ФАКУЛЬТЕТ КОМП’ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КИЇВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Іванов Є.О., Ліндер Я.М., Жереб К.А., Супрун О.О.

Представлення та обробка динамічних структур даних

Посібник першокурсника

Другий семестр

Київ – 2025

УДК 004.3/.4(075.8)

Рекомендовано до друку вченою радою факультету комп'ютерних наук та кібернетики (протокол № 14 від 20 травня 2025 року)

Ухвалено науково-методичною комісією факультету комп'ютерних наук та кібернетики (протокол № 9 від 19 травня 2025 року)

Рецензенти:

О.І. Провотар, д-р ф.-м. н. (професор, завідувач кафедри інтелектуальних програмних систем Київського національного університету імені Тараса Шевченка);

О.П. Нечипорук, д-р т. н. (професорка, завідувачка кафедри інтелектуальних кібернетичних систем Державного університету "Київський авіаційний інститут").

Іванов Є.О., Ліндер Я.М., Жереб К.А., Супрун О.О.

Представлення та обробка динамічних структур даних: навчальний посібник. – Електронне видання, – 2025. – 135 с.

У посібнику подано матеріали для вивчення курсу «Програмування» студентами першого курсу факультету комп'ютерних наук та кібернетики спеціальності «Інженерія програмного забезпечення». Навчальний посібник містить загальні відомості про представлення та обробку динамічних структур даних, зокрема списків, дерев та графів, з використанням мови програмування C++. Також розглянуто різні методи сортування та проведено їх порівняння. Крім того, у посібнику наведені приклади розв'язання типових задач та варіанти лабораторних робіт, рекомендованих до виконання студентами протягом другого семестру.

Зміст

Вступ

Програмування є однією з ключових дисциплін у підготовці сучасного фахівця в галузі інформаційних технологій. Оволодіння мовою програмування C / C++ є важливим етапом у навчанні, оскільки ця мова поєднує в собі ефективність, гнучкість та низькорівневий доступ до пам’яті, що дозволяє краще зрозуміти принципи роботи комп’ютерних систем.

Цей посібник призначений для студентів першого курсу, які у другому семестрі продовжують вивчення програмування мовою C / C++. У ній розглядаються основи динамічної роботи з пам’яттю, структури даних, алгоритми та методи їх обробки. Особлива увага приділяється реалізації базових структур даних, таких як списки, стеки, черги, дерева та графи, а також алгоритмам сортування та пошуку.

Матеріал подається у логічній послідовності, що сприяє поступовому засвоєнню тем. Теоретичні відомості супроводжуються прикладами коду та практичними завданнями, які допоможуть закріпити отримані знання. Крім того, до посібника включено лабораторні роботи, спрямовані на формування практичних навичок програмування.

Метою цього посібника є не лише ознайомлення студентів з основами мови C, але й розвиток алгоритмічного мислення, вміння аналізувати та ефективно вирішувати задачі з використанням структур даних і алгоритмів. Сподіваємось, що цей матеріал стане корисним і допоможе студентам у вивченні програмування та розв’язанні реальних інженерних задач.

1. Принципи динамічної роботи з пам’яттю

В попередньому курсі “Основи програмування” та подібних джерелах [1,2] розглядались “стандартні” способи організації даних, засновані на використанні скалярних та структурних типів. Спільна риса цих типів – структури даних мали фіксований розмір протягом роботи з ними (масиви, структури, масиви структур тощо). Це визначає суттєві обмеження. Для багатьох задач потрібно працювати з даними, розмір яких завчасно невідомий. При цьому можливі такі ситуації:

- розмір даних не змінюється під час роботи з ними, але невідомий завчасно. Приклад – програма запитує у користувача кількість студентів у групі, створює дані для кожного студента, обробляє ці дані, при цьому кількість студентів у групі не змінюється в процесі роботи програми
- розмір даних може змінюватись під час роботи з ними. Приклад – аналогічно попередньому, проте під час роботи програми можуть додаватись нові студенти, або вилучатись раніше додані.

В обох випадках потрібно виділяти пам’ять під дані динамічно, і використовувати спеціальний тип даних – покажчики (англ. pointer). Покажчик по суті описує адресу в пам’яті, де знаходяться певні блоки даних.

Пам`ять під дані може виділятися або на етапі компіляції (при цьому її розмір фіксується), або у ході виконання програми – областями вказаного розміру (розмір може змінюватись за потребою).

Виділення пам`яті під час виконання програми надає гнучкість у поданні даних. Пам`ять виділяється та звільняється блоками, що зв`язані між собою за допомогою покажчиків. Такий спосіб організації даних називають **динамічними структурами даних**, оскільки їх розмір може змінюватись у ході роботи.

Для роботи з адресами пам`яті (покажчиками) C++ пропонує досить чисельний набір інструментів:

- унарні оператори (для роботи з довільними покажчиками):
  - ***** – *розіменування;*
  - **&** – *отримання адреси*

- **new –** виділення пам’яті
- **delete** – звільнення пам`яті;

- бінарні оператори (для роботи зі структурами та покажчиками на них):
  - **.** *– отримання поля;*
  - -> *– розіменування поля*.
- спеціальна константа NULL (або nullptr у нових версіях C++) – позначає «пустий покажчик», тобто сигналізує що покажчик не вказує на пам’ять, яку можна використовувати; розіменування пустих покажчиків не дозволяється.

Приклад використання операторів для покажчиків на скалярні типи:

```
int value = 3; // змінна зберігає ціле значення
int *pointer = &value; // змінна зберігає покажчик на ціле значення, в даному випадку адресу змінної value
cout<<*pointer<<endl; // розіменування покажчика, виводить значення області пам’яті, куди вказує покажчик, тобто значення змінної value
*pointer = 5; // змінюємо значення області пам’яті, куди вказує покажчик; при цьому змінюється значення змінної value, але не змінюється значення змінної pointer
pointer = new int; // динамічно виділяємо пам’ять під ціле значення, зберігаємо адресу; при цьому змінюється значення змінної pointer, але не змінюється значення змінної value
*pointer = 7; // змінюємо значення області пам’яті, куди вказує покажчик; на відміну від попереднього випадку значення змінної value не змінюється, бо змінна pointer більше не вказує на адресу змінної value
delete pointer; // звільняємо динамічно виділену пам’ять
pointer = new int(11); // однією операцією динамічно виділяємо пам’ять та ініціалізуємо її
delete pointer; // звільняємо динамічно виділену пам’ять
pointer = NULL; // присвоюємо змінній pointer спеціальне значення – пустий покажчик – щоб випадково не використати її після того, як звільнили пам’ять
```

Приклад використання операторів для покажчиків на масиви (розмір масиву задається під час створення, але може бути невідомим під час компіляції):

```
int static_array[5]={1,2,3,4,5}; // змінна зберігає масив цілих чисел; якщо значення елементів масиву задаються під час ініціалізації – розмір масиву можна явно не задавати (int static_array[] {1,2,3,4,5};)
```

```
    cout<<static_array[2]<<endl; // вивести елементи масиву з індексом 2 (оскільки індекси починаються з 0, має вивести значення 3)
    int size = sizeof(static_array)/sizeof(static_array[0]); // розраховує розмір масиву (кількість елементів), лише для статично заданих масивів
    int *pointer = static_array; // змінна зберігає покажчик на масив, також на перший елемент масиву
    cout<<*pointer<<endl; // розіменування покажчика, виводить значення області пам'яті, куди вказує покажчик, тобто значення першого елемента масиву static_array[0]
    *pointer = 11; // змінюємо значення області пам'яті, куди вказує покажчик; при цьому змінюється значення елемента масиву static_array[0], але не змінюється значення змінної pointer
    int dynamic_size = get_size(); // отримуємо розмір масиву (функція get_size() має бути реалізована додатково, наприклад читати розмір з файлу чи запитувати у користувача)
    pointer = new int[dynamic_size]; // динамічно виділяємо пам'ять під масив цілих чисел розміру dynamic_size, зберігаємо адресу; при цьому змінюється значення змінної pointer, але не змінюється значення елементів масиву static_array
    for(int i=0; i<dynamic_size; i++) {
        pointer[i] = get_value();
    } // в циклі заповнюємо значення елементів динамічно створеного масиву (функція get_ value() має бути реалізована додатково)
    pointer[1] = 7; // змінюємо значення елементу динамічного масиву з індексом 1
    delete [] pointer; // звільняємо динамічно виділену пам'ять; важливо використати форму оператора delete для видалення саме масиву (з додатковими символами []), інакше буде втрачено пам'ять
    pointer = NULL; // присвоюємо змінній pointer спеціальне значення – пустий покажчик – щоб випадково не використати її після того, як звільнили пам'ять
```

Приклад використання операторів для покажчиків на структури:

```
    Node value; // змінна зберігає структуру типу Node
    value.d = 3; // змінюємо значення цілого поля структури
    value.p = NULL; // змінюємо значення поля-покажчика
    Node *pointer = &value; // змінна зберігає покажчик на структуру, в даному випадку адресу змінної value
    cout<<(*pointer).d<<endl; // розіменування поля покажчика
    cout<<pointer->d<<endl; // те саме одним оператором
    (*pointer).d = 5; // змінюємо значення поля
```

```
    pointer->d = 5; // те саме
    pointer = new Node; // динамічно виділяємо пам’ять під структуру, зберігаємо адресу
    pointer->d = 7; // змінюємо поле динамічно виділеної структури
    pointer->p = new Node; // виділіємо ще одну динамічну структуру, зберігаємо її адресу в полі першої структури
    pointer->p->d = 11; // змінюємо поле другої структури; оператор розіменування поля структури можна використовувати повторно для доступу до вкладених структур
    pointer->p->p = NULL; // змінюємо поле другої структури


    delete pointer->p; // звільняємо другу структуру
    delete pointer; // звільняємо першу структуру
    pointer = NULL; // присвоюємо змінній pointer спеціальне значення – пустий покажчик – щоб випадково не використати її після того, як звільнили пам’ять
```

Працювати з покажчиками треба дуже акуратно, щоб не виникли помилки одного з наступних типів:

- розіменування неініціалізованого покажчика, наприклад Node *ptr; ptr->d = 1; – треба обов’язково виділяти пам’ять (або отримувати адресу раніше виділеної пам’яті) перед тим як працювати з даними через покажчик (виправлений приклад Node *ptr **= new Node**; ptr->d = 1;);
- розіменування пустого покажчика, наприклад Node *ptr = NULL; ptr->d = 1; – перед роботою з покажчиком, який може бути пустим, треба обов’язково перевірити що він не є пустим (виправлений приклад Node *ptr = NULL; **if(ptr!=NULL)** {ptr->d = 1;});
- втрата пам’яті, або засмічення пам’яті (memory leak), наприклад Node *ptr = new Node; ptr = new Node; – треба обов’язково звільняти всю пам’ять, яку раніше виділили (виправлений приклад Node *ptr = new Node; **delete ptr;** ptr = new Node;);
- розіменування звільненого покажчика (use after free), наприклад Node *ptr = new Node; delete ptr; ptr->d = 1; – якщо покажчик залишається доступним в програмі, після звільнення рекомендується присвоїти йому

значення NULL, щоб випадково не працювати з ним (виправлений приклад Node *ptr = new Node; delete ptr; **ptr = NULL; if(ptr!=NULL)** {ptr->d = 1;});

- повторне звільнення покажчика (free after free), наприклад Node *ptr = new Node; delete ptr; delete ptr; – аналогічно попередньому, якщо покажчик залишається доступним в програмі, після звільнення рекомендується присвоїти йому значення NULL (виправлений приклад Node *ptr = new Node; delete ptr; **ptr = NULL; if(ptr!=NULL)** { delete ptr;})

Окрім операторів new та delete, стандартна бібліотека (<stdlib.h>) надає більш низькорівневі функції для керування розподілом пам`яті:

- calloc(), malloc(), realloc() – виділення;
- free() – звільнення.

В C++ рекомендовано для керування розподілом пам`яті використовувати оператори **new** та **delete** .

В мові C++ (на відміну від мови C) існує ще один тип даних – посилання (reference). Посилання працюють аналогічно покажчикам, з певними відмінностями. Посилання мають наступний синтаксис:

```
int value=5;
int& ref_value = value; // посилання на змінну цілого типу
cout<<ref_value<<endl; // виводить значення змінної, на яку посилається ref_value, тобто в даному випадку значення 5
ref_value = 7; // змінює значення змінної, на яку посилається ref_value, тобто в даному випадку змінної value
```

Деякі відмінності посилань від покажчиків:

- Посилання мають простіший синтаксис – не потребують явних операторів розіменування та взяття адреси;
- Посилання не можуть бути пустими (мати значення NULL) – це з одного боку спрощує роботу з ними (не потрібні перевірки на пусті значення), але з іншого боку обмежує можливі сфери використання;

- Посилання не передбачають динамічного виділення чи звільнення пам’яті – вони завжди мають вказувати на змінну, пам’ять під яку була виділена раніше (статично або динамічно);
- Посилання завжди вказує на одну область пам’яті (яка задається під час ініціалізації посилання) – можна змінити значення змінної, на яку вказує посилання, але не можна змусити посилання вказувати на іншу змінну, для цього треба створити інше посилання.

Використання посилань там, де це доцільно, може спростити код та зробити його більш зрозумілими – але треба розуміти обмеження посилань та їх відмінність від покажчиків. Також посилань немає в мові С, тому не варто використовувати їх в коді, який має співпрацювати з мовою С.

2. Динамічні структури даних. Лінійні списки

Довільна програма створюється для обробки даних, від способу організації яких суттєво залежать алгоритми її роботи, тому вибір структур даних має першочергове значення.

Використовують як лінійні, так й нелінійні динамічні структури:

- лінійні списки (стеки, черги);
- дерева (бінарні, двійкового пошуку, збалансовані, …).

Вказані структури розрізняються як способами зв`язку між окремими елементами структури, так й доступними операціями.

Динамічні структури дозволяють гнучко та ефективно працювати з даними, розмір яких заздалегідь невідомий, а також пришвидшити роботу з даними при виконанні операцій додавання, вилучення, пошуку, впорядкування даних [3-5]. Існуючі можливості для ефективного представлення та обробки різноманітної інформації є предметом подальшого розгляду. Для динамічних структур основними складовими є:

- базові типи даних – структури, покажчики, масиви;
- операції – виділення та звільнення пам`яті, доступу до даних через покажчики.

Деякі лінійні структури даних:

- зв’язні списки – зберігають кожен елемент як спеціальну структуру даних (вузол списку), що містить значення елементу та покажчик на сусідні вузли. Розрізняють однозв’язні списки (покажчик тільки на наступний елемент) та двозв’язні списки (покажчики на наступний та попередній елемент);
- списки на основі масивів – виділяють пам’яті більше, ніж потрібно для зберігання елементів, тому можна додавати елементи в завчасно виділену область пам’яті. При спробі додати більше елементів, ніж виділено

пам’яті, може або повертати помилку, або виділяти масив більшого розміру і копіювати існуючі елементи туди;

- циклічні списки – «ніколи не завершуються», оскільки наступним елементом після останнього вважається перший. Обхід циклічних списків треба реалізовувати акуратно, щоб не вийшло нескінченних циклів – зазвичай запам’ятовують, з якого елементу почали обхід, і завершують обхід коли знову дійшли до цього елементу;
- черга – спеціальний вид списків, який підтримує додавання з однієї сторони (в кінець черги), а видалення з іншої сторони (початок черги);
- стек – спеціальний вид списків, який підтримує додавання та видалення елементів лише з однієї сторони;
- двостороння черга (дек) - спеціальний вид списків, який підтримує додавання та видалення елементів лише з обох сторін (початок та кінець), але не в середині.

Кожна з розглянутих структур має як свої переваги, так і недоліки, які будуть детальніше розглянуті під час подальшого опису цих структур. Тому корисно знати про існування різних структур даних і використовувати в кожному випадку найбільш доцільну структуру даних.

Далі будемо розглядати, як можна було б реалізувати відповідні структури даних «з нуля», використовуючи лише можливості мови C/C++. Варто розуміти, що в стандартній бібліотеці мови C++ вже є готові реалізації для багатьох структур даних, зокрема двозв’язний список std::list, однозв’язний список std::forward_list, список на основі масиву std::vector, черга std::queue, стек std::stack, двостороння черга std::deque. Для багатьох задач можна використовувати ці реалізації. Проте вивчення того, як влаштовані ці структури, може бути корисним для вибору правильної структури в кожній задачі. Також іноді трапляються випадки зі спеціальними вимогами, яким не відповідають бібліотечні структури даних, і в цьому випадку вміння реалізовувати такі структури «з нуля» є корисним.

Елемент динамічної структури представляється структурою, що містить принаймні два поля: для збереження даних (довільні типи та кількість полів), для збереження зв'язків (поля покажчиків). Наприклад, для однозв’язного списку:

```
struct Node {
Data d; //тип Data повинен бути визначений раніше
Node *p; };
Node *newPtr;
```

Для створення та видалення елемента динамічної структури використовують відповідно фрагменти коду

```
newPtr = new Node;
delete newPtr;
```

Звернення до даних (поля структури):

```
newPtr -> d
```

або

```
(*newPtr).d
```

Найпростіші динамічні структури даних – лінійні зв`язані списки (linked list). Зв’язний список складається з послідовності вузлів, у кожному вузлі зберігаються дані та зв’язки з іншими вузлами (покажчики на них). Як мінімум зберігається зв`язок поточного вузла з наступним – такий список називається однозв’язним (singly linked list). Список задається покажчиком на початковий вузол (який також називають головою списку). В останньому вузлі (який також називають хвостом списку) покажчик на наступний вузол задається рівним NULL.

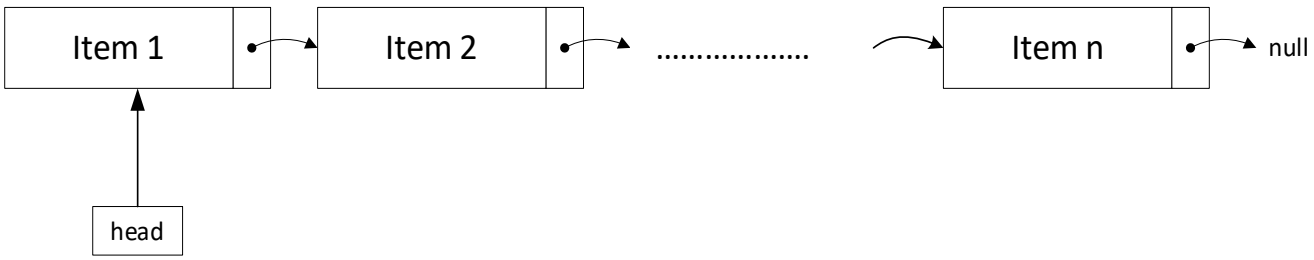


Рисунок 1. Схема однозв’язного списку

Для визначення типу елементу цієї динамічної структури використовують фрагмент коду:

```
struct Node {int dat; Node *next;};
```

Використовують також двозв’язні списки (doubly linked list), в яких кожен вузел зберігає покажчик не лише на наступний, а й на попередній вузол. Покажчик на наступний вузол в останньому вузлі (хвості списку) та покажчик на попередній вузол в першому вузлі (голові списку) задаються рівними NULL.

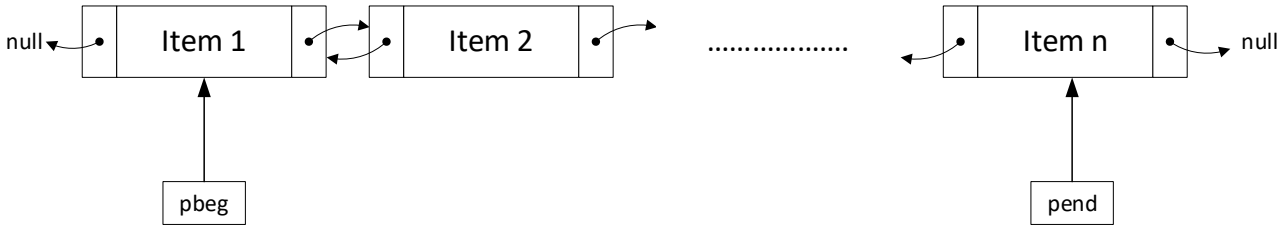


Рисунок 2. Схема двозв’язного списку

Для визначення типу елементу двозв’язного списку використовують фрагмент коду:

```
struct Node {int dat; Node* next; Node* prev};
```

(Важливо!) Втрата покажчиків (розрив зв’язків) призведе до недоступності відповідних елементів, або списку в цілому (поява “сміття”).

Для представлення списку можна використовувати і масиви. Проте таке представлення пов’язане з певними недоліками:

- фіксований розмір масиву призводить до обмеження на розмір списку та зменшення ефективності використання наявної пам`яті;
- якщо дозволяється збільшення розміру масиву шляхом виділення нового масиву – потрібно багато часу на копіювання всіх елементів з попереднього масиву;
- порядок елементів списку визначається розташуванням у масиві, отже вставка та видалення елементу під номером *i* передбачають переміщення всіх елементів після цього номеру. Зокрема, додавання елементу на початку масиву призводить до копіювання всіх елементів масиву.

Втім, представлення списку на масивах має і переваги:

- масив займає неперервну область пам`яті, отже швидкодія програми буде вищою;
- не завжди мови програмування надають явні можливості роботи з адресами.
- приймаючи рішення щодо використання однієї зі структур даних (зв’язних списків чи масивів), рекомендовано зважити різні вимоги конкретної задачі, а також провести виміри важливих параметрів (швидкодії, використання пам’яті тощо) на реалістичних даних.

2.1. Операції над лінійними списками

Для роботи з лінійними списками необхідно визначити основні операції над ними:

- вставка елементу у початок (голову) списку;
- вставка елементу у кінець (хвіст) списку;
- видалення елементу з початку списку;
- видалення елементу з кінця списку;
- вивід списку на екран;
- видалення списку;
- вставка елементу у список після елементу, заданого вказівником;
- видалення елементу зі списку після елементу, заданого вказівником;
- знаходження кількості елементів у списку;
- пошук елемента у списку.

Ми будемо реалізовувати ці операції для однозв'язного списку, заданого типом

```
struct Node {int dat; Node *next;};
```

Для зручності, ми будемо зберігати покажчики на початок і кінець списку у окремій структурі:

```
struct List
{Node *head,*tail;
List():head(NULL),tail(NULL){}
};
```

Строго кажучи, для роботи зі списком достатньо зберігати лише початок, оскільки кінець можна отримати, послідовно проходячи за покажчиками на наступний елемент. Але для багатьох операцій зберігання кінця дозволить реалізувати їх простіше та буде більш ефективним під час виконання.

Вставка елементу у початок списку

```
void add_head(List& lst,int dat)
```

```
    {
       Node* tmp=new Node; //створюємо новий елемент
       tmp->dat=dat; //заповнюємо його дані
       // прив'язуємо його до початку списку
       tmp->next=lst.head;
/* Якщо список був порожній то присвоюємо покажчику на кінець
списку покажчик на його початок */
       if (lst.head==NULL) lst.tail=tmp;
       lst.head=tmp; //зміщуємо початок списку
    }
```

Вставка елементу у кінець списку

```
    void add_tail(List& lst,int dat)
    {
       Node* tmp=new Node; //створюємо новий елемент
       tmp->dat=dat; //заповнюємо його дані
       //обнуляємо його вказівник на наступний елемент
       tmp->next=NULL;
       if (lst.tail!=NULL) //якщо список непорожній
       {
        lst.tail->next=tmp; //прив'язуємо його до кінця списку
        lst.tail=tmp; //зміщуємо кінець списку
       }
       //інакше і кінець і початок вказують на створений елемент
       else lst.tail=lst.head=tmp;
    }
```

Видалення елементу з початку списку

```
    void delete_head(List& lst)
    {
      //якщо список порожній то завершуєм роботу
      if (lst.head==NULL) return;
      // зберігаємо початок списку у тимчасовій змінній
      Node* tmp=lst.head;
      // зміщуємо початок списку
```

```
      lst.head=lst.head->next;
      // якщо список став порожнім то зануляємо і покажчик на кінець списку
      if (lst.head==NULL) lst.tail=NULL;
      delete tmp; // видаляємо елемент
    }
```

Видалення елементу з кінця списку

Ця операція вимагає знаходження елементу, що знаходиться безпосередньо перед кінцевим елементом списку. Для однозв'язних списків ця операція виконується за лінійний час від довжини списку.

```
    void delete_tail(List& lst)
    {
      //якщо список порожній то завершуєм роботу
      if (lst.head==NULL) return;
      /*якщо список складається з одного елементу, то видаляємо його і зануляємо покажчики на початок і кінець списку */
      if (lst.head==lst.tail)
      {
          delete lst.head;
          lst.head=lst.tail=NULL;
          return;
      }
      Node* tmp=lst.head;
      // знаходимо елемент перед останнім
      while (tmp->next!=lst.tail) tmp=tmp->next;
      // видаляємо останній елемент списку
      delete lst.tail;
      // змінюємо вказівник на кінець списку
      lst.tail=tmp;
      // зануляємо покажчик на наступний елемент у кінці списку
      lst.tail->next=NULL;
    }
```

Вивід списку на екран

Ця операція може бути реалізована як ітеративно, так і рекурсивно

```
void print_list(Node* head) //ітеративний підхід
{
while (head!=NULL) {
   cout<<head->dat<<" ";
   head=head->next;
}}
void print_list_rec(Node* head) //рекурсивний підхід
{
if (head==NULL) return;
cout<<head->dat<<" ";
print_list_rec(head->next);
}
```

Зауважимо, щоб вивести на екран вміст списку, починаючи з кінця і закінчуючи початком списку, необхідно поміняти місцями останні дві команди рекурсивної реалізації. Для ітеративної реалізації це буде зробити складніше. Проте слід розуміти, що максимальна кількість рекурсивних викликів обмежена, тому для дуже великих списків рекурсивна реалізація може не працювати.

