Прикарпатський національний університет імені Василя Стефаника

Ткачук В.М.

Навчальний посібник

Алгоритми та структура даних

Івано-Франківськ 2016

УДК 004.41=93(075.8)
ВВК 32
Т48

Рекомендовано до друку вченою радою факультету математики та інформатики
Прикарпатського національного університету імені Василя Стефаника
(протокол №2 від 2 травня 2016р.)

Рецензенти:

Козленко М.І.:- кандидат технічних наук, доцент кафедри інформаційних технологій факультету математиуи та інформатики Прикарпатського національного університету імені Василя Стефаника

Косаревич Р.Я.:- кандидат технічних наук, старший науковий співробітник відділу методів і систем дистанційного зондування ФМІ ім. Г.В.Карпенка НАНУ

Ткачук В.М.

Т48- Алгоритми і структура даних: Навчальний посібник / В.М.Ткачук. - Івано-Франківськ : Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016.-286 с.

У підручнику розглядаються основні питання алгоритмізації та застосування у програмування базових алгоритмів та структур даних. Підручник призначений для студентів, що навчаються на напрямі підготовки "Програмна інженерія"та вивчають сучасні інформаційні технології в рамках дисципліни "Алгоритми та структура даних".

УДК 004.41=93(075.8)
ББК 32)

©Ткачук В.М., 2016
©Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016

Частина I

Структура даних

Розділ 1

Структура даних

1.1 Поняття структури даних

Сучасна ЕОМ задумувалася та будувалась як пристрій, що автоматизує та полегшує складні математичні розрахунки, які вимагають значних затрат часу. На сьогоднішній день на перше місце виступає можливість зберігати значні об'єми даних із можливістю швидкого доступу до них. Можливість же виконувати обчислення при цьому відступила на другий план.

Ми говоримо про дані як про деяке абстрактне представлення реального світу: деякі несуттєві характеристики та властивості об'єктів при цьому ігноруються. Розв'язок будь-якої задачі навіть без використання ЕОМ вимагає певного рівня абстрагування - визначення множини даних, що характеризують реальну ситуацію.

На етапі розв'язку задачі та розробки відповідного програмного забезпечення необхідно визначити структуру та формат використовуваних даних. При цьому під структурою даних розуміють множину елементів даних та взаємозв'язків між ними. Незалежно від суті та складності будь-які дані в оперативній пам'яті комп'ютера задаються у вигляді послідовності двійкових розрядів (бітів), а їх значеннями є відповідні двійкові числа. Бітові послідовності є слабо структурованими та незручними для практичного використання. На практиці, за звичай, використовують більш складну організацію структури даних.

Будь-які достатньо великі програми проектуються шляхом декомпозиції задачі – виділення в задачі деяких структур і їхніх абстракцій. Абстрагування від проблеми передбачає ігнорування ряду деталей для того, щоб звести задачу до більш простої. Задачі абстрагування і наступна декомпозиція типові для процесу створення програм: декомпозиція використовується для поділу програм на компоненти; абстрагування ж передбачає продуманий вибір

компонентів для цієї задачі.

Структурний підхід до даних і алгоритмів дає можливість структурування складної програми. Розробляти сучасну програмну методом „все відразу" неможливо, вона повинна бути представлена в вигляді деякої структури – складових частин і зв'язків між ними. Правильне структурування дає можливість на кожному етапі розробки зосередити увагу розробника на одній оглядовій її частині або доручити реалізацію різних її частин різним виконавцям.

1.1.1 Концепція типу даних

Структури даних і алгоритми є тими матеріалами, з яких будуються програми. Більше того, сам комп'ютер складається з структур даних і алгоритмів. Вбудовані структури даних представлені регістрами й словами пам'яті, де зберігаються бінарні величини. Закладені в конструкцію апаратури алгоритми – це реалізація в електронних логічних схемах жорстких правил, за якими поміщені в пам'ять дані інтерпретуються як команди, що підлягають виконанню. Тому в основі роботи будь-якого комп'ютера лежить вміння оперувати лише з одним видом даних – з окремими бітами. Працює ж з цими даними комп'ютер тільки у відповідності з незмінним набором алгоритмів, які визначаються системою команд центрального процесора.

Задачі, які вирішуються за допомогою комп'ютера, рідко представляються мовою бітів. Як правило, дані мають форму чисел, літер, текстів, символів і більш складних структур типу послідовностей, списків і дерев.

Структура даних відноситься за своєю суттю до „просторових" понять: її можна звести до схеми організації інформації в пам'яті комп'ютера. Алгоритм ж є відповідним процедурним елементом в структурі програми – він служить рецептом розрахунку.

Структури даних, які застосовуються в алгоритмах, можуть бути досить складними. Вибір правильного представлення даних часто служить ключем до вдалого програмування і може в більшій мірі впливати на продуктивність програми, ніж деталі реалізації використовуваного алгоритму. Але, мабуть, ніколи не появиться загальна теорія вибору структур даних, у кожному конкретному випадку потрібно підходити до цього творчо.

Незалежно від змісту і складності будь-які дані в пам'яті комп'ютера представляються послідовністю бінарних розрядів, а їх значеннями є відповідні бінарні числа. Дані, які розглядаються у вигляді послідовності бітів, мають дуже просту організацію, тобто є слабо структурованими. Більш крупні й змістовніші, ніж біт, „будівельні блоки" для організації довільних даних отримуються на основі поняття „структури даних".

Поняття „фізична структура даних" відображає спосіб фізичного представлення даних в пам'яті машини і називається ще структурою зберігання. Розгляд структури даних без врахування її представлення в машинній пам'яті називається абстрактною або логічною структурою. В загальному випадку між логічною і відповідною їй фізичною структурами існує відмінність, міра якої залежить від самої структури і особливостей того середовища, в якому вона повинна бути відображена.

Розгляд структур даних утруднений досить незручним змішуванням аб-

страктних властивостей структур даних і проблемами представлення, які зв'язані з комп'ютером і машинною мовою. Важливо чітко розрізняти ці проблеми за допомогою багаторівневого опису. Будь-яка структура даних може описуватися, таким чином, на трьох різних рівнях:

- **функціональна специфікація** – вказує для деякого класу імен операції, які дозволені з цими іменами, і властивості цих операцій;
- **логічний опис** – задає декомпозицію об'єктів на більш елементарні об'єкти і декомпозицію відповідних операцій на більш елементарні операції;
- **фізичне представлення** – дає метод розміщення в пам'яті комп'ютера тих величин, які складають структуру, і відношення між ними, а також спосіб кодування операцій на мові програмування.

Одній і тій же функціональній специфікації можуть відповідати декілька логічних описів, які в свою чергу можуть реалізовуватися декількома фізичними представленнями. Проте, потрібно мати впевненість, що декомпозиція кожного нового рівня достатньо добре відображає декомпозицію безпосередньо вищого рівня.

Найпростіші структури даних, реалізовані мовою програмування, називають також стандартними типами даних . Багато мов програмування дозволяють на основі стандартних типів будувати типи даних, визначені програмістом (користувачем).

1.1.2 Класифікація структур даних

Із поняттям структури данных тісно пов'язане власне поняття типу даних. Розрізняють фізичну та логічну структуру даних. Фізична структура, на відміну від логічної, визначає спосіб представлення даних в пам'яті комп'ютера.

Розрізняють прості структури данных (типи) та складні (інтегровані). Прості структури не можуть бути поділені на складові частини, більші ніж біти. З точки зору фізичної структури для простого типу даних чітко визначений його розмір та спосіб розміщення в оперативній пам'яті комп'ютера. З точки зору логічної структури прості дані є неподільними одиницями.

Інтегровані структури даних включають у себе інші структури даних — прості чи інтегровані. Між окремими елементами структури можуть існувати явно задані зв'язки (не обов'язково). В залежності від цього розрізняють: незв'язані структури (вектори, матриці, рядки, стеки, черги) та зя'язані структури (звязані списки).

За ознакою можливості зміни розміру розрізняють структури статичні, напівстатичні та динамічні. Під зміною розуміють зміну числа елементів структури або зв'язків між її елементами. Класифікація структур даних за ознакою зміни приведена на рис.1.

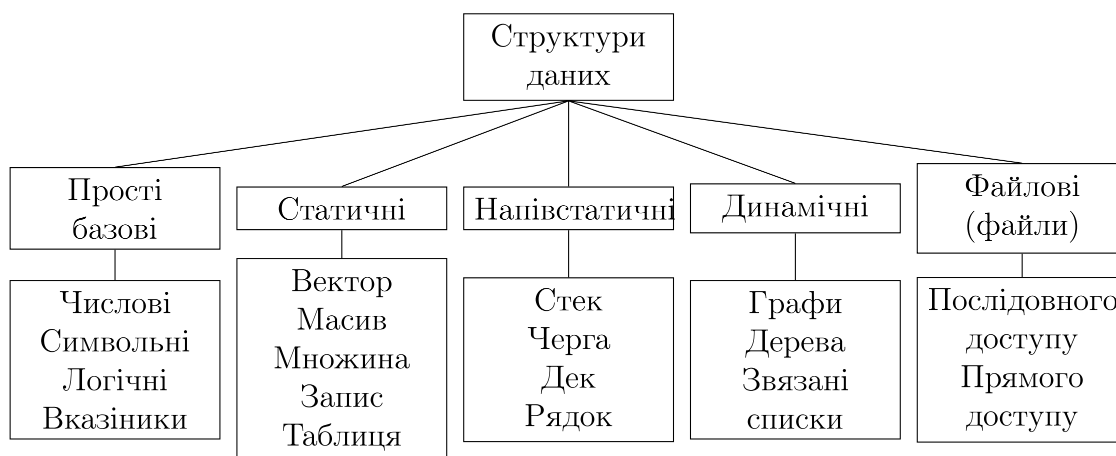


Рис. 1.1: Класифікація структур даних

За ознакою впорядкованості елементів структури можна поділити на лінійні та нелінійні. Прикладом нелінійних структур можуть бути багатозв'язні списки, дерева, графи.

Лінійні структури, в свою чергу, поділяються на структури із послідовним розподілом (вектори, рядки, масиви, стеки, черги) та структури із довільними розподілом (однозв'язні, двохзв'язні списки) за характером розподілу елементів в оперативній пам'яті.

Задання типу даних однозначно визначає:

- розмір пам'яті, який відводиться для збереження даної структури та спосіб її розміщення:
- діапазон допустимих значень, які може приймати структура;
- операції, які можна виконати над даного типу даними.

Прості структури даних є основою для побудови більш складних структур. Їх також називають примітивними чи базовими структурами (типами даних). До них відносять: числові, бітові, логічні, символьні, перераховувані, інтервальні, вказівники. Структура простих типів даних для мови C++ приведена на рис.2 (в інших мовах програмування набір та розмір простих типів можуть відрізнятися від приведеного).

Размер каждого типа данных указан на рисунке в байтах через запятую от названия типа. Как уже было сказано, разные типы данных имеют различный формат представления их в машинной памяти. На рис. 3.3—3.5 приведены примеры форматов числовых типов данных. На рис. 3.4 S обозначает знаковый разряд числа (если S=0, то число положительное, если 5=1— число отрицательное). Формат для представления чисел с плавающей точкой, приведенный на рис. 3.5, а, содержит поля мантиссы, порядка и знаков мантиссы и порядка фиксированной длины. Однако чаще вместо порядка используется характеристика, полученная

1.2 Прості структури даних

1.2.1 Арифметичні типи

Стандартні типи даних часто називають арифметичними, оскільки їх можна використовувати в арифметичних операціях. Для опису основних типів визначено ключові слова: цілий, символьний, логічний, дійсний.

За допомогою цілих чисел можна представити кількість об'єктів, яка є дискретною за своєю природою (тобто кількість об'єктів можна перерахувати). Внутрішнє представлення величин цілого типу – ціле число в бінарному коді. Внутрішнє представлення дійсного типу складається з двох частин – мантиси і порядку.

Результати логічного типу отримуються при порівнянні даних будь-яких типів. Величини логічного типу можуть приймати тільки значення $true$ і $false$. Внутрішня форма представлення значення $false$ – 0 (нуль). Будь-яке інше значення інтерпретується як $true$.

Значенням символьного типу є символи з деякої наперед визначеної множини. В більшості сучасних персональних комп'ютерів цією множиною є ASCII. Ця множина складається з 256 різних символів, які впорядковані певним чином і містить символи великих і малих букв, цифр і інших символів, включаючи спеціальні керуючі символи. Значення символьного типу займає в пам'яті 1 байт. Іншою широко використовуваною множиною для представлення символьних даних є код $Unicode$. В $Unicode$ кожний символ кодується двома байтами.

Над арифметичними типами, як і над всіма іншими, можливі перш за все чотири основних операції: створення, знищення, вибір, поновлення. Специфічні операції над числовими типами – додавання, віднімання, множення і ділення.

Ще одна група операцій над арифметичними типами – операції порівняння: "рівно "не рівно "більше "менше"і т.п. Говорячи про операції порівняння, потрібно звернути увагу на особливість виконання порівнянь на рівність/нерівність дійсних чисел. Оскільки ці числа представляються в пам'яті з деякою точністю, порівняння їх не завжди може бути абсолютно достовірним.

1.2.2 Перерахований тип

При написанні програм часто виникає потреба визначити наперед відому кількість іменованих констант, для яких потрібно, щоб усі вони мали певні, не обов'язково різні значення. Для цього зручно використовувати типом

перерахування *enum*. Всі можливі значення цього типу задаються списком цілочисельних констант:

enum [<ім'я типу>] <список констант> ;

Змінним перерахованого типу можна присвоювати кожне зі значень із списку констант, заданих при оголошенні типу. Імена перерахованих констант повинні бути унікальними (не можуть повторюватися), а самі перераховані константи задаються, ініціалізуються та опрацьовуються як звичайні цілі числа. При відсутності ініціалізації перша константа обнуляється, а кожна наступна отримує значення, на одиницю більше від попередньої. Значення констант можна задати тільки при визначенні перерахованого типу і поміняти його в програмі вже неможливо.

Приклад визначення перерахованого типу, де значення іменованих констант задано явним чином :

enum colors {RED = 5, GREEN = 20, BLACK = 50};

Будь-яке значення цілого типу можна явно привести до перерахованого типу, але при виході за межі його діапазону результат буде невизначеним. Допустимою є також процедура перетворення перерахованого типу до цілого числа. Наприклад:

```
enum colors {RED = 5, GREEN = 20, BLACK = 50};
main()
{
    colors col1;
    colors col2=GREEN;
    int i=20;
    //перетворення цілого числа до перерахованого типу
  col1=colors(i);
    //перетворення  перерахованого типу до цілого числа
    int i1=int(col2);
  .  .  .  .  .  .  .
   }
```

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

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

Нижче наведено приклад створення перерахованого типу $floor$ та можливостей його використання:

```
#include <iostream>
#include "windows.h"
using namespace std;
//визначення перерахованого типу
enum floor {Parking, Market, Boutiques, Spa, Club, Restaurant,Bar
   };


int main()
{
     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);

        int F; //вибір поверху користувачеи
        bool exit = true;//вийти чи продовжити подорож

    cout << " \n \t ________________________________________\n\n";
    cout << "\t Ласкаво просимо до торгового центру !!! \n";
    cout << "\t Відвідайте за допомогою ліфту всі поверхи! \n \n";
    cout << "\t __________________________________________ \n \n";

error:  while(exit)//поки exit рівний true
    {
cout << " \n \t Натисніть кнопку із номером поверху від( 0 до 6):
   ";
        cin >> F;
         switch(F)
        {
        case(Parking):
        cout <<  "\n Ви в паркінгу!!!" << endl;
        break;

        case(Market):
        cout << "\n Ви на першому поверсі!";
        cout << "\n Відвідайте наш магазин \n \n";
        break;
```

```
        case(Boutiques):
        cout << "\n Ви на другому поверсі!";
        cout << "\n Тут знаходиться магазин побутової техніки. \n
   \n";
        break;

        case(Spa):
        cout << "\n Ви на третьому поверсі!";
        cout << "\n Тут знаходиться спасалон-. \n \n";
        break;

        case(Club):
        cout << "\n Ви на четвертому поверсі!";
        cout << "\n Тут знаходиться наш більярдний клуб! \n \n";
        break;

        case(Restaurant):
        cout << "\n Ви на пятому поверсі!";
        cout << "\n Тут ви можете відвідати наш ресторан! \n \n";
        break;

    case(Bar):
        cout << "\n Ви на шостому поверсі!";
cout << "\n Тут ви можете приємно провести час в нашому барі! \n \
   n";
        break;

default: cout << " \n \n \n Помилка!
          В будівлі тільки 6 поверхів! \n \n";
      goto error;
        }

cout << "Якщо хочете вийти на цьому поверсі, то, натисніть 0. \n";
cout << "Якщо хочете продовжити подорож - натисніть 1: ";
        cin >> exit;
    }
return 0;
}
```

```
        Ласкаво просимо до торгового центру !!!
        Відвідайте за допомогою ліфту всі поверхи!


        Натисніть кнопку із номером поверху (від 0 до 6): 2

Ви на другому поверсі!
Тут знаходиться магазин побутової техніки.

Якщо хочете вийти на цьому поверсі, то, натисніть 0.
Якщо хлчете продовжити подорож - натисніть 1: 1

        Натисніть кнопку із номером поверху (від 0 до 6): 5

Ви на п'ятому поверсі!
Тут ви можете відвідати наш ресторан!

Якщо хочете вийти на цьому поверсі, то, натисніть 0.
Якщо хлчете продовжити подорож - натисніть 1: 0
Press any key to continue . . .
```

Рис. 1.2: Результат виконання програми

1.2.3 Вказівники

Використання вказівників дає можливість скоротити сам текст програми та підвищити її ефективність. Для цілого ряду задач вказінникових типів взагалі неможливо обійтися. Як правило вказівники використовують для виконання наступних операцій:

- робота із масивами та символьними змінними
- обробка даних у динамічній пам'яті
- створення динамічних структур
- передача у функцію даних для їх опрацювання

Вказівник (покажчик) - особливий тип даних, значення якого є адреса байта в оперативній пам'яті ЕОМ. В сучасних обчислювальних систем розмір комірки (мінімальної адресованої одиниці) складає один байт). Покажчик не є самостійним типом, він завжди пов'язаний з якимось іншим типом. Хоча фізична структура адреси не залежить від типу й значення даних, які зберігаються за цією адресою, компілятор вважає покажчики на різні типи такими, що мають різний тип. Таким чином, коли йде мова про покажчики, правильно говорити не про єдиний тип даних „покажчик", а про цілу множину типів: „покажчик на ціле", „покажчик на символ".

Вказівники можуть як константами, так і змінними:

- константні вказівники зберігають незмінну адресу певної області оперативної пам'яті. Таким типом вказівників є імена масиві та символьних рядків;
- вказівники-змінні є змінними програми. Їх значення можуть бути змінені в ході виконання програми.

Основними операціями, для яких може бути використано покажчики є: оголошення, присвоєння, отримання адреси, вибір. Перерахованих операцій достатньо для розв'язання більшості задач прикладного програмування. В C/C++ доступні також операції адресної арифметики.

До свого першого використання всі показники повинні бути оголошені згідно наступного синтаксису:

*<базовий тип> *<ім'я вказівника>*

тут: *<базовий тип>* - тип даних програми, адреси яких буде зберігати даний вказівник; $*$ ї ознакою того, що наступна змінна є вказівником; *<ім'я вказівника>* - ідентифікатор змінної. Наприклад:

*int *i, j,k*

*char *th*

*double *f*

Тут: i - адреса ділянки пам'яті, в якій розташоване ціле число; th - адреса даних типу $char$; вказівник f призначений для збереження даних типу $double$. Знак $*$ в оголошенні вказівника обов'язково повинен стояти перед кожною вказівниковою змінною. Змінні без $*$ будуть вважатися звичайними змінними відповідного типу.

Обсяг пам'яті, яку займає вказівник, визначається апаратно-програмною організацією збереження даних в оперативній пам'яті та, як правило, становить 2 або 4 байти. Вказівникові може бути присвоєна тільки адреса змінної відповідного типу: в C++ автоматичного перетворення типів вказівників не передбачено. У C++ визначено дві базові операції для роботи із вказівниками:

& - *адресація*: визначення адреси змінної. Результатом даної операції є адреса змінної, що стоїть перед знаком &. Наприклад, якщо у програмі оголошено цілу змінну k ($int\ k = 14$), тоді щоби її адресу присвоїти вказівнику m ($int\ *m$;), необхідно записати наступну команду:

$$m = \&k$$

В оголошеннях вказівника можна відразу ініціалізувати значення адреси об'єкту, що має відповідний до вказівника тип. Тепер до значення, яке зберігається у змінній k, можна звернутися не тільки за її ім'ям, але і через вказівник. При цьому необхідно застосувати операцію розадресації вказівника $*$.

$*$ -*розадресація*: отримання адреси змінної. Операндом операції $*$ є вказівник (адреса ділянки пам'яті), а результатом - значення об'єкту, адресу якого містить вказівник. Операція є зворотньою до операції взяття адреси &. Результатом розадресації є тип даних, який був заданий базовим при оголошенні вказівника. . Звернення через вказівник мобе бути використане всюди, десинтаксично може бути записаний об'єкт даного типу. Наприклад, якщо в програмі оголошено змінну та вказівник, що зберігає адресу цієї змінної:

```
double z;
double *vz;
*vz=&z;
```

то в результаті присвоєння:

```
*vz=3.123;
```

змінна z прийме значення 3.123.

За старшинством операція взяття адреси та операція розадресації поступаються тільки операції звернення до функції.

Переваги використання вказівників безпосередньо пов'язані із можливістю виконання над ними наступних операцій:

- присвоєння;
- порівняння;
- збільшення/зменшення;
- віднімання.

Перелічені операції ще називають адресною арифметикою. Операції присвоєння, порівняння та віднімання є бінарними: обидва операнди обов'язково повинні бути вказівниками на однакові базові типи. Операції збільшення/зменшення можуть бути як унарними (операції інкременту та декрименту), так і бінарними: збільшення чи зменшення адреси на цілочисленну величину.

Присвоєння вказівників Вказівнику можна присвоїти значення адреси, яка задається іншим вказівником чи обчислюється за дрпомогою виразу, що знаходиться справа від операції присвоєння. Операція присвоєння вимагає, щоби

базовий тип вказівника та виразу, що присвоюється, повинні бути однакові. Наприклад:

```
int *iz;
int *ik;
A=254;
. . . . . . .
iz=&A;
ik=iz;
```

В результаті виконання приведеного фрагменту програми вказівники iz та ik приймуть значення - адресу змінної A. Значення змінної iz буде рівне значенню змінної A .

Додавання та віднімання до вказівника цілого числа. Загальна формула обчислення значення вказівника при додаванні/відніманні цілого числа N має наступний вигляд:

```
нова< адресастара>=< адреса> +/- Nрозмір*< базового типу>;
```

В результаті виконання операції адреса, що зберігається у вказівнику, міняється кратно до розміру даних, на які вказує вказівник. Наприклад, фрагмент програми, приведений нижче, збільшить адресу, що міститься у iz, на $4 * 5 = 20$ байти. Тут враховано, що цілий тип даних займає в оперативній пам'яті 4 байти.

```
int *iz;
.  .  .  .  .  .
iz=iz+5;
```

При необхідності розмір базового типу може бути визначений за допомогою функції

$$sizeof(<\text{базовий тип}>)$$

Різниця однотипних вказівників. При відніманні вказівників обчислюється кількість комірок між адресами, що зберігаються у вказівниках. Загальна форма запису при відніманні однотипних вказівників:

```
кількість< комірок>=
       (< вказівник 1> - вказівник< 2)/ розмір базового типу>;
```

Операції інкременту (++) Адреса, що зберігається у вказівнику, збільшується на розмір базового типу. Наприклад, приведений нижче фрагмент програми збільшує значення вказівника iz на 4 байти (змінна типу int займає в пам'яті 4 байти):

```
    int *iz;
    .   .   .   .   .
    iz++
```

Аналогічно, *операція декрименту (- -)* зменшує вказівник на 1: віднімає розмір у байтах базового типу від адреси, що зберігається у вказівнику.

Порівняння вказівників. При операції порівнюються адреси, що зберігаються у вказівниках. Два вказівники рівні між собою, якщо вони містять однакові адреси: вказують на одну і ту саму комірку пам'яті. Більшим є той вказівник, який містить більшу адресу.Приклад порівняння вказівників:

```
    int *iz;
    int *ik;
    .   .   .   .   .
if (iz= =ik)
     printf("вказівники посилаються на одну і ту саму комірку \n")
   ;
if (iz>ik)
    printf("iz посилається на більшу за адресою комірку \n");
```

Інші операції над вказівниками, крім перерахованих вище, є недопустимими.

1.3 Статичні структури даних

Структуровані дані являють собою інтегрований об'єкт, елементами якого можуть бути дані одного чи декількох базових (примітивних) типів. Значення структурованого типу є сукупністю значень компонент. Якщо всі компоненти відносяться до одного типу, то їх називають базовим типом. Компоненти , в свою чергу, також можуть бути структурованими, що дозволяє будувати цілу ієрархічну структура даних, але при цьому кінцеві компоненти завжди є базовими. Характерною особливістю статичних структур є виділення пам'яті для їх збереження ще на етапі компіляції програми.

В літературі дається наступне означення структури: *структурою даних* називають сукупність елементів даних і множину зв'язків між ними. Структуру можна означити наступним чином:

$$S = (D, R)$$

тут D - множина елементів даних; R - множина зв'язків між елементами даних. Пара, що являє собою елемент даних та елемент відношень, називають елементом структури.

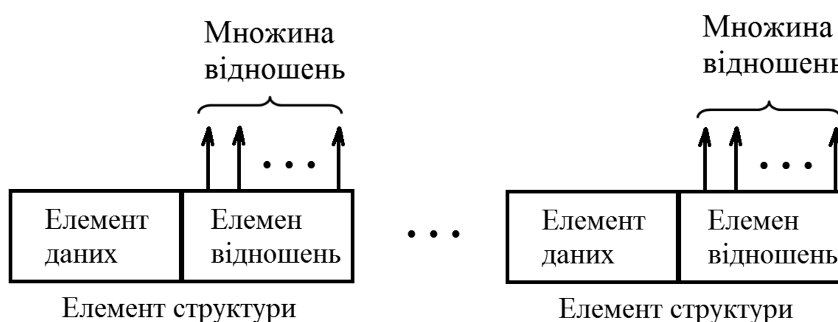


Рис. 1.3: Структура даних:стрілки означають зв'язки між елементами даних

Деякі структури даних визначаються безпосередньо в мовах програмування, наприклад масив, структура чи об'єнання, а деякі створюються самими розробниками в процесі розробки та реалізації алгоритму розв'язку задачі. В багатьох випадках відношення між елементами в явному вигляді можуть бути відсутніми. Наприклад при роботі із елементами масиву множина відношень визначається індексами елементів масиву. Часто разом зі структурою даних

пов'язується і специфічний перелік операцій, які можуть бути виконаними над даними, організованими в таку структуру. Відома формула "Програма = Алгоритми + Структури даних"дуже точно виражає необхідність відповідального ставлення до такого підбору. Тому іноді навіть не обраний алгоритм для обробки масиву даних визначає вибір тої чи іншої структури даних для їх збереження, а навпаки.

При описі структури вона, а в багатьох випадках і її складові частини отримують імена, за допомогою яких і реалізується доступ до елементів структури. Спосіб подання структури даних в пам'яті машини називають *структурою збереження даних.* Наприклад, вектор в структурі даних характеризується фіксованим числом однотипних елементів, доступ до яких здійснюється за допомогою індексу. В структурі збереження для вектора відводиться суцільний ланцюжок комірок пам'яті, достатній для розміщення всіх його елементі. Оскільки кожний елемент займає фіксовану кількість комірок пам'яті, то і для збереження вектора виділяється фіксований об'єм пам'яті. Доступ до елементів вектора відбувається за його адресою в оперативній пам'яті.

Структура збереження із одного боку повинна зберігати відношення між елементами даних, а з другого - забезпечувати найбільш прості та ефективні операції над структурою даних як в цілому, так і для окремих елементів. Такими операціями як правило є доступ до елементів, їх вставка чи видалення, і т.д. Відображення структури даних в структуру збереження реалізується в програмі засобами мови програмування та є найпростішим для скалярних типів.. При цьому дані розміщуються в послідовних комірках пам'яті, займаючи одну чи декілька комірок.

Дані можуть бути збережені або в статичній, або в динамічній пам'яті. Пам'ять називають статичною, якщо вона виділяється на етапі компіляції програми та визначається типом даних, які будуть в ній збережені. Вона виділяється на весь час виконання програми. Якщо пам'ять виділяється по "запиту"під час виконання програми, то вона називається динамічною. В запиті задається об'м необхідної пам'яті, а операційна система виділяє одну суцільну область комірок та повертає адресу початку цієї області. Така пам'ять може бути звільнена в ході виконання програми, якщо потреба в ній відпала.

1.3.1 Стандартна бібліотека шаблонів (STL)

Стандартна бібліотека шаблонів (*Standard Template Library*) - набір узагальнених алгоритмів, контейнерів, засобів доступу до їх вмісту та додаткових функцій в C++. Всі компоненти STL оформлені у вигляді шаблонів, тому можуть використовуватися із довільного типу даними. Для використання шаблону достатньо просто визначити відповідний контейнер а потім тільки викликати функції та алгоритми його обробки.

Робота стандартної бібліотеки шаблонів грунтується на взаємодії трьох базових компонент:

- **контейнер** - сховище набору однотипних даних . В STL представлено заготовки найбільш часто використовуваних структур даних, таких як вектор, список, множина, стек, черга і т. д. Конкретна структура формується користувачем за допомогою стандартних шаблонів контейнерів;
- **алгоритми (algorithms)** - операції над вмістом контейнера. Існують алгоритми для ініціалізації, сортування, пошуку та модифікації вмісту контейнера. Як правило на вхід алгоритм отримує пару ітераторів та для елементів із заданого інтервалу виконує деяку задачу обробки даних;
- **ітератор (iterator)** - об'єкт, що стосовно контейнера відіграє роль вказівника. За їх допомогою можна отримати доступ до вмісту контейнера аналогічно до використання вказівників для доступу до елементів масиву. Власне вони реалізують зв'язок між вмістом контейнера та алгоритмами їх обробки.

Шаблони контейнерів, алгоритми та й взагалі вся бібліотека винесені в окремий простір імен *std*. Для використання ключових слів із простору імен *std* необхідно після всіх підключених файлів використати директиву:

```
using namespace std;
```

Створення та використання контейнерів стандартної бібліотеки шаблонів передбачає виконання наступних кроків:

- створення екземпляру контейнера(його опис);
- додавання елементів в контейнер командою $insert()$;
- визначення кількості елементів в контейнері за допомогою команди $size()$;
- визначення, чи є контейнер пустим (елементи в ньому відсутні) за допомогою команди $empty()$;
- видалення елементів із контейнера викликом команди $erase()$;

- видалення всіх елементів із контейнера командою $clear()$;
- отримання ітератора, що вказує на початок послідовності (перший елемент) за допомогою $begin()$.
- отримання ітератора, що вказує на позицію за кінцем послідовності елементів контейнера командою $end()$;
- виконання циклічних процедур по елементах контейнера за допомогою ітераторів;
- обмін вмісту одного контейнера із вмістом іншого командою $swap()$;
- виконання операцій порівняння вмістів двох контейнерів (рівні, менший чи більший).

За типом всі контейнери бібліотеки STL поділяються на ряд категорій.

Послідовний контейнер - впорядкована множина елементів, в якій кожний елемент займають певну позицію, яка залежить від часу та місця вставки, але на зв'язана із значенням самого елементу. Наприклад, при послідовному додаванні ряду елементів в кінець контейнера вони будуть слідувати в порядку їх занесення в контейнер.

Табл. 1.1: Послідовні контейнери

vector	Вектор - C-подібний одномірний динамічний масив довільного доступу із автоматичною зміною його розміру при додаванні/видаленні елементу. Доступ до елементів здійснюється за їх індексом. Всі елементи вектора повинні бути одного типу та зберігаються послідовно в суміжній області пам'яті.
list	Список - двозв'язний список елементів. Кожний елемент списку займає окремий блок пам'яті та містить посилання на попередній та наступний елементи. Список не підтримує довільний доступ до елементів: для доступу до n - того елементу необхідно пройти по всьому ланцюжку попередніх елементів.
deque	Дек - двостороння черга. Являє собою динамічний масив, що може рости в обох напрямках: дозволяє вставляти/видалення елементів на обох кінцях черги.

Рядкові змінні типу *string* також можна розглядати як контейнер STL, оскільки вона аналогічна вектору із елементами символьного типу. За допомогою ітераторів можна перебрати всі символи рядка. Базові алгоритми STL також є доступними до звичайних статичних чи динамічних масивів.

Асоціативний контейнер - впорядкована множина елементів, в якій позиція елементу залежить від його значення за певним критерієм сортування. Тобто при додаванні елементів в контейнер порядок їх слідування буде залежати тільки від їх значення. Послідовність додавання елементів при цьому значення не має.

Для передачі критерію сортування асоціативні контейнери мають необов'язковий аргумент: по замовчуванню в якості критерія сортування використовується оператор $<$. Критерій сортування також використовується для перевірки рівності: два елементи рівні, якщо кожний із них не більший іншого.

Табл. 1.2: Асоціативні контейнери

Set	Множина - відсортована множина однотипних елементів, що не повторюються. Забезпечує стандартні операції над множинами: об'єднання, перерізу різниці.
Multiset	Мультимножина -множина, аналогічний **Set** із можливість дублювання елементів.
Map	Відображення - впорядкований асоціативний масив пар елементів, що складаються із унікальних ключів та відповідних їм значень. Порядок слідування елементів визначається ключами. Дублювання ключів не допускається.
multimap	Мультивідображення - аналогічний **Map** контейнер із можливістю збереження кількох однакових ключів

Адаптивний контейнер - спеціалізований контейнер, що використовує загальну архітектуру базових контейнерів, ітераторів та алгоритмів.

Табл. 1.3: Адаптивні контейнери

stack	**Стек** - контейнер, що реалізує стек, який працює за принципом: «останнім прийшов — першим пішов» (LIFO, last in, first out). Всі операції в стеку можна проводити тільки з одним елементом, який був введений в стек останнім та знаходиться на верхівці стеку .
queue	**Черга** - контейнер, що реалізує чергу, яка працює за принципом «перший прийшов — першим пішов» (LIFO): раніше буде обслужений той елемент, який потрапив у чергу раніше. Елементи попадають в контейнер із одного кінця, а на іншому «обслуговуються» (видаляються).
priority_queue	Черга із пріоритетом - послідовність читання елементів визначається їх пріоритетом. В пріоритетній черзі наступним елементом є не перший вставлений елемент, а елемент із максимальним пріоритетом. По замовчуванню елементи сортуються по спаданню

Псевдоконтейнер -на відміну від звичайних контейнерів STL кількість елементів фіксується на етапі компіляції, тому не підтримуються операції вставки та видалення елементів. В них також немає можливості роботи із ітераторами

Табл. 1.4: Псевдоконтейнери

bitset	Служить для збереження бітових масок. Подібний на *vector* < *bool* > фіксованого розміру. Розмір фіксується при оголошенні змінної типу *bitset*. Оптимізований за розміром пам'яті та не дозволяє використовувати ітератори.
basic_string	Контейнер для збереження та обробки рядків. Елементи а пам'яті зберігаються підряд єдиним блоком, що дозволяє організувати швидкий доступ до всієї послідовності. Елементами можуть бути прості (фундаментальні) типи даних.
valarray	Шаблон служить для збереження числових масивів та їх оптимізації з метою досягнення максимальної обчислювальної продуктивності. В деякій мірі подібний на *vector*, але в ньому відсутні більшість стандартних для контейнерів операцій.

Ітератор Ітератори (iterators) – це об'єкти, які стосовно контейнера відіграють роль вказівників. Вони дозволяють одержати доступ до вмісту контейнера аналогічно, як вказівники використовуються для доступу до елементів масиву. Ітератори забезпечують зв'язок між структурами даних , алгоритмами та іншим програмним кодом, який ці структури використовує. Їх універсальність дозволяє писати код, що не залежить від типу контейнера.

Контейнер задається двома ітераторами:

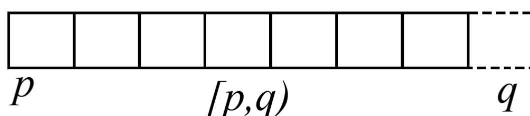


Рис. 1.4: Схематичне представлення ітератора

p вказує на перший елемент контейнера; q - на елемент, що слідує за останнім. Такій підхід зумовлений тим, що при заданні пустого контейнера обидва ітератори p та q не можуть вказувати на один і той-же елемент, бо тоді контейнер містить один елемент. Задання послідовності у вигляді півінтервалу $p, q)$ дозволяє задати порожній контейнер: якщо p та q вказують на один і той-же елемент, то контейнер порожній.

Ітератори підтримують наступні основні операції:

- $*$ - отримання значення елементу в поточній позиції ітератора. Якщо елементи складаються із декількох членів, то для звернення до них безпосередньо через ітератор використовується оператор ->;

- ++ - переміщення ітератора до наступного елементу. Багато ітераторів (залежно від типу контейнера) підтримують переміщення у зворотньому напрямі за допомогою оператора $--$;
- $==$ та $!=$ перевірка співпадіння (наспівпадіння) позицій , заданих двома ітераторами;
- = - присвоєння значення (ітератора).

Кожен із типів контейнерів визначає внутрішню поведінку ітератора, яка визначається структурою даних.

Табл. 1.5: Функції, загальні для всіх ітераторів

c.begin()	Повертає ітератор початку контейнера.
c.end()	Повертає ітератор кінця контейнера.
c.size()	Повертає розмір контейнера.
c.empty()	Перевіряє, чи порожній контейнер.
c.clear()	Очищає контейнер, видаляючи всі елементи.
c=c1	Присвоює вміст контейнера c1 однотипному контейнеру c.
c.assign(b,e)	Заміняє вміст контейнера елементами діапазону [b,e) іншого контейнера.
c.swap(c1)	Міняє місцями вміст двох однотипних контейнерів.
c==c1 (c!=c1)	Порівнює контейнери на рівність (нерівність).

Тут **c**- будь-який з створених типів контейнерів STL.

Табл. 1.6: Функції для контейнерів **vector** і **list**

c.insert(it,t)	Вставляє елемент t у контейнер c перед ітератором it.
c.insert(it,n,t)	Вставляє n елементів t у контейнер c перед ітератором it.
c.insert(it,b,e)	Вставляє діапазон елементів $[b, e)$ у контейнер с перед ітератором it.
c.erase(it)	Видаляє елемент у контейнері c в позиції ітератора it.
c.erase(b,e)	Видаляє діапазон елементів$[b, e)$ у контейнері c.
c.assign(n,t)	Присвоює контейнеру c n копій t.
c.front()	Повертає посилання на перший елемент.
c.back()	Повертає посилання на останній елемент.
c.push_back(t)	Додає t у кінець контейнера c.
c.pop_back()	Видаляє останній елемент.

Більш повний список функцій для контейнерів можна знайти у літературі. Частина функцій буде також розглянута у наступних розділах при детальнішому розгляді тих чи інших структур даних.

Алгоритми Алгоритми орієнтовані на ефективну роботу із набором елементів контейнера. Завдяки ітераторам алгоритми є досить незалежними та можуть бути відокремленими від від поняття "контейнер". Вони просто дозволяють працювати із елементами послідовності, що задається ітераторами. Особливостями роботи багатьох алгоритмів можна управляти.

STL-алгоритми представляють собою набір готових функцій, які можуть бути застосовані до того чи іншого типу контейнерів (і не тільки). Щоби використовувати алгоритми, необхідно в програму підключити заголовочний файл $< algorithm >$:

```
#include <algorithm>
```

З алгоритмами стандартної бібліотеки шаблонів можна ознайомитись, звернувшись до відповідної літератури. Нижче приведено найбільш часто використовувані алгоритми.

Табл. 1.7: Основні алгоритми, що не модифікують контейнери

find ()	Знаходить перше входження значення в послідовність.
count ()	Підраховує кількість входжень значення в послідовність.
search ()	Знаходить перше входження послідовності як підпослідовності.
search_n ()	Знаходить n-те входження значення в послідовність.

Табл. 1.8: Основні алгоритми, що модифікують послідовність елементів

copy()	Копіює послідовність, починаючи з першого елемента.
swap()	Міняє місцями два елементи.
replace()	Заміняє елементи із заданим значенням.
replace_copy()	Копіює послідовність, замінюючи елементи з заданим значенням.
fill()	Заміняє всі елементи даним значенням.
remove()	Видаляє елементи з даним значенням.
reverse()	Міняє порядок елементів на зворотний.
random_shuffle()	Випадковим чином тасує послідовність.
unique()	Видаляє рівні сусідні елементи.

Табл. 1.9: Алгоритми сортування

sort ()	Сортує послідовність.
partial_sort ()	Сортує частину послідовності.
stable_sort ()	Сортує послідовність, зберігаючи порядок слідування рівних елементів.
lower_bound ()	Знаходить перше входження значення у відсортованій послідовності.
upper_bound ()	Знаходить перший елемент, більший ніж задане значення.
binary_search ()	Визначає, є чи даний елемент у відсортованій послідовності.
merge ()	Зливає дві відсортовані послідовності.

Табл. 1.10: Алгоритми роботи із множинами

includes ()	Перевірка на входження.
set_union ()	Об'єднання множин.
set_intersection ()	Перетин множин.
set_difference ()	Різниця множин.

Табл. 1.11: Алгоритми пошуку мінімального та максимального значення

min ()	Менше із двох.
max ()	Більше із двох.
min_element ()	Найменше значення в послідовності.
max_element ()	Найбільше значення в послідовності.

1.3.2 Вектор

Вектор (одномірний масив) — скінчена впорядкована множина простих даних одного типу, які називають елементами вектора. Вектор характеризуєся наступними властивостями:

- сталість структури протягом всього часу існування:
- суміжність елементів;
- неперервність області пам'яті, виділеної відразу для всіх елементів структури.

Впорядковуючи елементи векторів можна пронумерувати їх послідовними цілими числами, які називаються індексами елементів. Кожному елементу вектора ставляться відповідно визначені значення індексу. що дає можливість ідентифікувати відповідні елементи. Під час опису логічної структури вектора зазвичай, йому присвоюється ім'я (ідентифікатор), задається максимальне значення індексу. Приклад оголошення вектора в мові програмування C++:

```
<тип> <імя_масиву> [розмір]=<ініціалізація>
```

Тут: **тип** - означає базовий тип елементів масиву; **розмір** - кількість елементів масиву. **Ініціалізація** - не обовязкова частина опису масиву, що дозволяє присвоїти елементам масиву початкові значення. При оголошенні статичного масиву його розміром може бути тільки числова константа. Наприклад:

```
int  a[100] ;
int  b[10]={3,6,-7,3,9,23,76,1,6};
int  c[ ] ={2,6,9,12};
char st[10];
char st_1[ ]="Рядок";
char st_1[ ]={'Р','я','д','о','к'};
```

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

```
a[i]=5;
```

В СІ існує тісний зв'язок між покажчиками й масивами. Будь-який доступ до елемента масиву, здійснюваний операцією індексування, може бути виконаний за допомогою покажчика. Будь-яка обробка масивів здійснюється шляхом виконання операцій над їх елементами.

Усі елементи одновимірного масиву записуються до розташованих поряд ділянок оперативної пам'яті, і весь масив може розглядатися як одна нерозривна область пам'яті. З математичної точки зору одновимірний масив — це вектор. Наприклад, вектор A, що має п'ять елементів, які у математиці записують у вигляді індексованих змінних a_1, a_2, a_3, a_4, a_5 можна зобразити значеннями цих змінних у сусідніх ділянках оперативної пам'яті:

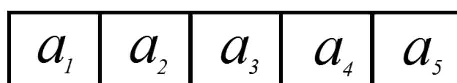


Рис. 1.5: Представлення вектора у пам'яті

Динамічний вектор (масив) - вектор, розмірність якого задається в ході виконання програми, а не на етапі її компіляції. В C++ для роботи з динамічними об'єктами, в тому числі і векторами, використовують спеціальні оператори *new* та *delete*. Вони служать для керування вільною оперативною пам'яттю, яка надається системою для об'єктів, час життя яких напряму задається розробником програми. Оператор *new* виділяє пам'ять під динамічний об'єкт, що створюється в ході виконання програми. Оператор *delete* видаляє об'єкт з пам'яті. Загальний синтаксис запису оператора *new* наступний:

```
new <ім'я_типу>;
new <ім'я_типу> <ініціалізація>;
new <ім'я_типу[вираз]>;
```

В результаті виконання оператору *new* в пам'яті виділяється відповідна кількість комірок, необхідних для збереження заданого типу. Оператор *delete* має наступний формат запису:

```
delete <вираз>;
delete[] <вираз>;
```

Приклад 3. 1. *Дано динамічний вектор, розмір якого задається на етапі виконання програми. Елементи вектора різні та генеруються випадковим чином. Визначити середнє арифметичне елементів масиву, що знаходяться між мінімальним та максимальним елементами включно.*

```
#include <iostream>
#include <windows.h>
```

```
using namespace std;

void GenerationArray(int *vector, int SIZE)
{
    for (int i=0; i<SIZE; i++)
   {
d1:  vector[i] =rand()%100-50;
     for (int j=0;j<i;j++)
  if(vector[j]==vector[i]&&i!=j)
goto d1;
}
}

void ShowArray(int *vector, int SIZE)
{
for (int i=0; i<SIZE; i++)
{     cout << vector[i] << " ";    }
cout << endl;
}

int main ()
{
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
int SIZE;
cout << "Введіть кількість елементів вектора "<<endl;
cin>>SIZE;
//Оголошення вказівника на масив
int *vector;
//Виділення динамічної памяті
vector= new int[SIZE];
GenerationArray(vector,SIZE);
ShowArray(vector,SIZE);
int i_min=0;
int i_max=0;
int v_min=vector[0];
int v_max=vector[0];
    for(int i=0;i<SIZE;i++)
    {
```

```
      if(vector[i]>v_max)
  {
    v_max=vector[i];
    i_max=i;
  }
      if(vector[i]<v_min)
  {
    v_min=vector[i];
    i_min=i;
  }
  }
  int temp;
  if(i_min>i_max)
  {
    temp=i_min;
  i_min=i_max;
  i_max=temp;
  }
  double Sered=0.0;
  for(int i=i_min;i<=i_max; i++)
  Sered=Sered+vector[i];
  Sered=Sered/(i_max-i_min+1);
  cout << "Середнє елементів між максимальним та мінімальним
 включно=  "<<Sered<<endl;
  //Звільнення динамічної памяті
  delete[] vector;
 cout << endl;
 return 0;
  }
```

Прості масиви в C++ можна розглядати як контейнери, а значить вони можуть оброблятися за допомогою алгоритмів стандартної бібліотеки шаблонів. В Інтерфейс ітераторів реалізований як вказівник, а значить звичайні вказівники можуть бути використані в алгоритмах STL.

