довжини, розділених на графи. Кожна графа відповідає визначеному полю елемента таблиці.

Зазвичай, одному з полів усіх елементів таблиці виділяється пам'ять для збереження ключа, що є унікальним для кожного елемента, використовуваного при доступі до відповідного елемента. Тому таблицю іноді визначають у вигляді списку пар (*K,V*), де *K* — поле ключа, а *V* — упорядкований набір полів призначених для збереження будь-якої іншої інформації. Приклад створення і доступу таблиці на *Pascal* [7]:

```
Type
 Element = Record
Num: Integer;
Name: String [15];
Fuc: String [3];
 Group: String [5];
End;
Var Table: Array [1..30] of Element;
```

Фізична структура таблиці представляє собою лінійну послідовність чарунок пам'яті, кількість яких визначається кількістю і розміром полів кожного елемента і кількістю елементів таблиці. Адресою таблиці вважається адреса слота, що відповідає першому полю першого елемента. Область пам'яті виділяється для таблиці в момент створення таблиці і надалі розмір цієї області, зазвичай, не змінюється.

Типові операції над таблицею — занесення нових елементів, пошук заданого елемента з метою перегляду його даних і видалення заданого елемента з таблиці. У тому випадку, коли таблиця невелика і частота звертання до неї мала, частіше усього застосовується послідовний перегляд таблиці з самого початку до елемента із заданим ключем або вільним елементом.

1.4. Багатовимірні масиви

Надамо ще одне формальне визначення масиву. *N*-вимірний масив, де *N*=1, 2, ... *n,* це — кінцева упорядкована множина (*n*–1)-вимірних масивів, усі елементи якої належать одному типу даних.

Мова *Pascal* та *C++* допускає роботу з масивами, розмірність, яких більше двох.

Наприклад: тривимірний масив у *Pascal* може бути оголошений у такий спосіб [5]:

```
VAR
    C: Array[1..3,1..2,1..4] of Byte;
```

Елемент у такому масиві адресується трьома індексами, *C*[*i,j,k*]. Перший індекс задає номер матриці, другий — номер рядка і третій — номер стовпчика. Такий масив корисний, якщо деяке значення визначається трьома параметрами. Наприклад: масив температур може бути проіндексований по широті, довжині, висоті.

Ось як оголошується багатовимірний масив в мові програмуванні *C++* [8]:

тип ім'я[розмір1] [розміру]...[розміри];

Наприклад, за допомогою наступного оголошення створюється тривимірний цілочисловий масив розміром 4×10×3.

```
int multidim[4][10][3];
```

З очевидних причин при виході за третій вимір геометрична аналогія неможлива. Проте, *Pascal* дозволяє задавати масиви з довільним числом розмірностей. Наприклад, 6-ти вимірний масив може бути оголошений у такий спосіб:

```
Var
D: Array [1..2,1..8,1..5,1..3,1..15,1..7] of Byte;
```

Для звернення до елемента такого масиву буде потрібно 6 індексів, наприклад: $D[a,b,c,d,e,f]$. Діапазон зміни індексу до заданої позиції індексу (розмір по якому-небудь виміру) дорівнює значенню верхньої межі для даного виміру мінус значення нижньої межі плюс одиниця. Кількість елементів у масиві є добутком розмірів по всіх вимірах. Наприклад: приведений раніше масив C містить $3\times2\times4=24$ елемента, а масив D містить $2\times8\times5\times3\times15\times7=25200$ елементів, нижня межа дорівнює 1.

Відображення масиву в пам'яті по стовпчиках може бути розширено і на масиви з розмірністю більше 2.

Масиви з кількістю вимірів, що перевищують три, використовуються нечасто, хоча б тому, що для їхнього зберігання потрібен великий обсяг пам'яті. Адже, як згадувалося вище, пам'ять, виділена для зберігання всіх елементів масиву, використовується протягом усього часу існування масиву. Наприклад, зберігання елементів чотиривимірного символьного масиву розміром $10\times6\times9\times4$ займе 2160 байт. А якщо кожну розмірність збільшити в 10 раз, то займана масивом пам'ять зросте до 21600000 байт. Як бачите, великі багатовимірні масиви здатні зайняти великий обсяг пам'яті, а програма, яка їх використовує, може дуже швидко зіштовхнутися із проблемою нестачі пам'яті.

1.5. Опрацьовування помилок, пов'язаних із неправильною індексацією

Припустимо, що програміст помилково зазначив індекс, що виходить за межі масиву. Треба сказати, що така ситуація виникає досить часто. Наприклад: програміст звертається до $A[i]$, де A є масивом з індексами, що змінюються в діапазоні від 1 до 100, а поточне значення i є 101. Такі помилки досить часті в тих випадках, коли в якості індексу використовується вираз, а також у середині циклу *For*, коли цикл повторюється на один раз більше

необхідного. Оскільки таке посилання є не дозволеним, результат такого звернення в мові *Pascal* не визначений [7].

Розглянемо декілька можливих дій, що можуть відбутися при виході індексу за межі масиву. Найпростішим випадком є відсутність будь-яких дій. Це означає, що при виникненні звертання до елемента масиву $A[i]$ машина продовжує обчислювати адресу елемента. Наприклад, якщо розмір кожного елементу масиву дорівнює одній чарунці пам'яті, а масив зазначено у межах 1 і 100, то посилання до елемента з індексом 101 призведе до отримання адреси, що стоїть уперед на 100 елементів від першого елемента масиву. Ця адреса знаходиться за межами самого масиву і може навіть знаходиться за межами пам'яті відведеної під програму. Система в цьому випадку може почати будь-яку підхожу дію. Якщо адреса лежить за межами пам'яті, відведеною програмою, то такою дією може бути видача повідомлення про помилку й припинення програми. Проте повідомлення про таку помилку не обов'язково має на увазі невірну індексацію. Воно може також означати спробу звернення до неіснуючої чарунки пам'яті або ж до пам'яті, не відведеної для даної програми.

Може трапитися так, що визначена адреса знаходиться в області пам'яті, відведеної програмі, проте інформація з цієї адреси не відповідає формату елемента, що адресується масиву. При спробі інтерпретувати цю інформацію як елемент масиву, система видає повідомлення про те, що інформація записана в невірному форматі. У цьому випадку також не буде ніяких відміток про те, що помилка відбулася внаслідок введення індексу, що є за межею масиву.

Одержання програмістом неточних повідомлень такого роду не виключає можливості виникнення й інших ситуацій. Набагато гірша ситуація може виникнути в тому випадку, якщо визначена адреса чарунки знаходиться у середині програмної області, а збережена в ній інформація знаходиться в необхідному форматі. У цьому випадку система скористається цією інформацією без ви-

дачі будь-яких повідомлень про помилку і на підставі цієї інформації буде видавати неправильні результати. В цьому випадку програміст не одержить ніяких повідомлень про те, що результати невірні. В інших випадках він може розуміти, що результат, очевидно, невірний, не знаючи при цьому, у якому місці великої програми відбулася помилка.

В усіх перерахованих вище випадках реалізація мови залежить від наявної системи виявлення помилок, що може бути реалізована апаратно або ж програмно, через операційну систему. У цьому випадку контроль за перебуванням індексу в середині меж масиву не провадиться. Оскільки індекс може бути змінною і виразом, його приналежність показаному діапазону неможливо визначити без явної перевірки такої відповідності. Така перевірка повинна проводитися кожного разу при виконанні оператора. Так, якщо оператор у циклі виконується 1000 разів, то це припускає проведення тисячі перевірок. Перевірка містить у собі не тільки визначення адреси, але і контроль на значимість. Це різко знижує ефективність роботи програми. Крім того, для перевірки значимості індексу необхідно постійно зберігати в пам'яті значення верхньої межі.

Є інший спосіб, при якому на перший план висуваються зручність роботи і легкість налагодження програми. Масив у цьому випадку подається не одними лише вхідними в нього елементами. Кожний масив містить також заголовок, у якому вказано межі масиву. Цей заголовок може бути розміщений на початку суцільної області, в якій знаходяться елементи масиву, або може розташовуватися у вигляді окремої одиниці і містити базову адресу масиву й значення його меж. При звертанні до елемента масиву в процесі роботи програми, проводиться перевірка на приналежність індексу припустимій області. Ця перевірка здійснюється до визначення адреси елемента. Якщо індекс лежить за межами припустимої області, то з’являється докладне повідомлення про помилку, що містить ім'я масиву і невірне значення індексу.

1.6. Структури й об'єднання в *C*++

В мові програмування *C*++ структури оголошуються за допомогою ключового слова *struct*. У мові *C* структура може містити тільки члени даних, але це обмеження в *C*++ не діє. В *C*++ структура, по суті, являє собою лише альтернативний спосіб визначення класу. В *C*++ в *struct* усі члени відкриті в структурі (*struct*) і закриті в класі (*class*) [6].

```
#include <iostream>
using namespace std;
struct Test {
  int get_i() { return i; }      // Ці члени відкриті
void put_i (int j) { i = j; } // (public) по замовченню.
private:
  int i;
};
int main ()
{
  Test s;
  s.put_i(10) ;
  cout << s.get_i();
  return 0;
}
```

Ця проста програма визначає тип структури *Test*, у якій функції *get_i*() і *put_i*() відкриті, а член даних *i* закритий. Зверніть увагу на використання ключового слова *private* для оголошення закритих елементів структури.

Об'єднання — це область пам'яті, що розділяється декількома різними змінними. Об'єднання створюється за допомогою ключового слова *union*. Його оголошення, як неважко переконатися на наступному прикладі, подібно оголошенню структури [6].

```
union utype {
  short int i;
  char ch;
} ;
```

Тут оголошується об'єднання, в якому значення типу *short int* і значення типу *char* розділяють ту саму область пам'яті. Необхідно відразу ж прояснити один момент: неможливо зробити так, щоб це об'єднання зберігало і цілочислове значення, і символ одночасно, оскільки змінні *i* та *ch* накладаються (у пам'яті) одна на одну. Але програма в будь-який момент може обробляти інформацію, що утримується в цьому об'єднанні, як цілочислове значення або як символ. Отже, об'єднання забезпечує два (або більше) способи представлення однієї і тієї ж порції даних.

Змінну об'єднання можна оголосити, розмістивши її ім'я наприкінці його оголошення або скориставшись окремою інструкцією оголошення. Щоб оголосити змінну об'єднання з ім'ям *u_var* типу *utype*, досить записати наступне:

```
utype u_var;
```

У змінній об'єднання *u_var* як змінна *i* типу *short int,* так і символьна змінна *ch* займають ту саму область пам'яті. (Безумовно, змінна *i* займає два байти, а символьна змінна *ch* використовує тільки один.) Як змінні *i* та *ch* розділяють одну область пам'яті, показано на рис. 1.2.

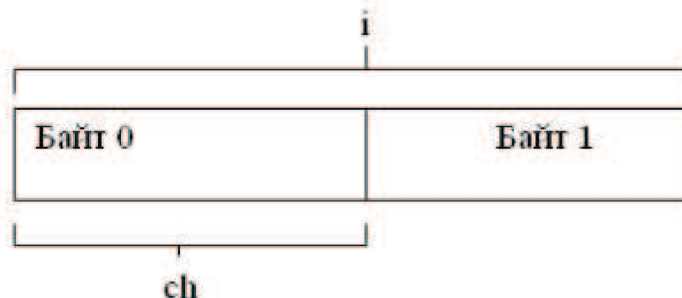


Рис. 1.2. Змінні *i* та *ch* разом використовують об'єднання *u_var*

Згідно «ідеології» *C++* об'єднання, по суті, є класом, у якому всі елементи зберігаються в одній області пам'яті. Тому об'єднання може включати конструктори і деструктори, а також функції-члени. Оскільки об'єднання успадковані від мови *C*, його члени відкриті (а не закриті) за замовчуванням.

Розглянемо програму, у якій об'єднання використовується для відображення символів, що складають молодший і старший байти короткого цілочислового значення (у припущенні, що «розмір» типу *short int* складає два байти) [6].

```
// Демонстрація  використання об'єднання.
#include <iostream>
using namespace std;
union u__type {
  u_type(short int a) { i = a; };
  u_type(char x, char y) {.ch[0] = x; ch[l] = y; }
  void showchars(){
    cout << ch[0] << " ";
    cout << ch[l] << "\n";
  }
  short int i;       // Ці члени даних об'єднання
  char ch[2j;        // поділяють одну й ту ж саму область
пам'яті
};
int main ()
{
  u_type u(1000);
  u_type u2('X', 'Y');
 // Дані в об'єкті типа u_type можна використовувати в
якості
  // цілочислового значення або двох символів
  cout << "Об'єднання u в якості цілого числа: ";
  cout << u.i << "\n";
  cout << " Об'єднання u в якості двох символів: ";
  u.showchars();
```

```
  cout << " Об'єднання u2 в якості цілого числа: ";
  cout << u2.i << "\n";
  cout << " Об'єднання u2 в якості двох символів: ";
  u2.showchars();
  return 0;
}
```

При використанні *C++*-об'єднань необхідно пам'ятати про деякі обмеження, пов'язані з їхнім застосуванням. Насамперед, об'єднання не може успадковувати клас і не може бути базовим класом. Об'єднання не може мати віртуальні функції-члени. Статичні змінні не можуть бути членами об'єднання. В об'єднанні не можна використовувати посилання. Членом об'єднання не може бути об'єкт, що перевантажує оператор привласнювання «=». Нарешті, членом об'єднання не може об'єкт, у якому явно визначено конструктор або деструктор.

Контрольні запитання та завдання для самоконтролю

1. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Упорядкувати масив по убуванню.
2. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Упорядкувати масив по зростанню.
3. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Знайти суму елементів масиву.
4. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Знайти добуток елементів масиву.
5. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Знайти найменший елемент масиву.
6. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Знайти найбільший елемент масиву.
7. Даний масив: 5, 17, 22, 4, 6, 7, 30, 11. Знайти всі парні елементи масиву.

8. Дано масив: 5, 17, 22, 4, 6, 7, 30, 11. Знайти всі непарні елементи масиву.

9. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Знайти всі парні елементи масиву й упорядкувати їх по зростанню.

10. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Знайти всі парні елементи масиву й упорядкувати їх по убуванню.

11. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Знайти всі непарні елементи масиву й упорядкувати їх по зростанню.

12. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Знайти всі непарні елементи масиву й упорядкувати їх по убуванню.

13. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Визначити парні і непарні елементи масиву. Упорядкувати їх по зростанню через один, починаючи з непарного (непарний / парний і т.д.).

14. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Визначити парні і непарні елементи масиву. Упорядкувати їх по зростанню через один, починаючи з парного (парний / непарний і т.д.).

15. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Визначити парні і непарні елементи масиву. Упорядкувати їх по убуванню через один, починаючи з непарного (непарний / парний і т.д.).

16. Дано масив: 13, 37, 5, 16, 29, 8, 4, 32, 69, 11, 2. Визначити парні і непарні елементи масиву. Упорядкувати їх по убуванню через один, починаючи з парного (парний / непарний і т.д.).

17. Дано масив: 3, 21, 17, 6, 10, 2, 21, 5. Знайти максимальний елемент масиву і визначити кількість входжень до масиву цього елемента.

18. Дано масив: 11, 24, 3, 10, 8, 3, 15, 3. Знайти мінімальний елемент масиву і визначити кількість входжень до масиву цього елемента.

19. Дано масив: 9, 8, 7, 13, 1, 2, 3, 4, 6, 5, 8, 9, 10. Знайти максимально довгу послідовність чисел, що йдуть підряд, задовольняючи умові: $AN=AN+1-1$.

20. Дано масив: 9, 8, 7, 13, 1, 2, 3, 4, 6, 5, 8, 9, 10. Знайти максимально довгу послідовність чисел, що йдуть підряд, задовольняючи умові: $AN=AN+1+1$.

Розділ 2. НАПІВСТАТИЧНІ СТРУКТУРИ: СТЕКИ, ЧЕРГИ, ДЕКИ

2.1. Список

Для початку введемо поняття «список». Лінійний список — це множина, що складається з n>0 вузлів (вузол — це група з n суміжних записів) $X[1]$, $X[2], ...,$ $X[n]$, структурні властивості якого по суті обмежуються лише лінійною (одномірною) залежністю вузлів, тобто такими умовам, що якщо n>0, то $X[1]$ є першим вузлом, якщо $1<k<n$, то k-му вузлу $X[k]$ передує $X[k–1]$ і за ним слідує $X[k+1]$, а $X[n]$ є останнім вузлом.

Визначений у такий спосіб список називають лінійним списком унаслідок лінійної упорядкованості його елементів. Упорядкованість елементів списку може задаватися неявно шляхом послідовного розташування його елементів як у логічній структурі, так і в пам'яті комп'ютера. З іншої сторони упорядкованість елементів може задаватися за допомогою спеціальних покажчиків, що розташовуються в елементах і дають можливість для кожного елемента визначити його попередника або послідовника. Такі списки називаються зв'язковими списками.

Операції, що можна виконувати з лінійними списками включають:

1. Одержати доступ до k-того вузла списку, щоб проаналізувати його або змінити вміст його поля.

2. Додати новий вузол безпосередньо перед k-м вузлом.

3. Видалити k-й вузол.

4. Об'єднати два (або більше) лінійних списків в один список.

5. Розділити лінійний список на два (або більше) списків.

6. Зробити копію лінійного списку.

7. Визначити кількість вузлів у списку.

8. Виконати сортування вузлів списку у зростаючому порядку по деяких полях у вузлах.

9. Знайти в списку вузли із заданим значенням у деякому полі.

Спеціальні випадки k=1 і k=n в операціях 1, 2 і 3 особливо виділяються оскільки в лінійному списку простіше одержати доступ до першого й останнього елементам, ніж до довільного.

Існує багато способів представлення лінійних списків, виходячи з цього, важко спроектувати єдиний метод представлення для лінійних списків, при якому всі перелічені операції виконуються ефективно. Наприклад, порівняно важко ефективно реалізувати доступ до k-того вузла в довгому списку для довільного k, якщо в той же час ми додаємо та видаляємо елементи в середині списку. Отже, ми будемо розрізняти типи лінійних списків по головних операціях, що з ними виконуються.

Дуже часто зустрічаються лінійні списки, у яких додавання, видалення, а також доступ до значень майже завжди виконуються в першому або в останньому вузлі, їм дано спеціальні назви: стек, черга та дек.

2.2. Стек

Операції вставки й витягання елементів зі звичайної послідовності адресні — вони використовують номер елемента (індекс). Якщо обмежити можливості зміни послідовності тільки її кінцями, одержимо структури даних, що називаються стеком і чергою.

Стек — лінійний список, у якому всі вставки й видалення (будь-який доступ) робляться в тільки в кінці списку (рис. 2.1) [9].

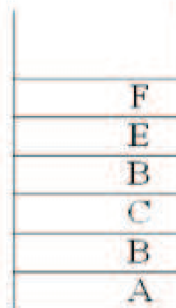


Рис. 2.1. Стек, що містить шість елементів

Іноді аналогія з переключенням залізничних колій, запропонована Е. Дейкстрой (рис. 2.2) допомагає зрозуміти механізм стека.

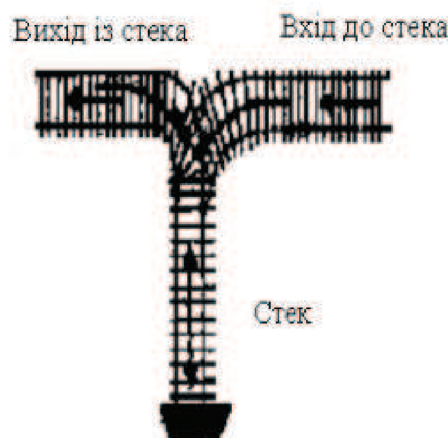


Рис. 2.2. Стек, наведений у вигляді залізничної колії

Зі стека ми завжди видаляємо «молодший» елемент із наявних у списку, тобто той, що був доданий пізніше інших.

Початок послідовності називається дном стека, кінець послідовності, до якого додаються елементи та з якого вони видаляються — вершиною стека. Операція додавання нового елемента (запис у стек) має загальноприйняту назву *Push* (занурити), операція видалення — *Pop*. Операції *Push* і *Pop* безадресні: для їхнього виконання ніякої додаткової інформації про місце розміщення елементів не потрібно.

Існують інші назви стеків, наприклад: стек називали пуш-даун (*push-down*) списком, реверсивною пам'яттю, гніздовою пам'яттю, магазином, списком типу *LIFO*. («*last-in-first-out*» — «останнім додається — першим видаляється»). Така розмаїтість назв цікава сама по собі, оскільки вона свідчить про важливість цих понять. Слово «стек» поступово стає стандартним терміном, із всіх інших словосполучень, перелічених вище лише «пуш-даун список» залишається ще досить поширеним, особливо в теорії автоматів.

Виняткова популярність стека в програмуванні пояснюється тим, що при заданій послідовності запису елементів у стек (наприклад, *А-В-С*) їх витягання відбувається в зворотному порядку (*С-В-А*). А саме ця послідовність дій відповідає таким поняттям, як вкладеність викликів функцій, вкладеність визначень конструкцій мови і т. д. Отже, скрізь, де мова йде про вкладеність процесів, структур, визначень, механізмом реалізації такої вкладеності є стек:

- при виклику функції, адреса повернення (адреса наступної за викликом команди) запам'ятовується в стеці, у такий спосіб створюється «історія» викликів функцій, яку можна відновити у зворотному порядку;

- при синтаксичному аналізі вкладених один до одного конструкцій мови транслятори використовують магазинні (стекові) автомати, стек при цьому містить не до кінця проаналізовані конструкції мови.

Інша важлива властивість стека — відносна адресація його елементів. Насправді для елемента, збереженого в стеці, важливо не його абсолютне положення в послідовності, а положення щодо вершини стека або його покажчика, яке відбиває «історію» його заповнення. Тому адресація елементів стека відбувається відносно до поточного значення покажчика стека, приклад роботи зі стеком за допомогою мови програмування *С++* [8]:

```
//Робота зі стеком по зміщенню:
int Get(int n){               // Отримати n елемент у стеку
return (Stack[SP–n]);}        // відносно покажчика стеку
```

Стеки дуже часто зустрічаються на практиці. Простим прикладом може служити ситуація, коли ми переглядаємо множину даних і складаємо опис станів або об'єктів, що повинні опрацьовуватися пізніше; коли початкова множина оброблена, ми повертаємося до цього списку і виконуємо наступне опрацювання, видаляючи елементи зі списку, поки список не стане порожнім. Для цієї цілі придатні як стек, так і черга, але стек, як правило, зручніше. При вирішенні задач наш мозок поводиться як «стек»: одна проблема призводить до іншої, а та у свою чергу до наступної; ми накопичуємо в стек ці задачі та підзадачі і видаляємо в міру того, як вони вирішуються. Аналогічно процес входів підпрограми й виходів із них при виконанні машиною програми подібний процесу функціонування стека. Стеки особливо корисні при опрацюванні мов, що мають структуру вкладень. До них відносяться мови програмування, арифметичні вирази і т.п. Взагалі, стеки частіше усього виникають в алгоритмах, що мають явно або неявно рекурсивний характер. Термін алгоритм використовується в комп'ютерних науках для опису методу рішення задачі, придатного для реалізації у вигляді комп'ютерної програми.

При описі алгоритмів, що використовують такі структури, прийнята спеціальна термінологія: новий елемент розміщується на верх стека, виймається теж верхній елемент (рис. 2.1). На дні стека знаходиться найменш доступний елемент і він не видалиться доти, поки не будуть видалені всі інші елементи. (Часто говорять, що елемент спускається (*push down*) в стек або, що стек піднімається (*pop up*), якщо видаляється верхній елемент. Стислість слів «опустити» і «підняти» має свою перевагу, але ці терміни помилково припускають прямування всього списку в пам'яті комп'ютера. Фізично нічого не спускається; елементи просто до-

даються поверх, як при стогуванні сіна або при укладці стосу коробок. У застосуванні до черг ми говоримо про наявність і кінець черги; об'єкти ставляться в кінець черги і віддаляються в момент, коли, нарешті досягають її початку.

Стек у мові програмування *C*++ зазвичай представляється масивом з додатковою змінною, яка вказує на останній елемент послідовності у вершині стека — покажчик стека (рис. 2.3).

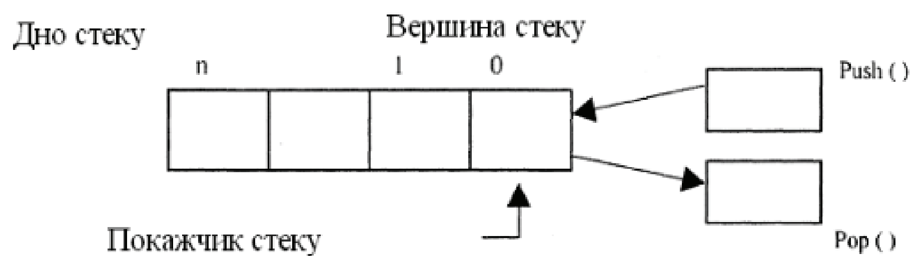


Рис. 2.3. Представлення стеку в масиві

У наступній програмі, виконаній за допомогою мови програмування *C*++, *N* елементів стека зберігаються як елементи масиву: *s*[0], ... , *s*[*N*–1], починаючи з першого занесеного елемента і завершуючи останнім. Верхівкою стека (позицією, в яку буде заноситися наступний елемент стека) є позиція *s*[*N*]. Максимальна кількість елементів, що може вміщати стек, програма-клієнт передає у вигляді аргументу в конструктор *STACK*, що розміщає в пам'яті масив даного розміру; однак код не перевіряє такі помилки, як переміщення елемента в переповнений стек (або виштовхування елемента з порожнього стека) [6].

```
template <class Item>
 class STACK
{ private:
Item *s; int N;
public:
STACK(int maxN)
```

```
{ s = new Item[maxN] ; N=0;}
 int empty() const
{ return N == 0; } void push (Item item)
{ s[N++] = item; } Item pop()
{ return s[– – N] ; }
```

Виконання операції додавання нового елементу означає запам'ятовування елемента в позиції масиву, що вказується індексом верхівки стека, а потім збільшення цього індексу на одиницю; виконання операції виштовхування елементу означає зменшення індексу на одиницю і видалення елемента, позначеного цим індексом. Операція створити (конструктор) здійснює розміщення масиву зазначеного розміру, а операція перевірити чи порожній стек перевіряє, чи не дорівнює індекс нулеві. Скомпільована разом із клієнтською програмою, ця реалізація забезпечує раціональний і ефективний стек. Один потенційний недолік застосування масиву для представлення стека відомий багатьом: як це зазвичай буває зі структурами даних, створюваними на базі масивів, до використання масиву необхідно знати його максимальний розмір, щоб розподілити під нього оперативну пам'ять. У розглянутій реалізації ця інформація передається в аргументі конструктора. Даний недолік — результат вибору реалізації на базі масиву; він не є невід'ємною частиною стеку.

Найчастіше важко визначити максимальну кількість елементів, яку програма буде заносити в стек: якщо вибрати занадто велике число, то така реалізація буде неефективно використовувати оперативну пам'ять, а це може бути небажано в тих додатках, де пам'ять є цінним ресурсом. Якщо вибрати занадто маленьке число, програма може взагалі не працювати. Застосування стеку дає можливість розглядати інші варіанти і змінювати реалізацію без зміни коду клієнтських програм. Наприклад, щоб стек міг елегантно збільшуватися і зменшуватися, можна було б віддати перевагу зв'язному спискові, як у наступній програмі, виконаній за допомогою мови програмування

C++. Стек організовано у зворотному порядку в порівнянні з реалізацією на базі масиву — починаючи з останнього занесеного елемента і завершуючи першим. Цей спосіб (рис. 2.4) дозволяє більш просто реалізувати базові стекові операції.

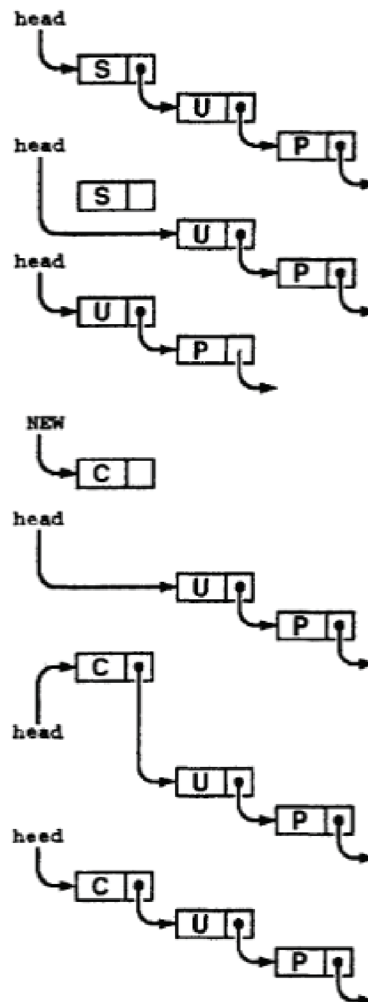


Рис. 2.4. Представлення стеку за допомогою зв’язного списку [8]

Щоб виштовхнути елемент, видаляється вузол на початку списку і витягається з нього елемент. Щоб додати елемент, створюється новий вузол і додається в початок списку. Оскільки всі операції зв'язного списку виконуються на початку списку, вузол, що відповідає верхівці стека є не потрібним [6].

```
template <class Item> class STACK
{ private:
struct node
{ Item item; node* next;
node(Item x, node* t)
{ item = x; next = t; }
};
typedef node *link; link head;
public:
STACK(int)
{ head = 0; } int empty () const
{ return head ==0; } void push(Item x)
{ head =* new node(x, head); } Item pop ()
{ Item v = head–>item; link t = head–>next; delete head; head = t;
return v; }
};
```

Наведена програма не перевіряє такі помилки, як спроба витягу елемента з порожнього стека, занесення елемента в переповнений стек або вихід за межі пам'яті. У відношенні перевірки двох останніх умов маються дві можливості. Їх можна трактувати як незалежні помилки і відслідковувати кількість елементів у списку, для чого при кожнім занесенні в стек необхідно перевіряти, чи не перевищує значення лічильник, переданий конструкторові як аргумент, і що *new* виконується успішно. Цілком доречно зайняти позицію, при якій не потрібно заздалегідь знати максимальний розмір стека, і, ігноруючи аргумент конструктора, повідомляти про те, що стек переповнений, тільки тоді, коли *new* завершується з помилкою.

Якщо потрібно створити стек великих розмірів, що зазвичай заповнюється практично цілком, очевидно, перевагу варто віддати реалізації на базі масиву. Якщо ж розмір стека варіюється в широких межах і присутні інші структури даних, яким потрібна пам'ять,

не використовувана під час, коли в стеці знаходиться кілька елементів, перевагу варто віддати реалізації на базі зв'язного списку.

В архітектурі практично всіх комп'ютерів використовується апаратний стек. Він представляє собою звичайну область внутрішньої (оперативної) пам'яті комп'ютера, з якою працює спеціальний регістр — покажчик стека. З його допомогою процесор виконує операції *Push* і *Pop* по збереженню й відновленню зі стека байтів і машинних слів різної розмірності. Єдина відмінність апаратного стека від розглянутої моделі (рис. 2.1) — це його розташування буквальне «нагору дном», тобто його заповнення від старших адрес до молодших.

Наведемо декілька додаткових способів запису, що будуть зручні при роботі зі стеками. Ми пишемо:

$$A \Leftarrow x \quad (1)$$

вказуючи, що значення x розміщується на верх стека.

Подібний запис:

$$x \Rightarrow A, \quad (2)$$

буде означати, що змінна x приймає значення верхнього елемента стека A. Запис (2) не має змісту, якщо A порожній, тобто якщо A не містить ніяких елементів.

При непустому стеку ми можемо визначати верхній елемент через:

$$top\ (A). \quad (3)$$

Найпростіший і найбільш природний спосіб збереження лінійного списку в пам’яті комп’ютера зводиться до розміщення вузлів списку в послідовних чарунках пам’яті один елемент за одним. У цьому випадку:

$$LOC\ (X[j+1])=LOC(X[j]+c). \quad (4)$$

Послідовний розподіл дуже зручний при роботі зі стеком. Для цього достатньо мати змінну T, що називають покажчиком стека (рис. 2.3). Коли стек порожній T=0. Щоб помістити новий елемент Y у стек, необхідно встановити:

$$T \leftarrow T+1\ ;\ X[T] \leftarrow Y, \qquad (5)$$

і, якщо стек не порожній, ми можемо встановити змінну Y рівну значенню, що зберігається в верхньому вузлі і видалити цей вузол діями, зворотими діям (5):

$$Y \leftarrow X[T];\ T \leftarrow T-1, \qquad (6)$$

(в силу (4) комп'ютеру ефективніше працювати зі значенням із T замість T. Такі зміни робляться легко і тому ми будемо і далі припускати, що $c=1$.)

2.3. Черга, дек

Черга — лінійний список, у якому всі вставки проводяться тільки на початку списку, а всі видалення (будь-який доступ) робляться в кінці списку.

Дек (черга з двома кінцями) — лінійний список, у якому всі вставки, видалення (будь-який доступ) робляться на обох кінцях списку [10].

Отже, дек має більшу спільність, чим стек або черга; він має деякі спільні властивості з колодою карт (в англійській мові ці слова співзвучні). Ми будемо розрізняти деки з обмеженим виходом або обмеженим входом; у таких деках відповідно вставка або видалення допускаються тільки на одному кінці.

Для черги справедливо наступне правило: видаляється завжди самий «старший» елемент; вузли покидають список у тому порядку, у якому вони в нього ввійшли. Чергу іноді називають циклічною пам'яттю або списком типу *FIFO* («*first-in-first-out*» — «першим додається — першим видаляється»). Протягом багатьох років бухгалтери використовували терміни *LIFO* і *FIFO* як назви методів при упорядкуванні прейскурантів. Ще один термін «архів» застосовувався в деках з обмеженим виходом, а деки з обмеженим входом називали «переліками» або «реєстрами».

Найпростіший спосіб представлення черги послідовністю, розміщеною від початку масиву, не зовсім зручний, оскільки при витягання із черги першого елемента всі наступні прийдеться постійно пересувати до початку. Альтернатива: у черги повинно бути два покажчики — на її початок у масиві й на її кінець. У міру постановки елементів у чергу її кінець буде просуватися до кінця масиву, те ж саме буде відбуватися з початком при видаленні елементів. Вихід з положення, що створилося — це «зациклити» чергу, тобто вважати, що за останнім елементом масиву іде знову перший. Подібний спосіб організації черги в масиві ще іноді називають циклічним буфером (рис. 2.5).

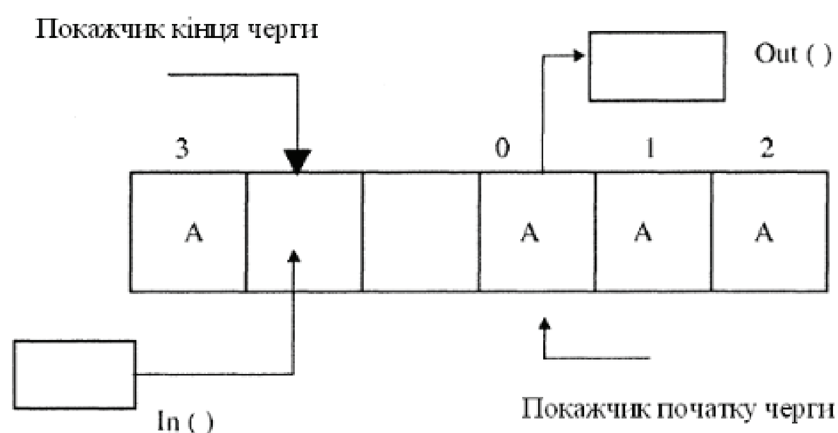


Рис. 2.5. Представлення черги

Наступна програма демонструє роботу з чергою за допомогою мови програмування *C++* [6].

```
// Основні операції з чергою (циклічний буфер):
#define SIZE 100            //Максимальна довжина черги
int QUEUE[SIZE];            //Масив елементів черги
int fst;                    //Покажчики на перший елемент черги
int lst;                    //Покажчик на наступний за останнім
void Clear () {fst=lst=0;}          //Очистити чергу
```

```
int In(int val){                    //Поставити в кінець черги
int next;
if((next=(lst+1)%SIZE)==fst)
return 0;                           //Переповнення черги
QUEUE[lst]=val;
lst=next; return 1;}
int Out() {                         //Взяти з початку черги
int val;
if (fst==lst)return 0;              //Черга порожня
val=QUEUE[fst++];
fst%=SIZE;                          //По досягненні fst==SIZE
return val;                         //прирівнюється до 0
}
```

На відміну від стека покажчик на кінець черги посилається не на останній зайнятий, а на перший вільний елемент масиву.

Поняття верху, низу, початку й кінця застосовуються іноді і до деків, якщо вони використовуються як стеки або черги. Не існує стандартних умов, що до того, де повинен бути верх, початок і кінець: зліва або справа.

Говорячи про деки, ми вказуємо лівий і правий кінці. Таким чином, ми бачимо, що в наших алгоритмах застосована багата розмаїтість описових слів: «зверху-вниз» — для стеків, «зліва-вправо» — для деків і «очікування в черзі» — для черг.

На рис. 2.6 представлено загальну схему функціонування напівстатичних структур: стеків, черг і деків.

При описанні черги запис:

$$A \Leftarrow x. \qquad (7)$$

Вказує, що x включається в кінець черги. Подібний запис:

$$x \Rightarrow A, \qquad (8)$$

буде означати, що змінна x приймає значення початкового елемента черги A, і це значення виключається з A.

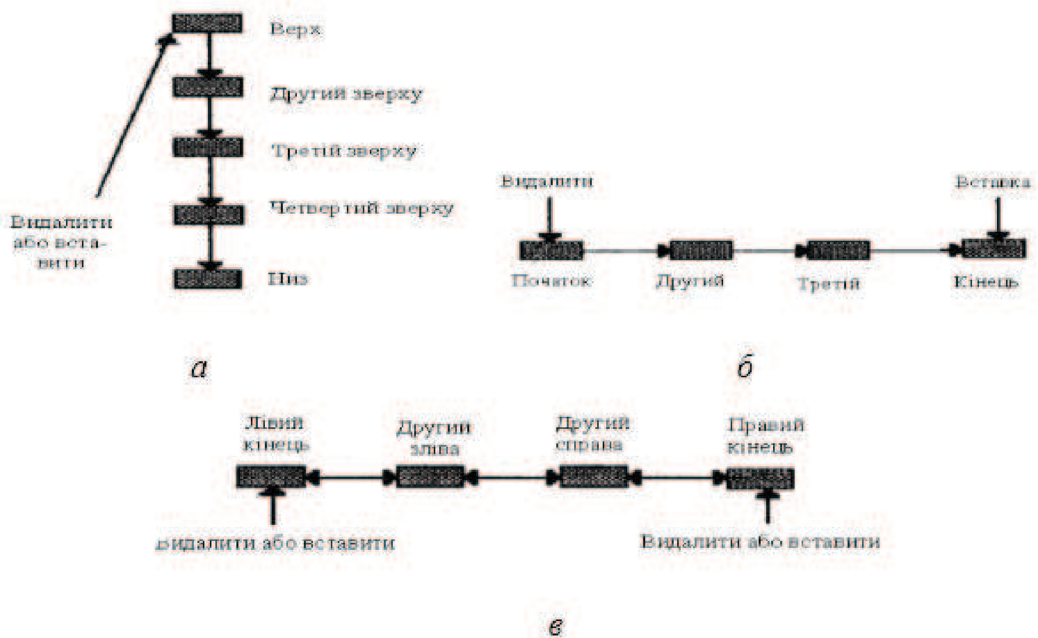


Рис. 2.6. Схема роботи напівстатичних структур [10]:
а — стек; *б* — черга; *в* — дек

Представлення черги або, більш того, загального дека потребує деяких хитрощів. Очевидне вирішення складається в тому, щоб мати два покажчики, скажемо F і R (для початку й кінця черги), причому $F=R=0$, якщо черга порожня. Тоді додавання елемента в кінець черги зводиться до операцій:

$$R \leftarrow R+1;\ X[R] \leftarrow Y \qquad (9)$$

а для видалення початкового вузла (F вказує місце безпосередньо перед початком черги) достатньо виконати:

$$F \leftarrow F+1;\ Y \leftarrow X[F];\ \text{якщо}\ \ F=R,\ \text{то}\ F \leftarrow R \leftarrow 0 \qquad (10)$$

Необхідно відзначити, що якщо R завжди випереджає F (і тоді в черзі є принаймні один вузол), то в таблиці послідовно розміщаються $X[1]$, $X[2]$,..., $X[100]$,... до безкрайності, що пов'язано з надзвичайно марнотратним використанням пам'яті.

Отже, простий метод (9), (10) повинен використовуватися лише в тій ситуації, коли відомо, що F досить регулярно наздоганяє

R (наприклад, якщо всі видалення з черги відбуваються спалахами, що спустошують чергу).

Можна обминути проблему виходу черги за межі пам'яті, фіксуючи M вузлів $X[1]$,.... $X[M]$ і неявно «замикаючи» їх в коло так, що за $X[M]$ випливає $X[1]$. Тоді процеси (9), (10) перетворяться до вигляду:

$$\text{якщо } R=M, \text{ то } R \leftarrow 1, \text{ інакше } R \leftarrow R+1;\ X[R] \leftarrow Y, \qquad (11)$$

$$\text{якщо } F=M, \text{ то } F \leftarrow 1, \text{ інакше } F \leftarrow F+1;\ Y \leftarrow R[F]. \qquad (12)$$

Приведений вище приклад сильно ідеалізований, оскільки по умовчанню передбачається, що не виникає ніяких колізій. Коли ми видаляємо вузол із стека або з черги, вважалося, що хоча б один вузол там є. Коли ми додавали вузол у стек або чергу, ми припускали, що для нього є місце в пам'яті. Але, очевидно, при методі (11), (12) у всій черзі не може бути більш M вузлів, а при методі (5), (6), (9), (10) у будь-який машинній програмі T і R не повинні перевершувати деякий максимальний розмір. Нижче показано, як повинні бути переписані розглянуті дії для загального випадку, коли ці обмеження можуть і не виконуватися:

$$\underset{\text{(вставити до стеку)}}{X \Leftarrow Y} \quad T \leftarrow T+1, \ \text{якщо } T \leftarrow M, \text{ то ПЕРЕПОВНЮВАННЯ; } X[T] \leftarrow Y. \qquad (13)$$

$$\underset{\text{(видалити зі стека)}}{Y \Leftarrow X} \quad \text{якщо } T=0, \text{ то НЕСТАЧА; } Y \leftarrow X[T];\ T \leftarrow T-1. \qquad (14)$$

$$\underset{\substack{\text{(включити}\\ \text{до черги)}}}{X \Leftarrow Y} \begin{cases} \text{якщо } R = M, \text{ то } R \leftarrow R+1; \text{інакше } R \leftarrow R+1 \\ \text{якщо } R = F, \text{ то ПЕРЕПОВНЕННЯ;} \\ X[R] \leftarrow Y \end{cases} \qquad (15)$$

$$\underset{\substack{\text{(видалити}\\ \text{з черги)}}}{X \Leftarrow Y} \begin{cases} \text{якщо } R = F, \text{ НЕСТАЧА;} \\ \text{якщо } F = M, \text{ то } F \leftarrow 1, \text{ інакше } F \to F+1 \\ Y \leftarrow X[F] \end{cases} \qquad (16)$$

Тут ми припускаємо, що $X[1]$,.... $X[M]$ складають той сумарний обсяг пам'яті, що відведено; ПЕРЕПОВНЮВАННЯ і НЕСТАЧА позначають надлишок або відсутність елементів.

Що ж робити, якщо трапляється ПЕРЕПОВНЮВАННЯ або НЕСТАЧА? НЕСТАЧА з'являється в тому випадку, коли ми намагаємося видалити неіснуючий елемент; звичайно така ситуація має сенс і не є помилковою, оскільки її можна використовувати для переходу з однієї ділянки програми на іншу. Наприклад, ми могли б багаторазово видаляти елементи доти, поки не виникне НЕСТАЧА. Проте ПЕРЕПОВНЮВАННЯ в більшості випадків свідчить про помилку; це означає, що таблиця вже повна, але існує ще інформація, яку необхідно помістити до списку. У такій ситуації, зазвичай повідомляють, що програма не може продовжувати свою роботу через відсутність пам'яті й виконання програми припиняється.

Зазвичай, не хотілося б припиняти роботу в ситуації ПЕРЕПОВНЮВАННЯ, якщо тільки один список переповнився, в той час як в інших списках тієї ж програми може залишатися значний запас простору. Дотепер мова йшла про програму з одним списком. Проте часто зустрічаються програми, що використовують декілька стеків, розмір кожного з яких динамічно змінюється. У такій ситуації не хотілося б мати необхідність, щоб кожний стек мав максимальний розмір, оскільки, зазвичай, цей розмір майже неможливо спрогнозувати, і навіть якщо задано максимальний розмір кожного зі стеків лише в рідкісних випадках усі стеки одночасно будуть заповнені на максимальну глибину.

Коли є два списки змінного розміру, то вони дуже добре можуть співіснувати разом, якщо ми дозволимо їм рости назустріч один одному (рис. 2.7).

Рис. 2.7. Спосіб розташування двох списків в пам'яті комп'ютера

Список 1 росте вправо, а список 2 (що зберігається в оберненому порядку) росте вліво. ПЕРЕПОВНЮВАННЯ не виникає доти, поки сумарний обсяг обох списків не вичерпає весь простір у пам'яті. Наведений на рис. 2.7 спосіб розподілу простору в пам'яті комп'ютера використовується дуже часто.

Треба зауважити, що немає способу, що дозволяє зберігати три (або більше трьох) послідовних списків змінного розміру так, щоб:

а) ПЕРЕПОВНЮВАННЯ виникало лише в тому випадку, коли сумарний розмір усіх списків перевищує відведену для них область пам'яті;

б) «нижній» елемент кожного списку мав фіксовану адресу.

Коли є, наприклад, десять або більше списків змінного розміру (а це цілком реальна ситуація), проблема розподілу пам'яті стає дуже важливою. Якщо ми хочемо задовольнити умові (а), нам доведеться порушити умову (б), тобто ми змушені припустити зміну позицій «нижніх» елементів цих списків. Це означає, що адреса L_0 у рівнянні (4) не є більше константою, тому що будь-які посилання в таблиці не можуть бути абсолютною адресою пам'яті, усі посилання повинні бути відносними з адресою бази L_0.

Важливий особливий випадок виникає, коли кожний із списків змінного розміру є стеком. Оскільки в цьому випадку в будь-який момент часу цікавить лише верхній елемент кожного зі стеків,

Припустимо, що в нас є n стеків; тоді, якщо $BASE[i]$ і $TOP[i]$ є змінними зв'язку для i-го стека, то алгоритми додавання й видалення є такими:

Додавання:

$TOP[i] \leftarrow TOP[i]+1$; якщо $TOP[i]>BASE[i+1]$, це ПЕРЕПОВНЮВАННЯ;

в іншому випадку установити:

$CONTENTS(TOP[i] \leftarrow Y)$ (17)

Видалити:

якщо $TOP[i]=BASE[i]$, це НЕСТАЧА,

в іншому випадку встановити $Y \leftarrow CONTENTS$

$(TOP[i], TOP[i] \leftarrow TOP[i]-1)$ (18)

У даному випадку $BASE\ [i+1]$ є базовою адресою $(i+1)$-*го* стека. Умова $TOP[i]=BASE[i]$ означає, що стек порожній.

У ситуації, що описана вище, ПЕРЕПОВНЮВАННЯ вже не є настільки критичним, як колись; ми можемо «перепакувати пам'ять», надавши таблиці, що переповнилася, простір, віднятий у таблиці, що ще не заповнена. Декілька можливих способів реалізації переупаковки напрошуються самі собою, тому що алгоритми дуже важливі у зв'язку з послідовним розподілом лінійних списків, ми розглянемо цю проблему у всіх подробицях. Почнемо з найпростішого з цих методів, а потім розглянемо деякі його варіанти.

Припустимо, що є n стеків і що зі значеннями $BASE[i]$ і $TOP[i]$ проводяться операції так само, як у (17) і (18). Стеки розташовуються в загальній області пам'яті, що складається з усіх чарунок L, для яких $L_0<L<=L_\infty$. (Тут L_0 і L_∞ — константи, що визначають область пам'яті, надану для використання.) Можна вважати, що спочатку усі стеки порожні і $BASE[i]=TOP[i]=L_0$ для всіх i. Ми також вважаємо, що $BASE[n+1]=L\infty$ і тоді (17) буде правильно виконуватися для $i=n$. Тепер ПЕРЕПОВНЮВАННЯ буде виникати кожен раз, коли в деякому стеці, за винятком стека n, виявляється елементів більше, чим було.

При переповнюванні стека реалізується одна з трьох можливостей:

а) Знайдемо найменше k (якщо таке існує), для якого

$$i<k<=n\times i\times TOP[k]<BASE[k+1].$$

Після цього зрушимо усе на одну позицію нагору:

$$CONTENTS(L+1)\leftarrow CONTENTS(L), \text{ для } TOP[k]>=L>BASE[i+1].$$

(Зауважте, що дії виконуються в порядку зменшення, а не збільшення значень L, щоб уникнути втрати інформації. Може трапитися, що $TOP[k]=BASE[i+1]$, і тоді не потрібно нічого зрушувати.) Встановлюємо $BASE[i]\leftarrow BASE[i]+1$, $TOP[i]\leftarrow TOP[i]+1$ для $i<j<k$.

б) Не можна знайти таке k, що задовольняє умові а), але є найбільше k, для якого $1<=k<i$ та $TOP[k]<BASE[k+1]$. Тепер усе зрушується униз на одну позицію:

$$CONTENTS(L-1)\leftarrow CONTENTS(L) \text{ для } BASE[k+1]<L<TOP[i].$$

(Зауважте, що це повинно виконуватися для зростаючих значень L.)

Встановлюємо

$$BASE[j]\leftarrow BASE[j]-1;\ TOP[j]\leftarrow TOP[j]-1 \text{ для } k<j<=i.$$

в) Для всіх $k<>i$ має місце рівність $TOP[k]=BASE[k+1]$. Тоді очевидно, ми не зможемо знайти місце для нового елемента стека, і роботу варто припинити.

Зрозуміло, що багато перших переповнювань стеків, що виникають при використанні цього методу, можна уникнути, якщо розумно вибирати початкові умови і не відводити із самого початку весь простір під n-ий стек.

Базуючись на досвід роботи з конкретною програмою, можна запропонувати більш підходящі початкові значення; проте, яким би гарним не був початковий розподіл, він дозволяє заощадити лише фіксовану кількість переповнювань, і ефект економії буде помічено тільки на ранній стадії роботи програми.

Описаний вище метод можна поліпшити, якщо при кожній переупаковці пам'яті готувати місце більш ніж для одного нового елемента. Операція переміщення таблиці в пам'яті потребує помітного часу, і ми можемо одержати виграш у швидкості, застосовуючи переміщення на 2 або на 3 позиції відразу, замість декількох переміщень на одну позицію.

Наступний фрагмент програми ілюструє реалізацію черги на базі масиву при використанні мови програмування *C++*. До вмісту черги відносяться всі елементи масиву, розташовані між індексами *head* і *tail*; при цьому враховується перехід з кінця на початок масиву. Якщо індекси *head* і *tail* рівні, черга вважається порожньою, однак якщо вони стали рівними в результаті виконання операції *put*, черга вважається повною. Зазвичай, перевірка на такі помилкові ситуації не виконується, але розмір масиву робиться на 1 більше максимальної кількості елементів черги. При необхідності програму можна розширити, включивши до неї подібного роду перевірки [6].

```
template <class Item>
 class QUEUE
{ private:
Item *q; int N, head, tail;
public:
QUEUE(int maxN)
{ q = new Item[maxN+l] ;
N = maxN+1; head = N; tail = 0; }
int empty() const
{ return head % N == tail; }
void put (Item item)
{ q[tail++] = item; tail = tail % N; }
 Item get()
{ head = head % N; return q[head++] ; }};
```

Представлення на базі масиву вимагає резервування оперативної пам'яті з обсягом, достатнім для запам'ятовування максимально очікуваної кількості елементів черги. У випадку ж представлення на базі зв'язного списку оперативна пам'ять використовується пропорційно кількості елементів у структурі даних; це відбувається за рахунок додаткової витрати пам'яті на зв'язки (між елементами) і додаткової витрати часу на розподіл і звільнення пам'яті для кожної операції.

Чергу також можна представити за допомогою зв’язного списку (рис. 2.8) [8].

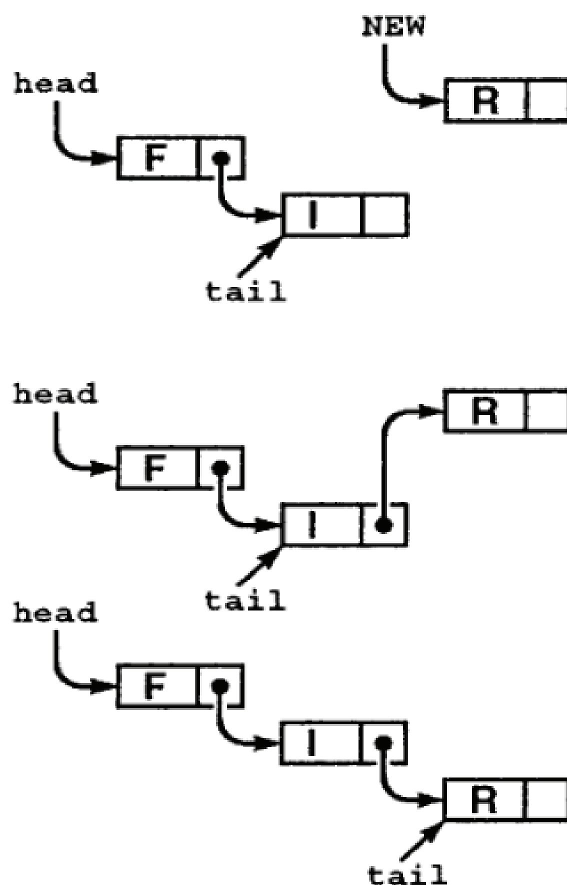


Рис. 2.8. Представлення черги за допомогою зв’язного списку

```
template <class Item> class QUEUE
{ private:
struct node
{ Item item; node* next;
node (Item x)
{ item = x; next =0; }
};
typedef node *link; link head, tail;
public:
QUEUE(int)
{ head =0; }
 int empty() const
{ return head ==0; }
void put(Item x) { link t = tail;
tail = new node(x); if (head == 0) head = tail; else t–>next = tail;}
Item get()
{ Item v = head–>item;
link t = head–>next;
delete head; head = t; return v;
}
};
```

У представленні черги у вигляді зв'язного списку нові елементи вставляються в кінець списку, тому елементи зв'язного списку від першого вставленого елементу до останнього розташовуються від початку до кінця черги. Черга представляється двома покажчиками: *head* (початок) і *tail* (кінець), що вказують, відповідно, на перший і останній елемент. Для витягання елемента з черги видаляється елемент на початку черги так само, як це робилося у випадку стека (рис. 2.4). Щоб занести в чергу новий елемент, поле зв'язку вузла, на який посилається покажчик *tail*, установлюється так, щоб воно вказувало на новий елемент, а потім обновляється покажчик *tail*.

Контрольні запитання та завдання для самоконтролю

1. Що таке стек?

2. Принцип роботи стека.

3. Що таке покажчик стека? У чому його потреба?

4. Дайте визначення черги.

5. Дек як особливий вид черги.

6. У чому відмінність стеків, деків і черг?

7. Що таке напівстатичні структури? Чому до них відносять стеки, деки, черги ?

8. Як функціонує черга, дек?

9. Які головні операції виконуються над півстатичними структурами?

10. Які проблеми можуть виникнути в процесі роботи зі стеками, деками, чергами?

11. Що необхідно робити у випадку виникнення переповнювання або нестачі?

12. Способи розміщення таких структур у пам'яті комп'ютера.

13. Дек з обмеженим входом є лінійним списком, у якому елементи можуть додаватися тільки з одного кінця, а видалятися з будь-якого кінця; очевидно, що дек з обмеженим входом може працювати або як стек, або як черга, якщо завжди будемо видаляти елементи з одного, із двох кінців. Чи може дек з обмеженим виходом також працювати або як стек, або як черга?

14. Чи існують які-небудь перестановки чисел 1, 2, , *n*, що не можна одержати за допомогою дека, що не має ні обмеженого входу, ні обмеженого виходу?