Видалення списку

Ця операція може бути реалізована як ітеративно, так і рекурсивно. Ітеративна реалізація полягає в послідовному виклику функції **delete_head,** поки список не порожній. Тому ми приведемо лише рекурсивну реалізацію операції.

```
//рекурсивна функція видалення
void delete_list_rec(Node* head)
{
if (head==NULL) return;
delete_list_rec(head->next);
```

```
delete head;
}
//допоміжна функція
void delete_list(List& a)
{
Delete_list_rec(a.head)
a.head=a.tail=NULL;
}
```

Вставка елемента у список після елементу, заданого вказівником

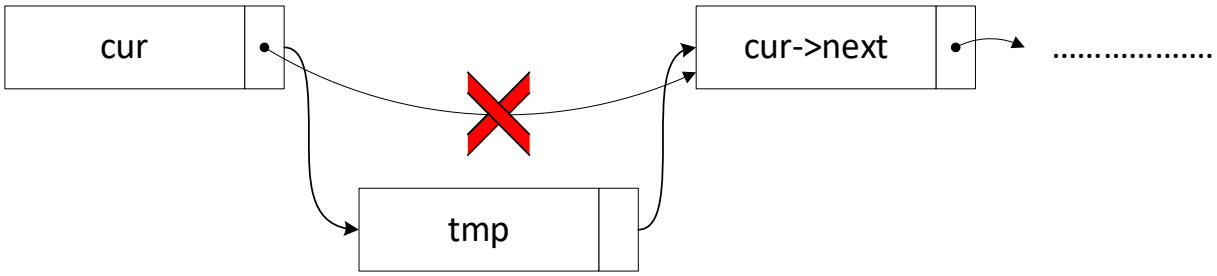


Рисунок 3. Схема операції вставки елемента

```
void add_after_pt(List& lst, Node* cur, int dat)
{
Node* tmp=new Node; //створюємо новий елемент
tmp->dat=dat; //заповнюємо його дані
//прив'язуємо до нього елемент наступний після cur
tmp->next=cur->next;
//
if (cur->next==NULL) lst.tail=tmp;
cur->next=tmp; // прив'язуємо до cur новий елемент.
}
```

Видалення елемента зі списку після елементу, заданого вказівником

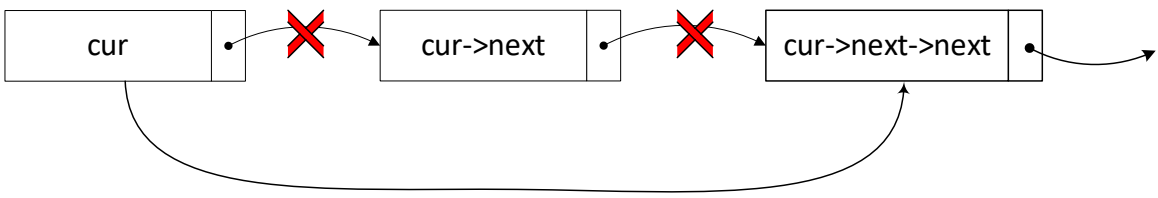


Рисунок 4. Схема операції видалення елемента

```
    void delete_after_pt(List& lst, Node* cur)
    {
       Node*  tmp=cur->next;  //Зберігаємо елемент, який
видалятимемо
          //якщо cur вказував на кінець списку, то завершуємо
роботу
       if (tmp==NULL) return;
       //зв'язуємо cur з наступним після видалення елементом
       cur->next=tmp->next;
       //якщо cur вказує на NULL то зсуваємо кінець списку
       if (cur->next==NULL) lst.tail=cur;
       delete tmp; // звільняємо пам'ять
    }
```

Знаходження кількості елементів у списку

Ітеративна реалізація	Рекурсивна реалізація

`int nelem(Node* head)`
`{`
`  int k=0;`
`  while (head!=NULL)`
`  {`
`  head=head->next;`
`  k++;`
`  }`
`  return k;`
`}`
	`int nelem(Node* head)`
	`{`
	`  if (head==NULL) return 0;`
	`  else`
	`  return 1+nelem(head->next);`
	`}`

Пошук елемента у списку

Ітеративна реалізація	Рекурсивна реалізація
`Node* felem(Node* head,int d)` `{`	`Node* felem(Node* head,int d)` `{`

```
    while (head!=NULL
           &&head->dat!=d)
        head=head->next;
  return head;
  }
```

```
    if (head==NULL||
    head->dat==d) return 0;
    else
    return felem(head->next,d);
}
```

Генерація списку

Під генерацією мається на увазі побудова списку за деяким правилом. У даному прикладі ми розглядаємо побудова списку з N елементів n, n-1, n-2,..., 2, 1.

Ітеративна реалізація

```
List gen_list(int n)
{
  List res;
  Node* p, t = NULL;
  for (int i = 1; i <= n; i++){
   p = new Node;
   p->dat = i;
   p->next = t;
   if (i==1) res.tail=p;
   t = p;
   }
   res.head=p;
   return res;
  }
```

Рекурсивна реалізація

```
List gen_list_rec(int n)
{
if (n==0) return List();
List res;
res.head=new Node;
res.head->dat = n;
List next=gen_list_rec(n-1);
res.head->next = next.head;
if (n==1) res.tail=res.head;
else res.tail=next.tail;
return res;
}
```

Основна складність у рекурсивній реалізації це зберегти посилання на кінець списку.

Робота з циклічним списком

Циклічний список реалізується аналогічно зв'язному списку (однозв'язному чи двозв'язному), з тою відмінністю, що останній елемент посилається не на пустий покажчик, а на початок списку, таким чином «зациклюючи» його. Для двозв'язного

циклічного списку аналогічно перший елемент посилається на останній. Реалізація більшості операцій аналогічна зв'язному списку, проте необхідно акуратно обходити елементи, щоб не утворився нескінченний цикл.

Розглянемо приклад ітеративної реалізації пошуку в однозв'язному циклічному списку.

```
Node* felem(Node* head,int d)
 {
     if (head == NULL) { return NULL; }
     Node* current = head;
     do {
         if (current->dat == d) { return current;}
         current = current ->next;
     } while (current!=head)
     return NULL; // не знайдено
    }
```

На відміну від аналогічної реалізації для зв'язного списку, ми не можемо зупинити пошук, коли поточний покажчик стане пустим – оскільки у непустому циклічному списку всі покажчики будуть непустими. Замість цього, ми порівнюємо значення поточного покажчика зі значенням початку списку – коли ця умова виконується, це означає, що ми обійшли весь список і повернулись на початок, тому треба завершувати пошук.

Задачі

- Лінійний зв`язний список містить послідовність цілих чисел. Написати функцію для:
    - знаходження максимального з чисел;
    - знаходження кількості чисел у послідовності;
    - перевірки належності заданого числа до послідовності;
    - перевірки наявності двох однакових чисел у послідовності.

- Рядки тексту являють собою прізвища, які можуть повторюватися. Потрібно прочитати текст і надрукувати кожне прізвище по одному разу. Порядок прізвищ не має значення. Умови:
    - джерело інформації - файл (текстовий);
    - кількість прізвищ не відома;
    - для тимчасового внутрішнього збереження інформації необхідно скористатися динамічною структурою.
- У файлі зберігається послідовність цілих чисел. Написати функції потрібні для побудови впорядкованого списку елементів заданої послідовності.
- Написати функцію для зчеплення двох лінійних однозв`язних списків (результат – список, отриманий в результаті додавання в кінець першого списку елементів з другого).
- Написати функції для визначення кількості додатних чисел у елементах списку:
    - нерекурсивну;
    - рекурсивну.
- Написати функції розташування елементів нециклічного однозв’зного списку у зворотному порядку (обернення списку):
    - у новому списку;
    - без побудови нового списку.
- Написати рекурсивні функції для виконання розглянутих операцій зі списками:
    - виведення елементів списку (розглянути порядок: починаючи з початку, з кінця списку);
    - додавання в кінець;
    - додавання в середину списку.
- Написати рекурсивну та нерекурсивну функції для визначення суми елементів числової послідовності, представленої зв`язним списком.

- Написати функції додавання та вилучення рядка зі списку за умови, що елемент списку зберігає вказівник на рядок і кількість m його повторень. При першому додаванні рядка полю m присвоюється 1, потім m збільшується на 1 при додаванні і зменшується на *1* при вилученні, а при вилученні рядка з $m = 1$ знищується елемент, що подає рядок.
- Гра - “лічилка” задається натуральними N і M. Числа 1, …, N розташовано колом, і від числа 1 починається відлік; M-е число вилучається. Відлік поновлюється з елемента, наступного за вилученим, і так до вилучення всіх чисел. Написати підпрограму друку за N і M послідовності чисел, які вилучаються, наприклад, при $N = 4$ і $M = 3$ це буде 3, 2, 4, 1. Пропонується реалізувати рішення з використанням зв’язного списку, масиву, рекурентного співвідношення.

2.2. Стеки та черги

Стеки та ***черги*** – структури даних, які можна розглядати як різновиди більш загальних спискових структур даних, але вони мають власні особливості. Стеки та черги знаходять найширше використання в алгоритмах розв`язання різноманітних задач. Ефективність роботи з ними значною мірою впливає на складність алгоритму в цілому. Тому вони заслуговують на окремий розгляд.

- ***Стек (stack)*** – лінійний список, де операції додавання, вилучення та доступу до елемента виконуються тільки в кінці списку. Принцип роботи з даними при використанні стеку називається LIFO (Last In First Out).

Аналогії:

- купа книг на столі;
- патрони у магазині автомату.

Для представлення стеків використовують послідовні та зв`язані способи.

- ***Черга (queue)*** – лінійний список, в якому елементи вилучаються з початку, а додаються в його кінець Принцип роботи з даними при використанні черги називається FIFO (First In First Out).

Аналогія – звичайна черга у магазині.

Двостороння черга (double-ended queue, або скорочено deque, дек) – лінійний список, в якому операції додавання та вилучення можна виконувати і на початку, і в кінці.

Аналогія – послідовність книг на полиці без проміжків, де доступ можливий з обох кінців.

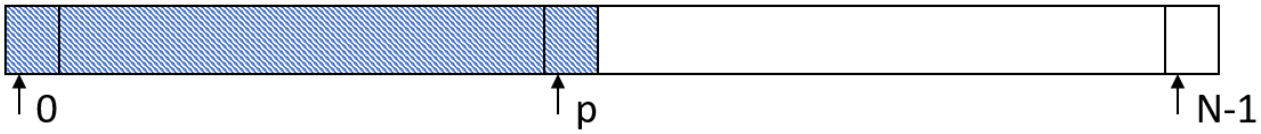
Послідовне зберігання стеків

Для послідовного зберігання одного стеку використовується масив d з N елементами.

Наприклад:

```
const int N=100;
```

```
int d[N], p=-1;
```



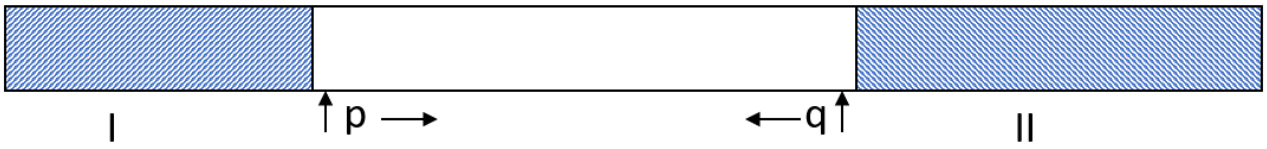
- p == -1 – порожній стек;
- додавання елемента: d[++p] = v;
- вилучення елемента: v = d[p--];

Проблеми: неоптимальне використання пам'яті, особливо якщо потрібно одночасно працювати з кількома стеками.

Для зберігання двох стеків доцільно використовувати один масив, при цьому один зі стеків зберігається на початку масиву і зростає в прямому напрямку, а інший – зберігається наприкінці масиву і зростає в зворотньому напрямку.

Наприклад:

```
const int N=100;
int d[N], p=-1, q=N;
```



- p==-1, q==N – порожні стеки;
- p == q-1 – переповнення стеків.

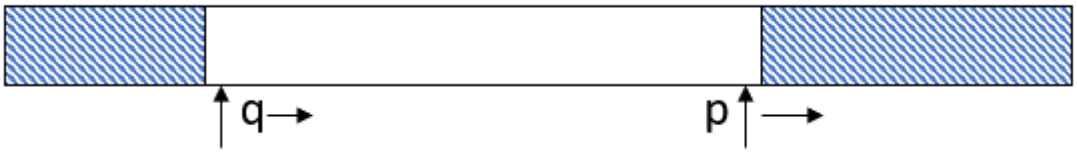
Більше 2 стеків:

- необхідний попередній розподіл пам`яті масиву;
- потрібен перерозподіл пам`яті у випадку переповнення одного з стеків (іноді з врахуванням швидкості зростання стеків).

Послідовне зберігання черг

Для послідовного зберігання однієї черги використовують масив. Зберігаються два індекси початку та кінця черги, при цьому масив вважається циклічним – тобто після останнього елементу іде перший. Наприклад:

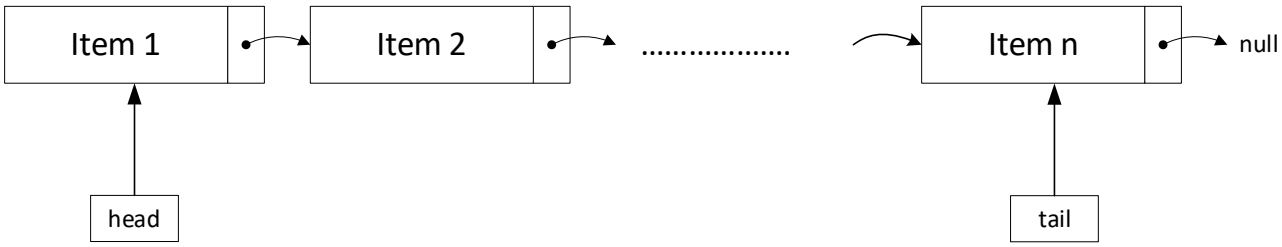
```
const int N=100;
int d[N], p=0, q=-1;
```



- на початку p==0; q==-1;
- порожня черга p>q;
- вилучення елемента v = d[(p++)%N];
- додавання елемента d[(++q)%N] = v;
- потрібен перерозподіл пам`яті коли q-p > N.

Зв`язане зберігання стеків та черг

При зв'язаному зберіганні стеків та черг проблеми переміщення та перерозподілу пам`яті зникають. Стеки реалізують аналогічно лінійним спискам. Черги реалізують аналогічно лінійним спискам але з двома покажчиками, на початок та на кінець списку.



Основні операції зі стеками та чергами:

- Вилучення елемента зі стеку (аналогічно вилученню з початку лінійного списку)
- Додавання елемента в стек (аналогічно додаванню елемента в початок лінійного списку)
- Вилучення елемента з черги (аналогічно вилученню з початку лінійного списку)

- Додавання елемента в чергу (аналогічно додаванню елемента в кінець лінійного списку)

Представляти чергу можна використовуючи циклічний список - досить одного покажчика dl.

Двостороння черга (дек) реалізується аналогічно звичайній. Легко додавати елементи з обох кінців та вилучати з початку, але складно вилучати з кінця.

Всі операції нескладно виконуються з двосторонньою чергою, реалізованою списком з двома зв`язками.

Отже, приходимо до таких висновків:

- Переваги та недоліки послідовних та зв`язних способів представлення стеків та черг значною мірою аналогічні до переваг та недоліків відповідних способів представлення лінійних списків.
- Варто враховувати, що множина операцій з стеками та чергами є суттєво обмеженою у порівнянні зі списками.

Стеки та рекурсія

Кожну рекурсивну програму можна переписати ітеративно за допомогою стеків. Загальний підхід до перетворення рекурсивної функції в ітеративну можна описати наступним чином:

- визначаємо, які значення змінюються між викликами рекурсивної функції. Зазвичай це аргументи функції, але також це можуть бути локальні змінні, які визначаються перед рекурсивним викликом, а використовуються після нього;
- створюємо стек, кожен елемент якого зберігає усі потрібні значення;

- на початку виконання функції додаємо в стек початкові значення відповідних параметрів;
- створюємо цикл з умовою «поки стек непустий»;
- всередині тіла циклу беремо параметри зі стеку, і далі продовжуємо обчислення з ними аналогічно тілу рекурсивної функції;
- замість рекурсивного виклику вставляємо додавання поточних значень параметрів на стек;
- потрібно акуратно організувати порядок виконання операцій в циклі, щоб він відповідав порядку операцій під час рекурсивних викликів.

Існує спеціальний випадок рекурсії, коли перетворення з рекурсивної в ітеративну функцію працює простіше і не вимагає використання стеку. Це так звана «хвостова рекурсія» (tail recursion), коли робиться лише один рекурсивний виклик, і він є останньою операцією в рекурсивній функції. В цьому випадку не треба використовувати стек – можна просто оновлювати значення локальних змінних.

Слід також зазначити, що використання цього підходу використання стеку для перетворення з рекурсивних до ітеративних функцій працює в усіх випадках, але отримана ітеративна функція буде виконувати приблизно таку ж кількість обчислень, як і початкова рекурсивна функція. Іноді рекурсивні реалізації алгоритмів можуть бути менш ефективними порівняно з ітеративними (наприклад, обчислення чисел Фібоначчі – рекурсивна реалізація працює за експоненційний час, тоді як ітеративна – за лінійний час). Тобто в деяких випадках можна використати додаткові структури даних та змінену структуру алгоритму для отримання більш ефективної ітеративної реалізації – але в цьому випадку немає загального підходу, треба будувати ітеративний алгоритм в кожному випадку окремо.

Розглянемо для прикладу функцію Аккермана:

$$A(n,x,y)=\begin{cases} x+1, & якщо\ \ n=0, \\ x, & якщо\ \ n=1\ \&\ y=0, \\ 0, & якщо\ \ n=2\ \&\ y=0, \\ 1, & якщо\ \ n=3\ \&\ y=0, \\ 2, & якщо\ \ n\geq 4\ \&\ y=0, \\ A(n-1,A(n,x,y-1),x), & якщо\ \ n\neq 0\ \&\ y\neq 0 \end{cases}$$

Властивості функції Аккермана:

- «вкладеність рекурсії»;
- існує рекурсивне означення;
- швидко зростає;
- дозволяє виразити традиційні арифметичні операції:
  - n=0 A(0, x, y) = x+1;
  - n=1 A(1, x, y) = x+y;
  - n=2 A(2, x, y) = x*y;
  - n=3 A(3, x, y) = x^y .

Рекурсивне обчислення функції Аккермана:

```
int accrec(int n, int x, int y){
  return (n&&y ?
     accrec(n-1, accrec(n, x, y-1), x) :
     smacc(n, x));
}
//функція Аккермана при n=0|y=0
int smacc(int n, int x){
  switch (n) {
    case 0: return x+1;
    case 1: return x;
    case 2: return 0;
    case 3: return 1;
    default: return 2;}
}
```

Ітеративне обчислення функції Аккермана:

```
// St - стек, обчислюємо  A(n,x,y)
∅ ⇒ St;
(n,x,y) ⇒ St;
repeat  { St ⇒ (n,x,y);
   if n≠0 & y≠0  then  (n,x,y-1) ⇒ St;
   else { t=smacc(n,x); St  ⇒  ;
            if St = ∅ then {break, t - result}
        else { St ⇒ (n,x,y)
              (n-1,t,x) ⇒ St }
            }}
```

Задачі

- Записати обчислення функції Аккермана з використанням стека. Здійснювати контроль переповнення стека, перевищення заданого числа кроків обчислення, та значень у стеку.
- Записати обчислення функції F(m,n) з використанням стека.

$$F(m,n)=\begin{cases} m+n+1, & \text{якщо } mn=0 \\ F(m-1,F(m,n-1)), & \text{якщо } mn\neq 0 \end{cases}$$

- Розглянути випадок зв`язного зберігання стеків для нерекурсивного обчислення A(n,x,y) та F(m,n).
- Написати рекурсивну та нерекурсивну функції для визначення суми елементів лінійного списку цілих чисел, якщо застосовано зберігання:
  - послідовне;
  - зв`язане.

2.3. Стисле та індексне зберігання списків

Нехай список $B = \langle k_1, k_2, \ldots, k_n \rangle$ містить кілька елементів з однаковим значенням ***v***. Побудуємо $B' = \langle k'_1, k'_2, \ldots, k'_m \rangle$, де $k'_i = (i.k_i)$.

Стислим зберіганням В є таке представлення B', в якому не записуються послідовні елементи із однаковим значенням.

Наприклад: $B =< C, Y, Y, Y, S, S, S, S, S, S, H, H, H, H, H, H, H, T, T >$;

$B' = \langle (1, C), (3, Y), (6, S), (7, H), (2, T) \rangle$.

У залежності від способу зберігання розрізняють *послідовне* (з використанням масивів) та *зв`язане* (з використанням зв'язних списків) *стислі зберігання* В.

Пошук і-го елемента списку при зв`язаному стислому зберіганні здійснюється методом послідовного перегляду, а при послідовному стислому зберіганні – методом бінарного пошуку.

Переваги та недоліки послідовного та зв`язаного стислих зберігань аналогічні перевагам та недолікам послідовного та зв`язаного зберігань лінійних списків.

Спосіб **індексного зберігання** передбачає, що початковий список $B = \langle k_1, k_2, \ldots, k_n \rangle$ ділиться на кілька списків $B_1, B_2, \ldots B_m$ так, що кожний елемент списку В потрапляє тільки в один з підсписків. Додатково використовується індексний список з m елементів, що показують на початки списків $B_1, B_2, \ldots B_m$.

Поділ списку В на підсписки $B_1, B_2, \ldots B_m$ здійснюється так, щоб всі елементи з певною властивістю p_i потрапили в один підсписок B_i. Для поділу списку В використовується деяка індексна функція $g(k)$.

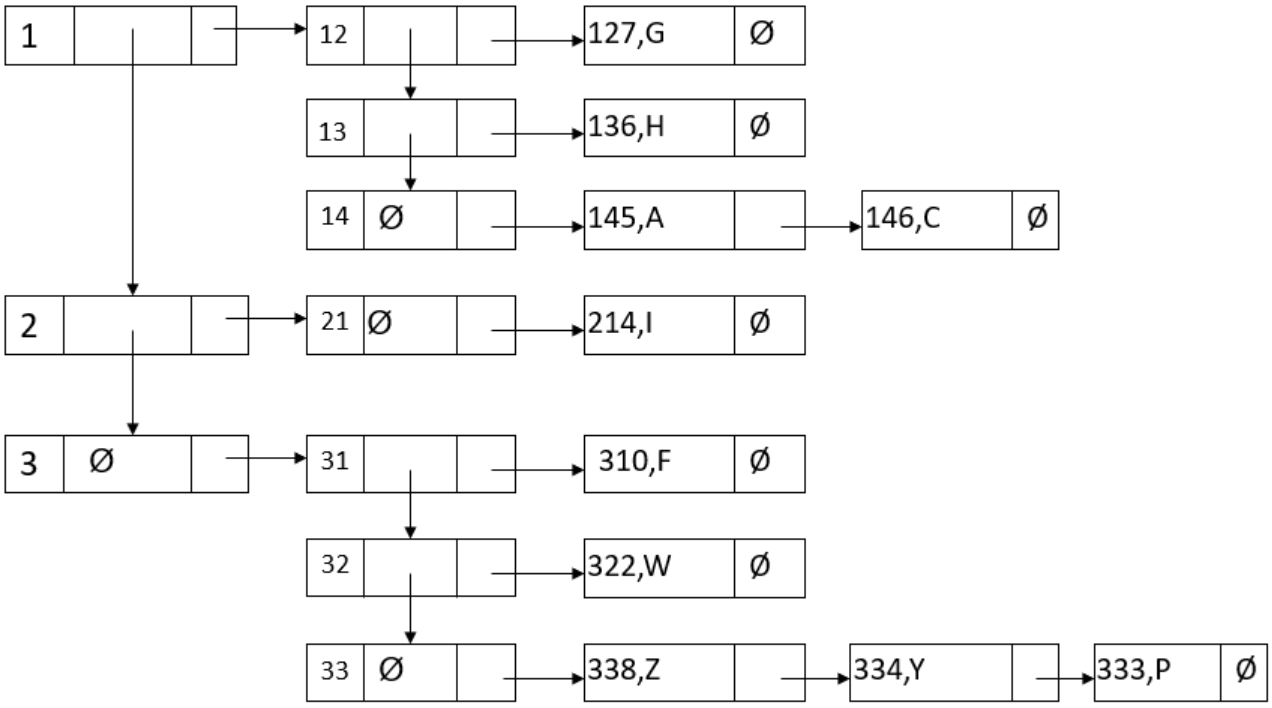
Переваги індексного зберігання в тому, що для пошуку елемента ***k*** із властивістю p_i досить переглянути тільки елементи підсписку B_i.

Оскільки довжина списку індексів Х як правило відома, часто використовують ***послідовно-зв`язане індексне зберігання***: Х – ***послідовно***, забезпечуючи прямий доступ до його елементів (підсписків); $B_1, B_2, \ldots B_m$ – ***зв`язано***, що спрощує додавання та вилучення елементів.

Звісно, що для індексного списку можна використовувати не обов`язково послідовне збереження.

Наприклад: B=<(338,Z), (145,A), (136,H), (214,I), (146,C), (334,Y), (333, P), (127,G), (310,F), (322,W)>.

Розподіл на підсписки та індексування здійснюємо таким чином, щоб до одного підсписку потрапляли елементи, які в першій компоненті мають однакові дві цифри.



Такий спосіб зберігання називають – ***(зв'язано-зв'язаним)-зв'язаним індексним зберіганням.***

Зауваження

- У розглянутих прикладах використовували стислі та індексні способи представлення, які є простішими й частіше використовуються.
- Можливі й інші варіанти з зв'язаних та послідовних способів для представлення індексних та інформаційних.
- Послідовні представлення надають можливості для прямого доступу, або бінарного пошуку.

- Зв'язані представлення дозволяють лише послідовний доступ до елементів.

Поради

- Обирати найбільш адекватний спосіб представлення даних, виходячи насамперед з основних пріоритетів.
- При обранні способу збереження для списку слід враховувати які операції і з якою частотою будуть виконуватися над списком.
- Враховувати як розмір задач так й значення, які потрібно представляти списками.
- Старанно підходити до обрання індексної функції.

Задачі

- На вході дві послідовності цілих чисел: M={m1,m2, … , m10000}, N={n1,n2, … , n10000}. Відомо, що не менше 92% елементів послідовності M дорівнюють 0. Представити списки у вигляді послідовного стислого зберігання у масиві структур з спеціальною відміткою кінця списку (за останнім елементом списку). Обчислити скалярний добуток векторів M та N, тобто величину $\sum_{i=1}^{10000} m_i n_i$.
- До лінійного списку F з m цілих чисел, більшість елементів якого дорівнюють 0, застосоване зв`язане стисле зберігання. Написати функцію для визначення і-го елемента списку.
- Лінійний список F цілих чисел зберігається як послідовно-зв`язаний індексний список так, що числа, які мають 2 однакові останні цифри, розміщені в один підсписок. Написати функцію перевірки: чи є в списку елемент зі значенням v?
- Многочлен від однієї змінної з цілими коефіцієнтами можна подати зв`язаним списком, впорядкованим за зростанням ступеня змінної без одночленів з нульовими коефіцієнтами. Написати функції, які реалізують над многочленами:

- додавання многочленів;
- множення многочлена на одночлен;
- множення двох многочленів;
- ділення двох многочленів з часткою та остачею у вигляді многочленів.

- На вході задана послідовність трійок (x_i,y_i,p_i), де x_i - німецьке слово, y_i - його англійський еквівалент, p_i - частота використання (у %) слова x_i у “типовому німецькому тексті”. Для послідовності пар (x_i,y_i), застосоване послідовно-зв`язане індексне зберігання. Елементи, що мають однакову першу букву німецького слова, вміщуються в один зв`язаний список, впорядкований за спаданням частоти використання. Написати програму формування цієї структури та здійснення послівного перекладу німецького речення. За відсутності перекладу конкретного слова залишати його неперекладеним.

- На вході задана послідовність цілих додатних чисел, менших за 1000. Написати функцію для читання чисел та організації їхнього (зв`язано-зв`язаного)-зв`язаного індексного зберігання. Тобто використовувати, по аналогії з розглянутим, - зв`язане, у вигляді сукупності списків, представлення для індексного списку та зв`язане представлення для інформаційних списків.

3. Сортування

Задача сортування в загальному вигляді полягає в тому, що значення елементів послідовності треба поміняти місцями так, щоб утворилася неспадна (або незростаюча) послідовність $v_1, v_2, \ldots, v_n$, тобто виконувалися нерівності $v_1 \leq v_2 \leq \ldots \leq v_n$ (або $v_1 \geq v_2 \geq \ldots \geq v_n$).

