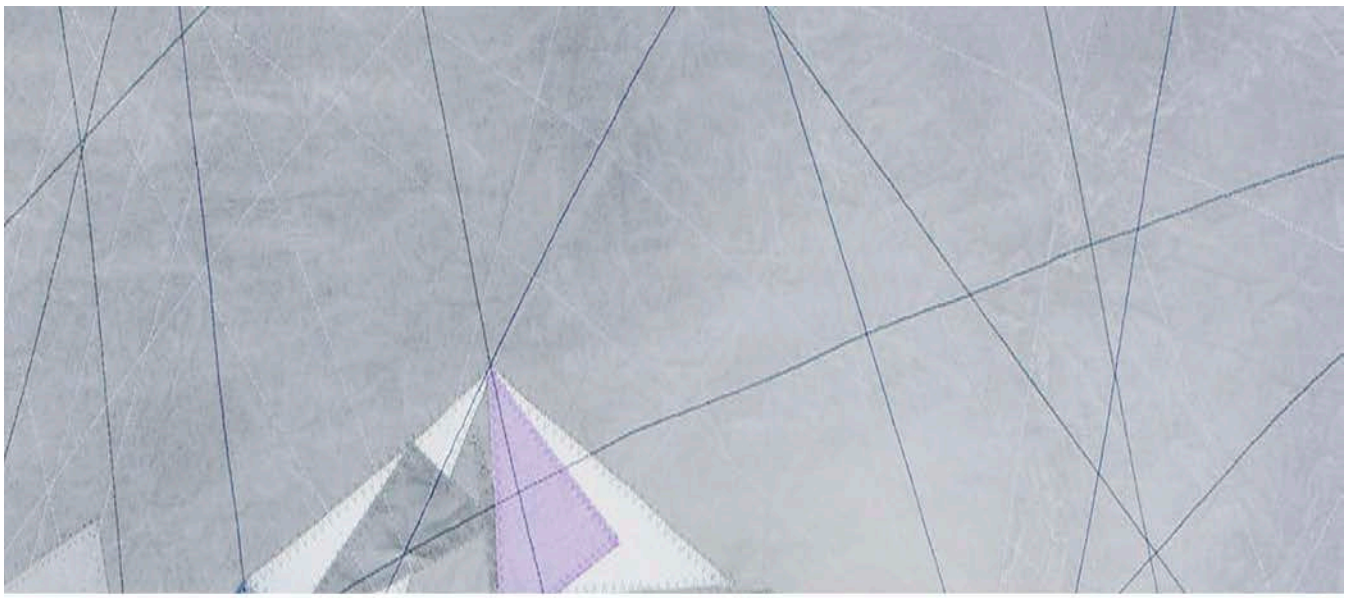
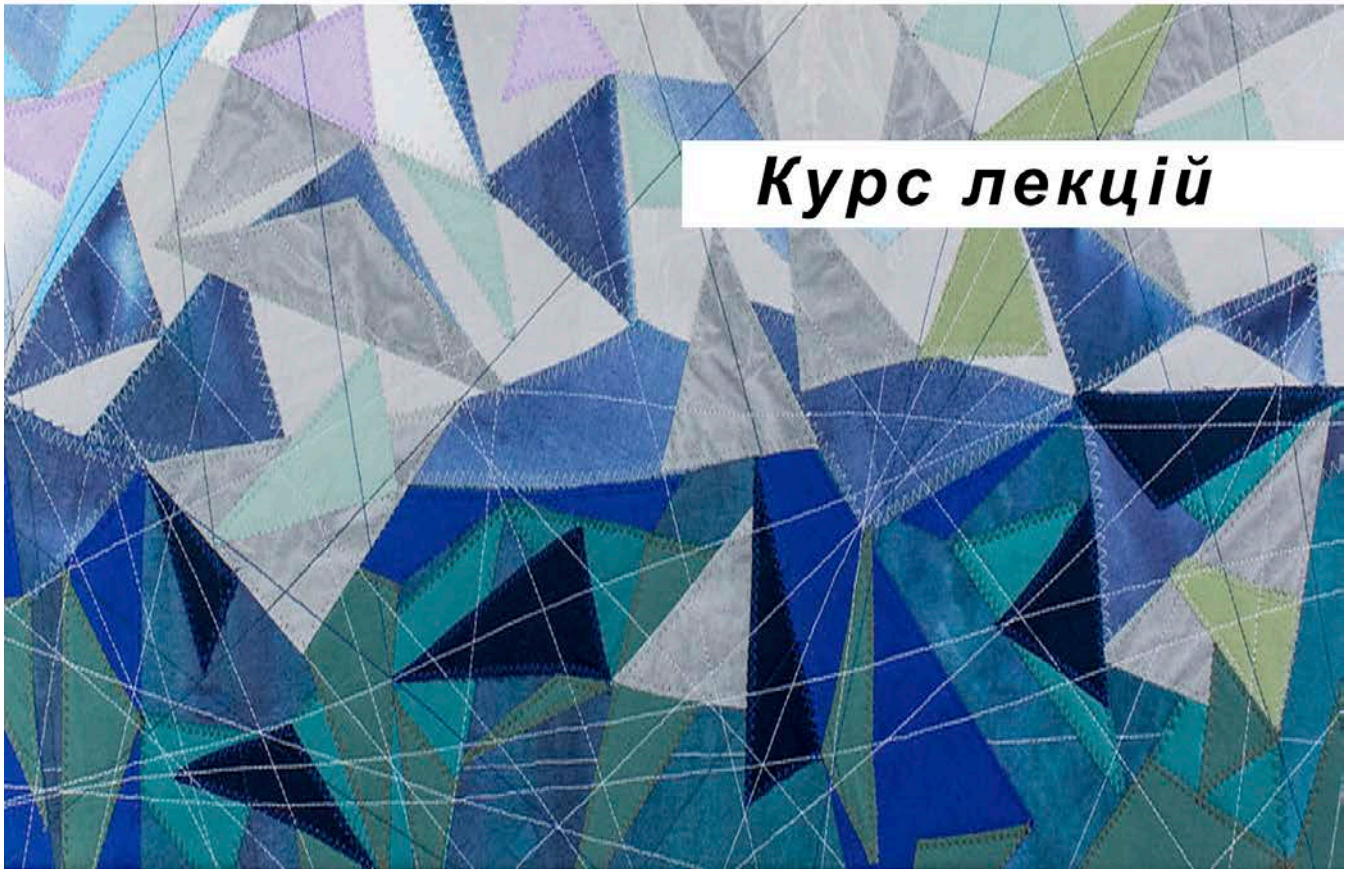


АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ



Ніжинський державний університет
імені Миколи Гоголя

АЛГОРИТМИ
ТА СТРУКТУРИ ДАНИХ

Курс лекцій

Упорядник:
В. М. Харченко

Ніжин – 2023

УДК 004.421(075.8)
Х22

Рекомендовано Вченою радою
Ніжинського державного університету імені Миколи Гоголя
(НДУ ім. М. Гоголя)
Протокол № 13 від 30.06.2023 р.

Рецензенти:

ЛИСЕНКО Ірина – доцент кафедри інформаційних технологій, фізико-математичних та економічних наук, кандидат фізико-математичних наук;

КРЕСАН Тетяна – доцент, канд. техн. наук, в.о. завідувача кафедри природничо-математичних та загальноінженерних дисциплін відокремленого підрозділу Національного університету біоресурсів і природокористування України «Ніжинський агротехнічний інститут».

Х22 Алгоритми та структури даних: курс лекцій / упорядник: В. М. Харченко. – Ніжин: НДУ ім. М. Гоголя, 2023. – 245 с.

Даний посібник відповідає програмі курсу «Алгоритми і структури даних» для спеціальності 122 Комп’ютерні науки. Він написаний на основі використання мови програмування С++ і вимагає знань, умінь та навичок курсів «Програмування», «Дискретна математика», «Архітектура комп’ютерних систем».

Кожна з 11 тем містить план, список використаної літератури, детальний розгляд питань. Основні положення підтверджуються прикладами. У переважній більшості, прикладами є коди функцій або програм, що написані мовою програмування С++.

Посібник може бути корисним учителям та учням загальноосвітніх шкіл при підготовці до олімпіад з інформатики.

УДК 004.421(075.8)

© В.М. Харченко, 2023
© НДУ ім. М. Гоголя, 2023

ЗМІСТ

ВСТУП

Мета та завдання навчальної дисципліни познайомити студентів з різними способами представлення даних в пам'яті комп'ютера, і з різними класами завдань і типами алгоритмів, що зустрічаються при розв'язуванні задач на сучасних комп'ютерах.

Після вивчення даної дисципліни студент повинен розуміти, що етап вибору або розробки алгоритму при розв'язуванні задач на сучасних комп'ютерах має велике і все зростаюче значення в міру розвитку обчислювальної техніки і засобів автоматизації програмування.

Формула

Програма = Алгоритм + Структури даних + Алгоритмічна мова

була запропонована в однойменній книзі Ніклауса Вірта. Дослідження алгоритмів і структур даних є однією з основ програмування.

Сучасний рівень програмування передбачає застосування структур даних, як необхідних атрибутів програмних конструкцій. У результаті вивчення даного курсу студент повинен ефективно вирішувати питання, що виникають на стадії розробки або вибору алгоритму. До цих питань відносяться: обґрунтування і вибір структури представлення даних, аналіз складності розробленого алгоритму, оцінка витрат на розробку алгоритму в залежності від класу розв'язуваних задач і наявних або потрібних для їх розв'язання обчислювальних засобів.

Даний курс слід вивчати тому, що розуміння основ алгоритмізації і тісно з нею взаємозалежної сфери організації структур даних необхідно для виконання серйозної роботи практично в будь-якій галузі інформатики.

Алгоритми все більше і більше використовуються для того, щоб по-новому поглянути на наукові проблеми за межами computer science та IT. Наприклад, дослідження квантових обчислень забезпечило новий обчислювальний погляд на квантову механіку. Цінові коливання на фінансових ринках можуть розглядатися як алгоритмічний процес. Навіть еволюцію можна розглядати як ефективний алгоритм пошуку.

Даний курс лекцій написаний на основі використання мови програмування C++. Курс вимагає знань курсів «Програмування», «Дискретна математика», «Архітектура комп'ютерних систем».

Курс може бути корисний учителям та учням загальноосвітніх шкіл при підготовці до олімпіад з інформатики.

Тема №1. АЛГОРИТМИ І ДАНІ

План лекції:

1. Вступ
2. Поняття алгоритму
3. Алгоритмічний аналіз
4. Які види задач розв'язують алгоритми?
5. Концепція структур даних
6. Класифікація структур даних

Джерела:

[1, §1.1], [2, Розділ 1,2], [3, §1.1 – 1.3], [4, Глава 1], [5, Chapter 1], [7, §1.1], [10, §1.1].

1. Вступ

Ми починаємо вивчати курс на 4 кредити ЄКТС: «Алгоритми та структури даних».

Дисципліна належить до переліку дисциплін професійно-практичної підготовки за освітнім рівнем «бакалавр». Предметом вивчення дисципліни є структури даних та типові алгоритми обробки інформації. Курс вимагає знань курсів «Програмування», «Дискретна математика», «Архітектура комп'ютерних систем».

Що таке алгоритми? Навіщо вивчати алгоритми? Яка роль алгоритмів для інших технологій, що використовуються в комп'ютерах? У цій лекції ми відповімо на ці запитання.

2. Поняття алгоритму

Неформально, **алгоритм** — це будь-яка добре визначена обчислювальна процедура, яка приймає деяке значення або набір значень як вхідні дані і видає деяке значення або набір значень як вихідні дані. Таким чином, алгоритм є послідовністю обчислювальних кроків, які перетворюють вхідні дані у вихідні.

Ми також можемо розглядати алгоритм як інструмент для розв'язання чітко визначеної обчислювальної задачі. Постановка проблеми в загальних рисах визначає бажане співвідношення введення/виведення. Алгоритм описує конкретну обчислювальну процедуру для досягнення цього зв'язку введення/виведення.

Алгоритм називається **правильним**, якщо для кожного вхідного екземпляра він зупиняється з правильним виведенням. Ми говоримо, що правильний алгоритм розв'язує задану обчислювальну задачу. Неправильний алгоритм може взагалі не зупинятися на деяких екземплярах уведення, або він

може зупинятися з неправильною відповіддю. Усупереч тому, що можна очікувати, неправильні алгоритми іноді можуть бути корисними, якщо можна контролювати їх частоту помилок. Але зазвичай ми будемо займатися лише правильними алгоритмами.

Загальновживаного означення алгоритму до даного часу не існує.

Наведемо деякі спроби дати означення:

Алгоритм – це розв'язання, що «складається із скінченної послідовності інструкцій, кожна із яких має чіткий смисл і може бути виконана зі скінченнимиобчислювальними затратами за скінченний час» [3, с.16].

Алгоритм – це кінцевий набір правил, що визначає послідовність операцій для рішення конкретної множини задач і володіє п'ятьма важливими рисами: скінченність, визначеність, введення, виведення, ефективність [7, с.4-5].

Алгоритм – « це будь-яка коректно визначена обчислювальна процедура, на вхід (input) якої подається деяка величина або набір величин і результатом виконання якої є вихідна (output) величина або набір значень» [5, с. 5].

Будемо вважати, що:

Алгоритм – ***це точний і зрозумілий для виконавця припис про здійснення скінченної послідовності визначених дій з метою розв'язання задачі певного типу або досягнення поставленої мети.***

Слово алгоритм походить від імені перського вченого, астронома та математика Абу Абдулли Абу Джафар Мухаммад ібн Муса аль-Хорезмі (близько 780 — близько 850). Приблизно 825 р. він написав трактат, в якому описав вигадану в Індії позиційну десяткову систему числення.

Алгоритми, як правило, створюються незалежно від базових мов. Це означає, що алгоритм може бути реалізований більш ніж однією мовою програмування.

З точки зору структури даних, деякі наступні алгоритми є важливими категоріями:

Пошук – алгоритм пошуку елемента в структурі даних.

Сортування – алгоритм розташування елементів у певному порядку.

Вставка – алгоритм для вставки елемента в структуру даних.

Оновлення – алгоритм оновлення існуючого елемента в структурі даних.

Видалення – алгоритм видалення існуючого елемента зі структури даних.

Властивостями алгоритму є дискретність, визначеність, виконуваність, скінченність, результативність і масовість.

Дискретність алгоритму означає, що його виконання зводиться до виконання окремих дій (кроків) у певній послідовності. Зауважимо, що кожна команда алгоритму повинна виконуватися за скінченний проміжок часу.

Визначеність алгоритму означає, що для заданого набору значень вхідних даних алгоритм однозначно визначає порядок дій виконавця і результат цих дій. Алгоритм не містить команди, які можуть сприйматися виконавцем неоднозначно, наприклад, «Узяти одну-дві ложки кави», «Поділити ціле число z на одне з двох заданих цілих чисел x або y» тощо. Алгоритми недопускають ситуації, коли після виконання чергової команди виконавцю неясно, яку команду він повинен виконувати наступною.

Виконуваність алгоритму означає, що алгоритм, призначений для певного виконавця, містить тільки команди, які входять до системи команд цього виконавця. Наприклад, алгоритм для виконавця «Учень четвертого класу» не може містити команду «Знайди корені квадратного рівняння», хоча така команда може бути в алгоритмі, який призначений для виконавця «Учень дев'ятого класу».

Виконавець алгоритму – це істота або неістота, яка може виконувати всі вказівки заданого алгоритму.

Основні характеристики виконавця алгоритму такі:

1. **Середовище виконавця** – умови, у яких може діяти виконавець.
2. **Елементарні дії** – найпростіші дії, які може виконати виконавець.
3. **Система команд виконавця** – сукупність допустимих команд виконавця.
4. **Допустимі команди** – команди, які зрозумілі виконавцю і можуть бути ним виконані. Недопустимі команди – команди, які не можуть бути виконані виконавцем.

Зазначимо, що виконавець повинен лише вміти виконувати кожну команду зі своєї системи команд, і не важливо, розуміє він її, чи ні. Говорять, що виконання алгоритмів виконавцем носить формальний характер: виконавець може не розуміти жодну з команд, може не знати мети виконання алгоритму, і все одно отримає результат. Так, наприклад, верстат з програмним керуванням не розуміє жодної з команд, яку він виконує, але завдяки своїй конструкції успішно виготовляє деталі.

Скінченність алгоритму означає, що його виконання закінчиться після скінченної (можливо, досить великої) кількості кроків і за скінченний час при будь-яких допустимих значеннях початкових даних.

Результативність алгоритму означає, що після закінчення його виконання обов'язково одержуються результати. Результативними вважаються також алгоритми, які визначають, що дану задачу не можна розв'язати, або дана задача не має розв'язків при заданому наборі початкових даних.

Масовість алгоритму означає, що алгоритм може бути застосований до цілого класу однотипних задач, для яких спільними є умова та хід розв'язування, і які відрізняються тільки початковими даними.

До характеристик алгоритму відносять:

Однозначність – Алгоритм повинен бути зрозумілим і недвозначним. Кожен із його кроків (або фаз), а також їхні входи/виходи повинні бути зрозумілими і вести лише до одного значення.

Вхідні дані — алгоритм повинен мати 0 або більше чітко визначених вхідних даних.

Вихідні дані — алгоритм повинен мати 1 або більше чітко визначених виходів і відповідати бажаному результату.

Скінченність — Алгоритми повинні завершитися після скінченної кількості кроків.

Можливість – має бути здійсненним за наявних ресурсів.

Незалежність — алгоритм повинен мати покрокові вказівки, які незалежать від будь-якого програмного коду.

До переваг алгоритмів відносять:

1. Алгоритм легко зрозуміти.
2. Він є покроковим поданням розв'язання заданої задачі.
3. В алгоритмі задача розбивається на дрібніші підзадачі або кроки отже, програмісту простіше його конвертувати у фактичну програму.

Недоліки алгоритмів :

1. Написання алгоритму займає багато часу, а тому це є трудомісткою задачею.
2. Часто розгалуження і цикли важко описати в алгоритмі.

Типи алгоритмів

Лінійні — алгоритми, в яких дії виконуються послідовно без перевірки будь-яких умов.

Розгалужені — алгоритми, в яких передбачені варіанти описів, в залежності від зміни умов (тобто перевірки умов "якщо - тоді " в обов'язковому порядку).

Циклічні — алгоритми, в яких окремі операції або групи операцій виконуються кілька разів. Їх ще називають алгоритмами з повтореннями.

Немає чітко визначених стандартів для написання алгоритмів. Скоріше, це залежить від проблем і ресурсів. Алгоритми ніколи не пишуться для підтримки конкретного програмного коду.

Ми знаємо, що всі мови програмування мають спільні базові конструкції коду, такі як цикли (do, for, while), оператори розгалуження (if-else) тощо. Тому доречно ці загальні конструкції використовувати для написання алгоритму.

Зазвичай, алгоритми пишуть крок за кроком, але це не завжди так. Написання алгоритму є процесом, який виконується після того, як область задачі чітко визначена.

Способи запису алгоритмів:

1. Графічний (наприклад, блок-схеми, послідовність зображень);

2. Мовний:

– звичайна мова (наприклад, кулінарний рецепт);

– псевдокод;

– мова програмування.

Блок-схема складається з окремих геометричних фігур – блоків, які з'єднуються напрямленими лініями. Лінії показують послідовність переходу від одного блока до наступного.

Псевдокод – компактна (найчастіше неформальна) мова опису алгоритмів, що використовує ключові слова імперативних мов програмування, але опускає несуттєві подробиці й специфічний синтаксис.

Загальний вигляд псевдокоду алгоритму може бути такий:

алг <назва алгоритму> (<тип змінних>)
арг <імена змінних-аргументів>
рез <імена змінних-результатів>
поч <тип та імена проміжних змінних>
  <тіло алгоритму>
кін

Записувати алгоритм у наведеному вигляді псевдокоду не обов'язково.

Алгоритмічна мова, конструкції якої однозначно перетворюються на команди комп'ютеру, називається мовою **програмування** (від грец. Programma – розпорядження, припис, оголошення). Текст алгоритму, записаний мовою програмування, називається **програмою**.

Спробуємо вивчити написання алгоритмів на прикладі.

Задача. Створити алгоритм для додавання двох чисел і відображення результату.

Запишемо алгоритм розв'язання у вигляді псевдокода:

```
алг додавання(дійсн a, b, дійсн c)
арг a, b;
рез c;
поч
        увести a, b;
        c := a+b;
        вивести c;
кін
```

Можна даний алгоритм записати так:

Крок 1 – ПОЧАТИ

Крок 2 − оголосити три цілих числа **a** , **b** & **c**

Крок 3 − визначте значення **a** і **b**

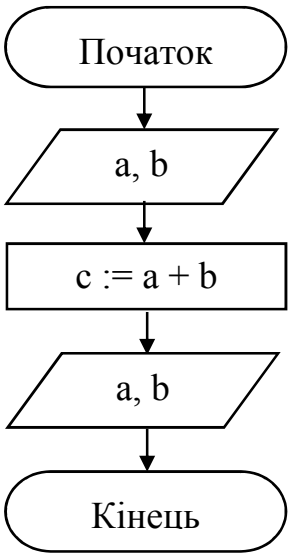
Крок 4 − додати значення **a** і **b**

Крок 5 – зберігти вихідні дані кроку 4 – **c**

Крок 6 − роздрукувати **c**

Крок 7 – СТОП

Наведемо цей же алгоритм поданий блок-схемою:



Напишемо цей алгоритм на C++:

```
#include <iostream>
using namespace std;
int main( ){
  int a, b, c;
  cin >> a >> b;
  c = a + b;
  cout << c << endl ;
  return 0;
}
```

Алгоритми розповідають програмістам, як кодувати програму. Альтернативно, алгоритм можна записати так:

Крок 1 – ПОЧАТИ ДОДАВАТИ

Крок 2 − отримати значення **a** і **b**

Крок 3 − c ← a + b

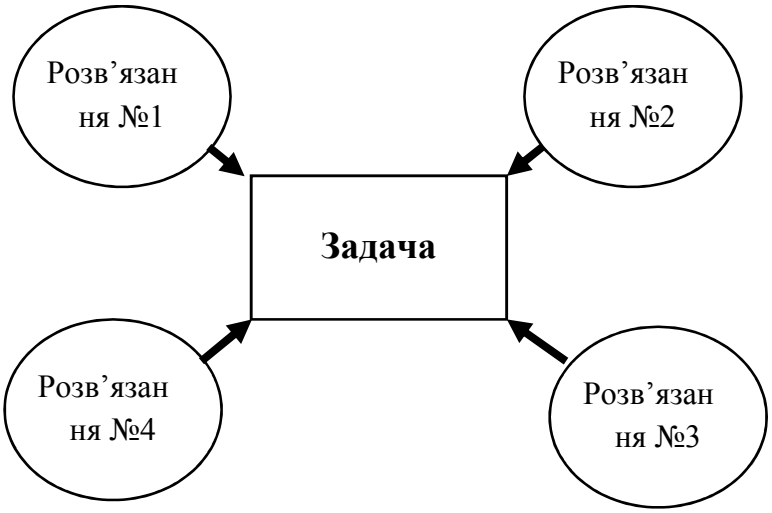
Крок 4 − відображення c

Крок 5 − СТОП

При розробці та аналізі алгоритмів зазвичай для опису алгоритму використовується другий метод. Це полегшує аналітику аналіз алгоритму, ігноруючи всі небажані визначення. При цьому можна спостерігати, які операції використовуються і як протікає процес.

Уведення номерів кроків необов'язкове.

Розробляючи алгоритм певної заданої задачі, проблему можна розв'язати кількома способами.



Отже, для даної задачі можна запропонувати багато алгоритмів розв'язання. Наступним кроком є аналіз запропонованих алгоритмів розв'язання та впровадження найкращого підходящого рішення.

3. Алгоритмічний аналіз

Ефективність алгоритму можна проаналізувати на двох різних етапах: до впровадження та після впровадження. Вони наступні:

- ***Апріорний*** **аналіз** – це теоретичний аналіз алгоритму. Ефективність алгоритму вимірюється, якщо припустити, що всі інші фактори, наприклад, швидкість процесора, є постійними і не впливають на реалізацію.
- ***Апостеріорний*** **аналіз** – це емпіричний аналіз алгоритму. Вибраний алгоритм реалізовано за допомогою мови програмування. Потім він виконується на цільовому комп'ютері. У даному аналізі збираються фактичні статистичні дані, такі як час роботи та необхідний адресний простір.

Ми дізнаємося про *апріорний* аналіз алгоритму. Аналіз алгоритму має справу з виконанням або часом виконання різних задіяних операцій. Час виконання операції можна визначити як кількість комп'ютерних команд, що виконуються за одну операцію.

Припустимо, що **X** є алгоритмом і **n** це розмір вхідних даних, час і простір, які використовує алгоритм X, є двома основними факторами, які визначають ефективність X.

- **Фактор часу** – час, що вимірюється шляхом підрахунку кількості ключових операцій.
- **Коефіцієнт простору** – адресний простір, що вимірюється шляхом підрахунку максимального простору пам'яті, необхідного алгоритму.

Складність алгоритму **f(n)** дає час роботи та/або простір для зберігання, що необхідний алгоритму в термінах, де **n** - розмір вхідних даних.

Означення. Об'ємна складність алгоритму – це обсяг пам'яті, необхідний алгоритму в його життєвому циклі.

Простір, що необхідний для алгоритму, дорівнює сумі двох наступних компонентів:

- Фіксована частина - це адресний простір, необхідний для зберігання певних даних і змінних, які не залежать від складності задачі. Наприклад, прості змінні та константи, розмір програми тощо.
- Змінна частина — це простір, необхідний для змінних, розмір якого залежить від складності задачі. Наприклад, динамічне виділення пам'яті, простір стека рекурсії тощо.

Складність простору T(P) будь-якого алгоритму P дорівнює T(P) = C + SP(I), де C — фіксована частина, а SP(I) — змінна частина алгоритму, яка залежить від характеристики екземпляра I. Це простий приклад, який намагається пояснити концепцію :

Алгоритм: SUM(A, B)

Крок 1 - СТАРТ

Крок 2 - C ← A + B + 10

Крок 3 - ЗУПИНИТИСЯ

Тут ми маємо три змінні A, B і C і одну константу. Отже , T(P) = 1 + 3. Тепер простір залежить від типів даних даних змінних і константних типів, і він буде відповідно помножений на певні величини..

4. Які види задач розв'язують алгоритми?

Практичні застосування алгоритмів зустрічаються повсюдно і включають такі приклади:

- Проект «Геном людини» має значний прогрес у досягненні цілей ідентифікації всіх 100 000 генів ДНК людини, визначення послідовностей 3 мільярдів пар хімічних елементів, які складають ДНК людини, збереження цієї інформації в базах даних та розробки інструментів для аналізу даних. Кожен із цих кроків вимагає складних алгоритмів. Він дає економію в часі, як людини, так

і машини, і в грошах, оскільки більше інформації можна отримати з лабораторних методів.

- Інтернет дозволяє людям у всьому світі швидко отримувати доступ до великих обсягів інформації. За допомогою розумних алгоритмів сайти в Інтернеті можуть керувати цим великим обсягом даних. Приклади проблем, які суттєво використовують алгоритми, включають пошук хороших маршрутів, за якими будуть переміщатися дані, і використання пошукової системи для швидкого пошуку сторінок, на яких міститься конкретна інформація.

Електронна комерція дозволяє обговорювати товари та послуги та обмінюватися ними в електронному вигляді, і це залежить від конфіденційності особистої інформації, такої як номери кредитних карток, паролі та виписки з банку. Основні технології, що використовуються в електронній комерції, включають криптографію з відкритим ключем і цифрові підписи, які базуються на числових алгоритмах і теорії чисел.

- Виробничі та інші комерційні підприємства часто мають потребу розподіляти дефіцитні ресурси найбільш вигідним чином. Нафтова компанія може захотіти знати, де розмістити свої свердловини, щоб максимізувати очікуваний прибуток. Політичний кандидат може побажати визначити, куди витрачати гроші, купуючи рекламу кампанії, щоб максимізувати шанси на перемогу на виборах. Авіакомпанія може зажадати призначити екіпаж на рейси найдешевшим можливим способом, переконавшись, що кожен рейс охоплюється і що урядові постанови щодо планування екіпажу виконуються. Провайдер Інтернет-послуг може забажати визначити, де розмістити додаткові ресурси, щоб обслуговувати своїх клієнтів більш ефективно.

- Нам дається дорожня карта, на якій позначено відстань між кожною парою сусідніх перехресть, і ми хочемо визначити найкоротший маршрут від одного перехрестя до іншого. Кількість можливих маршрутів може бути величезною, навіть якщо ми заборонимо маршрути, які перетинаються між собою. Як вибрати, який із усіх можливих маршрутів найкоротший? Тут ми моделюємо дорожню карту і хочемо знайти найкоротший шлях від однієї вершини до іншої в графі.

Ці списки завдань далеко не вичерпні, але мають дві характеристики, спільні для багатьох цікавих алгоритмічних задач:

1. У них є багато варіантів рішення, переважна більшість з яких не розв’язує проблему. Знайти той, який підходить, або той, який є «найкращим», може бути досить складним завданням.

2. Вони мають практичне застосування.

З проблем у наведеному вище списку найпростішими прикладами є пошук найкоротшого шляху. Транспортна фірма, така як автотранспортна або

залізнична компанія, має фінансовий інтерес у пошуку найкоротших шляхів через автомобільну або залізничну мережу, оскільки використання коротших шляхів приводить до зниження витрат на оплату праці та палива. Або вузлу маршрутизації в Інтернеті може знадобитися знайти найкоротший шлях через мережу, щоб швидко маршрутизувати повідомлення. Або людина, яка бажає проїхати з Ніжина до Львова, може захотіти знайти маршрути проїзду з відповідного веб-сайту, чи вона може використовувати свій GPS навігатор під час руху.

5. Концепція структур даних

У Computer Science під **структурою даних розуміють спосіб зберігання даних, що забезпечує її ефективне використання.**

Означення. Структури даних – це сукупність елементів даних і відношень між ними.

Під елементами даних розуміють як прості дані, так і структури даних. А під відношеннями між даними - функціональні зв'язки між ними і покажчики на те, де знаходяться ці дані.

Означення. Елемент відношень – це сукупність всіх зв'язків елемента з іншими елементами даних, що розглядається в структурі.

Структури даних і алгоритми є тим, з чого конструюються програми. Враховуючи знання з архітектури обчислювальних систем, можемо говорити, що комп'ютер сам складається з структур даних і алгоритмів. Вбудовані структури даних – це регістри й слова пам'яті, де зберігаються величини. Закладені в конструкцію апаратури алгоритми – це реалізація в електронних логічних схемах жорстких правил, за допомогою яких поміщені в пам'ять дані інтерпретуються як команди, що підлягають виконанню. Тому в основі роботи будь - якого комп'ютера лежить вміння оперувати лише з одним видом даних – з окремими бітами. Працює ж з цими даними комп'ютер тільки у відповідності з незмінним набором алгоритмів, які визначаються системою команд центрального процесора.

Задачі, які розв'язуються за допомогою комп'ютера, рідко представляються мовою бітів. За звичай, дані мають форму чисел, текстів, символів і більш складних структур (послідовностей, списків і дерев).

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

Структури даних, які застосовуються в алгоритмах, можуть бути досить складними. Вибір правильного представлення даних часто служить ключем до

вдалого програмування і може в більшій мірі впливати на продуктивність програми, ніж деталі реалізації використовуваного алгоритму. Але, мабуть, ніколи не появиться загальна теорія вибору структур даних, у кожному конкретному випадку потрібно підходити до цього творчо. Незалежно від змісту і складності будь-які дані в пам'яті комп'ютера представляються послідовністю двійкових розрядів, а їх значеннями є відповідні двійкові числа. Дані, які розглядаються у вигляді послідовності бітів, мають дуже просту організацію, тобто є слабо структурованими. Більш крупні й змістовніші, ніж біт, елементи отримуються на основі поняття „структури даних".

Поняття „фізична структура даних" відображає спосіб фізичного представлення даних в пам'яті машини і називається ще структурою зберігання.

Розгляд структури даних без урахування її подання в машинній пам'яті називається абстрактною або **логічною структурою**. У загальному випадку між логічною і відповідною їй фізичною структурами існує відмінність, міра якої залежить від самої структури і особливостей того середовища, в якому вона повинна бути відображена.

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

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

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

Операції над структурами даних

Над структурами даних можуть виконуватися такі загальні операції: створення, знищення, вибір (доступ), поновлення.

Операція створення – це виділення пам'яті для зберігання структури даних. Пам'ять може резервуватися в процесі виконання програми або на етапі компіляції. У деяких мовах для структурованих даних, які конструюються

програмістом, операція створення включає в себе також встановлення початкових значень параметрів структури (ініціалізація).

Незалежно від мови програмування, що використовується, наявні в програмі структури даних не появляються „з нічого”, а явно або неявно оголошуються операторами створення.

Операція знищення структур даних протилежна до операції створення – вона звільнює пам’ять, яку займала структура, для подальшого використання. Операція знищення дозволяє ефективно використовувати пам’ять.

Операція вибору використовується програмістами для доступу до даних в самій структурі. Форма операції доступу залежить від типу структури даних, до якої здійснюється звертання. Метод доступу – одна з найбільш важливих властивостей структур, тому що вона має безпосереднє відношення до вибору конкретної структури даних.

Операція поновлення дозволяє змінити значення даних в структурі даних. Операцією поновлення є, наприклад, операція присвоєння, або, більш складна форма – передача параметрів.

Зазначені чотири операції обов’язкові для всіх структур і типів даних. Крім них для кожної структури даних можуть бути визначені специфічні операції, які працюють тільки з даними конкретного типу чи структури.

6. Класифікація структур даних

Поняття класифікації структур даних, як і інших класифікацій, є дещо умовним. Усі структури ділять на прості (базові) структури даних та інтегровані (структуровані).

Простими називаються такі структури даних, які не можуть бути розділені на складові частини більші, ніж біти. З точки зору фізичної структури важливим є те, що у даній машинній архітектурі, системі програмування завжди можна наперед сказати, яким буде розмір вказаного простого типу і яка структура його розміщення в пам’яті. З логічної точки зору, прості дані є неподільними одиницями.

Інтегрованими називаються структури даних, складовими частинами яких є інші структури даних – прості або інтегровані. Інтегровані структури даних створюються програмістом із використанням засобів інтеграції даних, які надаються мовами програмування.

У залежності від наявності чи відсутності явно заданих зв’язків між елементами даних потрібно розрізняти **незв’язні структури** і **зв’язні структури**.

Досить важлива ознака структури даних – її змінність – зміна кількості елементів і/або зв'язків між елементами структури. За ознакою змінності розрізняють структури **статичні, напівстатичні, динамічні**.

Ще одна важлива ознака структури даних – характер впорядкованості її елементів. За цією ознакою структури можна поділити на лінійні й нелінійні структури.

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

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

За видом пам'яті, що використовується для збереження даних,існує поділ на структури даних для оперативної і для зовнішньої пам'яті.

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

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

🔔 Контрольні запитання:

1. Що таке алгоритм?
2. Які відомі способи зображення алгоритмів?
3. Які є властивості алгоритму? Охарактеризуйте їх.
4. Сформулюйте означення виконавця алгоритму.
5. Які основні характеристики виконавця алгоритму? Охарактеризуйте їх.
6. Що таке фактор часу?
7. Що таке об'ємна складність алгоритму?
8. Дайте означення структури даних.
9. Які операції над структурами даних виділяють?
10. Дайте класифікацію структури даних.

Тема №2. ПРОСТІ СТРУКТУРИ ДАНИХ

План лекції:

1. Арифметичні типи
2. Покажчики
3. Статичні структури даних

Джерела:

[1, §1.2], [4, §3.1, 3.2], [11, §1.5].

1. Арифметичні типи

Стандартні типи даних часто називають арифметичними, оскільки їх можна використовувати в арифметичних операціях. Для опису основних типів в C++ визначено ключові слова: **int, long, char, bool, float, double** тощо.

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

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

Значенням символьного типу **char** є символи з деякої наперед визначеної множини. У більшості сучасних персональних комп'ютерів цією множиною є ASCII. Ця множина складається з 256 різних символів, які впорядковані певним чином і містить символи великих і малих букв, цифр і інших символів, включаючи спеціальних керуючих символів. Значення символьного типу займає в пам'яті 1 байт. Іншою вживаною множиною для подання символьних даних є Unicode. У Unicode кожний символ кодується двома або чотирма байтами.

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

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

Перелічувальний тип

При написанні програм часто виникає потреба визначити кілька іменованих констант, для яких необхідно, щоб усі вони мали різні значення. Для цього зручно використовувати перелічувальний тип **enum**. Усі можливі значення даного типу задаються списком цілочисельних констант:

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

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

Діапазон значень перечислень визначається кількістю біт, які необхідні для подання всіх його значень. Будь-яке значення цілого типу можна явно привести до перелічувального типу, але при виході за межі його діапазону результат буде невизначеним. При відсутності ініціалізації перша константа набуває нульове значення, а кожній наступній присвоюється на одиницю більше значення від попереднього.

Наприклад:

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

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

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

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

```
#include <iostream>

using namespace std;
//визначення перерахованого типу
enum floor {Parking, Market, Boutiques, Spa, Club,
Restaurant };
int main()
{
  setlocale(LC_ALL,"ukr");
  int F; //вибір поверху користувачеи
  bool exit = true;//вийти чи продовжити подорож
  cout <<" \n \t _____\n\n";
  cout << "\t Ласкаво просимо до торгового центру !!! \n";
  cout << "\t Відвідайте за допомогою ліфту всі поверхи! \n \n";
  cout << "\t _____ \n \n";
  error: while(exit) // поки exit рівний true
  {
    cout << " \n \t Натисніть кнопку із номером поверху від 0 до 6: ";
  cin >> F;
  switch(F)
  {
    case(Parking):
      cout << "\n Ви на паркінгу!!!" << endl;
      break;
    case(Market):
      cout << "\n Ви на першому поверсі!";
      cout << "\n Відвідайте наш магазин \n \n";
      break;
    case(Boutiques):
      cout << "\n Ви на другому поверсі!";
      cout << "\n Тут знаходиться магазин побутової техніки. \n\n";
      break;
    case(Spa):
      cout << "\n Ви на третьому поверсі!";
      cout << "\n Тут знаходиться спа-салон. \n \n";
      break;
    case(Club):
      cout << "\n Ви на четвертому поверсі!";
```

```
      cout << "\n Тут знаходиться наш більярдний клуб! \n \n";
      break;
    case(Restaurant):
      cout << "\n Ви на пятому поверсі!";
      cout << "\n Тут ви можете відвідати наш ресторан! \n \n";
      break;
    default: cout << " \n \n \n Помилка! В будівлі тільки 6 поверхів! \n
\n";
          goto error;
  }
  cout << "Якщо хочете вийти на цьому поверсі, то, натисніть 0. \n";
  cout << "Якщо хочете продовжити подорож - натисніть 1: ";
  cin >> exit;
}
  return 0;
}
```

2. Покажчики

Тип «покажчик» являє собою адресу комірки пам’яті (у більшості сучасних обчислювальних систем розмір комірки – мінімальної адресованої одиниці пам’яті – складає один байт). При програмуванні на низькому рівні робота з адресами складає значну частину програм. При вирішенні прикладних задач з використанням мов високого рівня найчастіші випадки, коли програміст може використовувати покажчики, такі:

1. При необхідності представити одну й ту ж ділянку пам’яті, а значить, одні й ті ж фізичні дані, як дані різної логічної структури.

2. При роботі з динамічними структурами даних.

Покажчики використовують при роботі із наступними структурами даних:

• робота із масивами та символьними змінними;

• обробка даних у динамічній пам’яті;

• створення динамічних структур;

• передача у функцію даних для їх опрацювання.

У С++ розрізняють три види покажчиків – покажчик на об’єкт, на функцію і на пустий тип, які відрізняються властивостями і набором допустимих операцій. Покажчик не є самостійним типом, він завжди зв’язаний з якимось іншим типом.

У програмах на мовах високого рівня покажчики можуть бути типізованими і нетипізованими. При оголошені типізованого покажчика визначається й тип об’єкту в пам’яті, який адресується цим покажчиком.

Хоча фізична структура адреси не залежить від типу й значення даних, які зберігаються за цією адресою, компілятор вважає покажчики на різні типи такими, що мають різний тип. Таким чином, коли йде мова про типізовані покажчики, правильніше говорити не про єдиний тип даних «покажчик», а про цілу сім'ю типів: «покажчик на ціле», «покажчик на символ» тощо.

Нетипізований покажчик використовується для представлення адреси, за якою містяться дані невідомого типу. Робота з нетипізованими покажчиками суттєво обмежена, вони можуть використовуватися тільки для збереження адреси, звертатися за такою адресою не можна.

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

До свого першого використання усі покажчики повинні бути оголошені згідно наступного синтаксису:

<базовий тип> *<ім'я вказівника>

тут: <базовий тип> - тип даних програми, адреси яких буде зберігати даний вказівник; ∗ є ознакою того, що наступна змінна є вказівником;

<ім'я вказівника> - ідентифікатор змінної. Наприклад:

int *i, j, k;

char *th;

double *f;

Тут: i - адреса ділянки пам'яті, в якій розташоване ціле число; th – адреса даних типу char; вказівник f призначений для збереження даних типу double.

Знак ∗ в оголошенні вказівника обов'язково повинен стояти перед кожною змінною – вказівником. Змінні без ∗ будуть вважатися звичайними змінними відповідного типу.

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

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

m = &k

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

* - розадресація: отримання адреси змінної. Операндом операції * є вказівник (адреса ділянки пам’яті), а результатом - значення об’єкту, адресу якого містить вказівник. Операція є зворотньою до операції взяття адреси &.

Результатом розадресації є тип даних, який був заданий базовим при оголошенні вказівника. Звернення через вказівник може бути використане всюди, де синтаксично дозволяється записати об’єкт даного типу. Наприклад, якщо в програмі оголошено змінну та вказівник, що зберігає адресу цієї змінної:

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

то в результаті присвоєння:

```
*vz=3.123;
```

змінна z прийме значення 3.123.

Перед використанням покажчика у програмі його обов’язково необхідно ініціювати, іншими словами, необхідно присвоїти адресу якого-небудь даного, інакше можуть бути непередбачені результати.

За старшинством операція взяття адреси та операція розадресації поступаються тільки операції звернення до функції. Переваги використання покажчиків безпосередньо пов’язані із можливістю виконання над ними наступних операцій:

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

Перелічені операції ще називають **адресною арифметикою**.

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

Присвоєння покажчиків

Вказівнику можна присвоїти значення адреси, яка задається іншим вказівником чи обчислюється за допомогою виразу, що знаходиться справа від операції присвоєння. Операція присвоєння вимагає, щоби базовий тип вказівника та виразу, що присвоюється, були однакові.

Наприклад:

int *iz;

int *ik;

A = 254;

. . . . . .

iz = &A;

ik = iz;

У результаті виконання наведеного фрагменту програми вказівники iz та ik набудуть значення – адресу змінної A. Значення змінної iz буде рівне значенню змінної A .

Оскільки покажчики — це спеціальні змінні, то в операціях з іншими покажчиками вони можуть використовуватися без символа «*», тобто без розкриття посилання, наприклад:

```
float *pt1, *pt2, x=15, m[5];
pt1 = &x;
pt2 = pt1;
pt1 = m;   //pt1 = &m[0];
```

де m — ім’я масиву, що розглядається як спеціальний покажчик-константа.

Приклад. Ілюстративна програма з використанням покажчиків.

```
#include <iostream>

using namespace std;

int main ( )
{
  setlocale(LC_ALL,"ukr");
  int x = 10;
  int *px (&x);  // int *px = &x;
  cout << "x =" << x << endl;
  cout << "*px =" << *px << endl;
  x *= 2;      //x=x*2;
  cout << "Нове значення *px = " << *px << endl;*
  px += 2;     // *px=*px + 2;
  cout << "Результат *px, x = " << x << endl;
  return 0;
}
```

Додавання та віднімання до вказівника цілого числа.

Загальна формула обчислення значення вказівника при додаванні/відніманні цілого числа N має наступний вигляд:

нова < адреса стара> = < адреса> +/- N * < розмір базового типу>;

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

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

При необхідності розмір базового типу може бути визначений за допомогою функції

sizeof (<базовий тип>)

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

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

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

```
nt *iz;
. . . . .
iz++
```

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

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

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

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

3. Статичні структури даних

Статичні структури відносяться до класу структур, які являють собою структуровану множину примітивних, базових структур. Оскільки статичні структури відрізняються відсутністю змінюваності, пам'ять для них виділяється автоматично – як правило, на етапі компіляції, або при виконанні – в момент активізації того програмного блоку, в якому вони описані. Ряд мов програмування допускають розміщення статичних структур в пам'яті на етапі виконання за явною вимогою програміста, але й у цьому випадку обсяг виділеної пам'яті залишається незмінним до знищення структури. Виділення пам'яті на етапі компіляції є такою зручною властивістю статичних структур, що у ряді задач програмісти використовують їх навіть для подання об'єктів, які мають властивість змінюваності. Наприклад, коли розмір масиву невідомий наперед, для нього резервується максимально можливий розмір.

Статичні структури в мовах програмування зв'язані із структурованими типами. Останні в мовах програмування є тими засобами інтеграції, які дозволяють будувати структури даних будь-якої складності. До таких типів відносяться масиви, структури та їхні похідні типи.

Масиви

Логічно масив об'єднує елементи одного типу даних, тобто належить до однорідного типу даних. Більше формально його можна визначити як впорядковану сукупність елементів деякого типу, які адресуються за допомогою одного або кількох індексів.

Масиви можна класифікувати за кількістю розмірностей масиву: на одновимірні масиви (вектори), двохвимірні (матриці) і багатовимірні (трьох, чотирьох і більше).

Логічно масив – це така структура даних, яка характеризується:

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

Фізична структура масиву – це спосіб розміщення елементів масиву в пам'яті комп'ютера. Під елемент масиву виділяється кількість байт пам'яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам'яті для зберігання масиву.

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

Адресою масиву є адреса першого байту початкового компоненту масиву.

Індексація масивів в C/C++ обов'язково починається з нуля.

Опис n - мірного масиву виконується згідно наступного синтаксису:

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

Наприклад:

```
int a[100] [4];
double c[10][4][7];
/* Опис двовимірного масив з одночасною його ініціалізацією. */
int s[2][2]={3,6,-7,3,9,23,76,1,6};
// Двовимірний масив рядкових змінних.
string str3[7][8];
/* Двовимірний масив рядкових змінних з одночасною його ініціалізацією. */
char ch[3][2] = {‘ W’, ‘ 0’, ‘ R’, ‘ L’, ‘ D’,’ ! ‘ }
```

Перед збереженням даних необхідно вказати розмір масиву.

Нехай розмір масиву дорівнює n.

Час доступу: O(1) [Це можливо, оскільки елементи зберігаються в суміжних місцях]

Час пошуку:

- O(n) для послідовного пошуку;

  O(log n) для двійкового пошуку (якщо масив відсортований).

Час вставки: O(n). Найгірший випадок відбувається під час вставки на початок масиву і вимагає зміщення всіх елементів.

Час видалення: O(n). Найгірший випадок відбувається під час видалення з початку масиву і вимагає зміщення всіх елементів.

Приклад 1. Потрібно зберегти оцінки всіх студентів групи. Можемо використовувати для їх зберігання масив. Це допомагає зменшити використання кількості змінних, оскільки нам не потрібно створювати окрему змінну для оцінок кожного предмета. До всіх оцінок можна отримати доступ, просто обходячи масив.

```
#include <iostream>

using namespace std;

int main()
{
  setlocale(LC_ALL,"ukr");
  int n, s, a[10];
  float avereage_score;
  cout << "Уведіть кількість оцінок студентів: ";
  cin >> n;
  cout << "Уведіть оцінки студентів: ";
  for (int i = 0; i < n; i++)
    cin >> a[i];
  s = 0;
  for (int i = 0; i < n; i++)
    s += a[i];
  avereage_score = s / n;
  cout << "Середній бал оцінок = " << avereage_score;

  return 0;
}
```

Якщо масив оголошено без задання розміру, але ініціалізовано списком, то розмір обчислюється автоматично як результат підрахунку елементів списку.

Якщо розмір вказується явно, то кількість елементів не повинна перевищувати заданий розмір. Якщо в списку недостатньо елементів, інші елементи ініціалізуються нулями. Першими ініціалізуються елементи з найменшими індексами.