15. Скільки елементів одночасно може знаходитися в черзі, не викликаючи переповнювання, якщо операції з чергами задаються як у (4), (5)?

16. В послідовності

*EAS * Y * QUE *** ST*** 10 ******

буква означає операцію *put*, а зірочка — операцію *get*. Знайдіть послідовність значень, що повертаються операціями *get*, коли ця послідовність операцій виконується над спочатку порожньою чергою.

17. Модифікуйте приведену в тексті реалізацію черги на базі масиву так, щоб у ній викликалася функція *error*(), якщо клієнт намагається виконати операцію *get*, коли черга порожня, або операцію *put*, коли черга переповнена.

18. Модифікуйте приведену в тексті реалізацію черги *FIFO* на базі зв'язного списку так, щоб у ній викликалася функція *error*(), якщо клієнт намагається виконати операцію *get*, коли черга порожня, або якщо при виконанні *put* відсутня доступна пам'ять у *new*.

19. У послідовності

*EAs + Y + QUE ** + st+* + IO*n + + **

буква верхнього регістра означає операцію *put* на початку дека, буква нижнього регістра — операцію *put* наприкінці дека, знак плюс означає операцію *get* на початку, а зірочка — операцію *get* наприкінці. Знайдіть послідовність значень, що повертаються операціями *get*, коли ця послідовність операцій виконується над спочатку порожнім деком.

20. Дайте визначення наступним операціям, що виконуються зі стеком і чергою:

```
int sp =–1, LIFO[100];
int lst =0, fst =0, FIFO[100];
// 1
void F1()
{ int c; if (sp < 1) return;
c = LIFO[sp]; LIF0[sp] = LIF0[sp – 1 ] ; LIF0[sp–1 ]=c; }
// 2
int F2(int n)
{ int v , i;
if (sp < n) return (0);
V = LIFO[sp–n];
for (i=sp–n: i<sp; i++) LIF0[i] = LIF0[i + 1] ;
```

```
sp—; return v;}
// 3
void F3(){ LIFO[sp+1] = LIFO[sp]; sp++; }
// 4
int F4(int n)
{ int v,i1 ,i2;
i1 = (fst+n) %100;
v = FIF0[i1];
for (; i1!=lst; i1 = i2){
i2 = (i1 + 1) % 100;
FIF0[i1] = FIF0 [i2] ; }
1st = —1st % 100; return v;}
// 5
void F5()
{ int n;
if (fst==lst) return;
n = (lst–1) %100;
FIFO[lst] = FIFO[n];
lst = ++lst % 100;}
```

Розділ 3. ЛІНІЙНІ ДИНАМІЧНІ СТРУКТУРИ: ОДНОЗВ′ЯЗНІ І ДВОЗВ′ЯЗНІ СПИСКИ

3.1. Спискові структури

Спискові структури даних зазвичай динамічні. Цьому є дві причини:

- змінні таких структур створюються як динамічні змінні, тобто кількість їх може бути довільною;

- кількість зв'язків між змінними і їх характер також визначаються динамічно в процесі роботи програми. У програмі спискові структури даних доступні через покажчик на деякий її елемент, який називається заголовком.

Робота зі списками здійснюється винятково через покажчики. Кожний з них переміщається за списком (переустановлюється з елемента на елемент), здобуваючи одну зі значеннєвих інтерпретацій — покажчик на перший, останній, поточний, попередній, новий і інші елементи списку. Тут доречна аналогія з масивом (табл. 3.1) і індексом у ньому, але за умови, що індекс міняється лінійно, а не довільно, а поточна кількість заповнених елементів у масиві задана окремою змінною.

Таблиця 3.1. **Порівняння списку і масиву**

Опис, дія	Список	Масив
Визначення	*struct list {int val; list *next, *pred;};*	*int A*[100]; *int n;*
Порожня структура даних	*list *ph = NULL;*	*n*=0;
Перший	*list *p; p = ph;*	*int i*=0*;*
Наступний	*p–>next*	*i* +1
Попередній	*p–>pred*	*i* – 1
До наступного	*p = p–>next*	*i* ++

Продовження табл. 3.1

До попереднього	*p = p–>pred*	*i– –*
Перегляд	*for (p = ph; p! = NULL; p=p–>next)...p–>val...*	*for (i=0; i<n; i++) ...A[i]...*
Останній	*p–>next = = NULL*	*i= =n–1*
До останнього	*for (p=ph; p–>next!=NULL; p = p–>next);*	*i = n–1*
Новий	*list *q = new list; q–>val = v;*	*int v;*
Додати останнім	*for (p = ph; p–next!=NULL; p=p–>next); q–>next=NULL; p–>next=q;*	*A[n++]=v;*
Додати першим	*q–>next=ph; ph=q;*	*for (i = n; i>0; i– –) A[i]=A[i–1]; A[0]=v; n++;*

Статичний список представляє собою звичайні змінні — елементи списку, зв'язки між ними ініціалізуються транслятором, вся структура даних «зашивається» у програмний код (мова програмування *C*++) [6]:

```
struct list { Int val; list *next; } a={0,nuLL}, b={1,&a},
c={2,&b}, *ph = &c;
```

Відмітимо, що за умовами визначення змінних список створюється «хвостом вперед». Список може містити обмежену кількість елементів, узятих з масиву. Зв'язки встановлюються динамічно, тобто програмою. Такий варіант використовується, коли фіксована кількість елементів утворює декілька різних динамічних структур (наприклад, черг), в яких елементи списку переносяться з однієї структури в іншу (мова програмування *C*++) [4]:

```
list A[100],*ph;            // Створити список елементів,
for (1=0; i<99; i++) {
```

```
// розташованих в статичному масиві
A[i].next = A + i + 1 ;
A[i].val = i;
}
A[99].next = NULL;
ph = &A[0];
```

У динамічному списку елементи є динамічними змінними, зв'язки між ними встановлюються програмно (мова програмування *C++*) [4]:

```
list *ph = NULL;                    // Список порожній
for (int i=0; i<10; i++){           // Створити список з 10
елементів,
list *q = new list;                 // додаючи наступний в
начало
q–>val = l;                         // списку
q–>next=ph;
Ph=q; }
```

У програмі список зазвичай задається заголовком — покажчиком на перший елемент. Порожньому списку відповідає *Null*-покажчик. Функція, що працює зі списком, повинна мати обов'язковий параметр — заголовок списку (мова програмування *C++*) [4]:

```
// Формальний параметр — заголовок списку
void F1(list *p) {
for (; p! = NULL; p=p–>next) puts(p–>val);
}
```

Враховуючи той факт, що параметри в *C++* передаються за значенням (у вигляді копії), цей варіант корисний тільки в тому випадку, коли перший (по порядку) елемент списку залишається першим у процесі роботи зі списком, а якщо ні, то необхідна зміна самого покажчика (заголовка), яке може проводитися:

- поверненням зміненого значення заголовка у вигляді результату функції;

- передачею покажчика на заголовок списку (покажчика на покажчик);

- передачею посилання на заголовок. Нагадаємо, що посилання — неявний покажчик, що використовує при роботі синтаксис об'єкта, який «відображається» на відповідний йому фактичний параметр [6].

```
// Додавання до початку списку зі зміною заголовка
// Варіант 1. Змінений покажчик повертається
list *lns1(list *ph, int v)
{ list *q=new list;
q–>val=v; q–>next=ph; ph=q;
return ph; }
// Варіант 2. Використовується покажчик на заголовок
void Ins2(list **pp, int v)
{ list *q = new list;
q–>val=v; q–>next=*pp; *pp=q; }
// Варіант 3. Використовується посилання на покажчик
void Ins3(list *&pp, int v)
{ list *q = new list;
q–>val=v; q–>next=pp; pp=q; }
// Приклад виклику
void main(){
list *ph = NULL;                    // Порожній список
ph = lns1 (ph,5);                   // Зберегти новий заголовок
Ins2(&ph.66);                 // Передається адреса заголовка
Ins3(ph,7); }           // Передається посилання  на заголовок
```

Логічний порядок проходження елементів списку міняється шляхом переустановлення покажчиків в елементах списку, що проводиться операціями присвоювання покажчиків. Для їхнього розуміння користуються декількома змістовними інтерпретаціями.

1. Графічна інтерпретація присвоювання покажчика:

- у лівій частині операції присвоювання повинне перебувати позначення гнізда, у яке заноситься нове значення покажчика, причому воно може бути досягнуто тільки через наявні робочі покажчики. На рис. 3.1 цьому відповідає ланцюжок операцій *q–>pred–>next;*

- у правій частині операції присвоювання повинне перебувати позначення гнізда, з якого береться значення покажчика p, на рис. 3.1.

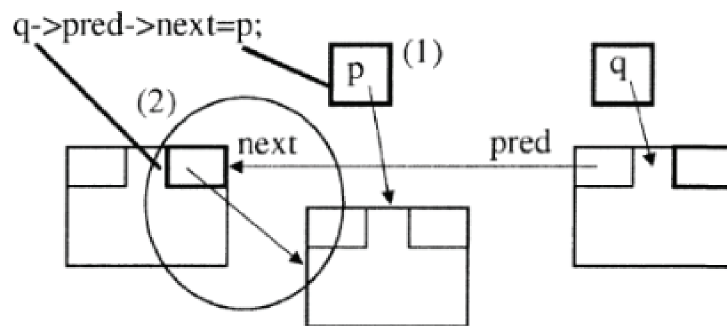


Рис. 3.1. Схема присвоювання покажчика

2. Адресна інтерпретація присвоювання покажчика (рис. 3.2). Вмістом покажчика є адреса указаної змінної.

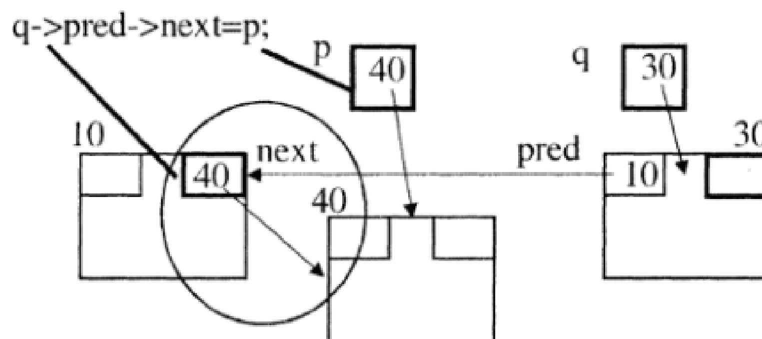


Рис. 3.2. Схема адресної інтерпретації присвоювання покажчик

3. Значеннєва інтерпретація присвоювання покажчика. При роботі зі списками кожний покажчик має певний сенс — перший, поточний, наступний, попередній і інші елементи списку. Поля *pred*, *next* також інтерпретуються як покажчики на наступний і попередній в елементі списку, доступному через покажчик. Тоді зміст присвоювання покажчиків однозначно переводиться у словесний опис. Наприклад, послідовність дій по додаванню нового елемента (покажчик *q*) у двозв'язний список перед поточним (покажчик *p*) коментується так (мова програмування *C++*) [6]:

```
q–>next=p;            // Наступний для нового = поточний
q–>pred = p–>pred; // Попередній для нового = попередній
if (p–>precl == NULL)                 // поточного
ph = q;                          // Додавання до начала списку
else                             // Додавання в середину
p–>pred–>next = q;  // Наступний для попереднього = новий
p–>pred=q;            // Попередній для поточного = новий
```

Основна властивість лінійних списків як структур даних – послідовність обходу списку залежить не від фізичного розміщення елементів списку в пам'яті, а від послідовності їх зв'язування покажчиками. Так само визначається нумерація елементів списку: логічний номер елемента в списку — це номер, одержуваний ним у процесі обходу списку.

Спискові структури представляють собою такий спосіб організації даних у пам'яті комп'ютера, при якому фізично розкидані елементи об'єднуються в логічно упорядковану сукупність за допомогою ланок зв'язку (посилань, покажчиків). Кожний елемент списку при цьому сам може бути списком.

Використання посилань при реалізація спискових структур дозволяє вводити потенційно нескінченні або циклічні структури і вказувати, що деяка підструктура належить декільком різним структурам. З цього випливає, що повинна існувати чітка система, що дозволяє відрізняти дані від посилань на них. Тому необхідно

вводити тип для даних, значення яких є посиланнями (покажчиками) на інші дані.

Найпростішими зв'язними списками є лінійні зв'язні списки: однозв'язний список і двозв'язний список.

Аналітичний запис спискової структури будується згідно таких правил:

- в якості елемента складного списку може виступати список будь-якої структури, що буде підсписком даного списку;
- кожний підсписок береться в круглі скобки;
- елемент списку записується послідовно один за одним відповідно до порядку їхнього проходження і відокремлюються один від одного комами;
- для ідентифікації значень елементів можуть використовуватися як великі, так і малі букви.

При визначенні структури списку зовнішні дужки не враховуються, а далі в кожній відкритій дужці знаходиться відповідна їй закрита дужка; елементи, вкладенні в дужки є підсписками й оголошуються складними елементами. Елементи, ідентифіковані буквами, оголошують простими.

Приклад 3.1. Нехай задано список такої структури (*b*, (*a*, (*b*, ( )), *b*), (*b*, (*a*))). Відкинувши зовнішні дужки, одержимо вираз *b*, (*a*, (*b*, ( )), *b*), (*b*, (*a*)), тобто в списку три елементи:

1-й простий елемент *b*;

2-й складний елемент (підсписок 1) (*a*, (*b*, ( )) *b*);

3-й складний елемент (підсписок 2) (*b*, (*a*)).

Це означає, що при графічній інтерпретації структури цього списку на першому етапі необхідно задати покажчики для трьох елементів. На другому етапі необхідно задати структуру підсписків 1 і 2.

Підсписок 1 включає три елементи:

- простий елемент *a*;
- складний елемент (підсписок 3) (*b*, ( ));
- простий елемент *b*.

Підсписок 2 включає два елементи:

- простий елемент *b*;

- складний елемент (підсписок 4) (*a*).

Таким чином, на другому етапі необхідно задати покажчики для п'яти елементів. На третьому етапі задається структура підсписків 3 і 4. Підсписок 3 містить два елементи:

- простий елемент *b*;

- складний елемент (підсписок 5) ( ), цей підсписок порожній, тобто не містить елементів.

Підсписок 4 містить усього один простий елемент *a*.

На третьому етапі повинні бути задані покажчики для трьох елементів. Таким чином, в аналітичній інтерпретації структури можна зазначити такі елементи:

Ел1 Ел 2 підсписок 1 Ел 3 підсписок 2
підсписок 3 підсписок 4
підсписок 5
(*b*, (*a*, (*b*, ( )), *b*), (*b*, (*a*)))

Графічне представлення спискових структур може бути виконане у вигляді однозв'язного, двозв'язного і кільцевого списків, що розглянуті нижче.

3.1.1. Лінійний однозв'язний список

Однозв'язний список є структурою, у якій елементи списку і підсписків «проглядаються» тільки в одному напрямку. Кожний елемент однозв'язного списку складається з двох різноманітних по призначенню полів: змістовного поля й поля покажчика (рис. 3.3).

У змістовному полі зберігаються дані, заради яких і створюється список. Змістовне поле в загальному випадку представляє собою запис, причому деякими його компонентами можуть бути покажчик додаткової інформації, що ставиться до елемента списку, але розташований не безпосередньо в змістовному полі.

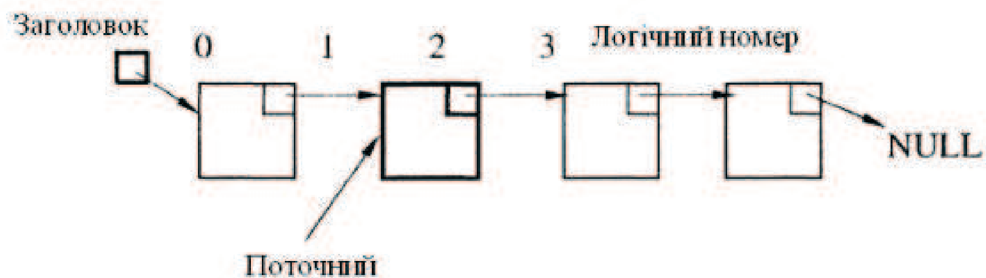


Рис. 3.3. Однозв'язний список

Поле покажчика зберігає адресу вибраного елемента списку. Користуючись покажчиком, можна одержати доступ до вибраного елемента списку, а з обраного елемента до чергового і т. д., поки не буде знайдено потрібний елемент; поле покажчика останнього елемента повинно містити спеціальну ознаку цільового або порожнього покажчика, що свідчить про кінець списку. Лінійність однозв'язного списку випливає з лінійної логічної упорядкованості його елементів: для кожного елемента (крім першого й останнього) є єдиний попередній і єдиний наступний. Логічна структура однозв'язного списку може бути представлена таким способом (рис. 3.4).

Як очевидно з рис. 3.4, частиною логічної структури однозв'язного списку є покажчик початку списку — УД, що містить адресу першого елемента списку. Елемент списку — ЕС містить адресу елемента списку *Ап* і значення даних *Dn*, якщо це простий елемент. Якщо елемент списку складний, то він містить адресу елемента списку *Ап* і адресу першого елемента підсписку *Апс*. Елемент підсписку містить адресу елемента підсписку й значення даних *Dnc*, якщо елемент простий. Якщо елемент підсписку складний, то він містить адресу елемента свого підсписку й адресу першого елемента свого підсписку. На рис. 3.4 також показано, що в полі покажчика останнього елемента списку або підсписку знаходиться спеціальна ознака 0, що свідчить про кінець списку.

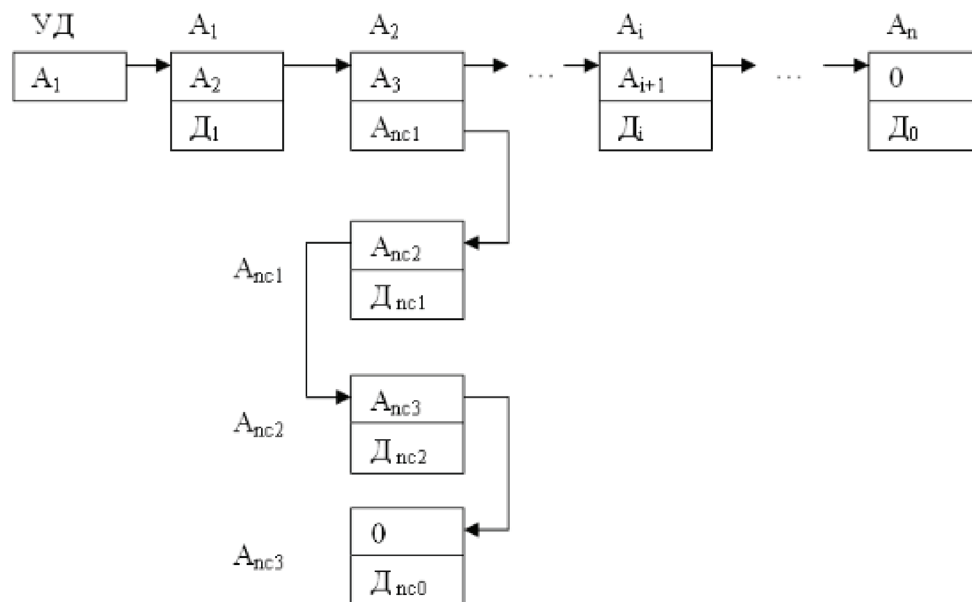


Рис. 3.4. Логічна структура однозв'язного списку

Фізична структура однозв'язного списку є сукупністю дескриптора й однакових по розміру й формату записів, розміщених довільно в деякій області пам'яті і пов'язаних один з одним лінійно упорядкованим ланцюжком за допомогою покажчиків. Таким чином, фізична суміжність елементів однозв'язного списку в пам'яті непотрібна, причому між елементами одного списку в пам'яті можуть бути елементи інших списків .

Дескриптор однозв'язного списку може бути реалізований у вигляді запису і містити інформацію про список: код структури, ім'я списку, адреса (покажчик) початого списку, кількість елементів у списку й опис елемента (наприклад, загальна довжина слова в байтах, тип структури для схову даних і т. п.). Дескриптор може знаходитися в тій області пам'яті, у якій розташовуються елементи списку або для нього виділяється будь-яке інше місце в пам'яті.

Приклад 3.2. Набір даних заданий таким способом (*A*, ((*B*, (*C*, *D*), *E*),(*X* (*Y*, *Z*)), *F*)) представити у вигляді однозв'язного списку, розмістити в пам'яті комп'ютера вільні чарунки з такими умовними адресами: від 21 до 34.

Розв'язання задачі представлено в графічному вигляді на рис. 3.5.

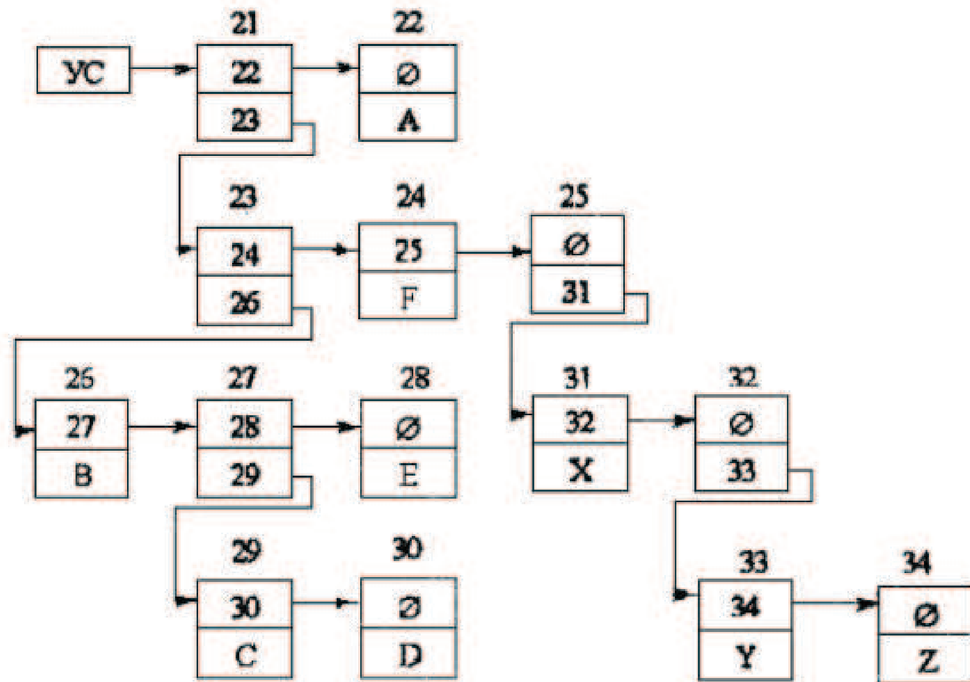


Рис. 3.5. Графічне подання розв'язання приклада 3.2

У деяких випадках додавання і видалення елементів потребує збереження покажчика на попередній елемент. Наприклад, для додавання в список зі збереженням порядку зростання місце вставки нового елемента — це перед першим більшим за введений при перегляді списку від початку. Це вимагає зміни значення покажчика в попередньому елементі списку [8].

// Додавання до однозв'язного списку зі збереженням порядку (C++):

```
// pr — покажчик на попередній елемент списку
void InsSort(list *&ph, int v)
{ list *q ,*pr,*p;
q = new list; q–>val = v;
// Перед переходом до наступного покажчик на поточний
```

```
// Запам'ятовується  як покажчик на попередній
for ( p=ph,pr=NULL; p!=NULL && v>p–>val; pr=p,p=p–>next);
if (pr==NULL)                    // Додавання перед першим
{ q–>next=ph; ph=q; }
else                             // Інакше після попереднього
{ q–>next=p;          // Наступний для нового = поточний
pr–>next=q; }}        // Наступний для попереднього = новий
```

Додаткова перевірка «крайніх» ситуацій показує, що фрагмент, який робить пошук місця вставки, коректно працює і у випадку порожнього списку (працює по гілці — додавання перед першим).

3.1.2. Лінійний двозв'язний список

Часто при перегляді лінійного списку виникає необхідність просування в будь-якому із двох напрямів по ланцюжку елементів. Таку можливість забезпечує двозв'язний список. Лінійний двозв'язний список відрізняється від однозв'язного списку тим, що кожний його елемент містить два покажчики, один із яких (прямий) адресує, як і в однозв'язному списку, елемент, а інший покажчик (зворотній) адресує попередній елемент списку (рис. 3.6).

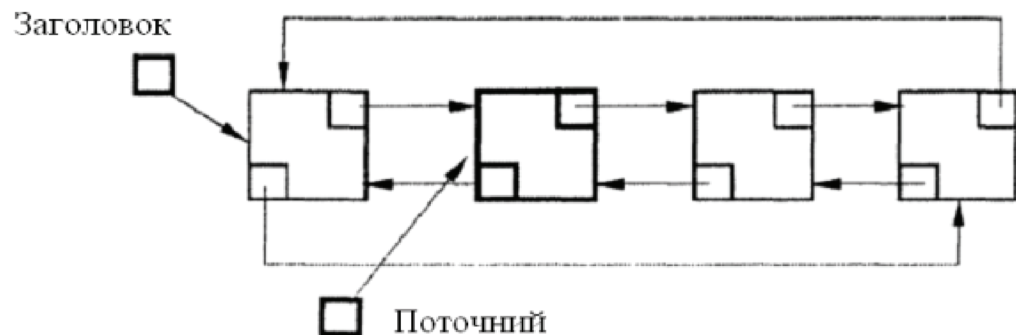


Рис. 3.6. Двозв'язний список

Лінійність двозв'язного списку випливає з того, що кожний із двох покажчиків у будь-якому елементі списку (крім крайніх елементів, у яких один із покажчиків порожній) задає лінійний порядок елементів, зворотній відносно порядку, який встановлюється іншим покажчиком. Логічна структура лінійного двозв'язного списку подана на рис. 3.7 [13].

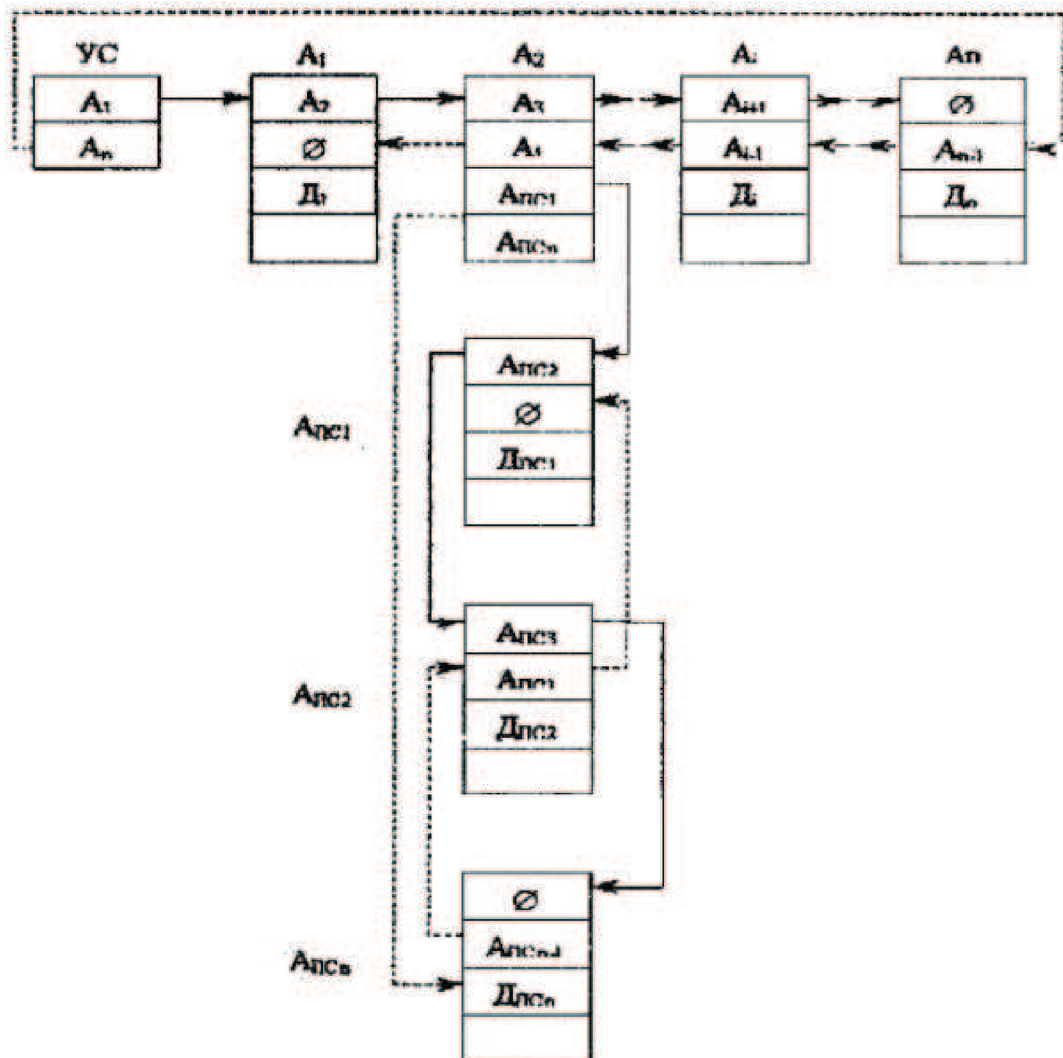


Рис. 3.7. Логічна структура двозв'язного списку

Двозв'язний список дозволяє рухатися по ланцюжкові елементів в обох напрямках, маючи доступними наступний і попередній елементи. «Розплачуватися» за це доводиться збільшенням кількості операцій над покажчиками. Наступний фрагмент програми, виконаний на мові програмування *C++*, показує видалення елемента списку по заданому логічному номеру [8].

```
// Видалення елемента списку по заданому логічному номеру
void Del(list *&pp, int n)
{ list *q;                          Покажчик на поточний елемент
for (q = pp; q!=NULL && n!=0; q = q–>next, n– –);
if (q==NULL) return;         Немає елемента з таким номером
if (q–>pred==NULL)                Видалення першого –
pp=q–>next;                       Корекція заголовка
else q–>pred–>next = q–>next;
// Наступний для попереднього = наступний за поточним
if (q–>next! = NULL)              // Видалення не останнього –
q–> next–>pred = q–>pred;
 Попередній для наступного =// попередній поточного
delete q; }
```

Приклад 3.3. Розв'яжемо наступне завдання. Список задано в табл. 3.2.

Таблиця 3.2.

Адреса даних	Вміст даних	Адреса зв'язку
210	Короп	213
211	Вовк	
212	Синиця	217
213	Йорш	219
214	Окунь	210
215	Ведмідь	
216	--------------	218
217	--------------	220
218	Ворона	212
219	Карась	216
220	--------------	

Побудувати структурну схему, що дозволяє читати заданий список у прямому і зворотному напрямках. У пам'яті вільні чарунки з умовними адресами від 314 до 320. Графічне подання розв'язку прикладу 3.3 наведено на рис. 3.8.

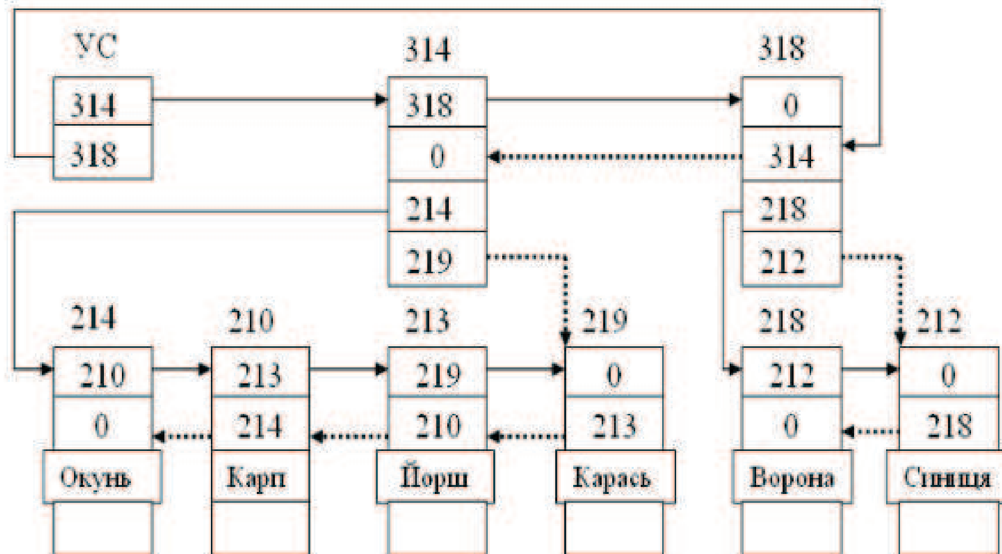


Рис. 3.8. Графічне розв'язання приклада 3.3.

3.1.3. Кільцеві структури

З розгляду логічної структури однозв'язного і двозв'язного списків очевидно, що для доступу до бажаного елемента необхідно переглянути список, у загальному випадку, з його початку, навіть якщо вихідною точкою перегляду є деякий елемент, що знаходиться в списку близько від шуканого елемента, але після нього в ланцюжку елементів. Це уповільнює операції доступу до елементів списку.

Кільцеві структури є такою формою організації пам'яті комп'ютера, при якій зчитування даних з одного або групи елементів відбувається доти, поки не виконається умова переходу до

шуканого елемента або групи елементів. Умова переходу, зазвичай, задається у вигляді кількості циклів «перегляду» попереднього елемента або групи елементів сполучених у кільце.

Застосовуються кільцеві структури в тому випадку, коли вихідний набір є повторюваною групою даних, а також, коли одиночний елемент або група елементів повинна бути повторена *n*-у кількість разів за умовами розв'язуваної задачі.

Кільцева структура, що передбачає *n*-кратний перегляд елементів в одному напрямку, називається однозв'язною.

Якщо перегляд елементів кільцевої структури може здійснюватися в двох напрямках, то така кільцева структура називається двозв'язною. У загальному випадку для задання кільцевої структури необхідно мати:

а) покажчик списку (УС), що містить інформацію про розташування першого елемента списку;

б) покажчик елемента списку (ЕС). Якщо цей елемент простий, то він містить значення даних і адресу покажчика переходу, якщо цей елемент складний, то він замість значення даних містить адресу першого елемента підсписку (кільця);

в) покажчик переходу (УП), що містить інформацію про розташування наступного елемента списку й інформацію про кількість переглядів попереднього елемента;

г) покажчик елемента підсписку (*Епс*), що містить інформацію про розташування елемента підсписку й значення даних, якщо цей елемент простий і, якщо елемент підсписку (кільця) сам є списком, то замість значення даних вказується адреса першого елемента чергового підсписку й адреса покажчика переходу і т .д.

Через те, що будь-який елемент будь-якого підсписку може бути списком, то побудувати загальну модель кільцевої структури неможливо. На рис. 3.9 наведено приклад представлення кільцевої структури в загальному вигляді, що дозволяє зрозуміти принцип її організації.

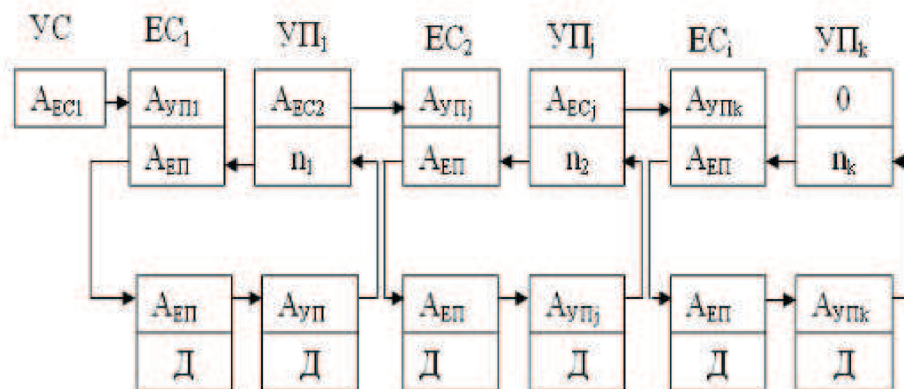


Рис. 3.9. Логічна структура кільцевого однозв'язного списку

На рис. 3.9 букви $n_1, n_2, ...n_k$ означають кількість переглядів відповідних елементів кільця.

Для великої кількості задач, кільцева структура може бути представлена в більш простій формі. У цьому випадку для задання кільцевої структури достатньо мати:

а) покажчик списку (УС), що містить інформацію про розташування першого елемента списку,

б) покажчик переходу (УП), що містить інформацію про кількість переглядів попереднього елемента і про розташування наступного елемента списку (підсписку);

в) елемент списку (ЕС). Якщо це простий елемент списку, то він містить значення даних і адресу покажчика переходу, а якщо це елемент кільця, то він містить значення даних і адресу такого елемента кільця, або адресу покажчика переходу, якщо це елемент останній у кільці.

Приклад 3.4. Представити графічне зображення для списків $(KB)F(CE)$ у вигляді однозв'язної кільцевої структури. Розв'язок завдання представлено на рис. 3.10.

Для задання двозв'язної кільцевої структури необхідно зазначити:

а) УС — покажчик списку, що містить адресу першого й останнього елемента списку, відповідно перший й останній у кільці;

б) ЕС — елементи списку, що містять дані, адреси елементів для прямого перегляду й адреси попередніх елементів для перегляду в зворотному напрямку.

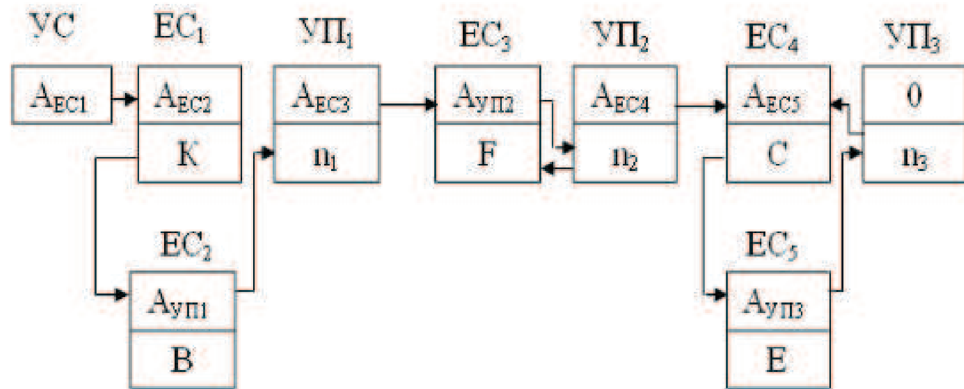


Рис. 3.10. Кільцевий однозв'язний список (*KB*)*F*(*CE*)

Перший і останній елементи кільця містять замість однієї адреси елемента адресу покажчика переходу (рис. 3.11).

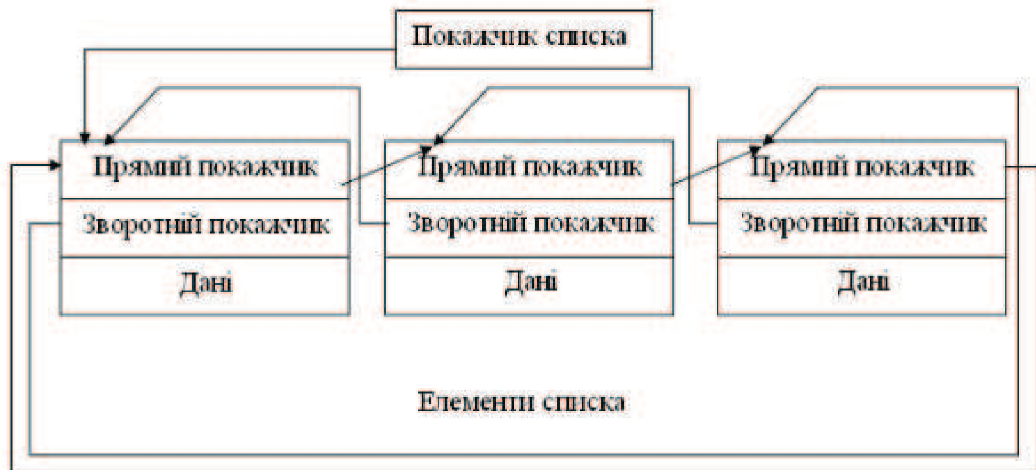


Рис. 3.11. Логічна структура кільцевого двозв'язного списку

Якщо необхідно переглянути простий елемент списку *k* один раз, то ЕС повинен містити лише адресу УП і дані. УП містить інформацію:

а) про кількість циклів у кільці;

б) адреса першого елемента даного кільця для прямого перегляду;

в) адреса останнього елемента попереднього кільця для зворотного перегляду .

УП першого й останнього кілець містить замість однієї з адрес, адреси для позначення кінця перегляду в прямому й зворотному напрямках .

Циклічний список дозволяє моделювати лінійні ланцюжки елементів, виключивши постійні перевірки на «перший» і «останній». Особливості такого списку:

- поле *next* останнього елемента посилається на перший елемент, а поле *pred* першого — на останній елемент списку;

- один елемент списку посилається сам на себе (*q–>next=q і q–>pred =q)*;

- операції вставки елемента в початок і в кінець списку ідентичні за винятком того, що в першому випадку міняється заголовок.

Цикл перегляду такого списку припускає повернення покажчика поточного елемента на початок списку в циклі з постумовою (мова програмування *C*++).

```
list *p=ph;
do {
// Тіло циклу для поточного елемента — p
p=p–>next;
} while (p!=ph);
```

3.2. Операції над списками

Розглянемо основні операції, що можна виконувати над списками різних видів. Головні операції, виконувані над списками: формування списку, вставка нового елемента в список, видалення елемента зі списку, пошук елемента в списку.

3.2.1. Формування списку

Для задання однозв'язного списку необхідно, щоб:

- покажчик початку списку містив адресу першого елемента списку;

- кожний елемент списку й елемент підсписку містили адреси елементів, що слідують далі.

Якщо елемент списку або елемент підсписку є складним елементом, то замість значення даних вони повинні містити адресу першого елемента чергового підсписку. Останні елементи списку й підсписків повинні містити ознаку кінця.

Для задання двозв'язного списку необхідно, щоб покажчик початку списку містив адреси першого й останнього елементів списку. Кожний елемент списку й підсписку містив адреси, як елементів, що слідують за ним, так і попередніх.

Якщо елемент списку або елемент підсписку є складним, то замість значення даних він повинен містити адреси першого й останнього елементів чергового підсписку. Останні елементи списку і підсписку повинні містити покажчик кінця в обох напрямках.

3.2.2. Вставка елемента до списку

Проілюструємо дану операцію для випадку однозв'язного списку. Список, у який слід додати новий елемент, може бути порожній або містити один, два елементи та більше. Новий елемент може додаватися до початку списку, в кінець або будь-куди у се-

редину списку. В залежності від цього операція вставки має свої особливості виконання. Припустимо, що оброблюваний функціональний список може містити довільну кінцеву кількість елементів і потрібно додати до нього новий елемент слідом за елементом цього ж списку, що адресується робочим покажчиком. Припустимо, що новий елемент уже є в пам'яті комп'ютера і його дані сформовані належним чином. Вихідна ситуація показана на рис. 3.12, *а*, на якому пунктирна стрілка вказує низку, що підлягає розриву [14].

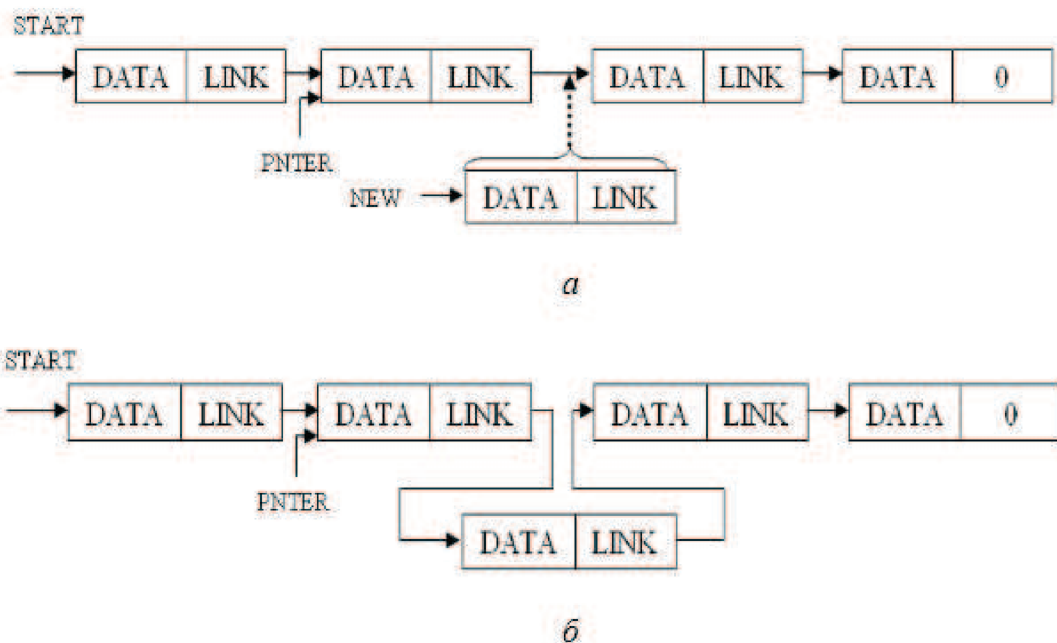


Рис. 3.12. Ілюстрація операції вставки нового елемента до однозв'язного списку: *а* — до вставки; *б* — після вставки

Операція вставки нового елемента до списку виконується таким способом. У поле списку, до якого додається елемент, заноситься вміст того елемента, слідом за яким у списку повинен бути додано новий елемент. Тим самим установлюється зв'язок, що включає елемент до списку. Потім у поле зв'язку елемента, слідом за яким додається новий елемент, заноситься значення по-

кажчика, що включає елемент, у результаті чого встановлюється зв'язок нового елемента з попереднім у списку елементів.

Для створення алгоритму цієї операції, введемо позначення змінних, що будуть потрібні для його опису. Дані, що зберігаються в змістовному полі елемента списку, зазначимо *DATA*. Відповідно до визначення зв'язкового списку *DATA* — це запис, причому структура цього запису однакова для кожного елемента списку. В елементів однозв'язного списку покажчик елемента будемо позначати *LINK*, а в елементі двозв'язного списку покажчики поточного і попередніх елементів — відповідно *SLINK* і *PLINK*. Покажчику початку списку, відносно до якого буде досліджуватися та чи інша операція, призначимо ім'я *START*. Відзначимо, що покажчик *START* зберігається в дескрипторі списку. Крім покажчика *START* у процесі опрацювання списку може знадобитися робочій покажчик, що приймає значення адреси того або іншого елемента списку. Дамо робочому покажчику ім'я *PNTER*.

Якщо покажчик не адресує ніякий елемент списку, будемо вважати, що його значення дорівнює 0.

Нам буде потрібен також список вільних елементів або вільний список, що повинен служити джерелом пам'яті при формуванні елементів для функціональних списків. Кожен елемент вільного списку має такий само формат полів, як і елемент функціонального списку, але вміст поля *DATA* в елементі вільного списку не визначений.

Домовимося також вважати, що вираз *DATA (PNTER)* подає дані того елемента списку, що адресується покажчиком *PNTER*. Відповідно вираз *LINK (PNTER)* має значення адреси або зв'язки елементів, що адресуються покажчиком *PNTER* і нарешті, запис *PNTER<–LINK(RNTER)* виражає привласнювання покажчику *PNTER* значення зв'язки *LINK* того елемента, що у даний момент адресується покажчиком *RNTER* (це відповідає переміщенню покажчика на вибраний елемент списку).

Нижче наведено алгоритм вставки нового елемента до однозв'язного списку, який використовує описані вище змінні. Передбачається, що вміст поля *DATA* нового елемента вже сформовано. Покажчик *NEW* адресує цей елемент.

Алгоритм 3.1:

1. [Формування поля зв'язки нового елемента.] Встановити *LINK*(*NEW*) <– *LINK*(*PNTER*).

2. [Модифікація поля зв'язки попереднього елемента.] Встановити *LINK*(*PNTER*) < –*NEW*.

Усі перераховані особливості можна побачити в прикладі вставки нового елемента зі збереженням упорядкованості при використанні мови програмування *C*++ [6].

```
// Вставка до циклічного списку зі збереженням порядку.
list *InsSort(list *ph, int v)    // Функція повертає новий заголовок
{ list *q = new list;                    // Новий елемент як єдиний
q–>val = v; q–>next = q–>pred = q;
if (ph == NULL) return q; // Список пустий — повернути новий
list *p = ph;
do { if (v < p–>val) break;    // Місце вставки перед першим,
p = p–>next;                               // більшим заданого, інакше –
} while (p!=ph);  // перед першим в списку (після останнього)
q–>next = p;                      // Наступний за новим = поточний
q–>pred = p–>pred;  // Попередній для нового = попередній
поточного
p–>pred–>next = q;  // Наступний для попереднього = новий
p–>pred = q;            //  Попередній для поточного = новий
if ( ph–>val > v) ph=q;            // Вставка перед першим –
return ph; }                                         // корекція заголовка
```

Пошук місця вставки завершується виявленням першого елемента, більше заданого або поверненням на початок списку. В обох випадках місце вставки перед поточним елементом вибирається коректно: вставка перед першим є вставкою після останнього.

Вставка у двозв'язний список зі збереженням порядку. Програма адекватно реагує на чотири ситуації: вставка до порожнього списку, у початок, у кінець і в середину списку (мова програмування *C*++) [6].

```
// Вставка до двозв'язного списоку зі збереженням порядку
void InsSort(list * &ph, int v)
{ list *q , *p = new list;              // Новий елемент списку
p–>val = v;
p–>pred = p–>next = NULL;
if (ph == NULL) {          // Вставка до порожнього списку
ph = p; return ;
}                                    // Пошук місця вставки — q
for (q = ph; q ! = NULL && v > q–>val; q=q–>next) ;
if (q == NULL)                        // Вставка до кінця списку
{                                  // Відновити покажчик на останній
for (q = ph; q–>next!=NULL; q=q–>next) ;
p–>pred = q;
q–>next = p;
return;
}                                     // Вставити перед поточним
p–>next=q;                 // Наступний за новим = поточний
p–>pred=q–>pred;// Попередній нового = попередній поточного
if (q–>pred = = NULL)      // Вставка до початку списку
ph = p;
else                                 // Вставка в середину
q–>pred–>next = p;         // Наступний за попереднім = новий
q–>pred = p; }              // Попередній поточного = новий
```

3.2.3. Видалення елемента зі списку

Як і при описанні операції вставки, будемо ілюструвати операцію видалення на прикладі однозв'язного списку, використовуючи ті ж позначення. Припустимо, що видаленню зі списку підлягає елемент, що розташований за елементом, що адресується робочим покажчиком *PNTER*. Після видалення елемента допоміжному покажчику *PTR* повинно бути привласнене значення адреси видаленого елемента. Припустимо, крім того, що елемент, який підлягає видаленню, дійсно є в списку (рис. 3.13, *а*) [15].

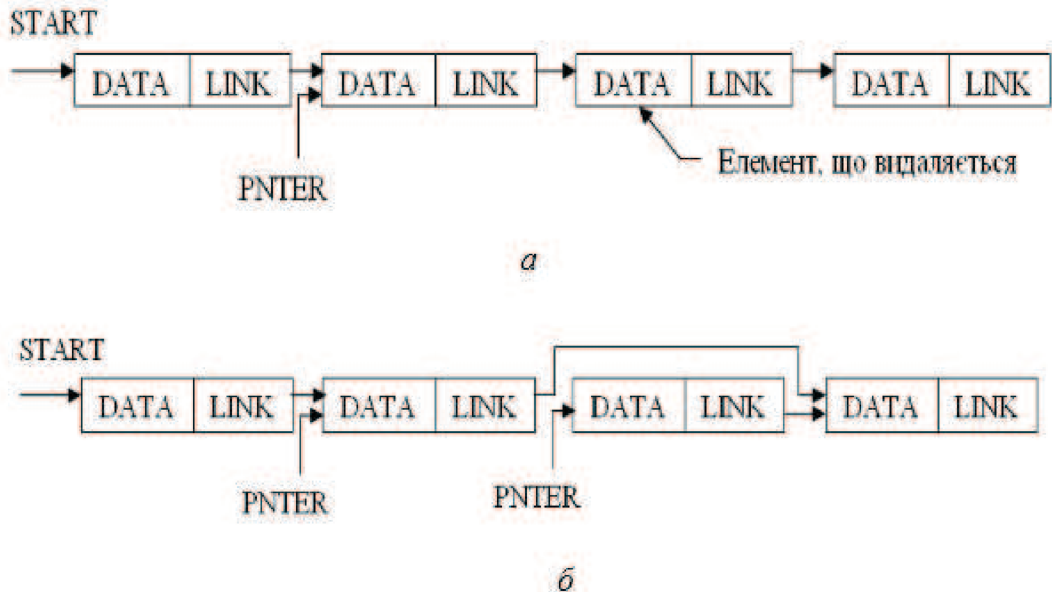


Рис. 3.13. Ілюстрація операції видалення елемента з однозв'язного списку: *а* — до видалення; *б* — після видалення

На першому кроці операції видалення допоміжному покажчику *PNTER* привласнюється вміст поля зв'язки елемента, що адресується покажчиком *PNTER*. На другому кроці, до поле зв'язки елемента, що адресується покажчиком *PNTER* зміниться вміст поля зв'язки елемента, що видаляється.

Відзначимо, що в полі зв'язки вилученого елемента, як і раніше зберігається адреса елемента, який видалили, і цією адресою при необхідності можна скористатися. (рис. 3.13, *б*).

Нижче приведено описи алгоритму видалення елемента з однозв'язного списку елементів, що знаходяться в списку слідом за елементом, що адресується покажчиком *PNTER.* Дані, що зберігаються в полі *DATA* доданого елемента, привласнюються змінній *FIELD,* що має таку ж структуру, що і поле *DATA.* Після цього видалений елемент вставляється в початок вільного списку, що адресується покажчиком *FREE.*

Алгоритм 3.2.

1. [Чи є слідуючий елемент?] Якщо *LINK*(*PNTER*)=, то надрукувати повідомлення «НЕОБХІДНИЙ ЕЛЕМЕНТ ВІДСУТНІЙ» і закінчити виконання алгоритму.

2. [Запам'ятовування адреси елемента, що видаляється.] Встановити *PTR LINK*(*PNTER*).

3. [Модифікація зв'язки в списку.] Вставити *LINK*(*PNTER*) *LINK*(*PTR*).

4. [Переписування даних із видаленого елемента.] Встановити *FIELD DATA*(*PTR*).

5. [Вставка доданого елемента в початок вільного списку.] Встановити *LINK*(*PTR*) *FREE, FREE PTR* і закінчити виконання алгоритму.

У двозв'язних списках обидві операції ускладнюються внаслідок необхідності модифікувати не один, а два покажчики у відповідних елементах.

3.2.4. Пошук елемента в списку

Одна з найчастіше вживаних операцій — пошук в списку елемента із заданим ключем *X.* У відмінності від масивів пошук у цьому випадку повинен проходити послідовно. Він закінчується або при виявленні елемента, або при досягненні кінця списку.

Алгоритм 3.3.

1. [Чи є необхідний елемент?] Якщо (*PNTER*)№*X AND* (*DATA*(*PNTER*)№*X*), *то PNTER LINK*(*PNTR*) і продовжуємо пошук, інакше видати повідомлення «НЕОБХІДНИЙ ЕЛЕМЕНТ ВІДСУТНІЙ» і закінчити виконання алгоритму. В результаті, у покажчику *PNTER* отримаємо адресу шуканого елемента

3.2.5. Способи прискорення пошуку в спискових структурах

Головною перевагою спискових структур є гнучкість і швидкість коригування, тому ці структури ефективно застосовуються до даних, що працюють у режимі коригування. На практиці не зустрічаються такі випадки, щоб дані працювали тільки в режимі коригування або пошуку. Як правило самі дані працюють як у режимі пошуку, так і в режимі коригування. Є способи, що дозволяють прискорити пошук у спискових структурах. До цих способів відноситься гніздовий спосіб упаковування спискових структур, адресна функція, *К*- і *А*- індекси та інше.

Сутність гніздового способу упаковування спискових структур полягає в тому, що декілька елементів об'єднуються в послідовну структуру й утворюють гніздо. А зв'язок між гніздами здійснюється за допомогою адрес зв'язку. При гніздовому способі зберігання списків, так само як і при ланцюговому, виділяється область опису об'єктів, що має аналогічну структуру і призначення. Для організації списку елементів створюється покажчик списку, у якому вказуються кількість робочих гнізд і адреса першого елемента першого гнізда.