Алгоритми сортування можна поділити на два великих класи:

- сортування *порівняннями (comparison sort)* — алгоритм може використовувати лише порівняння двох елементів і не може використовувати додаткові знання про типи елементів, їх внутрішню структуру, діапазони значень, розподіл значень тощо. Тобто не передбачається ніяких обмежень на значення елементів, які сортуються, — аби їх можна було порівнювати;
- алгоритми другого класу суттєво використовують обмеження множини значень елементів, або їхні структурні особливості.

З іншого боку, алгоритми сортування поділяються на алгоритми сортування масивів та списків у оперативній пам'яті (***внутрішнє сортування***) і файлів (***зовнішнє сортування***).

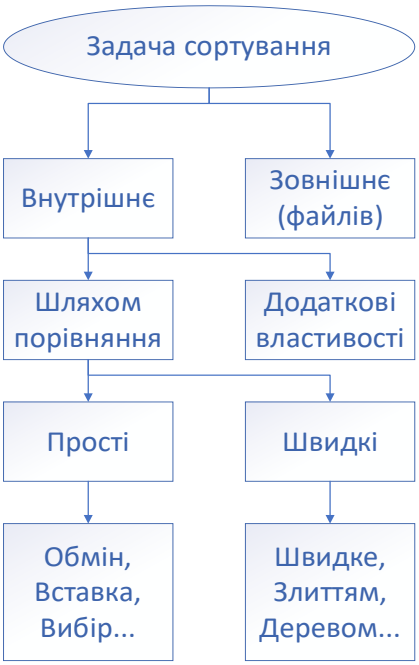


Рисунок 5. Типи та способи сортування

Прості алгоритми сортування

Серед методів сортування “на тому ж місці” (тобто без використання додаткової пам`яті) виділяють:

- **Обмінне сортування** – при перегляді здійснюється обмін двох сусідніх елементів, що не відповідають потрібному порядку, доки такі пари існують. Наприклад – “*сортування бульбашкою*” (bubble sort). Використовує $O(n^2)$ операцій (порівнянь). Функція для сортування бульбашкою наведена нижче

```
void bubble(int a[], int n){
  int is, i, c;
  do {is = 0;
    for (i=1; i<n; i++)
    if (a[i-1] > a[i]) {
      c = a[i]; a[i] = a[i-1];
      a[i-1] = c; is = 1;
    }
  } while (is);
}
```

- **Сортування вставкою (insertion sort)** – при перегляді для кожного наступного елемента знаходиться його «місце» у підмасиві попередніх елементів, куди і вставляється відповідний елемент. Таким чином, після k кроків сортування ми гарантовано отримуємо відсортований префікс довжини k у вихідному масиві. Функція для сортування вставкою наведена нижче

```
void insert(int a[], int n){
  int i, j, c;
  for (i=1; i<n; i++) {
    c = a[i];
    for (j=i-1; j>=0 && a[j]>c; j--)
      a[j+1] = a[j];
    a[j+1] = c;
  }
}
```

- **Сортування вибором (selection sort)** – кожну і-ту ітерацію сортування ми шукаємо найменший елемент серед тих, що залишилися серед елементів з індексами більшими або рівними за і, та поміщаємо його на позицію і. Функція для сортування вибором наведена нижче

```
void select(int a[], int n){
  int i, j, c;
  for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
      if (a[i] > a[j]) {
        c = a[i]; a[i] = a[j]; a[j] = c;
        }
}
```

Серед простих методів сортування з виділенням додаткової пам`яті виділяють:

- **Сортування визначенням позиції** – порівняннями кожного елемента з іншими визначають кількість менших за нього (k). Це визначає місце елемента у впорядкованому масиві (k+1). Потім переставляють елементи згідно з їх місцями. Кількість порівнянь – $(n^2-n)/2$, кількість переміщень –$(9n/4)$. Складність $O(n^2)$.
- **Сортування квадратичним вибором** має такий алгоритм:
  1) Масив B розбивається на частини $B_1, B_2, \ldots, B_m$, де $m=\sqrt{n}$ та $|B_i| = \sqrt{n}$.
  2) Для кожної частини B_i знаходять мінімальний елемент g_i .
  3) Мінімальний елемент для B буде рівним $\min(g_1, g_2, \ldots, g_m) = g_i$.
  4) g_i вилучають та замінюють новим мінімальним елементом в B_i .
  5) Алгоритм продовжують до вичерпання множин $B_1, B_2, \ldots, B_m$.

Потрібно $O(n\sqrt{n})$ порівнянь, але використовується додаткова пам`ять для зберігання $(g_1, g_2, \ldots, g_m)$.

Швидкі алгоритми сортування

Швидке сортування (quicksort)

1) певним чином вибирається *опорне значення (pivot) v;*

2) елементи масиву *A* обмінюються так, що масив розбивається на дві ділянки — ліву та праву: в лівій ділянці елементи мають значення не більше v, а у правій — не менше;

3) після цього достатньо окремо відсортувати ці дві ділянки.

Зауваження

- У якості опорного значення можна вибрати, наприклад, перший елемент, або елемент, що знаходиться посередині масиву. Від вибору опорного елемента залежить якість поділу масиву на дві частини, і відповідно швидкість алгоритму.
- Рекурсивна функція використовує покажчики low – перший елемент, hi – останній елемент ділянки масиву.
- Якщо hi-low > 1 – використовуються два покажчика p та q, що рухаються назустріч, виконуючи обміни. Ліворуч p – менші за опорне, праворуч q – більші. В кінці займає своє місце розділяючий елемент.
- Сортуються ділянки масиву low – (p-1) та (p+1) – hi
- Час сортування суттєво залежить від початкового списку. Час мінімальний – якщо на кожному кроку поділу отримуємо підсписки приблизно рівної довжини, тоді – O(n log n).
- Якщо початковий список мало відрізняється від упорядкованого – потрібно $n^2/2$ кроків. Складність “у найгіршому” – $O(n^2)$.
- Потребує додаткової пам`яті (порядку $O(\log_2 n)$) для виконання рекурсивної функції.
- Складність “у середньому” – O(n*log n).

Сортування злиттям (merge sort)

- В основі алгоритмів сортування злиттям лежить об'єднання двох упорядкованих послідовностей в одну.
- Використовує додатковий масив розміру n .
- За одне проходження виконується n порівнянь, а всього потрібно здійснити $\log_2 n$ таких проходжень.
- Час роботи: $T(n)= O(n \log n)$.

Алгоритм можна реалізувати як рекурсивно так і ітеративно.

Будь-який алгоритм сортування, оснований лише на порівнянні елементів, має *складність у "найгіршому випадку"*, яка оцінюється знизу як $O(n*\log n)$, тому *задача сортування має оцінку складності* $O(n \log n)$. У розглянутих реалізаціях алгоритмів можна використовувати як послідовні, так й зв`язні способи представлення списків для підвищення ефективності алгоритмів. Досить поширеними є застосування рекурсивних підходів у алгоритмах сортування.

Втім, для сортування даних, які мають деяку внутрішню структуру або обмеження на їх значення існують спеціальні методи з лінійною оцінкою часової складності.

Метод черпаків

Нехай у нас є масив $A=\{a_1, a_2, \ldots, a_n\}$ з обмеженнями на значення елементів $0 \le a_i \le m-1$.

Алгоритм:

1. Утворюємо m порожніх черг (черпаків). Кожна відповідає числу [0, m-1].
2. Масив $A=\{a_1, a_2, \ldots, a_n\}$ розкладаємо по чергам. Кожен елемент a_i заносимо у чергу з номером a_i .
3. Послідовно зливаємо черги. Для кожного *i* наповнення (i+1)-черги заносимо в кінець і-черги.

Складність алгоритму $O(m+n)$

Застосування методу черпаків:

- Цифрове сортування (розподільне);
- Бітове сортування;
- Сортування ланцюжків різної довжини.

Цифрове сортування (radix sort)

- Дії з черпаками (розкладання та злиття) здійснюються поступово за всіма десятковими розрядами чисел послідовності, починаючи з наймолодших розрядів чисел.
- Кількість операцій для сортування n t-розрядних чисел – O(n*t), тому й T(n)= O(n*t).
- Якщо для початкового списку й черпаків використовувати зв`язне представлення, то можна уникнути використання додаткової пам`яті, переміщення елементів здійснюючи за рахунок зміни покажчиків (зв`язків між елементами.)

Сортування файлів (зовнішнє сортування)

Технології зовнішнього сортування відмінні. *Причини*:

- Значний час доступу до даних (файлів) ;
- Немає прямого (довільного) доступу до даних.
- Безпосередньо доступний один елемент файлу.

Основні алгоритми основані на злитті частково впорядкованих (попередньо) файлів. Злиття використовує тільки прості структури даних, проходити які достатньо послідовно.

Один з найпростіших методів зовнішнього сортування називається *збалансованим злиттям*. *Відрізок* - максимальна послідовність елементів, упорядкована за зростанням значень. Наприклад, у послідовності <2, 8, 3, 7, 6, 5, 3, 4, 1> є шість відрізків: <2, 8>, <3, 7>, <6>, <5>, <3, 4>, <1>.

4) Спочатку відрізки по черзі копіюються в допоміжні файли F_3 і F_4 (*розподіл*). F_3: <2, 8, 6, 3, 4>; F_4: <3, 7, 5, 1>.

5) Пари відрізків файлів F_3 і F_4 об'єднуються у більш довгі відрізки і по черзі копіюються у F_1 і допоміжний файл F_2 (*злиття*). F_1: <2, 3, 7, 8, 1, 3, 4>; F_2: <5, 6>.

6) Потім пари відрізків файлів F_1 і F_2 об'єднуються у файли F_3 і F_4, і так далі, поки в результаті чергового злиття не утвориться єдиний відрізок. F_3: <2, 3, 5, 6, 7, 8>; F_4: <1, 3, 4>. F_1: <1, 2, 3, 3, 4, 5, 6, 7, 8>; F_2:<>.

Зауваження

- Якщо перед певним кроком було M відрізків, то після нього їх стає не більше $[(M+1)/2]$. Отже, таких кроків не більше $\log N$, де N – число елементів файлу.
- На кожному кроці N елементів копіюються в інші файли.
- Складність алгоритму $O(N \log N)$.
- Потрібно враховувати, що це операції з повільною (зовнішньою) пам`яттю.
- Кількість кроків й складність розглянутого алгоритму буде зменшуватись із зростанням розміру відрізків.
- Можна використати внутрішнє сортування (наприклад *сортуюче дерево*) для створення відносно великих відрізків.
- У реальних задачах часто сортують структури даних, складені з декількох різнотипних полів, найчастіше великого розміру. Один із способів прискорення роботи з даними полягає у створенні додаткового масиву індексів або вказівників на елементи основного масиву структур. Наприклад, дані про абонента телефонної мережі містять номер, прізвище, адресу та багато іншої інформації. Шукати дані про абонента треба як за номером, так і за прізвищем. <("Кнут", 31), ("Ахо", 86), ("Хопкрофт", 12), ("Ульман", 24)> Отже у файлі для кожного запису зберігається пара номерів (індексів).
- упорядкування за 1 компонентою - <1, 0, 3, 2>.
- упорядкування за 2 компонентою - <2, 3, 0, 1>.

Задачі

- Написати функцію, яка вставляє в упорядкований список у зв`язному зберіганні новий елемент, зберігаючи впорядкованість.
- Написати функцію, яка для двох скінчених множин цілих чисел, поданих зв`язними впорядкованими списками знаходить:
  - їхній перетин;
  - їхнє об`єднання.
- Написати функцію для сортування списку з n цілих чисел методом квадратичного вибору.
- Написати функцію для сортування списку з n цілих чисел методом “визначення позиції”.
- Написати функцію для сортування з використання методу черпаків.
- Написати функцію для цифрового сортування.
- Написати функцію для бітового сортування.
- Реалізувати метод «зовнішнього сортування».
- На вході задані неупорядковані списки студентів чотирьох клубів C_1, C_2, C_3, C_4, кожний з яких має не більше 250 членів. Скласти програму для визначення всіх студентів, що є членами принаймні трьох клубів. (Для спрощення припустити, що члени клубів вказані цілими додатними числами за номерами їхніх студентських квитків).

4. Матриці та багатовимірні масиви

Розглянемо матрицю B={k_{ij} | 1≤i≤m, 1≤j≤n}. Матрицю можна розглядати як лінійний список B=<s_1,s_2, …, s_m>, елементи якого s_i – рядки матриці. Таким чином, при зберіганні за рядками можемо представити матрицю у вигляді одного лінійного списку k_{11}, k_{12}, …, k_{1n}, k_{21}, …, k_{m1}, k_{m2}, …, k_{mn}.

Це надає можливості для прямого доступу до елементів, де адресу елемента k_{ij} можна визначити за допомогою рівності

$$Adr\ k_{ij} = Adr\ k_{11} + n(i-1) + j - 1$$

Матрицю можна розглядати також як лінійний список B=<g_1,g_2, …, g_n>, елементи якого g_i - стовпці матриці. Таким чином, при зберіганні за стовпчиками можемо представити матрицю у вигляді одного лінійного списку k_{11}, k_{21}, …, k_{m1}, k_{12}, …, k_{m2}, k_{1n}, …, k_{mn}.

Адресу елемента k_{ij} можна визначити за допомогою рівності

$$Adr\ k_{ij} = Adr\ k_{11} + m(j-1) + i - 1$$

Існують спеціальні типи матриць:

- ***Трикутна матриця*** - для j>i (j<i) елемент k_{ij}=v.
- ***Стрічкова матриця*** ширини d>0 – для abs(i-j)>d k_{ij}=v .
- ***Розріджена матриця***, якщо більшість її елементів дорівнює 0.

Для вказаних матриць використовують спеціальні стислі зберігання, в яких не записуються елементи із значенням v.

Зберігання трикутної матриці

У послідовному стислому зберіганні трикутних матриць з упорядкуванням за рядками вилучають всі елементи k_{ij} для j>i .

Потрібно $n(n+1)/2$ ділянок пам`яті замість n^2.

$$Adr\ k_{ij} = Adr\ k_{11} + i(i-1)/2 + j - 1 \text{ за умови } (i \geq j)$$

Наприклад: A(20,20) → B(210). ($a_{1,1}$, $a_{2,1}$, $a_{2,2}$, $a_{3,1}$, $a_{3,2}$, $a_{3,3}$, …, $a_{20,1}$, $a_{20,2}$, … $a_{20,20}$)

Зберігання стрічкової матриці

У послідовному стислому зберіганні стрічкової матриці з упорядкуванням за рядками вилучають всі елементи k_{ij} для *abs(i-j)>d* . Потрібно *(2d+1)n-2d* ділянок пам`яті.

$$Adr\ k_{ij} = Adr\ k_{11} + (i-1)(2d+1) + j - i \text{ за умови } (abs(i-j) \leq d).$$

Наприклад: A(20,20) ширини 4 $\rightarrow$ B(172). ($a_{1,1}$, $a_{1,2}$, $a_{1,3}$, $a_{1,4}$, $a_{1,5}$, $a_{2,1}$, $a_{2,2}$, $a_{2,3}$, $a_{2,4}$, $a_{2,5}$, $a_{2,6}$, …, $a_{20,16}$, $a_{20,17}$, $a_{20,18}$, $a_{20,19}$, $a_{20,20}$)

Зберігання розрідженої матриці

Для розрідженої матриці B створимо список $F' =< k'_{11}, k'_{12}, \dots, k'_{mn} >$, де $k'_{ij} = (i, j, k_{ij})$ та підсписок F'' з F' вилучивши всі елементи $k'_{ij} = (i, j, v)$.

Можливі варіанти стислого зберігання:

- **Словник за ключами** (DOK - Dictionary of Keys) будується як словник, де ключ це пара (рядок, стовпець), а значення це відповідний рядку та стовпцю елемент матриці.
- **Список списків** (LIL - List of Lists) будується як список рядків, де рядок це список вузлів виду (стовпець, значення).
- **Список координат** (COO - Coordinate list) зберігається список з елементів виду (рядок, стовпець, значення).
- **Стиснене зберігання рядком** (CSR - compressed sparse row, CRS - compressed row storage, Єльський формат). Ми представляємо вихідну матрицю $M^{n \times m}$, що містить N_{NZ} }ненульових значень у вигляді трьох масивів:
  - масив значень - масив розміру N_{NZ}, в якому зберігаються ненульові значення взяті підряд з першого непустого рядка, потім йдуть значення з наступного непустого рядка і т.д.
  - масив індексів стовпців - масив розміру N_{NZ} що зберігає номери стовпців, відповідних елементів із масиву значень.

- масив індексації рядків - масив розміру $n + 1$, що для індексу i зберігає кількість ненульових елементів у рядках до $i - 1$ включно. Варто зазначити, що останній елемент масиву індексації рядків збігається з N_{NZ}, а перший завжди дорівнює 0.

Приклад зберігання у форматі CSR

Нехай $M = \begin{pmatrix} 1 & 2 & 0 & 3 \\ 0 & 0 & 4 & 0 \\ 0 & 1 & 0 & 11 \end{pmatrix}$. Тоді масив значень = {1, 2, 3, 4, 1, 11}, масив індексів стовпців = {0, 1, 3, 2, 1, 3}, масив індексації рядків = {0, 3, 4, 6} – спочатку зберігається 0 як замикаючий елемент.

- **Стисне зберігання стовпцем** (CSC - compressed sparse column, CCS - compressed column storage) Те саме що і CRS, тільки рядки і стовпці змінюються ролями - значення зберігаємо по стовпцях, по другому масиву можемо визначити рядок, після підрахунків з третім масивом - дізнаємося стовпці.
- **Прямокутне зв`язане індексне зберігання** використовує масиви покажчиків для рядків та стовпчиків. Кожний вузол списку має два поля покажчиків – на наступний у рядку та наступний у стовпчику.

Багатовимірні масиви

Нехай маємо m-вимірний масив $B(n_1,n_2,\ldots,n_m)$. Доступ до елемента здійснюється через індекси $i_1,i_2,\ldots,i_m$. *Нехай* $p_0=n_1n_2\ldots n_m$; $p_1=n_2n_3\ldots n_m$; … $p_{m-1}=n_m$; $p_m=1$. H = {p_1, p_2, …, p_m} - інформаційний вектор масиву B. Адреса елемента з індексами $i_1 i_2 \ldots i_m$ задається рівнянням

$$adr\ k_{i_1 i_2 \ldots i_m} = adr\ k_{11\ldots1} + \sum_{t=1}^{m} (i_t - 1) p_t$$

Метод Айліфа для доступу до багатовимірних масивів не потребує операцій множення для доступу до його елементів, проте використовує додаткові масиви.

Розглянемо на прикладі масиву $C(n_1,n_2,n_3)$. Метод використовує: $B(n_1*n_2*n_3)$ для зберігання елементів масиву C , $G(n_1*n_2)$ та $F(n_1)$ - для зміщення:

$$G[i] = 1 + (i-1) * n_3\ ,\ \ F[j] = 1 + (j-1) * n_2.$$

Тоді адреса елементу за індексом ijk запишеться у вигляді

$$adrC_{ijk} = \ adrC_{111} + g[j - 1 + f[i]] + k - 1\ .$$

Задачі

- Написати функцію для побудови зв`язаного стислого зберігання розрідженої матриці B[10,40].
- Написати функцію для побудови послідовно-зв`язаного індексного зберігання розрідженої матриці B[10,40].
- Написати функцію для копіювання розрідженої матриці B[10,40], що задана у прямокутному зв`язаному стислому зберіганні.
- На вході елементи матриці A(10,10), впорядковані за рядками. Написати програму для введення матриці A та організації прямокутного зв`язного стислого зберігання. Обчислити матрицю $B=A^2$, запам`ятовуючи її як прямокутну зв`язну стислу структуру. Надрукувати елементи матриці B за рядками.
- Гра “Життя” відбувається на прямокутному полі комірок, кожна з яких має 8 сусідів і може вміщувати “організм”; *m(k)* – кількість сусідніх з *k* комірок, зайнятих “організмами”. Нове покоління “організмів” утворюється з попереднього покоління за правилами:
- “організм” в комірці k зберігається за умови $2 \le m(k) \le 3$, інакше вмирає;
- “організм” в порожній комірці k народжується при m(k)=3.
- Написати програму, яка читає початкову конфігурацію та кількість поколінь n, обчислює n-те покоління та відображає його.

5. Дерева.

5.1. Методи зберігання

Дерево - зв`язний ациклічний граф (теорія графів).

Але в програмуванні дерева насамперед використовуються як гнучкий й потужний спосіб представлення для складної інформації, оскільки дозволяють зберігати не тільки всю потрібну сукупність одиниць даних, а й наявні зв'язки між одиницями даних [6]. Використовуючи певну організацію даних в дереві, можна ефективніше працювати з інформацією ніж при її представленні лінійними списками.

Дерево можна розглядати як скінченну множину вузлів, можливо порожню, з одним виділеним вузлом, що називається коренем, а інші вузли розділені на $m \geq 0$ множин $B_1, B_2, \ldots, B_m$, які не перетинаються і кожна з яких є дерево.

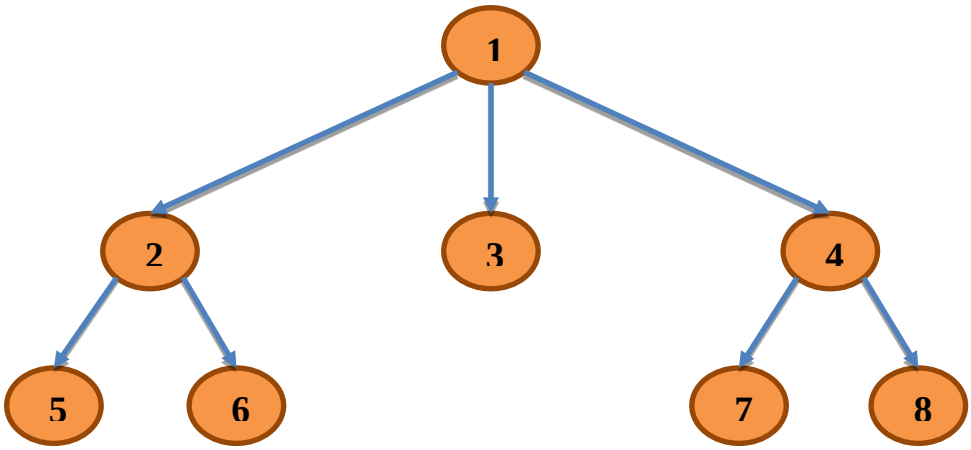


Рисунок 6. Приклад кореневого відміченого дерева

На множині вузлів (вершин) дерева визначені природнім чином бінарні відношення попередників (*батько*) та наступників (*син*).

Основні властивості:

- існує один вузол (*корінь*) без попередників;
- у кожного іншого вузла є попередник (*батько*);
- для будь-якого вузла існує один ланцюг, що зв`язує його з коренем.

Основні поняття:

- ***Листя дерева*** - вузли, що не мають синів. Інші вузли - внутрішні.
- ***Степінь вузла*** - це кількість його синів.
- ***Степінь дерева*** - найбільша степінь по всіх його вузлах.
- Дерево степені ***n*** називається ***повним***, якщо степінь кожного вузла дорівнює ***n*** або ***0***.
- ***Дерево впорядковане***, коли для кожного вузла множина синів є впорядкованою.
- ***Висота дерева*** – максимальна довжина ланцюга від кореня до листів дерева.
- ***Проходження дерева*** - систематичний перегляд всіх його вузлів у певному порядку.

Основні порядки проходження дерев:

Прямий порядок проходження дерева:

- відвідують корінь;
- у прямому порядку проходять всі піддерева у відповідності з їх впорядкуванням.

Обернений порядок проходження дерева:

- у оберненому порядку проходять всі піддерева у відповідності з їх впорядкуванням;
- відвідують корінь.

Порівневий порядок проходження дерева - відвідують корінь, потім поступово вузли на відстані від кореня 1, 2, 3, … .

Для прикладу дерева з рисунка 6 відвідування вузлів у:

- прямому порядку – <1, 2, 5, 6, 3, 4, 7, 8>;
- оберненому порядку – <5, 6, 2, 3, 7, 8, 4, 1>;
- порівневому порядку – <1, 2, 3, 4, 5, 6, 7, 8>.

Традиційні операції з деревами:

- пройти вузли в певному порядку;
- знайти вузол з заданою властивістю;
- визначити батька, або синів заданого вузла;
- вилучити вказаний вузол (піддерево);
- додати новий вузол.

Способи зберігання дерев:

- послідовні (використовують для зовнішнього представлення);
- зв`язані (використовують для внутрішнього представлення).

Послідовні способи - лінійні зображення дерев у вигляді "рядка". Зокрема - *рівневе*, *дужкове*.

Рівневий запис дерева:

- Подання дерева використовує номери рівнів. З кожним вузлом дерева v зв`язується номер рівня - ціле додатне число k_v, що задовольняє умови: якщо u - син v, то $k_u > k_v$, якщо u та w сини v то $k_u = k_w$.
- Це запис усіх вузлів з номерами рівнів у прямому порядку.

Для дерева з рисунка 6 – {(1,1), (2,2), (3,5), (3,6), (2,3), (2,4), (3,7), (3,8)}

Дужковий запис дерева:

- запис дерева з одного вузла - запис цього вузла;
- запис дерева з коренем w та піддеревами $B_1, B_2, \ldots, B_m$ - запис кореня w та в дужках дужковий запис його піддерев $B_1, B_2, \ldots, B_m$.

Для дерева з рисунка 6 – {1(2(5, 6), 3, 4(7, 8))}.

Різні форми зв`язаного зберігання дерев залежать від способу подання відношення "батько" - "син".

Використовують найчастіше наступні форми зберігання:

- ***стандартна***,

- ***обернена***,
- ***розширена стандартна***.

Традиційна домовленість – якщо у вершини відсутній і-тий син, то відсутні й всі наступні (i+1, i+2, …, m).

