МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Герцюк М.М., Гавор А.С.

АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Навчальний посібник

Київ 2025

УДК 004.421+004.422(075)

Рекомендовано Вченою радою Державного університету інформаційно-комунікаційних технологій (протокол № 12 від 21 жовтня 2025 року)

Герцюк М.М., Гавор А.С. Алгоритми і структури даних. - Навчальний посібник. – К.: ДУІКТ, 2025. – 75 с.

Рецензенти: **Складаний П.М.**, кандидант технічних наук, доцент, завідувач кафедри інформаційної та кібернетичної безпеки імені професора Володимира Бурячка, Київського столичного університету імені Бориса Грінченка,

Корнага Я.І., доктор технічних наук, професор, декан факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Навчальний посібник розроблений на підставі робочої програми кредитного модуля бакалавра з дисципліни “Алгоритми і структури даних”, для опанування теоретичних та практичних навичок, які необхідні майбутнім фахівцям з інженерії програмного забезпечення. Призначений для студентів, які навчаються за освітньою програмою «Технології цифрового розвитку» підготовки бакалаврів за спеціальністю 121 – «Інженерія програмного забезпечення» денної, заочної та дистанційної форм навчання. Спрямований на формування у студентів умінь та набуття практичних навичок, пов’язаних зі структурами даних та алгоритмами. Забезпечує студентів необхідними теоретичними знаннями для опанування відповідної теми лабораторної роботи та виконання завдань, запланованих впродовж семестру.

ISBN 978-617-8521-03-5

Зміст

Вступ

Курс лекцій "Алгоритми та структури даних" призначений для формування глибокого розуміння основ розробки, аналізу та застосування алгоритмів і структур даних у програмуванні. Він охоплює ключові концепції, такі як лінійні та нелінійні структури даних (масиви, списки, стеки, черги, дерева, графи), алгоритми сортування, пошуку, а також методи оцінки їхньої ефективності. Особлива увага приділяється практичному застосуванню знань для оптимізації програмного коду та розв’язання реальних задач. Курс розрахований на студентів із базовими знаннями програмування на C++, дискретної математики та математичної логіки, забезпечуючи міцну основу для подальшого вивчення комп’ютерних наук.

Лекція 1

Тема лекції: Введення в алгоритми та структури даних

План лекції

1. Ознайомлення з курсом
2. Визначення алгоритмів та їхній вплив на програмування
3. Значення структур даних в програмуванні
4. Аналіз ефективності алгоритмів

1. Ознайомлення з курсом

Курс закладає основу для розуміння алгоритмів, структур даних, ефективних алгоритмів обробки даних, оптимізації програмного коду, розробки продуктивних алгоритмів та структур даних, тощо. Основною метою курсу є отримання навичок з розробки, аналізу та використанні різноманітних алгоритмів та структур даних.

Курс розбито на дві частини. Перша розглядає структури даних, зокрема:

- такі лінійні структури даних, як масиви, кортежі, стек, черга, дек;
- такі нелінійні структури даних, як таблиця, хеш-таблиця, граф, дерево, бінарне дерево, купа, червоно-чорне дерево;
- хешування та хеш-функції.

Включаються, також і алгоритми, що потрібні для роботи з висвітленими структурами.

Друга частина розглядає алгоритми, зокрема визначення алгоритмів, оцінка складності та шляхи оптимізації алгоритмів, алгоритми сортування, алгоритми пошуку, евристичні алгоритми, динамічне програмування.

Очікувані результати курсу передбачають:

- використання та реалізація різних структур даних;
- розуміння основних алгоритмічних концепцій, їх розробка та застосування;

- аналіз ефективності алгоритмів, та їх оптимізація.

Курс розраховано на студентів, що мають базові знання з мови C++, математичної логіки, дискретної математики та дискретних структур.

2. Визначення алгоритмів та їхній вплив на програмування

Алгоритмом називається набір інструкцій, що описує порядок дій виконавця, для розв’язання задачі за скінченну кількість дій (за скінченний час).

Основні концепції та властивості алгоритмів визначають їхню природу та роль у вирішенні завдань. Існують різні класифікації проблем залежно від їх розв'язності. Включають розв’язні проблеми, напіврозв’язні проблеми та нерозв’язні проблеми.

Розв'язні проблеми – проблеми, для яких існує алгоритм, що завжди зупиняється і дає правильну відповідь для будь-якого допустимого вхідного значення за кінцевий час. Наприклад, перевірка чи є дане число простим.

Напіврозв'язні проблеми - ті, для яких існує алгоритм, який зупиняється і дає правильну відповідь для деяких вхідних даних, але може ніколи не зупинитися для інших. Приклад: проблема зупинки для конкретної програми та вхідного значення.

Нерозв'язні проблеми є такими, для яких не існує жодного алгоритму, який би міг вирішити їх для всіх можливих вхідних значень. Проблема зупинки є класичним прикладом нерозв'язної проблеми. Одним з прикладів нерозв’язних проблем є проблема зупинки.