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

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

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

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

У результаті буде ініціалізована наступна матриця:

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

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

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

Так, при оголошені, наприклад, двовимірного масиву

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

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

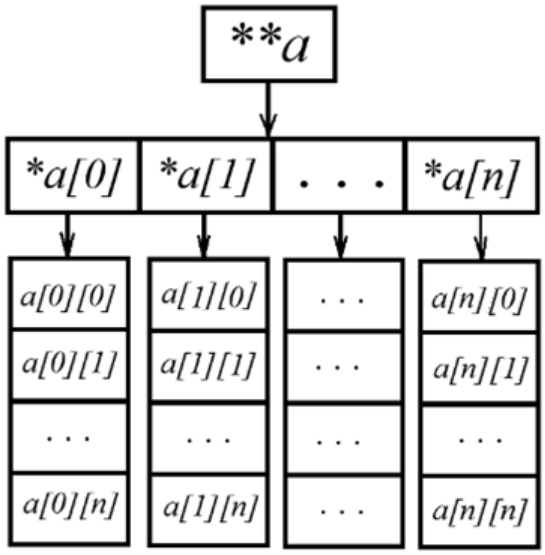


Рис 2.1. Двомірний масив як масив вказівників

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

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

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

Аналогічно можна розглянути, наприклад, тривимірний масив:

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

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

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

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

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

Приклад 2. Визначити найбільший та найменший елементи матриці, їх індекси. У функції для пошуку максимального елементу використати явну форму оголошення параметру як матриці. У функції для пошуку мінімального елементу параметр Mass оголосити як вказівник на масив N дійсних чисел.

```
#include <algorithm>
#include <iostream>

#define N 5

using namespace std;

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

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

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

//У функції використано явне оголошення параметра Mass як матриці

```
double Max_Elen(double Mass[][N],int *m_r, int *m_c)
{
 double max = Mass[0][0];
 *m_r = 0;
 *m_c = 0;
 for(int i = 0; i < N; i++)
  {
    for(int j = 0; j < N; j++)
     {
      if( Mass[i][j] > max)
      {
       max = Mass[i][j];
       *m_r = i;
       *m_c=j;
      }
     }
  }
 return max;
}
```

//У функції параметр Mass оголошено як вказівник на масив N дійсних чисел

```
double Min_Elen(double (*Mass)[N],int *m_r, int *m_c)
{
 double min = **Mass;
 double *elem = *Mass;
 *m_r = 0;
 *m_c = 0;
 for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++,elem++)
   if(*elem<min)
```

```
        {
         min = *elem;
         *m_r = i;
         *m_c = j;
        }
     return min;
    }
    int main()
    {
     setlocale(LC_ALL,"ukr");
     double Mass[N][N];

     Init_Mass(Mass);
     Show_Mass(Mass);

     double max_elem;
     double min_elem;
     int m_r = 0, m_c = 0;

     max_elem = Max_Elen( Mass, &m_r, &m_c);
     cout << "Масимальний елемент= " << max_elem << "
Позиція " << m_c+1 << " " << m_r+1 << endl;

     min_elem = Min_Elen( Mass, &m_r, &m_c);
     cout << "Мінімальний елемент= " << min_elem << "
Позиція " << m_c+1 << " " << m_r+1 << endl;

     return 0;
    }
```

Розріджені масиви

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

Розрізняють два типи розріджених масивів:

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

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

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

На практиці для роботи з розрідженим масивом розробляються функції:

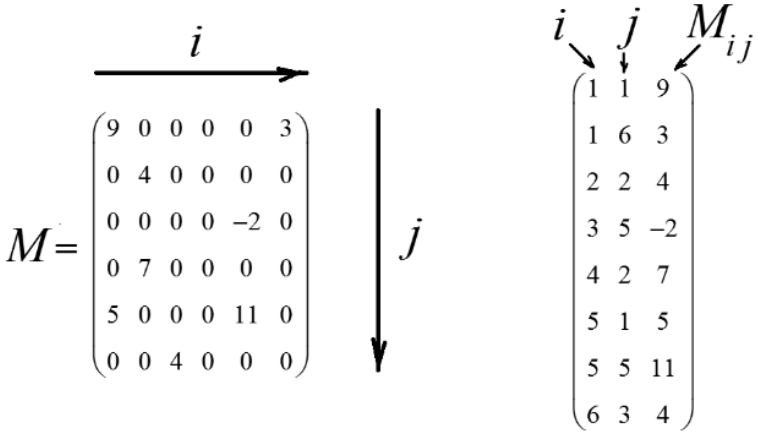
✓ для перетворення індексів масиву в індекс вектора;

✓ для отримання значення елемента масиву з його упакованого подання за індексами;

✓ для запису значення елемента масиву в його упаковане подання за індексами.

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

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



Пам'ять, необхідна для збереження розріджених матриць, складається з двох частин: основної пам'яті, у якій розміщуються числові значення елементів та додаткової пам'яті, де зберігаються індекси та інша інформація, що необхідна для формування структури матриць і яка забезпечує доступ до числових значень

їх елементів. Способи зберігання і використання даних, що зберігаються в основній і додатковій пам’яті, досить різноманітні і визначаються, головним чином, обраним методом.

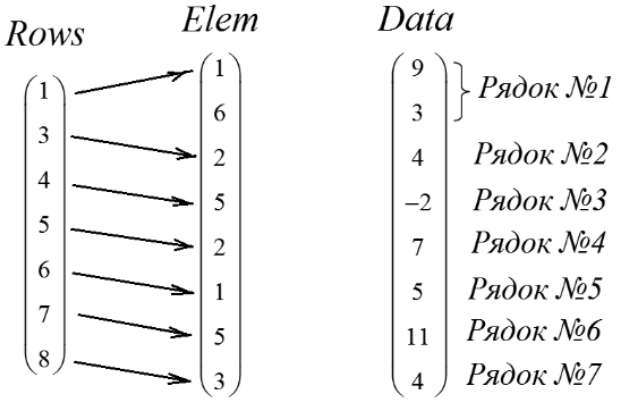
Якщо елементами масиву є цілі числі, то найпростіший підхід полягає у створенні двовимірного масиву розмірності n ∗ 3, де n - кількість ненульових елементів вихідної розрідженої матриці. Перші два елементи кожного рядка містять індекси рядка та стовпця ненульового елементу, а третій – значення самого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

Якщо ненульові елементи не цілі числа, то замість двовимірного масиву розмірності n ∗ 3 використовують два масиви: двомірний цілочисельний масив індексів ненульових елементів та одновимірний дійсний масив для збереження значень елементів. Алгорими обробки такого типу масивів принципових проблем не викликають. Можна дещо зменшити об’єм необхідної для збереження розрідженого масиву пам’яті, якщо створити три масиви наступної структури:

• Rows - вектор, довжина якого рівна кількості рядків вихідної розрідженої матриці;

• Elem - вектор,кількість елементів якого рівна кількості ненульових елементів вихідної розрідженої матриці;

• Data - вектор зі значеннями ненульових елементів вихідної розрідженої матриці.



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

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

Дане подання масиву скорочує вимоги до об’єму пам’яті більш ніж удвічі. Спосіб послідовного розподілу має й ту перевагу, що операції над матрицями можуть бути виконані швидше, ніж це можливо при поданні у вигляді послідовного масиву, особливо якщо розмір матриці великий.

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

Метод зв’язаних структур переводить структуру даних, що представляється, в інший розділ класифікації. При тому, що логічна структура даних залишається статичною, фізична структура стає динамічною.

Множини

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

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

Над множинами визначені наступні специфічні операції:

1. Об’єднання множин. Результатом є множина, що містить елементи початкових множин.

2. Переріз множин. Результатом є множина, що містить спільні елементи початкових множин.

3. Різниця множин. Результатом є множина, яка містить елементи першої множини, які не входять в другу множину.

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

5. Перевірка на входження елемента в множину. Результатом цієї операції є значення логічного типу, що вказує чи входить елемент в множину.

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

• генерація впорядкованої за зростанням множини (GenerationSet) ;

• вивід множини на екран(ShowSet);

• видалення елементу множини із перевіркою, чи такий елемент у множині присутній(DelSet);

• вставка елементу у множину із перевіркою, чи такий елемент у множині вже присутній (InsertSet);

• виконання операції об’єднання двох множин(UnionSet).

```
#include <iostream>

using namespace std;

int max_SIZE=1000; //Максимальна кількість елементів
множини

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

//Функція генерування вихідної матриці.
void GenerationSet(int *Set, int SIZE)
{
 int i = 0;
 Set[i] = 2000 - rand() % 1000;
 int temp;
```

```
 while( i < SIZE )
  {
    temp = 1000 - rand() % 2000;
    for (int j = 0; j < SIZE; j++)
     if (temp == Set[j]) continue;
    i++;
    Set[i]=temp;
  }
 SortSet( Set,SIZE);
}

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

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

```
 if(Set[SIZE-1] == x) return 0;
 for (int i=0; i<SIZE-1; i++)
  {
    if(Set[i]==x) return 0;
    if(Set[i]<x && Set[i+1]>x)
     {
      temp=i;
      break;
     }
  }
 for(int i = SIZE; i > temp; i--)
  Set[i] = Set[i - 1];
 Set[temp + 1] = x;
 SIZE++;
 return 1;
}

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

//Функція обєднання двох множин.
void UnionSet(int *Set, int SIZE, int *Set_1, int SIZE_1,
int *Set_0, int &SIZE_0)
{
```

```
 SIZE_0 = SIZE;
 for (int i = 0; i < SIZE; i++)
  Set_0[i] = Set[i];
 for (int i = 0; i < SIZE_1; i++)
  InsertSet(Set_0, Set_1[i], SIZE_0);
}

int main()
{
  setlocale(LC_ALL,"ukr");
//Блок оголошень та ініціалізації
  int SIZE = 0;
  int SIZE_1 = 0;
  int SIZE_0 = 0;
  int x;
  int *Set = new int[max_SIZE];
  int *Set_1 = new int[max_SIZE];
  int *Set_0 = new int[max_SIZE];

  cout << "Введіть розмір множини, яка має бути
згенерована" << endl;
  cin >> SIZE;
  GenerationSet(Set, SIZE);
  ShowSet(Set, SIZE);

  cout << "Введіть значення елементу для вставки у
множину" << endl;
  cin >> x;
  if (InsertSet(Set,x,SIZE) == 0) cout << "Такий елемент
у множині вже присутній" << endl;
  ShowSet(Set, SIZE);
  cout << "Введіть значення елементу для видалення із
множини" << endl;
  cin >> x;
  if( DelSet(Set,x,SIZE) == 0)cout << "Такий елемент у
множині відсутній" << endl;
  ShowSet(Set, SIZE);

  cout << "Введіть розмір другої множини, яка має бути
згенерована" << endl;
```

```
  cin >> SIZE_1;
  GenerationSet(Set_1, SIZE_1);
  ShowSet(Set_1, SIZE_1);

  cout << "Результат об\'єднання двох множин" << endl;
  UnionSet(Set, SIZE, Set_1, SIZE_1, Set_0, SIZE_0);
  ShowSet(Set_0, SIZE_0);

  return 0;
}
```

Можна використати стандартну бібліотеку шаблонів STL. Для роботи з множинами потрібно підключати відповідну бібліотеку командою

#include <set>;

У С++ розрізняють упорядковані множини **set** та мультимножини **multiset**.

Множини **set** містять тільки унікальні елементи, а мультимножини можуть містити дублікати.

Для реалізації множин можна використати і бібліотеку **<list>**, у якій вже реалізовані операції об'єднання, перерізу та різниці.

Структури

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

Означення. Структура – це іменована упорядкована група логічно зв'язаних змінних, що зберігаються в одному місці.

Загальний формат визначення структури виглядає так:

```
struct ім'я_типу_структури {
  тип ім'я_елемента1;
  тип ім'я_елемента2;
  тип ім'я_елемента3;
  . . .
  тип ім'я_елементаN;
} структурні_змінні;
```

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

Звернення до окремих полів структури замінюються на їхні адреси ще на етапі компіляції. Найважливішою операцією для структури є операція доступу

до вибраного поля структури – операція кваліфікації. Загальний формат доступу записується так:

```
ім’я_структурної_змінної.ім’я_члена
```

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

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

Об’єднання

Об’єднання представляють собою частковий випадок структури, усі поля якої розміщуються за однією ж і тою ж адресою. Формат опису такий же, як і в структури. Оголошення об’єднання подібне оголошенню структури.

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

Оголошення об’єднання починається із ключового слова **union**.

Довжина об’єднання рівна найбільшій із довжин його полів. У кожен момент часу в змінній типу об’єднання зберігається тільки одне значення, і відповідальність за його правильне використання лягає на програміста.

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

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

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

🔔 Контрольні запитання:

1. Які типи називають арифметичними?
2. Що являє собою перелічувальний тип?
3. Які є базові операції для роботи з покажчиками?
4. Що таке розріджений масив?
5. У чому суть методу послідовного розміщення?

Тема №3. СКЛАДНІСТЬ АЛГОРИТМІВ

План лекції:

1. Алгоритм та оцінка його складності
2. Алгоритм та його аналіз
3. Час виконання (running time)
4. Найкращий, найгірший та середній час виконання
5. Асимптотична оцінка складності алгоритмів. O – символіка
6. Класи алгоритмів

Джерела:

[1, §3.1.5], [2, Розділ 7], [3, Глава 9], [4, Глава 2], [5, Chapter 2], [6, Chapter 2], [7, §1.2.10, 1.2.11], [10, Chapter 2], [11, Chapter 2]

1. Алгоритм та оцінка його складності

Для того, щоб зрозуміти, чи підходить алгоритм для розв'язання певної задачі проводять його аналіз.

Алгоритмом називають набір інструкцій, який описує порядок дій виконавця, щоб розв'язати задачу за скінченну кількість дій (за скінченний час).

Як бачимо, алгоритм – це механізм, який не тільки повинен гарантувати те, що вирішення колись буде знайдене, але й те, що буде знайдене саме оптимальне, тобто найкраще вирішення. Крім того, алгоритм повинен мати наступні п'ять якостей:

1. **Обмеженість в часі** – робота алгоритму обов'язково повинна завершитись через деякий розумний період часу.
2. **Правильність** – алгоритм повинен знаходити правильне рішення.
3. **Детермінованість** – скільки б разів не виконувався алгоритм з однаковими вхідними даними, результат повинен бути однаковим.
4. **Скінченність** – опис алгоритму повинен мати скінчену кількість кроків.
5. **Однозначність** – кожний крок алгоритму повинен інтерпретуватися однозначно.

Охарактеризуємо поняття алгоритму не формально, а дескриптивно за допомогою таблиці 3.1.

Як видно з таблиці, евристика – це пряма протилежність класичному алгоритму, бо вона не дає ніяких гарантій на те, що рішення буде знайдене, так само, як і на те, що воно буде оптимальним. Між ними є два перехідні стани – приблизні алгоритми (рішення гарантоване, але його оптимальність – ні) і ймовірнісні алгоритми (рішення не гарантоване, але якщо воно буде знайдене, то обов'язково буде оптимальним).

Таблиця 3.1. Дескриптивний опис алгоритму

		Розв'язання гарантоване	
		Так	Ні
Чи обов'язкове оптимальне розв'язання	Так	Алгоритм	Ймовірнісний алгоритм
	Ні	Приблизний алгоритм	Евристичний алгоритм

У процесі вирішення прикладних задач вибір потрібного алгоритму викликає певні труднощі. Потрібно визначити на чому базувати свій вибір, якщо алгоритм повинен задовольняти наступні протиріччя:

1. Бути простим для розуміння, перекладу в програмний код і наладки.

2. Ефективно використовувати комп'ютерні ресурси і виконуватися швидко.

Якщо написана програма повинна виконуватися лише кілька разів, то перша вимога найбільш важлива. Вартість робочого часу програміста, звичайно, значно перевищує вартість машинного часу виконання програми, тому вартість програми оптимізується за вартістю написання (а не виконання) програми. Якщо мати справу з задачею, вирішення якої потребує значних обчислювальних затрат, то вартість виконання програми може перевищити вартість написання програми, особливо якщо програма повинна виконуватися багаторазово. Тому, з економічної точки зору, перевагу буде мати складний комплексний алгоритм (в надії, що результуюча програма буде виконуватися суттєво швидше, ніж більш проста програма). Але і в цій ситуації розумніше спочатку реалізувати простий алгоритм, щоб визначити, як повинна себе вести більш складна програма. При побудові складної програмної системи бажано реалізувати її простий прототип, на якому можна провести необхідні виміри й змоделювати її поведінку в цілому, перш ніж приступати до розробки кінцевого варіанту. Таким чином, програмісти повинні бути обізнані не тільки з методами побудови швидких алгоритмів, але й знати, коли їх потрібно застосувати.

Існує кілька способів оцінки складності алгоритмів. Програмісти, звичайно, зосереджують увагу на швидкості алгоритму, але важливі й інші вимоги, наприклад, до розмірів пам'яті, вільного місця на диску або інших ресурсів. Від швидкого алгоритму може бути мало користі, якщо під нього буде потрібно більше пам'яті, ніж встановлено на комп'ютері.

Важливо розрізняти практичну складність, яка є точною мірою часу обчислення і об'єму пам'яті для конкретної моделі обчислювальної машини, і теоретичну складність, яка більш незалежна від практичних умов виконання алгоритму і дає порядок величини вартості.

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

Інтуїтивно можна виділити такі основні складові складності алгоритму:

1. Логічна складність - кількість людино-місяців, витрачених на створення алгоритму.

2. Статична складність - довжина опису алгоритмів (кількість операторів).

3. Часова складність - час виконання алгоритму.

4. Ємнісна складність - кількість умовних одиниць пам'яті, необхідних для роботи алгоритму.

Щоб порівняти ефективність алгоритмів, Юріс Хартманіс і Річард Е. Стернс розробили міру ступеня складності алгоритму, яка називається **обчислювальною складністю**.

Більшість алгоритмів надає вибір між швидкістю виконання і ресурсами. Задача може виконуватися швидше, використовуючи більше пам'яті, або навпаки – повільніше з меншим обсягом пам'яті.

Прикладом в даному випадку може слугувати алгоритм знаходження найкоротшого шляху. Задавши карту вулиць міста у вигляді мережі, можна написати алгоритм, що обчислює найкоротшу відстань між будь-якими двома точками цієї мережі. Замість того, щоб кожного разу заново перераховувати найкоротшу відстань між двома заданими точками, можна наперед прорахувати її для всіх пар точок і зберігати результати в таблиці. Тоді, щоб знайти найкоротшу відстань для двох заданих точка, достатньо буде просто взяти готові значення з таблиці. При цьому отримують результат, практично, миттєво, але це потребує великий обсяг пам'яті. Карта вулиць для великого міста може містити сотні тисяч точок. Для такої мережі таблиця найкоротших відстаней містила б більше 10 мільярдів записів. У цьому випадку вибір між часом виконання і обсягом необхідної пам'яті очевидний.

Із цього зв'язку випливає ідея просторово-часової складності алгоритмів. При цьому підході складність алгоритму оцінюється в термінах часу і простору, і знаходиться компроміс між ними.

2. Алгоритм та його аналіз

Означення. Аналіз алгоритмів – це процес визначення обчислювальної складності алгоритмів, тобто кількості часу, пам'яті чи інших ресурсів, необхідних для виконання алгоритмів.

Термін «аналіз алгоритмів» був введений Дональдом Кнутом. Аналіз алгоритмів є важливою частиною більш загальної теорії обчислювальної складності, яка встановлює теоретичні оцінки ресурсів, необхідних будь-якому алгоритму, що розв'язує задану обчислювальну задачу.

Аналіз алгоритму полягає у визначенні та аналізі таких критеріїв як:

1) час роботи програми, як функцію від вхідних даних;
2) загальну кількість пам'яті, що необхідна для даних програми;
3) загальний об'єм програмного коду;
4) чи програма розв'язує коректно (правильно) поставлену задачу;
5) чи стресо-стійка програма, тобто як добре буде поводити себе програма з некоректними вхідними даними.
6) комплексність програми, тобто чи легко програму читати, розуміти та модифікувати.

Останні 4 пункти частіше за все покладаються на зовнішні аналізатори (інженери-тестувальники, автоматичні системи тестування, рецензування коду).

Перші два цілковито знаходяться у зоні відповідальності програміста, який створює програму.

3. Час виконання (running time)

Припустимо для розв'язання певної задачі побудований алгоритм. Потрібно оцінити на скільки оптимальним (швидким) є цей алгоритм. Виникає низка цілком закономірних питань:

- Як це зробити, тобто оцінити швидкодію цього алгоритму?
- У яких одиницях цю швидкодію вимірювати?
- Яка врешті буде складність цього алгоритму?

Звичайно більшість читачів, які вперше зіштовхнулися з цією проблематикою, скажуть, що швидкодію алгоритму можна оцінити як «час виконання програми» побудованої за цим алгоритмом і будуть правими. Проте, швидше за все, більшість з них буде вкладати в термін «час» кількість секунд, хвилин чи годин за які виконується програма написана за цим алгоритмом. А ось тут не все так однозначно. Таке означення не є коректним, оскільки воно лише частково характеризує швидкодію алгоритму. Дійсно, ні для кого не секрет, що швидкодія різних комп'ютерів різна і якщо на одному комп'ютері програма буде

виконуватися, припустимо 10 секунд, то на комп'ютері, що в 10 разів швидший – 1 секунду. Однак найбільш неефективний алгоритм, запущений на комп'ютері Fugaku, може виконуватися набагато швидше, ніж найефективніший алгоритм, запущений на ПК, тому час виконання завжди залежить від системи. Наприклад, щоб порівняти 100 алгоритмів, усі вони мають бути запущені на одній машині. Крім того, результати тестів під час виконання залежать від мови, якою написаний даний алгоритм, навіть якщо тести виконуються на одній машині. Якщо програми скомпільовані, вони виконуються набагато швидше, ніж під час інтерпретації. Програма, написана на C++ або Ada, може бути в 20 разів швидшою, ніж та сама програма, закодована на Python або Prolog.

Тому цілком логічним є те, що для оцінки складності алгоритму необхідно використовувати критерій, що не залежить від потужності ЕОМ або мови програмування на якій реалізовано алгоритм.

На час виконання програми впливають фактори:

1. Введення початкової інформації в програму.

2. Якість скомпільованого коду програми, що виконується.

3. Машинні інструкції (звичайні і прискорені), які використовуються для виконання програми.

4. Часова складність алгоритму відповідної програми.

Часова ефективність алгоритму визначається як функція від розміру даних, шляхом підрахунку кількості виконаних базових операцій.

Базова операція – це операція, яка робить основний внесок в загальний час виконання алгоритму. Зазвичай це операція з найбільшим часом роботи в найбільш глибоко вкладеному циклі.

Робота будь-якої програми, що виконується комп'ютером складається з елементарних операцій, які об'єднуються у блоки для утворення інструкцій мови програмування.

До елементарних операцій віднесемо операції з набору:

- Просте присвоювання: a = b;
- Одновимірна індексація a[i]: (адреса (a) + i * довжина елементу).
- Арифметичні операції: (*, /, -, +).
- Операції порівняння: a < b.
- Логічні операції {or, and, not}.

Виключимо команду переходу за адресою - вона пов'язана з операцією порівняння в конструкції розгалуження.

Як ви можете здогадуватися, час, який витрачає комп'ютер на різні типи інструкцій різний. Наприклад, час інструкцій звернення до об'єкту в пам'яті та присвоєння значення змінній може суттєво відрізнятися. Проте, для дослідження питань цього курсу, нам буде достатнього розглядати спрощену модель комп'ютера, вважаючи, що всі елементарні операції виконуються за однаковий час τ, і без обмежень загальності, можемо вважати, що

$\tau = 1$.

Таким чином

Означення. Часом виконання програми (eng. running time, time complexity) називається кількість елементарних операцій, які виконує комп'ютер під час виконання програми.

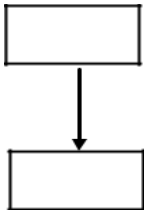
Очевидно, що ця кількість операцій може залежати від вхідних даних (inputs) задачі. Дійсно, наприклад очевидно, що для знаходження визначника матриці розміром 3 треба здійснити значно більше операцій, ніж для знаходження визначника розміром 2.

Час виконання програми будемо позначати

$$T(n)$$

де n – розмір вхідних даних.

Основні алгоритмічні конструкції та їх «трудомісткість»:



а) конструкція «Послідовного переходу»

Трудомісткість конструкції є сума трудомісткості блоків, які виконуються послідовно один за одним:

$$T_{\text{« Послідовного переходу»}} = t_1 + \ldots + t_k,$$

де k – кількість блоків.

б) конструкція «Розгалуження»

if (l)

f_{then} з ймовірністю p

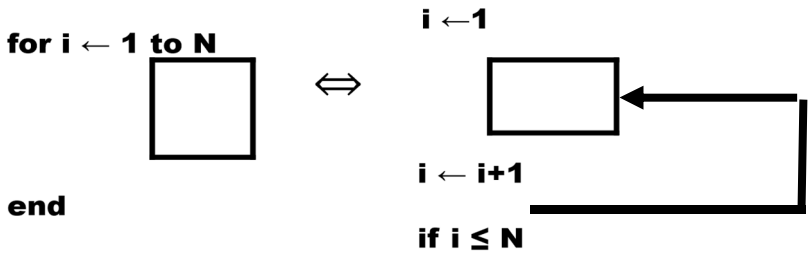
else

f_{else} з ймовірністю *(1-p)*

Загальна трудомісткість конструкції «Розгалуження» вимагає аналізу ймовірності виконання переходів на блоки «then» і «else» і визначається як:

$$T_{\text{«Розгалуження»}} = t_{then} * p + t_{else} * (1-p)$$

в) конструкція «Цикл»



Після зведення конструкції до елементарних операцій її трудомісткість визначається як:

$T_{«цикл»} = 1+3*N+N*t_{«тіло\ циклу»}$

Приклади аналізу простих алгоритмів

Приклад 1. Задача знаходження суми елементів квадратної матриці.

Рядок	Дія алгоритму		
1	int sumM(){		
2	int sum = 0;	sum = 0;	1
3	for (int i=0; i < n; i++)	i = 0;	1
4	for (int j=0; j < n; j++)	j = 0;	n
5	sum += S[i][j];	sum += S[i][j];	4*n*n
6		j = j+1 if j < n	3*n*n
		i = i+1 if I < n	3*n
	return sum; }	return sum;	1
T(n)= 1 + 1 + $7*n^2$ + 4*n +1 = $7*n^2$ + 4*n + 3			

Приклад 2. Знайдемо кількість операцій алгоритму, що обчислює суму компонент квадратної матриці розмірності n.

Рядок	Дія алгоритму	Кількість операцій
1	int sumM(){	
2	result = 0 ;	1
3	i = 0 ;	1
4	j = 0 ;	1
5	while (i < n){	n

Рядок	Дія алгоритму	Кількість операцій
6	while (j < n){	n
7	result += A[i][j];	4*n*n
8	j += 1; }	2*n*n
9	j = 0 ;	n
10	i += 1 ;	2*n
11	return result ; }	1
Всього операцій		$T(n)=6*n^2+5*n+4$

4. Найкращий, найгірший та середній час виконання

У попередніх прикладах знайдений час виконання програм залежав лише від розміру вхідних даних. Проте, для багатьох програм час виконання програми часто залежить не стільки від розміру вхідних даних, скільки від самих цих даних.

Означення. Часом виконання програми у найгіршому випадку (eng. worst-case running time, worst-case time complexity) будемо називати найбільшу кількість елементарних операцій, які виконує комп'ютер під час виконання програми для довільних вхідних даних розміру n.

Означення. Часом виконання програми у найкращому випадку (eng. best-case running time, best-case time complexity) будемо називати найменшу кількість елементарних операцій, які виконує комп'ютер під час виконання програми яка досягається для деякого набору вхідних даних розміру n.

Очевидно, що згадані вище випадки є крайніми та не завжди можуть об'єктивно відображати інформацію про швидкодію алгоритму на практиці для конкретних наборів вхідних даних. Тому, крім згаданих випадків, часто розглядають третій – час виконання в середньому.

Означення. Часом виконання програми в середньому (eng. average-case running time, average-case time complexity) будемо називати усереднену кількість елементарних операцій, які виконує комп'ютер під час виконання програми для всіх наборів вхідних даних розміру n.

Час виконання в середньому час залежить від ймовірнісного розподілу вхідних даних і, може бути визначеним, якщо ці ймовірнісні характеристики відомі. На практиці час виконання у середньому знайти значно складніше, аніж час виконання у найгіршому випадку, враховуючи математичну складність такої задачі. Тому в основному будемо використовувати час виконання у найгіршому випадку, як характеристику часової складності алгоритму.

Приклад 3. Знаходження максимального елемента в масиві

Рядок	Дія алгоритму		
1	int maxM(){		
2	int max_elem = S[0];	max_elem = S[0];	**2**
3	for (int i=1; i < n; i++)	i = 1	1
4	if (max_elem < S[i])	max_elem < S[i]	2*(n-1)
5	max_elem = S[i];	max_elem = S[i];	2*(n-1)
6		i = i+1 if (i<n)	3*(n-1)
	return max_elem; }	max_elem	1
Всього операцій:		T(n)= 7*n-3	

Даний алгоритм є кількісно-параметричним, тому для фіксованої розмірності вхідних даних необхідно проводити аналіз для гіршого, кращого і середнього випадків.

а) найгірший випадок

Максимальна кількість переприсвоєння максимуму (на кожному проході циклу) буде в тому випадку, якщо елементи масиву відсортовані за зростанням.

Трудомісткість алгоритму в цьому випадку дорівнює:

T(n) = 2+1+1+ (n-1) (3+2+2)=7*n – 3.

б) кращий випадок

Мінімальна кількість переприсвоєння максимуму (жодного на кожному проході циклу) буде в тому випадку, якщо максимальний елемент розміщено на першому місці в масиві.

Трудомісткість алгоритму в цьому випадку дорівнює:

T(n) = 2+1+1+ (n-1) (3+2)=5*n – 1.

в) середній випадок

Алгоритм пошуку максимуму послідовно перебирає елементи масиву, порівнюючи поточний елемент масиву з поточним значенням максимуму.

На черговому кроці, коли переглядається k-тий елемент масиву, переприсвоєння максимуму відбудеться, якщо в підмасиві з перших k елементів максимальним елементом є останній.

У випадку даних рівномірного розподілу вхідних, ймовірність того, що максимальний з к елементів, розташований у деякій (останній) позиції, дорівнює 1/k.

У масиві з n елементів загальна кількість операцій переприсвоєння максимуму визначається:

$$\sum_{i=1}^{n} \frac{1}{k} = Hn \approx \ln(n) + \gamma,\ \gamma = 0{,}57$$

Hn називається n-им гармонійним числом.

Точне значення (математичне очікування) середньої кількості операцій присвоювання в алгоритмі пошуку максимуму в масиві з n елементів визначається величиною Hn

T(n)=2 + (n-1)*(3+2) + 2*(Ln(n) + γ) = 5*n +2*Ln(n) – 3 +2*γ

Теорема 1. (Послідовна композиція). Найгірший час виконання алгоритму, що складається з послідовності виразів S1, S2, ..., Sm має асимптотичну складність

$O(\max\{T1(n),...,Tm(n)\})$,

де $T1(n)$, ... , $Tm(n)$ – час виконання відповідно інструкцій S1, ..., Sm.

Теорема 2. (Цикл while). Найгірший час виконання алгоритму, що містить цикл while

while(S1){S2}

має асимптотичну складність

$O(\max\{T1(n)\times(I(n)+1), T2(n)\times I(n)\})$,

де $T1(n)$, $T2(n)$ – час виконання відповідно інструкцій S1, S2, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.

Теорема 3. (Умовний оператор). Найгірший час виконання алгоритму

```
if (S1)
  S2;
else S3;
```

має асимптотичну складність

$O(\max\{T1(n), T2(n), T3(n)\})$,

де $T1(n)$, $T2(n)$, $T3(n)$ – час виконання відповідно інструкцій S1, S2, S3.

5. Асимптотична оцінка складності алгоритмів. O – символіка

Спробуємо зрозуміти, чому так важливо оцінювати часову складність алгоритмів. Припустимо, що ми розглядаємо два алгоритми: A і B для розв'язання деякої задачі. Нехай зроблено ретельний аналіз часу роботи кожного з алгоритмів і визначили їх як $T_A(n)$ та $T_B(n)$ відповідно, де n – розмір вхідних даних. Тоді кращий алгоритм той, у якого $T_A(n)<T_B(n)$. Якщо наперед нічого не відомо про вхідні дані, то який ефективніший алгоритм складно говорити.

Швидкість зростання всіх членів функції

Для прикладу розглянемо швидкість зростання функції

$$f(n) = n^2 + 100n + lgn + 1000.$$

Подивимося, як впливатимуть значення одночленів даної функції у залежності від значення n. Їх подано у наступній таблиці [6, С. 70].

n	f(n)	n^2		100n		$\lg n$		1000	
	Значення	Значення	%	Значення	%	Значення	%	Значення	%
1	1 101	1	0,1	100	9,1	0	0	1000	90,83
10	2 101	100	4,76	1000	47,6	1	0,05	1000	47,6
100	21 002	10 000	47,6	10 000	47,6	2	0,001	1000	4,76
1000	1101003	1000000	90,8	100000	9,1	3	0,0003	1000	0,09
10000	101001004	100000000	99,0	1000000	0,99	4	0,0	1000	0,001
100000	10010001005	10000000000	99,9	10000000	0,099	5	0,0	1000	0,00

Із даної таблиці можна зробити висновок, що, для різних значень змінної n роль одночленів на загальну швидкодію стає іншою. Навіть ігноруючи постійні множники, ми отримуємо чітке загальне уявлення про придатність алгоритма для розв'язання задачі певного розміру.

Давайте розглянемо такий приклад. Допустимо, що величина входу деякої задачі є n. Нехай три студенти по різному розв'язали цю задачу.

Нехай є три алгоритми розв'язання задачі, часова складність яких визначена:

1. $T_1(n) = 1962n^2 - 256n + 10$
2. $T_2(n) = 62n^3 + 19n^2 - 25n + 2$
3. $T_3(n) = 3 * 2^n + 1$

Який алгоритм кращий?

Який алгоритм кращий? У випадку, якщо розмір входу буде зовсім невеликий, наприклад n = 4, то останній алгоритм буде виконуватися за найменшу кількість операцій. Проте вже при n = 20 кращим буде другий алгоритм, а от, якщо n буде великим, наприклад 1000000, то другий алгоритм буде використовувати значно більше операцій ніж перший, а останній взагалі можна вважати нескінченним.

При оцінці швидкодії програми:

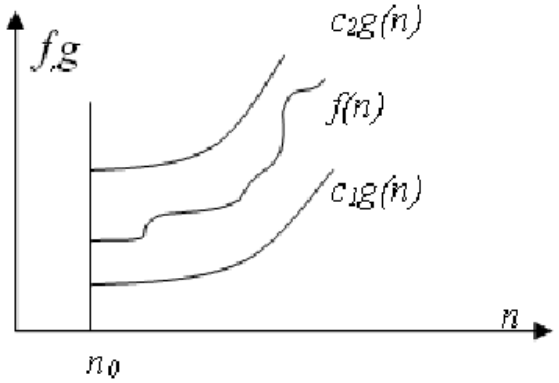
1. У формулі кількості операцій, враховують лише доданок, що зростає найшвидше.
2. Сталий множник при цьому доданку встановлюють рівними 1.
3. Отриману величину називають **порядком складності алгоритму.**

Для оцінки складності програм, залежно від вхідних даних, використовують O, Ω та Θ символіку яка є формалізованим обґрунтуванням порядку складності алгоритму. Основне їхнє призначення це «грубо» оцінити час виконання алгоритму, а також наскільки швидко зростає час роботи алгоритму зі збільшенням розміру вхідних даних. Фактично тут піде мова про

асимптотичну поведінку функцій часу виконання, що залежать від вхідних даних алгоритму.

Асимптотичне позначення O йде з підручника Бахмана по теорії простих чисел (Bachman, 1892), позначення Ω та Θ, введені Д.Кнутом (Donald Knuth).

Θ – символіка



Нехай f (n) і g (n) – додатні функції додатного аргументу, n ≥ 1 (кількість об'єктів на вході і кількість операцій – додатні числа), тоді:

f(n) = Θ (g(n)),

якщо існують додатні c1, c2, n0, такі, що:

c1 * g(n) ≤ f(n) ≤ c2 * g(n),

при n > n0.

Кажуть: функція g(n) є асимптотично точною оцінкою функції f(n), тому що, за визначенням, функція f(n) не відрізняється від функції g(n) з точністю до постійного множника.

З f(n) = Θ(g(n)) випливає:

g(n) = Θ(f (n)).

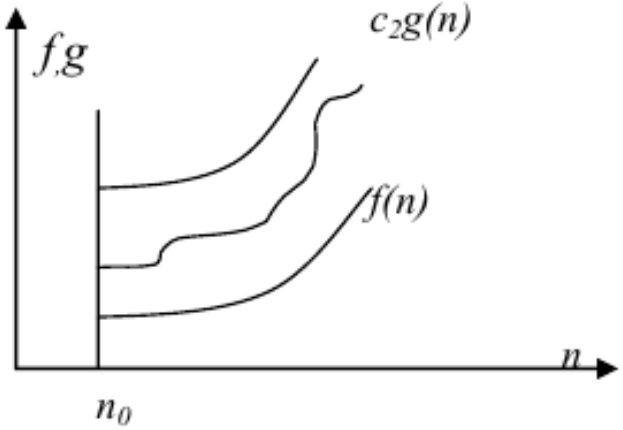
Приклади:

f(n)=4n^2+nlnN+174 ⇨ f(n)= Θ(n^2);

f(n)=Θ(1) – запис означає, что f(n) або дорівнює константі, яка не дорівнює нулю, або f(n) обмежена константою на ∞ :

f(n) = 7+1/n = Θ(1).

Оцінка O (O велике)



Дана оцінка потребує тільки, що б функція f(n) не перевищувала g(n) починаючи з n>n0, з точністю до постійного множника.

<u>Означення.</u> Кажуть, що f(n) = O(g(n)) якщо ∃ c>0 та ∃ n0 > 0, що ∀n > n0:

0 ≤ f(n) ≤ c * g(n).

Запис O(g(n)) означає клас функцій, які зростають не швидше, ніж функція g(n) з точністю до постійного множника.

Іноді говорять, що g(n) мажорує функцію f(n).

У такому разі також кажуть, що «f зростає не швидше ніж g».

Приклад 4. Розглянемо функцію $f(n) = 8n + 128$.

Очевидно, що $f(n) \geq 0$ для всіх натуральних n. Покажемо, що $f = O(n^2)$. Згідно з означенням, нам треба знайти таке натуральне n_0 і таку сталу $C > 0$, що для всіх $n \geq n_0$

$$8n + 128 \leq Cn^2$$

Не має значення величина сталої – головне чи вона існує. Припустимо, що $C = 1$

$$8n + 128 \leq n^2 \Rightarrow n^2 - 8n - 128 \geq 0 \Rightarrow (n - 16)(n + 8) \geq 0$$

Очевидно, що остання нерівність виконується при $n \geq 16$. Таким чином, $\exists C = 1$ та $\exists n_0 = 16$, що для всіх $n \geq n_0$

$$8n + 128 \leq n^2$$

Отже,

$$f(n) = O(n^2)$$

Очевидно, що існує безліч значень таких пар C та n_0.

Наприклад, для всіх функцій:

- f(n)=1/n,
- f(n)= 12,
- f(n)=3*n+17,
- f(n)=n*ln(n),
- f(n)=6*n^2 +24*n+77

буде вірною оцінка O(n^2).

Вказуючи оцінку O є сенс вказувати найбільш «близьку» функцію, оскільки, наприклад, для f(n) = n^2 є справедливою оцінка O (2^n), проте вона не має практичного сенсу.

Приклад 5.

```
for (i = 0; i < n; i++) {
  for (j = 1, sum = a[0]; j <= i; j++)
    sum += a[j];
  cout << "сума для підмасиву від 0 до" << i << " = " << sum << endl;  }
```

Оцінимо складність даного алгоритму:

$$1 + 3(n-1) + 6n + \sum_{i=1}^{n-1} 2i = 9n + 2\left(1 + 2 + \cdots + n - 1\right) - 2 =$$

$$= 9n + n\left(n - 1\right) - 2 = O(n) + O(n^2) = O(n^2)$$

Приклад 6.

```
for (i = 0, length = 1; i < n-1; i++) {
  for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
  if (length < i2 - i1 + 1)
```

length = i2 - i1 + 1;

}

Складність даного алгоритму $O(n^2)$.

Правила визначення асимптотичної поведінки функції

1. Мультиплікативні константи можна опускати, тобто для будь-якої додатної сталої C:

$Cn^3 = O(n^3)$;

2. Нехай b ≥a тоді $n^a = O(n^b)$;

3. Нехай a > 1. Тоді для будь-якого k ≥ 1, $n^k = O(a^n)$;

4. Функція $\log_a n = O(\log_b n)$ для будь-яких додатних чисел a≠1 і b≠1;

5. Якщо функція є сумою кількох функцій, то її асимптотична поведінка визначається доданком, що зростає найшвидше.

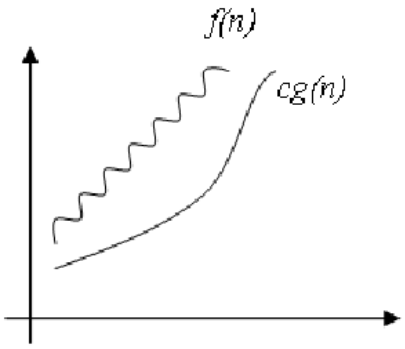
Наприклад: $196n^3+1023n^2+19n+15=O(n^3)$

Поширені асимптотичні складності

Складність	Коментар	Приклади
O(1)	Сталий час роботи не залежно від розміру задачі	Пошук у хеш-таблиці
O(loglog n)	Дуже повільне зростання необхідного часу	Очікуваний час роботи інтерполюючого пошуку n елементів
O(log n)	Логарифмічне зростання — подвоєння розміру задачі збільшує час роботи на сталу величину	Швидке обчислення x^n ; двійковий пошук у відсортованому масиві з n елементів
O(n)	Лінійне зростання — подвоєння розміру задачі подвоїть і необхідний час	Додавання/віднімання чисел з n цифр; лінійний пошук в масиві з n елементів
O(nlog n)	Лінеаритмічне зростання — подвоєння розміру задачі збільшить необхідний час трохи більше ніж вдвічі	Сортування злиттям або купою масиву з n елементів.

Складність	Коментар	Приклади
$O(n^2)$	Квадратичне зростання — подвоєння розміру задачі вчетверо збільшує необхідний час	Елементарні алгоритми сортування масивів з n елементів; Лінійний пошук у квадратній матриці розмірності n.
$O(n^3)$	Кубічне зростання — подвоєння розміру задачі збільшує необхідний час у вісім разів	Звичайне множення матриць
$O(a^n)$	Експоненціальне зростання — збільшення розміру задачі на 1 призводить до a -кратного збільшення необхідного часу; подвоєння розміру задачі підносить необхідний час у квадрат	Деякі задачі комівояжера; Алгоритми пошуку повним перебором

Оцінка Ω (омега)



Означення. Говорять, що $f(n)=\Omega(g(n))$, якщо знайдеться така константа $c> 0$ і таке число n_0, що $0 \le cg(n) \le f(n)$ для всіх $n \ge n_0$.

У такому разі також кажуть, що «f зростає не повільніше ніж g».

Це нижня оцінка, тобто складність алгоритму не менша деякої функції $cg(n)$ і геометрично знаходиться вище її графіку.

Приклад 7. Покажемо, що $5n^2 - 64n + 256 = \Omega(n^2)$.

Відповідно до означення, нам треба знайти таке натуральне $n0$ і таку сталу $C > 0$, що для всіх $n \ge n_0$

$5n^2 - 64n + 256 \ge Cn^2$.

Виберемо $C = 1$. Тоді

$5n^2 - 64n + 256 \ge n^2$

$\Rightarrow 4n^2 - 64n + 256 \ge 0$

$\Rightarrow 4(n - 8)^2 \ge 0$

Оскільки $(n - 8)^2 \ge 0$ для всіх $n \ge 0$, отримуємо, що $n_0 = 1$, що і доводить твердження прикладу.

Властивості

1. **Транзитивність:**

- $f(n) = \Theta(g(n))$ і $g(n) = \Theta(h(n))$ то $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ і $g(n) = O(h(n))$ то $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ і $g(n) = \Omega(h(n))$ то $f(n) = \Omega(h(n))$

2. **Рефлексивність:**

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

3. **Симетричність:**

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

6. Класи алгоритмів

На початку 1960-х років, у зв'язку з початком широкого використання обчислювальної техніки для розв'язання практичних завдань, виникло питання про межі практичної застосовності цього алгоритму розв'язання задачі в сенсі обмежень на її розмірність. Які завдання можуть бути вирішені на ЕОМ за реальний час? Відповідь на це питання була дана у працях Кобхема (Alan Cobham, 1964) та Едмондса (Jack Edmonds, 1965), де були введені класи складності задач.

Алгоритми, час роботи яких при розмірі входу n складає не більше $O(n^k)$, називаються **поліноміальними**.

Клас поліноміальних задач позначається **P** і має властивість замкнутості. Композиція двох поліноміальних алгоритмів також працює за поліноміальний час. Це випливає з того, що сума, добуток та композиція багаточленів є багаточлен.

Задачі, для яких існують перевірочні алгоритми, що працюють за поліноміальний час, утворюють клас **NP**. Це означає недетермінований поліноміальний час. Іншими словами, клас **P** це клас задач, які можна розв'язати швидко, а клас **NP** це клас задач, розв'язок яких може бути швидко перевірений.

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

1) відшукання експоненціального алгоритму, що працює для реальних даних за реальний час;
2) за поліноміальний час відшукати не оптимальне рішення, а деяке наближення до нього. Такі алгоритми називаються **наближеними алгоритмами**.

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

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

Контрольні запитання:

1. Що таке складність алгоритму?
2. Які фактори впливають на час виконання програми?
3. Що таке часова ефективність алгоритму?
4. Що таке час виконання програми?
5. Як визначається час виконання алгоритму, що містить цикл?
6. Дайте означення O(g(n)).
7. Які правила визначення асимптотичної поведінки функції?
8. Які розрізняють класи алгоритмів?

Тема № 4. ЛІНІЙНІ СТРУКТУРИ ДАНИХ

План лекції:

1. Однозв'язний список
2. Двозв'язний список
3. Стек
4. Черга

Джерела:

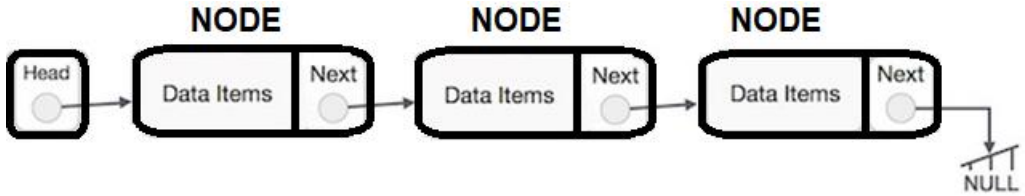
[1, §1.4], [2, Розділ 3, 5], [3, Глава 2], [4, §3.3, 3.4], [5, §10.1, 10.2], [6, Chapter 3, 4], [7, §2.2], [10, Chapter 3], [11, Chapter 3].

1. Зв'язний список

На відміну від масивів, зв'язаний список не зберігає елементи даних у суміжних місцях пам'яті.

Означення. Зв'язний список — це послідовність структур даних, які пов'язані між собою за допомогою посилань.

Зв'язний список — це лінійна структура даних (наприклад, масиви), де кожен елемент є окремим об'єктом. Кожен елемент (вузол) списку складається з двох елементів – даних і посилання на наступний вузол.



Зв'язний список може зберігатися:

- в оперативній пам'яті;
- у файлі.

У зв'язному списку виділяють окремий елемент (Element), який ще називається вузлом (Node).

Зв'язний список має такі властивості:

1) Послідовні елементи з'єднуються вказівниками.

2) Останній елемент вказує на NULL.

3) Може збільшуватися або зменшуватися в розмірі під час виконання програми.

4) Можна робити вузлів стільки, скільки потрібно (поки системна пам'ять не вичерпається).

5) Не витрачає місце в пам'яті (але займає додаткову пам'ять для вказівників). Він виділяє пам'ять у міру зростання списку.

Зв'язний список має переваги перед масивами:

1) Динамічний розмір

2) Простота вставки/видалення

До недоліків зв'язного списку відносять:

1) Довільний доступ не допускається.

2) Для кожного елемента списку потрібен додатковий простір для вказівника.

3) Не підтримує кеш.

Пов'язаний список представлений вказівником на перший вузол зв'язаного списку. Перший вузол називається головою. Якщо зв'язаний список порожній, значення голови = NULL.

Кожен вузол у списку складається щонайменше з двох частин:

1) дані;

2) вказівник (або посилання) на наступний вузол.

У C++ можемо представляти вузол за допомогою структур або класів. Наприклад вузол зв'язаного списку з цілочисельними даними можна подати через структури:

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

Типи зв’язних списків:

1. Однозв’язний список
2. Двозв’язний список

Однозв’язний список

Кожен вузол зберігає адресу або посилання наступного вузла в списку, а останній вузол має наступну адресу або посилання як NULL.

Наприклад , 1->2->3->NULL

Якщо вузол містить елемент даних, який є вказівником на інший вузол, то багато вузлів можна об’єднати разом, використовуючи лише одну змінну для доступу до всієї послідовності вузлів.

Якщо вузол має посилання лише на його наступника в цій послідовності, список називається **однозв’язаним списком**.

Вузол можна додати трьома способами:

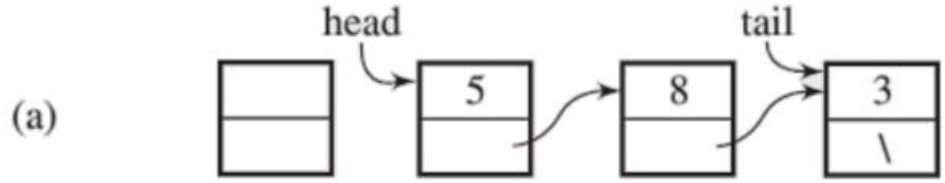
1) На початку зв’язного списку

2) Після заданого вузла.

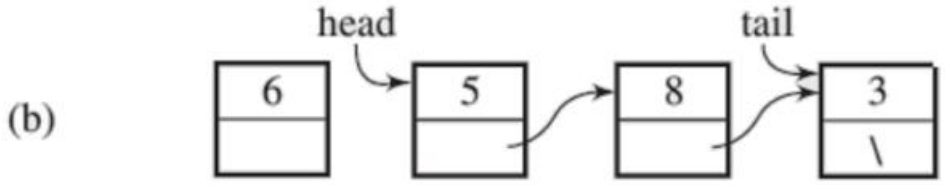
3) У кінці зв’язного списку.

Додати вузол на початку зв’язного списку:

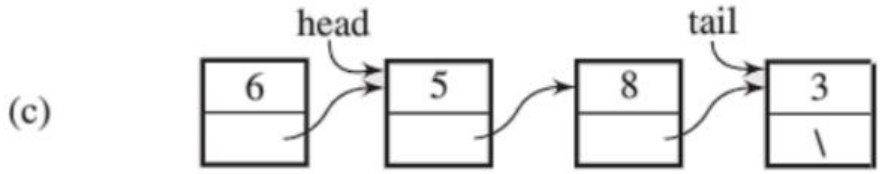
1. Створюється порожній вузол. (Малюнок a).



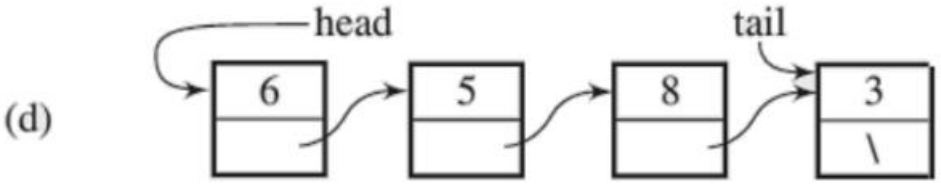
2. Інформаційний елемент вузла ініціалізується певним цілим числом (Малюнок b).



3. Оскільки вузол включається на початку списку, наступний член стає вказівником на перший вузол у списку; (Малюнок c).



4. Новий вузол передує всім вузлам у списку, але цей факт має бути відображений у значенні head; інакше новий вузол буде недоступним. Тому заголовок оновлюється, щоб стати вказівником на новий вузол (Малюнок d).

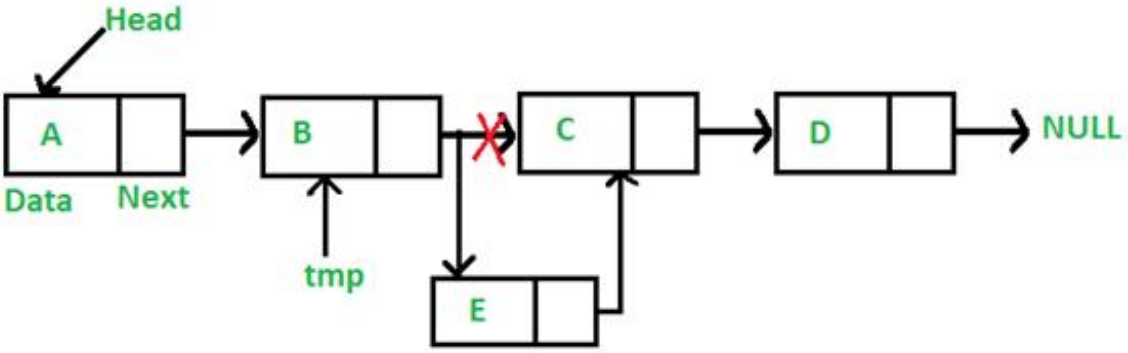


Часова складність алгоритму дорівнює O(1)

Додати вузол після заданого вузла

Дається вказівник на вузол, і новий вузол вставляється після даного вузла:

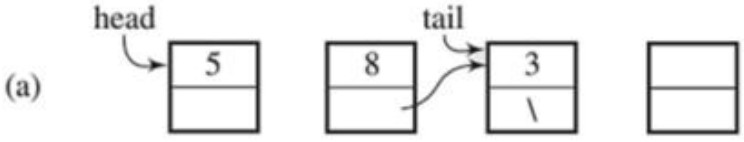
1. Перевірити, чи заданий prev_node є NULL
2. Виділити новий вузол.
3. Ввести дані.
4. Зробити наступний із нового вузла наступним за prev_node
5. Перемістити наступний за prev_node як new_node



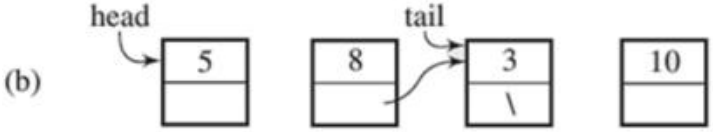
Часова складність insertAfter ( ) дорівнює O(1)

Додати вузол в кінець списку

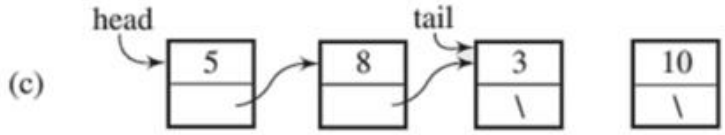
1. Створюється порожній вузол (Малюнок a).



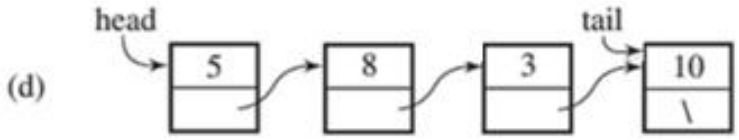
2. Інформаційний елемент вузла ініціалізується цілим числом new_node (Мал. b).



3. Оскільки вузол включено в кінець списку, для наступного члена встановлено значення Null (Малюнок c).