Структура списку така, що виділені в гніздо елементи розташовуються послідовно. Останній запис списку є покажчиком переходу на поточне гніздо. Довжина гнізда дорівнює кількості елементів у гнізді плюс одиниця.

Вільна пам'ять, що використовується при коригуванні, також упаковується таким способом. Для організації вільних гнізд

виділяється покажчик, в якому вказується їхня кількість і адреса чарунки першого вільного гнізда. Мінімальна кількість вільних гнізд спочатку дорівнює кількості робочих гнізд.

У зв'язку з тим, що пошук у середині гнізда здійснюється по упорядкованій послідовній структурі, швидкість пошуку в гніздовому списку вище, ніж у ланцюговому. Проте коригування при цьому складніше, і вільна пам'ять займає більший обсяг, тому що кожному робочому гнізду повинно відповідати одне вільне гніздо.

Щоб поліпшити коригування і скоротити обсяг вільної пам'яті, можна скористатися таким прийомом: робочі гнізда упакувати гніздовим способом, а вільну пам'ять — ланцюговим.

Адресна функція — залежність номера елемента від його ключової ознаки [16]:

$$I = f(p),$$

де i — номер (адреса) елемента в пам'яті; p — значення ключової ознаки

Вигляд функції f може бути будь-яким і вибирається при розробці системи пошуку.

Найпростіша адресна функція має вигляд :

$$i=kP–a,$$

де a — константа і для простоти опрацювання береться $a=k\,Pmin–1$, $Pmin$ — мінімальне значення ключової ознаки; k — константа, що вибирається таким чином, щоб була цілим числом.

Якщо можливе дублювання в значеннях елементів, то або для кожного елемента організується лічильник дублів, або елементи, що дублюються, виносяться в резервну зону, де їм задається ланцюгова організація. Можна використовувати інший вид адресної функції:

$$i=\{P/m\},$$

де m — ціле число; $\{P/m\}$ — залишок від ділення P на m.

Для упакування за допомогою цієї адресної функції виділяються дві зони: головна (що має суцільну ділянку) і резервна (що може зайняти будь-які ділянки пам'яті).

Значення m залежить у першу чергу від довжини суцільної ділянки пам'яті для головної зони. Коли вибрано m, це означає, що m номерів можуть займати елементи в головній зоні. Якщо поточний елемент, що претендує на один із номерів, не може бути переміщений у головну зону через зайнятість цієї позиції, то він розміщається в резервну зону. Між елементами, що претендують на ту саму позицію, встановлюється адресний зв'язок.

У результаті упаковування спискової структури за допомогою аналізованої адресної функції утвориться m ланцюжків. Для ефективного пошуку необхідно, щоб довжина всіх ланцюжків була приблизно однакова.

Існує спосіб прискорення пошуку за рахунок *K*- і *A*-індексів. Значення ключів записів, номера яких у рядку утворюють арифметичну прогресію називаються *K*-індексами.

Адреси записів, знаки ключів яких утворюють арифметичну прогресію називаються *A*-індексами.

У списковій структурі, позначеної тими або іншими індексами, пошук ведеться в дві стадії: у масиві індексів і в головній структурі.

При пошуку елемента з ключем q в списковій структурі, що має масив *K*-індексів, знаходиться адреса Ki, така, що $Ki<q<Ki+1$ (i — номер індексу в масиві індексів). Далі організуються вхід у список за адресою, записаною в Ki, і пошук у цій структурі методом перебору.

У випадку А-індексів для шуканого q обчислюється, як

$$I=q/z,$$

де z — інтервал значень ключа для *A*-індексу.

Далі елементи списку проглядаються послідовно, починаючи з адреси, записані в i-му *A*-індексі.

3.3. Приклад організації і представлення лінійних зв'язних списків у пам'яті комп'ютера

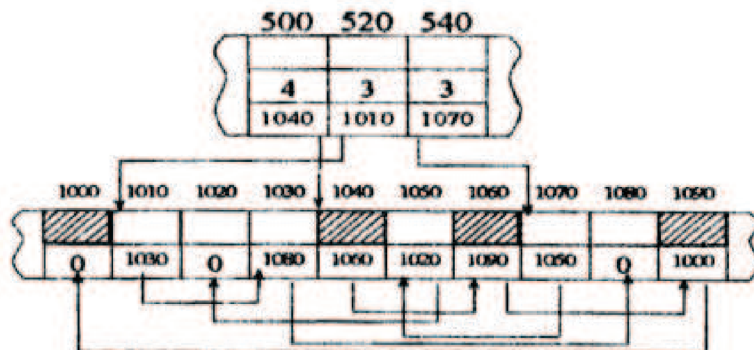
Зупинимося на питаннях організації й представлення лінійних зв'язних списків у пам'яті комп'ютера. З опису алгоритму видалення елемента з лінійного однозв'язного списку, наведеного вище, ми вже бачили, що дана операція логічно призводить до операції вставки видаленого елемента в інший список. Це означає, що при використанні спискових структур у системі створюється не менше двох списків, один з яких (функціональний список) призначається для збереження тієї або іншої корисної інформації, а інший містить вільні елементи. У системі можуть існувати одночасно багато списків. Якщо елементи усіх функціональних списків мають ту саму структуру і розмір, то на всю групу цих списків достатньо мати один вільний список. В іншому випадку прийдеться створювати і підтримувати стільки вільних списків, скільки є різних структур елементів функціональних списків у системі.

Спочатку, як правило, усі списки у системі, за винятком вільних списків, порожні, і кожний займає в пам'яті комп'ютера ділянку, яка необхідна лише для дескриптора. При цьому вільні списки мають максимальну кількість елементів. Кожний вільний список спочатку займає в пам'яті безперервну область. Фізична послідовність елементів у такій області неважлива, але, зазвичай, елементи вільного списку спочатку розташовуються у відведеній їм області в тому ж фізичному порядку, в якому вони пов'язані в список. Поля дескриптора вільного списку мають значення, обумовлені адресою першого елементу списку, кількістю елементів у списку й іншими розмірами, що включаються в дескриптор.

У процесі роботи програмно-інформаційної системи, в яку входять списки, відбувається багатократне перекачування із вільного списку у функціональні списки і навпаки, а також з одного функціонального напрямку в інший. У результаті змінюється кількість і послідовність елементів у різних списках, а в області

пам'яті, що займаються елементами того або іншого списку, можуть з'явитися численні «пробіли», у яких зберігаються елементи інших списків. На рис. 3.14 наведено приклад розподілу пам'яті для трьох однозв'язних списків у момент їхнього створення (рис. 3.14, *а*) і через якийсь час опрацювання (рис. 3.14, *б*).

а



б

Рис. 3.14. Приклад розподілу пам'яті для трьох однозв'язних списків: *а* — у момент створення списків; *б* — через якийсь час опрацювання

На рис. 3.14 один зі списків є вільним, а два інших — функціональні, причому елементи всіх трьох списків мають однаковий формат і розмір. Дескриптори цих списків займають по 20 одиниць пам'яті і мають адреси 500, 520, 540. Для простоти в кожному з трьох дескрипторів приведено тільки кількість елементів і значення покажчика першого елементу списку. Потрібне значення покажчика в дескрипторі є ознакою порожнього списку, в момент створен-

ня обидва функціональних списки є порожніми. Нульове значення адреси в полі покажчика того або іншого елемента списку є ознакою кінця відповідного списку. Поля даних в елементах вільного списку заштриховані. З рис. 3.14 зрозуміло, що в момент створення вільний список складається з 10 елементів, послідовно розташованих в пам'яті, розмір кожного елемента дорівнює 10 одиницям пам'яті. Через якийсь час опрацювання список вільних елементів містить чотири елементи, а функціональні списки — по три елемента кожний, причому елементи трьох списків фізично перемішані.

3.4. Ланцюговий спосіб організації збереження спискових структур

У даному підрозділі приводяться задачі по застосуванню ланцюгового способу організації спискових структур. Спосіб організації структури в пам'яті називається структурою зберігання. Ланцюгова структура зберігання списків є основною і, мабуть, єдиною, що зберігає цілісну логіку спискової структури. Ланцюгова структура зберігання може бути задана як простому, так і складному списку, що має ряд підсписків.

У загальному випадку при ланцюговому способі організації списку виділяють дві області в пам'яті: область опису об'єктів списку і спискову область.

Область опису об'єктів складається із записів, що цілком характеризують цей об'єкт, яким привласнені комп'ютерні найменування. Комп'ютерне найменування включає адресу запису елемента в області опису об'єктів і ключ пошуку. У списковій області організується два ланцюжки:

- ланцюжок об'єктів, кожний елемент містить його комп'ютерне найменування й адресу поточного елемента;

- ланцюжок вільних елементів пам'яті, кожний з яких містить адресу вибраного елемента.

Таким чином у списковій області насамперед повинні бути організовані два покажчики: покажчик списку і покажчик вільних елементів пам'яті. У покажчику списку вказуються кількість об'єктів у даному списку і адреса комп'ютерного найменування першого елемента. У покажчику вільних елементів пам'яті задаються кількість вільних елементів і адреса першого вільного елемента пам'яті.

Приклад 3.5. Нехай дана сукупність елементів, що складається із 6 записів фіксованої довжини по 20 слів кожна. Виділимо область опису об'єктів з адреси 1000 (табл. 3.3).

Таблиця 3.3.

Комп'ютерна назва елементу		Опис елемента
Адреса	Ключ пошуку	
1000	$K1$	Запис 1
1020	$K2$	Запис 2
1040	$K3$	Запис 3
1060	$K4$	Запис 4
1080	$K5$	Запис 5
1100	$K6$	Запис 6

Спискова область почнеться з адреси a+0 і буде мати такий вигляд (табл. 3.4). Адреса a+0 буде покажчиком списку, а a+1 — покажчиком списку вільних елементів пам'яті.

Таблиця 3.4.

Адреса	Комп'ютерна назва	Адреса зв'язку
a+0		
a+1		
a+2		
a+3		
a+4		
a+5		
…		
a+10		

Комп'ютерні найменування елементів можуть розташовуватися в списковій області довільно. Логічний порядок задається за допомогою адрес зв'язку. Оскільки в нас 6 елементів, то в покажчику списку вказується це число й адреса розташування машинного найменування першого елемента списку (a+4) (табл. 3.5).

Таблиця 3.5

Адреса	Комп'ютерна назва		Адреса зв'язку
a+0	6		a+4
a+1	3		a+5
a+2	1060	K4	a+10
a+3	1100	K6	KC1
a+4	1000	K1	a+6
a+5			a+7
a+6	1020	K2	a+8
a+7			a+9
a+8	1040	K3	a+2
a+9			KC2
a+10	1080	K5	a+3

В адресах a+5, a+7, a+9 розташовуються вільні елементи пам'яті. Для вставки нового елемента до списку, його опис розташовується в області опису об'єктів. Причому незалежно від логічного порядку проходження цього елемента його опис розташовується наприкінці області опису об'єктів.

Для вставки до спискової області нового об'єкта обирається будь-який вільний елемент пам'яті. Для цього в покажчику списку кількість вільних елементів зменшується на одиницю, в адресу зв'язку попереднього вільного елемента заноситься адреса, що стояла в адресі зв'язку, елемента який видаляють. Якщо елемент був перший, то адреса його зв'язку поміщається в покажчик вільних елементів пам'яті.

В обраний вільний елемент пам'яті заноситься комп'ютерне найменування нового елемента списку, а адресу зв'язку знаходять таким чином: якщо новий елемент вставляється в початок списку, то в його адресу зв'язку записується значення адреси зв'язку, взяте з покажчика списку, а на адресу зв'язку покажчика списку записується адреса, у якій розміщено новий елемент. Якщо новий елемент вставляється в середину списку, то спочатку знаходять попередній елемент списку, після якого потрібно додати новий елемент, а потім проводиться заміна адреси зв'язку: у попередньому елементі ставиться адреса розташування комп'ютерного найменування нового елемента, а в новому елементі ставиться адреса, взята з адреси зв'язку попереднього елемента.

Приклад 3.6. Нехай у вихідну сукупність необхідно вставити елементи 7 і 8, причому логічний порядок елементів повинний бути таким: 8, 1, 2, 3, 7, 4, 5, 6. Область опису об'єктів буде мати вигляд (табл. 3.6.).

Таблиця 3.6.

Комп'ютерна назва елементу		Опис елемента
Адреса	Ключ пошуку	
1000	*K*1	Запис 1
1020	*K*2	Запис 2
1040	*K*3	Запис 3
1060	*K*4	Запис 4
1080	*K*5	Запис 5
1100	*K*6	Запис 6
1120	*K*7	Запис 7
1140	*K*8	Запис 8

Після першого коригування, тобто після вставки елемента 7, спискова область буде мати вигляд (табл. 3.7).

Таблиця 3.7.

Адреса	Комп'ютерна назва		Адреса зв'язку
a+0	7		*a*+4
a+1	2		*a*+7
a+2	1060	*K*4	*a*+10
a+3	1100	*K*6	*KC*1
a+4	1000	*K*1	*a*+6
a+5	1120	*K*7	*a*+2
a+6	1020	*K*2	*a*+8
a+7			*a*+9
a+8	1040	*K*3	*a*+5
a+9			*KC*2
a+10	1080	*K*5	*a*+3

Машинне найменування елемента 7 перенесено в адресу *a*+5. Машинне найменування елемента 8 перенесено в адресу *a*+9. Після другого коригування, спискова область буде мати вигляд (табл. 3.8.).

Таблиця 3.8.

Адреса	Комп'ютерна назва		Адреса зв'язку
a+0	7		*a*+9
a+1	2		*a*+7
a+2	1060	*K*4	*a*+3
a+3	1100	*K*6	*KC*1
a+4	1000	*K*1	*a*+6
a+5	1120	*K*7	*a*+2
a+6	1020	*K*2	*a*+8
a+7			*a*+10
a+8	1040	*K*3	*a*+5
a+9	1140	*K*8	*a*+4
a+10			*KC*2

Процес видалення елемента зі списку здійснюється також шляхом заміни адреси зв'язку, де знаходиться елемент, що передує видаленню (для першого елемента це буде покажчик списку), і в

ньому адреса зв'язку замінюється на іншу адресу зв'язку. Одночасно відбувається зменшення кількості елементів у покажчику списку. Адреса, по якій розташовувався видалений елемент, оголошується вільною і вставляється в список вільних елементів пам'яті.

Приклад 3.7. Нехай у розглянутому вище списку необхідно вставити п'ятий елемент. Спискова область після коригування буде мати вигляд (табл. 3.9.).

Таблиця 3.9.

Адреса	Комп'ютерна назва		Адреса зв'язку
a+0	8		a+9
a+1	1		a+7
a+2	1060	*К*4	a+10
a+3	1100	*К*6	*КС*1
a+4	1000	*К*1	a+6
a+5	1120	*К*7	a+2
a+6	1020	*К*2	a+8
a+7			*КСП*
a+8	1040	*К*3	a+5
a+9	1140	*К*8	a+4
a+10	1080	*К*5	a+3

Контрольні запитання та завдання для самоконтролю

1. Спискова структура задана таким аналітичним виразом: $(a,(b,c,(d,e,(k)),d(a,\ c),b))$. Задати аналітичну і графічну інтерпретацію цього списку в однозв'язному, двозв'язному виг-лядах.

2. Спискова структура задана таким аналітичним виразом: $(((b),(a,((),(b,a,())),(a)),a))$. Потрібно задати аналітичну і графічну інтерпретацію цього списку в однозв'язному, двозв'язному кільцевих виглядах.

3. В умовах задачі 1 потрібно скоригувати спискову структуру для одержаної послідовності $(a,(b,c,(d,e,(k)),d(a,c),b))$ про $(a,(b,m,c,(d,k,(e,k)),d,(a,b,m)))$ про $(a,c,(b,k,(d,c),a,c,m)))$.

4. В умовах задачі 2 потрібно скорегувати спискову структуру для одержаної послідовності $(((b),(a,((\ ),(b,a(\ ))),(a)),a))$ про $(((b,a),(b,((a,(b)),((a,(b)),(a,(b))),(b,(a))),b)$.

5. Нехай 10 студентів отримали екзамени з ІС з такими оцінками:

Іванов — 3;
Петров — 4;
Сидорів — 5;
Волков — 5;
Морозова — 2;
Маслова — 4;
Дурова — 5;
Тарасинський — 2;
Дубова — 5;
Чорна — 3.

Потрібно задати спискову структуру заданого набору даних; внести необхідні корективи, виходячи з того, що студентка Морозова перездала іспит на 4, а студента Тарасинського виключили за неуспішність.

6. Потрібно задати спискову структуру для елементів предметного каталогу бібліотечного фонду спискової структури.

Рубрикація предметного каталогу така:

1. Технічне забезпечення АСУ
   1.1. Електронні обчислювальні машини
   1.2. Периферійне устаткування
2. Інформаційне забезпечення АСУ
   2.1. Організація інформації в АСУ
   2.2. Характеристики інформаційних даних
   2.3. Методи доручення первинної інформації
   2.4. Носії інформації

3. Математичне забезпечення АСУ
   3.1. Операційні системи
   3.2. Бібліотеки стандартних програм
   3.3. Мови програмування
   3.3.1. FOXPRO
   3.3.2. PASCAL
   3.3.3. Фортран
4. Прикладні програми

7. Дано сукупність елементів 3, 8, 1, 4, 9, 5, 7, 1, 3. Потрібно задати спискову структуру для упорядкованої сукупності цих елементів, упаковану ланцюговим способом. Спискову область почати з адреси 150. Здійснити коригування структури за рахунок вставки елементів 6, 0, 2, видалення елемента 5.

8. Дано набір даних, що складається з 9 записів фіксованої довжини. Довжина кожного запису 10 слів. Потрібно задати однозв'язну спискову структуру, упаковану ланцюговим способом, за умови, що адреса спискової області — 160, а області опису об'єктів — 1500. Здійснити коригування шляхом вставки трьох нових елементів, відповідно після четвертого елемента, перед першим елементом і після шостого елемента. Видалити третій і перший елементи.

9. В умовах задачі 8 задати двозв'язну спискову структуру, упаковану ланцюговим способом, і здійснити відповідне коригування.

10. Дано складний список такого вигляду:

$$((( ),(b,a,( )))),B,(A,(( ),B))).$$

Потрібно задати графічне представлення однозв'язного кільцевого списку.

11. Дано список такої структури:

$$((B,(A,( ),B),(B,A)), A(B(A,( )))).$$

Потрібно задати графічне представлення однозв'язного списку і здійснити його коригування з метою отримання списку такої структури:

$$(A,(A,(B)),(B,A)),(A,B),(B(A,(\ )))).$$

12. Дано список такої структури:

$$(A,(B(K,(B,A(\ )),A),(\ )),A))).$$

Потрібно задати графічне представлення однозв'язного і двозв'язного списку і здійснити коригування з метою отримання списку такої структури:

$$((K,A),(B(A,(K,A,(B)),A),((K,C)),A))).$$

13. Побудувати спискову структуру в пам'яті ЕОМ (графічно) таким чином, щоб в чарунках 31 — 36 розмістилися записи «топот» і «потоп», які можна було б читати як у прямому, так і в зворотному напрямках.

14. Побудувати в пам'яті ЕОМ спискову структуру: 112, 4, 82 — таким чином, щоб можна було б читати її як у прямому, так і в зворотному напрямках. В робочому полі вільні чарунки з умовними адресами 10, 11, 12, 13.

15. Побудувати структурні схеми однозв'язного і двозв'язного списку для розміщення в них набору чисел (1, 2 (3, 4, 5) 6), якщо для розміщення дано чарунки з умовними адресами 101 — 108.

16. Влаштувати структурну схему, що дозволяє читати список $(AB)C(BA)$ у прямому й зворотному порядку.

17. До ЕОМ увели деякий набір даних, що є списком матеріалів такого вигляду:

Металеві	Дерев'яні	Щебінь
1. Арматура	1. Дошки	
2. Балки	2. Планки	

Показати розміщення цих даних у вигляді двозв'язного списку, якщо для розміщення надано чарунки з умовними адресами у такій послідовності: 215, 181, 10, 37, 84, 85, 86, 87.

18. Набір даних — р, т, о, розмістити в пам'яті ЕОМ у вигляді кільцевого двозв'язного списку, якщо для розміщення надано чарунки з умовними адресами у такій послідовності: 171, 937, 560, 456, 766. Організувати слово «ротор».

19. У чарунках з умовними адресами 2, 4, 7, 12, 23, 24 зберігаються відповідні дані: П, Р, Т, В, І, И. Побудувати структурну схему кільцевого однозв'язного списку, що дозволяє читати слово «ПРИВІТ».

20. Побудувати структурну схему кільцевого однонапрямного списку для розміщення у ньому слова «ЗАЛИШОК», якщо для розміщення надано чарунки з умовними адресами 1, 12, 123, 2, 23, 3, 32, 321. Показати процес отримання з даного списку слова «ЗАЛИШОК».

21. За допомогою ілюстрації логічної структури двозв'язних списків пояснити процес вставки і видалення елемента. Написати алгоритм.

22. За допомогою ілюстрації логічної структури однозв'язних кільцевих списків пояснити процес вставки і видалення елемента. Написати алгоритм.

23. За допомогою ілюстрації логічної структури двозв'язних кільцевих списків пояснити процес вставки і видалення елемента. Написати алгоритм.

24. Побудувати структурну схему однозв'язного списку запису «ПАРКОВИЙ» і показати процес заміни елементів отриманого списку для того, щоб отримати запис «ПАРНИЙ». Дано П, А, Р, К, О, В, И, Й, Н зберігаються відповідно в чарунках з умовними адресами від 30 до 38.

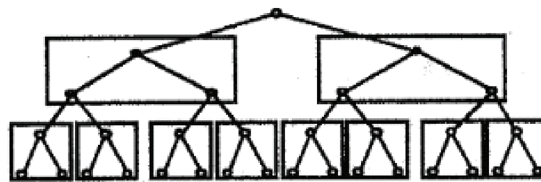
Розділ 4. БІНАРНІ ДЕРЕВА, *B*-ДЕРЕВА

4.1. Пошук по дереву в зовнішній пам'яті

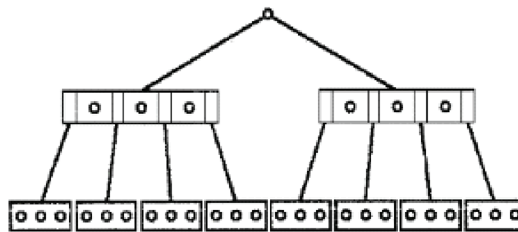
Специфікою інформаційних систем є те, що, по-перше: кількість даних надзвичайно велика, в той час, як частота використання окремих даних невисока, по-друге: дані зберігаються не в оперативній, а в зовнішній пам'яті. Для пошуку даних, що зберігаються в зовнішній пам'яті, дуже важливою є проблема скорочення кількості переміщень даних із зовнішньої пам'яті в оперативну.

Наприклад, бінарні дерева, які часто використовуються, виявляються більш корисними для пошуку в зовнішній пам'яті, через те, що їхня висота менша, отже при пошуку буде потрібно менше звертань до зовнішньої пам'яті. Кількість звертань до зовнішньої пам'яті можна зменшити й у випадку бінарного дерева, якщо в один блок зовнішньої пам'яті помістити декілька вузлів, розташованих на різних рівнях, як показано на рис. 4.1, *а*. Природно розглядати такі структури, як різновид дерев, що сильно розгалужуються, (рис. 4.1, *б*) [12].

Навіть у тому випадку, коли в опрацюванні беруть участь тільки окремі дані, що зберігаються в зовнішній пам'яті, більш швидким вважається не прямий доступ до «розкиданих» по пам'яті даних, а послідовний доступ до суміжних ділянок. Тому не дивно, що послідовний доступ намагаються використовувати всюди, де це можливо. Але упорядкованість даних у бінарному дереві погано відповідає порядку їхнього розташування по адресах пам'яті. В сильно розгалужених деревах, упорядкованість ключів у дереві більше відповідає їхньої упорядкованості у просторі адрес, тому що принаймні в кожному вузлі ключі розташовуються в порядку зростання їхніх значень.



а



б

Рис. 4.1. Схема дерев: *а* — розташування данних на різних рівнях дерева; *б* — дерево, що сильно розгалужено

Звідси можна зробити висновок, що для організації пошуку в зовнішній пам'яті вибір того або іншого типу дерева залежить від особливості зовнішньої пам'яті і мети пошуку. Розглянемо порівняно нещодавно розроблені *B*-дерева, що вже знайшли широке використання.

4.2. *B*–дерева

4.2.1. Визначення і пошук

Визначення дерева має винятково рекурсивну природу. Будь-який елемент цієї структури даних називається вершиною. Дерево є або окремою вершиною, або вершиною, що має обмежену кількість зв'язків з іншими деревами (гілками). Дерева, що розташовані нижче для поточної вершини називаються піддеревами,

а їх вершини — нащадками. По відношенню до нащадків поточна вершина називається предком (рис. 4.2). Вершина дерева — структурована змінна, що містить деяку кількість (окремі змінні, масив, динамічний масив) покажчиків на нащадків [11].

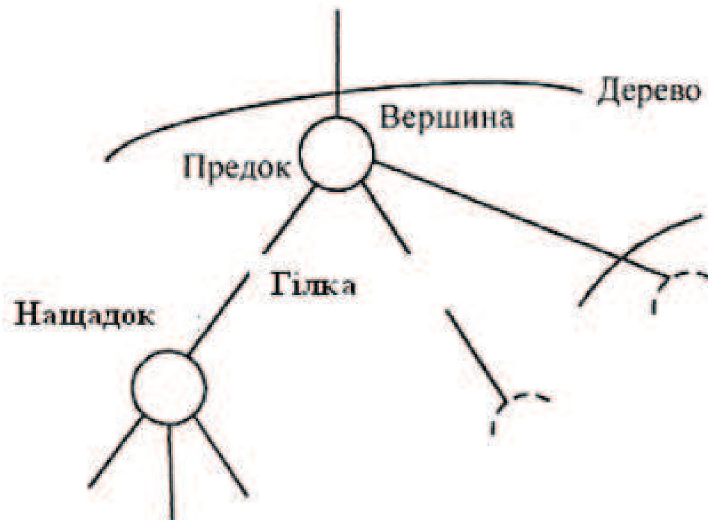


Рис. 4.2. Структура дерева

В мові програмування *C++* для роботи з деревами необхідно використовувати інструкцію *tree*, наступний фрагмент програми ілюструє створення дерева [6]:

```
struct tree{
int val;                    // Значення в вершині дерева
tree *p[4];             // Масив покажчиків на нащадків
};                          // (обмежена кількість)
```

При визначені доцільності використання структури дерева необхідно усвідомлювати відмінність дерев від списків і масивів, їх недоліки і переваги. Достоїнство списку — локальність здійснюваних у ньому змін, а саме: при вставці/виділенні елемента задіються тільки його сусіди, при цьому їх розташування у пам'яті не зміниться (рис. 4.3, *а*). Масиви (масиви покажчиків), навпаки, потребують у цьому випадку масового переміщення елементів

(рис. 4.3, *б*). Основна вада списку — винятково послідовний доступ. Деревоподібна структура даних забезпечує відомий компроміс: зміни у неї мають властивість локальності, а доступ хоча й не прямій, але принаймні логарифмічний (при заміні алгоритмів повного обходу дерева вибором однієї з його гілок) (рис. 4.3, *в*).

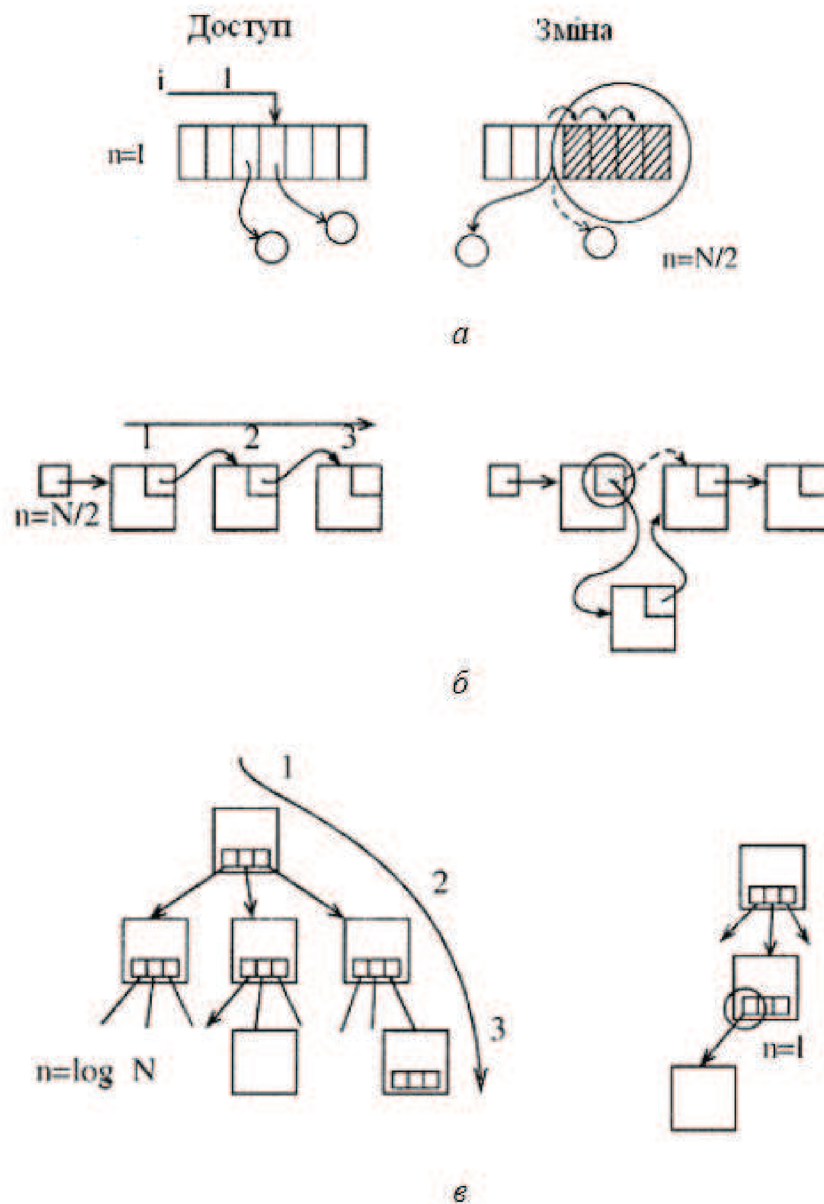


Рис. 4.3. Порівняння структур даних: *а* — масив; *б* — список; *в* — дерево

B-дерево має більш складну структуру, ніж бінарне дерево пошуку. Відомо декілька різновидів цієї структури, що поширює діапазон практичного використання *B*-дерев. Насамперед, дамо визначення головної структури *B*-дерева. Розшифровування символу «*B*» у назві дерева неоднозначне, це перший символ англійського слова «*balanced*» (збалансований). Дана структура розроблена в Науково-дослідній лабораторії фірми «Боїнг», авторами *B*-дерева є Р. Бейер і Є. Мак-Крейт, що і дали їй цю назву. *B*-деревом порядку n називається сильно розгалужене дерево ступеня $2n+1$, що володіє такими властивостями:

1. Кожний вузол, за винятком кореня, містить не менше n і не більше $2n$ ключів.

2. Корінь містить не менше одного і не більше $2n$ ключів.

3. Всі листи розташовані на одному рівні.

4. Кожен нелистовий вузол містить кількість покажчиків на одиницю більше кількості ключів.

5. Кожен нелистовий вузол містить два списки: упорядкований по зростанню список і список покажчиків, що відповідає йому, (для листових вузлів список покажчиків відсутній).

На рис. 4.4 показано приклад *B*-дерева порядку n=2.

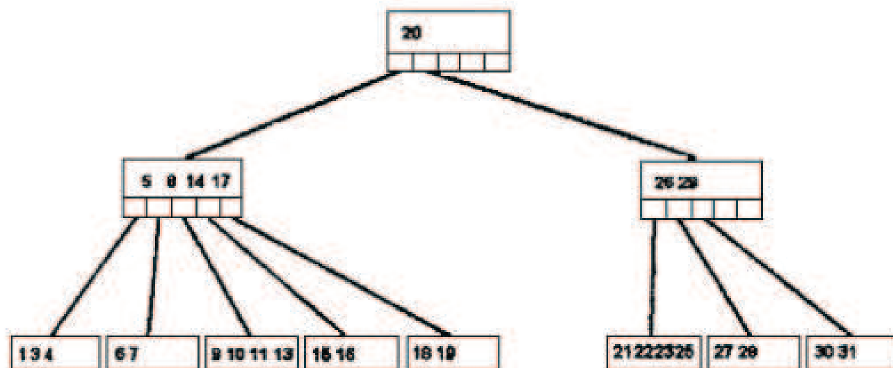


Рис. 4.4. Приклад *B*-дерева порядку n=2. (Для наочності використані числові значення із діапазону 1-31. Кількість ключів у корені від 1 до 4. Кількість ключів в інших вузлах від 2 до 4)

Нижче буде показано, що структура *B*-дерева має сенс для організації пошуку в зовнішній пам'яті тільки при великих значеннях порядку дерева, а дерева (рис. 4.4) більше підходять для пошуку в оперативній пам'яті. *B*-дерево порядку n=2 вибрано в якості прикладу тільки для зменшення і спрощення рисунка.

Оскільки *B*-дерево сильно розгалужується, його висота невелика, тому і кількість звернень до зовнішній пам'яті при пошуку також невелика, окрім того:

1) порівняно просто може бути реалізовано послідовний доступ, оскільки листи розташовані на одному рівні;

2) при додаванні і видалені ключів зміни обмежуються, як, правило, одним вузлом, а не торкаються всієї структури дерева, оскільки в усіх вузлах є вільні поля, максимальне з яких, за виключенням кореня, складає $n=(2n)/2$;

3) Область зміни є дуже вузькою, навіть у випадку, коли зміни торкаються всієї структури дерева, тому що ділянка, на якій відбуваються зміни, розташовується безпосередньо між нащадком і його предком. Щоб переконатися в справедливості перерахованих переваг структури *B*-дерева, розглянемо операції пошуку, вставки і видалення.

Розглянемо, як залежить кількість ключів *B*-дерева від його порядку n і висоти h. Знайдемо спочатку максимальну кількість ключів *NMAX*. Максимальною кількістю вузлів такого дерева буде [16]:

$$1+(2n+1)+(2n+1)^2+...+(2n+1)^{h-1}=\{(2n+1)^{h-1}\}/2n.$$

Оскільки максимальне число ключів у кожному вузлі дорівнює $2n$, то

$$NMAX=(2n+1)^{h-1}.$$

Щоб знайти мінімальну кількість ключів дерева *NMIN*, треба знати мінімально можливу кількість вузлів такого дерева. Без урахування кореня воно матиме вигляд:

$$2 + 2(n+1) + 2(n+1)^2 + ... + 2(n+1)^{h-2} = 2\{(n+1)^{h-2}\}/n.$$

Оскільки мінімальне можлива кількість ключів у вузлі дерева дорівнює n, то очевидно, що

$$NMIN = 2(n+1)^{h-2}.$$

Отже, для будь-якого B-дерева висоти h будь-який ключ може бути знайдений не більш ніж за h кроків пошуку. І навпаки, B-дерева, що містить N ключів, буде мати висоту h із діапазону

$$\log 2n + 1(N+1) \le h \le \log n + 1(N+1)/2.$$

При великих значеннях N і n межі діапазону h майже рівні $\log 2nN$ та $\log nN$. Якщо при пошуку аргументу по B-дереву висоти h витрати (час) на пошук у корені позначити через Cr, витрати на доступ до кожного з вузлів, за винятком кореня, і на пошук у вузлах — через C, а ймовірність переходу пошуку з (i–1)-го рівня на t — рівень — через pi, то загальні витрати на пошук Ct будуть виражатися як:

$$Ct = Crp1 + C(p1 + p2 + ... + ph);\ 1 = p1 > p2 > .. > ph,$$

але, оскільки загальна кількість ключів, що зберігаються у B-дереві від кореня до (i–1)-го рівня, відноситься до кількості ключів, що зберігаються на 1-му рівні, як число, укладене між n і $2n$, до одиниці, пошук у більшості випадків продовжується до листів. Таким чином, $pi \approx 1$, і можна вважати, що:

$$Ct = Cr + C(h-1).$$

При великих значеннях N верхня межа витрат на пошук визначається як

$$Ct \approx Cr + \log n + 1(N+1)/2.$$

Здається, що чим більше n, тим краще. Проте в дійсності з ростом n пропорційно ростуть витрати часу пошуку в вузлах, а якщо n перевищить визначений розмір, то можуть різко зрости

витрати часу і на доступ до піддерев. Таким чином, висоту дерева не варто робити близькою до двох за рахунок не обмеженого збільшення ступеня *n*.

При визначенні мінімальної (максимальної) довжини гілок дерева кожна вершина повинна одержати значення мінімальних довжин гілок від нащадків, вибрати з них найменшу й передати предковий результат, збільшивши його на 1 — «додавши себе».

Наступна програма, реалізована на мові програмування *C++*, демонструє роботу з деревом [6]:

```
struct tree{
int val;
tree *p[4];
};
// Визначення гілки мінімальної довжини
int MinLnt(tree *q){
if (q = = NULL) return 0;
int min= MinLnt(q~>p[0]);
for (int i = 1; i<4; i ++){
int x=MinLnt(q–>p[i]);
if (x < min) min=x;}
return min + 1;}
```

Для відстеження процесу «занурення» досить додаткової змінної, яка зменшує своє значення на 1 при черговому рекурсивному виклику, наприклад на мові програмування *C++* [6]:

```
// Вставка вершини до дерева на задану глибину
int Insert(tree *ph, int v, int d) { // d — поточна глибина вставки
if (d == 0) return 0;                    // Нижче не розгядати
for ( int i=0; i<4; i++)
if (ph–>p[i] == NULL){
tree *pn = new tree;
pn–>val=v;
for (int j = 0; j<4; j ++) pn –>p[i] = NULL;
```

```
ph–>p[i] = pn;
return 1; }
else
if (Insert(ph–>p[i], v , d–1)) return 1;        // Вершину додано
return 0; }
```

При виявленні в дереві першого значення, що задовольняє умові, необхідно перервати не тільки поточний крок рекурсії, але й усі попередні. Тому цикл рекурсивного виклику переривається відразу ж, як тільки нащадок поверне знайдене у піддереві значення. Поточна вершина повинна «ретранслювати» отримане від нащадка значення до власного предка («нагору по інстанції»), що показано в наступній програмі реалізованій за допомогою мови програмування *C++* [6]:

```
// Пошук в дереві строки, довжиною більше заданої
struct stree{
char *str;
stree *p[4];};
char *big_str(stree *q){
if (q==NULL) return NULL;
if (strlen(q–>str)>5) return q–>str;    // Знайдено в поточній
вершині
for (int i=0; i<4; i++){
char *child=big_str(q–>p[i]);               // Отримання
строки від нащадка
if (child!=NULL) return child;              // Повернути її
}
return NULL;}                // Немає ні у себе, ні у нащадків
```

Для здійснення пошуку максимального (мінімального) значення, що міститься у дереві проводиться повний обхід дерева, у кожній вершині — стандартний контекст вибору мінімального з поточного значення в вершині й значень, отриманих від нащадків при рекурсивному виклику функції. На мові програмування *C++*

пошук максимального значення в дереві може бути здійснено наступним чином:

```
// Пошук максимального в дереві
int GetMax(tree *q){
if (q==NULL) return – 1 ;
int max=q–>val;
for (int i=0; i<4; i++){
int x=GetMax(q–>p[i]);
if (x > max) max=x;}
return max;}
```

4.2.2. Вставка і видалення ключів, послідовний доступ

Для виконання операції вставки ключа у *B*-дерево необхідно:

1. Переконатися в тому, що аргумент пошуку не збігається з жодним із вже наявних у дереві ключів. Для цього необхідно послідовно переглядати вузли дерева, поки не буде досягнуто відповідний лист.

2. Якщо лист заповнено не повністю, виконати вставку аргументу у відповідне місце списку ключів і на цьому завершити операцію вставки (рис. 4.5).

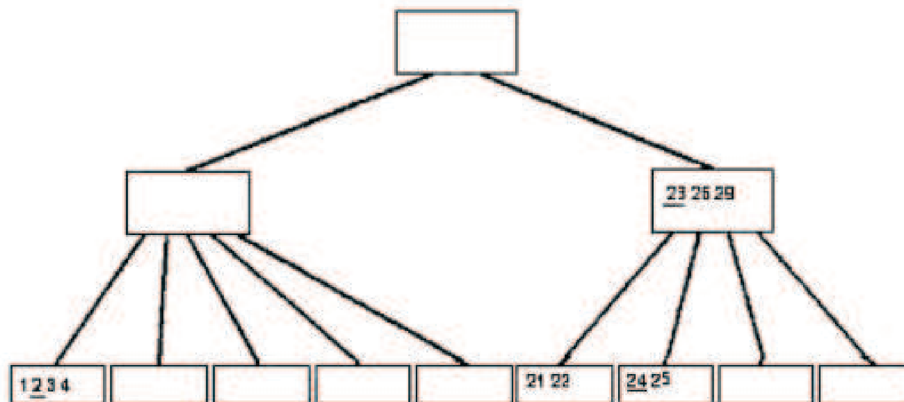


Рис. 4.5. Результат вставки ключів 2 і 24 у *B*-дерево

На рис. 4.5 показані тільки ключі, що мають відношення до процедури вставки. Жирними лініями позначені межі вузлів у місці розщеплення, підкреслено вставлені ключі.

3. Якщо в обраному листі вільного місця немає, то в результаті вставки кількість ключів дорівнюватиме $2n+1$ і виникне ситуація переповнення.

При переповненні: $(n+1)$-й (середній ключ списку) видалити з вузла; створити новий вузол (лист) і перемістити в нього n ключів списку з вузла, у якому виникло переповнення; видалений ключ вставити у вузол-корінь переповненого вузла. Таким чином, при переповненні відбувається розщеплення вузла: утворюються два нових замість одного старого (рис. 4.5) і ключ переносу.

4. Якщо в результаті вставки ключа переносу внутрішній вузол виявиться переповненим, він теж розщеплюється (при цьому половина списку ключів і списку покажчиків розташовується в новому вузлі) (рис. 4.6).

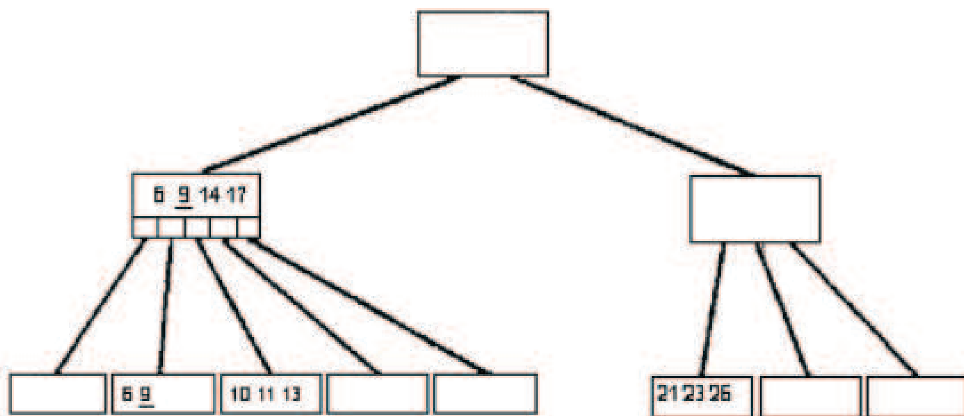


Рис. 4.6. Результат вставки ключа 12 у В-дерево

На рис. 4.6 показано тільки ключі, що мають відношення до вставки. Жирними лініями позначені межі вузлів у місці розщеплення, підкреслено вставлений ключ і ключ переносу.

5. Якщо ключ переносу досягне кореня і корінь після вставки також виявиться переповненим, його теж розщеплюють, що призводить до збільшення висоти дерева на одиницю і створення нового кореня, куди поміщається ключ переносу. Отримані в результаті розщеплення два вузли, стають коренями піддерева нового кореня.

Для виконання операції видалення ключа необхідно виконати такі дії:

1) Знайти вузол, що містить ключ, який повинен бути віддалений. Подальші дії залежать від того, чи міститься ключ, який повинен бути видалений, у листі або в не листовому вузлі.

2) Випадок, коли ключ, що повинен бути видалений, знаходиться в листі.

2.1) Якщо знайдений лист містить не менше (n+1) ключів, то ключ видаляють і операцію закінчують (рис. 4.7).

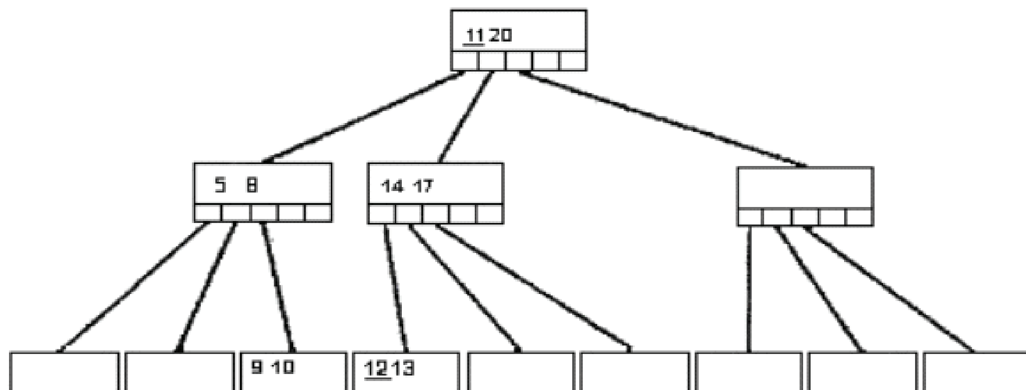


Рис. 4.7. Результат видалення ключів 7 і 22 із *B*-дерева

На рис. 4.7 підкреслено ключі, переміщені для відновлення збалансованості після видалення.

2.2) Якщо в знайденому листі міститься рівно n ключів, то для зберігання рівноваги достатньо виконати переміщення ключів із сусіднього листа. Для переміщення одного ключа із сусіднього листа необхідно: видалити ключ, що розділяє списки клю-

чів сусідніх листів, із вузла рівня, що передує, і вставити його до листа, що містить ключів менше *n*. На місце видаленого ключа розміщується крайній (лівий або правий) ключ із сусіднього листа (рис. 4.7). У випадку великої нерівноваги кількості ключів у сусідніх листах, для зменшення кількості балансувань дерева при послідовних видаленнях, краще переміщати одночасно максимально можливу кількість ключів (щоб після переміщення кількість ключів у сусідніх листах стала рівною або розмірно рівною) (рис. 4.8).

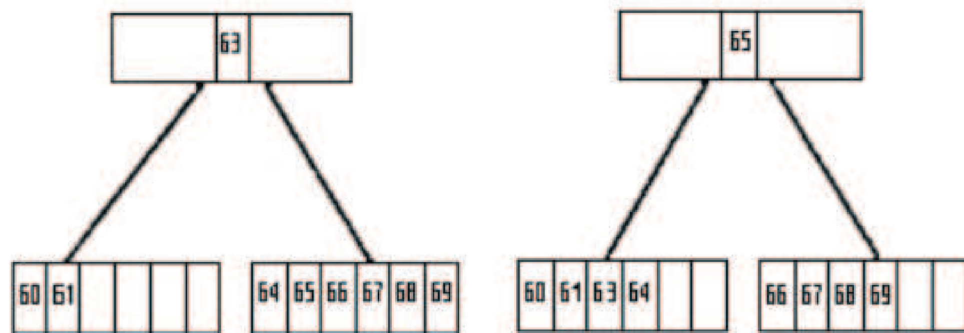


Рис. 4.8. Перерозподіл ключів у випадку сильного порушення балансу після видалення

2.3) Якщо знайдений і сусідній із ним лист містять по *n* ключів, то роблять їх конкатенацію (об'єднання) — списки ключів сусідніх листів об'єднуються і залишається один лист, а інший знищується. При конкатенації листів один із ключів попереднього рівня, «що розділяє» сусідні листи, стає зайвим, і його необхідно перемістити вниз (рис. 4.9). Це явище можна назвати «нестачею» при видаленні. Іноді в результаті «нестачі» у вузлах верхнього рівня кількість ключів може стати меншою за *n*. Тоді для зберігання рівноваги буде потрібно виконати переміщення.

Товстою лінією обведено вузол, отриманий шляхом конкатенації двох вузлів. Підкреслено ключ, видалений із вихідного ключа сусіднього вузла того ж рівня, приклад чого показано на рис. 4.10.

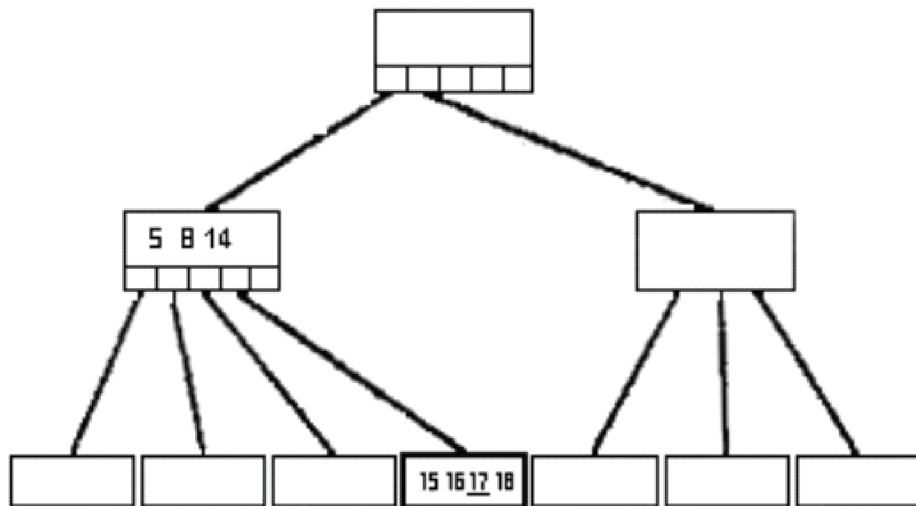


Рис. 4.9. Результат видалення ключа 18 із *B*-дерева, зображеного на рис. 4.4

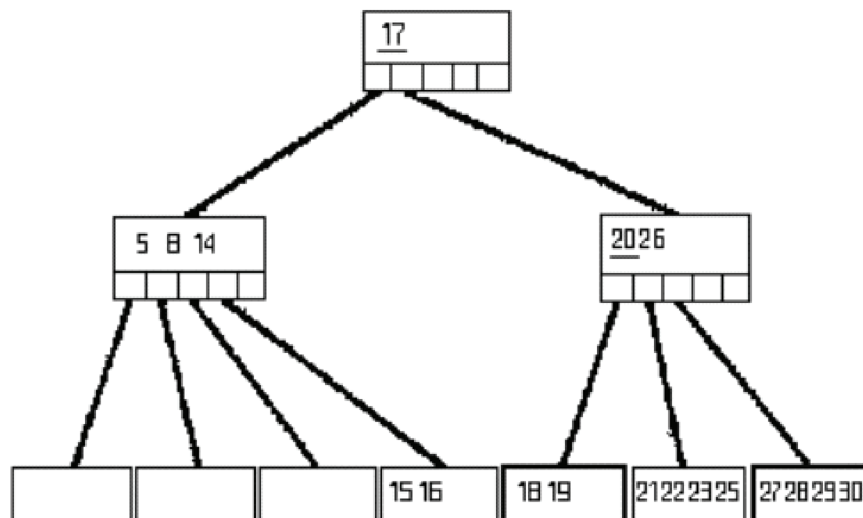


Рис. 4.10. Результат видалення ключа 31 з *B*-дерева, зображеного на рис. 4.4

На рис. 4.10 жирними рамками обведено вузол, отриманий шляхом конкатенації, і вихідний вузол, який змінився. Підкреслено підняті й опущені ключі.

Переміщення може вимагати видалення ключа з вузла більш високого рівня, а видалення у свою чергу призвести до ситуації нестачі на цьому рівні. Якщо цей процес досягне кореня дерева, а після видалення з кореня в ньому не залишиться ключів, то в цьому випадку висота дерева зменшиться на одиницю.

3) Випадок, коли ключ, що повинен бути видалений, знаходиться в не листовому вузлі .

3.1) Видаляємо заданий ключ і поміщаємо на його місце ключ, що є безпосередньо наступним за видаленим. Вузол, що містить ключ, наступний за видаленим, буде самим лівим листом правого піддерева видаленого ключа. Щоб знайти цей вузол, спускаємося вниз на один рівень по «правому» покажчику видаленого ключа, і якщо це лист, то пошук закінчено; у протилежному випадку, використовуючи самий лівий покажчик поточного вузла, спускаємося вниз по дереву доти, поки не буде досягнуто листа. Тепер видаляємо з листа самий лівий ключ списку.

3.2) Наступні дії будуть аналогічні видаленню ключа з листа. На рис. 4.11 показано випадок простого видалення ключів, а на рис. 4.12 — випадок, коли при видаленні ключа потрібно додаткове балансування дерева.

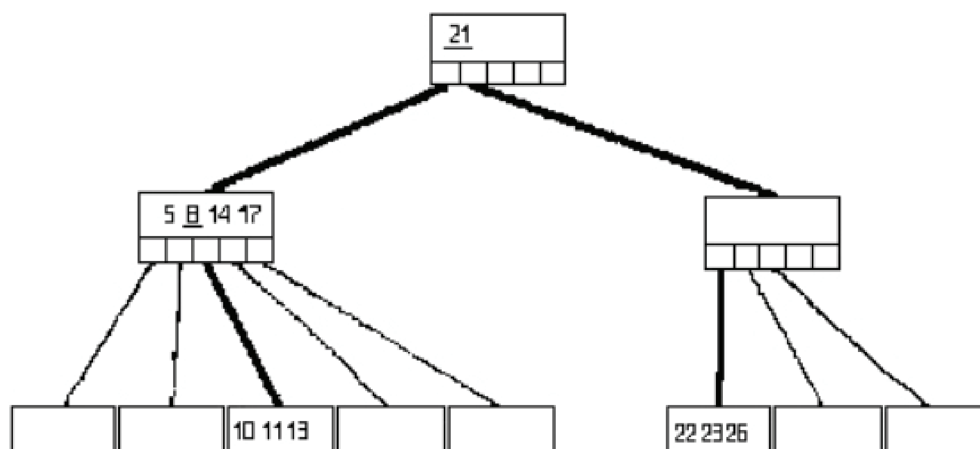


Рис. 4.11. Результат видалення ключів 8 і 20 із *B*-дерева, зображеного на рис. 4.4

На рис. 4.11 жирними лініями показані шляхи пошуку ключа, що розташований за видаленим. Підкреслено підняті ключі.

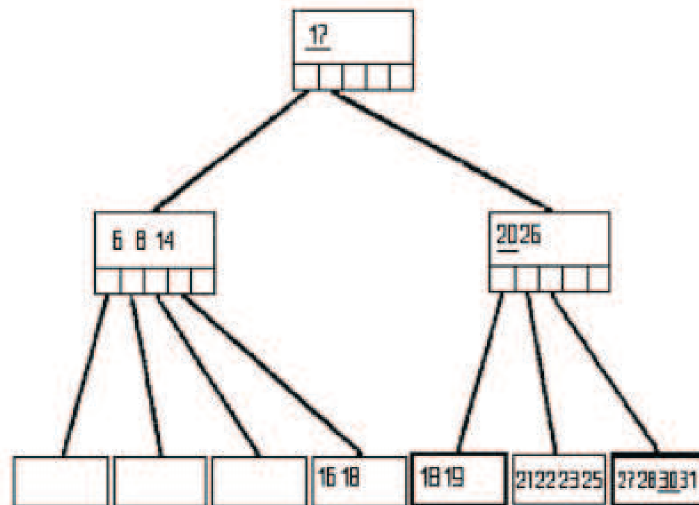


Рис. 4.12. Результат видалення ключа 29 із B-дерева, зображеного на рис. 4.4

Випадок, зображений на рис. 4.12, майже аналогічний випадку, поданому на рис. 4.10, за винятком того, що над ключом 30 виконується більш складне переміщення.