Проблема зупинки - проблемою пошуку універсальної програми, що дозволяє за записом довільної програми та вхідних даних установити, чи зупиниться обчислювальний пристрій, що діє відповідно до даної програми і обробляє вхідні дані, або ж він буде працювати нескінченно довго. У 1936 році Аллан Тьюринг її досліджував і довів нерозв'язною.

Таким чином, формулюється наступна задача:

Формулювання задачі: визначити, чи зупиниться алгоритм на певному вхідному значенні, або буде виконуватися нескінченно довго.

Доведення нерозв’язності: Аллан Тьюринг довів, що не існує загального алгоритму, який може вирішити проблему для всіх можливих пар програма-вхідне значення. Тобто, деякі проблеми не можуть бути вирішені жодним алгоритмом, існують межі того, що можуть робити комп'ютери і алгоритми.

Будь-який алгоритм має володіти основними властивостями:

- **масовість:** свідчить придатність алгоритму до багатьох задач певного класу;
- **визначеність:** свідчить про те, що кожна команда алгоритму визначається і тлумачиться однозначно;
- **дискретність:** означає, що алгоритм складається з послідовних завершених команд або дій;
- **результативність:** означає, що кожна дія приводить до певного результату;

- **формальність:** означає, що виконавець може виконати поставлене завдання, діючи за інструкціями, і при цьому може навіть не розуміти його змісту;
- **скінченність:** діючи за алгоритмом, можна отримати результат за скінченну кількість кроків.

Роль алгоритмів у вирішенні завдань у сфері програмування та інших областях є критичною. Алгоритми застосовуються для розв'язання задач, оптимізації процесів, автоматизації процесів, моделювання та аналізу, розробки програмного забезпечення, інтелектуальних систем, криптографії, графічних та візуальних ефектів, інтернету речей, тощо.

Алгоритми можуть бути подані різними способами:

- словами;
- блок-схемою. Приклад блок схеми показано на рис. 1.1, рис. 1.2;

Рис. 1.1 Простий приклад блок схеми.

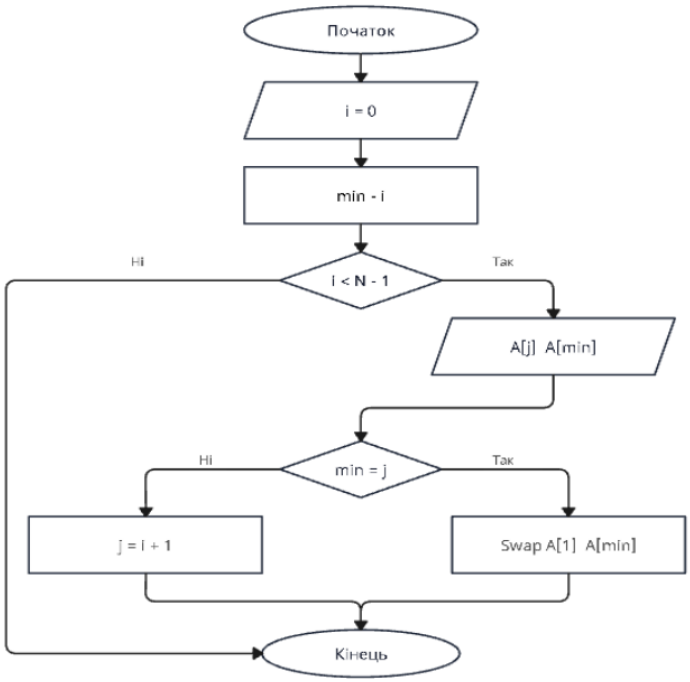


Рис. 1.2 Приклад складнішої блок-схема

- малюнками. Приклад такого малюнку показана на рис. 1.3;

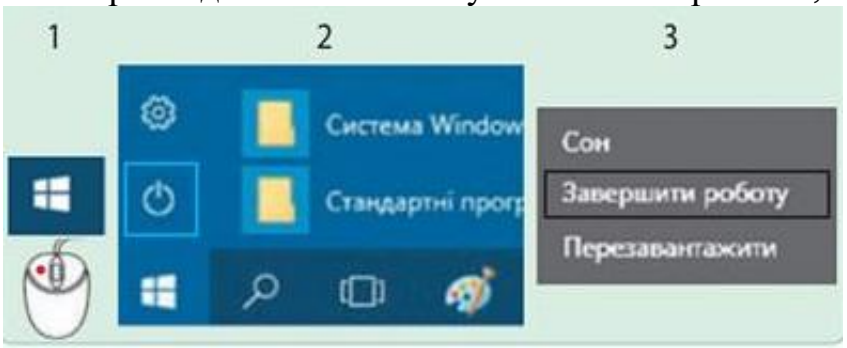


Рис. 1.3 Приклад малюнку

- будь-якою мовою програмування.

3. Значення структур даних в програмуванні

Структура даних - спосіб організації та зберігання даних в комп'ютері. Визначає, як дані можуть бути структуровані, і надає можливості для виконання різних операцій з ними.