У ***стандартній*** формі зберіганні:

- зв`язки - від батька до синів;
- представлення вузла дерева степені m містить m покажчиків;
- дерево задають покажчиком на корінь.

```
const int m = 3;
struct TrNd {int dat;
                    TrNd *s[m];};
TrNd*  trptr;
```

У ***оберненій*** формі зберіганні:

- зв`язки - від *синів* до *батька*;
- представлення вузла містить 1 покажчик;
- дерево задають списком покажчиків на листя.

```
const int n = 1000;
struct TrNd {int dat;
                    TrNd *f;};
TrNd*  arrp[n];
```

Розширена стандартна форма :

- є об`єднанням *стандартної* та *оберненої* форм;
- представлення вузла містить ***m+1*** покажчик - на всіх *синів* та *батька*.

```
const int m=3, n = 1000;
struct TrNd {int dat;
                    TrNd *s[m];
                    TrNd *f;};
TrNd  *trptr,  *arrp[n];
```

Функція для проходження дерева в прямому порядку:

```
//дерево подано в стандартній формі
void preord(TrNd* p)
{
  if (p) { cout << p->dat << ' ';
         for (int i=0; i<m; i++) preord(p->s[i]);
         }
}
```

Функція для проходження дерева в оберненому порядку:

```
//дерево подано в стандартній формі
void postord(TrNd* p)
{ if (p) {
          for (int i=0; i<m; i++) postord(p->s[i]);
          cout << p->dat << ' ';
          }
}
```

Для дерева поданого в стандартній формі не рекурсивне рішення використовує стек:

```
st ← Ø;
st ← корінь;
while (st <> Ø) do {
                  p ← st;
                  обробка вузла p ;
                  for (i=m-1; i>=0; i--) st ← (p→s[i]);
                                }
```

Реалізація:

```
const int Mst=100;
int sp=0;
TrNd* st[Mst];
//занесення у стек
void push(TrNd* p)
    { if (sp < Mst) st[sp++] = p;
  else cout << "Stack is FULL !!!" << endl;
    }
```

```
//вилучення зі стеку
TrNd* pop()
   { if (sp) return st[--sp];
     cout << "Stack is EMPTY !!!" << endl;
     return NULL;
    }
//виведення дерева в прямому порядку
void preord_st(TrNd* p)
   { push(p);
      while(sp){
      if (p = pop()) {
              cout << p->dat << ' ';
              for (int i=m-1; i>=0; i--) //перебір піддерев
                 push(p->s[i]);  //занесення у стек
                     }
               }
   }
```

Функція для побудови дерева степені 3 за його поданням у рівневому зображенні. На вході послідовність пар чисел $l_1\, w_1\, l_2\, w_2$ … -1 -1 (l_i - номер рівня, w_i - значення відмітки вузла, -1 -1 - маркер кінця). Побудуємо представлення дерево в розширеній стандартній формі.

```
TrNd* input(int lf, TrNd* pf)
{ int  l;  TrNd*  p;
  if (lev > lf) {
    p = new TrNd; l = lev; p->dat = dn; p->f = pf;
    cout << "Lev = "; cin >> lev;
    cout << "Data = "; cin >> dn; cout << endl;
    for (int i=0; i<m; i++)  p->s[i] = input(l, p);
   }
    else p = NULL;
  return p;
}
```

Функція використовує зовнішні змінні **int lev, dn; ,** в які перед зверненням зчитується перша пара вхідних даних. Звернення до функції може мати вигляд –

if (lev > -1) TrNd* p = input(0, NULL); .

Функція для створення копії дерева степені 3, поданого в розширеній стандартній формі.

```
TrNd* copytr(TrNd*  t, TrNd*  r)
{ TrNd* p;
  if (!t) return t;
  p = new TrNd;  p->dat = t->dat;  p->f = r;
  p->s[0] = copytr(t->s[0], p);
  p->s[1] = copytr(t->s[1], p);
  p->s[2] = copytr(t->s[2], p);
  return p;
}
```

Звернення до функції може мати вигляд – **TrNd* q = copytr(p, NULL);**.

Підсумки

- Були розглянуті основні підходи до представлення дерев (довільної степені m).
- Способи зв`язаного збереження найчастіше використовують для внутрішнього представлення дерев, що визначають прикладні данні.
- Способи послідовного збереження використовують для зовнішнього представлення дерев й надають можливості простого введення відповідних даних.
- У розглянутих прикладах обмежились лише найпростішими й найбільш вживаними реалізаціями послідовних та зв`язаних способів збереження дерев.

Поради

- Обирати (у разі можливості) найбільш адекватний спосіб представлення дерев, виходячи насамперед з наступних потрібних дій.

- Розібратися як з рекурсивними, так й з не рекурсивними способами обходу дерев, оскільки довільна нетривіальна обробка дерева найчастіше передбачає відповідний перебір його вершин.
- Розібратися з термінологією.

Задачі

- Написати функції для побудови внутрішнього представлення дерева степені 3 у стандартній формі за його поданням на вході:
  - у рівневому зображенні;
  - у дужковому зображенні.
- Написати функцію виведення відміток вузлів дерева, заданого у стандартній формі при порівневому обході.
- Написати функцію для визначення висоти дерева степені 3, що зберігається у стандартній формі.
- Написати функцію для визначення кількості вершин дерева степені 3, що зберігається у стандартній формі.
- Написати функцію для копіювання дерева степені 3, заданого у стандартній формі.
- Написати функцію для звільнення пам`яті, що зайнята деревом степені 3, заданим у стандартній формі.
- Написати функції для визначення значення найбільшого елемента у вузлах дерева степені 3, заданого у стандартній формі:
  - рекурсивне рішення;
  - ітеративне рішення.
- Написати функції для пошуку в дереві степені 3, представленого у стандартній формі, заданого значення:
  - рекурсивне рішення;
  - ітеративне рішення.

- Написати функцію для вилучення з дерева степені 3, представленого у стандартній формі, вузла з заданим значенням (вважаємо, що значення у вузлах не повторюються).
- Написати функцію для додавання до дерева степені 3, представленого у стандартній формі, вузла з заданим значенням.
- Написати не рекурсивну функцію проходження дерева степені 3, представленого у стандартній формі, в оберненому порядку.
- Дерево степені 3 зберігається в оберненій формі (масив покажчиків A[n] на листя дерева). Написати функцію для створення копії цього дерева, що зберігається у стандартній формі.

5.2. Бінарні дерева

Бінарні дерева заслуговують на окремий розгляд з декількох причин. Вони знаходять найширше використання та мають певну специфіку. Інформацію, що представлена довільним деревом, й навіть послідовністю дерев довільної степені, можна подати також й бінарним деревом. Вони є основою дуже потужних способів подання інформації, наприклад, – *дерев бінарного пошуку*.

Бінарне дерево – це упорядковане дерево степені 2. Воно має наступні властивості:

- може бути порожнім;
- будь-який вузол може мати другого (правого) сина за відсутності першого (лівого).

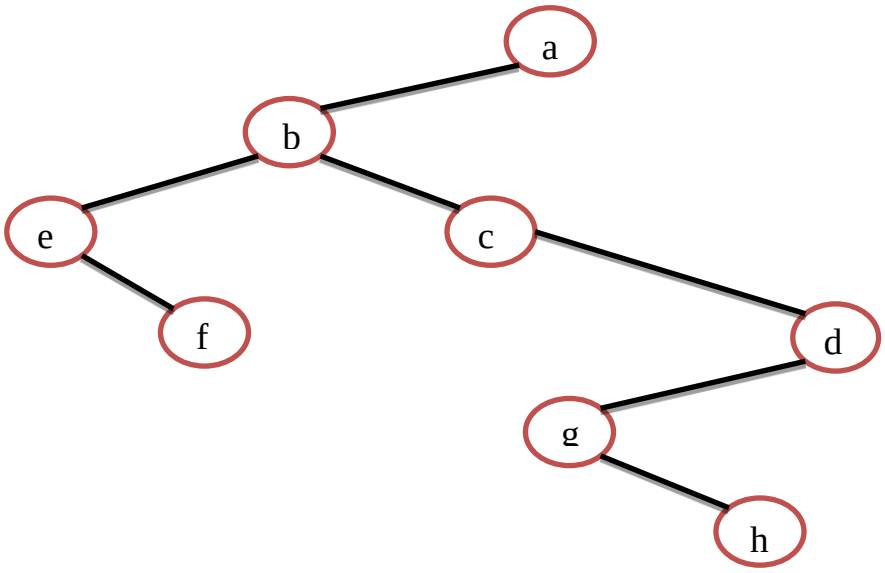


Рисунок 7. Приклад бінарного дерева

Бінарні дерева зберігаються аналогічно звичайним деревам. При розміщенні в стандартній формі:

```
struct BTN {int dat;
     BTN *lt, *rt; };
```

Звісно що можна використовувати й розглянуті вище *обернену* та *розширену стандартну* форми.

Використовують деякі “традиційні” способи представлення “звичайного” дерева, й навіть послідовності дерев довільної степені, бінарними деревами. Існує взаємно однозначна відповідність між скінченними послідовностями звичайних впорядкованих дерев та бінарними деревами.

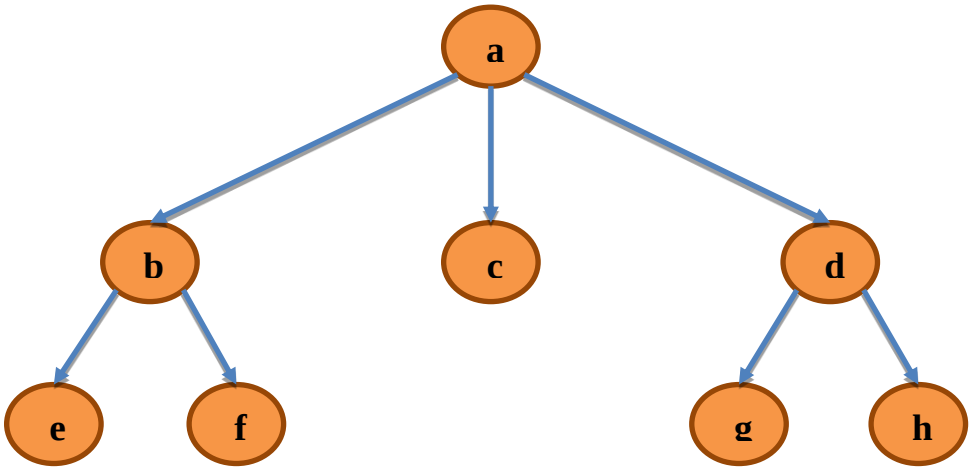


Рисунок 8. Дерево для представлення бінарним деревом

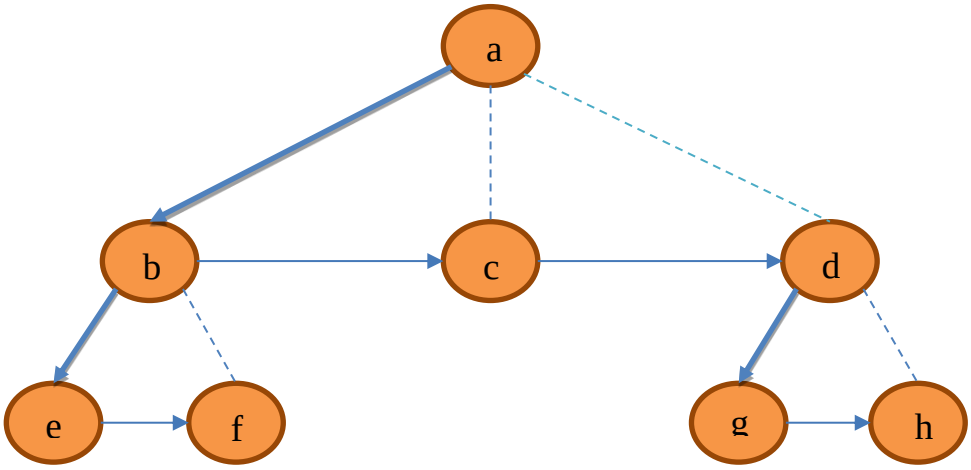


Рисунок 9. Представлення бінарним деревом (зміна зв’язків)

Зберігаються зв’язки *“батько – перший син”*, всі решта вилучаються, також додаються зв’язки між синами одного батька.

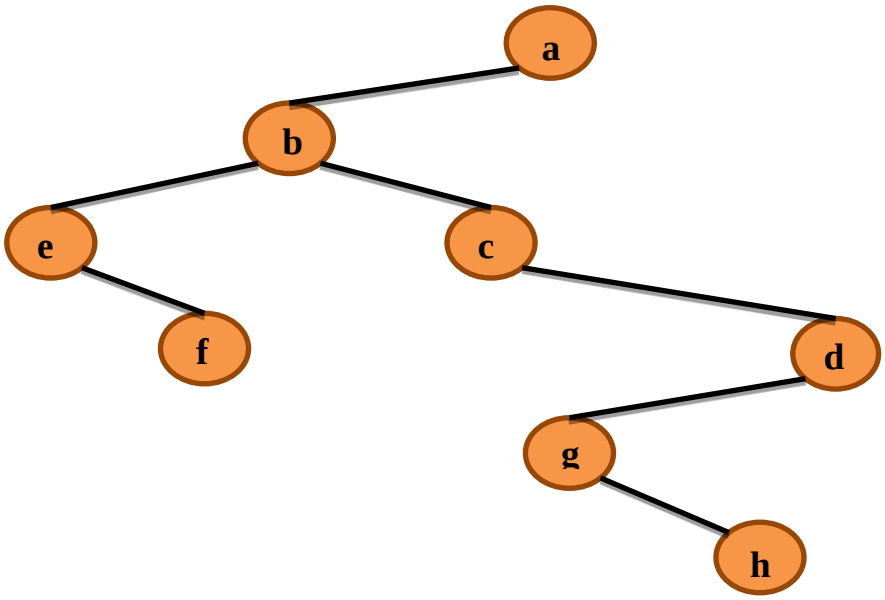


Рисунок 10. Представлення бінарним деревом

У кореневому вузлі вільне поле для правого зв`язку дозволяє приєднати бінарне представлення наступного дерева з послідовності.

Запишемо функцію, що створює відповідне бінарне дерево для послідовності з $n \leq 3$ дерев степені 3. Маємо наступні типи:

```
const int m = 3;
struct TrNd
{ int dat;
  TrNd *s[m];
};

struct BTN
{ int dat;
  BTN *lt, *rt;
};
```

Функція переходу від послідовності з $n \leq 3$ дерев степені 3 до бінарного дерева:

```
BTN* beta(TrNd* p, TrNd* q, TrNd* r)
{   BTN* t;
    if (p)
    {   t = new BTN;
        t->dat = p->dat;
```

```
        t->lt = beta(p->s[0], p->s[1], p->s[2]);
        t->rt = beta(q, r, NULL);
    }
    else  t = NULL;
    return t;
}
```

Проходження бінарних дерев.

Крім розглянутих вище варіантів перегляду всіх вузлів дерева в *прямому*, *оберненому* та *рівневому* порядках, для бінарних дерев дуже часто використовують ***симетричний порядок обходу***, який полягає в наступному:

- проходимо у симетричному порядку ліве піддерево;
- проходимо корінь;
- проходимо у симетричному порядку праве піддерево.

Для прикладу дерева з рисунка 7 послідовність відвідування вузлів у *симетричному порядку* - <e, f, b, c, g, h, d, a>.

Функція для проходження дерева в симетричному порядку

```
//дерево подано в стандарній формі
//виведення дерева в симетричному порядку
void symord(BTN* p)
{   if(p)
    {   symord(p->lt);          //обходимо ліве піддерево
        cout << p->dat << “, “; //обробка вузла
        symord(p->rt);          //обходимо праве піддерево
    }
}
```

Аналогічно можемо реалізувати прямий та обернений обхід

```
//виведення дерева в прямому порядку
void preord(BTN* p)
{    if (p) {   cout << p->dat << ", ";
                preord(p->lt);
```

```
                preord(p->rt);
        }
    }
    //виведення дерева в оберненому порядку
    void postord(BTN* p)
    { if (p) {   postord(p->lt);
                postord(p->rt);
                cout << p->dat << ", ";
        }
    }
```

Не складно для наявної інформації побудувати відповідне представлення у вигляді максимально симетричного бінарного дерева. Складніше буває підтримувати такий стан в ході наступної обробки дерева, що передбачає додавання та вилучення вузлів дерева. Але є відповідні технології, які дозволяють підтримувати "хороший стан" дерева в ході обробки.

Побудуємо бінарне дерево з максимально симетричною структурою (мінімальної висоти) для заданої кількості вузлів. Правило рівномірного розподілу ***n*** вузлів можна визначити рекурсивно:

- перший вузол – корінь дерева;
- створюємо ліве піддерево з кількістю вузлів *nleft = n div 2* ;
- створюємо праве піддерево з кількістю вузлів *nright = n – nleft – 1* .

```
BTN* build(int nn)
{   BTN* p;
    int dd, nleft, nright;
    if(!nn) return NULL;   //порожнє дерево
    nleft = nn/2;      //к-сть вузлів у лівому піддереві
    nright = nn – nleft – 1; // к-сть вузлів у правому
    cout << "Enter node data: ";
    cin << dd;
    cout << endl;
    p = new BTN;  //створюємо корінь
    p->dat = dd;
    p->lt = build(nleft)   //будуємо ліве піддерево
```

```
        p->rt = build(nright)  //будуємо праве піддерево
        return p;
    }
```

Спеціальні способи зберігання. Прошиті дерева

Варто згадати й про інші досить поширені способи представлення бінарних дерев. Спочатку проста арифметика – n вершин бінарного дерева передбачають - 2n полів для зв'язків. З теорії графів – ребер у такому дереві n-1, звідки n+1 поле для зв'язків мають значення NULL. А чи не можна з більшою користю заповнювати "NULL-поля" для зв'язків?

Прошите дерево будується з бінарного дерева шляхом заміни NULL-покажчиків:

- NULL-покажчик на відсутнього лівого сина замінюється покажчиком на попередній вузол у симетричному порядку;
- NULL-покажчик на відсутнього правого сина - покажчиком на наступний вузол у тому ж порядку;
- щоб відрізнити покажчики "*нитки*" від звичайних покажчиків, використовують два додаткових поля в кожному вузлі прошитого бінарного дерева;
- є прив`язка до порядку обходу.

Розглянемо приклад прошитого дерева з симетричним порядком для бінарного дерева з рисунка 7, де послідовність відвідування вузлів у *симетричному порядку* - <e, f, b, c, g, h, d, a>.

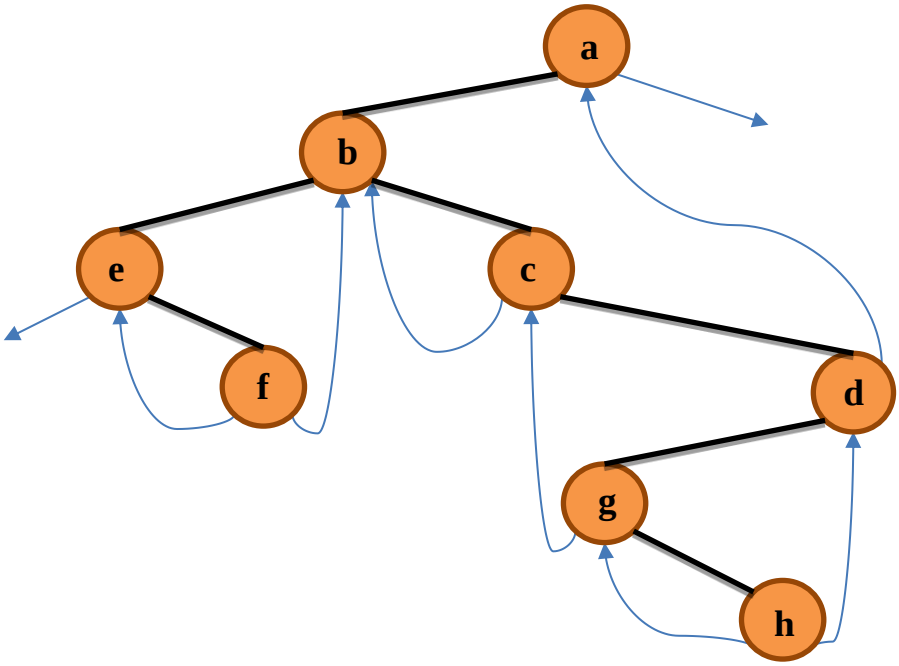


Рисунок 11. Приклад прошитого дерева

Структура вузла такого дерева та потрібні типи даних визначаються:

```
struct TH
{   int dat;
    TH *lt, *rt;
    bool li, ri;
};
```

Якщо **p->li = true**, **то p->lt** - показує на лівого сина, інакше на попередній вузол у симетричному порядку. Аналогічно **p->rt** - показує на правого сина, або на наступний вузол, залежно від значення **p->ri**.

Проходження і побудова прошитого дерева (симетричний порядок)

Функції для проходження прошитого дерева

```
TH* suc(TH* p) //отримання наступного для вузла  p
{   TH* q = p->rt;
    if(p->ri)
        while(q->li)  q = q->lt;
    return q;}
void symm(TH* p)
{   while(p->li) p = p->lt; //вихід на початковий вузол
    while(p) //рух по дереву
```

```
    {   cout << p->dat << “, “;  p = suc(p);
    }
}
```

Запишемо функції для побудови прошитого дерева за звичайним бінарним

```
//допоміжна функція
TH* buildo(BTN* p, TH* sl, TH* sr)
{   TH* t = new TH;
    t->dat = p->dat;
    if((t->li = (p->lt != NULL)))
        t->lt = buildo(p->lt, sl, t)
    else t->lt = sl;
    if ((t->ri = (p->rt != NULL)))
        t->rt = buildo(p->rt, t, sr);
    else t->rt = sr;
    return t;
}
//функція побудови прошитого дерева з бінарного
TH* buildh(BTN* p)
{  return (p ? buildo(p, NULL, NULL) : NULL);
}
```

Послідовні способи зберігання

Знаходять також досить широке практичне застосування послідовні способи “*внутрішнього*” зберігання для бінарних дерев, наприклад, в алгоритмах сортування (“сорт деревом”).

Зберігання бінарного дерева можна організувати в масиві структур з розміщенням вузлів дерева в прямому порядку.

Лівий син вузла p, якщо він існує, завжди розміщується поряд з p (праворуч), а місцезнаходження правого сина вказується значенням поля (r) в структурі. Наявність чи відсутність лівого сина фіксується в структурі логічним полем (isl).

Маємо потрібний тип даних:

```
const int M = 100;     //обмеження на к-сть вузлів в дереві
```

```
struct
 {  int dat;
    unsigned int r;    //розташування правого сина
    bool isl;          //ознака існування лівого сина
 } BTR[M];
int n;                 //к-сть вузлів у дереві
```

dat	a	b	e	f	c	d	g	h
r	0	4	3	0	5	0	7	0
isl	1	1	0	0	0	1	0	0

Рисунок 12. Зберігання дерева з рисунка 7 відповідним масивом

Для вузла з індексом i в масиві ($0 \le i < n$) ми завжди можемо визначити індекс вузла j за певною умовою (за його відсутністю – j = -1):

- лівий син

```
j = (BTR[i].isl ? i + 1 : -1);
```

- правий син

```
j = BTR[i].r-1;
```

- наступний вузол у прямому порядку

```
j = ((i < n - 1) ? i + 1 : -1);
```

- попередній вузол у прямому порядку

```
j = i - 1;
```

- батько

```
if (i)
{   j = i-1;
    if (!BTR[j].isl)
        while (BTR[j].r != i+1)
            j--;
}
else j = -1;
```

Підсумки

- У розглянутий прикладах обмежились лише найбільш принциповими й вживаними способами подання бінарних дерев та здійснення обробки даних, що представлені деревом.
- “Прошивка” бінарного дерева залежить від обраного способу обходу й спрощує подальші дії, що основані саме на цьому способі обходу.
- Повністю аналогічним чином можна організувати “зворотній” обхід “прошитого” дерева у відповідному порядку, починаючи з останнього вузла.

Поради

- Обирати (у разі можливості) найбільш адекватний спосіб представлення бінарних дерев, виходячи насамперед з наступних потрібних дій.
- Розібратися як з рекурсивними, так й з не рекурсивними способами здійснення обробки даних, представлених бінарними деревами.

Задачі

- Написати функції для визначення у бінарному дереві, яке подане у стандартній формі:
  - висоти;
  - кількості листів;
  - кількості вузлів;
  - вузла з заданим значенням;
  - значення найбільшого елемента у вузлах.
- Написати функцію для звільнення всіх ділянок пам`яті, зайнятих вузлами бінарного дерева, поданого у стандартній формі.
- Написати функцію, що будує копію бінарного дерева
- Написати не рекурсивну функцію для визначення кількості вузлів у “прошитому” дереві.

- Написати не рекурсивні функції для “зворотнього” проходження “прошитого” дерева в симетричному порядку, починаючи з останнього вузла.
- Написати не рекурсивні функції (з використанням стека) для друкування відміток вузлів бінарного дерева, поданого у стандартній формі, при його проходженні:
    - в симетричному порядку;
    - в оберненому порядку.

5.3. Дерева бінарного пошуку

Дерево бінарного пошуку, як динамічна структура даних, надає дуже гнучкі та потужні можливості для обробки впорядкованої інформації, а також виступає основою для побудови інших ефективних інструментів подання та обробки даних.

Дерево бінарного пошуку - бінарне дерево, кожний вузол якого ***v*** відмічений значенням ***l(v)*** так що:

- ***l(u)*** < ***l(v)*** для кожного вузла ***u*** з лівого піддерева вузла ***v***;
- ***l(u)*** > ***l(v)*** для кожного вузла ***u*** з правого піддерева вузла ***v***;
- кожне з піддерев є деревом бінарного пошуку.

Дерева бінарного пошуку мають наступну властивість: відмітки вузлів бінарного дерева, що отримані в симетричному порядку обходу утворюють впорядковану послідовність.

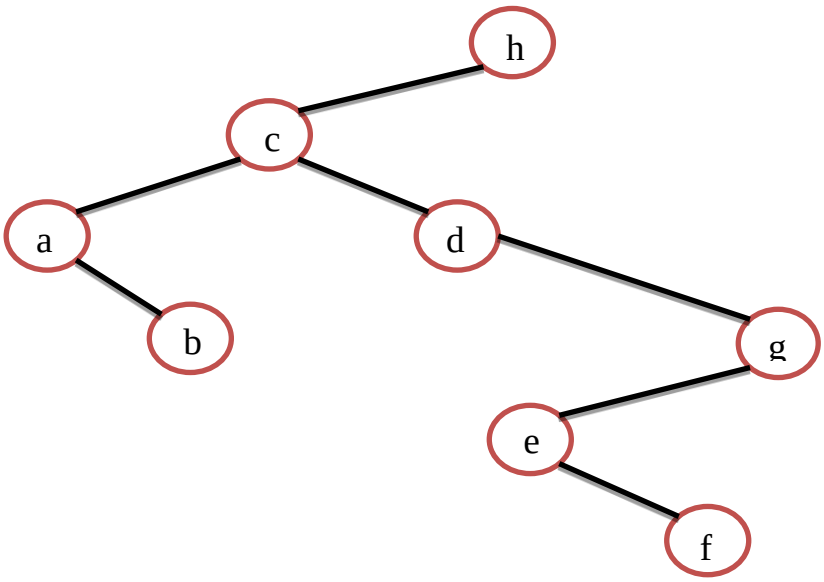


Рисунок 13. Дерево бінарного пошуку. Симетричний порядок обходу вузлів <a, b, c, d, e, f, g, h>

Основні операції для роботи з деревами бінарного пошуку:

- пошук вузла за ключовим значенням;
- вставка вузла у дерево бінарного пошуку;
- вилучення вузла з дерева бінарного пошуку;
- пошук найменшого та найбільшого значення;

- побудова дерева бінарного пошуку за множиною значень;
- перевірка дерева, чи є воно деревом бінарного пошуку.

Реалізуємо ці операції для дерева бінарного пошуку, заданого типом

```
struct BTN {int dat;
                    BTN *lt, *rt;
                   };
```

Пошук у дереві бінарного пошуку

Вхід: дерево T і ключ K.

Задача: якщо є вузол зі значенням K в дереві T, то повернути покажчик на цей вузол.

Алгоритм: Для порожнього дерева повернути покажчик NULL, інакше порівняти K зі значенням кореня X:

Якщо K=X, видати покажчик на цей вузол.

Якщо K>X, шукати ключ K в правому піддереві T.

Якщо K<X, шукати ключ K в лівому піддереві T.