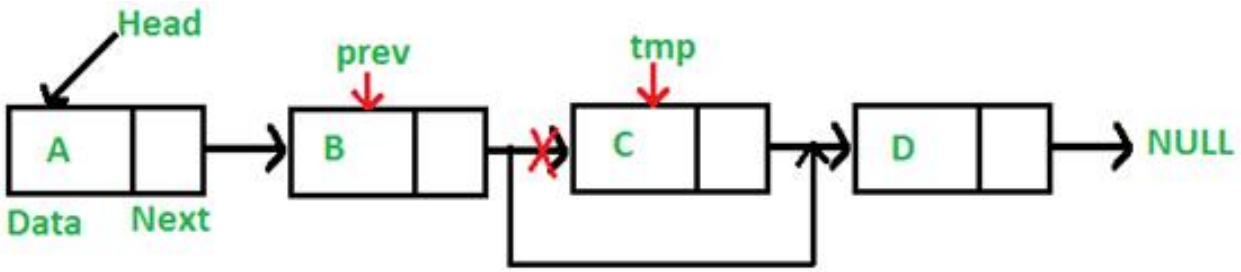


4. Тепер вузол включено до списку, зробивши посилання наступного члена останнього вузла списку вказівником на щойно створений вузол (Малюнок d).



Щоб видалити вузол зі зв'язаного списку

1) Знайти попередній вузол для вузла, який потрібно видалити.
2) Змінити наступний вузол у попереднього вузла.
3) Звільнити пам'ять вузла, який потрібно видалити.



Нижче наведено лістинг програми з реалізацією усіх алгоритмів на C++.

```
/* Повна робоча програма C++ для демонстрації однозв'яз-
ного списку */
#include <bits/stdc++.h>
using namespace std;

struct Node
{
int data;
Node *next;
};

/* Посилання (покажчик на вказівник) на голову списку та
int елемент,
    вставляє новий вузол на початку списку. */
void push(Node** head_ref, int new_data)
```

```
{
 /* 1. створити вузол */
 Node* new_node = new Node();
 /* 2. ввести дані */
 new_node->data = new_data;
 /* 3 . Зробити наступний новий вузол головою */
 new_node->next = (*head_ref);
 /* 4. перемістити голову, щоб вказати на новий вузол */
 (*head_ref) = new_node;
}

    /* Дано вказівник на вузол prev_node, вставити новий
вузол після заданого prev_node */
void insertAfter(Node* prev_node, int new_data) {
  // 1. Перевірити, чи заданий prev_node має значення NULL
  if (prev_node == NULL) {
    cout << "Даний попередній вузол не може бути NULL";
    return;  }
  // 2. Створити новий вузол
  Node* new_node = new Node();
  // 3. Ввести дані
  new_node->data = new_data;
  // 4. Зробити наступний з нового вузла наступним з попереднього_вузла
  new_node->next = prev_node->next;
   // 5. перемістити наступний з prev_node як new_node
  prev_node->next = new_node;
}

/* Дано покажчик (вказівник на вказівник) на голову списку
та int елемент
додає новий вузол у кінець списку */
void append(Node** head_ref, int new_data)
{

    // 1. створити вузол
    Node* new_node = new Node();

    // Використовується на кроці 5
    Node *last = *head_ref;
```

```
    // 2. Ввести дані
    new_node->data = new_data;

    // 3. Цей новий вузол буде останній вузол,
    //  тому зробити покажчик на наступний вузол
     //  як NULL
    new_node->next = NULL;

    // 4. Якщо зв'язаний список порожній, зробити
    // новий вузол головою
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    // 5. Інакше перехід до останнього вузла
    while (last->next != NULL)
    {
        last = last->next;
    }
```

// 6. Змінити покажчик наступного вузла на створений вузол

```
    last->next = new_node;
    return;
}

/* Дано посилання (покажчик на вказівник) на голову списку
та ключ, видаляє перше входження ключа у зв'язаний список
*/
void deleteNode(Node** head_ref, int key) {
  // Зберегти головний вузол
  Node* temp = *head_ref;
  Node* prev = NULL;
  /* Якщо сам головний вузол містить ключ, який потрібно
видалити */
  if (temp != NULL && temp->data == key)   {
    *head_ref = temp->next; // Змінити голову
    delete temp;      // знищити стару голову
    return;
```

```
  }
  /* Інакше шукаємо ключ, який потрібно видалити,
відстежуємо попередній вузол, оскільки нам потрібно
змінити попередній->наступний */
   else   {
  while (temp != NULL && temp->data != key)   {
    prev = temp;
    temp = temp->next;
  }
  // Якщо ключ відсутній у зв’язаному списку
  if (temp == NULL)
    return;
  // Від’єднти вузол від пов’язаного списку
  prev->next = temp->next;
  // Free memory
  delete temp;
  }
}

/* Ця функція друкує вміст зв’язаного списку, починаючи з
даного вузла */
void printList(Node* node)
{
  while (node != NULL)
  {
    cout << node->data << " ";
    node = node->next;
  }
}

// Код програми
int main() {
    setlocale(LC_ALL,"ukr");
   // Почати з порожнього списку
  Node* head = NULL;
   // Додати елементи в зв’язаний список
  push(&head, 7);
  push(&head, 1);
  push(&head, 3);
  push(&head, 2);
```

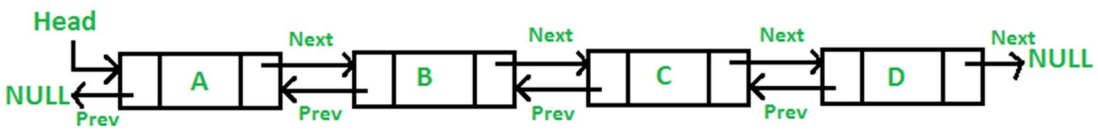
```
  puts("Створений однозв'язаний список:");
  printList(head);

  deleteNode(&head, 1);
  puts("\nЗв'язаний список після видалення 1:");
  printList(head);

  Node* temp = head;
  while (temp != NULL && temp->data != 3)
    temp = temp->next;

  insertAfter(temp, 9);
  puts("\nЗв'язаний список після додавання 9 після 3:");
  printList(head);

  append(&head, 10);
  puts("\nЗв'язаний список після додавання у кінець 10:");
  printList(head);

  return 0;
}
```

2. Двозв'язний список

Двозв'язний список схожий на звичайний однозв'язний список, але елементи у ньому зберігають посилання не лише на наступний, а й на попередній елемент. Завдяки цій властивості, можна переміщуватись по списку вперед і назад.

У цьому типі зв'язного списку є два посилання, пов'язані з кожним вузлом: одна з опорних точок на наступний вузол і одна - на попередній вузол.

Перевага цієї структури даних: ми можемо проходити в обох напрямках, і для видалення нам не потрібно мати явний доступ до попереднього вузла.



Переваги перед однозв'язним списком

- У двозв'язному списку можна рухатися як вперед, так і назад.

➢ Операція видалення у двозв'язному списку ефективніша, якщо вказано вказівник на вузол, який потрібно видалити.

Недоліки перед однозв'язним списком:

1) Кожен вузол двозв'язного списку потребує додаткового місця для покажчика попереднього вузла.

2) Усі операції вимагають попередньо утвореного додаткового покажчика.

Вставити вузол у двозв'язному списку можна:

1) На початку двозв'язного списку.

2) Після заданого вузла.

3) У кінці двозв'язного списку.

4) Перед заданим вузлом.

Розглянемо алгоритм вставлення нового вузла попереду голови списку.

1) Додати вузол спереду

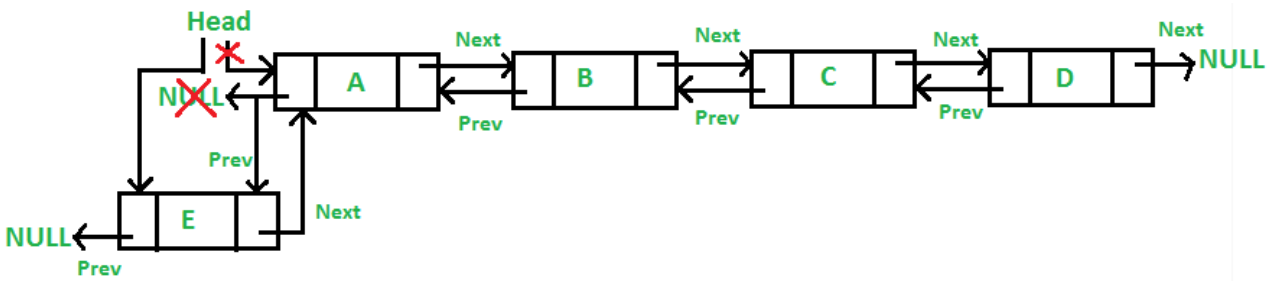
1. Створити новий вузол.

2. Присвоїти значення в поле data.

3. Покажчик нового вузла next присвоїти посилання на голову списку, а покажчику prev нового вузла присвоїти NULL.

4. Змінити попередній вузол голови на новий вузол.

5. Перемісттити голову, щоб вказати на новий вузол.



2) Додати вузол після заданого вузла:

1. Перевірити, чи даний prev_node має значення NULL.

2. Створити новий вузол.

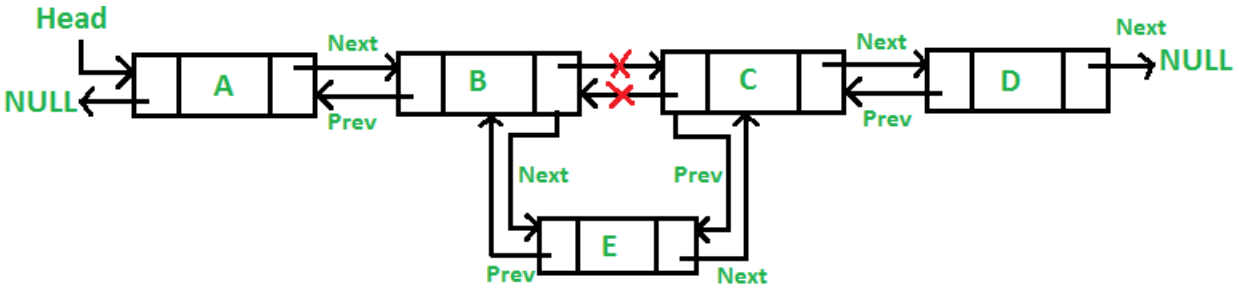
3. Присвоїти значення в поле data.

4. Зробити вказівник next нового вузла наступним попереднього вузла.

5. Змінити вказівник next вузла prev_node на новий вузол new_node.

6. Установити вказівник на попередній вузол нового вузла new_node на вузол prev_node.

7. Змінити попередній вузол вузла new_node на вузол new_node.



3) Додати вузол у кінці списку:

1. Створити новий вузол.

2. Присвоїти значення в поле data.

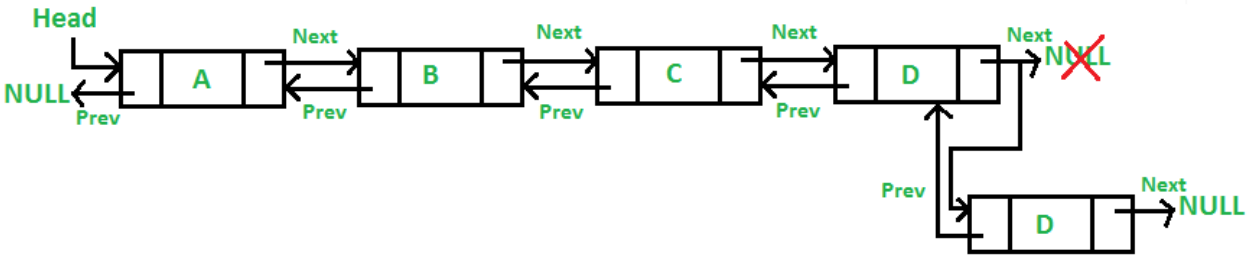
3. Цей новий вузол буде останнім вузлом, отже присвоїти вказівнику next вузла new_node значення NULL.

4. Якщо пов’язаний список порожній, зробити новий вузол головою.

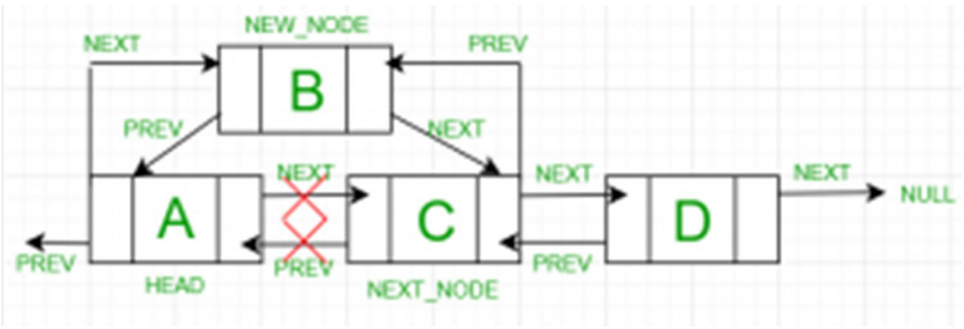
5. Інакше перехід до останнього вузла списку.

6. Змінити вказівник next останнього вузла списку на new_node.

7. Зробити останній вузол списку як вказівник prev для нового вузла new_node.



4) Додати вузол перед даним вузлом списку:



1) Перевірити, чи вказівник next_node дорівнює NULL чи ні. Якщо дорівнює NULL, то вийти з алгоритму, оскільки будь-який новий вузол не можна додати перед NULL

2) Виділити пам’ять для нового вузла, нехай він буде називатися new_node

3) Присвоїти значенню поля data нового вузла значення new_data.

4) Установити попередній покажчик вузла new_node, як попередній вузол вузла next_node.

5) Установити попередній покажчик вузла next_node як вузол new_node.

6) Установити наступний покажчик вузла new_node як наступний вузол next_node.

7) Якщо попередній вузол new_node не NULL, тоді встановити наступний покажчик цього попереднього вузла як new_node (new_node->prev->next = new_node).

8) Інакше, якщо prev для new_node дорівнює NULL, це буде новий головний вузол. Отже, слід установити (*head_ref) = new_node.

Алгоритм видалення вузла з двозв'язного списку:

Нехай вузол, який потрібно видалити, буде del.

1. Якщо вузол, який потрібно видалити, є головою списку, змінити вказівник голови на наступний вузол.

2. Якщо наступний вузол за del існує, то зв'язати його з попереднім до del вузлом.

3. Якщо del не голова списку, то зв'язати попередній вузол із наступним за del вузлом.

```
// Повна програма C++ для демонстрації всіх методів
#include <bits/stdc++.h>
using namespace std;
// Зв'язаний вузол списку
struct Node {
  int data;
  Node* next;
  Node* prev;
};
/* Дано покажчик (вказівник на вказівник) на голову списку
та int елемент,
вставляє новий вузол на початок списку. */
void push(Node** head_ref, int new_data) {
/* 1. Створити новий вузол */
  Node* new_node = new Node();
/* 2. Увести дані */
  new_node->data = new_data;
/* 3. Зробити покажчик next нового вузла на голову, а prev
— NULL */
  new_node->next = (*head_ref);
```

```
  new_node->prev = NULL;
/* 4. Змінити покажчик prev вузла голови на новий вузол */
  if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;
/* 5. Перемістіть голову на новий вузол */
  (*head_ref) = new_node;
}
/* Дано вузол як prev_node, вставити новий вузол після
даного вузла */
void insertAfter(Node* prev_node, int new_data) {
  /*1. Перевірити, чи даний prev_node має значення NULL */
  if (prev_node == NULL) {
    cout<<"Даний попередній вузол не може бути NULL";
    return;
  }
 /* 2. Виділити новий вузол */
  Node* new_node = new Node();
/* 3. Увести дані */
  new_node->data = new_data;
/* 4. Указати покажчику next нового вузла на покажчик next
попереднього_вузла */
  new_node->next = prev_node->next;
/* 5. Зробити покажчик next prev_node як new_node */
  prev_node->next = new_node;
/* 6. Зробити prev_node як попередній для new_node */
  new_node->prev = prev_node;
/*  7.  Змінити  попередній  вузол  наступного  вузла  для
new_node */
  if (new_node->next != NULL)
    new_node->next->prev = new_node;
}
/*  Дано  покажчик  (вказівник  на  вказівник)  на  голову
ДВОЗВ'ЯЗНОГО СПИСУ та int елемент, додає новий вузол у
кінці */
void append(Node** head_ref, int new_data) {
 /* 1. виділити вузол */
  Node* new_node = new Node();
  Node* last = *head_ref; /* used in step 5*/
 /* 2. ввести дані */
  new_node->data = new_data;
```

```
  /* 3. Цей новий вузол буде останнім вузлом, отже
присвоїти покажчику next вузла new_node значення NULL*/
  new_node->next = NULL;
/* 4. Якщо зв'язний список порожній, зробити новий вузол
головою */
  if (*head_ref == NULL)  {
    new_node->prev = NULL;
    *head_ref = new_node;
    return;  }
/* 5. Інакше перехід до останнього вузла */
  while (last->next != NULL)
    last = last->next;
/* 6. Установити покажчик next останнього вузла на new_node
*/
  last->next = new_node;
/* 7. Установити покажчик prev нового вузла на останній
вузол списку */
  new_node->prev = last;
  return;
}

   /* Функція для видалення вузла в двозв'язному списку.
    head_ref - покажчик голове списку.
    del - покажчик на вузол, який потрібно видалити. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
/* базовий випадок */
    if (*head_ref == NULL || del == NULL)
        return;

 /* Якщо вузол, який потрібно видалити, є головою списку
*/
    if (*head_ref == del)
        *head_ref = del->next;


    /* Змінити покажчик next, лише якщо вузол, який
потрібно видалити,
    НЕ останній вузол */
    if (del->next != NULL)
```

```
        del->next->prev = del->prev;

       /* Змінити покажчик prev, лише якщо вузол, який буде
видалено,
        НЕ є першим вузлом */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Нарешті, звільнити пам'ять, зайняту del*/
    free(del);
}

/* Функція для видалення вузла в заданій позиції
    у двозв'язному списку */
void deleteNodeAtGivenPos(struct Node** head_ref, int n)
{
/* якщо список має значення NULL або вказано недійсну
позицію */
    if (*head_ref == NULL || n <= 0)
        return;

    struct Node* current = *head_ref;
    int i;

 /* перехід до вузла в позиції 'n' від початку */
    for (int i = 1; current != NULL && i < n; i++)
        current = current->next;

  /* якщо 'n' більше, ніж кількість вузлів у двозв'язному
списку */
    if (current == NULL)
        return;

 /* видалити вузол, на який вказує 'current' */
    deleteNode(head_ref, current);
}

/* Ця функція друкує вміст зв’язаного списку, починаючи з
даного вузла */
void printList(Node* node) {
```

```
  Node* last;
  cout<<"\nОбхід у прямому напрямку\n";
  while (node != NULL)   {
    cout<<" "<<node->data<<" ";
    last = node;
    node = node->next;  }
    cout << endl;
  cout<<"\nОбхід у зворотному напрямку\n";
  while (last != NULL)   {
    cout<<" "<<last->data<<" ";
    last = last->prev;  }
    cout << endl;
}

/*  Основна програма для тестування вищезазначених
функцій*/
int main() {
  setlocale(LC_ALL, "ukr");
/* Почати з порожнього списку */
  Node* head = NULL;
// Вставити 6. Таким чином зв’язаний список стає 6->NULL
  append(&head, 6);
// Вставити 7 на початку. Таким чином зв’язаний список стає
7->6->NULL
  push(&head, 7);
 // Вставити 1 на початку. Таким чином зв’язаний список
стає 1->7->6->NULL
  push(&head, 1);
 // Вставити 4 у кінці. Таким чином зв’язаний список стає
1->7->6->4->NULL
  append(&head, 4);

  cout << "Створений двозв\'язний список: ";
  printList(head);

  // Вставити 8 після 7. Таким чином пов’язано
  // список стає 1->7->8->6->4->NULL
  insertAfter(head->next, 8);
  cout << "Список після вставляння 8 після 7: ";
  printList(head);
```

```
  //видалення із списку 7
  deleteNodeAtGivenPos(&head, 2);
  cout << "Список після видалення 2-го вузла: ";
  printList(head);

  return 0;
}
```

Аналіз складності:

Часова складність: O(1). Оскільки обхід двозв'язного списку не потрібен, то час складності є постійним.

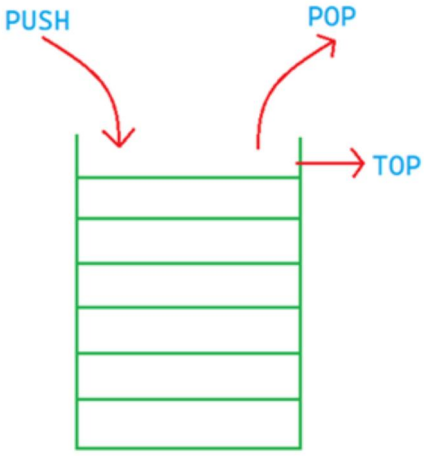
Просторова складність: O(1). Оскільки додатковий простір не потрібен, тому складність простору є постійною.

3. Стек

Означення. Стек — це лінійна структура даних, яка дотримується певного порядку виконання операцій.

Порядок може бути LIFO (останній прийшов, перший вийшов) або FILO (першим прийшов останній вийшов).

Схематично стек можна візуалізувати так, як подано на малюнку.



В основному в стеку виконуються такі три основні операції:

Push: додає елемент у стек. Якщо стек заповнений, то це називається умовою переповнення.

Pop: видаляє елемент зі стека. Елементи видаляються в зворотному порядку до того, у якому вони записувалися в стек. Якщо стек порожній, це називається умовою недостатнього наповнення.

Peek або **Top**: повертає верхній елемент стека.

isEmpty: повертає true, якщо стек порожній, інакше повертає false.

Як практично зрозуміти стек?

Є багато реальних прикладів стека. Розглянемо простий приклад стосу тарілок, складених одна над одну в кафе. Тарілка, яка знаходиться вгорі, береться першою, тобто тарілка, яка була розміщена найнижчій позиції, залишається в стосу найдовше. Отже, можна просто побачити, що слідує порядку LIFO/FILO.

Наведемо одну з можливих реалізацій стеку:

```
// C++ програма для зв'язаного списку реалізації стека
#include <bits/stdc++.h>
using namespace std;
```

```
// Структура для представлення стека
struct StackNode {
    int data;
    StackNode* next;
};

StackNode* newNode(int data)
{
    StackNode* stackNode = new StackNode();
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(StackNode* root)
{
    return !root;
}

void push(StackNode** root, int data)
{
    StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    cout << data << " занесено до стека\n";
}

int pop(StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}
int peek(StackNode* root)
```

```
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

int main()
{
    StackNode* root = NULL;
    setlocale(LC_ALL, "ukr");
    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    cout << pop(&root) << " вилучено зі стека\n";

    cout << "Верхній елемент є " << peek(root) << endl;

    cout<<"Елементи в стеку : ";
    //надрукувати всі елементи стеку:
    while(!isEmpty(root))
    {
        // надрукувати верхній елемент у стеку
        cout<<peek(root)<<" ";
        // видалити верхній елемент у стеку
        pop(&root);
    }

    return 0;
}
```

Часові складності операцій над стеком:

push(), pop(), isEmpty() і peek() займають O(1) часу. Ми не запускаємо жодного циклу в жодній із цих операцій.

Застосування стека на ІТ-практиці:

• Балансування символів;

• Перетворення інфікса на постфікс/префікс;

• Функції повторного скасування в багатьох програмах – редакторах, наприклад, Photoshop;

• Функція переходу вперед і назад у веб-браузерах;

• Використовується в багатьох алгоритмах, таких як Ханойська вежа, обхід дерев тощо;

• Зворотне відстеження є одним із методів проектування алгоритму. Деякими прикладами зворотного відстеження є проблема проблема N-Ферзів, пошук шляху в лабіринті тощо. У них відбувається занурення в певне середовище якимось чином, якщо цей спосіб неефективний, ми повертаємося до попереднього стану і пробуємо піти іншим шляхом. Щоб повернутися з поточного стану, нам потрібно зберегти попередній стан, для цього нам потрібен стек.

• У графових алгоритмах, таких як топологічне сортування та пошуку сильно зв’язаних компонент.

• В управлінні пам’яттю будь-якого сучасного комп'ютера використовується стек, як основний спосіб керування роботою програм. Кожна програма, яка виконується в комп’ютерній системі, має власний розподіл пам’яті у стеці.

• Перевертання рядків також є іншим застосуванням стека. Тут один за одним кожен символ вставляється в стек. Таким чином, перший символ рядка знаходиться в нижній частині стека, а останній елемент рядка – у верхній частині стека. Після виконання операцій pop над стеком ми отримуємо рядок у зворотному порядку.

4. Черга

Означення. **Черга** (англ. *queue*) в програмуванні – динамічна упорядкована структура даних, що працює за принципом «перший прийшов – перший пішов» (англ. *FIFO — first in, first out*).

Черга має ***голову*** (англ. *head*) та ***хвіст*** (англ. *tail*). Новий елемент черги додається у хвіст черги. З черги видаляють елемент, який знаходиться в її голові.

Існує кілька способів реалізації черги:

✓ за допомогою одновимірного масиву;

✓ за допомогою пов'язаного списку;

✓ за допомогою класу об'єктно-орієнтованого програмування.

Найпростіші операції з чергою:

✓ `enqueue()` – додавання елемента x у кінець черги `queue` (`queue` – покажчик на чергу);

✓ `dequeue()` – видалення елемента з голови черги `queue`;

✓ `front()` – виведення першого елемента черги `queue`;

✓ `print()` – виведення елементів черги `queue`.

У наступному прикладі використаємо стандартну бібліотеку STL для реалізації вказаних операцій.

```
#include <iostream>
#include <list>
//Бібліотека забезпечує реалізацію двозв'язного списку
using namespace std;

// Функція для додавання елемента до черги
void enqueue(list<int>& queue, int x) {
    queue.push_back(x);
}

// Функція для видалення елемента з черги
void dequeue(list<int>& queue) {
    if (!queue.empty()) {
        queue.pop_front();
    }
}

// Функція для виведення першого елемента черги
void front(const list<int>& queue) {
    if (!queue.empty()) {
        cout << "Перший елемент черги: " << queue.front()
<< endl;
    }
    else {
        cout << "Черга порожня." << endl;
    }
}


// Функція для виведення стану черги
void print(const list<int>& queue) {
    if (queue.empty()) {
        cout << "Черга порожня." << endl;
    }
    else {
        cout << "Елементи черги: ";
        for (const auto& item : queue) {
            cout << item << " ";
        }
        cout << endl;
    }
}

int main() {
    setlocale(LC_CTYPE, "ukr");
```

```
    list<int> queue; // Створення пустої черги
    do {
        cout << "Оберіть опцію:" << endl;
        cout << "1. Додати елемент до черги" << endl;
        cout << "2. Видалити перший елемент з черги" <<
endl;
        cout << "3. Вивести перший елемент черги" << endl;
        cout << "4. Вивести стан черги" << endl;
        cout << "0. Вийти" << endl;

        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Уведіть елемент: ";
            cin >> item;
            enqueue(queue, item);
            break;
        case 2:
            dequeue(queue);
            break;
        case 3:
            front(queue);
            break;
        case 4:
            print(queue);
            break;
        case 0:
            cout << "Програма завершена." << endl;
            break;
        default:
            cout << "Невірний вибір. Спробуйте ще раз." <<
endl;
            break;
        }
    } while (choice != 0);
     return 0;
}
```

🔔 Контрольні запитання:

1. Що таке зв’язний список?
2. Чим відрізняється однозв’язний список від двозв’язного?
3. Яка особливість побудови стека?
4. Що таке черга?
5. Чим відрізняється черга від стека?

Тема №5. РЕКУРСІЯ

План лекції:

1. Вступ
2. Види рекурсії
3. Особливості програмування рекурсивних функцій

Джерела:

[1, Розділ 4], [3, §9.2-9.4], [4, Глава 5], [6, Chapter 5], [11, §1.3]

1. Вступ

Означення. Рекурсивним називається спосіб побудови об'єкта, у якому визначення об'єкта включає аналогічний об'єкт у вигляді його частини.

Означення. Функція називається **рекурсивною**, якщо її значення для даного аргументу визначається через значення тієї ж функції для попередніх аргументів.

Означення. Процес, у якому функція викликає себе прямо чи опосередковано, називається **рекурсією**, а відповідна функція називається **рекурсивною функцією**.

Будь-яке рекурсивне завдання слід розбити на ряд етапів. При цьому, у рекурсивній функції однозначно визначено, як розв'язати найпростішу частину завдання – базову. Базову частину рекурсивної функції ще називають якорем.

Якщо функція викликається для розв'язання базового завдання, то вона повертає результат. В іншому випадку, для вирішення складнішого завдання, функція ділить це завдання на дві частини: одну частину, яку функція вміє розв'язувати, та іншу, яку функція розв'язувати не вміє.

Щоб зробити рекурсію виконуваною, остання частина має бути схожою на початкове завдання, але бути в порівнянні з нею простішою і меншою. Оскільки це нове завдання подібне до початкового, функція викликає копію самої себе, щоб почати працювати над меншою проблемою – це називається **рекурсивним викликом** або **кроком рекурсії**.

Крок рекурсії виконується до того часу, поки вихідне звернення до функції не закривається, тобто доки не закінчено виконання функції.

Щоб завершити процес рекурсії, щоразу, як функція викликає саму себе з меншими значеннями аргументів, що зводяться до якоря. У цей момент функція розпізнає базове завдання, повертає результат попередньої функції, і послідовність повернень повторює весь шлях назад, доки не дійде до початкового виклику і не поверне кінцевий результат.

Техніка рекурсії – це корисна техніка, яка запозичена з математики. Рекурсивний код, як правило, коротший і його легше писати, ніж ітеративний

код. Рекурсія найкорисніша для завдань, які можна визначити в термінах подібних підзадач.

Отже, рекурсивне визначення складається з двох частин:

✓ У якірній частині перераховані основні елементи, які є конструкторськими блоками всіх інших елементів набору.

✓ У другій частині наводяться правила, які дозволяють будувати нові об'єкти з основних елементів або об'єктів, які вже були побудовані.

Наприклад, для побудови множини натуральних чисел виділяється один якірний елемент 1, а операція збільшення на 1 задається так:

1. 1 ∈ N;

2. якщо n ∈ N, то (n + 1) ∈ N;

3. у множині N немає інших об'єктів.

Зручніше використовувати наступне визначення, яке охоплює весь спектр арабської цифрової спадщини:

1. 1, 2, 3, 4, 5, 6, 7, 8, 9 ∈ N;

2. якщо n ∈ N, то n1, n2, n3, n4, n5, n6, n7, n8, n9 ∈ N;

3. це єдині натуральні числа.

Рекурсивні визначення мають дві цілі:

1) створення нових елементів,
2) перевірити, чи належить елемент до набору.

У разі тестування проблема вирішується шляхом зведення її до більш простої задачі, а якщо простіша задача все ще занадто складна, її зводять до ще більш простої задачі, і так далі, доки вона не буде зведена до задачі, зазначеної в якір.

Приклад 1. Визначити суму перших n натуральних чисел.

Існує кілька способів зробити це, але найпростіший підхід – просто додати числа, починаючи від 1 до n.

Отже, функція виглядає просто так:

Підхід (1) – просто додавати по одному

f(n) = 1 + 2 + 3 +…+ n

Підхід (2) – Рекурсивне додавання

$$f(n) = \begin{cases} 1, & if\ n = 1, (\text{якір}) \\ n + f(n-1), if\ n > 1\ (\text{індуктивний крок}). \end{cases}$$

Напишемо на основі підходу(1) програму на C++:

```
#include <iostream>
using namespace std;
int main()
{
```

```
  long n,sum = 0;
  cout << "Enter the number n: " << endl;
  cin >> n;
  for (int i = 1; i <= n; i++)
    sum += i;
  cout << "sum(1+..." << n <<")= " << sum << endl;
  return 0;
}
```

Давайте напишемо програму на C++ на основі підходу (2):

```
#include <iostream>
using namespace std;
long sum( int n){
  if (n == 1) return 1;
  else
   return n +sum(n-1);
}

int main() {
  long n;
  cout << "Enter the number n: " << endl;
  cin >> n;
  cout << "sum(1+..." << n <<")= " << sum(n) << endl;
  return 0;
}
```

Зробимо аналіз складності алгоритму:

T(1) = k1.

Якщо n > 1, функція виконає фіксовану кількість операцій k2 і, крім того, зробить рекурсивний виклик Sum(n-1).

Загалом T(n) = k2 + T(n-1).

Нехай k1 = k2 = 1.

Щоб знайти часову складність для функції суми, можна потім звести до розв’язування рекурентного відношення

T(1) = 1, (*)

T(n) = 1 + T(n-1), коли n > 1. (**)

Повторно застосовуючи ці співвідношення, ми можемо обчислити T(n) для будь-якого додатного числа n.

T(n) = (**)
1 + T(n-1) = (**)
1 + (1 + T(n-2)) = 2 + T(n-2) = (**)
2 + (1 + T(n-3)) = 3 + T(n-3) = (**)
…
k + T(n-k) = …= (**)
n – 1 + T(1) = (**)
n – 1 + 1= Θ(n)

2. Види рекурсії

Є кілька класифікацій рекурсії. Зокрема, виділяють:

- ✓ лінійну;
- ✓ змішану;
- ✓ розгалужену;
- ✓ вкладену.

Лінійна рекурсія

Найпростішим прикладом рекурсії є лінійна рекурсія, при якій функція містить єдиний умовний виклик себе. У такому разі рекурсія стає еквівалентною звичайному циклу. Будь-який циклічний алгоритм можна перетворити на лінійно-рекурсивний і навпаки. Приклад лінійної рекурсії є обчислення факторіалу: n! = 1 · 2 · 3 ... (n - 1) · n = (n - 1)! · n. Математично можна записати обчислення факторіалу у такому вигляді:

$$n! = \begin{cases} 1, \ if\ n = 0, \ (\text{якір}) \\ n \cdot (n-1)!, \ if\ n > 0 \ (\text{індуктивний крок}). \end{cases} \quad (5.1)$$

У першому рядку формули (5.1) явно зазначено, як вирахувати факторіал, якщо аргумент дорівнює нулю. У будь-якому іншому випадку для обчислення n! необхідно обчислити попереднє значення (n - 1)! та помножити його на n. Значення, що зменшується, гарантує, що врешті-решт виникне необхідність знайти 0!, який обчислюється безпосередньо.

Приклад 2. Функцію n! факторіалу можна визначити таким чином:

$$n! = \begin{cases} 1, \quad if\ n = 0, \ (\text{якір}) \\ n \cdot (n-1)!, \ if\ n > 0 \ (\text{індуктивний крок}). \end{cases}$$

Рекурсивна програма на C++ може мати вигляд:

```
#include <iostream>
using namespace std;
unsigned long long fact(int n){
if (n == 0) return 1;
else
```

```
  return n * fact(n-1);
}
int main() {
  int n;
  cout << "Enter the number n: " << endl;
  cin >> n;
  cout << n <<"!= " << fact(n) << endl;
  return 0;
}
```

Щоб зрозуміти, як виконуватиметься ця функція fact(), пригадаємо, що на час виконання допоміжного алгоритму основний алгоритм припиняється. При виклику нової копії рекурсивного алгоритму знову виділяється місце всім змінних, оголошених у ньому, причому змінні інших копій будуть недоступні. При видаленні копії рекурсивного алгоритму з пам'яті видаляються і всі його змінні. Активізується попередня копія рекурсивного алгоритму, що стають доступними її змінні.

Нехай необхідно обчислити 4! Основний алгоритм: вводиться

n = 4, виклик factorial(4) . Основний алгоритм зупиняється, викликається та працює factorial(4): 4 ≠ 0, тому factorial := factorial(3)·4. Робота функції призупиняється, викликається та працює factorial(3): 3 ≠ 0, тому factorial := factorial(2)·3. Зауважте, що на даний момент у пам'яті комп'ютера дві копії функції factorial. Викликається та працює factorial(2): 2 ≠ 0, тому factorial:= factorial(1)·2. У пам'яті комп'ютера вже три копії функції factorial і викликається четвертою. Викликається та працює factorial(1): 1 ≠ 0, тому factorial:= factorial(0)·1. Викликається та працює factorial(0): 0 = 0 тому factorial(0)= 1. Робота цієї функції завершена, продовжує роботу factorial(1): factorial(1):= factorial(0)·1 = 1 * 1 = 1 . Робота функції завершена та працює factorial(2):= factorial(1) · 2 = 1 · 2 = 2.

Робота цієї функції також завершена і продовжує роботу функція factorial(3): factorial(3):= factorial(2) · 3 = 2 · 3 = 6.

Завершується робота цієї функції, і продовжує роботу функція factorial(4): factorial(4):= factorial(3) · 4 = 6 · 4 = 24 .

Управління передається до основної програми.

Лінійна рекурсія – найбільш простий і найпоширеніший вид рекурсії.

Змішана рекурсія

У цьому вигляді рекурсії дві чи більше функції викликають одна одну циклічно. Умови завершення можуть міститись у всіх або в одній з функцій.

Приклад 3. Розглянемо завдання визначення парності числа. Число є парним, якщо попереднє число непарне, і навпаки – число непарне, якщо попереднє парне. Математично:

$$isOdd(n) = \begin{cases} \text{false}, n = 0, \\ \text{isEven}(n-1), n > 0; \end{cases}$$

$$isEven(n) = \begin{cases} true, n = 0, \\ \text{isO}dd(n-1), n > 0. \end{cases}$$

Код на C++ буде виглядати так:

```
bool isEven(int n)
{
  if (n == 0)  // умова закінчення
   return true;
  else
   return isOdd(n - 1);
}

bool isOdd(int n)
{
  if (n == 0)  // умова закінчення
   return false;
  else
   return isEven(n - 1);
}
```

Графічно схема виконання функцій для n = 4 представлена на рис. 5.1.

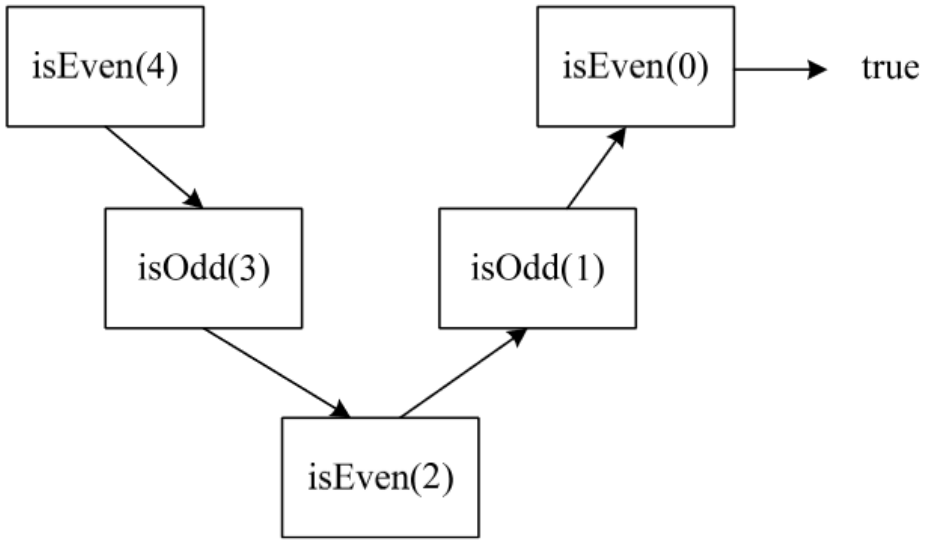


Рис. 5.1. Схема виконання функцій перевірки парності

Розгалужена рекурсія

У разі розгалуженої рекурсії функція викликається більше одного разу. Окремий випадок цього виду – бінарна рекурсія (виклик інших функцій).

Приклад 4. Розглянемо приклад – обчислення послідовності чисел Фібоначчі. Послідовність чисел Фібоначчі: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... починається з 0 і 1 та має ту властивість, що кожен наступний член послідовності є сумою двох попередніх членів. Ця послідовність часто зустрічається у природі, зокрема вона описує форму спіралі. Математично послідовність Фібоначчі можна записати так:

$$Fibonacci(n) = \begin{cases} 0, & n = 0; \\ 1, & n = 1; \\ \text{Fibonacci(n} - 1)\text{Fibonacci(n} - 2), & \text{n} > 1. \end{cases}$$

Код програми буде виглядати так:

```
int Fibonacci (int n)
{
 if (n < 0) return -1; //помилка
 if (n == 0)
  return 0;
 else
if (n == 1)
return 1;
else
 return Fibonacci (n-1)+ Fibonacci (n-2);
}
```

На рис. 5.2 наведено схему обчислення чисел Фібоначчі для n = 3.

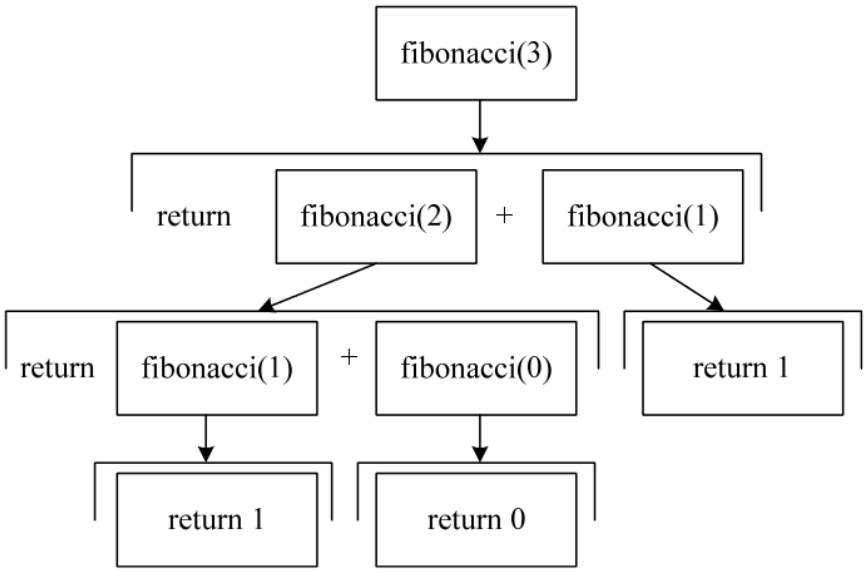


Рис. 5.2. Схема виклику функції обчислення чисел Фібоначчі

Розглянемо часову складність даного алгоритму:

T(n) = T(n-1) + T(n-2), що є експоненційним часом.

T(n)=T(n-1)+T(n-2)+O(1).

Це означає, що час обчислення Fibonacci(n) дорівнює сумі часу, необхідного для обчислення Fibonacci(n-1) і Fibonacci(n-2).

Розв'язуючи наведене вище рекурсивне рівняння, ми отримуємо верхню межу Фібоначчі як $O(2^n)$, але це не жорстка верхня межа.

Оскільки

$$T(n) = O(\left(\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right),$$

то ми можемо записати так:

$$T(n) = O((\left(\frac{1+\sqrt{5}}{2}\right)^n)$$

$$T(n) = O((1.6180)^n)$$

Вкладена (гніздова) рекурсія

Розглянуті вище рекурсії можуть бути замінені на ітераційний цикл або ітераційний цикл зі стеком. Однак даний вид рекурсії складно уявити у вигляді циклічної конструкції.

Більш складний випадок рекурсії зустрічається у визначеннях, коли функція не визначається лише в термінах сама по собі, але також використовується як один із параметрів.

Приклад 5. Потрібно обчислити за невід'ємним цілим числом m функцію Акермана. Вона визначається таким чином:

$$Akkerman(n, m) = \begin{cases} n+1, & m = 0; \\ \text{Akkerman}(\text{m}-1,1), & m > 0, n = 1; \\ \text{Akkerman}\big(\text{m}-1, \text{Akkerman}(\text{m}, n-1)\big), & m > 0\ n > 0. \end{cases}$$

Код програми буде виглядати:

```
int Ackkerman(int m, int n)
{
  if (m < 0 || n < 0)
return -1;
  if (m == 0)
return n+1;
  else       //лінійна рекурсія
```

```
if (m > 0 && 0 == n)
return Ackkerman(m-1, 1);
//Вкладена рекурсія
else
return Ackkerman(m-1, Ackkerman(m, n-1));
}
```

Є й інша класифікація рекурсії. Рекурсію поділяють на пряму та непряму. Пряму рекурсію ділять на:

- ✓ хвостову;
- ✓ рекурсію в голові;
- ✓ деревоподібну
- ✓ вкладену.

Хвостова рекурсія

Означення. Хвостова рекурсія характеризується використанням лише одного рекурсивного виклику в самому кінці реалізації функції.

Наприклад:

```
void tail(int i) {
   if (i > 0) {
    cout << i << ' ' ;
    tail(i-1);
   }
  }
```

Функція визначена як не хвостова рекурсія:

```
  void nonTail(int i)
{
   if (i > 0) {
    nonTail(i-1);
    cout << i << ' ' ;
    nonTail(i-1);
}
}
```

Хвостові рекурсивні функції вважаються кращими, ніж нехвостові рекурсивні функції, оскільки хвостова рекурсія може бути оптимізована компілятором. Компілятори зазвичай виконують рекурсивні процедури за допомогою стека. Цей стек складається з усієї відповідної інформації,

включаючи значення параметрів, для кожного рекурсивного виклику. Коли процедура викликається, її інформація поміщається в стек, а коли функція завершується, інформація повертається зі стеку. Таким чином, для функцій без хвостової рекурсії глибина стека (максимальна кількість простору стека, що використовується в будь-який момент під час компіляції) є більшою. Ідея, яку використовують компілятори для оптимізації хвостово-рекурсивних функцій, проста, оскільки рекурсивний виклик є останнім оператором, у поточній функції не залишається нічого робити, тому збереження фрейму стека поточної функції не потрібно.

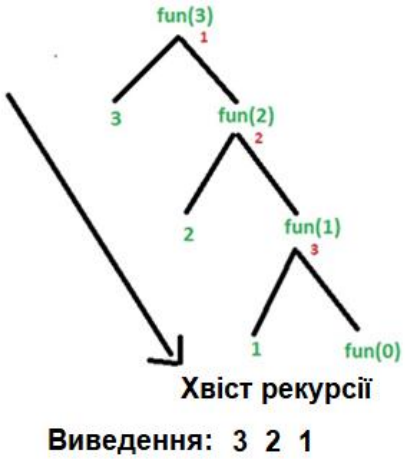


Рис. 5.3. Демонстрація виклику хвостово-рекурсивної функції

```
// Програма, яка демонструє хвостову рекурсію
#include <iostream>
using namespace std;
// Рекурсивна функція
void fun(int n){
  if (n > 0) {
    cout << n << " ";
    // Останній оператор у функції
    fun(n - 1);
  }
}
// Driver Code
int main(){
  int x;
  setlocale(0,".1251");
  cout << "Уведіть ціле число x"<<endl;
  cin >> x;
  fun(x);
  return 0;
```

```
}
```

Якщо x=3, то виклики функції демонструються на рис. 5.3.

Часова складність для хвостової рекурсії: O(n).

Просторова складність для хвостової рекурсії: **O(n).**

Примітка: складність часу та простору наведена для цього конкретного прикладу. Для іншого прикладу це може відрізнятися.

Рекурсія в голові

Якщо рекурсивна функція, яка викликає саму себе, і цей рекурсивний виклик є першим оператором у функції, тоді вона відома як **рекурсія в голові (Head Recursion).**

У таких алгоритмах немає жодного оголошення, жодної операції перед рекурсивним викликом. Функція не повинна обробляти або виконувати будь-які операції під час виклику, і всі операції виконуються під час повернення.

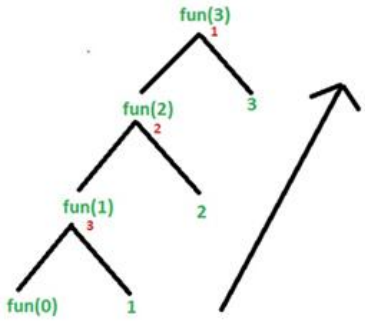


Рис. 5.4. Демонстрація виклику функції Head Recursion

```
// C++ програма, що демонструє Head Recursion
#include <iostream>
using namespace std;
// Recursive function
void fun(int n){
  if (n > 0) {
    // Перший оператор у функції
    fun(n - 1);
    cout << " "<< n;
  }
}
// Driver code
int main(){
  int x = 3;
  fun(x);
```

```
  return 0;
}
```

Часова складність для рекурсії в голові: O(n).

Просторова складність для рекурсії в голові: O(n).

Деревоподібна рекурсія

Якщо рекурсивна функція викликає себе більше одного разу, вона відома як **деревоподібна рекурсія**.

Означення. Функція з кількома рекурсивними викликами називається **дереворекурсивною**.

Оскільки кожен виклик розгалужується на кілька менших викликів, кожен з яких розгалужується на ще менші виклики, так само, як гілки дерева стають меншими, але більш численними, коли вони відходять від стовбура. Дерево рекурсії корисно для візуалізації того, що відбувається під час повторення.

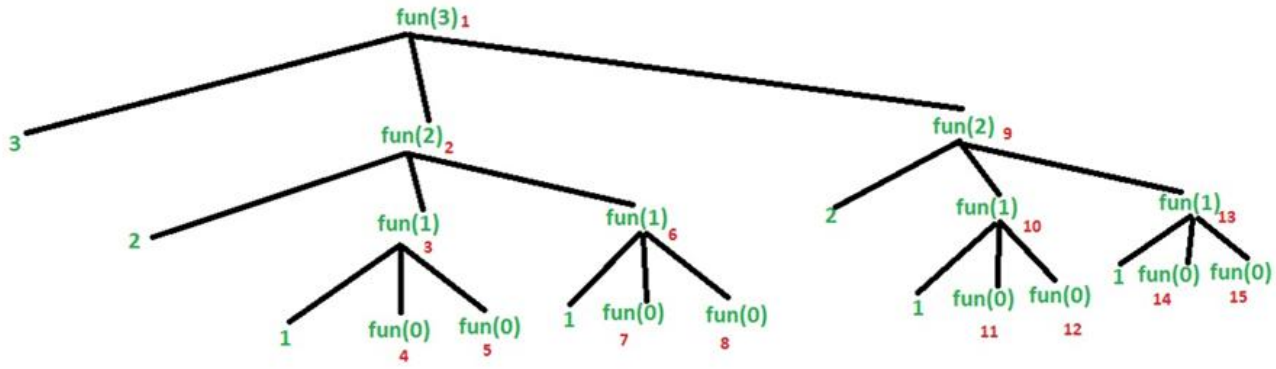


Рис. 5.5. Демонстрація виклику дерево-рекурсивної функції

```
// C++ програма, що демонструє Tree Recursion
#include <iostream>
using namespace std;
// Рекурсивна функція
void fun(int n){
  if (n > 0)  {
    cout << " " << n;
    // Виклик функції перший
    fun(n - 1);
    // Виклик функції другий
    fun(n - 1);
  } }
// Driver code
int main(){
  fun(3);
```

```
  return 0;
}
```

Виклики функції демонструються на рис. 5.

Часова складність для рекурсії в голові: $O(2^n)$.

Просторова складність для рекурсії в голові: $O(n)$.

Непряма рекурсія

Непряма рекурсія (або взаємна рекурсія) виникає, коли функція викликає іншу функцію, що в кінцевому підсумку призводить до повторного виклику вихідної функції.

Ланцюжок проміжних викликів може мати довільну довжину, наприклад:

f() -> f1() -> f2() ->... -> fn() -> f()

Існує також ситуація, коли f() може викликати себе опосередковано через різні ланцюжки. Таким чином, на додаток до щойно вказаного ланцюга може бути можливий інший ланцюг. Наприклад

f() -> g1() -> g2() -> ... -> gm() -> f()

Прикладом цієї ситуації можуть бути три функції, які використовуються для декодування інформації. Функція receive() зберігає вхідну інформацію в буфері, decode() перетворює її в розбірливу форму, а store() зберігає її у файлі. receive() заповнює буфер і викликає decode(), який, у свою чергу, після завершення своєї роботи подає буфер з декодованою інформацією до store(). Після того, як store() виконає свої завдання, вона викликає receive(), щоб перехопити більше закодованої інформації, використовуючи той самий буфер. Отже, маємо ланцюжок викликів

receive() → decode() → store() → receive() → decode() → ...