Основними призначеннями структур даних є:

1. ефективне зберігання та звертання до даних: структури даних дозволяють ефективно зберігати дані та швидко звертатися до них. Тому, мають визначати способи доступу до окремих елементів та груп даних;
2. використання в операціях та алгоритмах: вдало використані структури даних допомагають виконувати різні операції та алгоритми над даними. Приклад: пошук, сортування, вставка та видалення елементів;
3. управління пам'яттю: структури даних допомагають економно використовувати пам'ять, управляючи виділення та звільненням ресурсів відповідно до потреб програми;
4. побудова та збереження даних: надають зручний та організований спосіб для побудови та збереження даних, спрощуючи роботу з даними;
5. взаємодія з алгоритмами: є ключовим елементом для реалізації різноманітних алгоритмів, дозволяючи їм ефективно працювати з великим обсягом даних;
6. моделювання реальних систем: використовуються для моделювання реальних систем, де структуризація та взаємодія даних важливі для відображення складних відносин та властивостей;
7. інкапсуляція та абстракція: дозволяють приховати деталі роботи з даними та надають абстракції, що спрощує розробку та управління кодом;
8. підтримка об'єктно-орієнтованого програмування: є основними складовими для побудови класів та об'єктів, що визначають властивості та методи;

В контексті роботи зі структурами даних доцільно відмітити, що вони мають бути вдало підібрані для вирішення конкретної ситуації. Для цього, потрібно знати їх правила роботи та особливості використання. В результаті,

можуть бути оптимізовані наступні аспекти роботи програми. З цією метою, потрібно наступні характеристики:

1. доступ до даних: ефективні структури даних дозволяють швидший доступ до елементів;
2. кількість затраченої пам'яті: при правильному використанні, структури даних, дозволяють використовувати пам'ять з економією, оскільки розділяють дані у вигляді вузлів та допомагають уникнути зайвого використання ресурсів;
3. оптимізація операцій: використання відповідних структур даних дозволяє оптимізувати такі операції, як вставка, видалення, сортування та пошук елементів, що призводить до покращення продуктивності програм;
4. простота реалізації: такі структури даних, як масиви або списки, є досить простими для реалізації та зрозуміння. Це, в свою чергу, сприяє швидкому розробленню програм;
5. зручність управління даними: структури даних дозволяють ефективно управляти та організовувати дані, що полегшує роботу з ними та сприяє збереженню порядку;
6. підтримка комплексних операцій: деякі структури даних підтримують комплексні операції, такі як транзакції в базах даних чи операції з об'єктами в об'єктно-орієнтованому програмуванні;
7. покращення читабельності та обслуговування коду: структури даних сприяють створенню більш зрозумілого та обслуговуваного коду, що полегшує розробку та підтримку програмного продукту;
8. підвищення надійності та стабільності: використання відповідних структур даних може покращити надійність та стабільність систем, особливо при обробці великого обсягу даних.

Аналіз структур даних з точки зору описаних аспектів дозволяє розробити оптимізовану програму, або оптимізувати її при подальшій підтримці, розширення, або покращення стану коду. Тому, важливо знати особливості та правила їх роботи.

4. Аналіз ефективності алгоритмів

Аналіз ефективності алгоритмів – це процес визначення та оцінки продуктивності різних алгоритмів при розв'язанні однієї й тієї ж задачі чи виконанні конкретної операції.

Основною метою є визначення швидкості та ефективності алгоритму для вирішення завдання при різних обсягах вхідних даних.

Розглядаються два види складності – **часова** та **просторова**.

Часова складність – характеристика продуктивності алгоритму. Визначається **кількістю елементарних операцій**, які потрібно виконати для реалізації алгоритму. Вказує на те, скільки часу алгоритм займає при роботі з різними обсягами вхідних даних.

Просторова складність алгоритму – характеристика обсягу пам'яті, необхідного для розв'язання задачі. Вимірюється в кількості пам'яті, яку використовує алгоритм для зберігання та обробки даних.

Асимптотична складність

В контексті оцінки складності варто згадати про **асимптотичну складність**, що є способом оцінювання продуктивності алгоритму при збільшенні розміру вхідних даних. Може бути застосована для часової та просторової складності. Асимптотичний, в даному випадку, означає наближення до певного значення, межі або поведінки без досягнення її в кінцевій точці. Умовно, може бути представлений у вигляді деякої кривою-лінією, що відображатиме залежність розміру вхідних даних до кількості операцій. Приклад такого графіку для нотації «Великого О» показано на рис. 1.3.

Існують різні аналітично-математичні нотації, що описуються складність. Ними є велике О, мале О, велике Омега і велике Тета. Найбільш використовуваною є нотація великого О, що вказує на верхню межу, або оцінку вищу, ніж фактична кількість операцій, які виконує алгоритм. Основними складностями є наступні:

- константна (O(1)) - алгоритм виконується за постійний час, незалежно від розміру вхідних даних;
- логарифмічна (O(log n)) - алгоритм виконується логарифмічно відносно розміру вхідних даних;
- лінійна (O(n)) – складність алгоритму збільшується виконується відносно розміру вхідних даних. Такий тип складності характерний для алгоритмів лінійного пошуку або простих ітерацій.
- лінійно-логарифмічна (O(n log n)) - характерно для багатьох ефективних алгоритмів сортування, таких як швидке сортування та злиття;
- квадратична ($O(n^2)$) - алгоритм виконується пропорційно квадрату розміру вхідних даних. Такий тип складності може бути характерним для бульбашкового сортування та інших простих алгоритмів сортування.
- експоненційна $O(2^n)$ - алгоритм виконується експоненційно відносно розміру вхідних даних. Це характерно для деяких найгірших алгоритмів, таких як рекурсивний перебір.