Рекурсивна реалізація	Не рекурсивна реалізація

`BTN* find(BTN* p, int key)`
`{ if (p && p->dat != key)`
`  return (p->dat > key ?`
`  find(p->lt, key):`
`  find(p->rt, key));`
`  return p;`
`}`
	`BTN* fnd(BTN* p, int key)`
	`{ while (p)`
	`  { if (p->dat > key)`
	`  p = p->lt;`
	`  else`
	`  if (p->dat < key)`
	`  p = p->rt;`
	`  else break;`
	`  }`
	`  return p;`
	`}`

Пошук найменшого значення

Рекурсивна реалізація	Не рекурсивна реалізація

```
int fmin(BTN* p)
{ if (p->lt)
        return (fmin(p->lt));
    return (p->dat);
}
```

```
int vmin(BTN* p)
{  while (p->lt)
        p->lt;
    return (p->dat);
 }
```

Побудова дерева бінарного пошуку

Визначимо функцію для створення нового вузла бінарного дерева пошуку:

```
BTN* nwnode(int v, BTN* pl, BTN* pr)
{   BTN* p = new BTN;
    p->dat = v;
    p->lt = pl;
    p->rt = pr;
    return p;
}
```

Скористаємось цією функцією для побудови дерева за даними впорядкованого масиву:

```
BTN* build(int a[], int n)
{ int m;
  if (n) {  m = n/2;
      //Середній елемент масиву є відміткою кореня
       return (nwnode(a[m], build(&a[0], m),
                 build(&a[m+1], n-m-1)));
    }
    return NULL;
}
```

Перевірка чи є дерево деревом бінарного пошуку

```
// vv – зовнішня змінна з дуже малим значенням
bool test(BTN* p)
{  if (!p) return 1; //для порожнього дерева - так
```

```
   if (!test(p->lt)) return 0;  //перевірка лівого
if (p->dat < vv) return 0;  //контроль кореня
vv = p->dat;  //коригування зовнішньої змінної
return (test(p->rt));  //перевірка правого
}
```

Вставка в дерево бінарного пошуку

Вхід: дерево T й значення V.

Задача: додати вузол з значенням V в дерево T (якщо вузол відсутній).

Алгоритм: Якщо дерево порожнє, замінити його на дерево з одним кореневим вузлом (V, NULL, NULL). Інакше порівняти V зі значенням вузла X:

Якщо V<X, рекурсивно додати V в ліве піддерево T.

Якщо V>X, рекурсивно додати V в праве піддерево T.

```
BTN* insert(BTN* p, int v)
{ if (!p) return (nwnode(v, NULL, NULL)); //для порожнього
  if (p->dat>v) p->lt = insert(p->lt, v);
  else if (p->dat < v) p->rt = insert(p->rt, v);
  return p;
}
```

Вилучення з дерева бінарного пошуку

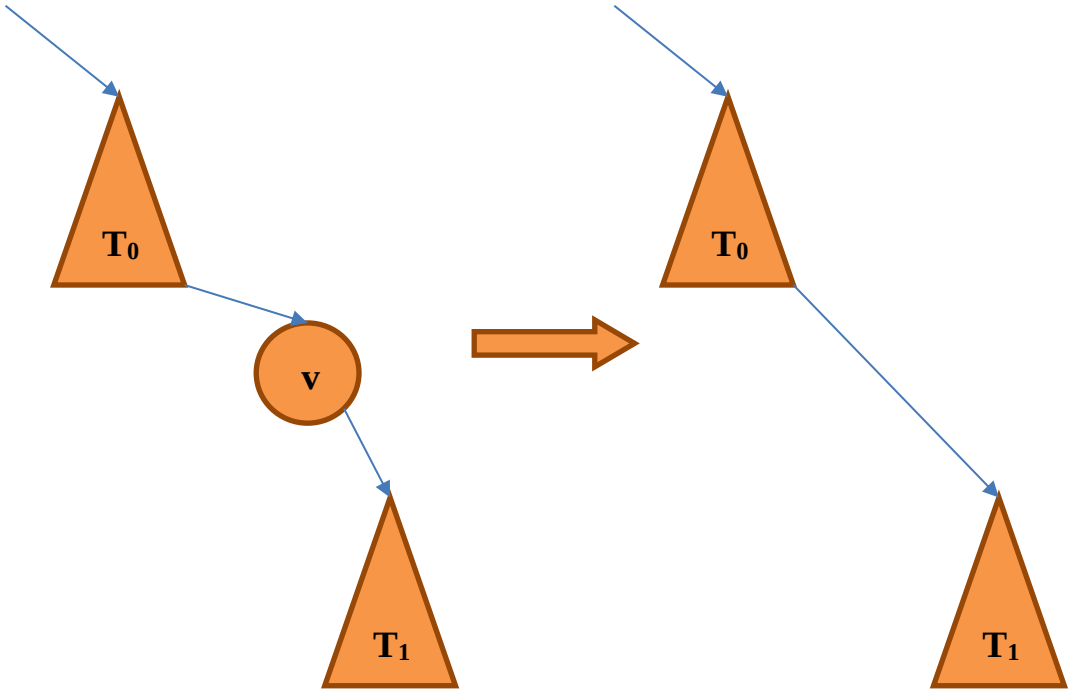


Рисунок 14. Вилучення з дерева бінарного пошуку (випадок одного сина)

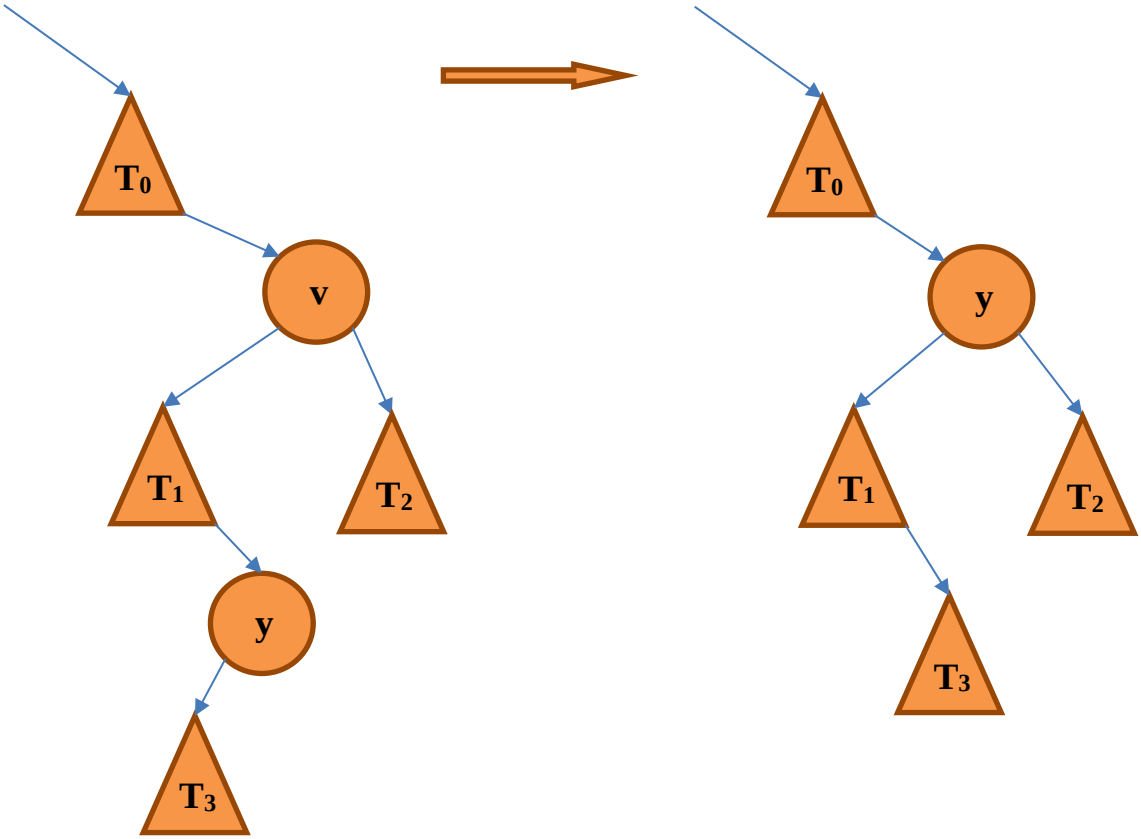


Рисунок 15. Вилучення з дерева бінарного пошуку (загальний випадок)

Вхід: дерево T й ключ V.

Задача: вилучити з дерева T вузол зі значенням V.

Алгоритм: Якщо дерево T порожнє, завершення. Інакше порівняти V зі значенням X кореневого вузла:

- Якщо V>X, рекурсивно вилучити V з правого піддерева T;
- Якщо V<X, рекурсивно вилучити V з лівого піддерева T;
- Якщо V=X, то необхідно розглянути два випадки:
  - Якщо вузол має не більше одного сина його вилучаємо;
  - Якщо вузол має двох синів, то:
    - Знайдемо вузол зі значенням Y, що йому безпосередньо передує у симетричному обході, здійснюємо заміну V←Y й вилучаємо вузол зі значенням Y.
    - Визначимо функцію для пошуку і видалення вузла, який передує вузлу **p** при симетричному обході, з переміщенням його значення у вузол **p**:

```
BTN* delmax(BTN* p, BTN* t)
{  BTN* q;
   if (p->rt) p->rt = delmax(p->rt, t);
    else
       { t->dat = p->dat;
         q = p;
         p->p->lt;
         delete q;
       }
    return p;
}
```

- Скористаємось цією функцією для вилучення вузла з дерева:

```
BTN* ndel(BTN* p, int v)
{  BTN* q;
   if (p)
     { if (p->dat > v) p->lt = ndel(p->lt, v);
       else if (p->dat < v)  p->rt = ndel(p->rt, v);
            else if (p->lt)  p->lt = delmax(p->lt, p);
                 else {
```

```
                        q = p;
                        p = p->rt;
                        delete q;
                       }
    }
    return p;
}
```

Підсумки

Дерева бінарного пошуку виступають у ролі досить привабливої структури даних для представлення різноманітної лінійно впорядкованої інформації, наприклад, таблиць (з ключовими полями, за якими здійснюється впорядкування) й підтримують ефективне виконання основних операцій: пошуку, додавання, вилучення.

Якщо n - кількість вузлів, а h – висота дерева (довжина найдовшого ланцюга від кореня) то:

- **Витрати пам`яті** – O(n).
- **Часова складність** для всіх дій:
  - "*в середньому*" - O(h);
  - "*в найгіршому*" - O(n) (h може дорівнювати n).

Часова складність основних дій з деревами бінарного пошуку визначається висотою дерева.

Не складно, за час O(n) побудувати дерево бінарного пошуку з висотою O(log n) (*ідеальне збалансо*ване *дерево*), що представляє впорядковану сукупність даних. Але дії з деревом: додавання та вилучення можуть порушувати "*збалансованість*", поступово перетворюючи його на дерево з висотою O(n). При цьому структура даних стає схожою на список й втрачає основні свої привабливі риси.

Поради

- Розібратися з розглянутими можливостями обробки інформації, представленої деревами бінарного пошуку.
- Самостійно реалізувати інші дії з інформацією, що представлена деревами бінарного пошуку;
- Не зловживати рекурсією.

Задачі

- Написати функції для визначення найбільшого значення у вузлах непорожнього дерева бінарного пошуку (рекурсивну, не рекурсивну.).
- Написати функцію, що обчислює кількість вершин дерева бінарного пошуку із значеннями меншими за v.
- Написати функцію, що обчислює кількість вершин дерева бінарного пошуку із значеннями більшими за v.
- Написати функцію для перевірки входження значення з інтервалу [v, u] до дерева бінарного пошуку.
- Задана послідовність цілих чисел a_1, a_2, …, a_n . Написати програму модифікації дерева бінарного пошуку B, спочатку порожнього, за умовами:
  - якщо $a_i > 0$, то додати a_i в B;
  - якщо $a_i < 0$, то вилучити $-a_i$ з B (при відсутності видається повідомлення);
  - якщо $a_i = 0$, то закінчити виконання програми.
- Написати не рекурсивні функції для вставки та вилучення елементів.
- Записати у файл відмітки вузлів дерева бінарного пошуку у порядку їх зростання.
- У текстовому файлі PROG – “Сі-програма”, з ідентифікаторами довжиною не більше 9. Надрукувати в алфавітному порядку всі ідентифікатори, вказавши кількість їх входження (великі й маленькі літери розрізняються).

- Розв`язати попередню задачу, але разом з ідентифікатором друкувати у зростаючому порядку номери всіх рядків тексту програми, де він зустрічається.
- Розв`язати попередню задачу, якщо максимальна довжина ідентифікаторів заздалегідь не відома.
- Написати функцію для розбиття дерева двійкового пошуку на два зі значеннями у деревах відповідно $< K$ та $\geq K$.
- Написати функцію для об`єднання двох множин представлених деревами двійкового пошуку. Результатом є дерево двійкового пошуку.
- Написати функцію для об`єднання двох дерев двійкового пошуку зі значеннями у деревах відповідно $<K$ та $\geq K$.
- Як на вашу думку можна виправляти “незбалансованість” дерев двійкового пошуку?

5.4. Збалансовані дерева бінарного пошуку

Дерева бінарного пошуку надають гнучкі та потужні можливості для обробки впорядкованої інформації, але:

- Час пошуку (й інших операцій) визначається висотою дерева. Найменший час пошуку $O(\log_2 n)$ має місце в “повністю збалансованих” деревах.
- Не є проблемою побудувати “повністю збалансоване” дерево (розглянута вище функція build). Але в результаті додавання та вилучення елементів збалансованість може бути втраченою. Дерево може поступово “наблизитись” до лінійного списку (з висотою $O(n)$). Збільшується час на виконання операцій.
- Підтримка “повної збалансованості” інколи вимагає повної перебудови дерева, що означає великі “накладні” витрати часу (Рисунок 16).

Повністю збалансоване дерево бінарного пошуку – для кожної вершини якого кількість вершин у його лівому та правому піддеревах відрізняється не більше ніж на 1. Часова складність алгоритму побудови такого дерева – $O(n)$.

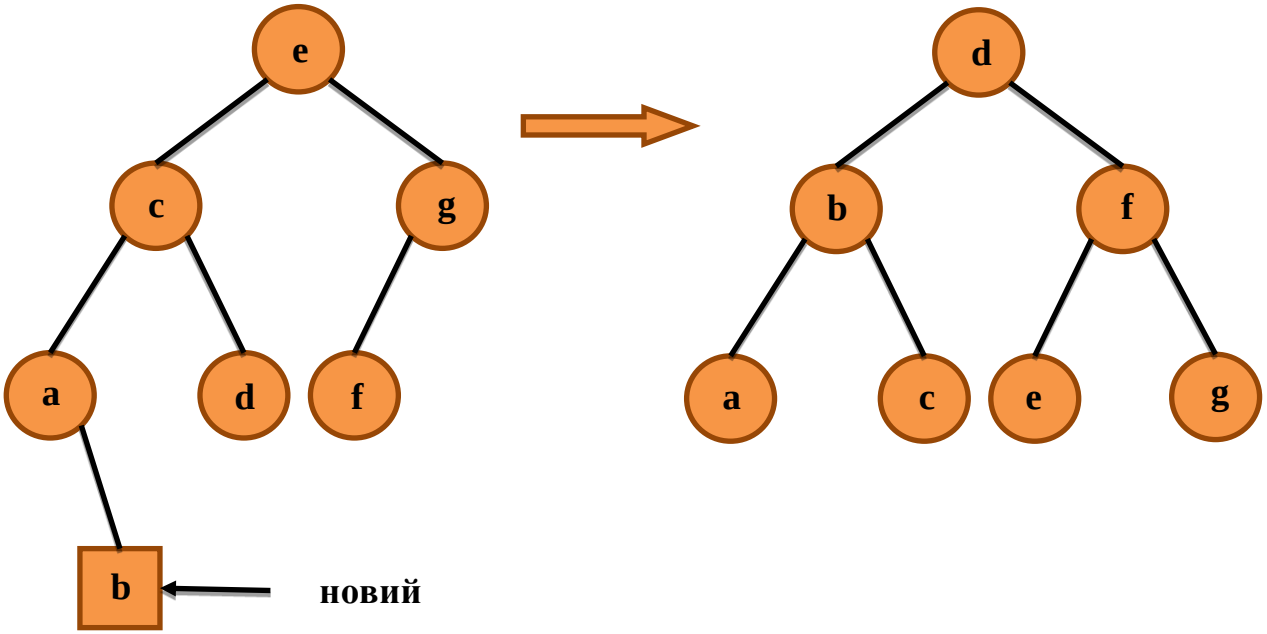


Рисунок 16. Відновлення повної збалансованості після додавання вершини

Наведений приклад демонструє, що змінились майже всі зв’язки між вершинами дерева. Звідки маємо, що відновлення повної збалансованості потребує часу $O(n)$.

Тому в програмуванні використовують дерева, які мають “часткову” збалансованість. Існують різні підходи до реалізації часткової збалансованості: “АВЛ-дерева”, “червоно-чорні дерева”, “2-3 дерева”, та інші. Спільним є те, що висота таких дерев ~ O(log n), що визначає час виконання основних операцій, а збалансованість підтримувати “значно дешевше”, здійснюючи відповідні локальні пере-творення, пройшовши від деякого вузла (де відбулись зміни) шлях не більший ніж до кореня дерева. Розглянемо суттєво різні підходи, що реалізовані в АВЛ (AVL) та 2-3 деревах.

АВЛ-дерева

АВЛ–дерево – для кожної вершини висоти його лівого та правого піддерев відрізняються не більше ніж на 1.

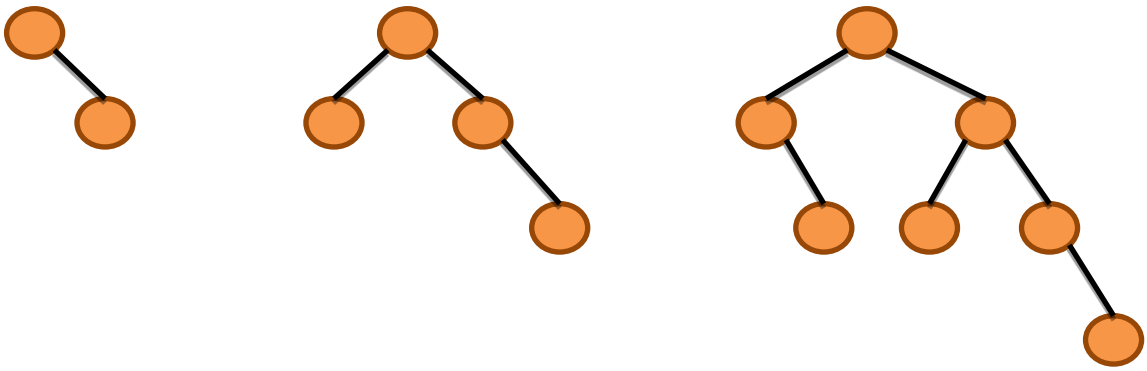


Рисунок 17. Приклади найбільш “асиметричних” АВЛ-дерев

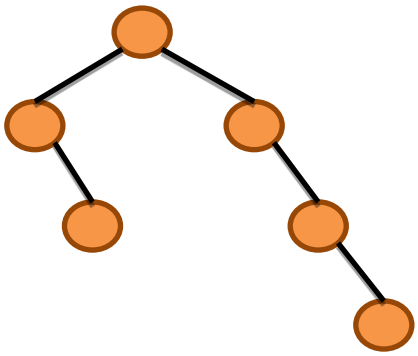


Рисунок 18. Не АВЛ-дерево

Теорема (Адельсон-Вельський, Ландіс):

АВЛ-дерево за висотою ніколи не буде перевищувати висоти повністю (“ідеально”) збалансованого дерева більше ніж на 45%, незалежно від кількості вершин.

Оцінки “середнього випадку” ще більш оптимістичні. “Середній час” пошуку в найбільш асиметричних АВЛ-деревах лише на 4% більше ніж у повність збалансованих. До того ж найбільш асиметричні АВЛ-дерева – рідкість.

Означення типу даних для вузла АВЛ-дерева, що зберігається у стандартній формі, може мати знайомий вигляд:

```
struct BTN
{  int dat, bal;
   BTN *lt, *rt;
};
```

Де поле `bal` – відповідає за різницю висот лівого та правого піддерев (баланс: -1, 0, 1).

Перетворення, що називають “*обертанням*” та “*подвійним обертанням*” лежать в основі відновлення збалансованості.

Обертання АВЛ-дерева

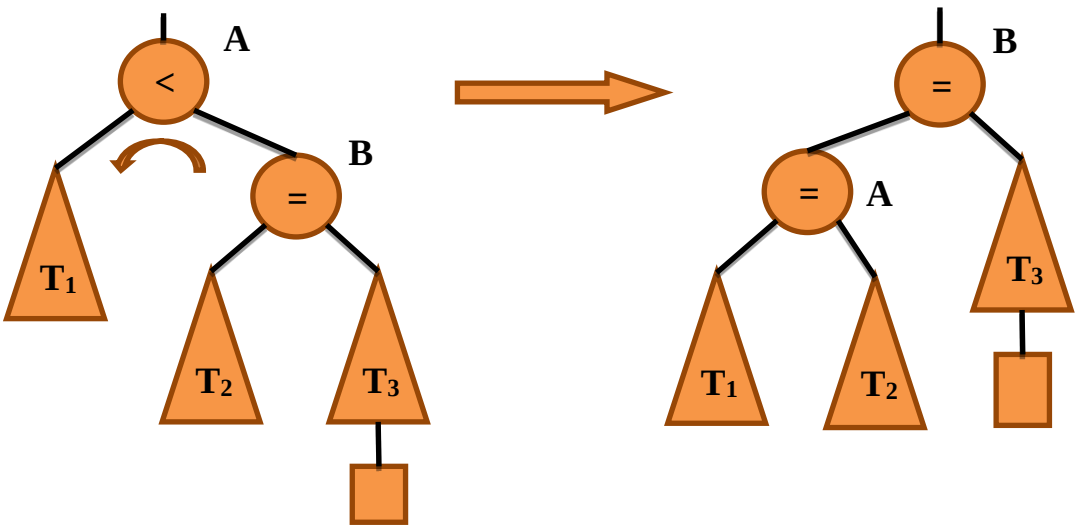


Рисунок 19. Ліве обертання АВЛ-дерева

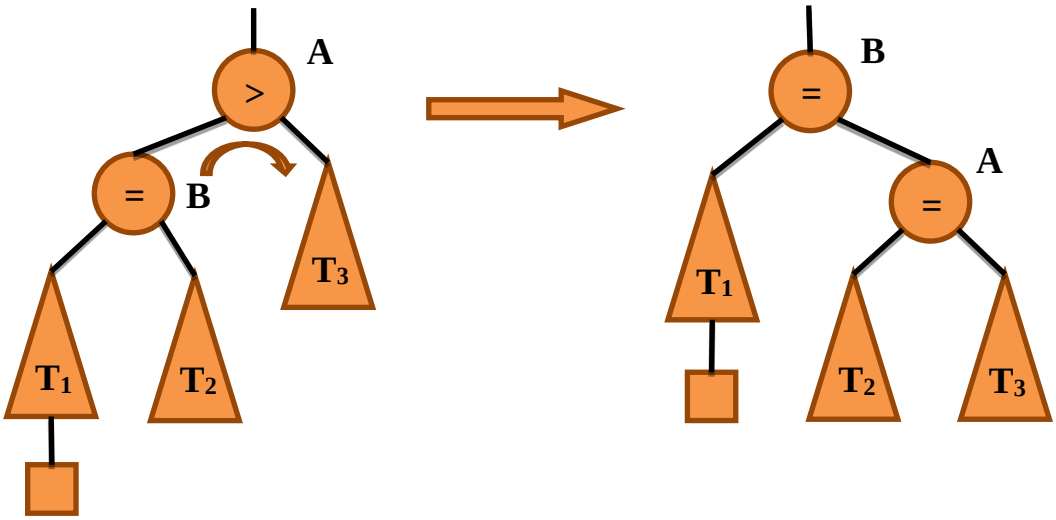


Рисунок 20. Праве обертання АВЛ-дерева

```
void right(BTN** p, int sa, int sb)
{   BTN * a, b;
    a = *p;
    b = a->lt;
    a->lt = b->rt;
    a->bal = sa;
    b->rt = a;
    b->bal = sb;
    *p = b;
}
```

Аналогічно функції для *правого обертання* можна записати функцію, що здійснює *ліве обертання* - `void left(BTN** p  , int sa , int sb)`.

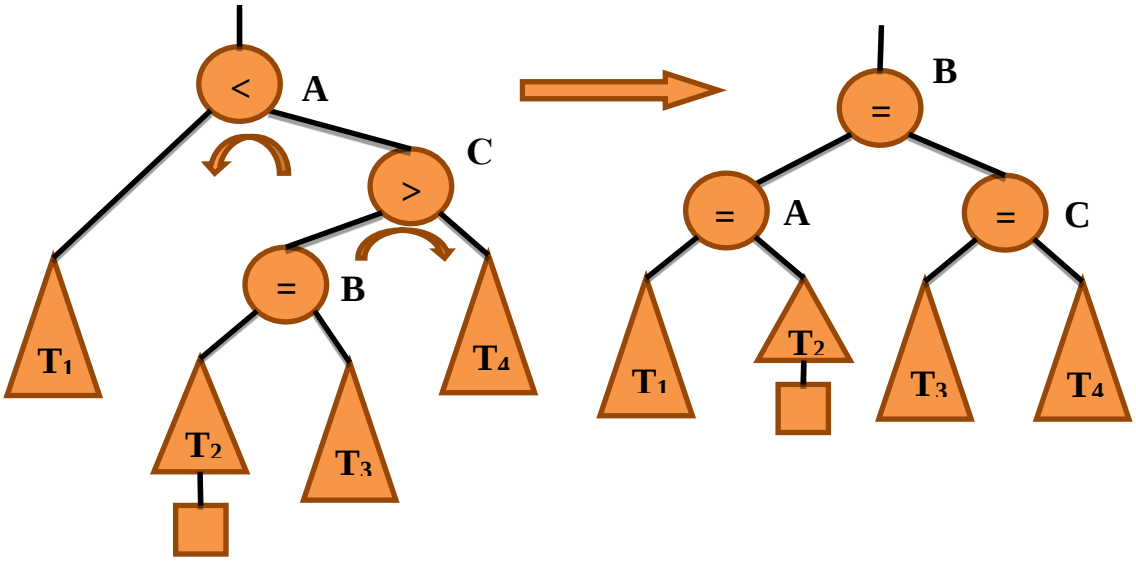


Рисунок 21. Подвійне ліве обертання АВЛ-дерева

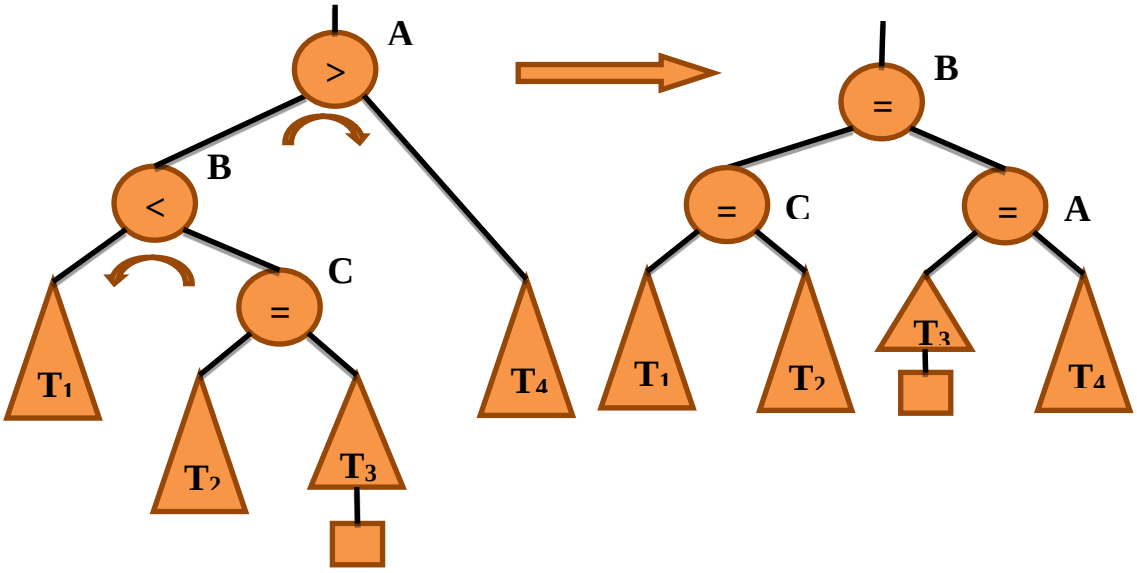


Рисунок 22. Подвійне праве обертання АВЛ-дерева

Функція *подвійного правого обертання.*

```
void dbright (BTN** p, int sa, int sb, int sc)
{   left(&((*p)->lt), sb, sc);
    right(p, sa, sc);
}
```

Аналогічно можна записати функцію, що здійснює *подвійне ліве обертання.*

Алгоритм додавання до АВЛ-дерева

Після додавання вершини у дерево може порушуватись збалансованість і потрібно її відновити, тому відбувається рух у зворотному напрямку від нового вузла до кореня. Дії визначаються значенням балансу вершини (bal) та можливо двох останніх кроків. Потрібні дії можна сформулювати у вигляді правил:

- Баланс вершини «=» , змінюється на:
  - «>», якщо останній крок був з лівого піддерева;
  - «<», якщо останній крок був з правого піддерева;

Рух до кореня продовжується;

- Баланс вершини «>», або «<» й останній крок був з більш короткого піддерева, змінюється на «=». Рух завершується.
- Баланс вершини «>», або «<» й останній крок був з більш високого піддерева:
  - Два останніх кроки були в одному напрямку – виконується відповідне обертання;
  - Два останніх кроки були в протилежних напрямках - виконується відповідне подвійне обертання;

Рух завершується.