який закінчується, коли не надходить нова інформація. Ці три функції працюють у такий спосіб:

```
receive(buffer)
    while buffer is not filled up
        if information is still incoming
        get a character and store it in buffer;
         else   exit();
 decode(buffer);

decode(buffer)
    decode information inbuffer;
    store(buffer);
```

```
store(buffer)
transfer information frombuffer  to file;
receive(buffer);
```

3. Особливості програмування рекурсивних функцій

Принципи програмування рекурсивних функцій

Принцип програмування рекурсивних функцій має багато спільного з методом математичної індукції. Цей метод, який використовується для підтвердження правильності встановлення для нескінченної послідовності стану і неявно застосовується при розробці рекурсивних функцій. Дійсно, сама рекурсивна функція є переходом з n-го в (n + 1)-ий стан деякого процесу.

Якщо цей перехід коригується, то є дотримання деяких умов на вхідних функціях і відповідає їх дотриманню на вихідному (що є в рекурсивному виклику), ці умови будуть підтримуватися по всьому ланцюжку поточного стану (при безумовній коректності початкового).

Спираючись на метод математичної індукції, можна сформулювати наступні правила:

• рекурсивна функція розробляється як узагальнений крок процесу, що викликається у випадкових початкових умовах і який призводить до наступного кроку в деяких нових умовах;

• узагальнені початкові умови кроку – формальні характеристики функції;

• початкові умови наступного кроку – фактичні параметри рекурсивного виклику;

• рекурсивна функція повинна перевіряти умови завершення рекурсії, при яких наступний етап процесу не виконується;

• локальними змінними функціями повинні бути оголошення всіх змінних, які мають відношення до протікання поточного кроку процеса.

Рекурсії та ітерації

Як згадувалося вище, деякі рекурсивні функції можуть бути реалізовані ітеративно. Порівняємо ці два підходи.

Як ітерації, так і рекурсії включають повторення: ітерації використовують структуру повторення явним чином, рекурсії реалізують повторення за допомогою повторних викликів. Як ітерації, так і рекурсії включають перевірку умов закінчення: ітерації закінчуються після порушень умови продовження циклу, рекурсії закінчуються після розпізнавання базової задачі. Як ітерації, так і рекурсії можуть виявитися нескінченими.

Незважаючи на те, що запис рекурсивних функцій буває дуже коротким, вони потребують великих затрат. При кожному виклику в стеку повинні бути

розміщені всі параметри та локальні змінні. На цю роботу (як і на наступне звільнення) витрачається і час, і простір – у програмі відображаються відповідні команди, які виконуються при кожному виклику, пам’ять, що витрачається під стек. Оскільки рекурсивний виклик виконується до завершення виконання функцій – розмір стека збільшується при кожному виклику до тих пір, поки не буде досягнута точка, коли виконується повернення. Розмір стека пропорційний глибині рекурсії.

Глибина рекурсії – це максимальна ступінь вкладеності рекурсивних викликів. У загальному випадку глибина буде залежати від вхідних даних. Тому при розробці рекурсивних функцій необхідно мінімізувати кількість і розміри локальних змінних і параметрів. Існує також й інший недолік – лишні обчислення. Одним із цих методів боротьби з недоліком є скорочення кількості рекурсивних звернень у текст (якщо є можливість замість двох викликів залишити один).

Контрольні запитання:

1. Яка функція називається рекурсивною?
2. Із чого складається рекурсивне визначення?
3. Назвіть класифікацію рекурсивних функцій.
4. Яка рекурсія є лінійною?
5. У чому суть змішаної рекурсії?
6. Яка рекурсія є розгалуженою?
7. Чим характеризується хвостова рекурсія?
8. Назвіть особливості рекурсії в голові.
9. Чим характерна непряма рекурсія?
10. Перелічіть принципи програмування рекурсивних функцій.
11. Дайте означення глибини рекурсії.

Тема №6. АЛГОРИТМИ ПОШУКУ В С++

Питання до розгляду:

1. Лінійний пошук
2. Бінарний пошук
3. Тернарний пошук

Джерела:

[1, §5.3], [2, Розділ 9], [4, Часть 4], [5, Chapter 12, 13, 32], [6, §6.3], [9, Chapter 6], [10, §3.4, 4.9], [11, §4.3]

1. Лінійний пошук

У контексті лінійного пошуку можуть виникати наступні задачі:

- ✓ визначити наявність заданого елементу в масиві (наборі даних);
- ✓ визначити кількість входжень заданого елементу в масиві;
- ✓ визначити номер позиції або масив номерів позицій розміщення заданого елементу в масиві.

Приклад 1. Дано одновимірний невпорядкований масив з цілих чисел, і потрібно перевірити, чи міститься задане число у цьому масиві.

Наприклад,

Вхід: arr[] = {1, 2, 8, 3, 6, 5, 11, 10, 13, 17}

Знайти індекс елемента x = 11;

Вихід: 6

Елемент x присутній під індексом 6

Вхід: arr[] = {1, 2, 8, 3, 6, 5, 11, 10, 13, 17}

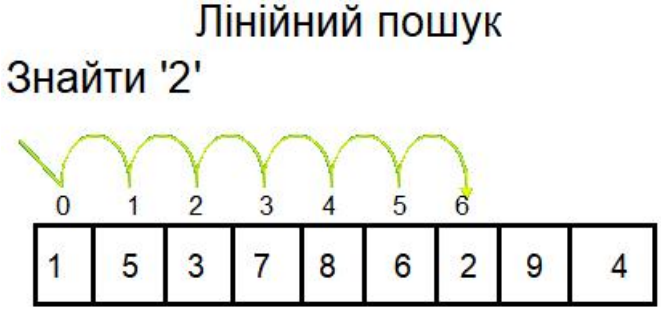
Знайти індекс елемента x = 175;

Вихід: -1

Елемент x відсутній в arr[].

Простим підходом є виконання **лінійного пошу**ку:

1. Почати з крайнього лівого елемента arr[] і порівняти x з кожним елементом arr[]
2. Якщо x збігається з елементом, повернути індекс.
3. Якщо x не збігається з жодним із елементів, повернути -1.



Реалізуємо даний алгоритм на С++:

```
#include <iostream>
#include <iomanip>
#include <ctime>
using namespace std;

//прототипи функцій
int linSearch(int arr[], int requiredKey, int size);
// Лінійний пошук
void showArr(int arr[], int size); // показ масиву

int main()
{
 setlocale(LC_ALL, "ukr");
 const int arrSize = 50;
 int arr[arrSize];
 int requiredKey = 0; // Шукане значення (ключ)
 int nElement = 0; // Номер елементу масиву
 srand(time(NULL));

 // Запис випадкових чисел у масив від 1 до 50
 for (int i = 0; i < arrSize; i++)
 {
 arr[i] = 1 + rand() % 50;
 }

 showArr(arr, arrSize);

 cout << "Яке число необхідно шукати?";
 cin >> requiredKey; // Введення шуканого числа

 //Пошук шуканого числа та запис номера елемента
 nElement = linSearch(arr, requiredKey, arrSize);
 if (nElement != -1)
 {
 /*якщо у масиві знайдено шукане число - виводимо індекс
елемента на екран */
 cout << "Значення " << requiredKey << " знаходиться в
комірці з індексом: " << nElement << endl;
 }
 else
```

```
 {
 //якщо у масиві не знайдено шукане число
 cout << "У масиві немає такого значення" << endl;
 }
 return 0;
}

//Виведення масиву на екран
void showArr(int arr[], int arrSize)
{
      for (int i = 0; i < arrSize; i++)
      {
         cout << setw(4) << arr[i];
         if ((i + 1) % 10 == 0)
         {
             cout << endl;
         }

      }
      cout << endl << endl;
}

//лінійний пошук
int linSearch(int arr[], int requiredKey, int arrSize)
{
 for (int i = 0; i < arrSize; i++)
 {
      if (arr[i] == requiredKey)
      return i;
 }
 return -1;
}
```

Функція, що виконує лінійний пошук linSearch повертає в програму -1 у разі, якщо значення, яке шукає користувач, не знайдено у масиві. Якщо значення буде знайдено – функція поверне індекс елемента масиву, у якому це значення зберігається.

Часова складність вищевказаного алгоритму $O(n)$. Лінійний пошук рідко використовується практично, оскільки інші алгоритми пошуку, такі як алгоритм

двійкового пошуку та хеш-таблиці, дозволяють значно швидший пошук порівняно з лінійним.

Зауважимо, що для задання ширини відображення елементів приєднана бібліотека ***iomanip*** та використана функція ***setw***.

Щоб повністю задавати випадковим чином елементи масиву, використано функцію ***srand***, яка виконує ініціалізацію генератора випадкових чисел ***rand***. Для того, щоб генерувати випадкові числа, функція ***srand*** зазвичай ініціалізується деякими різними значеннями, наприклад, такі значення генеруються функцією ***time*** бібліотеки ***ctime***.

Зазвичай лінійний пошук застосовується для одиночного пошуку в невеликому масиві, який не відсортовано. В інших випадках, краще та ефективніше спочатку відсортувати масив та застосовувати інші алгоритми пошуку. Наприклад, двійковий (бінарний) пошук або інші.

Лінійний пошук можна дещо покращити. Для цього можна розглянути "бар'єрний" метод, який може бути корисним у багатьох завданнях. Для використання бар'єрного методу масив повинен мати один додатковий елемент (тобто його довжина має бути не меншою, ніж n + 1 елемент). Зазначимо, що у такий спосіб можна шукати лише перше входження елемента:

```
a [n + 1] = k;
for (i = 0; a[i] != k; i++);
```

Якщо елемент ***k*** зустрічається в масиві, то його індекс буде перебувати в змінній ***i***, якщо такий елемент у масиві не зустрічається, то ***i*** буде дорівнює ***n+1***.

Покращення складності лінійного пошуку в найгіршому випадку

Якщо елемент знайдено, то складність алгоритму буде в межах від O(n) до O(1).

Можна здійснювати пошук з лівого та правого краю масиву. Це те саме, що й попередній метод, оскільки тут ми виконуємо 2 операції «якщо» за одну ітерацію циклу, а в попередньому методі ми виконали лише 1 операцію «якщо». Це робить обидві часові складності однаковими.

```
#include<bits/stdc++.h>
using namespace std;

void search(vector<int> arr, int search_Element)
{
  int left = 0;
  int length = arr.size();
  int position = -1;
   int right = length - 1;
```

```
  // Виконуємо цикл від 0 вправо
  for(left = 0; left <= right;)
  {

   // Якщо search_element знайдено з лівого краю
    if (arr[left] == search_Element)
    {

      position = left;
      cout << "Елемент знайдено у масиві в "
         << position + 1 << " позиції з "
         << length << " елементів" << endl;

      break;
    }

   // Якщо search_element знайдено з правого краю
    if (arr[right] == search_Element)
    {
      position = right;
      cout << "Елемент знайдено у масиві в "
         << position + 1 << " позиції з "
         << length << " елементів" << endl;

      break;
    }
    left++;
    right--;
  }

  // Якщо елемент не знайдено
  if (position == -1)
    cout << "Не знайдено в масиві з "
       << length << " елементами" << endl;;
}

// Основна програма
int main()
{
```

```
    vector<int> arr{ 1, 2, 3, 4, 5 };
    setlocale(LC_ALL, "ukr");
    int search_element = 5;


    // Виклик функції
    search(arr, search_element);

    search_element = 20;
    // Виклик функції
    search(arr, search_element);

    return 0;
}
```

2. Бінарний пошук

Означення. ***Бінарний пошук*** – це алгоритм пошуку, який використовується у відсортованому масиві шляхом багаторазового ділення інтервалу пошуку навпіл.

Ідея двійкового пошуку полягає в тому, щоб використовувати інформацію про сортування масиву та зменшити часову складність до $O(Log\ n)$.

Двійковий пошук можна використовувати тільки в тому випадку, якщо є масив, всі елементи якого впорядковані (відсортовані). Бінарний пошук не використовується для пошуку максимального або мінімального елементів, так як в відсортованому масиві ці елементи містяться на початку і в кінці масиву відповідно, залежно від того як відсортований масив, за зростанням або за спаданням. Тому алгоритм бінарного пошуку застосуємо, якщо необхідно знайти деякий ключовий елемент в масиві. Тобто організувати пошук по ключу, де ключ - це певне значення в масиві.

Розглянемо задачу пошуку номера елемента упорядкованої за *зростанням* послідовності $\{a_n\}$, який (елемент) має значення b. Під упорядкованим масивом розумітимемо масив, упорядкований по неспаданню, тобто $a[1] \leq a[2] \leq \ldots \leq a[N]$.

На початку виконання цього алгоритму:

j — менша межа діапазону номерів, на якому шукають потрібний номер, дорівнює меншій межі діапазону номерів масиву;

k — більша межа діапазону номерів, на якому шукають потрібний номер, дорівнює більшій межі діапазону номерів масиву.

Алгоритм бінарного пошуку для упорядкованого за *неспаданням* масиву має такий вигляд:

1. ***Поки справджується нерівність j ≤ k, робити таке:***

- ***надати значення змінній l = [(j + k)/2] — знайти індекс l, що поділяє досліджуваний діапазон навпіл;***
- ***якщо $a_l = b$, завершити пошук зі знайденим номером l;***
- ***якщо $a_l < b$, надати значення змінній j = (l + 1), тобто вибрати для подальшого пошуку діапазон, розташований правіше від l;***
- ***якщо $a_l > b$, надати значення змінній k = (l – 1), тобто вибрати для подальшого пошуку діапазон, розташований лівіше від l.***

2. ***Завершити пошук, вважаючи його неуспішним.***

Тут квадратні дужки (у першій дії всередині циклу) позначають операцію визначення цілої частини.

Наведемо програму на C++ повністю:

```
/* Програма C++ для реалізації ітеративного двійкового пошуку */
#include <bits/stdc++.h>
using namespace std;
// Ітеративна функція двійкового пошуку. Воно повертає
// розташування x у заданому масиві arr[l..r], якщо є,
// інакше -1
int binarySearch(int arr[], int l, int r, int x)
{
  while (l <= r) {
    int m = l + (r - l) / 2;

    // Перевіряємо, чи є x у середині
    if (arr[m] == x)
      return m;
```

```
    // Якщо x більше, ігнорувати ліву половину
    if (arr[m] < x)
      l = m + 1;

    // Якщо x менше, ігноруємо праву половину
    else
      r = m - 1;
  }

  // якщо дійшли до даного рядка, то елемента не було
  return -1;
}

//Виведення масиву на екран
void showArr(int arr[], int arrSize)
{
 for (int i = 0; i < arrSize; i++)
 {
 cout << setw(4) << arr[i];
 if ((i + 1) % 10 == 0)
 {
 cout << endl;
 }

 }
 cout << endl << endl;
}

int main(void)
{
  int arr[50];
  int x, n = 50;
  setlocale(LC_ALL, "ukr");
  srand(time(NULL));
  for (int i = 0; i < n; i++) {
    arr[i] = 1 + rand() % 50;
  }
  //Сортування масиву з використанням STL
   sort(arr, arr+n);
```

```
   //Виведення масиву на екран
   showArr(arr, n);

   cout << "Яке число необхідно шукати?";
   cin >> x; // Введення шуканого числа

   //Пошук шуканого числа та запис номера елемента
  int result = binarySearch(arr, 0, n - 1, x);
  (result == -1)
    ? cout << "У масиві немає такого значення"
    : cout << "Елемент знаходиться в комірці з індексом: "
<< result;
  return 0;
}
```

Складність алгоритму бінарного пошуку становить $O(log\ N)$, де N – кількість елементів у масиві.

Вбудовані функції C++

У C++ існує вбудована функція ***binary_search***, яка перевіряє, чи є у відсортованому діапазоні елемент, що дорівнює вказаному значенню. Функція повертає значення true, якщо вказаний елемент є в діапазоні і false – якщо такий елемент відсутній. Функція вимагає приєднання бібліотеки <algorithm>.

Таку програму можна записати:

```
/* Програма C++ для реалізації ітеративного двійкового
пошуку */
#include <bits/stdc++.h>

using namespace std;

//Виведення масиву на екран
void showArr(int arr[], int arrSize)
{
 for (int i = 0; i < arrSize; i++)
 {
 cout << setw(4) << arr[i];
 if ((i + 1) % 10 == 0)
 {
```

```
 cout << endl;
 }

 }
 cout << endl << endl;
}

int main(void)
{
  int arr[50];
  int x, n = 50;
  setlocale(LC_ALL, "ukr");
  srand(time(NULL));
  for (int i = 0; i < n; i++) {
    arr[i] = 1 + rand() % 50;
  }
  //Сортування масиву з використанням STL
   sort(arr, arr+n);
   //Виведення масиву на екран
   showArr(arr, n);

   cout << "Яке число необхідно шукати?";
   cin >> x; // Введення шуканого числа

   //Пошук шуканого числа та запис номера елемента
  int result = binary_search(arr, arr+n, x);
  (result == -1)
    ? cout << "У масиві немає такого значення"<< endl
    : cout << "У масиві є таке значення" << endl;
  return 0;
}
```

Наведемо інші корисні функції STL C++. Функція **lower_bound** повертає ітератор першого елемента е відсортованому діапазоні, який дорівнює або більший за елемент, що шукається.

```
    #include <iostream>
    #include <algorithm>
    using namespace std;
    int a[10] = { 1, 2, 3, 4, 5, 5, 5, 8, 9, 10};
    int main()
```

```
{
int b = 5;
cout << * (lower_bound (a, a + 10, b)) << endl;
b = 6;
cout << * (lower_bound (a, a + 10, b));
}
```

Програма виведе значення 5, тому що в масиві є елемент 5 і значення 8 – значення першого елемента більшого 6. Аналогічно функції lower_bound працює функція upper_bound, яка повертає ітератор на елемент, який строго більше від шуканого в заданому діапазоні впорядкованих елементів.

2.2. Бінарний пошук для монотонних функцій

Бінарний пошук може використовуватися не лише для пошуку елементів у масиві, але й для пошуку коренів рівнянь та значень монотонних (зростаючих чи спадних) функцій. Нагадаємо, що функція називається **зростаючою**, якщо ∀x1, x2 : x1 > x2 ⇒ f(x1) > f(x2) (для будь-яких x1 і x2, якщо x1 > x2, то f(x1) також більше f(x2)).

Дійсно, так само як і в масиві, ми можемо виключити з розгляду половину поточної області, якщо нам відомо, що там не існує рішення. Якщо ж функція не монотонна, то користуватися бінарним пошуком не можна, бо він може видавати неправильну відповідь, або шукати в повному обсязі відповіді.

Наприклад розглянемо завдання пошуку кубічного кореня. Кубічним коренем числа x (позначається $\sqrt[3]{x}$) називається таке число y, що $y^3 = x$.

Сформулюємо задачу так: для цього дійсного числа x (x ≥ 1) знайти кубічний корінь з точністю не менше 5 знаків після десяткової крапки.

Функція при x ≥ 1 обмежена зверху числом x, а знизу - одиницею.

Таким чином, за нижню межу ми вибираємо 1, за верхню – саме число x. Після цього ділимо поточний відрізок навпіл, підносимо середнє значення до кубу і якщо куб більше x, то замінюємо верхню грань, інакше - нижню.

Код буде виглядати так:

```
r = x ;
l = 1 ;
while (fabs ( l -r) > eps ) {
m= ( l+r ) /2 ;
if (m * m * m < x) l = m ;
else r = m ;
}
```

Щоб користуватися функцією ***fabs***, необхідно підключити бібліотеку ***cmath***.

Пошук кореня рівняння алгоритмом бінарного пошуку

Алгоритм бінарного пошуку зручно застосовувати визначення коренів рівняння (так званий дійсний бінарний пошук).

Задача. Знайдіть таке число x, що $x^2 + \sqrt{x} = C$, з точністю не менше ніж 6 знаків після десяткової крапки.

Вхідні дані

У єдиному рядку міститься дійсне число $1.0 \leq C \leq 10^{10}$.

Вихідні дані

Виведіть одне число - шукане x.

Програма на C++:

```
#include <iostream>
#include <cmath>
using namespace std;
const long double eps = 1e-10;
long double f(long double x){
return x * x + sqrt(x);
}
int main(){
long double c, left = 0, right = 1e15, middle;
 setlocale(LC_ALL, "ukr");
 cout << "Уведіть значення константи рівняння C: " << endl;
cin >> c;
while (fabs(right - left) > eps) {
// for (int i = 0; i < 100; ++i) на практиці використовують
for
middle = (left + right) / 2.0;
if ((f(middle)-c)<0)
  left = middle;
else
  right = middle;
}
cout << fixed;
cout.precision(7);
cout << right;
return 0;
```

```
}
```

У наведеній програмі закінчення пошуку відбувається в тому випадку, коли проаналізований відрізок стане меншим заданої похибки. Приблизна кількість ітерацій алгоритму дорівнюватиме $log\,(\frac{R-L}{\varepsilon})$. У проаналізованому прикладі зсув меж пошуку відбувається з урахуванням порівняння значення функції з нулем: **f(middle) – c < 0**. Якщо немає інформації щодо монотонності функції, але точно відомо, що у інтервалі пошуку вона має єдиний корінь, то розумніше перевірятиме наявність різних знаків функції на межах нової області пошуку: у випадку $\boldsymbol{f(l) * f(m) \leq 0}$ пересувати правий кордон (переходимо в лівий проміжок), а у випадку $\boldsymbol{f(r) * f(m) \leq 0}$ у правий проміжок).

3. Тернарний пошук

Тернарний пошук використовують для пошуку:

✓ максимума функції f на заданому проміжку [l, r] за умови, що функція спочатку строго зростає, потім досягає максимуму в одній точці або на відрізку, потім строго спадає;

✓ мінімума функції f на заданому проміжку [l, r] за умови, що функція спочатку строго спадає, потім досягає мінімуму в одній точці або на відрізку, потім строго зростає.

Алгоритм (тернарного пошуку максимуму за умови, що він досяжний лише в одній точці)

1. Вибрати довільні точки m_1 і m_2, що задовольняють нерівності: $l < m_1 < m_2 < r$. Наприклад, надати значень:

$m_1 = l + (r - l)/3$;

$m_2 = r - (r - l)/3$.

2. Порахувати значення функції $f(m_1)$ і $f(m_2)$.

3. Якщо $f(m_1) < f(m_2)$ — шуканий максимум поза проміжком $[l, m_1)$ — надати значення:

$l = m_1$.

4. Якщо $f(m_1) > f(m_2)$ — шуканий максимум поза проміжком $(m_2, r]$ — надати значення:

$r = m_2$.

5. Якщо $f(m_2) = f(m_2)$ — шуканий максимум поза проміжками $[l, m_1)$ і $(m_2,r]$ при строгому зростанні й спаданні f до і після точки максимума при зростанні аргумента — надати значення:

$l = m_1$;

$r = m_2$.

6. Якщо r – l < ε, вивести значення (l + r)/2, значення f у цій точці і закінчити виконання алгоритму. Інакше перейти до пункту 1.

Тут ε — дійсне число — деяка наперед задана точність пошуку.

Обчислити найбільше значення функції $f(x) = 1 - x^2$ на проміжку [−3, 2].

Код програми на C++:

```
#include <iostream>
using namespace std;

static double f(double x) {return 1-x*x;}

int main()
{ double m1,m2,rs,l=-3, r=1, eps=1e-10;
 setlocale(LC_ALL, "ukr");
 while (r-l > eps)
 { m1 = l + (r-l)/3,
  m2 = r - (r-l)/3;
  if (f(m1) < f(m2)) l=m1; else
  if (f(m1) > f(m2)) r=m2; else {l=m1; r=m2;}
 }
 cout.setf(ios::fixed);
 cout.precision(11);
 rs=((r+l)/2);
 cout<<"Найбільше значення функції: f("<<rs<<") =
"<<f(rs)<<endl;
 return 0;
}
```

Часова складність: $O(\log_3 n)$, де n — розмір масиву.

Просторова складність: O(1)

🔔 Контрольні запитання:

1. У яких задачах доречно використовувати лінійний пошук?
2. Яка часова складність лінійного пошуку?
3. У чому полягає ідея бінарного пошуку?
4. Які вбудовані функції C++ дозволяють здійснити двійковий пошук?
5. У чому суть бінарного пошуку для монотонних функцій?
6. Коли використовують тернарний пошук?
7. Яка ідея тернарного пошуку?

Тема №7. ХЕШУВАННЯ

План лекції:

1. Вступ
2. Хеш-таблиця
3. Функції хешування
4. Хешування
5. Відкрита адресація

Джерела:

[4, Глава 14], [5, Chapter 11], [6, Chapter 10], [9, §6.4], [11, Chapter 5]

1. Вступ

Раніше часто говорилося, що хешування дозволяє більш оптимально виконувати дії з деревами чи послідовностями даних.

Можна навести інший приклад. Нехай, ми хочемо розробити систему зберігання записів про співробітників за номерами телефонів. Потрібно, щоб наступні запити були виконані ефективно:

1. Ввести номер телефону та відповідну інформацію.

2. Знайти номер телефону та отримати інформацію.

3. Видалити номер телефону та пов’язану інформацію.

Шляхи розв’язання даної проблеми:

1. Масив телефонних номерів і записів.

2. Пов’язаний список телефонних номерів і записів.

3. Збалансоване двійкове дерево пошуку з телефонними номерами як ключами.

4. Таблиця прямого доступу.

Для масивів і зв’язаних списків потрібно шукати лінійним способом, що на практиці може бути дорогим задоволенням. Якщо ми використовуємо масиви та зберігаємо дані відсортованими, то номер телефону можна шукати за час O(log n) за допомогою двійкового пошуку, але операції вставки та видалення стають складними, оскільки нам доводиться підтримувати установлений раніше порядок.

Завдяки збалансованому бінарному дереву пошуку ми отримуємо помірний час пошуку, вставки та видалення. Гарантовано, що всі ці операції виконуються за час O(log n).

Інше рішення, яке можна придумати, – це використовувати таблицю прямого доступу, де ми створюємо великий масив і використовуємо номери телефонів як індекс у масиві. Запис у масиві NULL, якщо номер телефону відсутній, інакше запис масиву зберігає вказівник на записи, що відповідають

номеру телефону. За складністю часу це рішення є найкращим серед усіх, ми можемо виконувати всі операції за O(1) час. Наприклад, щоб вставити номер телефону, ми створюємо запис із деталями даного номера телефону, використовуємо номер телефону як індекс і зберігаємо покажчик на створений запис у таблиці.

Але таке рішення має багато практичних обмежень. Перша проблема полягає в тому, що необхідний додатковий простір пам'яті для зберігання буде величезним. Наприклад, якщо номер телефону складається з n цифр, нам потрібно O(m * 10n) простору для таблиці, де m — розмір покажчика для запису. Інша проблема полягає в тому, що ціле число в мові програмування може не зберігати n цифр.

Через зазначені вище обмеження таблицю прямого доступу не завжди можна використовувати. Хешування – це рішення, яке можна використовувати майже в усіх подібних ситуаціях і на практиці працює надзвичайно добре в порівнянні з вищенаведеними структурами даних, такими як масив, зв'язаний список, збалансоване бінарне дерево пошуку BST. За допомогою хешування ми отримуємо O(1) середній час пошуку (за розумних припущень) і O(n) у гіршому випадку.

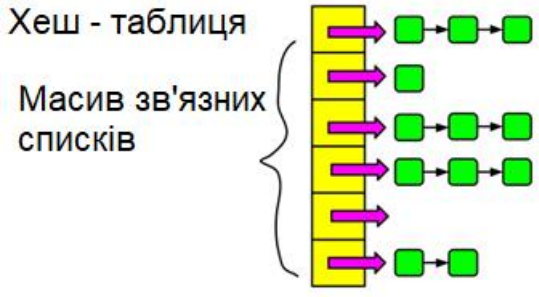
Хешування є покращенням порівняно з таблицею прямого доступу. Ідея полягає у використанні хеш-функції, яка перетворює заданий номер телефону або будь-який інший ключ у менше число та використовує невелике число як індекс у таблиці, яка називається хеш-таблицею.

2. Хеш-таблиця

Означення. **Хеш-таблиця** – це структура даних для зберігання пар ключ-значення.

На відміну від базового масиву, який використовує номери індексів для доступу до елементів, хеш-таблиця використовує ключі для пошуку записів таблиці. Це робить керування даними більш керованим для користувача-людини, оскільки простіше каталогізувати записи даних за їхніми атрибутами, а не за їх кількістю у гігантському списку.

У C++ ми реалізуємо хеш-таблицю як масив зв'язних списків. Це щось на зразок багатовимірного масиву. У двовимірному масиві, наприклад, елементи складаються з рядків фіксованої довжини. Однак у хеш-таблиці елементи (або сегменти) можуть розширюватися або стискатися, щоб вмістити практично нескінченну кількість записів таблиці.



Основні операції для хеш-таблиці:

• **Search** – виконує пошук елемента в хеш-таблиці.

• **Insert** – вставляє елемент у хеш-таблицю.

• **Delete** — видаляє елемент із хеш-таблиці.

Визначити структуру даних, що має деякі дані та ключ, на основі яких буде проводитися пошук у хеш-таблиці можна таким чином:

```
struct DataItem {
  int data;
  int key;
};
```

Визначити метод хешування для обчислення хеш-коду ключа елемента даних можна, наприклад, так, як подано нижче:

```
int hashCode(int key){
  return key % SIZE;
}
```

Операція Search

Щоразу, коли потрібно шукати елемент, слід обчислити хеш-код переданого ключа та знайти елемент, використовуючи цей хеш-код як індекс у масиві. Доречно використовувати лінійне зондування, щоб визначити елемент, якщо елемент не знайдено в обчисленому хеш-коді.

```
struct DataItem *search(int key) {
  // отримати хеш
  int hashIndex = hashCode(key);

  //переміщення в масиві до порожнього
  while(hashArray[hashIndex] != NULL) {

   if(hashArray[hashIndex]->key == key)
     return hashArray[hashIndex];

   //перехід до наступної комірки
   ++hashIndex;

   //покрити таблицю
   hashIndex %= SIZE;
  }
```

```
    return NULL;
  }
```

Операція insert

У випадку, коли потрібно вставити елемент, слід обчислити хеш-код переданого ключа та знайти індекс, використовуючи цей хеш-код як індекс у масиві. Використовуйте лінійне зондування, щоб випередити елемент, якщо елемент не знайдено в обчисленому хеш-коді. Коли буде знайдено, збережіть там фіктивний елемент, щоб зберегти продуктивність хеш-таблиці.

```
void insert(int key,int data) {
  struct DataItem *item = (struct DataItem*) malloc(
sizeof( struct DataItem ));
  item->data = data;
  item->key = key;

  // отримати хеш
  int hashIndex = hashCode(key);

  /*переміщення в масиві до порожньої або видаленої
комірки */
  while(hashArray[hashIndex] != NULL &&
hashArray[hashIndex]->key != -1) {
   //перейти до наступної комірки
   ++hashIndex;

   //покрити таблицю
   hashIndex %= SIZE;
  }

  hashArray[hashIndex] = item;
}
```

Операція delete

Якщо елемент потрібно видалити, то слід обчислити хеш-код переданого ключа та знайти індекс, використовуючи цей хеш-код як індекс у масиві. Використовуйте лінійне зондування, щоб випередити елемент, якщо елемент не знайдено в обчисленому хеш-коді. Коли буде знайдено, збережіть там фіктивний елемент, щоб зберегти продуктивність хеш-таблиці.

```
    struct DataItem* delete(struct DataItem* item) {
      int key = item->key;

      // отримати хеш
      int hashIndex = hashCode(key);

      //переміщення в масиві до порожнього
      while(hashArray[hashIndex] !=NULL) {

       if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];

         /*призначити фіктивний елемент у видаленій
позиції */
         hashArray[hashIndex] = dummyItem;
         return temp;
       }

       //перейти до наступної комірки
       ++hashIndex;

       //покрити таблицю е
       hashIndex %= SIZE;
      }

      return NULL;
    }
```

З точки зору ефективності, хеш-таблиця є компромісом між масивом і зв'язаним списком. Він використовує як індексацію, так і обхід списку для зберігання та отримання елементів даних.

Пошук елементів за індексом робить масив дуже ефективним. Незалежно від того, де в масиві зберігається елемент, його отримання завжди займає однакову кількість часу. З технічної точки зору, отримання елемента з масиву – це операція O(1) або «постійний час».

Пошук елементів у зв'язаному списку набагато менш ефективний. Не можна просто отримати прямий доступ до будь-якого вузла в списку. Замість цього слід пройти по списку до кінця, поки не знайдеться цільовий елемент. Якщо елемент, який потрібно знайти, опиняється на початку списку, пошук є

операцією O(1), оскільки потрібно перейти лише на один вузол. Якщо елемент знаходиться в кінці списку, його знаходження буде операцією складності O(n), де n – загальна кількість вузлів у списку.

Підсумовуючи, у міру збільшення кількості елементів у масиві час виконання для доступу до елемента за його індексом залишається незмінним. Зі зв'язаним списком час, необхідний для доступу до певного елемента, збільшується лінійно з кількістю елементів.

Ідея полягає у використанні двовимірного масиву хеша розміру [MAX+1][2].

Алгоритм :

1. Призначити всім значенням хеш-матриці 0.

2. Обхід заданого масиву:

Якщо елемент ele не від'ємний, то призначити

hash[ele][0] = 1.

В іншому випадку, взяти абсолютне значення ele і призначити hash[ele][1] = 1.

3. Для пошуку будь-якого елемента x в масиві.

Якщо X є невід'ємним, перевірити, чи хеш [X][0] == 1 чи ні. Якщо хеш [X][0] дорівнює одиниці, то число присутнє, інакше – немає.

Якщо X від'ємне, взяти абсолютне значення X і перевірити, чи хеш [X][1] == 1 чи ні. Якщо хеш [X][1] дорівнює одиниці, то число присутнє.

Наведемо приклад коду програми на C++ .

```
/* Програма CPP для реалізації прямого відображення індексу
з дозволеними від'ємними значеннями. */
#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

// Оскільки масив є глобальним, він ініціалізується як 0.
bool has[MAX + 1][2];

// пошук, присутнє X у заданому масиві чи ні.
bool search(int X)
{
  if (X >= 0) {
    if (has[X][0] == 1)
```

```
      return true;
    else
      return false;
  }

  // якщо X від'ємне, взяти абсолютне значення X.
  X = abs(X);
  if (has[X][1] == 1)
    return true;

  return false;
}

void insert(int a[], int n)
{
  for (int i = 0; i < n; i++) {
    if (a[i] >= 0)
      has[a[i]][0] = 1;
    else
      has[abs(a[i])][1] = 1;
  }
}

// Основна програма
int main()
{
  int a[] = { -1, 9, -5, -8, -5, -2 };
  int n = sizeof(a)/sizeof(a[0]);
  setlocale(LC_ALL, "ukr");

  insert(a, n);
  int X = -5;

  if (search(X) == true)
    cout << "Присутній";
  else
    cout << "Відсутній";
  return 0;
}
```

3. Функції хешування

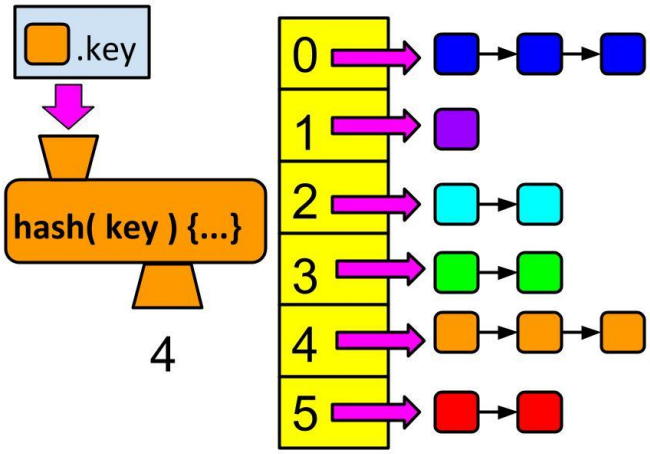
Означення. **Хеш-функцією** називається будь-яка функція, яка може використовуватися для відображення даних довільного розміру у значенні фіксованого розміру.

Значення, які повертає хеш-функція, називаються хеш-значеннями, хеш-кодами, дайджестами або просто хешами. Значення зазвичай використовуються для індексації таблиці фіксованого розміру, яка називається **хеш-таблицею**. Використання хеш-функції для індексації хеш-таблиці називається **хешуванням** або адресацією розсіяного сховища.

Хеш-функції та пов'язані з ними хеш-таблиці використовуються в програмах зберігання та пошуку даних для доступу до даних за короткий і майже постійний проміжок часу. Для них потрібно лише трохи більше пам'яті, ніж загальний простір, необхідний для самих даних або записів.

Хеш-функція вирішує, де зберігати та отримувати елементи в хеш-таблиці. Він приймає ключ елемента як свій параметр і повертає розташування індексу для цього конкретного елемента. Як правило, хеш-функція використовує модульну арифметику. Зокрема, значення ключа ділиться на довжину таблиці, щоб створити номер індексу в таблиці. Цей номер індексу відноситься до розташування або сегмента в хеш-таблиці.

У попередньому прикладі: функція, яка перетворює цей великий номер телефону в маленьке практичне ціле число. Відображене ціле значення використовується як індекс у хеш-таблиці. Простіше кажучи, хеш-функція відображає велике число або рядок у невелике ціле число, яке можна використовувати як індекс у хеш-таблиці.



Приклад 2. Припустимо, що хеш-функція приймає рядок як параметр, а потім додає значення ASCII усіх символів у цьому рядку, щоб отримати ціле число. Якщо ключ «Гора», то сума значень ASCII кодування буде 689. Після цього хеш-функція приймає значення модуля цього числа на довжину таблиці, щоб отримати номер індексу. Якщо довжина таблиці дорівнює 13, то 689 за модулем 13 дорівнює 0. Таким чином, елемент з ключем «Гора» потрапить у покажчик №0 хеш-таблиці.

```
int hash( string key )
{
  int value = 0;
  for ( int i = 0; i < key.length(); i++ )
    value += key[i];
  return value % tableLength;
}
```

Хеш-функції та пов’язані з ними хеш-таблиці використовуються в програмах для зберігання та пошуку даних для доступу до даних за невеликий і майже постійний час на пошук. Вони вимагають обсягу пам’яті лише трохи більше, ніж загальний простір, необхідний для самих даних або записів.

Хешування – це ефективна з точки зору обчислень і пам’яті форма доступу до даних, яка дозволяє уникнути нелінійного часу доступу до впорядкованих і невпорядкованих списків і структурованих дерев, а також часто експоненційних вимог до зберігання, пов’язаних з прямим доступом до просторів станів великої або змінної довжини ключів.

Використання хеш-функцій залежить від статистичних властивостей взаємодії ключів і функцій: поведінка в гіршому випадку є дуже поганою з малою ймовірністю, а поведінка в середньому випадку може бути майже оптимальною.

Гарна хеш-функція повинна мати такі властивості

1) Ефективно обчислюватися.

2) Повинна рівномірно розподіляти ключі (кожна позиція таблиці однаково ймовірна для кожного ключа).

Наприклад, для телефонних номерів погана хеш-функція – це використовувати перші три цифри. Кращою функцією є розглянути останні три цифри. Зверніть увагу, що це може бути не найкраща хеш-функція. Можливо, є кращі способи.

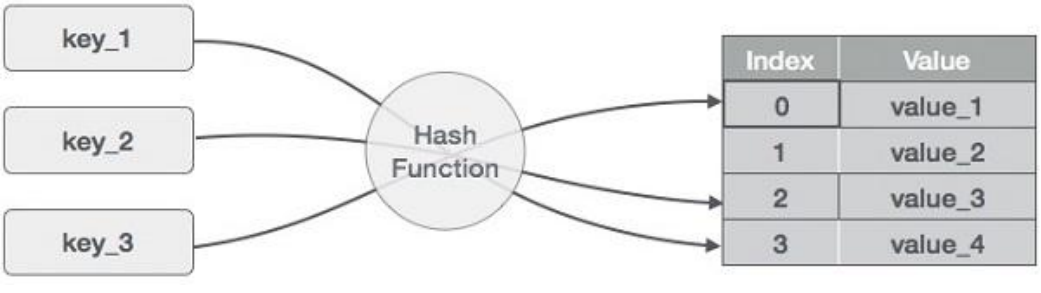
4. Хешування

Означення. **Хешування** – це метод перетворення діапазону значень ключів у діапазон індексів масиву.

Ми будемо використовувати оператор за модулем, щоб отримати діапазон значень ключів. Розглянемо приклад хеш-таблиці розміру 20, і слід зберегти наступні елементи. Елемент у форматі (ключ, значення).

(1, 20)

(2, 70)
(42, 80)
(4, 25)
(12, 44)
(14, 32)
(17, 11)
(13, 78)
(37, 98)



Старший номер	Ключ	Хеш	Масив покажчик
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Управління колізіями

Оскільки хеш-функція отримує невелике число для ключа, який є великим цілим числом або рядком, ймовірно, що два ключі отримають одне і те ж значення. Ситуація, коли нещодавно вставлений ключ відповідає вже зайнятому слоту в хеш-таблиці, називається **колізією**, і її необхідно обробляти за допомогою спеціальної техніки обробки колізій.

У цьому прикладі відбулося виникнення колізії з індексом масиву17.

Яка ймовірність виникнення колізії у великій таблиці?

Колізії дуже ймовірні, навіть якщо у нас є велика таблиця для зберігання ключів. Важливим спостереженням є парадокс дня народження. Якщо всього 23 людини, то ймовірність того, що у двох людей однаковий день народження, становить 50%.

Як боротися з колізіями?

В основному існує два методи боротьби з виникненням колізій:

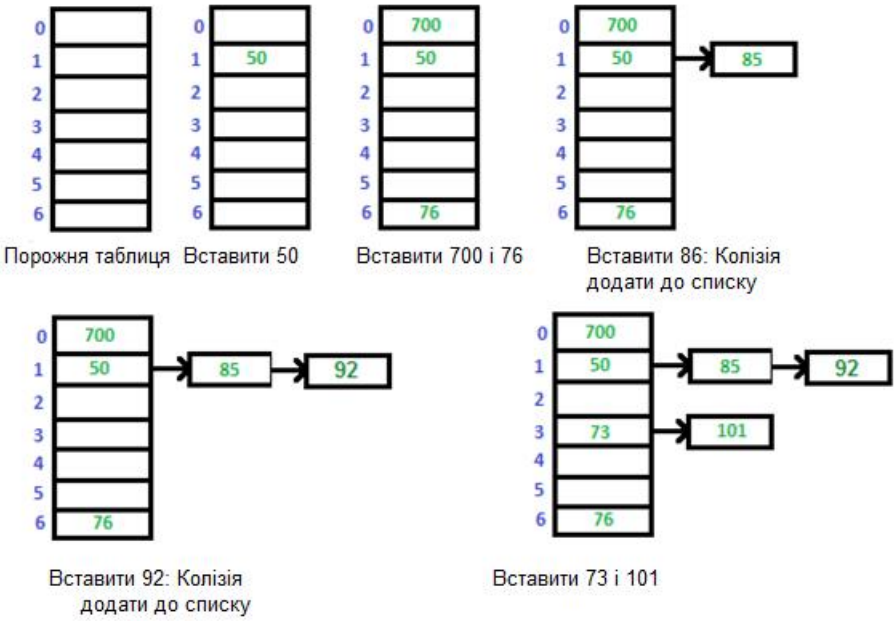
1) Окремий ланцюжок

2) Відкрита адресація

Окремий ланцюжок:

Ідея полягає в тому, щоб кожна комірка хеш-таблиці вказувала на зв’язаний список записів, які мають однакове значення хеш-функції.

Розглянемо просту хеш-функцію як "ключ за модулем 7", а послідовність ключів як 50, 700, 76, 85, 92, 73, 101.



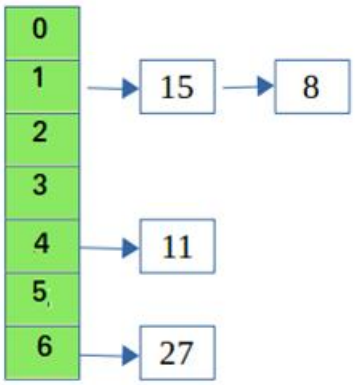
Ідея полягає в тому, щоб кожна клітинка хеш-таблиці вказувала на зв’язаний список записів, які мають однакове значення хеш-функції. Давайте створимо хеш-функцію, щоб хеш-таблиця мала кількість сегментів «N». Щоб вставити вузол у хеш-таблицю, нам потрібно знайти хеш-індекс для даного ключа. І його можна обчислити за допомогою хеш-функції.

Приклад : hashIndex = key % noOfBuckets

Insert: Перехід до сегмента відповідає обчисленому вище хеш-індексу та вставленню нового вузла у кінець списку.

Розглянемо хеш-таблицю з 7 сегментами (0, 1, 2, 3, 4, 5, 6).

Ключі надходять по порядку (15, 11, 27, 8). Тоді отримаємо хеш-таблицю, яка зображена на рисунку.



Delete: Щоб видалити вузол з хеш-таблиці, слід обчислити хеш-індекс для ключа, перейти до сегмента, який відповідає обчисленому хеш-індексу,

переглянути список у поточному сегменті, щоб знайти та видалити вузол із заданим ключем (якщо знайдено).

Переваги:

1) Алгоритм простий у виконанні.

2) Хеш-таблиця ніколи не заповнюється, ми завжди можемо додати більше елементів до ланцюжка.

3) Менш чутливий до хеш-функції або коефіцієнтів навантаження.

4) Здебільшого використовується, коли невідомо, скільки і як часто ключів можна вставляти чи видаляти.

Недоліки:

1) Продуктивність кешу ланцюжка не гарна, оскільки ключі зберігаються за допомогою зв'язаного списку. Відкрита адресація забезпечує кращу продуктивність кешу, оскільки все зберігається в одній таблиці.

2) Втрата простору (деякі частини хеш-таблиці ніколи не використовуються)

3) Якщо ланцюжок стає довгим, то в гіршому випадку час пошуку може стати O(n).

4) Використовує додатковий простір для посилань.

Продуктивність ланцюга:

Ефективність хешування можна оцінити за припущення, що кожен ключ з однаковою ймовірністю хешується в будь-який слот таблиці (просте рівномірне хешування). Якщо:

m – кількість слотів у хеш-таблиці

n – кількість ключів, які потрібно вставити в хеш-таблицю, то

коефіцієнт навантаження $\alpha = n/m$.

Очікуваний час пошуку дорівнює $O(1 + \alpha)$.

Очікуваний час видалення дорівнює $O(1 + \alpha)$.

Час вставки дорівнює $O(1)$.

Часова складність пошуку вставки та видалення дорівнює $O(1)$, якщо α дорівнює $O(1)$.

Конструкції для зберігання даних у ланцюгу :

Зв'язаний список

Search: O(l) , де l = довжина з пов'язаний список

Delete: O(l)

Insert: O(l)

Не підтримує кеш

Динамічні масиви (вектори в C++):

Search: O(*l*) , де *l* – довжина динамічного масиву.

Delete: O(*l*).

Insert: O(*l*).

Кеш підтримується.

5. Відкрита адресація

Як і окремий ланцюжок, відкрита адресація є методом обробки колізій. У відкритій адресації всі елементи зберігаються в самій хеш-таблиці. Тому в будь-який момент розмір таблиці має бути більшим або дорівнювати загальній кількості ключів (зауважте, що ми можемо збільшити розмір таблиці, копіюючи старі дані, якщо потрібно).

Insert(k): Продовжувати пробувати, поки не знайдеться порожній слот. Знайшовши порожній слот, вставити k.

Search(k): продовжувати пробувати, доки ключ слота не стане рівним k або не буде досягнуто порожнього слота.

Delete(k): операція видалення. Якщо просто видалити ключ, то пошук може зірватися. Тому слоти видалених ключів позначаються спеціально як «видалені».

Insert(k) може вставити елемент у вилучений слот, але пошук не зупиняється на видаленому слоті.

Для виконання вставки при відкритій адресації ми послідовно перевіряємо комірки хеш-таблиці доти, поки не знаходимо порожню комірку, у яку розташовуємо ключ, що вставляється.

Нехай hash(x) – це індекс слота, обчислений за допомогою хеш-функції, а S – розмір таблиці.

Якщо слот hash(x) % S заповнений, намагаємося потрапити в (hash(x) + 1) % S.

Якщо (hash(x) + 1) % S також заповнений, то намагаємося потрапити в (hash(x) + 2) % S.

Якщо (hash(x) + 2) % S також заповнений, то намагаємося потрапити в (hash(x) + 3) % S і т.д.

Кожен слот містить або ключ, або значення NIL (якщо він не заповнений).

🔔 Контрольні запитання:

1. Що таке хеш-таблиця?
2. Які основні операції виділяють для хеш-таблиць?
3. Дайте означення хеш-функції.
4. Які властивості у гарної хеш-функції?
5. Що таке хешування?
6. Що таке колізія при хешуванні?
7. Які є методи обробки колізій? Охарактеризуйте їх.

Тема №8. АЛГОРИТМИ СОРТУВАННЯ В ОДНОВИМІРНИХ МАСИВАХ

План лекції:

1. Вступ
2. Сортування вставкою
3. Сортування методом бінарної вставки
4. Сортування злиттям
5. Швидке сортування
6. 3-стороннє швидке сортування

Джерела:

[1, §5.1], [2, Розділ 8], [3, Глава 8], [4, Часть 3], [5, Chapter 5 - 8], [6, Chapter 9], [9, Chapter 5], [10, Chapter 4], [11, Chapter 7]

1. Вступ

У словниках слово «сортування» (sorting) визначається як «розподіл, відбір по сортах; поділ на категорії, сорту, розряди», проте програмісти зазвичай використовують це слово в більш вузькому сенсі, позначаючи перегрупування масиву елементів в деякому певному порядку. Цей процес, напевно, слід було б назвати упорядкуванням (ordering) або ранжуванням (sequencing).

Означення. Сортуванням називається це процес перегрупування заданої множини об’єктів в деякому певному порядку.

Мета сортування – полегшити пошук елементів.

Означення. Алгоритмом сортування називається алгоритм для впорядкування елементів у списку.

Алгоритм сортування використовується для перевпорядкування заданого масиву або елементів списку відповідно до оператора порівняння для елементів. Оператор порівняння використовується для визначення нового порядку елемента у відповідній структурі даних.

Усі алгоритми, які розглядаються у цій темі, будуть взаємозамінними. Кожному буде переданий масив, що містить елементи; вважаємо, що всі позиції масиву містять дані для сортування. Нехай N – це кількість елементів, що передається програмам сортування.

Зауважимо, що STL (англ. Standard Template Library) надає багато функцій сортування, зокрема в бібліотеці <algorithm>.

У STL сортування виконується за допомогою функції сортування шаблону. Параметри для сортування представляють початковий і кінцевий маркер контейнера і необов'язковий компаратор:

void sort(Iterator begin, Iterator end);

void sort(Iterator begin, Iterator end, Comparator cmp);

Ітератори повинні підтримувати випадковий доступ. Алгоритм сортування не гарантує, що однакові елементи збережуть свій початковий порядок (якщо це важливо, потрібно використовувати **`stable_sort`** замість **`sort`**).

Наприклад:

```
std::sort( arr.begin( ), arr.end( ) );
std::sort( arr.begin( ), arr.end( ), greater<int>{ } );
std::sort( arr.begin( ), arr.begin( ) + ( arr.end( ) -
arr.begin( ))/2);
```

перший виклик сортує весь контейнер, `arr`, у неспадному порядку.

Другий виклик сортує весь контейнер, `arr`, у порядку зростання.

Третій виклик сортує першу половину контейнера, `arr`, у порядку спадання.

Розглянемо класичні алгоритми сортування одновимірних масивів. Спочатку введемо деякі властивості, які характерні для методів сортування.

Алгоритм сортування на місці використовує постійний додатковий простір для отримання результату (змінює лише даний масив). Він сортує список, лише змінюючи порядок елементів у списку.

Наприклад, сортування вставкою та сортування вибором є алгоритмами сортування на місці, оскільки вони не використовують жодного додаткового місця для сортування списку, а типова реалізація сортування злиттям не на місці, а також реалізація сортування підрахунком не на місці.