Описані та інші види складності можна умовно відобразити на графіку, що показано на рис. 1.4.

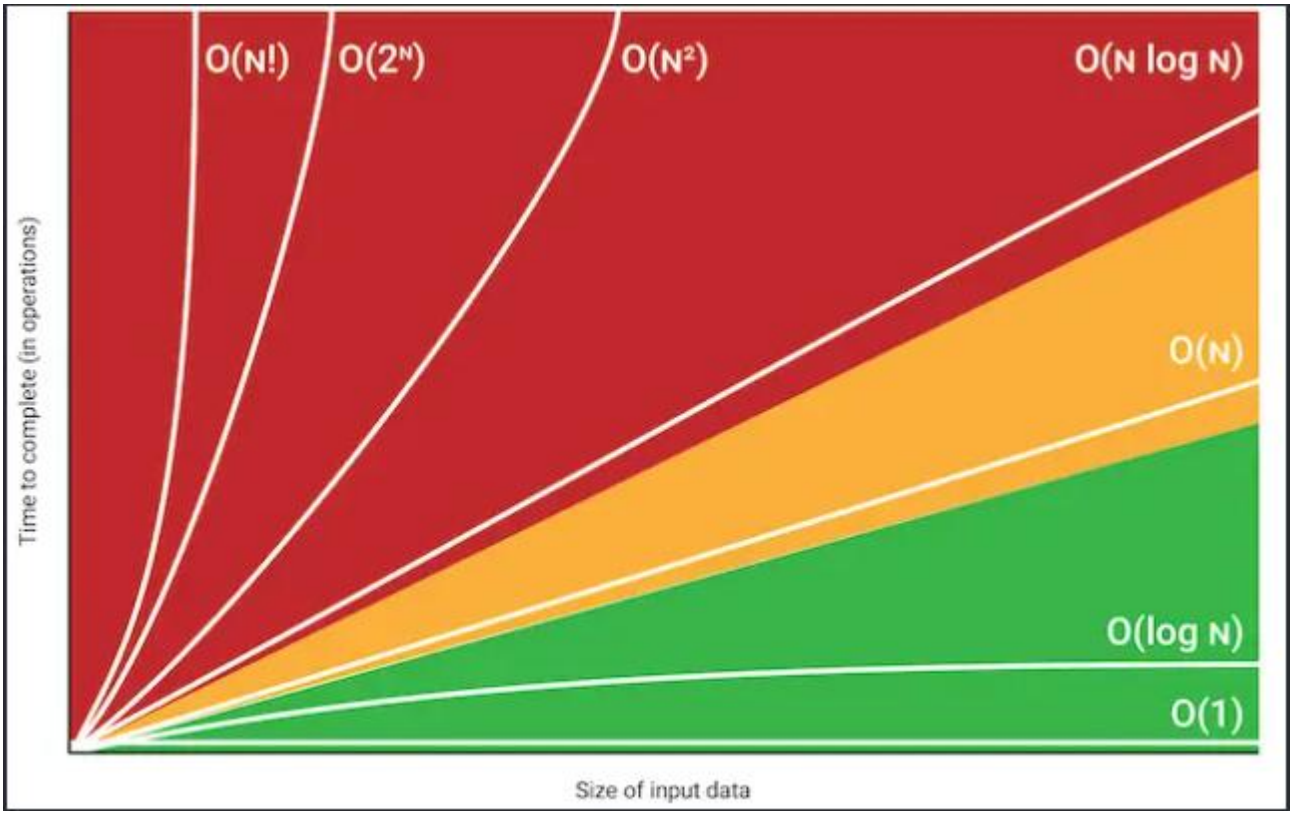


Рис. 1.4 Графік, що відображає нотації «великого О»

Графік відображає залежність розміру вхідних даних до часу виконання. Можна побачити, що найкращими складностями для великої кількість даних є константна та логарифмічна. Такі інші складності, як квадратична, логічно-логарифмічна та експоненційна є небажаними. Лінійна складність є деяким балансом між двома залежностями.

Ці складності важливо враховувати при побудові складних алгоритмів, розробці програм та проектуванні систем, оскільки прямопропорційно впливають на оптимізацію коду.

Як же оцінити складність алгоритму? Для кожного типу складності алгоритм оцінювання свій.

Для часової складності:

1. Ідентифікуємо основні операції та визначимо їхню частоту виконання залежно від розміру вхідних даних.
2. Потім потрібно підсумувати час виконання операцій для отримання загального часу виконання алгоритму.
3. Результат потрібно висловити у нотації.

Просторова складність має наступним алгоритм:

1. Перш за все, потрібно визначити кількість використаної пам’яті, як для зберігання даних, так і роботи алгоритму;
2. Далі, потрібно оцінити, як обсяг пам’яті залежить від розміру вхідних даних.
3. Результат потрібно висловити у нотації.