Основна частина з реалізації алгоритму додавання до АВЛ-дерева може мати вигляд:

```
//додавання вузла в АВЛ-дерево
//формування нового вузла збалансованого дерева
BTN* newbl(v)
{ BTN * t;
  t = new BTN;
  t->lt = t->rt = NULL;
  t->bal = 0; t->dat = v;
  return t;
}
//додавання вузла
//повертає ознаку того, що висота збільшилась
bool binclude(BTN** p, int v)
```

```
    { BTN* t;
      if (!(t = *p)) {*p = newbl(v); return 1;}
      else if (t->dat == v) return 0;
           else return (t->dat > v ?
                       (binclude(&(t->lt), v) ? lbal(p) : 0) :
                       (binclude(&(t->rt), v) ? rbal(p) : 0));
    }
    //ліве перебалансування при додаванні
    //повертає ознаку того, що висота збільшилась
    bool lbal(BTN** p) {
    { BTN* t;
      switch((t = *p)->bal)
      { case 1:  t->bal = 0; return 0;
        case 0:  t->bal = -1; return 1;
        case -1: if ((t = t->lt)->bal == -1) right(p, 0, 0);
             else if ((t->rt)->bal == -1) dbright(p, 1, 0, 0);
                    else dbright(p, 0, -1, 0);
             return 0;
      }
    }
    //аналогічний вигляд має функція rbal(p) для правого
//перебалансування
```

Алгоритм вилучення з АВЛ-дерева

Вилучення з АВЛ-дерева відбувається так саме, як розглянуте раніше вилучення з дерева бінарного пошуку (функція `BTN* ndel(BTN* p, int v)`, Рисунок 15). Але при вилученні вузла з дерева збалансованість може порушуватись. Для відновлення збалансованості потрібно рухатись у напрямку до кореня й враховувати можливі зменшення висоти. Дії (у вигляді відповідних обертань) залежать від значення балансу (bal), напрямку останнього кроку й можливо від балансу (bal) сина вершини. При відновленні висоти піддерева подальший рух завершується.

Основна частина з реалізації алгоритму вилучення з АВЛ-дерева може мати вигляд:

```
    //вилучення вузла з АВЛ-дерева
    //повертає ознаку того, що висота зменшилась
    bool bdelete(BTN** p, int v)
    { BTN* t;
      if (!(t = *p)) return 0;
      else if (t->dat > v)
              return(bdelete(&(t->lt), v) ? lbald(p) : 0);
           else if (t->dat < v)
                  return(bdelete(&(t->rt), v) ? rbald(p) : 0);
                else if (t->lt)
                  return(bdelmax(&(t->lt), t) ? lbald(p) : 0);
                     else {*p = t->rt; delete t; return 1;}
    }
    //вилучення  вузла  з  максимальним  значенням  після
//пересилання його значення у вузол з покажчиком t
    //повертає ознаку того, що висота зменшилась
    bool bdelmax(BTN** p, BTN* t)
    {  BTN* q;
      if ((q = *p)->rt)
        return(bdelmax(&(q->rt), t) ? rbald(p) : 0);
      else {t->dat = q->dat; *p = q->lt; delete q; return 1;}
    }
    //ліве перебалансування при вилученні
    //повертає ознаку того, що висота зменшилась
    bool lbald(BTN** p)
    { BTN* t;
      switch((t = *p)->bal)
      { case -1: t->bal = 0; return 1;
        case 0:  t->bal = 1; return 0;
        case 1:  switch((t = t->rt)->bal)
               { case -1:
                   switch((t->lt)->bal)
                   { case -1: dbleft(p, 0, 1, 0);  return 1;
                     case 0:  dbleft(p, 0, 0, 0);  return 1;
                     case 1:  dbleft(p, -1, 0, 0); return 1;
                   }
                 case 0: left(p, 1, -1); return 0;
                 case 1: left(p, 0, 0);  return 1;
```

```
                }
            }
        }
        //аналогічний вигляд має функція rbald(p) для правого
//перебалансування
```

2-3 дерева

2-3 деревом називається дерево, в якому кожна вершина, що не є листом, має двох, або трьох синів, а довжини всіх шляхів від кореня до листів однакові.

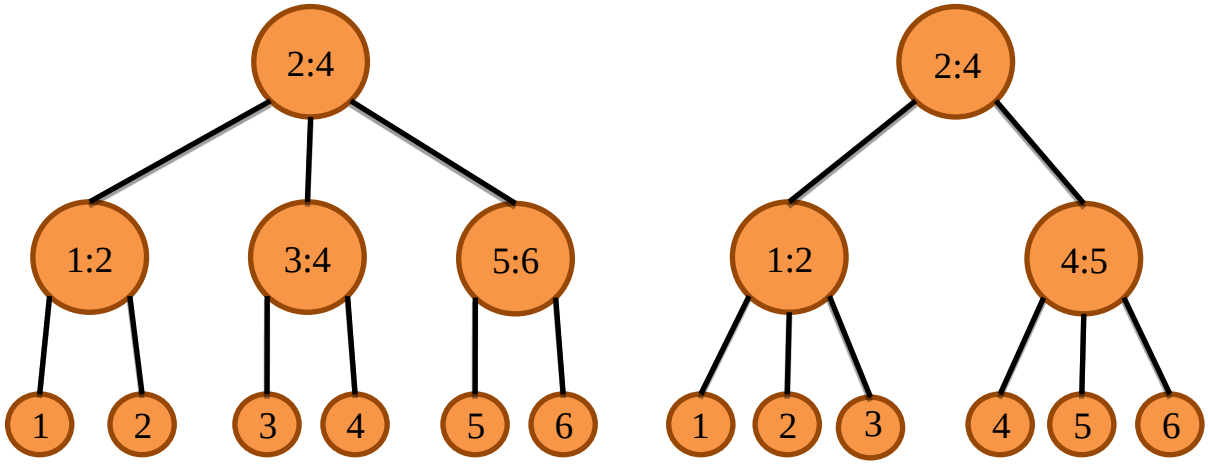


Рисунок 23. Приклади 2-3 дерев

Теорема.

Нехай T є 2-3 деревом висоти h. Кількість вершин дерева T обмежена 2^{h+1} -1 та $(3^{h+1}-1)/2$, а кількість листів 2^h та 3^h.

Для розміщення лінійно впорядкованої множини S використовують лише листя дерева, а “внутрішні” вершини v відмічені парою - L[v]:M[v], де L[v] – найбільший елемент S у піддереві коренем якого є лівий син v, M[v] - найбільший елемент S у піддереві коренем якого є другий син v. Час пошуку у такому дереві – O(log n). Наведені на рисунку 23 приклади свідчать, що лінійно впорядкована множина (S={1, 2, 3, 4, 5, 6}) може бути представлена різними 2-3 деревами.

Наявність вказаних відміток у внутрішніх вузлах 2-3 дерева забезпечує прості й очевидні можливості для операції пошуку. Дії, пов’язані з додаванням та

вилученням значень, що може потребувати відновлення збалансованості дерева спробуємо поясними на прикладах, схематично наведені на рисунках 24–26.

Приклади роботи алгоритмів додавання та вилучення для 2-3 дерева

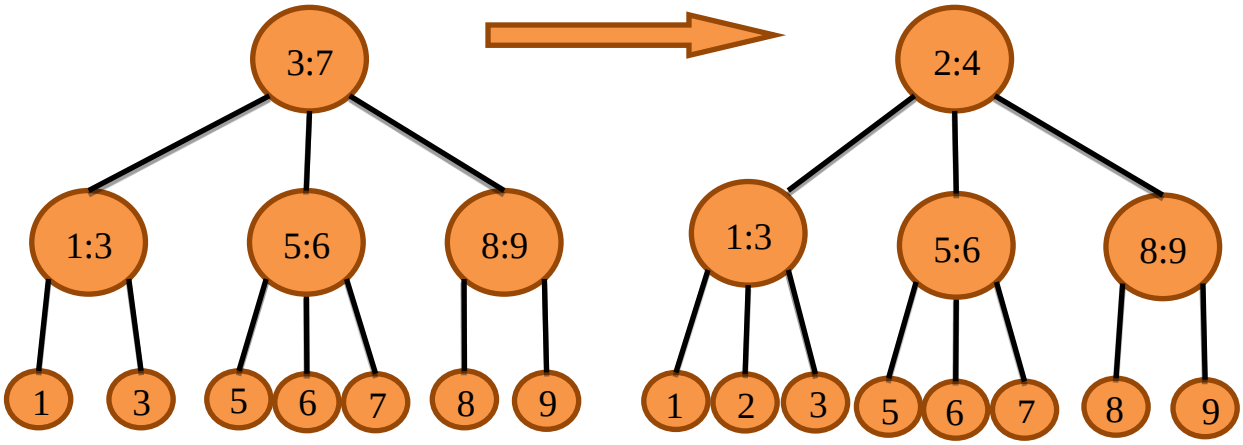


Рисунок 24. Додавання у 2-3 дерево елемента 2

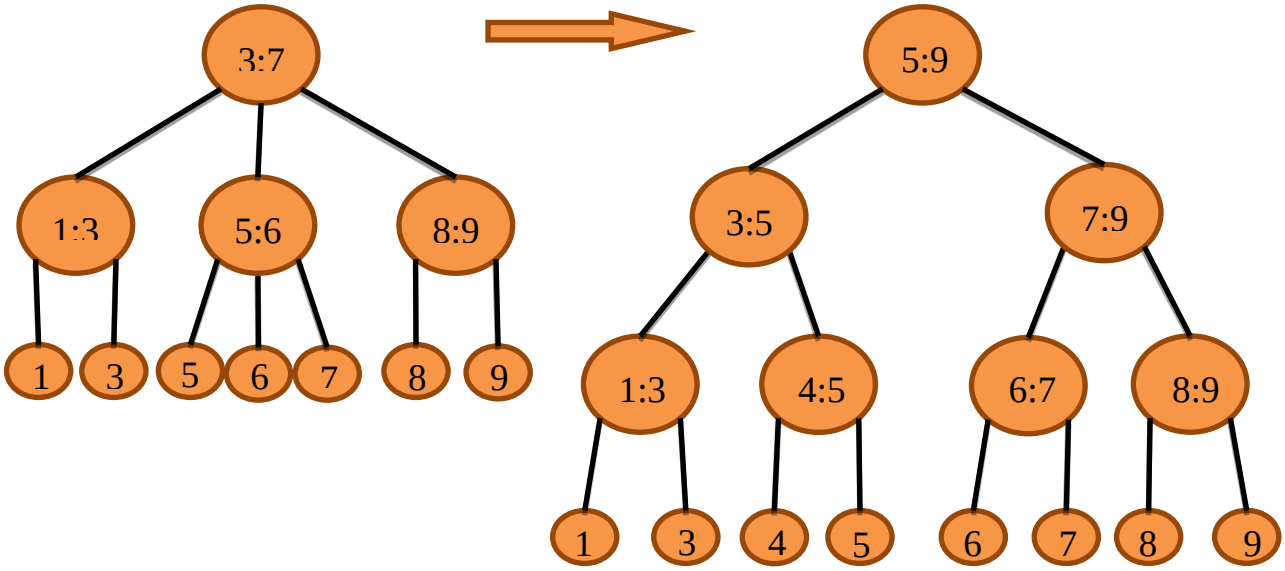


Рисунок 25. Додавання у 2-3 дерево елемента 4

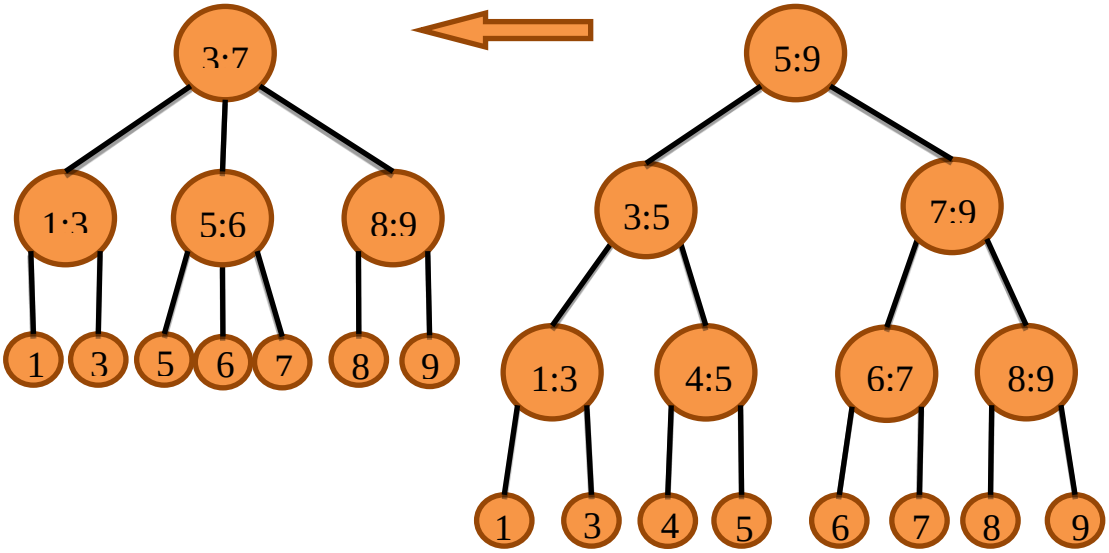


Рисунок 26. Вилучення з 2-3 дерева елемента 4

Підсумки

З`ясувавши суттєві переваги використання саме збалансованих дерев бінарного пошуку для роботи з лінійно впорядкованими даними, а також принципову складність підтримки *повної збалансованості*, при розгляді підходів до *часткової збалансованості* дерев бінарного пошуку, обмежились двома “класичними” й одночасно суттєво різними підходами, які здатні забезпечувати ефективність обробки інформації.

- Існують інші схеми *часткової збалансованості* дерев бінарного пошуку.
- Відомі підходи до *часткової збалансованості* дерев бінарного пошуку дозволяють підтримувати висоту відповідних дерев на рівні ~O(log n), для даних розміру n, й забезпечувати потрібну ефективність виконання основних дій, пов’язаних з пошуком, додаванням та вилученням даних.

Поради

- Розібратися з розглянутими можливостями обробки інформації, представленої збалансованими деревами бінарного пошуку.
- Самостійно розглянути інші підходи до забезпечення збалансованості дерев бінарного пошуку.

- Самостійно розв`язати запропоновані задачі.

Задачі

- Написати функцію для *лівого обертання*.
- Написати функцію для *подвійного лівого обертання*.
- Написати функцію перевірки, чи є бінарне дерево АВЛ-деревом.
- Реалізувати алгоритм включення у АВЛ-дерево.
- Реалізувати алгоритм вилучення з АВЛ-дерева.
- Довести, що алгоритм вилучення з АВЛ-дерева може вимагати [h/2] обертань та подвійних обертань, але не більше (h – висота дерева).
- Якщо в означенні збалансованого за висотою дерева допустити щоб максимальна різниця між висотами двох піддерев дорівнювала 2, побудувати алгоритми відновлення збалансованості, які гарантують логарифмічний час пошуку. Навести найбільш асиметричні збалансовані дерева в цьому випадку. Який час пошуку в “найгіршому” та “середньому” в найбільш асиметричних деревах?
- Записати функцію для пошуку в 2-3 дереві.
- Записати алгоритм вставки в 2-3 дерево. Реалізувати алгоритм відповідною функцією.
- Записати алгоритм вилучення з 2-3 дерева. Реалізувати алгоритм відповідною функцією.
- Записати алгоритм побудови 2-3 дерева за впорядкованою множиною. Реалізувати алгоритм відповідною функцією.

5.5. Дерева і вирази

Крім розглянутих вище застосувань дерев, як потужних й гнучких засобів для обробки різноманітної інформації, варто звернути увагу на природні зв'язки між деревами та математичними виразами (формулами). Вирази – дозволяють визначати зв'язки між вхідними й результуючими даними. Розглядали вище різні способи запису виразів, можливості обчислення значення виразу, при заданих значеннях змінних. Практично важливою є також задача здійснення *"аналітичних обчислень виразів"*, де результатом обчислень є відповідний вираз. Виникають питання адекватного внутрішнього представлення виразів, для зручної роботи з ними. Також продемонструємо технологічні підходи до побудови програмних рішень задач, що виникають при роботі з виразами, й основані на відповідних формалізаціях задач.

Подання виразів деревами

Розглянемо можливості використання дерева для подання виразу. В представленні арифметичного виразу, що складається з цілих чисел, змінних, операцій, у бінарному дереві знак останньої (в порядку виконання) операції розміщується в корені дерева, перший операнд цієї операції представлений лівим піддеревом, другий - правим. Всі числа та змінні виразу представлені листками, а операції - внутрішніми вузлами.

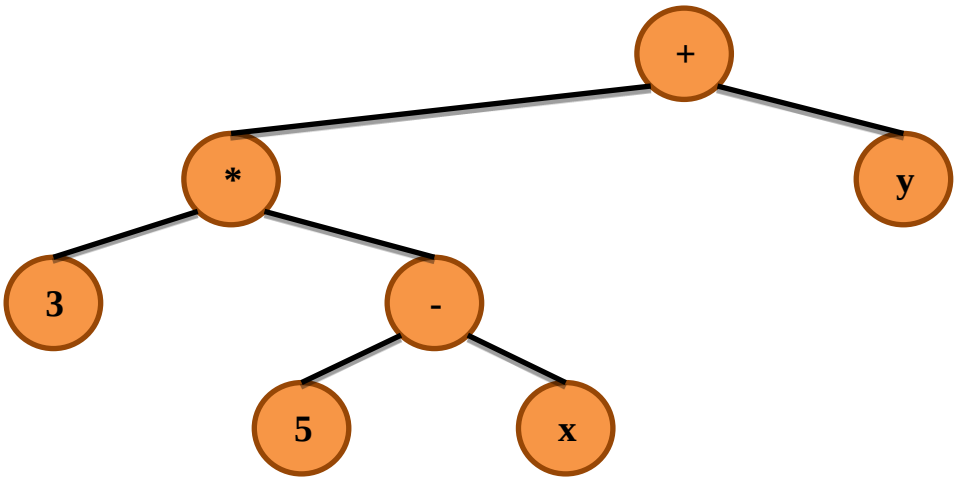


Рисунок 27. Подання виразу 3(5 – x) + y за допомогою дерева*

Прямий та обернений порядки обходу відповідають *прямому* та *інверсному польським записам виразу*: “+ * 3 - 5 x y”; “3 5 x - * y +”.

Проходження дерева в симетричному порядку відповідає *повному дужковому запису*: ((3 * (5 - x)) + y), або *частковому дужковому запису*, якщо враховані властивості операцій: 3 * (5 - x) + y.

Дерево будемо задавати *стандартною формою*:

```
struct BTN {int dat;
                    BTN *lt, *rt;
                    };
```

Обробка виразів, представлених деревами

Розглянемо, як приклад, арифметичні вирази, що містять тільки невід`ємні цілі числа, знаки операцій {+, -, *, /}, змінні, позначені x, y, z та дужки (). Припускаємо, що пропуски та символи нового рядка можливі в будь якому місці запису виразу, представленому *повним дужковим записом*. Бінарне дерево будемо зберігати у *стандартній формі,* використовуючи розглянуті раніше типи даних. Вважаємо, що у дереві знаки операцій та змінні у вузлах задані кодами своїх символів, а числа зберігаються з оберненими знаками. Це означає, що у вузлах все розмаїття відміток кодується цілими числами: додатне значення – код операції, або змінної; від’ємне – код числа. Звісно, що це не єдиний можливий спосіб для кодування відміток вузлів.

Основні підзадачі:

- введення виразів (використовуємо *повний дужковий запис*);
- виведення виразу (*повний дужковий запис*);
- обробка виразу:
  - аналітичне диференціювання виразів;
  - спрощення виразів.

Введення виразів

Виходячи з того, що програмування є поєднанням мистецтва та технологій, звернемо увагу на технології. Для формалізації задачі введення виразів згадаємо, що повний дужковий запис виразу є рекурсивною структурою, синтаксис якої можна подати БНФ (від англ. *Backus-Naur form* – спосіб запису правил формальною мовою):

```
<вираз>::= <число>|<змінна>|<(><вираз><операція><вираз><)>
<число> ::= {<цифра>}*
<цифра> ::= 0|1|2|3|4|5|6|7|8|9
<змінна> ::= x|y|z
<операція> ::= +|-|*|/
```

Це і визначає організацію функцій для розпізнавання. Для спрощення скористаємось у функціях зовнішньою змінною:

```
int next=' ';
```

Для основної функції введення виразу, що заданий у повному дужковому записі, спочатку сформуємо службові функції для зчитування символів виразу, отримання з символів числа, формування вузла дерева, розпізнавання виразу записаного, згідно наведених правил БНФ:

```
//введення чергового символу в зовнішню змінну next
void nsym()
{  if (next == EOF)
```

```
        return;
    while ((next = getchar()) == ' ' || next == '\n');
}
//введення числового операнда виразу
int numb()
{   int v;
    if (next <'0' || next > '9')
    {   cout << "ERROR " << (char) next << endl;
        return -1;
    }
    for (v=0; next >= '0' && next <= '9'; nsym())
        v = v * 10 + (next - '0');
    return v;
}
//формування вузла бінарного дерева
BTN* nwnode(int v, BTN* pl, BTN* pr)
{ BTN* p = new BTN;
  p->dat = v; p->lt = pl; p->rt = pr;
  return p;
}
//розпізнавання виразу
BTN* expr()
{   BTN* p;
    char nx;
    if (next == '(')
    {   nsym();
        p = expr();
        if (strchr ("+-*/", next))
        {   nx = next;
            nsym();
            p = nwnode(nx, p, expr());
        }
        if (next != ')')
        {  cout << "ERROR " << (char) next << endl;
           return NULL;
        }
        nsym();
        return p;
```

```
        }
        if(strchr("xyz", next))
        {   nx = next;
            nsym();
            return nwnode(nx, NULL, NULL);
        }
        else  return nwnode(-numb(), NULL, NULL);
    }
    //введення виразу в повному дужковому записі
    BTN* inpexp()
    {   nsym();
        return expr();
    }
```

Використання БНФ, для формалізації зовнішнього запису виразів, дозволяє будувати програмне рішення використовуючи суто технологічний підхід, де правилам БНФ відповідають програмні функції, а зв'язкам між правилами – виклики відповідних функцій. При цьому кожна функція, що представляє відповідне правило, є досить простою та прозорою й природнім чином будується за правилом. Прозорість й структурованість програмного рішення надає хороші можливості для подальшого супроводження та розвитку.

Виведення виразів

Для виведення виразу у повному дужковому записі з його внутрішнього подання деревом, використовується обхід дерева у *симетричному порядку*.

```
    //виведення виразу в повному дужковому записі
    void prnexp(BTN* p)
    {   int v;
        v = p->dat;
        if (v <= 0)  cout << -v;
        else if (strchr("+-*/", v))
        {   cout << '('; prnexp(p->lt);
            cout << (char) v;
```

```
        prnexp(p->rt);
        cout << ')';
    }
    else  cout << (char) v;
}
```

Аналітичне диференціювання виразів

Аналітичне диференціювання виразів (за деякою змінною, наприклад - x) здійснюється згідно відомих правил:

- x` = 1; a = 0; (a – константа, або інша змінна)
- (u + v)` = u` + v`;
- (u – v)` = u` – v`;
- (u * v)` = u` * v + v` * u;
- (u / v)` = (u` * v – u * v`) / (v * v)

Наведені формули й визначають організацію функції для диференціювання.

```
//диферецювання виразу за змінною var
BTN* diff(BTN* p, int var)
{ if (p->dat == var)
        return nwnode(-1, NULL, NULL);
  switch (p->dat)
  {  case '+':
         return nwnode('+', diff(p->lt, var),
                            diff(p->rt, var));
    case '-':
        return nwnode('-', diff(p->lt, var),
                           diff(p->rt, var));
    case '*':
        return nwnode('+',
                      nwnode('*',diff(p->lt,var),
                             copytr(p->rt)),
                      nwnode('*',copytr(p->lt),
                             diff(p->rt, var)));
    case '/':
        return nwnode('/',
```

```
                        nwnode('-',
                               nwnode('*', diff(p->lt, var),
                                      copytr(p->rt)),
                               nwnode('*',copytr(p->lt),
                                      diff(p->rt, var))),
                       nwnode('*', copytr(p->rt),
                              copytr(p->rt)));
    default:
        return nwnode(0, NULL, NULL);
    }
}
```

Копіювання дерева-виразу можна реалізувати наступною функцією.

```
//створення копії бінарного дерева
BTN* copytr(BTN* p)
{ if (p)
    return nwnode(p->dat, copytr(p->lt),
                          copytr(p->rt));
  else
    return NULL;
}
```

При диференціюванні отримуємо не зовсім звичну форму для результату.

Наприклад:

((3 * (x + 1)) – (z / x))` = (0 * (x + 1) + 3 * (1 + 0)) – (0 * x – z * 1) / (x * x)

Причина – відсутні спрощення при виконанні диференціювання. Варто реалізувати спрощення отриманого дерева-виразу.