Коли всю інформацію, яку потрібно відсортувати, неможливо одночасно помістити в пам'ять, сортування називається **зовнішнім сортуванням.** Зовнішнє сортування використовується для великої кількості даних. Сортування злиттям і його варіанти зазвичай використовуються для зовнішнього сортування. Деякі зовнішні накопичувачі, такі як жорсткий диск тощо, використовуються для зовнішнього зберігання.

Коли всі дані розміщуються в пам'яті, сортування називається **внутрішнім сортуванням.**

Параметри алгоритмів сортування:

Час сортування – основний параметр, що характеризує швидкодію алгоритму.

Пам'ять – один з параметрів, що характеризується тим, що ряд алгоритмів сортування вимагають виділення додаткової пам'яті під тимчасове зберігання

даних. При оцінці використаної пам'яті не буде враховуватися місце, що займає вихідний масив даних і незалежні від вхідної послідовності витрати, наприклад, на зберігання коду програми.

Природність поводження – параметр, який вказує на ефективність методу при обробці вже відсортованих, або частково відсортованих даних. Алгоритм поводиться природно, якщо враховує цю характеристику вхідної послідовності й працює краще.

Використання операції порівняння. Алгоритми, що використовують для сортування порівняння елементів між собою, називаються заснованими на порівняннях. Мінімальна трудомісткість гіршого випадку для цих алгоритмів становить $O(n \log n)$, але вони відрізняються гнучкістю застосування. Для спеціальних випадків (типів даних) існують більш ефективні алгоритми.

Стійкість (або стабільність) – це параметр, який відповідає за те, що сортування не змінює взаємного розташування рівних елементів. Наприклад, якщо алфавітний список групи сортується за оцінками, то стійкий метод створює список у якому прізвища студентів з однаковими оцінками будуть впорядковані за алфавітом, а нестійкий метод створить список у якому, можливо, вихідний порядок буде порушений.

Стабільність в алгоритмах сортування в основному важлива, коли у нас є пари значень ключів із можливими повторюваними ключами. Наприклад, імена людей як ключі та їх деталі як значення. І потрібно відсортувати ці об'єкти за ключами.

Алгоритм сортування називається стабільним, якщо два об'єкти з однаковими ключами з'являються в тому ж порядку у відсортованому масиві, як вони з'являються у вхідному для сортування масиві.

Формально стабільність можна визначити так:

Нехай A — масив, і нехай "<" — строгий слабкий порядок на елементах A.

Алгоритм сортування є стабільним, якщо

$$i < j \text{ та } A[i] \equiv A[j] \text{ означає } \pi(i) < \pi(j),$$

де π — перестановка сортування (сортування переміщує A[i] у позицію π(i).

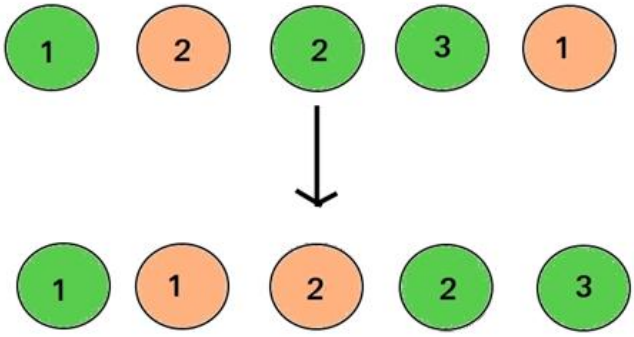


Рис. 8.1 Приклад стабільного сортування

Неформально стабільність означає, що еквівалентні елементи зберігають свої взаємні позиції після сортування.

Коли рівні елементи не можна розрізнити, наприклад за допомогою цілих чисел, або, загалом, будь-які дані, де весь елемент є ключовим, стабільність не є проблемою. Стабільність також не є проблемою, якщо всі ключі різні.

Будь-який даний алгоритм сортування, який не є стабільним, може бути змінений на стабільний. Можуть бути певні алгоритми сортування, щоб зробити його стабільним, але загалом будь-який алгоритм сортування на основі порівняння, який не є стабільним за своєю природою, можна змінити на стабільний, змінивши операцію порівняння ключів, щоб порівняння двох ключів розглядало позицію як коефіцієнт для об'єктів з рівними ключами.

Методи сортування даних у масивах

Для прискорення пошуку у множинних об'єктах, як правило, застосовують спочатку сортування (упорядкування) елементів. Існує велика кількість алгоритмів сортування, також як і алгоритмів пошуку, котрі описані у літературних джерелах. Нижче надається знайомство з поширеними алгоритмами.

До відомих алгоритмів сортування можна віднести:

За час $O(n^2)$:

1. Сортування вибором — пошук найменшого або найбільшого елемента і переміщення його в початок або кінець впорядкованого списку.

2. Сортування вставкою(включенням) — визначаємо місце де поточний елемент повинен знаходитися в упорядкованому списку, і вставляємо його туди.

3. Сортування обміном (сортування бульбашкою) — для кожної пари індексів проводиться обмін, якщо елементи розташовані не по порядку.

4. Сортування методом бінарної вставки

За час $O(nlog\ n)$

5. Плавне сортування

6. Пірамідальне сортування

7. Швидке сортування

8. Сортування злиттям

9. Timsort

За час $O(n)$з використанням додаткової інформації про елементи:

10. Сортування підрахунком

11. Сортування за розрядами

12. Сортування комірками

2. Сортування вставкою

Метод сортування вставки застосовується тоді, коли масив тільки що створений або триває його створення, і його треба заповнювати так, щоб після вставлення кожного нового елемента збереглася його впорядкованість. Для цього здійснюється пошук підходящого для вставки місця у вже заповненій частині масиву `a[0]...a[m]`. Пошук триває від кінця до початку масиву. Місце для нового елемента звільняється шляхом зсуву елементів масиву в кінець масиву. Масив практично розділений на відсортовану та несортовану частини.

Алгоритм

Щоб відсортувати масив розміром n у порядку зростання необхідно:

1. Повторювати від arr[1] до arr[n] по масиву.
2. Порівняти поточний елемент (ключ) з його попередником.
3. Якщо ключовий елемент менший, ніж його попередник, порівняти його з елементами раніше. Перемістити більші елементи на одну позицію праворуч, щоб звільнити місце для зміненого елемента.

Наведемо код на C++:

```
/* Функція сортування масиву за допомогою сортування вставкою*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
/* Перемістити елементи arr[0..i-1], які є більшими за key, на одну позицію праворуч щодо їх поточної позиції */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Аналіз сортування вставкою

Найгірший випадок – дані розташовані в зворотному порядку. У цьому випадку для кожного *i* елемент arr[i] менший за кожен елемент arr[0], ... , arr[i-1],

і кожен з них переміщується на одну позицію. Для кожної ітерації зовнішнього циклу for є i порівнянь, і загальна кількість порівнянь для всіх ітерацій цього циклу дорівнює

$$\sum_{i=1}^{N-1} i = 1 + 2 + 3 + 4 + \cdots + (N-1) = \frac{N(N-1)}{2} = \mathrm{O}(N^2).$$

Кількість виконання присвоєння у внутрішньому циклі for можна обчислити за допомогою тієї ж формули. До цього додається кількість разів, коли ключ завантажується та вивантажується у зовнішньому циклі for, що призводить до загальної кількості ходів:

$$\frac{N(N-1)}{2} + 2(N-1) = \frac{N^2 + 3N - 4}{2} = O(N^2)$$

Крім того, ця межа є жорсткою, оскільки уведення в зворотному порядку може досягти цієї межі. Точний розрахунок показує, що кількість тестів у внутрішньому циклі програми не більше $i + 1$ для кожного значення i. Підсумовуючи всі *i*, маємо загальну суму

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2).$$

З іншого боку, якщо вхідні дані попередньо відсортовані, час виконання дорівнює O(N), тому що тест у внутрішньому циклі for завжди завершується негайно. Справді, якщо вхідні дані майже відсортовані, сортування вставкою буде виконуватися швидко. Через ці широкі варіації варто проаналізувати поведінку цього алгоритму в середньому випадку. Виявляється, середній випадок є $\Theta(N^2)$ для сортування вставками, а також для ряду інших алгоритмів сортування.

Часова складність: $O(N^2)$.

Просторова складність алгоритму: O(1).

Граничні випадки: сортування вставками займає максимальний час для сортування, якщо елементи відсортовані в зворотному порядку. І воно займає мінімальний час (порядок n), коли елементи вже відсортовані.

Алгоритмічна парадигма: інкрементний підхід.

Сортування на місці: Так.

Стабільний алгоритм: Так.

Використання: Сортування вставкою використовується, коли кількість елементів невелика. Це також може бути корисно, коли вхідний масив майже відсортований, лише деякі елементи не розміщені в повному великому масиві.

3. Сортування методом бінарної вставки

Ми можемо використовувати двійковий пошук, щоб зменшити кількість порівнянь у звичайному сортуванні вставкою. Сортування двійковою вставкою використовує двійковий пошук, щоб знайти правильне місце для вставки вибраного елемента на кожній ітерації. При звичайній вставці сортування займає $O(i)$ (на і-й ітерації) у гіршому випадку. Ми можемо зменшити його до $O(log\ i)$, використовуючи двійковий пошук. Алгоритм в цілому все ще має час роботи в найгіршому випадку $O(N^2)$ через серію замін, необхідних для кожної вставки.

```
/* C++ програма для реалізації бінарного сортування вставкою */
#include <iostream>
using namespace std;

/* Функція на основі бінарного пошуку для пошуку позиції куди потрібно вставити елемент a[low..high] */
int binarySearch(int a[], int item, int low, int high)
{
    if (high <= low)
        return (item > a[low]) ? (low + 1) : low;

    int mid = (low + high) / 2;

    if (item == a[mid])
        return mid + 1;

    if (item > a[mid])
        return binarySearch(a, item, mid + 1, high);
    return binarySearch(a, item, low, mid - 1);
}

// Функція сортування масиву a[] розміром n
void insertionSort(int a[], int n)
{
    int i, loc, j, k, selected;

    for (i = 1; i < n; ++i)
    {
        j = i - 1;
        selected = a[i];
```

```
    // знайти місце, куди потрібно вставити вибране
            loc = binarySearch(a, selected, 0, j);

    /* Перемістити усі елементи після розташування, щоб
створити простір */
            while (j >= loc)
            {
                a[j + 1] = a[j];
                j--;
            }
            a[j + 1] = selected;
        }
    }

    // Код програми
    int main()
    {
        setlocale(LC_ALL,"ukr");

    // Блок декларації змінних
        int n;
        cout << "Уведіть розмір масиву n:"<<endl;
        cin >> n;
        int arr[n];

    //Блок уведення даних
        cout<<"Уведіть масив:"<<endl;
        for(int i = 0; i < n; i++)
            cin >> arr[i];
        cout << endl;

        insertionSort(arr, n);
        cout <<"Відсортований масив: \n";
        for (int i = 0; i < n; i++)
            cout <<" "<< arr[i];

        return 0;
    }
```

4. Сортування злиттям

Розглянемо сортування злиттям. Воно виконується в найгіршому випадку $O(N\ log\ N)$, і кількість використаних порівнянь майже оптимальна. Дане сортування є гарним прикладом рекурсивного алгоритму.

Основною операцією в цьому алгоритмі є злиття двох відсортованих частин списку. Функція merge() використовується для об'єднання двох половин. Злиття merge (arr, l, m, r) є ключовим процесом, який передбачає, що arr[l..m] і arr[m+1..r] відсортовані та об'єднує два відсортованих підмасиви в один.

Псевдокод для рекурсивної функції MergeSort:

MergeSort(arr[], l, r)

1 if r > l

2 $m = l + (r - l)/2$

3 MergeSort(arr, l, m)

4 *MergeSort(arr, m+1, r)*

5 *Merge(arr, l, m, r)*

6 else

return

MergeSort використовує допоміжну процедуру *Merge(arr, l, m, r)*, що здійснює об'єднання частин масиву arr з l-го по m-й елемент і з m+1-го по r-й елемент в один впорядкований підмасив.

Merge(arr, *l, m, r*)

1 ***i*** ← ***l***

2 **j** ← ***m+1***

3 **for** k ← *l* **to** *r*

4 **do if** j > r або (i ≤ m і arr[i] ≤ arr[j])

5 **then** B[k] ← arr[i]

6 i ← i + 1

7 **else** B[k] ← arr[j]

8 j ← j + 1

9 **for** *k* ← *l* **to** *r*

10 **do** arr[k] = B[k]

Якщо уважно проаналізувати алгоритм, то побачимо, що масив рекурсивно ділиться на дві половини доти, доки розмір не стане 1. Як тільки розмір стає 1, процеси злиття вступають в дію і починають об'єднувати масиви назад, доки не буде об'єднано повністю масив.

```
// Програма C++ для сортування злиттям
#include <iostream>
```

```
using namespace std;

// Об’єднує два підмасиви масиву[].
// Перший підмасив - arr[begin..mid]
// Другий підмасив - arr[mid+1..end]
void merge(int array[], int const left, int const mid,
int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Створення тимчасових масивів
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];

    /* Копіювання даних у тимчасові масиви leftArray[] і
rightArray[] */
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

// Початковий індекс першого підмасиву
    auto indexOfSubArrayOne = 0,
        indexOfSubArrayTwo = 0;
// Початковий індекс другого підмасиву
    int indexOfMergedArray = left;
// Початковий індекс об’єднаного масиву

    /* Злиття тимчасових масивів назад у
array[left..right] */
    while (indexOfSubArrayOne < subArrayOne &&
indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne] <=
rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray] =
leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
```

```
            array[indexOfMergedArray] =
rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Копіювати решту елементів
    // left[], якщо такі є
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray] =
leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Копіювати решту елементів
    // right[], якщо такі є
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray] =
rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
}
/* begin призначений для лівого індексу, а end – для
правого індексу підмасиву arr для сортування */
void mergeSort(int array[], int const begin, int const
end)
{
    if (begin >= end)
        return; // Рекурсивне повернення

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

// Функція для друку масиву
void printArray(int A[], int size)
{
```

```
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}

// Код основної програми
int main()
{
setlocale(LC_ALL,"ukr");

// Блок оголошення змінної
    int arr_size;
    cout << "Уведіть розмір масиву n:"<<endl;
    cin >> arr_size;
    int arr[arr_size];

//Блок уведення даних
    cout<<"Уведіть масив:"<<endl;
    for(int i = 0; i < arr_size; i++)
        cin >> arr[i];
    cout << endl;

    mergeSort(arr, 0, arr_size - 1);

    cout << "\n Відсортований масив: \n";
    printArray(arr, arr_size);
    return 0;
}
```

Аналіз Mergesort

Сортування злиття є класичним прикладом методів, які використовуються для аналізу рекурсивних процедур: ми повинні написати рекурентне відношення для часу виконання. Будемо вважати, що N є степенем 2, тому завжди ділимо на парні половини. Для N = 1 час сортування злиттям є постійним, який позначатимемо 1. В іншому випадку, час для сортування злиттям N чисел дорівнює часу виконання двох рекурсивних сортування злиття розміром N/2 плюс час злиття, який є лінійним. Запишемо тоді такі рівняння:

$T(1) = 1$

$T(N) = 2T(N/2) + N$

Отримали стандартне рекурентне співвідношення, яке можна розв'язати кількома способами. Покажемо лише один метод, ідея якого полягає в тому, щоб поділити відношення повторюваності на N. Причина цього стане очевидною згодом. Згідно з даним методом маємо:

$$\frac{T(N)}{N} = \frac{T(\frac{N}{2})}{\frac{N}{2}} + 1$$

Дане рівняння справедливе для будь-якого N, що є степенем 2, тому ми також можемо записати

$$\frac{T(\frac{N}{2})}{\frac{N}{2}} = \frac{T(\frac{N}{4})}{\frac{N}{4}} + 1$$

і

$$\frac{T(\frac{N}{4})}{\frac{N}{4}} = \frac{T(\frac{N}{8})}{\frac{N}{8}} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Тепер додамо усі рівняння. Це означає, що ми додаємо усі доданки з лівої частини та встановлюємо результат рівним сумі всіх доданків у правій частині. Звернемо увагу, що термін T(N/2)/(N/2) з'являється з обох сторін і називається **телескопіюванням** суми. Після того, як все додано, остаточний результат буде таким:

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

бо всі інші доданки ***i*** скорочуються, і є в рівнянні log N, бо всі 1 в кінці цих рівнянь утворюють log N. Помножимо на N праву й ліву частину рівняння й отримаємо:

$$T(N) = N\,logN + N \approx O(N\,logN)$$

Зверніть увагу, що якби ми не поділили на N на початку розв'язання, сума не була б телескопічною. Ось чому довелося ділити на N.

Просторова складність алгоритму: O(n).

Алгоритмічна парадигма: розділяй і володарюй.

Сортування на місці: ні, в типовій реалізації.

Стабільний алгоритм: так.

Сортування злиттям корисно для сортування зв'язних списків за час $O(nLog\,n)$. У випадку зв'язних списків алгоритм відрізняється в основному через різницю у розподілі пам'яті масивів і зв'язних списків. На відміну від масивів, вузли зв'язного списку можуть не бути суміжними в пам'яті. Крім того,

у зв’язний список можна вставляти елементи посередині за O(1) додатковий простір і O(1) час. Таким чином, операцію злиття сортування злиттям можна реалізувати без додаткового місця для зв’язних списків.

У масивах здійснюють довільний доступ, оскільки елементи суміжні в пам’яті. Скажімо, є цілочисельний (4-байтовий) масив A і нехай адреса A[0] буде x, тоді для доступу до A[i] ми можемо отримати прямий доступ до пам’яті за адресою (x + i*4). На відміну від масивів, не можна робити випадковий доступ до зв’язного списка. Сортування злиттям отримує доступ до даних послідовно, і потреба у довільному доступі невелика.

5. Швидке сортування

Як і сортування злиттям, QuickSort є алгоритмом стратегії «Поділяй і володарюй», яку ми будемо обговорювати пізніше. Він вибирає елемент як опорну точку та розбиває заданий масив навколо вибраної опорної точки.

Існує багато різних версій швидкого сортування, які вибирають опорну точку по-різному:

1. Завжди вибирати перший елемент як опорний.
2. Завжди вибирати останній елемент як опорний (реалізований нижче).
3. Вибирати випадковий елемент як опорну точку.
4. Вибирати медіану як опорну точку.

Ключовим процесом у швидкому сортуванні є функція `partition()`. Метою поділів є масив і елемент *x* масиву, як опорна точка, розміщення *x* у правильне положення у відсортованому масиві та розміщення всіх менших елементів (менші за *x*) перед *x* і розміщення всіх більших елементів (більших за *x*) після *x*. Все це потрібно робити за лінійний час. Псевдокод для рекурсивної функції швидкого сортування:

```
/* low --> Початковий індекс, high --> Кінцевий індекс */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi є індексом розділення, arr[pi] тепер в
потрібному місці */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // До pi
        quickSort(arr, pi + 1, high); // Після pi
```

```
    }
}
```

Алгоритм поділу

Існує багато способів зробити поділ. Логіка проста, ми починаємо з крайнього лівого елемента і відстежуємо індекс менших (або рівних) елементів з *i*. Під час проходження, якщо знаходимо менший елемент, ми міняємо поточний елемент місцями з arr[i]. Інакше ми ігноруємо поточний елемент.

```
/* C++ реалізація QuickSort */
#include <bits/stdc++.h>
using namespace std;

// Функція обміну двох значень a і b
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* Ця функція отримує останній елемент як опорний, розміщує
опорний елемент у його правильне положення у відсортованому
масиві, розміщує всі менші за опорну точку елементи ліворуч
від опорної точки, а всі більші елементи — праворуч від
опорної точки */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // опорна точка
    int i = (low - 1); /* Індекс меншого елемента та вказує
правильне положення опорної точки, знайденої на даний
момент */
    for (int j = low; j <= high - 1; j++)
    {
        // Якщо поточний елемент менший за опорну точку
        if (arr[j] < pivot)
        {
            i++; // збільшення індексу меншого елемента
            swap(&arr[i], &arr[j]);
```

```
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* Основна функція, яка реалізує QuickSort
arr[] --> Масив для сортування,
low --> Початковий індекс,
high --> Кінцевий індекс */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi - це індекс розділення, arr[p] тепер у
потрібному місці */
        int pi = partition(arr, low, high);

// Окремо сортуємо елементи перед опорною точкою
// і після неї
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Функція для друку масиву */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Код основної програми
int main()
{
        setlocale(LC_ALL,"ukr");

// Блок оголошення масиву
```

```
    int n;
    cout << "Уведіть розмір масиву n:"<<endl;
    cin >> n;
    int arr[n];

//Блок уведення даних
    cout<<"Уведіть масив:"<<endl;
    for(int i = 0; i < n; i++)
        cin >> arr[i];
    cout << endl;

    quickSort(arr, 0, n - 1);
    cout << "Відсортований масив: \n";
    printArray(arr, n);
    return 0;
}
```

Аналіз QuickSort

Час, затрачений на QuickSort, загалом можна записати так.

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

Перші два терми призначені для двох рекурсивних викликів, останній терм для процесу розбиття. У формулі k – кількість елементів, менших за опорний.

Час, який витрачається на QuickSort, залежить від вхідного масиву та стратегії розділів.

Найгірший випадок: Найгірший випадок відбувається, коли процес розділу завжди вибирає найбільший або найменший елемент як опорний. Якщо ми розглянемо вищенаведену стратегію поділу, де останній елемент завжди вибирається як опорний, найгірший випадок буде, коли масив вже відсортований у порядку збільшення або зменшення. Нижче наведено повторення в гіршому випадку.

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

що еквівалентно

$$T(n) = T(n-1) + \Theta(n)$$

Розв’язок вказаного повторення є $\Theta(n^2)$.

Найкращий випадок: найкращий випадок має місце, коли процес поділу завжди вибирає середній елемент як опорний. Нижче наведено час виконання для кращого випадку.

$$T(n) = 2T(n/2) + \Theta(n)$$

Час виконання сортування з вищевказаною опорною точкою є $\Theta(nLog\ n)$.

Середній випадок:

Щоб провести аналіз середнього випадку, потрібно розглянути усі можливі перестановки масиву та обчислити час, затрачений на кожну перестановку, що виглядає непросто.

Можемо отримати уявлення про середній випадок, розглянувши випадок, коли розбиття поміщає $O(n/9)$ елементів в один набір і $O(9n/10)$ елементів в інший набір. Нижче наведено результат для цього випадку.

$$T(n) = T(n/9) + T(9n/10) + \Theta(n)$$

Час виконання сортування з вищевказаним розбиттям є також $O(nLogn)$.

Незважаючи на те, що найгірший час складності QuickSort становить $O(n^2)$, що більше, ніж у багатьох інших алгоритмах сортування, таких як сортування злиттям і сортування купою, на практиці QuickSort є швидшим, оскільки його внутрішній цикл можна ефективно реалізувати на більшості архітектур і в більшість даних реального світу. Швидке сортування можна реалізувати різними способами, змінивши вибір опорної точки, так що найгірший випадок рідко трапляється для даного типу даних. Однак сортування злиттям, як правило, вважається кращим, коли дані великі та зберігаються у зовнішньому сховищі.

Реалізація QuickSort за замовчуванням не є стабільною. Однак будь-який алгоритм сортування можна зробити стабільним, враховуючи індекси як параметр порівняння.

Відповідно до широкого визначення алгоритму на місці, QuickSort кваліфікується як алгоритм сортування на місці, оскільки використовує додатковий простір лише для зберігання рекурсивних викликів функцій, але не для маніпулювання уведенням даних.

6. 3-стороннє швидке сортування

У простому алгоритмі швидкого сортування ми вибираємо елемент як опорну точку, розбиваємо масив навколо опорної точки і повторюємо для підмасивів ліворуч і праворуч від опорної точки.

Розглянемо масив, який містить багато зайвих елементів. Наприклад, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. Якщо 4 вибрано як опорний пункт у Simple QuickSort, ми вибираємо лише одне значення 4 і рекурсивно обробляємо решту випадків. У 3-Way QuickSort масив arr[l..r] ділиться на 3 частини:

а) елементи arr[l..i] менші опорних.

б) елементи arr[i+1..j-1] рівні опорній точці.

в) елементи arr[j..r], більші за опорні.

```
// C++ програма для 3-way quick sort
#include <bits/stdc++.h>
using namespace std;

/* Ця функція розбиває a[] на три частини
    a) a[l..i] містить усі елементи, менші за опорний
елемент
    b) a[i+1..j-1] містить усі випадки рівні опорній точки
    c) a[j..r] містить усі елементи, більші за опорний
елемент*/
void partition(int a[], int l, int r, int& i, int& j)
{
    i = l - 1, j = r;
    int p = l - 1, q = r;
    int v = a[r];

    while (true) {
        /* Зліва знаходимо перший елемент, більший ніж або
дорівнює v. Цей цикл обов'язково буде завершуватися,
оскільки v є останнім елементом */
while (a[++i] < v)
            ;

        // З правого боку знаходимо перший елемент,
        // який менший або дорівнює v
        while (v < a[--j])
            if (j == l)
                break;

        // Якщо i та j перетинаються, то ми закінчили
        if (i >= j)
            break;

// Поміняти місцями, щоб менший був ліворуч,
// більший йшов праворуч
        swap(a[i], a[j]);

// Переміщуємо ліве входження опорної точки
//на початок масиву та ведемо підрахунок за допомогою p
        if (a[i] == v) {
```

```
            p++;
            swap(a[p], a[i]);
        }

/* Переміщуємо праве входження опорної точки в кінець
масиву і продовжуємо підрахунок за допомогою q  */
        if (a[j] == v) {
            q--;
            swap(a[j], a[q]);
        }
    }

// Переміщення опорного елемента до його
// правильного індексу
    swap(a[i], a[r]);

// Перемістити всі однакові входження ліворуч,
// починаючи від сусіднього до arr[i]
    j = i - 1;
    for (int k = l; k < p; k++, j--)
        swap(a[k], a[j]);

// Перемістити всі однакові входження праворуч з кінця
// до сусіднього з arr[i]
    i = i + 1;
    for (int k = r - 1; k > q; k--, i++)
        swap(a[i], a[k]);
}

// Швидке сортування на основі 3-х шляхів
void quicksort(int a[], int l, int r)
{
    if (r <= l)
        return;

    int i, j;

// Зверніть увагу, що i та j передаються як посилання
    partition(a, l, r, i, j);
```

```
    // Рекурсія
    quicksort(a, l, j);
    quicksort(a, i, r);
}

// Допоміжна функція для друку масиву
void printarr(int a[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d  ", a[i]);
    printf("\n");
}

// Код основної програми
int main()
{
    setlocale(0,".1251");

    // Оголошення блоку змінних
    int n;
    cout << "Уведіть розмір масиву n:"<<endl;
    cin >> n;
    int arr[n];

    //Блок уведення даних
    cout<<"Уведіть масив:"<<endl;
    for(int i = 0; i < n; i++)
        cin >> arr[i];
    cout << endl;

    printarr(arr, n);
    // Функція виклику сортування
    quicksort(arr, 0, n - 1);
    printarr(arr, n);
    return 0;
}
```

Слід пам’ятати, що різні алгоритми ефективні для певної кількості елементів у одновимірному масиві. Щоб визначити, який алгоритм доречніше використовувати при розв’язуванні певної задачі, наведемо таблицю оцінок складності сортування різними алгоритмами.

Алгоритм сортування	**Часова складність алгоритму**		
	Найкращий	Середній	Найгірший
Вибором	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Бульбашкою	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Вставками	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Пірамідальне	$\Omega(n\,log(n))$	$\theta(n\,log(n))$	$O(n\,log(n))$
Швидке	$\Omega(n\,log(n))$	$\theta(n\,log(n))$	$O(n^2)$
Злиттям	$\Omega(n\,log(n))$	$\theta(n\,log(n))$	$O(n\,log(n))$
Блочне	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$
За розрядами	$\Omega(nk)$	$\theta(nk)$	$O(nk)$
Підрахунком	$\Omega(n+k)$	$\theta(n+k)$	$O(n+k)$

🔔 Контрольні запитання:

1. Дайте означення сортування.
2. Що таке алгоритм сортування?
3. Який формат функції сортування у STL?
4. У чому суть алгоритму сортування на місці?
5. Яке сортування називають зовнішнім, а яке внутрішнім?
6. Перелічіть параметри алгоритмів сортування.
7. У чому полягає ідея сортування вставкою?
8. Як відбувається сортування методом бінарної вставки?
9. Охарактеризуйте сортування злиттям.
10. Опишіть ідею швидкого сортування.
11. У чому суть 3-стороннього швидкого сортування?

Тема №9. ДЕРЕВА

План лекції:

1. Дерева. Попередні означення
2. Префіксні дерева
3. Бінарні дерева
4. Двійкове дерево пошуку
5. Алгоритм Прима

Джерела:

[1, §2.1], [2, §6.2], [3, Глава 3], [4, Глава 2], [5, Chapter 12, 13,18], [6, Chapter 6], [7, §2.3], [10, §3.4], [11, Chapter 4]

Представлення списків мають фундаментальне обмеження: або пошук, або вставку можна зробити ефективними, але не обидві операції одночасно. Хоча стеки та черги відображають певну ієрархію, вони обмежені лише одним виміром.

Для великої кількості введених даних лінійний час доступу до пов'язаних списків є непомірним. У цій темі ми розглянемо просту структуру даних, для якої середній час виконання більшості операцій дорівнює $O(log\ N)$. Буде описана концептуально проста модифікація цієї структури даних, яка гарантує зазначений вище час, обмежений у гіршому випадку, і обговоримо другу модифікацію, яка по суті дає $O(\ log\ N)$ часу виконання на операцію для довгої послідовності інструкцій.

Деревовидні структури дозволяють як ефективний доступ, так і оновлення до великих колекцій даних.

Структура даних, про яку ми згадуємо, відома як двійкове дерево пошуку. Дерево бінарного пошуку є основою для реалізації двох класів бібліотечних колекцій, set і map, які використовуються в багатьох програмах. Дерева взагалі є дуже корисними абстракціями в інформатиці.

Зокрема, бінарні дерева часто вживані й відносно легко реалізовані. Але двійкові дерева корисні для багатьох речей, окрім пошуку. Лише кілька прикладів додатків, які дерева можуть прискорити, включають визначення пріоритетів завдань, опис математичних виразів і синтаксичних елементів комп'ютерних програм або організацію інформації, необхідної для керування алгоритмами стиснення даних.

1. Попередні означення

На відміну від натуральних дерев, ці дерева зображені догори дном з корінням угорі, а листям (кінцевими вузлами) внизу. Дерево можна визначити кількома способами. Розглянемо рекурсивне означення дерева.

Означення.

1. Порожня структура – це порожнє дерево.

2. Якщо t1, . . . , tk – це роз'єднані дерева, тоді структура, корінь якої має дочірніми коріннями t1, . . . , tk також дерево.

3. Деревами є лише структури, створені за правилами 1 і 2.

Вказане означення ще називають індуктивним. Наведемо також і родове означення.

Означення. Дерево – це структура даних, що являє собою сукупність елементів і відношень, які утворюють ієрархічну структуру цих елементів.

Дерево — це сукупність вузлів. Колекція може бути порожньою; в іншому випадку дерево складається з виділеного вузла *r*, який називають коренем, і нуля або більше непорожніх (під)дерев T1, T2, ..., Tk, кожне з коренів яких з'єднане направленою дугою від r.

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

Над деревами визначено наступні основні операції:

• пошук вузла із заданим ключем;

• включення вузла у дерево;

• видалення вузла;

• пошук по дереву;

• обхід дерева.

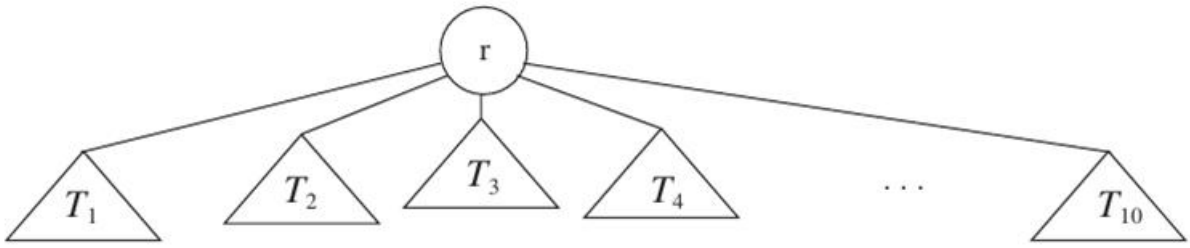
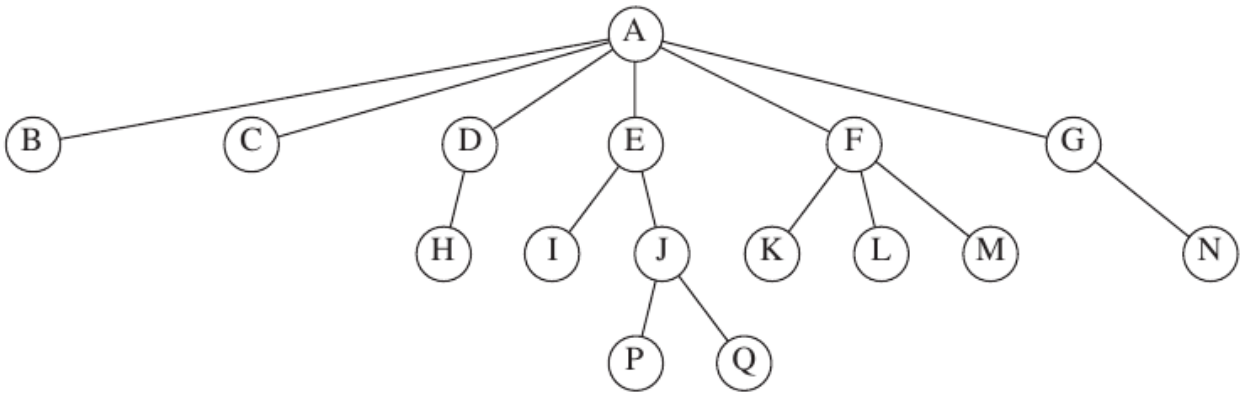


Рис. 9.1. Узагальнене дерево

Кожна така множина ***ti***, в свою чергу, є деревом та називається **піддеревом** вузла ***r***. Кажуть, що корінь кожного піддерева є нащадком r, а ***r*** є батьківським коренем кожного піддерева. На рис. 9.1 показано типове дерево з використанням рекурсивного визначення.

З рекурсивного визначення ми знаходимо, що дерево — це набір з N вузлів, один з яких є коренем, який має N − 1 ребро. Те, що існує N − 1 ребро, випливає з того факту, що кожне ребро з'єднує деякий вузол зі своїм батьківським, а кожен вузол, крім кореня, має одного предка (див. рис. 9.2).



Малюнок 9.2. Дерево

У дереві на рис. 9.2 коренем є A. Вузол F має A як предка і K, L та M як нащадків. Кожен вузол може мати довільну кількість нащадків, можливо, нуль.

Означення. Листками дерева називають вершини, в які входить одна дуга і не виходить жодної дуги.

Листки на дереві 9.2 є: B, C, H, I, P, Q, K, L, M і N. Вузли з одним і тим же предком є братами і сестрами; таким чином, K, L і M є братами і сестрами.

Кожне дерево має такі властивості:

1) існує вузол, в який не входить ні одна дуга (корінь);

2) у кожну вершину, крім кореня, входить одна дуга.

Шлях від вузла n_1 до n_k визначається як послідовність вузлів n_1, n_2, ..., n_k таких, що n_i є батьківським для n_{i+1} для $1 \leq i < k$. Довжиною цього шляху є кількість ребер на шляху, а саме $k - 1$. Від кожного вузла до самого себе є шлях нульової довжини. Зверніть увагу, що в дереві є рівно один шлях від кореня до кожного вузла.

Означення. Висота (глибина) дерева називається число, яке визначається кількістю рівнів, на яких розташовуються його вершини.

Фактично, для будь-якого вузла n_i **глибина n_i** є довжиною унікального шляху від кореня до n_i. Таким чином, корінь знаходиться на глибині 0. Висота n_i — це довжина найдовшого шляху від n_i до листка. Таким чином, усі листки знаходяться на висоті 0. Висота дерева дорівнює висоті кореня. Для дерева на малюнку 2 вершина E знаходиться на глибині 1 і висоті 2; F знаходиться на глибині 1 і висоті 1; висота дерева 3. Глибина дерева дорівнює глибині найглибшого листка; це завжди дорівнює висоті дерева.

Якщо є шлях від n_1 до n_2, то n1 є предком n_2 і n_2 є нащадком n_1. Якщо $n1 \neq n2$, то n_1 є власним предком n_2 і n_2 є власним нащадком n_1.

Означення. Степенем вершини в дереві називається кількість дуг, які з неї виходить.

Степінь дерева дорівнює максимальному степеню вершин, що входять у дерево. Листками у дереві є вершини, що мають степінь нуль.

За величиною степеня дерева розрізняють два типи дерев:

- ✓ **бінарні** – степінь дерева не більше двох;
- ✓ **сильнорозгалужені** – степінь дерева довільний.

Оголошення дерева на С++ може бути:

```
typedef struct TreeNode
    {
    int Data; //Поле даних
    TreeNode *firstChild;
    TreeNode *nextSibling;
    };
```

2. Дерева та префіксні дерева

Давайте розглянемо дерево імен, записаних англійськими літерами (див. рис 9.3). Кожен з прямокутників має 27 індексів, адже в іменах можуть бути і апострофи.

Будемо називати цю структуру **префіксне дерево** або **trie** (trie — це таке дотепне ім'я для дерева, що оптимізоване для пошуку — reTRIEval, вимовляється як "трай").

Означення. Префіксним деревом називається структура даних, що дозволяє зберігати асоціативний масив, ключами якого найчастіше є рядки.

Уважається, що префіксне дерево містить рядок-ключ тоді і тільки тоді, коли цей рядок можна прочитати на шляху з кореня до деякого (єдиного для цього рядка) виділеного вузла. У листку дерева не зберігається ключ. Значення ключа можна отримати проглядянням всіх батьківських вузлів, кожний з яких зберігає один або кілька символів алфавіту. Корінь дерева пов'язаний з порожнім рядком. Нащадки вузла мають загальний префікс. Значення, пов'язані з ключем, пов'язані тільки з листками і, можливо, деякими внутрішніми вузлами.

Ці префіксні дерева використовують багато пам'яті, адже кожен вузол виділяє місце для багатьох індексів, й більшість з цих місць буде порожнім, принаймні спочатку, коли є лише кілька рядків.

Але ми отримуємо швидкість та витрачаємо менше часу, адже час вставки дорівнює $O(1)$, бо не існує нескінченно довгих імен. Найдовше слово у словнику цього практичного завдання матиме 40 літер, але це буде стала величина, яка не залежатиме від інших імен у цій структурі. Щодо часу виконання, він залежатиме від довжини рядка, яка асимптотично сягає $O(1)$.

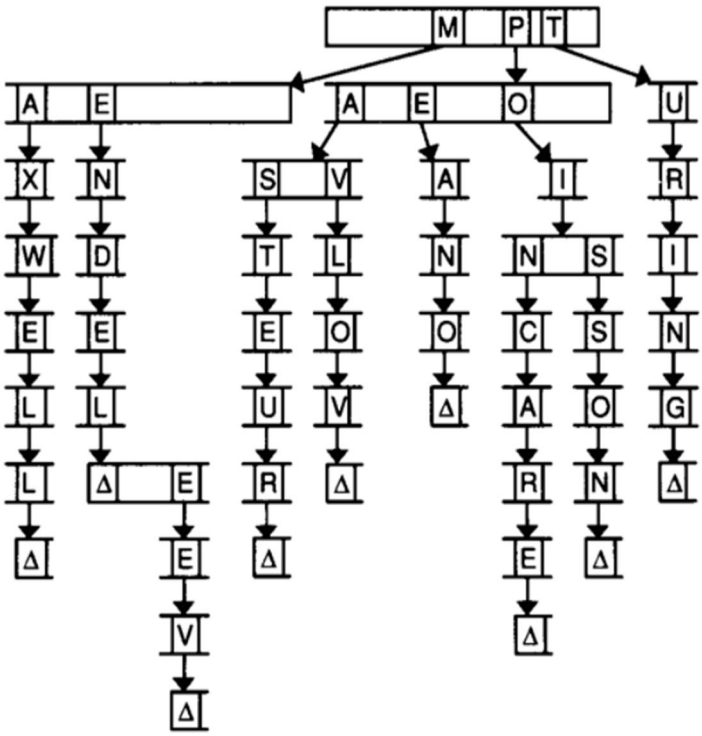


Рис. 9.3. Префіксне дерево

Ми можемо реалізувати це в наступний спосіб:

```
typedef struct node
{
  bool word;
  struct node* children[27];
}
node;
…
node* trie;
```

Кожен вузол node матиме булеве значення `bool`, яке відповідатиме за кінець слова та масив із 27 вказівниками на інші вузли.

Щоб оголосити наше префіксне дерево, ми повинні мати один node*, що вказує на кореневий елемент, як у зв’язних списках.

Виділяють три основні операції над префіксним деревом:

1) перевірка наявності ключа в дереві;
2) видалення ключа з дерева;
3) вставка нового ключа (можливо з якоюсь додатково пов’язаною інформацією).

Нехай n – довжина рядка, який запитують/ видаляють/ вставляють, а σ – розмір алфавіту, тобто кількість різних символів на ребрах цього префіксного дерева.

Уважатимемо, що вузол x має k синів (при цьому $k \leq \sigma$). Позначимо через $x_1, x_2, \ldots, x_k$ посилання цих синів, а через $a_1, a_2, \ldots, a_k$ – символи, які позначають ребра, що з'єднують x з відповідними синами. Найбільш простий спосіб організувати навігацію у вузол x – зберігати динамічний масив пар (a_i, x_i). При такому підході всі три операції виконуються за $O(n\sigma)$.

Якщо вставка та видалення не використовуються, то краще відсортувати пари по ключу a_i тоді операцію перевірки наявності ключа в префіксному дереві можна буде виконувати за $O(n \log \sigma)$ за допомогою бінарного пошуку у вузлах.

Можна домогтися часу виконання $O(n \log \sigma)$ для всіх трьох операцій, якщо зберігати пари (a_i, x_i) відсортованими за ключом a_i у якомусь збалансованому бінарному дереві пошуку, наприклад, у червоно-чорному дереві або АВЛ-дереві.

3. Бінарні дерева

Означення. Дерево, елементи якого мають не більше 2 дочірніх елементів, називається **бінарним деревом**.

Оскільки кожен елемент у двійковому дереві може мати лише 2 дочірніх, ми зазвичай називаємо їх лівим і правим дочірнім елементом.

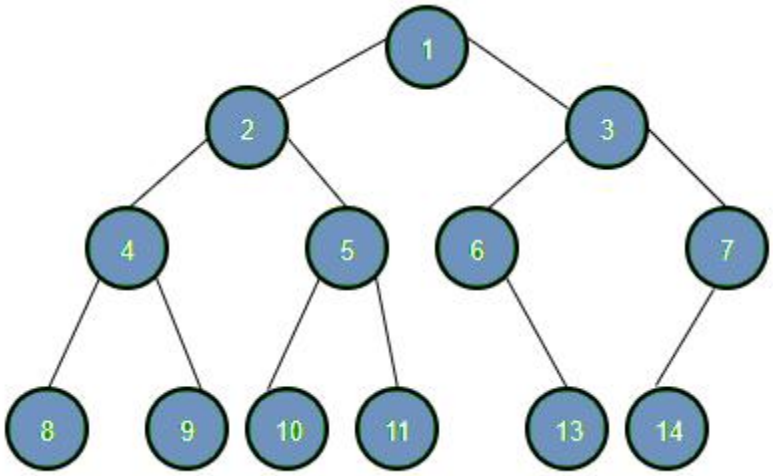


Рис. 9.3. Двійкове дерево

Вузол бінарного дерева містить такі частини.

1. Дані.
2. Вказівник на ліву дитину.
3. Вказівник на праву дитину.

Оголошення бінарного дерева є типовим:

```
struct TreeNode
{
    Object element;
```

```
TreeNode* left;
TreeNode* right;
};
```

Обхід дерев (Inorder , Preorder та Postorder)

На відміну від лінійних структур даних (масив, зв'язаний список, черги, стеки тощо), які мають лише один логічний спосіб їх проходження, дерева можна обходити різними способами. Нижче наведено загальновживані способи проходження дерев для дерева, що зображене на рис. 9.4.

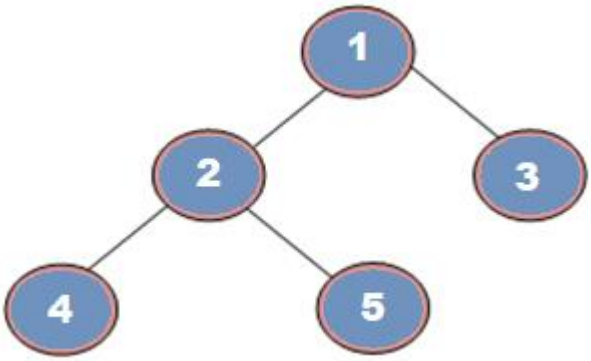


Рис. 9.4. Бінарне дерево

Перші проходи в глибину:

(a) Симетричний обхід дерева (ліворуч, корінь, правий): 4 2 5 1 3

(b) Обхід дерева у прямому порядку (корінь, ліворуч, праворуч) : 1 2 4 5 3

(c) Обхід дерева у зворотному порядку (ліворуч, праворуч, корінь):
4 5 2 3 1

Обхід порядку в ширину або порівневий обхід: 1 2 3 4 5

Алгоритм симетричного обходу дерева (Inorder):

1. Обхід лівого піддерева, тобто виклик Inorder (ліве піддерево)
2. Відвідати корінь.
3. Обхід правого піддерева, тобто виклик Inorder (праве піддерево)

Використання алгоритму симетричного обходу дерева

У випадку бінарних дерев пошуку (BST), обхід за порядком дає вузли в порядку, що не зменшуються. Щоб отримати вузли BST в порядку, не зростаючому, можна використовувати варіант обходу.

Приклад 1. Симетричний обхід дерева для наведеної вище фігури дорівнює 4 2 5 1 3.

Алгоритм обходу дерева у прямому порядку:

1. Відвідати корінь.
2. Обхід лівого піддерева, тобто виклик Preorder(left-subtree)
3. Обхід правого піддерева, тобто виклик Preorder(right-subtree)

Використання алгоритму обходу прямого порядку

Обхід дерева у прямому порядку використовується для створення копії дерева. Обхід прямого порядку також використовується для отримання префіксного виразу в дереві виразів.

Приклад 2. Обхід дерева у прямому порядку (рис. 9.4) становить 1 2 4 5 3.

Алгоритм обходу дерева у зворотньому порядку:

1. Обхід лівого піддерева, тобто виклик Postorder (ліве піддерево)
2. Обхід правого піддерева, тобто виклик Postorder (праве піддерево)
3. Відвідати корінь.

Використання алгоритму обходу у зворотньому порядку

Для видалення дерева використовується обхід дерева у зворотньому порядку (Postorder). Обхід дерева у зворотньому порядку також корисний для отримання постфіксного виразу дерева виразів.

Приклад 3. Обхід дерева з рис. 9.4 у зворотньому порядку буде 4 5 2 3 1.

```
// Програма C++ для різних обходів дерев
#include <iostream>

using namespace std;

/* Вузол бінарного дерева має дані, вказівник на ліву
дочірню частину і вказівник на праву дочірню частину */
struct Node {
  int data;
  struct Node *left, *right;
  Node(int data)
  {
    this->data = data;
    left = right = NULL;
  }
};

//Друкування дерева у зворотному порядку
void printPostorder(struct Node* node)
{
  if (node == NULL)
    return;

// перший повтор у лівому піддереві
```

```
  printPostorder(node->left);

// потім повторюється на правому піддереві
  printPostorder(node->right);

// Тепер розберіться з вузлом
  cout << node->data << " ";
}

//Друкування дерева симетричним обходом
void printInorder(struct Node* node)
{
  if (node == NULL)
    return;

/* перший повтор на лівій дитині */
  printInorder(node->left);

/* потім надрукувати дані вузла */
  cout << node->data << " ";

/* тепер повторюється для правої дитини */
  printInorder(node->right);
}

//Друкування дерева у прямому порядку
void printPreorder(struct Node* node)
{
  if (node == NULL)
    return;

/* перший друк даних вузла */
  cout << node->data << " ";

/* потім повторюється на лівому піддереві */
  printPreorder(node->left);

/* тепер повторюється в правому піддереві */
  printPreorder(node->right);
}
```

```
int main( )
{
  struct Node* root = new Node(1);
  root->left = new Node(2);
  root->right = new Node(3);
  root->left->left = new Node(4);
  root->left->right = new Node(5);

  cout << "\ nПопередній обхід бінарного дерева є \n";
  printPreorder (root);

  cout << "\ nОбхід двійкового дерева в порядку \n";
  printInorder(root);

  cout << "\ nПостордер обхід бінарного дерева \n";
  printPostorder (root);

  return 0;
}
```

Часова складність алгоритму: O(n).

Давайте розглянемо різні граничні випадки.

Функція складності T(n) — для всіх задач, де задіяний обхід дерева – може бути визначена як:

$$T(n) = T(k) + T(n - k - 1) + c,$$

де k – кількість вузлів з одного боку кореня і $n - k - 1$ з іншого.

Зробимо аналіз граничних умов:

Випадок 1. Перекошене дерево (одне з піддерев порожнє, а інше непорожнє) k дорівнює 0 у цьому випадку.

$T(n) = T(0) + T(n-1) + c$

$T(n) = 2\,T(0) + T(n-2) + 2c$

$T(n) = 3\,T(0) + T(n-3) + 3c$

$T(n) = 4\,T(0) + T(n-4) + 4c$

……

$T(n) = (n-1)T(0) + T(1) + (n-1)c$

$T(n) = nT(0) + (n)c$

Значення T(0) буде деякою константою, скажімо, d. (обхід порожнього дерева займе деякий константний час).

$T(n) = n(c + d)$

$T(n) = \Theta(n)$

Випадок 2. І ліве, і праве піддерева мають рівну кількість вузлів.

$T(n) = 2T(\lfloor n/2 \rfloor) + c$

Функція **floor** — це функція, яка отримує як вхідне дійсне число *x*, і повертає на вихід найбільше ціле число, менше або рівне *x*, позначається floor(*x*) або $\lfloor x \rfloor$. Наприклад, $\lfloor 2{,}4 \rfloor = 2$, $\lfloor -2{,}4 \rfloor = -3$.

Ця рекурсивна функція має стандартну форму $(T(n) = aT(n/b) + (-)(n))$ для головного методу. Якщо ми розв'язуємо цим методом, то отримаємо n.

Допоміжний простір: якщо ми не враховуємо розмір стеку для викликів функцій, тоді $O(1)$, інакше $O(h)$, де h – висота дерева.

Висота перекошеного дерева дорівнює n (кількість елементів), тому найгірша складність простору – $O(n)$, а висота – $O(log\ n)$ для збалансованого дерева, тому найкраща складність простору —– $O(log\ n)$.

Симетричний обхід дерева без рекурсії

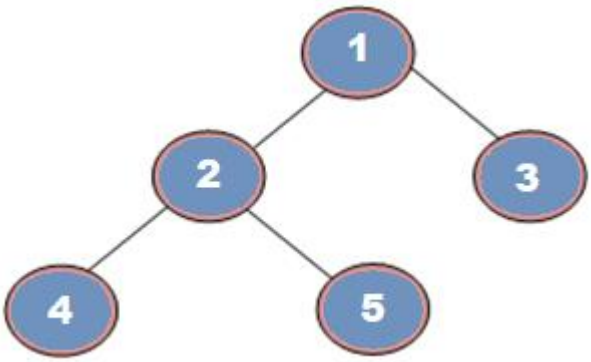


Рис. 9.5. Двійкове дерево прикладу 4.

Використання структури Stack є очевидним способом обходу дерева без рекурсії. Нижче наведено алгоритм обходу бінарного дерева за допомогою стека.

1) Створити порожній стек S.

2) Ініціалізувати поточний вузол як root.

3) Вставити поточний вузол до S і встановити current = current->left, доки current не стане NULL

4) Якщо current == NULL і стек не порожній, то

а) Вилучити верхній елемент зі стека.

б) Роздрукувати елемент – верхівку стека, встановити current = popped_item->right

в) Перейдіть до кроку 3.

5) Якщо current == NULL і стек порожній, то закінчити.

Приклад 4. Розглянемо наведене дерево на рис. 9.5. Здійснити симетричний обхід дерева без рекурсії.

Крок 1. Створити порожній стек: S = NULL.

Крок 2. Встановити поточний вузол, як адресу root: current -> 1.

Крок 3. Занесення поточного вузла до стеку і встановити current = current->left.

Доки current не стане NULL:

current -> 1

push 1: Stack S -> 1

current -> 2

push 2: Stack S -> 2, 1

current -> 4

push 4: Stack S -> 4, 2, 1

current = NULL

Крок 4. Демонстрація та вилучення з S

a) Pop 4: Stack S -> 2, 1

b) print "4"

c) current = NULL /* праворуч від 4 і перейти до кроку 3.

Оскільки поточне посилання NULL, крок 3 нічого не робить. */

Знову демонстрація та вилучення з кроку 4.

a) Pop 2: Stack S -> 1

b) print "2"

c) current -> 5/*праворуч від 2 і перейти до кроку 3 */

Крок 3. Занесення 5 до стека і присвоєння поточного покажчика NULL

Stack S -> 5, 1

current = NULL

Крок 4. демонстрація та вилучення з S

a) Pop 3: Stack S -> NULL

b) print "3"

c) current = NULL /*праворуч від 5 і перейти до кроку 3.

Оскільки поточне посилання NULL, крок 3 нічого не робить. */

Знову демонстрація та вилучення з кроку 4.

a) Pop 1: Stack S -> NULL

b) print "1"

c) current -> 3 /*праворуч від 1 */

Крок 3. Занесення 3 до стека і створення поточного посилання NULL

Стек S -> 3

current = NULL

Крок 4. демонстрація та вилучення з S

a) Pop 1: Stack S -> NULL

b) print "1"

c) current -> 3 /*праворуч від 3 */

Обхід припиняється, оскільки стек S порожній, а поточне посилання NULL.