Приклад 3. 2. *Ілюстрація обробки звичайного масиву з використанням ітераторів та алгоритмів STL.*

```
  #include <algorithm>
```

```
#include <iostream>
#include <windows.h>
#include <numeric>
using namespace std;

void GenerationArray(int *myArray, int SIZE)
{
    for (int i=0; i<SIZE; i++)
    myArray[i] =rand()%100-50;
}

void ShowArray(int *myArray, int SIZE)
{
  cout << endl;
for (int i=0; i<SIZE; i++)
{     cout << myArray[i] << " ";    }
cout << endl;
}

int main()
{
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
int SIZE;
int Sum;
    cout << "Введіть кількість елементів масиву";
cin>> SIZE;

int *myArray=new int[SIZE];
int *myArray_0=new int[SIZE];

GenerationArray(myArray, SIZE);
//Використання числового алгоритму accumulate для знаходження
// суми елементів масиву myArray.
 Sum=accumulate(myArray,myArray+SIZE,0);
cout << "Сума елементів масиву="<<Sum<<endl;
cout << "Масив до сортування:";
    ShowArray(myArray, SIZE);
```

```
    // Сортування перших 4 елементів масиву
        sort(myArray, myArray+4);

    cout << "Масив після сортування :";
    ShowArray(myArray, SIZE);

    cout << "Масив після тосування випадковим чином :";
    random_shuffle(myArray,myArray+SIZE);

    ShowArray(myArray, SIZE);
  int min_myArray;
  int axm_myArray;
  int *presult;

  //Мінімальний елемент масиву
    min_myArray=*min_element(myArray,myArray+SIZE);

    cout << "Мінімальний елемент масиву:";
    cout << min_myArray<<endl;
    presult =find(myArray, myArray+SIZE,min_myArray);
    int pos = presult - myArray;
    cout << "Індекс мінімального елементу:"<<pos<<endl;
    cout << "Максимальний елемент серед перших 6 елементів
 масиву:"<<endl;

    //Максимальний елемент масиву
    axm_myArray=*max_element(myArray,myArray+4);
    cout << axm_myArray<<endl;

    //Видалення елементу зі значенням 19
    if (find(myArray,myArray+SIZE,19)) cout<<"Входить 19"<<endl;
    remove(myArray,myArray+SIZE,-50);
    SIZE=SIZE-1;
    cout << "Вивід модифікованого масиву з використанням ітератора
 :";
  cout << endl;
    copy(myArray, myArray+SIZE, ostream_iterator<int>(cout," "));
    cout << endl;
            int SIZE_0=7;
```

```
  copy(myArray,myArray+SIZE_0, myArray_0);
  cout << "Вивід другого масиву : ";
 cout << endl;
  copy(myArray_0, myArray_0+SIZE_0, ostream_iterator<int>(cout, "
  "));
  cout << endl;
  //Підрахунок кількості елементів, що рівні заданому значенню.
  int number=count(myArray,myArray+SIZE, -16);
  cout << "Кількість елементів, рівних -16,
 дорівнює "<<number<<endl;

  //Приклад обробки текстових змінних
  char str[]="The Standard Template Library (STL) is a software
 library for the C++ programming language that influenced many
 parts of the C++ Standard Library. It provides four components
 called algorithms, containers, functional, and iterators.";
  //Кількість символів
  int n=strlen(str);
  //Пошук букви е
  int numb=count(str,str+n, 'e');
  cout << "Кількість входжень букви е рівна "<< numb << endl;

  delete[] myArray;
  delete[] myArray_0;
  return 0;
  }
```

Примітка: для можливості використання числових алгоритмів (*accumulate*) при обробці масиву даних необхідно обов'язково підключити заголовочний файл *numeric*:

```
    #include <numeric>
```

Результат виконання програми:

Для більш широкої функціональності в стандартній бібліотеці шаблонів передбачено контейнер *vector*. Він фактично являє собою звичайний динамічний масив, для якого визначено ряд додаткових функцій. Для використання контейнера *vector* необхідно підключити заголовочний файл $< vector >$ та для розширення функціональних можливостей контейнера заголовочний файл $< algorithm >$:

```
Введіть кількість елементів масиву:25
Сума елементів масиву=-1
Масив до сортування:
-9 17 -16 -50 19 -26 28 8 12 14 -45 -5 31 -23 11 41 45 -8 -23 -14 41 -46 -48 3 42
Масив після сортування :
-50 -16 -9 17 19 -26 28 8 12 14 -45 -5 31 -23 11 41 45 -8 -23 -14 41 -46 -48 3 42
Масив після тосування випадковим чином :
-46 28 12 42 -5 31 41 3 14 -16 -23 -9 41 -14 -45 -8 -23 8 11 -50 -26 17 45 -48 19
Мінімальний елемент масиву:-50
Індекс мінімального елементу:19
Максимальний елемент серед перших 6 елементів масиву:
42
Входить 19
Вивід модифікованого масиву з використанням ітератора :
-46 28 12 42 -5 31 41 3 14 -16 -23 -9 41 -14 -45 -8 -23 8 11 -26 17 45 -48 19
Вивід другого масиву :
-46 28 12 42 -5 31 41
Кількість елементів, рівних -16, дорівнює 1
Кількість входжень букви е рівна 14
Press any key to continue . . .
```

```
        #include <vector>
        #include <algorithm>
        using namespace std;
```

Контейнера типу *vector* можна оголосити наступним чином:

```
    //порожній контейнер <vector> типу <int>
        vector<int> myVector;
//зарезервовано память під 10 елементів
        myVector.reserve(10);
//додатково 10 елементів ініціалізуються нулями
        vector<int> myVector1(10);
```

Метод *reserve*() використовується для гарантованого виділення відповідного об'єму пам'яті для заданої кількості елементів. Якщо при роботі із вектором зарезервованої пам'яті буде недостатньо (при додаванні нових елементів), то його довжина буде автоматично збільшуватися. При цьому також буде перераховано його розмір. Наприклад:

```
 vector<int> myVector1(10);
//визначення розміру вектора myVector1
int elements = myVector1.size();
cout<<"В myVector1 виділено місце під збереження  "<<elements<<"
   елементів" <<endl;
cout<<endl;
//Ініціалізація 10 елементів масиву випадковими значеннями
for (int i=0;i<10;i++)
```

```
myVector1[i]=rand()%100;
//Додавання ще 10 елементів в кінець масиву
for (int i=10;i<20;i++)
myVector1.insert(myVector1.end(),rand()%100);
//Новий розмір вектора
elements = myVector1.size();
cout<<"Новий розмір вектора= "<< elements<<endl;
cout<<endl;
// Вивід елементів вектора
cout<<"Елементи вектора "<< endl;
cout<<endl;
for (int i=0;i<elements;i++)
cout<<myVector1[i]<<"  ";
cout<<endl;
cout<<endl;
```

```
В myVector1 виділено місце під збереження  10  елементів
Новий розмір вектора= 20
Елементи вектора
41  67  34  0  69  24  78  58  62  64  5  45  81  27  61  91  95  42  27  36
```

Рис. 1.6: Результат виконання фрагменту коду

Крім прямого доступу до елементів вектора за допомогою індексів, елементи вектора також можна отримати за допомогою ітераторів. За звичай ітератори використовуються у парі: один задає текуче значення ітератора, а другий служить для задання точки відліку (початок/кінець контейнера). Функція *begin*() визначає ітератор (вказівник) на перший елемент; *end*() - на неіснуючий елемент, що слідує за останнім елементом контейнера. Для вектора використовується ітератор довільного доступу, який дозволяє обходити контейнер в будь-якому порядку, міняючи при цьому (при необхідності) елементи вектора. Можливі виникнення проблем при роботі із ітератором, якщо міняється розмір вектора.

```
//оголошення ітератора довільного доступу
//the_iterator - імя задається користувачем
        vector<int>::iterator the_iterator;
//початкове значення ітератора - перший елемент контейнера
```

```
        the_iterator = myVector1.begin();
        int summ = 0;
//поки не буде досягнуто кінець вектора (myVector1.end())
    while (the_iterator != myVector1.end())
            {
                summ += *the_iterator;
             ++the_iterator;
            }
    cout << "suma= " << summ << endl;
```

Вектор зберігає порядок його елементів при вставці нових елементів чи на початок, чи в середину вектора: наступні за місцем вставки елементи зміщуються на відповідну кількість позицій. Як наслідок - ітератори елементів після місця вставки стають недійсними. Приклад використання ітераторів для вставки елементів в різні позиції в межах вектора:

```
 cout<<endl << "Вихідний вектор" <<  endl;
elements = myVector1.size();
  for (int i=0;i<elements;i++)
cout<<myVector1[i]<<"  ";
//Вставка елементу на початок вектора
   myVector1.insert(myVector1.begin(), -123);
//Вставка елементу на третю відносно початку позицію
     myVector1.insert(myVector1.begin()+2, -123);
//Вставка кількох елементів (3)
//рівних (4) починаючи із заданої позиції (6)
     myVector1.insert(myVector1.begin()+5, 3, 4);
//Вставка елементу в кінець вектора
// myVector1.insert(myVector1.end(), -32);
elements = myVector1.size();
cout<<endl << "Вектор після вставки елементів" <<  endl;
for (int i=0;i<elements;i++)
cout<<myVector1[i]<<"  ";
```

Аналогічно видалення елементів вектора можна виконати із використанням функції $erase()/$. Додатково до вектора визначено наступні функції:

- $v.clear()$; - очистити вектор v ;

```
Вихідний вектор
41  67  34  0  69  24  78  58  62  64  5  45  81  27  61  91  95  42  27  36
Вектор після вставки елементів
-123  41  -123  67  34  4  4  4  0  69  24  78  58  62  64  5  45  81  27  61  91  95  42  27  36
```

Рис. 1.7: Результат виконання фрагменту коду

- $empty()$ - перевірка, чи є вектор порожнім.
  Наприклад:$if(v.empty())$ $cout <<$ "v порожній."; ;
- операції порівняння векторів.
  Наприклад:$if(v > v1)$ $cout <<$ "v більший ніж $v1$.";
- $swap()$- обмін елементами двох векторів.
  Наприклад $v.swap(v1)$;.
- $max_element$ пошук максимального елементу вектора.
  Наприклад

```
vector<int>::iterator largest = max_element( myvector1.begin()
   , myvector1.end() );
   cout << "Найбільший елемент " << *largest << endl;
cout << "Індекс цього елемента " << largest - myvector1.begin
   () << endl;
```

- $find$ пошук заданого числа (3).
  Наприклад

```
vector<int>::iterator p = find(myvector1.begin(), myvector1.
   end(), 3);
if (p != myvector1.end())cout << "\n Знайдено " << *p << endl;
```

- $find_end$ - пошук підвектора sub у векторі $myvector1$.
  Наприклад:

```
vector<int>::iterator p;
p=find_end(myvector1.begin(),myvector1.end(),sub.begin(),sub.
   end());
if(p!= myvector1.end())cout << "Послідовність знайдено \n";
else cout << " Послідовність не знайдено \n" ;
```

- видалення ідентичних елементів впорядкованого вектора.
  Наприклад

```
myvector1.erase(unique(myvector1.begin(), myvector1.end() ),
  myvector1.end() );
```

До контейнера *vector* визначено також алгоритм сортування елементів за зростанням *sort*, який дозволяє сортувати елементи як в межах цілого вектора, так і вибраного діапазону, заданого за допомогою ітераторів. Сортування за спадання може бети реалізоване за допомогою функції *reverse*.

Приклад №1:

```
//сортування цілого вектора
sort (myvector.begin(), myvector.end());
```

```
Вектор до сортування 32 71 12 45 26 80 53 33
Відсортований вектор 12 26 32 33 45 53 71 80
```

Рис. 1.8: Результат сортування

Приклад №2:

```
//сортування перших чотирьох елементів вектора
sort (myvector.begin(), myvector.begin()+4);
```

```
Вектор до сортування 32 71 12 45 26 80 53 33
Відсортований вектор 12 32 45 71 26 80 53 33
```

Рис. 1.9: Результат сортування

Приклад розв'язку наступної задачі з використанням контейнера *vector*:
Приклад 3. 3. *Виступ спортсмена оцінюється кількома суддями, після чого з усього масиву виставлених оцінок відкидається найбільша та найменша, а для оцінок, що залишилися, визначається середнє арифметичне, яке і*

є заліковими балами студентами. Якщо найвищу чи найнижчу оцінку виставили декілька суддів, то вилучається тільки одна із них. Для кількість суддів $n > 3$ визначити, яка остаточна оцінка піде в залік спортсмену.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "windows.h"
using namespace std;
int main ()
{
  SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
  vector<int> Baly;
    vector<int>::iterator it_Baly;
 cout << "Введіть кількість суддів  "<<endl;
 // кількість суддів
 int N;
 int bal;
 cin>>N;
 Baly.reserve(N);
 // формування вектора оцінок
 for (int i=0;i<N;i++)
 {
   cout << "Введіть оцінку судді №  "<<i+1<<endl;
   cin>>bal;
Baly.insert(Baly.end(),bal);
 }

cout << "Оцінки спортсмена"<<endl;
  for (it_Baly=Baly.begin(); it_Baly!=Baly.end(); ++it_Baly)
    cout << " " << *it_Baly;
  cout << endl;
//Сортування елементів вектора
sort (Baly.begin(), Baly.end());

cout << "Посортовані за зростанням оцінки спортсмена"<<endl;
  for (it_Baly=Baly.begin(); it_Baly!=Baly.end(); ++it_Baly)
    cout << " " << *it_Baly;
```

```
  cout << endl;

// Видалення останньої оцінки
 Baly.erase(Baly.begin()+N-1);
 // Видалення першої оцінки
Baly.erase(Baly.begin());
cout << "Оцінки спортсмена після видалення найбільшої та найменшої"
   <<endl;
  for (it_Baly=Baly.begin(); it_Baly!=Baly.end(); ++it_Baly)
    cout << " " << *it_Baly;
  cout << endl;
//Знаходження середнього арифметичного елементів,
//що залишились
  int elements = Baly.size();
 double summ = 0;
//поки не буде досягнуто кінець вектора (myVector1.end())
  it_Baly=Baly.begin();
    while (it_Baly != Baly.end())
            {
                summ += *it_Baly;
             ++it_Baly;
            }
    cout << endl;
    cout << "Остаточна оцінка в залік спортсмена= " << summ/
   elements << endl;
  cout << endl;
  return 0;
}
```

```
Введіть кількість суддів
7
Введіть оцінку судді №  1
102
Введіть оцінку судді №  2
99
Введіть оцінку судді №  3
103
Введіть оцінку судді №  4
87
Введіть оцінку судді №  5
87
Введіть оцінку судді №  6
104
Введіть оцінку судді №  7
99
Оцінки спортсмена
 102 99 103 87 87 104 99
Посортовані за зростанням оцінки спортсмена
 87 87 99 99 102 103 104
Оцінки спортсмена після видалення найбільшої та найменшої
 87 99 99 102 103

Остаточна оцінка в залік спортсмена= 98
```

Рис. 1.10: Результат сортування

1.3.3 Масив

Масив є похідним типом даних - найбільш широко використовуваних структурою. Логічно масив об'єднує елементи одного типу даних, тобто належить до однорідного типу даних. Формально його можна визначити як впорядковану сукупність елементів деякого типу, які адресуються за допомогою одного або декількох індексів. Кількість елементів масиву є фіксованим, тому об'єм пам'яті під його збереження є незмінним в ході виконання програми.

До базових операцій над матрицями та їх елементами належать:

- ввід та вивід матриць;
- заповнення нової матриці елементами за заданим алгоритмом;
- обчислення узагальнюючих характеристик (сум, добутків і кількості елементів)
- пошук елементів матриці за певним критерієм;
- виконання певних операцій над компонентами матриць (перестановка рядків (стовпців), множення матриць, тощо);
- виконання операцій сортування матриць чи пошуку елементів за заданим критерієм;

- визначення, чи задовольняє матриця або окремі її елементи певній властивості.

За кількістю розмірностей масиву вони поділяються на одновимірні масиви (вектори), двовимірні (матриці) і багатовимірні (трьох, чотирьох і більше). Логічно масив – це така структура даних, яка характеризується:

- фіксованим набором елементів одного і того ж типу;
- кожний елемент має унікальний набір значень індексів;
- кількість індексів визначають розмірність масиву;
- звернення до елемента масиву виконується за його ім'ям та значенням індексів.

Використання індексів з одного боку забезпечує гнучкість доступу до елементів, а з другого - при відсутності відповідного контролю може приводити до виходу за допустимі межі самого масиву.

Фізична структура масиву – це спосіб розміщення елементів масиву в пам'яті комп'ютера. Під елемент масиву виділяється кількість байтів пам'яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам'яті для збереження масиву. Найважливішою операцією фізичного рівня над масивом є доступ до заданого елемента. Як тільки реалізовано доступ до елемента, над ним може бути виконана будь-яка операція, що має сенс для того типу даних, якому відповідає елемент. Перетворення логічної структури масиву у фізичну називається процесом лінеаризації, в ході якого багатомірна логічна структура масиву перетвориться в одновимірну фізичну структуру.

Адресою масиву є адреса першого байта початкового елементу масиву. Індексація масивів в C/C++ обов'язково починається з нуля. Масив будь - якої розмірності у пам'яті зберігається у вигляді вектора: всі елементи знаходяться у суміжних областях підряд, починаючи із нульового елементу. Так, елементи двомірного масиву в C++ розташовуються порядково (останній індекс міняється найшвидше). Приклад розміщення масиву A розміру $m \times n$ елементів:

$$a_{0,0}, a_{0,1}, a_{0,2}, \ldots, a_{0,n-1}, a_{1,0}, a_{1,1}, \ldots, a_{1,n-1}, \ldots, a_{m-1,n-1}$$

Вся інформація, необхідна для управлінням елементами масиву, задається на етапі його опису, який включає ім'я масиву, тип його елементів (визначають довжину елементів у байтах) , діапазон зміни індексів та їх кількість. Опис n - мірного масиву виконується згідно наступного синтаксису:

```
<тип>
<імя_масиву>[розмір 1][розмір 2]...[ розмір n ]=<ініціалізація>
```

Наприклад:

```
int  a[100] [4];
double  c[10][4][7];
// Опис двовимірного масив з одночасною його ініціалізацією.
int  s[2][2]={3,6,-7,3,9,23,76,1,6};

// Двовимірний масив рядкових змінних.
string str3[7][8];
 // Двовимірний масив рядкових змінних з одночасною його
ініціалізацією.
char ch[3][2] = {'W', 'O', 'R', 'L', 'D',' ! '}
```

Якщо масив оголошено без задання розміру, але ініціалізовано списком, то розмір обчислюється автоматично як результат підрахунку елементів списку. Якщо розмір вказується явно, то кількість елементів не повинна перевищувати заданого розміру. Якщо в списку недостатньо елементів, інші елементи ініціалізуються нулями. Першими ініціалізуються елементи з найменшими індексами.

Додаткові фігурні дужки дозволяють ініціалізувати окремі фрагменти багатовимірного масиву, де кожна пара фігурних дужок задає значення, пов'язані з певною розмірністю, але не допустимим є використання порожніх дужок. Наприклад:

```
int Array
[3][3][3]={{{0}},{{10},{2,3},{9}},{{7},{2,4},{6,3,0}}};
```

Наступний приклад ілюструє ініціалізацію цілочисленного масиву у вигляді трикутної матриці розміру 5×5 :

```
int x[5][5]={{6},{5,3},{8,4,6},{7,3,9,4},{3,7,1,2,8}};
```

В результаті буде ініціалізована наступна матриця:

$$\begin{pmatrix} 6 & 0 & 0 & 0 & 0 \\ 5 & 3 & 0 & 0 & 0 \\ 8 & 4 & 6 & 0 & 0 \\ 7 & 3 & 9 & 4 & 0 \\ 3 & 7 & 1 & 2 & 8 \end{pmatrix}$$

Вказівник на багатовимірний масив у C++ – це фактично масиви масивів.

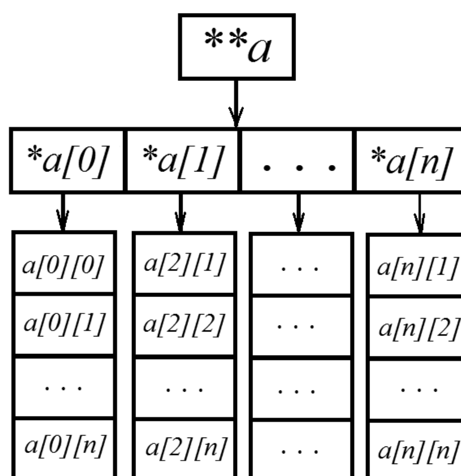


Рис. 1.11: Двомірний масив як масив вказівників

Так, при оголошені, наприклад, двомірного масиву

```
int Mass[3][5]
```

у пам'яті комп'ютера виділяється область для збереження значення змінної $Mass$, яка є вказівником на масив із трьох вказівників. Для цього масиву з трьох вказівників в свою чергу також виділяється область пам'яті. Кожен із цих трьох вказівників містить адресу одномірного масиву із п'яти елементів типу int. Фактично в пам'яті виділяються три області для збереження трьох масивів цілих чисел, кожна з яких складається з п'яти елементів.

Доступ до елементів масиву вказівників здійснюється із зазначенням одного індексного виразу наступним чином: $Mass[2]$ або $*(Mass{+}2)$. Для доступу до елементів двовимірного масиву може бути використано два індексні вирази у формі $Mass[1][2]$ чи еквівалентних їй $*(*(arr+1)+2)$ та $(*(arr+1))[2]$.

Приклад доступу до елементів двомірного масиву $Masa$ розмірності 3×2 з використанням вказівників:

p	$p+1$	$p+2$	$p+3$	$p+4$	$p+5$
$a[0][0]$	$a[0][1]$	$a[1][0]$	$a[1][1]$	$a[2][0]$	$a[2][1]$

Аналогічно можна розглянути, наприклад, тривимірний масив:

```
int Tenz[5][7][8]
```

Його можна інтерпретувати як вказівник на двовимірний масив розміром 7×8 елементів:

```
int (*pb)[7][8];
pb=Tenz;
```

Фактично ім'я $Tenz$ являє собою адресу тривимірного масиву: його першого елемента. Тоді $Tenz[0]$, $Tenz[1]$, ... , $Tenz[4]$ – адреси відповідних двовимірних масивів, $Tenz[i][j]$ – одновимірних. Наприклад:

```
//Присвоєння значення елементу b[1][0][0].
***(pb++)=12;
//Присвоєння значення елементу b[1][0][1].
*(**(pb++)+1)=51;
```

Приклад 3. 4. *Визначити найбільший та найменший елементи матриці та їх індекси. У функції для пошуку максимального елементу використати явну форму оголошення параметру як матриці. У функції для пошуку мінімального елементу параметр Mass оголосити як вказівник на масив N дійсних чисел.*

```
#include <algorithm>
#include <iostream>
#include <windows.h>
#define N 5
using namespace std;

//У функції використано явне оголошення параметра Mass як
 матриці
//Випадкові значення генеруються в межах [-100,100]
    void Init_Mass(double Mass[][N])
{
for(int i=0;i<N; i++)
     for(int j=0;j<N; j++)
     Mass[i][j]=(rand()%2000)/10.0-100.0;
}

//У функції використано явне оголошення параметра Mass як
 матриці
void Show_Mass(double Mass[][N])
```

```
    {
      for(int i=0;i<N; i++)
      {
          cout<<endl;
         for(int j=0;j<N; j++)
         cout<<Mass[i][j]<<"\t";
        }
    cout<<endl;
    }

    //У функції використано явне оголошення параметра Mass як
  матриці
    double Max_Elen(double Mass[][N],int *m_r, int *m_c)
    {
    double max=Mass[0][0];
    *m_r=0;
    *m_c=0;
    for(int i=0; i<N; i++)
        {
          for(int j=0; j<N; j++)
      {
        if(Mass[i][j]>max)
         {
          max=Mass[i][j];
          *m_r=i;
          *m_c=j;
        }
     }
    }
  return max;
    }

    //У функції параметр Mass оголошено як вказівник на масив N
  дійсних чисел
    double Min_Elen(double (*Mass)[N],int *m_r, int *m_c)
    {
    double min=**Mass;
    double *elem=*Mass;
    *m_r=0;
    *m_c=0;
```

```
        for(int i=0; i<N; i++)
         for(int j=0; j<N; j++,elem++)
         if(*elem<min)
         {
          min=*elem;
          *m_r=i;
          *m_c=j;
         }
     return min;
    }

    int main()
    {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
     double Mass[N][N];
    Init_Mass(Mass);
    Show_Mass(Mass);

    double max_elem;
    double min_elem;
    int m_r=0, m_c=0;
    max_elem=Max_Elen(Mass,&m_r, &m_c);
    cout<<"Масимальний елемент=  "
                        <<max_elem<<" Позиція  "<<m_c+1 <<" "<<
   m_r+1<<endl;
    min_elem=Min_Elen(Mass,&m_r, &m_c);
        cout<<"Мінімальний елемент=  "
                         <<min_elem<<" Позиція  "<<m_c+1 <<" "<<
   m_r+1<<endl;
    return 0;
}
```

Якщо розмір масиву не відомий наперед, то для його збереження використовується динамічне виділення пам'яті. Нижче приведено приклад динамічного виділення пам'яті для двовиимірного масиву цілих чисел $Mass$ розміром $n \times m$ та його заповнення випадковими числами:

```
    //Mass - вказівник на масив вказівників.
    int** Mass;
```

```
-95.9    -53.3    -66.6    -50      16.9
72.4     47.8     35.8     -3.8     -53.6
70.5     -85.5    28.1     -17.3    96.1
-50.9    -0.5     94.2     -17.3    43.6
-60.9    -39.6    90.2     -84.7    -70.8
Масимальний елемент=  96.1   Позиція  5 3
Мінімальний елемент=  -95.9   Позиція  1 1
Press any key to continue . . .
```

Рис. 1.12: Результат роботи програми

```
//Виділення пам'яті для масиву вказівників на n рядків.
Mass = new int*[n];
    for(int i = 0; i < n; i++)
{
//Виділення пам'яті для кожного рядка масиву розмірністю m.
Mass[i] = new int[m];
    for (int j=0;j<m;j++)
//Присовоення елементам масиву випадкових значень.
    Mass[i][j]=rand()%100+1;
}
```

Тут при описі динамічного масиву (матриці) $**Mass$ - вказівник на вказівник на int: виділяється пам'ять під масив вказівників на рядки матриці. Наступний цикл - цикл для виділення пам'яті під кожен рядок масиву $a[i] = new\ int[m]$. Кожному елементу масиву вказівників на рядки присвоюється адреса початку ділянки пам'яті, виділеної під рядок із m елементів типу int. Звільнити пам'ять можна командою $delete[]\ a$ згідно наступного прикладу:

```
for(int i = 0; i < n; i++)
//Звільнення пам'яті від кожного рядка.
 delete[] Mass[i];
 //Звільнення пам'яті від масиву вказівників.
delete[] Mass;
```

Як статичний, так і динамічний двовимірний масив у функцію можна передати як подвійну вказівку (**). Проблема в тому, що в C++, власне кажучи, немає двовимірних масивів. У вас двомірний масив є масивом одномірних масивів - тому у функцію, крім самого вказівника на перший елемент, треба передати і всі розміри (кількість рядків та стовпців). Якщо всі одномірні ве-

ктори однакової довжини, то достатньо передавати два розміри, m та n. При необхідності довжина кожного із векторів може бути різною (пилкоподібний масив).

Приклад 3. 5. *Обчислити визначник квадратної матриці, розмір якої задається на етапі виконання програми, а сама матриця генерується випадковим чином.*

Алгоритм обчислення

Для обчислення визначника вихідну матриці зводимо до трикутного вигляду відносно головної діагоналі. Для цього скористаємось властивостями визначника Δ:

- додавання чи віднімання будь яких рядків чи стовпців у масиві не міняє значення визначника;
- при перестановці двох стовпців чи рядків визначник міняє свій знак на протилежний;
- множення всіх елементів будь-якого стовпця на число k рівнозначно множенню визначника на k;
- значення визначника не зміниться, якщо до будь-якого стовпця додати інший стовпець, помножений на число k.

На першому етапі перетворимо визначник Δ так, щоб одержати визначник, в якому всі елементи першого стовпчика, крім першого, дорівнюють 0. Для цього від другого рядка віднімемо перший, помножений на $\frac{a_{21}}{a_{11}}$. Далі від третього рядка віднімемо перший, помножений на $\frac{a_{31}}{a_{11}}$. Продовжуючи цей процес, нарешті від n-того рядка віднімемо $\frac{a_{\text{т}1}}{a_{11}}$. Далі аналогічно перетворюємо визначник так, щоб всі елементи другого стовпчика, починаючи з третього, дорівнювали нулю.Продовжуючи цей процес до останнього рядка n отримаємо визначник:

$$\Delta = \begin{vmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ 0 & x_{22} & x_{23} & \dots & x_{2n} \\ 0 & 0 & x_{33} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & x_{nn} \end{vmatrix} = a_{11} \cdot a_{22} \cdot \ldots \cdot a_{nn}$$

При виконанні операції перетворення матриці до трикутної на кожному із етапів необхідно необхідно відслідковувати можливість виникнення ситуації,

коли $a_{ii} = 0$. В цьому випадку необхідно поміняти місцями сусідні рядки (функція $SIGN$) із порядковими номерами i та $i+1$, пам'ятаючи, що така процедура міняє знак визначника (змінна $sign$)

```
#include <iostream>
#include <windows.h>
using namespace std;
int sign=1;

//Функція генерування вихідної матриці.
void GenerationArray(double **Mas, int SIZE)
{
    for (int i=0; i<SIZE; i++)
     for (int j=0; j<SIZE; j++)
                Mas[i][j]=rand()%10;
    }

//Функція виводу матриці на екран.
void ShowArray(double **Mas, int SIZE)
{
cout.precision(2);
for (int i=0; i<SIZE; i++)
{
  cout<<endl;
    cout << endl;
for (int j=0; j<SIZE; j++)
    cout << Mas[i][j] << "\t\t ";
  }
  cout<<endl;
}

//Функція обміну двох стовпців матриці.
void SIGN(double **Mas, int SIZE, int i)
{
double temp;
for (int j=0; j<SIZE; j++)
{
 temp=Mas[j][i];
 Mas[j][i]=Mas[j][i+1];
 Mas[j][i+1]=temp;
```

```
  }
     sign=-sign;
  }

  //Функція перетворення вихідної матриці до трикутної
  // та обчислення визначника.
  void determinant(double **Mas, int SIZE)
  {
  double koef;
  for(int k=1;k<SIZE; k++)
  {
  for(int j=k;j<SIZE;j++)
  {
if(Mas[k-1][k-1]==0) {SIGN(Mas, SIZE, k-1); cout<<"sign"<<endl;}
  koef=Mas[j][k-1]/Mas[k-1][k-1];
for(int i=0; i<SIZE;i++)
  Mas[j][i]=Mas[j][i]-Mas[k-1][i]*koef;
  }
  }
   // Обчислення визначника трикутної матриці.
  double det=1;
  for (int i=0;i<SIZE;i++)
  det=det*Mas[i][i];
  cout<<endl;
  // Врахування зміни знаку при перестановці стовпців.
  det=det*sign;
  cout<< fixed<<"Визначник=  "<<det<<endl;
  cout << endl;
  }

  int main()
  {
  cout.width(8);
SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
int SIZE;
cout<<"Введіть розмір квадратної матриці "<<endl;
cin>>SIZE;
   // Створення динамічного масиву.
```

```
    double **Mas=new double *[SIZE];
  for (int i=0; i<SIZE; i++)
  Mas[i]=new double [SIZE];
  GenerationArray(Mas, SIZE);

  cout<<"Вихідна згенерована матриця"<<endl;
  ShowArray(Mas, SIZE);
  determinant( Mas, SIZE);

  cout<<"Вихідна матриця, перетворена до трикутної"<<endl;
  ShowArray(Mas, SIZE);
  cout<<endl;
  for(int i = 0; i < SIZE; i++)
  //Звільнення пам'яті від кожного рядка.
  delete[] Mas[i];
  //Звільнення пам'яті від масиву вказівників
  delete Mas;
return 0;
  }
  }
```

Для роботи із цілочисленними матрицями можна модифікувати алгоритм обчислення, врахувавши, що множення всіх елементів будь-якого стовпця на число k рівнозначно множенню визначника на k.

```
Введіть розмір квадратної матриці
4
Вихідна згенерована матриця

1               7               4               0

9               4               8               8

2               4               5               5

1               7               1               1

Визначник=  -748.00

Вихідна матриця, перетворена до трикутної

1.00            7.00            4.00            0.00

0.00            -59.00          -28.00          8.00

0.00            0.00            1.75            3.64

0.00            0.00            0.00            7.26

Press any key to continue . . .
```

Рис. 1.13: Результат роботи програми

1.3.4 Розріджені масиви

На практиці зустрічаються масиви, які через певні причини можуть займати пам'ять не повністю, а частково. Це особливо актуально для масивів великих розмірів, таких що для їхнього збереження в повному об'ємі може бути недостатньо пам'яті комп'ютера. **Розріджений масив** – це масив, більшість елементів якого рівні між собою, так що зберігати достатньо лише невелику кількість значень, відмінних від основного (фонового) значення. Нема єдиного визначення, яка кількість ненульових елементів має бути в матриці, щоб вона була розрідженою. При роботі із розрідженими масивами питання розташування їх в пам'яті реалізуються на логічному рівні з врахуванням типу елементів.

Простим прикладом роботи із розрідженими масивами можуть бути електронні таблиці: попри те, що робочий лист документу має розмір 16384×1048576 елементів (для Excel 2010) зайнятими є не більше 1 відсотка від потенційно доступних комірок.

Розрізняють два типи розріджених масивів:

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

До масивів з математичним описом розташування елементів відносяться масиви, в яких існують функціанальні закономірності в розташуванні елементів із значеннями, відмінними від фонового. Елементи, значення яких є фоновими, називають нульовими. Елементи, значення яких відмінні від фонового називають ненульовими. Фонове значення не обов'язково рівне нулю. Ненульові значення зберігаються, як правило, в одновимірному масиві, а зв'язок між розташуванням у розрідженому масиві і в новому, одновимірному, описується математично за допомогою формули, що перетворює індекси масиву в індекси вектора.

На практиці для роботи з розрідженим масивом розробляються функції:

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

До масивів з випадковим розташуванням елементів відносяться масиви, в яких не існує закономірностей у розташуванні елементів із значеннями, відмінними від фонового. Один з основних способів збереження подібних розріджених матриць полягає у збереженні ненульових елементів в одновимірному масиві із ідентифікацією кожного елемента масиву індексами. Спосіб послідовного розподілу має також ту перевагу, що операції над матрицями можуть бути виконані швидше, ніж це можливо при представленні у вигляді послідовного масиву, особливо якщо розмір матриці великий.

Пам'ять, необхідна для збереження розріджених матриць, складається з двох частин: основної пам'яті, у якій розміщуються числові значення елементів та додаткової пам'яті, де зберігаються індекси та інша інформація, необхідна для формування структури матриць і яка забезпечує доступ до числових значень їх елементів . Способи зберігання і використання даних, що зберігаються в основній і додатковій пам'яті, досить різноманітні і визначаються, головним чином, обраним методом.

Якщо елементами масиву є цілі числі, то найпростіший підхід полягає у створенні двомірного масиву розмірності $n * 3$, де n - кількість ненульових елементів вихідної розрідженої матриці. Перші два елементи кожного рядка містять індекси рядка та стовпця ненульового елементу, а третій - значення самого елементу.

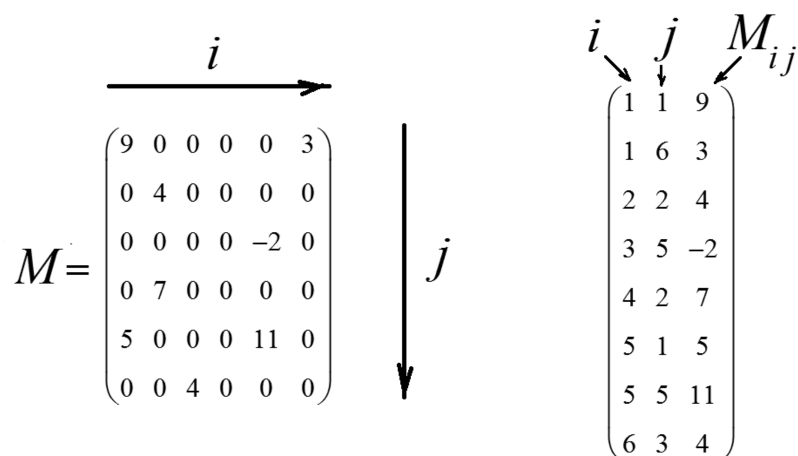


Рис. 1.14: Представлення розрідженої матриці цілих чисел

Якщо ненульові елементи не є цілими числами, то замість двомірного масиву розмірності $n * 3$ використовують два масиви: двомірний цілочисленний масив індексів ненульових елементів та одномірний дійсний масив для збереження значень елементів.. Алгоритми обробки такого типу масивів принци-

пових проблем не викликають. Можна дещо зменшити об'єм необхідної для збереження розрідженого масиву пам'яті, якщо створити три масиви наступної структури:

- $Rows$ - вектор, довжина якого рівна кількості рядків вихідної розрідженої матриці;
- $Elem$ - вектор,кількість елементів якого рівна кількості ненульових елементів вихідної розрідженої матриці;
- $Data$ - вектор зі значеннями ненульових елементів вихідної розрідженої матриці.

Масив $Elem$ містить послідовно індекси стовпців ненульових елементів елементів вихідної розрідженої матриці: спочатку для першого рядка, потім для другого і т. д. Кожний i-тий елемент $Rows$ містить індекси початку i-тої групи елементів в векторі $Elem$. Групою є кількість ненульових елементів розрідженої матриці, що знаходяться у одному рядку.

При такому представленні алгоритми обробки розрідженої матриці дещо ускладнюється, бо кількість ненульових елементів у кожному рядку є різним, сам масив $Elem$ не містить ознаки кінця групи і як наслідок вимагає додаткового використання інформації із вектора $Rows$.

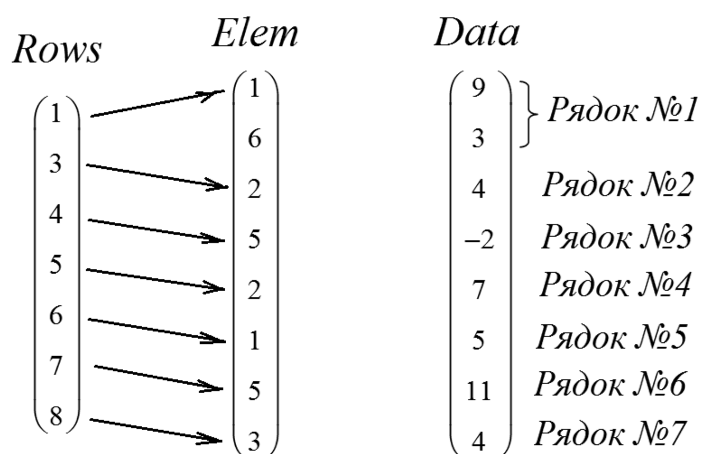


Рис. 1.15: Представлення розрідженої матриці для зменшення необхідного об'єму пам'яті

Приклад 3. 6. *Дана розріджена матриця дійсних чисел, розмір якої задається на етапі виконання програми. Реалізувати її представлення у*

вигляді двох масивів: двомірного масиву індексів ненульових елементів розрідженої матриці та одномірного масиву дійсних чисел, що містить значення ненульових елементів матриці. Для розрідженого масиву реалізувати наступні функції:

- *генерація розрідженої матриці заданого розміру (GenerationArray);*
- *видалення ненульового елементу в заданій позиції(Del_Elem);*
- *вставка ненульового елементу в задану позицію (Ins_Elem);*
- *вивід фрагменту розрідженої матриці для заданого інтервалу(ShowArray);*
- *обчислення суми діагональних елементів розрідженої матриці(Diag_Sum).*

```
 #include <iostream>
 #include <windows.h>
 using namespace std;
 // Функція генерації розрідженого масиву заданого розміру
 void GenerationArray(double *DANI, int *ELEM,
                                   int SIZE, int SIZE_i, int
SIZE_j)
 {
     int i0;
     int j0;
     for (int i=0; i<SIZE; i++)
         {
         ident:  i0=rand()%SIZE_i;
       j0=rand()%SIZE_j;
 //Перевірка, чи є такий елемент.
         for(int j=0;j<SIZE;j++)
      if (i0==*(ELEM+j) && j0==*(ELEM+SIZE+j)) goto ident;
     DANI[i] =rand()%1000/10.0;
 *(ELEM+i)=i0;
 *(ELEM+SIZE+i)=j0;
         }
 }

 // Функція вставки елементу в розріджений масив
 void Ins_Elem(double *DANI, int *ELEM, int &SIZE,
                                int N_i, int N_j,double Dane)
 {
```

```
    //Перевірка, чи є такий елемент.
    for (int k=0;k<SIZE; k++)
  if(*(ELEM+k)==N_i && *(ELEM+k+SIZE)==N_j)
  {
          DANI[k]=Dane;
      SIZE=SIZE-1;
        goto w2;
   }
    DANI[SIZE]=Dane;
    *(ELEM+2*SIZE+1)=N_j;
    for(int k=2*SIZE;k>SIZE;k--)
    *(ELEM+k)=*(ELEM+k-1);
    *(ELEM+SIZE)=N_i;
    w2:SIZE=SIZE+1;
    }

    // Функція обчислення суми діагональних елементів.
    void Diag_Sum(double *DANI, int *ELEM, int SIZE, double &Sum)
    {
        Sum=0;
        for (int k=0;k<SIZE; k++)
         if(*(ELEM+k)==*(ELEM+k+SIZE)) Sum=Sum+DANI[k];
    }

    // Функція видалення елементу в розрідженому масиві
    void Del_Elem(double *DANI, int *ELEM, int &SIZE, int N_i, int
    N_j)
    {
        int N_0;
        for (int k=0;k<SIZE; k++)
     if(*(ELEM+k)==N_i && *(ELEM+k+SIZE)==N_j)
    {
      N_0=k;
      for(int i=N_0;i<SIZE;i++)
    {
      DANI[i]=DANI[i+1];
      *(ELEM+i)=*(ELEM+i+1);
      *(ELEM+i+SIZE)=*(ELEM+i+1+SIZE);
    }
```

```
  for(int i=SIZE+-1;i<2*SIZE;i++)
{
  *(ELEM+i)=*(ELEM+i+1);
  }
  SIZE=SIZE-1;
  }
  }

  // Функція виводу фрагменту розрідженого масиву
  void ShowArray(double *DANI, int *ELEM, int SIZE,
                        int i_min, int i_max, int j_min, int
 j_max)
  {
  double DANI_0;
  for (int i=i_min; i<=i_max; i++)
  {
  cout << endl;
  for (int j=j_min; j<=j_max; j++)
   {
  DANI_0=0;
  for (int k=0;k<SIZE; k++)
  if(*(ELEM+k)==i && *(ELEM+k+SIZE)==j)
   {
  DANI_0=*(DANI+k);
    break;
}
  cout<<"\t"<<DANI_0;
}
}
   cout << endl;
  }
  int main ()
  {
  SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
  int SIZE_i;
  int SIZE_j;
  int SIZE;
  cout << "Введіть кількість рядків розрідженої матриці
```

```
"<<endl;
cin>> SIZE_i;
cout << " Введіть кількість стовпців розрідженої
матриці "<<endl;
cin>> SIZE_j;
cout << "Введіть кількість ненульових елементів розрідженої
матриці "<<endl;
cin>> SIZE;
//Оголошення вказівника на масив даних.
double *DATA;
//Виділення динамічної памяті під масив індексів.
DATA= new double[SIZE+10];
int *ELEM;
ELEM = new int[SIZE*2+10];
GenerationArray(DATA,ELEM,SIZE,SIZE_i,SIZE_j);
int i_min, i_max;
int j_min, j_max;
novyi:cout << "Введіть i_min та i_max "<<endl;
cin>>i_min>>i_max;
cout << " Введіть j_min та j_max "<<endl;
cin>>j_min>>j_max;
ShowArray(DATA,ELEM,SIZE,i_min,i_max,j_min,j_max);
int N_i=0;
int N_j=0;
cout << " Введіть позицію для видалення "<<endl;
cin>>N_i>>N_j;
Del_Elem(DATA, ELEM, SIZE, N_i, N_j);
cout << "Введіть позицію для вставки "<<endl;
cin>>N_i>>N_j;
double Dane;
cout << "Введіть значення для вставки "<<endl;
cin>>Dane;
Ins_Elem(DATA, ELEM, SIZE, N_i, N_j,Dane);
double Sum;
Diag_Sum(DATA, ELEM, SIZE, Sum);
cout << " Сума діагональних елементів= "<<Sum<<endl;
cout << endl;
goto novyi;
return 0;
```

```
    }
```

```
  Введіть кількість рядків розрідженої матриці
1000
  Введіть кількість стовпців розрідженої матриці
1000
  Введіть кількість ненульових елементів розрідженої матриці
2000
  Введіть i_min та i_max
510 520
  Введіть j_min та j_max
600 610

        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       88.8    0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
  Введіть позицію для видалення
```

Рис. 1.16: Генерація розрідженої матриці та вивід її фрагменту

```
  Введіть позицію для вставки
516 605
  Введіть значення для вставки
13.34
  Сума діагональних елементів= 65.5

  Введіть i_min та i_max
510 520
  Введіть j_min та j_max
600 610

        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       13.34   0       0       88.8    0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0       0
  Введіть позицію для видалення
```

Рис. 1.17: Встака елементу в розріджену матрицю та обчислення суми діагональних елементів

Розглянуті вище способи представлення розрідженої матриці можна використовувати у тому випадку, коли кількість та розміщення ненульових елементів залишається незмінним. Інакше більш зручним є представлення розрідженої матриці з використанням двонаправленого списків із динамічним

виділенням пам'яті для збереження елементів списку. В цьому випадку для реалізації розрідженого масиву необхідно створити структуру, що містить наступні поля:

- значення ненульового елементу розрідженої матриці;
- логічна позиція (індекси) ненульового елементу в масиві ;
- вказівники на наступний та попередній елементи списку.

<table>
<tr><td colspan="3">Вказівник на наступний елемент в списку рядків</td><td colspan="3">Вказівник на наступний елемент в списку стовпців</td></tr>
<tr><td colspan="2">Номер рядка</td><td colspan="2">Номер стовпця</td><td colspan="2">Значення елементу</td></tr>
</table>

Доступ до масиву реалізується шляхом переходу по вказівниках. Наприклад, для реалізації розрідженого масиву цілих чисел можна створити наступну структуру:

```
  struct Element
      {
          int Log[2];    //Логічна позиція (індекси)  елементу
в масиві.
          int d;           //Поле даних.
          Element* next;  //Вказівник на наступний елемент.
          Element* prev;  //Вказівник на попередній  елемент.
      }
          ;
```

Принципова перевага методу реалізації розрідженого масиву за допомогою двозв'язного списку полягає у простоті реалізації та ефективному використанні пам'яті — місце виділяється тільки для ненульових елементів. Недолік є необхідність застосовувати лінійний пошук для доступу до відповідних ненільових елементів. Процедура збереження також використовує лінійний пошук, щоби знайти місце вставки нового елементу.

Ці проблеми можна обійти, побудувавши розріджений масив на основі двійкового дерева, реалізованого у вигляді двозв'язаного список. Його основна перевага полягає в можливості швидкого пошуку, що дозволяє ефективно виконувати операції вставки та доступу до елементів розрідженого масиву. Щоб використати двійкове дерево для реалізації розрідженого масиву, необхідно створити наступну структуру:

```
   struct Element
       {
             int Log[2];   //Логічна позиція (індекси)  елементу в
масиві.
             int d;                            //Поле даних.
             struct Element *left;        // Вказівник на ліве
піддерево.
             struct Element *right;      // Вказівник на праве
піддерево.
               }
             ;
```

Більш детально робота із такими двозв'язними списками та двійковими деревами буде розглянута пізніше при розгляді відповідних динамічних структур даних.

1.3.5 Множини

Множина - структура, яка є набором даних одного і того ж типу, що не повторюються (кожен елемент множини є унікальним). Порядок слідування елементів множини на має принципового значення Набір елементів асоціюється із ім'ям множини, і з нею можна порівнювати інші величини, щоби визначити їх приналежність до цієї множини. При необхідності множина може і не містити жодного елементу - бути порожньою (нульовою).

Множина є підмножиною іншої множини, якщо в цій другій множині можна знайти усі елементи, які є в першій множині. Відповідно, множина вважається надмножиною іншої множини, якщо вона містить усі елементи цієї другої множини. Кожен окремий елемент є членом множини, якщо він входить до складу елементів множини. Число, що дорівнює кількості елементів множини, називається її потужністю. Від масиву множина відрізняється відсутністю індексації її елементів, а від запису — тим, що елементи множини є однотипними і не позначаються ідентифікаторами. Дві множини вважаються еквівалентними тоді і тільки тоді, коли всі їх елементи однакові, причому порядок проходження елементів в множині байдужий. Якщо всі елементи однієї множини входять також і в інше, говорять про включення першої множини в друге. Порожня множина включається в будь-яке інше.

Тип множини використовують у задачах, у яких має значення тільки належність або неналежність елемента деякій множині. В основному реалізація множин у програмуванні повторює відомі операції та відношення для множин у математиці.

Над множинами визначені наступні специфічні операції:

1. Об'єднання множин.
2. Перетин множин.
3. Різниця множин.
4. Симетрична різниця.
5. Перевірка на входження елемента в множину.

Досить часто використовують більш складну ніж проста множина структуру: **навантажена множина**. У ній кожен елемент міститься разом з додатковою інформацією, яку називають *навантаженням елемента.* При додаванні елемента в множину необхідно також задавати навантаження, яке він несе. Така структура даних у стандартній бібліотеці шаблонів реалізована у вигляді карти (Map). Прикладом навантаженої множини може бути звичайний словник: множини - іноземне слово; навантаження - переклад слова (певний текст).

Тип даних "множина"не передбачений як стандартний тип мови програмування С/С++, але реалізується засобами визначення типів користувача.

Для простоти розглянемо алгоритм реалізації звичайної множини. Реалізація навантаженої множини аналогічна, тільки паралельно з елементами необхідно додатково зберігати навантаження елементів. У звичайній реалізації множини її елементи зберігаються у вигляді звичайного одномірного масиву. Додаткова змінна містить поточну кількість елементів.

При додаванні елемента x до множини необхідно спочатку визначити, чи міститься він у множині: якщо так, то множина не змінюється. Для цього використовується функція пошуку елемента. Та ж функція пошуку використовується і при видаленні елемента із множини. При додаванні елемента він дописується в комірку масиву з індексом, рівним числу елементів та число елементів збільшується на одиницю. Для видалення елемента достатньо останній елемент множини переписати на місце, що видаляється та зменшити кількість елементів на одиницю.

У звичайній реалізації елементи множини зберігаються в масиві в довільному порядку. Це означає, що при пошуку елемента x доведеться послідовно перебрати всі елементи, поки не буде знайдемо x (або його немає). Тому така реалізація доцільна тільки для невеликих множин. Прискорити швидкість обробки елементів множини можна впорядкувавши їх при розміщенні в одномірному масиві за певним ключем, наприклад за зростанням. При цьому дещо ускладняться алгоритми вставки та видалення елементів.

Приклад 3. 7. *Реалізувати впорядковану по зростанню множину цілих чисел. Елементи множини зберігаються у звичайному одномірному масиві Для множини реалізувати наступні функції:*

- *генерація впорядкованої за зростанням множини (GenerationSet) ;*
- *вивід множини на екран(ShowSet);*
- *видалення елементу множини із перевіркою, чи такий елемент у множині присутній(DelSet);*
- *вставка елементу у множину із перевіркою, чи такий елемент у множині вже присутній (InsertSet);*
- *виконання операції об'єднання двох множин(UnionSet). Про суть операції об'єднання дивись рис. 1.20 .*

```
#include <iostream>
#include <windows.h>
```

```
  using namespace std;
  //Максимальна кількість елементів множини
  int max_SIZE=1000;

  //Функція сортування елементів множини.
  void SortSet(int Set[],int SIZE)
  {
   int i,j,k,p;
      for (i=0; i<SIZE-1; i++)
      {
      // Ознака процесу обміну.
      p=0;
  for (j=SIZE-1; j>i; j--)
  if (Set[j]<Set[j-1])
  {
k=Set[j];
Set[j]=Set[j-1];
Set[j-1]=k;
p=1; // Пройшов процес обміну.
  }
  // Якщо обміну не було, то сортування завершено
  if (p==0) return;
  }
  }

  //Функція генерування вихідної матриці.
  void GenerationSet(int *Set, int SIZE)
  {
  int i=0;
  Set[i]=2000-rand()%1000;
  int temp;
      while(i<SIZE)
      {
       temp=1000-rand()%2000;
       for (int j=0; j<SIZE; j++)
         if (temp==Set[j]) continue;
         i++;
       Set[i]=temp;
      }
```

```
    SortSet( Set,SIZE);
  }

    //Функція виводу матриці на екран.
    void ShowSet(int *Set, int SIZE)
    {
  cout<<endl;
  for (int i=0; i<SIZE; i++)
        cout << Set[i] << "\t";
   cout<<endl;
 }

    //Функція вcтаки елементу у відсортовану множину.
    int InsertSet(int *Set, int x, int &SIZE)
    {
  int temp;
  //Елемент попадає на першу позицію у відсортованій множині
  if(x<Set[0])
    {
     for(int i=SIZE;i>0;i--)
     Set[i]=Set[i-1];
    Set[0]=x;
    SIZE++;
    return 1;
    }
  if(x>Set[SIZE-1])
    {
    Set[SIZE]=x;
    SIZE++;
    return 1;
    }
  if(Set[SIZE-1]==x) return 0;
  for (int i=0; i<SIZE-1; i++)
  {
    if(Set[i]==x) return 0;
    if(Set[i]<x && Set[i+1]>x)
    {
    temp=i;
    break;
```

```
    }
  }
     for(int i=SIZE;i>temp;i--)
     Set[i]=Set[i-1];
    Set[temp+1]=x;
    SIZE++;
     return 1;
   }

    //Функція видалення елементу із відсортованої множини.
    int DelSet(int *Set, int x, int &SIZE)
    {
    int temp=-1;
  for (int i=0; i<SIZE; i++)
     {
      if(Set[i]==x)
     {
      temp=i;
      break;
    }
     }
    if(temp==-1) return 0;
  for(int i=temp;i<SIZE;i++)
     Set[i]=Set[i+1];
  SIZE--;
  return 1;
  }

    //Функція обєднання двох множин.
    void  UnionSet(int *Set, int SIZE, int *Set_1, int SIZE_1, int
    *Set_0, int &SIZE_0)
    {
  SIZE_0=SIZE;
  for (int i=0; i<SIZE; i++)
    Set_0[i]=Set[i];
  for (int i=0; i<SIZE_1; i++)
     InsertSet(Set_0,Set_1[i],SIZE_0);
  }
```

```
  int main()
  {
int SIZE=0;
int SIZE_1=0;
int SIZE_0=0;
int x;
SetConsoleCP(1251);
 SetConsoleOutputCP(1251);
 int *Set=new int[max_SIZE];
 int *Set_1=new int[max_SIZE];
 int *Set_0=new int[max_SIZE];
 cout<<"Введіть розмір множини, яка має бути
згенерована"<<endl;
 cin>>SIZE;
GenerationSet(Set, SIZE);
ShowSet(Set, SIZE);
 cout<<"Введіть значення елементу для вставки у множину"<<endl;
 cin>>x;
 if (InsertSet(Set,x,SIZE)==0) cout<<"Такий елемент у множині
вже присутній"<<endl;
  ShowSet(Set, SIZE);
 cout<<"Введіть значення елементу для видалення із
множину"<<endl;
 cin>>x;
if(DelSet(Set,x,SIZE)==0)cout<<"Такий елемент у множині
відсутній"<<endl;
ShowSet(Set, SIZE);
  cout<<"Введіть розмір другої множини, яка має бути
згенерована"<<endl;
 cin>>SIZE_1;
GenerationSet(Set_1, SIZE_1);
ShowSet(Set_1, SIZE_1);
  cout<<"Результат обєднання двох множин"<<endl;
  UnionSet(Set, SIZE, Set_1, SIZE_1, Set_0, SIZE_0);
  ShowSet(Set_0, SIZE_0);
return 0;
  }
```

У випадку, якщо простір значень для елементів множини відносно невеликий та відомий наперед (наприклад множина букв українського алфавіту),

```
Введіть розмір множини, яка має бути згенерована
6

-724    -169    500     533     666     1959
Введіть значення елементу для вставки у множину
532

-724    -169    500     532     533     666     1959
Введіть значення елементу для видалення із множину
501
Такий елемент у множині відсутній

-724    -169    500     532     533     666     1959
Введіть розмір другої  множини, яка має бути згенерована
5

-705    38      536     855     1642
Результат обєднання двох множин

-724    -705    -169    38      500     532     533     536     666     855     1642    1959
```

Рис. 1.18: Результат виконання програми

є можливим реалізувати більш ефективну структуру даних типу множина з використанням бітових масивів. Так, якщо прості значень складається із елементів a_1, a_2, ... a_n, то множину можна задати за допомогою n - елементного бітового масиву (вектора): якщо i - тий біт рівний 1, то елемент a_i належить множині; якщо k - тий біт рівний 0, то елемент a_k не належить множині. Таке представлення дозволяє як зменшити необхідний для збереження множини об'єм оперативної пам'яті, так і ефективно реалізувати базові операції над множинами з використанням побітних операцій.

Бітові операції - це операції, що виконуються над окремими бітами (а не числом в в цілому). Існують наступні чотири бітові операції;

1. **побітове І ('&' - AND)** - якщо обидва біти операндів рівні 1, то результат дорівнює 1. Якщо хоча б один операнд дорівнює 0, то результуючий біт дорівнює 0;
2. **побітове заперечення НЕ ('∼' NOT)** - унарна операція, в результаті виконання якої якщо операнд був 0, то стане 1, і навпаки;
3. **побітове АБО ('|' OR)**- бінарна операція, дія якої еквівалентно застосування логічного **АБО** до пари операндів. Якщо обидва операнди рівні 0, результатом буде 0. Якщо ж хоча б один із операндів дорівнює 1, то результат дорівнює 1;
4. **Додавання за модулем два ('∧' XOR) (виключне АБО)**- бінарна операція, результат якої дорівнює 1, якщо сума операндів непарна, інакше результат рівний 0.

Всі операції з бітами можуть проводитися тільки із даними цілого типу чи

рядковими змінними. Стандартні побітові операції над множинами і вирази, що їм відповідають при роботі з масками:

- об'єднання множин: A | B;
- переріз множин: A & B;
- різниця множин: A & ∼ B;
- доповнення множини: ALL_BITS ∧ A (тут ALL_BITS – маска, в якій всі біти, що відповідають елементам, рівні 1).

Для представлення множини, яка може містити до 32 елементів можна використовувати ціле число стандартного 32 - розрядного цілого типу даних (при використанні 64 - розрядних цілих максимальна кількість елементів складатиме до 64 елементів відповідно). В такому числі, якщо i - тий біт дорівнює 1, то i - тий елемент входить в множині; якщо i - тий біт дорівнює 0, то i - тий елемент відсутній в множині . Таке ціле число, яким представляється множина, називають бітовою маскою множини.

Для маніпуляції бітами використовуються наступні операції:

- « – зсув вліво;
- » – зсув вправо:
- додавання i-того елементу в множину (встановлення i - го біту): $A| = 1 << i$;
- видалення i-того елемента: $A\& =\sim (1 << \text{i})$;
- перевірка, чи i - тий елемент належить множині: $(A\&(1 << \text{i}))! = 0$.

Оператор зсуву вліво $a << b$ зсуває біти числа a вліво на b позицій. Аналогічно оператор зсуву вправо зсуває біти вправо. При цьому, біти, що дописуються до числа, заповнюються нулями. Наприклад, якщо A = 1010, то

```
A << 2 = 101000;
A >> 1 = 101;
```

Доступу до значень відповідних бітів 32 - розрядного двійкового числа можна отримати за допомогою операції зсуву вправо. Приклад функції, що виводить побітово двійкове число за допомогою операції $>>$:

```
void DisplayBits(unsigned int  number)
{
const int length = 31;
int bitPos;
```

```
    int k;
    cout << "Бітове представлення числа"<< number<<endl;
        for (bitPos = length; bitPpos >= 0; bitPos --)
        {
        k=number >> bit_pos & 1;
        cout<<k;
        }
    cout<<endl;
}
```

Для підрахунку кількості бітів, що дорівнюють 1 (кількість елементів множини), достатньо видаляти встановлені біти із числа, поки воно не стане рівним 0:

```
    int Ones(int x)
    {
    int k = 0;
    // Поки число не стане рівним 0
    while (x != 0)
        {
            k++;
    //Видалення з маски x наймолодший встановлений біт.
            x &= x - 1;
        }
    return k;
    }
```

Більш детально робота із бітовими масками буде розглянута трохи пізніше у наступних розділах.

Приклад 3. 8. *Реалізація впорядкованої множини букв українського алфавіту з використанням бітових масивів. Для множини реалізовано наступні функції:*

- *генерація впорядкованої множини (InsertBit) ;*
- *вивід множини на (ShowSet);*
- *вивід бітового масиву (ShowBit);*
- *видалення елементів із множини (DelBit);*
- *виконання операції об'єднання двох множин(UnionBit).*

- *виконання операції перерізу двох множин (IntersectsonBit).*

```
#include <iostream>
#include <stdio.h>
#include <windows.h>
#include <string.h>
using namespace std;
int max_SIZE=70;
//Простір значень елементів множини.
char abc[70]="
абвгґдеєжзиіїйклмнопрстуфхцчшщьюяАБВГҐДЕЄЖЗИІЇЙКЛМНОПРСТУФХЦЧШЩЬЮЯ
.,";

//Функція виводу бітового масиву.
void ShowBit(bool *masca)
{
for (int i=0; i<max_SIZE; i++)
    cout << masca[i];
  cout<<endl;
}

//Функція виводу множини.
void ShowSet(bool *masca)
{
 for (int i=0; i<max_SIZE; i++)
 if(masca[i]==1) cout << abc[i] <<" ";
 cout<<endl;
 }

//Функція виводу елементів у множину.
void InsertBit(bool *masca, char *slowo)
{
 for(int i=0; i<strlen(slowo);i++)
 {
        //Перевірка, чи символ є елементом множини.
   if(strchr(abc,slowo[i])==NULL) continue;
        //Формування бітового вектора.
 masca[max_SIZE-1-strlen(strchr(abc,slowo[i]))]=1;
 }
```

```
    }

    //Функція видалення елементів із множини.
    void DelBit(bool *masca, char *slowo)
    {
      for(int i=0; i<strlen(slowo);i++)
     {
            //Перевірка, чи символ є елементом множини.
           if(strchr(abc,slowo[i])==NULL) continue;
          masca[max_SIZE-1-strlen(strchr(abc,slowo[i]))]=0;
      }
    }

     //Обєднання множин.
     void UnionBit(bool *masca, bool *masca_1, bool *masca_0)
    {
       for(int i=0; i<max_SIZE;i++)
            masca_0[i]=masca[i]| masca_1[i];
   }

    //Переріз множин.
    void IntersectsonBit(bool *masca, bool *masca_1, bool *masca_0
   )
    {
      for(int i=0; i<max_SIZE;i++)
       masca_0[i]=masca[i]& masca_1[i];

    }

    int main()
    {
     char slowo[100];
     SetConsoleCP(1251);
       SetConsoleOutputCP(1251);
        bool masca[70]={0};
        bool masca_1[70]={0};
      bool masca_0[70]={0};
      bool masca_2[70]={0};
    cout<<"Введіть першу множину"<<endl;
```

```
gets_s(slowo);
InsertBit(masca, slowo);
cout<<endl<<"Введіть елементи для видалення"<<endl;
gets_s(slowo);
DelBit(masca, slowo);
cout<<endl<<"Введіть другу множину"<<endl;
gets_s(slowo);
InsertBit(masca_1, slowo);
cout<<endl<<"Обєднання множин"<<endl;
UnionBit(masca,masca_1,masca_0);
ShowSet(masca_0);
cout<<endl<<"Переріз множин"<<endl;
IntersectsonBit(masca,masca_1,masca_2);
ShowSet(masca_2);
cout<<endl<<"Бітова маска першої множини"<<endl;
ShowBit(masca);
cout<<endl<<"Бітова маска другої множини"<<endl;
ShowBit(masca_1);
return 0;
}
```

```
Введіть першу множину
Приклад викоритстання бітових векторів для реалізації множини.

Введіть елементи для видалення
елементи

Введіть другу множину
Використання бітових операцій

Обєднання множин
а б в д е ж з и і ї й к н о п р с т х ц я В П   .

Переріз множин
а б в і к о р с х ц я

Бітова маска першої множини
11100100110110100010110001100000100000000000000000000100000000000001100

Бітова маска другої множини
111000100011011001111110011000001001000000000000000000000000000000001000
Press any key to continue . . .
```

Рис. 1.19: Результат виконання програми

Якщо можливий набір елементів є достатньо великим, то множину зручні-

ше реалізувати у вигляді багатоелементних структур даних, таких як зв'язані списки, дерева, графи і т. д. В цьому випадку елементи задаються , як мінімум, двома полями: інформаційним, де зберігається значення елементу множини та вказівником на наступний елемент. На відміну від попередніх способів представлення, при такому підході об'єм пам'яті, необхідний для збереження множини, пропорційний кількості елементів, а не загальній кількості можливих значень елементів. Алгоритми обробки множин будуть залежати від логічної структури даних, що цтворюють множину, та більш детально будуть розглянуті пізніше при розгляді відповідних структур даних.

Повний функціонал для роботи зі структурою даних типу 'множина' в C++ можна отримати, використавши шаблон класу бібліотеки STL, описаний у заголовочному файлі <set> .

Контейнер *Set* забезпечує наступну функціональність:

1. створення порожньої множини;
2. додавання елементу до множини: дублювання елементів виключається;
3. видалення елементів із множини;
4. перевірка наявності елементів у множині;
5. визначення кількості елементів в контейнері;
6. перевірка наявності певних елементів в контейнері.
7. вибір елементів із множини.

Порожня множина може бути оголошена згідно наступного синтаксису:

```
Set <type> змінна;
```

Тут: *type* - тип елементів множини. Приклад оголошення порожньої множини цілих чисел *mySet*:

```
// Підключення заголовочних файлів для роботи із даними типу
множина.
#include <algorithm>
#include <set>

    .  .  .  .  .  .  .  .  .

set<int> mySet;
```

Приклад поелементного додавання елементів у множину *mySet*:

```
  // Додавання одного цілого числа (81) у множину.
  mySet.insert(81);
  // Додавання у множину 10 випадкових чисел.
for(int i=1; i<10;i++)
   mySet.insert(rand()%100 );
```

Приклад іншого алгоритму додавання елементів у множину *mySet*:

```
   // Формування масиву цілих чисел.
   int as[6]={91,21,3,4,-50,6};
   set<int> mySet;
   // Додавання у масив перших чотирьох елементів масиву as.
      mySet.insert(as,as+5);
   // Визначення кількості елементів в контейнері mySet;
   int N = mySet.size();
   // Вивід максимальної кількості елементів,
   // що можна помістити у контейнер mySet
   cout << "Максимально можлива кількість елементів"
                      <<mySet.max_size()<<endl;
```

Видалення елементу із заданим значенням із множини *mySet*:

```
  // Видалення із множини числа 5
  mySet0.erase(5);
```

Приклад видалення елементів множини *mySet*, що знаходяться в заданому інтервалі значень. Приклад буде працювати, якщо елементи x_1 та x_2 присутні в множині *mySet*:

```
  set<int>::iterator it1, it2;
  it1 = mySet0.find(x1);
  it2 = mySet0.find(x2);
  // Збільшення ітератора it2 на 1 для видалення
  // елемента x2 включно.
  it2++;
  mySet0.erase(it1, it2);
```

Для контейнера *set* нема поняття *індекс елементу*, тому єдиним способом доступу до даних, що знаходяться у *set* є використання ітераторів. Приклад знаходження суми елементів множини:

```
 // Обчислення суми елементів множини.
 int Sum = 0;
     for(set<int>::const_iterator it = mySet0.begin();
                                     it != mySet0.end();
it++)
         Sum=Sum+(*it);
 cout <<  endl;
 cout<<"Сума елементів множини mySet0 = " <<Sum<<  endl;
```

Типовим прикладом використання ітераторів може бути також організація виводу елементів множини:

```
 for (set<int>::const_iterator it = mySet.begin(),end = mySet.
end();
                          it != end; ++it)
 cout <<"  "<< *it;
 cout << endl;
```

Виконання пошуку реалізується за допомогою операції *count*(), яка повертає або 1 (елемент присутній в множині), або 0 (елемент відсутній у множині). Приклад визначення, чи належить ціле число x до множини $mySet$.

```
 if(mySet.count(x))
         cout <<  " Належить множині "<<endl;
 else cout <<  " Не належить множині "<<endl;
```

Об'єднанням двох множин A та B називається множина, що складається із елементів, які входять хоча-б в одну із множин A або B.

$$A \cup B = \{x | x \in A \text{ або } x \in B\}$$

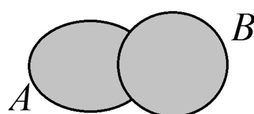


Рис. 1.20: Об'єднання двох множин

Приклад об'єднання двох випадковим чином згенерованих множин $mySet$ та $mySet0$.

```
    set<int> mySet;
    set<int> mySet1;
    //Перша множина.
  for(int i=1; i<10;i++)
  mySet.insert(rand()%100 );
    //Друга множина .
  for(int i=1; i<12;i++)
    mySet1.insert(rand()%100 );
    //Вивід першої множини.
    cout << "Перша множина"<<endl;
    for (set<int>::const_iterator it = mySet.begin(),
              end = mySet.end(); it != end; ++it)
      cout <<"  "<< *it;
  cout << endl;
  cout << " Друга множина"<<endl;
  //Вивід другої множини.
    for (set<int>::const_iterator it = mySet1.begin(),
                    end = mySet1.end(); it != end; ++it)
    cout <<"  "<< *it;
  cout << endl;
    set<int> result;
    // Операція обєднання множин.
    set_union(mySet1.begin(), mySet1.end(),
                mySet.begin(), mySet.end(),
                       inserter(result, result.end()));
    cout << endl;
    cout << " Обєднані множини"<<endl;
  //Вивід обєднаної множини.
    for (set<int>::const_iterator it = result.begin(),
            end = result.end(); it != end; ++it)
  cout <<"  "<< *it;
  cout << endl;
```

Перерізом двох множин A та B називається множина, що складається із елементів, які одночасно входять і в множин A і в множину B.

$$A \cap B = \{x | x \in A \text{ та } x \in B\}$$

```
Перша множина
 0  24  34  41  58  62  67  69  78
Друга множина
 5  27  36  42  45  61  64  81  91  95

Обєднані множини
 0  5  24  27  34  36  41  42  45  58  61  62  64  67  69  78  81  91  95
```

Рис. 1.21: Результат виконання фрагменту коду

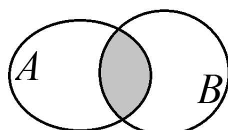


Рис. 1.22: Переріз двох множин

```
set_intersection(mySet1.begin(), mySet1.end(),
         mySet.begin(), mySet.end(),
            inserter(result, result.end()));
```

```
Перша множина
 0  5  24  34  41  58  61  62  67  69  78  81
Друга множина
 5  27  36  42  45  61  64  81  91  95

Переріз множин
 5  61  81
```

Рис. 1.23: Результат перерізу множини mySet1 та mySet

Різницею двох множин A та B називається множина, що складається із елементів множин A, які не входять в множину B.

$$A\backslash B = \{x|x \in A \text{ але } x \notin B\}$$

$$B\backslash A = \{x|x \in B \text{ але } x \notin A\}$$

```
// Різниця множин mySet1 та mySet.
set_difference(mySet1.begin(), mySet1.end(),
        mySet.begin(), mySet.end(),
            inserter(result, result.end()));
// Різниця множин mySet та mySet1.
```

Рис. 1.24: Різниця двох множин A та B

```
        set_difference(mySet.begin(), mySet.end(),
             mySet1.begin(), mySet1.end(),
                inserter(result_1, result_1.end()));
```

```
Перша множина
 0  5  24  34  41  58  61  62  67  69  78  81
Друга множина
 5  27  36  42  45  61  64  81  91  95

Різниці множин mySet1 та mySet
 27  36  42  45  64  91  95

Різниці множин mySet та mySet1
 0  24  34  41  58  62  67  69  78
```

Рис. 1.25: Переріз множин mySet1 та mySet

Симетрична різниця. Результатом є множина, яка містить елементи, які входять до складу однієї або другої множини (але не обох одночасно).

Рис. 1.26: Симетрична різниця двох множин

$$A \Delta B = \{(A \backslash B) \cup (B \backslash A\}$$

```
        set_symmetric_difference(mySet.begin(), mySet.end(),
            mySet1.begin(), mySet1.end(),
        inserter(result, result.end()));
```

```
Перша множина
 0  5  24  34  41  58  61  62  67  69  78  81
Друга множина
 5  27  36  42  45  61  64  81  91  95


Симетрична різниця множин mySet та mySet1
 0  24  27  34  36  41  42  45  58  62  64  67  69  78  91  95
```

Рис. 1.27: Симетрична різниця двох множин

Приклад 3. 9. *Програма початкової роздачі карт при грі в дурня з використанням контейнера Set. Кількість гравців задається із клавіатури.*

```
  #include <iostream>
  #include <windows.h>
  #include <algorithm>
  #include <set>
  #include <string>
  using namespace std;

  int main()
  {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
   string num[] ={"6","7","8","9","10",
      "Валет","Дама","Король","Туз"};;
  string karta;
int kilk;
set<string> koloda_Set;
set<string> karty_Set[4];
set<string>:: iterator it = koloda_Set.begin();
  // Формування множини: колода карт.
for (int i=0 ; i<8; i++)
      {
   koloda_Set.insert(num[i]+" Черва");
   koloda_Set.insert(num[i]+" Бубна");
   koloda_Set.insert(num[i]+" Трефа");
   koloda_Set.insert(num[i]+" Піка");
      }
  cout << endl;
```

```
 cout <<"Введіть кількість граків"<< endl;
  cin>>kilk;
  // Кількість граків.
  for (int j1=0;j1<kilk;j1++)
      {
      // Кількість карт.
  for(int j=0;j<6;j++)
      {
      // Вибір випадковим чином карти.
      int i_0=koloda_Set.size();
      i_0=rand()%i_0;
      it = koloda_Set.begin();
      // Перевід ітератора на вибрану карту.
      advance(it,i_0);
      // Вибір карти
      karta=*it;
      // Видалення карти із колоди
      koloda_Set.erase(karta);
      // Формування множини карт для кожного із граків
      karty_Set[j1].insert(karta);
      }
      }
  cout << "\t\t\t Здача карт"<<endl;
  // Вивід здачі на екран/
  for(int i=0;i<kilk;i++)
      {
  cout << " \t\t Карти грака №= "<<i+1<<endl;
  cout <<endl;
  // Власне вивід сформованих нових множин.
  copy( karty_Set[i].begin(), karty_Set[i].end(),
              ostream_iterator<string>(cout, " | "));
  cout <<endl;
  cout <<endl;
  }
  cout <<endl;
  return 0;
  }
```

```
Введіть кількість граків
4
                         Здача карт
                 Карти грака №= 1

6 Бубна  |  7 Піка  |  8 Бубна  |  9 Черва  |  Валет Черва  |  Дама Трефа  |

                 Карти грака №= 2

6 Трефа  |  7 Бубна  |  7 Трефа  |  8 Трефа  |  8 Черва  |  Валет Трефа  |

                 Карти грака №= 3

10 Піка  |  10 Черва  |  6 Піка  |  9 Бубна  |  Дама Бубна  |  Король Трефа  |

                 Карти грака №= 4

10 Бубна  |  6 Черва  |  7 Черва  |  8 Піка  |  Дама Піка  |  Король Бубна  |

Press any key to continue . . .
```

Рис. 1.28: Результат роботи програми

1.3.6 Структури

Структура - складений тип даних, змінні якого можуть містити дані різних типів. Елементи структури називаються полями структури і можуть мати довільний тип, крім типу цієї ж структури, але можуть бути покажчиками на неї. Структура описується згідно наступного синтаксису:

```
        struct <ім'я структури>
    {
    <тип поля_1>  <ім'я поля_1>;
    <тип поля_2>  <ім'я поля_2>;

    .  .  .  .  .  .  .  .  .  .  .  .

    <тип поля_k>  <ім'я поля_k>;
    }
    список< змінних>;
```

Приклад оголошення та опису масиву структур, що мітить наступну інформацію про студента: прізвище; ім'я; по батькові; номер групи; чотири оцінки здачі сесійних екзаменів; середній бал за результатами сесії.

```
struct student
       {
            char surname[15];
            char name[15];
            char number[7];
            int score[4];
            double sb;
        }
ms[1000];
```

В результаті буде створено новий тип даних *student*, який може оголошуватися у програмі аналогічно до стандартних типів. Змінна типу *student* має п'ять полів: три рядки (*surname* , *name* та *number*), одномірний масив *score* цілих чисел (містить оцінки за екзамен) та дійсне число *middle* (середній бал за здані екзамени). При оголошенні типу *student* послідовно під усі поля структури виділяється пам'ять:

- змінна *surname*: 15 байтів;
- змінна *name*: 15 байтів;

- змінна *name*: 7 байтів;
- змінна *score*: 16 байтів;
- змінна *sb*: 8 байтів.

Сумарна пам'ять, що виділяється під збереження структури, може бути більшою для так-званого вирівнювання адрес. Так, у наведеному прикладі сумарна пам'ять для збереження полів рівна 61 байт, а реальний обсяг виділеної компілятором під збереження структури пам'яті рівний 64 байти.

Одночасно із оголошенням структури вона може бути ініціалізована згідно наведеного нижче прикладу:

```
student st={"Федорів","Петро", "ПІ-11",5, 4, 4, 3, 4.000};
```

При ініціалізації масиву структур кожен елемент масиву записується у фігурних дужках. Найважливішою операцією для структури є операція доступу до вибраного поля структури. Доступ здійснюється за допомогою операції "крапка"згідно наступного синтаксису:

```
<структурна змінна>.<ім'я поля>
```

Приклад присвоєння значень полям структури для змінної *st*:

```
  strcpy(st.surname,"Федорів");
strcpy(st.name,"Петро");
strcpy(st.number,"ПІ-11");
st.score[0]=5;
st.score[1]=4;
st.score[2]=4;
st.score[3]=3;
st.sb=(st.score[0]+st.score[1]+st.score[2]+st.score[3])/4.0;
```

Структури можуть розміщатись у динамічній пам'яті за допомогою оператора *new* та оголошуватися як вказівники. У цьому випадку доступ до полів структури здійснюється за допомогою операції вибору – "стрілка" (->):

```
// Виділення пам'яті.
student *st1=new student;
// Присвоєння оцінки полю score[0].
st1->score[0]=5;
// Присвоєння значення полю surname.
strcpy(st1->surname," Федорів");
// Присвоєння оцінки полю score[3] з використанням
вказіваника.
```

```
(*st1).score[3]=3;
```

Важливо, що якщо $st1$ - вказівник на структуру, то $(*st1)$ - сама структура і доступ до її полів здійснюється за допомогою крапки.

Приклад виводу полів створеної раніше структури, розміщеної в динамічній пам'яті :

```
// Вивід прізвища.
cout << "Студент "<<(*st1).surname<<endl;
//Вивід оцінки.
cout << "Оцінка "<<(*st1).score[3]<<endl;
```

Полями структури можуть бути дані довільного типу, в тому числі і структури, за виключенням типу тієї-ж самої структури. Для прикладу створимо дві структури для збереження інформації про студента, аналогічну до раніше створеної структури *student*.

```
struct sesion
  {
 int score[4];
 double sb;
  };
struct student
  {
    char surname[15];
    char name[15];
    char number[7];
    // Поле структури - структура sesion.
    sesion rez;
   };
```

Присвоєння значення полям структури:

```
  strcpy(st.surname,"Федорів");
strcpy(st.name,"Петро");
strcpy(st.number," ПІ-11");
st.rez.score[0]=5;
st.rez.score[1]=4;
st.rez.score[2]=4;
st.rez.score[3]=3;
st.rez.sb=(st.rez.score[0]+st.rez.score[1]+
          st.rez.score[2]+st.rez.score[3])/4.0;
```

Вивід полів структури на екран:

```
cout << "Прізвище="<<st.surname<<endl;
cout << "Номер групи="<<st.number<<endl;
cout << "Середній бал="<<st.rez.sb<<endl;
```

Приклад роботи зі структурою, що буде розміщена в динамічній пам'яті: оголошення змінної *dani* типу структури *student*

```
student *dani=new student;
```

присвоєння значення полям:

```
strcpy(dani->surname,"Федорів");
strcpy(dani->name,"Петро");
strcpy(dani->number,"ПІ-11");
dani->rez.score[0]=5;
dani->rez.score[1]=4;
dani->rez.score[2]=4;
dani->rez.score[3]=3;
dani->rez.sb=((*dani).rez.score[0]+(*dani).rez.score[1]+
            (*dani).rez.score[2]+(*dani).rez.score[3])/4.0;
```

вивід значення полів:

```
cout << "Прізвище="<<(*dani).surname<<endl;
cout << "Номер групи="<<(*dani).number<<endl;
cout << "Середній бал="<<(*dani).rez.sb<<endl;
```

Над вибраним полем структури можливі будь-які операції, які допустимі для типів цього поля. Більшість мов програмування підтримує деякі операції, які працюють із структурою, як з єдиним цілим, а не з окремими її полями. Це операція присвоєння значення одного запису іншому однотипному запису, при цьому відбувається поелементне присвоєння всіх полів.

```
student st;
student dani;

. . . . . . . .

//Присвоєння однотипних структур.
dani=st;
```

Структура у C++ крім звичайних полів може містити і елементи-функції. Приклад доповнення раніше створеної структури *student* функцією $uniwer_bal$, яка переводить середній бал студента в 100-бальній шкалі в університетську шкалу (відмінно,добре, задовільно, незадовільно) та функцією *ocinka*, що переводить середній бал в п'ятибальну шкалу.

```
   #include <iostream>
   #include <windows.h>
   using namespace std;
   struct student
      {
           char surname[15];
           char name[15];
           char number[7];
           int score[4];
           double sb;
 //Функція переводу середнього балу в університетську шкалу
 void uniwer_bal()
 {
     if (sb>=90)cout << "Відмінно"<<endl;
   if (sb<90&&sb>=70)cout << "Добре"<<endl;
   if (sb<70&&sb>=50)cout << "Задовільно"<<endl;
   if (sb<50)cout << "Незадовільно"<<endl;
 }
 //Функція переводу середнього балу в п'ятибальну шкалу
 int ocinka()
 {
   if (sb>=90) return 5;
   if (sb<90&&sb>=70)return 4;
   if (sb<70&&sb>=50)return 3;
   if (sb<50)return 2;
 }
  }
   ;
       int main()
   {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
 student st;
   //Присвоєння значень полям структури st.
```

```
strcpy_s(st.surname,"Федорів");
strcpy_s(st.name,"Петро");
strcpy_s(st.number,"ПІ-11");
st.score[0]=85;
st.score[1]=74;
st.score[2]=78;
st.score[3]=93;
st.sb=(st.score[0]+st.score[1]+st.score[2]+st.score[3])/4.0;
  //Вивід значень відповідних полів структури st
  cout << "Прізвище=  "<<st.surname<<endl;
  cout << "Номер групи=  "<<st.number<<endl;
  cout << "Середній бал=  "<<st.sb<<endl;
  cout << "Оцінка за університетською шкалою  ";
  //Виклик функції uniwer_bal().
 st.uniwer_bal();
  //Перевід оцінки до п'ятибальної шкали за допомогою функції
 ocinka().
 cout << "Оцінка за пятибальною шкалою  "
                  <<st.ocinka()<<endl;;
  return 0;
  }
```

```
Прізвище=  Федорів
Номер групи=  ПІ-11
Середній бал=  82.5
Оцінка за університетською шкалою  Добре
Оцінка за пятибальною  шкалою  4
Press any key to continue . . .
```

Рис. 1.29: Результат виконання програми

Структури до функції також можуть бути передані як звичайні змінні:

```
  // Функція переведення на наступний курс.
  void Perevid(student &S,int n)
{
  int j;
  for(j=0; j<n; j++)
          if(S.score[j]<50)
        {
           cout << "Студент не може бути переведений
```

```
                  на наступний курс  "<<endl;
          return;
      }

   cout << " Студент може бути переведений
                  на наступний курс "<<endl;
 }
```

Звернення до функції *Perevid* в програмі для структури *st* виглядає наступним чином:

```
   st.Perevid(st,4);
```

При необхідності у структурі оголосити два поля так, щоби вини не могли одночасно приймати значення: їх можна об'єднати в єдине ціле, скориставшись структурою даних типу об'єднанням *union*, яке більш детально буде розглянуто в наступному розділі . Приклад такого поля в структурі:

```
   union
 {
   int children;
   char war[15];
 };
```

1.3.7 Об'єднання

Об'єднання - формат даних, який може містити різні типи даних, але водночас тільки один із них. Фактично об'єднання представляють собою частковий випадок структури, усі поля якої розміщуються за однією ж і тою ж адресою в оперативній пам'яті. Таким чином можна створити ділянку пам'яті спільного користування, до якої можна звертатися через єдину змінну, що має тип об'єднання. Формат опису аналогічний до опису структури:

```
        union ім<'я об'єднання>
    {
                тип< поля_1>  ім<'я поля_1>;
                тип< поля_2>  ім<'я поля_2>;
                .  .  .  .  .  .  .  .  .  .  .  .
                тип< поля_k>  ім'я поля_k;

        };
```

Наприклад:

```
        union demo
            {
                int  i;
                float x;
                double z;
                char ch;
            } Dem;
```

У наведеному прикладі обєднання може містити або дані типу int, або $float$, або $double$, або $char$. Об'єм пам'яті, необхідний для збереження даних визначається найбільшим за розміром його полів. У кожен момент часу в змінній типу об'єднання може зберігатися тільки одне значення, а контроль на його правильне використання покладається на програміста. Об'єднання використовується для економії оперативної пам'яті, коли відомо, що більше одного поля одночасно не буде потрібно.

Всі чотири поля змінної Dem мають однакове нульове зміщення відносно адреси початку ділянки, яку займає Dem. Це означає, що в області молодших байтів всі поля об'єднання накладаються (спільне використання області пам'яті).

Змінна типу об'єднання може бути ініціалізована: ініціалізується тільки значення першого поля обєднання. Для наведеного прикладу це змінна i типу

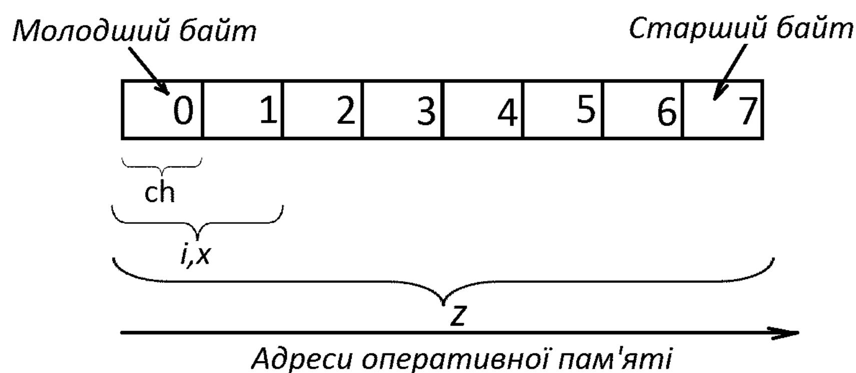


Рис. 1.30: Взаємне накладання полів обєднання Dem

int. Наприклад

$$demo\ Dem = \{43\}$$

До змінних типу об'єднання доступними є всі ті ж самі операції, що і до структурних змінних - це присвоєння, визначення адреси змінної, виділення окремих елементів поля. Для звернення до елементів об'єднання використовуються операції "крапка"(.) та "стрілка"($->$):

$$\text{ім'я змінної об'єднання . ім'я поля}$$

$$\text{вказівник на об'єднання}- > \text{ім'я поля}$$

Наприклад, якщо виконати присвоєння

$$Dem.z = 23.14$$

то полю z змінної Dem буде присвоєно значення 23.14

Об'єднання застосовуються для економії пам'яті в тих випадках, коли відомо, що більше одного поля одночасно не потрібно, а також для різної інтерпретації одного і того ж бітового представлення. Поля об'єднання можуть бути довільного типу - простого чи складеного. Так, поле об'єднання може бути структурою, вкладеним об'єднанням чи вказівником. Об'єднання можуть бути також елементами масиву обо складовими частинами структур чи інших об'єднань.

Приклад використання об'єднання як поля структури. Для визначення, який саме елемент об'єднання використовується в даний момент часу, в структурі використовується додаткове поле $type$.

```
        struct scr
            {
```

```
        int  type;
        union ident
                  {long num; char ch[25];  };
    }
              price;
    .  .  .  .  .  .  .
    if (price.type= =1) cin>>price.ident.num;
        else cin>>price.ident.ch;
```

В залежності від значення *type* поле *ident* буде приймати одне із значень: або типу *long*. , або типу *char*

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

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

1.3.8 Бітові типи

Бітові поля - особливий вигляд елементів структур та об'єднань. Такі поля являють собою послідовність бітів, записаних у дане цілого типу.

Частіше всього такий тип даних використовують при необхідності мати доступ до окремих бітів в середині байта чи слова даних. Наприклад, така потреба виникає при управлінні апаратними засобами комп'ютера, наприклад при зверненні до регістрів відеоадаптера. Іншим прикладом використання бітових полів може бути стиснення даних для їх компактного збереження: коли дані є булівського типу (так чи ні) і для їх збереження достатньо одного біта Мінімально адресована комірка пам'яті – 1 байт, а для зберігання прапорця достатньо одного біта. Бітові поля описуються за допомогою структурного типу.

Оголошення бітового поля здійснюється за допомогою наступної синтаксичної конструкції:

$$< \text{тип поля} > \quad < \text{ім'я бітового поля} > \quad : \quad < \text{розмір поля} >$$

Тут *тип поля* може бути тільки *int* (з додатковим специфікатором *unsignet*) чи *signet*; *ім'я бітового поля* - звичайний ідентифікатор змінної; *розмір поля* - ціле число, що задає ширину поля в бітах.

Доступ до поля здійснюється звичайним способом – за іменем. Над бітовими типами можливі три групи специфічних операцій: операції алгебри логіки, операції зсуву, операції порівняння.

Операції зсуву виконують зміщення бінарного коду на задану кількість розрядів ліворуч або праворуч. Із трьох можливих типів зсуву (арифметичний, логічний, циклічний) в мовах програмування частіше реалізується лише логічний.

Бітові поля (bit fields) - особливий вид полів структури. Вони дають можливість задавати кількість бітів, в яких зберігаються елементи цілих типів. Бітові поля дозволяють раціонально використовувати пам'ять за допомогою зберігання даних в мінімально потрібній кількості бітів.

При оголошенні бітового поля вслід за типом елемента ставиться двокрапка (:) і вказується цілочисельна константа, яка задає розмір поля (кількість бітів). Розмір поля повинен бути константою в діапазоні між 0 і заданим загальним числом бітів, яке використовується для зберігання даного типу даних.

Для роботи із бітовими полями служить також спеціальний контейнер *bitset*, який використовується для збереження бітових значень: елементи контейнера можуть приймати тільки значення 0 (*істина*) або 1 (*хиба*). Фактично *bitset* - бітовий масив, для якого реалізовано операції довільного доступу,

зміни значень окремих бітів та обробки масиву в цілому. Нумерація бітів починається із нуля та відбувається справа наліво. Для бітової множини не визначене поняття ітератора, тому вона не вважається контейнером в чистому вигляді, бо не забезпечує стандартний інтерфейс.

Шаблон контейнера *bitset* визначено в заголовочному файлі $< bitset >$.