Нижче, приводиться приклад різного коду C++, що відповідає деякій нотації:

Константна (O(1)):

```
void f(){
```

```
for(int i=0; i<100;i++){
…
}
}
```
, бо в будь-якому випадку відпрацює за 100 циклів.

Лінійна - O(N):
```
void f(int n){
for(int i=0; i<n;i++){
…
}
}
```
, бо n циклів.

Лінійна - O(N):

```
void f(int n){
for(int i=0; i<n;i++){
…
}
for(int i=0; i<n;i++){
…
}
}
```
n циклів

або
```
void f(int n){
for(int i=0; i<n;i++){
…
}
for(int i=0; i<n;i++){
…
}
}
```
, хоча і циклів n * 2, але при оцінці залежності кількість операцій буде залежать від змінної n.

O(N+K):
```
void f(int n, int k){
for(int i=0; i<n;i++){
…
}
for(int i=0; i<k;i++){
…
}
}
```
, бо залежить від двох змінних – n і k.

O(N*K):
```
void f(int n, int k){
for(int i=0; i<n;i++){
        for(int i=0; i<k;i++){
        …
        }
}
}
```
, бо на одну ітерацію n припадає k циклів.

Квадратична O(N^2):

```
void f(int n){
for(int i=0; i<n;i++){
        for(int i=0; i<n;i++){
        …
        }
}
}
```

, бо на одну ітерацію n припадає n циклів.

Таким чином, оцінка складності алгоритму є важливим етапом в аналізі програмного коду з метою його оптимізації та адаптації.

Практичне завдання

1. Змоделюйте ріст часу роботи різних алгоритмів пошуку й сортування для масивів різного розміру (10, 100, 1 000, 10 000, …). Реалізуйте простий пошук та найпростіший метод сортування. Побудуйте графіки залежності часу від розміру вхідних даних.

Питання до самоконтролю

1. Що таке алгоритм?
2. Які є способи вираження алгоритмів?
3. Які є властивості алгоритмів?
4. Як оцінюється складність алгоритмів?
5. Що таке структура даних?
6. Яке є призначення структури даних?
7. Назвіть деякі з можливих переваг використання ефективних структур даних.

Лекція 2

Тема лекції: Масиви, множини, кортежі

План лекції

1. Масиви
2. Множини
3. Кортежі

1. Масиви

Масив - структура даних, що зберігає набір елементів. Кожен з його елементів має індекси, за якими його можна знайти у масиві. Іншими словами, вони можуть бути ідентифікованими або адресованими. Елементи розташовані упорядковано та забезпечують ефективне зберігання та роботу з даними.

Нумерація масиву починається з нуля. Масив має фіксовану довжину, яка визначається при його створенні. Розмір залишається незмінним протягом всього життєвого циклу.

Елементи масиву мають однаковий тип даних, що означає, що вони мають однаковий розмір у пам'яті.

Масиви бувають статичні та динамічні.

Статичний масив є масивом, розмір якого фіксується при його оголошенні та залишається незмінним протягом виконання програми. Оголошення та виділення пам'яті для статичного масиву зазвичай відбувається на етапі компіляції програми. Величина статичного масиву повинна бути вказана до початку виконання програми, і вона не може змінюватися під час її роботи.

Приклад оголошення статичного масиву з розміром в 5 елементів в мові програмування C++:

```
int staticArray[5];
```

, де int – тип масиву, staticArray – назва масив, 5 – розмір.

Динамічний масив - масив, розмір якого може визначатись під час виконання програми. Різниця від статичного масиву полягає у тому, що розмір динамічного масиву можна задати під час ініціалізації, на відміну від статичного масиву, де визначається під час оголошення.

Приклад створення динамічного масиву C++:

```
int* dynamicArray = new int[5];
```

, де int* dynamicArray – є змінною вказівника на масив з типом int, new int[5] – ініціалізація самого масиву за посиланням.

Розмірності масивів. Розрізняють одновимірні та багатовимірні масиви. Багатовимірні масиви застосовуються для представлення, як матриць, у випадку двох вимірів, так і інших складних структур даних, якщо що мають більше.

Двовимірний масив. Двовимірний масив є матрицею, де кожний елемент належить одночасно двом лінійним послідовностям – стовпцю. Приклад оголошення матриці мовою С++:

int array[i][j];

, де i – рядок, j – стовпець.

Прикладом двовимірного масиву є зберігання та відображення пікселів зображення.

Тетраедральний масив. Тетраедральний масив (трикутний масив) – є тривимірним масивом, де кожний елемент належить стовпцю, колонці та сторінці одночасно. Має 3 індекси. Приклад оголошення такого масиву мовою С++:

int array[i][j][k];

, де i – рядок, j – стовпець, k – сторінка.

Приклад описаної схеми тетраедрального масиву показано на рис. 2.1.