```
// Програма C++ для друку впорядкованого обходу з
// використанням стека.
#include<bits/stdc++.h>

using namespace std;

/* Вузол бінарного дерева має дані, вказівник на ліву
дочірню частину і вказівник на праву дочірню частину */
struct Node
{
  int data;
  struct Node* left;
  struct Node* right;

  Node (int data)
  {
    this->data = data;
    left = right = NULL;
  }
};

/* Ітеративна функція для дерева із симетричним порядком
обходу */
void inOrder ( struct Node *root)
{
  stack<Node *> s;
  Node *curr = root;

  while (curr != NULL || s.empty() == false)
  {
    /* Досягти крайнього лівого вузла поточного вузла */
    while (curr != NULL)
    {
```

```
      /* розмістити вказівник на вузол дерева в стеку
перед тим, як обійти ліве піддерево вузла */
      s.push(curr);
      curr = curr->left;
    }

    /* На цьому етапі поточний покажчик має бути NULL */
    curr = s.top();
    s.pop();

    cout << curr->data << " ";

    /* ми відвідали вузол і його ліве піддерево. Тепер
настала черга правого піддерева */
    curr = curr->right;

  } /* end of while */
}

/* Програма драйвера для перевірки вищевказаних функцій*/
int main( )
{

/* Побудоване двійкове дерево є
1
/ \
2 3
/ \
4 5
*/
  struct Node *root = new Node(1);
  root->left    = new Node(2);
  root->right    = new Node(3);
  root->left->left = new Node(4);
  root->left->right = new Node(5);

  inOrder(root);
  return 0;
}
```

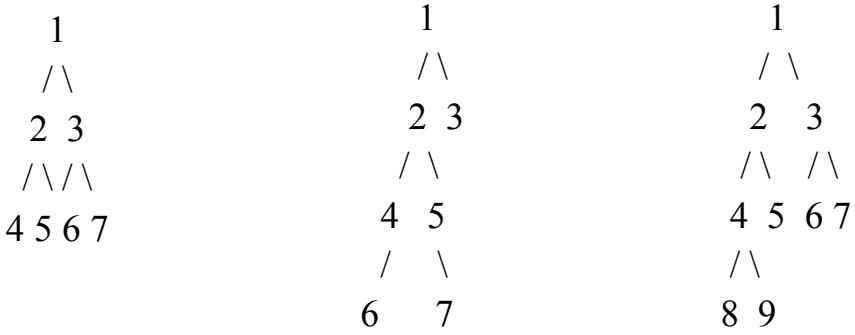
Часова складність алгоритму: O(n).

4. Побудова повного бінарного дерева з заданими прямим та оберненим порядком обходу

Приклад 5. Враховуючи два масиви, які представляють прямий і зворотній обхід повного бінарного дерева, побудувати двійкове дерево.

Повне бінарне **дерево** – це двійкове дерево, в якому кожен вузол має 0 або 2 вузли - нащадки.

Нижче наведено приклади повних дерев.



Неможливо побудувати загальне двійкове дерево на основі прямого та зворотного обходів. Але якщо знати, що бінарне дерево повне, то можна побудувати дерево без двозначності. Давайте розберемося в цьому за допомогою наступного прикладу.

Розглянемо два задані масиви як `pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7}` і `post[] = {8, 9, 4, 5, 2, 6, 7 , 3, 1};`

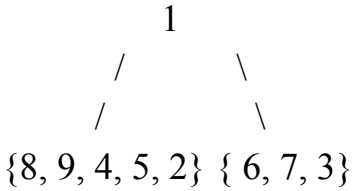


Рис. 9.6

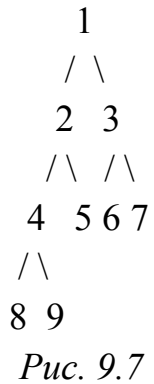


Рис. 9.7

У `pre[]` крайній лівий елемент є коренем дерева. Оскільки дерево – повне, а розмір масиву більше 1, то значення поряд із 1 у `pre[]` має бути лівим коренем нащадка. Отже, ми знаємо, що 1 є кореневим, а 2 – лівим нащадком. Як знайти всі вузли в лівому піддереві? Ми знаємо, що 2 є коренем усіх вузлів лівого піддерева. Усі вузли перед 2 в `post[]` мають бути в лівому піддереві. Тепер ми

знаємо, що 1 – це корінь, елементи {8, 9, 4, 5, 2} знаходяться в лівому піддереві, а елементи {6, 7, 3} – у правому піддереві (рис. 9.6).

Ми рекурсивно дотримуємося наведеного вище підходу і отримуємо наступне дерево (рис. 9.7).

```
/* програма для побудови повного бінарного дерева */
#include <bits/stdc++.h>
using namespace std;

/* Вузол бінарного дерева має дані, вказівник на ліве
піддерево і вказівник на праве піддерево */
struct node
{
  int data;
  node *left;
  node *right;
};


// Допоміжна функція для створення вузла
node* newNode (int data)
{
  node* temp = new node();

  temp->data = data;
  temp->left = temp->right = NULL;

  return temp;
}

/* Рекурсивна функція для побудови Full з pre[ ] і post[].
  preIndex використовується для відстеження індексу в pre[
].
  l – найменший індекс, а h – найбільший індекс для
поточного підмасиву в post[ ] */
node* constructTreeUtil (int pre[], int post[], int*
preIndex,
                int l, int h, int size)
{
```

```
  // Base case
  if (*preIndex >= size || l > h)
    return NULL;

  /* Перший вузол при прямому обході - це корінь. Тому слід взяти вузол у Preindex прямого обходу та зробити його коренем та збільшити Preindex на 1 */
  node* root = newNode ( pre[*preIndex] );
  ++*preIndex;

  /* Якщо поточний підмасив має лише один елемент, не потрібно повторювати */
  if (l == h)
    return root;

// Шукати наступний елемент pre [] в post []
  int i;
  for (i = l; i <= h; ++i)
    if (pre[*preIndex] == post[i])
      break;

  /* Використовується індекс елемента, знайденого в postorder, щоб розділити масив postorder на дві частини: ліве піддерево та праве піддерево */
  if (i <= h)
  {
    root->left = constructTreeUtil (pre, post, preIndex, l, i, size);
    root->right = constructTreeUtil (pre, post, preIndex, i + 1, h-1, size);
  }

  return root;
}
/* Основна функція для побудови повного двійкового дерева. Ця функція в основному використовує constructTreeUtil ( ) */
node *constructTree (int pre[], int post[], int size)
{
  int preIndex = 0;
```

```
  return constructTreeUtil (pre, post, &preIndex, 0, size
- 1, size);
}

/* Функція - утиліта для друку симетричного обходу
двійкового дерева*/
void printInorder (node* node)
{
  if (node == NULL)
    return;
  printInorder(node->left);
  cout<<node->data<<" ";
  printInorder(node->right);
}

// Програма для перевірки вищевказаних функцій
int main ()
{
  int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};
  int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};
  int size = sizeof( pre ) / sizeof( pre[0] );
  setlocale(LC_ALL, "ukr");

  node *root = constructTree(pre, post, size);

  cout<<"Симетричний обхід дерева: \n";
  printInorder(root);

  return 0;
}
```

5. Вставлення в двійкове дерево в порівневому порядку

Приклад 6. За допомогою двійкового дерева та ключа вставити ключ у двійкове дерево у першій доступній позиції в порівневому порядку.

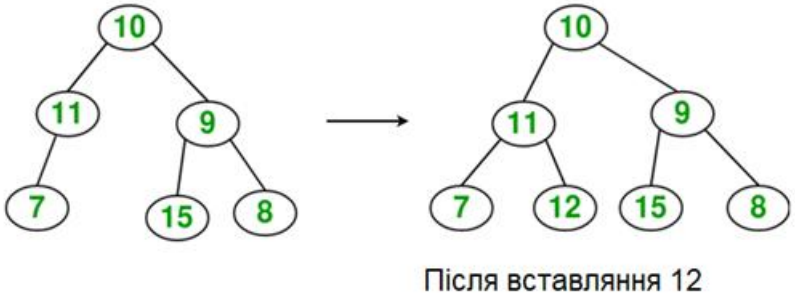


Рис. 9.8. Вставлення нового вузла

Ідея полягає в тому, щоб виконати ітераційний обхід заданого дерева в порівневому порядку за допомогою черги. Якщо ми знайдемо вузол, лівий вузол - нащадок якого порожній, ми створюємо новий ключ як лівий вузол - нащадок. Інакше, якщо ми знайдемо вузол, правий вузол - нащадок якого порожній, ми робимо новий ключ як правий вузол - нащадок. Продовжуємо обходити дерево, доки не знайдемо вузол, лівий або правий елемент - нащадок якого порожній.

```
// Програма C++ для вставки елемента в двійкове дерево
#include <iostream>
#include <queue>
using namespace std;

/* Вузол бінарного дерева має дані, вказівник на ліву
дочірню частину і вказівник на праву дочірню частину */

struct Node {
  int data;
  Node* left;
  Node* right;
};

/* Функція створення нового вузла */

Node* CreateNode(int data)
{
  Node* newNode = new Node();
  if (!newNode) {
    cout << "Memory error\n";
    return NULL;
  }
  newNode->data = data;
```

```
  newNode->left = newNode->right = NULL;
  return newNode;
}

/* Функція для вставки елемента в двійкове дерево */

Node* InsertNode(Node* root, int data)
{
// Якщо дерево порожнє, призначити нову адресу вузла root
  if (root == NULL) {
    root = CreateNode(data);
    return root;
  }

/* В іншому випадку виконуємо обхід у порівневому
порядку, поки не знайдемо порожнє місце, тобто або лівий
дочірній вузол, або правий дочірній вузол вказує на
NULL.*/
  queue<Node*> q;
  q.push(root);

  while (!q.empty()) {
    Node* temp = q.front();
    q.pop();

    if (temp->left != NULL)
      q.push(temp->left);
    else {
      temp->left = CreateNode(data);
      return root;
    }
    if (temp->right != NULL)
      q.push(temp->right);
    else {
      temp->right = CreateNode(data);
      return root;
    }
  }
}
/* Обхід двійкового дерева в симетричному порядку */
```

```
void inorder(Node* temp)
{
  if (temp == NULL)
    return;
  inorder(temp->left);
  cout << temp->data << ‘  ‘ ;
  inorder(temp->right);
}

// Код драйвера
int main( )
{
    setlocale(LC_ALL, "ukr");
    Node* root = CreateNode ( 10);
    root->left = CreateNode ( 11);
    root->left->left = CreateNode ( 7);
    root->right = CreateNode ( 9);
    root->right->left = CreateNode ( 15);
    root->right->right = CreateNode ( 8);

    cout << " Обхід у симетричному порядку перед
вставкою: ";
    inorder(root);
    cout << endl ;

    int ключ = 12;
    root = InsertNode(root, key);
    cout << " Обхід у симетричному порядку після вставки:
";
    inorder(root);
    cout << endl ;

    return 0;
}
```

6. Видалення в двійковому дереві

Приклад 7. Враховуючи двійкове дерево, видаліть з нього вузол, переконавшись, що дерево зменшується знизу (тобто видалений вузол замінюється на найнижчий і крайній правий вузол).

Дана дія відрізняється від видалення вузла в BST. Тут не встановлений порядок між елементами, тому ми замінюємо вилучений елемент останнім елементом.

Алгоритм

1. Починаючи з кореня, знайти найглибший крайній правий вузол у двійковому дереві та вузол, який потрібно видалити.

2. Замінити дані вузла, який потрібно видалити, на дані найглибшого крайнього правого вузла.

3. Потім видалити найглибший крайній правий вузол.

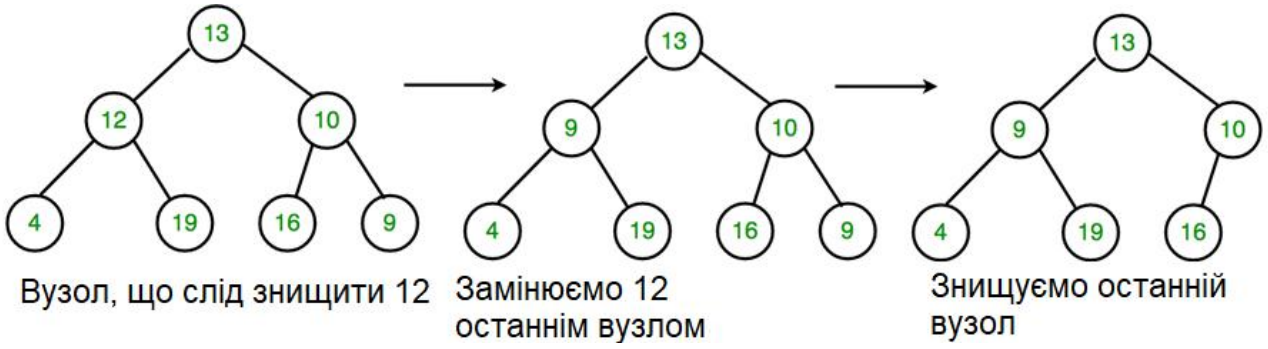


Рис. 9.9. Знищення вузла в двійковому дереві

```
/* Програма C++ для видалення елемента у двійковому
дереві */
#include <bits/stdc++.h>
using namespace std;

/* Вузол бінарного дерева має ключ, покажчик лівий
дочірний елемент і вказівник на правий дочірній елемент */
struct Node {
  int key;
  struct Node *left, *right;
};

/* функція для створення нового вузла дерева і покажчик
повернення */
struct Node* newNode(int key)
{
  struct Node* temp = new Node;
  temp->key = key;
  temp->left = temp->right = NULL;
  return temp;
```

```
};

/* Симетричний обхід двійкового дерева*/
void inorder(struct Node* temp)
{
  if (!temp)
    return;
  inorder(temp->left);
  cout << temp->key << " ";
  inorder(temp->right);
}

/* функція для видалення даного найглибшого вузла
( d_node ) у двійковому дереві */
void deletDeepest(struct Node* root,
         struct Node* d_node)
{
  queue<struct Node*> q;
  q.push(root);
  // Do level order traversal until last node
  struct Node* temp;
  while (!q.empty()) {
    temp = q.front();
    q.pop();
    if (temp == d_node) {
      temp = NULL;
      delete (d_node);
      return;
    }
    if (temp->right) {
      if (temp->right == d_node) {
        temp->right = NULL;
        delete (d_node);
        return;
      }
      else
        q.push(temp->right);
    }
    if (temp->left) {
      if (temp->left == d_node) {
```

```
            temp->left = NULL;
            delete (d_node);
            return;
          }
          else
            q.push(temp->left);
        }
      }
    }

    /* функція видалення елемента у двійковому дереві */
    Node* deletion(struct Node* root, int key)
    {
      if (root == NULL)
        return NULL;
      if (root->left == NULL && root->right == NULL) {
        if (root->key == key)
          return NULL;
        else
          return root;
      }
      queue<struct Node*> q;
      q.push(root);

      struct Node* temp;
      struct Node* key_node = NULL;
    /* Виконати обхід, щоб знайти найглибший вузол (temp)
і вузол, який потрібно видалити (key_node) */
      while (!q.empty()) {
        temp = q.front();
        q.pop();
        if (temp->key == key)
          key_node = temp;
        if (temp->left)
          q.push(temp->left);
        if (temp->right)
          q.push(temp->right);
      }
      if (key_node != NULL) {
        int x = temp->key;
```

```
        deletDeepest(root, temp);
        key_node->key = x;
      }
      return root;
    }
    // Код головної програми
    int main()
    {
      setlocale(LC_ALL, "ukr");
      struct Node* root = newNode(10);
      root->left = newNode(11);
      root->left->left = newNode(7);
      root->left->right = newNode(12);
      root->right = newNode(9);
      root->right->left = newNode(15);
      root->right->right = newNode(8);
      cout << " Обхід у симетричному порядку перед
видаленням: ";
      inorder(root);

      int key = 11;
      root = deletion(root, key);
      cout << endl;
      cout << " Обхід у симетричному порядку після
видалення: ";
      inorder(root);
      return 0;
    }
```

7. Двійкове дерево пошуку

Означення. Двійкове дерева пошуку (BST) — це структура даних на основі вузлів, яка має такі властивості:

✓ Ліве піддерево вузла містить лише вузли з ключами, меншими за ключ вузла. Праве піддерево вузла містить лише вузли з ключами, більшими за ключ вузла. Кожне ліве та праве піддерево також має бути двійковим деревом пошуку.

✓ Не повинно бути повторюваних вузлів.

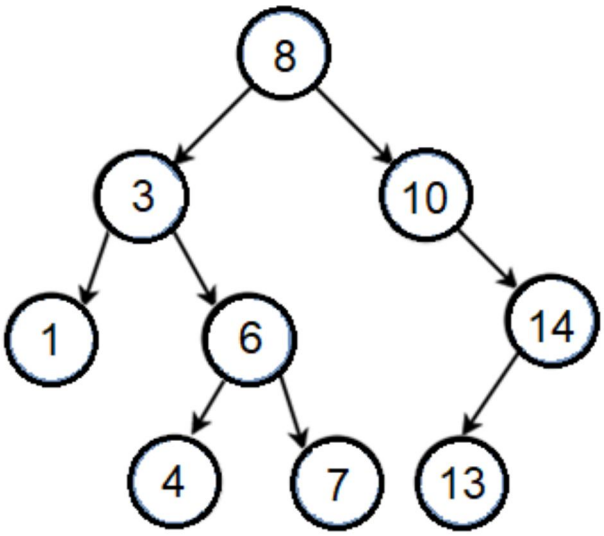


Рис. 9.10. Двійкове дерева пошуку

Наведені вище властивості двійкового дерева пошуку забезпечують упорядкування між ключами, щоб такі операції, як пошук, мінімум і максимум, можна було виконувати швидко. Якщо немає впорядкування, можливо, доведеться порівнювати кожен ключ, щоб шукати даний ключ.

Пошук ключа

Для пошуку значення, якби у нас був відсортований масив , ми могли б виконати двійковий пошук. Операція пошуку в дереві двійкового пошуку буде дуже схожою. Скажімо, ми хочемо шукати число, тому почнемо з кореня, а потім порівняємо значення для пошуку зі значенням кореня, якщо воно дорівнює, ми закінчимо пошук, якщо це менше, ми знаємо, що нам потрібно перейти до лівого піддерева, тому що в дереві двійкового пошуку всі елементи в лівому піддереві менші, а всі елементи в правому піддереві більші. Пошук елемента в дереві двійкового пошуку, по суті, це обхід, при якому на кожному кроці ми будемо рухатися вліво або вправо, а отже, на кожному кроці ми відкидаємо одне з піддерев. Якщо дерево збалансоване, ми почнемо з простору пошуку “n” вузлів і коли відкинемо одне з піддерев ми відкинемо вузли “n/2”, тому простір пошуку буде зменшено до “n/2”, а потім на наступному кроці буде зменшено простір пошуку до “n/4” і продовжимо скорочувати так, поки ми не знайдемо елемент або поки простір пошуку не буде зменшено лише до одного вузла.

Алгоритм пошуку у двійковому дереві пошуку (BST):

1. Почати з кореня.

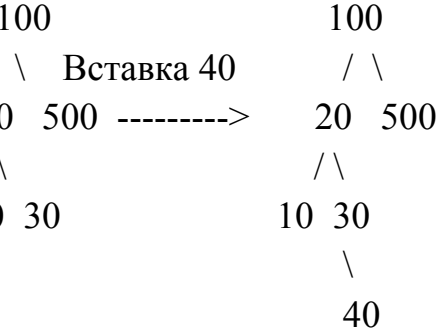
2. Порівняти пошуковий елемент з коренем, якщо менше кореня, то рекурсивно задати пошук для лівого піддерева, інакше – рекурсія для правого піддерева.

3. Якщо елемент для пошуку знайдено де-небудь, повернути true, інакше повернути false.

```
// C++ функція для пошуку заданого ключа в заданому BST
struct node* search(struct node* root, int key)
{

/* Основні випадки: root є нульовим або ключ присутній у
корені */
if (root == NULL || root->key == key)
    return root;

// Ключ більший за ключ root
  if (root->key < key)
    return search(root->right, key);

// Ключ менший за кореневий ключ
  return search(root->left, key);
}
```

Вставка ключа

Новий ключ завжди вставляється в листок. Ми починаємо пошук ключа з кореня, поки не потрапимо на листковий вузол. Як тільки листовий вузол знайдено, новий вузол додається як дочірній вузол листка.




```
    // Допоміжна функція для вставки нового вузла
    // із заданим ключем у BST
    struct node* insert(struct node* node, int key){
        // Якщо дерево порожнє, повертаємо новий вузол
        if (node == NULL)
            return newNode(key);
```

```
        // В іншому випадку повторювати вниз по дереву
        if (key < node->key)
            node->left = insert(node->left, key);
        else if (key > node->key)
            node->right = insert(node->right, key);

        // Повертає (незмінений) покажчик вузла
        return node;
    }
```

Коли ми видаляємо вузол, виникають три можливості.

1) ***Вузол, який потрібно видалити, це листок*** **–** просто приберати з дерева.

```
     50                    50
    /  \    delete(20)    /   \
   30  70   --------->  30     70
   / \   /\              \     /\
  20  40 60 80            40 60 80
```

2) ***Вузол, який потрібно видалити, має лише один вузол нащадок:*** скопіювати вузол - нащадок у вузол, який потрібно видалити, і видалити вузол - нащадок.

```
     50                    50
    /  \    delete(30)    / \
   30   70   --------->   40  70
   \   / \                  / \
   40 60  80               60  80
```

3) ***Вузол, який потрібно видалити, має два вузли - нащадки.*** Знайти у піддереві наступника вузла. Скопіювати вміст наступника до вузла, який слід видалити, та видалити наступника вузла. Зауважте, що попередник inorder також можна використовувати.

```
     50                    60
    /  \    delete(50)    / \
   40   70   --------->   40  70
       /\                     \
      60  80                   80
```

Важливо відзначити, що наступник вузла потрібен лише тоді, коли потрібний елемент - нащадок не порожній. У цьому конкретному випадку наступник у порядку можна отримати, якщо знайти мінімальне значення в правому піддереві - нащадку.

```
// Програма C++ для демонстрації
// операція видалення в двійковому файлі
// дерева пошуку
#include <bits/stdc++.h>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Допоміжна функція для створення нового вузла BST
{
  struct node* temp
    = (struct node*)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// Допоміжна функція
// симетричний обхід BST
void inorder(struct node* root)
{
  if (root != NULL) {
    inorder(root->left);
    cout << root->key;
    inorder(root->right);
  }
}

/* Допоміжна функція для вставки нового вузла із заданим
ключем в BST */
struct node* insert(struct node* node, int key)
```

```
{
/* Якщо дерево порожнє, повертаємо новий вузол */
  if (node == NULL)
    return newNode(key);

/* Інакше повторювати вниз по дереву */
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

/* повертаємо (незмінений) покажчик вузла */
  return node;
}

/* Враховуючи непорожнє двійкове дерево пошуку,
повертаємо вузол з мінімальним значенням ключа, знайденим
у цьому дереві. Зауважте, що не потрібно шукати все
дерево. */
struct node* minValueNode(struct node* node)
{
  struct node* current = node;

/* прокручуємо вниз, щоб знайти крайній лівий листок */
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

/* Ця функція має двійкове дерево пошуку та ключ
видаляє ключ і повертає новий корінь */
struct node* deleteNode(struct node* root, int key)
{
  // base case
  if (root == NULL)
    return root;

/* Якщо ключ, який потрібно видалити - менший від ключа
кореня, то він знаходиться в лівому піддереві */
```

```
  if (key < root->key)
    root->left = deleteNode(root->left, key);

/* Якщо ключ, який потрібно видалити більший за ключ
кореня, то він лежить у правому піддереві */
  else if (key > root->key)
    root->right = deleteNode(root->right, key);

// якщо ключ такий самий, як ключ кореня, то це вузол
// для видалення
  else {
// вузол не має дочірніх
    if (root->left==NULL and root->right==NULL)
      return NULL;

// вузол лише з одним дочірнім або без нього
    else if (root->left == NULL) {
      struct node* temp = root->right;
      free(root);
      return temp;
    }
    else if (root->right == NULL) {
      struct node* temp = root->left;
      free(root);
      return temp;
    }

/* вузол з двома дочірніми елементами: отримати
наступника вузла (найменший у правому піддереві)*/
    struct node* temp = minValueNode(root->right);

// Скопіюйте вміст наступника вузла в цей вузол
    root->key = temp->key;

// Видалити наступника порядку
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}
```

```
// Код основної програми
int main()
{
/* Давайте створимо наступний BST
50
/ \
30 70
/ \ / \
20 40 60 80 */
  struct node* root = NULL;
  root = insert(root, 50);
  root = insert(root, 30);
  root = insert(root, 20);
  root = insert(root, 40);
  root = insert(root, 70);
  root = insert(root, 60);
  root = insert(root, 80);

  cout << " Обхід заданого дерева в непорядковому порядку
\n";
  inorder(root);

  cout << "\ nВидалити 20\n";
  root = deleteNode(root, 20);
  cout << " Обхід модифікованого дерева в непорядковому
порядку \n";
  inorder(root);

  cout << "\ nВидалити 30\n";
  root = deleteNode(root, 30);
  cout << " Обхід модифікованого дерева в непорядковому
порядку \n";
  inorder(root);

  cout << "\ nВидалити 50\n";
  root = deleteNode(root, 50);
  cout << " Обхід модифікованого дерева в непорядковому
порядку \n";
  inorder(root);
```

```
  return 0;
}
```

Часова складність: Найгірша часова складність операції видалення – $O(h)$, де h – висота дерева двійкового пошуку. У гіршому випадку, можливо, нам доведеться подорожувати від кореня до найглибшого листового вузла. Висота перекошеного дерева може стати n, а складність операції видалення може стати O(n).

Даний алгоритм видалення можна удосконалити**.** Можна уникнути рекурсивних викликів, відстежуючи батьківський вузол спадкоємця, щоб просто видаляти наступника, зробивши дочірнє значення батьківського вузла NULL. Відомо, що наступником завжди буде листковий вузол.

8. AVL дерево

Означення. Деревом AVL називається самозбалансоване двійкове дерево пошуку (BST), у якому різниця між висотами лівого та правого піддерев не може бути більшою за одиницю для всіх вузлів.

Дерево AVL (Адельсона - Вельського та Ландіса) – це двійкове дерево пошуку з умовою балансу. Умова балансу має бути легко підтримуваною, і це гарантує, що глибина дерева дорівнює $O(\ log\ N)$. Найпростіша ідея – вимагати, щоб ліве і праве піддерева мали однакову висоту.

Як показує рис. 9.11, ця ідея не змушує дерево бути неглибоким.

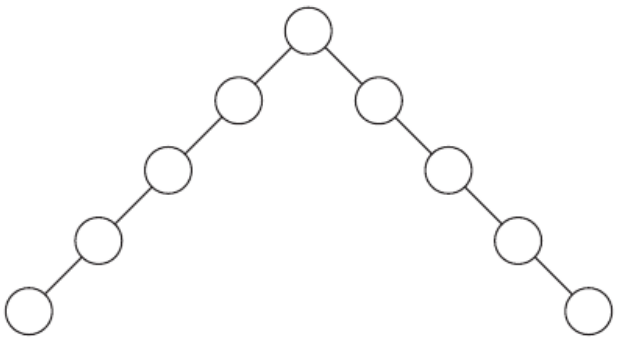


Рис. 9.11. Дерево AVL

Інша умова балансу буде полягати в тому, що кожен вузол повинен мати ліве і праве піддерева однакової висоти. Якщо висота порожнього піддерева визначена як −1 (як зазвичай), то лише ідеально збалансовані дерева з 2k − 1 вузлом задовольнятимуть цьому критерію. Таким чином, хоча це гарантує дерева невеликої глибини, умова рівноваги занадто жорстка, щоб бути корисною, і її потрібно замінити.

Приклад дерева, яке є деревом AVL, вказано на рис. 9.12.

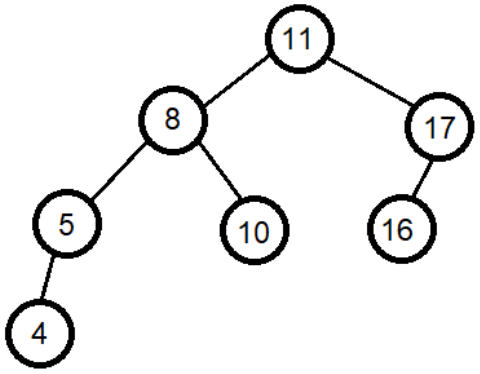


Рис. 9.12. Приклад AVL дерева

Наведене вище дерево є AVL, оскільки різниця між висотами лівого та правого піддерев для кожного вузла менша або дорівнює 1.

Приклад дерева, яке НЕ є деревом AVL

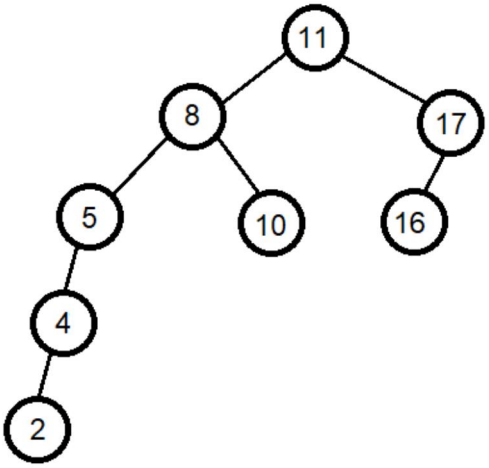


Рис. 9.13. Двійкове дерево, яке не є AVL деревом

Наведене вище дерево не є AVL, оскільки різниця між висотами лівого та правого піддерев для 8 та 11 більша за 1.

Більшість операцій BST (наприклад, пошук, максимум, мінімум, вставка, видалення тощо) займають O(h) часу, де h – висота BST. Швидкість виконання цих операцій може стати O(n) для скошеного двійкового дерева. Якщо ми переконаємося, що висота дерева залишається $O(\log \ n)$ після кожної вставки та видалення, то можемо гарантувати верхню межу $O(\log \ n)$ для всіх цих операцій. Висота дерева AVL завжди дорівнює $O(\log \ n)$, де n — кількість вузлів у дереві.

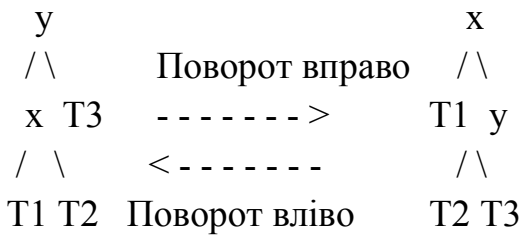
Вставлення елемента

Щоб переконатися, що дане дерево залишається AVL після кожної вставки, слід розширити стандартну операцію вставки BST для виконання деякого перебалансування. Нижче наведено дві основні операції, які можна виконати, щоб перебалансувати BST без порушення властивості BST (keys(left) < key(root) < keys(right)).

1) Поворот вліво

2) Поворот вправо

T1, T2 і T3 – це піддерева дерева, укорінені через *y* (з лівого боку) або *x* (з правого боку)



Ключі в обох вищевказаних деревах мають наступний порядок

$$keys(T1) < key(x) < keys(T2) < key(y) < keys(T3).$$

Тож властивість BST ніде не порушується.

Нехай новий вставлений вузол буде w. Розглянемо спосіб перетворення утвореного дерева у AVL дерево.

1) Виконати стандартну вставку у BST для w.

2) Починаючи з w, рухатися вгору і знати перший незбалансований вузол. Нехай z – перший незбалансований вузол, y – дочірній вузол z, який знаходиться на шляху від w до z, а x – онук z, який знаходиться на шляху від w до z.

3) Знову збалансувати дерево, виконавши відповідні обертання на піддереві, укоріненому з z. Можуть бути 4 можливі випадки, які потрібно обробити, оскільки x, y і z можна організувати 4 способами. Нижче наведено 4 можливі варіанти:

a) y є лівим дочірнім елементом z, а x є лівим дочірнім елементом y;

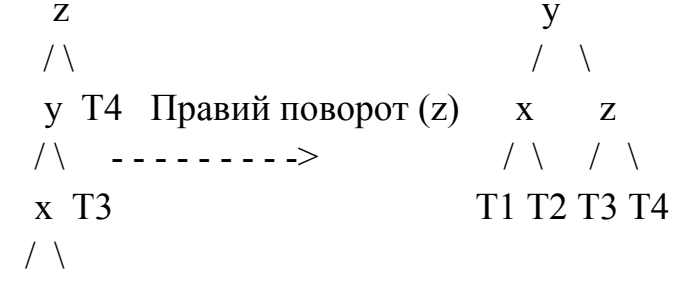
b) y є лівим дочірнім елементом z, а x правим дочірнім елементом y;

в) y – правий дочірній до z, а x – правий дочірній до y ;

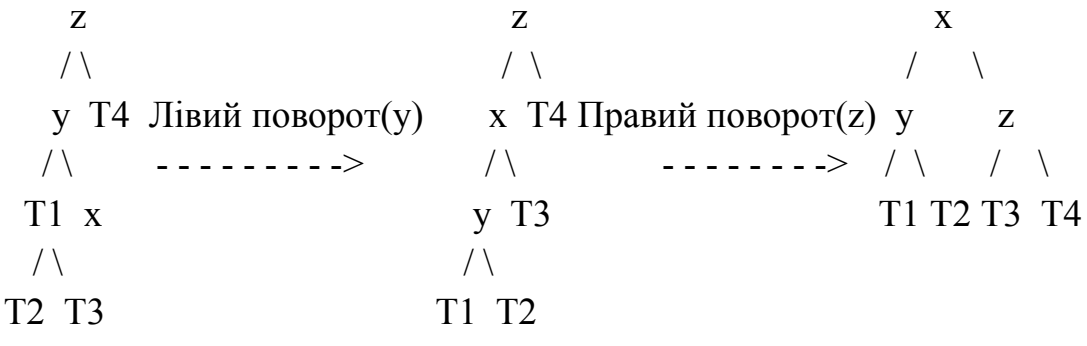
d) y є правим дочірнім елементом z, а x є лівим дочірнім елементом y.

В усіх випадках нам потрібно лише знову збалансувати піддерево, вкорінене з z, і повне дерево стане збалансованим, оскільки висота піддерева (після відповідних обертань), укоріненого з z, стає такою ж, якою була до вставки.

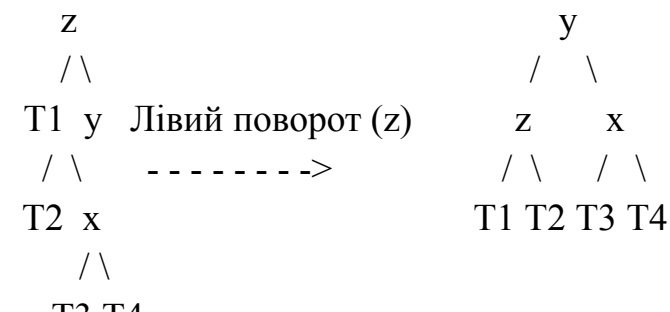
a) Лівий лівий вибір (T1, T2, T3 і T4 є піддеревами).



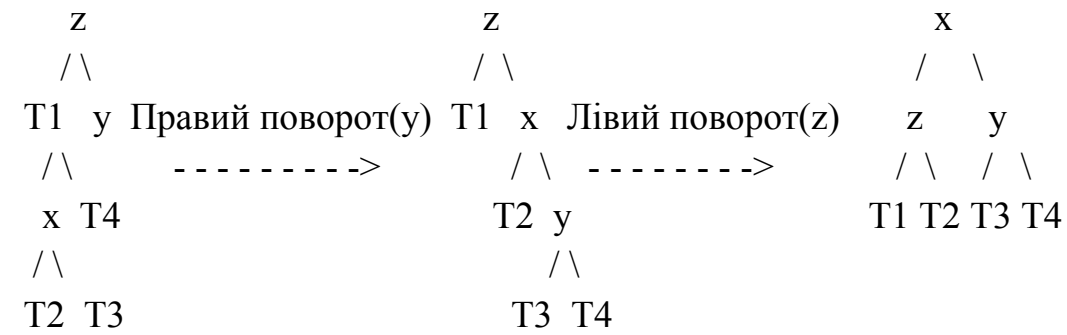
б) Лівий правий вибір.



в) Правий правий вибір



г) Правий лівий вибір



Реалізація

Нижче наведено реалізацію для вставлення вузла в дерева AVL. Ця реалізація використовує рекурсивну вставку BST для вставки нового вузла. У ній після вставки ми отримуємо покажчики на всіх предків по черзі знизу вгору. Тому не потрібний покажчик на батьків, щоб здійснювати рух вгору. Сам рекурсивний покажчик здійснює рух вгору і відвідує всіх предків для вставленого вузла.

Алгоритм для вставлення вузла в дерева AVL:

1) Виконати звичайну вставку BST.

2) Поточний вузол повинен бути одним із предків щойно вставленого вузла. Оновити висоту поточного вузла.

3) Отримати коефіцієнт балансу (висота лівого піддерева мінус висота правого піддерева) поточного вузла.

4) Якщо коефіцієнт балансу більше 1, то поточний вузол незбалансований, і виконавець знаходиться або в лівому лівому випадку, або лівому правому. Щоб

перевірити, лівий лівий випадок це чи ні, потрібно порівняти щойно вставлений ключ із ключем у лівому корені піддерева.

5) Якщо коефіцієнт балансу менший за -1, то поточний вузол незбалансований, і виконавець алгоритму знаходиться або в правому правому випадку, або в правому лівому. Щоб перевірити, чи це правий правий випадок чи ні, порівняти щойно вставлений ключ із ключем у правому корені піддерева.

```
// Програма C ++ для вставки вузла в дерево AVL
#include<bits/stdc++.h>
using namespace std;

// вузол AVL дерева
struct Node
{
  int key;
  int height;
  Node *left;
  Node *right;
 };

// Функція для отримання висоти дерева
int height(Node *N)
{
  if (N == NULL)
    return 0;
  return N->height;
}

/* Допоміжна функція, щоб отримати максимум двох цілих
чисел */
int max(int a, int b)
{
  return (a > b)? a : b;
}

/* Функція, яка виділяє новий вузол із заданим ключем
та NULL лівими та правими покажчиками. */
Node* newNode(int key)
{
```

```
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // новий допоміжний вузол
                      // додається зліва
    return(node);
  }

  /* Допоміжна функція правого повороту піддерева з
коренем y
   Див.діаграму вище. */
  Node *rightRotate(Node *y)
  {
    Node *x = y->left;
    Node *T2 = x->right;

    // Виконати обертання
    x->right = y;
    y->left = T2;

    // Оновлення висоти
    y->height = max(height(y->left),
            height(y->right)) + 1;
    x->height = max(height(x->left),
            height(x->right)) + 1;

    // Повернення нового кореня
    return x;
  }

  /* Допоміжна функція лівого повороту піддерева з
коренем x
     Див.діаграму вище. */
   Node *leftRotate(Node *x)
  {
    Node *y = x->right;
    Node *T2 = y->left;

    // Виконати обертання
```

```
    y->left = x;
    x->right = T2;

    // Оновлення висоти
    x->height = max(height(x->left),
            height(x->right)) + 1;
    y->height = max(height(y->left),
            height(y->right)) + 1;

    // Повернення нового кореня
    return y;
  }

  // отримати коефіцієнт балансу вузла N
  int getBalance(Node *N)
  {
    if (N == NULL)
      return 0;
    return height(N->left) - height(N->right);
  }

  /* рекурсивна функція для вставки ключа в піддерево,
з коренем вузлом і
   повертає новий корінь піддерева. */
  Node* insert(Node* node, int key)
  {
  /* 1. Виконати звичайну вставку BST */
    if (node == NULL)
      return(newNode(key));

    if (key < node->key)
      node->left = insert(node->left, key);
    else if (key > node->key)
      node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
      return node;

    /* 2. Оновлення висоти цього вузла предка */
    node->height = 1 + max(height(node->left),
             height(node->right));
```

```
      /* 3. Отримати коефіцієнт балансу цього вузла
предка, щоб перевірити, чи цей вузол став незбалансованим
*/
      int balance = getBalance(node);

      /* Якщо цей вузол стає незбалансованим, то є 4
випадки */
       // Лівий лівий випадок
      if (balance > 1 && key < node->left->key)
        return rightRotate(node);

      // Правий правий випадок
      if (balance < -1 && key > node->right->key)
        return leftRotate(node);

      // Лівий правий випадок
      if (balance > 1 && key > node->left->key)
      {
        node->left = leftRotate(node->left);
        return rightRotate(node);
      }

      // Правий лівий випадок
      if (balance < -1 && key < node->right->key)
      {
        node->right = rightRotate(node->right);
        return leftRotate(node);
      }

      /* Повернути (не змінений) вказівник вузла */
      return node;
    }

    /* Функція для друку прямого обходу дерева.
       Функція також друкує висоту кожного вузла */
    void preOrder(Node *root)
    {
      if(root != NULL)
      {
```

```
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
      }
    }

    // Основна програма
    int main()
    {
      Node *root = NULL;
      setlocale(LC_ALL, "ukr");

      /*  Побудова  дерева,  наведеного  наведеного  у
коментарі, що знаходиться нижче */
      root = insert(root, 10);
      root = insert(root, 20);
      root = insert(root, 30);
      root = insert(root, 40);
      root = insert(root, 50);
      root = insert(root, 25);

      /* Побудоване дерево AVL буде
            30
           /  \
         20   40
        / \    \
       10 25   50
      */
      cout << "Обхід дерева у прямому порядку з "
          "побудовою AVL дерева є: \n";
      preOrder(root);

      return 0;
    }
```

9. Алгоритм Прима

Розглянемо один алгоритм побудови каркасного дерева найменшої ваги.

Алгоритм Прима належить до жадібних алгоритмів. Він розпочинається з порожнього каркасного (остовного) дерева. Ідея полягає в тому, щоб

підтримувати два набори вершин. Перший набір містить вершини, які вже включені в MST (дерево найкоротшого шляху), інший набір – містить вершини, які ще не включені. На кожному кроці алгоритм розглядає всі ребра, які з'єднують два набори, і вибирає ребро мінімальної ваги з цих ребер, яке не утворює циклу в побудованому дереві. Після вибору кінцевої вершини він переміщує іншу кінцеву вершину ребра до набору, що містить MST.

Група ребер, яка сполучає дві множини вершин в графі, називається **розрізом** в теорії графів. Отже, на кожному кроці алгоритму Прима ми знаходимо розріз, вибираємо ребро мінімальної ваги з розрізу та включаємо цю вершину до множини MST.

Алгоритм Прима:

1) Створити набір mstSet, який містить вершини, що вже включені в MST.

2) Призначити значення ключа всім вершинам у вхідному графі. Ініціалізувати всі значення ключів як INFINITE. Призначте значення ключа 0 для першої вершини, щоб вона була обрана першою.

3) Якщо mstSet не включає всі вершини

….a) Вибирати вершину u, якої немає в mstSet і має мінімальне значення ключа.

….b) Включити u до mstSet.

….c) Оновити значення ключа всіх сусідніх вершин з вершиною u. Щоб оновити значення ключа, слід повторити обчислення для усіх сусідніх вершин. Для кожної сусідньої вершини v, якщо вага ребра {u, v} менша за попереднє значення ключа v, то слід оновити значення ключа як вагу {u, v}.

Ідея використання ключових значень полягає в тому, щоб вибрати ребро мінімальної ваги з розрізу.

Розглянемо такий приклад: дано зважений граф, який подано на рис. 9.14.

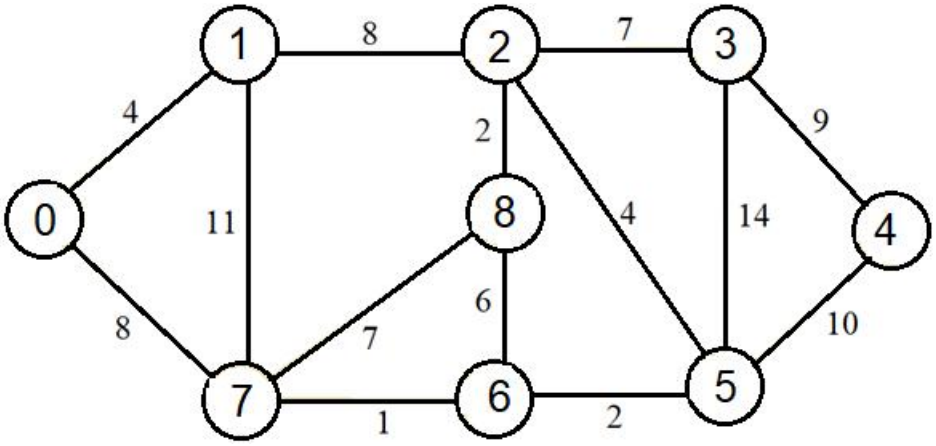
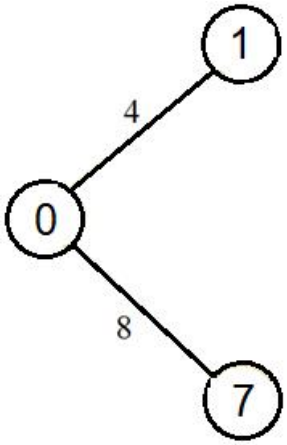


Рис. 9.14. Зважений граф

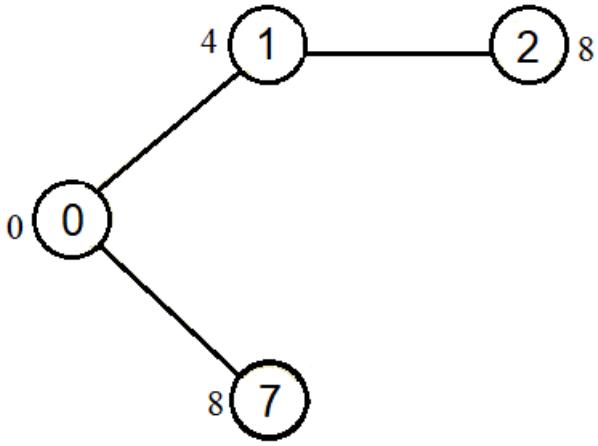
Побудувати дерево найменшої ваги, використовуючи алгоритм Прима.

Набір mstSet спочатку порожній ($mstSet = \emptyset$), а ключі, призначені вершинам: {0, INF, INF, INF, INF, INF, INF, INF, INF}, де INF означає



нескінченність. Тепер слід вибрати вершину з мінімальним значенням ключа. Вибрано вершину 0, слід включити її в mstSet. Отже, $mstSet = \{0\}$. Після включення до mstSet оновити значення ключів сусідніх вершин. Суміжні вершини 0 — це 1 і 7. Ключові значення 1 і 7 оновлюються як 4 і 8, тоді маємо множину ключів {0, 4, INF, INF, INF, INF, INF, 8, INF}.

Наступний крок показує вершини та їхні ключові значення, показано лише вершини зі скінченними значеннями ключа. Вершини, що входять до MST, показані зеленим кольором.



Слід вибрати вершину з мінімальним значенням ключа, яка ще не включена в MST (не в mstSET). Вершина 1 вибирається і додається до mstSet. Отже, $mstSet = \{0, 1\}$. Оновити ключові значення сусідніх вершин 1. Ключове значення вершини 2 стає 8, тоді маємо множину ключів {0, 4, 8, INF, INF, INF, INF, 8, INF}.

Слід вибрати вершину з мінімальним значенням ключа, яка ще не включена в MST (не в mstSET). Ми можемо вибрати вершину 7 або вершину 2, нехай вибрано вершину 7. Отже, $mstSet = \{0, 1, 7\}$. Оновити ключові значення сусідніх вершин числа 7. Ключове значення вершин 6 і 8 стають кінцевими (1 і 7 відповідно), тоді маємо множину ключів {0, 4, 8, INF, INF, INF, 1, 8, 7}.

Повторюємо вищезазначені кроки, поки mstSet не включатиме всі вершини даного графа. Нарешті отримуємо наступне дерево.

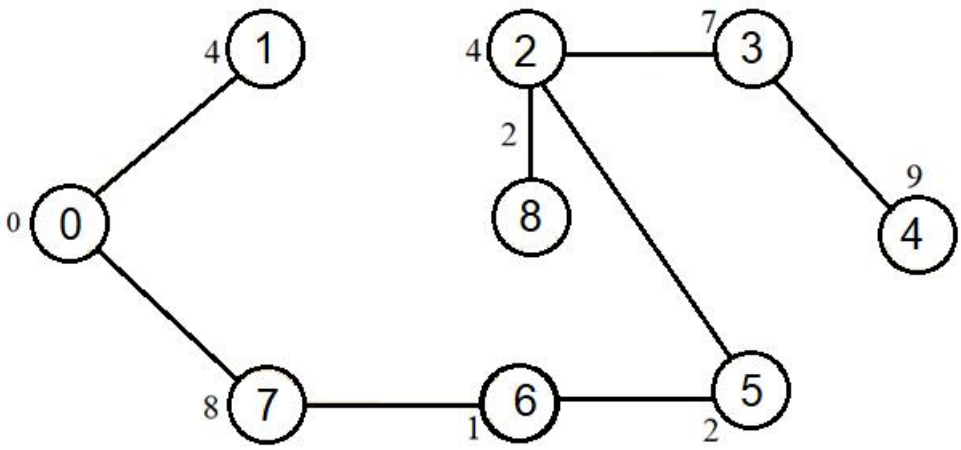


Рис. 9.16. Дерево мінімальної ваги

Ми використовуємо логічний масив mstSet[] для представлення набору вершин, включених до MST. Якщо значення mstSet[v] відповідає істині, то вершина v включається в MST, інакше ні. Масив key[] використовується для зберігання значень ключів усіх вершин. Інший масив parent[]для зберігання індексів батьківських вузлів у MST. Батьківський масив є вихідним масивом, який використовується для відображення створеного MST.