```
#include <bitset>
```

Приклади оголошення бітової множини:

задається розмір контейнера із ім'ям *bitSet*, значення елементів якого по замовчуванню рівні 0. В наведеному прикладі формується масив 32 бітів:

```
bitset <32> bitSet;
```

оголошення контейнера із його одночасною ініціалізацією. Для ініціалізації необхідно задати беззнакове ціле число як аргумент конструктора. Початкові N позицій (залежить від аргументу) отримають значення двійкових розрядів аргументу:

```
bitset <30> bit2 (10);
//Аргумент задано як число в 16-ковій системі числення.
bitset <10> bit3 (0x1f);
```

в якості аргументу задається рядкова змінна типу *string*, яка складається із нулів та одиниць:

```
string a="1001111111111101";
bitset <16> bitSet1(a);
//Підрядок, починаючи із 8-мої позиції.
bitset <16> bitSet2(a,10);
//Підрядок, починаючи із 5-тої позиції довжиною 5 символів.
bitset <16> bitSet3(a,5,5);
```

```
bitSet1=0100111111111101
bitSet2=0000000001111101
bitSet3=0000000000011111
```

Рис. 1.31: Результат ініціалізаці бітових множин

Кожен байт (елемент *bitset*) є доступним незалежно від інших байтів як за допомогою відповідних функцій, так і за допомогою індексів аналогічно до елементів звичайного масиву. До бітової множини визначені також стандартні бітові операції та операції порівняння: $==$ (рівні) чи $!=$ (не рівні).

Табл. 1.12: Основні операції із контейнером *bitset*

Операція	**Значення**
test(pos)	Перевіряє, чи значення біта в позиції *pos* рівне 1.
any()	Перевіряє, чи хоча-б один біт рівний 1.
none()	Перевіряє, чи жоден біт не рівний 1.
count()	Повертає кількість бітів, рівних 1.
size()	Повертає загальну кількість бітів.
flip()	Міняє значення всіх бітів.
flip(pos)	Міняє значення біта в позиції pos.
set()	Всім бітам присвоюється значення, рівне 1.
set(pos)	Біту в позиції pos присвоюється значення, рівне 1.
reset()	Всім бітам присвоюється значення, рівне 0.
reset(pos)	Біту в позиції pos присвоюється значення, рівне 1.

Для контейнера *bitset* визначено також дві функції для перетворення бітової множини до іншого типу:

перетворення до рядкової змінно типу *string*, що складається із символів нулів та одиниць:

```
  string ST_bit;
  //bitSet - попередньо задана бітова множина.
ST_bit=bitSet.to_string();
```

перетворення бітового вектора до цілого числа типу unsigned long (якщо він поміщається в даний тип)

```
  unsigned long ki;
  //bitSet - попередньо визначена бітова множина.
ki=bitSet.to_ulong();
```

Приклад 3. 10. *Програма генерує 6 випадкових цілих чисел в діапазоні від 1 до 36 та пропонує їх вгадати. Результат роботи - кількість вгаданих чисел та їх значення.*

```
  #include <iostream>
  #include <stdio.h>
  #include <windows.h>
  #include <bitset>
```

```
  #include <string.h>

  using namespace std;
  #define Max_SIZE 37

  //Генерація 6 різних чисел.
  void Generator(bitset <Max_SIZE> &bit_st)
  {
int i=0;
int n;
      while(i<6)
      {
          n=rand()%36+1;
          if (bit_st[n]==1) continue;
          i++;
          bit_st[n]=1;
      }
  }

  void Vvid(bitset <Max_SIZE> &bit_st0)
  {
      int n;
      int i=0;
      // Ввід 6 чисел із перевіркою коректності їх вводу.
      cout<<"Введіть шість чисел"<<endl;
      while(i<6)
          {
          cout<<"Число № "<<i+1<<endl;
          cin>>n;
          if ( n<1 || n>36)
          {
          cout<<"Введено некоректне число"<<endl;
          continue;
          }
          if (bit_st0[n]==1)
          {
       cout<<"Таке число вже було введено"<<endl;
          continue;
          }
```

```
            i++;
            bit_st0[n]=1;
            }
        }

    int main()
    {
  int n;
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    bitset <Max_SIZE> bit_st;
    bitset <Max_SIZE> bit_st0;
    bitset <Max_SIZE> Rez;
    Generator(bit_st);
    Vvid(bit_st0);
    // Вазначення кількості вгаданих чисел як
    // переріз множин.
    Rez=bit_st0 & bit_st;
    n=Rez.count();
    // Вивід вгаданих чисел.
    cout<<"Ви вгадали "<<n<<" чисел"<<endl;
    for(int i=0;i<Max_SIZE; i++)
    if(Rez.test(i)) cout<<i<<"  ";
    cout<<endl;
    return 0;
    }
```

```
Введіть шість чисел
Число № 1
5
Число № 2
19
Число № 3
18
Число № 4
1
Число № 5
29
Число № 6
-1
Введено некоректне число
Число № 6
18
Таке число вже було введено
Число № 6
10
Ви вгадали 3 чисел
5  18  29
Press any key to continue . . .
```

Рис. 1.32: Результат роботи програми

1.3.9 Таблиці

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

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

Іноді розрізняють таблиці з фіксованою і із змінною довжиною структури. Очевидно, що таблиці, які об'єднують структури ідентичних типів, будуть мати фіксовані довжини структур. Необхідність в змінній довжині може виникнути в задачах, подібних до тих, які розглядалися для об'єднань. Як правило таблиці для таких задач і складаються із структур до складу яких входять об'єднання, тобто зводяться до фіксованої (максимальній) довжини

структури. Значно рідше зустрічаються таблиці з дійсно змінною довжиною структури. Хоча в таких таблицях і економиться пам'ять, але можливості роботи з такими таблицями обмежені, оскільки за номером структури неможливо визначити її адресу.

Таблиці із структурами змінної довжини обробляються тільки послідовно – в порядку зростання номерів структур. Доступ до елемента такої таблиці звичайно здійснюється в два кроки. На першому кроці вибирається постійна частина структури, в якій міститься, – в явному чи неявному вигляді – довжина структури. На другому кроці вибирається змінна частина структури у відповідності з її довжиною. Додавши до адреси поточної структури її довжину, одержують адресу наступної структури.

У хеш-таблиці замість безпосереднього використання ключа як індексу масиву, індекс обчислюється за значенням ключа. Функція, що відображає елемент множини ключів $\{0, 1, ..., n-1\}$ на множину індексів $\{0, 1, ..., m-1\}(m < n)$, називається хеш-функцією. Якщо два ключі хешуються в одну й ту саму комірку, то говорять про виникнення колізії. За способом вирішення колізій розрізняють:

- відкрите хешування — усі елементи, що хешуються в одну комірку, об'єднуються у зв'язний список;
- закрите хешування — усі елементи зберігаються безпосередньо у хеш-таблиці, при потраплянні у зайняту комірку обирається послідовність інших хеш-значень.

Необхідно забезпечити для хеш-таблиці реалізацію основних операторів:

- пошук елемента;
- запис елемента;
- читання елемента.

При закритому хешуванні хеш-таблиця є масивом, елементи якого занумеровані від 0 до $m-1$. При відкритому хешуванні хеш-таблиця є масивом, кожна комірка якого містить покажчики на заголовок списку всіх елементів, хеш-значення ключа яких дорівнює індексу комірки.

Значенням хеш-функції є остача від ділення ключа на розмір хеш-таблиці.

$$h(x) = x\%m$$

1.4 Динамічні структури даних

Для статичних змінних характерним є те, що для їх збереження область в оперативній пам'яті виділяється на етапі компіляції програми та не міняється в процесі її виконання. У багатьох задачах обсяг оперативної пам'яті, необхідний для збереження даних, неможливо визначити наперед. Для збереження такого типу даних використовуються змінні, які можуть бути створені та видалені в ході виконання програми.

Динамічні структури даних характеризуються наступними ознаками:

1. вони мають змінну довжину і прості процедури її зміни;
2. зміна довжини структури відбувається в певних межах, не перевищуючи певного максимального (граничного) значення.

Якщо динамічну структуру розглядати на логічному рівні, то це послідовність даних, зв'язана відносинами лінійного списку. Доступ до елемента може здійснюватися за його порядковим номером.

Фізичне представлення динамічної структур даних в пам'яті – це звичайно послідовність комірок в пам'яті, де кожний наступний елемент розташований в пам'яті в наступній комірці. Фізичне представлення може мати також вид однонаправленого зв'язного списку (ланцюжки), де кожний наступний елемент адресується покажчиком, який знаходиться в поточному елементі. У цьому випадку обмеження на довжину структури менш строгі.

1.4.1 Список

Зв'язані списки та масиви. Списки мають деякі переваги над масивами. Вони досить ефективні щодо операцій додавання або видалення елементу в довільному місці списку, виконуючи їх за постійний час, тоді як масиви для цього потребують часу $O(n)$, тобто час зростає з ростом кількості елементів масиву. В списках також не існує проблеми "розширення яка рано чи пізно виникає в масивах фіксованого розміру, коли виникає необхідність включити в нього додаткові елементи. Точно так, фіксований масив, з якого було видалено багато елементів (або вони просто не використовуються) є дуже неефективним з точки зору використання пам'яті. Функціонування списків можливо в ситуації, коли пам'ять комп'ютера фрагментована, тоді як масиви переважно потребують неперервної області для зберігання.

З іншого боку, масиви дозволяють безпосередній доступ до будь-якого елементу. Однобічно зв'язані списки, натомість, потребують проходження усіх попередніх елементів. Це призводить до складнощів застосування списків в

задачах, де необхідно швидко знаходити елемент за його індексом, наприклад в алгоритмах сортування. Кешування списків в таких випадках майже не дає ефекту.

Іншим очевидним недоліком списків є необхідність разом з корисною інформацією додаткового збереження інформації про вказівники, що позначається на ефективності використання пам'яті цими структурами.

Списки використовуються замість масивів для зберігання й опрацювання однотипних даних, кількість яких заздалегідь є невідома й може змінюватися у процесі роботи. У списках просто організувати процеси видалення елементу чи його вставку на довільне місце. Принципова перевага списку перед масивом полягає у їх структурній гнучкості: порядок слідування елементів списку може не співпадати із порядком розміщення елементів у пам'яті комп'ютера, а порядок обходу списку завжди задається явним чином за допомогою його внутрішніх зв'язків

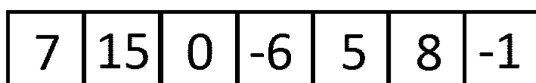
Лінійний зв'язаний список- це набір однотипних компонентів, які послідовно пов'язані між собою за допомогою покажчиків. Кожен компонент списку може складатися із кількох інформаційних полів та покажчика на наступний елемент. Інформаційні поля елементу списку можуть бути змінними будь-яких типів

Зі такими структурами типу список можуть бути реалізовані наступні операції:

- отримання k-того елемента списку для читання чи запису в нього нового значення:
- додавання нового елемента в будь-яку позицію в списку;
- видалення елемента списку;
- об'єднання в одному списку двох або більше лінійних списків;
- розбиття списку на два або більше фрагментів;
- створення копії списку;
- визначення кількості елементів в списку;
- сортування елементів списку;
- обмін елементів списку місцями;
- пошук елемента, що задовільняють певним критеріям.

За означенням, список — це послідовність з $n \geq 0$ елементів $x[0], x[1], \ldots x[n-1]$, для якої виконується наступна умова: якщо $n > 0$ та $x[0]$ — перший елемент у списку, а $x[n-1]$ — останній, то k-тий елемент розташований між $x[k-1]$ та $x[k+1]$ елементами для усіх $1 < k < n$.

У оператичній пам'яті одновимірний масив з 7-ти елементів (наприклад масив цілих чисел $7, 15, 0, -6, 5, 8, -1,$) розміщується наступним чином:



Це означає, що елементи розташовані в пам'яті один за одним, що дає можливість їх нумерувати, але ускладнює в ході роботи програми додавання нових елементів, чи їх вилучення.

На відміну від масиву елементи списку розташовані в оперативній пам'яті випадковим чином, що дозволяє без проблем додавати чи видаляти елементи у довільному місці. Наприклад, список елементів, аналогічний попередньому масиву, може бути розташований у пам'яті наступним чином:

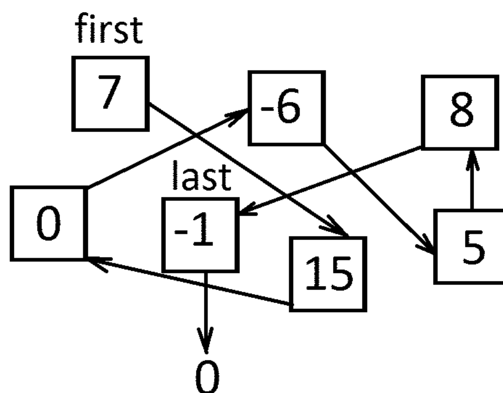


Рис. 1.33: Розміщення елементів списку в оперативній пам'яті

На рисунку стрілками задано порядок проходження елементів списку. Для того, щоби працювати з елементами як зі зв'язною структурою, необхідно задати, де в пам'яті розташовано кожний з елементів, тобто фактично знати їхні адреси. Цілком достатньо зберігати тільки адресу першого елемента (покажчик на елемент $First$). Кожний наступний елемент списку повинен містити інформацію про розташування наступного за ним елементу. При цьому немає необхідності в даному елементі зберігати інформацію про розташування решти елементів. До будь-якого i-того елементу можна добратись, послідовно пройшовши по ланцюжку від першого до другого, від другого до третього і т. д аж до i-того елементу.

У наведеному прикладі списку елемент зі значенням 7 містить інформацію (вказівник) про розташування елементу зі значенням 15 і т. д.. Щоби перейти до елементу зі значенням -6, необхідно взяти із першого елементу (7) адресу другого (15), із нього адресу третього (0), і на останньому етапі отримати адресу четвертого елементу (-6). Безпосередньо напряму отримати значення четвертого елементу неможливо.

Для зручності та наочності список, як правило, зображають наступним чином:

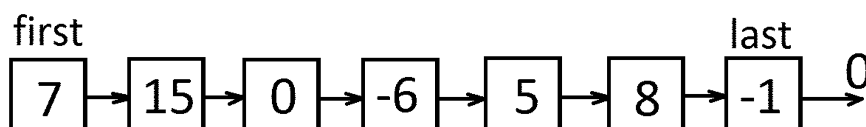


Рис. 1.34: Наочний спосіб представлення списку

Для організації списку кожен елемент, окрім даних, повинен також містити ще одне значення: адресу наступного елемента. Зручно подати елементи списку у вигляді структури, яка містить два поля: поле даних (значення елементу) та поле адреси (вказівник на наступний елемент) . Після останнього елементу більше елементів немає, тому вказівник на наступний елемент повинен залишатися порожнім ($NULL$) дорівнювати 0:

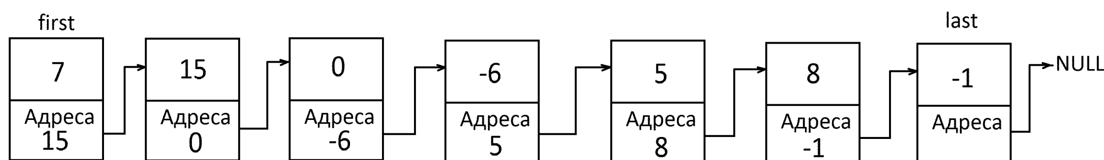


Рис. 1.35: Організація списку за допомогою структури

Верхня частина кожного елементу задає його числове значення (поле даних), а нижня - задає адресу наступного елементу (поле $next$). Оголошення структури типу "елемент списку" матиме наступний вигляд:

```
struct <елемент списку>
    {
         < поле даних>;
         поле< адреси>*  ;
    };
```

Приклад оголошення елементу структури *List*, що складається із даних цілого типу:

```
struct List
{
 //Інформаційне поле.
  int Data;
  //Поле адреси.
  List *next;
};
```

Покажчик на перший елемент цього списку можна оголосити наступним чином:

```
List *first;
first=NULL;
```

Його початкове значення завжди дорівнює 0: тобто списку поки-що нема. Після створення списку $first$ прийме деяке ненульового значення - значення адреса пам'яті ЕОМ, у якій розташовано перший елемент. Наприклад:

```
first=003B21F0
```

Знаючи покажчик на перший елемент, до другого елемента можна звернутися наступним чином: $first- > next$, до третього: $first- > next- > next$ і т. д. Числове значення першого елемента: $first- > Data$. Важливо, що $first$ не є власне елементом списку, а лише покажчиком на нього, тобто є покажчиком на структуру. Тому звертання до полів структури відбувається не через крапку, а через стрілку -> .

Переміщення по елементах списку, як правило, реалізується із використанням циклу *while*. Циклічна процедура повторюється, поки не добралися до потрібного елементу, чи список не завершено. Для цього використовується допоміжний покажчик $Current$, який оголошується наступним чином:

```
List *Current;
```

Перемістившись спочатку до першого елементу ($Current = first$;) у циклі, доки не завершено список з поточним елементом списку виконується необхідна дія, після чого здійснюється перехід до наступного елемента списку:

```
    // Допоки список не завершено,
    while (Current!=NULL);
            {
                    ... // Виконання дії з елементом списку.
                        // Перехід до наступного елемента.
                        Current = Current->next;
            }
```

Приклад функції, що виводить елементи списку на екран:

```
    void Show_List(List *first)
    {
      cout<<endl<<"Лінійний список"<<endl;
      do
      {
        cout<<first->Data<<"  ";
            //Перехід до наступного елемента.
        first = first->next;
      }
      //Поки не досягнуто кінець списку
      while (first!=NULL);
      cout<<endl;
    }
```

Приклад функції, що підраховує кількість елементів списку :

```
    int Size_List(List* first)
    {
      int i=0;
      do
      {
        i++;
        first = first->next;
     }
      //Поки не досягнуто кінець списку
      while (first!=NULL);
      return i;
    }
```

Приклад функції, що виконує пошук заданого елементу $Item$ списку :

```
bool Find_Element(List* first, int Item)
{
  //Поки не досягнуто кінець списка
  while (first != NULL){
    if (Item == first->Data) return true;
    else first = first->next;
 }
 return false;
}
```

Алгоритм створення першого елементу списку:

1. виділити місце у пам'яті для збереження елементу:

```
List *NewItem=new List;
```

2. присвоїти елементу числове значення $Item$:

```
 NewItem->Data=Item;
```

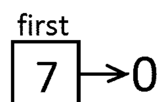
3. присвоїти покажчику на наступний елемент значення $NULL$:

```
NewItem->next = NULL;
```

4. покажчику на елемент $first$ присвоїти адресу створеного елементу:

```
(*first) = NewItem;
```

Схематично створений список, що складається із одного елементу, рівного 7, виглядає наступним чином:



Алгоритм додавання нового елементу (-3) в кінець списку:

1. виділити під елемент місце в пам'яті :

```
List *NewItem=new List;
```

2. присвоїти полю даних його значення (-3):

```
Item=-3;
NewItem->Data=Item;
```

3. покажчику на наступний елемент присвоїти значення $NULL$:

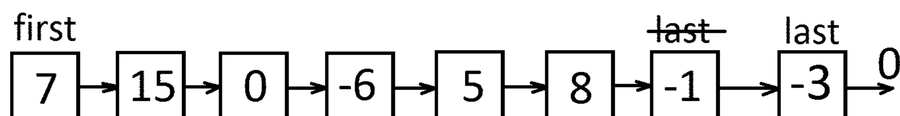
```
NewItem->next = NULL;
```

4. створений елемент необхідно додати до списку після останнього елементу. Фактично адресу нового елемента необхідно записати у поле $next$ останнього елементу списку. Для доступу до останнього елементу скористаємось допоміжним вказівником $Current$:

```
List *Current=(*first);
//Поки не досягнуто кінець списку
while (Current->next!=NULL)
Current=Current->next;
```

5. присвоїти покажчику на останній елемент $last$ покажчик на новий елемент $NewItem$:

```
Current->next = NewItem;
```



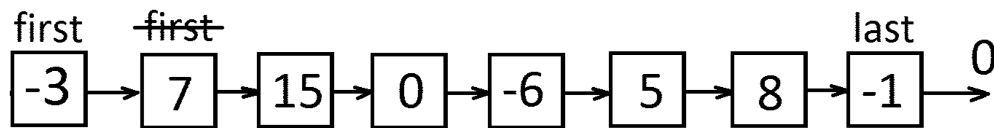
Для вставки нового елементу на першу позицію у списку необхідно після виконання пунктів 1 та 2 виконати наступні дії:

3. новий елемент додати до списку - в його поле $next$ записати адресу першого елементу:

```
NewItem->next=(*first);
```

4. покажчикові $first$ присвоїти покажчик на новий елемент:

```
*first)=NewItem;
```



Для додавання нового елементу до списку після елементу, заданого порядковим номером $Number$, необхідно виконати наступні дії:

1. виділити місце в пам'яті під новий елемент $NewItem$ та записати до нього дані ($Item$):

```
List *NewItem=new List;
NewItem->Data=Item;
```

2. перейти у списку до елементу із номером $Number$. Для переходу скористаємось допоміжним вказівником $Current$:

```
List *Current=(*first);
for(int i=1; i < Number ; i++)
Current=Current->next;
```

3. пов'язати новий елемент із елементом, що слідує після елемента із номером $Number$: до поля $next$ елементу, що вставляється, записати адресу елемента $Number + 1$:

```
NewItem->next = Current->next;
```

4. пов'язати новий елемент із елементом номером $Number$: до поля $next$ елементу $Number$ записати адресу нового елементу :

```
Current->next = NewItem;
```

5. якщо елемент списку $Number$ був останнім, то тепер останнім буде новий елемент :

```
if (Current->next != NULL)
Current->next = NewItem;
```

В цілому функція $Insert_Element$ вставки елементу в будь-яку позицію в межах списку приведена нижче. Тут реалізовано наступні особливості вставки елемента у список:

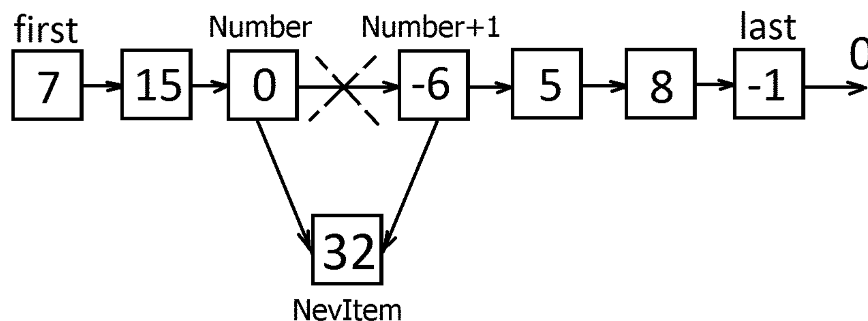


Рис. 1.36: Вставка нового елементу до списку

- якщо список порожній, то елемент, що вставляється, стане першим елементом списку;
- якщо в списку елементів менше, ніж $Number$, то елемент, що вставляється, стане останнім елементом списку;
- для вставки елемента на першу позицію використано функцію $Add_Element_0$

```
//Функція вставки елемента із заданим значенням Item
//у задану позицію Number
void Insert_Element(List** first, int Number, int Item)
{
  Number--;
  List *NewItem=new List;
  NewItem->Data=Item;
  NewItem->next = NULL;
  //Якщо список порожній
  if ((*first) == NULL)
  {
    //Введений елемент стає першим у списку
    (*first) = NewItem;
  }
  //Якщо список не порожній
  else {
    List *Current=(*first);
    for(int i=1; i < Number && Current->next!=NULL; i++)
      Current=Current->next;
    if (Number == 0){
      //Вставляємо елемент на першу позицію
```

```
          //з використанням додаткової функції
          Add_Element_0(&(*first), Item);
        }
        //Вставляємо елемент на задану позицію
        else {
          if (Current->next != NULL)
            NewItem->next = Current->next;
          Current->next = NewItem;
        }
      }
    }
```

```
    //Функція створення першого елементу списку
    void Add_Element_0(List **first, int Item)
    {
      // Створюємо новий елемент списку
     List *NewItem = new List;
     NewItem->Data=Item;
     NewItem->next=(*first);
      // Першим стає введений елемент
     (*first)=NewItem;
    }
```

Для видалення першого елементу списку його адресу необхідно запам'ятати в допоміжному покажчику $Current$, а покажчик $first$ перемістити на другий елемент. Після цього можна видалити елемент $Current$ із пам'яті:

```
    List *Current = (*first);
    (*first) = (*first)->next;
    delete(Current);
```

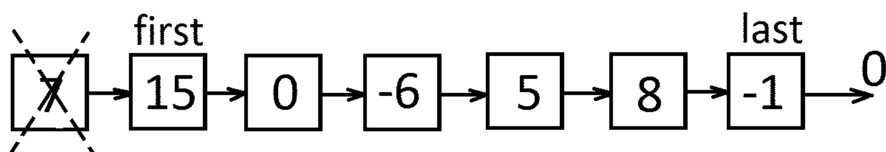


Рис. 1.37: Видалення першого елементу списку

Для видалення елементу c із номером $Number$ зі списку необхідно спочатку знайти покажчик на попередній елемент c_1 ($Number - 1$) та виконати наступні операції:

1. пов'язати елемент c_1 із розташованим після c елементом. Для цього до поля *next* елементу c_1 необхідно записати адресу елементу $c->next$: c_1->$next$ = c->$next$;
2. якщо елемент c був останнім, то тепер останнім буде c_1: $if(c == last)\ last = c_1$;
3. якщо елемент c був першим, то тепер першим стане наступний після c елемент: $if(c == first)\ current = first->next$; $first = current$;
4. видалити елемент *c* із пам'яті: *delete* c;

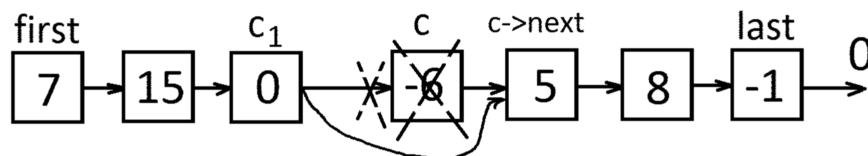


Рис. 1.38: Видалення елементу c зі списку

Описаний вище алгоритм видалення елементу можна реалізувати у вигляді наступної функції:

```
//Функція видалення елементу із однонаправленого списку
void Delete_Element_1(List** first, int Number)
{
  //Допоміжний вказівник
  List *ptr;
  List *Current = (*first);
  for (int i = 1; i < Number && Current != NULL; i++)
    Current = Current->next;
      {
    //Видаляємо перший елемент
    if (Current ==(*first))
    {
      (*first) = (*first)->next;
      delete(Current);
      Current = (*first);
    }
```

```
        //Видаляємо елемент, що не стоїть на першій позиції
        else {
          ptr = (*first);
          while (ptr->next != Current)
            ptr = ptr->next;
          ptr->next = Current->next;
          delete(Current);
          Current=ptr;
        }
      }
    }
```

Видалення всього списку із пам'яті можна реалізувати у вигляді наступної рекурентної функції:

```
    void Delete_List(List* first)
    {
      if (first != NULL){
          Delete_List(first->next);
          delete first;
      }
    }
```

Важливими частинними випадками лінійних списків, в яких операції додавання та вилучення елементів можуть бути виконані лише для першої чи останньої позиції списку, є:

- **стек**
  лінійний список, в якому операції додавання та видалення виконуються лише на одному кінці списку (вершині стеку) ;
- **черга (однобічна черга)**
  лінійний список, в якому усі операції додавання виконується на одному кінці (в голові черги), а операції видалення — на іншому кінці (в хвості черги).
- **дек або двобічна черга**
  лінійний список, в якому всі операції вставки та видалення виконуються на обох кінцях списку.

1.4.2 Різновиди списків

За кількістю зв'язків між елементами списку розрізняюьт :

- однозв'язні списки;
- двозв'язні списки.

За просторовою топологією списки можуть бути:

- лінійні;
- циклічні.

Зв'язані списки мають ряд переваг порівняно з масивами. Зокрема, в них набагато ефективніше (за час O(1), тобто незалежно від кількості елементів) виконуються процедури додавання та вилучення елементів. Натомість, масиви набагато кращі в операціях, які потребують безпосереднього доступу до кожного елементу, що у випадку зі зв'язаними списками неможливо та потребує послідовного перебору усіх елементів, які передують даному.

Найпростішим є лінійний одноз'язний (однонапрямлеий) список, у якому кожний попередній елемент посилається тільки на наступний. У такому списку можна рухатись лінійно тільки в одному напрямі: від першого до останнього елементу. Іншими словами зв'яки між елементами мають чітку однонапрямлену структуру.

В однобічно зв'язаному списку, який є найпростішим різновидом зв'язаних списків, кожний елемент складається з двох полів: $data$ або даних, та вказівника $next$ на наступний елемент. Якщо вказівник не вказує на інший елемент (інакше: $next = NULL$), то вважається, що даний елемент — останній в списку.

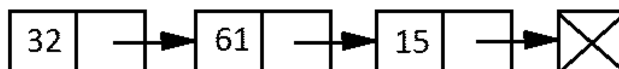


Рис. 1.39: Однобічно зв'язаний список з трьох елементів

В двобічно зв'язаному списку елемент складається з трьох полів — вказівника на попередній елемент $prev$, поля даних $data$ та вказівника $next$ на наступний елемент. Якщо $prev = NULL$, то в елемента немає попередника (тобто він є «головою» списку), якщо $next = NULL$, то в нього немає наступника («хвіст» списка).

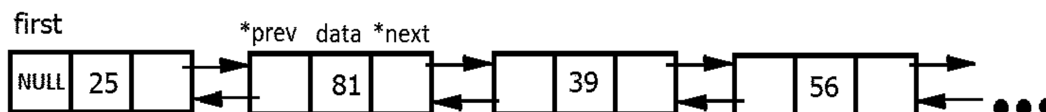


Рис. 1.40: Двобічно зв'язаний список

Лінійний двозвязний список

При необхідності проходити список в обох напрямах та мати інформацію як про наступний, так і про попередній елемент використовують двозв'язні списки. Для його організації до елементу необхідно додати ще одне поле, яке буде містити адресу попереднього елементу: кожний елемент буде мати два зв'язки із найближчими сусідами.

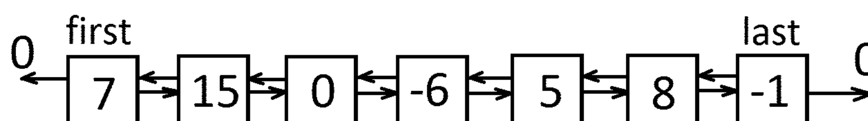


Рис. 1.41: Організація двонапрямленого списку

Такий список називається лінійним двозв'язним, бо зв'язки мають два напрямки: від першого елемента до останнього і від останнього до першого. Кожен елемент має три поля: поле даних та два додаткових поля із адресами попереднього та наступного елементів списку. Це дозволяє проходити список як у прямому, так і зворотньому напрямах.

Приклад оголошення елементу лінійного двозв'язного списку за допомогою структури матиме наступний вигляд:

```
struct Double_List
{
 //Поле даних.
 int Data;
 //Вказівник на наступний елемент.
  Double_List *next;
  //Вказівник на попередній елемент.
  Double_List *prev;
};
```

Для попередньо створеного порожнього списку:

```
//Список порожній.
Double_List *first=NULL;
Double_List *prev=NULL;
```

функції, що створює перший елемент двозв'язного списку матиме наступний вигляд:

```
    void Make_List_0(Double_List** first)
{
  //Виділення памяті під новий елемент списку.
  (*first) = new Double_List();
  //Ввід значення в інформаційне поле.
  cout << endl<<"Введіть значення елементу"<<endl;
  cin >> (*first)->Data;
  (*first)->prev = NULL;
  //Більше елементів немає.
  (*first)->next=NULL;
}
```

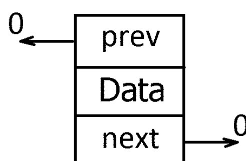


Рис. 1.42: Створений двонапрямлений список, що складається із одного елемента

Двозв'язний список, що складається із деякої кількості елементів, можна реалізувати за допомогою наступної рекурсивної функції

```
void Make_List(int Num,Double_List** first,Double_List* prev)
{
  if (Num > 0) {
    //Виділення пам'яті під новий елемент.
        (*first) = new Double_List();
    cout <<endl<< "Введіть значення елементу"<<endl;
    //Ввід значення в інформаційне поле
        cin >> (*first)->Data;
    (*first)->prev = prev;
    (*first)->next=NULL;
    Make_List(Num -1,&((*first)->next),(*first));
```

```
    }
    else (*first) = NULL;
  }
```

Для додавання нового вузла на початок списку необхідно:

- виділити під елемент місце в пам'яті;

```
    List *NewItem=new List;
```

- встановити вказівник *next* елементу *NewItem* на голову існуючого списку, а вказівник $prev = NULL$ (перший елемент списку).

```
NewItem->prev=NULL;
NewItem->next = (*first);
```

- встановити вказівник *prev* вузла, що був першим, на елемент *NewItem*:

```
(*first)->prev = NewItem;
```

- встановити голову списку на новий вузол:

```
(*first) = NewItem;
```

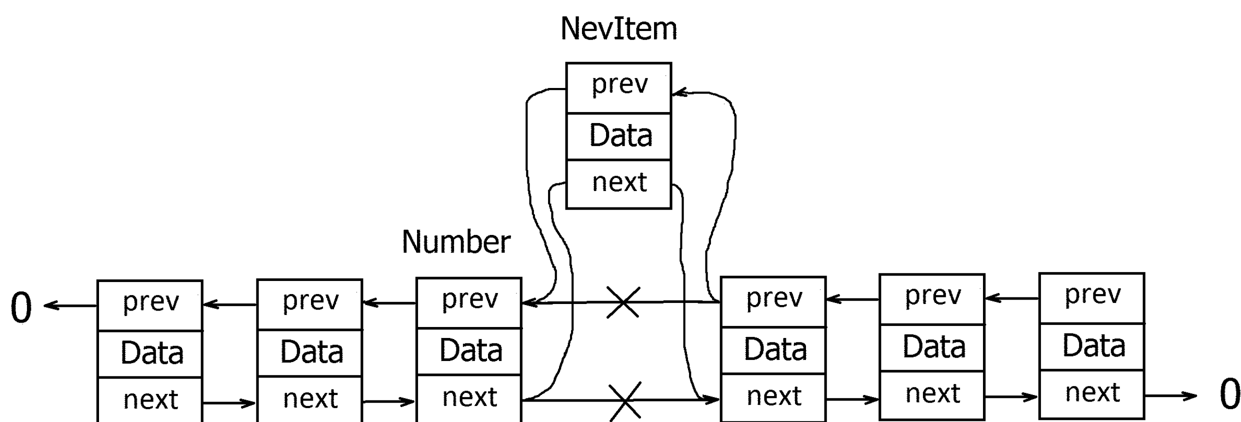


Рис. 1.43: Алгоритм додавання нового елемента в середину двонапрямленого списку

Зважаючи на те, що двозвязний список є симетричним, алгоритм додавання нового вузла в його кінець є аналогічним до алгоритму його вставки на початок списку.

Для додавання елементу *NewItem* в середину списку (задано його порядковий номер *Number*) необхідно виконати наступні операції:

- виділити під елемент місце в пам'яті;

```
List *NewItem=new List;
```

- перейти до елементу із номером $NewItem$, скориставшись допоміжним вказівником $Current$:

```
Double_List *Current=(*first);
for(int i=1; i < Number; i++)
Current=Current->next;
```

- встановити вказівник $next$ елемента $NewItem$ на наступний за номером $Number$ елемент. Вказівник $prev$ встановити на попередній елемент:

```
NewItem->next = Current->next;
NewItem->prev = Current;
```

- встановити вказівники елементів зліва та справа від місця вставки так, щоби включити елемент $NewItem$ в список:

```
Current->next->prev = NewItem;
Current->next = NewItem;
Current = NewItem;
```

Операції додавання елементу в кінець двонапрямленого списку, додання елементу до списку перед першим елементом та вставки нового елемента на задану позицію $Number$ можна реалізувати у вигляді наступної функції:

```
void Insert_Item(Double_List** first, int Number, int Item)
{
  Number--;
  Double_List *NewItem=new(Double_List);
  NewItem->Data=Item;
  NewItem->prev=NULL;
  NewItem->next = NULL;
    //Якщо список порожній
  if (first == NULL) {
    (*first) = NewItem;
  }
    //Якщо список не порожній
  else {
    Double_List *Current=(*first);
```

```
        for(int i=1; i < Number && Current->next!=NULL; i++)
          Current=Current->next;
        if (Number == 0){
          //Вставка елементу на нову позицію
          NewItem->next = (*first);
          (*first)->prev = NewItem;
          (*first) = NewItem;
        }
            //Вставка елементу на деяку позицію, відмінну від
першої.
        else {
          if (Current->next != NULL) Current->next->prev = NewItem
;
          NewItem->next = Current->next;
          Current->next = NewItem;
          NewItem->prev = Current;
          Current = NewItem;
        }
      }
    }
```

Для видалення вузла достатньо поміняти значення відповідних адресних полів. Операція видалення елементу із двонаправленого списку фактично аналогічна видаленню елемента із однонаправленого списку: на першому етапі необхідно знайти вказівник на елемент із заданим номером $Number$ та вказівники для сусідніх вузлів (справа та зліва відносно елементу із номером $Number$) задати так, якби елемент, що видаляється, був відсутній. На наступному кроці необхідно звільнити пам'ять, яку займає видалений елемент. Додатково необхідно перевірити, чи не є елемент, що видаляється, першим чи останнім у списку.

Операції видалення елементу із початку, кінця двонапрямленого списку чи довільного елементу, що знаходиться на позиції $Number$ можна реалізувати у вигляді наступної функції:

```
    void Delete_Item(Double_List** first, int Number)
    {
        //Допоміжний вказівник
      Double_List *ptr;
     Double_List *Current = (*first);
     for (int i = 1; i < Number && Current != NULL; i++)
```

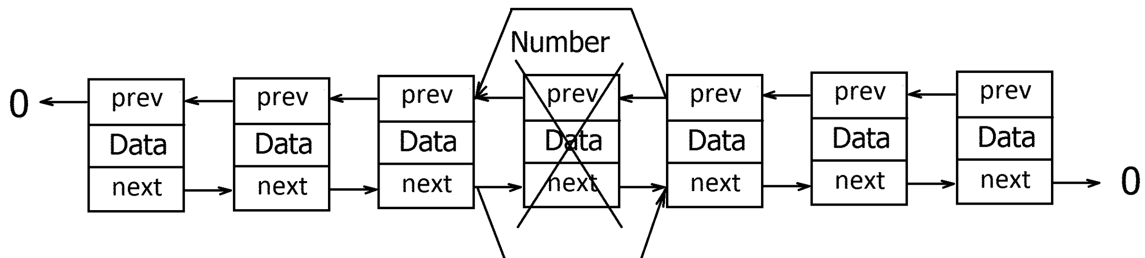


Рис. 1.44: Алгоритм видалення елементу із двонапрямленого списку

```
          Current = Current->next;
        //Перевірка, чи список не є порожнім
      if (Current != NULL)
            {
            //Виділення першого елементу
        if (Current->prev == NULL)
            {
            (*first) = (*first)->next;
            delete(Current);
            (*first)->prev = NULL;
            Current = (*first);
        }
            //Видалення елементу (За виключенням першого)
          else
                {
                //Видалення останнього елементу списку
            if (Current->next == NULL)
                {
            Current->prev->next = NULL;
            delete(Current);
            Current = (*first);
            }
                //Видалення елементу із середини списку
            else
                {
            ptr = Current->next;
            Current->prev->next =Current->next;
            Current->next->prev =Current->prev;
              delete(Current);
```

```
              Current = ptr;
            }
          }
        }
     }
```

Приклади функцій обробки елементів двозв'язного списку:

- перевірка, чи є список порожнім:

```
    bool Empty_List(Double_List* first){
  if (first!=NULL)  return false;
  else  return true;
}
```

- видалення списку(рекурсивне):

```
    void Delete_List(Double_List* first)
    {
  if (first != NULL){
    Delete_Double_List(first->next);
    delete first;
  }
}
```

- пошук останнього елементу списку:

```
    Double_List* Find_End(Double_List* fitst)
  {
  Double_List *ptr;
  ptr = fitst;
  while (ptr->next != NULL){
    ptr = ptr->next;
  }
  return ptr;
}
```

- вивід елементів списку:

```
    void Show_List(Double_List *first)
    {
      cout<<endl<<"Лінійний двозвязний список"<<endl;
      do
      {
```

```
      cout<<first->Data<<"  ";
          //Перехід до наступного елемента.
      first = first->next;
    }
    //Поки не досягнуто кінець списку
    while (first!=NULL);
    cout<<endl;
  }
```

- вивід елементів списку у зворотньому порядку:

```
  void Show_Revers_List(Double_List *first)
  {
  Double_List *ptr;
  ptr = Find_End(first);
    cout<<endl<<"Лінійний двозв'язний список (реверс)"<<endl;
  do
    {
      cout<<ptr->Data<<"  ";
    ptr = ptr->prev;
      }
    while (ptr!=NULL);
    cout<<endl;
  }
```

1.4.3 Контейнер *list*

Контейнер *List* являє собою двозв'язний список, що зберігається в довільних областях пам'яті. Для використання контейнера необхідно підключити однойменний заголовний файл:

```
#include <list>
#include <algorithm>
```

Табл. 1.13: Базові функції для обробки контейнера *list*

Операція	**Призначення**
sp.sort()	Сортування списку *sp*
sp.sort(Pred)	Сортування списку *sp* з використанням критерію сортування , заданого бінарним предикатом *Pred*
sp.push_front(t)	Додавання елемента *t* у початок списку *sp*
sp.push_back(t)	Додавання елемента *t* у кінець списку *sp*
sp.pop_front()	Видалення першого елемента списку *sp*
sp.pop_back()	Видалення останнього елемента списку *sp*
sp.remove(t)	Видалення всіх елементів t зі списку *sp*
sp.clear()	Видалення всіх елементів зі списку *sp*
sp.unique()	Видалення рядом розташованих дублікатів зі списку *sp*
sp.unique(BinPred)	Видалення рядом розташованих дублікатів зі списку *sp* з використанням бінарного предикату *BinPred*
sp.merge(sp2)	Об'єднання двох списків. Якщо вони відсортовані, то впорядкованість зберігається
sp.reverse()	Запис елементів списку *sp* у зворотному порядку
sp.size()	Кількість елементів списку *sp*
sp.max_size()	Максимально можлива кількість елементів списку *sp*
sp.empty()	Перевірка, чи є список *sp*порожнім
sp.remove_if(Pred)	Видалення всіх елементів зі списку *sp*, що задовільняють предикату *Pred*

Для двох списків визначеними є також стандартні операції порівняння ($<$, $<=$, $>$, $>=$, $==$, $!=$)

Поняття предиката Деякі із алгоритмів та контейнерів використовують спеціальний тип функцій, які називаються предикатами. Ці функції повертають результатом або *true*, або *false*, а сама функція визначається розробником. Існують два варіанти предикатів: унарні (приймають один аргумент) та бінарні (приймають два аргументи).

Табл. 1.14: Ітератори для доступу до елементів контейнера *list*

Ітератор	**Призначення**
begin()	Ітератор першого елементу списку
rbegin()	Зворотній ітератор першого елементу списку
end()	Ітератор кінця списку
emphrend()	Зворотній ітератор кінця списку
erase(i)	Видалення елементу, на який вказує ітератор i
insert(i,elem)	Вставляє новий елемент, рівний *elem* перед елементом, заданим ітератором i

Деякі приклади використання контейнера *list* Створення порожнього списку *List* та опис ітератора *iter*:

```
list<int>List;
list<int>::iterator iter;
```

Заповнення списку

```
const int SIZE = 10;
  // Функція push_back().
  for (i = 0; i < SIZE; i++) List.push_back(i);
  // Функція push_front()
  for (i = 0; i < SIZE; i++) List.push_front(rand()%100);
```

Вивід вмісту контейнера, що складається із цілих чисел, за допомогою прямого ітератора *iteratorit*:

```
void print(list<int> List)
{
  for (list<int>::iterator it = List.begin(); it != List.end()
; it++)
    cout<<*it<<" ";
  cout<<endl;
}
```

Вивід вмісту контейнера цілих чисел у зворотному порядку, за допомогою ітератора *reverse_iteratorit*:

```
 void print_revers(list<int> List)
{
```

```
    for (list<int>::reverse_iterator it = List.rbegin(); it !=
 List.rend(); it++)
      cout<<*it<<" ";
    cout<<endl;
  }
```

Пошук та видалення елемента із заданим значенням:

```
  iter = find(List.begin(),List.end(),69);
if( iter != List.end()) List.erase(iter);
else printf("Такого числа в списку нема");
```

Можливості модифікації створеного списку:

```
  // Видалення елементу із початку списку
if(!List.empty())List.pop_front();
// Видалення елементу із кінці
if(!List.empty())List.pop_back();
printf("Після видалення числа з кінця списку\n");
// Вивід максимального розміру списку
int ListMaxSize = List.max_size();
printf("Максимальний розмір списку = %d\n", ListMaxSize);
// Визначення поточного розміру списку
int ListSize = List.size();
// Вставка на початок списку 7 одиничок
List.insert(List.begin(),7, 1);
// Видалення останнього елементу
List.pop_back();
// Створення новогоnewList контейнера та копіювання в нього
 вмісту списку List
list<int>newList(List.begin(),List.end());
// Заповнення списку  newList нулями
newList.insert(newList.begin(),newList.size(),0);
// Допис в список newList уміст списку List
newList.assign(List.begin(),List.end());
// Видалення списку newList
   newList.clear();
  // Видалення дублікатів
   newList.unique();
// Злиття списків
newList.merge(List);
```

```
// Упорядкування списку
newList.sort();

// Функція splice()
  newList.splice(newList.begin(),List);
printf("Після застосування функції splice\n");

// Порівняння списків
if(List == newList) printf("List == newList\n");
if(List > newList) printf("List > newList\n");
if(List >= newList) printf("List >= newList\n");
if(List < newList) printf("List < newList\n");
if(List <= newList) printf("List <= newList\n");
```

Поняття предиката Деякі із алгоритмів та контейнерів використовують спеціальний тип функцій, які називаються предикатами. Ці функції повертають результатом або *true*, або *false*, а сама функція визначається розробником. Існують два варіанти предикатів: унарні (приймають один аргумент) та бінарні (приймають два аргументи). Предикати використовуються в алгоритмах сортування, пошуку. В залежності від виконання необхідної умови (наприклад сортування) вона повертає або істину, або хибу: результат порівняння двох елементів за певною ознакою. Наприклад, при його використання в функції сортування *sort* алгоритм міняє порядок елементів контейнера *list* таким чином, щоби для будь-яких двох елементів вини задовільняли умові предикату.

Приклад 4. 11. *Програма створює список, елементами якого є структура student, виводить його на екран аналізує його із використання предикатів.*

```
  #include <windows.h>
  #include <iostream>
  #include <list>
  using namespace std;
  double ser_bal;
  int ocinka;

  struct student
  {
```

```
    char name[15];
    int oc[4];
  };

  // Предикат для сортуванням за середнім балом студента (за
зростанням)
  bool Sort(student a, student b)
  {
    return (a.oc[0]+a.oc[1]+a.oc[2]+a.oc[3])/4.0 < (b.oc[0]+b.oc
[1]+b.oc[2]+b.oc[3])/4.0;
  }
  ;

  // Предикат для сортуванням за середнім балом студента (за
спаданням)
  bool Sort_revers(student a, student b)
  {
    return (a.oc[0]+a.oc[1]+a.oc[2]+a.oc[3])/4.0 > (b.oc[0]+b.oc
[1]+b.oc[2]+b.oc[3])/4.0;
  }
  ;
  // Предикат для сортуванням за прізвищем студента
  bool sort_name (student a, student b)
  {
    return strcmp(a.name,b.name)<0;
  }

  // Предикат для відбору студентів за середнім балом
  bool selest_sb(student a) {
    return  ((a.oc[0]+a.oc[1]+a.oc[2]+a.oc[3])/4.0 < ser_bal);
  }

  // Предикат для відбору студентів за оцінкою із першого
екзамену
  bool bal(student a )
  {
    return (a.oc[0]!=ocinka);
  }

  // Функція виводу списку на екран (із середнім балом)
```

```
void print1(list<student> List)
{
  if (List.size()==0) cout<<"Список порожній";
  else  for (list<student>::iterator it = List.begin(); it !=
List.end(); it++)
    cout<<it->name<<" ("<<(it->oc[0]+it->oc[1]+it->oc[2]+it->
oc[3])/4.0<<")  ";
  cout<<endl;
}

// Функція виводу списку студентів на екран
void print2(list<student> List)
{
  for (list<student>::iterator it = List.begin(); it != List.
end(); it++)
    cout<<it->name<<" ";
  cout<<endl;
}

//Вивід списку на екран у зворотному порядку
void print_revers(list<student> List)
{
  list<student> ::reverse_iterator it = List.rbegin();
  for (it; it != List.rend(); it++)
    cout<<it->name<<" ("<<(it->oc[0])<<")  ";
  cout<<endl;
}

int main ()
{
  SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
  student a1={"Михайлів",4,5,5,4};
  student a2={"Федорів",3,4,3,3};
  student a3={"Кравець",3,2,4,3};
  student a4={"Василів",4,4,5,3};
  student a5={"Антонів",3,4,5,2};
  student a6={"Вацеба",2,4,3,2};
  list<student> stud;
```

```
    stud.push_front(a1);
    stud.push_front(a2);
    stud.push_front(a3);
    stud.push_front(a4);
    stud.push_front(a5);
    stud.push_front(a6);
    cout<<"Список студентів по алфавіту"<<endl;
    stud.sort(sort_name);
    print2(stud);
    cout<<endl<<"Список студентів за зростанням середнього
балу"<<endl;
    stud.sort(Sort);
    print1(stud);
    cout<<endl<<"Список студентів за спаданням середнього
балу"<<endl;
    stud.sort(Sort_revers);
    print1(stud);
    cout<<"Загальна кількість студентів=  "<<stud.size()<<endl;
    cout<<endl<<"Введіть середній бал, за яким відібрати
студентів"<<endl;
    cin>>ser_bal;
    cout<<endl<<"Список відібраних за середнім балом студентів
"<<endl;
    list<student> stud1;
    stud1=stud;
    stud1.remove_if(selest_sb);
    print1(stud1);
    cout<<"Кількість відібраних студентів=
"<<stud1.size()<<endl;
   cout<<endl<<"Введіть оцінку першого екзамена, за якою
відібрати студентів"<<endl;
   cin>>ocinka;
   stud1=stud;
   stud1.remove_if(bal);
   cout<<endl<<"Відібрані за першим екзаменом студенти"<<endl;
   print_revers(stud1);
   cout<<"Кількість відібраних студентів=
"<<stud1.size()<<endl;
  return 0;
  }
```

```
Список студентів по алфавіту
Антонів Василів Вацеба Кравець Михайлів Федорів

Список студентів за зростанням середнього балу
Вацеба (2.75 )  Кравець (3 )  Федорів (3.25 )  Антонів (3.5 )  Василів (4 )  Михайлів (4.5 )

Список студентів за спаданням середнього балу
Михайлів (4.5 )  Василів (4 )  Антонів (3.5 )  Федорів (3.25 )  Кравець (3 )  Вацеба (2.75 )
Загальна кількість студентів=   6

Введіть середній бал, за яким відібрати студентів
3.4

Список відібраних за середнім балом студентів
Михайлів (4.5 )  Василів (4 )  Антонів (3.5 )
Кількість відібраних студентів=   3

Введіть оцінку першого екзамена, за якою відібрати студентів
3

Відібрані за першим екзаменом студенти
Кравець (3 )  Федорів (3 )  Антонів (3 )
Кількість відібраних студентів=   3
Press any key to continue . . .
```

Рис. 1.45: Результат роботи програми

Циклічний однозв'язний список

У циклічному однозв'язному списку за останнім елементом іде перший елемент. Це означає, що у полі *next* останнього елемента записано адресу першого елементу: $last- > next = first$;. Схематично такий список можна зобразити наступним чином:

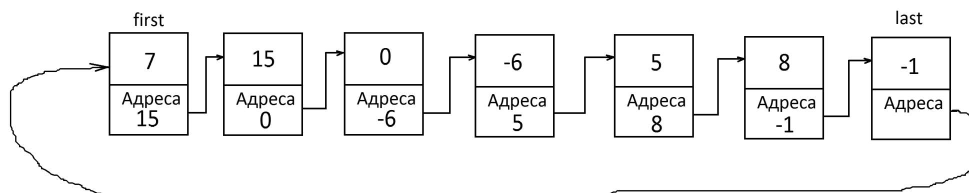


Рис. 1.46: Циклічний однозв'язний список

Оголошення циклічного однозв'язного списку аналогічне оголошенню лінійного однозв'язного списку. Функція створювання першого елементу списку:

```
    void fir(int x)
        {
            first=new Element;
```

```
            first->d=x;
            first->next=first;
            last=first;
}
```

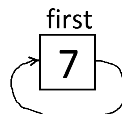


Рис. 1.47: Створення першого елементу циклічного однонаправленого списку

Функція додавання нового елементу до списку між останнім і першим:

```
    void add_el(int x)
        {
            Element* c=new Element;
            c->d=x;
            c->next=first;
            last->next=c;
            last=c;
        }
```

Функція вставки нового елемента після елементу c_1:

```
    void insert_el(Element* c1, int x)
        {
            Element* c=new Element;
            c->d=x;
            c->next=c1->next;
            c1->next=c;
                if (c1==last) last=c;
        }
```

Функція видалення зі списку елементу, який слідує після c_1:

```
    void del_el(Element* c1)
        {
            Element* c=c1->next;
            c1->next=c->next;
            if(c==first) // Якщо вилучається перший елемент,
            first=c->next; // першим стає другий.
            if(c==last) // Якщо вилучається останній елемент,
```

```
            last=c1; // останнім стає передостанній.
            delete c;
}
```

Функція видалення списку із пам'яті :