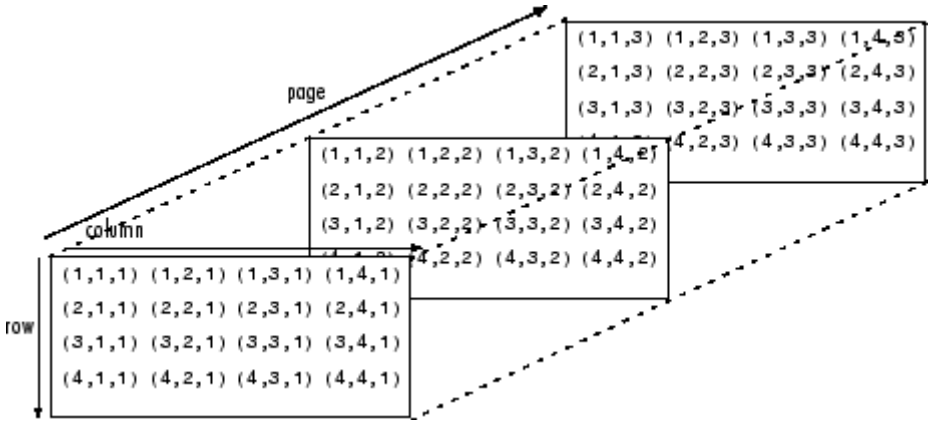


Рис. 2.1 Схема тетраедрального масиву

Аналогічним чином може бути визначено і більше вимірів. Багатовимірні масиви допомагають ефективно зберігати й обробляти великі обсяги інформації в програмі, що може бути важливим для розв’язання завдань у різних галузях, як-от наукові дослідження, комп’ютерний зір, оброблення зображень і звуку та багато інших.

Розріджена матриця. Розрідженою матрицею є структура даних, що застосовується для зберігання матриці, у якої багато значень дорівнюють значенню на замовчанням, таким чином, не потребуючи зберігання в пам'яті. Така матриця особливо ефективна для матриць, де більшість елементів рівні нулю. Прикладами є задачі лінійної алгебри, відображення графів через масиви, обчислення з розділенням, тощо.

Основні операції над масивами наступні:

- пошук елемента за заданим індексом;
- запис елемента в масив;
- злиття масивів і розбиття масиву на частини;
- сортування масивів за деякими правилами;
- копіювання масивів.

Приклад ініціалізації динамічного та статичного масиву мовою С++:

```
// Статичний масив цілих чисел розміром 5
```

```
int arr[5] = {10, 20, 30, 40, 50};

// Динамічний масив із 5 елементів
int* arr = new int[5]{10, 20, 30, 40, 50};
```

2. Множина

Множина – структура даних: набір елементів, кожен з яких є унікальним. Елементи у множині не впорядковані. Нульова множина – та, у якої немає елементів.

Підмножиною є множина є іншої множини. Елемент є членом множини, якщо він входить до складу елементів підмножини. Надмножиною є множина, що мітить усі елементи множини.

Над множинами визначені наступні специфічні операції:

1. об’єднання множин: операція, що зливає елементи початкових множин;
2. перетин множин: операція, що відокремлює спільні елементи початкових множин;
3. різниця множин: операція, що виокремлює елементи першої множини, які не входять до другої;
4. перевірка на входження елемента в множину: операція повертає булеве значення “true/false”, що вказує чи входить елемент в множину.

Тип даних „множина” мовою програмування C/C++, але може бути реалізований його користувацький варіант.

Приклад застосування множини мовою C++:

```
// Множина з унікальними значеннями
std::set<int> s = {2, 3, 5, 7, 11};

// Додаємо новий елемент
s.insert(13);

// Перевірка наявності елемента
if (s.count(5)) {
   // 5 присутнє у множині
}
```

3. Кортеж

Кортеж - структура даних, що дозволяє зберігати кілька елементів різних типів в одній змінній.

Основна відмінність кортежу від інших структур даних, полягає в тому, що кортеж може містити елементи різних типів. Розмір кортежу фіксований і традиційна його реалізація є фіксованою, але деякі реалізації передбачають його, як динамічну структуру.

Приклад застосування кортежу мовою C++:

```
#include <iostream>
#include <tuple>
```

```
int main() {
    // Створення кортежу
    std::tuple<int, double, std::string> myTuple(42, 3.14, "Hello");

    // Звертання до елементів кортежу
    int intValue = std::get<0>(myTuple);
    double doubleValue = std::get<1>(myTuple);
    std::string stringValue = std::get<2>(myTuple);

    // Вивід значень
    std::cout << "Int: " << intValue << std::endl;
    std::cout << "Double: " << doubleValue << std::endl;
    std::cout << "String: " << stringValue << std::endl;

    return 0;
}
```

4. Порівняння структур даних

Не менш важливим питанням у використанні структур даних є розуміння їх різниці. Порівняння структур даних масиву, множини та кортежу приведено у табл. 2.1.

Таблиця 2.1

Порівняльна таблиця: переваги та недоліки

Структура даних	Переваги	Недоліки
Масив	- Простота використання - Швидкий доступ за індексом - Економне використання пам’яті	- Фіксований розмір (статичний) - Не можна динамічно додавати/видаляти елементи
Множина	- Зберігає лише унікальні значення - Легко перевіряти наявність елемента	- Немає доступу за індексом - Порівняно повільніше додавання/пошук, ніж у масиві для невеликих обсягів
Кортеж	- Може зберігати значення різних типів - Легко групувати кілька змінних разом	- Не підтримує зміну розміру - Не можна перебирати елементи у циклі як у масиві

Розуміння переваг та недоліків дає розуміння того, в яких ситуація доцільно використовувати різні структури даних.