```
/* Програма C++ для знаходження мінімального дерева
алгоритмом Прима. Програма використовує для подання графа
матрицю суміжності */
#include <bits/stdc++.h>

using namespace std;

#define V 5  // Кількість вершин в графі

/* Функція для пошуку вершини з мінімальним значенням ключа
з набору вершин, які ще не включені в MST */
int minKey(int key[], bool mstSet[])
{
// Ініціалізація мінімального значення
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (mstSet[v] == false && key[v] < min)
      min = key[v], min_index = v;

  return min_index;
}

/* Допоміжна функція для друку створеного MST, збереженого
в parent[] */
void printMST(int parent[], int graph[V][V])
{
  cout<<"Ребро \tВага\n";
  for (int i = 1; i < V; i++)
    cout<< parent[i] << " - " << i<< " \t" <<
graph[i][parent[i]] << endl;
}
```

```
/* Функція для побудови та друку MST для графа, поданого
за допомогою представлення матриці суміжності */
void primMST(int graph[V][V])
{
// Масив для зберігання побудованого MST
  int parent[V];

/* Значення ключа, що використовується для вибору вершину
мінімальної ваги в розрізі */
  int key[V];

// Для представлення набору вершин, включених до MST
  bool mstSet[V];

// Ініціалізуємо всі ключі як INFINITE
  for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = false;

/* Завжди включати першу 1-у вершину в MST. Зробити ключ
0, щоб ця вершина була обрана як перша вершина. */
  key[0] = 0;
  parent[0] = -1; // Перший вузол завжди є коренем MST

  // MST міститеме вершину V
  for (int count = 0; count < V - 1; count++)
  {
/* Вибираємо мінімальну ключову вершину з набору вершин,
які ще не включені в MST */
    int u = minKey(key, mstSet);

// Додаємо вибрану вершину до набору MST
    mstSet[u] = true;

/* Оновлення значення ключа та батьківського індексу
сусідніх вершин вибраної вершини.
Розглянемо лише ті вершини, які ще не включені в MST */
    for (int v = 0; v < V; v++)

/* graph[u][v] не дорівнює нулю лише для сусідніх вершин m
mstSet[v] є хибним для вершин, які ще не включені в MST.
```

```
Оновити ключ, лише якщо графік [u][v] менший за ключ [v]
*/
      if (graph[u][v] && mstSet[v] == false && graph[u][v]
< key[v])
        parent[v] = u, key[v] = graph[u][v];
  }

// друкуємо побудований MST
  printMST(parent, graph);
}

// Основна програма
int main()
{
/* Створимо наступний графік
     2  3
  (0)--(1)--(2)
   |  / \  |
  6| 8/  \5 |7
   | /   \ |
  (3)-------(4)
      9   */
  int graph[V][V] = { { 0, 2, 0, 6, 0 },
            { 2, 0, 3, 8, 5 },
            { 0, 3, 0, 0, 7 },
            { 6, 8, 0, 0, 9 },
            { 0, 5, 7, 9, 0 } };

  setlocale(LC_ALL, "ukr");

  // Друкування розв'язку
  primMST(graph);

  return 0;
}
```

🔔 Контрольні запитання

1. Дайте рекурсивне означення дерева.
2. Сформулюйте родове означення дерева.
3. Які основні операції визначені на дереві?

4. Що називають листками дерева?
5. Дайте означення висоти дерева.
6. Які типи дерев розрізняють за величиною степеня дерева?
7. Яке дерево називають префіксним? Для чого його використовують?
8. Дайте означення бінарного дерева.
9. Спробуйте відтворити алгоритми симетричного, прямого та зворотного обходу дерев.
10. Яка ідея алгоритму побудови повного бінарного дерева?
11. На основі яких ідей побудовані алгоритми вставлення і видалення вузлів у двійковому дереві?
12. Що називають двійковим деревом пошуку?
13. Запишіть алгоритм пошуку у BST.
14. Яке дерево називають AVL деревом?
15. Яка ідея вставлення вузла у AVL дерево?

Тема 10. ГРАФИ

План лекції:

1. Означення
2. Подання графів у C++
3. Пошук у шир або BFS у графі
4. Пошук вглиб або DFS пошук
5. Топологічне сортування
6. Алгоритм Дейкстри
7. Алгоритм Флойда -Уоршелла

Джерела:

[1, §2.2], [3, Глава 6, 7, 8.4], [4, Глава 9], [5, Part IV], [6, Chapter 8], [10, Chapter 5, 6, 15], [11, Chapter 9].

1. Означення

Графом називається структура даних, яка складається з наступних двох компонентів:

1. Скінченної множини вершин, які також називають вузлами.

2. Скінченної множини впорядкованих або ж невпорядкованих пар виду (u, v) чи {u, v}, що називаються ***ребрами***.

Іншими словами, граф – це впорядкована пара $G = (V, E)$, яка складається з множини вершин V і множини ребер E. Кожне ребро є парою (v, w) або $\{v, w\}$, де $v, w \in V$. Ребра (v, w) іноді називають ***дугами***. Якщо пари впорядковані, то

граф називається ***орієнтованим***. Орієнтовані графи іноді називають ***орграфами***. **Вершина w суміжна з v** тоді й тільки тоді, коли $(v, w) \in E$ або $\{v, w\} \in E$. **У неорієнтованому графі** з ребром $\{v, w\}$, а отже, $\{w, v\}$, вершина w суміжна з вершиною v і навпаки. Іноді ребро має третій компонент, відомий як вага або вартість.

На рис. 10.1 показано приклад неорієнтованого графа з 5 вершинами.

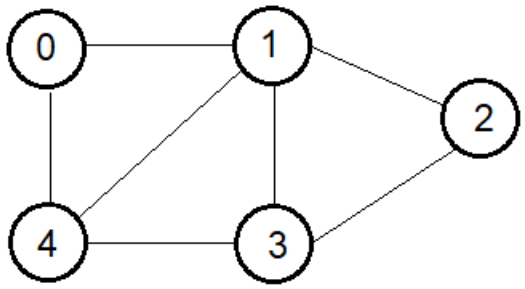


Рис 10.1. Неорієнтований граф

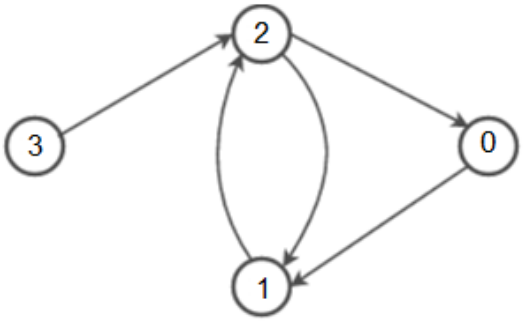


Рис. 10.2. Орієнтований граф

На рис. 10.2 наведено приклад орієнтованого графа з 4 вершинами.

Шлях у графі – це послідовність вершин $(w_1, w_2, w_3, \ldots, w_N)$, таких, що $(w_i, w_{i+1}) \in E$ або $\{w_i, w_{i+1}\} \in E$ для $1 \leq i < N$. Довжиною такого шляху є кількість ребер на шляху, що дорівнює $N - 1$. Розглянемо шлях від вершини до себе. Якщо цей шлях не містить ребер, то довжина шляху дорівнює 0. Це зручний спосіб визначити тривіальний випадок. Якщо граф містить ребро (v, v) або $\{v, v\}$, тоді його називають ***петлею***. Шлях $< v, v >$ називають **циклом**. Графи, які ми розглянемо, загалом не мають петель. **Простий шлях** – це шлях, у якому всі вершини різні, за винятком того, що перша і остання можуть бути однаковими.

Цикл у орієнтованому графі – це шлях довжиною щонайменше 1 такий, що $w_1 = w_N$. Цей цикл простий, якщо шлях простий. Для неорієнтованих графів вимагається, щоб ребра у циклі були різними. Логіка цих вимог полягає в тому, що шлях (u, v, u) в неорієнтованому графі не слід вважати циклом, оскільки $\{u, v\}$ і $\{v, u\}$ є одним і тим же ребром. У орієнтованому графі це різні ребра, тому має сенс називати це циклом. Орієнтований граф є ***ациклічним***, якщо він не має циклів. Орієнтований ациклічний граф іноді називають його абревіатурою DAG.

Неорієнтований граф є ***зв'язним***, якщо є шлях від кожної вершини до будь-якої іншої вершини. Орієнтований граф з цією властивістю називається ***сильно зв'язним***. Якщо орієнтований граф не є сильно зв'язним, але неорієнтовний граф зв'язний, то граф називається ***слабо зв'язним***.

Повний граф – це граф, у якому є ребро між кожною парою вершин.

Графи використовуються для представлення багатьох реальних додатків:

Мережі: можуть включати шляхи у стаціонарній або мобільній телефонній мережі або в комунікаційній мережі.

Карти Google: дозволяють зв'язати свою подорож від початку до кінця.

Соціальні мережі: друзі з'єднуються один з одним за допомогою ребра, де кожен користувач представляє вершину. Наприклад, у Facebook кожна людина представлена вершиною (або вузлом). Кожен вузол є структурою і містить таку інформацію, як ідентичність, ім'я, стать і місце розташування.

Система рекомендацій: дані взаємозв'язку між рекомендаціями користувача використовують графи для підключення.

2. Подання графів

Наступні два є найчастіше вживаними поданнями графів в програмуванні:

1. Матриця суміжності.
2. Список суміжності.

Існують також інші подання, такі як матриця інцидентності і список інцидентності. Вибір графічного зображення графа залежить від ситуації, зокрема залежить від типу виконуваних операцій і зручності використання.

Матриця суміжності

Матриця суміжності – це двовимірний масив розміру VxV, де V — кількість вершин у графі. Нехай двовимірний масив позначено adj[] [], тоді $adj[i]\,[j] = 1$ вказує, що від вершини i до вершини j є ребро. Матриця суміжності для неорієнтованого графа завжди симетрична відносно головної діагоналі. Тоді як матриця суміжності для орієнтованого графа не є симетричною. Матриця суміжності також використовується для представлення зважених графіків. Якщо adj[i] [j] = w, то від вершини і до вершини j є ребро з вагою w.

Матриця суміжності для прикладу неорієнтованого графа, показаного на рис. 10.1 запишеться:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

```
// Програма C++ для наведеного вище підходу
#include<iostream>

using namespace std;

int vertArr[20][20];
```

```
//матриця суміжності спочатку ініціалізується 0

// Функція виведення матриці суміжності графа
void displayMatrix(int v) {
  int i, j;
  for(i = 0; i < v; i++) {
   for(j = 0; j < v; j++) {
     cout << vertArr[i][j] << " ";
   }
   cout << endl;
  }
}

//функція для додавання ребра до матриці
void add_edge(int u, int v) {
  vertArr[u][v] = 1;
  vertArr[v][u] = 1;
}

main(int argc, char* argv[]) {
  int v = 5;  // у графі 5 вершин
  setlocale(LC_ALL, "ukr");

  add_edge(0, 1);
  add_edge(0, 4);
  add_edge(1, 2);
  add_edge(1, 3);
  add_edge(1, 4);
  add_edge(2, 3);
  add_edge(3, 4);

  cout << " Матриця суміжності неорієнтованого графа:
" << endl;
  displayMatrix(v);

  return 0;
}
```

Матриця суміжності для прикладу орієнтованого графа, показаного на рис. 10.2:

	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	1	0	0
3	0	0	1	0

```
// Програма C++ для наведеного вище підходу
#include<iostream>

using namespace std;

int vertArr[20][20];
//матриця суміжності спочатку ініціалізується 0

// Функція виведення матриці суміжності графа
void displayMatrix(int v) {
  int i, j;
  for(i = 0; i < v; i++) {
   for(j = 0; j < v; j++) {
     cout << vertArr[i][j] << " ";
   }
   cout << endl;
  }
}

//функція для додавання ребра до матриці
void add_edge(int u, int v) {
  vertArr[u][v] = 1;
}

main(int argc, char* argv[]) {
  int v = 4;  // в графі 4 вершини
  setlocale(LC_ALL, "ukr");

  add_edge(0, 1);
  add_edge(1, 2);
  add_edge(2, 0);
  add_edge(2, 1);
```

```
    add_edge(3, 2);

    cout << " Матриця суміжності орієнтованого графа: "
<< endl;
    displayMatrix(v);
    return 0;
}
```

Переваги: Таке подання графа легше реалізувати та використовувати. Видалення ребра займає O(1) часу. Запити на кшталт того, чи є ребро від вершини «u» до вершини «v», є ефективними і їх можна виконати за O(1) часу.

Недоліки: займає більше місця – $O(|V|^2)$. Навіть якщо граф розріджений, він займає той самий простір. Додавання вершини займає $O(|V|^2)$ часу.

Список суміжності

Використовується масив списків. Розмір масиву дорівнює кількості вершин. Нехай масив буде array[]. Масив array[i] представляє список вершин, суміжних з і-ою вершиною. Це подання також можна використовувати для представлення зваженого графіка. Вагу ребер можна представити у вигляді списків пар (див. рис. 10.3).

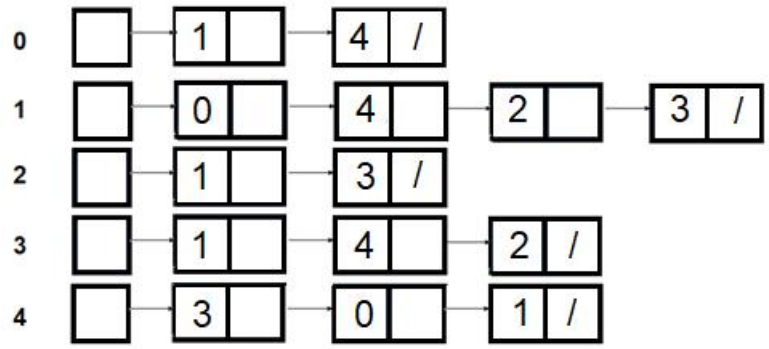


Рис.10.3. Список суміжності графа з рис. 10.1

Зауважте, що в наведеній нижче реалізації ми використовуємо динамічні масиви (векторні в C++) для представлення списків суміжності замість зв'язаного списку. Векторна реалізація має переваги зручності кешу.

```
// Просте відображення графа за допомогою STL
#include<bits/stdc++.h>
using namespace std;

// Допоміжна функція додавання ребра в
// неорієнтований граф.
```

```
void addEdge(vector<int> adj[], int u, int v)
{
  adj[u].push_back(v);
  adj[v].push_back(u);
}

// Допоміжна функція друкування списку суміжності
// що представляє граф
void printGraph(vector<int> adj[], int V)
{
  for (int v = 0; v < V; ++v)
  {
    cout << "\n Список суміжності вершини "
       << v << "\n голова ";
    for (auto x : adj[v])
      cout << "-> " << x;
    cout << "\n";
  }
}

// Основна програма
int main()
{
  int V = 5;
  setlocale(LC_CTYPE,"ukr");
  vector<int> adj[V];
  addEdge(adj, 0, 1);
  addEdge(adj, 0, 4);
  addEdge(adj, 1, 2);
  addEdge(adj, 1, 3);
  addEdge(adj, 1, 4);
  addEdge(adj, 2, 3);
  addEdge(adj, 3, 4);
  printGraph(adj, V);
  return 0;
}
```

Переваги: економить місце O(|V|+|E|) . У гіршому випадку в графі може бути C_V^2 кількість ребер, що використовує простір $O(V^2)$. Додати вершину простіше.

Недоліки: такі запити, як чи є ребро від вершини u до вершини v, неефективні і їх можна виконати за час порядку $O(V)$.

3. Пошук у шир або BFS у графі

Пошук вшир (обхід вшир, breadth-first search) - це один з основних алгоритмів на графах. У результаті такого пошуку знаходиться шлях найкоротшої довжини в незваженому графі, тобто. шлях, що містить найменшу кількість ребер.

Наступна схема алгоритму буде характерна і для пошуку вглиб.

1. Завести PLAN пошуку – контейнер даних, де зберігатимуться вершини в яких планується побувати. Спочатку він порожній.
2. Додати в PLAN пошуку вихідну вершину, з якої запропоновано почати.
3. Поки PLAN не порожній і мета пошуку не досягнута робити таке:
   a) GET: Витягнути з PLAN якусь вершину v.
   b) Відвідати вершину v. Якщо не просто потрібно обходити вершини, а щось шукати, то саме час обшукати вершину v на предмет досягнення мети пошуку.
   c) Позначити, що вершина v вже відвідана.
   d) PUT: Додати до PLAN усі сусідні з v вершини, які ще не були відвідані.
4. Вивести результат пошуку.

Найважливішим для реалізації цієї схеми є PLAN. Це контейнер даних, в якому потрібні дві функції GET – щоб щось із контейнера дістати і PUT – щоб у контейнер щось покласти. Звичайно, краще використовувати вже готові контейнери.

У BFS слід використовувати для зберігання плану ЧЕРГУ, алгоритм змінює стратегію та здійснює пошук вшир. Оскільки вершини будуть відвідуватись у тому порядку, в якому їх додавали, це дуже схоже на поширення хвилі з початкової точки.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
   // Читання вихідних даних
   int n, v;
   cin >> n >> v;
   int matrix [n] [n];
   for (int i = 0; i < n; i++)
```

```
      for (int j = 0; j < n; j++)
        cin >> matrix[i][j];

    queue <int> plan; // План відвідування у вигляді черги
    plan.push(--v); // нумеруємо з 0, а не з 1
    matrix [v] [v] = 1;
// відзначаємо, що ця вершина вже заносилася до плану
    int counter = 1; // Початкову вже порахували
    while (!plan.empty()) {
      v = plan.front();
// відвідуємо наступну за планом вершину
      plan.pop(); // видаляємо її із плану відвідування
      for (int u = 0; u < n; u++) {
// перебираємо сусідні з нею
        if (matrix[v][u] and !matrix[u][u]) {
// якщо нова, то
          plan.push(u); // Додаємо її в план
          matrix[u][u] = 1; // зазначаємо, що не нова
          counter++; // Вважаємо, скільки було вершин
        }
      }
    }
    cout << counter << endl;
}
```

Часова складність: Якщо граф задавати списком суміжності, то для не зваженого графа, можемо вирішити цю задачу за час $O(|V| + |E|)$. У даній програмі часова складність власне алгоритму BFS становить $O(n)$.

Просторова складність

Оскільки в пам’яті зберігаються всі розгорнуті вузли, то просторова складність алгоритму становить $O(|V| + |E|)$.

Задача. Задано орієнтовний граф. Знайдіть найкоротшу відстань від вершини x до усіх інших вершин графа.

Вхідні дані

У першому рядку містяться два натуральних числа n та x ($1 \le n \le 1000$, $1 \le x \le n$) - кількість вершин у графі та стартова вершина відповідно. Далі у n рядках по n чисел - матриця суміжності графа: в і-му рядку на j-му місці стоїть "1", якщо вершини і та j з’єднані ребром, і "0", якщо ребра між ними немає. На головній діагоналі матриці стоять нулі.

Вихідні дані

Виведіть через пропуск числа d[1], d[2], ..., d[n], де d[i] дорівнює -1, якщо шляхів між x та i немає, у протилежному випадку це мінімальна відстань між x та i.

Розв'язання

Це завдання розв'язуватимемо пошуком вшир. Сам граф зберігатимемо у двовимірному масиві graph[][], в масиві d[] зберігатимемо найкоротшу відстань між заданою вершиною та і -ою вершиною. У черзі `queue <int> plan` будемо зберігати вершини, що опрацьовуються. На самому початку там буде зберігатися тільки вихідна вершина, потім поки в черзі зберігаються якісь вершини дістаємо верхню і дивимося ребра, що виходять з неї, якщо раніше вершини не були задіяними, то поміщаємо в кінець черги. Черга спорожніє, коли буде переглянуто всі вершини, в які можна потрапити з вихідної.

```
#include <iostream>
#include <queue>
using namespace std;
int graph[1001][1001], d[1001];
queue <int> plan;

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  int n, x, v;
  cin >> n >> x;
  x--;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cin >> graph[i][j];
    }
  }
  for (int i = 0; i < n; i++) {
    d[i] = -1;
  }
// Алгоритм BFS
  plan.push(x);
  d[x] = 0;
  while (!plan.empty()) {
    v = plan.front();
```

```
    plan.pop();
    for (int i = 0; i < n; i++) {
      if (d[i] == -1 && graph[v][i]) {
        plan.push(i);
        d[i] = d[v] + 1;
      }
    }
  }
// Виведення найкоротших відстаней
  for(int i = 0; i < n; i ++) {
    cout << d[i] << " ";
  }
  return 0;
}
```

Використання алгоритму BFS:

1. Пошук компонент зв'язності у графі за O(n+m).

Для цього просто запускають обхід вшир від кожної вершини, за винятком вершин, що залишилися відвіданими (used = 1) після попередніх запусків. Таким чином, виконують звичайний запуск у ширину від кожної вершини, але не обнуляємо щоразу масив used[], за рахунок чого ми щоразу обходитимемо нову компоненту зв'язності, а сумарний час роботи алгоритму становитиме, як і раніше, O(n+ m) (такі кілька запусків обходу на графі без обнулення масиву used називаються серією обходів завширшки).

2. Знаходження вирішення будь-якої задачі (гри) з найменшою кількістю ходів, якщо кожен стан системи можна уявити вершиною графа, а переходи з одного стану до іншого – ребрами графа.

Класичний приклад – гра, де робот рухається полем, при цьому він може пересувати ящики, що знаходяться на цьому ж полі, і потрібно за найменшу кількість ходів пересунути ящики в необхідні позиції. Вирішується це обходом вшир за графом, де станом (вершиною) є набір координат: координати робота, та координати всіх коробок.

3. Знаходження найкоротшого шляху в 0-1-графі (тобто графі зваженому, але з вагами рівними тільки 0 або 1): досить трохи модифікувати пошук вшир: якщо поточне ребро нульової ваги, і відбувається поліпшення відстані до якоїсь вершини, то цю вершину додаємо не вкінець, а на початок черги.

4. Знаходження найкоротшого циклу в орієнтованому незваженому графі: проводимо пошук вшир з кожної вершини. Як тільки в процесі обходу ми

намагаємося піти з поточної вершини якимсь ребром у вже відвідану вершину, то це означає, що ми знайшли найкоротший цикл, і зупиняємо обхід вшир. Серед усіх таких знайдених циклів (по одному від кожного запуску обходу) вибираємо найкоротший.

5. Знайти всі ребра, що лежать на якомусь найкоротшому шляху між заданою парою вершин (a, b). Для цього треба запустити 2 пошуки вшир: з a, і b. Позначимо через d_a[] масив найкоротших відстаней, отриманий у результаті першого обходу, а через d_b[] – внаслідок другого обходу. Тепер для будь-якого ребра (u, v) легко перевірити, чи він лежить на якомусь найкоротшому шляху: критерієм буде умова d_a[u] + 1 + d_b[v] = d_a[b].

6. Знайти всі вершини, що лежать на якомусь найкоротшому шляху між заданою парою вершин (a, b). Для цього треба запустити 2 пошуки вшир: з a, і b. Позначимо через d_a[] масив найкоротших відстаней, отриманий у результаті першого обходу, а через d_b[] — внаслідок другого обходу. Тепер для будь-якої вершини легко перевірити, чи лежить він на якомусь найкоротшому шляху: критерієм буде умова d_a[v] + d_b[v] = d_a[b].

7. Знайти найкоротший парний шлях у графі (тобто шлях парної довжини). Для цього треба побудувати допоміжний граф, вершинами якого будуть стани (v, c), де v — номер поточної вершини, c = 0 … 1 – поточна парність. Будь-яке ребро (a, b) вихідного графа в цьому новому графі перетвориться на два ребра ((u, 0), (v, 1)) та ((u, 1), (v, 0)). Після цього, на графі треба обходом вшир знайти найкоротший шлях зі стартової вершини в кінцеву, з парністю, що дорівнює 0.

Практичне застосування BFS обходу:

Збір інформації-сміття: Техніка збору сміття, "Алгоритм Чейні" використовує обхід BFS для копіювання сміття.

Передавання інформації в мережах: пакет подорожує з одного вузла до іншого, використовуючи техніку BFS у мережі мовлення, щоб охопити усі вузли.

GPS - навігація: Ми можемо використовувати BFS в навігації GPS, щоб знайти всіх сусідів або сусідні вузли розташування.

Веб -сайти соціальних мереж: З огляду на людину "P", ми можемо знайти всіх людей на відстані "D" від P, використовуючи BFS до рівня D.

Однорангових мережах: Знову BFS можна використовувати в однорангових мережах, щоб знайти всі сусідні вузли.

Найкоротший шлях і мінімальне остовне дерево, що не зважене: Техніка BFS використовується для пошуку найкоротшого шляху, тобто шляху з найменшою кількістю ребер у незваженому графі. Аналогічно, ми також можемо знайти мінімальне дерево, що охоплює граф, використовуючи BFS на графі, що зважений.

4. Пошук вглиб або DFS пошук

Алгоритм пошуку вглиб (*Depth-first search, DFS*) – алгоритм для обходу дерева, структури подібної до дерева, або графу. Робота алгоритму починається з кореня дерева (або іншої обраної вершини в графі) і здійснюється обхід в максимально можливу глибину до переходу на наступну вершину.

Нехай G=(V, E) – простий зв'язний граф, усі вершини якого позначено попарно різними символами. У процесі пошуку вглиб вершинами графу G надають номери (DFS-номери) та певну структуру даних для збереження множин, яку називають стеком. Адже організація доступу до елементів стеку така, що ми насамперед відвідуватимемо ті вершини, які потрапили в план останніми. Зі стеку можна вилучити тільки той елемент, котрий було додано до нього останнім: стек працює за принципом «останній прийшов – перший вийшов» (LIFO). Інакше кажучи, додавання й вилучення елементів у стеку відбувається з одного кінця, який називається верхівкою стеку. DFS-номери вершини x позначають DFS(x). Отже, у раніше розглянутому алгоритмі PLAN буде стеком.

Єдине важливе пояснення. Щоб позначити, що вершина вже відвідувалася, використовуємо діагональ матриці суміжності графа. В умові спеціально підкреслили, що там завжди нулі, а отже можна поставити matrix[v][v] = 1, щоб позначити вершину як вже відвідану. Або ж слід завести інший масив **used**.

Алгоритм

Наведемо кроки алгоритму

1. Почати з довільної вершини v. Виконати DFS(v):=1. Включити цю вершину в стек.

2. Розглянути вершину у верхівці стеку: нехай це вершина x. Якщо всі ребра, інцидентні вершині x, позначено, то перейти до кроку 4, інакше – до кроку 3.

3. Нехай $\{x, y\}$ – непозначене ребро. Якщо DFS(y) уже визначено, то позначити ребро $\{x, y\}$ штриховою лінією та перейти до кроку 2. Якщо DFS(y) не визначено, то позначити ребро $\{x, y\}$ потовщеною суцільною лінією, визначити DFS(y) як черговий DFS-номер, включити цю вершину в стек і перейти до кроку 2.

4. Виключити вершину x зі стеку. Якщо стек порожній, то зупинитись.

```
#include <iostream>
#include <stack>

using namespace std;
```

```
int main() {
   // Читання вихідних даних
   int n, v;
   cin >> n >> v;
   int matrix [n] [n];
   for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
       cin >> matrix[i][j];

   stack <int> plan; // План відвідування як стека
   plan.push(--v); // ми нумеруємо з 0, а не з 1
   matrix [v] [v] = 1;
// відмічаємо, що ця вершина вже заносилася до плану
   int counter = 1; // Початкову вже порахували

   while (!plan.empty()) {
     v = plan.top();
// відвідуємо наступну за планом вершину
     plan.pop(); // видаляємо її із плану відвідування
     for (int u = 0; u < n; u++) {
// перебираємо сусідні з нею
       if (matrix[v][u] and !matrix[u][u]) {
                 // якщо нова, то
         plan.push(u); // Додаємо її в план
         matrix[u][u] = 1;
                 // відмічаємо, що не нова
         counter++;
                 // Рахуємо, скільки було вершин
       }
     }
   }

   cout << counter << endl;
   return 0;
}
```

Аналіз складності

Часова складність: $O(|V| + |E|)$, де V – кількість вершин, а E – кількість ребер у графа.

Просторова складність: $O(|V|)$, оскільки потрібен додатковий масив розміру V.

Даний алгоритм можна запрограмувати з оголошенням масиву used. Наведемо фрагмент коду з використанням структури вектора:

```
vector<vector<int>> g; // граф, список суміжності
int n; // кількість вершин
vector<bool> used (n); // відвідані вершини

void dfs(int v)
 {
  used [v] = true;
  for (int i = 0; i < g[v].size(); i++)
   {
     int to = g[v][i];
     if (!used [to])
   {
       dfs(to);
      }
    }
 }
```

Інколи необхідно знати кольори вершин (0 - не відвідана вершина; 1 - відвідана вершина, але досі у стеці; 2 - відвідана і більше немає у стеці) та "час" заходу та виходу з вершин. Ці допоміжні величини використовуються у застосуваннях алгоритму.

```
vector<vector<int>> g; // граф, список суміжності
int n; // кількість вершин

vector<int> color; // кольори вершин (0, 1, або 2)

vector<int> time_in, time_out;
      // "час" заходу та виходу з вершин
int dfs_timer = 0; // "таймер" для визначення "часу"

void dfs(int v)
 {
  time_in[v] = dfs_timer++;
  color[v] = 1;
```

```
  for (int i = 0; i < g[v].size(); i++) {
    int to = g[v][i];
    if (color[to] == 0) {
      dfs(to);
    }
  }
  color[v] = 2;
  time_out[v] = dfs_timer++;
}
```

Застосування DFS:

1) Виявлення циклу у графі

Граф має цикл, тоді і тільки тоді, коли вийшовши з певної вершини за допомогою DFS обходу, маркер активності знову повертається у дану вершину.

2) Пошук будь-якого шляху у графі.

Ми можемо перевизначити алгоритм DFS так, щоб знайти шлях між двома заданими вершинами u і z.

a) Викликати ***DFS (g, u)*** з *u* як початкової вершини.
b) Використати стек S, щоб відстежувати шлях між початковою вершиною та поточною вершиною.
c) як тільки зустрічається вершина *z* , то слід повернути шлях як вміст стека.

3) Топологічне сортування

4) Перевірка чи граф є двочастинним.

5) Пошук сильно зв’язаних компонентів графа.

6) Розв’язання лабіринтів тощо.

Задача. Розглянемо приклад використання DFS (задача 2383 https://www.eolymp.com/uk/problems/2383)

Є електрична мережа будинку з певною кількістю вузлів, деякі з яких сполучені дротами. Будь-яка пара вузлів може бути з’єднана максимум одним дротом. Жодний вузол не може бути підключений сам до себе. Кожний вузол мережі є або домашньою розеткою, або підключений до основної зовнішньої електромережі.

Ви хочете зробити схему безпечною та надлишковою, додавши стільки додаткових дротів, скільки це можливо. Єдина складність полягає в тому, що жодні два зовнішніх вузла електромережі на даний момент не з’єднані між собою (прямо чи опосередковано), і Вам слід зберегти цю властивість, інакше відбудеться замикання. Знайдіть максимальну кількість нових дротів, яку можна додати до схеми.

Вхідні дані

Складається з декількох тестів. Перший рядок кожного тесту містить кількість вузлів схеми **n** (**1** ≤ **n** ≤ **50**), пронумерованих цілими числами від **0** до **n - 1**. Наступні **n** рядків описують присутні дроту: **x**-ий символ рядка **y** дорівнює **1** (одиниця), якщо вершини **x** та **y** сполучені дротом, та **0** (нуль) інакше. Наступний рядок містить кількість вузлів, під'єднаних до основної електромережі, за якою йде список самих вузлів.

Вихідні дані

Для кожного тесту вивести в окремому рядку максимальну кількість нових дротів, яку можна додати в електричну мережу.

Розбір

Побудуємо матрицю суміжності графа m. Підрахуємо в змінній Edges загальну кількість наявних дротів (ребер графа): для цього слід підрахувати загальну кількість одиниць у масиві m і поділити його на 2 (кожне ребро буде підраховано двічі, оскільки граф є неорієнтованим).

Кожна вершина, підключена до зовнішньої мережі, разом із вершинами, що з нею (навіть безпосередньо), утворюють окрему компоненту зв'язності. За допомогою пошуку у глибину виділимо ці компоненти. Очевидно, що граф розпадеться на множину компонентів зв'язності, кожна з яких містить одну вершину, підключену до зовнішньої мережі. При цьому залишиться кілька вершин, що не входить в жодну з компонент. Нехай остання множина містить LeftVertex вершин. Між вершинами, що входять у різні компоненти зв'язності, дроти протягувати не можна, оскільки станеться замикання. Між двома вершинами, що входять в одну компоненту, можна додати провід, якщо він не був між ними спочатку. Між будь-якими двома з вершин, що залишилися LeftVertex, можна додати дріт. Для максимізації загального числа дротів множину LeftVertex вершин необхідно приєднати до тієї компоненти зв'язності, яка має найбільше вершин. У змінній MaxConnComp обчислюватимемо величину найбільшої компоненти зв'язності. За допомогою глобальної змінної Vertex функції dfs будемо підраховувати число вершин, що входять в поточну компоненту зв'язності. Якщо поточна компонента зв'язності містить Vertex вершин, максимум вона може містити Vertex * (Vertex – 1) / 2 ребер (повний граф). У змінній res обчислюємо найбільше ребер, що може міститися у всіх компонентах, породжених вершинами, підключеними до зовнішньої мережі. Приєднуємо дротами множину з LeftVertex вершин до найбільшої компоненти зв'язності: між LeftVertex вершинами можна провести LeftVertex * (LeftVertex – 1) / 2 ребер, між LeftVertex вершинами множини ребер, що залишилися, і MaxConnComp вершинами з найбільшої компоненти зв'язності. Залишається від

загального числа ребер res відняти кількість вже наявних Edges, щоб знайти максимальну кількість дротів, яку можна додати.

Електричну мережу представляємо у вигляді неорієнтованого графа і зберігаємо як матрицю суміжності в масиві m. Масив used використовується при пошуку в глиб для позначення вже пройдених вершин. Список вершин, що приєднані до основної електромережі, зберігаємо у масиві gridConnections.

```
#include <iostream>
#include <algorithm>
#include <cstring>
#include <vector>

using namespace std;

int i, j, n, num, val, Edges, Vertex, m[51][51], used[51];
vector<int> gridConnections;

/*Пошук у глибину використовується для виділення компонент зв'язності графа.
У глобальній змінній Vertex підраховуємо кількість вершин, що входять до поточної компоненти зв'язності. */
void dfs(int v)
{
      int i;
      used[v] = 1;
      Vertex++;
      for (i = 0; i < n; i++)
            if (m[v][i] && !used[i])
                  dfs(i);
}

/*Функція maxNewWires повертає максимальну кількість дротів, яку можна додати до електричної схеми. */
int maxNewWires()
{
      int i, res = 0, LeftVertex = n, MaxConnComp = 0;
      memset(used, 0, sizeof used);
```

```
/*Оскільки за умовою завдання в кожну компоненту зв'язності
може входити максимум один зовнішній вузол електромережі,
запускаємо пошук у глибину кожного такого вузла. */
      for (i = 0; i < gridConnections.size(); i++)
      {
            Vertex = 0;
            dfs(gridConnections[i]);

      /*У  компоненті  зв'язності,  що  містить  вершину
             gridConnections[i], входить Vertex вершин.
      Через Vertex вершин можна провести
      Vertex* (Vertex – 1)/2 дротів.
      MaxConnComp містить кількість вершин у
      найбільшій компоненті зв'язності. */

            res += Vertex * (Vertex - 1) / 2;
            LeftVertex -= Vertex;
            MaxConnComp = max(MaxConnComp, Vertex);
      }

  /*Залишилося LeftVertex вершин, які навіть побічно не
пов'язані із зовнішніми вузлами електромережі.
  З'єднуємо дротами всі можливі пари цих (LeftVertex *
(LeftVertex – 1) / 2 нових дротів), а також приєднуємо
кожен з них до всіх вершин з найбільшої компоненти
зв'язності (MaxConnComp * LeftVertex нових дротів). */
      res += LeftVertex * (LeftVertex - 1) / 2 + MaxConnComp
* LeftVertex;
      return res - Edges;
}

int main() {
      /* Читаємо вхідні дані. Підраховуємо кількість ребер
у графі Edges. */
      while (scanf("%d", &n) == 1)
      {
            for (Edges = i = 0; i < n; i++) {
                  for (j = 0; j < n; j++)
                  {
                        scanf("%1d", &m[i][j]);
```

```
                    if (m[i][j]) Edges++;
                }
            }
            scanf("%d", &num);
            gridConnections.clear();
            for (i = 0; i < num; i++)
   scanf("%d", &val), gridConnections.push_back(val);

    /* Оскільки граф неорієнтований, кожне ребро буде
пораховано двічі. */
            Edges /= 2;
            printf("%d\n", maxNewWires());
     }
     return 0;
}
```

5. Топологічне сортування

Топологічне сортування – це впорядкування вершин орієнтованого ациклічного графа, таке, що якщо є шлях від vi до vj, то vj з'являється після vi у впорядкованому масиві вершин.

Зрозуміло, що топологічне впорядкування неможливе, якщо граф має цикл, оскільки для двох вершин v і w циклу, v передує w і w передує v. Крім того, упорядкування не обов'язково є єдиним.

Простий алгоритм пошуку топологічного впорядкування полягає в тому, щоб спочатку знайти будь-яку вершину без вхідних ребер. Потім ми можемо надрукувати цю вершину та видалити її разом із ребрами з графа. Потім застосовуємо цю ж стратегію до решти графа.

Отже, топологічне сортування для орієнтованого ациклічного графа (DAG) – це лінійне впорядкування вершин таким чином, що для кожного орієнтованого ребра (u, v) вершина u стоїть перед v в упорядкуванні. Топологічне сортування для графа неможливе, якщо граф не є DAG.

Наприклад, топологічне сортування наступного графа – «5 4 2 3 1 0» (див. рис. 10.4). Для графа може бути більше одного топологічного сортування. Наприклад, інше топологічне сортування наступного графіка – «4 5 2 3 1 0». Першою вершиною в топологічному сортуванні завжди є вершина зі степенем 0 (вершина без вхідних ребер).

У DFS ми друкуємо вершину, а потім рекурсивно викликаємо DFS для її сусідніх вершин. При топологічному сортуванні нам потрібно надрукувати вершину перед її сусідніми вершинами. Наприклад, у наведеному графі вершина

«5» повинна бути надрукована перед вершиною «0», але на відміну від DFS, вершина «4» також повинна бути надрукована перед вершиною «0». Тому топологічне сортування відрізняється від DFS. Наприклад, DFS показаного графа – «5 2 3 1 0 4», але це не топологічне сортування.

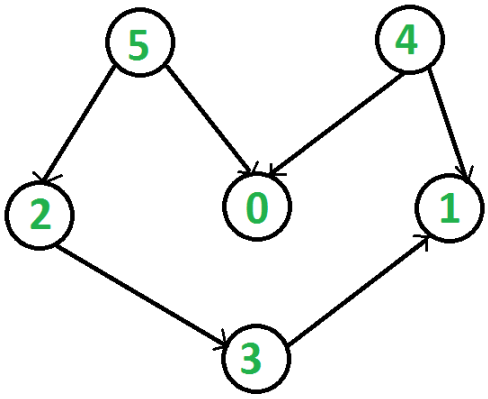
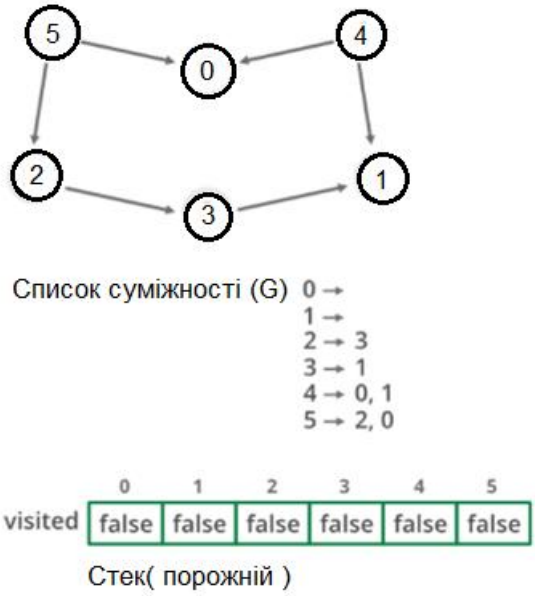


Рис. 10.4. Граф – приклад DAG

Алгоритм пошуку топологічного сортування:

Ми можемо змінити DFS, щоб знайти топологічне сортування графа. У DFS ми починаємо з вершини, спочатку друкуємо її, а потім рекурсивно викликаємо DFS для сусідніх вершин. У топологічному сортуванні ми використовуємо тимчасовий стек. Ми не друкуємо вершину відразу, ми спочатку рекурсивно викликаємо топологічне сортування для всіх сусідніх вершин, а потім поміщаємо її в стек. Нарешті, роздруковуємо вміст стека. Зауважте, що вершина переміщується в стек лише тоді, коли всі її суміжні вершини (і суміжні вершини суміжних вершин тощо) вже є в стеку.

Нижче наведено зображення вказаного вище підходу:

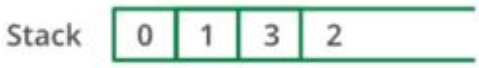


Крок 1: TopologicalSort(0), visited[0] = true

Список порожній. Відсутній рекурсивний виклик.

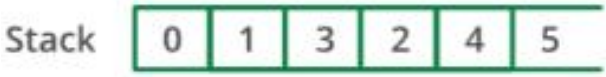
Крок 2: TopologicalSort(1), visited[1] = true
Список порожній. Відсутній рекурсивний виклик.

Крок 3: TopologicalSort(2), visited[2] = true
↓
TopologicalSort(3), visited[3] = true
‘ 1’ уже відвідали. Немає більше рекурсивних викликів.



Крок 4: TopologicalSort(4), visited[4] = true
↓
‘ 0’ і ‘ 1’ уже відвідали. Немає більше рекурсивних викликів.

Крок 5: TopologicalSort(5), visited[5] = true
↓
‘ 2’ і ‘ 0’ уже відвідали. Немає більше рекурсивних викликів.



Крок 6: Друкування всіх елементів стеку зверху до низу.
Покажемо реалізацію топологічного сортування на C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

//перевірка графа на циклічність
bool cyclic (int v, int &cycle_st, vector <int> *graph,
vector <int> &color ) {
   color[v] = 1;
   for (size_t i = 0; i < graph[v].size(); ++i) {
     int to = graph[v][i];
     if (color[to] == 0) {
//якщо у вершину не входили жодного разу
```

```
      if (cyclic (to, cycle_st, graph, color)) return
true;
    }
    else if (color[to] == 1) {
/*якщо у вказану вершину раніше входили, це означає, що
знайдено цикл */
      cycle_st = to;
// і змінюємо значення індикатора
      return true;
    }
  }
  color[v] = 2;
/*Вказуємо, що у вершину більше жодного разу входити не
будемо */
  return false;
}

// функція dfs пошуку
void dfs (int v, vector <int> *graph, vector<bool> &used,
vector <int> &answer) {
  used [v] = true;
//Вказуємо, що використовували цю вершину
  for (int i=0; i < graph[v].size(); i++) {
    int to = graph[v][i];
/*За списком проходимо по всіх вершинах, до яких можна
пройти від вершини v */
    if (!used[to])
/*і якщо вершину не розглядали, то застосовуємо алгоритм
пошуку в глибину для неї */
      dfs (to, graph, used, answer);
  }
  answer.push_back (v+1);
// заносимо вершину у вектор, що зберігає результат
}

// функція топологічного сортування
void  topological_sort(int  n,  vector  <int>  *graph,
vector<bool> &used, vector <int> &answer) {
  for (int i = 0; i < n; i++)
//вказуємо, що жодна вершина не була використана
```

```
      used[i] = false;
    for (int i = 0; i < n; i++)
      if (!used[i])
/*якщо під час попередніх операцій вершина не
використовувалася */
        dfs (i, graph, used, answer);
// щось викликаємо для неї алгоритм пошуку в глибину
    reverse (answer.begin(), answer.end());
}

int main() {
    int n, m; // Число вершин і ребер
    cin >> n >> m;
    int a, b;
    vector <int> graph[100001]; // граф
    vector <bool> used (n);
    vector<int> answer;
    vector<int>color (n,0);
/* масив, що зберігає кількість відвідувань для цієї
вершини. */
    int cycle_st = -1;
    for (int i = 0; i < m; i++) {
        cin >> a >> b;
        graph [a-1]. push_back (b-1);
      }
    for (int i = 0; i < n; i++) {
      if (cyclic (i, cycle_st, graph, color))
 // Перевірка графа на ациклічність
        break;
    }
    if (cycle_st! = -1) {
      cout << "-1";
    }
    else {
      topological_sort(n, graph, used, answer);
      for (int i = 0; i < answer.size(); i++)
      {
        cout << answer[i] << " ";
      }
      cout << endl;
```

```
    }
    return 0;
}
```

Аналіз складності

Часова складність: $O(|V| + |E|)$.

Наведений вище алгоритм – це просто DFS з додатковим стеком. Таким чином, складність за часом така ж, як і у DFS.

Просторова складність: $O(|V|)$.

Для стека потрібний додатковий простір.

6. Алгоритм Дейкстри

Задано граф і вихідну вершину, знайти найкоротші шляхи від вершини - джерела до всіх вершин у даному графі.

Алгоритм Дейкстри дуже схожий на алгоритм Прима для мінімального каркасного дерева. Як і MST Prim, ми генеруємо SPT (дерево найкоротшого шляху) із заданим джерелом як коренем. Ми підтримуємо два набори: один набір містить вершини, включені в дерево найкоротших шляхів; інший набір включає вершини, які ще не включені в дерево найкоротших шляхів. На кожному кроці алгоритму ми знаходимо вершину, яка знаходиться в наборі ще не включених вершин і має мінімальну відстань від вершини – джерела.

Нижче наведено докладні кроки, які використані в алгоритмі Дейкстри, щоб знайти найкоротший шлях від однієї вихідної вершини до всіх інших вершин у даному графі.

Алгоритм

1) Створити набір sptSet (набір найкоротших шляхів дерева), який відстежує вершини, включені в дерево найкоротшого шляху, тобто мінімальна відстань яких до вершини - джерела обчислюється та остаточно завершується. Спочатку цей набір порожній.

2) Призначити значення відстані всім вершинам у вхідному графі. Ініціалізувати всі значення відстані як INFINITE. Призначити значення відстані як 0 для вихідної вершини - джерела, щоб вона була обрана першою.

3) Якщо sptSet не включає всі вершини

….a) Вибрати вершину u, якої немає в sptSet і до якої є мінімальне значення відстані.

….b) Включити u до sptSet.

….c) Оновити значення відстані для всіх сусідніх вершин u. Щоб оновити значення відстані, повторити обчислення для усіх сусідніх вершин. Для кожної

сусідньої вершини v, якщо сума значення відстані u (від джерела) і ваги ребра u-v менша за значення відстані v, то оновити значення відстані v.

Давайте розберемося на такому прикладі (див. рис.10.5):

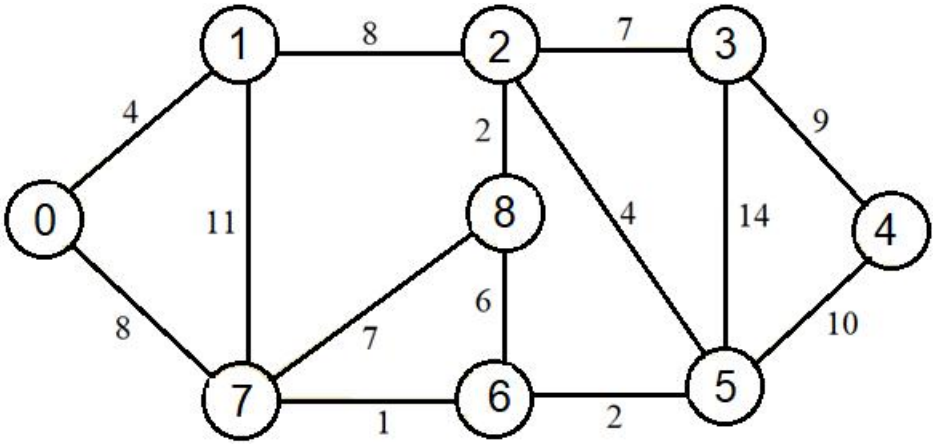
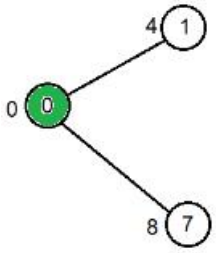
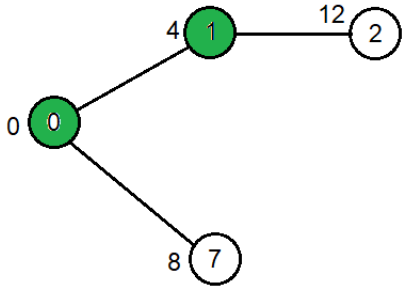


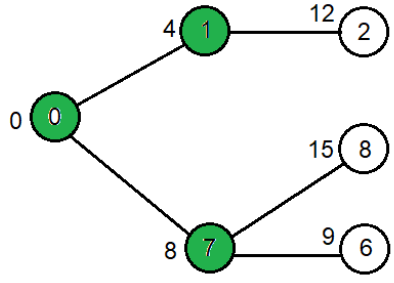
Рис. 10.5. Зважений граф

Набір sptSet спочатку порожній, а відстані, призначені вершинам, будуть {0, INF, INF, INF, INF, INF, INF, INF, INF}, де INF означає нескінченність. Тепер слід вибрати вершину з мінімальним значенням відстані. Вибрано вершину 0, включити її в sptSet. Отже, sptSet = {0}. Після включення 0 до sptSet оновити значення відстані його сусідніх вершин. Суміжні вершини 0 – це 1 і 7. Значення відстані 1 і 7 оновлюються як 4 і 8. Тоді множина відстаней буде дорівнювати {0, 4, INF, INF, INF, INF, INF, 8, INF}.У наступному підпункті показано вершини та їх значення відстані, показано лише вершини з кінцевими значеннями відстані. Вершини, що входять до SPT, показані зеленим кольором.



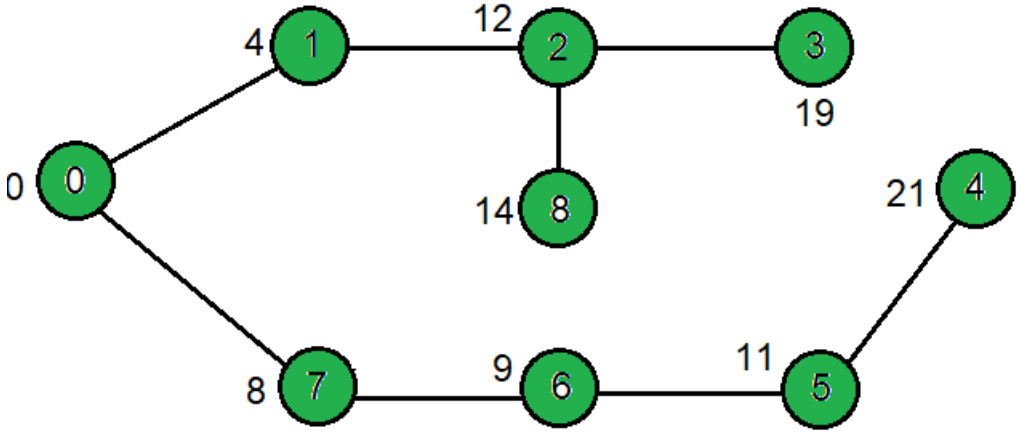


Виберіть вершину з мінімальним значенням відстані, яка ще не включена в SPT (не в sptSET). Вершина 1 вибирається та додається до sptSet. Отже, sptSet тепер стає {0, 1}. Оновіть значення відстані суміжних вершин 1. Значення відстані до вершини 2 стає 12. Тоді множина відстаней буде дорівнювати {0, 4, 12, INF, INF, INF, INF, 8, INF}.



Вибрати вершину з мінімальним значенням відстані, яка ще не включена в SPT (не в sptSET). Вибрано вершину 7. Отже, sptSet = {0, 1, 7}. Оновити значення відстані сусідніх вершин 7. Значення відстані до вершин 6 і 8 стає кінцевим (15 і 9 відповідно). Тоді множина відстаней буде дорівнювати {0, 4, 12, INF, INF, INF, 9, 8, 15}.

Ми повторюємо вищезазначені кроки, поки sptSet не включатиме всі вершини даного графа. Нарешті, ми отримуємо наступне дерево найкоротшого шляху (SPT).



Ми використовуємо булевий масив sptSet[] для представлення набору вершин, включених до SPT. Якщо значення sptSet[v] є істинним, то вершина v буде включена в SPT, інакше ні. Масив dist[] використовується для зберігання найкоротших значень відстані всіх вершин.