Нерідко потрібно виконати опрацювання всіх ключів B-дерева із заданого діапазону в порядку зростання їх значень. Типовий приклад такого опрацювання — надрукувати упорядковану таблицю, що містить усі ключі *B*-дерева. Очевидно, що основу алгоритму розв'язання подібної задачі повинен складати той або інший алгоритм обходу дерева. Використовуємо рекурсивний алгоритм обходу і будемо вважати, що *trav* (*node*) означає виклик алгоритму з передачею йому покажчика *node* на деякий вузол дерева. Тоді, щоб почати опрацювання, достатньо записати *trav* (*root*), враховуючи, що *root* є покажчиком на корінь *B*-дерева .

Алгоритм *trav* (*node*):

1. Якщо вузол *node* є листом, то надрукувати всі ключі і закінчити обчислення.
2. Якщо *node* не лист, то позначити ключі вузла як *key* [*1*],...*key*[*k*]; а покажчик на піддерева як *pt*[1],...*pt*[*k*].
   2.1 Виконати *trav* (*pt*[1]).
   2.2 Для *j*=1,...,*k*.
      2.2.1 Надрукувати значення ключа *key*.
      2.2.2 Виконати *trav*(*pt*[*i*+1]).

3. Кінець.

Для того, щоб мати можливість надрукувати ключі тільки з заданого діапазону, наприклад від *k*0 до *k*1 у цей алгоритм достатньо додати одну змінну *fi* логічного типу. Тепер при *fi*=0 здійснюється пошук, як тільки буде знайдено ключ, для якого *key*[*?*]=*k*0, достатньо змінити *fi* на *fi*=1 й далі використовувати алгоритм, аналогічний вище приведеному. Можна також замість рекурсії використовувати стек, в який при спуску вниз по дереву розміщати покажчики на вузли, що ще не оброблено. При цьому очевидно, що максимальний розмір стека не буде більше висоти *B*-дерева.

Оскільки велика частина ключів *B*-дерева містяться в листах, то головні витрати при роботі алгоритму *trav* припадають на спуск вниз по дереву й однорідному послідовному опрацюванню ключів, розташованих у листах. Проте, не можна зневажати і витратами, пов'язаними з поверненням на більш високий рівень дерева тому, що для цього потрібні звертання до зовнішньої пам'яті.

4.3. Різновиди *B*-дерев. *B*+-дерево

B-дерево, у якому точні значення ключів зберігаються тільки в листах (кінцевих вузлах), будемо називати *B*+-деревом. У внутрішніх вузлах *B*+-дерева зберігаються не точні значення

ключів, а ключі-роздільники, що задають діапазон зміни ключів для піддерева. В багатьох випадках вони збігаються з точними значеннями ключів або їхньої частини, але, як буде показано нижче, це не обов'язково. Оскільки всі ключі розташовані на одному рівні — у листах, то зв'язавши їх покажчиками, послідовний доступ стане особливо простим при високій швидкості доступу по дереву.

Відзначимо, що пошук по *B*+-дереву завжди закінчується в листі. В листах *B*-дерев також міститься велика частина ключів, і тому витрати часу на пошук по *B*+-дереву майже не збільшуються навіть без урахування того, що сама операція порівняння аргументу з ключем-роздільником швидша, ніж порівняння з самим ключем. При розщепленні листа в результаті вставки, ключ з листа не видаляється, а в якості ключа переносу використовується його копія. Операція видалення з *B*+-дерева має ту перевагу, що видалення завжди проводиться з листа. Наприклад при видаленні ключа 20 на рис. 4.13, *б* немає необхідності в зміні ключа-роздільника 20.

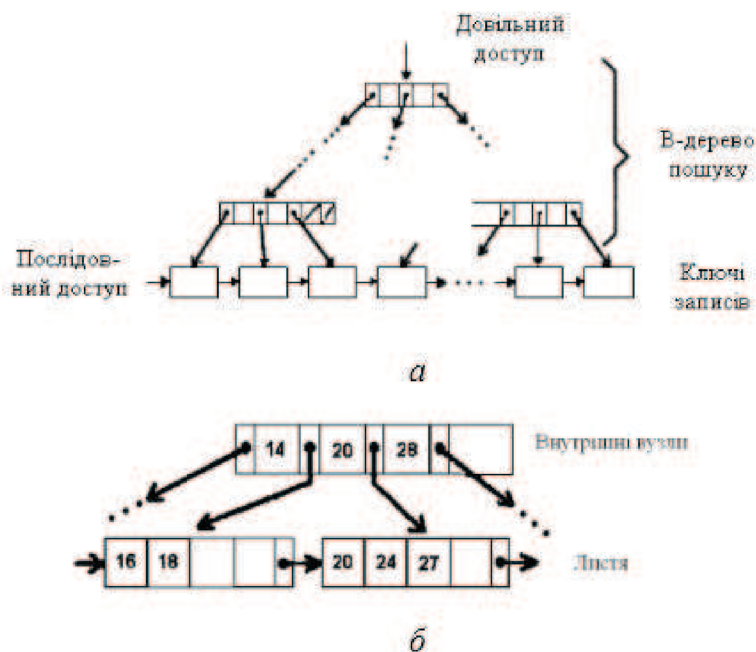


Рис. 4.13. B+-дерево: *а* — загальний вид *B*+-дерева;
б — видалення з *B*+ -дерева

Необхідність у зміні ключа-роздільника виникає тільки у випадку нестачі, коли необхідна перестановка ключів і встановлення нових зв'язків при балансуванні. Всі інші операції, крім розглянутих вище, виконуються аналогічно операціям над *B*-деревами. Таким чином, *B*+-дерева багато в чому перевершують *B*-дерева. Поступаються вони *B*-деревам у тому, що потребують більше пам'яті для представлення. Особливою перевагою *B*+-дерев є можливість послідовного доступу.

4.4. Зміни структури *B*-дерева

Деякі автори *B*+-дерева називають *B**-деревами. Проте *B**- і *B*-дерева не еквівалентні. Структурно *B**-дерева дуже близькі до *B*-дерев і відрізняються від них тільки більш високим коефіцієнтом використання пам'яті. Якщо для *B*-дерева кожний вузол повинен бути заповнений ключами не менше ніж наполовину, то для *B**-дерева потрібно, щоб кожний вузол був заповнений ключами не менше, ніж на 2/3. Щоб і у випадку переповнення при вставці забезпечити значення коефіцієнта заповнення кожного вузла не менше 2/3, необхідно спочатку об'єднати два сусідніх вузли, розташовані на однім рівні, а потім розщепити його на три частини. Якщо підвищити коефіцієнт використання пам'яті, то тільки за рахунок цього можна зменшити висоту дерева, а отже, і кількість кроків пошуку. Разом із цим може виявитися, що витрати на зміну структури дерева при вставках і видаленнях зростуть.

Ми трохи торкнулися структури лісу (*trie*-дерева), що використовують для пошуку простих слів, коли говорили про використання сильнорозгалужених дерев для пошуку в оперативній пам'яті. Якщо слова, для яких використовується таке дерево пошуку, стають занадто довгими, краще використовувати *B*+-дерево. Якщо список слів заздалегідь відомий, то витрати на пошук можна зменшити шляхом штучного вибору ключа-роздільника між словами.

4.5. Бінарне дерево

Бінарне дерево — це кінцева множина елементів, яка є порожньою або містить один елемент, який називається коренем дерева, а інші елементи множини поділяються на дві підмножини, що не перетинаються (кожна з яких є бінарним деревом).

Ці підмножини називаються лівим і правим піддеревами вихідного дерева. Кожний елемент бінарного дерева називається вузлом дерева. Бінарне дерево (*binary tree*) — це упорядковане дерево, що складається з вузлів двох типів: зовнішніх вузлів без дочірніх вузлів і внутрішніх вузлів, кожний з яких має рівно два дочірніх вузли [10]. Оскільки два дочірніх вузли кожного внутрішнього вузла упорядковані, говорять про лівий дочірній вузол (*left child*) і про правий дочірній вузол (*right child*) внутрішніх вузлів. Кожен внутрішній вузол повинен мати і лівий, і правий дочірні вузли, хоча один з них або обоє можуть бути зовнішніми вузлами.

На рис. 4.14 показано узвичаєний спосіб представлення бінарного дерева. Це дерево складається з дев'яти вузлів, *A* — корінь дерева.

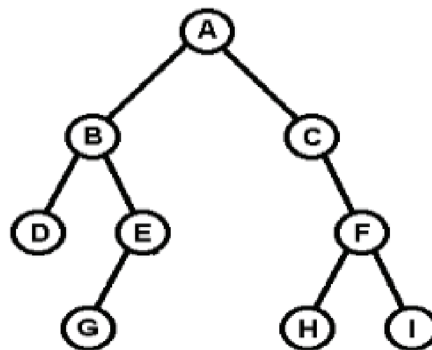


Рис. 4.14. Бінарне дерево

Ліве піддерево бінарного дерева, зображеного на рис. 4.14 має корінь *B*, а праве — корінь *C*. Це виражається двома гілками, що виходять з *A*: лівою — до *B* і правою — до *C*. Відсутність гілок означає порожнє піддерево. Наприклад, ліве піддерево бінарного дерева з коренем *C* і праве піддерево бінарного дерева з коренем *E* обидва порожні. Бінарні дерева з коренями *D, G, H* і *I* мають порожні ліві і праві піддерева. Якщо *A* — корінь бінарного дерева і *B* — корінь його лівого або правого піддерева, то говорять, що *A* — батько *B*, а *B* — лівий або правий син *A*. Вузол, що не має синів (такі вузли як *D, G, H* і *I*), називається листом. Вузол $n1$ — предок вузла $n2$ (а $n2$ — нащадок $n1$), якщо $n1$ — або батько $n2$, або батько деякого предка $n2$. Наприклад, у дереві на рис. 4.14 *A* — предок *G*, а *H* — нащадок *C*, але *E* не є ні предком, ні нащадком *C*. Вузол $n2$ — лівий нащадок вузла $n1$, якщо $n2$ є або лівим сином $n1$, або нащадком лівого сина $n1$. Схожим способом може бути визначено правий нащадок. Два вузли є братами. якщо вони сини того ж самого батька.

Якщо кожний вузол бінарного дерева, що не є листом, має непусті праві і ліві піддерева, то дерево називається строго бінарним деревом. Таким чином, дерево на рис. 4.15 є строго бінарним, у той час як дерево на рис 4.14 — ні (оскільки вузли *C* і *E* мають по одному синові).

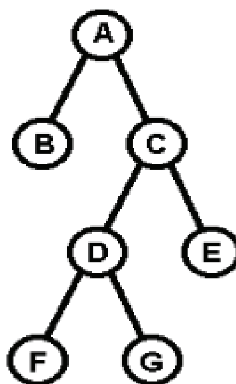


Рис. 4.15. Строго бінарне дерево

Строго бінарне дерево з n листами завжди містить $2n-1$ вузли. Рівень вузла в бінарному дереві може бути визначений у такий спосіб. Корінь дерева має рівень 0, а рівень будь-якого іншого вузла дерева має рівень на 1 більше рівня свого батька. Наприклад, у бінарному дереві на рис. 4.14 вузол E — вузол рівня 2, а вузол H — рівня 3. Глибина бінарного дерева — це максимальний рівень листа дерева, що дорівнює довжині найдовшого шляху від кореня до листа дерева. Тобто глибина дерева на рис. 4.14 дорівнює 3. Повне бінарне дерево рівня n — це дерево, у якому кожний вузол рівня n є листом, і кожний вузол рівня менше n має не порожні ліве й праве піддерева.

На рис. 4.16 приведено приклад повного бінарного дерева рівня 3.

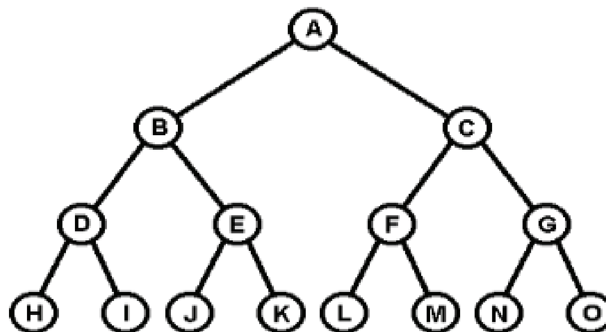


Рис. 4.16. Повне бінарне дерево

Майже повне бінарне дерево — це бінарне дерево, для якого існує невід'ємне ціле k таке, що:

1. Кожний лист у дереві має рівень k або $k+1$.

2. Якщо вузол дерева має правого нащадка рівня $k+1$, тоді усі його ліві нащадки, що є листами, також мають рівень $k+1$.

На рис. 4.17, *а—г* наведено різновиди бінарних дерев.

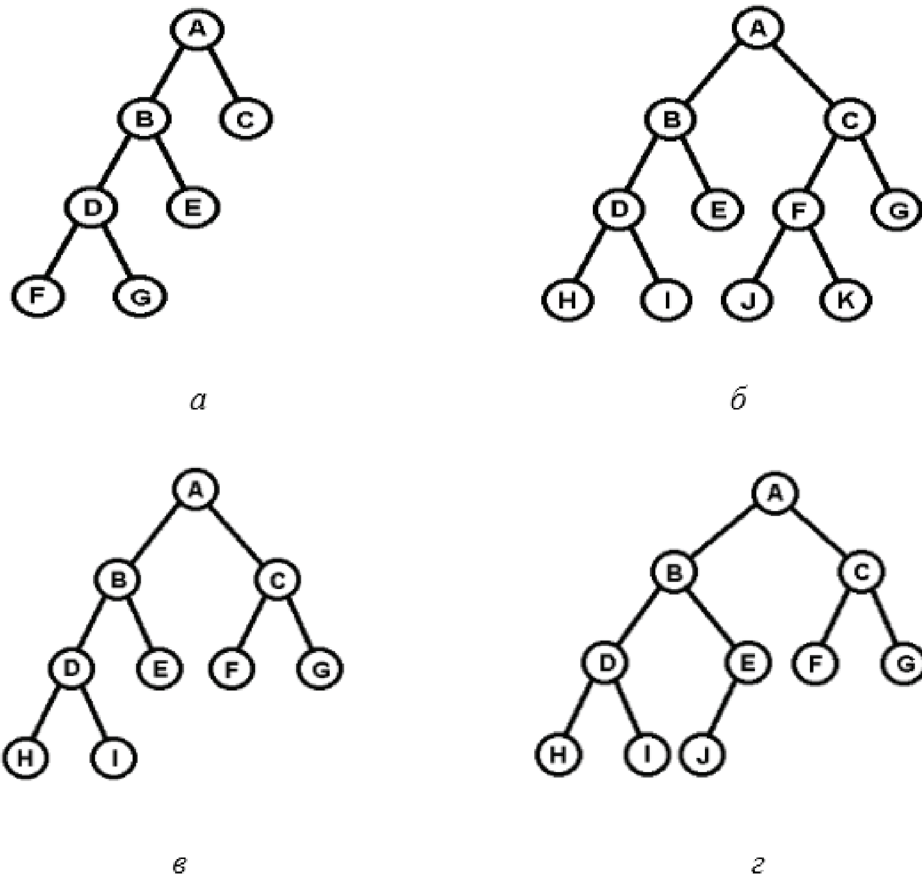


Рис. 4.17. Приклади бінарних дерев

Строго бінарне дерево з рис. 4.17, *а* не є майже повним, оскільки воно містить листи рівнів 1, 2 і 3, порушуючи тим самим умову 1. Строго бінарне дерево на рис. 4.17, *б* задовольняє умові 1 тому, що кожний лист має рівень 2 або 3. Проте, при цьому порушується умова 2, оскільки *A* має правого нащадка рівня 3(*J*), але так само і лівого нащадка, що є листом рівня 2(*E*). Строго бінарне дерево на рис. 4.17, *в* задовольняє обома умовам і, отже, є майже повним бінарним деревом. Бінарне дерево на рис. 4.17, *г* — також майже повне бінарне дерево, але воно не є строго бінарним, оскільки вузол *E* має лише лівого сина.

Вузли майже повного бінарного дерева можуть бути пронумеровані так, що кореню призначається номер 1, лівому сину — подвоєний номер його батька, а правому — подвоєний номер батька плюс одиниця. Рис. 4.18 ілюструє нумерацію вузлів дерева, показаного на рис. 4.17, *в*. При такій схемі нумерації кожному вузлу майже повного бінарного дерева присвоюється унікальний номер, що визначає позицію вузла рівня дерева.

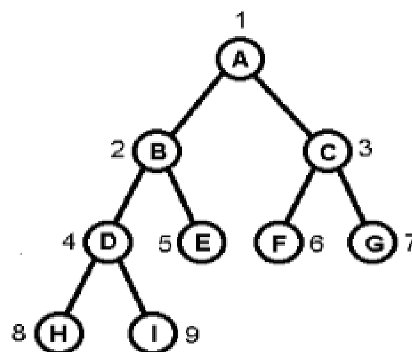


Рис. 4.18. Нумерація вузлів бінарного дерева

Майже повне строго бінарне дерево з *n* листами має, як і будь-яке інше строго бінарне дерево з *n* листами, $2n$–1 вузли. Майже повне бінарне дерево з *n* листами, що не є строго бінарним, має $2n$ вузлів. Існує два різноманітних, майже повних бінарних дерева з *n* листами: одне з них строго бінарне, а друге — ні. Наприклад, обидва дерева на рис. 4.17, *в* і рис. 4.17, *г* майже повні і мають по п'ять листів; проте дерево на рис. 4.17, *в* строго бінарне, тоді як дерево на рис. 4.17, *г* — ні. Існує тільки одне майже повне бінарне дерево з *n* вузлами. Воно є строго бінарним тільки, якщо *n* непарне. Таким чином, дерево на рис. 4.17, *в* — це єдине майже повне бінарне дерево з дев'ятьма вузлами, і воно ж є строго бінарним, тому що 9 — непарне число. У той же час дерево на рис. 4.17, *г* — єдине майже повне бінарне дерево з 10 вузлами, і воно не є строго бінарним тому, що число 10 парне.

4.6. Застосування бінарних дерев

Бінарне дерево — корисна структура даних у тих випадках, коли в кожній точці процесу повинно бути прийняте одне рішення з двох можливих. Наприклад, припустимо, що ми хочемо знайти всі дублікати в списку чисел. Один спосіб виконання цієї задачі складається в порівнянні кожного числа з попереднім. Проте, для цього потрібно велика кількість порівнянь. Шляхом використання бінарного дерева кількість порівнянь може бути зменшена. Зчитується перше число й поміщається у вузол, що стає коренем бінарного дерева з порожніми лівим і правим піддеревами. Потім кожне наступне число зі списку порівнюється з числом, що знаходиться в корені дерева. Якщо значення збігаються, то це дублікат. Якщо нове число має менше значення в корені, то процес повторюється для лівого піддерева, а якщо більше, то для правого. Так продовжується доти, поки не зустрінеться дублікат або, поки ми не досягнемо порожнього піддерева. В останньому випадку число розташовується в новий вузол порожнього піддерева.

На рис. 4.19 наведно дерево, що було б створено при введені таких чисел: 14 15 4 9 7 18 3 5 16 4 20 17 9 14 5.

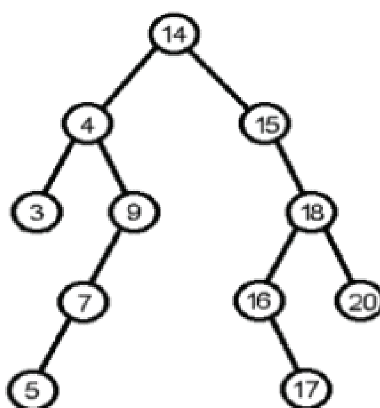


Рис. 4.19. Приклад створення бінарного дерева

В результаті розв’язання задачі, буде видано відповідь, що числа 4, 9, 14 і 5 — дублікати.

Ще одна звичайна операція — проходження бінарного дерева, тобто треба обминути усе дерево, відмічаючи кожний вузол один раз. Можна було, друкувати вміст кожного вузла, коли ми відмічаємо його або опрацювати його яким-небудь іншим способом.

Очевидний порядок проходу вузлів лінійного списку — від першого до останнього. Проте для вузлів дерева не існує такого «природного» лінійного порядку. Тому, у різноманітних випадках для проходження використовують різноманітні способи упорядкування. Ми визначимо три методи проходження. У кожному з цих методів не потрібно ніяких дій для проходження порожнього бінарного дерева. Методи визначаються рекурсивно так, що проходження бінарного дерева потребує перевірки кореня і проходження його лівого й правого піддерев.

Розглянемо алгоритми для реалізації найбільш загальної функції обробки дерев — обходу дерева; при наявності покажчика на дерево потрібно систематично обробити всі вузли в дереві. При роботі з бінарними деревами існують два зв'язки і, отже, три основних порядки можливого обходу вузлів:

- прямій обхід (зверху вниз), при якому ми відвідуємо вузол, а потім ліве і праве піддерево;

- поперечний обхід (ліворуч праворуч), при якому ми відвідуємо ліве піддерево, потім вузол, а потім праве піддерево;

- зворотний обхід (знизу нагору), при якому ми відвідуємо ліве і праве піддерево, а потім вузол.

Ці методи можна легко реалізувати за допомогою рекурсивної програми, приклад якої, виконаної на мові програмування *C*++ наведено нижче.

Ця рекурсивна функція приймає як аргумент посилання на дерево і викликає функцію *visit* для кожного з вузлів дерева.

У приведеному прикладі функція реалізує прямий обхід; якщо помістити звертання до *visit* між рекурсивними викликами, вийде поперечний обхід; а якщо помістити звертання до *visit* після рекурсивних викликів — то зворотній обхід [8].

```
void traverse(link h, void visit(link))
{
if (h ==0) return;
visit(h);
traverse(h–>l, visit);
traverse(h–>r, visit);
}
```

Логічна нумерація вершин дерев при використанні рекурсії проводиться згідно послідовності їх рекурсивного обходу. Рекурсивна функція у цьому випадку одержує посилання або покажчик на лічильник вершин, який вона збільшує на 1 при обході поточної вершини. Залежно від того, хто нумерується раніше — предок або нащадки, існують різні способи обходу й нумерації (рис. 4.20).

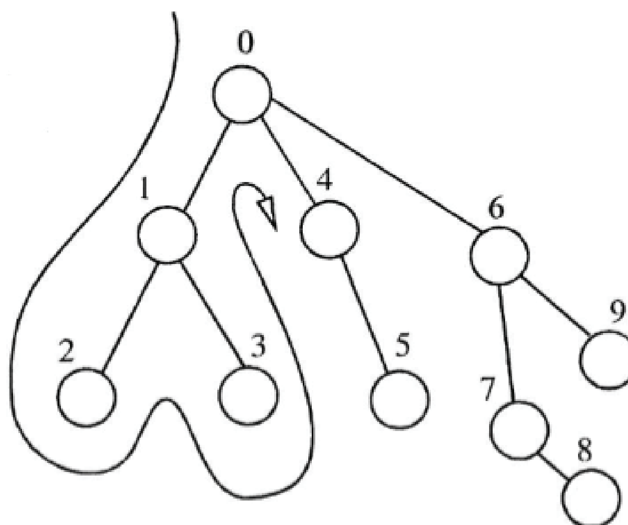


Рис. 4.20. Нумерація дерева при рекурсивному обході

Наступний фрагмент програми (мова програмування *C++*) демонструє обхід дерева з нумерацією вершин зверху вниз [6]:

```
void ScanNum (tree *q, int &n )
 {
if (q==NULL) return;
printf("n = %d val = %d\n",n++,q–>val);
for (int i=0; i<4; i++) ScanNum(q–>p[i],n);
}
```

Обхід з нумерацією у звичайнім дереві використовується для витягання вершини по логічному номеру. При досягненні вершини із заданим номером обхід припиняється (аналогічно алгоритму пошуку першого підходящого). Витягання по логічному номеру з повним обходом дерева наведено в наступному фрагменті програми (мова програмування *C++*) [6]:

```
int GetNum (tree *q, int &n, int num )
{
if (q==NULL) return – 1 ;
if ( n++ ==num) return q–>val;          // Номер поточної співпав з тим, що вимагають
for (int i=0; i<4; i++) {                 // Обхід нащадків,
int vv=GetNum (q–>p[i],n,num );       // доки не перевищено номер
if (n > num) return vv;
}
return – 1 ; }
```

Якщо кожна вершина дерева буде містити додатковий параметр — кількість вершин у пов'язаному з нею піддереві, то витягання по логічному номеру виконується за допомогою циклічного алгоритму або лінійної рекурсії, завдяки чому можна відразу ж визначити, в якому піддереві перебуває вершина, що нас цікавить. Лічильники вершин можна корегувати в самому процесі додавання/видалення вершин.

// Витягання по логічному номеру (лічильник вершин в піддереві) (C++)

```
struct tree
{
int nodes;                    // Лічильник вершин в піддереві
int val;
ctree *p[4];
};
int GetNum(ctree *q, int num, int n0)
{
if (q==NULL) return 1;    // n0 початковий номер в поточному піддереві
if (n0++==num) return q–>val;    // Початковий номер співпав з тим, що вимагали
for (int i=0; i<4; i++)
{
if (q–>p[i] = = NULL) continue;
int nc= q–>p[i]–>nodes;  // Кількість вершин у нащадка
if (n0+ nc > num)                   // Вибрано нащадок
return GetNum(q–>p[i],num,n0);      // з діапазоном номерів
else                          // Корегувати початковий номер
n0+=nc;                       // для наступного нащадка
}
}
```

Корисно також розглянути нерекурсивні реалізації, в яких використовується явний стек. Для простоти ми почнемо з розгляду абстрактного стека, що містить елементи дерева, ініціалізованого деревом, що потрібно обійти. Потім ми ввійдемо в цикл, у якому виштовхується й обробляється верхній запис стека, цикл продовжується, поки стек не спорожніє. Якщо виштовхнутий запис є елементом, ми відвідуємо його; якщо виштовхнутий запис —

дерево, ми виконуємо послідовність операцій виштовхування, що залежить від необхідного порядку:

- для прямого обходу заштовхується праве піддерево, потім ліве піддерево, а потім вузол;
- для поперечного обходу заштовхується праве піддерево, потім вузол, а потім ліве піддерево;
- для зворотного обходу заштовхується вузол, потім праве піддерево, а потім ліве піддерево.

Нульові дерева в стек не заштовхуються. На рис. 4.21 показано вміст стека при використанні кожного з цих трьох методів для обходу дерева.

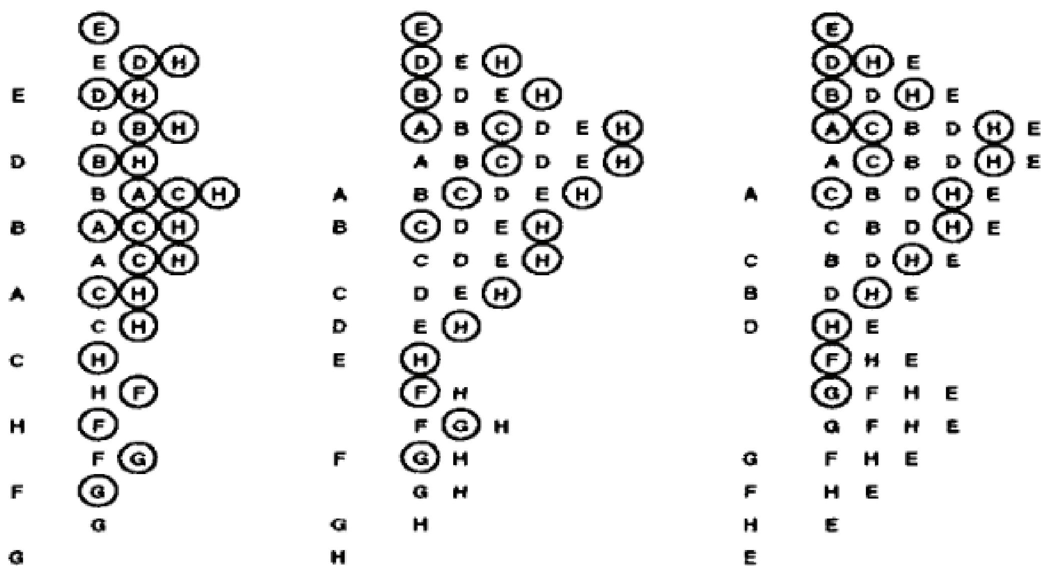


Рис. 4.21. Зміна вмісту стека при обході дерева

Послідовності, що наведені на рис. 4.21 відображають вміст стека для прямого (ліворуч), поперечного (у центрі) і зворотного (праворуч) обходу дерева. Методом індукції можна легко переконатися, що для будь-якого бінарного дерева цей метод призводить до такого ж результату, як і рекурсивний метод.

Описана в попередньому абзаці схема є концептуальною, що охоплює три методи обходу дерева, однак реалізації, використовувані на практиці, трохи простіші. Наприклад, для виконання прямого обходу не обов'язково заштовхувати вузли в стек (ми відвідуємо корінь кожного дерева, що виштовхується), можна скористатися простим стеком, що складається тільки з одного типу елементів (зв'язків дерева), як це зроблено в нерекурсивній реалізації наведеної нижче програми (мова програмування *C*++). Системний стек, що підтримує рекурсивну програму, містить адреси повернення і значення аргументів, а не елементи або вузли, але фактична послідовність виконання обчислень (відвідування вузлів) залишається однаковою для рекурсивного методу і методу з використанням стека [8].

```
void traverse(link h, void visit(link))
 { STACK<link> s(max);
 s.push(h);
while (!s.empty())
{
visit (h = s.pop());
if (h–>r != 0) s.push(h–>r);
if (h–>1 != 0) s.push(h–>1) ;
}
}
```

4.7. Приклади застосування бінарних дерев в *Pascal*

Для представлення вузла двійкового дерева в мові програмування *Pascal* зручно користуватися записами:

```
type
 зв'язок = ^вузол;
 вузол =
 record
```

```
  ліве, праве: зв'язок ;
  дані: тип
 end;
```

Як було сказано вище перегляд двійкового дерева може проводитися рекурсивно: для кожного вузла потрібно виконати три дії. Для позначення операції, що необхідно виконати над кожним вузлом дерева, використовується термін «дослідити»:

- дослідити вузол;
- переглянути ліве піддерево;
- переглянути праве піддерево.

Ці три кроки можуть бути виконані шістьма різноманітними послідовностями, тому для перегляду дерева існує шість різноманітних способів. Завдяки існуванню традиційного ствердження, що ліве піддерево завжди проглядається перед правим, кількість різноманітних способів перегляду знижується із шести до трьох. Три способи, що залишилися, мають спеціальні найменування: при прямому (*preorder*) перегляді спочатку досліджується вузол, а потім ліве і праве піддерева; при зворотному (*postorder*) перегляді досліджується ліве піддерево, вузол, і потім праве піддерево; при кінцевому (*endorder*) перегляді вузол досліджується після перегляду дерев.

Рекурсивна процедура, що виконує перегляд двійкового дерева, виконана на мові програмування *Pascal:*

```
Procedure перегляд (дерево, зв'язок);
 Begin
 If дерево <> nil
   then
   begin
     досліджувати (дерево);
     перегляд (дерево^ . ліве);
    перегляд (дерево^ . праве);
   end;
End;
```

Така процедура здійснює прямий перегляд дерева. Інші види перегляду можуть бути отримані шляхом перестановки трьох операторів, що входять у внутрішній складовий оператор. При різноманітних порядках перегляду дерева, вузли досліджуються в такій послідовності:

Прямий: ДБАГВЕЗЖИ
Зворотній: АБВГДЕЖЗИ
Кінцевій: АВГБЖИЗЕД

Зворотний перегляд дерева призвів до виникнення упорядкованості вузлів за алфавітом. Це не випадковий збіг. Процедура вставити добавляє до двійкового дерева пошуку один вузол, зберігаючи при цьому упорядкованість дерева (мова програмування *Pascal*).

```
Procedure вставити (var дерево : зв'язок; нові дані : тип даних );
Begin
If дерево = nil
then
begin
new(дерево);
with дерево^ do
Begin
ліве:= nil
 праве:= nil
 дані:=нові дані
   end;
  End {with}
  else
  with дерево^   do
               if нові дані < дані
                  then   уставити (ліве, нові дані
  elseif   нові дані >
                         then   уставити ( праве,нові дані)
                      else   { дублювання інформації }
End;
```

Наступна функція виділяє в результаті роботи посилання на вузол дерева, що містить необхідні дані, або порожнє посилання, якщо потрібна інформація в дереві відсутня (мова програмування *Pascal*).

```
Function знайти (дерево : зв'язок; ключ : тип даних) : зв'язок
begin
if дерево = nil
then знайти := nil
else
with дерево^do
      if ключ<дані
then знайти:=знайти (ліве,ключ)
{ використовуємо рекурсію}
 else if ключ > дані
then := знайти (праве, ключ)
 else знайти := дерево;
end;
```

Функцію «знайти» можна написати і не застосовуючи рекурсію (мова програмування *Pascal*):

```
var
кінець: boolean ;
 begin
 кінець :=false ;
 repeat
  if дерево = nil ;
  then кінець := true
     else
      with дерево^ do
           if ключ< дані
              then дерево :=ліве
              else if ключ>дані
               then дерево := праве
                else кінець :=true ;
```

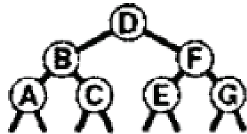
```
    until кінець;
    знайти :=дерево ;
  end;
```

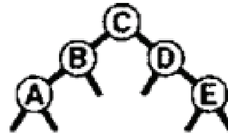
Зауважимо, нарешті, що елементи двійкового дерева пошуку можуть бути перераховані в правильній послідовності за допомогою зворотного перегляду. Єдиною операцією, що потребує значних зусиль, є операція видалення вузла, що не є листом дерева. Двійкове дерево пошуку є більш зручною структурою для збереження і пошуку даних ніж масив. При роботі з таким деревом треба попередньо переконатися, що нові дані не надходять у порядку зростання або убування, тому що в цьому випадку дерево вироди́ться в лінійний список. У багатьох практичних випадках, дані надходять у випадковому порядку, що дозволяє будувати більш-менш ефективні дерева пошуку. У випадковому дереві, що складається з N вузлів, час, потрібний для додавання або видалення вузла, пропорційний $log(N)$, тоді як відповідний час проведення лінійного пошуку по масиві пропорційний N.

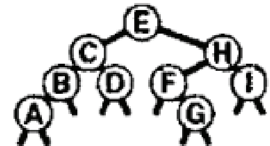
Контрольні запитання та завдання для самоконтролю

1. Що таке структура дерево?
2. Що таке бінарне дерево?
3. Що таке строго бінарне дерево?
4. Що таке корінь дерева?
5. Що таке лист дерева?
6. Опишіть схему прямого проходження дерева.
7. В чому полягає сутність зворотного проходження дерева?
8. В чому полягає сутність поперечного проходження дерев?
9. Яка відмінність проходження дерев при використанні рекурсії і без неї?

10. Наведіть порядок обходження вузлів для прямого, поперечного, зворотного обходу по рівнях для наступних бінарних дерев:







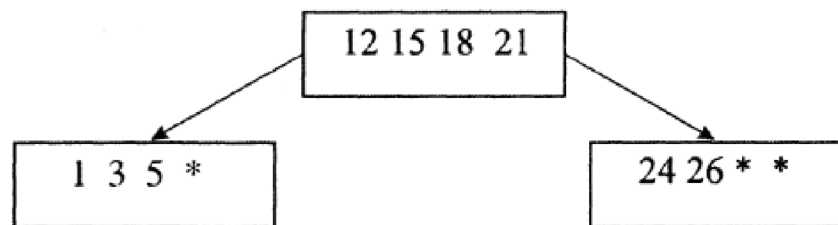
11. Приведіть нерекурсивну реалізацію поперечного обходу дерева.

12. Приведіть нерекурсивну реалізацію зворотного обходу бінарного дерева.

13. Створіть схему упорядкованого дерева, що визначене набором ребер 0–1, 1–2, 1–3, 1–4, 4–5.

14. Створіть програму, в якій всі елементи бінарного дерева порівнюються операцією *operator==*. Створіть рекурсивну функцію, що видаляє в бінарному дереві елементи, що дорівнюють введеному.

15. Створіть програму, в якій вершина двійкового дерева містить масив цілих чисел і два покажчики на праве і ліве піддерева. Масив цілих чисел у кожнім елементі упорядкований, дерево в цілому також упорядковано. Функція додає до дерева цілу змінну зі збереженням упорядкованості, використовуючи наступний рисунок.



16. Визначите вид дерева і дії, що над ними виконуються. Напишіть виклик функції для статичного дерева, складеного з ініціалізованих змінних.

Приклад оформлення тестового завдання.

Завдання:

```
Struct tree { char *s; tree *p[4]; };
int F( tree *q) {
if (q==NULL) return 0;
for (int v=strlen(q–>s), i=0; i<4; i++)
v+=F(q–>p[i]);
return v; }
```

Розв'язання завдання.

Мова йде про дерево, що підтверджується наявністю рекурсії для покажчиків на «сусідні» елементи структури даних. Вершина дерева містить покажчик на рядок. У кожній вершині виконується підсумовування довжини рядка, що утримується в ній, і результатів рекурсивного виклику нащадків. Підсумок: функція повертає сумарну довжину рядків у вершинах дерева.

```
tree A1={"aaa",NULL,NULL.NULL.NULL};
tree A2 = {"bb",NULL,NULL,NULL.NULL};
tree A3 = {"cccc",&A1,&A2,NULL,NULL};
tree A4={"dd",NULL,NULL,NULL,NULL};
tree A5 = {"aaa",NULL,NULL.NULL.NULL};
tree A6={"fff",&A3,&A4,&A5.NULL};
void main()
{ printf("F = %d\n",F(&A6)); }
```

//Завдання № 2

```
struct XXX { int v; xxx *p[4]; };
int F1 (xxx *q)
{ int i.n.rn;
if (q==NULL) return 0;
for (n = F1(q–>p[0]),i = 1; i<4; i++)
If ((m = F1(q–>p[i])) >n) n = m;
return n + 1; }
```

```
//Завдання №3
struct zzz { int v; zzz *l,*r; };
int F2(zzz *p) {
if (p==NULL) return(O);
return (1 + F2(p–>r) + F2(p–>l)); }
//Завдання №4
int F3(xxx *q)
{ int i,n,m;
if (q==NULL) return 0;
for (n=q–>v,i=0; i<4; i++)
if ((m = F3(q–>p[i])) >n) n=m;
return n; }
//Завдання №5
void F4(int a[], int n, int v) {
if (a[n] ==–1) { a[n]=v; return; }
if (a[n] ==v) return;
if (a[n] >v) F4(a,2*n,v);
else F4(a,2*n + 1,v);
}
void z3() {
int B[256],i;
for (i=0; i<256; i++) B[i] = – 1 ; F4(B,1,5); F4(B,1,3); }
```

Розділ 5. РЕКУРСІЯ

5.1. Поняття рекурсії й основні визначення

Слово «рекурсія» — похідне від терміна *recurrence,* що в перекладі з англійської означає повернення, повторення [10]. Усі добре знають проблему курки й яйця: курка вилупилася з яйця, але яйце знесла курка, що в свою чергу вилупилася з яйця і т.д. Щоб визначити, що з'явилося раніш: яйце або курка, треба пройти весь ряд курка — яйце — курка — яйце і т.д. до його початку. Проблема курки й яйця — це класичний приклад рекурсії, в якій, у цьому випадку, відповідь не дається.

Математики не зустрічають труднощів у цій задачі вони, зазвичай, задають, що є початковим, і коли це задано, одержують все інше. Як приклад розглянемо ряд Фібоначчі:

$$1\ \ 1\ \ 2\ \ 3\ \ 5\ \ 8\ \ 13\ \ 21$$

Перші два члени цього ряду чисел одиниці, і тут початкова ситуація цілком визначена. Потім можна побачити, що кожний наступний член ряду знаходиться з виразу

$$T(n)=T(n-1)+T(n-2),$$

де $T(n)$ знаходиться як сума двох попередніх членів $T(n-1)$ і $T(n-2)$. Наприклад, третій член $T(3)$ дорівнює сумі другого $T(2)=1$ і першого $T(1)=1$ членів ряду. В такий спосіб $T(3\square)=1+1=2$. Четвертий член $T(4)$ дорівнює сумі третього $T(3)=2$ і другого $T(2)=1$ членів, іншими словами, $T(4)=2+1=3$ і т.д.

Тут також використані дві рекурсії: для того, щоб знайти кожний новий член ряду, треба повернутися до двох попередніх членів. Між іншим, ряди Фібоначчі завжди відігравали велику роль у математиці. Якщо послідовно ділити два поточні останні члени ряду один на одний, то будемо одержувати результат, відомий під назвою «золотий перетин».

У програмуванні прикладів використання рекурсії ще більше:

- рекурсивне визначення в синтаксисі мови. Наприклад, визначення будь-якого конкретного оператора (умовний, цикл, блок) в якості складових частин включає довільний оператор;

- рекурсивна структура даних — елемент структури даних містить один або кілька покажчиків на аналогічну структуру даних. Наприклад, однозв'язний список можна визначити як елемент списку, що містить покажчик *NULL* або покажчик на аналогічний список;

- рекурсивна функція — тіло функції містить прямій або непрямий (через іншу функцію) власний виклик.

Рекурсивним називається об'єкт, що частково визначається через самого себе.

Очевидно, що рекурсія не може бути безумовною, у цьому випадку, вона стає нескінченною. Рекурсія повинна мати в собі умови завершення, по якім її черговий крок вже не проводиться. Інша, ще не відзначена, особливість полягає в тому, що поряд з лінійною рекурсією, коли визначення об'єкта містить у собі єдиний аналогічний об'єкт, існує ще й розгалужена рекурсія, коли таких об'єктів багато.

Застосування рекурсії часто дозволяє давати більш «прозорі» алгоритми, ніж без рекурсії. Велику кількість математичних функцій можна визначити рекурсивно. Найбільш типовим є приклад операції привласнювання, у котрій змінна визначається через саму себе, раніш обчислену:

$$X: =X + Y\ (Pascal);$$

$$x+=y\ (C++).$$

Більш складним є використання рекурсивних підпрограм:

а) Цілий ступінь числа.

$$xn = \begin{cases} 1, & якщо \quad n = 0; \\ x^{*}xn - 1, & якщо \quad n > 0; \end{cases}$$

б) Обчислення факторіала.

$$n! = \begin{cases} 1, & якщо \quad n = 0; \\ n^*(n-1)!, & якщо \quad n > 0. \end{cases}$$

в) Поліном Лежандра.

$$Pn(x) = \begin{cases} 1, & якщо \quad n = 0; \\ x, & якщо \quad n = 1; \\ (2n-1)xPn-1(x)-(n-1)Pn-2(x)/n, & якщо \quad n > 1. \end{cases}$$

По приведених вище визначеннях неважко скласти описи рекурсивних функцій. Для третього приклада, що є рекурсивним визначенням полінома Лежандра при використанні мови програмування *Pascal*, одержимо:

```
Function p(n:Integer; x:Real):Real;
Begin
if n=0 then p:=1
        else if n=1 then p:=x
              else p:=((2*n–1)*x*p(n–1,x)–(n–1)*p(n–2,x))/n;
End;
```

Проте, в більшості випадків рекурсивне рішення не є кращим, оскільки більш просте рішення може бути знайдено за допомогою ітерації.

Використання рекурсії дозволяє легко запрограмувати обчислення по рекурентним формулам. Наприклад, програма, що використовує рекурсивну функцію для обчислення факторіала має, при використанні мови програмування *Pascal,* наступний вигляд:

```
Program Factorial;
Var
  n:Integer;
function Fact(i:Integer):LongInt;
begin
if i=1 then Fact:=1
```

```
        else Fact:=i*Fact(i–1);
end;
Begin
Write('Введіть число n:');
ReadLn(n);
WriteLn('Факторіал n! =',Fact(n));
End.
```

Ця ж програма без використання рекурсії з використанням мови програмування *Pascal* мала б вигляд:

```
Program Factorial;
Var
Fact:LongInt;
 n,i:Integer;
Begin
Write('Введіть число n:');
ReadLn(n);
Fact:=1;
for i:=1 to n do Fact:=Fact*i;
WriteLn('Факторіал n! =', Fact);
End.
```

Визначення факторіалу при використанні мови програмування *C++* буде мати наступний вигляд:

```
int factorial(int N)
{
if (N == 0) return 1;
return N*factorial(N–1);
 }
```

Далі наведено приклад компактної реалізації алгоритму Евкліда для пошуку найбільшого взаємного дільника для двох цілих чисел. Алгоритм ґрунтується на спостереженні, що найбільший загальний дільник двох цілих чисел x та y, коли $x > y$, збі-

гається з найбільшим загальним дільником числа *y* і *x* по модулю (в залишку від розподілу *x* на *y*). Число *t* поділяє і *x,* і *y* тоді, і тільки тоді, коли воно поділяє і *y,* і *x mod y* (*x* по модулю *y*), оскільки *x* дорівнює *x mod y* плюс число, кратне *y*. При реалізації алгоритму Евкліда глибина рекурсії залежить від арифметичних властивостей аргументів (вона зв'язана з ними логарифмічною залежністю). Аргумент функції — значення, що передається функції. Лістинг програми, що реалізує алгоритм Евкліда подано на мові програмування *C++:*

```
int gcd(int m, int n)
{
if (n == 0) return m;
return gcd(n, m % n) ;
}
```

Вибір між рекурсією й ітерацією визначається вимогами до обсягу робочої пам'яті. Оскільки, при кожному новому виклику підпрограми виділяється нове місце в стеку для локальних змінних, то може не вистачити місця в пам'яті, і проблема стає нерозв’язною.

У розглянутих вище випадках процедури або функції викликали самі себе, це називається прямою рекурсією. Крім того, процедура *P* може викликати процедуру *Q*, що в свою чергу викликає процедуру *P* – це називається непрямою рекурсією.

Для створення рекурсивних алгоритмів необхідні і достатні знання поняття процедури і функції. Процедури і функції дозволяють задати будь-якій послідовності дій (операторів) ім'я, за допомогою якого можна буде цю послідовність дій викликати.

Програми, в яких використовуються рекурсивні процедури, відрізняються простотою, наочністю і компактністю тексту. Такі якості рекурсивних алгоритмів витікають з того, що рекурсивна процедура вказує, що потрібно робити, а нерекурсивна більше акцентує увагу на тому, як потрібно робити.

Виконання рекурсивних процедур вимагає значно більшого розміру оперативної пам'яті під час виконання, ніж нерекурсивних. При кожному рекурсивному виклику для локальних змінних, а також для параметрів процедури, що передаються за значенням, виділяються нові чарунки пам'яті. Таким чином, якійсь локальній змінній *A* на різних рівнях рекурсії будуть відповідати різні чарунки пам'яті, що можуть мати різні значення. Тому, скористатися значенням змінної *A* *i*-го рівня рекурсії можна знаходячись тільки на цьому *i*-му рівні.

Максимальна кількість рекурсивних викликів процедури без повернень, що відбувається під час виконання програми, називається глибиною рекурсії. Кількість рекурсивних викликів у кожний конкретний момент часу називається поточним рівнем рекурсії.

5.2. Форми рекурсивних функцій

У загальному випадку будь-яка рекурсивна процедура *Rec* містить у собі деяку множину операторів *S* і один або декілька операторів рекурсивного виклику *P*.

Рекурсивні функції лише на перший погляд виглядають як звичайні фрагменти програм, щоб відчути їхню специфіку, досить подумки простежити по тексту програми процес її виконання. У звичайній програмі ми будемо проходити по ланцюжкові викликів функцій, але жодного разу повторно не ввійдемо в той самий фрагмент, поки з нього не вийшли. Можна сказати, що процес виконання програми «лягає» однозначно на текст програми. Інша справа — рекурсія. Якщо спробувати відстежити по тексту програми процес її виконання, то ми прийдемо до такої ситуації: увійшовши в рекурсивну функцію *F*, ми «рухаємося» по її тексту доти, поки не зустрінемо її виклик, після чого ми знову почнемо виконувати ту ж саму функцію спочатку. При цьому слід зазначити найважливішу властивість рекурсивної функції —

її перший виклик ще не закінчився. Зовні створюється враження, що текст функції відтворюється (копіюється) щораз, коли функція сама себе викликає:

void main()	*void F();*	*void F();*	*void F()*
{	*{*	*{*	*{*
F();	*..if () F();*	*..if() F();*	*..if() F();*
}	*}*	*}*	*}*

Насправді цей ефект відтворюється в комп'ютері. Однак, при цьому копіюється не весь текст функції (не вся функція), а тільки її частини, пов'язані з локальними даними (формальні, фактичні параметри, локальні змінні й крапка повернення). Алгоритмічна частина (оператори, вираження) рекурсивної функції й глобальні змінні не міняються, тому вони присутні в пам'яті комп'ютера в єдиному екземплярі.

Як було показано вище, безумовні рекурсивні процедури призводять до безкінечних процесів, і на цю проблему потрібно звернути особливу увагу тому, що практичне використання функцій із безкінечним самовикликом неможливе. Така неможливість пояснюється тим, що для кожної копії рекурсивної процедури необхідно виділяти додаткову область пам'яті, а нескінченої пам'яті не існує. Отже, головна вимога до рекурсивних функцій полягає в тому, що виклик рекурсивної функції повинен виконуватися за умови, що на якомусь рівні рекурсії вона стане помилковою. Якщо умова істинна, то рекурсивний спуск продовжується. Коли вона стає помилковою, то спуск закінчується, і починається почергове рекурсивне повернення з усіх викликаних на даний момент копій рекурсивної функції.

Кожний рекурсивний виклик породжує новий «екземпляр» формальних параметрів і локальних змінних, причому старий «екземпляр» не знищується, а зберігається в стеці за принципом вкладеності. Тут має місце випадок, коли одному імені змінної в процесі роботи програми відповідає декілька її екземплярів.

Відбувається це в такій послідовності:

- у стеці резервується місце для формальних параметрів, в які записуються значення фактичних параметрів. Зазвичай, це проводиться в порядку, зворотному їхньому проходженню в списку;

- при виклику функції в стек записується крапка повернення — адреса тієї частини програми, де перебуває виклик функції;

- на початку тіла функції в стеці резервується місце для локальних (автоматичних) змінних.

Рекурсивний виклик, «екземпляр» рекурсивної функції є одним з ідентичних повторюваних кроків деякого процесу, який в цілому й вирішує поставлене завдання. У термінах процесу і його кроків основні параметри рекурсивної функції одержують додатковий зміст:

- формальні параметри рекурсивної функції є початковим станом для поточного кроку процесу;

- фактичні параметри рекурсивного виклику є початковим станом для наступного кроку — переходу з поточного при рекурсивному виклику;

- автоматичні змінні є внутрішніми характеристиками процесу на поточному кроці його виконання;

- зовнішні змінні є глобальним станом усієї системи, через які окремі кроки в послідовності можуть взаємодіяти.

Це значить, що формальні параметри рекурсивної функції, глобальні й локальні змінні не можуть бути взаємозамінні, як це іноді робиться в звичайних функціях.

Специфіка рекурсивних алгоритмів полягає в тому, що вони повністю виключають «історичний» підхід до проектування програми. Спроби логічно простежити послідовність рекурсивних викликів заздалегідь приречені на провал. Їх можна прокоментувати приблизно такою фразою: «Функція *F* виконує ... і викликає *F*, яка виконує ... і викликає *F*...». Ясно, що для логічного аналізу програми в цьому мало користі. Проте, ця фраза смутно нагадує

нам спроби «історичного аналізу циклічних програм». Там для того, щоб зрозуміти, що робить цикл, пропонувалося використовувати деякий інваріант (умова, співвідношення), що зберігається в циклі. Наявність такого інваріанта дозволяє «не заглядати уперед» до наступних і «не обертатися назад» до попередніх кроків циклу тому, що на них робиться те ж саме.

Аналогічна ситуація має місце в рекурсії. Тільки вона збільшується тим, що при розгалуженій рекурсії «історичний» підхід взагалі не застосовується, оскільки: «Функція F виконує ... і викликає F другий раз, яка виконує ... і викликає F в третій раз ... а потім, коли знову повернеться в перший виклик, викличе F ще раз у другий раз...». Звідси перша заповідь: алгоритм повинен розроблятися, не виходячи за рамки поточного рекурсивного виклику. Інші принципи:

- рекурсивна функція розробляється як узагальнений крок процесу, який викликається в довільних початкових умовах і призводить до наступного кроку в деяких нових умовах;
- для кроку процесу рекурсивного виклику, необхідно визначити інваріанти, що зберігаються в процесі виконання алгоритму, умови й співвідношення;
- початкові умови чергового кроку повинні бути формальними параметрами функції;
- початкові умови наступного кроку повинні бути сформовані у вигляді фактичних параметрів рекурсивного виклику;
- локальними змінними функції повинні бути оголошені усі змінні, які мають відношення до протікання поточного кроку процесу і до його стану;
- у рекурсивній функції обов'язкова перевірка умов завершення рекурсії, за яких наступний крок процесу не виконується.

Результат рекурсивної функції, зазвичай, пов'язаний зі способом перебору варіантів і методом досягнення мети в процесі рекурсивного пошуку.

1. Використовується повний перебір можливих варіантів і видача (збереження) всіх варіантів, що відповідають меті. Зазвичай, рекурсивна функція має результат *void*, отже, вона не може вплинути на характер наступного протікання процесу пошуку. Якщо при пошуку знаходяться підходящі варіанти (успішне завершення рекурсії), то вони можуть зберігатися в глобальній структурі даних, з якою працюють усі кроки рекурсивного алгоритму.

2. Рекурсивна функція виконує пошук першого успішного варіанта. Її результатом, зазвичай, є логічне значення. При цьому істина відповідає успішному завершенню пошуку, а неправда — невдалому. Загальна для всіх алгоритмів схема: якщо рекурсивний виклик повертає істину, то вона повинна бути негайно «передана наверх», тобто поточний виклик також повинен бути завершений зі значенням істина. Якщо рекурсивний виклик повертає неправда, то пошук повинен бути продовжений. При завершенні повного перебору всіх варіантів рекурсивна функція також повинна повернути значення «неправда». Характеристики оптимального варіанта можуть бути повернуті в глобальних даних або по посиланню.

3. При пошуку проводиться вибір найбільш оптимального серед підходящих варіантів. Зазвичай, для цього використовується мінімум або максимум якої-небудь характеристики обираного варіанта. Тоді рекурсивна функція повертає значення, яке служить оцінкою для всіх переглянутих нею варіантів, а поточний рекурсивний виклик вибирає з них мінімум або максимум з урахуванням даних поточного кроку.

Найпростішим прикладом рекурсії є лінійна рекурсія, коли функція містить єдиний умовний виклик самої себе. У такому випадку рекурсія стає еквівалентною звичайному циклу. Дійсно, будь-який циклічний алгоритм можна перетворити в лінійно-рекурсивний і навпаки. Наступні фрагменти програми реалізовані на мові програмування *C++*, дають змогу обчислювати факторіал [4].

```
// Рекурсивний алгоритм обчислення факторіала
int fact(int n)
{
if (n = = 1) return 1;
return n * fact(n–1);
}
// Циклічний алгоритм обчислення факторіала
int fact(int n)
{
for (int s = 1; n!=0; n– –) s * = n;
return s;
}
```

Структура рекурсивної функції може приймати три різні форми, приклади функцій наведені з використанням мови програмування *Pascal:*

1. Форма з виконанням дій до рекурсивного виклику (із виконанням дій на рекурсивному спуску).

```
Procedure Rec;
begin
    S;
    if умова then Rec;
```

2. Форма з виконанням дій після рекурсивного виклику (із виконанням дій на рекурсивному поверненні).

```
Procedure Rec;
begin
    if умова then Rec;
    S;
end;
```

3. Форма з виконанням дій як до, так і після рекурсивного виклику (із виконанням дій як на рекурсивному спуску, так і на рекурсивному поверненні).

```
Procedure Rec;
begin
    S1;
    if умова then Rec;
    S2;
end;
```

або

```
Procedure Rec;
begin
    if умова then
    begin
        S1;
        Rec;
        S2;
    end;
    end;
```

Всі форми рекурсивних процедур знаходять застосування на практиці. При розв'язанні багатьох завдань, у тому числі обчислення факторіала, неважливо, яка використовується форма рекурсивної процедури. Проте є класи завдань, при розв'язанні яких програмісту потрібно свідомо управляти ходом роботи рекурсивних процедур і функцій.