```
    void ochistka()
        {
            Element* c;
            last->next=0;
        while(first!=0)
            {
                c=first;
                first=first->next;
                delete c;
            }
        }
```

Циклічний двозвязний список

Циклічний двозв'язний (двонапрямлений) список – це циклічний список, кожний елемент якого містить покажчики як на наступний так і попередній елементи. Схематично його можна представити наступним чином:

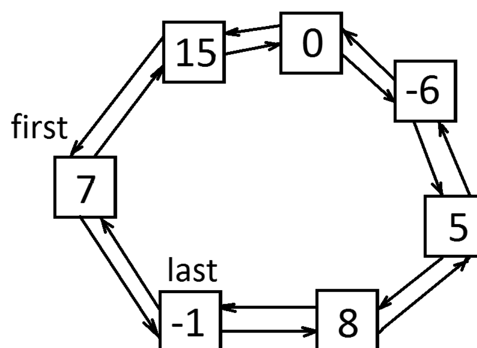


Рис. 1.48: Циклічний двозв'язний список

Функція створення першого елемента:

```
    void fir(int x)
```

```
    {
        first=new Element;
        first->d=x;
        first->next=first;
        first->prev=first;
        last=first;
    }
```

Функція вставки нового елемента до списку (поміж останнім і першим):

```
void add_el(int x)
    {
         Element* c=new Element;
        c->d=x;
        c->next=first;
        c->prev=last;
        last->next=c;
        first->prev=c;
        last=c;
    }
```

Функція вставки нового елемента після елементу c_1:

```
void insert_el(Element* c1, int x)
    {
         Element* c=new Element;
        c->d=x;
        c->next=c1->next;
        c1->next->prev=c;
        c1->next=c;
        c->prev=c1;
            if(c1==last) last=c;
    }
```

Функція видалення зі списку елемента, який слідує після c_1:

```
void del_el(Element* c1)
    {
         Element* c=c1->next;
```

```
            c1->next=c->next;
            c->next->prev=c1;
                if(c==first) first=c->next;
                if(c==last) last=c1;
            delete c;
        }
```

Функція видалення списку із пам'яті:

```
    void ochistka()
        {
            Element* c;
            last->next=0;
            first->prev=0;
            while(first!=0)
                {
                     c=first;
                     first=first->next;
                    delete c;
                }
            last=0;
        }
```

1.4.4 Стек

Стек - різновид лінійного списку, структура даних, яка працює за принципом "останнім прийшов - першим пішов"(LIFO, last in, first out). Всі операції в стеку можна виконувати тільки із одним елементом, який знаходиться на його верхівці та був введений в останнім.

Стек можна розглядати як певну аналогію до стопки книжок, з якої можна взяти тільки верхню, і на яку можна покласти книжку тільки зверху. Інша назва стеку - "магазин за аналогією з принципом роботи магазину в автоматичній зброї. Його можна зобразити наступним чином:

До елементів стеку визначено наступні базові операції:

- **push ("заштовхнути елемент")**
  додати елемент в стек, розмістивши його на вершині (*root*). Розмір стеку

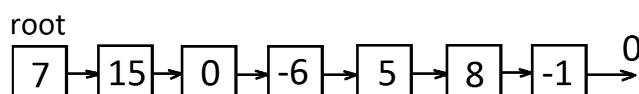


Рис. 1.49: Схематичне зображення стеку

збільшується на одиницю. При перевищенні граничного розміру стеку відбувається його переповнення ($stackoverflow$):

- **pop ("виштовхнути елемент")**
  отримати елемент із верхівки стеку. Елемент видаляється зі стеку, а його місце в верхівці стеку займає наступний за ним відповідно до правила $LIFO$ Розмір стеку зменшується на одиницю. При намаганні "виштовхнути"елемент із вже пустого стеку відбувається ситуація "незаповнення"стеку ($stackunderflow$).

В пам'яті комп'ютера стек може бути організований як масив або множина комірок в певній області оперативної пам'яті з додатковим зберіганням і вказівника на верхівку стека. Заштовхування першого елемента в стек збільшує адресу вказівника, виштовхування елементу зменшує її. Таким чином, адреса вказівника завжди відповідає комірці масиву, в якій на даний момент знаходиться верхівка стеку.

Взагалі стек можна реалізувати за допомогою будь-якої структури, яка дозволяє зберігати декілька елементів.

Для роботи зі стеком необхідно пам'ятати покажчик на вершину стеку ($root$). Кожен елемент повинен також містити адресу наступного елемента: елементу, який знаходиться нижче нього та було додано у стек раніше за нього. Елемент стека, як і списку, є структурою, яка містить одне (чи більше) поле даних та поле адреси попереднього елемента (покажчик на елемент). Тип елемента стеку оголошується аналогічно до оголошення елементу списку, Приклад оголошення елементу стеку, що складається із даних цілого типу:

```
    struct Elem
        {
            int i;          // поле даних ціле- число
            Elem *next  ; //вказівник на наступний елемент
        };
```

Початкове оголошення вершини стеку:

```
Elem *root=0;
```

У стек на його схематичному представленні вище елемент зі значенням -1 було додано першим, а видалений він може бути тільки після того, як будуть видалі усі попередні елементи . Елемент зі значенням 7 було записано у стек останнім, тому видаленим він може бути тільки першим. Новий елемент до стеку може бути доданий тільки ліворуч) від 7.

При записі нового елементу у стек поле $next$ цього елементу буде містити адресу попереднього елемента - елементу зі значенням 7. Після запису вершиною стеку $root$ стане новий елемент. Аналогічно до роботи зі списком для роботи зі стеком використовується проміжний покажчик с.

Кожна з цих операцій зі стеком виконується за фіксований час $O(1)$ і не залежить від розміру самого стеку. Додаткові операції (присутні не у всіх реалізаціях стеку):

- **isEmpty**:
  перевірка наявності елементів в стеку: результатом є істина ($true$), якщо елементів у стеку нема;
- **isFull**:
  перевірка заповненості стека. Результат є істина, якщо новий елемент в стек додати неможливо;
- **clear**:
  очистити стек (видалити усі елементи);
- **top**:
  отримати значення верхнього елементу (без виштовхування);
- **size**:
  отримати розмір (кількість елементів) стеку;
- **swap**:
  поміняти місцями два верхніх елементи стеку.

Приклад Добавити!!!

1.4.5 Черга

Черга - динамічна структура даних, що працює за принципом "перший прийшов - перший пішов"(англ. $FIFO—firstin, firstout$). У черги є голова (англ. $head$) та хвіст (англ. $tail$). Елемент, що додається до черги, опиняється в її хвості. Елемент, що видаляється з черги, знаходиться в її голові. Така черга повністю аналогічна звичайній "магазинній"черзі, в якій хто перший встав в

неї, той першим буде обслуженим (але, на відміну від реальної черги, імовірність пройти поза чергою виключена).

Основні операції з чергою:

- **enqueue — "поставити в чергу"**
  операція додавання елемента в "хвіст"черги. При цьому довжина черги збільшується на одиницю. Якщо відбувається намагання додати елемент у вже заповнену чергу, відбувається її переповнення (англ. queue overflow);
- **dequeue — "отримання з черги"**
  операція, яка повертає елемент з голови та видаляє його з черги, таким чином встановлюючи голову на наступний за видаленим елемент та зменшуючи довжину на одиницю. При намаганні видалити елемент з пустої черги, виникає ситуація "незаповнення"(англ. queue underflow):

Черга може бути реалізована за допомогою масива $Q[1..n]$, в якому зберігаються дані та двох додаткових змінних $head[Q]$ та $tail[Q]$, в яких зберігаються індекси відповідно "голови"та "хвоста"черги, $lenght[Q]$ - довжина черги.

Тоді операції *enqueue* та *dequeue* запишуться так:

```
ENQUEUE (Q, x)
 Q[tail[Q]] := x
 if tail[Q] = length[Q]
 then tail[Q] := 1
 else tail[Q] := tail[Q] + 1
```

```
DEQUEUE (Q)
 x := Q[head[Q]]
 if head[Q] = length[Q]
 then head[Q] := 1
 else head[Q] := head[Q] + 1
 return x
```

Черга з пріорітетами (англ. priority queue) — це структура даних, що призначена для обслуговування множини S, з кожним елементом якої пов'язано певне значення, що зветься ключем (англ. key). Черга з пріорітетами може бути неспадною або незростаючою. В незростаючій черзі з пріорітетами підтримуються наступні операції:

1. **Insert(S,x)**
   вставляє елемент x в множину S;

2. **Maximum(S)**
   повертає елемент множини S з найбільшим ключем;
3. **Extract-Max**
   повертає елемент з найбільшим ключем, видаляючи його при цьому з множини S;
4. **Change-Key(S,x,k)**
   змінює значення ключа для елемента x, шляхом заміни його ключем зі значенням k.

Черга з пріорітетами може бути реалізована на різних структурах даних. В залежності від обраної структури змінюється ефективність виконання операцій з чергою. Найбільш часто вживаними є масиви.

1.4.6 Дек

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

Так само можна сформулювати поняття деку, як стек, в якому включення і виключення елементів може здійснюватися з обох кінців.

Деки рідко зустрічаються у своєму первісному визначенні. Окремий випадок деку – дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури деку аналогічні логічній і фізичній структурі кільцевої черги. Проте, стосовно деку доцільно говорити не про голову і хвіст, а про лівий і правий кінець. Над деком доступні наступні операції:

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

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

Розділ 2

Нелінійні структури даних

2.1 Дерева

2.1.1 Основні поняття та визначення

Дерево є найбільш ефективним способом представлення та збереження інформації. Дерево - це ієрархічна структура даних, що складається елементів (вузлів), зв'язаних між собою відношеннями типу "батьківська вершина "вершина нащадка".

Формально деревом називають скінчену множину вузлів T, з'єднаних гілками (ребрами), серед яких є один виділений вузол t, який називають *коренем*, а інші вузли розбиті на $m \geq 0$ множин, що не перетинаються $T_1, T_2, \ldots, T_m$. Кожна така множина T_i, в свою чергу, є деревом та називається **піддеревом** вузла t.

Кількість вузлів u та кількість ребер v зв'язані наступним чином:

$$u = v + 1$$

Корінь дерева t називають предком або батьківським вузлом. Вузли $t_1, t_2, \ldots, t_n$ називають нащадками, або дочірними вузлами. Ступенем вузла (degree) називають кількість нащадків даного вузла. Ступенем дерева називають найбільшу ступінь всіх його вузлів. Вузли нульового ступеня не мають нащадків та називаються листками, чи кінцевими вузлами дерева. Повним називають дерево, у якого ступінь всіх вузлів, що не є листками, рівна ступеню дерева.

Рівнем вузла називають кількість вузлів у шляху, що з'єднує даний вузол із коренем дерева. Висотою дерева називають максимальний рівень для вузлів дерева. Дерево називають впорядкованим, якщо для будь-якого вузла дерева, за виключенням кореневого, відомо, яким по рахунку нащадком є даний вузол.

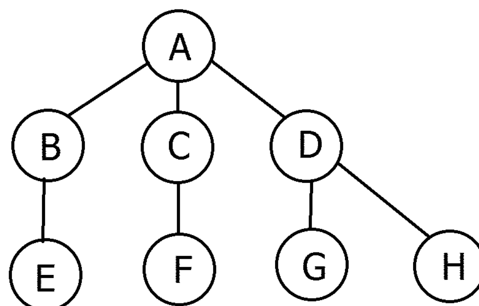


Рис. 2.1: Приклад представлення дерева

Дерево, приведено на рисунку, є деревом третього степеня та висоти 2. Коренем дерева є вершина A. Листками дерева є вершини E, F, G, H. Вузол D є коренем піддерева, що складається із вузлів D, G, H. Рівень кореня (вершина A) рівний 0. Кожний нащадок кореневої вершини (B, C, D) є вузлом першого рівня, а наступні нащадки (E, F, G, H) - вузлами другого рівня.

Дерево є рекурсивною структурою даних, так як кожне піддерево є також деревом. Дії з такими структурами даних простіше всього описувати за допомогою рекурсивних алгоритмів.

Над деревами визначено наступні основні операції:

- пошук вузла із заданим ключем;
- включення вузла у дерево;
- видалення вузла;
- пошук по дереву;
- обхід дерева.

Для розв'язку багатьох задач виникає необхідність обходу дерева - послідовний перегляд всіх вузлів дерева в певному порядку. Розрізняють декілька алгоритмі обходу дерева:

- прямий обхід (pre-order walk), при якому кожен вузол-предок проглядається раніше його нащадків ;
- зворотній обхід (post-order walk), при якому спочатку проглядаються нащадки, а потім предки.
- симетричний обхід , при якому обходиться спочатку ліве піддерево, потім вузол, потім — праве піддерево

2.1.2 Бінарне дерево

Бінарним деревом називають впорядковане дерево другого ступеня. При цьому першого нащадка будь - якого вузла називають лівим піддеревом, о другого - правим піддеревом. Будь-який вузол може не мати нащадків взагалі, мати тільки ліве чи праве піддерево, або мати обидва піддерева. Бінарне дерево також може бути порожнім.

Бінарне дерево називається ідеально збалансованим, якщо для кожного вузла кількість вузлів в його правому піддереві відрізняється від кількості вузлів у його правому піддереві не більше ніж на 1. Повним називають бінарне дерево, у якого кожна вершина, що не є листом, має рівно два нащадки, а всі листки знаходяться на останньому рівні.

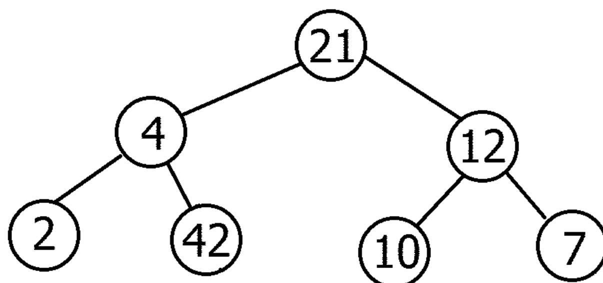


Рис. 2.2: Приклад повного бінарного дерева

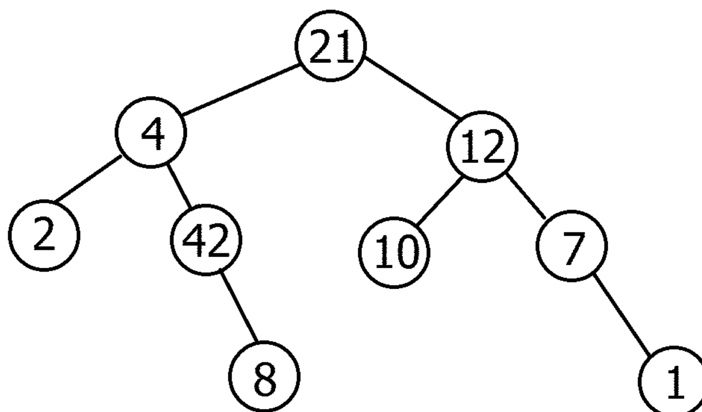


Рис. 2.3: Приклад неповного бінарного дерева

Збереження бінарного дерева можна організувати послідовно у вигляді масиву, чи за допомогою зв'язаних структур. При зв'язаному способі збереження бінарного дерева кожен його вузол являє собою структуру, що має два адресні поля, які використовуються для вказівників на ліве та праве піддерево, відповідно:

```
struct BinaryTree
{
 //Поле даних
 int Data;
 //Вказівник на лівого нащадка
 BinaryTree* Left;
 //Вказівник на правого нащадка
 BinaryTree* Right;
};
```

Для доступу до вузлів дерева використовується вказівник на кореневий елемент:

```
//Дерево поки-що порожнє.
BinaryTree* BTree = NULL;
```

Алгоритм побудови ідеального збалансованого дерева, якщо кількість вершин n відома наперед, можна реалізувати рекурсивно, врахуваваши, що для досягнення мінімальної висоти дерева необхідно, по можливості, розміщувати максимально можливу кількість вершин на всіх рівнях, за виключенням найнижчого. Це просто зробити, якщо поміщати всі вершини, з яких будується дерево, порівно зліва та справа від кожної вершини. Алгоритм можна реалізувати у вигляді наступної рекурсивної процедури:

- взяти першу вершину в якості кореня дерева;
- побудувати ліве піддерево із $n_{Left} = n \ / \ 2$ (ділення націло) вершинами ;
- побудувати праве піддерево із $n_{Right} = n - n_{Left} - 1$ вершинами;

Описаний алгоритм може бути реалізований к вигляді наступної рекурсивної функції:

```
// Побудова ідеально збалансованого дерева,
// що містить n вершин.
// *Tree - вказівник на корінь дерева.
void Tree_Balance(int n,BinaryTree **Tree)
{
```

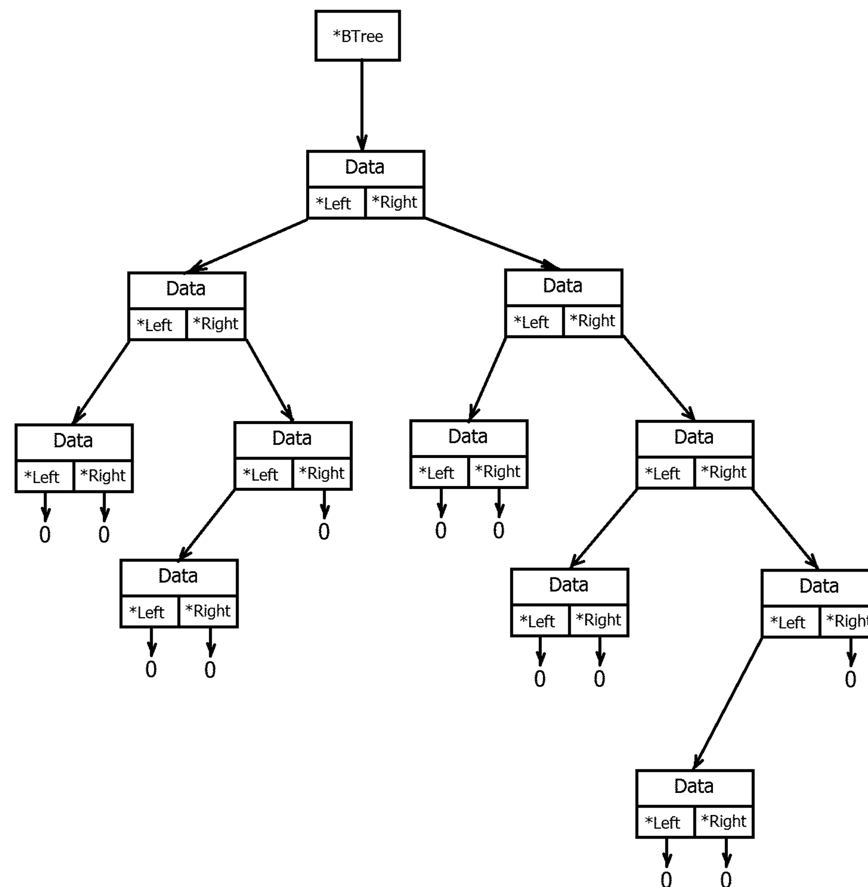


Рис. 2.4: Представлення бінарного дерева у вигляді зв'язаної структури

```
      BinaryTree *now;
      int x,nl,nr;
      now = *Tree;
      if  (n==0) *Tree = NULL;
      else
      {
        nl = n/2; nr = n - nl - 1;
        // Ввід даних для чергового вузла
        cin>>x;
        now = new BinaryTree;
        (*now).Data = x;
        Tree_Balance (nl,&((*now).Left));
        Tree_Balance (nr,&((*now).Right));
        *Tree = now;
```

```
    }
  }
```

Для набору вхідних даних $7, 32, 25, 14, 10, 17, 3, 11, -9, 18, 29, 44, -24, 15$ функцією буде згенероване приведене нижче ідеально збалансоване дерево:

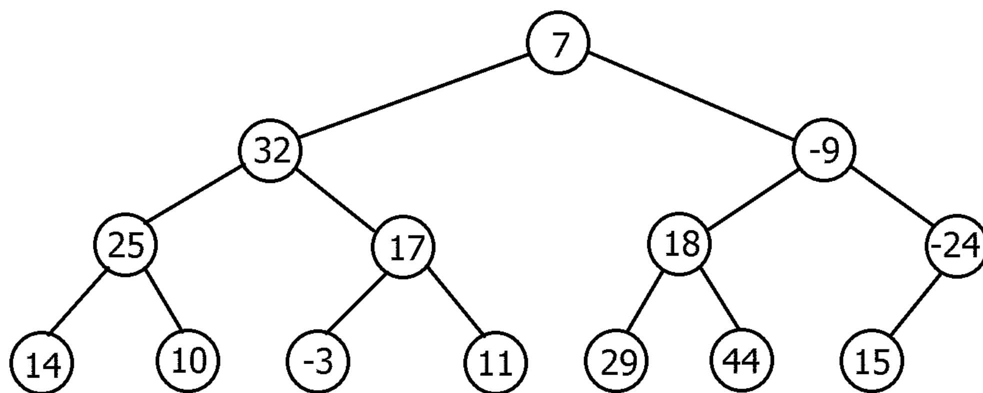


Рис. 2.5: Ідеально збалансоване дерево, сформоване функцією $Tree_Balance$

Вивід бінарного дерева у консольне вікно найпростіше реалізувати з використанням рекурсивної функції. Виведене дерево буде "лежати "на лівому боці. Спочатку виводиться ліве піддерево: при цьому чергова вершина зсувається відносно лівого краю консольного вікна на величину, пропорційну глибині вершини - відстані від кореня дерева до даної вершини. Такий відступ формується за допомогою наступного циклу:

```
for  (int i=1; i<=l; i++) cout<<"   ";
```

Такий зсув реалізовано використанням змінної l , значення якої збільшується на 1 при кожному рекурсивному звертанні до функції. Початкове значення змінної l визначає загальне положення бінарного дерева відносно лівого краю консольного вікна

```
  // Функція виводу бінарного дерева Tree
  // у консольному вікні.
  void Print_Tree(BinaryTree **Tree, int l)
  {
    if  (*Tree!=NULL)
    {
      //Рекурсивна функція виводу правого піддерева
```

```
        Print_Tree(&((**Tree).Right),l+2);
        for  (int i=1; i<=l; i++) cout<<"   ";
        //Вивід кореня дерева
        cout<<(**Tree).Data<<endl;
        //Рекурсивни функція виводу лівого піддерева
        Print_Tree(&((**Tree).Left),l+2);
  }
}
```

```
                -24
                     15
           -9
                     44
                18
                     29
      7
                     11
                17
                     -3
           32
                     10
                25
                     14
```

Рис. 2.6: Приклад виводу дерева, сформованого функцією *Tree_Balance*, в консольне вікно

Обхід бінарного дерева передбачає відвідування усіх вершин бінарного дерева, при цьому кожна з вершин відвідується тільки один раз. Існують три види таких обходів, кожний з яких визначається рекурсивно:

Прямий порядок обходу бінарного дерева (CLR – center, left, right), при якому кожна вершина відвідується до того, як будуть відвідані її нащадки. Такий порядок означає обхід в напрямку зверху в низ, коли після проходження чергового вузла продовжується просування в глибину дерева, поки не будуть пройдені всі його нащадки. Такий порядок обходу бінарного дерева називають ще обходом в глибину. Для кореня дерева рекурсивно виконується наступна процедура :

- відвідати корінь;
- відвідати ліве піддерево;
- відвідати праве піддерево

```
   //Функція виводу дерева в прямому порядку
void pre_order(BinaryTree *Tree)
{
  if (!Tree) return;
  //Обхід кореня
  cout<<Tree->Data<<"  ";
  //Обхід лівого піддерева
  pre_order(Tree->Left);
  //Обхід правого піддерева
  pre_order(Tree->Right);
}
```

```
Вивід дерева в прямому порядку.
7  32  25  14  10  17  -3  11  -9  18  29  44  -24  15
```

Рис. 2.7: Вивід дерева, сформованого вище, у прямому порядку

Зворотний порядок (LCR – left, center, right), при якому кожна вершина відвідується лише після того, як будуть відвідані її нащадки. Для кореня дерева рекурсивно виконується наступна процедура :

- відвідати ліве піддерево;
- відвідати праве піддерево;
- відвідати корінь

```
//Функція виводу дерева в зворотньому порядку
void post_order(BinaryTree *Tree)
{
  //Якщо дерева немає, то завершити роботу
  if (Tree==NULL)    return;
  //Обхід лівого піддерева
  post_order(Tree->Left);
  //Обхід правого піддерева
  post_order(Tree->Right);
  //Обхід кореня
  cout<<Tree->Data<<"  ";
}
```

```
Вивід дерева у зворотньому порядку.
14  10  25  -3  11  17  32  29  44  18  15  -24  -9  7
```

Рис. 2.8: Вивід дерева, сформованого вище, у зворотньому порядку

Симетричний порядок, при якому кожна вершина відвідується між відвіданням лівого та правого нащадків. Для кореня дерева рекурсивно виконується наступна процедура :

- відвідати ліве піддерево;
- відвідати корінь;
- відвідати праве піддерево

```
//Функція виводу дерева в симетричному порядку
void in_order(BinaryTree *Tree)
{
  //Якщо дерева немає, то завершити роботу
  if (Tree==NULL)    return;
  //Обхід лівого піддерева
  in_order(Tree->Left);
  //Обхід кореня
  cout<<Tree->Data<<"  ";
  //Обхід правого піддерева
 in_order(Tree->Right);
}
```

```
Вивід дерева у симетричному порядку.
14  25  10  32  -3  17  11  7  29  18  44  -9  15  -24
```

Рис. 2.9: Вивід дерева, сформованого вище, у симетричному порядку

Такий порядок особливо часто застосовується в бінарних деревах пошуку, тому що дає можливість обходу вершин у порядку збільшення значень їх ключів. При симетричному порядку дерево проходиться зліва направо, виводячи впорядковану послідовність ключів вузлів.

Ще одним прикладом обходу дерева може бути його обхід по рівнях (в ширину), починаючи від кореня. Алгоритм такого обходу з використанням черги є наступним:

- відвідати корінь;
- поки проходимо рівень L рівень $L + 1$ тимчасово зберігаємо в стеку;
- переходимо до наступного рівня та проходимо всі його вузли;
- процес повторюємо до тих пір, поки не будуть пройдені всі рівні.

```
// Функція обходу бінарного дерева по рівнях
void levelOrderTree(BinaryTree *Tree)
{
  //Якщо дерева немає, то завершити роботу
  if (Tree == NULL) return;
  // Створюємо чергу
  queue<BinaryTree *> q;
  // Вставляємо корінь дерева в чергу
  q.push(Tree);
  // Поки черга не порожня
  while (!q.empty() )
  {
    // Беремо перший елемент черги
    BinaryTree* temp = q.front();
    // Видаляємо перший елемент із черги
    q.pop();
    // Виводимо значення першого елемента черги
    cout << temp->Data << " ";
    // Вставляємо в чергу лівого нащадка (якщо він є)
    if ( temp->Left != NULL )
      q.push(temp->Left);
    // Вставляємо в чергу правого нащадка (якщо він є)
    if ( temp->Right != NULL )
      q.push(temp->Right);
  }
}
```

Для роботи функції необхідно підключити бібліотеку для роботи із контейнером - черга:

```
#include<queue>
```

```
Вивід дерева по рівнях.
7 32 -9 25 17 18 -24 14 10 -3 11 29 44 15
```

Рис. 2.10: Вивід дерева, сформованого вище, по рівнях

2.1.3 Бінарне дерево пошуку

Бінарним деревом пошуку називають таке бінарне дерево, для будь-якого вузла t якого виконуються наступні вимоги:

- значення всіх полів даних лівого піддерева вузла t є меншими, ніж самого вузла t ;
- значення всіх полів даних вузлів правого піддерева вузла t є більшими, ніж самого вузла t .

Основне використання дерева пошуку - організація ефективного пошуку даних. При пошуку можуть виконуватися наступні операції: знаходження заданого елемента, визначення найбільшого чи найменшого елемента, визначення входження елемента, і т.д. В загальному для одног і того=ж самого набору даних існує достатньо велика кількість двійкових дерев пошуку різної форми. Важливим моментом є той факт, що значення ключових полів вузлів дерева, виведені в симетричному порядку, утворюють впорядковану за зростанням послідовність.

Алгоритм формування дерева пошуку (додавання вузла) наступний: при поступленні чергового вузла із значенням ключа *data* він порівнюється, починаючи із кореневого, із ключем поточного вузла. В залежності від результатів порівняння процес продовжується в лівому та правому піддеревах до тих пір, поки не буде досягнуто вузол, до якого можна додати вхідні дані. В залежності від результату порівняння новий вузол буде розміщено зліва або справа та стане листком дерева.

Структура дерева в результаті вставки буде змінена таким чином, щоб відображати ці зміни, але при цьому зберігати властивість бінарного дерева пошуку. Якщо до дерева не ставиться вимога збалансованості, процедура вставки вузла в дерево відбувається на місці порожнього піддерева, до якого приводить пошук по ключовому значенню.

Приклад функції, яка із вхідного потоку даних формує бінарне дерево пошуку із використанням рекурсивної функції додавання вузла *add_Node* :

```
// Функція додавання вузла до бінарного дерева пошуку
BinaryTree *add_Node(BinaryTree *Tree, int data)
{
  if (!Tree)
  {
    // Створення нового вузла
    Tree = (BinaryTree *) new BinaryTree;
    // Ввід даних
```

```
      Tree->Data = data;
      // Обнулення вказівників
      Tree->Left = NULL;
      Tree->Right = NULL;
    }
    else if (data <= Tree->Data)
      // Це лівий нащадок
      Tree->Left = add_Node(Tree->Left, data);
    else
      // Це правий нащадок
      Tree->Right = add_Node(Tree->Right, data);
    return Tree;
  }
```

Для вхідного набору даних:

```
BinaryTree *A=NULL;
int Tr[]={7,32,25,14,10,17,-3,11,-9,18,29,44,-24,15};
int n=14;
for(int i=0;i<n;i++)
  A=add_Node(A, Tr[i]);
```

буде сформоване наступне бінарне дерево пошуку:

Для цього - самого набору вхідних даних, тільки з іншим порядком слідування:

```
  int Tr[]={32,7,-24,14,25,17,10,-3,-9,11,18,44,29,15};
```

буде сформоване інше бінарне дерево пошуку:

Якщо вихідний потік даних буде впорядкованим:

```
  int Tr[]={-24,-9,-3,7,10,11,14,15,17,18,25,29,32,44};
```

то бінарне дерево пошуку при такому способі його побудови виродиться у лінійний список:

Висотою бінарного дерева називають максимальний із рівнів всіх його піддерев. Висота порожнього дерева рівна 0, а висота дерева, що містить тільки кореневай вузол, рівна 1. Алгоритм пошуку висоти бінарного дерева полягає у визначенні максимального значення серед висот його лівого та правого піддерев (збільшивши значення на 1 - корінь дерева). Рекурсивна функція визначення висоти бінарного дерева може бути реалізована наступним чином

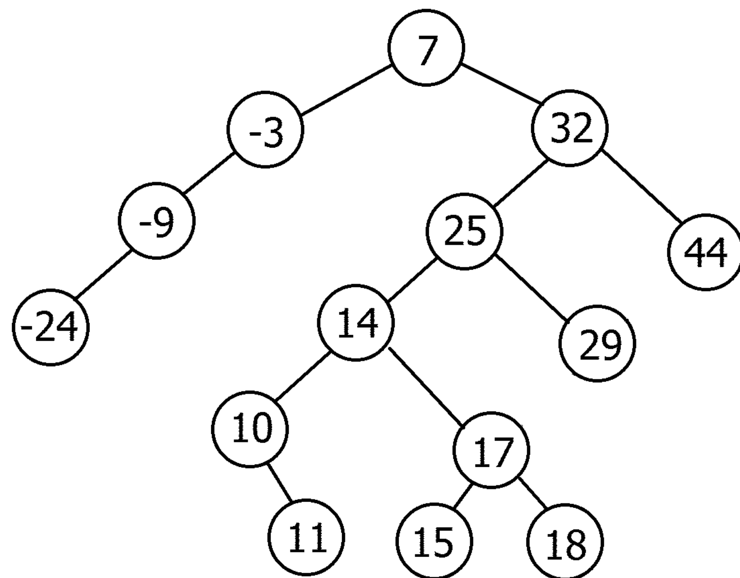


Рис. 2.11: Бінарне дерево пошуку(перший варіант)

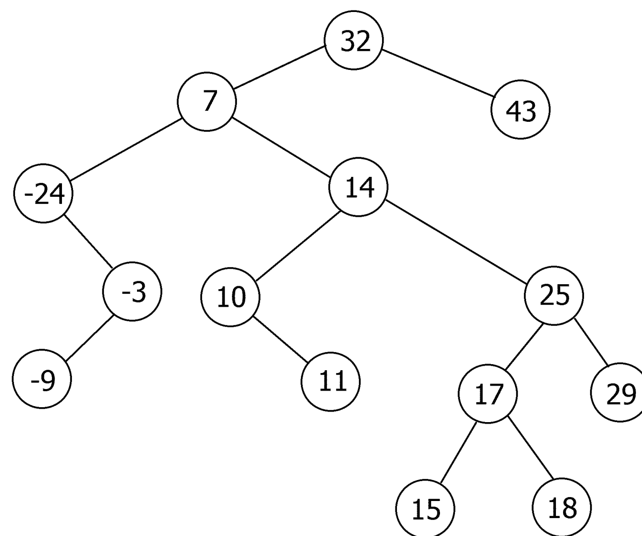


Рис. 2.12: Бінарне дерево пошуку(другий варіант)

```
//Визначення висоти бінарного дерева.
    int Height (BinaryTree **Tree)
    {
    //Висота лівого піддерева
```

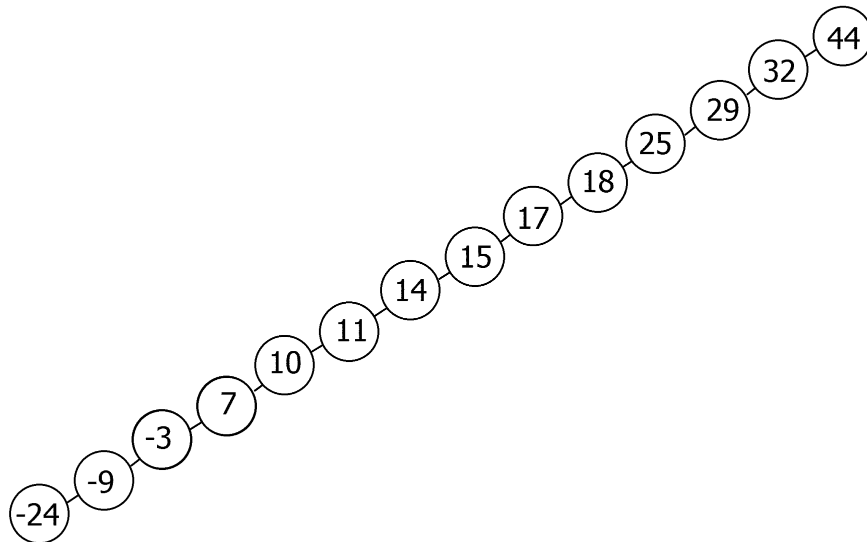


Рис. 2.13: Бінарне дерево пошуку(третій варіант)

```
    int h_left;
    //Висота правого піддерева
    int h_right;
    //Поки дерево не порожнє
    if  (*Tree==NULL) return (0);
    else
    {
      h_left = Height (&((**Tree).Left));
        h_right = Height (&((**Tree).Right));
      }
        //Більша із двох висот
        if  (h_left > h_right) return (1 + h_left);
        else  return (1 + h_right);
    }
```

Пошук вузла в дереві може виконуватися аналогічно до алгоритмів перегляду дерева, але в цьому випадку втрачається ефективність використання бінарного дерева пошуку і пошук фактично стає лінійним.

Для забезпечення ефективності пошуку необхідно використовувати рекурсивну процедуру пошуку значення із заданим ключем. Пошук в такому дереві починається із кореня дерева і на кожному наступному кроці порівнюються шукане значення із ключовим полем поточного вузла. У випадку, якщо зна-

чення співпадають, процедура пошуку завершена, інакше якщо шукане значення менше значення ключового поля , то пошук продовжується в лівому піддереві, інакше - в правому піддереві. Якщо пошук приводить до порожнього піддерева - шукане значення в дереві відсутнє.

```
//Пошук елемента у бінарному дереві
int Find(BinaryTree *Tree, int x)
{
  if (!Tree) return 0;
  // Знайдено шукане значення
  else if (Tree->Data==x) return 1;
  // Пошук в лівому піддереві
  else if (x < Tree->Data) return Find(Tree->Left, x);
  // Пошук в правому піддереві
  else return Find(Tree->Right, x);
}
```

Елемент із максимальним значенням ключа можна знайти, рухаючись за вказівником $Right$ від кореневого вузла до тих пар, поки не зустрінеться значення $NULL$. Якщо у вузла немає правого піддерева, то оскільки всі ключі в правому піддереві згідно властивості дерева пошуку є меншими, максимальний ключ знаходиться в даному вузлі:

```
int find_Max (BinaryTree *Tree)
{
  if (Tree  == NULL)
    return NULL;
while   (Tree->Right)
    Tree = Tree->Right;
return Tree->Data;
}
```

Аналогічно, мінімальний елемент можна знайти, рухаючись за вказівником $Left$:

```
int find_Min (BinaryTree *Tree)
{
  if (Tree  == NULL)
    return NULL;
while   (Tree->Left)
    Tree = Tree->Left;
return Tree->Data;
```

```
    }
```

Видалення вузла із дерева є достатньо складним процесом, бо вузол може бути кореневим, лівим чи правим нащадком або взагалі бути початком цілої вітки. Видалення вузла із бінарного дерева пошуку реалізується наступним чином: якщо вузол, що видаляється, є листком дерева, то вузол стає порожнім піддеревом: для цього достатньо скоректувати відповідний вказівник попереднього вузла.

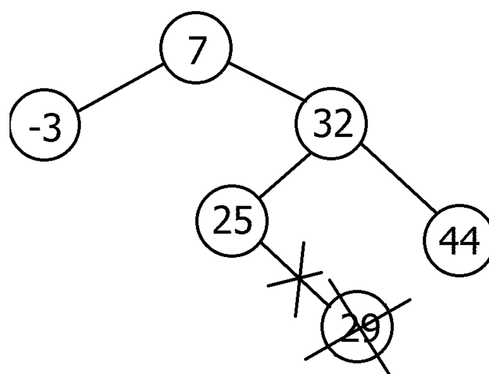


Рис. 2.14: Видалення вузла, що є листком бінарного дерева

Якщо вузол має одного нащадка, то він заміщається своїм нащадком: ми видаляємо вузол z, створюючи новий зв'язок між батьківським та дочірнім вузлом вузла z.

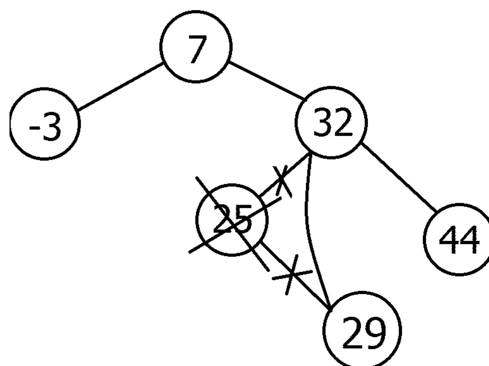


Рис. 2.15: Видалення вузла, що має одного нащадка

Якщо у вузла z є два нащадки, його необхідно замінити самим правим елементом із його лівого піддерева. Для цього відбувається пошук наступного за ним вузла y, в якого немає лівого нащадка, видалення його з позиції, де

він знаходився раніше за рахунок створення нового зв'язку між його батьком та нащадком, та заміною ним вузла z.

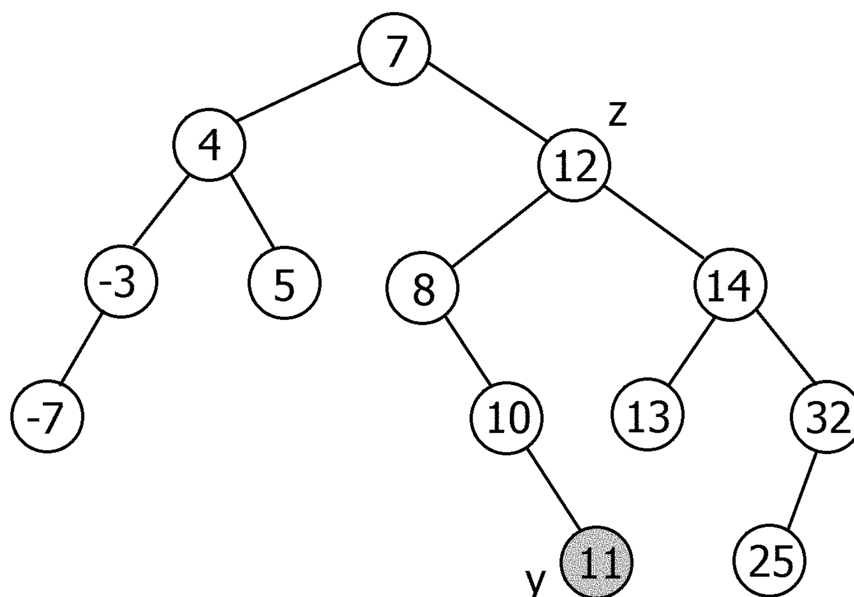


Рис. 2.16: Перший етап видалення вузла, пошук вузла y

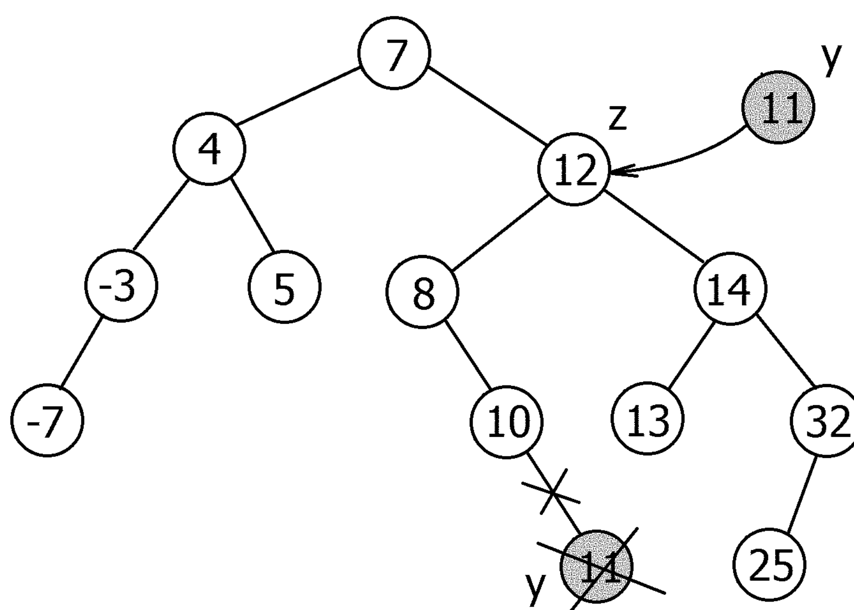


Рис. 2.17: Другий етап видалення вузла: видалення вузла y та його вставка на місце вузла z

Для прикладу, описаний вище алгоритм видалення може бути реалізований у вигляді наступної функції:

```
  //Видалення елемента у бінарному дереві
  BinaryTree *Delete(BinaryTree *Tree, int x)
  {
    BinaryTree* P, *v;
    if (!Tree) cout << "Такий елемент в дереві
відсутній!" << endl;
    else if (x < Tree->Data) Tree->Left = Delete(Tree->Left, x);
    else if (x > Tree-> Data) Tree->Right = Delete(Tree->Right,
x);
    else {P = Tree;
    // Випадок 1
    if (!Tree->Right) Tree = Tree->Left;
    // Випадок 1
    else if (!Tree->Right) Tree = Tree->Right;
    // Випадок 2
    else
    { v = Tree->Left;
    if (v->Right)
    {
      while (v->Right->Right) v = v->Right;
      Tree->Data = v->Right->Data;
      P = v->Right; v->Right = v->Right->Left;
    }
    else
    {
      Tree->Data = v->Data;
      P = v;
      Tree->Left=Tree->Left->Left;
    }
    }
    }
    return Tree;
  }
```

Функція підрахунку вузлів всього дерева, алгоритм роботи якої аналогічний функціям обходу всього дерева. Кількість вузлів зберігається в змінній p. Відповідний їй фактичний параметр повинен прийняти значення 0 перед

викликом функції.

```
void nodes_Tree(BinaryTree *&Tree, int &p)
{
   if (Tree == NULL) return;
   p++;
   nodes_Tree(Tree->Left, p);
   nodes_Tree(Tree->Right, p);
}
```

Функція визначення довжини мінімальної гілки:

```
int min_Depth(BinaryTree *Tree)
{
        if(Tree == NULL)
            return 0;
        if(Tree->Left == NULL || Tree->Right == NULL)
    {
            return 1 + (Tree->Left == NULL ? min_Depth(Tree->
Right) : min_Depth(Tree->Left));
        }
        return (1 + min(min_Depth(Tree->Left), min_Depth(Tree
->Right)));
    }
```

Функція видалення всього дерева, алгоритм роботи якої аналогічний функціям обходу всього дерева:

```
void del_all(BinaryTree *&Tree)
{
  if (!Tree) return;
  del_all(Tree->Left);
  del_all(Tree->Right);
  delete Tree;
  Tree = NULL;
}
```

2.1.4 Збалансоване дерево

В програмуванні збалансоване дерево в загальному розумінні цього слова — це такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем є мінімальною. Ця властивість є важливою тому, що час виконання більшість алгоритмів на бінарних деревах пошуку пропорційний до їхньої висоти, і звичайні бінарні дерева пошуку можуть мати досить велику висоту в тривіальних випадках.

Бінарне дерево називається збалансованим, якщо для кожного його вузла кількість вузлів лівого і правого піддерев відрізняється не більше ніж на одиницю. Ідеально збалансоване дерево, що містить n вузлів, має висоту $(log_2 n) - 1$. Така висота є мінімальною для бінарного дерева, що містить n вузлів.

Пошук в ідеально збалансованому дереві виконується за логарифмічний час. Побудова збалансованого дерева для впорядкованої послідовності значень не викликає трудностей, однак підтримка збалансованості при видаленні та вставці вузлів є достатньо трудоємкою задачею. Томудля побудови збалансованого дерева використовують наступне формулювання збалансованості.

Дерево називають збалансованим тоді і тільки тоді, коли висоти двох піддерев кожної вершини відрізняються не більше ніж на одиницю. Такі дерева також називають **АВЛ -деревами** . В АВЛ -деревах для кожного вузла вводять показник збалансованості та виконують балансування в тому випадку, якщо показник збалансованості порушується. В кожному вузлі **АВЛ -дерева**, крім поля даних та вказівників на лівого та правого нащадків зберігається показник балансу - різниця висот правого та лівого піддерев (або його висоту).