Спрощення виразів

Для реалізації спрощення виразів скористаємось наступними правилами:

- u + 0 = u; 0 + u = u;
- u – 0 = u;
- u * 0 = 0; 0 * u = 0;
- u * 1 = u; 1 * u = u;

- 0 / u = 0;
- u / 1 = u;

Наведені формули й визначають організацію функції для спрощення. Окремою функцією реалізовано вилучення дерева.

```
//вилучення бінарного дерева
void deltr(BTN* p)
{  if (!p) return;
    deltr(p->lt);
    deltr(p->rt);
    delete p;
}
//спрощення виразу
BTN* simpl(BTN* p)
{  BTN* pl, pr;
   if (!p) return p;
   if ((p->dat<0) ||(strchr("xyz", p->dat)))
      return p;
   pl = p->lt = simpl(p->lt);
   pr = p->rt = simpl(p->rt);
   switch(p->dat)
    { case '+':
        if (pl->dat == 0)
        {  delete pl;
           delete p;
           return pr;
        }
        if (pr->dat == 0)
        {   delete pr;
            delete p;
            return pl;
      }
       return p;
    case '-':
        if (pr->dat == 0)
        {   delete pr;
            delete p;
            return pl;
        }
```

```
            return p;
        case '*':
            if ((pl->dat == 0) || (pr->dat == 0))
            {   p->dat = 0;
                p->lt = p->rt = NULL;
                deltr(pl);
                deltr(pr);
                return p;
            }
            if (pl->dat == -1)
            {   delete pl;
                delete p;
                return pr;
            }
            if (pr->dat == -1)
            {   delete pr;
                delete p;
                return pl;
            }
            return p;
        case '/':
            if ((pl->dat == 0) || (pr->dat == -1))
            {   deltr(pr);
                delete p;
                return pl;
            }
            return p;
        }
    }
```

Якщо після розглянутого вище диференціювання виразу застосувати спрощення, отриманий результат буде мати більш традиційний вираз.

Наприклад:

((3 * (x + 1)) – (z / x))` = (3 – (0 - z) / (x * x))

Звісно, що розмаїття способів представлення, навіть для арифметичних виразів, не зводиться до розглянутих. Наприклад, для арифметичного виразу, що містить невід`ємні цілі числа, змінні{x, y, z} та операції {+, -, *, /} й представлений

бінарним деревом у стандартні формі, можна використати й інше кодування операндів та операцій, наприклад:

- цілі числа представляти своїми значеннями;
- операції - кодами: ‘+’ -1, ‘-’ -2, ‘*’ -3, ‘/’ -4 ;
- змінні - кодами: x -5, y -6, z -7 .

Не є принциповим обрані обмеження – лише до трьох змінних у виразі: x, y, z (легко розширюється на довільні кількість та імена змінних довжини 1). Можливості спрощення дерев-виразів нескладно також суттєво розширити, наприклад, за рахунок здійснення обчислення константних виразів. Можна також узагальнити задачу аналізу помилкових ситуацій при введенні виразів.

Підсумки

- Використання дерев для представлення формул виявилось зручним для виконання нетривіальної роботи з формулами, здійснення аналітичних перетворень.
- Застосування відповідної формалізації, для розглянутих задач, дозволило скористатись технологічними підходами й отримати добре структуровані програмні рішення придатні для подальшого вдосконалення й супроводження.
- Внаслідок рекурсивної природи даних, досить прозорими й привабливими виглядають саме рекурсивні алгоритми та їх програмна реалізація.

Поради

- Розібратися з розглянутими, а також іншими можливостями подання та обробки виразів..
- Самостійно розв`язати запропоновані задачі.

Задачі

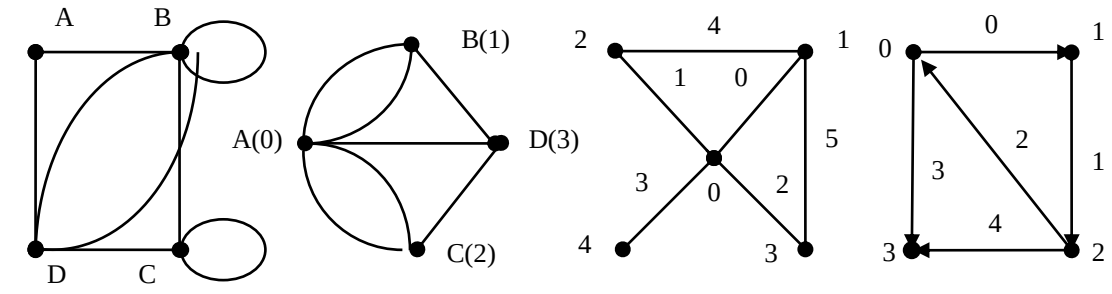
- Написати функцію для друкування виразу, поданого бінарним деревом, що містить цілі числа, змінні x, y, z та знаки операцій +, -, *, / з мінімальною кількістю дужок. Вважати, що зовнішні дужки не друкуються, операції однакового старшинства виконуються зліва направо.
- Написати функцію для обчислення значення виразу , поданого бінарним деревом, що містить цілі числа, змінні x, y, z та знаки операцій +, -, *, /, при заданих значеннях змінних.
- Написати функцію для спрощення арифметичного виразу, зображеного бінарним деревом. Передбачити можливість обчислення константних виразів й врахування обчислених значень при спрощенні.
- Розглянути питання представлення та обробки логічних виразів. Записати функції для роботи з формулами, що містять логічні константи, змінні, операції: кон'юнкції, диз'юнкції, заперечення, суми за модулем два.
- Написати функцію для перевірки, чи представляє дерево «правильний арифметичний вираз».
- Написати функцію для перевірки, чи представляє дерево «правильний логічний вираз».
- Написати функцію для спрощення арифметичного виразу, зображеного бінарним деревом, використовуючи тотожності:
    - $((f_1+f_2)*f_3) = (f_1*f_3 + f_2*f_3)$;
    - $((f_1-f_2)*f_3) = (f_1*f_3 - f_2*f_3)$;
    - $(f_1*(f_2+f_3)) = (f_1*f_2 + f_1*f_3)$;
    - $(f_1*(f_2-f_3)) = (f_1*f_2 - f_1*f_3)$.
- Написати функцію для спрощення арифметичного виразу, зображеного бінарним деревом, використовуючи тотожності “обернені” до запропонованих у попередній задачі.

6. Графи

6.1. Способи їх представлення та обробка.

Основні поняття графа

Неформально граф $G = (V, E)$ складається зі скінченої множини вершин $V = \{v_1, v_2, \dots, v_n\}$ та скінченої множини ребер (дуг) $E = \{e_1, e_2, \dots, e_m\}$ [7,8]. Ребро визначається парою вершин $e_k = (v_i, v_j)$.



а) псевдомультиграф б) мультиграф в) граф г) орграф

Рисунок 28. Різновиди графів

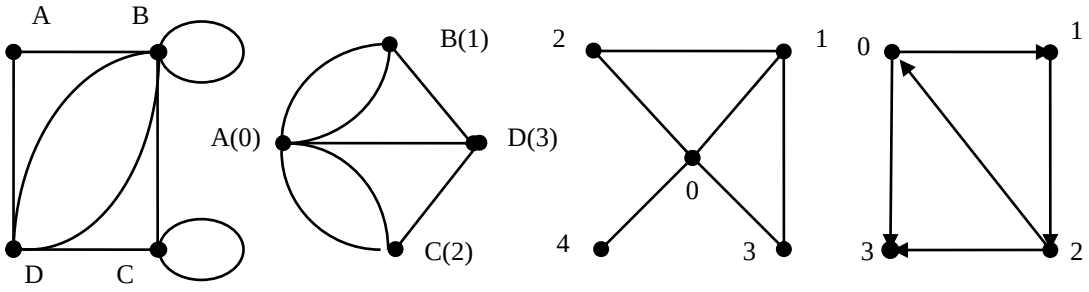
Способи подання графа

- матриця суміжності
- матриця інцидентності
- список суміжності
- структура суміжності

Матриця суміжності

Для $G = (V, E)$, де $|V| = n$, $|E| = m$ – ***матриця суміжності*** A розмірами $n \times n$ задається таким чином: $a_{ij} = 1$, якщо вершини v_i та v_j суміжні, інакше $a_{ij} = 0$ (зокрема, для звичайного графа $a_{ii} = 0$). Для мультиграфів a_{ij} — це число ребер, які з'єднують вершини v_i та v_j. Для орієнтованих графів $a_{ij} = 1$, якщо дуга має

початок у v_i та кінець у v_j, інакше $a_{ij} = 0$. Матриця суміжності задає відношення суміжності R на множині V, $R \subseteq V \times V$.



а) псевдомультиграф б) мультиграф в) граф г) орграф

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 2 & 2 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Рисунок 29. Приклади різновидів графів та їх матриці суміжності

Матриця інцидентності

Матриця інцидентності B розмірами $n \times m$, що відповідає (n, m)-графу G: $b_{kl} = 1$, якщо вершина v_k і ребро e_l інцидентні, $b_{kl} = 0$ інакше. Для орграфів: $b_{kl} = 1$, якщо вершина v_k є початком дуги e_l; $b_{kl} = -1$, якщо v_k — кінець e_l; $b_{kl} = 0$, якщо v_k і e_l не інцидентні. Матриця інцидентності задає в*ідношення інцидентності* $R \subseteq V \times E$.

Графи на рис. 36 в) та г) подаються матрицями інцидентності $\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ та $\begin{pmatrix} 1 & 0 & -1 & 1 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix}$ відповідно.

Список суміжності

Список суміжності графа G — це множина $S = \{(v_i, v_j), \ldots, (v_l, v_k)\}$, утворена парами вершин, суміжних у графі. Наприклад, граф на рис. 36 в) подається списком суміжності {(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3)}.

Для мультиграфів замість пар потрібні трійки, щоб відрізняти ребра (дуги) між тими самими суміжними вершинами. Мультиграфу на рис. 36 відповідає список суміжності {(0, 1, 0), (0, 1, 1), (0, 2, 0), (0, 2, 1), (0, 3, 0), (1, 3, 0), (2, 3, 0)}.

Структура суміжності

Структура суміжності для графа з *n* вершинами утворюється списком з *n* елементів, які відповідають вершинам. Кожен елемент списку є списком з номерів вершин, суміжних із “поточною” вершиною. Кожну дугу орграфа подано в цій структурі один раз, кожне ребро — двічі.

Зазвичай структуру суміжності представляють масивом з n покажчиків на списки у зв`язному зберігання, що містять номери всіх суміжних вершин. Наприклад, графам на рис. 36 *в), г)* відповідають наступні структури суміжності:

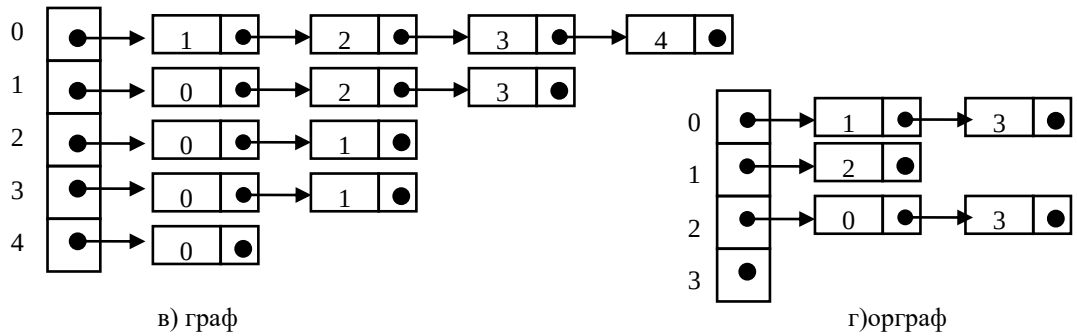


Рисунок 30. Структури суміжності графів в) та г)

Порівняння матричних та спискових способів подання графів

Особливості матричного подання:

- легко програмується;
- матриця суміжності ефективна з погляду витрат пам’яті для графів близьких до повних;
- застосовується лише при відносно невеликій кількості вершин.

Особливості спискового способу подання:

- перевірка суміжності двох заданих вершин, або знаходження ребра, що з’єднує дві вершини, вимагає перегляду всього списку вершин, суміжних, як мінімум, з однією з них.
- Немає необхідності у створенні копії графа при додаванні/видаленні вершини.
- Списковий спосіб подання ефективний з погляду витрат пам’яті для розріджених графів.

Поради щодо подання графів

- Перед розв’язанням задач обробки графів необхідно перейти до найбільш зручного подання, а після розв’язання — конвертувати граф у початкове подання.
- Спосіб подання графа, повинен враховувати специфіку задачі й алгоритмів її розв’язання.
- Перехід між різними способами представлення графів простий і відбувається за $O(n^2)$ операцій.

Приклади задач на подання графів

Реалізувати функцію, яка за структурою суміжності графа будує матрицю суміжності.

Визначимо типи даних, що використовуватимуться в задачах:

```
const int MaxN=16;
struct Node
{
int nn;
  Node *next;
};
struct StrAdj
{
int n;
  Node* e[MaxN];
};
bool a[MaxN][MaxN];
```

Тепер власне опишемо функцію, яка за заданою структурою суміжності будує матрицю суміжності

```
void AdjMtr(StrAdj s, bool A[MaxN][MaxN], int *nn)
{
Node* p;
  int i, j, k;
  *nn = k = s.n;
  for (i=0; i<MaxN; i++)
     {
       for (j=0; j<MaxN; j++)
    {
      A[i][j] = 0;
     }
   }
  for (i=0; i<k; i++)
  {
    p = s.e[i];
       while (p)
    {
         A[i][p->nn] = 1;
      p = p->next;
    }
  }}
```

Функція додавання дуги в орієнтований граф, заданий структурою суміжності

```
void add(StrAdj gr, int i, int j)
{
  Node* p, q;
  if (i<0 || j<0 || i>=gr.n || j>=gr.n) return;
  for (p=q=gr.e[i]; q; q=q->next)
  {
    if (q->nn == j) return;
   }
  gr.e[i]=q = new Node;
  q->nn = j;
  q->next = p;
}
```

При реалізації матриці суміжності замість бітової матриці можна також використовувати вектор, елементами якого є цілі компоненти, що задають своїми двійковими представленнями рядки бітової матриці.

```
const int MaxN=16;
struct VctAdj
{
  int n;
  unsigned e[MaxN];
};
```

Наприклад, для матриці суміжності $\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$

```
VctAdj a1 = {4, {5, 2, 9, 0}};
```

Перевірка наявність ребра в графі, заданому матрицею суміжності

```
#define ised(gr, i, j) ((gr.e[i]>>(j))&01)
bool isedge(VctAdj gr, int i, int j)
{
  return (( i<gr.n && j<gr.n && i>=0 && j>=0)?
              (gr.e[i]>>gr.n-1-j & 01): 0);
}
```

Функція **isedge** здійснює додатково перевірку коректності аргументів. Макровизначення **ised** можна використовувати, якщо перевірка не потрібна.

Представлення графів може суттєво впливати на ефективність алгоритмів, на наочність програм. Часто буває доцільним задавати алгоритм незалежно від представлення графів, в термінах вершин та дуг.

Підрахунок кількості дуг орграфа G=(V,E) та степені кожної вершини

Алгоритм розв'язання задачі:

```
count(G, cnt, s) {
s ← 0;
for (v ∈ V) do
     {
    cnt[v] ← 0;
        for (t ∈ Adj[v]) do  cnt[v] ←cnt[v]+1;
        s ← s+cnt[v];
      }
}
```

Реалізація алгоритму:

```
#define ised(gr, i, j) ((gr.e[i]>>(j))&01)
//відношення суміжності задане вектором цілих
const int MaxN=16;
struct Vct {int n; unsigned e[MaxN];};
void count(Vct a, int cnt[MaxN], int *s)
```

```
{
  int v, t;
  for (v=0; v<MaxN; v++) cnt[v] = 0;
  for (*s=v=0; v<a.n; v++)
    {
   for (t=0; t<a.n; t++)
          if (ised(a, v, t)) cnt[v]++;
       (*s) += cnt[v];
}
}
```

Обчислення транзитивного замикання графа

Для орграфа G=(V,E) - транзитивне замикання - $G^*=(V,E^*)$, де $e=(v,w) \in E^* \Leftrightarrow$ існує шлях в G з v до w.

Матрицю A^* можна обчислити через матриці A^0, A^1, …, $A^n = A^*$, елементи яких визначаються таким чином: $a^0_{ij}=a_{ij}$, $a^k_{ij}= a^{k-1}_{ij} \vee a^{k-1}_{ik} \wedge a^{k-1}_{kj}$, k = 1, 2, …, n.

Фактично $a^k_{ij}= 1$ при існуванні шляху з $v_i \in V$ до $v_j \in V$ через вершини $\{v_1, v_2, \ldots, v_k\} \subset V$.

Обчислення транзитивного замикання графа, представленого матрицею суміжності.

```
const int MaxN=100;
void tran(bool A[MaxN][MaxN], int nn)
{
  for (int k=0; k<nn; k++)
    for (int i=0; i<nn; i++)
      if (A[i][k])
        for (int j=0; j<nn) A[i][j] = A[i][j] || A[k][j];
}
```

Обчислення транзитивного замикання графа, представленого вектором суміжності.

```
#define ised(gr, i, j) ((gr.e[i]>>(j))&01)
const int MaxN=16;
struct Vct
{
  int n;
  unsigned e[MaxN];
};
void count(Vct a)
{
  for (int k=0; k<a.n; k++)
    for (int i=0; i<a.n; i++ )
      if (ised(a, i, k)) a.e[i] |=a.e[k];
}
```

Зауваження

- Перед розв’язанням задач обробки графів за зовнішнім поданням графа треба будувати внутрішнє, а після розв’язання — навпаки.
- Один з найбільш вживаних способів внутрішнього представлення графів є структури суміжності.
- Матриця суміжності, як спосіб представлення графа, ефективна з погляду витрат пам’яті для графів близьких до повних.

Підсумки

- Були розглянуті основні підходи до представлення орієнтованих та неорієнтованих графів.
- В розглянутих прикладах, при різних способах представлення графів, обмежились простими діями, перевіркою простих властивостей, які не вимагали обходів графа.

- Перехід між різними способами представлення графів простий і відбувається за $O(n^2)$ операцій.

Поради

- Спосіб внутрішнього подання графа, повинен враховувати специфіку задачі й алгоритмів її розв'язання.
- По можливості формулювати алгоритм розв`язання задачі в термінах вершин та ребер, основних характеристик графа, не обмежуючись конкретним представленням графа.
- Самостійно розв`язати запропоновані нижче задачі.

Задачі

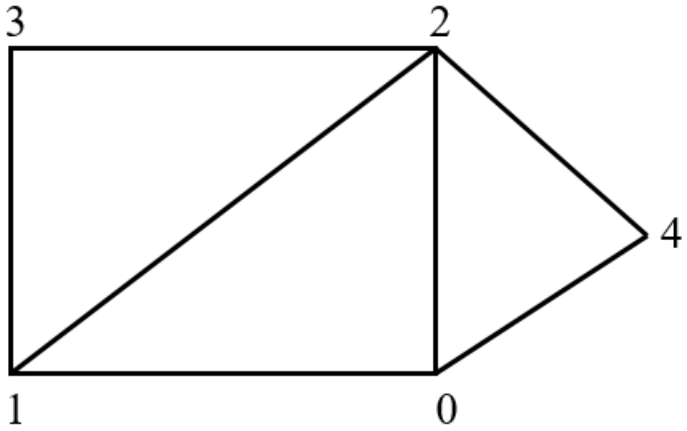
- Написати функцію, яка за матрицею суміжності графа будує структуру суміжності.
- Написати функцію, яка за матрицею суміжності графа будує вектор суміжності.
- Написати функцію, яка здійснює введення графу – за зовнішнім представленням графу будує внутрішнє у вигляді структури суміжності. Оберіть самостійно вид зовнішнього представлення графа.
- Написати функцію, яка перевіряє зв`язність орієнтованого графа, поданого матрицею суміжності.
- Написати функцію, яка перевіряє ациклічність орієнтованого графа, поданого:
    - матрицею суміжності;
    - структурою суміжності.
- Написати функцію, яка перевіряє чи є граф, поданий матрицею суміжності деревом.
- Орієнтований граф $G=(V,E)$ - транзитивний, якщо для нього виконується властивість: якщо $(v,u)\in E$ та $(u,w)\in E \rightarrow (v,w)\in E$. Написати функцію для перевірки транзитивності графа. Граф заданий матрицею суміжності.

6.2. Алгоритми на графах

В основі багатьох алгоритмів на графах лежить систематичний перебір вершин та ребер графа [9,10]. Розрізняють два базових алгоритми обходу графів:

- **Пошук у глибину**
- **Пошук у ширину**

Пошук у глибину



Порядок обходу вершин:

<0, 1, 2, 3, 4>

Рисунок 31. Пошук у глибину з початкової вершини 0

Стратегія: рухатись вглиб, поки є вихідні дуги, що не пройдені; повертатись й шукати інший шлях, коли таких дуг немає. Якщо залишились вершини, які не пройдені, брати одну з них і повторювати процес доки є не пройдені вершини.

Алгоритм:

```
//G=(V,E)
bool num[|V|]
search(G)
{
  for (v ∈ V) do num[v]←1;
  for (v ∈ V) do  if (num[v]) then  dfs(G,v);
}
dfs(G,v)
{
  num[v] ← 0;  {обробка v;}
  for (w ∈ Adj(v)) do  if (num[w]) then dfs(G,w);
```

```
}
```

- Складність алгоритму dfs для зв’язного (n, m)-графа має оцінку $O(m)$.
- Якщо граф подано матрицею суміжності, складність реалізації $O(n^2)$.
- Якщо граф подано структурою суміжності, складність реалізації $O(n+m) = O(m)$ (для зв`язного графа).

Набір даних для реалізації алгоритму на основі структури суміжності:

```
const int MaxN=16;
struct Node
{
  int nn;
  Node *next;
};
 struct StrAdj
{
  int n;
  Node* e[MaxN];
};
bool num[MaxN];
```

Реалізація алгоритму на основі структури суміжності:

```
void search_st(StrAdj s)
{
  int v;
  for (v=0; v<MaxN; v++)
    num[v] = v < s.n;
  for (v=0; v<s.n; v++)
    if (num[v]) dfs_st(s, v);
}

void dfs_st(StrAdj s, int v)
{
  Node* w;
  cout << v << ", ";
```

```
  num[v] = 0;
  for (w=s.e[v]; w; w=w->next)
    if (num[w->nn]) dfs_st(s, w->nn);
}
```

Набір даних та макросів для реалізації алгоритму на основі вектора суміжності:

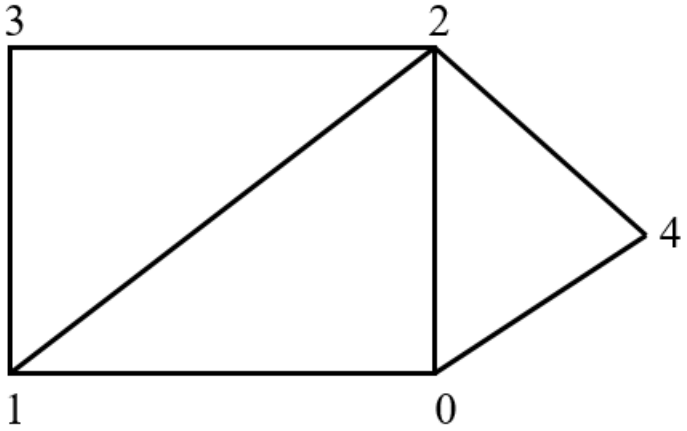
```
#define ised(gr, i, j) ((gr.e[i]>>(j))&01)
const int MaxN=16;
struct Vct {int n;
                unsigned e[MaxN];};
bool num[MaxN];
```

Реалізація алгоритму на основі вектора суміжності:

```
void search_st(Vct a)
{
  int v;
  for (v=0; v<MaxN; v++)
    num[v] = v < a.n;
  for (v=0; v<a.n; v++)
    if (num[v]) dfs_vc(a, v);
}

void dfs_vc(Vct a, v)
{
  cout << v << ", "; num[v] = 0;
  for (int w=0; w<a.n; w++)
    if (ised(a, v, w) && num[w]) dfs_vc(a, w);
}
```

Пошук у ширину



Порядок обходу вершин:

<0, 1, 2, 4, 3>

Рисунок 32. Пошук у ширину з початкової вершини 0

Алгоритм:

```
bool num[|V|]
search(G)
{
  for (v ∈ V) do num[v]←1;
  for (v ∈ V) do  if (num[v])  then  bfs(G,v);
}
bfs(G,v)
{
 //Q - черга
  Q ←Ø;  Q ← v;
  num[v] ← 0;
  while (Q ≠ Ø) do
{
  u ← Q;
 {
   обробка u;
  }
     for (w ∈ Adj(u)) do
       if (num[w])  then  {Q ← w;  num[w]←0;}
} }
```

- Складність алгоритму bfs для зв’язного (*n*, *m*)-графа має оцінку *O*(*m*).

- Якщо граф подано матрицею суміжності, складність реалізації $O(n^2)$.
- Якщо граф подано структурою суміжності, складність реалізації $O(n+m) = O(m)$ (для зв`язного графа).

```
//пошук у ширину
//представлення структурою суміжності
void search_bst(StrAdj s)
{
  int v;
  for (v=0; v<MaxN; v++)
    num[v] = v < s.n;
  for (v=0; v<s.n; v++)
    if (num[v]) bfs_st(s, v);
}

void bfs_st(StrAdj s, int v)
{
    int u, w;
    Node* dl = NULL, l = NULL, p;
    addq(&dl, &l, v); num[v] = 0;
    while (dl)
    {
        u = popq(&dl, &l);
        cout << u << "  ";
        for (p = s.e[u]; p; p = p->next)
        {
            w = p->nn;
            if (num[w])
            {
                addq(&dl, &l, w); num[w] = 0;
            }
  }}}
//додавання в чергу
void addq(Node** dl, Node** l, int v)
{
    Node* q;
    q = new Node;
```

```
    q->nn = v;  q->next = NULL;
    if (*dl)  (*l)->next = q;
    else *dl = q;
    *l = q;
}

//вилучення з черги
int popq Node** dl, Node** l)
{
    int v;
    Node* q;
    if (!(q = *dl)) return 0; //черга порожня
    if (q == *l) *dl = *l = NULL;
    else (*dl) = q->next;
    v = q->nn; delete q;
    return v;
}
```

Пошук компонент зв'язності

Вхід: Неорієнтований граф $G = (V, E)$.

Вихід: Вектор *cnm*, де елементи, що відповідають вершинам однієї компоненти зв'язності мають однакове значення.

Алгоритм: linkcm.

```
bool num[V]
linkcm(G, cnm)
{
    for (v ∈ V) do
    {
        num[v]←1;
        cnm[v] ← 0;
    }
    c ← 0;
    for (v ∈ V) do
        if (num[v]) then
        {
```

```
            c ← c + 1;
            comp(G, v, cnm, c);
        }
}

comp(G, v, cnm, c)
{
    num[v] ← 0;
    cnm[v] ← c;
    for (w ∈ Adj(v)) do
        if (num[w])  then  comp(G, w, cnm, c);
}
```

Реалізація:

```
void linkcm(StrAdj s, int cnm[MaxN])
{
    int v, c;
    for (v = 0; v < MaxN; v++)
    {
        num[v] = v < s.n;
        cnm[v] = 0;
    }
    for (c = v = 0; v < s.n; v++)
        if (num[v])
        {
            c++;
            cout << c << " :  ";
            comp(s, v, cnm, c);
            cout << endl;
        }
}

//перебір однієї компоненти
void comp(StrAdj s, int v, int cnm[], int c)
{
    Node* w;
    cout << v << " ";
```

```
        num[v] = 0;
        cnm[v] = c;
        for (w = s.e[v]; w; w = w->next)
            if (num[w->nn]) comp(s, w->nn, cnm, c);
    }
```

Топологічне сортування

Топологічне сортування вершин орієнтованого графа полягає у присвоєнні його вершинам чисел 1,2, …, |V| так, щоб при наявності в графі дуги (v_i,v_j) виконувалась умова i<j.