Також необхідно надати інформацію про трудомісткість рекурсивних алгоритмів. Трудомісткість — це залежність часу виконання алгоритму від розмірності вхідних даних. У рекурсивних функціях розмірність вхідних даних визначає глибину рекурсії, де глибина рекурсії — це максимальний ступінь вкладеності викликів функцій у ході обчислення. У загальному випадку глибина буде залежати від даних, що вводяться. Якщо мається розгалужена рекурсія — цикл із m повторень, то при глибині рекурсії N загальна кількість рекурсивних викликів буде порядку mN, оскільки з кожним кроком рекурсії вона збільшується в m раз. Показо-

вий характер функції говорить про те, що трудомісткість рекурсивних алгоритмів значно перевищує трудомісткість відомих алгоритмів сортування і пошуку.

Рекурсивний спуск і рекурсивне повернення розглянемо на прикладі обчислення факторіалу.

5.3. Виконання дій на рекурсивному спуску

Для реалізації універсального алгоритму обчислення факторіала, що діє на спуску, у рекурсивну функцію потрібно додатково ввести два параметри:

Mult — для виконання рекурсивного виклику (на спуску) операції добутку накопичуваного значення факторіала на черговий множник;

m — для забезпечення незалежності рекурсивної функції від імені конкретної глобальної змінної, тобто для підвищення універсальності функції.

Програма *Factorial_Down* виконана на мові програмування *Pascal*, яка використовує рекурсивну функцію *Fac_Dn*, що виконує обчислення на спуску, має такий вигляд:

```
Program Factorial_Down;
Var
n:Integer;
function Fact_Dn (Mult: LongInt; i, m: Integer): LongInt;
begin
 Mult:=Mult*i; {Накопичення факторіала стоїть до}
        {оператора рекурсивного виклику. }
       {Отже обчислення виконується}
        { на спуску.       }
 if i=m then Fact_Dn:=Mult;
  else Fact_Dn:=Fact_Dn (Mult, i+1, m);
```

```
end;
begin
Write ('Введіть число n:');
ReadLn(n);
WriteLn('Факторіал n! =',Fact_Dn(1,1,n));
end.
```

Для демонстрації виконуваних функцією *Fact_Dn* дій приведемо таблицю трасування її параметрів по рівнях рекурсії.
У табл. 5.1 розглянемо конкретний випадок для *n*=5.

Таблиця 5.1

Поточний рівень рекурсії	Рекурсивний спуск	Рекурсивне повренення
0	Введення *(n=5)* *Fact_Dn(1,1,5);*	Вивод *n! =120*
1	*Mult:=1*1(1); i=1; Fact_Dn(1,2,5);*	*Fact_Dn:=120;*
2	*Mult:=1*2(2); i=2; Fact_Dn(2,3,5);*	*Fact_Dn:=120;*
3	*Mult:=2*3(6); i=3; Fact_Dn(6,4,5);*	*Fact_Dn:=120;*
4	*Mult:=6*4(24); i=4; Fact_Dn(24,5,5);*	*Fact_Dn:=120;*
5	*Mult:=24*5(120); i=5; Fact_Dn:=120;*	*Fact_Dn:=120;*

5.4. Виконання дій на рекурсивному поверненні

Розглянута на початку програма *Factorial*, що використовує рекурсивну функцію *Fact*, виконує обчислення факторіала на поверненні. Але це не зовсім очевидно, оскільки у функції *Fact* рекурсивний виклик і операція множення сполучені в однім опера-

торі присвоювання. Для більш зрозумілої демонстрації роботи на поверненні, приведемо програму *Factorial_Up*, що використовує функцію *Fact_Up*, у якої рекурсивний виклик і оператор накопичення факторіала розділені явно.

```
Program Factorial_Up;
Var
n:Integer;
function Fact_Up (i: Integer):LongInt;
Var
Mult: LongInt;
Begin
if i=1 then Mult:=1
      else Mult:=Fact_Up(i–1);
Fact_Up:=Mult*i; {Накопичення факторіала стоїть післяоператора рекурсивного виклику. Отже обчислення виконується на поверненні. }
 end;
begin
Write (‘Введіть число n:’);
ReadLn(n);
WriteLn(‘Факторіал n! =’,Fact_Up(n));
 end.
```

В табл. 5.2 представлено трасування по рівнях рекурсії, для функції *Fact_Dn*.

Таблиця 5.2.

Поточний рівень рекурсії	Рекурсивний спуск	Рекурсивне повернення
0	Введення *(n=5)* *Fact_Up(5)*	Вивод*: n! =120*
1	*i=5;* *Mult:=Fact_Up(4);*	*Fact_Up:=24*5 (120);*
2	*i=4;* *Mult:=Fact_Up(3);*	*Fact_Up:=6*4 (24);*
3	*i=3;* *Mult:=Fact_Up(2);*	*Fact_Up:=2*3 (6);*
4	*i=2;* *Mult:=Fact_Up(1);*	*Fact_Up:=1*2 (2);*
5	*i=1;* *Mult:=1;*	*Fact_Up:=1*1 (1);*

5.5. Виконання дій як на рекурсивному спуску, так і на рекурсивному поверненні

Третю форму рекурсивних підпрограм покажемо на прикладі наступного завдання.

Завдання:

Вивести на друк символи введеного рядка *'HELLO'* в зворотному напрямку.

Рішення цього завдання виконано у вигляді показаної нижче програми *Reverse_String*, виконаної на мові програмування *Pascal,* що використовує рекурсивну процедуру *Reverse*. Нагадаємо, що функція повертає значення, що дорівнює *True*, коли зчитується останній символ рядка і значення, що дорівнює *False*, якщо рядок ще не закінчився.

```
Program Reverse_String;
Procedure Reverse;
Var  Ch:Char;
begin
if not eoln then
begin
    Read(Ch);      Reverse;      Write(Ch);
end;
end;
begin
Reverse;
end.
```

Якщо після запуску програми на виконання в якості вхідного рядка ввести слово *‘HELLO’*, то таблиця трасування (табл. 5.3.) по рівнях рекурсії буде мати наступний вигляд:

Таблиця 5.3.

Поточний рівень рекурсії	Рекурсивний спуск	Рекурсивне повернення
0	Reverse;	Вивод: ‘H’;
1	eoln=False; Введення:’H’; Reverse;	Вивод: ‘E’;
2	eoln=False; Введення:’E’; Reverse;	Вивод: ‘L’;
3	eoln=False; Введення:’L’; Reverse;	Вивод: ‘L’;
4	eoln=False; Введення:’L’; Reverse;	Вивод: ‘O’;
5	eoln=False; Введення:’O’; Reverse;	Вивод: ‘H’;
6	eoln=True;	

5.6. Приклади використання рекурсії в *Pascal* та *C++*

Наприкінці дев'ятнадцятого сторіччя в Європі з'явилася гра Ханойські вежі. Популярності гри сприяли розповіді про те, що цією грою зайняті служителі храму брахманів і, що завершення гри буде означати кінець світу. Задача складається з наступного: дано три стовпчики *A, B, C*. На стовпчику А знаходяться чотири диски різного діаметра, причому вони розташовуються так, що кожний менший диск знаходиться на більшому (рис. 5.1).

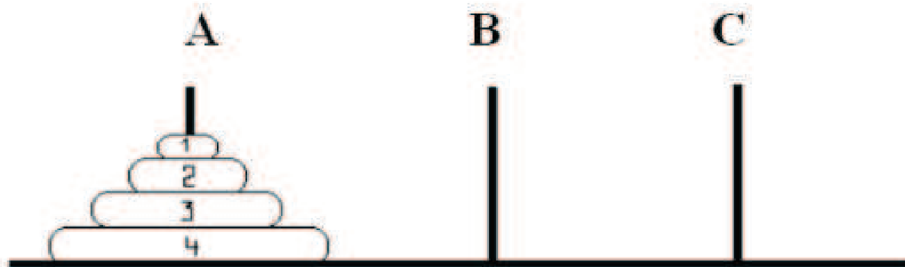


Рис. 5.1. Графічне подання задачі про Ханойські вежі

Мета гри — перенести вежу з лівого стрижня на правий, при чому за один раз можна переносити тільки один диск, крім того, забороняється розміщувати більший диск на менший. Для реалізації створеного алгоритму розв’язання цієї задачі будемо використовувати мову програмування *Pascal*.

Для визначення підходу до розв’язання поставленого завдання, розглянемо більш загальний випадок із n дисками. Якщо ми зможемо сформулювати рішення для n дисків у термінах рішення для n–1 диска, то дана проблема буде вирішена, оскільки завдання для n–1 дисків можна буде, в свою чергу, вирішити в термінах n–2 дисків і так далі до випадку одного диска. А для випадку одного диска (n=1) рішення елементарне. Потрібно просто перенести єдиний диск із стовпчика *A* на стовпчик *C*.

Таким чином, якщо сформулювати рішення для n дисків у термінах n–1 диска, то ми фактично одержуємо алгоритм рекурсивної процедури, за допомогою якої можна легко досягти мети даного завдання.

Розглянемо словесний опис такого алгоритму.

if n = 1 (then)

1. Перемістити цей єдиний диск із стовпчика *A* на стовпчик *C* і зупинитися.

(else)

2. Перемістити верхні n–1 диск із стовпчика *A* на стовпчик *B*, використовуючи стовпчик *C* як допоміжний.
3. Перемістити нижній диск, що залишився, із стовпчика *A* на стовпчик *C*.
4. Перемістити n–1 диск із стовпчика *B* на стовпчик *C*, використовуючи стовпчик *A* як допоміжний.

Можна з упевненістю сказати, що така послідовність дій дасть коректне рішення для будь-якого значення n. Це витікає з наступного. Якщо n=1, то коректність очевидна. Якщо n=2, то ми знаємо, що вже маємо рішення для випадку (n–1)-го диска, що дорівнює 1. Аналогічно, якщо n=3, ми знову маємо рішення для (n–1)-го диска, що дорівнює 2. Подібно можна показати, що це рішення працює для n=1, 2, 3, 4... і якогось іншого n.

Наведемо програму *Hanoi_Towers*, що розв'язує поставлене завдання за допомогою рекурсивної процедури *Move_Disks.*

```
Program Hanoi_Towers;
Uses WinCrt;
Var
n:Integer;
Procedure Move_Disks (n:Byte; Source, Dest, Tmp: Char);
{n — кількість дисків на стовпчику Source       }
{Source — вихідний стовпчик            }
```

```
{Dest — стовпчик, на який потрібно переставити диски}
{Tmp — Допоміжний стовпчик          }
    begin
    if n=1 then WriteLn('Переставити диск номер 1 із стовпчи-
ка',Source, 'на стовпчик', Dest)
     else
     begin
     {Переставляємо n – 1 верхніх дисків із вихідного стовпчика
на допоміжний, використовуючи цільовий диск як проміжний}
    Move_Disks (n – 1, Source, Tmp, Dest);
    WriteLn('Переставити диск номер',n:1, ' із стовпчика',
Source, ' на стовпчик', Dest);
    {Переставляємо усі n – 1 диски, розташовані на допоміж-
ному стовпчику, на цільовий, використовуючи вихідний диск як
проміжний}
    Move_Disks(n – 1,Tmp, Dest, Source);
    end;
    end;
    Begin
    ClrScr;
     Write('Введіть число дисків: ');
    ReadLn(n);
       WriteLn;
     WriteLn('Послідовність інструкцій для рішення завдання: ');
     WriteLn;
    Move_Disks(n, 'A', 'C', 'B');
    End.
```

Результат роботи програми *Hanoi_Towers* для кількості вихідних дисків на стовпчику *A*, що дорівнює 4 наведено на рис. 5.2. Аналізуючи результати розглянутого завдання, слід зазначити аналогію між методом математичної індукції, рекурсивним методом математичної індукції і рекурсивним методом про-

грамування. Дійсно, спочатку було сформульоване рішення для одного диска. Потім було припущено існування рішення для $(n-1)$-го диска, і на основі цього припущення побудована процедура переміщення n дисків, що викликає сама себе.

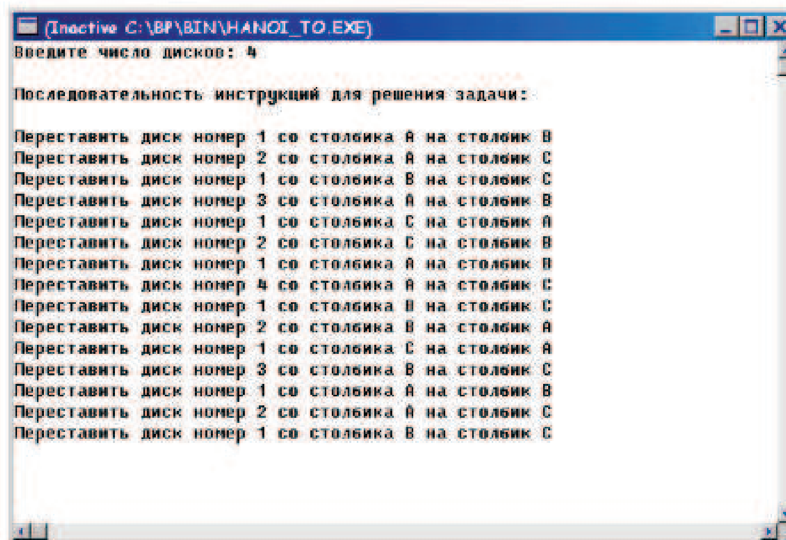


Рис. 5.2. Екрана форма виводу результатів розв'язання задачі про Ханойські вежі

Зазначена аналогія показує потужність рекурсивних методів програмування тому, що з їхньою допомогою вдається в природному стилі програмувати розв'язання як алгоритмів, побудованих по рекурентним формулам, так і алгоритмів більш загального характеру таких, як завдання про Ханойські вежі і швидке сортування.

Реалізація методу швидкого сортування масиву є прекрасним прикладом використання рекурсії. Цей метод був розроблений у 1962 р. професором Оксфордського університету К. Хоаром.

Наступним прикладом використання рекурсії буде розв'язання задачі про вісім ферзів при використанні мови програмування *С++*. Мета: розташувати вісім ферзів на шаховій дошці так, щоб вони не перебували друг у друга «під боєм». Опишемо алгоритм розв'язання вказаної задачі.

1. Оскільки ферзі «б'ють» по вертикалі (тобто на кожної вертикалі їх не більш одного), то крок рекурсії полягає у виставлянні ферзів на чергову вертикаль. Інваріант процесу — перші *i*–1 ферзів уже коректно виставлені, крок додає ще одного ферзя, зберігаючи коректність. Формальний параметр кроку — номер вертикалі (*i*), фактичний параметр рекурсивного виклику — номер наступної вертикалі (*i*+1). Алгоритм шукає перше підходяще розміщення й повертає логічне значення — розміщення знайдено (1) або не знайдено (0). Загальні дані представляють собою дошку із уже виставленими ферзями, досить мати одномірний масив, індекс у якому позначає позицію ферзя по вертикалі, а значення — позицію по горизонталі.

```
int R[8];
int step(int i)
{
... step{i + 1);
}
```

2. Перебір варіантів полягає в послідовнім виставленні чергового ферзя на всі вісім кліток вертикалі. Якщо після виставлення він перебуває під боєм, клітка пропускається. Якщо ні, то проводиться спроба викликом рекурсивної функції виставити наступного за ним. Схема пошуку першого підходящого варіанта говорить про те, що при позитивному результаті рекурсивного виклику необхідно перервати пошук і повернути цей варіант. У протилежному випадку — перебір триває. По закінченню перегляду — повернути 0.

```
int R[8];
int step(int i){
for (int j=0; j<8; j++){
R[i]=i;
if (!TEST(i)) continue;       // Під боєм — пропустити
if (step(i + 1)) return 1;    // Ланцюг добудовано — вийти
}
return 0;}                    // Цикл завершено — невдача
```

3. Оскільки кожний ферзь «виставляється» в глобальному масиві, то по завершенню ланцюжка «успішних» виходів з рекурсивних викликів у ньому й буде перебувати перший підходящий варіант. Ферзі вважаються успішно виставленими, якщо рекурсивна функція досягає неіснуючої вертикалі. Ця перевірка повинна бути зроблена в самому початку тіла функції. Функція *TEST* перевіряє знаходження *i*-го ферзя з усіма попередніми ферзями на одній горизонталі й діагоналі. Спочатку функція викликається для *i*=0.

Лістинг програми, створеної на основі розробленого алгоритму розв'язання задачі, наведено нижче [8].

```
// Задача про вісім ферзів
int R[8];
int TEST{int i){
for (int j = i – 1 ; j>=0; j – – ){
jf(R[i] = = R[j]) return 0;               // По горизонталі
if(abs(R[i]–R[j])==i–j) return 0;         // По діагоналі
}
return 1; }
int step(int i){
if (i = = 8) return 1;
for (int j = 0; j<8; j + + ){
R[i]=j;
if (!TEST(i)) continue;              // Під боєм — пропустити
if (step(i + 1)) return 1;      // Ланцюг добудовано — вийти
}
return 0;}                      // Цикл завершено — невдача
#include <stdio.h>
void main(){ step(0);
for (int i=0; i<8; !++) printf("%d ".R[!]);
printf("\n");
 }
```

Рекурсія також може бути дуже корисною при розв'язанні задач пошук найкоротшого шляху в графові або відстані між містами. Ця задача в загальному випадку має наступне формулювання: відстані між містами задані матрицею. Для кожної пари міст *i, j* елемент *R*(*i, j*) матриці містить значення відстані між ними або 0, якщо вони не зв'язані безпосередньо (матриця симетрична відносно головної діагоналі). Реалізацію алгоритму розв'язання цього завдання будемо провадити за допомогою мови програмування *C++*. Для початку — потрібно знайти значення мінімального шляху між двома заданими містами. Схема рекурсивного процесу пошуку для цього завдання принципово не відрізняється від попередніх. Крок рекурсії — переміщення з поточного міста в сусідній у пошуках шляху. Формальні параметри функції (початковий стан процесу) — індекс поточного міста в матриці відстаней і індекс міста-мети. «Успішне обмеження рекурсії» — формальні параметри збігаються. Функція містить цикл перебору всіх сусідів і рекурсивного виклику для кожного з них, якщо між ними є прямій шлях. «Зациклення» запобігається міткою пройдених міст.

```
#define N 5
int
R[N][N]={{0,4,2,0,0},{4,0,0,1,3},{2,0,1,0.6},{0,0,3,0,0},{0,0,6,0,0}};
int M[N] = {0.0,0.0,0};
... step(int src, int dst){
if (src ==dst) return...
if (M[src] = = 1) return...
M[src] = 1;
for (int i=0; i<N; i++){
if (R[src][i]==0) continue;
... step(i,dst);
}
M[src]=0;
return ...; }
```

Далі випливають особливості оптимального пошуку. Насамперед, рекурсивний процес забезпечує повний перебір. Рекурсивний виклик повертає оптимальне значення — мінімальна відстань від поточного міста до міста-мети, або — 1, якщо шлях відсутній. Поточний крок рекурсії повинен зберегти цей інваріант, отриманий від сусідів. Для цього він додає до кожного припустимому (не рівному 1) результату рекурсивного виклику відстань від поточного міста до сусіда й вибирає з них мінімальний, повертаючи в якості «свого» результату.

Лістинг програми визначення мінімального шляху між містами має наступний вигляд [8]:

```
#define N 5
int R[N][N] =
{{0,4,2,0,0},{4,0,0,1,3},{2,0,0,0,6},{0,1,0,0,0},{0,3,6,0,0}};
int M[N] = {0,0,0,0,0};              // Мітка пройдених міст
int step(int src, int dst){
if (src ==dst) return 0;             // Успіх від мети до мети 0
if (M[src] = = 1) return – 1 ;       // Повторне проходження  –1
M[src] = 1;
int min = – 1 ;                      // Мінімальний шлях від src до dst
for (int 1=0; i<N; !++){
if (R[src][i]==0) continue; // Сусіди не зв'язані – пропустити
int x=step(i,dst);                   // Результат від сусіда до мети
if (x= = –1) continue;               // Шлях не знайдено – пропустити
x+ = R [ s r c ] [ i ] ;             // Додати відстань до сусіда
if (min = = –1 II x < min)           // Зафіксувати мінімум
min=x;}
M[src]=0;                            // Зняти мітку
return min;}
```

Контрольні запитання та завдання для самоконтролю

1. Що таке рекурсія?
2. Властивості рекурсії.
3. Основні принципи рекурсії.
4. Що таке рекурсивна структура даних?
5. Що таке рекурсивна функція?
6. Які результати виконання рекурсивних функцій?
7. Що таке лінійна рекурсія?
8. Що таке виконання дій на рекурсивному спуску?
9. Що таке виконання дій на рекурсивному поверненні?
10. Опишіть схему виконання дій як на рекурсивному спуску, так і на рекурсивному поверненні.
11. Що таке інваріант рекурсивної функції?
12. Яка трудомісткість рекурсивних функцій?
13. Що таке глибина рекурсії?
14. Відстані між містами задані матрицею (якщо між містами i, j є прямий шлях з відстанню N, те елементи матриці $A(i, j)$ і $A(j, i)$ містять значення N, інакше 0). Написати програму пошуку мінімального шляху обходу всіх міст без відвідування двічі того ж самого міста.
15. Напишіть рекурсивну програму для обчислення lg(*!N*).
16. Укажіть глибину рекурсії алгоритму Евкліда при введенні двох послідовних чисел Фібоначчі (*FN і FN*+1).
17. Змініте для обчислення *N! mod M*, щоб переповнення більше не грало ніякої ролі.
18. Визначите вид рекурсії (лінійна, розгалужена), сформулюйте змістовний результат рекурсивного алгоритму:

Завдання №1

```
long F1 (int n)
{ if (n = = 1) return 1;
return (n * F1(n–1)); }
```

Завдання №2

```
double F2(double *pk, double x, int n)
{
if (n ==0) return(*pk);
return *pk + X *F2(pk + 1 ,x,n–1);
}
void z3()
{ double B[] ={ 5.,0.7,4.,3. } ,X=3., Y;
Y = F2(B,X,4); }
```

Завдання №3

```
void F3(int in[], int a, int b)
{ int i, j, mode;
if (a> = b) return;
for (i=a, j = b, mode=1; i != j ; mode >0 ? i++ : j ––)
if (in[i] > in[j])
{ int c;
c = in[i]; in[i] = in[j]; in[j]=c; mode = –mode;
}
F3(in,a,i–1); F3(in,i + 1 ,b);
}
```

Завдання №4

```
char *F4(char *p, char *s) {
if ( *s =='\0') return p;
*p++ = *s;
p=F4(p, s + 1);
*p++ = *s;
return p;}
void z4()
{ char *q, S[80];
*F4(S, "abcd")=0; }
```

Завдання №5

```
void F5(char *&p, char *s){
if ( *s = = '\0') return;
*p++ = *s;
F5(p, s + 1);
*p++ = *s; }
void z5()
{ char *q, S[80];
q = S; F5(q,"abcd"); *q = 0; )
```

Розділ 6. ЛОГІЧНІ СТРУКТУРИ ТА СПОСОБИ ОПРАЦЮВАННЯ ФАЙЛІВ

6.1. Введення

Перед тим, як ознайомитися з логічними структурами і способами опрацювання файлів необхідно ознайомитись з поняттям файлу. Файл — це сукупність записів одного типу. Файл визначається як набір однотипних записів, збережений на зовнішніх пристроях, що запам'ятовують. Типовими зовнішніми пристроями, що запам'ятовують, є дисководи з магнітними дисками. Запис є змістовною одиницею збереженої у файлі інформації. Розмір запису залежить від розв'язуваної задачі. Логічний запис — це сукупність даних про деякий об'єкт. Він утворюється з полів або елементів даних, що вказують на різноманітні атрибути, властиві конкретному об'єкту. Елемент даних, зазвичай, є значенням, що є частиною опису об'єкта або події. Для того, щоб мати можливість доступу в файлі до конкретного запису, кожному запису привласнюють унікальний номер або ім'я, що служить його ідентифікатором і розташовується в окремому полі. Цей ідентифікатор називають ключем запису. Елементарним прикладом записів і відповідних їм ключів є бібліотечна картотека, де записами виступають картки з даними про абонента (дані про книги, про самого клієнта), а ключем запису виступає порядковий номер абонента. Або у випадку, коли «об'єктами» для зубного лікаря є пацієнти, то кожний логічний запис повинен зберігати інформацію про пацієнта, а його полями можуть бути ім'я пацієнта, номер його лікарняної картки, вік, номер телефону та ін. У даному прикладі в якості ключа виступає номер лікарняної картки пацієнта.

Для вирішення інформаційних задач використовуються файли з різноманітною організацією доступу до даних [17]:

1. Одержати всі (або більшість) записи із файла даних. У цьому випадку з файла вилучається від 10 до 100 % усіх записів. Найбільш ефективною для опрацювання файла даних буде послідовна організація.

2. Одержати унікальний запис. Цей випадок протилежний першому. Із набору записів вилучається один запис, і тут найкраще використовувати пряму організацію (функції хешування і т.п.).

3. Одержати деякі записи. З файла вилучається від 1 до 10 % записів. Найбільш ефективна для використання індексно-послідовна організація.

У сучасних обчислюваних системах (ОС) найбільш широко використовуються послідовні, довільно організовані й індексно-послідовні типи логічних структур файлів. У деяких ОС використовуються також і інші типи файлових структур. Дуже часто спеціальні файлові структури реалізуються за допомогою комбінації або модифікації головних типів файлових структур. Надалі для стислості замість терміна «тип логічної структури файла» будемо вживати «тип файла» або «спосіб організації файла».

6.2. Стислі характеристики логічних структур

Послідовний тип файла або послідовний файл характеризується тим, що фізичне проходження його записів на пристрої зовнішньої пам'яті цілком збігається з логічним порядком проходження записів. Для доступу до деякого внутрішнього запису послідовного файла необхідно переглянути файл із самого початку доти, поки не буде знайдено необхідний запис. Деякі пристрої зовнішньої пам'яті допускають тільки послідовну організацію файла. Послідовна організація має дві особливості. Перша — удосконалення організації полягає в тому, що запис даних упорядковується у визначеній послідовності; друга — в тому, що атрибути даних розподілені по категоріях так, що окремі записи містять зна-

чення всіх атрибутів даних у тому самому порядку і, можливо, в одній і тій же позиції. У цьому випадку імена атрибутів даних в описі файла повинні зустрічатися тільки один раз.

Замість того, щоб зберігати пари «ім'я атрибута-значення», із кожним ім'ям зв'язується повний набір (стовпчик) значень. Така організація схожа на добре відому табличну організацію, що зазвичай використовується при виводі даних. Записи тут мають фіксовану довжину.

Послідовні файли найбільш часто використовуються при вирішенні комерційних задач у пакетному режимі типів файлів. Для того, щоб об'єднати дані з декількох послідовних файлів і логічно упорядкувати їхні записи, виконуються операції сортування. Тоді всі необхідні дані можуть бути знайдені шляхом послідовного перегляду заданих файлів. Незручність полягає в тому, що файли можуть бути упорядковані тільки відповідно до одного первинного ключа. Тому часто виникає необхідність у повторному сортуванні такого файла по іншому ключу із тим, щоб об'єднати інші набори файлів. У тих випадках, коли дані опрацьовуються тільки періодично, як, наприклад, при щомісячному упорядкуванні накладних, послідовні файли більш вигідніші ніж інші з погляду витрат. У залежності від виконуваної операції ефективність використання послідовних файлів може бути як дуже високою, так і неприпустимо низькою.

Довільно організований файл (у деяких джерелах він зветься рандомізованим файлом) характерний тим, що його записи можуть бути розміщені в зовнішній пам'яті в довільному порядку, причому доступ до будь-якого запису файла може бути виконано безпосередньо по його фізичній адресі, без необхідності перегляду попередніх записів. Внаслідок можливості безпосереднього або прямого звернення до будь-якого запису такий файл називається також файлом прямого доступу. Він може розміщатися тільки на пристрої прямого доступу, що запам'ятовує (ППДЗ). Файл прямого доступу дозволяє найбільш тісно зв'язати аргумент

пошуку, використовуваний для вибірки запису, із фізичними можливостями пристроїв прямого доступу. Зміст прямого доступу до файла полягає в тому, що заздалегідь відомо, де знаходяться дані на пристрої. Вперше файли з прямим доступом знайшли застосування в електромеханічних машинах, у яких відперфороване на карті число використовувалося для того, щоб визначити, де зберігається залишок інформації, що утримується на карті. Метод прямого доступу забезпечує швидкий пошук запису, оскільки він дозволяє уникнути виконання проміжних операцій над файлом, але пошук ведеться лише по одному ключовому атрибуту. При цьому не потрібно, щоб дані були пов'язані з попередніми записами.

Файли прямого доступу часто використовуються для упорядкування довідників, таблиць вартостей, розкладів, списку імен і т. д. Пряма організація файлів особливо зручна в тих випадках, коли записи мають невеличкі розміри і фіксовану довжину, коли необхідний швидкий доступ і, коли доступ до даних завжди здійснюється просто. Крім того, файли прямого доступу часто використовуються в якості допоміжних елементів у більш складних організаціях файлів.

В індексно-послідовному файлі записи можуть бути розміщені фізично в довільному порядку, але логічний порядок проходження записів визначається ключами записів. Такий факт завжди доповнюється одним або декількома пов'язаними індексами, кожний з яких є таблицею, що містить ключі і фізичні адреси записів файла. Важлива особливість індексно-послідовного файла — можливість здійснювати як прямий, так і послідовний доступ до запису. При прямому доступі спочатку по ключу запису визначається індексна фізична адреса запису, після чого звернення до неї здійснюється як у файлі прямого доступу. При послідовному доступі в індексі проглядається послідовність ключів у їхньому логічному порядку і відбувається звернення до записів у порядку перегляду їх ключів. Індексно-послідовний файл завжди розміща-

ється на ППДЗ. Поява індексно-послідовних файлів — це прагнення перебороти недоліки послідовної організації файла, що виявляються при доступі до даних, не гублячи при цьому всіх переваг цієї організації. Одна відмінна риса індексно-послідовних файлів полягає в наявності індексу, що дозволяє здійснювати менш упорядкований доступ до записів; інша особливість полягає в наявності засобів опрацювання доповнень до файла.

Індексно-послідовні файли визначеного типу широко використовуються при опрацюванні економічної інформації. Особливо часто вони використовуються в тих випадках, коли тимчасові інтервали, протягом яких необхідно зберегти найбільш нову копію файла менші, ніж інтервали опрацювання, припустимі при періодичній реорганізації послідовних файлів (інтервал опрацювання визначає фактичний «час життя» конкретної структури деякого файла, після чого проводиться реорганізація і файл набуває іншої структури). Наприклад, для упорядкування списку товарів індексно-послідовний файл може використовуватися щодня, а його реорганізація (разом із процесом, що формує замовлення на товари, запасів яких не достає) проводиться щодня.

Індексно-послідовні файли також широко використовуються для опрацювання інформаційних записів, але при цьому потрібно, щоб у запису був визначений ключовий атрибут. Ця вимога часом призводить до того, що в окремих індексно-послідовних файлах зберігаються копії тих самих даних, але упорядкованих по різних ключах. У цьому випадку зростають витрати на відновлення і збільшується необхідний обсяг пам'яті.

Користувачі часто не розуміють призначення того або іншого варіанта організації даних. Внаслідок чого, використання, наприклад, індексно-послідовних файлів в деяких системах веде до невиправданого збільшення часу опрацювання даних. Випадки, коли нові дані надходять у файл групами, теж можуть призвести до великих витрат часу при їхньому перегляді, оскільки ланцюжки, що утворюються, можуть виявитися дуже довгими. В дійсності

групи нових даних часто приєднуються до кінця файла. Якщо їх опрацьовувати або якщо заздалегідь до реорганізації індексно-послідовного файла виділити в ньому додаткову область і резервувати значення індексу, то можна уникнути ряду проблем, що виникають при додаванні записів. У межах конкретного методу опрацювання індексно-послідовного файла можливості для модифікації режимів невеликі, проте автори програмного забезпечення і виробники комп'ютерів усе ж надають ряд додаткових варіантів опрацювання.

Необхідно зауважити, що обмеження, відповідно до якого тільки один ключовий атрибут визначає головну послідовність записів файла, є загальним для всіх файлів.

6.3. Метод доступу

Приступаючи до фізичного проектування бази даних, ми насамперед з'ясовуємо, які апаратні засоби ми будемо використовувати. Важливу роль тут відіграють пристрої зовнішньої пам'яті. З огляду на характеристики цих пристроїв ми відповідно повинні обрати найбільш ефективні. Порівнюючи логічну схему бази даних і фізичні пристрої збереження даних, ми повинні вибрати найбільш ефективний метод доступу до них. Що ж таке метод доступу? Метод доступу — це сполучення типу файла і стандартних програмних засобів, що забезпечують визначений режим передачі даних між файлом і головною пам'яттю. У методі доступу виділяються два компоненти: структура пам'яті і механізм пошуку.

Структура пам'яті задає обмеження на утворення шляхів доступу до даних. Механізм пошуку — це алгоритм, що визначає специфічний шлях доступу, що можливий у рамках заданої структури пам'яті і кількість кроків уздовж цього шляху для перебирання шуканих даних.

6.4. Опрацювання файлів

Найбільш загальною процедурою, що лежить в основі більшості операторів змістовного опрацювання даних, є пошук і вибірка записів вихідних (оброблюваних) файлів відповідно до керуючого файла і виконання деяких операцій над виділеними записами. Надалі під опрацюванням будемо розуміти реалізацію стандартної процедури пошуку і вибірки. Для вибору методів реалізації процедур пошуку і вибірки необхідно знати характеристики інформації, файлів й опрацювання.

Характеристики інформації — це обсяг, активність і динамічність.

Обсяг оброблюваного файла визначається кількістю записів, що зберігаються в ньому. Чим більший обсяг, тим у більшій мірі правильний вибір методу організації файла й алгоритму опрацювання впливає на ефективність.

Активність записів визначається частотою звернення до них при опрацюванні. Чим більша активність, тим вищі вимоги до організації даних і алгоритму їхнього опрацювання. Процедура опрацювання повинна будуватися так, щоб полегшити пошук і вибірку найбільш активних записів.

Динамічність визначається частотою й обсягом операцій по зміні, видаленню і додаванню записів. Висока динамічність є непрямою причиною частої реорганізації, що, у свою чергу, висуває визначені додаткові вимоги до ефективності організації файла й алгоритму його опрацювання.

Структура й організація конкретних машинних файлів є похідним чинником стосовно характеристик інформації і використовуваних типів фізичної пам'яті. Різноманіття методів організації файлів може бути зведене до трьох головних: простий послідовний, послідовно-каталогізований і прямий (довільний).

Оскільки в послідовному файлі записи у фізичній пам’яті розташовуються один за одним. Це цілком характеризує структуру

файла і ніякої іншої інформації про місце знаходження окремих записів немає. Така структура файла характерна для інформації з високою активністю записів і її рівномірного розподілу, а також для пам'яті з послідовним доступом.

В індексно-послідовних файлів записи в пам'яті розташовуються, як правило, послідовно. Проте, місце розташування окремих груп записів визначається за допомогою додаткової інформації, організованої у вигляді каталогу, що зв'язує ключові ознаки (індекси) початкових записів кожної групи з їхніми адресами пам'яті. Таку структуру застосовують до малодинамічної інформації з невисокою активністю записів при підвищених вимогах до швидкості їхнього опрацювання.

У файлі з прямою організацією існує визначене співвідношення між ключовими ознаками кожного запису і його адресою в пам'яті. Це співвідношення, що дає практично повну інформацію про розташування кожного запису, задається заздалегідь установленою (при первинній організації файла) функціональною залежністю «ключова ознака — адреса». Таку структуру застосовують при малодинамічній інформації або при особливо високих вимогах до швидкості опрацювання записів.

Вибір методів опрацювання (пошуку і вибірки записів) залежить від методів організації файла і характеристик інформації.

В організації процесу опрацювання істотне значення мають:

1. Характер вибірки записів з оброблюваного файла;
2. Порядок витягання записів з оброблюваного файла.

У першому випадку можливе послідовне і довільне опрацювання. При послідовному опрацюванні в першому випадку записи файла вибираються в ключовій послідовності, тобто в порядку зростання ключової ознаки. У випадку, коли записи неупорядковані ні по яких реквізитах, ключовою ознакою запису є його порядковий номер. При довільному опрацюванні характер вибірки записів відмінний від ключової послідовності.

У другому випадку можливе упорядковане опрацювання, коли упорядковують керуючий файл, тобто управляють вибіркою записів з головного файла, і неупорядковане, коли порядок вибірки визначається жорстокою послідовністю записів керуючого файла.

Таким чином, можна виділити чотири загальні методи опрацювання: послідовно-упорядкована, послідовно-неупорядкована, довільно-упорядкована, довільно-неупорядкована.

6.4.1. Опрацювання файлів із послідовною організацією

Послідовна організація файлів є найбільш простою. Для пристроїв пам'яті з послідовним машинним доступом (типу магнітних стрічок, перфострічок, перфокарт) вона є практично єдино можливою, але застосовується також для пам'яті з довільним доступом. Наприклад, на магнітних дисках записи послідовно-організованого файла розташовуються один за одним у порядку зростання номерів записів, доріжок, циліндрів. При постійній довжині записів послідовна структура характеризується таким співвідношенням:

$$Xij = Xio + JK3,$$

де Xij — відносна адреса i-го реквізиту запису; $K3$ — константа, що визначає довжину запису в обраних одиницях вимірювання (байт, слово) [18].

Для спрощення задачі аналізу методів опрацювання припустимо r=1. Оброблюваний файл (обсягом N записів) будемо називати головним файлом, список K необхідних записів головного файла рівні відношенню K і N.

Для вибору методу опрацювання істотне значення мають організація формування керуючого файла і припустимий час затримки результату опрацювання. Зокрема, якщо надходження записів у керуючий файл розподілено в часі і немає можливості

очікувати його накопичення й упорядкування, потрібно застосувати неупорядковане опрацювання.

Послідовно-неупорядковане опрацювання полягає в тому, що з керуючого файла витягається ідентифікатор (ключ) чергового шуканого запису, що шукається в головному файлі методом послідовного перебору й оброблювання. Потім виконується перехід до шуканого ключа і т. д.

У середньому на один ключ керуючого файла, тобто на опрацювання одного активного запису, потрібно $(N+1)/2$ вибірок і порівнянь записів головного файла, тому послідовно-неупорядковане опрацювання застосовується рідко.

Якщо головний файл розміщується в пам'яті з довільним доступом і його записи розташовані в ключовій послідовності (упорядковані по ідентифікатору) можливе довільне опрацювання. У цьому випадку шукані записи по черзі знаходяться методом дихотомії (розподіл навпіл) або іншим методом, що не потребує послідовної вибірки записів головного файла. При дихотомічному пошуку питома кількість переглянутих записів головного файла дорівнює $\log_2 N$. Може застосовуватися і комбінація методів довільного і послідовного опрацювань. Наприклад, у пам'яті вінчестеру циліндр із шуканим записом може відшукуватися методом дихотомії, а сам запис у межах циліндра — методом послідовного перебору.

У цілому, неупорядковане опрацювання є нетиповим випадком для послідовної організації. Зазвичай, застосовується упорядковане опрацювання, коли головний файл упорядковано в одній і тій же ключовій послідовності (синфазно). Проте переваги упорядкованого опрацювання не мають абсолютного характеру. У деяких випадках при правильно організованому обміні з пам’яттю неупорядковане опрацювання може виявитися ефективніше.

6.4.2. Опрацювання індексно-послідовних файлів

Сутність послідовно-каталогізованої організації файла полягає в наступному. Сукупність записів файла, розташованих у пам'яті в ключовій послідовності, попередньо ділиться на групи. Результати розбивки відбиваються в багаторівневому каталозі, що фіксує інтервали відповідних значень, ключів. Позначимо через *g* кількість рівнів у каталозі, через *mi* — деяке число, що показує, скільки інтервалів *i*-го рівня зберігаються в одному інтервалі (*i*–1)-го рівня. Таким чином, перший рівень каталогу відбиває результати розбивки файла на *m*1 груп першого рівня розбивки обсягом *N/m*1, другий рівень каталогу відбиває результати розбивки кожної групи першого рівня на *m* підгруп обсягом *N/m*1*m*2 і т.д [19].

Структура каталогу може бути описана плоским графом, що має мережу вузлів, розміщених по рівнях ієрархії і пов'язаних гілками. Кожна гілка *i*-го рівня графа відповідає інтервалу значень ключів у групі файлів, що містить запис. Кожному вузлу, крім вузлів *g*-го рівня, відповідає адреса в каталозі початку підгрупи інтервалів ключів поточного рівня, вузлам *g*-го рівня відповідають початкові адреси мінімальних груп файла. Процес пошуку запису, за допомогою такого каталогу, складається з етапів послідовного порівняння заданого ключа з інтервалами значень ключів гілки *i*-го рівня і визначення адреси відповідної групи інтервалів (*i*+1)-го рівня. При виконанні умови *i*=*g* з останнього рівня каталогів зчитується початкова адреса мінімальної групи файлів, що містять записи, серед яких шуканий знаходиться «звичайними» методами пошуку, наприклад, методом перебору.

Практично кожна група інтервалів *i*-го рівня каталогу містить, зазвичай, *vi*=*mi*–1 пар «адреса — ключ — адреса», оскільки результати аналізу останнього ключа в групі інтервалів можуть однозначно визначати адреси однієї з двох підгруп інтервалів поточного рівня.

Оскільки каталог і файл можуть знаходитися в різноманітних типах пам'яті, кількість порівнянь у загальному випадку не може служити адекватним критерієм трудомісткості опрацювання.

При організації індексно-послідовного файла та при визначенні структури і розміщення каталогу на практиці необхідно:

- забезпечити мінімальний час доступу (середнє або максимальне) при заданому обсязі пам'яті каталогу;

- забезпечити мінімальний обсяг каталогу при заданому часі пошуку і вибірки;

- раціонально вибрати обмеження по обсягу пам'яті або часу доступу, тобто побудувати залежність мінімального часу доступу від обсягу і структури каталогу.

Розглянемо два варіанти організації індексно-послідовного файла.

Перший варіант — це суперфайл (бібліотека файлів), розміщений у зовнішній пам'яті, і каталог, що визначає адреси файлів у пам'яті.

Другий варіант є файлом на магнітних дисках з індексно-послідовною організацією.

Файл із послідовно-каталогізованою структурою, на відміну від простого послідовного, практично однаково добре пристосований як для послідовного, так і для довільного опрацювання.

Розглянемо три методи упорядковано-довільного опрацювання, ефективні для пам'яті із лінійно-послідовними властивостями доступу (типу магнітних дисків і стрічок). Ціль упорядковано-довільного опрацювання полягає в скороченні загальної довжини блукань механізму, що при неупорядкованому опрацюванні має хаотичний характер.

Перший найбільш простий метод полягає в упорядкуванні керуючого файла в ключову послідовність, що і головний файл, і застосуванні звичайної схеми ієрархічного пошуку по каталогу, тобто стандартного методу доступу. Завдяки синфазній упорядкованості прямування механізму зчитування з головного файла отримується

поступовий односпрямований характер. Застосування цього методу ефективне тільки тоді, коли рівні каталогу не знаходяться на одному пристрої з головним файлом. Якщо позначити максимальну довжину лінійного переміщення механізму доступу через L, то для неупорядкованого опрацювання K активних записів потрібно сумарне переміщення, що дорівнює $0,33LK$. Упорядковане опрацювання виконується за один новий прогін механізму. При цьому на один активний запис припадає довжина пошуку, приблизно рівна L/K.

Другий метод відрізняється від першого тим, що пошук по каталогу починається не з самого верхнього рівня, а з деякого проміжного j, номер якого залежить від активності файла. Значення j вибирається таким чином, щоб при даній загальній активності файла майже всі гнізда рівня j були активними, тобто до відповідних інтервалів ключів (підгруп головного файла) входили активні записи. Отже, чим більше активність, тим більше повинно бути значення j. Цей метод майже цілком аналогічний методу довільно-послідовного опрацювання послідовного файла.

Третій метод упорядкованого опрацювання базується на можливості повного урахування особливостей конкретного розміщення активних записів за допомогою каталогів і застосування оптимальних стратегій їх вибірки. Такий метод ефективний, коли каталог розміщається цілком в оперативній пам'яті. Прикладом реалізації методу є групова вибірка файлів у суперфайлі записаному на магнітній стрічці.

При оптимальній стратегії вибірки отримують значно кращі характеристики опрацювання, ніж при довільній неупорядкованій вибірці.

6.4.3. Опрацювання файлів із довільною організацією

Сутність довільної організації полягає в тому, щоб виконувати перетворювання «ключ — адреса» не за допомогою каталогу, а за допомогою деякої процедури, що виділяє адресу по заданому ключу. Таким чином, значення «адреса — ключ —

адреса» пов'язані деякою заздалегідь установленою функціональною залежністю.

Найпростішим видом довільної структури є, так звана, пряма організація, у якій ключ використовується замість адреси, тобто функціональна залежність між ключем і адресою така, що ключ дорівнює адресі, при цьому він може бути опущений і не зберігатися в явному вигляді у записі. Простим прикладом такої організації є машинний файл, записаний в оперативну пам'ять. Тут оброблюваними файлами є машинні слова, а їхніми ключами — адреси. Пряма організація файла є ідеальною для опрацювання розташованих в порядку зростання адрес, одночасно ключі розташовуються строго в порядку зростання. Отже, і довільне, і послідовне опрацювання є максимально ефективними.

Можливості застосування прямої організації не має обмежень, крім припустимого обсягу пам'яті. У більшості випадків застосовується довільна організація з непрямою адресацією, в якій використовуються більш складні функціональні залежності «ключ — адреса», що забезпечують стискання області можливих значень адрес. Практично завжди ці залежності неоднозначні, тобто різноманітним значенням ключів можуть відповідати однакові значення адрес.

Записи, що завантажувалися у відповідні позиції першими, називаються первинними або головними. Записи, відображувані в тій же позиції, називаються синонімами. Записи, що в результаті перетворення ключа одержали зайняті адреси, називаються записами переповнювання. Отже, синоніми завжди є записами переповнювання. Проте, деякі головні записи можуть виявитися записами переповнювання, якщо їхня власна позиція зайнята чужим синонімом. Функцію перетворення «ключ — адреса» прийнято називати функцією розставляння або рандомізації.

Повний середній час пошуку і вибірки одного запису у файлі визначається, як:

$$T_{ср} = t_р + t_{ср.виб},$$

де t_p — час використання адреси, $t_{\text{ср.виб}}$ — середній час вибірки запису, пропорційний кількості порівнянь при пошуку.

Для організації завантаження і подальшого опрацювання записів переповнювання довільного файла необхідно вибрати методи рандомізації (функції розставляння, що забезпечують мінімальну кількість синонімів), а також методи завантаження і подальшого опрацювання записів переповнення.

Вибір методів рандомізації в значній мірі залежить від особливостей конкретного розподілу значення ключів. При ідеальній функції розставляння для «розріджень» у розподілі ключів не повинні генеруватися значення адрес, а для «згущень» ключів повинна генеруватися підвищена кількість адрес. Цього можна досягнути, представивши опис конкретної функції розподілу ключів у вигляді таблиці. Отже, порушується головна ціль довільної організації — отримати гарні результати довільного опрацювання без попереднього опису структури файла. Крім того, тривалість обчислення значення функції є складовою частиною загальних витрат часу на пошук і опрацювання. Тому час обчислення для занадто складних функцій розставляння може перекрити економію, одержувану за рахунок зменшення $t_{\text{ср.виб}}$. У цілому, для вибору функції розставляння характерна така тенденція. Якщо файл розташовано в оперативній пам'яті, то за інших умов доцільно застосувати більш прості і швидко обчислювальні функції, що генерують синоніми. Для зовнішнього типу магнітних дисків час вибірки одного запису порівняно великий. У даному випадку для скорочення середнього часу пошуку і вибірки доцільно функцію розставляння вибирати більш трудомістку, але ефективнішу.

Файл із довільною структурою орієнтований, в основному, на довільно-неупорядковане опрацювання. Алгоритм опрацювання складається з двох етапів: обчислення процесів і пошуку запису. Конкретна реалізація етапів і характеристики трудомісткості опрацювання залежать від обраного методу рандомізації,

прийнятого способу організації записів переповнювання, ступеня заповнення файла, типу пристрою, що запам'ятовує, в якому розташовано файл.

Послідовне опрацювання довільного файла ускладнене тому, що записи розташовані, зазвичай, не в ключовій послідовності, тому це потрібно робити методом довільного опрацювання, обираючи запис у порядку зростання значень ключів. Таким чином, у цьому випадку фактично здійснюється довільно-неупорядковане опрацювання.

Побудова функції рандомізації, що зберігає ключову послідовність адрес, дозволяє забезпечити звичайне послідовне опрацювання. Користуючись цією функцією, можна одержати задовільні результати лише при достатньо рівномірному розподілі ключів. У протилежному випадку генерується значна кількість синонімів, що погіршує характеристики довільного опрацювання.

6.5. Критерії оцінювання і вибору структур файлів і методів опрацювання

Оскільки різні типи структур файлів відрізняються різноманітною ефективністю опрацювання, формулюється постановка задачі оптимального вибору структури і методів опрацювання для стабільних і динамічних файлів. Критерієм оптимальності є середньо вагомий час опрацювання одного активного запису.

Урахування динамічності має два аспекти:

1. Визначення оптимальної періодичності реорганізації файлів;

2. Вибір структури і методів опрацювання з урахуванням витрат часу на реорганізацію.

Обидві задачі вирішуються на основі дослідження моделі, що описує узагальнену постановку задачі оптимальної реоргані-

зації. В основі моделі — система обслуговування, процес функціонування якої включає дві складові:

а) робочий процес обслуговування заявок;

б) процес розбудови системи, що викликається зверненням.

Розбудова полягає в тому, що в міру її накопичення збільшується її тривалість обслуговування заявок. У деякий момент часу провадиться реорганізація системи, що знищує розбудову і відновлює початкові властивості системи.

Контрольні запитання та завдання для самоконтролю

1. Поняття файлу і запису.
2. Охарактеризуйте послідовний файл. Сфери використання.
3. Охарактеризуйте довільно організований файл.
4. Де набули застосування довільно-організовані файли?
5. Чим характерний індексно-послідовний файл?
6. Де застосовуються файли з індексно-послідовною організацією?
7. Що таке метод доступу?
8. Основні характеристики інформації. Стислий опис.
9. Опрацювання файлів із послідовною організацією.
10. Опрацювання індексно-послідовних файлів.
11. Основні критерії використання каталогізованих файлів на практиці.
12. Опрацювання файлів із довільною організацією.
13. Що таке функція рандомізації?
14. Методи рандомізації.
15. Які існують критерії оцінки і вибору структур файлів і методів опрацювання?

Розділ 7. ПОСЛІДОВНІ ФАЙЛИ

7.1. Структура послідовних файлів

Для розв'язання задач інформаційного характеру використовуються різноманітні способи організації файлів для доступу до даних. Якщо при вирішенні задачі необхідно одержати всі або більшу частину записів файлу, то доцільно застосовувати послідовну організацію файлу. Послідовне розміщення записів допускає фізичне блокування, тим самим зменшуючи час доступу до даних. При обробці даних невеликого обсягу, коли їх значення відносно часто піддаються змінам, ефективно використовувати збереження цих даних у вигляді послідовно з'єднаних ділянок (зв'язана послідовна структура).

Послідовний файл визначається послідовним розташуванням своїх записів у порядку їхнього надходження у файл. Доступ до записів файлу такого типу здійснюється в порядку їхнього розміщення на носії інформації.

Логічне представлення послідовного файлу має вигляд (рис. 7.1):

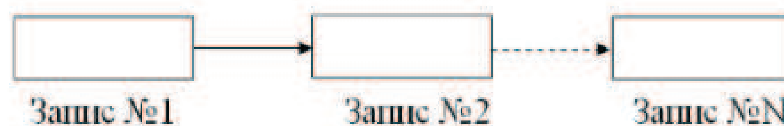


Рис. 7.1. Логічна структура послідовного файла

Як видно з рис. 7.1 усі записи послідовного файлу розташовані друг за другом, тобто кожному запису послідовного файлу, крім останнього, відповідає єдиний запис, розташований на носії інформації фізично слідом за ним. Отже, для доступу до i-го запису файлу попередньо необхідно пропустити (i–1) запис цього файлу.

При послідовній організації файлу існує дві особливості:

1) записи файлу упорядковуються у визначеній послідовності;

2) атрибути даних розподілені по категоріях так, що окремі записи містять значення всіх атрибутів даних у тому ж порядку й в одній і тій же позиції.

У другому випадку імена атрибутів даних в описі файлу обов'язково не повинні повторюватися. Прикладом такої організації легко може служити проста таблична організація даних (табл. 7.1.):

Таблиця 7.1. **Послідовний файл**

Прізвище	Вік	Зріст	Стать
Вітютнев	19	189	М
Ровенський	17	178	М
Коломієць	18	183	М
Тагінцева	19	175	Ж
Завойко	17	180	Ж

Досить часто при роботі з послідовним файлом необхідно однозначно визначити записи у відповідності їхнім ключам. У такому випадку записи файлу впорядковуються у відповідності ключовим атрибутам. Один ключовий атрибут виконує роль первинного ключа сортування (ключа вищого порядку). Якщо цей ключ не в змозі однозначно визначити порядок, то доти, поки порядок не буде цілком визначений, можна задавати повторний і додаткові ключові атрибути. Тепер почергове зчитування записів файлу в цьому порядку може бути виконано послідовно.

Табл. 7.2 ілюструє записи послідовного файлу, що мають один ключовий атрибут:

Таблиця 7.2. **Послідовний файл із простим ключовим атрибутом — номер студента**

Номер студента	Спеціальність	Теорія	Практика	Вік
1543	ПЗ	4	5	19
1995	ТЕП	3	3	18
3214	ПМ	5	3	20
3321	ПЗ	5	5	19
4433	ПЗ	4	4	19
5521	ЄС	4	3	18
5812	ПМ	5	4	18
8123	ПМ	4	4	17
8810	ПЗ	5	5	18
9123	ПМ	4	5	19

Табл. 7.3, на відміну від табл. 7.2, не має ключа, що однозначно визначає запис таблиці, але в даному випадку як ключ може виступати пара полів запису:

Таблиця 7.3. **Послідовний файл із складовим ключовим атрибутом — предмет — номер студента**

Предмет	Номер студента	Група	Іспит	Вік
Фізика	1923	ПМ	5	19
Фізика	4232	ПЗ	3	18
Фізика	5001	ПМ	3	20
Інф.системи	134	ПЗ	5	19
Інф.системи	4232	ПЗ	4	19
Інф.системи	5001	ПМ	3	18
Філософія	134	ПЗ	4	18
Філософія	5001	ПМ	4	17

У табл. 7.2 і табл. 7.3 присутнє поле запису, що зберігає номер студента. Такі ключі допомагають однозначно визначити запис і одержати єдиний ключовий атрибут. Якщо такі ключі відсутні, то їх у деяких випадках доцільно додавати.

Послідовний файл характеризується обмеженим і наперед визначеним набором атрибутів. Усі записи одного послідовного файлу мають ідентичну структуру. У випадках, коли необхідно в деякий запис внести зміни (додати новий атрибут), то приходиться реорганізувати весь файл. При цьому, кожний запис файлу переписується, для того, щоб виділити місце для нового елемента даних. Тому для послідовних файлів іноді із самого початку визначають резервну область (кілька стовпців залишають незаповненими).

7.2. Пошук у послідовних файлах

Для роботи з послідовними файлами необхідно знати яким чином здійснюються основні операції роботи з послідовними файлами: пошук, додавання, видалення, модифікація запису, реорганізація усього файлу.

Для того, щоб пошук відбувався найбільш ефективно і не був вибагливим до часових характеристик, файл потрібно впорядкувати одним із методів сортування. Зрозуміло, що якщо потрібно тільки одноразовий перегляд (пошук) у файлі, то швидше буде зробити послідовний пошук, де сортування зовсім не обов'язкове. Але у випадках частого повторення пошуку ефективніше буде впорядкувати файл, а потім методом бінарного (двійкового) пошуку можна значно знизити час пошуку необхідного запису. Цей вид пошуку можливий тільки для того типу атрибута, по якому впорядковано файл.

Принцип послідовного пошуку складається з пошуку ключа. Така послідовна процедура є найпростішим способом пошуку. Для звичайного масиву фрагмент програми, що визначає, чи має один з його елементів задане значення, виглядає так (використовуємо мову програмування *C*++) [4]:

```
for (i = 0; i<n; i++)
if (A[i] = = B) break;
if (i != n) ...знайдено...
```

Те, що ми одержуємо в даному фрагменту тільки факт наявності елемента масиву з даним значенням, не має ніякого значення. У реальних програмах «елементами масиву» є, зазвичай, не прості змінні, а більш складні утворення (наприклад, структуровані змінні). Та частина елемента даних, яка ідентифікує його й використовується для пошуку, називається ключем. Інша частина несе в собі змістовну інформацію, яка добувається і використовується зі знайденого елемента даних. Ключ — частина елемента даних, яка використовується для його ідентифікації й пошуку серед безлічі інших таких елементів.