```
struct ABL_Tree
{
 //Поле даних
 int Data;
 //Вказівник на лівого нащадка
ABL_Tree *Left;
 //Вказівник на правого нащадка
ABL_Tree *Right;
 //Висота дерева
 int height;
};
```

Алгоритми пошуку, вставки та видалення є аналогічними до відповідних

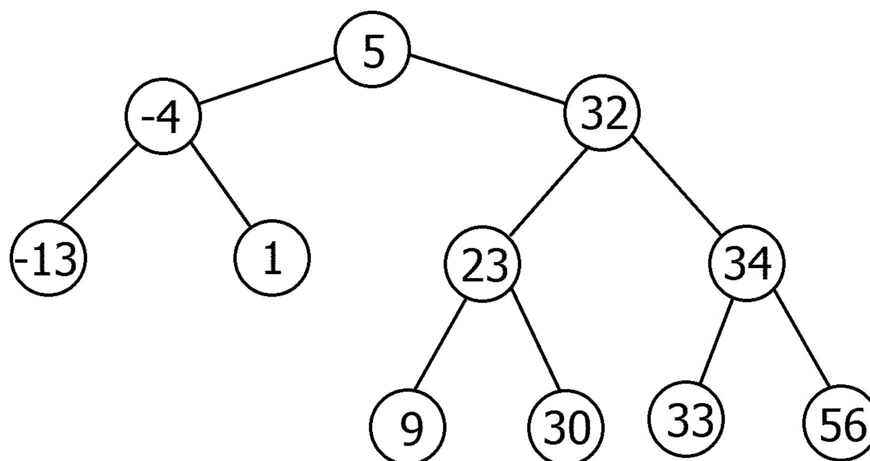


Рис. 2.18: Приклад ABL дерева

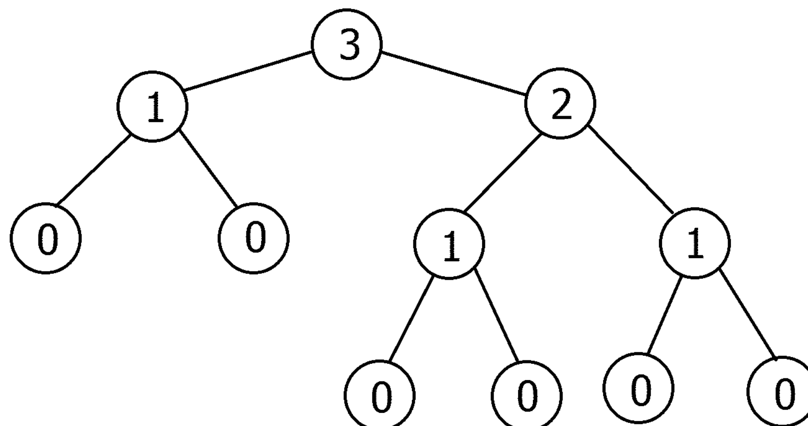


Рис. 2.19: Приклад ABL дерева. У вузлі задано значення висоти відповідного дерева

операцій для бінарного дерева пошуку. Однак після будь - яких операцій, що міняють структуру дерева (вставка чи видалення елементу) баланс може бути порушений: висоти двох піддерев для деякого вузла відрізняються більше ніж на 1. Балансуванням називають приведення дерева до висоти $log_2 n$ (n - число елементів дерева).

В цьому випадку необхідно застосовувати алгоритми балансування дерева, які грунтуються на операціях обертання. Розрізняють 4 типи обертань:

Мале ліве обертання.

2.1.5 Червоно-чорне дерево

У бінарному дереві пошуку реалізують всі базові операції над динамічними множинами за час $O(h)$, де h – висота дерева. Таким чином, операції виконуються тим швидше, чим меншою є висота дерева. Однак в найгіршому випадку продуктивність бінарного дерева пошуку виявляється не кращою за продуктивність зв'язаного списку.

Червоно-чорні дерева представляють собою одну з багатьох схем збалансованих дерев пошуку, які гарантують виконання операцій над динамічною множиною за час $O(lgn)$ навіть у найгіршому випадку.

Червоно-чорне дерево представляє собою бінарне дерево пошуку з одним додатковим бітом кольору в кожному вузлі. Колір вузла може бути або чорним, або червоним. Окрім цього на вузли червоно-чорного дерева накладаються наступні вимоги:

- кожний вузол є червоним або чорним;
- корінь дерева є чорним;
- кожний листок дерева (NULL) є чорним;
- якщо вузол червоний, то обидва його дочірніх вузла – чорні;
- будь-який шлях від кореня до листка (NULL) містить однакову кількість чорних вузлів.

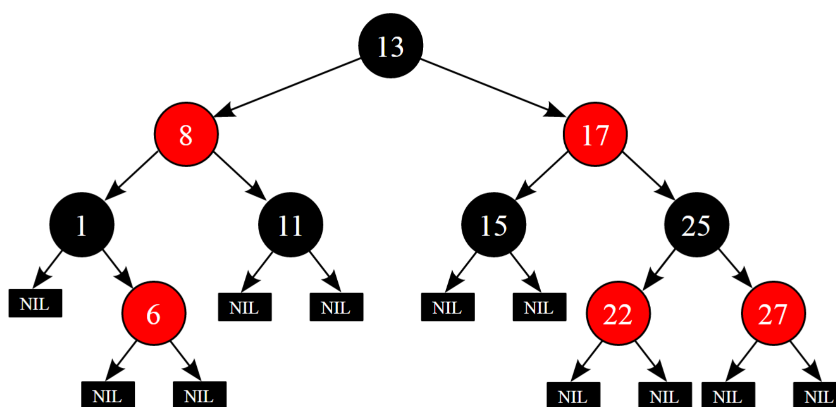


Рис. 2.20: Приклад червоно-чорного дерева

Для простоти реалізації в дерево добавляються фіктивні чорні вершини: для кожної вершини дерева при відсутності у неї нащадків, на місце відповідного нащадка вставляється фіктивна чорна вершина. При визначення висоти дерева фіктивні вершини не враховуються

Для задання однієї вершини червоно-чорного дерева можна скористатися наступною структурою:

```
struct SBTree_
{
  int IsRed;
  int value;
  struct STree_ *par;
  struct STree_ *left, *right;
} SBTree;
```

Тут:

- **par** -вказівник на батьківський елемент даної вершини;
- **left** та **right** - казівники на лівого та правого нащадків відповідно;
- **IsRed** задає колір вершини (червона чи чорна);
- **value** - ключ вершини (дані, що в ній зберігаються).

Висота червоно-чорного дерева, що містить N вершин рівна:

$$h = O(log_2(N))$$

2.2 Графи

2.2.1 Основні поняття та визначення

Алгоритми на графах Поняття графа та його зображення в пам'яті комп'ютера. Найкоротші шляхи у графі. Обхід графу. Для опису та аналізу різних складних систем, що складаються із зв'язаних між собою елементів, часто використовують графічні схеми. На них елементи зображають за допомогою кружків (точок, прямокутників), а зв'язки між ними - лініями або стрілками, що з'єднують елементи:

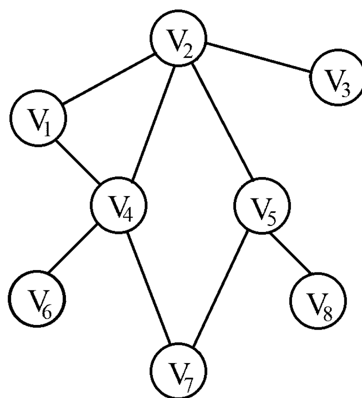


Рис. 2.21: Формальне представлення складної системи

Кружки називають вершинами (вузлами) графа, а з'єднувальні лінії - ребрами. Наведений вище граф складається із восьми вершин $V_1, V_2 \ldots, V_8$ вершин та 8 ребер $[V_1, V_2]$, $[V_2, V_3]$, ..., $[V_5, V_8]$.

Фактично граф – це складна нелінійна багатозв'язна структура, що відображає властивості і зв'язки складного об'єкту. У вузлах (кружках) міститься інформація про елементи об'єкту. Зв'язки між вузлами задаються ребрами графа. Одне ребро з'єднує тільки одну пару вершин. Дві вершини, що з'єднані між собою ребром, називають *суміжними вершинами*. Прикладами таких систем можуть бути: географічна карта (міста та дороги що їх з'єднують); гіпертекст (гіпертекстові документи мережі та лінії зв'язку), мікросхеми (транзистори, діоди, резистори і т.д. та їх електричні з'єднання) і т. д.

Ця багатозв'язна структура має наступні властивості:

- на кожний елемент (вузол, вершину) може бути довільна кількість посилань;

- кожний елемент може мати зв'язок з будь-якою кількістю інших елементів;
- кожний зв'язок (ребро, дуга) може мати напрям і вагу.

Формально *граф (graph)* $G(V, E)$ це сукупність двох множин: непорожня множина V (множина вершин) та множина E - множина ребер:

$$G(V, E) =< V; E >, \ V \neq \varnothing$$

Кількість вершин графа G (потужність множини V) називають *розміром графа*, а кількість ребер (потужність множини E) називають *потужністю графа.*

Ребро e_i називають *інцидентним* до вершини V_i, якщо V_i є одним із кінців e_i. Дві вершини називають суміжними, якщо вони іцедентні одна одній. Два ребра називають суміжними, якщо вони мають одну спільну вершину.

Розрізняють два основних види графів: орієнтовані та неорієнтовані. Якщо ребра графу мають напрям від однієї вершини до іншої, то такий граф називають ірієнтованим, а відповідні вершини називають початком та кінцем дуги. Якщо напрям ребер не задано (значення не має), то такий граф називають орієнтованих. Для більшості задач важливим власне наявність ребер між окремими вершинами.

В більшості випадків важливими є базові структурні властивості, які можна визначити, аналізуючи множину ребер графу.

Шлях (path) у графі є послідовністю вершин, у якій два сусідні елементи інцидентні, причому будь-яка вершина не зустрічається двічі. Відстань між двома вершинами графа — найменша довжина шляху, що з'єднує ці вершини.

Відстань між двома вершинами графа - довжина найкоротшого шляху, що з'єднує ці дві вершини.

Гамільтонів шлях - простий шлях в графі, що містить всі вершини графа точно по одному разу.

Цикл (простий цикл) - простий шлях в графі, що містить всі вершини графа точно по одному разу та починається і закінчується в одній і тій самій вершині.

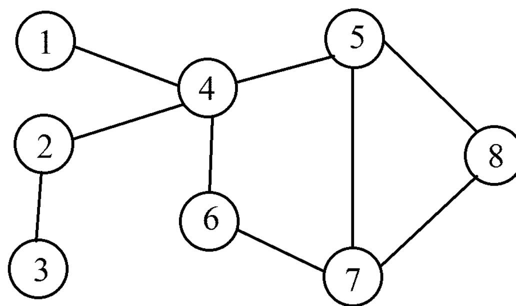
2.2.2 Способи представлення графів

Надалі із всієї множини різних типів графів ми будемо розглядати прості графи, для яких в графі немає кратних ребер (для кожної пари вершин є тільки одне і тільки одне ребро, що їх з'єднує) та вершини, що утворюють ребра повинні бути різними (петлі відсутні).

Нехай G - граф, що містить n вершин. Побудуємо квадратну матрицю A порядку n, у якій кожний елемент $A_{i,j}$ визначається наступним чином:

$$\begin{cases} 1, & \text{якщо } e_{i,j} \in E \\ 0, & \text{якщо } e_{i,j} \notin E \end{cases}$$

Тут $e_{i,j}$ - ребро, що зв'язує вершину i та j. Така матриця називається матрицею суміжності графа G. Для звичайного графа в матриці суміжності на головній діагоналі стоять нулі і вона симетрична відносно головної діагоналі. Нижче приведено приклад графа із пронумерованими вершинами і побудована для нього матриця суміжності. Порядок нумерації значення не має:



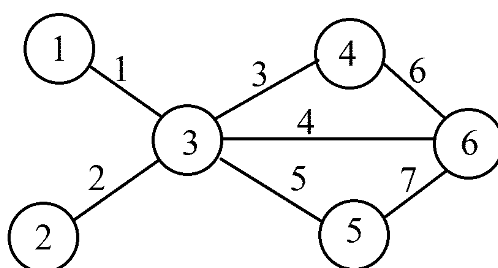
$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Рис. 2.22: Матриця суміжності

Інший спосіб представлення графа - матриця інцедентності. Для її побудови всі вершини графа пронумеруємо числами від 1 до n, а ребра - від 1 до

m. Матриця інцедентності має n рядків та m стовпців. Її елементи рівні 1, якщо вершина із номером i інцедентна ребру із номером j, інакше елемент рівний нулю.

Нижче приведено приклад графа із прономерованими вершинами та ребрами і побудована для нього матриця інцедентів:



$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Рис. 2.23: Матриця інцедентів

У кожному стовпці матриці інцедентності є тільки два елементи, відмінних від 0, тому такий спосіб задання графа є досить неекономним по відношенню до оперативної пам'яті. Відношення інцедентності можна також реалізувати у вигляді списку ребер, у якому кожен рядок списку відповідає ребру, в якому записано номери вершин, інцедентним йому. Порядок цих вершин для орієнтованого графу є довільним.

За списком ребер можна легко побудувати матрицю інцедентності. Фактично кожний рядок списку відповідає стовпцю матриці інцедентності з тим-самим номером. Для неорієнтованого графу в рядку списку записуються номери елементів стовпця матриці інцедентності, що дорівнюють 1.

При моделюванні реальних систем вершинам та ребрам графа можуть присвоюватися деякі значення. Значення цих чисел визначається природою самої системи. Наприклад, якщо граф моделює систему автомобільних доріг, то ребра можуть задавати відстань між відповідними містами, а вершини, наприклад, вартість пального на заправках даного міста. Такі числа, не залежно від їх природи називають вагою. Зважений граф — граф, кожному

Ребро	Вершини
1	1, 3
2	2, 3
3	3, 4
4	3, 6
5	3, 5
6	4, 6
7	5, 6

Табл. 2.1: Список ребер попереднього графа

ребру якого (чи вершині) поставлено у відповідність деяке значення. Наприклад:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 \end{pmatrix}$$

Рис. 2.24: Матриця суміжності зваженого орієнтованого графа

Як видно, крім того, що граф є зваженим, він є і орієнтованим, бо матриця суміжності не є симетричною відносно головної діагоналі.

2.2.3 Мінімальне покриваюче (каркасне) дерево

Мінімальне покриваюче дерево у зваженому, неорієнтованому графі — це кістяк цього графа, що має мінімальну можливу вагу, де під вагою дерева розуміється сума ваг його ребер. За означенням для графа $G = (V, E)$ із відомими для кожного ребра, що з'єднує вершини (u, v) вагами $w(u, v)$ це є множина ребер T, що з'єднує всі вершини графа та забезпечує мінімум повної ваги:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

Існує кілька алгоритмів знаходження мінімального покриваючого дерева. Одним із найвідоміших серед них є алгоритм Прима. Він дозволяє виділити тільки ті ребра графа, за допомогою яких можна обійти всі вершини графа і при цьому сумарна вага цих ребер буде мінімальною. Шуканий мінімальний каркас будується поступово, добавляючи до нього одне ребро за іншим. На першому етапі каркас складається із однієї, довільним чином вибраної вершини. Наступним кроком є вибір інцедентного до неї ребра із мінімальною вагою. Процес добавлення вершин із мінімальними вагами продовжується до тих пір, поки каркас не біде містити всі вершини графа.

Більш формально алгоритм можна представити наступним чином. Введемо логічний масив $visited[N]$, який спочатку заповнимо нулями ($false$) та виконаємо наступні кроки

1. виберемо наприклад першу вершину та відмітимо її як використану $visited[0] = true$;;
2. найдемо серед всіх ребер w, що з'єднують дану використану вершину із іншими невикористаними вершинами ($visited[0] = false$;) ребро із мінімальною вагою. Якщо ребер із мінімальною вагою декілька, то вибираємо будь-яку із них.Якщо таких ребер немає, то завершуємо виконання алгоритму
3. добавляємо інцедентну знайденому ребру із мінімальною вагою вершину до каркасу;
4. відмічаємо додану до каркасу вершину j як використану $visited[j] = true$;
5. переходимо до виконання пункту 2.

Побудоване таким чином каркасне дерево буде мати мінімальну сумарну вагу, бо на кожному етапі його роботи із всіх можливих вибирається ребро

із мінімальною вагою. Для даного зваженого зв'язаного графу мінімальне каркасне дерево є єдиним, якщо ваги всіх ребер різні. Якщо є кілька ребер однакової ваги, то вигляд побудованого каркасного дерева залежить від порядку перегляду ребер із однаковими вагами. Якщо вихідний граф не є зв'язаним, то каркасне дерево не буде побудоване.

Для практичної реалізації алгоритму Прима в матриці суміжності графа, якщо ребра, що з'єднує вершини i та j не існує, ставиться нескінченість - деяке велике число, яке більше ваги будь -якого ребра графа. У приведеному нижче прикладі таке число приймається рівним $INT_MAX = 2147483647$.

Приклад 2. 12. *Дано N міст, які необхідно з'єднати дорогами так, щоби можна було добратися із одного міста до іншого напряму чи через інші міста. У вхідному файлі graf.txt задана вартість будівництва кожної такої дороги. Якщо дорогу між містами збудувати неможливо, то в матриці суміжності стоять нулі. Необхідно мінімізувати загальну вартість побудови системи доріг.*

Примітка: задача може бути зведена до задачі згаходження мінімального каркасного дерева в зваженому неорієнтованому графі. У файлі $graf.txt$ задано кількість вершин графа та його матриця суміжності:

```
graf.txt
13
0 1 2 11 0 0 0 0 0 0 0 0 0
1 0 4 0 31 0 0 0 0 0 0 0 0
2 4 0 0 0 2 0 0 0 0 0 0 0
11 0 0 0 2 0 0 0 6 0 0 0 0
0 31 0 2 0 9 0 1 0 0 0 0 0
0 0 2 0 9 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 4 0 0 0
0 0 0 0 1 0 0 0 2 7 0 0 0
0 0 0 6 0 0 0 2 0 0 9 11 0
0 0 0 0 0 0 4 7 0 0 5 0 8
0 0 0 0 0 0 0 0 9 5 0 14 21
0 0 0 0 0 0 0 0 11 0 14 0 3
0 0 0 0 0 0 0 8 0 0 21 3 0
```

Рис. 2.25: Структура файлу $graf.txt$

2.2.4 Пошук в ширину

Розв'язок задач аналізу графів вимагає обходу графу із проходом всіх вершин, які доступні із стартової вершини. Для реалізації такого обходу найчастіше використовують дві стратегії: пошук в глибину та пошук в ширину.

При пошуку в ширину спочатку розглядаються всі вершини, суміжні зі стартовою - тобто знаходяться від неї на відстані 1 (найближчі сусіди), потім вершини, що знаходяться на відстані 2 (наступні за найбдижчим сусіди)і т.д. Такий порядок обходу забезпечується тому, що в якості активної завжди вибирається та із відкритих вершин, яка стала відкритою раніше інших. Такий порядок обходу легко реалізувати, якщо для збереження відкритих вершин використовувати чергу - якщо нова вершина стає відкритою, то вона добавляється в кінець черги, а активна вибирається із її початку.

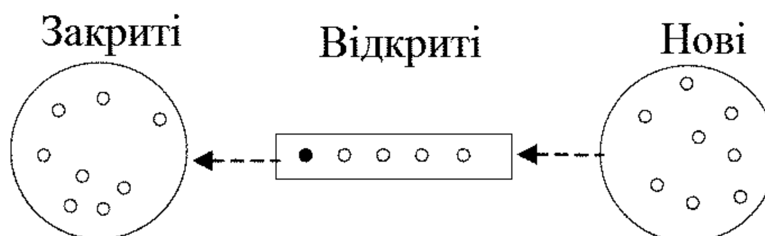


Рис. 2.26: Схема "перенесення"вершин

В ході роботи алгоритму кожна вершина міняє свій статус, послідовно проходячи три етапи: спочатку вона нова (ще не зустрічалася в процесі обходу графа), потім стає відкритою (вершина вже зустрічалася, але її околиці ще не досліджено), і накінець переходить в розряд закритих. Чорним кружком показано активну вершину. При такому способі вибору вершина буде вибиратися в якості активної до тих пір, поки не буде досліджено весь її окіл.

Цей алгоритм пошуку в графі також називають хвильовим алгоритмом, бо обхід графа йде за принципом поширення хвилі: хвиля розтікається рівномірно в усі боки з однаковою швидкістю. На i - тому кроці будуть вибрані всі суміжні вершини, досяжні за i ходів - переходів із однієї вершини в іншу.

Для практичної реалізації алгоритму необхідно задіяти три структури даних:

- матриця суміжності графа G;
- черга $quen$;
- масив вершин, які вже були відвідані $visited$. Спочатку масив повинен бути заповнений значеннями $false$ (жодна з вершин не була відвідана).

Дві перші структури є структурами цілого типу, масив *visited* - логічного. Процес обходу графа можна розбити на наступні етапи:

1. Обнуляємо масив *visited*: жодна вершина графа ще не була відвідана.
2. Вибираємо в якості стартової деяку вершину s та поміщаємо її в чергу *quen*.
3. Досліджуємо вершину s (відмічаємо як відвідана у масиві *visited*) та всі суміжні із нею вершини поміщеємо в кінець черги *quen*.
4. Якщо на даному етапі черга виявилася порожньою, то припиняємо роботу, інакше відвідуємо вношину, що стоїть на початку черги, відмічаючи її як відвідану у масиві *visited*. Всі її найближчі сусіди заносимо в кінець черги.
5. Переходимо до виконання пункту 4.

Приклад 2. 13. *Обхід графа у ширину з використанням матриці суміжності. Для організації черги використано контейнер queue стандартної бібліотеки шаблонів. Простий граф генерується випадковим чином за допомогою окремої функції (Generator).*

```
#include <iostream>
#include <windows.h>
#include <queue>
using namespace std;
//Кількість вершин графа.
const int n=6;
//Матриця суміжності графа
int GM[n][n];
//Черга
queue<int> Q;

// Функція генерування простого графа випадковим чином.
void Generator(int n)
{
int temp;
for(int i=0;i<n;i++)
    {
    for(int j=i;j<n;j++)
    {
```

```
        if(i==j)
      {
            GM[i][j]=0;
        continue;
    }
    temp=rand()%100;
    if (temp>50) GM[i][j]=0,GM[j][i]=0;
    else GM[i][j]=1, GM[j][i]=1;
        }
     }
  }

    //Функція пошуку в ширину.
    void BFS(bool *visited, int unit)
    {
    // Помічаємо вершину графу як відвідану.
    visited[unit]=true;
    //Поміщаємо вершину графу в чергу.
    Q.push(unit);
    // Поки черга не стане порожньою
    while (!Q.empty())
    {
    //Вибір елементу, що знаходиться на початку черги.
      unit=Q.front();
    //Видалення вершини із черги.
      Q.pop();
    //Вивід вершини графу на екран.
    //Враховано, що нумерація починається із нуля.
     cout<<unit+1<<" ";
    // Обхід всіх суміжних вершин
  //із перевіркою, чи не були вони відвідані.
    for (int i=0; i<n; i++)
        if (GM[unit][i] && !visited[i])
        {
    //Додавання вершини в чергу, якщо
    // вона не була відвідана та до неї є ребро.
     Q.push(i);
    // Помічаємо вершину графу як відвідану
    visited[i]=true;
```

```
        }
     }
  }

    void main()
    {
   SetConsoleCP(1251);
     SetConsoleOutputCP(1251);
    Generator(n);
    int start;
    cout<<"Введіть стартову вершина обходу графа >> ";
    cin>>start;
    // Логічний динамічний масив вершин, що були відвідані.
    bool *visited=new bool[n];
    cout<<"Матриця суміжності згенерованого графа: "<<endl;
    // Початкове обнулення масиву visited
    for (int i=0; i<n; i++)
        {
        visited[i]=false;
    // Вивід матриці суміжності.
     for (int j=0; j<n; j++)
        cout<<" "<<GM[i][j];
     cout<<endl;
    }
    cout<<"Порядок обходу: "<<endl;
    BFS(visited, start-1);
    delete []visited;
    cout<<endl;
    }
```

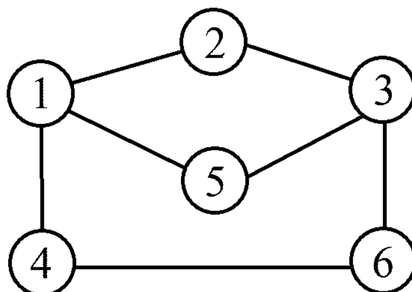


Рис. 2.27: Згенерований простий неорієнтований граф

```
Введіть стартову вершина обходу графа >> 3

Матриця суміжності згенерованого графа:
 0 1 0 1 1 0
 1 0 1 0 0 0
 0 1 0 0 1 1
 1 0 0 0 0 1
 1 0 1 0 0 0
 0 0 1 1 0 0

Порядок обходу:
3 2 5 6 1 4
Press any key to continue . . .
```

Рис. 2.28: Результат роботи програми

2.2.5 Пошук в глибину

При пошуку в глибину в якості активної вершини вибирається та із відкритих вершин, яка була відкрита останньою. Для реалізації такого алгоритму вибору найбільш зручною структурою збереження множини відкритих вершин є стек: вершини, що відкриваються, поміщаються в стек в тому порядку, в якому вони відкриваються, а в якості активної вибирається остання.

Фактично рух здійснюється в глибину - від однієї вершини до іншої, з нею суміжної, поступово віддаляючись від початкової вершини , поки не дійдемо до тупика. Вершина буде тупиком, якщо всі суміжні вершини вже відвідані. Повернення із тупика здійснюється вздовж вже пройденого шляху, поки не буде знайдено вершину, в якої є ще не відвідані суміжні вершини. Після чого рух продовжується в новому напрямку. Процес завершується, коли повернемось в стартову вершину і всі її суміжні вершини вже були відвідані.

Для практичної реалізації алгоритму необхідно задіяти три структури даних:

- матриця суміжності або інцедентів графа G;

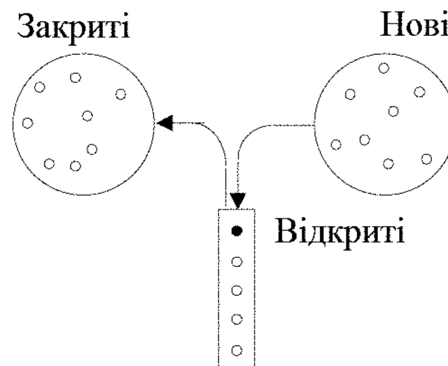


Рис. 2.29: Схема "перенесення"вершин

- стек *stack*;
- масив вершин, які вже були відвідані *visited*. Спочатку масив повинен бути заповнений значеннями *false* (жодна з вершин не була відвідана).

Процес обходу графа можна розбити на наступні етапи:

1. Обнуляємо масив *visited*: жодна вершина графа ще не була відвідана.
2. Вибираємо в якості стартової деяку вершину s та поміщаємо її в стек *stack* та відмічаємо як відвідана в масиві *visited*.
3. Читаємо вершину зі стеку та вибираємо будь-яку суміжну із нею вершину і поміщаємо на вершину стека *stack*.
4. Якщо вершина на верхівці стеку є тупиком, то виводимо її на екран (як порядок обходу) та видаляємо зі стеку.
5. Якщо стек ще не порожній, то переходимо до пункту 3.
6. Завершуємо роботу.

Приклад 2. 14. *Обхід графа у глибину з використанням матриці інцедентів. Для організації стеку використано контейнер stack стандартної бібліотеки шаблонів. Матриця інцедентів ініціалізується на етапі її опису.*

У прикладі лістінгу програми задана таблиці інцедентності, що визначає наступний граф:

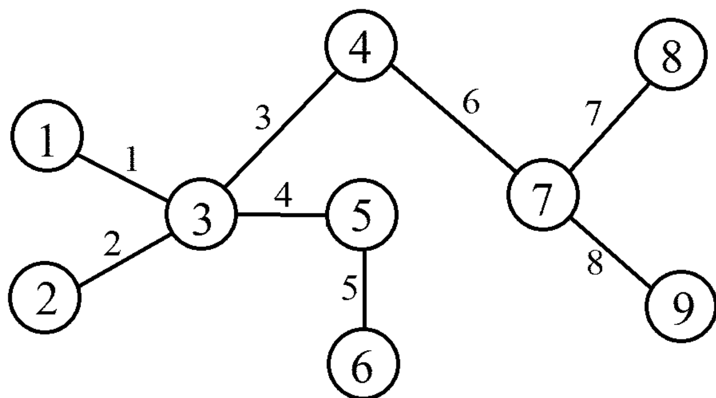


Рис. 2.30: Схема графа

```
 #include <iostream>
  #include <windows.h>
  #include <stack>
  using namespace std;
  const int n=9;
  const int m=8;

  //Матрица інцедентів
  int GM[n][m] =
  {
      {1, 0, 0, 0, 0, 0, 0, 0},
      {0, 1, 0, 0, 0, 0, 0, 0},
      {1, 1, 1, 1, 0, 0, 0, 0},
      {0, 0, 1, 0, 0, 1, 0, 0},
      {0, 0, 0, 1, 1, 0, 0, 0},
      {0, 0, 0, 0, 1, 0, 0, 0},
      {0, 0, 0, 0, 0, 1, 1, 1},
      {0, 0, 0, 0, 0, 0, 1, 0},
      {0, 0, 0, 0, 0, 0, 0, 1}
  }
  ;

  //Стек
  stack<int> Q;
  //Функція пошуку в глибину.
  void DFS(bool *visited, int unit)
  {
  // Помічаємо вершину графу як відвідану.
  visited[unit]=true;
  //Поміщаємо першу вершину графу в стек.
  Q.push(unit);
  // Поки стек не стане порожньою.
  while (!Q.empty())
  {
  //Вибір елементу, що знаходиться на вершині стеку.
  q1: unit=Q.top();
//Пошук сусідів у матриці інцедентів.
  for (int i=0; i<m; i++)
```

```
  {
    if(GM[unit][i]!=0)
  {
  for (int j=0; j<n; j++)
  if (GM[j][i]!=0 && visited[j]==false)
  {
  visited[j]=true;
    Q.push(j);
  goto q1;
  }
  }
  }
  //Якщо нових сусідів не знайдено, то видаляється вершина зі
 стеку.
  //та вивести її на екран
  cout<<" "<<Q.top()+1;
  Q.pop();
  }
  }

  void main()
  {
SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
  //Generator(n);
  int start;
  cout<<"Введіть стартову вершина обходу графа » ";
  cin>>start;
  // Масив вершин, що були відвідані.
  bool *visited=new bool[n];
  cout<<endl;
  cout<<"Матриця інцедентів графа: "<<endl;
  // Жодна вершина ще не була відвідана.
  for (int i=0; i<n; i++)
  {
  visited[i]=false;
  // Вивід матриці інцедентів.
  for (int j=0; j<m; j++)
  cout<<" "<<GM[i][j];
```

```
    cout<<endl;
    }
    cout<<endl;
    cout<<"Порядок обходу: "<<endl;
    DFS(visited, start-1);
    delete []visited;
    cout<<endl;
    }
```

```
Введіть стартову вершина обходу графа >> 1

Матриця інцедентів графа:
 1 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0
 1 1 1 1 0 0 0 0
 0 0 1 0 0 1 0 0
 0 0 0 1 1 0 0 0
 0 0 0 0 1 0 0 0
 0 0 0 0 0 1 1 1
 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 1

Порядок обходу:
 2 8 9 7 4 6 5 3 1
Press any key to continue . . .
```

Рис. 2.31: Результат виконання програми

2.2.6 Ейлерів цикл

Циклом в теорії графів називають замкнутий обхід, в якому перша та остання вершина збігаються. Цикл, що містить всі усі ребра графу тільки по одному разу, називається Ейлеровим. Ейлерів цикл в неорієнтованому графі можна побудувати тільки тоді, коли граф є зв'язаним та в ньому відсутні вершини із непарними ступенями. Нагадаємо, що ступенем вершини називають кількість суміжних із нею вершин.

Ейлеровим шляхом називається маршрут, який включає всі ребра графа по одному разу. Він існує тоді, коли граф є зв'язаним та містить не більше двох вершин із непарними ступенями. При цьому зрозуміло, що одна вершина із непарним ступенем в графі існувати не може і одна із них повинна бути вибрана за початок маршруту (ейлерівського шляху), а друга буде його кінцем. Якщо з'єднати ці дві вершини фіктивним ребром, то всі ступені вершин стануть парними і задача може бути зведена до пошук циклу Ейлера.

Алгоритм побудови циклу Ейлера починає свою роботу із деякої довільної вершини v: будується цикл починаючи із вершини v і при цьому вершини шляху поміщаються у стек, а відповідні ребра видаляються. Якщо шлях більше неможливо продовжити, то це означає, що що із графа був видалений деякий цикл. При цьому степінь всіх вершин, як і у вихідному графі, залишається парною. Поточна вершина видаляється із стеку. Верхній елемент стеку (вершина) стає поточним і шлях шукається вже із неї. Процес продовжується до тих пір, поки стек не спорожніє.

Доведення достатності подамо алгоритмом знаходження ейлерового шляху. Не обмежуючи загальності міркувань, розглянемо зв'язний граф, у якому всі вершини мають парний степінь. Якщо це не так, долучимо до графа ще одну вершину і сполучимо її ребрами з двома вершинами, що мають непарний степінь.

Поки не вилучено усі ребра, робимо таке:

Припустимо тепер, що всі вершини графу G мають парний степінь. Почнемо маршрут P_1 із довільної вершини v_1 і продовжимо його, наскільки це можливо, вибираючи щоразу нове ребро. Позаяк степені всіх вершин парні, то, увійшовши в будь-яку вершину, відмінну від v_1, ми завжди маємо можливість вийти з неї через іще не пройдене ребро. Тому шлях P_1 завершиться у вершині v_1, тобто P_1 обов'язково виявиться циклом. Якщо з'ясується, що маршрут P_1 містить усі ребра графу G, то це ейлерів цикл. У протилежному випадку вилучимо з G всі ребра циклу P_1 і всі вершини, інцидентні лише вилученим ребрам. Отримаємо якийсь зв'язаний граф G_1. Оскільки P_1 та G мають вершини лише парних степенів, то, очевидно, і граф G_1, матиме цю

властивість. Окрім того, позаяк граф G зв'язаний, то графи P_1 і G_1 мають принаймні одну спільну вершину v_2. Тепер із вершини v_2 побудуємо цикл P_2 в графі G_1 аналогічно до того, як ми будували цикл P_1 у графі G. Цикл P_2 вставимо в цикл P_1 на місце вершини v_2. Одержимо цикл P_3. Серед вершин побудованого циклу P_i. вибираємо ту, що є кінцем невилученого ребра. Відсутність такої вершини означає, що всі ребра уже пройдено або граф не є зв'язним;

Алгоритм побудови ейлеревого циклу

1. Вийти з довільної вершини v_1. Кожне ребро, що вже пройдене, закреслити. Якщо шлях l_1 замикається в v_1 і проходить через всі ребра графа, то отримаємо шуканий ейлерів цикл.
2. Якщо залишилися ребра, що не пройдені, то повинні існувати вершини v_2 , що належать l_1 , і ребру що не ввійшов в l_1.
3. Так як v_2 – парне, то число ребер, яке належить v_2 і які не ввійшли до шляху l_1 , також парне. Почнемо новий шлях l_2 з v_2 і використовуємо тільки ребра, що не належать l_1. Цей шлях закінчиться в v_2.
4. Об'єднаємо обидва цикли: v_1 пройдемо по шляху l_1 до v_2, а потім по l_2 і, повернувшись в v_2, пройдемо по залишеній части l_1 назад в v_1.
5. Якщо знову знайдуться ребра, які не ввійшли в шлях, то почнемо нові цикли. Так як число ребер і вершин кінцеве, то процес закінчиться.

Існує й інший алгоритм побудови ейлерового циклу, який дає змогу побудувати цей цикл одразу. Це алгоритм Флері.

Робота алгоритму полягає в нумерації ребер у процесі побудови ейлерового циклу.

1. Починаємо з довільної вершини u та присвоюємо довільному ребру (u, v) номер 1. Викреслюємо ребро (u, v) та переходимо у вершину v.
2. Нехай v – вершина, у яку ми перейшли на попередньому кроці, k – останній присвоєний номер. Вибираємо довільне ребро, інцидентне вершині v, причому міст вибираємо лише тоді, коли немає інших можливостей. Присвоюємо вибраному ребру номер (k + 1), викреслюємо його та збільшуємо лічильник k на 1.
3. Якщо всі ребра графу викреслено та пронумеровано – ці номери задають послідовність ребер в ейлеревому циклі. Інакше перейти на крок 2.

Примітка: Не проходимо по ребру, якщо видалення даного ребра приводить до розбиття графа на дві зв'язані компоненти - необхідно перед видаленням ребра перевіряти, чи не є воно мостом. Міст - ребро в графі, видалення якого збільшує число компонент звязності.

Класичним прикладом задачі на ейлерів цикл є задача про доміно:

Приклад 2. 15. *Є N каменів доміно, на двох кінцях яких записано по одному числу (за звичай від 0 до 6, хоча в загальному випадку це не має значення). Необхідно викласти всі доміно в ряд так, щоби у двох сусідніх каменів числа, записані на спільній стороні, співпадали. Доміно можна повертати.*

Задача може бути переформульована наступним чином: числа, записані на каменях доміно - це номери вершин графу, а самі камені доміно - ребра графу. Кожен камінь із числами (a,b) - це ребро (a,b) або (b,a). В такому представленні задача зводиться до задачі знаходження ейлерового циклу на побудованому таким чином графі.

```
#include <iostream>
#include <fstream>
#include <windows.h>
#include <stack>
using namespace std;

// Старотова вершина для побудови ейлерівського циклу
const int start = 5;

//Список суміжності для задання графа
struct S_Graf
{
  int inf;
  S_Graf *next;
};

//Додавання суміжної вершини
void add(S_Graf* &list,int data)
{
  if(!list)
  {
    list=new S_Graf;
```

```
      list->inf=data;
      list->next=0;
      return;
    }
    S_Graf *temp=list;
    while(temp->next)
      temp=temp->next;
    S_Graf *elem=new S_Graf;
    elem->inf=data;
    elem->next=NULL;
    temp->next=elem;
  }

  // Видалення вершини key зі списку
  void del(S_Graf* &l,int key)
  {
    if(l->inf==key)
    {
      S_Graf *tmp=l;
      l=l->next;
      delete tmp;
    }
    else
    {
      S_Graf *tmp=l;
      while(tmp)
      {
        // Є наступна вершина
        if(tmp->next)
          if(tmp->next->inf==key)
            // Вона є шуканою
          {
            S_Graf *tmp2=tmp->next;
            tmp->next=tmp->next->next;
            delete tmp2;
          }
          tmp=tmp->next;
      }
    }
```

```
}

// Визначення, чи можна побудувати ейлерівський шлях
int eiler(S_Graf **gr,int num)
{
  int count;
  for(int i=0;i<num;i++)
  {
    count=0;
    S_Graf *tmp=gr[i];
    // Рахуємо степінь вершини
    while(tmp)
    {
      count++;
      tmp=tmp->next;
    }
    // Степінь непарна
    if(count%2==1)return 0;
  }
  // Всі степені парні
  return 1;
}

void eiler_path(S_Graf **gr,int SIZE_0)
{ //Побудова циклу Ейлера
 // Стек для збереження пройдених вершин
 stack <int> S_0;
 // Перша вершина вибирається довільним чином
 int v=start;
 int u,N=0;
 //Поміщення вершини графа в стек
 S_0.push(v);
 //Поки стек не порожній
 while(!S_0.empty())
 {
    // Поточна вершина
    v = S_0.top();
    if(!gr[v]){
  //Виводимо вершину
```

```
      v = S_0.top();
      S_0.pop();
      N++;
      cout<<v<<"  ";
      if(N!=1 && N!=int(SIZE_0/2)+1) cout<<endl<<v<<"  ";
    }
    else
    {
      //Переходимо на наступну вершину
      u=gr[v]->inf;
      S_0.push(u);
      //Видаляємо пройдене ребро
      del(gr[v],u);
      del(gr[u],v);
        }
      }
    }

    int main()
    {
      SetConsoleCP(1251);
      SetConsoleOutputCP(1251);
      // Масив списків суміжності
      S_Graf **graf;

      //Файл с із матрицею суміжності
      ifstream F_graf("graf.txt");

     if( ! F_graf) cout<<"Не можливо відкрити файл із матрицею
  суміжності графа"<<endl;
     // Кількість вершин графа
     int SIZE;
     int SIZE_0=0;
     // Поточне значення прочитаного ребра суміжності
     int zn;
     F_graf>>SIZE;
     graf=new S_Graf*[SIZE];
     for(int i=0;i<SIZE;i++)
        graf[i]=0;
```

```
    // Заповнюємо масив списків суміжності
    for(int i=0;i<SIZE;i++)
      for(int j=0;j<SIZE;j++)
      {
        F_graf>>zn;
        // Формування списку ребер, інцедентних вершині i
        if(zn!=0)
        {
          add(graf[i],j);
          SIZE_0++;
        }
      }
      cout<<"Кількість каменів доміно= "<<int(SIZE_0/2)<<endl;
      if(eiler(graf,SIZE))eiler_path(graf,SIZE_0);
      else cout<<"Граф не є ейлеровим";
      cout<<endl;
      return(0);
  }
```

```
Кількість каменів доміно= 21
5  6
6  4
4  5
5  3
3  6
6  2
2  4
4  3
3  2
2  5
5  1
1  6
6  0
0  4
4  1
1  3
3  0
0  2
2  1
1  0
0  5
Press any key to continue . . .
```

Рис. 2.32: Результат виконання програми

2.2.7 Гамільтонів цикл

Якщо граф має простий цикл, який містить усі вершини графу (по одному разу), то такий цикл називається гамільтоновим циклом, а граф називається гамільтоновим. Гамільтонів маршрут - це маршрут, який містить усі вершини графа по одному разу. Гамільтонові цикл та маршрут, взагалі кажучи, не містять усіх ребер графу.

Задача знаходження гамільтонового циклу отримала свій розвиток як продовження ряду практичних задач, однією із яких є задача комвіявожера, яка передбачає знаходження гамільтонового циклу.

Не всі зв'язані графи мають гамільтонів цикл хоча б тому, що такий граф має бути двозв'язним (тобто граф, який має точки з'єднання, не може мати гамільтонового циклу). Незважаючи на зовнішню подібність формулювань задач про існування ейлерового та гамільтонового циклів, ці задачі принципово різні.

Відповісти на питання, чи має граф гамільтонів цикл, зазвичай, дуже важко. Вивчення достатніх умов наявності в графі гамільтонового циклу – один із важливих напрямків у теорії графів. Інтуїтивно зрозуміло, що граф із багатьма ребрами, достатньо рівномірно розподіленими, з великою ймовірністю має гамільтонів цикл.

Існує ряд достатніх умов існування гамільтонового циклу. Наприклад, граф буде гамільтоновим, якщо степінь будь-якої його вершини задовільняє нерівності (теорема Дірака):

$$deg\ v \geq \frac{n}{2}$$

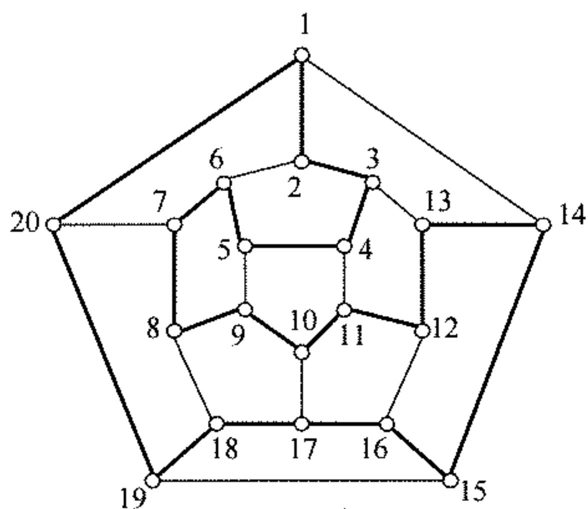


Рис. 2.33: Приклад циклу Гамільтона

Як знайти гамільтонів цикл або переконатись, що його немає? Очевидний алгоритм, який можна застосувати, - це повний перебір усіх можливостей, тобто n! перестановок усіх вершин графу та відповідних перевірок.

Існує алгоритм побудови гамільтонового циклу в графі (якщо він існує) на основі бектрекінгу. Він полягає в наступному. Починаємо з довільної вершини. Будуємо шлях без повторення вершин, доки це можливо. Якщо вдалося пройти всі вершини, то перевіряємо, чи існує ребро, що з'єднує останню та початкову вершини цього шляху. Якщо описаний процес у певний момент неможливо продовжити, то повертаємося на одну вершину назад і намагаємося продовжити побудову шляху (без повторення вершин) іншим способом. Цей алгоритм значно зменшує кількість ітерацій в порівнянні з повним перебором.

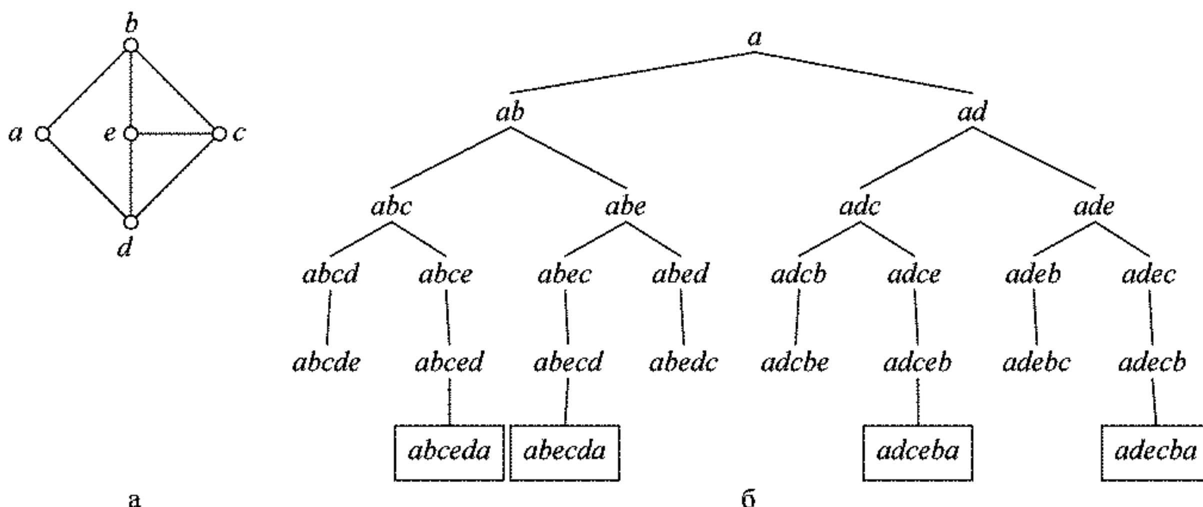


Рис. 2.34: Приклад роботи алгоритму бектрекінгу

2.2.8 Найкоротший шлях

Постановка задачі: для заданого простого зваженого графа $G = (V, E)$ якщо існує кілька шляхів із вершини i до вершини j знайти той із них що має мінімальну вагу. Для простого (незваженого) графа - той, що містить мінімальну кількість дуг. Якщо резцльтатом буде нескінченість, то це означає, такого шляху просто не існує.

Алгоритм Дейкстри

Для заданої вершини i знаходяться довжини найкоротших шляхів до всіх інших вершин. Алгоритм будується на поступовому розростанні шляхів від стартової вершини до всіх інших вздовж дуг мінімальної ваги. Вихідним параметром роботи алгоритму є матриця суміжності із вагами дуг розміром $n \times n$. Якщо між відповідними елементами дуга відсутня, то відповідний елемент матриці рівний нескінченості: присвоюється велике число рівне "машинній нескінченості". Для програмної реалізації в якості нескінченості задається достатньо велике число, яке завідомо більше можливої довжини шляху. Аналогічно до попередніх алгоритмів пошуку знадобиться також масив вершин, які вже були відвідані $visited$. Результатом роботи алгоритму буде вектор d довжин від заданої вершини до всіх інших.

На початку роботи алгоритму у векторі d тільки i - та координата дорівнює 0: шлях від i- тої вершини до самої себе. Всі інші вершини є нескінченими:

$$d_j = \begin{cases} 0, & i = j \\ \infty, & i \neq j \end{cases}$$

Сам алгоритм Дейкстри складається із n ітерацій та полягає в наступному:

1. Серед ще не переглянутих вершин обираємо вершину v із найменшим значенням величини вектора $d(v)$ серед ще не відмічених. Якщо таких вершин кілька, то обираємо довільну із них. На прешому кроці роботи алгоритму обирається стартова вершина, бо для неї, в силу визначення вектора d відстань мінімальна (рівна нулю) та відмічається як переглянута ($true$).
$$d[v] = \min_{p:u[p]=false} d[p]$$
2. Вибрана чином вершина v відмічається як відвідана.
3. Для поточної ітерації із вершини v проводиться **релаксація**: переглядаються всі ребра, що виходять із вершини v. і для кожної такої вершини t_0

алгоритм пробує покращити значення $d[t_0]$. Тобто, якщо довжина поточного ребра len, то процес релаксації можна описати наступним чином:

$$d[t_0] = \min(d[t_0], d[v] + len)$$

4. На цьому поточна ітерація завершується і алгоритм переходить до наступної ітерації..
5. Після виконання n ітерацій всі вершини графа відмічені як пройдені і алгоритм завершує свою роботу.
6. Якщо не всі вершини досяжні із стартової вершини,то знайдені для них значення $d[v]$ так і залишаться нескінченно великими. Це означає, що якщо після декількох послідовних ітерацій алгоритм можна зупинити, як тільки в якості вибраної вершини береться вершина із нескінченною відстанню.

Відновлення маршруту для найкоротших шляхів. По завершенню роботи алгоритму Дейкстри ми маємо вектор d - множину найкоротших шляхів до кожної вершини. Для відновлення маршруту найкоротшого шляху необхідно мати ще один вектор p розміру n, який називають вектором предків. У кожній його координаті p_i зберігається номер вершини, що передує вершині із номером v_i у найкоротшому шляху до v_i . Оскільки для початкової вершини предків немає, то $p_s = 0$. Тут використано адитивність найкоротшого шляху: якщо із найкоротшого шляху до v_i вилучити останню вершину, то отримаємо найкоротший шлях до вершини із номером p_i. Тобто маючи такий вектор предків найкоротший маршрут до вершини v_t можна побудувати із кінця, додаючи у шлях наступну вершину - предка попередньої, поки предком не буде 0:

$$P = \{v_t; p(v_t); p(p(v_t)); p(p(p(v_t))); \ldots; v_s\}$$

Сам вектор p будується наступним чином: при кожній успішній релаксації (коли при порівнянні буде $d_1 + b_{it} < d_i$) записуємо, що наразі предком v_t буде вершина із номером i:

$$p_t = p(t) = i$$

На наступних ітераціях предок v_t може і змінитися, якщо буде знайдений ще коротший шлях.

Приклад 2. 16. *Для графа, приведеного нижче, побудувати матрицю суміжності та реалізувати алгоритм Дейкстри із виводом мінімального шляху та маршруту від заданої вершини до всіх інших вершин графа.*

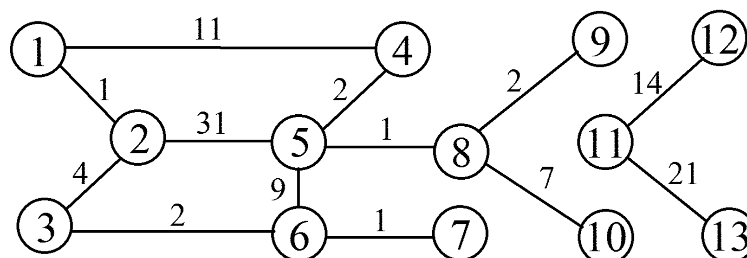


Рис. 2.35: Приклад зваженого графа. Біля кожного ребра приведено його вагу