Топологічне сортування – визначення на множині вершин лінійного порядку, в який вкладається частковий порядок, що визначається дугами графа.

Топологічне сортування можливе тільки для ациклічного орієнтованого графа.

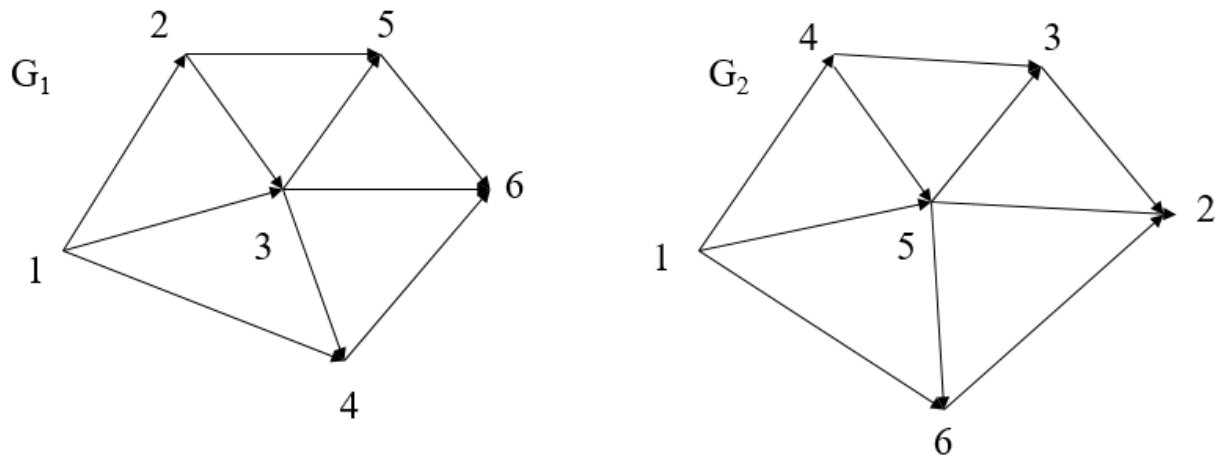


Рисунок 33. Топологічно відсортовані(G1) та невідсортовані(G2) вершини графа

Вхід: ациклічний орієнтований граф.

Вихід: нумерація вершин числами 1, 2, …, |V|.

Алгоритм:

1. C←|V|;
2. Відшукується вершина, з якої не виходять дуги. Вершина отримує номер C та вилучається разом з інцидентними ребрами.
3. C← C-1.

4. Переходимо до 2.

Алгоритм:

```
tsort(G, label)
for (v ∈ V) do
{
    num[v]←1;
    label[v] ← 0;
}
c ← | V | +1;
for (v ∈ V) do
if (num[v])  then  tops(G, v, label, c);

tops(G, v, label, c)
{
    num[v] ← 0;
    for (w ∈ Adj(v)) do
        if (num[w])  then  tops(G, w, label, c);
        else  if (label[w] = 0)  then{ G має цикл };
    c ← c - 1;
    label[v] ← c;
}
```

Реалізація алгоритму:

```
//представлення структурою суміжності
void tsort(StrAdj s, int label[MaxN])
{
    int v, c;
    for (v = 0; v < MaxN; v++)
    {
        num[v] = v < s.n;
        label[v] = 0;
    }
    for (c = s.n + 1, v = 0; v < s.n; v++)
        if (num[v]) tops(s, v, label, c);
    cout << endl;
}
```

```
//перебір однієї компоненти
void tops(StrAdj s, int v, int label[], int& pc)
{
    Node* w;
    num[v] = 0;
    for (w = s.e[v]; w; w = w->next)
        if (num[w->nn]) tops(s, w->nn, label, pc);
        else if (!label[w->nn]) cout << "cycle " << w->nn;
    pc--;
    label[v] = pc;
    cout << v << " ";
}
```

Пошук найкоротших шляхів у графі

Ця задача може бути конкретизована у декілька способів:

- Пошук найкоротших шляхів між всіма парами вершин графа.
- Пошук найкоротших шляхів між виділеною вершиною та всіма іншими вершинами графа;
- Пошук найкоротшого шляху між двома виділеними вершинами графа;

Обчислення відстаней між всіма парами вершин

Знайдемо довжини найкоротших простих ланцюгів для всіх пар вершин.

Відстані подаються у вигляді матриці *D*.

Початкова ініціалізація:

1. $d_{i\,i} \leftarrow 0$;
2. $d_{i\,j} \leftarrow 1$, якщо різні вершини *i* та *j* суміжні;
3. $d_{i\,j} \leftarrow \infty$, якщо вершини *i* та *j* несуміжні.

Поступово будемо додавати вершини, які можуть бути проміжними в цих ланцюгах.

Вхід. Неорієнтований зв'язний граф $G = (V, E)$ з вершинами 1, …, *n* у вигляді матриці $D(1{:}n, 1{:}n)$.

Вихід. Матриця *D*, в якій d_{ij} — це відстань між вершинами *i* та *j* в графі.

***Алгоритм** (Флойда).*

```
for k ←1 to n do
  for i←1 to n do
    for j ←1 to n do
      dij ←  min(dij, dik+dkj);
```

Складність алгоритму Флойда $O(n^3)$. Для знаходження самих шляхів між вершинами необхідно динамічно заповнювати додаткову матрицю A де у кожній клітинці A[i][j] зберігатиметься наступна вершина на шляху від вершини i до вершини j.

Обчислення відстаней від заданої вершини

Алгоритм обчислення відстаней від заданої вершини до всіх інших - на основі обходу графа в ширину

Вхід. Неорієнтований зв'язний граф $G = (V, E)$ і початкова вершина v_0.

Вихід. Для кожної вершини v її відстань $d(v)$ від v_0.

***Алгоритм** (Дейкстри).*

```
//v0 – початкова вершина
for (v ∈ V) do d(v)← ∞;
     d(v0) ← 0;
     Q ← Ø; //Q – черга
     Q ← v0;
     while (Q ≠ Ø) do
{
       u ← Q;
            for (w ∈ Adj (u)) do
              if  (d(w) = ∞)  then
   {
               d(w) ← d(u)+1;
     Q ← w;
    }
 }
```

Обчислення найкоротшого шляху між двома визначеними вершинами базується на попередньому алгоритмі. Вищеописані алгоритми можна модифікувати у випадку графа зі зваженими ребрами. У такому випадку Довжиною шляху вважають сумарну “вагу” по всіх його ребрах.

Побудова кістякового дерева мінімальної ваги

Нехай G=(V,E) – зв`язний неорієнтований граф, на множині ребер, E - задана функція ваги.

Кістякове дерево – підграф S=(V,T), що містить всі вершини G і є деревом. Вага ребер природнім чином визначає вагу кістякового дерева.

Алгоритм Крускала побудови КДМВ

Вхід. Неорієнтований зв’язний граф G = (V, E).

Вихід. Список T ребер кістякового дерева мінімальної ваги.

Цей алгоритм називають “Жадібним алгоритмом”. Його складність O(|E|log|E|).

Упорядкувати множину ребер E за неспаданням ваги;

```
for (v ∈ V) do cmp(v) ← v;
 ncomp ← n;
 i ← 0;
T ← Ø;
 while ncomp > 1 do
{
  i ← i+1;
  обрати E[i] з кінцями u і w;
   if cmp(u) ≠ cmp(w) then
  {
      об’єднати КЗ, яким належать u і w;
      додати ребро (u, w) до T;
      ncomp ← ncomp-1;
      cmp(w) ← cmp(u)
   }
 }
```

Алгоритм Дейкстри-Пріма

Вхід. Неорієнтований зв'язний граф $G = (V, E)$.

Вихід. Список T ребер кістякового дерева мінімальної ваги.

Алгоритм ("Алгоритм найближчого сусіда") має складність Складність O(|V| 2) у найгіршому випадку, коли граф є повним.

```
U ← {v1};
T ← Ø;
while (U ≠ V) do
{
  серед ребер між U та V\U знайти ребро (u, w) мінімальної ваги;
   T ← T ∪{(u, w)};
   U ← U ∪ {w};
 }
```

Побудова кістякового дерева пошуком у глибину

Пошук у глибину на зв`язному неорієнтованому графі G=(V,E) призводить до розбиття множини ребер E на дві підмножини: T – "деревних" ребер; E \ T – "обернених" ребер.

У зв'язку з поставленою задачею природнім чином виникають запитання:

- Практичні застосування задачі?
- Яка кількість кістякових дерев графа?
- Яка кількість мінімальних кістякових дерев графа?
- Як узагальнити задачу на незв`язний неорієнтований граф?
- Як узагальнити задачу на орієнтовані графи?

Зауваження

- Для розглянутих задач та алгоритмів їх розв`язання спосіб внутрішнього представлення графів не є таким вже й суттєвим. Реалізація

алгоритму досить просто враховує довільний “класичний” спосіб представлення графів, але спосіб представлення впливає на оцінки складності.

- Алгоритми на графах суттєво використовують стеки та черги при роботі з даними.

Поради

- Спосіб внутрішнього подання графа, повинен враховувати специфіку задачі й алгоритмів її розв’язання.
- По можливості формулювати алгоритм розв`язання задачі в термінах вершин та ребер, основних характеристик графа, не обмежуючись конкретним представленням графа.
- Самостійно розв`язати запропоновані нижче задачі.

Задачі

- Стягнути дві вершини графа в одну.
- Написати функцію, що здійснює обхід графа пошуком у глибину. Граф представлений матрицею суміжності.
- Написати нерекурсивну функцію, що здійснює обхід графа пошуком у глибину. Граф представлений структурою суміжності.
- Написати функцію, що здійснює обхід графа пошуком у ширину. Граф представлений матрицею суміжності.
- Реалізувати алгоритм обчислення довжини найкоротшого шляху між двома виділеними вершинами.
- Запропонувати й реалізувати алгоритм для знаходження найкоротшого шляху між двома виділеними вершинами.
- Модифікувати розглянутий алгоритм (Флойда) для знаходження найкоротших шляхів між всіма парами вершин графу.
- Які наслідки в задачах про найкоротші шляхи будуть мати від`ємні “довжини” ребер?
- Реалізувати алгоритм Крускала.

- Модифікувати алгоритм Крускала для незв`язного графу.
- Реалізувати алгоритм Прима.
- Реалізувати алгоритм побудови кістякового дерева пошуком у глибину.
- Реалізувати алгоритм побудови кістякового дерева пошуком в ширину.
- Нехай R={(i,j)| mod(i,j)=0}- бінарне відношення на множині M={2, …, 100}. Написати програму яка друкує елементи M у топологічному порядку. Скористатись тим, що бінарному відношенню R природнім чином відповідає граф G=(M,E) : $(i,j)\in E \leftrightarrow (i,j)\in R$.

7. Лабораторні роботи

Лабораторна робота №1. Списки

Реалізувати функції та створити базові структури, необхідні для роботи зі списком, а саме введення елементів списку користувачем та його виведення в консоль. Тип та особливості будови списку визначаються студентом самостійно. Також реалізувати функції та провести операції відповідно до варіанту:

1) Занести в список перші N раціональних чисел, використовуючи діагональний метод Кантора. Реалізувати функції: пошук числа у списку, видалення заданого числа зі списку, сума всіх елементів у списку, додавання наступного числа згідно діагональним методом Кантора.
2) Список зберігає структуру з полями «коефіцієнт» та «степінь», впорядковану за другим полем та кодує поліном $f(x)$. Визначити всі можливі остачі від ділення $f(x)$ на число A.
3) Список зберігає структуру з полями «коефіцієнт» та «степінь», впорядковану за другим полем та кодує поліном від змінної x. Визначити остачу від ділення цього поліному на $Ax + B$. A, B задаються з клавіатури
4) Два списки зберігають структуру з полями «коефіцієнт» та «степінь» впорядковані за другим полем та кодують два поліноми однієї змінної. Створити список, що кодує добуток цих двох поліномів
5) У списку представлено булеву функцію від n змінних у формі ДНФ, кожен елемент списку – один з доданків (елементарна кон’юнкція). Вивести таблицю значень функції.
6) У списку представлено булеву функцію від n змінних у формі КНФ, кожен елемент списку – один з доданків (елементарна диз’юнкція). Вивести таблицю значень функції.
7) У списку представлено булеву функцію від n змінних у формі поліному Жегалкіна, кожен елемент списку – один з доданків. Вивести таблицю значень функції.

8) У списку представлено булеву функцію від n змінних у формі ДНФ, кожен елемент списку – один з доданків (елементарна кон'юнкція). Записати функцію у вигляді поліному Жегалкіна (також у списку)
9) У списку представлено булеву функцію від n змінних у формі поліному Жегалкіна, кожен елемент списку – один з доданків. Записати функцію у вигляді ДНФ (також у списку)
10) Два списки, у яких представлено булеві функції f, g від n змінних у формі ДНФ, кожен елемент списку – один з доданків. Визначити, чи імплікує функція f функцію g.
11) У списку представлено булеву функцію від n змінних у формі ДНФ, кожен елемент списку – один з доданків. Знайти скорочену ДНФ цієї функції. (знайти всі мінімальні ДНФ цієї функції +1 бал.)
12) Розкласти многочлен $(ax + b)^n$ та записати відповідні коефіцієнти поліному у вигляді списку.
13) Шахова дошка кодується чотири-зв'язним списком, у якому зберігаються числа, що кодують колір фігури та її тип. Вивести списком всі клітинки, на які може походити фігура з заданої клітинки.
14) У чотири-зв'язному списку кодуються чорні клітинки дошки для шашок. Визначити, скільки шашок протилежного кольору може побити задана шашка.
15) Для двох множин поданих списками реалізувати основні операції: об'єднання, перетину, різниці.
16) Для двох множин поданих списками реалізувати порівняння (=, $\subset$, $\subseteq$) .

*** Використання масивів та вбудованих контейнерів не допускається!

*** Для зчитування даних у список рекомендується використовувати файли

Лабораторна робота №2. Дерева

Операції над деревами

Реалізувати функції та створити базові структури, необхідні для роботи з деревом в стандартному представленні, а саме введення елементів користувачем та його виведення в консоль. Тип та особливості будови дерева визначаються студентом самостійно. Також реалізувати функції та провести операції відповідно до варіанту:

1) За посиланням на корінь дерева, вивести його елементи по рівнях (Спочатку кореневий елемент, потім його сини, потім сини синів тощо.)
2) За посиланням на корінь дерева, з'єднати посиланнями сусідні вузли дерева, що знаходяться на одному і тому самому рівні (Вважати що у кожного вузла є відповідне поле)
3) За посиланням на корінь дерева знайти найдовший ланцюг у цьому дереві (розглядаємо дерево як зв'язний ациклічний граф)
4) За посиланням на корінь дерева, знайти ланцюг у цьому дереві з максимальною сумою значень елементів (розглядаємо дерево як зв'язний ациклічний граф)
5) За посиланням на корінь дерева, знайти центр цього дерева (розглядаємо дерево як зв'язний ациклічний граф)
6) За посиланнями на два вузли дерева, знайти найближчого спільного батьківського елемента цих вузлів, не обов'язково безпосереднього (вважаємо що кожен вузол має посилання на батьківський елемент)
7) За посиланням на корінь дерева, перетворити його на двозв'язний список без виділення додаткової пам'яті (прохід дерева центрований, Inorder traversal)

*** Реалізувати порівняння дерев на рівність, ізоморфізм.

Задачі на різновиди дерев

1) Реалізувати AVL-дерево
2) Реалізувати червоно-чорне дерево
3) Реалізувати 2-3 дерево (https://en.wikipedia.org/wiki/2%E2%80%933_tree)
4) Реалізувати B-tree
5) Реалізувати дії з «прошитими» деревами, а також визначення їх основних характеристик.
6) Реалізувати дерево відрізків для суми відрізка елементів у масиві. Реалізувати запит суми та зміну елемента масиву з перерахунком.
7) Реалізувати дерево відрізків для НОК відрізка елементів у масиві. Реалізувати запит суми та зміну елемента масиву з перерахунком).

Задачі на дерева виразів

1) Реалізувати перевід рядка у дерево виразів. (змінні, дійсні числа, операції +,-,*,/, унарний мінус, sin, cos). Обчислити вираз, роблячи запит по змінним
2) Реалізувати перевід рядка у дерево виразів(змінні, дійсні числа, операції +,-,*,/,^, унарний мінус, exp, log(y,x) де y -- база логарифмування). Обчислити вираз, роблячи запит по змінним
3) Реалізувати перевід рядка у дерево виразів (змінні, дійсні числа, операції +,-,*,/, унарний мінус, степінь). Обчислити вираз, роблячи запит по змінним
4) Реалізувати перевід рядка у дерево виразів(змінні, цілі числа, операції +,-,*, унарний мінус, ділення націло, остача від ділення, піднесення до натуральної степені). Обчислити вираз, роблячи запит по змінним
5) Реалізувати перевід рядка у дерево виразів(змінні, дійсні числа, операції +,-,*,/). Символьно продиференціювати вираз по вибраній змінній та вивести на екран результат у вигляді рядка.
6) Реалізувати перевід рядка у дерево виразів(змінні, дійсні числа, операції +,-,*, sin, cos). Символьно продиференціювати вираз по вибраній змінній та вивести на екран результат у вигляді рядка.

7) Реалізувати перевід рядка у дерево виразів(змінні, дійсні числа, операції +,-,*, exp). Символьно продиференціювати вираз по вибраній змінній та вивести на екран результат у вигляді рядка.

*** графічно намалювати побудоване дерево +1 бал.

Лабораторна робота №3. Графи

Реалізувати функції та створити базові структури, необхідні для роботи з графами, а саме введення елементів користувачем та його виведення в консоль. Метод збереження та особливості графу визначаються студентом самостійно. Також реалізувати функції та провести операції відповідно до варіанту:

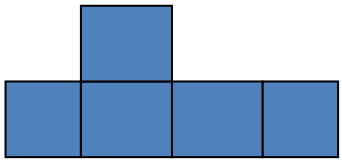
1) Для орієнтованого графа знайти мінімальну підмножину вершин з яких є досяжними всі його інші вершини.
2) Перевірити чи є граф ациклічним.
3) Перевірити чи є граф транзитивним.
4) Знайти діаметр графа (відстань між двома найбільш віддаленими вершинами)
5) Визначити хроматичний многочлен та хроматичне число графа.
6) Визначити визначник матриці суміжності графа.
7) Визначити число компонент зв’язності графа.
8) Визначити компоненти двозв’язності графа.
9) Визначити мінімальний цикл Гамільтона графа
10) Знайти максимальний повний підграф у заданому графі.
11) Перевірити, чи є орієнтований граф сильно зв’язним
12) Визначити обхват графа – найменший по довжині цикл у графі.
13) Побудувати мінімальне кістякове (остовне) дерево у графі.
14) Визначити всі «мости» графа.
15) Визначити всі вершини - «точки зчеплення» графа.
16) Перевірити граф на планарність.
17) Знайти мінімальну кількість ребер, після вилучення яких граф втратить зв’язність.
18) Знайти мінімальну кількість вершин, після вилучення яких граф втратить зв’язність.

*** реалізувати зчитування графу з файлу

Лабораторна робота №4. Пошук з поверненням

1) Розставити на шаховій дошці 8 ферзів так, щоб вони не били одна одну.
2) Кінь розміщується на першій клітинці порожньої шахової дошки і, рухаючись за правилами шахів, повинен один раз відвідати кожну клітинку. Вивести на екран маршрут коня.
3) У множині елементів (цілі числа), що задається масивом, знайти підмножину, сума елементів якої дорівнює K.
4) Квадратна матриця складається з нулів та одиниць. Знайти найдовший шлях у матриці між двома заданими точками, що складається тільки з одиниць. Доступні кроки вправо, вліво, вверх, вниз.
5) Дано послідовність додатних чисел. Розставити між ними знаки «+» та «-» так, щоб у результаті ми одержали 0. Якщо це не можливо, вивести відповідне повідомлення.
6) Вивести на екран всі перестановки чисел від 1 до N, при яких $\forall i\ a[i] \neq i$.
7) Побудувати послідовність з N літер в алфавіті, що складається з трьох елементів (наприклад, 1, 2, 3), таку, що ніякі дві сусідні підпослідовності не співпадають.
8) Знайти всі послідовності довжини N, що не містять однакових сусідніх підпослідовностей.
9) Запропонувати та реалізувати не рекурсивний алгоритм для задачі “Ханойські вежі”.
10) Є набір кісток доміно (не обов`язково повний). Знайти правильну послідовність з наявних кісток максимальної довжини.
11) Для лабіринту знайти шлях з однієї зазначеної точки в іншу.
12) Знайти мінімальну множину прямих, на яких можна розташувати всі точки заданої множини.
13) Є набір слів. Скласти з них правильний ланцюжок максимальної довжини (по кількості слів, або кількості букв). Ланцюжок є правильним, якщо перша буква наступного слова співпадає з останньою буквою попереднього.

14) Обчислити всі найдовші спільні підрядки двох рядків та їх довжину. Наприклад, у рядків abcab та ccabc такими є abc та cab.
15) На шаховій дошці розміром n*n потрібно розставити n ферзів, таким чином, щоб жодний ферзь не знаходився під ударом іншого.
16) Для шахової дошки розміром N*M підрахувати кількість обходів фігурою кінь, при яких кожне поле шахової дошки відвідується в точності по одному разу.
17) Є предмети N різних типів (кількість предметів кожного типу не обмежена). Кожний предмет типу i має вагу w_i та вартість v_i. Потрібно визначити максимальну вартість рюкзака, вага якого не перевищує W.
18) Написати програму для визначення всіх можливих розміщень k ферзів (k<8), що не атакують один одного, на шахівниці розміром 8*8 так, щоб кожна її клітина перебувала під боєм.
19) Написати функцію, яка для заданого n>0 перевіряє, чи є можливість покрити прямокутник розміром 12*5n фігурами



де сторона кожного квадрата дорівнює 1.
20) Для задачі про шахового коня розглянути постановку де потрібно знаходити та друкувати тільки один варіант обходу. Оптимізувати рішення задачі. (Правило Варнсдорфа – при обході слід обирати поле, з якого існує мінімальна кількість переміщень на незайняті поля. Тоді в першу чергу займають граничні та кутові поля й кількість “повернень” – зменшується.).

Ігри для одного гравця

1) П’ятнашки. Забезпечити розв’язання головоломки комп’ютером. https://uk.wikipedia.org/wiki/П%27ятнашки
2) Тетріс. https://uk.wikipedia.org/wiki/Тетріс
3) Сапер. https://uk.wikipedia.org/wiki/Сапер_(відеогра)
4) Змійка. https://uk.wikipedia.org/wiki/Змійка(гра)

5) Кубик-рубик. Забезпечити розв’язання головоломки комп’ютером. https://uk.wikipedia.org/wiki/Кубик_Рубіка
6) Кульки. https://uk.wikipedia.org/wiki/Кульки
7) Bubble Shooter https://uk.wikipedia.org/wiki/Bubble_Shooter
8) Судоку. Реалізувати генерацію головоломки та можливість її автоматичного розв'язання. https://uk.wikipedia.org/wiki/Судоку

Ігри з симуляцією фізики

1) Більярд «Карамболь» https://uk.wikipedia.org/wiki/Карамболь
2) Doodle Jump https://uk.wikipedia.org/wiki/Doodle_Jump
3) Cut the Rope. https://uk.wikipedia.org/wiki/Cut_the_Rope
4) Brick Breaker https://en.wikipedia.org/wiki/Brick_Breaker
5) Asteroids https://uk.wikipedia.org/wiki/Asteroids

Настільні ігри з почерговими ходами

1) Морський бій, проти комп’ютера. https://uk.wikipedia.org/wiki/Морський_бій_(настільна_гра)
2) Score four, проти комп’ютера https://en.wikipedia.org/wiki/Score_four
3) Ґомоку, проти комп’ютера. https://uk.wikipedia.org/wiki/Ґомоку
4) «Чотири в ряд», проти комп’ютера. https://uk.wikipedia.org/wiki/Чотири_в_ряд
5) Реверсі, проти комп’ютера. https://uk.wikipedia.org/wiki/Реверсі
6) Турецькі шашки, проти комп’ютера. https://uk.wikipedia.org/wiki/Турецькі_шашки
7) Шашки, проти комп’ютера. https://uk.wikipedia.org/wiki/Шашки

*** першочергова увага приділяється розробленому алгоритму (емуляція фізики, стратегія комп’ютерного гравця), а не візуальній компоненті.

8. Перелік джерел посилань

1. Є. О. Іванов, Я. М. Ліндер, і К. А. Жереб, Основи мови програмування С++: навчальний посібник. Київ: Логос, 2020, 90 с. — ISBN 978-617-7631-24-7.
2. Бєлов Ю.А., Карнаух Т.О., Коваль Ю.В., Ставровський А.Б. Вступ до програмування мовою С++. Організація обчислень. – К.: ВПЦ "Київський університет", 2012. – 175 с.
3. Карнаух Т.О., Коваль Ю.В., Потієнко М.В., Ставровський А.Б. Вступ до програмування мовою С++. Організація даних. – К.: ВПЦ "Київський університет", 2015. – 151 с.
4. Проценко В.С., Чаленко П.Й., Ставровський А.Б. Техніка програмування мовою Сі. Навч. посібник. — К.: Либідь, 1993. — 224 с. — ISBN 5-325-00321-6.
5. В.В. Зубенко, Л.Л. Омельчук Програмування. Поглиблений курс. – К.:Видавничо-поліграфічний центр "Київський університет", 2011. - 623 с. Бібліогр.: с. 602-609. ISBN 978-966-439-380-2.
6. В.В. Зубенко, Л.Л. Омельчук Програмування. Поглиблений курс. – К.:Видавничо-поліграфічний центр "Київський університет", 2011. - 623 с. Бібліогр.: с. 602-609. ISBN 978-966-439-380-2.
7. Narsingh Deo. Graph Theory with Applications to Engineering and Computer Science (Dover Books on Mathematics)/ Deo Narsingh. — Dover Publications; First Edition, First (August 17, 2016). — 496 p.
8. Веклич Р.А., Карнаух Т. О., Ставровський А. Б. Вступ до програмування мовою С++ : структури даних . –К. : ВПЦ "Київський університет", 2018. – 99 с.
9. Карнаух Т. О. Теорія графів у задачах / Т. О. Карнаух, А. Б. Ставровський. – К. : КНУ, 2012. – 90 с.
10. Кузьменко, І. М. Теорія графів [Електронний ресурс] : навчальний посібник для здобувачів ступеня бакалавра за спеціальністю 122 «Комп'ютерні науки» / І. М. Кузьменко ; КПІ ім. Ігоря Сікорського. – Електронні текстові. – Київ : КПІ ім. Ігоря Сікорського, 2020. – 71 с.