Наведений фрагмент програми забезпечує послідовний, або лінійний, пошук в неупорядкованому масиві, а середня кількість переглянутих елементів для масиву розмірності N буде дорівнювати $N/2$.

7.2.1. Послідовний або лінійний пошук

Лінійний пошук у послідовному файлі відбувається наступним чином. Припустимо, що є файл записів $R_1, R_2, R_3, ..., R_n$, яким відповідають ключі $K_1, K_2, ..., K_n$. Алгоритм для пошуку запису з даним ключем K, що можна описати таким чином «почни спочатку і просувайся, поки не знайдеш потрібний ключ, тоді зупинися», доцільно представити наступною послідовністю дій:

1. [Початкова установка.] Встановити i=1;
2. [Порівняння.] Якщо K=K_i, то алгоритм вдало завершується.
3. [Просування.] Збільшення i на одиницю.
4. [Кінець файлу] Якщо $i \leq N$, то повернутися до кроку 2. У протилежному випадку алгоритм закінчується невдало (рис. 7.2):

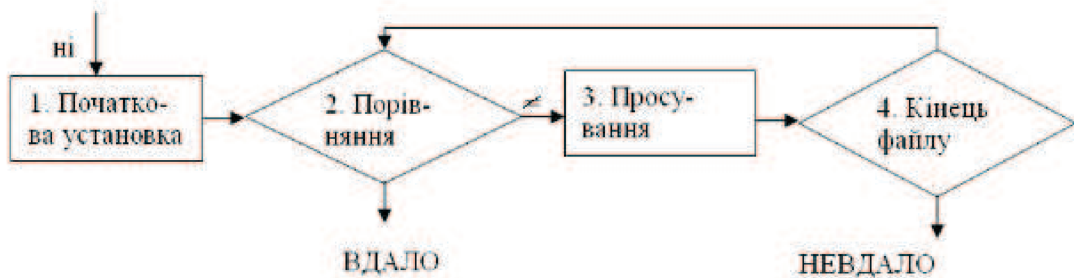


Рис. 7.2. Послідовний пошук

Зауважимо, що в цього алгоритму може бути два різних виходи: вдалий (коли знайдено положення потрібного ключа) і невдалий (коли встановлено, що шуканого аргументу немає).

7.2.2. Двійковий пошук

Один із способів реалізації методу двійкового пошуку використовує два покажчики — v і u, що відповідають верхній і нижній межі пошуку відповідно. Якщо елементи даних упорядковані, то знайти те, що нас цікавить можна значно швидше.

Алгоритм двійкового або бінарного пошуку засновано на розподілі навпіл поточного інтервалу пошуку. В основі його лежить той факт, що при однократнім порівнянні шуканого елемента і деякого елемента масиву ми можемо визначити, праворуч або ліворуч від поточного слід шукати. Найпростіше вибирати елемент на середині інтервалу, в якім проводиться пошук. Тоді одержимо такий алгоритм:

- шуканий інтервал пошуку ділиться навпіл, і за значенням елемента масиву в крапці розподілу визначається в якій частині необхідно шукати значення на наступному кроці циклу;
- для обраного інтервалу пошук повторюється;
- при досягненні значення інтервалу 0 — пошук припиняється;
- у якості початкового інтервалу вибирається весь масив.

Реалізація наступного алгоритму (рис. 7.3) дає результат у вигляді аргументу K із файлу записів R_1, R_2, R_3, ..., R_N, ключі яких розташовані в порядку зростання, тобто $K_1 < K_2 < \ldots < K_N$ [20]:

Рис. 7.3. Двійковий пошук

Алгоритм двійкового пошуку:

1. [Початкова установка] Встановити v=1 і u=n;

2. [Знаходження середини.] (Якщо K є присутнім у файлі, то природно виконується умова $K_1 < K < K_u$). Якщо $u < v$, то алгоритм закінчується невдало, тобто шуканий елемент не знайдено. У протилежному випадку потрібно встановити значення I=[(v+u)/2], де I вказує на середину аналізованої частини файлу;

3. [Порівняння.] Якщо $K < K_I$, то перейти до кроку 4, якщо ж $K > K_I$, то перейти до кроку 5, а якщо $K = K_I$, то алгоритм закінчується з позитивним результатом;

4. [Корегування нижньої межі u.] Встановити u=I–1 і перейти до кроку 2 ;

5. [Корегування верхньої межі v.] Встановити v=I+1 і перейти до кроку 2 .

Згідно описаного алгоритму наведемо приклад пошуку деякого числа методом двійкового пошуку: Пошук числа 300:

Крок перший: Встановлюємо верхню межу v=1 і нижню межу u=13;

Крок другий: Знаходимо середину I і встановлюємо її порядковий номер, що дорівнює 7; [123 233 274 289 300 432 <u>543</u> 692 903 992 1002 4993 5521]

Крок третій: Порівнюємо шукане число 300 з числом, що відповідає середині. Оскільки в нашому прикладі значення середини перевищує значення шуканого числа, то, відповідно до алгоритму, переходимо до виконання кроку номер чотири;

Крок четвертий: Встановлюємо значення нижньої границі u=I–1 і переходимо до кроку номер два;

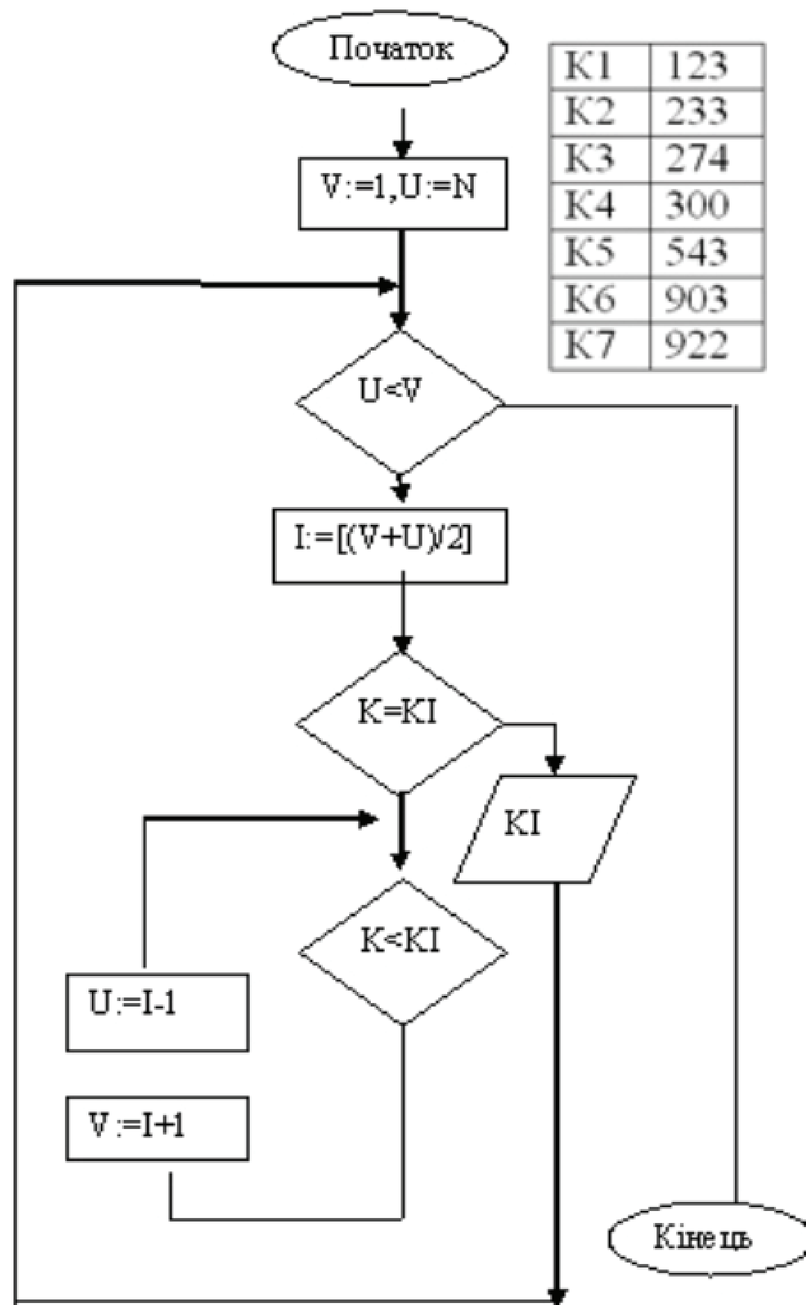
[123 233 274 289 300 432] 543 692 903 992 1002 4993 5521
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
123 233 274 [289 <u>300</u> 432] 543 692 903 992 1002 4993 5521

В остаточному підсумку пошук закінчується вдало — число 300 знайдено (рис. 7.4).

Представимо схему бінарного (двійкового) пошуку числа, аналогічну попередній, але в цьому випадку число знайдено не буде, тому що воно відсутнє:

Пошук числа 123:
[012 054 064 089 094 102 <u>120</u> 164 177 235 654 344 532]
012 054 064 089 094 102 120 [164 177 <u>235</u> 654 344 532]
012 054 064 089 094 102 120 [164 177] 235 654 344 532
012 054 064 089 094 102 120 164][177 235 654 344 532

Як видно зі схеми число не знайдено і в даній ситуації верхня границя перевищила значення нижньої границі.



K1	123
K2	233
K3	274
K4	300
K5	543
K6	903
K7	922

Рис. 7.4. Блок-схема алгоритму двійкового пошуку числа 300: V — верхня границя пошуку; U — нижня границя пошуку; K — шуканий елемент; X — масив елементів; I — значення, яке відповідає середині аналізованої частини X; N — розмірність масиву X; N=7.

Приклад двійкового пошуку в упорядкованому масиві при використанні мови програмування *C++* може бути виконаний наступним чином [6]:

```
int binary(int c[], int n, int val)
{                                   //Повертає індекс знайденого
int a,b,m;                          // Ліва, права границі і середина
for(a=0,b=n–1; a <= b;)
 {
m = (a + b)/2;                         // Середина інтервалу
if (c[m] == val)                       // Значення знайдено
return m;                            // Повернути індекс знайденого
if (c[nn] > val)
b = m–1;                               // Обрати ліву половину
else
a = m + 1;
}                            // Обрати праву половину
return – 1 ;
}                            // Значення не знайдено
```

Оцінимо кількість порівнянь, які необхідно виконати для пошуку необхідного значення. Оскільки після першого порівняння інтервал зменшується в 2 рази, після другого — в 4 рази і т. д., то кількість порівнянь буде не більше за відповідний ступінь числа 2, що дає розмірність масиву n, або $2s = n$, тоді $s = \log2(n)$. Для масиву з 1000 елементів їх буде 10, з 1000000 — 20. Саме заради цього й існують численні алгоритми сортування. З невеликими змінами даний алгоритм може використовуватися для визначення місця вставки нового елементу в упорядкований масив. Для цього необхідно обмежити розподіл інтервалу до одержання єдиного елемента ($a=b$), після чого додатково перевірити, куди слід робити вставку.

7.3. Основні операції над послідовними файлами

До основних операцій, що можна провадити над файлами взагалі і над послідовними файлами безпосередньо є додавання запису, видалення шуканого запису та відновлення запису, що було видалено. При цьому кожна операція створює необхідність реорганізації файлу, тобто приведення його до стану, який є характерним для файлу заданого вигляду.

7.3.1. Додавання запису

Додавання записів в основний файл неможливе, тому що при цьому записи додаються в кінець файлу, тим самим порушуючи упорядкованість у ньому. Тому, при виконанні процедури додавання записів у послідовний файл, записи збираються у файл коригувань доти, поки останній не стане достатньо великим, а потім здійснюється пакетне відновлення. Ця процедура виконується за допомогою реорганізації файлу.

Після створення файлу корегувань він, як і основний файл, сортується за аналогічним ключем. Після виконання необхідних операцій з файлом корегувань створюється новий файл, що є відновленим вихідним файлом. При виконанні таких операцій користуються визначеною термінологією: основний файл є файлом старшого покоління, файл отриманий після реорганізації основного файлу за допомогою файлу корегувань називається файлом молодшого покоління. Слід зазначити, що файл молодшого покоління — це зовсім новий файл, записаний на нове місце на носії інформації, ніяк не пов'язане з адресою файлу старшого покоління.

Рис. 7.5 ілюструє приклад вставки нових записів у послідовний файл із використанням файлу корегувань. Записи з файла коригувань і записи з файла старшого покоління записуються в утворюваний файл відповідно до порядку своїх ключових атрибутів. Спочатку в новий файл запишуться три записи з файла ста-

ршого покоління, після цього у файл молодшого покоління буде записано запис із файла коригувань, потім знову з файлу старшого покоління. Це буде продовжуватися доти, поки не будуть переписані всі записи з файлу старшого покоління і файла коригувань.

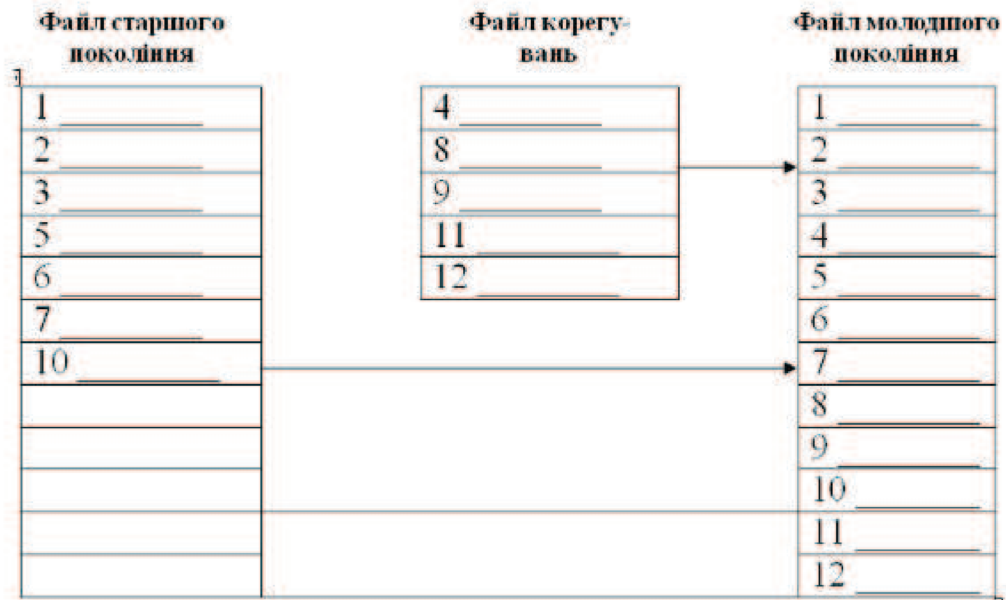


Рис. 7.5. Вставка записів у послідовний файл

7.3.2. Видалення запису

Аналогічно з додаванням, видалення записів з файлу так само неможливо здійснити, тому що в цьому випадку знову ж порушується упорядкованість файлу. Тому для видалення записів файлу використовується наступний спосіб: записи збирають в окремому новому файлі (файл корегувань); виконується пакетне відновлення файлу.

Процедура видалення записів файлу виконується за допомогою реорганізації файлу. Після створення файлу корегувань він, як і основний файл, сортується за аналогічним ключем. Після

виконання необхідних операцій з файлом корегувань створюється новий файл, що є відновленим вихідним послідовним файлом, який вже не містить записів, що було потрібно видалити. Файл корегувань у даному випадку містить записи, що також присутні у файлі старшого покоління і які потрібно видалити.

Розглянемо схему, що ілюструє процес видалення записів із послідовного файлу (рис. 7.6):

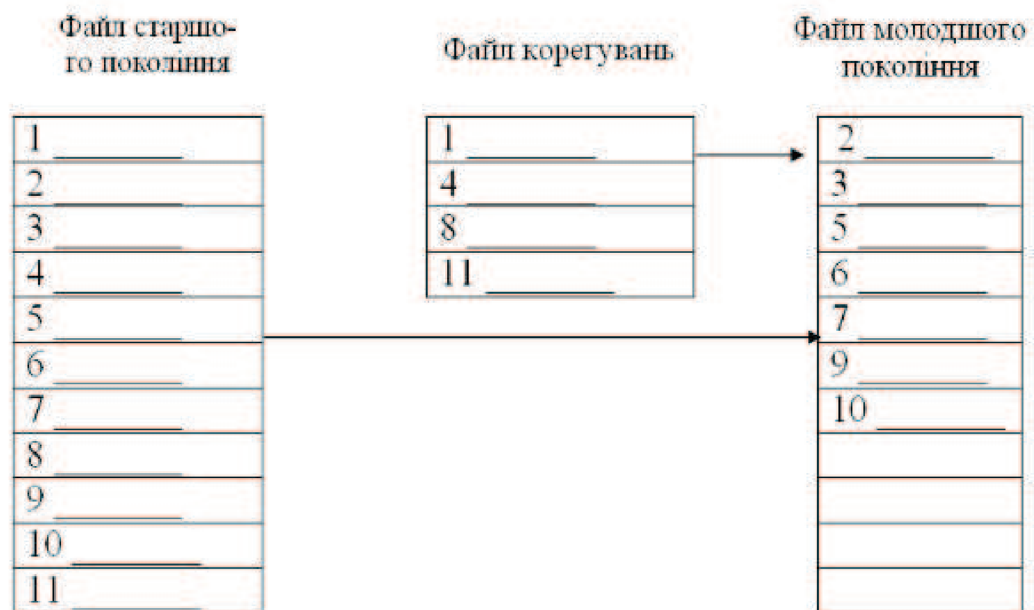


Рис. 7.6. Видалення запису із послідовного файлу

Як видно з рис. 7.6 при створенні нового файлу молодшого покоління деякі записи з файлу старшого покоління переписуються у файл молодшого покоління без усяких змін. У нашому випадку запис з ключем 1 не переписується, тому що саме його потрібно видалити (він міститься у файлі корегувань). І так далі, поки всі записи з файлу старшого покоління, що не потребують видалення, не будуть переписані у файл молодшого покоління.

7.3.3. Відновлення запису

Відновлення записів у послідовному файлі аналогічне зі вставкою записів у послідовний файл. Відновлення запису є заміною вже наявних даних у визначеному записі з деяким ключем на нові дані, що заносяться в цей же запис. При відновленні запису (оновлюваний запис не перевищує розміру старого запису) новий файл не створюється, а відновлення відбувається прямо в основному файлі, при цьому значення ключових атрибутів повинні бути однаковими (рис. 7.7).

Рис. 7.7. Схема відновлення записів у послідовному файлі

7.4. Реорганізація послідовних файлів

Принцип реорганізації послідовного файлу полягає в створенні нового файлу, шляхом об'єднання вмісту основного файлу і файлу корегувань. Для того, щоб цей процес (процес об'єднання) відбувався з найменшими витратами часу, рекомендується і доцільно, щоб файл старшого покоління і файл коригувань були впо-

рядковані за однаковим ключем (досить впорядкувати файл корегувань по ключі, по якому вже впорядковано основний файл).

У процесі об'єднання відсортованих даних, записи з файлу старшого покоління і файлу корегувань копіюються в новий файл.

Заключним кроком реорганізації файлу є видалення вихідного файлу, а потім і перейменування отриманого нового файлу, якому привласнюється ім'я старого (видаленого) файлу. Це забезпечує подальшу присутність файлу на носії інформації.

7.5. Сортування

При розв’язанні задачі сортування, зазвичай, висувається вимога мінімального використання додаткової пам'яті, з якої випливає неприпустимість застосування додаткових масивів. Для порівняння властивостей алгоритмів сортування важливо не те, скільки конкретно часу вони виконуються для даних відомого обсягу, а як вони поведуть себе при збільшенні цього обсягу в 10, 100, 1000 раз і т.д., тобто тенденція збільшення часу обробки, а вона в свою чергу залежить від кількості базових операцій над елементами даних — вибірок, порівнянь, перестановок. Із цією метою введене поняття трудомісткості. Ще раз нагадаємо, що трудомісткість алгоритму — це залежність кількості базових операцій алгоритму від розмірності вхідних даних.

Трудомісткість показує не абсолютні витрати часу в секундах або хвилинах, що залежить від конкретних особливостей комп'ютера, а в якій залежності зростає час виконання програми при збільшенні обсягів оброблюваних даних. Оцінимо трудомісткості відомих нам алгоритмів (рис. 7.8):

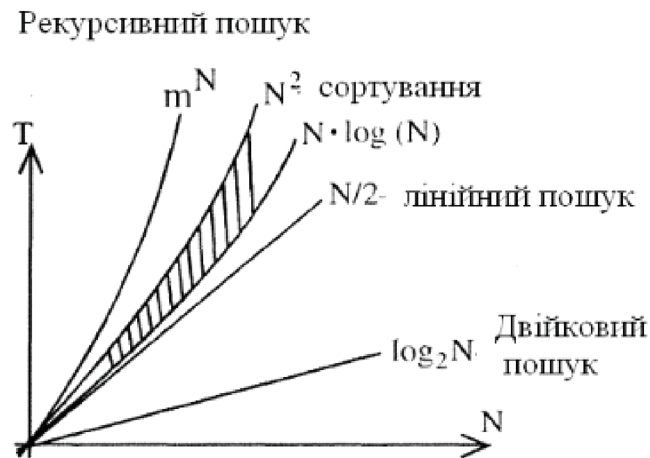


Рис. 7.8. Трудомісткість алгоритмів сортування і пошуку

Виходячи з даних, наведених на рис. 7.8:

- трудомісткість лінійного пошуку — $N/2$ — лінійна залежність;

- трудомісткість двійкового пошуку — залежність логарифмічна $\log_2 N$;

- для сортування, зазвичай, використовується цикл у циклі. Звідси видно, що трудомісткість навіть найгіршого сортування не може бути більше N×N — залежність квадратична. За рахунок оптимізації вона може бути знижена до $N \times \log(N)$;

- алгоритми рекурсивного пошуку, засновані на повному переборі варіантів, мають, зазвичай, показову залежність трудомісткості від розмірності вхідних даних (mN).

Алгоритми сортування можна класифікувати по декількох ознакам. Вид сортування по розміщенню елементів: внутрішнє — в оперативній пам'яті (сортування масивів), зовнішнє — сортування файлів. Внутрішнє сортування — це написаний алгоритм сортування, орієнтований (призначений) для збереження результатів сортування в оперативній пам'яті чи в одному файлі (файл використовується, як «сховище інформації»: зчитують, обробляють зчитане, записують зміни).

Зовнішнє сортування — це написаний алгоритм сортування, орієнтований (призначений) для роботи з дисковим простором (створюється група файлів з іменами, що вказують на збережену в них інформацію).

Вид сортування по виду структури даних, що містить елементи, які необхідно відсортувати: сортування масивів, масивів покажчиків, списків, даних в файлах і інших структур даних.

Основна ідея алгоритму. В основі різноманіття сортувань лежить різноманіття ідей. Тут потрібно відразу ж відокремити ідею алгоритму від варіантів його технічної реалізації, а також від поліпшень основного методу. Крім того, стосовно до різних структур даних той самий алгоритм сортування буде виглядати по-різному.

Для оцінки швидкодії алгоритмів різних методів сортування, як правило, використовують два показники:

- кількість присвоювань;
- кількість порівнянь.

Насамперед, виділимо сортування, в яких у процесі роботи створюється впорядкована частина — розмір її збільшується на 1 за кожний крок зовнішнього циклу. Сюди відносять дві групи сортувань:

- сортування вставками: черговий елемент розташовується по місці свого розташування у вихідну послідовність (масив);
- сортування вибором: вибирається черговий мінімальний елемент і розташовується в кінець послідовності.

Дві інші групи використовують поділи на частині, але за різними принципами і з різною метою:

- сортування поділом: послідовність (масив) розділяється на дві частково впорядковані частини за принципом «більше-менше», які потім можуть бути відсортовані незалежно (можливо навіть тим же самим алгоритмом);
- сортування злиттям: послідовність регулярно розподіляється на декілька незалежних частин, які потім поєднуються (злиття).

Сортування цих груп відрізняються від «банальних сортувань» тим, що процес упорядкування в них у явному вигляді не відслідковується (сортування без сортування).

Окрема група обмінних сортувань із численними оптимізаціями заснована на ідеї регулярного обміну сусідніх елементів.

Окремо існує сортування підрахунком. У ньому визначається кількість елементів, більших або менших даного, визначається його місце розташування у вихідній послідовності (масиві).

Практично кожний алгоритм сортування можна розбити на три частини:

- порівняння, що визначає упорядкованість пари елементів;
- перестановку, що змінює місцями пару елементів;
- алгоритм сортування, що здійснює порівняння і перестановку елементів доти, поки всі елементи не будуть упорядковані.

Алгоритм сортування — це процедура, що реорганізує (змінює) файл записів чи інформацію, завантажену в оперативну пам'ять таким чином, щоб елементи інформації постали в зростаючому (чи убутному) порядку. Завдяки такому упорядкованому розташуванню, стає можливою більш ефективна обробка інформації, що відсортована по одному ключу (елементу); створюється основа для ефективних алгоритмів роботи з інформацією.

7.6. Внутрішнє сортування

Числа a і b зв'язані відношенням порядку $a \leq b$, якщо виконуються такі умови [21]:

1) Рефлективність: $a \leq b$, $b \leq$ □□ a.

2) Антисиметричність: якщо $a \leq b$, $b \leq a$, то $a = b$.

3) Транзитивність: якщо $a \leq b$, то $a \leq c$.

4) Лінійність: для довільних a і b справедливо $a \leq b$ чи $b \leq a$.

Множина чисел, що задовольняють цим умовам у математиці називаються цілком упорядкованими.

Розглянемо елементи x_1, x_2, ..., x_n цілком упорядкованої безлічі. Послідовність таких елементів, розміщених в одномірному масиві в порядку зростання значень, будемо називати відсортованою послідовністю. Якщо є можливість переставляти вихідну послідовність елементів у довільному порядку (у випадку, якщо всі x_j різноманітні, утвориться n! перестановок), то процес перерозміщення елементів у відсортовану послідовність називається сортуванням.

Припустимо є файл, записи якого складаються з двох атрибутів: прізвище і вік, причому записи розташовані за абеткою по атрибуті прізвище. Припустимо, що при сортуванні записів у порядку збільшення віку ми хочемо зберегти алфавітний порядок розташування записів усередині групи кожного віку (рис. 7.9.).

Рис. 7.9. Схема стійкого алгоритму сортування

Зазвичай, алгоритм сортування, що зберігає упорядкованість елементів послідовності в кожній групі записів з тим самим ключем, називають стійким алгоритмом. Якщо сортування записів виконується тільки за значенням одного атрибута, то часто має сенс використовувати стійкий алгоритм сортування.

Нажаль, стійкими є простіші алгоритми, а складні, високоефективні алгоритми, як правило, нестійкі. Наприклад, у наступному прикладі алгоритм є нестійким. Розглянемо безліч, що складається з n груп чисел вигляду (x_{j1}, ... x_{jp}) (j = 1, ..., n). Відсортуємо ці групи за значенням першого елемента. Якщо кілька груп мають однакові значення перших елементів, відсортуємо їх по другим. Будемо виконувати ці дії доти, поки не одержимо відсортовану послідовність n груп чисел. Якщо представити дату у форматі 6-розрядної числової послідовності (наприклад, рік 82 серпень 15 буде виглядати як 82, 08, 15), тоді розглянутим способом можна відсортувати деяку безліч дат. При цьому для аналізу погрішностей у даних природно використовувати інформацію про діапазони значень елементів, що складають дату, тобто другий елемент міститься в інтервалі [1, 12], а третій — в інтервалі [1, 31] (чи в більш вузькому інтервалі в залежності від значення другого елемента). Розглянутим способом можна відсортувати досить великий масив даних. Стійкість алгоритму можна забезпечити порядком розгляду елементів груп, що сортуються: вони повинні вибиратися справа наліво. Приклад такого сортування приведено на рис. 7.10.

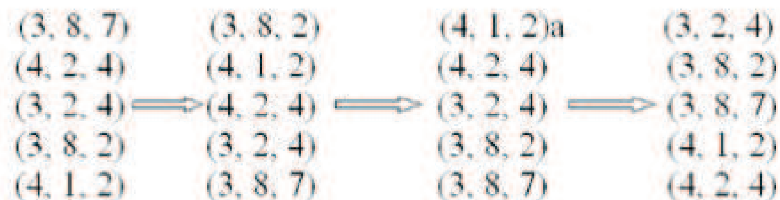


Рис. 7.10. Сортування по основі системи із застосуванням стійкого алгоритму

Тут сортування обов'язково треба починати з правих елементів послідовностей, тобто спочатку треба провести розподілене сортування по молодших цифрах ключів, хоча на перший погляд здається, що треба робити навпаки.

Часто сортування необхідне у випадках, коли записи файлу розташовані в залежності від значень їх ключів, тобто ключам записів визначено положення у файлі. Нижче розглянемо випадок, коли існує можливість змінити положення запису у файлі разом із ключем запису. У цьому випадку при заданих розмірах файлу, що підлягає сортуванню, необхідно враховувати можливість його обробки тільки з використанням оперативної пам'яті. Очевидно, що сортування в оперативній пам'яті і сортування в зовнішній пам'яті істотно відрізняються по ефективності; будуть відрізнятися також алгоритми і критерії оцінки алгоритмів.

7.6.1. Сортування простими вставками

Принцип методу полягає в наступному. Масив розділяється на дві частини: відсортовану і невідсортовану. Елементи з невідсортованої частини по черзі вибираються та вставляються у відсортовану частину так, щоб не порушувати в ній упорядкованість елементів. На початку роботи алгоритму в якості відсортованої частини масиву приймають тільки один перший елемент, а в якості не відсортованої частини — всі інші елементи. Технічні деталі: можна проводити лінійний пошук від початку впорядкованої частини до першого більшого за даний, з кінця — до першого меншого за даний (трудомісткість алгоритму по операціям порівняння — $N \times N/4$), використовувати двійковий пошук місця в упорядкованій частині (трудомісткість алгоритму — $N \times \log(N)$).

Вербально алгоритм сортування простими вставками можна описати наступним чином. Розглянемо послідовність $(x_1, ..., x_n)$ з порядком розташування елементів $1,...,n$. Порівняємо x_1 з x_2, якщо $x_1 \leq x_2$, то залишимо їх у цьому ж порядку. У протилежному випадку змінемо їх місцями. Позначимо, що утворилась послідовність $x_1^{(2)}, x_2^{(2)}$. Порівняємо x_3 з $x_2^{(2)}$ і якщо $x_2^{(2)} \leq x_3$ залишимо цей порядок, у протилежному випадку пересунемо $x_2^{(2)}$ вправо. Порівняємо x_3 з $x_1^{(2)}$ і якщо $x_1^{(2)} \leq x_3$ вставимо x_3 на місце $x_2^{(2)}$, у протилеж-

ному випадку підсунемо $x_1^{(2)}$ вправо, а x_3 вставимо на місце $x_1^{(2)}$ і т.д. Операції порівняння і переміщення виконуються до одержання відсортованої послідовності.

Схематично алгоритм сортування простими вставками можна відобразити у вигляді блок-схеми (рис. 7.11).

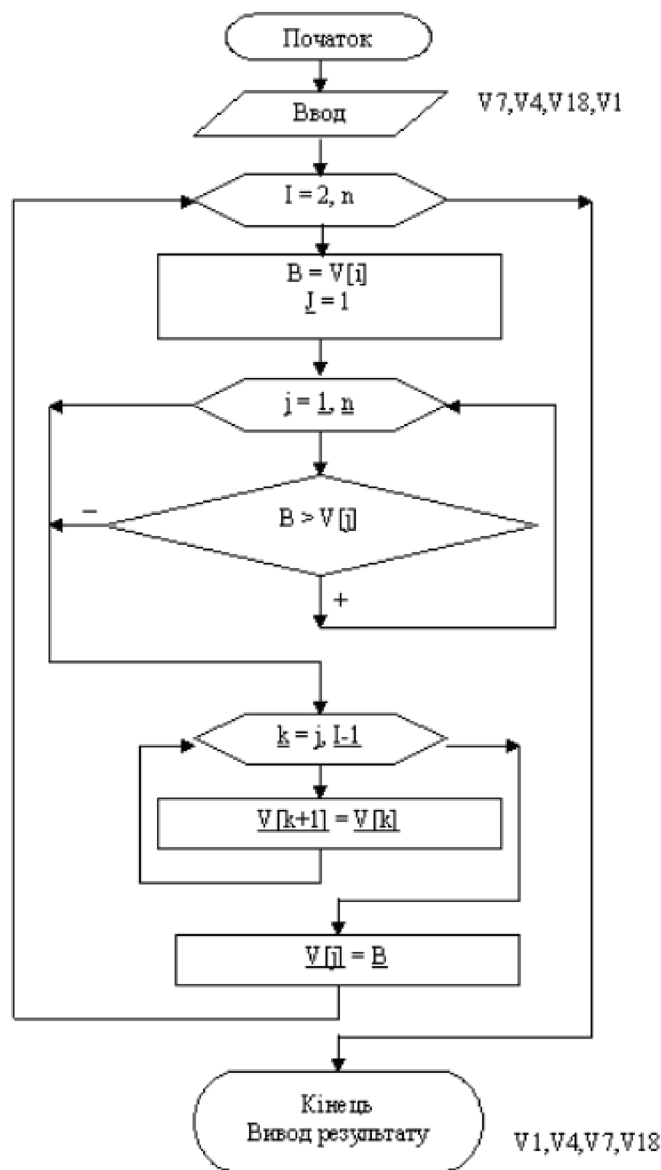


Рис. 7.11. Блок-схема алгоритму сортування простими вставками: *V[X]* — масив елементів; *I, j, k* — індетифікатори циклу, *B* — проміжна змінна.

Узагальнюючи вище сказане, алгоритм сортування буде складатися з n–1-го проходу (n — розмірність масиву), кожний з яких буде включати чотири дії:

1. Узяття чергового I-го не відсортованого елемента і збереження його в додатковій змінній;

2. Пошук позиції j у відсортованій частині масиву, в якій присутність узятого елемента не порушить упорядкованості елементів;

3. Зрушення елементів масиву від I–1-го до j–1-го вправо, щоб звільнити знайдену позицію вставки;

4. Вставка узятого елемента до знайденої j-ї позиції.

Наочно з принципом дії метода сортування вставками можна ознайомитися за допомогою рис. 7.12, де зображено схему сортування масиву букв.

```
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A O S R T I N G E X A M P L E
A O R S T I N G E X A M P L E
A O R S T I N G E X A M P L E
A I O R S T N G E X A M P L E
A I N O R S T G E X A M P L E
A G I N O R S T E X A M P L E
A E G I N O R S T X A M P L E
A E G I N O R S T X A M P L E
A A E G I N O R S T X M P L E
A A E G I M N O R S T X P L E
A A E G I M N O P R S T X L E
A A E G I L M N O P R S T X E
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

Рис. 7.12. Сортування масиву букв при використанні метода простих вставок

Під час першого проходу сортування вставками (рис. 7.12) елемент *S*, що займає другу позицію, більше *A*, тому торкати його не треба. На другому проході, коли в третій позиції зустрічається елемент *O*, він міняється місцями з *S*, в результаті послідовність *A O S* стає відсортованою і т.д. Незаштриховані елементи, що не взяті в кружок — це ті, що були пересунені на одну позицію вправо.

У наступному прикладі, реалізованого на мові програмування *C++*, послідовність дій по вставці чергового елемента в упорядковану частину «розкладена по поличках» у вигляді послідовності чотирьох дій, зв'язаних змінними [6].

```
// Проста вставка
void sort(int in[], int n){
for ( int i = 1; i < n; i++) {              // Для чергового i
int v=in[i];                               // Роби 1 : зберегти черговий
for (int k=0; k<i; k++)                    // Роби 2 : пошук місця вставки
if(in[k]>v) break;                         перед першим, більшим v
for(int j = i – 1 ; j> = k; j – – )        // Роби 3: зрушення на 1 вправо
in[j + 1] = in[j];                         // від чергового до знайденого
in[k]=v;                     // Роби 4 : вставка чергового на місце
}}                           // першого, більшого за нього
```

Цей алгоритм сортування є стійким. Якщо кількість порівнянь і переміщень розділити на мінімальну кількість порівнянь (n–1), то буде отримана характеристика, що має назву «кількість інверсій». Для випадку, коли послідовність із самого початку відсортована в зростаючому порядку, інверсія мінімальна і дорівнює 0. Якщо із самого початку послідовність відсортована в убутному порядку, інверсія максимальна і дорівнює $(n–1)n/2$. Інверсія — це кількість випадків, коли x_k менше елементів із послідовності $(x_1, ..., x_{k–1})$ для k=2,…, n.

Необхідно зауважити, що час виконання сортування вставками залежить, головним чином, від вихідного порядку ключів при введенні. Наприклад, якщо файл великий, а ключі записів вже упорядковані (або майже упорядковані), то сортування вставками виконується швидко.

7.6.2. Сортування простим вибором

Цього разу при перегляді масиву (x_1, ..., x_n) ми будемо шукати найменший елемент, порівнюючи його з першим x_1. Якщо такий елемент знайдено, поміняємо його місцями з першим. Позначимо послідовність, що утворилася, як x_1(..., $x_n^{(2)}$). Потім повторимо цю операцію, але почнемо не з першого елемента, а з другого, знайшовши найменше значення серед елементів послідовності $x_2^{(2)}$ (..., $x_n^{(2)}$), міняємо його місцями з $x_2^{(2)}$. І будемо продовжувати подібним чином, поки не відсортуємо весь масив. Після (n–1) кроків порівняння й обміну одержимо послідовність відсортовану в зростаючому порядку (рис. 7.13).

При видалені обраного елемента з масиву на його місце може бути записано «дуже велике число», що виключає його повторний вибір. Обраний елемент може видалятися шляхом зрушення частини, що залишилася, мінімальний елемент може мінятися місцями з «наступним». Трудомісткість алгоритму — $N \times N/2$.

Наглядно ознайомитися з принципом роботи розглядуваного методу сортування можна за допомогою рис. 7.14, де показано, як буде змінюватися масив букв від первинного стану до відсортованого.

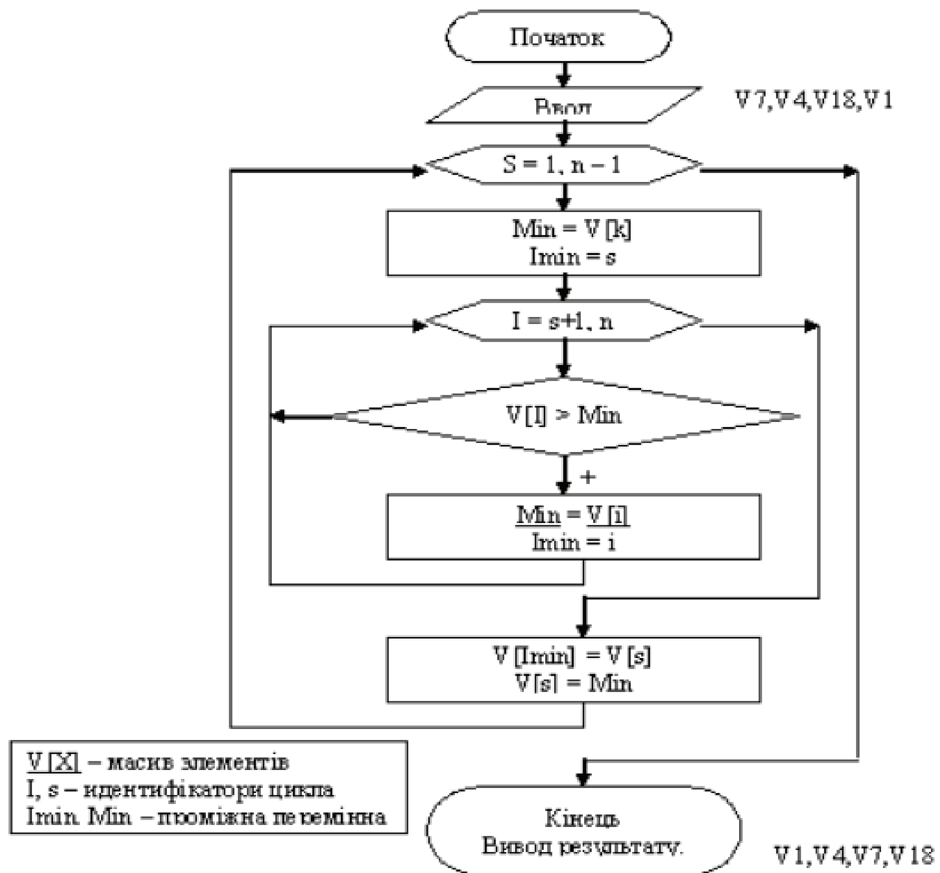


Рис. 7.13. Блок-схема алгоритму сортування простим вибором

```
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A A O R T I N G E X S M P L E
A A E R T I N G O X S M P L E
A A E E T I N G O X S M P L R
A A E E G I N T O X S M P L R
A A E E G I N T O X S M P L R
A A E E G I L T O X S M P N R
A A E E G I L M O X S T P N R
A A E E G I L M N X S T P O R
A A E E G I L M N O S T P X R
A A E E G I L M N O P T S X R
A A E E G I L M N O P R S X T
A A E E G I L M N O P R S X T
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

Рис. 7.14. Сортування масиву букв при використанні сортування простим вибором

У прикладі, зображеному на рис. 7.14, перший прохід не дав результату, оскільки ліворуч від *A* в масиві немає елемента, меншого за *A*. На другому проході інший елемент *A* виявився найменшим серед тих, що залишилися, тому він міняється місцями з елементом *S*, що займає другу позицію. Далі, на третьому проході, елемент *E*, що знаходиться в середині масиву, міняється місцями з *O*, що займає третю позицію; потім, на четвертому проході, ще один елемент *E* міняється місцями з *R*, що займає четверту позицію і т.д. до моменту, коли всі букви займають визначені місця.

Наступний приклад — один із численних варіантів «мирного співіснування» упорядкованої й неупорядкованої частин в одному масиві. Упорядкована частина перебуває ліворуч, і її розмірність відповідає кількості виконаних кроків зовнішнього циклу. Неупорядкована частина розташована праворуч, тому пошук мінімуму із запам'ятовуванням індексу мінімального елемента відбувається в інтервалі від *i* до кінця масиву (використовуємо мову програмування *C++)* [8].

```
//  Сортування простим вибором
void sort(int in[], int n){
for ( int i=0; i < n–1; i++){                    // Для чергового і
for ( int j = i + 1, l< = i; j<n; j++) //k — індекс мінімального
if (in[j] < in[k]) k=j;                   // в діапазоні i..n–1
int c=in[k]; in[k] = in[i]; in[i] = c;
}}
```

У сортуванні вибором контекст вибору мінімального елемента, зазвичай, помітний «неозброєним оком». Але в наступному варіанті він сполучений із процесом обміну й тому не видний: Мінімальний елемент відразу ж переміщається на чергову позицію [6]:

```
// „Законспіроване” сортування вибором
void sort(int in[], int n){
for ( int i=0; i < n-1; i++)                 // Для чергового і
```

```
for ( int j = i + 1, k=i; j<n; j++)        // Для всіх залишившихся
if (in[j] < in[ i]) {                      // в діапазоні і..n–1
int c=in[i]; in[i] = in[j]; in[j] = c;  // одразу ж міняти з черговим
}}                                         // Вибір сумісний з обміном
```

Недолік сортування вибором полягає в тому, що час його виконання лише в невеликій мірі залежить від того, наскільки упорядковано вихідний файл. Процес пошуку мінімального елемента за один прохід файлу дає дуже мало інформації про те, де може знаходитися мінімальний елемент на наступному проході цього файлу. Наприклад, користувач, що застосовує цей метод, буде чимало здивований, коли довідається, що на сортування майже відсортованого файлу або файлу, записи якого мають однакові ключі, потрібно стільки ж часу, скільки і на сортування файлу, упорядкованого випадковим чином.

Незважаючи на всю простоту й очевидний примітивізм підходу, сортування вибором перевершує більш досконалі методи сортування в одному з важливих аспектів: цьому методові сортування файлів віддається перевага в тих випадках, коли записи файлу величезні, а ключі займають незначний простір. У подібного роду додатках витрати ресурсів на переміщення записів набагато перевершують витрати на операції порівняння, а що стосується переміщення даних, то ніякий алгоритм нездатний виконати сортування файлу з меншою кількістю переміщень даних, ніж метод вибору.

Якщо порівнювати властивості цього алгоритму з властивостями алгоритму сортування простими вставками, отримаємо, що у випадку сортування простим вибором кількість порівнянь більша, а кількість переміщень менша. У цілому великої різниці в ефективності немає. Сортування вставками більш підходить для випадку, коли дані, що підлягають сортуванню надходять послідовно, а сортування простим вибором — коли всі дані знаходяться в пам’яті, а відсортовані дані виводяться послідовно.

7.6.3. Просте обмінне сортування («пухирцеве» сортування)

Ідея цього методу відображена в його назві. Найлегші елементи масиву «спливають» наверх, самі важкі — «тонуть». Алгоритмічно це можна реалізувати наступним чином. Ми будемо переглядати послідовність ($x_1, ..., x_n$), виконуючи порівняння кожних двох сусідніх елементів послідовності. Якщо лівий елемент із пари більше правого — змінюємо їх місцями. Тепер елемент, що має найменше значення, знаходиться в самій крайній лівій позиції. Якщо повторювати ці дії для N–2 елементів, що залишилися невідсортованими (тобто для тих, котрі лежать «нижче» першого), то найменший із цих елементів пересунеться на другу позицію зліва. Алгоритм зупиняється, коли вся послідовність стає відсортованою. Усього потрібно N–1 прохід.

Як видно, алгоритм досить простий, але, як іноді зауважують, він є неперевершеним у своїй неефективності. Оцінити трудомісткість алгоритму можна через середню кількість порівнянь, яка дорівнює $(N \times N - N)/2$.

Обмінні сортування мають ряд особливостей. Насамперед, вони чутливі до ступеня вихідної впорядкованості масиву. Повністю впорядкований масив буде переглянуто один раз, у той час як вибір або вставка будуть «зображувати бурхливу діяльність». Крім того, основна властивість, на якій заснована їхня оптимізація, безпосередньо не спостерігається в тексті програми: елемент із більшим значенням «підхоплюється» сукупністю послідовних обмінів і «спливає» до кінця масиву, поки не зустріне елемент, більший за себе. Із знайденим процес сортування продовжується.

Алгоритм метода «пухирця» можна представити у вигляді блок-схеми, у такий спосіб, як це зроблено на рис. 7.15.

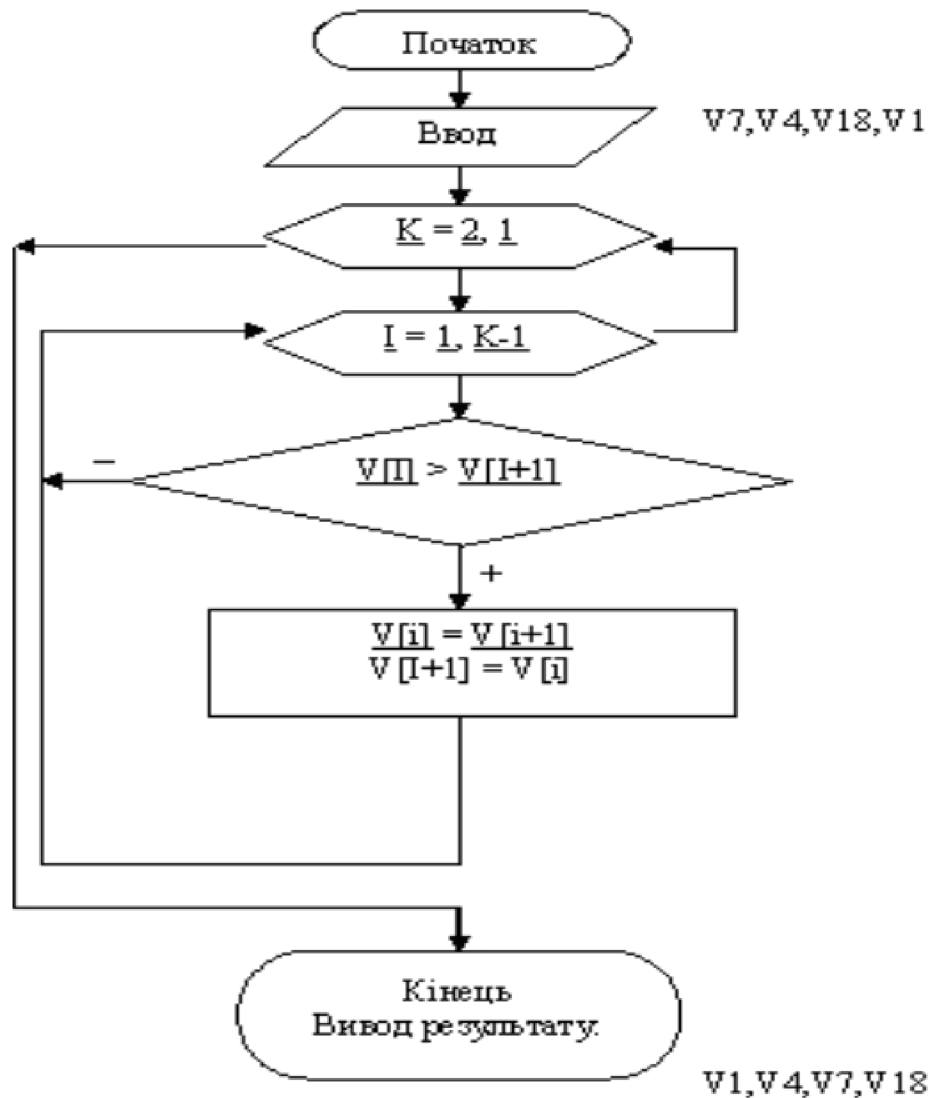


Рис. 7.15. Блок-схема алгоритму сортування «пухирцем»:
V[*X*] — це масив елементів, а *I, K* — проміжні змінні

Наведемо ще один приклад сортування методом «пухирця», з якого можна наочно ознайомити зі схемою функціонування цього методу. Нехай дано неупорядкований масив букв, що необхідно впорядкувати методом пухирця. Схема переходу невідсортованого заданого масиву до відсортованого наведено на рис. 7.16, де показано, що в процесі «пухирцевого» сортування ключі з малими зна-

ченнями спрямовуються вліво. Оскільки сортування здійснюється в напрямку справа наліво, кожний ключ міняється місцями з ключем ліворуч доти, поки не буде виявлено ключ із меншим значенням.

```
A S O R T I N G E X A M P L E
A A S O R T I N G E X E M P L
A A E S O R T I N G E X L M P
A A E E S O R T I N G L X M P
A A E E G S O R T I N L M X P
A A E E G I S O R T L N M P X
A A E E G I L S O R T M N P X
A A E E G I L M S O R T N P X
A A E E G I L M N S O R T P X
A A E E G I L M N O S P R T X
A A E E G I L M N O P S R T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

Рис. 7.16. Сортування масиву букв за допомогою метода «пухирця»

На першому проході (рис. 7.16) *E* міняється місцями з *L, P* і *M*, поки не зупиниться праворуч від *A*; далі вже *A* просувається до початку файлу, поки не зупиниться перед іншим *A*, що вже займає остаточну позицію, *i*-й по величині ключ встановлюється у своє остаточне положення після *i*-ого проходу, як і у випадку сортування вибором, але при цьому й інші ключі також наближаються до своїх остаточних позицій.

Фрагмент програм, що реалізує сортування методом «пухирця», виконаний на мові програмування *C++* при використанні циклу з параметром і циклу з постумовою, наведено нижче [4]:

```
// Сортування методом „пухирця”
void sort(int A[], int n){
int i,found;                          // Кількість порівнянь
do {                               // Повторювати перегляд
```

```
found =0;
for (i=0; i<n–1; i++)
if (A[i] > A[i + 1]) {                    // Порівняти сусідів
int cc = A[i]; A[i]=A[i + 1]; A[i + 1]=cc;
found++;                                  // Переставити сусідів
}
} while(found !=0); }                     //доки є перестановки
```

Також, для реалізації алгоритму можна використовувати два цикли з параметром так, як це наведено в наступному фрагменті коду (мова програмування *C++)* [6]:

```
void bubble(Item a[], int 1, int r)
{int temp;
for (int i= 1; i < r; i++)
for (int j = r; j > i; j– –)
{ int temp = a[i];
 a[i]=a[i + 1];
 a[i + 1]=temp;
}
}
```

Дії, що виконуються в наведеному прикладі описуються наступним чином. Для кожного i від 1 до r–1 внутрішній цикл (j) поміщає мінімальний елемент серед елементів послідовності $a[i],...,a[r]$ у $a[i]$, переходячи від елемента до елемента справа наліво і, виконуючи при цьому операції порівняння значень сусідніх елементів та обміну місцями наступних друг за другом елементів. Найменший елемент за допомогою операцій порівнянь переміщається вліво і «спливає як пухирець» до початку файлу. Як і у випадку сортування методом вибору, в умовах якої індекс i переміщається по файлі зліва направо, елементи ліворуч від нього знаходяться у своїх остаточних позиціях.

Якщо файл практично відсортований, можна змусити сортування працювати з більшою ефективністю, перевіряючи, чи не

виявився черговий прохід таким, що на ньому не було виконано жодної операції перестановки елементів місцями (що рівносильне повному упорядкуванню файлу, у зв'язку з чим можна залишити зовнішній цикл). Впровадження в програму подібного удосконалення дозволяє підвищити швидкість дії «пухирцевого» сортування для деяких типів файлів, однак у загальному випадку воно не досягне тієї ефективності, що характерна для програми сортування вставками.

Кількість порівнянь при використанні методу обмінного сортування складає мінімум $(n-1)$, максимум — $(n-1)n/2$. Перше значення відповідає випадку найкращого варіанта кількості обмінів, друге — найгіршого.

7.7. Характеристики продуктивності елементарних методів сортування

Алгоритми сортування вибором, вставками і «пухирцеве» сортування за часом виконання знаходяться в квадратичній залежності від кількості елементів як у найбільш складних, так і в звичайних випадках, але в той же час вони не мають потреби в додатковій пам'яті. Таким чином, час їхнього виконання відрізняється тільки постійним коефіцієнтом пропорційності цієї залежності, хоча принципи їхньої роботи істотно розрізняються.

У загальному випадку час виконання алгоритму сортування пропорційний кількості операцій порівняння, виконуваних цим алгоритмом, кількості переміщень або змін місцями елементів, а, можливо, і обом одразу. У випадку введення даних, упорядкованих випадковим чином, порівняння зазначених методів сортування припускає вивчення розходжень між відповідними постійними коефіцієнтами пропорційності в залежності від кількості виконуваних операцій порівнянь і зміни елементів місцями, а також розходжень відповідних постійних коефіцієнтів пропорційності в за-

лежності від кількості виконуваних операцій у внутрішніх циклах. У випадку введення даних зі спеціальними характеристиками час виконання різних видів сортувань може відрізнятися в більшій мірі, ніж за значеннями зазначених вище постійних коефіцієнтів пропорційності.

Сортування вставками робить у середньому приблизно $N2/4$ операцій порівняння і $N2/4$ операцій напівобміну елементів місцями (переміщень) і в два рази більше операцій у найгіршому випадку. Сортування, виконане програмою, що реалізує алгоритм сортування вставками виконує ту ж саму кількість порівнянь і переміщень. Ця величина легко прослідковується на діаграмі розміром N на N, представленої на рис. 7.12, що докладно ілюструє роботу алгоритму сортування вставками. Тут ведеться підрахунок елементів, що лежать під головною діагоналлю, причому в найгіршому випадку враховуються всі такі елементи. Очікується, що для випадково розподілених вхідних даних кожний елемент пройде в середньому половину шляху назад, отже, необхідно враховувати тільки половину елементів, що лежать нижче діагоналі.

Сортування вибором робить приблизно $N/2$ операцій порівняння і N операцій обміну елементів місцями. Можна легко перевірити цю властивість на прикладі даних, приведених на рис. 7.14, що представляють собою таблицю розмірністю N на N, на якій не заштриховані букви відповідають порівнянням. Приблизно половина елементів цієї таблиці незаштрихована, ці елементи розташовані над діагоналлю. Кожному з $N–1$ елементів (за винятком завершального елемента) на діагоналі відповідає операція обміну. Точніше, дослідження програмного коду показує, що на кожне i від 1 до $N–1$ приходиться одна операція обміну та $N–i$ порівнянь так, що всього виконується $N–1$ операцій обміну та $(N–1)+(N–2)\ +...+2+1=N(N–1)/2$ операцій порівняння [21]. Ці властивості зберігаються незалежно від природи вхідних даних. Єдиний показник сортування вибором, що залежить від характеру

вхідних даних — це кількість операцій присвоювання змінній *min* нових значень. У найгіршому випадку ця величина також стає квадратично залежною, однак у середньому вона характеризується значенням $(N\log N)$, тому ми вправі розраховувати на те, що час виконання сортування вибором не суттєво залежить від природи вхідних даних.

«Пухирцеве» сортування робить у середньому приблизно $N/2$ операцій порівняння і $N/2$ операцій обміну як у середньому, так і в найгіршому випадках. На i-му проході «пухерцевого» сортування потрібно виконати $N–i$ операцій порівняння-обміну. Якщо алгоритм удосконалено таким чином, що його виконання припиняється, як тільки він виявляє, що файл уже відсортовано, то час його виконання залежить від природи вхідних даних. Якщо файл уже відсортовано, то досить усього лише одного проходу, однак i-й прохід вимагає виконання $N–i$ операцій порівняння й обміну, якщо файл відсортовано у зворотному порядку. Як відзначалося раніше, ефективність сортування для звичайного випадку ненабагато вища, ніж для самого складного випадку.

На рис. 7.17 наведено схему сортування вхідного файлу при використанні розглядуваних методів. На рис. 7.17 показані розходження в тому, як сортування вставками, вибором, а також «пухирцеве» сортування приводять конкретний файл у порядок. Файл, що підлягає сортуванню, представлено у вигляді відрізків, що сортуються по куті нахилу. Чорні відрізки відповідають елементам, до яких у процесі сортування виконується доступ на кожнім проході; сірі відрізки відповідають елементам, що не зачіпаються. В умовах сортування вставками (ліворуч) елемент, що вставляється, проходить приблизно половину шляху назад через відсортовану частину на кожнім проході. Сортування вибором (у центрі) і «пухирцеве» сортування (праворуч) проходять на кожнім проході через усю невідсортовану частину масиву з метою виявлення там наступного найменшого елемента; розходження між цими методами полягають у тому, що «пухирцеве» сорту-

вання змінює місцями кожну пару сусідніх елементів, які порушують порядок, у той час, як сортування вибором поміщає мінімальний елемент в остаточну позицію. Це розходження виявляється в тому, що в міру виконання «пухирцевого» сортування, невідсортована частина масиву стає все більш упорядкованою.

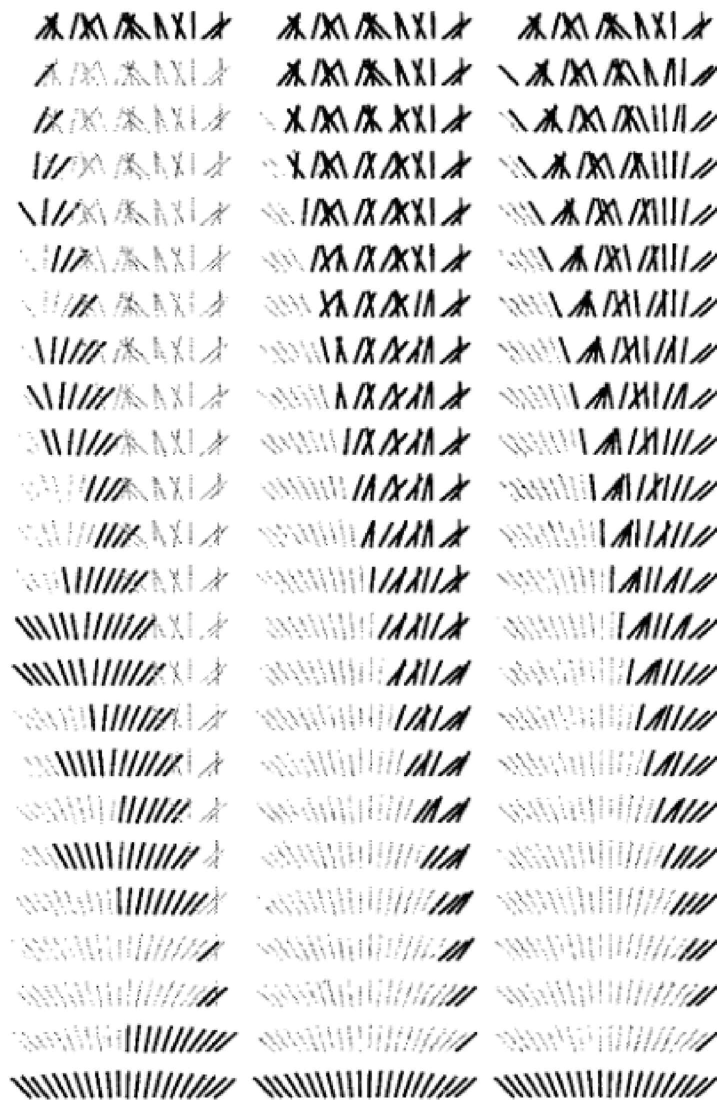


Рис. 7.17. Операції порівняння й обміну в умовах елементарних сортувань

Наприклад, розглянемо, як працює сортування вставками на файлі, що вже є відсортованим. Відразу ж з'ясовується, що всі елементи файлу знаходяться на своїх місцях, а загальні витрати часу на сортування лінійно залежать від кількості елементів. Це ж твердження справедливе й по відношенню до «пухирцевого» сортування, але для сортування вибором ця залежність залишається квадратичною.

7.8. Зовнішнє сортування

Ми переходимо до розгляду іншого аспекту задачі сортування, що виникає, коли файл, що необхідно від сортувати не знаходиться в оперативній пам'яті комп'ютера. Для опису такого роду ситуацій використовується термін зовнішнє сортування (*external sorting*) [10]. Існує безліч різних типів пристроїв зовнішнього сортування, що накладають різні обмеження на атомарні операції, застосовувані при реалізації таких видів сортування. Крім того, буде корисно вивчити методи сортування, що використовують дві найпростіші базові операції: операція зчитування (*read*) даних із зовнішнього запам'ятовуючого пристрою в оперативну пам'ять і операція запису (*write*) даних з оперативної пам'яті на зовнішній запам'ятовуючий пристрій.

У даному розділі розглянемо сортування, що проводиться для послідовностей записів, які зберігаються у файлах із послідовним доступом, розташованих на пристроях зовнішньої пам'яті.

Для здійснення зовнішнього сортування виконуються такі операції:

1) Знаходять потрібний блок у зовнішній пам'яті.

2) Роблять зчитування і передачу даних по каналу вводу-виводу в оперативну пам'ять.

3) В оперативній пам'яті здійснюють операції порівняння і переміщення.

4) Виконують запис у зовнішню пам'ять відповідно до 1 і 2.

При виборі того або іншого алгоритму сортування необхідно враховувати такі параметри:

а) обсяг оперативної пам'яті, використаної в якості робочої області;

б) тип, кількість, обсяг зовнішньої пам'яті, використовуваної в якості робочої області;

в) кількість каналів вводу-виводу;

г) кількість і довжину записів;

д) час виконання операцій в оперативній пам'яті.

Абстрактна модель сортування, якою ми надалі будемо користуватися, побудована на припущенні, що файл, який підлягає сортуванню занадто великий, щоб цілком поміститися в оперативну пам'ять, і що він визначає значення двох інших параметрів: часу виконання сортування (кількість проходів по даним) і кількість використовуваних зовнішніх пристроїв, тобто в нашому розпорядженні маються:

- N записів на зовнішньому пристрої, сортування яких ми повинні виконати;

- простір оперативної пам'яті, достатній для розміщення M записів;

- $2P$ зовнішніх пристроїв, якими ми можемо користуватися під час сортування.

Ми привласнюємо мітку 0 зовнішньому пристроєві, на якому знаходиться файл вхідних даних, і мітки 1, 2, ..., $2P–1$ всім іншим зовнішнім пристроям. Ціль сортування полягає в тому, щоб повернути запис на пристрій 0 у відсортованому вигляді.

Велика частина методів зовнішнього сортування дотримує наступних принципів загального характеру. Виконується перший прохід файла, що сортується, у процесі якого виконується його розбивка на блоки, розмір яких приблизно відповідає просторові оперативної пам'яті, після чого виконується сортування цих бло-

ків. Потім здійснюється злиття відсортованих блоків, при необхідності з цією метою виконуються кілька проходів файлу, при цьому з кожним проходом ступінь упорядкованості зростає, поки весь файл не виявиться відсортованим.

7.8.1. Сортування злиттям

Найпростіший метод зовнішнього сортування — сортування-злиттям, що одержала назву збалансованого багатошляхового злиття (*balanced multiway merging*), схема якого показана на рис. 7.18.

```
A S O R T I N G A N D M E R G I N G E X A M P L E W I T H F O R T Y F I V E R E C O R D S • S

A O S • D M N • A E X • F H T • E R V • S
I R T • E G R • L M P • O R T • C E O • S
A G N • G I N • E I W • F I Y • D R S • S

A A G I N O R S T • F F H I O R T T Y • S
D E G G I M N N R • C D E E O R R S V • S
A E E I L M P W X • S

A A A D E E E G G G I I I L M M N N N O P R R S T W X • S
C D E E F F H I O O R R R S T T V Y • S
• S

A A A C D D E E E E E F F G G G H I I I I L M M N N N O O O P R R R R R S S T T T V W X Y • S
```

Рис. 7.18. Схема багатошляхового злиття

На проході початкового розподілу беремо елементи *A S O* (рис. 7.18) з набору вхідних даних, сортуємо їх і поміщаємо відсортовану сукупність *A O S* на перший пристрій для вихідних даних. Далі ми беремо елементи *R T I* з набору вхідних даних, сортуємо їх і поміщаємо відсортовану сукупність *I R T* на другий пристрій для вихідних даних. Продовжуючи таким чином, роблячи відповідні операції на вихідних пристроях у циклі, ми в остаточному результаті отримаємо 15 відрізків: по п'ять на кожному вихідному пристрої. На першій стадії злиття ми здійснюємо злиття

відрізків *A O S, I R T* та *A G N*, в результаті чого одержуємо послідовність *A A G I N O R S T*, яку ми записуємо на перший вихідний пристрій, потім ми виконуємо злиття других відрізків на вхідних пристроях і одержуємо послідовність *D E G G I M N N R*, що ми записуємо на другому вихідному пристрої, і т.д.; таким способом ми виконуємо збалансований розподіл даних на три пристрої. Сортування завершується двома додатковими проходами, на яких виконується злиття.

Сортування злиттям складається з проходу, що здійснює початковий розподіл (*initial distribution*), за яким чередують кілька проходів багатошляхового злиття (*multiway merging passes*). На початковому проході ми здійснюємо розподіл вхідних даних (входу) по зовнішніх пристроях P, $P + 1,...,$ $2P–1$ у вигляді відсортованих блоків даних, кожний з яких містить M записів (за винятком, можливо, заключного блоку, що менший за інші, якщо N не кратно M). Такий розподіл неважко виконати — ми зчитуємо перші M записів із пристрою введення, сортуємо їх і записуємо отриманий блок даних на пристрій P; потім зчитуємо наступні M записів із вхідного пристрою, сортуємо їх і записуємо відсортований блок на пристрій $P+1$ і т.д. Якщо, досягнувши пристрою $2P–1$, у нас усе ще залишаються неопрацьовані дані (тобто, якщо $N>PM$), ми поміщаємо на пристрій $P+1$ другий відсортований блок і т.д. доти, поки всі введення не будуть вичерпано. По завершенні процедури розподілу кількість відсортованих блоків, розміщених на кожнім пристрої, дорівнює значенню N/M, округленому до найближчого цілого числа вбік зменшення або збільшення. Якщо N є кратним M, то розміри всіх блоків однакові і дорівнюють N/M (якщо це не так, то всі блоки, за винятком останнього, дорівнюють N/M). Для невеликих N кількість блоків може виявитися менше P, через це один або більша кількість пристроїв можуть виявитися порожніми.

У загальному випадку алгоритм багатошляхового злиття полягає в наступному:

1. Використовуючи оперативну пам'ять, сформувати якомога більш довгі початкові відрізки з елементів заданого файла; розподілити їх підходящим способом на декілька файлів.

2. Сформувати більш довгі відрізки шляхом злиття відрізків декількох файлів, удруге розподілити їх по декількох файлах.

3. Повторювати цю операцію.

4. Якщо наприкінці утвориться один відрізок, закінчити сортування.

Окремий випадок сортування злиттям — це двошляхове злиття. Суть якого полягає в наступному. Оберемо якомога довгі ділянки, які є відсортованими послідовності вигляду ak, $ak+1,...,al-1$, al у кінцевій числовій послідовності $a1$, $a2$, ..., an. Ці ділянки назвемо відрізками. Точніше, це будуть зростаючі відрізки. Розглянемо приклад числової послідовності:

$$\underline{56}\ \underline{2489}\ \underline{3}\ \underline{17}$$

Підкреслені ділянки є відрізками довжиною 2, 4, 1, 2. Відсортована послідовність складається з одного відрізка довжиною n, а послідовність, відсортована в зворотному порядку, складається *з* n відрізків довжиною l. Припустимо є послідовність $a1$, $a2$, ..., an різноманітних чисел. Можна зробити висновок, що для рівноймовірності можливі $n!$ варіантів розташування елементів цієї послідовності, а середнє значення кількості відрізків дорівнює $(n+1)/2$. Це виходить з того, що математичне очікування кількості відрізків довжиною не менше p буде складати $1/p!+(p/(p+1)!)(n-p)$; при $p=1$ кількість відрізків дорівнює $(n+1)/2$. Очікуване значення довжин відрізків залежить від початкової позиції відрізків у послідовності, проте, можна вважати, що воно дорівнює приблизно *2*. Це можна припустити із визначення для очікуваного значення кількості відрізків.

Розглянемо далі злиття, за допомогою якого при заданих відрізках ключів записів файла шляхом їхніх об'єднань утворюють один більш довгий відрізок. Насамперед, розглянемо простий ви-

падок, коли два відрізки входять у різноманітні файли. При цьому роблять порівняння початкових елементів кожного з відрізків: менший елемент видаляють, а на місце видаленого вставляють наступний за видаленим елемент. Цей елемент стає новим початковим елементом відповідного відрізка. Цю операцію повторюють доти, поки один із відрізків не закінчиться; елементи іншого відрізка, що залишилися, виводять, не змінюючи порядку (рис. 7.19):

2 4 5 6 8 9 ← { 5 6 / 2 4 8 9 }

Рис. 7.19. Схема двошляхового злиття

Спробуємо поширити двошляхове злиття на випадок, коли в кожному з двох вхідних файлах є по декілька відрізків. При цьому незалежно від кількості відрізків в обох файлах злиття буде провадиться тільки в один відрізок (рис. 7.20).

2 4 5 6 8 9 1 3 7 ← { 5 6 3 / 2 4 8 9 1 7 }

а

2 4 5 6 8 9 ∞ 137 ∞ ← { 5 6 ∞ 3 ∞ / 2 4 8 9 ∞ 137 ∞ }

б

1 2 3 5 6 7 9 ← { 2 3 6 9| / 1 5 7| } ← { 2 9|1 7 / 3 6 |5| } ← { 9|6|1|5| / 2|3|7| }

в

Рис. 7.20. Злиття файлів, що містять декілька відрізків.

Алгоритм злиття відрізків двох файлів полягає в наступному:

1. Увести початкові елементи кожного файла. Якщо один із відрізків є останнім у файлі, перейти до опрацювання файла.
2. Повторювати до закінчення якогось із відрізків наступне:
   2.1. Порівняти два елементи і вивести той, що не більше.
   2.2. Ввести елемент файла на місце елемента, що був видалений.
      2.2.1. Якщо кінець відрізка не досягнутий, перейти до кроку 2.1.
      2.2.2. Якщо кінець відрізка досягнуто, виводити елементи іншого файла до закінчення відрізка і перейти до кроку 1.
3. (Опрацювання кінця файла.) Якщо в іншому файлі є відрізки (хоча б один), робити ввод-вивід елементів цих відрізків без зміни порядку розташування елементів, закінчити виконання алгоритму.

На відміну від попереднього алгоритму в наступному — необхідно визначити кінець відрізка. Це можна зробити різними способами:

1. Ключ видаленого запису порівнюють із ключем запису, що вставляється. Якщо значення ключа нового запису менше, то відрізок закінчено (рис. 7.20, *а*).

2. В якості мітки, що вказує на кінець відрізка, установлюють спеціальний ключ, в якості якого вибирають число, що не входить у діапазон значень ключа. Як тільки буде введене це значення, кінець відрізка досягнуто.

При використанні цього методу необхідно обов'язково переконатися в тому, що обрана ознака кінця відрізка не входить у діапазон значень ключа (рис. 7.20, *б*).

3. Відрізки у файлах роблять фіксованої довжини.

Розглянемо ще один метод. Не звертаючи уваги на вміст заданого вихідного файла, його розглядають як складений із відрізків довжиною *l*. Розділяючи цей файл на два підфайли і здійснюючи їхнє злиття, одержують відрізки довжиною 2. Повторюючи

цю операцію, доводять довжину відрізків до 2*k* (*k*=0,1,2, ...). Якщо при цьому довжина файла *n* не є ступенем *2*, то по закінченню злиття останній відрізок буде мати надлишок елементів, рівний *n*/2*k*. На рис. 7.20, *в* наведено приклад, у якому для простоти в якості файла використано сім однорозрядних чисел.

Зазвичай, при двошляховому злитті з 2*r* відрізків середньої довжини *l* одержують *r* відрізків середньої довжини 2*l*, тобто якщо починають з відрізків довжиною *l*, то для отримання довжини відрізка 210=1024 буде потрібно десять злиттів, тому при зовнішньому сортуванні необхідно починати з файла, що містить найдовші відрізки.

7.9. Приклади програм створення та модифікації файлу в *Pascal* та *C++*

Програма виконує вставку нових записів до файлу, сортування даних, пошук необхідних записів за параметрами, видалення записів, а також перегляд всіх раніше введених даних (*Pascal*):

```
Uses crt;
Type
        Zap: string [11];
                Tel :string [11];
        Mg:string [20];
        Mr:string [20];
        Pas :string[20];
End;
Var
        A:array [1.. 20 ] of zap;
        F,f1,f3:file of zap   ;z1,z2:zap; s,st:string ; x,y,i,j:integer;
flag,f1:Boolean ;
        key:byte ; c:char ; f3:longint ;
Program Knop;{Визначити яка кнопка була натиснута}
```

```
        begin
        flag::=false ; key:=0;
        if keypressed then
                begin
        key :=ord(readkey);
        if key=0 then
                        begin
        key:=ord(readkey); flag=true;
                        end;
                end;
else
        mov ax ,3{зчит.клав. і коорд. миші }
        int 33 h{перерив. миші}
        mov key ,bl {читан. Яка клав .миші вкл.1-лв ,2-пр ,3-лв і
прав, 4-серед.}
        mov x,cx{коорд по x}
        mov y,dx{коорд по y}
        end;
end;
Procedure Vkl;
Begin
        Clrscr;
        Seek (f,filesize(f));
        Write ('введіть прізвище :');
        Readln(z1.fam);
         Write ('введіть телефон:');
        Readln(z1.tel);
        Write ('введіть місце проживання:');
        Readln(z1.mg);
        Write ('введіть паспортні данні:');
        Readln(z1.pas);
        Write (f,z1);
```

```
End;
Procedure Pros;
    Begin
    Clrcsr; Textcolor (4); Gotoxy (1,1);
    Write ('прізвище ':–13); Gotoxy (13,1);
    Write ('телефон':–12); Gotoxy (25,1);
    Write (місце проживання':–21); Gotoxy (46,1);
    Write ('місце роботи':–22); Gotoxy (68,1);
    Write ('паспорт':–12); Textcolor (yellow);
    i:=0; fs:=filesize(f);
    while fs>i do
        begin
seek(f,i); read (f,z1); Gotoxy (1,i+2); Write (z1.fam:–12);
Gotoxy (13,i+2); Write (z1.tel:–12); Gotoxy (25,i+2); Write (z1.mg:–22);
Gotoxy (46,i+2); Write (z1.mr:–22); Gotoxy (68,i+2); Write (z1.pas:–12);
i:=i+1;
        end;
readkey;
end;
Procedure Del(st:string);
Begin
    Fl:=false; Assign (f1,'noname.bak'); Rewrite (f1);
    i:=0; fs :=filesize (f);
    while fs>i do
        begin
    seek(f,i); read (f,z1);
    if z1.fam <> st then
            begin
    write (f1,z1);
            end
```

```
        else
        f1:=true; i:=i+1;
                end;
close(f); erase (f1); close(f1); rename(f1,s); assign(f,s); reset(f);
end;
Procedure Sort;
Var
j:integer;
        begin
f1:=false; assign(f3,'noname1.bak'); rewrite (f3); j:=0;
repeat
seek(f,0); read(f,z2); i:=1; fs:=filesize(f);
while fs>i do
                begin
        seek(f,i); read(f,z1);
        if z1.fam<z2.fam then
        z2:=z1; i:=i+1;
                end;
        write (f3,z2); del(z2.fam); j:=j+1;
        until i=1;
        close(f);   erase(f);   close(f3);   rename(f3,s);   assign(f,s);
reset(f);
        write ('сортування закінчене');
        readkey;
end;
Procedure Poisk(j:integer);
Begin
        Clrscr;
        Write('введіть '); St:='';
        Case j of
        1:begin write('прізвище:'); end;
        2 :begin write('телефон:'); end;
```

```
            3:begin  write('адреса;'); end;
            4 :begin write ('місце роботи:'); end;
            5: begin write('паспорт:'); end;
    end;
    readln; readln(st); i:=0;
    fs:=filesize (f);
    while fs>1 do
                begin
    seek(f,i); read(f,z1); begin
    wrieln('прізвище–',z1.fam); writeln('телефон–',z1.tel);
    writeln('місце проживання'–,z1.mg); writeln('місце роботи'–
,z1.mr);
    writeln('паспорт'–,z1.pas);
                end;
    if (z1.tel=st) and (j=2)then
    begin
    wrieln('прізвище–',z1.fam); writeln('телефон–',z1.tel);
    writeln('місце проживання'–,z1.mg); writeln('місце роботи'–
,z1.mr);
    writeln('паспорт'–,z1.pas);
    end;
    if (z1.mg=st) and (j=3)then
    begin
    wrieln('прізвище–',z1.fam); writeln('телефон–',z1.tel);
    writeln('місце проживання'–,z1.mg); writeln('місце роботи'–
,z1.mr);
    writeln('паспорт'–,z1.pas);
    end;
    if (z1.mr=st) and (j=4)then
    begin
    wrieln('прізвище–',z1.fam); writeln('телефон–',z1.tel);
```

```
writeln('місце проживання'–,z1.mg); writeln('місце роботи'–
,z1.mr);
writeln('паспорт'–,z1.pas);
end;
if (z1.pas=st) and (j=5)then
begin
wrieln('прізвище–',z1.fam); writeln('телефон–',z1.tel);
writeln('місце проживання'–,z1.mg); writeln('місце роботи'–
,z1.mr);
writeln('паспорт'–,z1.pas);
end;
i:=i+1;
end;
writeln('The end'); readkey; clrscr;
end;
Procedure Fon;
Begin
Clrscr;
Writeln('F1 вставка '); Writeln('F2 перегляд'); Writeln('F3 со-
ртування ');
Writeln('F4 видалення запису'); Writeln('F5 пошук');
Writeln('F10 вихід');
End;
Begin
Textbakground(1); Textcolor(yellow); Clrscr;
Write('введіть ім'я файлу:'); Rideln (s);
If s='' then s:='noname.dan';
Assign(f,s);
{$I–}
reset(f);
{$I+}
```

```
if ioresult<>0 then
begin
rewrite(f); close(f); reset(f);
end;
Fon;
Repeat
Knop;
If ((key=59)and(flag=true)) then
Begin
Vkl; Fon;
End;
If ((key=60)and(flag=true)) then
Begin
Pros; Fon;
End;
If ((key=61)and(flag=true)) then
Begin
Sort; Fon;
End;
If ((key=62)and(flag=true)) then
Begin
Clrscr; Write ('Введіть прізвище для видалення :'); Readln
(st); Del(st);
If fl then write('запис із прізвищем',st,'видалений')
Else
Write('прізвище',st,'не знайдене'); Ridkey; Fon;
End;
If ((key=63)and(flag=true)) then
Begin
```

```
Clrscr;  Writeln('1–по прізвищу'); Writeln('2–по телефону');
Writeln('3–по адресі'); Writeln('4–по роботі'); Writeln('5–по паспо-
рту');
Repeat
{$I–}
read (j);
{$I+}
until ioresult=0;
Poisk(j); Fon;
End;
If ((key=68)and(flag=true)) then
Break; Until (false);
Close(f);
End.
```

У *С++* файл відкривається шляхом зв'язування його з потоком. Як ви знаєте, існують потоки трьох типів: уведення, виводу і введення-виводу. Щоб відкрити вхідний потік, необхідно оголосити потоковий об'єкт типу *ifstream*. Для відкриття вихідного потоку потрібно оголосити потік класу *ofstream*. Потік, що передбачається використовувати для операцій як уведення, так і виводу, повинен бути оголошений як об'єкт класу *ofstream*. Наприклад, при виконанні наступного фрагмента коду буде створено вхідний потік, вихідний і потік, що дозволяє виконувати операції в обох напрямках [4]:

```
ifstream in;                // вхідний потік
ofstream out;               // вихідній потік
fstream both;               // потік уведення-виводу
```

Створивши потік, його потрібно зв'язати з файлом. Це можна зробити за допомогою функції *open*(), причому в кожнім із трьох потокових класів є своя функція-член *open*(). Представимо їх прототипи.

```
void ifstream: : open (const char *filename,
        ios::openmode mode = ios::in);
void ofstream::open (const char *filename,
        ios::openmode mode = ios::out | ios::trunc);
void ofstream::open (const char *filename,
        ios::openmode mode = ios::in | ios::out);
```

Тут елемент *filename* означає ім'я файлу, що може включати специфікатор шляху. Елемент *mode* визначає спосіб відкриття файлу. Він повинен приймати одне або кілька значень перерахування *openmode*, що визначені в класі *ios*. Нижче приведені значення цього перерахування.

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Кілька значень перерахування *openmode* можна поєднувати за допомогою логічного додавання (АБО). Вставка значення *ios::app* у параметр *mode* забезпечить приєднанням до кінця файлу всіх даних, що виводяться. Це значення можна використовувати тільки до файлів, відкритих для виводу даних. При відкритті файлу з використанням значення *ios::ate* пошук буде починатися з кінця файлу. Не дивлячись на це, операції введення-виводу можуть як і раніше виконуватися по всьому файлі.

Значення *ios::in* говорить про те, що даний файл відкривається для вводу даних, а значення *ios::out* забезпечує відкриття файлу для їх виводу.

Значення *ios::binary* дозволяє відкрити файл у двійковому режимі. По замовчуванню всі файли відкриваються в текстовому режимі. У текстовому режимі можуть відбуватися деякі перетворення символів (наприклад, послідовність, що складається із символів по-

вернення каретки і переходу на новий рядок, може бути перетворена на символ нового рядка). Але при відкритті файлу в двійковому режимі ніякого перетворення символів не виконується. Варто мати на увазі, що будь-який файл, що містить форматований текст або ще неопрацьовані дані, можна-відкрити як у двійковому, так і в текстовому режимі. Єдине розходження між цими режимами складається в перетворенні (або не перетворенні) символів.

Використання значення *ios::trunc* приводить до руйнування змісту файлу, ім'я якого збігається з параметром *filename*, а цей файл скорочується до нульової довжини. При створенні вихідного потоку типу *ofstream* будь-який існуючий файл з ім'ям *filename* автоматично скорочується до нульової довжини.

При виконанні наступного фрагмента коду відкривається звичайний вихідний файл.

```
ofstream out;
out.open("тест") ;
```

Оскільки параметр *mode* функції *open*() по умовчанню встановлюється рівним значенню, що відповідає типові потоку, який відкривається, в попередньому прикладі взагалі немає необхідності задавати його значення.

Невідкритий у результаті невдалого виконання функції *open*() потік при використанні в булевом вираженні встановлюється рівним значенню НЕПРАВДА. Цей факт може служити для підтвердження успішного відкриття файлу, наприклад, за допомогою такої *if*-інструкції:

```
if (!mystream){
cout << "Не вдалося відкрити файл.\n";
// опрацювання помилки }
```

Перш ніж робити спробу одержати доступ до файлу, варто завжди перевіряти результат виклику функції *open*(). Можна також перевірити факт успішного відкриття файлу за допомогою

функції *is_open*(), що є членом класів *fstream, ifstream* і *ofstream*. Ось її прототип:

```
bool is_open() ;
```

Ця функція повертає значення ІСТИНА, якщо потік зв'язано з відкритим файлом, і НЕПРАВДА — у протилежному випадку. Наприклад, використовуючи наступний код, можна довідатися чи відкритий у даний момент потоковий об'єкт *mystream*:

```
if (!mystream.is_open () ){
cout << "Файл не відкрито \n";
// ...
```

Хоча цілком коректно використовувати функцію *open*() для відкриття файлу, в більшості випадків це робиться по-іншому, оскільки класи *ifstream, ofstream* і *fstream* включають конструктори, що автоматично відкривають заданий файл. Параметри в цих конструкторах і їх значення (що діють по умовчанню) збігаються з параметрами і відповідними значеннями функції *open*(). Тому найчастіше файл відкривається так, як це показано в наступному прикладі:

```
ifstream mystream ("myfile") ;    // файл відкривається для вводу
```

Якщо з якоїсь причини файл відкрити неможливо, потокова змінна, що зв'язується з цим файлом, устанавлюється рівною значенню НЕПРАВДА.

Щоб закрити файл, використовуйте функцію-член *close*(). Наприклад, щоб закрити файл, зв'язаний з потоковим об'єктом *mystream*, використовуйте таку інструкцію:

```
mystream.close ();
```

Функція *close*() не має параметрів і не повертає ніякого значення.

Найпростіше зчитувати дані з текстового файлу або записувати їх у нього за допомогою операторів << та >>. Наприклад, у наступній програмі виконується записування до файлу *test* цілого числа, дійсного числа і рядка [4].

```
// Запис даних у файл.
#include <iostream>
 #include <fstream>
 using namespace std;
int main()
{
ofstream out ("test");         // Створюємо файл з ім'ям  test та
                                // відкриваємо його для виводу даних.
if(!out) {
cout << "Не вдалося відкрити файл.\n";
return 1;
}
out << 10 <<  " " << 123.23 << "\n";              //     Виводимо
дані в файл
out << "Це короткий текстовий файл.";
out.close ();                                   // Закриваємо файл.
return 0;
}
```

Наступна програма зчитує ціле число, *float*-значення, символ і рядок з файлу, створеного при виконанні попередньої програми.

```
// Считывание данных из файла.
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
char ch; int i; float f; char str[80];
ifstream in ("test");          // Відкриваємо файл для вводу даних
 if(!in) {
cout << "Не вдалося відкрити файл \n";
return 1; }
in >> i;                                     // Зчитуємо дані з файла
```

```
in >> f;
 in >> ch;
 in >> str;
cout << i<< " " <<f<< " " << ch << "\n";
cout << str;
in.close();                          // Закриваємо файл.
return 0; }
```

Варто мати на увазі, що при використанні оператора >> для зчитування даних з текстових файлів відбувається перетворення деяких символів. Наприклад, символи *Space* опускаються. Якщо необхідно заборонити які б то ні було перетворення символів, відкрийте файл у двійковому режимі доступу. Крім того, пам'ятайте, що при використанні оператора >> для зчитування рядка, введення припиняється при виявленні першого символу *Space*.

Щоб зчитувати і записувати до файлу блоки даних, використовуються функції-члени *read*() та *write*(). Їх прототипи мають наступний вигляд:

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

Функція *read*() зчитує *num* байт даних зі зв'язаного з файлом потоку і розміщує їх у буфер, що адресується параметром *buf*. Функція *write*() записує *num* байт даних до зв'язаного з файлом потоку з буфера, що адресується параметром *buf*.

При виконанні наступної програми спочатку до файлу записується масив цілих чисел, а потім він же зчитується з файлу [4].

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
int n[5]={1,2,3,4,5};
register int i;
```

```
ofstream out (:test", ios::out| ios::binary);
if(!out){
cout<<"Не вдалося відкрити файл\n";
return 1;
}
out.write((char*)&n, sizeof n);          //Записуємо блок даних
out.close();
for(i=0;i<5;i++)                                  // очищуємо масив
n[i]=0;
ifstream in("test", ios::in | ios::binary);
if(!in){
cout<<"Не вдалося відкрити файл\n";
return 1;}
in.read((char*) &n, sizeof n);           // Зчитуємо блок даних
for(i=0; i<5; i++)//Відображаємо значення, зчитані з файлу
cout<< n[i]<< " ";
in.close();
return 0;
}
```

Зверніть увагу на те, що в інструкціях звертання до функцій *read*() та *write*() виконуються операції приведення типу, які є обов'язковими при використанні буфера, визначеного не у вигляді символьного масиву.

Якщо кінець файлу буде досягнуто до того, як буде зчитано *num* символів, функція *read*() просто припинить виконання, а буфер буде вміщати стільки символів, скільки вдалося зчитати до цього моменту. Точну кількість зчитаних символів можна узнати за допомогою ще однієї функції-члена *gcount*(), яка має такий прототип:

```
streamsie gcount();
```

Функція *gcount*(); повертає кількість символів, зчитаних в процесі виконання останньої операції вводу даних.

Контрольні запитання та завдання для самоконтролю

1. Що таке послідовний файл?
2. Загальна схема дії пошуку в послідовному файлі.
3. Поняття послідовного або лінійного пошуку.
4. В чому полягає двійковий пошук?
5. Основні операції з послідовними файлами і схеми їх виконання.
6. Що таке сортування?
7. В чому полягає відмінність внутрішнього і зовнішнього сортування?
8. Принцип дії сортування вставками.
9. Принцип дії сортування вибором.
10. Принцип дії сортування обміном. Сортування «пухирцем».
11. Принципи вибору метода сортування, їх недоліки та переваги.
12. Загальна схема сортування злиттям.
13. Сутність двошляхового злиття.
14. Створити програму, що реалізує сортування вставками. Місце розміщення чергового елемента до відсортованої частини визначити за допомогою двійкового пошуку. Двійковий пошук оформити у вигляді окремої функції.
15. У стилі рис. 7.14 показати, як сортується початковий файл EASYQUESTION методом вибору.
16. Якої максимальної величини досягає кількість обмінів для будь-якого конкретного елемента в процесі сортування вибором? Що собою представляє середня кількість обмінів, що приходиться на один елемент?
17. У стилі рис. 7.12 показати, як сортується початковий файл EASYQUESTION методом вставок.
18. Розробити програмну реалізацію сортування вставками, в якій у внутрішньому циклі використовується оператор

while, що завершується по одній з двох умов, опис яких приводиться в програмі.

19. У стилі рис. 7.16 показати, як сортується початковий файл EASYQUESTION методом «пухирцевого» сортування.

20. По тексту програми визначите алгоритм сортування, призначення окремих змінних і циклів:

Завдання №1:

```
void F1 (int in[],int n)
{ int i,j,k,c;
for (i = 1; i<n; i++){
for (k=i; k !=0; k—)
{ if (in[k] > in[k–1]) break;
c=in[k]; in[k]=in[k–1]; in[k–1]=c;
}
}}
```

Завдання №2:

```
void F2(int in[],int out[],int n)
{ int i,j ,cnt;
for (i=0; i< n; i++) {
for ( cnt=0,j=0; j<n; j++)
if (in[j] > in[i]) cnt++;
else
if (in[j]==in[i] && j>i) cnt+ + ;
out[cnt] = in[i] ;
} }
```

21. Визначити двійковий чи лінійний пошук розташування значення в масиві *out*[*n*] виконується в наступних програмах:

Завдання №1:

```
int find(int out[],int n, int val);
void F4(int in[], int n){
int i,j,k;
for (i = 1; i<n; i++)
{ int c; c = in[i]; к = find(in,i.c);
for (j = i; j ! = k ; j – – ) in[j] = in[j–1];
in[k] = c; }
}
```

Завдання №2:

```
void F5(int in[], int n){
int i,j,c,k;
for (i=0; i < n–1; i++){
for (j = i + l ,c = in[i],k = i; i<n; j++)
if (in[j] > c) { c = in[j]; k=j; }
in[k] = in[i]; in[i] = c;
}
}
```

Розділ 8. ІНДЕКСНО-ПОСЛІДОВНА ОРГАНІЗАЦІЯ

8.1. Основні поняття. Визначення використовуваних термінів

На практиці найбільш поширеною є індексно-послідовна організація. Саме цей вид організації і буде розглянуто далі. Прямий підхід до побудови індексу полягає в збереженні масиву з посиланнями на ключі й елементи, упорядкованого по ключах, з наступним використанням бінарного пошуку.

Метод доступу який поєднує послідовну організацію ключів з індексним доступом називається індексно-послідовним доступом (рис. 8.1.).

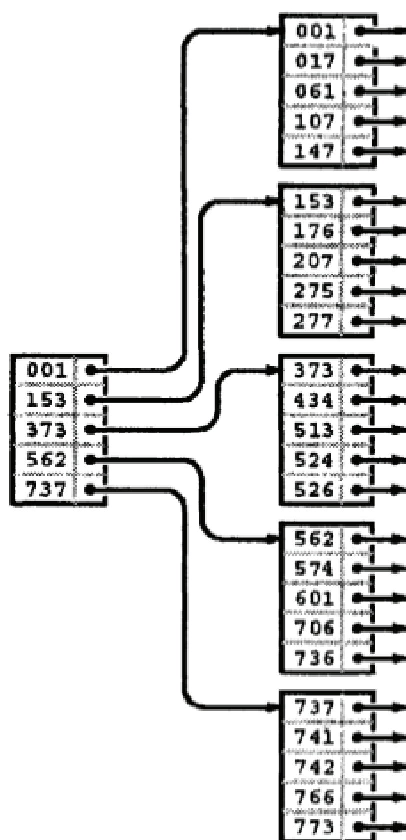


Рис. 8.1. Структура індексно-послідовного файлу

У послідовному індексі ключі зберігаються один за одним в повних сторінках (праворуч), причому індекс указує на найменший ключ на кожній сторінці (ліворуч). Для додавання ключа буде потрібно перебудувати всю структуру даних.

Цей метод зручний для додатків, у яких зміни в базі даних виконуються рідко. Іноді сам індекс називають каталогом. Недолік використання індексно-послідовного доступу полягає в тому, що зміна каталогу є операцію, сполученою з великими витратами. Наприклад, для додавання одного ключа може знадобитися перебудова буквально всієї бази даних із присвоєнням нових позицій багатьом ключам і новим значенням індексам. Для подолання згаданого недоліку і забезпечення можливості збільшення бази даних в ранніх системах резервувалися зайві сторінки на дисках і зайвий простір на сторінках; але, у кінцевому рахунку, подібна технологія виявлялася не дуже ефективною в динамічних ситуаціях.

Для виконання пошуку в індексно-послідовному файлі потрібно виконання тільки постійну кількість зондувань, однак вставка може потребувати перебудови всього індексу. В даному випадку термін «постійне» використовується дещо «вільно» для позначення значення, що пропорційне $\log MN$ для великого значення M. Як уже відзначалося, це виправдано для розмірів файлів, що зустрічаються на практиці. Навіть при наявності 128-розрядного ключа пошуку, придатного для вказівки неймовірно великої кількості різних елементів, рівного 2128, елементів із даним ключем можна було б знайти за допомогою всього 13 зондувань, при 1000-позиційному розгалуженні. Ми не будемо розглядати реалізації, що забезпечують пошук і побудову індексів подібного типу, оскільки вони є спеціальними випадками більш загальних механізмів [8].

Нагадаємо, що над файлами будь-яких типів можливі такі дії: вставка запису, видалення запису, пошук запису, модифікація запису, реорганізація файла.

Надалі, при розгляданні матеріалу будуть необхідні такі визначення: Елемент даних або поле — найменша одиниця іменованих даних. Може складатися з будь-якої кількості бітів або байтів.

Агрегат даних — іменована сукупність елементів даних у середині запису, аналізована як єдине ціле. Наприклад, агрегат ДАТА може складатися з елементів даних МІСЯЦЬ, ДЕНЬ, РІК.

Запис — іменована сукупність елементів або агрегатів даних. Логічна одиниця даних, що може складатися з декількох фізичних блоків, розміщених на зовнішньому носії, в оперативній пам'яті і т.п.

8.2. Індексно-послідовні файли. Їх структура

Головна відмінна риса індексно-послідовних файлів полягає в наявності індексу, що дозволяє здійснювати менш упорядкований доступ до записів; інша особливість полягає в наявності засобів опрацювання доповнень до файла. Логічні дані розташовуються в одній або декількох областях даних (які мають також назву «первинна область даних», що позначається *PRIME*), які представляють собою послідовні набори даних. Крім цих областей даних, індексно-послідовний файл має області ще двох типів: довідникову область, або область індексів (*INDEX*) і область переповнення (*OVFLOW*) (рис. 8.2) [22].

Рис.8.2. Індексна організація файла

Послідовні файли зручні при переборі записів, але не дуже підходять у тих випадках, коли потрібно швидко знайти якийсь визначений запис. Для цього більш підходить індексна організація файлу, при якій запис має власне ім'я, назване індексом або ключем. Наприклад, якщо ключем є деяке символьне поле, то упорядкування може виконуватися в прямому або зворотному алфавітному порядку. Для роботи з індексним файлом операційна система заводить спеціальну таблицю індексів, у якій зберігаються всі імена записів файлів, їх довжина і відповідні посилання — внутрішні імена.

При індексній організації таблиця індексів повинна містити індекси всіх записів файлу. Підвищення ефективності пошуку записів досягається за рахунок того, що пошук ведеться в таблиці індексів, яка має значно менші розміри, ніж область даних. Найвища ефективність досягається в тих випадках, коли індекси в таблиці упорядковані і можливе застосування швидких методів.

Область переповнення призначена для розміщення записів, витиснутих із доріжок основної області при додаванні записів до раніше сформованого набору. Кожний циліндр може мати свою область переповнення. Допускається також створення незалежної області переповнення, куди записи будуть надходити при відсутності вільного місця на доріжках переповнення циліндрів. Переповнений файл повинен бути підданий реорганізації, в ході якої сам файл створюється заново на новому місці, записи з областей переповнення займають належні місця в областях даних, а області переповнення стають вільними. Відповідним чином корегується і вміст довідникової області.

8.3. Поняття індексу. Одно- та багаторівневі індекси

Якщо файл упорядковано по ключах, то, зазвичай, використовується таблиця названа індексом, довідником або таблицею індексів. При звернені до таблиці задається ключ шуканого запису, а результатом процедури пошуку в таблиці є відносна або абсолютна адреса в зовнішній пам'яті (у нашому випадку — у файлі). Індекс складається з набору елементів, кожний з яких містить значення ключового атрибута, що відповідає деякому запису даних, і покажчик, що дозволяє здійснювати безпосередній доступ до цього запису [15]. Для записів великих розмірів елементи індексу займають значно менше місця, ніж сам запис даних. Отже весь індекс буде меншим за файл, тому що пошук буде проводитися по меншій області. Для того, щоб здійснювати швидкий пошук у самому індексі, його елементи упорядковують відповідно до заданого атрибута, навіть якщо файл упорядковано іншим способом. Індекс можливо визначити як таблицю, з якою пов'язана процедура, що отримує на вході інформацію про деякі значення атрибутів і видає на виході інформацію, що сприяє швидкій локалізації запису або записів, які містять задане значення атрибутів. Первинним індексом є індекс, що використовується в якості вхідної інформації, ідентифікатор запису (первинний ключ) і видає на виході інформацію, що відноситься до фізичної адреси даного запису. Повторний індекс — це індекс, що використовує в якості вхідної інформації.

Індекс містить певним чином підібрані ключі записів і відповідні їм адреси, по яких можна знайти ці записи у файлі. Початок кожного циліндра на диску, відведеного під файл, займає індекс доріжок. У ньому на кожну доріжку області даних заведено елемент, що містить дві частини — нормальний елемент (або інша його назва — нормальний вхід) і елемент переповнення (вхід переповнення).

Нормальний елемент містить адресу першого запису (фактично номер доріжки) і ключ останнього запису на цій доріжці. Елемент переповнення повинен містити ключ самого старшого запису й адресу самого молодшого запису на доріжці переповнення. При створенні (або реорганізації) індексно-послідовного файлу, коли доріжки переповнення порожні, елементи переповнення збігаються за значенням із нормальними елементами своїх доріжок.

Індекс доріжок — індекс найнижчого рівня, він будується завжди системою. Якщо файл займає декілька циліндрів, система будує індекс більш високого рівня — індекс циліндрів. Принцип його побудови аналогічний: кожний елемент індексу циліндрів відповідає визначеному циліндрові і містить номер циліндра і ключ останнього запису на циліндрі. При дуже великому обсязі файлу можливе створення індексу вищого рівня — головного індексу, що полегшує пошук в індексі циліндрів (рис. 8.3.)

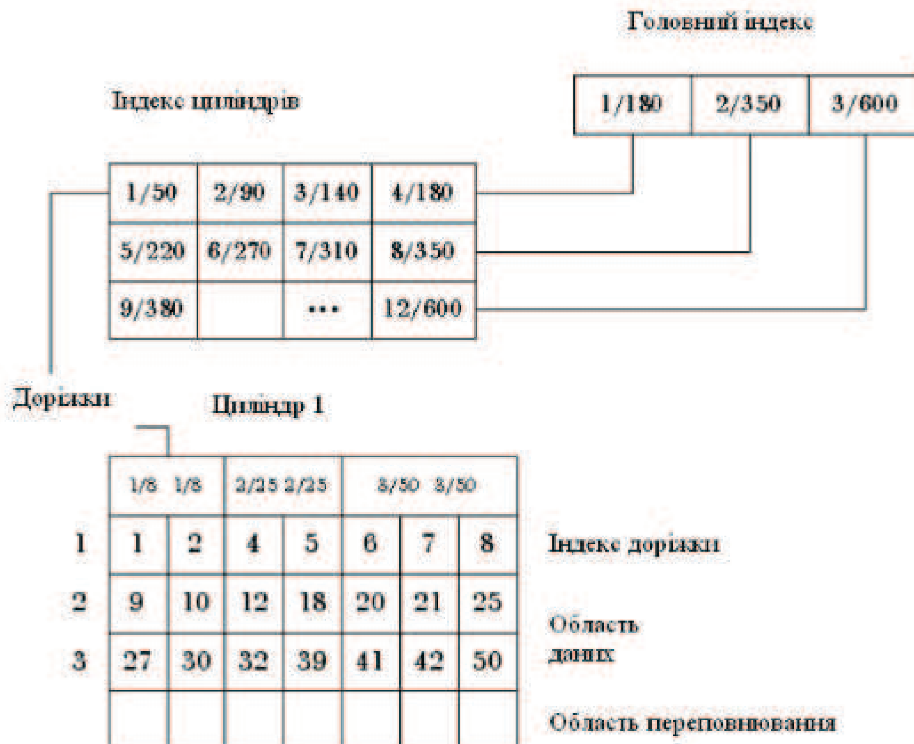


Рис. 8.3. Фізичне розташування індексно-послідовного файлу

Дробові числа на рис. 8.3 означають: чисельник — номер доріжки або номер циліндра; знаменник — значення ключа. В індексі доріжок показані елементи — нормальні і переповнення. Файл показано після створення, тому значення нормальних елементів і елементів переповнення доріжок збігаються.

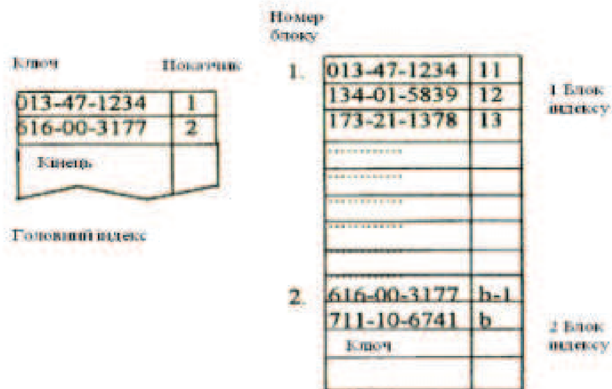
При створенні індексно-послідовного файлу система спочатку виділяє пам'ять необхідного обсягу, потім поділяє її на частини, необхідні для розміщення областей індексів, даних і переповнення, далі розміщає записи в областях даних і заповнює адресами і значеннями елементи індексів необхідних рівнів.

Розглянемо приклад індексно-послідовного файлу СЛУЖБОВЦІ із такою структурою: СЛУЖБОВЦІ (ТАБ_ НОМЕР, ІМ'Я, ПРІЗ, ПО БАТ., ПОС,…, ПП), де: ТАБ _ НОМЕР — табельний номер службовця; ІМ'Я — ім'я; ПРІЗ — прізвище службовця; ПО БАТ — по батькові; ПОС — посада; ПП — покажчик переповнювання. Схематично організацію індексно-послідовного файлу СЛУЖБОВЦІ можна представити, як показано на рис. 8.4. [19].

На рис. 8.4 показано тільки поля ТАБ_НОМЕР, ІМ'Я інші поля опущено. Обравши ТАБ_НОМЕР ключовим атрибутом вважаємо, що файл упорядковано по табельному номеру. Для наведеного приклада розбивка індексу провадилась по табельному номеру: файл був проіндексований по початкових цифрах, а потім був складений індекс другого рівня (головний індекс), що складається всього з двох елементів (013-47-1234 і 616-00-3177), який поділяє індекс першого рівня на дві частини.

У залежності від розміру файла і складності його структури запису, індекс може мати декілька рівнів (рис. 8.5).

Кількість рівнів індексу може бути довільним, у залежності від розміру файла даних. Для нашого приклада доцільно застосовувати дворівневий індекс, як це показано на рис. 8.4, *а*.



а

№ блоку запису	ТАБ_НОМЕР	ІМ'Я	Інші дані	Покажчик переповнення
11.1	013-47-1234	Василь		-
11.2	028-18-2341	Ганна		111.1
11.2	128-15-3415	Юлія		110.2
12.1	134-01-5859	Олена		-
12.2	143-09-0711	Євген		111.2
12.3	156-88-4321	....		-
13.1	193-21-1378	....		-
13.2	....	....		-
57.1	616-00-3177	Ігор		-
	633-89-9343	Алла		110.3
	704-43-0314	Інна		-
58.1	711-10-6741	Петро		-
	743-51-0030	Олексій		-
	896-43-4111	Іван		-

Записи в файлі СЛУЖБОВЦІ

б

Записи в файлі СЛУЖБОВЦІ

110.1	075-17-6317	Ірина		-	Блок переповнення 1
110.2	129-14-5301	Сергій		-	
110.2	676-43-1701	Зінаїда		-	
111.1	063-76-3340	Валерій		110.1	Блок переповнення 2
111.2	151-73-1901	Володимир		-	
Вільний простір		....	....	....	
....	....	....	....	....	
....	....	....	....	....	

в

Рис. 8.4. Організація індексно-послідовного файлу СЛУЖБОВЦІ:
а — індекс для файлу СЛУЖБОВЦІ; *б* — записи в файлі СЛУЖБОВЦІ;
в — область переповнення

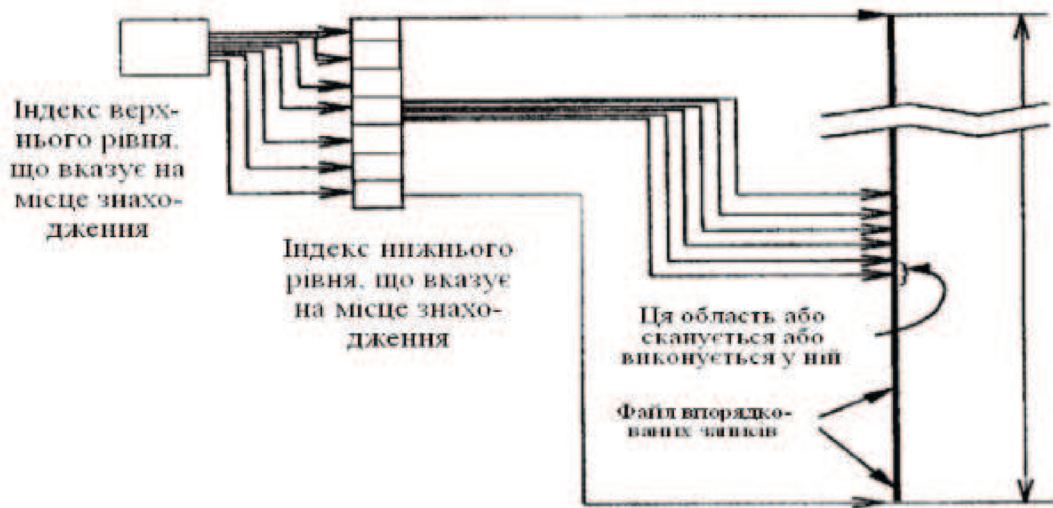


Рис. 8.5. Багаторівневий індекс

З ростом рівня індексу розмір індексу зменшується. Індекси верхнього рівня мають невеличкі розміри і можуть зберігатися в оперативній пам’яті. Необхідність мати багато рівнів індексації виникає рідко. Альтернативою багаторівневому індексуванню є використання методу двійкового або блокового пошуку по індексному файлу.

8.4. Основні операції над індексно-послідовними файлами

Для локалізації заданого запису використовується індекс. Процес вибірки складається з перегляду головної таблиці, установлення механізму читання-запису, зчитування індексу й зчитування блока даних. При цьому під час зчитування запису необхідно аналізувати переповнювання (якщо такі існують).

Під вибіркою розуміють доступ до деякого компонента даних — запису файлів. Для того, щоб отримати вибраний запис з індексно-послідовного файлу, його пошук треба починати від поточного запису даних, не беручи до уваги індекс. Необхідно

визначити, чи може упорядковане зчитування записів виконуватися послідовно або треба перейти до іншої області. У залежності від розташування попереднього і наступного запису можливі різні варіанти пошуку.

Всі основні операції з файлом базуються на операції вибірки запису.

Для того, щоб здійснювати додавання записів до індексно-послідовного файлу необхідно зарезервувати деяку вільну область. Записи можна розміщати в окремому файлі або зарезервувати для них область у кожному блоці. Виділення області в кожному блоці можливо тільки в тих випадках, коли блоки мають великі розміри і додавання достатньо розподілено, в протилежному випадку ця область може швидко заповнитися. Розумного компромісу можна досягти, якщо резервну область лишати після кожного блока (рис. 8.6), створюючи додаткові порожні записи між кожним через 2, *3,…, n* записом у залежності від інтенсивності внесення нової інформації. Цей метод називається методом розподілення вільної пам'яті (*distributed free space*).

1	Иванов	
2		
3	Петров	
4		
5	Сидоров	
…		
n		

Рис.8.6. Створення додаткових порожніх записів

В якості додаткової області пам'яті також можна зарезервувати повторну область переповнювання, що використовується в тих випадках, коли область переповнювання якогось блока сама переповнюється.

Додавання записів до індексно-послідовного файлу відбувається в такий спосіб. Спочатку визначається місце на доріжці, де у відповідності зі значенням ключа повинен розташовуватися запис. Далі система зрушує вправо записи, що мають ключ, більший, ніж у запису, що додається, звільняючи тим самим для нього місце. У результаті новий запис займає належне місце у файлі, а з доріжки видаляється запис з максимальним для даної доріжки ключем. Нове значення максимального ключа доріжки записується в нормальний елемент. Витиснутий запис переноситься в область переповнення і в елементі переповнення корегується облікова інформація.

Алгоритм вставки запису (або записів) в індексно-послідовний файл має наступний вигляд:

1. Проходимо по ланцюжку індексів до досягнення деяких даних.

2. Коли дані знайдено, перевіряємо, чи немає можливості записати нові дані в деяке вільне місце.

3. Якщо така можливість є — записуємо.

4. Інакше, створюємо покажчик на область переповнювання, що містить внесену інформацію.

Для того, щоб показати існування доданого запису, використовується покажчик. Якщо кожному запису файла відповідає елемент індексу, то в індекс може бути додано також покажчик переповнювання. При опрацюванні послідовної частини файла ці покажчики в елементах індексу використовуються для доступу до доданих записів.