```
/* Програма C++ для алгоритму найкоротшого шляху Дейкстри з одною вершиною - джерелом.
Програма призначена для матричного представлення графа */
#include <iostream>
#include <climits>

using namespace std;

// Кількість вершин в графі
#define V 9

/* Функція для пошуку вершини з мінімальним значенням відстані з набору вершин, які ще не включені в дерево найкоротшого шляху */
int minDistance(int dist[], bool sptSet[])
{

  // Ініціалізація мінімальних значень
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min)
```

```
            min = dist[v], min_index = v;

      return min_index;
    }

    // Функція для друку створеного масиву відстаней
    void printSolution(int dist[])
    {
      cout <<"Vertex \t Distance from Source" << endl;
      for (int i = 0; i < V; i++)
        cout << i << " \t\t"<<dist[i]<< endl;
    }

    /* Функція, яка реалізує алгоритм найкоротшого шляху
Дейкстри з одним джерелом для графа, представленого за
допомогою матриці суміжності */
    void dijkstra(int graph[V][V], int src)
    {
      int dist[V]; /* Вихідний масив. dist[i] зберігатиме
найкоротшу відстань від src до i */

      bool sptSet[V]; /* sptSet[i] буде істинним, якщо
вершина i включена в дерево найкоротшого шляху або
найкоротша відстань від src до i завершена

       Ініціалізуємо всі відстані як INFINITE і stpSet[]
як false */
      for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

      /* Відстань від вершини-джерела до самої себе завжди
дорівнює 0 */
      dist[src] = 0;

      // Знайти найкоротший шлях до всіх вершин
      for (int count = 0; count < V - 1; count++) {
        /* Вибрати вершину з мінімальною відстанню з
набору ще не оброблених вершин. u завжди дорівнює src на
першій ітерації. */
        int u = minDistance(dist, sptSet);
```

```
        // Позначити виділену вершину як опрацьовану
        sptSet[u] = true;

        /* Оновлення значення dist сусідніх вершин
вибраної вершини. */
        for (int v = 0; v < V; v++)
          /* Оновлюємо dist[v] лише якщо не в sptSet, є
ребро від u до v, а загальна вага шляху від src до v через
u менша за поточне значення dist[v] */
          if (!sptSet[v] && graph[u][v] && dist[u] !=
INT_MAX
            && dist[u] + graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
      }

      // друкуємо побудований масив відстаней
      printSolution(dist);
    }

    // програма для перевірки алгоритму Дейкстри
    int main()
    {

      /* Нехай задамо граф, який обговорювався вище */
      int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

      dijkstra(graph, 0);

      return 0;
    }
```

Зауваження:

1) Код обчислює найкоротшу відстань, але не знаходить інформацію про шлях. Ми можемо створити батьківський масив, оновити його, коли відстань оновлюється (наприклад, реалізація prim) і використовувати його, щоб показати найкоротший шлях від джерела до різних вершин.

2) Код призначений для неорієнтованих графів, але цю ж функцію Дейкстри можна використовувати і для орієнтовних графів.

3) Код знаходить найкоротші відстані від вершини - джерела до всіх вершин. Якщо нас цікавить лише найкоротша відстань від джерела до однієї вершини, ми можемо розірвати цикл for, коли обрана вершина набуває мінімальну відстань.

4) Часова складність реалізації $O(|V|^2)$. Якщо вхідний граф представлений за допомогою списку суміжності, його можна зменшити до $O(E\ log\ |V|)$ за допомогою двійкової купи.

5) Алгоритм Дейкстри не працює для графів з циклами від'ємної ваги. Він може дати правильні результати для графа з від'ємними ребрами, але тоді потрібно дозволити, щоб вершину можна було відвідати кілька разів, і ця версія втратить свою швидкість за часом. Для графів з ребрами та циклами з від'ємною вагою можна використовувати алгоритм Беллмана–Форда.

7. Алгоритм Флойда-Уоршелла на C++

Дано орієнтований або неорієнтований зважений граф G з n вершинами. Потрібно знайти значення всіх величин d_ij – довжини найкоротшого шляху з вершини i у вершину j.

Передбачається, що граф не містить циклів від'ємної ваги (тоді відповіді між деякими парами вершин може просто не існувати – вона буде дуже маленькою).

Алгоритм був одночасно опублікований у статтях Роберта Флойда (Robert Floyd) та Стівена Уоршелла (Варшалла) (Stephen Warshall) у 1962 р., на честь яких цей алгоритм і називається в даний час. Втім, у 1959 р. Бернард Рой (Bernard Roy) опублікував практично ідентичний алгоритм, але його публікація залишилася непоміченою.

Ключова ідея алгоритму – розбиття процесу пошуку найкоротших шляхів на фази.

Перед k-ою фазою (k = 1 .. n) вважається, що в матриці відстаней d[][] збережені довжини таких найкоротших шляхів, які містять як внутрішні вершини тільки вершини з множини {1, 2, …, k -1} (вершини графа нумеруємо, починаючи з одиниці).

Іншими словами, перед k-ою фазою величина d[i][j] дорівнює довжині найкоротшого шляху з вершини i у вершину j, якщо цьому шляху дозволяється заходити тільки до вершин з номерами, меншими k (початок і кінець шляху не рахуються).

Легко переконатись, що щоб дана властивість виконалася для першої фази, достатньо в матрицю відстаней d[][] записати матрицю суміжності графа: d[i][j] = g[i][j] – вагу ребра з вершини i у вершину j . При цьому, якщо між якимись вершинами ребра немає, то слід записати величину "нескінченність". З вершини в себе завжди слід записувати величину 0, це критично для алгоритму.

Нехай тепер алгоритм знаходиться на k-ій фазі, і слід перерахувати матрицю d[][] таким чином, щоб вона відповідала вимогам вже для k+1 фази. Фіксуються якісь вершини i та j. Виникає два принципово різні випадки:

Найкоротший шлях з вершини i до вершини j, якому дозволено додатково проходити через вершини { 1, 2, …, k }, збігається з найкоротшим шляхом, якому дозволено проходити через вершини множини { 1, 2, ..., k- 1 }. У такому разі величина d[i][j] не зміниться під час переходу з k-ої на k+1-у фазу.

«Новий» найкоротший шлях став кращим за «старий» шлях. Це означає, що «новий» найкоротший шлях проходить через вершину k. Відразу зазначимо, що не буде втрачена спільність, розглядаючи далі лише прості шляхи.

Тоді зауважимо, що якщо буде розбито цей «новий» шлях вершиною k на дві половинки (одна, що йде i → k, а інша - k → j), то кожна з цих половинок вже не заходить у вершину k. Але тоді виходить, що довжина кожної з цих половинок була порахована ще на k-1-ій фазі або ще раніше, і нам достатньо взяти просто суму d[i][k] + d[k][j], вона і дасть довжину «нового» найкоротшого шляху.

Об’єднуючи ці два випадки, отримуємо, що на k-ій фазі потрібно перерахувати довжини найкоротших шляхів між усіма парами вершин i та j наступним чином:

new_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);

Отже, уся робота, яку потрібно зробити на k-ій фазі, – це перебрати всі пари вершин і перерахувати довжину найкоротшого шляху між ними. У результаті виконання n-ої фази в матриці відстаней d[i][j] буде записана довжина найкоротшого шляху між i та j, або нескінченність, якщо шляху між цими вершинами не існує.

Останнє зауваження: можна не створювати окрему матрицю new_d[][] для тимчасової матриці найкоротших шляхів на k-ій фазі: всі зміни можна робити відразу в матриці d[][]. Справді, якщо ми покращили (зменшили) якесь значення у матриці відстаней, ми могли погіршити тим самим довжину найкоротшого шляху якихось інших пар вершин, опрацьованих пізніше.

Реалізація

На вхід до програми подається граф, заданий у вигляді матриці суміжності – двовимірного масиву d[][] розміру $n \times n$, в якому кожен елемент задає довжину ребра між відповідними вершинами.

Потрібно, щоб виконувалось d[i][i] = 0 для будь-яких i.

```
for (int k=0; k<n; ++k)
    for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
            d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Передбачається, що якщо між двома якимись вершинами немає ребра, то в матриці суміжності було записано якесь велике число (досить велике, щоб воно було більшим за довжину будь-якого шляху в цьому графі). Тоді це ребро завжди невигідно брати, і алгоритм спрацює правильно.

Якщо не вжити спеціальних заходів, то за наявності в графі ребер від'ємної ваги, в результуючій матриці можуть з'явитися числа виду $\infty - 1$, $\infty - 2$, і т.д., які означають, що між відповідними вершинами взагалі немає шляху. Тому за наявності у графі від'ємних ребер алгоритм Флойда краще написати так, щоб він не виконував переходи з тих станів, в яких вже стоїть «немає шляху»:

```
for (int k=0; k<n; ++k)
   for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
         if (d[i][k] < INF && d[k][j] < INF)
            d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Відновлення самих шляхів

Легко підтримувати додаткову інформацію – про «предків», якими можна буде відновлювати найкоротший шлях між будь-якими двома заданими вершинами як послідовності вершин.

Для цього досить крім матриці відстаней d[][] підтримувати також матрицю предків p[][], яка для кожної пари вершин міститиме номер фази, де було отримано найкоротшу відстань з-поміж них. Зрозуміло, що цей номер фази є не чим іншим, як «середньою» вершиною шуканого найкоротшого шляху, і тепер нам просто треба знайти найкоротший шлях між вершинами i та p[i][j], а також між p[i][j] та j. Звідси виходить простий рекурсивний алгоритм відновлення найкоротшого шляху.

Аналіз алгоритму

Асимптотика алгоритму, очевидно, становить $O(n^3)$. Алгоритму потрібно $O(n^3)$ пам'яті для збереження матриць. Однак кількість матриць можна легко

скоротити до двох, щоразу переписуючи непотрібну матрицю або взагалі перейти до двомірної матриці.

Щодо часу роботи – три вкладені цикли від 1 до n – дають асимптотичний час $\Theta(n^3)$.

🔔 Контрольні запитання

1. Що називають графом у алгоритмах та структурах даних?
2. Які дві вершини графа є суміжними?
3. Що називають простим шляхом у графі?
4. Які найчастіше вживані подання графів в програмуванні? У чому їх суть?
5. Яка схема алгоритму пошука вшир?
6. Яку структуру даних використовують при пошуку вшир?
7. Яке практичне застосування BFS обходу?
8. У чому суть алгоритму пошуку вглиб?
9. Що таке топологічне сортування?
10. Запишіть алгоритм Дейкстри.
11. Яка ключова ідея алгоритму Флойда – Уоршелла?

Тема №11. АЛГОРИТМІЧНІ СТРАТЕГІЇ

План лекції:

1. Перебір варіантів
2. Перебір із поверненням (backtracking)
3. Розділяй та володарюй
4. Поняття «жадібного» алгоритму
5. Динамічне програмування

Джерела:

[3, Глава 10], [4, §5.2, 5.3], [5, Chapter 4, 15, 16], [10, Chapter 7,8], [11, Chapter 10]

1. Перебір варіантів

Є цілий клас завдань, розв’язання яких зводиться до перебору різних варіантів, серед яких вибирається такий, що задовольняє умову задачі.

Розглянемо організацію перебору на класичному прикладі: знайти дільники цілого числа N.

Ціле число K є дільником N, якщо залишок від ділення N на K дорівнює 0. Щоб знайти всі дільники, достатньо перебрати всі числа від 1 до N і перевірити, чи є вони дільниками. Цей алгоритм можна реалізувати за допомогою програми:

```
#include <iostream>

using namespace std;

int main()
{
  int n, i;
  setlocale(0, ".1251");
  cout << "Уведіть натуральне число N: " << endl;
  cin >> n;
  cout << "Дільниками числа " << n << " є: " << endl;
  for( i = 1; i <= n; i++)
  if (n % i == 0){
    cout << i << " ";
  }

  return 0;
}
```

З цього прикладу стає зрозумілою загальна структура розв'язання завдань за допомогою перебору варіантів. Для організації перебору використовують цикл, кожен крок виконання якого відповідає розгляду одного з варіантів. Усередині циклу стоїть перевірка, чи цей варіант підходить під умову завдання.

Розв'язуючи завдання методом перебору, завжди слід аналізувати, а чи не можна якимось чином скоротити кількість варіантів, що перебираються. Наприклад, у даному випадку зауважимо, що будь-яке число ділиться на себе і на 1. Тому ці варіанти можна виключити з перебору. Більш того, дільником, відмінним від числа N може бути число, яке не більше за $\sqrt{N}$. Продовжуючи аналіз щодо скорочення перебору, зауважимо, що якщо j є дільником, то частка від ділення j на i – також буде дільником. Таким чином, виводячи дільники парами, можна обмежитися перебором до $\sqrt{N}$:

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
```

```
{
  int n, i;
  setlocale(0, ".1251");
  cout << "Уведіть натуральне число N: " << endl;
  cin >> n;
  cout << "Дільниками числа " << n << " є: " << endl;
  cout << 1 << " " << n << " ";
  for( i = 2; i <= ceil(sqrt(n)); i++)
  if (n % i == 0){
    cout << i << " " << n / i << " ";
  }

  return 0;
}
```

Розглянемо ще один приклад, у якому буде продемонстровано не тільки перебір, а й те, що лічильником циклу for буде дійсна змінна.

Задача. Знайти мінімуми функції $f(x) = x^4 - x^2$ з точністю до 0.001 на відрізку від –5 до 5.

Для пошуку мінімуму перебиратимемо всі числа від –5 до 5 з кроком 0.001. Умовою знаходження мінімуму вважатимемо те, що значення функції f(x), яке менше, ніж у сусідніх точках. А саме: $f(x_{min} - 0.001) > f(x_{min}) > f(x_{min} + 0.001)$. Позначимо через змінну MinX ліву межу значень аргумента функції, MaxX – праву межу значень аргумента функції, а Step – точність обчислень та крок обчислень. Цей алгоритм можна реалізувати за допомогою програми:

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
  double Step, MinX, MaxX, x;
  setlocale(0, ".1251");
  cout << "Уведіть ліву межу значень аргумента MinX:
" << endl;
  cin >> MinX;
  cout << "Уведіть праву межу значень аргумента MaxX:
" << endl;
```

```
      cin >> MaxX;
      cout << "Уведіть точність значень аргумента Step: "
<< endl;
      cin >> Step;
      for ( x = MinX; x <= MaxX; x += Step){
        if ( ( pow(x, 4) - pow(x, 2) ) < (pow(x - Step,
4)- pow(x - Step, 2))&& (pow(x, 4)- pow(x, 2) < pow(x +
Step, 4)- pow(x + Step, 2)))
          cout << x << " ";
      }

      return 0;
    }
```

Крім перебору варіантів, тут на початку програми використано ще один важливий прийом, що зветься **параметризацією**. Такі характеристики, як точність і діапазон пошуку, були записані в окремі змінні, а потім замість чисел використовувалися імена цих змінних. Такий підхід дозволяє легко модифікувати програму, якщо параметри завдання зміняться, програма стає універсальнішою. Крім того, програма стає зрозумілішою, особливо якщо імена змінних обрані так, щоб відповідати змісту параметра, що зберігається в них.

Зверніть увагу, що значення аргументу функції обчислюються за формулою:

$x = x + Step$.

Отже, **вичерпний пошук** (англ. *exhaustive search*) або **метод грубої сили** (англ. *brute-force*) або ж **повний перебір** – загальний метод розв'язування задач та алгоритмічна парадигма, суть якого в систематичному перенумеровані всіх можливих кандидатів на розв'язок і перевірці кандидата на виконання умов.

Хоча повний перебір легко реалізувати і він завжди буде знаходити відповідь (за умови її існування), час виконання при цьому буде пропорційним кількості кандидатів на розв'язок. А в більшості практичних задач це призводить до дуже швидкого зростання кількості кандидатів, коли зростає розмір задачі (це називається комбінаторним вибухом). Тому пошук повним перебором зазвичай використовується, тільки для задач невеликого розміру. Наприклад, евристичний алгоритм може скоротити кількість кандидатів на розв'язок до прийнятного розміру. Також метод повного перебору можна використовувати коли простота реалізації важливіша за швидкість.

При розробці таких алгоритмів зазвичай необхідно вирішити два завдання.

1. Встановити порядок на елементах, що підлягають перерахунку.

2. Навчитися переходити від довільного елемента до елемента, що знаходиться безпосередньо за ним.

Альтернативний підхід до вирішення подібних завдань полягає у використанні рекурсії - для цього потрібно показати, що перебір N елементів включає як підзавдання перебір меншої кількості елементів (наприклад, N-1), і т.д. рекурсивно.

2. Перебір із поверненням (backtracking)

Перебір із поверненням (backtracking) – це загальний метод упорядкованого перебору. Перебір із поверненням особливо зручний для розв'язання завдань, що вимагають перевірки потенційно великої, але скінченної кількості рішень.

Термін backtracking був введений в 1950 р. американським математиком Дерріком Генрі Лемером.

У загальному випадку ми вважаємо, що рішення можна записати як вектор (масив змінної довжини) $B = (b_1, b_2, \ldots, b_n)$, що задовольняє заданим умовам та обмеженням, або як множина таких векторів. При цьому в одних завданнях розмірність рішення (число n) може бути відома наперед, а в інших – наперед не визначена.

Метод перебору з поверненням заснований на тому, що при пошуку рішення багаторазово робиться спроба розширити поточне часткове рішення, тобто його продовжити. Якщо розширення неможливо, відбувається повернення до попереднього більш короткого часткового рішення, і робиться спроба його розширити іншим можливим способом.

Як вихідне часткове рішення ми вибираємо порожній вектор, який позначатимемо ( ). На основі заданих обмежень з'ясовуємо які елементи є кандидатами в b_1; позначимо цю підмножину через S_1. Як b_1 вибираємо перший елемент S_1 і отримуємо часткове рішення (b_1). У загальному випадку, за частковим рішенням $(b_1, b_2, \ldots, b_{k-1})$ на основі обмежень завдання будується S_k, з якого обираються кандидати для розширення часткового рішення $(b_1, b_2, \ldots, b_{k-1})$ до $(b_1, b_2, \ldots, b_k)$. Якщо $S_k = \{\}$, тобто часткове рішення $(b_1, b_2, \ldots, b_{k-1})$ не може бути розширено, ми повертаємось та вибираємо новий елемент b_{k-1}. Якщо новий елемент b_{k-1} вибрати не можна, ми повертаємось ще на один крок назад і вибираємо новий елемент b_{k-2} і т.д.

Реалізація перебору з поверненням зазвичай призводить до експоненційних алгоритмів, що пов'язано з експоненціальним числом досліджуваних вершин у дереві пошуку з поверненням. Процес вирішення завдання перебором із поверненням поділяється на окремі підзадачі, які зручно описувати з використанням рекурсії. При використанні рекурсії відпадає

необхідність безпосередньо організовувати повернення та відстежувати правильність їх виконання. Вони стають інтегрованою частиною механізму виконання рекурсивних викликів. Нижче наведено рекурсивну реалізацію перебору з поверненням на псевдокоді:

procedure backtracking(vector, i)
if vector не має сенсу розширювати
then exit
if vector є рішенням
then записати vector
побудувати Si
for $s \in S_i$ **do**
backtracking(vector+s, i+1)
Для запуску процедури необхідно викликати її:
backtracking((), 1)

Усі завдання, для вирішення яких можна застосувати перебір із поверненнями, можна умовно розділити на кілька великих груп:

- ✓ Пошук одного рішення
- ✓ Пошук всіх рішень
- ✓ Пошук оптимального рішення

Розглянемо алгоритм перебору із поверненням (backtracking) з прикладу завдання про проходження лабіринту (пошук всіх рішень).

Дано лабіринт, опинившись усередині якого необхідно знайти вихід назовні. Переміщатися можна лише у горизонтальному та вертикальному напрямках. На малюнку показані всі варіанти шляхів виходу із центральної точки лабіринту.

Для отримання програми розв'язання цього завдання необхідно вирішити дві проблеми:

• як організувати дані;

• як побудувати алгоритм.

Інформацію про форму лабіринту зберігатимемо в квадратній матриці символьного типу розміром N x N, де N – непарне число (щоб була центральна точка). На профіль лабіринту накладається сітка так, щоб у кожній її комірці знаходилася або стіна, або прохід. Матриця відображає заповнення сітки:

елементи, що відповідають проходу, дорівнюють пробілу, а стіні – будь-якому символу (наприклад, літері m). Шлях руху по лабіринту відзначатиметься символами +.

m	m	m	m		m	m	m
m	m				m	m	m
m	m		m	m	m		m
m							
m	m	m		m	m	m	m
m	m	m	m	m	m	m	m

Вхідні дані – профіль лабіринту (вхідна матриця без хрестиків), а результат – всі можливі траєкторії виходу з центральної точки лабіринту (для кожного шляху виводиться матриця з траєкторією, позначеною хрестиками). Алгоритм перебору із поверненням ще називають **методом проб**.

Суть методу:

1. З кожної чергової точки траєкторії проглядаються можливі напрямки руху в одній і тій же послідовності (наприклад, вгору-вниз-вправо-вліво);

- крок проводиться у першу ж виявлену вільну сусідню клітинку;
- клітинка, в яку зроблено крок, позначається хрестиком.

2. Якщо з чергової клітини далі шляху немає (тупик), слід повернутися на крок назад і проглянути ще не випробувані шляхи руху з цієї точки; при поверненні назад покинута клітина позначається пробілом.

3. Якщо чергова клітина, в яку зроблено крок, опинилася на краю лабіринту (на виході), то на друк виводиться знайдений шлях.

Процедура GO намагається зробити крок у клітину з координатами x, y. Якщо ця клітина виявляється на виході з лабіринту, пройдений шлях виводиться на друк. Якщо ні, то відповідно до встановленої вище послідовності робиться крок у сусідню клітину. Якщо клітина тупикова, виконується крок назад. Зі сказаного вище слід зазначити, що процедура носить рекурсивний характер. Запишемо спочатку загальну схему процедури без деталізації:

```
void go(...)
{
  if клітина вільна
{
 крок у клітину
 if край лабіринту
друкуємо шлях
 else
{
Спроби зробити крок у сусідні клітини у певній послідовності
```

}

повернення назад на 1 крок

}

}

Для виведення знайдених траєкторій складається процедура printlab.

```
#include<iostream>
#include<string>
#include<iomanip>
using namespace std;
int n,m;
void printlab(char**&lab)
{
  int i,j;
  cout<<"--------------------------------"<<' \n';
  for(i=0;i<n;i++){
    for(j=0;j<m;j++)
      cout<<lab[i][j];cout<<' \n';
   }
}
void go(char**&lab, int x,int y)
{
  if(lab[x][y]==' ' ) //клітина вільна
    {
      lab[x][y]=' +'; //крок у клітину
      if((x==0)||(x==n-1)||(y==0)||(y==m-1)) //край
        printlab(lab);
      else {
          go(lab,x+1,y);
          go(lab,x-1,y);
          go(lab,x,y+1);
          go(lab,x,y-1);
          }
       lab[x][y]=' ' ; //Повернення назад
    }
}
int main()
{
  int j,i;
  string s;
```

```
  cin>>n>>m;
  char**lab;
  lab = new char * [n];
  for(i=0;i<n;i++)
    lab[i] = new char[m];
  cin.ignore(); /*необхідно для того, щоб пропустити в
кінці попереднього рядка символ перехід на новий рядок,
щоб він не зчитувався командою getline */
  for(i=0;i<n;i++)
  {
    getline(cin,s);
    for(j=0;j<m;j++)
      lab[i][j]=s[j];
  }
  go(lab,n/2,m/2);

  return 0;
}
```

Для лабіринта вказаного в умові задачі, буде отримано два розв’язки:

```
6 8
mmmm mmm
mm    mmm
mm mmm m
m
mmm mmmm
mmmmmmmm
--------------------------------
mmmm mmm
mm    mmm
mm mmm m
m    ++++
mmm mmmm
mmmmmmmm
--------------------------------
mmmm+mmm
mm+++mmm
mm+mmm m
m +++
mmm mmmm
mmmmmmmm
```

3. Розділяй та володарюй

Парадигма **«Розділяй та володарюй»** (англ. *divide and conquer*) – розробка алгоритмів, що полягає в рекурсивному розбитті розв'язуваної задачі на дві або більше підзадачі того ж типу, але меншого розміру, і комбінуванні їх розв'язків для отримання відповіді до вихідного завдання. Розбиття виконуються доти, поки всі підзавдання не стануть елементарними.

За [5] парадигма, яка лежить в основі «Розділяй і володарюй» на кожному рівні містить три кроки:

1. **Розділення** задачі на кілька підзадач, які є меншими екземплярами тієї ж задачі.

2. **Володарювати** над підзадачами, шляхом їх рекурсивного розв'язання. Якщо розміри підзадачі достатньо малі, то вони можуть розв'язуватися як базові в алгоритмі.

3. **Комбінування** підзадач в розв'язанні задачі.

Прикладом використання даного метода є сортування масиву шляхом злиття, яке ми розглядали раніше. При цьому відбувалися кроки:

1. Розділення n - елементного масиву на дві n/2 елементні послідовності.

2. Володарювання. Рекурсивно сортуємо дані дві підпослідовності з використанням алгоритму злиття.

3. Комбінування. З'єднуємо дві відсортовані підпослідовності для отримання відсортованого масиву.

Код алгоритму MergeSort на C++, як і його аналіз був даний у темі 8.

4. Поняття «жадібного» алгоритму

Жадібні алгоритми – це такі алгоритми, які прагнуть зробити оптимальний вибір у кожний момент часу.

Означення. Жадібний алгоритм (greedy algorithm) – метод рішення оптимізаційних задач, заснований на тому, що процес прийняття рішення можна розбити на елементарні кроки, на кожному з яких приймається окреме оптимальне рішення.

Цей алгоритм на кожному кроці робить локально оптимальний вибір, сподіваючись у результаті отримати глобально оптимальне рішення.

Алгоритми називаються жадібними, коли вони використовують жадібну властивість: «У кожний момент часу, що є найкращим вибором?»

Дані алгоритми цікавить лише найкраще рішення на даний момент. Але загальне оптимальне рішення може відрізнятиметься від рішення, яке вибирає алгоритм на кожному кроці своєї роботи. У жадібних алгоритмах ніколи не

озираються назад для того, щоб переосмислити що зроблено, чи потрібна глобальна оптимізація.

До оптимізаційної задачі можна застосувати **принцип жадібного вибору**, *послідовність локально оптимальних виборів дає глобально оптимальний розв'язок*. Доведення оптимальності здійснюється за такою схемою: спочатку доводиться, що жадібний вибір на першому етапі не унеможливлює шляху до оптимального розв'язку: для будь-якого розв'язку є інший, узгоджений із жадібним і не гірший від першого. Далі доводиться, що підзадача, яка виникла після жадібного вибору на першому етапі, аналогічна початковій, і міркування закінчується за індукцією.

Існує евристичне правило для розуміння застосовності жадібного підходу. Якщо обидві властивості наведені нижче справджуються, жадібний алгоритм може бути застосований до розв'язання задачі.

1. **Принцип жадібного вибору**.

2. **Оптимальна підструктура**. Задача має оптимальну підструктуру, якщо оптимальний розв'язок цілої задачі містить оптимальний розв'язок для будь якої підзадачі. Іншими словами, після завершення певного кроку алгоритму залишається розв'язати задачу, для якої жадібний підхід також працює.

Жадібні алгоритми набагато швидші, ніж алгоритми «розділяй і володарюй» та динамічного програмування.

Приклади популярних жадібних алгоритмів:

- ✓ Dijkstra's Algorithm (Алгоритм Дейкстри)
- ✓ Kruskal's algorithm (Алгоритм Крускала)
- ✓ Prim's algorithm (Алгоритм Пріма)
- ✓ Huffman trees (Дерева Хаффмана)

Задача 8788 (eolimp.com). Монети

У Вас є нескінчена кількість монет номіналами від **1** до **n**. Ви хочете вибрати певний набір монет сумою **s**. Дозволено мати в наборі монети з однаковим номіналом. Яку мінімальну кількість монет необхідно взяти, щоб набрати суму **s**.

Вхідні дані

Два цілих чисел **n** та **s** ($\mathbf{1} \leq \boldsymbol{n} \leq \mathbf{10^5}$, $\mathbf{1} \leq \boldsymbol{s} \leq \mathbf{10^9}$).

Вихідні дані

Виведіть мінімальну кількість монет, необхідну для взяття суми **s**.

Вхідні дані	Вихідні дані
5 11	3
6 16	3

Розв’язання

Очевидно, що слід вибирати монети найбільшого номіналу і брати їх якомога більше. Це реалізується за допомогою жадібного алгоритму.

```
#include <iostream>

using namespace std;

int main()
{
  long n, // максимальний номінал монет
     s, // сума, яку слід набрати монетами
     k = 0; //кількість монет
  cin >> n >> s;
  for ( int i = n; i > 0; i-- ){
/*вибираємо максимальну кількість монет найбільшого
номіналу */
    k += s / i;
    s %= i; //залишок необхідної суми
//якщо сума набрана, то вийти з циклу
    if ( s == 0)
      break;
  }
// друк результату
  cout << k << endl;

  return 0;
}
```

5. Динамічне програмування

Нерідко не вдається поділити задачу на невелику кількість задач меншого розміру, об’єднання розв’язків яких дозволить отримати рішення початкової задачі. У таких випадках пробують поділити задачу на стільки задач, скільки необхідно, потім кожну поділену задачу ділять на ще кілька менших і так далі. Якщо б весь алгоритм зводився саме до такої послідовності дій, то в результаті отримали б алгоритм з експоненціальним часом виконання.

Задача про коника

На числовій прямій сидить коник, який може стрибати вправо на одну або на дві одиниці. Спочатку коник знаходиться в точці з координатою 0. Визначте кількість різних маршрутів коника, що приводять його в точку з координатою n.

Рекурсивне розв'язання

Позначимо кількість маршрутів коника, що ведуть в точку з координатою n, як $F(n)$. Тепер навчимося обчислювати функцію $F(n)$. Насамперед зазначимо, що $F(0) = 1$ (це вироджений випадок, існує рівно один маршрут з точки 0 в точку 0 – він не містить жодного стрибка), $F(1) = 1$, $F(2) = 2$. Як обчислити $F(n)$? У точку n коник може потрапити двома способами – з точки $n-2$ за допомогою стрибка довжиною 2 і з точки $n-1$ стрибком довжини 1. Тобто число способів потрапити в точку n дорівнює сумі числа способів потрапити в точку $n-1$ і $n-2$, що дозволяє виписати рекурентне співвідношення: $F(n) = F(n-2) + F(n-1)$, правильне для всіх $n \geq 2$.

Код на C++:

```
#include <iostream>

using namespace std;

long long f( int n )
{
  if (n < 2)
    return 1;
  else
    return f(n-1) + f(n-2);
}

int main()
{
  int coord_grasshopper;
  setlocale(0, ".1251");
  cout << "Уведіть цілу координату для коника: " << endl;
  cin >> coord_grasshopper;
  cout << f(coord_grasshopper) << endl;
  return 0;
}
```

Час роботи функції зі збільшенням n зростає експоненціально, тобто таке рішення є неприйнятним за складністю. Причина цього полягає в тому, що при

обчисленні рекурсивної функції підзадачі, для яких обчислюється розв'язання, «перекриваються». Тобто для того, щоб обчислити $F(n)$ нам потрібно викликати $F(n-1)$ і $F(n-2)$.

Нерекурсивне розв'язання

Насправді нескладно побачити, що значення рекурсивної функції в даному випадку майже будуть збігатися з числами Фібоначчі, бо обчислюються за тими ж рекурентним співвідношенням. А для обчислення чисел Фібоначчі можна використовувати цикл і масив, а не рекурсію – наступне число Фібоначчі визначається, як сума двох попередніх.

```
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
  clock_t t;
  long long f[100];
  int coord_grasshopper;
  setlocale(0, ".1251");
  cout << "Уведіть цілу координату для коника: " << endl;
  cin >> coord_grasshopper;
  t = clock();
  f[0] = 1;
  f[1] = 1;
  for( int i = 2; i <= coord_grasshopper; i++ ){
    f[i] = f[i - 1] + f[i - 2];
  }
  cout << f[coord_grasshopper] << endl;

  t = clock() - t;
  cout << ((double)t)/CLOCKS_PER_SEC << " c" << endl;
  return 0;
}
```

Складність такого рішення буде $O(n)$. Складність обчислення зменшується за рахунок того, що для кожного проміжного i значення $F(i)$

обчислюється один раз і зберігається в списку, щоб згодом використовувати це значення кілька разів для обчислення $F(i+1)$ і $F(i+2)$.

Такий прийом називається динамічним програмуванням.

Означення. Динамічним програмуванням називається метод розв'язування задач, при якому задача розбивається на підзадачі, відповіді до яких записуються в структуру даних і використовуються при розв'язуванні задачі.

Для розв'язання в кожній з підзадач використовується один і той же метод.

Власне термін «динамічне програмування» належить Річарду Беллману. Слово «програмування» в назві «динамічне програмування» мало не той сенс, що зараз. Коли Річард Беллман вигадав цей термін, програмування означало «планування» і під «динамічним програмуванням» розумілося оптимальне планування багатоступінчастих процесів. Значний внесок у створення загального формалізму послідовного аналізу варіантів даного методу належить київській школі математиків на чолі з В. С. Михалевичем (схема формалізації Михалевича, метод «київський вінник»).

Одним із способів розв'язання задач, для яких не можна побудувати оптимальний розв'язок на кожному кроці є перевірка всіх можливих послідовностей розв'язків. Динамічне програмування значно зменшує кількість варіантів, що слід перевірити, за рахунок виключення послідовностей розв'язків, які не можуть бути оптимальними.

Оптимальної послідовності розв'язків досягають завдяки явному зверненню до **принципу оптимальності:**

оптимальна послідовність розв'язків має таку властивість, що, незважаючи на початковий стан та розв'язок в початковий момент, серед розв'язків, що залишились, завжди міститься оптимальна послідовність розв'язків щодо стану, який утворився після прийняття першого розв'язку.

Нехай S_i – це стан задачі, який утворюється після вибору розв'язку r_i, $1 \le i \le j$. Нехай Γ_i буде оптимальною послідовністю розв'язків відповідно до стану задачі S_i. Тоді, згідно з принципом оптимальності, оптимальну послідовність розв'язків відповідно до S_0 визначають як найкращу послідовність розв'язків $r_i\Gamma_i, 1 \le i \le j$.

Для більшості задач рекурсивна природа задання принципу оптимальності приводить до рекурентного типу співвідношень. Алгоритми динамічного програмування розв'язують ці рекурентні рівняння, щоб знайти оптимальний розв'язок заданої задачі.

Формулювати рекурентні співвідношення динамічного програмування можна за допомогою використання двох підходів перегляду варіантів розв'язань: **прямого та оберненого**.

Нехай $x_1, x_2, \ldots, x_n$ – змінні, що характеризують шукану послідовність розв'язків.

У прямому перегляді побудова розв'язку x_i формулюється в термінах оптимальної послідовності розв'язків для $x_{i+1}, \ldots, x_n$. Для оберненого перегляду формулювання тверджень щодо розв'язку x_i здійснюють у термінах оптимальної послідовності розв'язків для $x_1, x_2, \ldots, x_{i-1}$.

Якщо рекурентні співвідношення сформульовані з використанням прямого підходу, тоді рівняння розв'язують за технікою «перегляду назад» – починаючи з останнього розв'язку. Якщо співвідношення сформульовані з використанням оберненого підходу – їх розв'язують за технікою «перегляду вперед».

Завдяки використанню принципу оптимальності послідовності розв'язків, що містять підпослідовності, які не є оптимальними, здебільшого не розглядаються. Тому алгоритми динамічного програмування, як правило, мають поліноміальну оцінку.

Інша важлива риса динамічного програмування полягає у тому, що оптимальні розв'язки підзадач можна зберігати в такому вигляді, який запобігає перерахуванню їх значень у разі подальших використань цих підзадач. Здебільшого для цього використовують зберігання у структурах (одновимірний чи двовимірний масив, різного роду списки). Використання структурних значень робить природним перетворення рекурсивних рівнянь в ітеративні програми.

За оберненого підходу побудова загального розв'язку із розв'язків підзадач меншої розмірності, обчислення йде від менших підзадач до більших, і відповіді запам'ятовуються в структурі даних.

Перевага використання структур даних полягає в тому, що, як тільки будь-яку підзадачу розв'язано, її відповідь заносять у структуру і ніколи вже не обчислюють знову.

Для аналізу застосовності метода перевіряють умови:

1. У задачі можна виділити однотипні підзадачі різних розмірів.

2. Серед виділених підзадач є тривіальні, які мають малий розмір і очевидне розв'язання.

3. Оптимальні розв'язання підзадачі більшого розміру можна побудувати із оптимальних розв'язань менших підзадач.

4. Одні і ті ж нетривіальні менші підзадачі використовуються при розв'язуванні різних великих підзадач.

5. Кількість різних підзадач розумна і для запам'ятовування результатів не потрібно багато пам'яті.

Перевірка 3 пункта – головний етап в аналізі застосовності динамічного програмування до задачі.

Основні кроки динамічного програмування:

1. Задача розбивається на певну кількість підзадач.

Підзадача – це та ж сама задача, але з меншою кількістю аргументів, або ж тією кількістю аргуметрів, але з меншими їх значеннями.

2. Записуються рекурентні співвідношення, що виражають розв'язання задачі через підзадачі.

3. Здійснюється пошук оптимального розв'язання кожної підзадачі і формування його у вигляді структури даних.

4. Розв'язування задачі будується на основі оптимальних розв'язків підзадач.

Відшукання розв'язку здійснюється так: спочатку здійснюється вибір останнього в часі розв'язання, потім у зворотному порядку вибирається решта розв'язань аж до початкового.

Динамічне програмування використовує ті ж рекурентні співвідношення, що і рекурсивне розв'язання, але на відміну від рекурсії в динамічному програмуванні значення обчислюються в циклі і зберігаються в структурі даних, наприклад, масиві. При цьому заповнення структури даних йде від менших значень до більших, тоді як в рекурсії – навпаки, рекурсивна функція викликається для великих значень, а потім викликає сама себе для менших значень.

Задача. Нехай коник стрибає на одну або дві точки вперед, а за стрибок в кожну точку необхідно заплатити певну вартість, різну для різних точок. Вартість стрибка в точку *i* задається значенням $Price[i]$ списку *Price*. Необхідно знайти мінімальну вартість маршруту коника з точки 0 в точку n.

Розв'язання

Нехай $C(n)$ – мінімальна вартість шляху з 0 в n. Виведемо рекурентне співвідношення для $C(n)$. Щоб потрапити в точку n ми повинні потрапити в точку $n-1$ або $n-2$, мінімальні вартості цих маршрутів дорівнюватимуть $C(n-1)$ і $C(n-2)$ відповідно. До них доведеться додати значення $Price[n]$ за стрибок в точку n. Але з двох точок $n-1$ і $n-2$ потрібно вибрати той маршрут, який має найменшу вартість. Отримали рекурентне співвідношення:

$C(n) = min(C(n-1), C(n-2)) + Price(n)$.

Обчислити значення цільової функції також краще за допомогою методу динамічного програмування, а не рекурсії:

```
#include <iostream>

using namespace std;

int main()
{
```

```
    long long C[100], price[100];
    int coord_grasshopper;
    setlocale(0, ".1251");
    cout << "Уведіть цілу координату для коника: " <<
endl;
    cin >> coord_grasshopper;
    for (int i = 1; i <= coord_grasshopper; i++)
      cin >> price[i];
    C[1] = price[1];
    for (int i = 2; i <= coord_grasshopper; i++)
      C[i] = min(C[i-1], C[i-2]) + price[i];
    cout << C[coord_grasshopper] << endl;
    return 0;
  }
```

Після виконання цього циклу в масиві C буде записана мінімальна вартість маршруту для всіх точок від 0 до n.

Відновлення відповіді

Задача. На числовій прямій сидить коник, який може стрибати вправо на одну або на дві одиниці. Спочатку коник знаходиться в точці з координатою 0. Нехай за стрибок в кожну точку необхідно заплатити певну вартість, різну для різних точок. Вартість стрибка в точку *i* задається значенням $Price[i]$ списку *Price*. Необхідно знайти мінімальну вартість маршруту коника з точки 0 в точку n і сам маршрут мінімальної вартості.

Розв'язання

Для відновлення відповіді будемо для кожної точки i запам'ятовувати номер точки $Prev[i]$, з якої коник потрапив в точку i, якщо він буде пересуватися по шляху мінімальної вартості. Тобто $Prev[i]$ – це точка, що передує точці з номером i на шляху мінімальної вартості (також кажуть, що $Prev$ – це масив попередників). Як визначити $Prev[i]$?

Якщо $C[i-1] < C[i-2]$, то коник потрапив в точку i з точки $i-1$, тому $Prev[i] := i-1$, інакше $Prev[i] := i-2$. Модифікуємо алгоритми обчислення значень цільової функції, одночасно обчислюючи значення $Prev[i]$.

Приклад програми на мові C++:

```
#include <iostream>

using namespace std;

int main()
{
```

```
	long long c[100], price[100];
	int prev[100], Path[100], i , j;
	int coord_grasshopper;
	setlocale(0, ".1251");
	cout << "Уведіть цілу координату для коника: " <<
endl;
	cin >> coord_grasshopper;
	for ( i = 1; i <= coord_grasshopper; i++)
	  cin >> price[i];
	prev[0] = 0;
	c[1] = price[1];
	prev[1] = 0;
	for ( i = 2; i <= coord_grasshopper; i++){
	  if (c[i - 1] < c[i - 2] ) {
	    c[i] = c[i - 1] + price[i];
	    prev[i] = i - 1;
	  }
	  else {
	    c[i] = c[i - 2] + price[i];
	    prev[i] = i - 2;
	  }
	}

	cout << c[coord_grasshopper] << endl;

	// відновлення шляху
	i = coord_grasshopper;
	j = 1;
	while (i > 0){
	  Path[j] = i;
	  i = prev[i];
	  j++;
	}
	Path[j] = 0;

	// друк шляху
	for ( i = j; i > 1; --i)
	  cout << Path[i] << " " ;
	cout << Path[1] << endl;
```

```
  return 0;
}
```

У кінці в список *Path* додається початкова вершина з номером 1, яка не була оброблена в основному циклі, а потім весь список *Path* розгортається в зворотному порядку (бо вершини додаються в зворотному порядку, від кінцевої до початкової).

Динамічне програмування з двома параметрами: таблиці

Розглянемо задачі, схожі на задачі **динамічного програмування з одним параметром**, але замість одновимірних задач на рух по прямій будуть розглядатися переміщення в двовимірному просторі – наприклад, переміщення на шахівниці або на аркуші паперу в клітинку.

Задача. Розглянемо шахівницю, в лівому верхньому кутку якої знаходиться король. Король може рухатися тільки вправо, вниз або по діагоналі вправо-вниз на одну клітку. Необхідно визначити кількість різних маршрутів короля, що приводять його в правий нижній кут.

Розв'язання

Співставимо кожному полю шахівниці її координати (i, j), де i буде позначати номер рядка на дошці, j – номер стовпця. Нумерувати рядки будемо зверху вниз, стовпці – зліва направо, нумерація починається з 0. Тоді початкове положення короля буде поле $(0,0)$.

Позначимо через $F(i, j)$ кількість способів прийти з поля $(0,0)$ в поле (i, j). У поле (i, j) можна прийти з трьох полів – зліва з $(i, j-1)$, зверху з $(i-1, j)$ і по діагоналі з $(i-1, j-1)$. Тому число маршрутів, які ведуть в поле дорівнює числу маршрутів з усіх її попередників, а саме:

$$F(i, j) = F(i, j-1) + F(i-1, j) + F(i-1, j-1)$$

Окремо потрібно задати значення для граничних полів, тобто коли $i = 0$ або $j = 0$. У результаті вийде таблиця заповнена наступним чином:

1	1	1	1	1
1	3	5	7	9
1	5	13	25	41
1	7	25	63	129
1	9	41	129	321

Для заповнення цієї таблиці і підрахунку числа маршрутів можна використовувати наступний фрагмент програми, в якому спочатку створюється двовимірна таблиця, потім заповнюються крайні поля (перший стовпець і перший рядок), потім заповнюються інші елементи таблиці за допомогою наведеного вище рекуррентного співвідношення. У даному прикладі n – число рядків, m – число стовпців на дошці.

…

```
for( int i = 0; i <= n; i++)
 f[i][0] = 1;
for( int j = 0; j <= n; j++)
 f[0][j] = 1;
for( int i = 1; i <= n; i++)
 for( int j = 1; j <= n; j++)
   f[i][j] = f[i][j-1] + f[i-1][j] + f[i-1][j-1];
```

На цьому прикладі можна скласти загальний план розв'язання задачі методом динамічного програмування. Його можна використовувати для розв'язання будь-яких задач за допомогою динамічного програмування:

1. Записати те, що потрібно знайти в задачі, як цільову функцію від деякого набору аргументів (числових, рядкових або ще яких-небудь).

2. Звести розв'язання задачі для довільного набору параметрів до розв'язання аналогічних підзадач для інших наборів параметрів (як правило, з меншими значеннями параметрів). Якщо завдання нескладне, то корисно буває виписати явне рекурентне співвідношення, що задає значення функції для даного набору параметрів.

3. Задати початкові значення функції, тобто ті набори аргументів, при яких задача є тривіальною і можна явно вказати значення функції.

4. Створити масив (або іншу структуру даних) для зберігання значень функції. Як правило, якщо функція залежить від одного цілочисельного параметра, то використовується одновимірний масив, для функції від двох цілочисельних параметрів - двовимірний масив і т. д.

5. Організувати заповнення масиву з початкових значень, визначаючи черговий елемент масиву за допомогою виписаного на кроці 2 рекурентного співвідношення або алгоритму.

Задача. Мишка і зернинки (https://www.eolymp.com/uk/problems/15)

В індійському храмі підлогу прямокутної форми вимощено однаковими квадратними плитками **1** x **1**, на кожну з яких насипано від **0** до **k** зернинок (**k** ≤ **30000**). Розміри підлоги **m** x **n**. Мишка вибігає з лівого нижнього кута підлоги храму і рухається до входу у іншу нірку, розміщену у протилежному кутку. Мишка може рухатись лише праворуч або вперед, збираючи всі зернинки з плитки, на якій вона знаходиться. Знайти маршрут, рухаючись по якому мишка збере найбільшу кількість зернин.

Вхідні дані

Перший рядок містить числа **m** та **n** - розміри підлоги (**1** ≤ **m**, **n** ≤ **100**). Далі йде **m** рядків, починаючи з верхнього, у кожному з яких розміщено **n** чисел – кількість зернинок на відповідній плитці.

Вихідні дані

Вивести маршрут руху мишки у форматі: **RRFFFRF** (**F** - крок вперед, **R** - крок праворуч).

Аналіз алгоритму

Нехай на плитці з координатами (i, j) знаходиться a[i][j] зернят. Нехай res[i][j] містить максимальну кількість зернят, які можна зібрати під час руху з лівого нижнього кута в клітину (i, j). Оскільки на плитку (i, j) можна потрапити або з плитки (i - 1, j), або з (i, j - 1), то

res[i][j] = max(a[i – 1][j], a[i][j – 1]) + a[i][j]

Для зручності написання коду лівій нижній плитці краще присвоїти координати (1, 1). У цьому випадку індекси i – 1 та j – 1 залишатимуться невід'ємними. Можна також обійтися і без масиву res, здійснюючи обчислення в масиві a. Будемо проходити плитки підлоги (комірки масиву) знизу вгору та зліва направо, поклавши

a[i][j] = max(a[i – 1][j], a[i][j – 1]) + a[i][j]

Після цих обчислень a[i][j] вже міститиме максимальну кількість зернят, яку можна зібрати після досягнення плитки (i, j).

Код програми на C++:

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    int a[102][102];
    int n, m, i, j;
    string t;
// Уводимо розмір матриці
    cin >>m >>n;
//Визначаємо, яке із чисел більше
    if (m>n) j=m;
        else j=n;
//Межі матриці заповнюємо числом -1
    for (i=0;i<=j;i++) {
        a[i][0]=-1;
        a[i][n+1]=-1;
        a[0][i]=-1;
        a[m+1][i]=-1;
    }
// Зчитуємо кількість зернинок на плитках
    for (i=1;i<=m;i++)
        for(j=1;j<=n;j++)
```

```
            cin >>a[i][j];
    for (i=m;i>=1;i--)
        for(j=1;j<=n;j++)
            if (!((i==m)&&(j==1)))
              if(a[i][j-1]>a[i+1][j])
                    a[i][j]=a[i][j]+a[i][j-1];
                else
                    a[i][j]=a[i][j]+a[i+1][j];
//Відновлення шляху
    i=1;
    j=n;
    for (;;) {
        if((i==m)&&(j==1)) break;
        if (a[i][j-1]>a[i+1][j]) {
             t='R'+t;
             j--;}
        else {
             t='F'+t;
             i++;}
   }
   cout <<t;
   return 0;
    }
```

Контрольні запитання

1. Що називають повним перебором?
2. Які завдання потрібно розв’язати при розробці алгоритмів повного перебору?
3. У чому суть методу перебору з поверненням?
4. Яка основна ідея парадигми «Розділяй і володарюй»?
5. Які алгоритми називають жадібними?
6. Сформулюйте принцип жадібного вибору.
7. Що називають динамічним програмуванням?
8. Сформулюйте принцип оптимальності для динамічного програмування.
9. Які основні кроки при розв’язуванні задачі методом динамічного програмування?

ЛІТЕРАТУРА

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

2. Алгоритми та структури даних. Навчальний посібник / Т. О. Коротєєва. Львів : Видавництво Львівської політехніки, 2014. – 280 с.

3. Ахо А. Структуры данных и алгоритмы. / Альфренд Ахо, Джон Хопкрофт, Джеффри Ульман. – Изд-ий дом «Вильямс», 2000. – 382 с.

4. Седжвик Р. Фундаментальные алгоритмы на С++. Анализ/ Структуры данных/ Сортировка/ Поиск. – К.: ДиаСофт, 2001.- 688 с.

5. Cormen T. H. Introduction to Algorithms. Third Edition. / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. – Massachusetts: The MIT Press, 2009. – 1292 p.

6. Drozdek A. Data Structures and Algorithms in C++. – Cengage Learning, 2013. – 792 p.

7. Knuth D. E. Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Edition. – Addison-Wesley Professional, 1997. – 672 p.

8. Knuth D. E. Art of Computer Programming, Volume 2: The Seminumerical Algorithms, 3rd Edition. – Addison-Wesley Professional, 1997. – 784 p.

9. Knuth D. E. Art of Computer Programming, Volume 3: The: Sorting and Searching. – Addison-Wesley Professional, 1997. – 800 p.

10. Skiena S. S. The Algorithm Design Manual. Second Edition. – London: Springer-Verlag, 2008. – 748 p.

11. Weiss M. A. Data structures and algorithm analysis in C++ . – Fourth edition. – New York : Pearson, 2014. – 635 p.

Навчальне видання

Упорядник:
В. М. ХАРЧЕНКО

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Курс лекцій

Технічний редактор – І. П. Борис
Верстка, макетування – О. В. Борщ

Книга друкується в авторському редагуванні.

Підписано до друку 10.10.23 р. | Формат 60x84/16 | Папір офсетний
Гарнітура Times | Обл.-вид. арк. 8,4 | Електронне вид-ня
Замовлення № 836 | Ум. друк. арк. 14,3

ВИДАВНИЦТВО НДУ ім. М. В. ГОГОЛЯ

Ніжинський державний університет
імені Миколи Гоголя.
м. Ніжин, вул. Воздвиженська, 3А
(04631) 7–19–72
E-mail: vidavn_ndu@ukr.net
www.ndu.edu.ua

Свідоцтво суб'єкта видавничої справи
ДК № 2137 від 29.03.05 р.