Практичні завдання

1. Створіть кортеж з трьох елементів різних типів (int, double, string) та продемонструйте доступ до кожного елемента. Реалізуйте функцію, що приймає кортеж і виводить його елементи на екран.
2. Напишіть програму для знаходження найбільшого та найменшого елементів у масиві Використайте як статичний масив.
3. Виконайте основні операції над двома std::set вручну. Задано два std::set<int> — знайти об’єднання, перетин та різницю (без використання стандартних алгоритмів set_union тощо). Поясніть, як працює кожна з операцій та на яких задачах множини доцільніше використовувати замість масиву.

Питання до самоконтролю

1. Що таке масив?
2. Як визначається розмір масиву?
3. Що таке множина?
4. Що таке кортеж?
5. Які різниця між множиною і кортежем?

Лекція № 3

Тема лекції: Лінійні структури даних. Стек. Черга. Дек.

План лекції

1. Лінійні структури даних
2. Стеки
3. Черги
4. Деки

1. Лінійні структури даних

Лінійні структури даних є типом структур даних, що відображає послідовну логіку розташування елементів, тобто елементи розташовуються послідовно, один за одним. У таких структурах кожен елемент має свого попередника і наступника. Винятком є перший і останній елемент (якщо елемент не має попередника – означає, що це перший елемент, не має наступника – елемент є останнім).

Основні типи лінійних структур даних включають:

- масив;
- кортеж;
- список;
- стек;
- черга;
- дек.

2. Стек

Стек – це лінійна структура даних, що задовольняє наступним умовам:

- є динамічною;
- новий елемент додається і дістається з одного боку послідовності;
- елемент зберігається в колекції до момент вилучення.

Працює за принципом LIFO (last in, first out) – останній пішов, перший прийшов.

Вершиною стеку є самий „верхній” елемент стеку. В даному випадку це є останній доданий. Тілом стеку є інші елементи, що зберігаються в структурі.

Основні операції над стеком наступні:

- **push** - включення нового елемента;
- **pop** - вибірка(вилучення) елемента зі стеку;

Корисною може бути операція визначення поточної кількості елементів в стеку.

Прикладами стеку є стос тарілок та магазин патронів рушниці.

Стек можна реалізувати, використовуючи наступні структури даних:

1. масивами. Стек може бути реалізований як масив, де елементи додаються та видаляються з кінця масиву (зазвичай з кінця з індексом, що збільшується або зменшується на 1). Важливо контролювати верхню границю масиву, щоб уникнути переповнення чи видалення з порожнього стеку;
2. вказівниками. Стек може бути реалізований за допомогою вказівників (вказівників на вершину стеку). Додавання елементу відбувається за допомогою зсуву вказівника, а витягування - зворотнього зсуву;
3. зв'язний список. Зв'язний список може бути використаний для реалізації стеку, де кожен вузол містить дані та вказівник на попередній вузол;
4. динамічний масив (вектор). Стек можна реалізувати за допомогою динамічного масиву, що може автоматично змінювати свій розмір під час додавання чи видалення елементів;
5. рекурсія. Рекурсивна функція може служити стеком, де кожен виклик функції додає свій стан в стек, а повернення з функції видаляє його;
6. дек. Двостороння черга може служити ефективною реалізацією стеку, оскільки обидва кінці можна використовувати для додавання та видалення елементів.

Основні алгоритми додавання та витягування елементів, у випадку реалізації стеку вказівником відображено на рис. 3.1 та рис. 3.2.

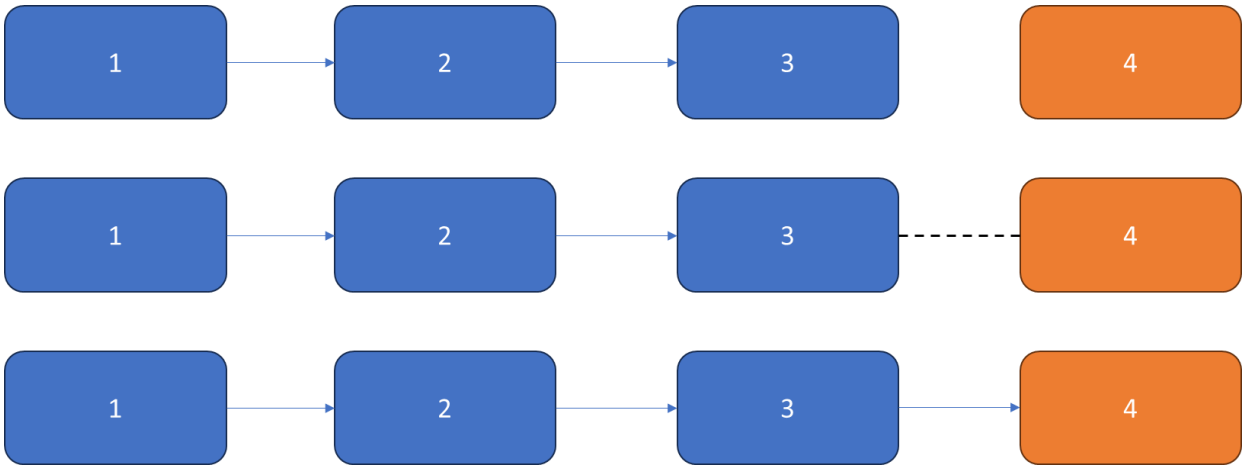


Рис. 3.1 Додавання елементів до стеку у випадку реалізації вказівником (PUSH)