Якщо при внесенні інформації в область переповнювання виникла колізія з вже розташованими там даними — розташовуємо їх в покажчик переповнювання — це називається методом розділених ланцюжків. На рис. 8.7 наведено вставка двох нових записів методом зчеплених ланцюжків.

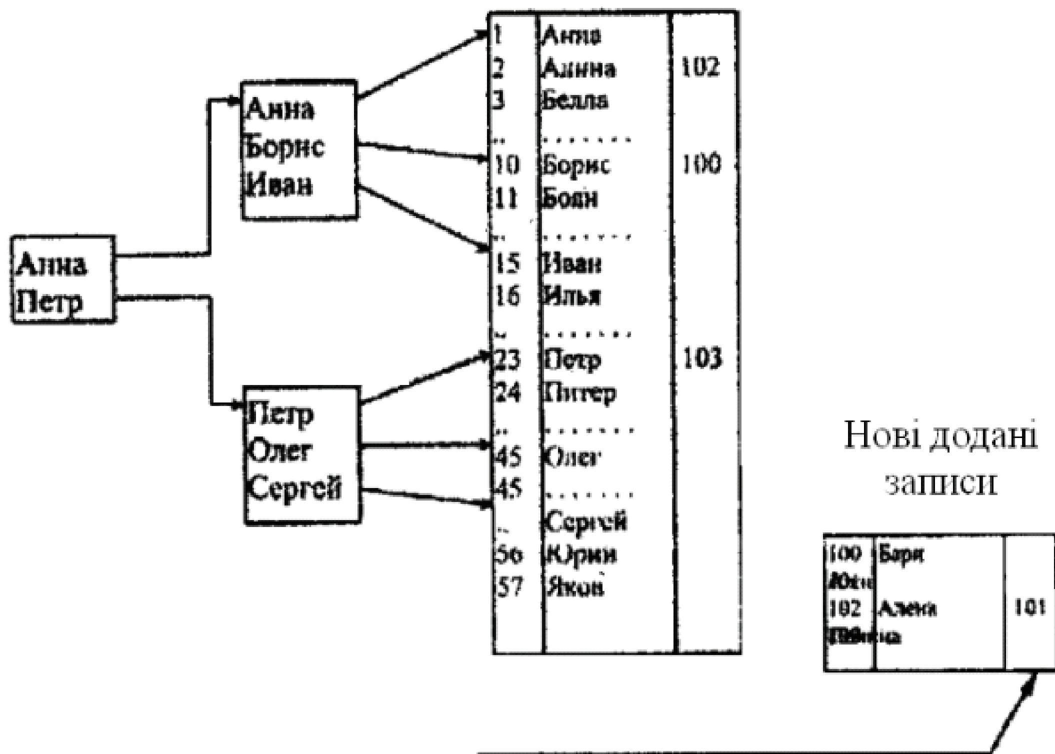


Рис. 8.7. Операція вставки до індексно-послідовного файлу

Області переповнювання при додаванні нових записів, зазвичай, зростають, тому що записи надходять у файл неупорядковані по ключу.

Пошук в індексно-послідовному файлі починається в головному індексі, звідси здійснюється посилання на потрібну частину індексу циліндрів. По індексі циліндрів система визначає номер циліндра, що містить шуканий запис. По індексу доріжки, розташованому на нульовій доріжці циліндра, визначається, на якій доріжці розташовано необхідний запис і в якій області — основній або в області переповнення (аналізуються відомості, що утримуються в нормальному елементі й елементі переповнення). Якщо запис знаходиться на основній доріжці, то він виявляється послідовним переглядом з аналізом ключів записів. Якщо необхідний запис був витиснутий в область переповнення, то перег-

ляд цієї області починається з запису, що має найменший ключ. Кожний запис має посилання на запис із великим значенням ключа. Така система посилань забезпечує пошук у напрямку зростання ключів і призводить до виявлення необхідного запису, якщо звичайно, він є у файлі.

Операція модифікації відрізняється від операції запису фактично тим, що при її виконанні у файл розміщують новий запис, але зі старим ключем. Для кожної операції в якості параметра вказується індекс запису, а для операцій запису і модифікації необхідна ще і довжина запису.

Видалення запису відбувається аналогічно, замість старого запису залишається «дірка», згодом файл фрагментується і нераціонально використовується дисковий простір. Таким чином, у залежності від інтенсивності використання файла необхідна його реорганізація, що буде розглянута нижче.

8.5. Анкерні крапки

Окремі записи в блоці можуть бути знайдені за допомогою послідовного пошуку, тому немає необхідності для кожного запису зберігати відповідний йому елемент індексу, а достатньо лише зазначити один запис у кожному блоці. Цей довідниковий запис називається анкерною крапкою, а в індексі зберігається тільки значення ключа анкерної крапки і покажчик блока [11]. Зазвичай, анкерні крапки вибираються для блоків. Витрати часу на пошук запису в блоці незначні, оскільки, коли потрібно, у пам’ять зчитується весь блок, який може зберігатися в буфері. Кількість записів у блоці дорівнює B/R, де B — розмір блока в байтах, R — довжина одного запису. Отже, кількість елементів в індексі дорівнює $V+P$, де V — розмір значення збереженого в елементі індексу, а P — розмір покажчика.

Якщо в індексі зберігаються тільки анкерні крапки блоків, то однією лише перевіркою індексу не можна визначити, чи існує запис, що відповідає даному аргументу. Необхідно також прочитати відповідний блок даних. Для того, щоб визначити, чи виходить значення аргументу пошуку за значення останнього елемента файла, необхідно вибрати останній блок даних, тому що анкерна крапка вказує на перший запис у цьому блоці. Якщо необхідність вставки записів у кінець файла виникає часто, зручніше зберігати в індексі значення ключа останнього (ніж першого) запису кожного блоку. Тоді для заданого аргументу відповідний блок обирається шляхом пошуку мінімального елемента індексу, який є не меншим за значення аргументу.

8.6. Коефіцієнт розширення індексу

Для оцінювання необхідної кількості рівнів індексації розглянемо приклад порівняно великого файла (один мільйон записів) і індексу скріпленого з блоками даних. Для того, щоб оцінити час доступу при виборці запису, варто визначити необхідну кількість рівнів індексу для файла такого розміру.

Приклад 8.1. Організація елементів індексу. Розмір індексу в нашому прикладі дорівнює (14+6)=20 байт, а розмір блока y, як і раніше дорівнює 2000. Отже, y=100, 105 елементів індексу нижнього рівня займають 103 блоки, для вказівки яких потрібно 103 елементи індексу другого рівня. Індекси другого рівня будуть займати пам’ять обсягом 20×(1000)=20000 байт, що є занадто великим для оперативної пам’яті, тому необхідно визначити третій рівень індексу. На вищому рівні буде потрібно усього 20000/2000=10 елементів, що будуть займати 200 байт. Термін головного індексу використовується для вказівки самого верхнього рівня. Рівні індексу нумеруються від 1 для найближчого до даних рівнів до x (тут 3) для головного рівня.

Можна зауважити, що блоки індексів містять велику кількість елементів, тобто мають великий коефіцієнт розширення, тому кількість рівнів невелика. У розглянутому вище прикладі в ході опрацювання два рівні індексу будуть зчитуватись з диска, а головний індекс залишається доступним в оперативній пам'яті. Для вибірки запису даних буде потрібно прочитати три блока.

8.7. Доступ до області переповнювання за допомогою індексу

При опрацюванні послідовної частини файлу покажчики в елементах індексу використовуються для доступу до доданих записів. У цьому випадку при пошуку заданого запису рішення про перехід може бути прийнято на основі інформації, що міститься в індексі, при цьому вибірка первинного блока не потрібна. Відповідний індекс показано на рис. 8.8.

Рис. 8.8. Схема розташування покажчиків на новий запис

Покажчики на вставку записів розташовуються з попередніми записами в первинних блоках даних. Ключ доданого запису тут не збігається; в послідовний файл розміщається тільки пока-

жчик так, що пошук будь-якого проміжного запису переноситься на область переповнювання. При додаванні записів за допомогою даної процедури не потрібно модифікувати індекс, проте для кожної вибірки доданого запису додатково зчитується один блок. Запит на неіснуючий запис вимагає переходу до файла переповнювання, коли аргумент пошуку розташований за ключем первинного запису з покажчиком на область переповнювання.

Первинний файл не містить полів, що вказують на область переповнювання. У тих випадках, коли файл оновлюється необхідно ввести індекс, що розширюється. Для запропонованої вище схеми, введення індексу спрощується шляхом розташування покажчика переповнювання з кожним елементом запису, тобто шляхом подвоєння розміру індексу. Проте, з ростом розміру індексу зростає також час його опрацювання. Для індексів, приєднаних до записів даних, збільшення розмірів буде дуже значним. Цей метод є інтересним, головним чином, у тих випадках, коли область переповнювання і відповідні первинні записи розташовані на різних циліндрах так, що економія часу на вибірку стає суттєвою.

Для того, щоб визначити місце розташування декількох записів переповнювання з одного первинного запису покажчики поміщаються в записи, що знаходяться в областях переповнювання. Такі записи зв'язуються в ланцюжок, за допомогою декількох блоків області переповнювання. До цього ж ланцюжка може бути приєднаний новий запис так, що послідовний порядок зберігається.

У тих випадках, коли вибірка здійснюється серед великої кількості доданих записів, у великій кількості блоків, проходження по ланцюжку до визначеного запису, в дійсності, може виявитися менш ефективним, ніж звичайний вичерпний пошук по всій області переповнювання. З іншого боку, можливість перегляду ланцюжка істотно спрощує упорядковане опрацювання. Для того, щоб при опрацюванні даних не загубити дані з буфера послідовного файла, необхідно мати спеціальний буфер переповнювання.

а

11.1	013-47-1234		
	028-18-2341		
	063-76-5340		111.1
12.1	134-01-5839		
	143-09-0711		
	151-73-1901		111.2
13.1	173-21-1378		
	і т.д.		
b-1.1	616-00-3177		
	633-89-9343		
	676-43-1701		110.3
b.1	711-10-6741		
	743-51-0030		
	896-43-4111		–

Послідовний файл (первинна область)¶

110.1	128-15-3413		110.2
110.2	129-14-5301		–
110.3	704-43-0314		
	075-17-6317		110.1
111.1	156-88-4321		–
111.2	....		
111.3	....		

б

Рис. 8.9. Процес проштовхування при додаванні нового запису

Інший метод розміщення записів переповнювання заснований на зберіганні послідовності ключів у блоках первинного файла. Нові записи додаються після своїх безпосередніх попередників; наступні записи проштовхуються в напрямку до кінця блоку. Записи з кінця первинного блока проштовхуються в область переповнювання. На рис. 8.9, *а* представлено процес проштовхування, а кінцевий стан файла показано на рис. 8.9, *б*.

Тепер у первинному блоці необхідно мати тільки один покажчик на область переповнювання, і з кожного блока здійснюється тільки один перехід до файла переповнювання. Проте, файл переповнювання опрацьовується так само, як було показано вище, а ланцюжки будуть довші. Вибірка запису, переміщеного до області переповнювання, в середньому буде здійснюватися довше.

8.8. Реорганізація індексно-послідовних файлів та області використання

Необхідність у реорганізації файла виникає в той момент (або до того моменту), коли переповняються самі області переповнювання. Реорганізація може знадобитися й у тих випадках, коли через утворення довгих ланцюжків час, необхідний на вибірку або упорядковане опрацювання записів, стає неприпустимо великим. У процесі реорганізації файл зчитується так само, як при виконанні упорядкованого опрацювання, і записується заново. При цьому видаляються всі записи, що позначені, а всі нові і ті, що залишилися, послідовно записуються в головну область нового файла. У процесі цього опрацювання програми реорганізації створюються нові індекси, в яких використовуються нові анкерні крапки. Частота такої реорганізації залежить від інтенсивності вставки записів у файл. На практиці реорганізація проводиться від одного разу в день до одного разу на рік. Оскільки для реорганізації може знадобитися велика

кількість часу, вона, зазвичай, виконується до того, як файл цілком заповнюється, для того, щоб уникнути неприємних несподіванок під час роботи з файлом. Реорганізація може виконуватися або періодично із заздалегідь обчисленим періодом, або в зручний момент після того, як кількість елементів в області переповнювання перевищить визначену межу.

Індексно-послідовні файли широко використовуються при опрацюванні економічної інформації. Особливо часто вони використовуються в тих випадках, коли тимчасові інтервали, в яких необхідно зберегти нову копію файла, менші ніж інтервали опрацювання, припустимі при періодичній реорганізації послідовних файлів. Наприклад, для упорядкування списку товарів індексно-послідовний файл може використовуватися щодня, а його реорганізація (разом із процесом, що формує замовлення на товари, запаси яких недостатні) проводитися щотижня.

Індексно-послідовні файли також широко використовуються для опрацювання інформаційних запитів, але при цьому потрібно, щоб у записі був визначено ключовий атрибут. Ця вимога часом призводить до того, що в окремих індексно-послідовних файлах зберігаються копії тих самих даних, але упорядкованих по різних ключах. У цьому випадку зростають витрати на відновлення і збільшується необхідний обсяг пам'яті.

Контрольні запитання і завдання для самоперевірки

1. Що таке індексно-послідовна адресація ? Її визначення.
2. Поняття індексу. Первинний і повторний індекси.
3. Які атрибути можна використовувати при індексуванні? Чи можна використовувати повторні ключі?
4. Розкрийте поняття анкерних крапок.
5. Що таке коефіцієнт розширення індексу? Яким способом він обчислюється і чому необхідне його існування? Залежність

індексно-послідовної організації від комп’ютера — наскільки вона суттєва?

6. Організація переповнювання. У чому суть методу зчеплення ланцюжків?

7. Організація переповнювання. Суть методу проштовхування.

8. Реорганізація індексно-послідовного файла. Розповісти яким чином відбувається доступ до запису у файлі.

9. Вибірка записів з індексно-послідовного файла. Розповісти яким чином відбувається доступ до запису у файлі.

10. Намалювати блок-схему вибірки запису з індексно-послідовного файла і пояснити її.

Розділ 9. ФАЙЛИ ПРЯМОГО ДОСТУПУ

9.1. Введення

Для ознайомлення з файлами прямого доступу необхідно визначити основні поняття, що використовуються в термінології файлів. Адресний простір файла — множина адрес його блоків. Потужність такої множини дорівнює кількості блоків у файлі.

Простір ключів — множина можливих ключів. Підмножина такої множини — реальні значення, що в дійсності приймають ключі. Цю підмножину надалі будемо називати множиною фактичних значень ключів. Коефіцієнт завантаження файла — відношення кількості зайнятих блоків до загальної кількості блоків у файлі. Для порожнього файла коефіцієнт завантаження дорівнює нулю, а для повністю зайнятого коефіцієнт дорівнює одиниці.

Хеш-функція, функція рандомізації, функція перетворення ключа на адресу — це одне поняття.

Хешовані файли, файли прямого доступу, рандомізовані файли, файли з прямою організацією — це одне поняття.

Тепер будуть описані файли прямого доступу, для яких головним є саме прямий доступ до блоків.

Файл прямого доступу може створюватися тільки на пристріоі прямого доступу, що запам'ятовує (ППДЗ). Створивши файл прямого доступу і виділивши необхідну для нього область пам'яті на ППДЗ можна записувати дані в будь-які блоки файла і зчитувати дані з будь-яких блоків.

За бажанням програміста блоки у файлі прямого доступу можуть мати формат «лічильник-дані» або «лічильник — ключ — дані». Ключ ідентифікує відповідний йому блок і використовується для визначення місця у файлі при записі блока і при його зчитуванні. Значенням ключа може бути номер блока в деякій системі нумерації, символічне ім'я або будь-яка інша інформація, що однозна-

чно ідентифікує блок. Ключі широко застосовуються при зверненнях до файлів прямого доступу.

Пряма організація розрахована на довільну обробку записів, хоча припустима і послідовна. Ключ запису використовується для встановлення адреси цього запису у файлі. Систему розміщення і пошуку записів у файлі визначає програміст.

При прямій організації існує декілька типових способів адресації записів у файлі. Обмін з файлом здійснюється блоками. Якщо програміст хоче використовувати блок як сукупність логічних записів, він усі дії по блокуванню і деблокуванню повинен здійснювати сам у своїй програмі. У файлах із прямою організацією для вказівки місця розташування записів може бути використана абсолютна або відносна адресація.

Абсолютна адресація заснована на тому, що адреса кожного запису у файлі є її фізичною адресою на диску і містить номер циліндра, номер доріжки і номер блоку на доріжці. У силу цього даний спосіб адресації називається також фізичним. Головний недолік такого способу — тверда фіксація місця запису на диску, в результаті чого стає неможливим переміщення такого файлу до зовнішішньої пам'яті. Перевага — висока швидкість пошуку.

При використанні відносної адресації записів програміст застосовує спеціальні прийоми, що дозволяють однозначно вказувати місце розташування записів на диску. Відносна адресація використовує порядкові номери, відлічувані від початку файлу: або це номера блоків, або номера доріжок у сполученні з номерами блоків на зазначених доріжках. Існує декілька різновидів відносної адресації.

9.2. Способи адресації блоків

При опрацюванні файла прямого доступу важливо правильно вибрати спосіб адресації блоків файла при записі і зчитуванні даних. Існують такі способи адресації блоків: пряма, таблична, рандомізована.

У випадку прямої адресації для записування або зчитування вмісту блока використовується фізична адреса блока на ППДЗ. Фізична адреса блока може задаватися в абсолютній або відносній формах. Абсолютна форма відповідає покажчику номерів циліндра, доріжки, сектори, а відносна форма — покажчику адреси у вигляді номера блока. Передбачається, що всі блоки області ППДЗ, виділеної для файла, пронумеровані із самого початку в порядку їхнього фізичного розташування на ППДЗ. По заданому номеру блока, знаючи місце розташування файла на ППДЗ, операційна система автоматично обчисляє абсолютну фізичну адресу блока [23].

Пряма адресація при прямій організації. Цей спосіб є найбільш простим, розповсюдженим і ефективним. Адреса будь-якого блоку інформації — деяке натуральне число, що представляє порядковий номер блоку відносно початку файлу. Дуже часто в якості такого номеру використовується деяке поле запису або в «чистому вигляді», або після мінімальних перетворень. Наприклад, з поточного значення ознаки віднімається найменше з його можливих значень. Результатом і буде відносний номер блоку у файлі. При прямій адресації досягається взаємно однозначна відповідність між відносним номером блоку і відносною фізичною адресою його у файлі, тобто кожний запис займає у файлі тільки своє місце. При прямій адресації можливий і зворотний перехід — від номера блоку до значення відповідної ознаки.

До основних недоліків прямого методу адресації можна віднести наступні:

- використання тільки записів фіксованої довжини;
- перед створенням файлу необхідна розмітка ділянки магнітного диска, відведеного під файл;
- якщо значення ознаки, що визначає адресу, розташовані в інтервалі нерівномірно, файл буде неефективно використовувати пам'ять через порожні ділянки.

При табличній адресації в головній пам'яті формується індексна таблиця. Вона містить дві колонки. Перша містить ключ блока, а друга — фізичну адресу блока. Для звернення до блока необхідно знайти ключ блока в таблиці. Потім витягається адреса блока, що відповідає знайденому ключу і використовуються для звернення до блока. Недоліки табличної адресації — це необхідність додаткових витрат пам'яті для індексної таблиці та збільшення часу доступу до файла.

Непряма адресація (рандомізована) при прямій організації полягає в наступному. Адреса є результатом перетворень, чинених над ключем запису. Ці перетворення істотно складніші, ніж при прямій адресації, і носять більш загальний характер. Цей процес називають рандомізацією. На відміну від прямої адресації, що ефективна у випадку рівномірного розподілу значень ознак, непряма адресація може бути ефективна при нерівномірному розподілі. Обраний конкретний спосіб рандомізації повинен забезпечити перетворення на адресу на диску будь-якого значення ключа з описуваного діапазону. Однак, не завжди бажаним ефектом рандомізації є одержання синонімів, тобто записів, що мають різні значення ознаки, але в результаті рандомізації однакові адреси, і тому «посягають» на те саме місце в пам'яті. При невеликій кількості синонімів їх можна розташовувати у файлі в послідовних вільних місцях. При більшій кількості синонімів необхідна спеціальна область переповнення.

Якщо проектувальник бази даних взмозі передбачити в пам'яті місце для кожного запису, зумовлене унікальним значенням його первинного ключа, тоді можна побудувати просту функцію перетворення ключа на адресу, що забезпечує запам'ятовування і точну вибірку кожного запису за один прямий доступ до блока. З тією ж ціллю кожній сукупності ключових значень записів, розташованих в одному фізичному блоці, можна привласнити відносний номер блока (відносна фізична адреса). Насправді, дані збе-

рігаються відповідно до розташування ключів, але при цьому їм неявно послідовно привласнюються відносні номера блоків.

Наприклад, у нас є парк автомобілів. Усього є 48 автомобілів, до кожного автомобіля закріплено водій і гараж. Тоді одержуємо запис Автомобіль (номер, водій, гараж). У кожному фізичному блоці зберігаються, наприклад 6 записів, тому потрібно 8 блоків. Тобто функція перетворення ключа на адресу (номера автомобілів від 1 до 48). Відносна адреса блока = [(*номер автомобіля –* –1)/6]+1, де [...] означає отримання цілої частини.

Наприклад, запис про 27-ий автомобіль зберігається в блоці з адресою 5. Така абсолютна відповідність між ключем і відносною адресою блока є головною відмінною рисою прямого доступу. Тому, його використання найбільш доцільне в тих випадках, коли можна управляти значеннями ключів із метою мінімізації витрат пам'яті. У розглянутому прикладі корисне використання пам'яті складає 100 %.

У випадку реальних даних не завжди можливі послідовні значення ключа, тому часто треба будувати більш складні функції перетворення.

Наприклад, якщо підприємство спеціалізується на сталях 18 марок, із номерами Ст0-Ст6 і 10, 15, 20,..., 60, то прямі адреси (рис. 9.1) можуть бути отримані у відповідності з такими правилами:

1. Якщо перші два знаки в номері марки рівні Ст, то потрібно до числової частини додати 1.

2. Якщо є тільки числова частина, то потрібно число розділити на 5 і додати 6.

Рис. 9.1. Одержання адреси з номером 1—18

Із цього прикладу можна зробити висновок, що прямий доступ дуже ефективний щодо часу вибірки і мінімізації використовуваної пам'яті. Будь-яка операція потребує 1-го довільного звертання до блока.

Прямий метод доступу був би поширеніший, якби в усіх додатках існувала можливість керувати значеннями ключів. Проте, некерованість значень ключів є звичайною проблемою. Наприклад, у деякій організації нараховується 1000 студентів, потрібно в якості первинного ключа записів даних про студентів використовувати номер залікової книжки. Серед можливих 10 у 5-ому ступеню ключових значень не можна зазначити явні підмножини, що можна було б ігнорувати, не можна зменшити обсяг необхідної пам'яті. Внаслідок цього, для збереження записів із кожних 100 буде використовуватися тільки одне місце в пам'яті, таке використання дуже неефективне. А якщо використовувати повне ім'я, що є складеним з 25 символів, то усього припустимо 33 у 25 ступеню ключових значень, крім того, неминучі повторювані імена, що не дозволяє забезпечити унікальність адреси блоків, одержуваних у результаті перетворення ключів. Тому використовувати повне ім'я людини в якості первинного ключа краще і не намагатися.

Загальний обсяг пам'яті необхідний для блоків бази даних.

$$V=[\text{кількість значень ключа}/EBF]\times\gamma \text{ (байт)},$$

де EBF — коефіцієнт корисного блокування (кількість записів в однім блоці), γ розмір блока в байтах. Кожне значення ключа відповідає одному запису, що виділяється.

Значення первинного ключа, використовуване в якості вхідного параметра функції перетворення ключа на адресу, можна розглядати як символічну адресу. Значення функції перетворення є відносною фізичною адресою, яку операційна система, в свою чергу, повинна перетворити в абсолютну фізичну адресу. Роздільне виконання показаних перетворень підвищує міру фізичної незалежності даних і, отже, зміна фізичного місця розташування

даних не призводить до зміни програмного забезпечення. Збережені адреси такі, як прямі покажчики, часто не є абсолютними фізичними адресами, а є відносними номерами блоків. Символічні покажчики надають більш високий рівень адресації в порівнянні з прямими покажчиками, хоча той і інший тип покажчиків може бути явно заданий у записах. Важливими аспектами при проектуванні файла прямого доступу є:

1. Типи опрацювання. Довільна вибірка (одержати унікальну), вставити, видалити, модифікувати і фізично послідовна вибірка.

2. Функція перетворення ключа на адресу. Впливає на час опрацювання *CPU*.

3. Кількість можливих значень ключа. Визначає розмір бази даних.

4. Використовуваний рівень адресації даних. Впливає на службове використання *CPU* і міру незалежності даних.

9.3. Основні операції над файлами прямого доступу

Для опрацювання файла прямого доступу необхідно знати яким чином здійснюються основні операції тобто: пошук, вставка, видалення, модифікація.

Пошук. Припустимо є значення V ключа. Визначимо $h(v)$, що дасть нам номер блока, наприклад i. Далі звернемося до блока з номером i. Після цього необхідно досліджувати кожний запис в цьому блоці і з'ясувати, чи містить він запис із значенням ключа V. Позитивний результат означає, що ми знайшли необхідний запис. Якщо ж запис не знайдено і заголовок блока містить покажчик на подальші блоки, то досліджуємо той блок, на який є покажчик.

Процес продовжується доти, поки або не буде знайдено запис із значенням ключа V, або не буде досліджено останній блок у ланцюгу блоків.

Вставка. Застосуємо описану раніше процедуру пошуку. Виявлення запису із заданим значенням ключа V може кваліфікуватися як помилка, оскільки немає сенсу додавати запис, якщо у файлі вже є деякий запис із тим же значенням ключа. Встановивши, що запис зі значенням ключа V не існує, знайдемо перше вільне місце для запису в даному блоці, що має адресу $h(v)$ і помістимо запис у цей фрагмент. Якщо ж вільного фрагмента не існує, то викликаємо одну з описаних раніше процедур опрацювання переповнювання і розміщуємо запис в інший блок.

Видалення. Для видалення запису зі значенням ключа V знову скористаємося процедурою пошуку, що дозволяє знайти цей запис. Далі можна просто зробити місце, що містить даний запис, вільним, встановивши в нуль біти «заповнений/вільний» у його заголовку. При цьому місце для запису стає доступним для повторного використання.

Модифікація. Нехай необхідно модифікувати одне або більше полів запису зі значенням ключа V. Для цього здійснюємо пошук цього запису, як було описано вище. Потім зчитуємо цей запис у робочу зону, модифікуємо поле або поля і записуємо цей запис на те ж місце у файлі прямого доступу.

9.4. Хешування ідентифікатора

Метод хешування ідентифікатора широко поширений як метод доступу, що забезпечує швидку довільну вибірку, вставку, видалення, модифікацію записів [16]. Було сказано, що прямий метод доступу залежить від «керованості» значень ключа і часто потребує невиправдано великих службових витрат пам'яті. Хешування ідентифікатора забезпечує вибірку, вставку, видалення, модифікацію окремих записів по заданому значенню первинного ключа. Платою за цю ефективність є порушення упорядкованості файла і втрата можливості виконувати пакетне

опрацювання або генерацію звітів, що базується на упорядкованості записів по первинному ключі.

Розглянемо деякі поняття, що знадобляться надалі. Ідентифікатор — це атрибут, що унікально визначає кожний примірник деякої сутності предметної області. У контексті бази даних або файла ідентифікатор є типом елемента даних, що унікально ідентифікує примірники конкретного типу записів, тобто первинний ключ. Хешуванням ідентифікатора або просто хешуванням називається метод доступу, що забезпечує пряму адресацію даних шляхом перетворення значення ключа у відносну або абсолютну фізичну адресу. Функцію перетворення ключа часто називають функцією хешування.

Алгоритми пошуку, що використовують хешування, складаються з двох окремих частин. Перший крок — обчислення хеш-функції, що перетворить ключ пошуку на адресу в таблиці. В ідеалі різні ключі повинні були б відображатися на різні адреси, але часто два і більше різних ключів можуть перетворюватися в ту саму адресу в таблиці. Тому друга частина пошуку методом хешування — процес розв’язання конфліктів, що обробляє такі ключі. В одних методах розв’язання конфліктів використовуються зв'язні списки, тому вони знаходять безпосереднє застосування в динамічних ситуаціях, коли заздалегідь важко передбачати кількість ключів пошуку. В інших методах розв’язання конфліктів висока продуктивність пошуку забезпечується для елементів, що зберігаються у фіксованому масиві.

Хешування — наочний приклад компромісу між часом і обсягом пам'яті [3]. Якби на обсяг використовуваної пам'яті обмеження не накладалися, будь-який пошук можна було б виконати за рахунок усього лише одного звертання до пам'яті, просто використовуючи ключ як адресу пам'яті, як це робиться при пошуку з використанням індексування по ключу. Однак, часто цей ідеальний випадок виявляється недосяжним, оскільки необхідний обсяг пам'яті може бути неприйнятним, коли ключі є довгими.

З іншого боку, якби не існувало обмежень на час виконання, можна було б обійтися мінімальним обсягом пам'яті, використовуючи метод послідовного пошуку. Хешування надає спосіб використання як допустимого обсягу пам'яті, так і прийнятного часу з метою досягнення компромісу між цими двома крайніми випадками. Зокрема, можна підтримувати будь-яке обране співвідношення, просто надбудовуючи розмір таблиці, а не переписуючи код або вибираючи інші алгоритми.

Альтернативний хешуванню термін «довільний доступ» підказує, що для перетворення, можливо неоднорідної множини значень ключа в однорідну множину фізичних адрес використовується деякий клас рандомізації. У прямому методі доступу розглядається множина ключових значень розміром *S* байтів, визначених у довіднику розміром *N*, і для кожного *N* у ступені *S* можливих ключових значень потрібна окрема фізична адреса. Якщо в базі даних використовується тільки *nr* різноманітних значень, то щільність ідентифікатора складає *nr/*(*N^S*). У методі хешування є спроба побудувати рівномірне відображення множини *N^S* значень у множину *O*(*nr*) фізичних адрес. У прямому методі доступу має місце відображення 1:1, а в методі хешування *m:*1. При використанні функцій рандомізації можливо перетворення двох або більш значень ключів в ту саму фізичну адресу, так названу власну адресу. Такі ключі називаються синонімами, а випадок перетворення ключа у вже зайняту власну адресу називають колізією. У прямому методі доступу синонімію ключів і колізії уникають ціною великих витрат пам'яті. При виникненні колізії під час початкового завантаження бази даних або при додаванні до неї нового запису в процесі експлуатації в методі хешування доводиться приймати рішення про те, де зберегти новий запис, що містить ключ-синонім [1].

На рис. 9.2 показано найбільш поширений на практиці спосіб організації хешування.

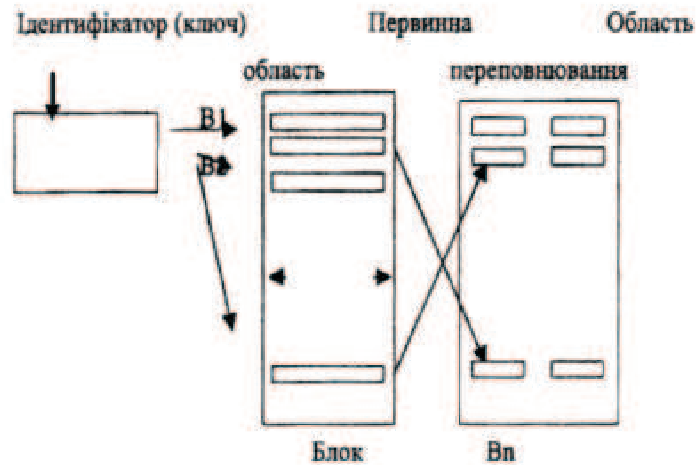


Рис. 9.2. Хешування файлу

Весь адресний простір, безпосередньо доступний функції хешування, ділиться на декілька областей фіксованого розміру, названих блоками. В якості блока можна використовувати сектор, доріжку — будь-яку ділянку пам'яті, що адресується як єдине ціле. В свою чергу блок може складатися з більш дрібних фізичних одиниць даних таких, як доріжка або збережений запис. Найменша складова одиниця блока, використовувана при аналізі продуктивності методу хешування, називається записом.

Процес розв'язання колізій синонімів складається з двох кроків. На першому кроці виконується перегляд блока з метою виявлення в ньому поруч із першим записом вільного простору для нового запису. При наявності вільного простору перегляд припиняється. У протилежному випадку повинен бути здійснений другий крок — опрацювання переповнювання. Умовимося називати область пам'яті, що містить блоки, первинною областю, а частину доступної області, що залишилася в пам'яті — областю переповнювання. Методи опрацювання переповнювання покликані забезпечити ефективне збереження записів переповнювання. Оцінювання різних методів опрацювання переповнювання є од-

ною з головних задач. Головні чинники, аналізовані при проектуванні методу хешування:

1. Тип опрацювання. Тільки довільна вибірка, вставка, видалення, модифікація записів.

2. Розподіл ключових значень. Впливає на розподіл власних адрес і кількість синонімів.

3. Функція хешування. Впливає на розподіл власних адрес і кількість синонімів.

4. Упорядкованість даних при початковому завантаженні. Впливає на загальну продуктивність у випадку використання методу відкритої адресації.

5. Адресний простір (кількість блоків). Впливає на кількість синонімів, дозволяє також здійснити зміну адрес для ключів, що потребує модифікації функції хешування.

6. Розмір блока (кількість записів). Забезпечує гнучкість опрацювання колізій без використання області переповнювання, по своїй дії цей параметр подібний коефіцієнту завантаження.

7. Коефіцієнт завантаження. Впливає на можливість виникнення переповнювання.

8. Метод опрацювання переповнювання. Впливає на час обслуговування вводу-виводу для операцій завантаження, вибірки, вставки, видалення, модифікації записів.

9.5. Функція хешування

Найкращою функцією хешування є функція, що відображає nr значень ключа в точності в nr власних адрес без синонімів. Теоретично існує nr способів такого ідеального відображення, проте, якщо врахувати, що існує nr у ступені nr способів присвоєння nr ключам nr власних адрес, можливість цього ідеального відображення незначна. Практичний досвід підказує нам необхідність направити зусилля на розробку задовольняючих по продуктивності

функцій хешування, що перетворять значення ключів на адреси, рівномірно розподілені по всьому адресному просторі.

В області конструювання ефективних функцій хешування була проведена велика дослідницька робота. В якості одиниць вимірювання ефективності функції хешування можна використовувати такі характеристики, як час *CPU* на виконання перетворення і час обслуговування вводу-виводу для доступу до блока. Як правило, час опрацювання *CPU* буває незначним у порівнянні з часом вводу-виводу, необхідним для доступу до даних; також необхідно підкреслити той факт, що найбільш важливим аспектом ефективності функції хешування є її спроможність рандомізувати значення ключа рівномірно по всьому адресному просторі. У протилежному випадку часто мають місце синоніми (колізії) і спостерігається спад продуктивності через необхідність виконання пошуку в області переповнювання. Доведено, що в силу залежності розподілу адрес від розподілу значень ключа (ідентифікатора) жодний із методів перетворення не можна назвати «найкращим» із погляду рандомізації. Проте, стійка продуктивність і простота методу хешування розподілом і його варіанти дозволяють рекомендувати їх в якості найкращих методів для звичайних застосувань.

Вибір функції хешування — досить важка задача, і вона повинна виконуватися програмістом з урахуванням таких вимог:

1. Функція хешування повинна відображати значення ключів до адрес блоків таким чином, щоб забезпечувався достатньо рівномірний розкид значень ключів по адресах.

2. Функція хешування повинна як можна рідше призводити до ситуацій переповнювання блока, коли два або більше значення ключів відображаються до тієї ж адреси блока у файлі.

3. Функція хешування не повинна зберігати якийсь зв'язок між значеннями ключів.

4. Функція хешування повинна бути простою, щоб не було потрібно багато часу на обчислення адреси блока по заданому значенню ключа.

Приклад 9.1. Для заданої послідовності значень ключа 36, 26, 12, 5, 95 і для адресного простору з 10 блоків побудувати функцію хешування, що не призводить до колізій.

Найпростіша функція хешування розподілом.

	Вихідний ідентифікатор	Перетворення в адресу
1)	36	$36 mod 10+1=7$
	26	$26 mod 10+1=7$
	12	$12 mod 10+1=3$
	5	$5 mod 10+1=6$
	95	$95 mod 10+1=6$

2) (скласти цифри вихідного ідентифікатора) $mod 10=1$

$26(2+6) mod 10+1=1$

$95(9+5) mod 10+1=5$

$36(3+6) mod 10+1=10$

Якщо пронумерувати блоки від 1 до 10 і застосувати найпростішу функцію хешування розподілом f = ідентифікатор mod 10+1, то виявиться, що 36 і 26 є синонімами, обидва значення відображаються в блок 7. Аналогічно 5 і 95 є синонімами і відображаються в блок 6. Завантаження показаних п'ятьох ключових значень призводить принаймні до двох колізій. Проте, якщо замінити функцію перетворення ідентифікатора на функцію «сума цифр mod10+1», то будуть отримані такі пари вигляду «ключове значення (власна адреса)»: 36(10), 26(9), 12(4), 5(6) і 95(5). Даний підхід подає комбінацію методів хешування згортком (додаванням цифр) і розподілом. Очевидно, що кожне значення ключа в цьому випадку відображається в унікальну адресу без колізій. Хоча для деяких послідовностей значень ключа, як наприклад, 36, 72, 27, 54, 63 ця функція може призводити до колізії. У цілому, для великих наборів ключових значень вона забезпечує гарну ефективність рандомізації.

У розглянутому прикладі в якості фізичних адрес використані відносні адреси блоків. У деяких випадках, особливо коли хешування виконується низькорівневими програмами операційної системи, доцільно обчисляти абсолютні адреси.

Приклад 9.2. Потрібно побудувати функцію хешування, що відображає номер страхового поліса за адресою конкретної доріжки на диску. Потрібно рівномірно розподілити значення номера страхового поліса між ними.

Насамперед відзначимо, що номер страхового поліса має вигляд *xxx-xx-xxx*, тому теоретично можливо 10 у ступені 9 різноманітних значень. Припустимо, що до теперішнього моменту задіяно тільки 2×10^9 номерів. Всю наявну повторну пам'ять можна представити у вигляді 32 пристроїв, $32\times555=17760$ циліндрів або $32\times555\times30=532800$ доріжок. Максимальна ємність пам'яті складає $10{,}14\times10^9$ байт тому кожному з $0{,}2\times10^9$ елементів даних можна виділити приблизно 50 байт.

Використаний алгоритм створює ще один різновид хешування згорткою і розподілом, коли обчислюються різноманітні суми і діляться на кожний із трьох модулів: кількість пристроїв, кількість циліндрів і кількість доріжок. На рис. 9.3 наведено приклад застосування показаного алгоритму до ключа 527-45-6783.

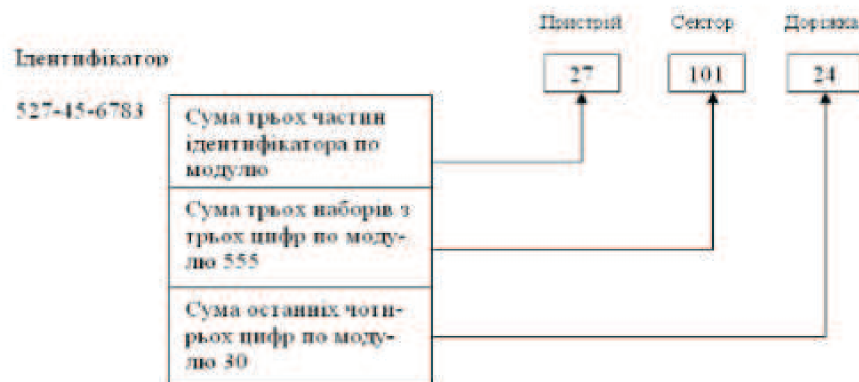


Рис. 9.3. Функція хешування для ключа 527-45-6783

Перша частина адреси дорівнює (527+45+6783)*mod*32=27, друга частина адреси дорівнює (527+456+783)*mod*555=101 і, нарешті, третя частина дорівнює (6+7+8+3)*mod*30=24. У задачах подібного типу, коли неможливо описати адресний простір як континуум за допомогою однієї послідовності чисел, необхідно кожну частину адреси обчисляти окремо.

9.6. Методи опрацювання переповнювання

Переповнювання виникає в тому випадку, коли потрібно вставити новий запис-синонім, а місця для запису у заданому блоці вже заповнені іншими записами-синонімами. Вибір методу опрацювання переповнювання — це одне з найважливіших проектних завдань у довільному методі доступу. Методику аналізу цих способів можна також застосовувати до проектування інших методів доступу.

9.6.1. Метод відкритої адресації

У цьому методі запис переповнювання заноситься до першого невикористаного («відкритого») місця для запису в блоці, адреса якого була отримана за допомогою хеш-функції. Подібний вид пошуку називається лінійним пошуком або лінійним випробуванням. Кількість місць, що підлягає перегляданню, перед тим, як запам'ятати запис, називається зсувом запису. Коли група записів перетвориться як синоніми, то спостерігається значне збільшення зсуву деяких записів у порівнянні з очікуваним усередненим зсувів. Аналогічно при значенні коефіцієнта завантаження, близькому до 1, спостерігається швидке зростання середньої кількості звернень до пам'яті для пошуку. Необхідно зазначити, що в даному методі опрацювання переповнювання всі записи переповнювання розташовуються в первинну область, а не в окрему область переповнювання. Пошук вільного місця здійснюється в межах одного блока.

9.6.2. Метод нелінійного пошуку

Хоча метод відкритої адресації, як правило, передбачає застосування лінійного пошуку, у дійсності, для пошуку відкритого місця в первинній області можна використовувати багато інших видів пошуку. Для запобігання проблем, пов'язаних із кластеризацією і високою щільністю записів при лінійному пошуку, були розроблені різноманітні методи нелінійного доступу до блоків. У методі рехешування для пошуку відкритого місця, із метою вставки нового запису або його наступної вибірки, використовується послідовність функцій перетворення ідентифікатора. У методі квадратичного пошуку, у випадку виникнення колізії, для обчислення послідовних адрес використовується квадратне рівняння вигляду $A(i^2)+Bi+C$ (по модулю рівному потужності адресного простору), де i позначає номер проби, C — початкова власна адреса початкового запису, а A та B — довільні константи.

9.6.3. Метод роздільних ланцюжків

У цьому методі для кожного блока первинної області виділяються локальні блоки для записів переповнювання [23]. Це дозволяє усунути переобтяженість блоків первинної області ланцюжками записів переповнювання за рахунок можливих втрат пам'яті в тих блоках переповнювання, що заповнені не цілком. Якщо потім блок, що містить запис переповнювання, буде потрібен для розміщення запису, власна адреса якого відповідає даному блоку, то для переміщення запису переповнювання необхідні деякі додаткові службові витрати.

Подібних витрат можна уникнути, якщо виділити всі блоки переповнювання в окрему область переповнювання і заборонити використання їх адрес у якості власних адрес записів. Визначення окремої області переповнювання потребує прийняття проектних рішень щодо використання блоків, записів.

На рис. 9.4 на прикладі простої бази даних проілюстровано основні методи опрацювання переповнювання.

Метод обробки переповнювання	Кількість звернень до записів	Кількість звернень до блоків	Кількість звернень до бакетів	Час обслуговування вводу-виводу (1 випадок), мс	Час обслуговування вводу-виводу (2 випадок), мс
1. Відкрита адресація	7	4	2	40	70
2. Нелінійний список	6	3	2	30	90
3. Зчеплені ланцюжки	7	4	2	40	100
4. Роздільні ланцюжки	5	3	2	40	90

Рис. 9.4. Порівняння методів обробки переповнювання

У кожному з цих чотирьох методів записи переповнювання розміщаються по-різному. Наведені на рис. 9.4 характеристики операції вибірки проясняють один важливий момент, при оцінці продуктивності методу хешування, виходячи з кількості звернень до записів, не враховується вплив розмірів блока. Як відомо, оцінка продуктивності у вигляді кількості звернень до блоків ближче до стандартної оцінки продуктивності у вигляді часу обслуговування вводу-виводу, ніж оцінка у вигляді кількості звернень до записів.

Наприклад, у випадку коефіцієнта блокування первинної області, рівного чотирьом, кількість звернень до записів для вставки нового запису у всіх чотирьох методах опрацювання переповнювання залишиться старим, але буде потрібно тільки два звернення до блоків. З іншого боку, взаємозв'язок між кількістю послідовних звернень до блоків і розміром блока для різних методів опрацювання переповнювання різний. Внаслідок чого, неможливо надати порівняльну оцінку продуктивності у вигляді часу обслуговування вводу-виводу тільки на основі кількості звертань до записів, блокам або бакетам. Достовірний порівняльний аналіз може бути виконаний тільки у вигляді часу обслуговування вводу-виводу.

Приклад 9.3. Припустимо, що для бази даних час довільного звернення до блока (*rba*) на диску в середньому складає 40 мс, а час послідовного звернення до блока (*sba*) дорівнює в середньому 10 мс. Потрібно встановити в якому з методів опрацювання переповнювання час обслуговування вводу-виводу буде мінімальний. Обчислити для таких випадків**:**

1. Первинна область і область переповнювання розташовані на однім секторі (циліндрі);

2. Кожний бакет є сектором на довільній доріжці й область переповнювання також розташована на довільному секторі.

Зауважимо, що у випадку використання єдиного сектора довільне звертання еквівалентне за часом послідовному.

Контрольні запитання і завдання для самоперевірки

1. Чи можна сказати, що існує функція хешування, яка буде перетворювати значення ключа в адреси без колізій, якщо значення ключів — це натуральні числа розподілені в порядку зростання без пропусків?

2. Чому метод роздільних ланцюжків, при всіх значеннях коефіцієнта завантаження, дає саму високу продуктивність?

3. Нехай є файл Робітники (Іден. номер; Стать; Посада; Зар.платня; Паспорт. дані; Становище), і надходять запити:

а) видати всіх чоловіків по всіх цехах;

б) видати паспортні дані робітників даного цеху;

в) видати кількість усіх слюсарів, працюючих на підприємстві.

Чи доцільно організовувати показаний файл як файл прямого доступу?

4. Чому при прямій адресації абсолютна форма покажчика адреси блока рідше використовується, ніж відносна?

5. Що відбувається, якщо функція хешування не відображає значення ключів в адреси блоків так, щоб забезпечувався достатньо рівномірний розкид значень ключів по адресах блоків?

6. Нехай у деякій базі даних вироби кодуються послідовністю трьох груп додатних цілих десяткових чисел *AA MM PPPP*, в якій число *AA* означає клас виробу, *MM* — номер складу, і *PPPP* — номер виробу в класі. Яка буде верхня межа коефіцієнта завантаження, якщо інформацію про кожний виріб зберігати у файлі прямого доступу, виділяючи в ньому по однім блоці на виріб, і якщо в якості ключа використовувати приведену вище послідовність із восьми десяткових чисел? При цьому відомо, що існує не більше 50 класів виробів, не більше 10 складів і, кількість виробів будь-якого класу на будь-якому складі не перевищує 2000.

7. Що визначає розмір бази даних:

а) кількість можливих значень ключа;

б) кількість фактичних значень ключа.

8. Що виступає в якості вхідної інформації для функції хешування?

9. За допомогою чого окремі блоки перетворюються в ланцюжки блоків?

ЛІТЕРАТУРА

1. *Далека В. Д., Деревянко А. С., Кравец О. Г., Тамановская Л. Е.* Модели и структуры данных. — Харьков: ХГПУ, 2000. — 241с.

2. *Вирт Н.* Алгоритмы и структуры данных. — Пер. с англ. Д. Б. Подшивалова. — М.: Мир, 1989 — 406 с.

3. *Альфред В. Ахо, Джон Хопкрофт, Джеффри Д. Ульман*. Структуры данных и алгоритмы. — М: Вильямс, 2000. — 384с.

4. *Шилдт Г.* С++. Руководство начинающим, 2-е изд.: пер с англ. — М.: Вильямс, 2005. — 672 с.

5. *Джулиан М. Бакнелл*. Фундаментальные алгоритмы и структуры данных в Delphi: пер. с англ. — СПб: ДиаСофтЮП, 2003. — 560 с.

6. *Романов Е. Л.* Практикум по программированию на С++: [уч. пособие]. — СПб: БХВ-Петербург; Новосибирск: НГТУ, 2004. — 432 с.

7. *Эллиот Б. Коффман*. Turbo Pascal, 5-е изд. — М.: Вильямс, 2005. — 896 с.

8. *Р. Седжвик*. Фундаментальные алгоритмы на С++: пер. с англ. — К.: ДиаСофт, 2001. — 688 с.

9. *Фаронов В. В.* Turbo Pascal 7.0. Учебный курс. — К.: Просвіта, 2008. — 368 с.

10. *Воройский Ф. С.* Информатика. Энциклопедический словарь-справочник: введение в современные информационные и телекоммуникационные технологии в терминах и фактах. — М.: ФИЗМАТЛИТ, 2006. — 768 с.

11. *Окулов С. И.* Программирование в алгоритмах. — СПб.: ДиаСофт, 2007. — 450с.

12. Сборник задач по курсу «Информационные системы и структуры данных»: [учеб. пособие] / С. М. Диго, Г. Н. Клешко, А. И. Мишенин, Е. А. Петров. — М.: Статистика, 1981. — 216 с.

13. *Сибуя М., Ямамото Т.* Алгоритмы обработки данных. — пер. с япон. — М.: Мир, 1986. — 632 с.

14. *Майкл Мейн, Уолтер Савитч*. Структуры данных и другие объекты в C++, 2-е изд. — М.: Вильямс, 2002. — 832с.

15. *Хусаинов Б. С.* Структуры и алгоритмы обработки данных. — М.: ФИЗМАТЛИТ, 2004. — 464 с.

16. *Т. Тиори, Дж. Фрай*. Проектирование структур баз данных. Т. 2. —М.: Мир, 1985. — 645 с.

17. *Гооз Г., Бауэр Ф. Л.* Информатика вводный курс. В 2 ч. 2-е издание, полностью переработанное и расширенное. — М.: Наука, 1992. — 432 с.

18. *Кнут Д.* Искусство программирования. 3-е изд.: Пер. с англ.: [уч. пос]. — М.: Вильямс, 2000 — 720 с.

19. *Й. Лэнгсам, М. Огенстайн, А. Тененбаум*. Структуры данных для персональных ЭВМ. — М.: Мир, 1989. — 214 с.

20. *Моргунов А. Н.* Программирование на языке Паскаль. Основы обработки структур данных. — М.: Диалектика, 2005. — 576 с.

21. *Бабичев А. В.* Распознание и спецификация структур данных. — М.: Наука, 2008. — 192 с.

22. *Теслер Г. С.* Обработка данных с помощью компьютера. — К.: Вища школа, 1991. — 321 с.

23. *Алексеев В. Е., Таланов В. А.* Графы и алгоритмы. Структуры данных. Модели вычислений. — К.: Оствіта, 2000. — 315 с.

ПРЕДМЕТНИЙ ПОКАЖЧИК

Абсолютна форма фізичної
адреси 262
Абстрактна модель зовнішнього
сортування 217
Агрегат даних 242
Адресація 261
абсолютна 261
відносна 261
пряма 261
непряма 261, 263
блоків 263
таблична 262
Адресна функція 83
Адресний простір 260
А-індекси 84
Алгоритм
видалення елемента з
однозв'язного списку 81
вставки елемента в
однозв'язний список 78
вставки запису в
індексно-послідовний
файл 250
Евкліда 138
задачі про вісім ферзів 156
злиття відрізків файлів 222
нестійкий алгоритм 199
обчислення факторіалу 145
опрацювання ключів дерева 113
ханойські вежі 153
пошуку елемента
в двозв'язному списку 82
пошуку елемента в дереві 121
пошуку найкоротшого шляху
між містами 158
рекурсії 142
сортування 197
сортування багатофазним
злиттям 219
сортування вставками 200
сортування вибором 205
сортування методом
«пухирця» 208
стійкий алгоритм 198
Анкерна крапка 252
Антисиметричність 197
Аргумент функції 139

B-дерево 101
B+-дерево 113
*B**-дерево 115

Багатовимірний масив 22
Бінарне дерево 116
глибина 118
майже повне бінарне дерево 118
повне бінарне дерево 118
строго бінарне дерево 117

Відносна форма фізичної
адреси 261
Вектор 9
Верхня межа масиву 11
Вершина стеку 31
Вибірка даних 235

Глибина рекурсії 146

Двовимірний масив 15
Дек 41

Дерево 98
вершина 98
гілка 98
піддерево 98
лист 110
нащадок 99
Дескриптор 10
однозв'язного списку 85
Динамічність файлу 169

Елемент даних 163, 242
нормальний елемент 250
переповнювання 250

Заголовок 58
Запис 18, 180, 242
активність записів 169
ключовий запис 242

Ідентифікатор 268
Інверсія 203
Індекс 242
вищого рівня 244
доріжки 245
первинний індекс 244
повторний індекс 244
циліндра 245
Індексно-послідовний
файл 166, 242

К-індекси 84
Кільцева структура 70
двозв'язна 71
однозв'язна 71
Ключ 163, 242
первинний ключ 181
Конкатенація 109

Ланцюгове збереження 83
Лінійність 198
Лінійний список 61
однозв'язний 63
двозв'язний 67
Ліс дерев 109
Логічний запис 163

Масив 10
Метод доступу 168
відкритої адресації 275
зчеплених ланцюжків 235
нелінійного пошуку 276
розподілення вільної пам'яті 249
розділених ланцюжків 250, 276
хешування індентифікатора 267
Механізм пошуку 168

Нащадки дерев 99
Нижня межа масиву 11

Обхід дерев 122
зворотній обхід 122
прямий обхід 122
поперечний обхід 122
Об'єднання 26
Область даних 242
область індексів 242
первинна 242
переповнювання 242
Обсяг файлу 169
Одновимірний масив 11

Піддерева 94
Покажчик
списку 56
стеку 33
Поле даних 242

Посилання 59
Потік даних 230
  вводу 230
  вводу-виводу 230
  виводу 230
Пошук
  двійковий 185
  послідовний, лінійний 184
  рекурсивний 196
Простір ключів 260

Рандомізація 263
Рекурсія 135
  глибина рекурсії 140
  лінійна 144
  розгалужена рекурсія 146
Рекурсивна структура даних 136
Рекурсивна функція 136
  об'єднання спуску і повернення 150
  рекурсивне повернення 148
  рекурсивний спуск 147
Рекурсивний об'єкт 128
Реорганізація
  індексно-послідовного файлу 257
  послідовного файлу 193
Рефлективність 197

Слот 9
Сортування 194
  багатошляховим злиттям 218
  вибором 196, 204
  внутрішне 195, 197
  вставками 196, 200
  двошляховим злиттям 220
  злиттям 196, 218
  зовнішне 195, 216
  обміном (метод «пухирця») 208
  підрахунком 197
  поділом 196
Список 56
  динамічний список 58
  кільцевий 68
  лінійний 61
  статичний список 57
Стек 31
Структура зберігання 87
Структура 26

Таблиця 20
Транзитивність 197
Трудомісткість алгоритмів 146, 194

Файл 163
  динамічність файлів 169
  індексно-послідовні 164, 240
  корегувань 190
  молодшого покоління 190
  обсяг 163
  послідовні 164
  прямого доступу 164, 260
  старшого покоління 190
Функція рандомізації 176
Хеш-функція 268

Циклічний буфер 42

Черга 41

Навчальне видання

МИХАЛЬОВ Олександр Ілліч
КРАМАРЕНКО Володимир Васильович
ЯЛОВА Катерина Миколаївна
НОВІКОВА Катерина Юріївна

СТРУКТУРИ ДАНИХ ТА АЛГОРИТМИ

Навчальний посібник

Підписано до друку 12.11.10. Формат 60×84 1/16
Папір друк. Друк – різограф. Ум.-друк. арк. 16,62
Обл.-вид. 15,26. Тираж – 300. Зам. № 19/11

Свідоцтво про внесення суб'єкта видавничої справи
до державного реєстру видавництв серія ДК № 1944

Друкарня
51918, Дніпродзержинськ
ДДТУ, вул. Дніпробудівська, 2