```
#include <windows.h>
#include <iostream>
using namespace std;
//Кількість вершин графа
const int SIZE_G=13;
bool visited[SIZE_G];
int distance[SIZE_G];

//Функція виводу маршруту
void Pach_G(int *parent,int st, int u)
{
  cout<<u+1;
    while(u!=st)
 {
    u=parent[u];
    cout<<"->"<<u+1;
    }
cout<<endl;
}

//Функція початкової ініціалізації масиву відстаней
// та масиву відвіданих вершин
void Init_G(bool *visited,int *distance, int st)
{
for (int i=0; i<SIZE_G; i++)
{
    distance[i]=INT_MAX;
    visited[i]=false;
```

```
  }
  distance[st]=0;
  }

  //Функція, що реалізує алгоритм Дейкстри
  void Dijkstra(int GR[SIZE_G][SIZE_G], bool *visited, int *
distance, int st)
  {
  int count, index, i, u, m=st+1;
  int parent[SIZE_G];
  for (int i = 0; i < SIZE_G; i++)
  parent[i] = 0;
  for (count=0; count<SIZE_G-1; count++)
  {
  int min=INT_MAX;
  for (i=0; i<SIZE_G; i++)
  if (!visited[i] && distance[i]<=min)
      {
      min=distance[i];
      index=i;
      }
  u=index;
  visited[u]=true;
  for (i=0; i<SIZE_G; i++)
      if (!visited[i] && GR[u][i] && distance[u]!=INT_MAX
                      && distance[u]+GR[u][i]<distance[i])
      {
   distance[i]=distance[u]+GR[u][i];
   parent[i] = u;
      }
  }
  cout<<"Довжина шляху та маршрут
            від стартової до всіх інших:\t\n";
  for (i=0; i<SIZE_G; i++)
  {if (distance[i]!=INT_MAX)
  {
    cout<<m<<" -> "<<i+1<<" = "<<distance[i]<<"\t\t";
    Pach_G(parent, st,  i);
  }
```

```
      else cout<<m<<" -> "<<i+1<<"\t \t "<<"Маршрут
   недоступний"<<endl;
        }
    }

    void main()
    {
     SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        bool visited[SIZE_G];
        int distance[SIZE_G];
    //Ініціалізація матриці суміжності
    int start, GR[SIZE_G][SIZE_G]={
    {0, 1, 0, 11, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 4, 0, 31, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 4, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0},
    {11, 0, 0, 0, 2, 0,0, 0, 0, 0, 0, 0, 0},
    {0, 31, 0, 2, 0, 9, 0, 1, 0, 0, 0, 0, 0},
    {0, 0, 2, 0, 9, 0, 1, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 0, 0, 0, 2, 7, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 14, 21},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 14, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 21, 0, 0},
    };
    cout<<"Стартова вершина >> "; cin>>start;
    Init_G( visited, distance, start-1);
    Dijkstra(GR, visited, distance, start-1);
    system("pause");
    }
```

```
Стартова вершина >> 1
Довжина шляху та маршрут від стартової до всіх інших:
1 -> 1 = 0              1
1 -> 2 = 1              2->1
1 -> 3 = 5              3->2->1
1 -> 4 = 11             4->1
1 -> 5 = 13             5->4->1
1 -> 6 = 7              6->3->2->1
1 -> 7 = 8              7->6->3->2->1
1 -> 8 = 14             8->5->4->1
1 -> 9 = 16             9->8->5->4->1
1 -> 10 = 21            10->8->5->4->1
1 -> 11         Маршрут недоступний
1 -> 12         Маршрут недоступний
1 -> 13         Маршрут недоступний
Press any key to continue . . .
```

Рис. 2.36: Результат виконання програми

Алгоритм Флойда - Воршола

Алгоритм дозволяє побудувати матрицю найкоротших шляхів між усіма парами вершин графів. Цей алгоритм є більш загальним у порівнянні із алгоритмом Дейкстри, бо дозволяє знаходити найкоротші шляхи між будь-якими двома вершинами графа.

Метод грунтується на тому факті, що в графі з додатними вагами ребер будь-який неелементарний, що містить більше одного ребра, найкоротший шлях , складається із інших найкоротших шляхів.

В алгоритмі використовується матриці A розміром $n\times$, за допомогою якої обчислюються довжини найкоротших шляхів. Елемент $A[i, j]$ рівний відстані від вершини i до внршини j. Значення елемента є скінченим, якщо такий шлях існує, та нескінчeності, якщо шлях між вершинами відсутній.

Ідея алгоритму полягає в наступному: нехай для трьох вершин i, j таk відомі відстані між ними. Якщо виконується нерівність

$$A[i, k] + A[k, j] < A[i, j]$$

то то шлях з вершини i до вершини j необхідно замінити шляхом $i \to k \to j$. Такі заміни виконуються регулярно в процесі виконання даного алгоритму.

На початку роботи алгоритму визначаємо матрицю відстаней A_0 та матрицю послідовності вершин S_0. Діагональні елементи матриць рівні нулю = елементи не використовуються для виконання обчислень. На початку роботи алгоритму параметр $k = 1$.

Основний крок по k розглядає можливість заміни, описаної вище, до всіх

елементів $A[i, j]$ матриці S_{k-1}. Якщо виконується нерівність:

$$A[i, k] + A[k, j] < A[i, j], (i \neq k)(j \neq k)(i \neq j)$$

то виконуються наступні операції

- створюємо матрицю A_k заміною в A_{k-1} елемента $A[i, j]$ на суму $A[i, k] + A[k, j]$;
- створюємо матрицю S_k заміною в S_{k-1} елемента $S[i, j]$ на k;
- приймаємо $k = k + 1$ та повторюємо крок k

Таким чином, алгоритм виконує n ітерацій і після i-тої ітерації матриця A буде містити довжини найкоротших шляхів між будь-якими діома парами вершин при умові, що ці шляхи проходять через вершини від першої до i-тої. На кожній ітерації перебираються всі пари вершин та шлях між ними скорочується за допомогою i-тої вершини.

Необхідно зауважити, що якщо граф неорієнтований, то всі матриці, що отримуються в результаті роботи алгоритму, є симетричні відносно головної діагоналі і тому достатньо обчислювати тільки елементи, що знаходяться вище головної діагоналі. Якщо граф задано матрицею суміжності, то час виконання алгоритму має порядок $O(n^3)$, оскільки містить три вкладені цикли.

Для відновлення найкоротшого шляху між двома вершинами необхідно крім матриці відстаней зберігати також матрицю нащадків, яка для кожної пари вершин містить номер фази, на якій був отриманий найкоротший шлях між ними. Номер фази є "середньою"вершиною шуканого найкоротшого шляху: просто необхідно знайти найкоротший шлях між вершинами i та $p[i][j]$, а також між $p[i][j]$ та j. Звідсіля слідує простий рекурсивний процес відновлення найкоротшого шляху.

Нижче приведено приклад реалізації алгоритму

```
#include <iostream>
#include <windows.h>
using namespace std;
const int n = 9;

int main()
{
  SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
```

```
int inf = 9999;
int from = 1, to = 6;

int  a[n][n] = {
  { 0, inf, inf, inf, inf, inf, 1, 3, 20 },
  { 5, 0, 5, 6, inf, inf, inf, 3, inf },
  { inf, inf, 0, inf, inf, inf, 2, inf, inf },
  { inf, inf, inf, 0, 2, inf, inf, inf, 6 },
  { inf, 14, inf, inf, 0, 9, 6, inf, inf },
  { inf, inf, inf, inf, inf, 0, inf, inf, inf },
  { inf, inf, inf, inf, inf, inf, 0, inf, 1 },
  { inf, inf, inf, inf, 4, inf, 4, 0, inf },
  { inf, inf, inf, inf, inf, inf, inf, inf, 0 },
};

int c[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    c[i][j] = i;

for (int k = 0; k < n; k++)
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (a[i][k] + a[k][j] < a[i][j])
      {
        a[i][j] = a[i][k] + a[k][j];
        c[i][j] = c[k][j];
      }
      for(int i=0; i<9;i++)
      {
        from=i;
        cout << "Найкоротший шлях від " << from + 1 << " до " <<
  to + 1 << ": " << to + 1 << "  ";
        int t = c[from][to];
        while ( t != from) {
          cout << t + 1 << "  ";
          t = c[from][t];
        }
        cout << from + 1 << endl;
```

```
            cout << "Довжина найкоротшого шляху = " << a[from][to]
   << endl;
        }
        return 0;
}
```

```
Найкоротший шлях від 1 до 7: 7  1
Довжина найкоротшого шляху = 1
Найкоротший шлях від 2 до 7: 7  1  2
Довжина найкоротшого шляху = 6
Найкоротший шлях від 3 до 7: 7  3
Довжина найкоротшого шляху = 2
Найкоротший шлях від 4 до 7: 7  5  4
Довжина найкоротшого шляху = 8
Найкоротший шлях від 5 до 7: 7  5
Довжина найкоротшого шляху = 6
Найкоротший шлях від 6 до 7: 7  6
Довжина найкоротшого шляху = 9999
Найкоротший шлях від 7 до 7: 7  7
Довжина найкоротшого шляху = 0
Найкоротший шлях від 8 до 7: 7  8
Довжина найкоротшого шляху = 4
Найкоротший шлях від 9 до 7: 7  9
Довжина найкоротшого шляху = 9999
Press any key to continue . . .
```

Рис. 2.37: Результат роботи програми

Частина II

Структура даних

Розділ 3

Поняття алгоритму і його властивості

3.1 Поняття алгоритму

3.1.1 Поняття алгоритму

Поняття алгоритму в інформатиці є фундаментальним, тобто таким, яке не визначається через інші, більш прості поняття.

Слово «алгоритм» походить від «algorithmi» — латинської форми написання імені великого математика аль-Хорезмі, який сформулював правила виконання арифметичних дій. Тому спочатку під алгоритмом розуміли тільки правила виконання чотирьох арифметичних дій над багатоцифровими числами в десятковій системі числення. Алгоритм може являти собою деяку послідовність обчислень, а може — послідовність дій нематематичного характеру. Але, у кожному разі, перед його складанням повинні бути чітко визначені початкові умови й те, що має бути одержано.

Алгоритм – послідовність дій, спрямованих на розв'язання поставленої задачі.

Алгоритм — кінцева послідовність кроків у рішенні завдання, що приводить від вихідних даних до необхідного результату.

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

Під виконавцем алгоритму ми розуміємо будь-яку істоту (живу чи неживу), яка спроможна виконати алгоритм. Кожний алгоритм створюється з розрахунку на конкретного виконавця, тому можна сказати, що алгоритм — це точні розпорядження (указівки, команди, операції, інструкції) виконавцеві здійснити послідовність дій, спрямованих на розв'язання поставленої задачі.

Алгоритм складається із команд - окремих указівок виконавцеві викона-

ти деякі конкретні дії. Команди алгоритму виконуються одна за одною, і на кожному кроці відомо, яка команда повинна виконуватися. Почергове виконання команд за кінцеве число кроків приводить до розв'язання задачі. Для того щоб виконавець міг розв'язати задачу за заданим алгоритмом, він повинен уміти виконувати кожну з дій, що вказується командами алгоритму.

Виконавцями алгоритмів можуть бути людина, тварини, автомати, тобто ті, хто розуміє та може виконати вказівки алгоритму.

Система команд виконавця — сукупність команд, які можуть бути виконані виконавцем; кожна команда алгоритму входить до системи команд виконавця.

В основі роботи автоматичних пристроїв лежить положення, що найпростіші операції, на які розпадається процес розв'язання задачі, може виконати машина, яка спеціально створена для виконання окремих команд алгоритму і виконує їх у послідовності, вказаній в алгоритмі.

Розробляти алгоритми може тільки людина. Виконують алгоритми люди й усілякі пристрої — комп'ютери, роботи, верстати, супутники, складна побутова техніка й навіть деякі дитячі іграшки.

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

3.1.2 Властивості алгоритму

Виконуючи алгоритм, виконавець може не вникати в зміст того, що він робить, і разом із тим отримати потрібний результат, тобто виконавець діє формально. Тому для правильної побудови алгоритму необхідно знати систему команд виконавця, бути впевненим, що виконання алгоритму завершиться за кінцеве число кроків. Тому кажуть про деякі загальні властивості алгоритмів.

Зрозумілість. Щоб виконавець міг досягти поставленої перед ним мети, використовуючи даний алгоритм, він повинен уміти виконувати кожну його вказівку, тобто розуміти кожну з команд, що входять до алгоритму. Зрозумілість - це властивість алгоритму, що полягає в тім, що кожен алгоритм повинен бути написаний у командах, зрозумілих даному виконавцю.

Дискретність. Алгоритм розв'язання задачі повинен складатися з послідовності окремих кроків — відокремлених одна від одної команд (указівок), кожна з яких виконується за кінцевий час. Тільки закінчивши виконання однієї команди, виконавець переходить до виконання іншої.

Визначеність (однозначність). Кожна команда алгоритму однозначно визначає дії виконавця і не припускає подвійного тлумачення. Суворо визначеним є й порядок виконання команд.

Формальність. Будь-який виконавець, який володіє заданою системою команд, може виконати заданий алгоритм, не вникаючи в суть задачі.

Скінченність алгоритму. Послідовність команд, які потрібно виконати, має бути скінченою.

Ефективність алгоритму. Для розв'язанні задачі може використовуватися обмежений обсяг комп'ютерних ресурсів.

Результативність. Виконання алгоритму не може закінчуватися невизначеною ситуацією або зовсім не закінчуватися. Будь-який алгоритм передбачає, що його виконання при допустимих початкових даних за кінцеве число кроків приведе до очікуваного результату.

Масовість. Алгоритм має передбачати можливість зміни початкових (вхідних) даних у деяких допустимих межах і можливість використання його для розв'язання задач одного класу (універсальність алгоритму). Отож, під масовістю алгоритму мається на увазі можливість його застосування для вирішення великої кількості однотипних завдань.

Саме через ці властивості часто дається визначення поняття алгоритму як скінченної однозначно визначеної послідовності операцій, формальне виконання якої приводить до розв'язання певної задачі за кінцеве число кроків.

3.1.3 Способи представлення алгоритмів

Існує кілька методів запису алгоритмів, вибір яких залежить від виконавця та того, хто його задає.

Першій спосіб - це **словесний опис алгоритму**. Це, по суті, звичайна мова, але з ретельним відбором слів і фраз, що не допускають зайвих слів, двозначностей і повторень. Доповнюється мова звичайними математичними позначеннями й деякими спеціальними відношеннями. Алгоритм описується у вигляді послідовності кроків. На кожному кроці визначається склад виконуваних дій і напрямок подальших обчислень. При цьому, якщо на поточному кроці не вказується який крок повинен виконуватися наступним, то здійснюється перехід до наступного кроку.

Другий спосіб - це подача алгоритму у вигляді **таблиць, формул, схем, малюнків** тощо. Наприклад, всіх вас вчили в дитячому садочку правилам поведінки на дорозі. І найкраще діти, вочевидь, сприймають алгоритм, що поданий у вигляді схематичних малюнків. Дивлячись на них, дитина, а потім і доросла людина, відпрацьовує ту лінію поведінки, що їй пропонується. Аналогічно можна навести приклади алгоритмів, що записані у вигляді умовних позначок на купленому товарі, щодо його користування (заварювання чаю, прання білизни тощо). У математиці наявність формул дозволяє розв'язати задачу, навіть "не використовуючи слів".

Третій спосіб - запис алгоритмів за допомогою **блок-схеми**. Цей метод був запропонований в інформатиці для наочності представлення алгоритму за допомогою набору спеціальних блоків. Основні з цих блоків наступні:

Четвертий спосіб - навчальні **алгоритмічні мови (псевдокоди)**. Ці мови мають жорстко визначений синтаксис і вже максимально наближені до машинної мови (мови програмування). Але створені вони з навчальною метою, тому мають зрозумілий для людей вигляд. Таких псевдокодів зараз існує велика кількість, починаючи з графічних середовищ "Алгоритміка "Роботландія "Лого-світи "Черепашка"тощо і закінчуючи текстовими "національними"реалізаціями алгоритмічних мов, подібних до Бейсіка, Паскаля. Ці псевдокоди мають програмну реалізацію і дуже широко застосовуються на етапі навчання основам програмування.

П'ятий спосіб максимально наближений до комп'ютера - це **мови програмування**. Справа в тому, що найчастіше у практиці виконавцем створеного людиною алгоритму являється машина (комп'ютер) і тому він повинен бути написаний мовою, зрозумілою для комп'ютера, тобто мовою програмування.

При побудові алгоритму часто виникає необхідність пояснити виконавцю деякі складні дії, якщо їх виконання не входить в систему команд виконавця. Наприклад, перший раз даючи дитині завдання пришити ґудзик до плаття, їй треба пояснити, як необхідно підбирати нитки для шиття, як вдягати нитку в голку, як тримати голку та ґудзик при роботі, яка різниця між пришиванням ґудзика до тоненької сорочки та товстої куртки (в другому випадку ґудзик робиться на "ніжці"). В подальшому такі пояснення будуть вже зайві, бо алгоритм "пришивання ґудзика"стає вже командою в системі команд виконавця "дитина".

Взагалі кажучи, кожна дія людини (якщо вона її може виконати) може вважатися командою її "системи команд хоча колись, на етапі навчання, учитель або хтось інший ретельно пояснював, яку треба виконати послідовність дій, щоб досягти поставленої мети.

Узагальнюючи сказане, можна сказати, що кінець кінцем кожну задачу можна вважати окремою командою виконавцю, якщо його навчено виконувати поставлене завдання. Якщо ж виконавець не знає, як розв'язувати запропоновану задачу, виникає потреба розкласти її на такі підзадачі, що являються "посильними"для виконання, тобто входять до системи команд виконавця. Продовжуючи цей процес, остаточно отримують алгоритм, що складається з простих команд, зрозумілих виконавцю, або остаточно переконуються, що дана задача непосильна для вибраного виконавця, тому що в його системі команд не існує необхідних для цього команд. Наприклад, як би ми не деталізували алгоритм побудови багатоповерхової будівлі для дитини, задача кінець кінцем являється для неї непосильною.

Запропонований підхід до конструювання алгоритмів називається методом покрокової деталізації зверху вниз. Вочевидь, що при такому підході кожна операція остаточно буде подана у вигляді лише одного з трьох типів базових структур алгоритмів - лінійної (в літературі часто ця структура називається слідування), розгалуження або повторення (циклу). Степінь деталізації алгоритму при цьому сильно залежить від того, на якого виконавця його орієнтовано.

Алгоритми, що складаються для розв'язування окремих підзадач основної задачі, називаються допоміжними. Вони створюються при поділі складної задачі на прості або при необхідності багаторазового використання одного ж того набору дій в одному або різних алгоритмах.

Допоміжний алгоритм повинен мати тільки один вхід та один вихід, причому того, хто користується ним, зовсім не цікавить, як реалізований цей алгоритм. Головне, щоб всі команди, що входять до складу допоміжного алгоритму входили до системи команд обраного виконавця. Зверніть увагу ще на те, що в реальному житті допоміжні алгоритми можуть виконувати, навіть, зовсім інші виконавці. Наприклад, якщо батьки вдома вирішили зробити ремонт, це не значить, що вони самостійно повинні зробити собі шпалери та клей. Алгоритми виробництва матеріалів існують і їх хтось виконує, а ми тільки користуємось результатами їх роботи.

Таким чином, можна вважати допоміжний алгоритм своєрідним "чорним ящиком на вхід якого подаються деякі вхідні дані, а на виході ми отримуємо очікуваний результат. Головне чітко домовитись про правила оформлення вхідних даних та вигляд результату. Невиконання домовленостей може привести до збою у виконанні допоміжного алгоритму або до отримання неочікуваного результату.

Описаний метод послідовної деталізації лежить в основі технології структурного програмування і широко застосовується при використанні таких мов програмування, як Бейсік, Паскаль, C, C++ та інших.

При описуванні програми для комп'ютера мовами високого рівня допоміжні алгоритми реалізовуються у вигляді підпрограм. Правила опису, звернення до них та повернення в точку виклику визначаються конкретною мовою програмування. Для зручності часто використовувані підпрограми можна об'єднувати в бібліотечні модулі і при необхідності підключати їх в свої програми.

3.1.4 Базові структури алгоритмів

Базові структури алгоритму — це структури, за допомогою яких створюється алгоритм для розв'язання певної задачі. Існують три основні (базові) алгоритмічні структури, або три основні типи алгоритмів: лінійний, розгалужений та циклічний.

Лінійний алгоритм (послідовне виконання, структура слідування) — це алгоритм, який забезпечує отримання результату шляхом одноразового виконання послідовності дій, незалежно від вхідних даних і проміжних результатів. Дії в таких алгоритмах виконуються послідовно, одна за однією, тобто лінійно. Розглянемо алгоритм такої задачі: за даними довжинами катетів a, b обчислити площу прямокутного трикутника.

Розгалужений алгоритм (умова, структура вибору) — у класичному варіанті ця структура розглядається як вибір дій у разі виконання або невиконання заданої умови. Галуження бувають повними і неповними.

Повне галуження — це галуження, в якому певні дії визначені й у разі виконання, і в разі невиконання умови. Неповне галуження — це розгалуження, в якому дії визначені тільки у разі виконання (або у разі невиконання) умови.

Якщо логічний вираз, то команда 1, інакше команда 2. Серія команд – це декілька команд.

Циклічний алгоритм (цикл, структура повторення) — це алгоритм, у якому передбачено повторення деякої серії команд. За допомогою цієї структури описуються однотипні дії, що повторюються декілька разів. Такі алгоритми забезпечують виконання довгої послідовності дій, записаних порівняно короткою послідовністю команд. Саме використання циклів дозволяє у повній мірі реалізувати швидкодію комп'ютерів.

Циклом називають процес повторення дій. Циклічні алгоритми забезпечують повторне виконання деяких команд скінчену кількість разів.

Доки логічний вираз, виконати команди

Основна особливість базових алгоритмічних структур — це їх повнота, тобто цих структур достатньо для створення найскладнішого алгоритму.

Розглянемо алгоритм задачі: підрахувати кількість цифр у записі натурального числа n. Ідея розв'язання: із заданого числа вибирати із молодшого розряду цифру за цифрою, доки число не вичерпається, тобто не стане рівним нулю; кожну цифру враховувати, тобто підраховувати їх кількість.

3.1.5 Аналіз складності алгоритму

При побудові алгоритму розв'язку задачі необхідно забезпечити виконання двох протилежних умов:

1. бути простим для розуміння, реалізації та відлагодження;
2. бути ефективним в сенсі затрат машинного часу та об'єму необхідної оперативної пам'яті ЕОМ.

Такий підхід до оцінки ефективності залишається актуальним не дивлячись на постійний ріс як швидкодії ЕОМ, так і об'єму доступної оперативної пам'яті. Вибір пріоритету для першого чи другого моментів визначається особливістю самої задачі. Якщо час виконання програми не є критичним, то віддається перевага як-правило першій умові, оскільки вартість розробки (як часова, так і фінансова) програми набагато вища вартості машинного часу, що витрачається на розв'язок задачі.

Якщо для розв'язку задачі необхідна затрата значних обчислювальних ресурсів, або час розв'язку обмежений деякими зовнішніми подіями, так як це є в системах реального часу, то ефективність алгоритму виступає на перший план. За звичай ефективні алгоритми є більш складними в реалізації.

Метою аналізу ефективності алгоритму є оцінка об'єму пам'яті або часу роботи, які необхідні алгоритму для обробки конкретного набору даних. При аналізі із задачею пов'язується деяке число, яки відображає міру кількості вихідних даних. В якості такої міри може виступати розмір масиву, кількість вершин чи дуг графу, кількість записів у файлі і т. д. Час виконання програми $T(n)$ залежить від розміру n, і при цьому одиниці вимірювання $T(n)$ не конкретизуються. Традиційно в найгіршому випадку приймають час виконання $T(n)$ як міру ефективності алгоритму.

Час, що витрачається на виконання алгоритму як функція розміру задачі, називають часовою складнісю алгоритму. Поведінка складності в границі при рості розміру задачі n асимптотичною складністю по часу. Аналогічно визначається ємнісна складність та асимптотична ємнісна складність. Асимптотична ємнісна складність визначає розмір задачі, яку можна розв'язати за допомогою даного алгоритму.

Якщо алгоритм A має час виконання $T_A(n) = c \cdot n^2$, а алгоритм B $T_B(n) = k \cdot d^n$, де c, d, k - деякі константи, то в залежності від їх співвідношення залежить від величини n. Як правило для малих n більш ефективним є алгоритм B, а для великих n ефективнішим буде алгоритм B. Це зумовлено тим, що функції $T_A(n)$ та $T_B(n)$ мають різні швидкості росту.

Для опису швидкості росту функції використовується O- символіка. Так, $T(n) = O(n^2)$ означає, що час виконання програми $T(n)$ має порядок $O(n^2)$ і розуміється, що існують додатні константи c та n_0 такі, що для всіх n, більших чи рівних n_0, виконується нерівність $T(n) \leq c \cdot n^2$. Аналогічно $T(n)$ має порядок $O(f(n))$, якщо існують константи c та n_0 такі, що для всіх n, більших чи рівних n_0, виконується нерівність $T(n) \leq c \cdot f(n)$, тоді $f(n)$ визначає степінь росту. $f(n)$ ще називають верхньою границею складності задачі.

Для прикладу визначимо складність алгоритму сортування масиву методом бульбашки. Основою алгоритму є подвійний цикл аналізу елементів масиву.

```
for (i=0; i<nn-1; i++)
{
for (j=nn-1; j>i; j--)
if (A[j]<A[j-1])
{
  k=A[j];
  A[j]=A[j-1];
  A[j-1]=k;
}
```

Для перестановки елементів (тіло циклу) необхідно затратити певний фіксований час. Для кожного фіксованого значення i, що задається зовнішнім циклом по i, внутрішній цикл по j виконується $(n-i)$ разів, азагальна цільність кроків визначається сумою

$$\sum_{i=0}^{n-2}(n-i) = \frac{n^2-n}{2}$$

Це означає, що в найгіршому випадку кількість перестановок рівна цьому значенню, а часова складність сортування методом бульбашки - $O(n^2)$.

При обчисленні часу виконання програми з використанням O-символіки використовують правила сум та добутків.

Правило сум: якщо час виконання двох фрагментів програми $T_1(n)$ та $T_2(n)$ має порядок $O(f_1(n))$ та $O(f_2(n))$ відповідно, то час послідовного виконання цих фрагментів обчислюється як $T_1(n)+T_2(n) = O(f_1(n))+O(f_2(n))$ та має порядок $O(max(f_1(n), f_2(n)))$

Правило добутків: $T_1(n) \cdot T_2(n) = O(f_1(n) \cdot f_2(n))$

З використанням цих правил можна визначити час виконання операторів базових структур програми.

1. Час виконання послідовності операторів визначається за правилом сум: тобто визначається найбільшим часом виконання одного оператора в даній послідовності.
2. Час виконання операторів розгалуження визначається найбільшим часом виконання однієї із віток.

3. Час виконання циклу визначається як добуток часу виконання тілу циклу на кількість повторів циклу.

При аналізі складності алгоритмів вводять поняття детермінованих та недетермінованих алгоритмів. В кожний момент часу виконання алгоритм знаходиться в певному визначеному стані. Стан алгоритму визначається сукупністю значень даних, що обробляються та виконуваною операцією. Властивість детермінованості означає, що для кожного існуючого на даний момент стану або існує точно один наступний стан, або не існує жодного - це кінцевий стан.

Нехай тепер для довільного стану алгоритму може бути більше одного наступного. Коли детермінований алгоритм доходить до такого місця, де необхідно вибрати одну із альтернатив досліджує їх послідовно одна за одною та вибирає серед них одну-єдину. Недетермінований алгоритм переглядає всі альтернативи не послідовно, а всі одночасно, копіюючи самого себе. Ці копії, в свою чергу, можуть породжувати нові копії і т. д. Всі копії продовжують виконуватися одночасно до тих пір, поки одна чи декілька альтернатив не знайдуть розв'язок, після чого зупиниться виконання всіх копій.

Всі задачі, які можна розв'язати за допомогою детермінованих алгоритмів за поліноміальний час, утворюють клас P - задач (від англійського Роіїпотіаі) та вважаються такими, які можуть бути легко розв'язані.

Задача, складність якої не менша f^n (тут f - константа або поліном від n)вважається експоненційною та відноситься до класу E. При невеликих значеннях n експоненційний алгоритм може бути більш швидким, ніж поліноміальний. При великих значеннях n відмінність між цими алгоритмами є значною.

Є задачі, які не попадають ні в клас P, ні в клас E.

Розділ 4

Рекурсивні алгоритми

4.1 Поняття рекурсії

Рекурсією називається такий спосіб організації обробки даних, при якому програма (функція) викликає сама себе безпосередньо, або з інших програм (функцій). Функція називається рекурсивною, якщо під час її роботи відбувається виникає її повторний виклик безпосередньо, або опосередковано через ланцюжок викликів інших функцій. Як правило, в основі рекурсивного алгоритму лежить рекурсивне визначення якогось поняття, тобто визначення, яке задає деякий обє'кт у термінах більш простого випадку цього ж об'єкту.

Розрізняють *пряму* і *непряму* рекурсії. Функція називається прямо рекурсивною, якщо містить у своєму тілі виклик самої себе. Якщо ж функція викликає іншу функцію, що у свою чергу викликає першу, то така функція називається непрямо рекурсивною.Рекурсивні функції найчастіше використовуються для компактної реалізації рекурсивних алгоритмів.

Основою для розробки рекурсивних алгоритмів служать рекурентні співвідношення (формули), що встановлюють залежності між результатами якихнебудь дій (операцій) на n-ному кроці від результатів аналогічних дій (операцій), отриманих на попередньому $(n-1)$-ому кроці:

Класичні приклади використання рекурсії — реалізація операції піднесення до степеня і обчислення факторіалу числа. Ці приклади носять тільки ілюстративний характер через їхню зручність для пояснення поняття рекурсії, однак практично не дають виграшу в програмній реалізації порівняно з ітераційним способом розв'язання цих задач.

Наприклад, про факторіал числа N можна сказати, що

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

$$n! = (n-1)! \cdot n$$

При цьому $1! = 1$.

Приклад використання рекурсивної процедури для організації обчислення $n!$:

```
int factor(int n)
{
  int D=1;
  if (n==1) D=D*1;
  else D=n*factor(n-1);
  return D;
}
```

Важливим моментом є той факт, що в наведеному прикладі рекурсивний виклик стоїть в середині оператора умови. Це є необхідною умовою того, щоби рекурсивний процес рано чи пізно завешився (рекурсія не продовжувалася до нескінченості). Важливим також є той факт, що функція викликає себе із іншим значенням параметра: не тим, із яким сама вона була викликана.

Схематично такий процес можна зобразити наступним чином: якщо випадок найпростіший, то розвязати його напряму, інакше виконувати рекурсивний виклик до настання найпростішого випадку.

Якщо, наприклад, викликати функцію $factor$ зі значенням параметра $n = 4$: $factor(4)$, то схематично рекурсивний процес можна зобразити наступним чином:

Функція $factor$ викликається зі значенням параметра $n = 4$. Вона, в свою чергу, містить виклик функції $factor$ зі значенням параметра $n = 3$. Фактично в оперативній памяті компютера створюється ще одна функція $factor$ і до завершення її роботи попередня функція роботу не закінчить. Процес рекурсивних викликів завершиться при значенні параметра $n = 1$: в цей момент виконаються всі 4 функції.

Цей процес можна зобразити у вигляді наступної діаграми, яка показує обидві фази функції обчислення факторіалу із використанням рекурсії.

В рекурсивних функціях можна виділити два процеси: рекурсивне занурення функції у себе, що відбувається доти, доки параметр не досягне деякого граничного значення, та рекурсивне повернення з підпрограми, що відбувається доти, доки параметр не сягне початкового значення. Після виклику функції з параметром n виконується ще $n-1$ викликів, і загальна кількість незавершених викликів сягає n. Величина, що характеризує максимальну кількість незавершених рекурсивних викликів, називається глибиною рекурсії.

Рекурсивний виклик може бути непрямим. В цьому випадку функція A звертається до себе опосередковано, шляхом виклику іншої B, в якій місти-

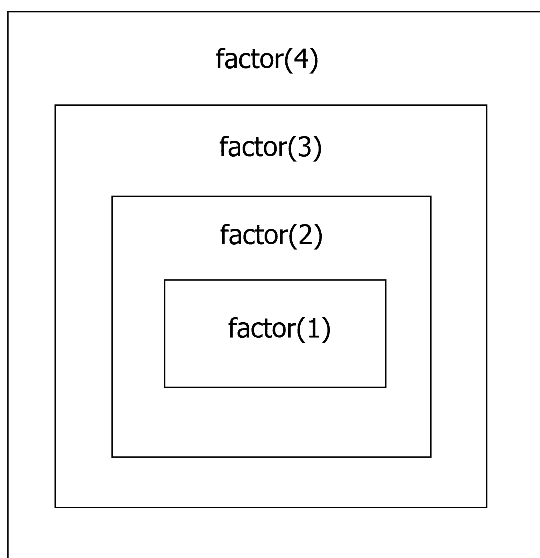


Рис. 4.1: Виконання рекурсивної функції $factor(4)$

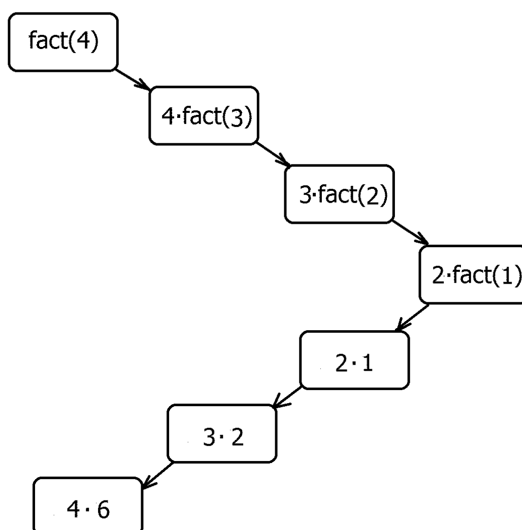


Рис. 4.2: Діаграма обчислення факторіалу із виконання рекурсивної функції $factor(4)$

ться звернення до першої, створючи рекурсивний ланцюжок. Умова завершення в такій рекурсії може бути в одній або всіх функціях.

Важливим є те, що перша описана функція повинна викликати ще не описану функцію. Для цього необхідно використати опис функції B до її вико-

ристання.

Приклад 1. 17. *Рекурсивний алгоритм обчислення значення виразу*

$$a = \frac{x^n}{n!}$$

```
#include <iostream>
using namespace std;
double pow(int, int); // опис функції
double calc(int x, int n)
{
  return pow(x, n)/n;
}
double pow(int x, int n)
{
  if (n == 1) return x;
  else return x*calc(x, n - 1);
}
int main()
{
  int n, x;
  cout << "n = "; cin >> n;
  cout << "x = "; cin >> x;
  double a = calc(x, n);
  cout << a<<endl;
  return 0;
}
```

4.2 Рекурсивний алгоритм сортування

Класичним прикладом використання рекурсії є також швидке сортування — один з найефективніших методів сортування. Загальний алгоритм швидкого сортування може бути реалізований наступним чином:

- в масиві вибираємо дечкий опорний елемент a_i (як правило центральнпий);
- переміщуємо всі елементи, менші чи рівні a_i, вліво від нього, а більші a_i - вправо від нього;
- результатом є масив, що складається із двох підмасивів: лівий менший або дорівнює правому;
- застосувавши цей алгоритм до лівого і правого підмасивів (поки вони складаються із більше ніж двох елементів) отримаємо повністю відсортовану послідовність.

Приклад 2. 18. *Відсортувати одновимірний масив A за зростанням , використовуючи рекурсивну функцію швидкого сортування.*

В заданому масиві A виконаємо розділення елементів згідно наступного алгоритму:

- виберемо в масиві центральний елемент m, відносно якого будемо здійснювати розподіл;
- введемо індекси і та j, які на початку алгоритму вказують, відповідно, на лівий та правий елементи послідовності;
- будемо рухатися із кроком 1 за допомогою індексу і у напрямку до кінця масиву, поки не знайдемо елемент $\mathrm{a_i} \geq \mathrm{c}$. Аналогічним чином будемо рухатися за допомогою індексу j від кінця масиву до початку, доки не знайдемо $\mathrm{a_i} \leq \mathrm{c}$;
- якщо $i \leq j$, міняємо елементи a_i та a_j місцями і продовжуємо рухати і та j за тими ж правилами:
- повторюємо попередній крок, доки $i \leq j$.

```
#include <iostream>
#include <windows.h>
using namespace std;
```

```
void QuickSort (int *arr, int a, int b)
{
  int temp;
  int i=a, j=b;
   int m = (b-a)/2+a ;
  int x = * (arr+m);
  do
  {
    while (i<=b && * (arr+i) < x) i++;
    while (j>=a && * (arr+j) > x) j--;
    if (i <= j)
    {
      if (i < j)
      {
        temp=* (arr+i);
        * (arr+i) =* (arr+j);
        * (arr+j) =temp;
      }
      i++;
      j--;
    }
  } while (i <= j);
  if (i < b) QuickSort (arr, i,b);
  if (a < j) QuickSort (arr,a,j);
}

//
-----------------------------------------------------------------------

void main()
{
   SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
  int a[] = {2, 7, 6, 9, 45, 4, 3, 67, 104, 1, 99, 72, 43, 8,
4, 28, 100};
  int n = sizeof(a)/sizeof(int);
  QuickSort(a,0, n-1);
  cout << "\n Результат швидкого сортування масиву" << endl;
```

```
    for (int i = 0; i<n; i++)
      cout << a[i] <<' ';
  }
```

```
Результат швидкого сортування масиву
1 2 3 4 4 6 7 8 9 28 43 45 67 72 99 100 104 Press any key to continue . . .
```

Рис. 4.3: Скрін-шот виконання програми

Розділ 5

Алгоритми обробки структур даних

5.1 Методи сортування

Сортуванням називають впорядкування елементів множини за зростанням чи спаданням.Із впорядкованими множинами простіше працювати, ніж із невпорядкованими: легше знаходити необхідні елементи, видаляти чи вставляти нові. Сортування використовується при трансляції програм, при організації даних на зовнішніх носіях, при створенні каталогів, бібліотек, баз даних і т.д.

Основною вимогою до алгоритмів сортування, як і до будь-яких інших алгоритмів, є вимоги по мінімізації часу виконання та по використанню машинної пам'яті. Це передбачає, що сортування елементів масиву виконується на місці , без їх передачі в результуючий масив.

Хорошою мірою ефективності алгоритму по часу виконання може бути кількість необхідних порівнянь ключів C, що визначають процес сортування та кількість пересилань M,які визначаються кількістю елементів n. Найпростішими методами сортування є прямі методи сортування, які вимагають $O(n^2)$ порівнянь ключів.

Швидші(покращені) методи сортування в найкращому випадку вимагають порядку $O(n \cdot log_2 n)$ порівнянь

Прямі методи сортування, що не вимагають додаткової пам'яті, можна розділити на три категорії:

1. сортування методом прямого включення:
2. сортування методом прямого вибору;
3. сортування методом прямого обміну (метод бульбашки).

Швидкі методи сортування, як правило, є покращенням прямих методів сортування

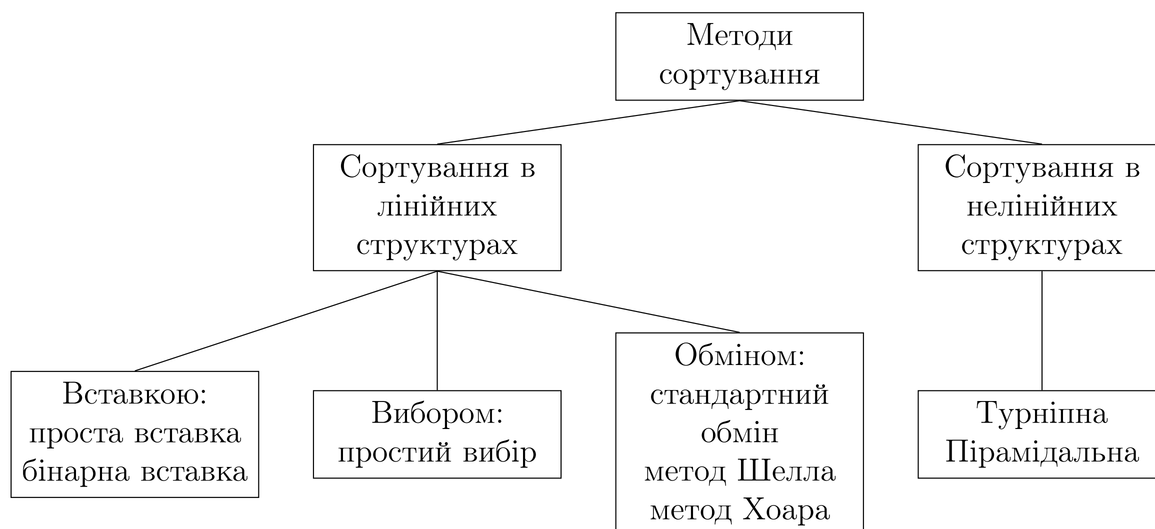


Рис. 5.1: Класифікація методів сортування

5.1.1 Сортування методом прямого включення

Нехай дано послідовність елементів a_0,a_1,a_2, ..., a_{n-1}. Ця послідовність чисто умовно ділиться на дві послідовності: впорядковану послідовність a_0,a_1,a_2, ..., a_{i-1} та невпорядковану послідовність a_i,a_{i+1},a_{i+2}, ..., a_{n-1}. Очевидно, що вихідна впорядкована послідовність складається із одного єдиного елементу a_0. На кожному кроці, починаючи із $i = 1$ і збільшуючи кожен раз i на одиницю, із вихідної послідовності вибирається елемент a_i та вставляється у впорядковану послідовність у відповідне місце. При цьому необхідно зсунути елементи готової послідовності на одну позицію вправо включно до i-тої позиції.

Пошук відповідного місця вставки визначається за допомогою порівняння a_i із елементами a_j $j = j - 1.j - 2.j - 3\ldots$, 9,та одночасним обміном місцями a_i та a_j, тобто зсувом a_j на одну позицію вправо.

Процес сортування завершується при виконанні однієї із умов:

1. $a_i \geq a_j$:
2. досягнуто лівий кінець готової послідовності.

Метод забезпечує стійке сортування.

Приклад програми, що реалізує метод прямого включення.

```
// Сортування методом прямого включення
#include <stdio.h>
#include <windows.h>
```

```
// Розмірність вихідного масиву
  const int n=15;
  int A[n];

/*_______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
    for (int i=0; i<n; i++)
         A[i] = rand()%100;
}
/*_______________________________________________________*/
// Функція сортування прямим включенням
void Sis(int A[],int nn)
{ int i,j,k;
printf("\n Вивід покроркового процесу сортування");
  for (i=1; i<n; i++)
     {
      k=A[i];
       j=i-1;
  while (k<A[j] && j >=0)
      {
         A[j+1]=A[j];
         j--;
       }
 A[j+1]=k;

printf("\n i=%d",i);
   for (j=0; j<nn; j++)
    printf("\t%d",A[j]);
}
}
/*_______________________________________________________*/

int main()
{
  SetConsoleOutputCP(1251);
```

```
  SetConsoleCP(1251);
       GenerationArray();
int j;
printf("\n Вихідний масив: \n\t");
  for (j=0; j<n; j++)
      printf("%d\t",A[j]);
      printf("\n");
      Sis(A,n);
printf("\n Відсортований масив :\n\t");
   for (j=0; j<n; j++)
       printf("%d\t",A[j]);
printf("\n");
system("pause");
return 0;
}
/*_______________________________________________________*/
```

```
Вихідний масив:
       41      67      34      0       69      24      78      58      62      64      5       45      81      27      61

Вивід покрокового процесу сортування
i=1    41      67      34      0       69      24      78      58      62      64      5       45      81      27      61
i=2    34      41      67      0       69      24      78      58      62      64      5       45      81      27      61
i=3    0       34      41      67      69      24      78      58      62      64      5       45      81      27      61
i=4    0       34      41      67      69      24      78      58      62      64      5       45      81      27      61
i=5    0       24      34      41      67      69      78      58      62      64      5       45      81      27      61
i=6    0       24      34      41      67      69      78      58      62      64      5       45      81      27      61
i=7    0       24      34      41      58      67      69      78      62      64      5       45      81      27      61
i=8    0       24      34      41      58      62      67      69      78      64      5       45      81      27      61
i=9    0       24      34      41      58      62      64      67      69      78      5       45      81      27      61
i=10   0       5       24      34      41      58      62      64      67      69      78      45      81      27      61
i=11   0       5       24      34      41      45      58      62      64      67      69      78      81      27      61
i=12   0       5       24      34      41      45      58      62      64      67      69      78      81      27      61
i=13   0       5       24      27      34      41      45      58      62      64      67      69      78      81      61
i=14   0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
 Відсортований масив :
       0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
Press any key to continue . . .
```

Рис. 5.2: Результат виконання програми із виводом ходу процесу сортування

Алгоритм сортування прямим включенням можна покращити, врахувавши той факт, що масив, куди вставляється новий елемент , вже частково впорядкований. Місце вставки елементу шукається методом двійкового (бінарного пошуку. Такий покращений алгоритм сортування називають методом прямого включення із поділом пополам. Використання такого алгоритму дає перевагу тільки у випадку сортування масивів великої розмірності.

Приклад програми, що реалізує метод прямого включення із поділом пополам.

```
// Сортування методим прямого включення
// із поділом пополам

#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
/*______________________________________________________*/
// Функція сортування прямим включенням із поділом пополам

void Binlns(int A[],int nn)
{
int i,j,x,m,L,R;
printf("\n Покроковий процес сортування ");
for (i=1; i<nn; i++)
{
x=A[i];
L=0;
R=i;
printf("\n %d",i);
while (L<R)
{ m=(L+R)/2;
if (A[m] <=x) L=m+1;
else R=m;
}
for (j=i; j>=R; j--)
A[j]=A[j-1];
A[R]=x;
 for(j=0; j<nn;j++)
```

```
    printf("\t %d",A[j]);
}
}


int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
int j;
printf("\n Вихідний масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
BinIns(A,n);
printf("\n Відсортований масив \n");
for (j=0; j<n;j++)
printf("\t%d",A[j]);
printf("\n");
system("pause");
return 0;
}
```

```
Вихідний масив
        41      67      34      0       69      24      78      58      62      64      5       45      81      27      61

Покроковий процес сортування
1       41      67      34      0       69      24      78      58      62      64      5       45      81      27      61
2       34      41      67      0       69      24      78      58      62      64      5       45      81      27      61
3       0       34      41      67      69      24      78      58      62      64      5       45      81      27      61
4       0       34      41      67      69      24      78      58      62      64      5       45      81      27      61
5       0       24      34      41      67      69      78      58      62      64      5       45      81      27      61
6       0       24      34      41      67      69      78      58      62      64      5       45      81      27      61
7       0       24      34      41      58      67      69      78      62      64      5       45      81      27      61
8       0       24      34      41      58      62      67      69      78      64      5       45      81      27      61
9       0       24      34      41      58      62      64      67      69      78      5       45      81      27      61
10      0       5       24      34      41      58      62      64      67      69      78      45      81      27      61
11      0       5       24      34      41      45      58      62      64      67      69      78      81      27      61
12      0       5       24      34      41      45      58      62      64      67      69      78      81      27      61
13      0       5       24      27      34      41      45      58      62      64      67      69      78      81      61
14      0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
 Відсортований масив
        0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
Press any key to continue . . .
```

Рис. 5.3: Результат виконання програми із виводом ходу процесу сортування

5.1.2 Сортування методом прямого вибору

Алгоритм сортування полягає в наступному:

1. у вихідній послідовності із n елементів шукається елемент із найменшим ключем;
2. знайдений елемент міняється місцем із першим елементом послідовності;
3. в послідовності із $n-1$ елементів шукається мінімальний елемент та міняється місцем із другим елементом послідовності і т. д., поки не залишиться останній, найбільший елемент послідовності.

За швидкодією алгоритм прямого вибору має деякі переваги у порівнянні із методом прямого вибору.

Наведемо приклад програми сортування методом прямого вибору та результати її виконання.

```
// Сортування методим прямого вибору


#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
/*______________________________________________________*/
// Функція сортування прямим вибором
void StrSel(int A[],int nn)
{int i,j,x,k;
for (i=0; i<nn-1; i++)
{ x=A[i];
k=i;
printf("\n %d",i);
```

```
for (j=i+1; j<nn;j++)
if (A[j]<x)
{ k=j;
x=A[k];
}
A[k]=A[i];
A[i]=x;
for (j=0; j<nn;j++)
printf("\t %d",A[j]);
}
}

int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
int j;
printf("\n Вихідний масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
StrSel(A,n);
printf("\n Відсортований масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
system("pause");
return 0;
}
```

5.1.3 Сортування методом прямого обміну (метод бульбашки)

Алгоритм прямого обміну грунтується на порівнянні та обміні місцями пари сусідніх елементів послідовності. В результаті одного проходу через послідовність в напрямку від кінця до її початку найменший (найлегший) елемент опиняється на початку послідовності. При зворотньому ході в кінці послідовеності опиняється найбільший елемент послідовності. При повторному

```
Вихідний масив
       41      67      34      0       69      24      78      58      62      64      5       45      81      27      61

0      0       67      34      41      69      24      78      58      62      64      5       45      81      27      61
1      0       5       34      41      69      24      78      58      62      64      67      45      81      27      61
2      0       5       24      41      69      34      78      58      62      64      67      45      81      27      61
3      0       5       24      27      69      34      78      58      62      64      67      45      81      41      61
4      0       5       24      27      34      69      78      58      62      64      67      45      81      41      61
5      0       5       24      27      34      41      78      58      62      64      67      45      81      69      61
6      0       5       24      27      34      41      45      58      62      64      67      78      81      69      61
7      0       5       24      27      34      41      45      58      62      64      67      78      81      69      61
8      0       5       24      27      34      41      45      58      61      64      67      78      81      69      62
9      0       5       24      27      34      41      45      58      61      62      67      78      81      69      64
10     0       5       24      27      34      41      45      58      61      62      64      78      81      69      67
11     0       5       24      27      34      41      45      58      61      62      64      67      81      69      78
12     0       5       24      27      34      41      45      58      61      62      64      67      69      81      78
13     0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
Відсортований масив
       0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
Press any key to continue . . .
```

Рис. 5.4: Результат виконання програми із виводом ходу процесу сортування

проході послідовності наступний найменший елемент зсуваєтьяс до початку послідовності. Процес повторюється до тих пір, поки не будуть впорядковані всі елементи послідовності.

Алгоритм можна покращити, врахувавши той факт, що якщо на даному етапі проходу послідовності жодного обміну елементів не відбулося, то послідовність вже повністю впорядкована.

Наведемо приклад програми сортування методом прямого вибору та результати її виконання.

```
// Сортування методим бульбашки
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*_____________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
/*_____________________________________________________*/
// Функція сортування методом бульбашки
```

```
void BblSort(int A[],int nn)
{
  int i,j,k,p;
for (i=0; i<nn-1; i++)
{ p=0;
/* ознака відсутності обюміну */
for (j=nn-1; j>i; j--)
if (A[j]<A[j-1])
{
  k=A[j];
  A[j]=A[j-1];
  A[j-1]=k;
  p=1; // пройшов процес обміну
}
// Якщо обміну не було, то сортування завершено
if (p==0) return;
printf("\ni=%d",i);
for (j=0; j<nn;j++)
printf("\t %d",A[j]);
}
}

int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
 int j;
printf("\n Вихідний масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
BblSort(A,n);
printf("\n Відсортований масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
system("pause");
return 0;
```

```
}
```

```
 Вихідний масив
       41      67      34      0       69      24      78      58      62      64      5       45      81      27      61

i=0     0       41      67      34      5       69      24      78      58      62      64      27      45      81      61
i=1     0       5       41      67      34      24      69      27      78      58      62      64      45      61      81
i=2     0       5       24      41      67      34      27      69      45      78      58      62      64      61      81
i=3     0       5       24      27      41      67      34      45      69      58      78      61      62      64      81
i=4     0       5       24      27      34      41      67      45      58      69      61      78      62      64      81
i=5     0       5       24      27      34      41      45      67      58      61      69      62      78      64      81
i=6     0       5       24      27      34      41      45      58      67      61      62      69      64      78      81
i=7     0       5       24      27      34      41      45      58      61      67      62      64      69      78      81
i=8     0       5       24      27      34      41      45      58      61      62      67      64      69      78      81
i=9     0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
 Відсортований масив
      0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
Press any key to continue . . .
```

Рис. 5.5: Результат виконання програми із виводом ходу процесу сортування

5.1.4 Шейкерне сортування

Особливістю більбашкового сортування полягає в тому, що сегкі елементи випливають зразу, за один прохід від кінця до початку послідовності, а тяжкі елементи тонуть повільно - на одну позицію за один прохід послідовності. Якщо поміняти напрям перегляду послідовності, то тяжкий елемент опустився-б за один прохід. Ця особливість покладено в основу роботи алгоритму шейкерного сортування, в якому чергуються напрямки послідовних переглядів послідовності. Особливо ефективним метод є в тому випадку, коли елементи послідовності майже впорядковані.

Наведемо приклад програми сортування методом прямого вибору та результати її виконання.

```
// Шейкерне сортування
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*-------------------------------------------------------*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
```

```
  A[i] = rand()%100;
}
/*______________________________________________________*/
// Функція шейкерного сортування
void ShkrSort(int A[],int nn)
{
  int i,j,k,x,L,R;
L=1;
R=nn-1;
k=nn-1;
do
{
for (j=R; j>=L; j--)
if (A[j-1] > A[j])
{
  x=A[j-1];
  A[j-1]=A[j];
  A[j]=x;
  k=j;
 }
L=k+1;
printf("\nL=%d",L);
for (i=0; i<nn; i++)
printf("\t%d",A[i]);
for (j=L; j<=R; j++)
if (A[j-1] > A[j])
{
  x=A[j-1];
  A[j-1]=A[j];
  A[j]=x; k=j;
}
R=k-1;
 printf("\nR=%d",R);
 for (i=0; i<nn; i++)
  printf("\t%d", A[i]);
}
while (L<R);
}
```

```
int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
 int j;
printf("\n Вихідний масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
ShkrSort(A,n);
printf("\n Відсортований масив \n");
for (j=0; j<n; j++)
printf("\t%d",A[j]);
printf("\n");
system("pause");
return 0;
}
```

```
 Вихідний масив
        41      67      34      0       69      24      78      58      62      64      5       45      81      27      61

L=2     0       41      67      34      5       69      24      78      58      62      64      27      45      81      61
R=13    0       41      34      5       67      24      69      58      62      64      27      45      78      61      81
L=3     0       5       41      34      24      67      27      69      58      62      64      45      61      78      81
R=11    0       5       34      24      41      27      67      58      62      64      45      61      69      78      81
L=4     0       5       24      34      27      41      45      67      58      62      64      61      69      78      81
R=10    0       5       24      27      34      41      45      58      62      64      61      67      69      78      81
L=10    0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
R=8     0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
 Відсортований масив
        0       5       24      27      34      41      45      58      61      62      64      67      69      78      81
Press any key to continue . . .
```

Рис. 5.6: Результат виконання програми із виводом ходу процесу сортування

5.2 Швидкі методи сортування

Всі прямі методи сортування фактично переміщають будь-який елемент послідовності на елементарній операції сортування на одну позицію. Тому вони вимагають порядку $O(n^2)$ таких кроків. Це означає, що в основу покращених методів сортування повинні бути покладені алгоритми, що забезпечують принцип переміщення елементів на кожному елементарному кроці на по можливості на більші відстані.

5.2.1 Метод Шелла

Метод Шелла є покращенням методу сортування з допомогою прямого включення та грунтується на сортуванні включеннями із відстаннями, щозменшуються. Спочатку окремо групуються і сортуються методом прямих включень елементи, що знаходяться на деякій відстані h_1 один від одного, потів на відстані $h_2 < h_1$ і т. д. Остання відстань повинна бути рівна одиниці. Тобто, якщо для сортування буде використано t відстаней, то $h_t = 1 < h_{t-1} < \ldots < h_1$. Бажано, щоби відстані забезпечували взаємодію різних ланцюжків якнайчастіше.

Показано, що має зміст використовувати наступні послідовності (тут вони записані у зворотному порядку):

$$1, 4, 13, 40, 121, \ldots h_{k-1} = 3h_{k+1}, t = \log_3(n) - 1$$

$$1, 3, 7, 15, 31, \ldots h_{k-1} = 2h_{k+1}, t = \log_2(n) - 1$$

В останньому випадку для сортування n елементів методом Шелла затрати пропорційні $n^{1.2}$, що значно краще традиційних методів сортування для яких затрати пропорційні n^2.

Часто використовують $h_1 = n/2$, $h_2 = n/4$, …, і т. д.

На кожному етапі сортуються групи $a_1, a_{1+h}, a_{1+2h} \ldots a_{1+kh}$. Оскільки початковий крок є найбільшим, то кількість елементів масиву в початковій групі є найменшим. На другому проході масиву крок зменшується, а кількість елементів у групі зростає, а на останньому проході при $h = 1$ група включає повністю всю послідовність.

Для прикладу розглянемо процес сортування масиву цілих чисел $5, 6, 2, 4, 8, 3, 1, 7$

При $h_1 = 4$ сортуються чотири групи по два елементи в кожній. Тобто:$5, 8; 6, 3; 3, 2$
В результаті перестановок $6 - 3$ та $2 - 1$ отримаємо частково впорядковану послідовність $5, 3, 1, 4, 8, 6, 2, 7$.

При $h_2 = 2$ сортуються дві групи по чотири елементи в кожній. Тобто: $5, 1, 8, 2$ та $3, 4, 6, 7$. В результаті в першій групі після перестановок отримаємо

послідовність $1, 2, 5, 8$. Оскільки друга група впорядкована, то отримаємо для $h_2 = 2$ наступну частково впорядковану послідовність $1, 3, 2, 4, 5, 6, 8, 7$.

Сортування із кроком $h_3 = 1$ приведе до остаточного впорядкування послідовності $1, 2, 3, 4, 5, 6, 7, 8$.

Наведемо приклад програми сортування методом Шелла та результати її виконання.

```
// Шейкерне сортування
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*_______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
/*_______________________________________________________*/
// Функція  сортування методом Шелла

void Shell(int A[],int nn)
{
int i,j,k,x;
k=(nn+1)/2;
while (k >=1)
{
for (i=k; i<nn; i++)
{
  if (A[i-k] > A[i])
     {
    x=A[i];
  j=i-k;
M: A[j+k]=A[j];
  if (j>k)
   {
```

```
  if (A[j-k]>x)
  {
  j=j-k;
   goto M;
    }
     }
   A[j]=x;
     }
    }
printf("\nk=%d ",k);
for (i=0; i<nn; i++)
printf("%d ",A[i]);
if (k>2) k=(k+1)/2;
          else k=k/2;
}
}

int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
 int j;


printf("\n Вихідний масив: \n");
     printf("\t");
     for (j=0; j<n; j++)
       printf("%d  ",A[j]);
printf("\n Вивід ходу сортування");
     Shell(A,n);
printf("\n Відсортований масив: \n");
for (j=0;j<n;j++)
printf("%d  ",A[j]);
printf("\n");
system("pause");
return 0;

}
```

```
 Вихідний масив:
        41  67  34  0  69  24  78  58  62  64  5  45  81  27  61
 Вивід ходу сортування
k=8 41 64 5 0 69 24 61 58 62 67 34 45 81 27 78
k=4 41 24 5 0 62 27 34 45 69 64 61 58 81 67 78
k=2 5 0 34 24 41 27 61 45 62 58 69 64 78 67 81
k=1 0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
 Відсортований масив:
0  5  24  27  34  41  45  58  61  62  64  67  69  78  81
Press any key to continue . . .
Press any key to continue . . .
```

Рис. 5.7: Результат виконання програми із виводом ходу процесу сортування

5.2.2 Сортування за допомогою дерева (піраміди)

Сортування за допомогою дерева є удосконаленням методу прямого вибору, що грунтується на пошуку найменшого елементу спочатку серед n елементів, потім серед $n-1$ елементів і т. д. Недоліком такого підходу є той факт, що за один прохід послідовності знаходиться тільки один елемент серед всіх n елементів послідовності не запамятовуючи ніякої іншої інформації.

В 1964 році Вильямсом був запропонований алгоритмсортування за допомогою дерева. Дерево має спеціальну структуру, яка називається пірамідою. Вона складається із поміченого двійкового дерева висотою h, яке володіє наступними властивостями:

1. кожна кінцева вершина має висоту h або $h-1$;
2. кожна кінцева вершина висоти h находиться лівіше кінцевої вершини висоти $h-1$;
3. ключ будь-якої вершини більший ключа наступної за нею вершини (нащадка). Це означає, що ключ кореня є найбільшим клбчом дерева.

Виходячи із перших двох властивостей будь-який масив, що складається із n елементів можна представити у вигляді двійкового дерева, яке потім можна перетворити в піраміду. Нехай маємо елементи масиву, що приймають наступні значення $5, 3, 7, 27, 9, 11, 14, 2, 8$. Із елементів масиву побудуємо дерево наступним чином. Перший елемент (5) утворює корінь дерева (нульовий рівень). Наступні два елементи $(7, 7)$ утворюють перший рівень. Наступні чотири елементи формують другий рівень і т. д. На кожному рівні елементи розміщуються зліва направо. В результаті отримаємо дерево:

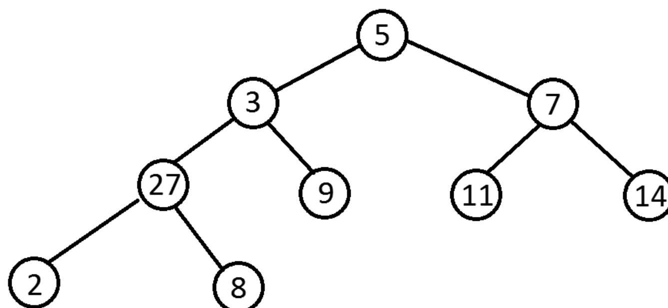


Рис. 5.8: Вихідне дерево для побудови піраміди

Із рисунка видно, що для будь-якої вершини дерева a_i нащадками є вершини a_j та a_{j+1}, $j = 2 \cdot i$, і остання нелистова вершина утворюється елементами a_k, $k = nmod2$.

Із отриманого дерева піраміда будується із врахуванням того, що ключ будь-якої вершини піраміди повинен бути більшим ключів нащадків (третя властивість двійкового дерева). Починаючи із самої останньої нелистової вершини $i = nmod2$ перевіряємо виконання даної властивості. Якщо вона не виконується, то ключі вершини та нащадка необхідно поміняти місцями. Таку процедуру необхідно проробити із вершиною $i = i - 1$ і т. д., поки не буде досягнуто корінь дерева.

Для розглядуваного прикладу $i = 9mod2 = 4$, $a_4 > a_8$ та $a_4 > a_9$ ($27 > 2$ та $27 > 8$), тому перестановок виконувати нема потреби. Для $i = 3$ a_3 (7) менше a_6 та a_7 (11 та 14), а $a_7 > a_7$, тому міняємо місцями a_3 та a_7. На наступному етапі при $i = 2$ міняємо місцями a_2 та a_4, потім a_4 та a_9. При $i = 1$ послідовно міняємо місцями a_1 та a_2,a_2 та a_5.

В результаті отримаємо піраміду

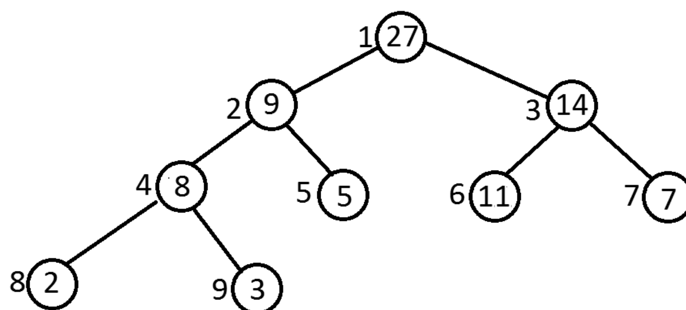


Рис. 5.9: Піраміда

Таким чином на першому етапі вихідний масив перетворено до структури піраміди, де a_1 - найбільший елемент.

На наступному етапі, використовуючи цю структуру, виконаємо впорядкування елементів масиву. Для цього поміняємо місцями перший та останній елементи: $a_1 \Longleftrightarrow a_9$ ($27 \Longleftrightarrow 3$). В результаті отримаємо нове дерево, яке тепер вже не є пірамідою.

Виключивши із розгляду останній елемент дерева $a_9 = 27$, отримаємо нове дерево, що складається із $n - 1$ елементів. В ньому тільки коренева вершина порушує властивість 3 піраміди. Послідовно переставляючи її: $a_1 \Longleftrightarrow a_3 \Longleftrightarrow a_6$, отримаємо нову піраміду із вершиною $a_1 = 14$.

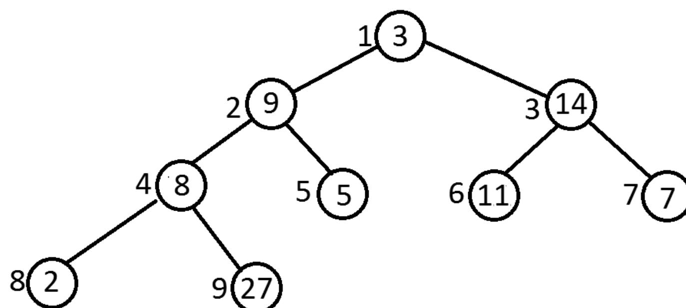


Рис. 5.10: Перестановка найбільшого елементу в низ піраміди

Помінявши місцями $a_1 = 14$ та $a_{n-1} = 2$ і виключивши із розгляду $an - 1$, отримаємо нове дерево, що складається із $n - 2$ елементів

Операція повторюється до тих пір, поки не отримається впорядкований за зростанням масив. Для впорядкування масиву за спаданням необхідно використовувати піраміду, для якої ключ будь-якої вершини є менший ключа наступної за нею вершини. При цьому ключ кореня є найменшим ключем дерева.

Приклад програми сортування за зростанням з використанням піраміди.

```
// Сортування пірамідою за зростанням
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
// ФУНКЦІЯ ПОБУДОВИ ПІРАМІДИ
void Sift(int A[],int L, int R)
{
```

```
  int i,j,x,k;
  i=L;
  j=2*L+1;
  x=A[L];
if ((j<R) && (A[j]<A[j+1]))
j++;
while ((j<=R) && (x<A[j]))
{
k=A[i];
A[i]=A[j];
A[j]=k;
i=j;
j=2*j+1;
if ((j<R) && (A[j]<A[j+1]))
j++;
}
}
/*______________________________________________________*/
// Функція  сортування пірамідою

void HeapSort(int A[],int nn)
{
int L,R,x,i;
L=nn/2;
R=nn-1;
/* Побудова піраміди із вихідного масмву */
while (L>0)
{
  L=L-1;
Sift(A,L,R);
printf("\nL=%d: ",L);
for (i=0;i<nn;i++)
printf("%d ",A[i]);
}
// Сортування: піпаміда=> відсортований масив
while (R>0)
{ x=A[0];
A[0]=A[R];
A[R]=x;
```

```
R--;
printf("\nR=%d: ",R);
for (i=0;i<nn;i++)
printf("%d ",A[i]);
Sift(A,L,R);
}
}


int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
int j;
printf("\n Сортування пірамідою");
printf("\n Вихідний масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n Хід процесу сортування");
HeapSort(A,n);
printf("\n Відсортований масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n");
system("pause");
return 0;
}
```

Приклад програми сортування за спаданням з використанням піраміди.

```
// Сортування пірамідою за спаданням
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*______________________________________________________*/
```

```
 Сортування пірамідою
 Вихідний масив
        41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
 Хід процесу сортування
L=6: 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
L=5: 41 67 34 0 69 81 78 58 62 64 5 45 24 27 61
L=4: 41 67 34 0 69 81 78 58 62 64 5 45 24 27 61
L=3: 41 67 34 62 69 81 78 58 0 64 5 45 24 27 61
L=2: 41 67 81 62 69 45 78 58 0 64 5 34 24 27 61
L=1: 41 69 81 62 67 45 78 58 0 64 5 34 24 27 61
L=0: 81 69 78 62 67 45 61 58 0 64 5 34 24 27 41
R=13: 41 69 78 62 67 45 61 58 0 64 5 34 24 27 81
R=12: 27 69 61 62 67 45 41 58 0 64 5 34 24 78 81
R=11: 24 67 61 62 64 45 41 58 0 27 5 34 69 78 81
R=10: 34 64 61 62 27 45 41 58 0 24 5 67 69 78 81
R=9: 5 62 61 58 27 45 41 34 0 24 64 67 69 78 81
R=8: 24 58 61 34 27 45 41 5 0 62 64 67 69 78 81
R=7: 0 58 45 34 27 24 41 5 61 62 64 67 69 78 81
R=6: 0 34 45 5 27 24 41 58 61 62 64 67 69 78 81
R=5: 0 34 41 5 27 24 45 58 61 62 64 67 69 78 81
R=4: 0 34 24 5 27 41 45 58 61 62 64 67 69 78 81
R=3: 0 27 24 5 34 41 45 58 61 62 64 67 69 78 81
R=2: 0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
R=1: 0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
R=0: 0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
 Відсортований масив
        0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
Press any key to continue . . .
```

Рис. 5.11: Результат виконання програми із виводом ходу процесу сортування

```
// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
// ФУНКЦІЯ ПОБУДОВИ ПІРАМІДИ


void Sift(int A[],int L, int R)
{
int i,j,x,k;
i=L;
```

```
j=2*L+1;
x=A[L];
if ((j<R) && (A[j]>A[j+1]))
j++;
while ((j<=R) && (x>A[j]))
{k=A[i];
A[i]=A[j];
A[j]=k;
i=j;
j=2*j+1;
if ((j<R) && (A[j]>A[j+1]))
j++;
}
}

/*_____________________________________________________*/
// Функція  сортування пірамідою

void HeapSort(int A[],int nn)
{
  int L,R,x,i;
L=nn/2;
R=nn-1;
// Побудова піраміди із вихідного масиву
while (L>0)
{
L=L-1;
Sift(A,L,R);
printf("\nL=%d: ",L);
for (i=0;i<nn;i++)
printf("%d ",A[i]);
}
// Сортування: піраміда=> відсортований масив
while (R>0)
{ x=A[0];
A[0]=A[R];
A[R]=x;
R--;
printf("\nR=%d: ",R);
```

```
for (i=0;i<nn;i++)
printf("%d ",A[i]);
Sift(A,L,R);
}
}


int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
int j;
printf("\n Сортування пірамідою");
printf("\n Вихідний масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n Хід процесу сортування");
HeapSort(A,n);
printf("\n Відсортований масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n");
system("pause");
return 0;
}
```

```
 Сортування пірамідою
 Вихідний масив
        41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
 Хід процесу сортування
L=6: 41 67 34 0 69 24 27 58 62 64 5 45 81 78 61
L=5: 41 67 34 0 69 24 27 58 62 64 5 45 81 78 61
L=4: 41 67 34 0 5 24 27 58 62 64 69 45 81 78 61
L=3: 41 67 34 0 5 24 27 58 62 64 69 45 81 78 61
L=2: 41 67 24 0 5 34 27 58 62 64 69 45 81 78 61
L=1: 41 0 24 58 5 34 27 67 62 64 69 45 81 78 61
L=0: 0 5 24 58 41 34 27 67 62 64 69 45 81 78 61
R=13: 61 5 24 58 41 34 27 67 62 64 69 45 81 78 0
R=12: 78 41 24 58 61 34 27 67 62 64 69 45 81 5 0
R=11: 81 41 27 58 61 34 78 67 62 64 69 45 24 5 0
R=10: 81 41 34 58 61 45 78 67 62 64 69 27 24 5 0
R=9: 69 41 45 58 61 81 78 67 62 64 34 27 24 5 0
R=8: 64 58 45 62 61 81 78 67 69 41 34 27 24 5 0
R=7: 69 58 64 62 61 81 78 67 45 41 34 27 24 5 0
R=6: 67 61 64 62 69 81 78 58 45 41 34 27 24 5 0
R=5: 78 62 64 67 69 81 61 58 45 41 34 27 24 5 0
R=4: 81 67 64 78 69 62 61 58 45 41 34 27 24 5 0
R=3: 69 67 81 78 64 62 61 58 45 41 34 27 24 5 0
R=2: 78 69 81 67 64 62 61 58 45 41 34 27 24 5 0
R=1: 81 78 69 67 64 62 61 58 45 41 34 27 24 5 0
R=0: 81 78 69 67 64 62 61 58 45 41 34 27 24 5 0
 Відсортований масив
        81 78 69 67 64 62 61 58 45 41 34 27 24 5 0
Press any key to continue . . .
```

Рис. 5.12: Результат виконання програми із виводом ходу процесу сортування

5.2.3 Швидке сортування Хоара (сортування поділом)

Даний метод є удосконаленням методу сортування обміном. Не дивлячись на те , що метод сортування бульбашкою є найменш ефективним, метод сортування Хоара є найбільш ефективним із відомих на даний момент методів. Його суть полягає в наступному:

Вибирається навмання будь-який елемент x вихідного масиву. Зліва від вибраного елементу будемо аналізувати масив до тих пір, поки не буде знайдено елемент $a_i > x$. Справа від вибраного елементу будемо аналізувати масив, поки не знайдемо елемент $a_j \leq x$. Поміняємо місцями ці два знайдені елементи та продовжимо процес пошуку до тих пір, поки обидва перегляди не зустрінуться. В результаті вихідний масив розіб'ється на дві частини. Приава частина міститиме елементи, менші чи рівні x, а права частина - елементи, що більші x. Застосувавши аналогічну процедуру до лівої і правої частин масиву від точки зустрічі, отримаємо чотири частини і т. д., поки кожна частина

буде містити тільки по одному елементу.

При рівноімовірному розподілі елементів масиву кожний раз вибирають медіану в якості елемента поділу. За звичай в якості границі вибирають середній елемент, що забезпечує не погані показники ефективності методу.

Приклад швидкого сортування Хоара із використанням рекурсивної функції Quicksort.

```
// Швидке сортування Хоара
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
const int n=15;
int A[n];
/*______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
// Рекурсивна функція швидкого сортування
void QuickSort(int A[],int L,int R)
{
  int i,j,k,x,m;
i=L;
j=R;
x=A[(L+R)/2];
do
{
while (A[i]<x)
  i++;
while (x<A[j])
j--;
if (i <=j)
{
  k=A[i];
  A[i]=A[j];
  A[j]=k;
```

```
  i++;
  j--;
 printf("\n i=%d j=%d x=%d: ",i-1,j+1,x);
   for (m=0; m<n; m++)
     printf(" %d ",A[m]);
}
}
while (i<j);
if (L<j)
{
  printf("\t L=%d j=%d",L,j);
    QuickSort(A,L,j);
}
if (i<R)
{ printf("\t i=%d R=%d",i,R);
QuickSort(A,i,R);
}
}

int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
int j;
printf("\n Сортування Хоара");


printf("\n Вихідний масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n Хід процесу сортування");
QuickSort(A,0,n-1 );
printf("\n Відсортований масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n");
system("pause");
return 0;
```

```
}
```

```
Сортування Хоара
Вихідний масив
       41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
Хід процесу сортування
i=1 j=13 x=58:  41  27  34  0  69  24  78  58  62  64  5  45  81  67  61
i=4 j=11 x=58:  41  27  34  0  45  24  78  58  62  64  5  69  81  67  61
i=6 j=10 x=58:  41  27  34  0  45  24  5  58  62  64  78  69  81  67  61
i=7 j=7 x=58:  41  27  34  0  45  24  5  58  62  64  78  69  81  67  61        L=0 j=6
i=0 j=3 x=0:  0  27  34  41  45  24  5  58  62  64  78  69  81  67  61         i=1 R=6
i=3 j=6 x=41:  0  27  34  5  45  24  41  58  62  64  78  69  81  67  61
i=4 j=5 x=41:  0  27  34  5  24  45  41  58  62  64  78  69  81  67  61        L=1 j=4
i=2 j=4 x=34:  0  27  24  5  34  45  41  58  62  64  78  69  81  67  61        L=1 j=3
i=1 j=3 x=24:  0  5  24  27  34  45  41  58  62  64  78  69  81  67  61        L=1 j=2
i=1 j=1 x=5:  0  5  24  27  34  45  41  58  62  64  78  69  81  67  61         i=2 R=3
i=2 j=2 x=24:  0  5  24  27  34  45  41  58  62  64  78  69  81  67  61        i=3 R=4
i=3 j=3 x=27:  0  5  24  27  34  45  41  58  62  64  78  69  81  67  61        i=5 R=6
i=5 j=6 x=45:  0  5  24  27  34  41  45  58  62  64  78  69  81  67  61        i=8 R=14
i=10 j=14 x=69:  0  5  24  27  34  41  45  58  62  64  61  69  81  67  78
i=11 j=13 x=69:  0  5  24  27  34  41  45  58  62  64  61  67  81  69  78      L=8 j=12
i=8 j=10 x=61:  0  5  24  27  34  41  45  58  61  64  62  67  81  69  78       L=8 j=9
i=8 j=8 x=61:  0  5  24  27  34  41  45  58  61  64  62  67  81  69  78        i=9 R=12
i=9 j=10 x=62:  0  5  24  27  34  41  45  58  61  62  64  67  81  69  78       i=10 R=12
i=11 j=11 x=67:  0  5  24  27  34  41  45  58  61  62  64  67  81  69  78      i=12 R=14
i=12 j=13 x=69:  0  5  24  27  34  41  45  58  61  62  64  67  69  81  78      i=13 R=14
i=13 j=14 x=81:  0  5  24  27  34  41  45  58  61  62  64  67  69  78  81
Відсортований масив
       0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
Press any key to continue . . .
```

Рис. 5.13: Результат виконання програми із виводом ходу процесу сортування

Алгоритм швидкого сортування можна реалізувати і без використання рекурсивної функції. При цьому рекурсія заміняється ітерацією, що в свою чергу вимагає використання додаткових операцій для збереження потрібної інформації з використанням стеку. На кожному етапі поділу виникають дві задачі по розділенню. Одну із них можна виконати зразу, а інформацію про другу необхідно запамятати в стеку, які необхідно виконати пізніше. Кожна вимога на поділ просто задається лівою і правою границями - індексами елементів масиву та заноситься в стек, організований у вигляді масиву структур.Вибір зі стеку здійснюється у зворотньому порядку Максимальний розмір стеку рівний n (розмір вихідного масиву). Це значення є максимальним та практично ніколи не досягається.

Приклад швидкого сортування Хоара без використанням рекурсивної функції .

```
// Швидку сортування Хоара
#include <stdio.h>
#include <windows.h>
// Розмірність вихідного масиву
```

```
const int n=15;
int A[n];
/*______________________________________________________*/

// Функція ініціалізації вихідного масиву
// з використанням генератора випадкових чисел
void GenerationArray()
{
  for (int i=0; i<n; i++)
  A[i] = rand()%100;
}
// Нерекурсивна функція швидкого сортування

void NRQuickSort(int A[],int nn)
{
  int i,j,k,x,m,s,L,R;
#define d 16
  struct stack {int L; int R;} st[d];
s=1;
st[1].L=0;
st[1].R=nn-1;
do
{
L=st[s].L;
R=st[s].R;
s--;
 do
{
  i=L;
  j=R;
  x=A[(L+R)/2];
  do
{
while (A[i]<x)
i++;
while (x<A[j])
j--;
if (i <=j)
{
```

```
  k=A[i];
  A[i]=A[j];
  A[j]=k;
i++;
j--;
printf("\n i=%d j=%d x=%d: ",i-1,j+1,x);
for (m=0; m<n; m++)
printf(" %d",A[m]);
}
}
while (i<j);
if (i<R)
{
  s++;
  st[s].L=i;
  st[s].R=R;
  printf(" i=%d R=%d s=%d",i,R,s);
}
R=j;
}
while (L<R);
}
while (s!=0);
}


int main()
{
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
GenerationArray();
int j;
printf("\n Сортування Хоара");


printf("\n Вихідний масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n Хід процесу сортування");
```

```
NRQuickSort(A,n);
printf("\n Відсортований масив \n\t");
for (j=0; j<n; j++)
printf("%d ",A[j]);
printf("\n");
system("pause");
return 0;
}
```

```
 Сортування Хоара
 Вихідний масив
        41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
 Хід процесу сортування
 i=1 j=13 x=58:  41 27 34 0 69 24 78 58 62 64 5 45 81 67 61
 i=4 j=11 x=58:  41 27 34 0 45 24 78 58 62 64 5 69 81 67 61
 i=6 j=10 x=58:  41 27 34 0 45 24 5 58 62 64 78 69 81 67 61
 i=7 j=7 x=58:  41 27 34 0 45 24 5 58 62 64 78 69 81 67 61 i=8 R=14 s=1
 i=0 j=3 x=0:  0 27 34 41 45 24 5 58 62 64 78 69 81 67 61 i=1 R=6 s=2
 i=3 j=6 x=41:  0 27 34 5 45 24 41 58 62 64 78 69 81 67 61
 i=4 j=5 x=41:  0 27 34 5 24 45 41 58 62 64 78 69 81 67 61 i=5 R=6 s=2
 i=2 j=4 x=34:  0 27 24 5 34 45 41 58 62 64 78 69 81 67 61 i=3 R=4 s=3
 i=1 j=3 x=24:  0 5 24 27 34 45 41 58 62 64 78 69 81 67 61 i=2 R=3 s=4
 i=1 j=1 x=5:  0 5 24 27 34 45 41 58 62 64 78 69 81 67 61
 i=2 j=2 x=24:  0 5 24 27 34 45 41 58 62 64 78 69 81 67 61
 i=3 j=3 x=27:  0 5 24 27 34 45 41 58 62 64 78 69 81 67 61
 i=5 j=6 x=45:  0 5 24 27 34 41 45 58 62 64 78 69 81 67 61
 i=10 j=14 x=69:  0 5 24 27 34 41 45 58 62 64 61 69 81 67 78
 i=11 j=13 x=69:  0 5 24 27 34 41 45 58 62 64 61 67 81 69 78 i=12 R=14 s=1
 i=8 j=10 x=61:  0 5 24 27 34 41 45 58 61 64 62 67 81 69 78 i=9 R=12 s=2
 i=8 j=8 x=61:  0 5 24 27 34 41 45 58 61 64 62 67 81 69 78
 i=9 j=10 x=62:  0 5 24 27 34 41 45 58 61 62 64 67 81 69 78 i=10 R=12 s=2
 i=11 j=11 x=67:  0 5 24 27 34 41 45 58 61 62 64 67 81 69 78
 i=12 j=13 x=69:  0 5 24 27 34 41 45 58 61 62 64 67 69 81 78 i=13 R=14 s=1
 i=13 j=14 x=81:  0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
 Відсортований масив
        0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
Press any key to continue . . .
```

Рис. 5.14: Результат виконання програми із виводом ходу процесу сортування

5.3 Алгоритми пошуку

При роботі з інформацією дуже важливим є пошук. Пошук - процес знаходження у множині даних якоїсь інформації. Зазвичай у даних містяться ключі. Ключ пошуку – це поле запису, використовуючи значення якого, відбувається пошук. Їх використовують щоб розрізняти записи. Загальна мета пошуку – знайти усі збіги записів з певним значенням ключа.

Пошук є однією з найчастіше використовуваних операцій у програмуванні. Взагалі, різноманітних видів пошуку дуже багато і вони розрізняються між собою способом організації даних. Але у кожному алгоритмі пошуку є свої переваги і недоліки.

Загальний алгоритм пошуку можна описати наступним чином:

1. Обчислення елемента, що часто передбачає отримання значення елемента, ключа елемента і т.д.
2. Порівняння елемента з еталоном або порівняння двох елементів.
3. Перебір елементів множини, тобто проходження по елементах масиву.

Різняться види пошуку методом перебору і стратегіями пошуку. В лінійних структурах існують такі основні алгоритми.

5.3.1 Послідовний або лінійний пошук

Це самий простий вид пошуку деякого елемента серед інших, що здійснюється за допомогою перевірки кожного елемента до тих пір, поки вони не будуть збігатися. Загальна ідея цього виду пошуку така: усі елементи розглядаються послідовно, один за одним. Це дає змогу не пропустити жодного елемента. Якщо збіг буде знайдено, то пошук припиняється і його результат є позитивним. Якщо не знайдено, то результат буде негативним. Перевагами такого пошуку є простота його реалізації, він не потребує додаткового об'єму пам'яті або додаткової роботи з функціями. Це дозволяє працювати вже під час отримання даних.

Також існує певний покращений послідовний алгоритм, який пришвидшує пошук. У множині встановлюється бар'єр, тобто елемент, який задовольняє пошуку. У циклі відпадає необхідність перевірки умови, зв'язаної з границями множини. Таким чином буде обмежена зміна індексу. Як і у попередньому випадку, алементи проглядаються по черзі, але для зменшення кількості порівнянь після останнього елемента додається елемент з ключем, що дорівнює шуканому значенню.

5.3.2 Бінарний або двійковий пошук

Це такий вид пошуку, у якому пошук елемента з множини відбувається за допомогою ділення деякої кількості раз цієї множини навпіл. Елемент, який треба знайти, колись потрапить або не потрапить в одну з цих двох частин. Бінарний пошук застосовується для відсортованих множин. Ідея цього пошуку така: Пошук даного значення серед деякого масиву починається, коли визначається центральний елемент цього масиву. Значення цього елемента порівнюється зі значенням елемента, який треба знайти. Якщо потрібне нам значення збігається з центральним, то пошук завершено. Якщо воно або більше або менше, то створюється масив, який складається з елементів, що знаходяться ліворуч або праворуч від центрального значення і тепер пошук відбувається в цьому масиві.

Перевагами такого пошуку є швидкість пошуку, якщо порівнювати з послідовним пошуком. Але є й такий недолік, що двійковий пошук може використовуватись лише на упорядкованій множині.

5.3.3 Інтерполяційний пошук

Для цього виду пошуку теж є обмеження: масив повинен бути впорядкований за величиною ключів кожного елемента. У цьому алгоритмі пошуку величини повинні бути рівномірно розподілені у деякому їх інтервалі від х до у. З цього виходить, що якщо є відома величина C ключа пошуку, то положення потрібного нам запису можливо передбачити, а не шукати у всьому файлі. Цей пошук використовується частіше, ніж бінарний.

Алгоритми пошуку рядка - важливий клас рядкових алгоритмів, що намагаються знайти місце де один або декілька текстових рядків входять у довший рядок або текст.

5.3.4 Алгоритм послідовного (прямого) пошуку

Найбільш очевидний та простий в реалізації алгоритм. Нехай m і n - довжини слів S і X відповідно. Можна порівняти зі словом X всі подслова S, які починаються з позицій $1, 2, ..., m - n + 1$ в слові S; у разі рівності виводиться відповідна позиція. Це не ефективний алгоритм, бо в ньому присутні два цикли (один вкладений), час роботи зовнішнього більшим ступенем залежить від n, а внутрішній у гіршому випадку робить m операцій. Таким чином, час роботи всього алгоритму є $O((n - m + 1)m)$. Для маленьких рядків пошук пропрацює швидко.

Алгоритм Рабіна являє собою модифікацію лінійного алгоритму. У слові A, довжина якого дорівнює m, шукається зразок X довжини n. Виріжемо "вікно"розміром n - воно буде рухатися по вхідному слову. Чи збігається слово у "вікні"із заданим зразком? Фіксується деяка числова функція на словах довжини n, тоді завдання зводиться до порівняння чисел, що є значно швидшою процедурою. Якщо значення цієї функції на слові у "вікні"та зразку різні, то збігу немає. Тільки якщо значення однакові, необхідно перевіряти послідовно збіг по буквах.

Цей алгоритм виконує лінійний прохід по рядку (n кроків) і лінійний прохід по всьому тексту (m кроків). Час роботи алгоритму лінійно залежить від розміру рядка і тексту. Замість того, щоб перевіряти кожну позицію на предмет відповідності зі зразком алгоритм передбачає перевірки тільки ті, які «нагадують» зразок.

Прикладом зручної для обчислення функції може бути заміна всіх букв у слові і зразку їх номерами, що представляють собою цілі числа. Тоді зручною функцією є сума цифр. При зсуві "вікна"потрібно додати нове число і відняти "зникле".

Алгоритм Рабіна і алгоритм послідовного пошуку є найпростішими алгоритмами, однак вони не є найбільш оптимальними

5.3.5 Алгоритм Кнута, Моріса, Пратта

Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм, який фактично потребує лише N порівнянь навіть в самому поганому випадку. Новий алгоритм базується на тому, що після часткового збігу початкової частини слова з відповідними символами тексту фактично відома пройдена частина тексту і можна „обчислити" деякі відомості (на основі самого слова), за допомогою яких потім можна швидко пересунутися текстом.

Приведений приклад пошуку слова $ABCABD$ показує принцип роботи такого алгоритму. Символи, які пройшли порівняння, – підкреслені.

Зверніть увагу: при кожному незбігу пари символів слово зсовується на всю пройдену відстань, оскільки менші зсуви не можуть привести до повного збігу.

5.3.6 Алгоритм Боуєра та Мура

КМП-пошук дає справжній виграш тільки тоді, коли невдачі передувала деяка кількість збігів. Лише у цьому випадку слово зсовується більше ніж на одиницю. На жаль, це швидше виняток, ніж правило: збіги зустрічаються значно рідше, ніж незбіги. Тому виграш від практичного використання КМП-стратегії в більшості випадків пошуку в звичайних текстах досить незначний. Метод, який запропонували Р. Боуєр і Д. Мур в 1975 р., не тільки покращує обробку самого поганого випадку, але й дає виграш в проміжних ситуаціях. БМ-пошук базується на незвичних міркуваннях – порівняння символів починається з кінця слова, а не з початку. Як і у випадку КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Нехай для кожного символу x із алфавіту величина dx – відстань від самого правого в слові входження x до правого кінця слова. Уявимо, що виявлена розбіжність між словом і текстом. У цьому випадку слово відразу ж можна зсунути праворуч на $dpM - 1$ позицій, тобто на кількість позицій, швидше за все більше одиниці.

Якщо символ, який не збігся, тексту в слові взагалі не зустрічається, то зсув стає навіть більшим, а саме зсовувати можна на довжину всього слова.

5.3.7 Чисельні алгоритми

Алгоритми рішення нелінійних рівнянь

Алгоритми рішення систем лінійних рівнянь

Алгоритми інтерполяції

Алгоритми апроксімації

Алгоритми рішення

екстремальних задач

. Алгоритми рішення диференційних рівнянь

Література

[1] *Львовский С.М.* Набор и верстка в пакете LaTeX. М. : Космосинформ, 1995. -373с.

[2] *Плис А. И., Сливина Н. А.* MathCAD 2000:Математический практикум. М.: Финансы и статистика, 2002. 655 с.

[3] *Половко Ф.М., Ганичев И.В.* Mathcad для студента СП.: БХВ-Петербург, 2006.- 226с.

[4] *Спивак М.* Восхитительный ТеХ.М.: Мир, 1993. -285с

Зміст

I Структура даних **3**

1 Структура даних **4**

1.1 Поняття структури даних 4
1.1.1 Концепція типу даних 6
1.1.2 Класифікація структур даних 8
1.2 Прості структури даних 10
1.2.1 Арифметичні типи 10
1.2.2 Перерахований тип 10
1.2.3 Вказівники 14
1.3 Статичні структури даних 19
1.3.1 Стандартна бібліотека шаблонів (STL) 21
1.3.2 Вектор 30
1.3.3 Масив 44
1.3.4 Розріджені масиви 57
1.3.5 Множини 67
1.3.6 Структури 88
1.3.7 Об'єднання 95
1.3.8 Бітові типи 98
1.3.9 Таблиці 103
1.4 Динамічні структури даних 105
1.4.1 Список 105
1.4.2 Різновиди списків 118
1.4.3 Контейнер *list* 127
1.4.4 Стек 138
1.4.5 Черга 140
1.4.6 Дек 142

2 Нелінійні структури даних **143**

2.1 Дерева 143
2.1.1 Основні поняття та визначення 143

2.1.2 Бінарне дерево . . . 145
2.1.3 Бінарне дерево пошуку . . . 154
2.1.4 Збалансоване дерево . . . 163
2.1.5 Червоно-чорне дерево . . . 165
2.2 Графи . . . 167
2.2.1 Основні поняття та визначення . . . 167
2.2.2 Способи представлення графів . . . 169
2.2.3 Мінімальне покриваюче (каркасне) дерево . . . 172
2.2.4 Пошук в ширину . . . 174
2.2.5 Пошук в глибину . . . 178
2.2.6 Ейлерів цикл . . . 184
2.2.7 Гамільтонів цикл . . . 191
2.2.8 Найкоротший шлях . . . 194

II Структура даних 203

3 Поняття алгоритму і його властивості 204
3.1 Поняття алгоритму . . . 204
3.1.1 Поняття алгоритму . . . 204
3.1.2 Властивості алгоритму . . . 206
3.1.3 Способи представлення алгоритмів . . . 207
3.1.4 Базові структури алгоритмів . . . 214
3.1.5 Аналіз складності алгоритму . . . 220

4 Рекурсивні алгоритми 227
4.1 Поняття рекурсії . . . 227
4.2 Рекурсивний алгоритм сортування . . . 231

5 Алгоритми обробки структур даних 234
5.1 Методи сортування . . . 234
5.1.1 Сортування методом прямого включення . . . 235
5.1.2 Сортування методом прямого вибору . . . 240
5.1.3 Сортування методом прямого обміну (метод бульбашки) 241
5.1.4 Шейкерне сортування . . . 244
5.2 Швидкі методи сортування . . . 247
5.2.1 Метод Шелла . . . 247
5.2.2 Сортування за допомогою дерева (піраміди) . . . 251
5.2.3 Швидке сортування Хоара (сортування поділом) . . . 259

5.3 Алгоритми пошуку 266
5.3.1 Послідовний або лінійний пошук 267
5.3.2 Бінарний або двійковий пошук 268
5.3.3 Інтерполяційний пошук 269
5.3.4 Алгоритм послідовного (прямого) пошуку 270
5.3.5 Алгоритм Кнута, Моріса, Пратта 271
5.3.6 Алгоритм Боуєра та Мура 274
5.3.7 Чисельні алгоритми 276