При додаванні нового елементу, його загальна структура має відображати, як значення, так і посилання на наступний елемент. При цьому, в коді має міститись посилання на голову списку – останній доданий та перший витягуваний елемент. Приклад такої структури мовою C++:

```
struct StackElement {
    int data;
    StackElement* next; // Вказівник на наступний елемент стеку
};
```

Схема відображає ситуацію, коли до 3-х елементів потрібно додати 4-й.

Така структура містить значення та посилання на наступний елемент. Для додавання 4-о елементу наступний – потрібно додати його, як(умовно):

4-й елемент.next = 3-й елемент;

При цьому, головою списку стає 4-й елемент. Таким чином, головою стеку буде останній доданий елемент, а новий елемент буде додано в кінець стеку.

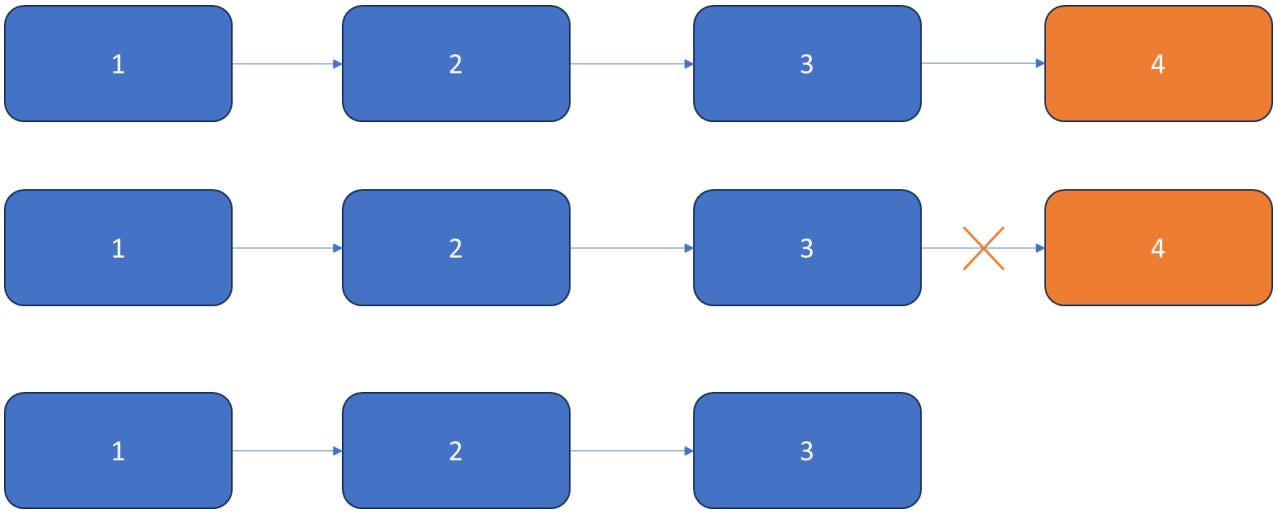


Рис. 3.2 Витягування елементів до стеку у випадку реалізації вказівником (POP)

Витягування елементу відбувається шляхом видалення посилання
3-ї елемент.next = nullptr;
, попередньо зберігши його значення з тимчасовій змінній. Після цього структуру 4-о елементу потрібно очистити з пам’яті.

Приклад реалізації стеку масивом мовою C++:

```
class ArrayStack {
    const int MAX_SIZE = 100;
    int data[MAX_SIZE];
    int top;
public:
    ArrayStack() : top(0) {}

    void push(int x) {
        if (top < MAX_SIZE)
            data[top++] = x;
        else
            throw std::overflow_error("Stack overflow");
    }

    int pop() {
        if (top > 0)
            return data[--top];
        throw std::out_of_range("Stack is empty");
    }
};
```

Даний приклад відображає реалізацію методів push та pop для стеку, реалізованого на основі масиву розміром 100 елементів.

3. Черга

Черга – лінійна структура даних, для якої виконуються такі умови:

- є динамічною;
- новий елемент приєднується завжди з одного, вилучення – з протилежного.

Працює за принципом FIFO(first in, first out) – останній пішов, перший прийшов.

Основні операції з чергою:

- enqueue - додавання елемента в кінець черги;
- dequeue - вибірка елемента з початку черги;

Корисною може бути також допоміжна операція визначення поточної кількості елементів в черзі.

Черги використовуються в різних областях для управління послідовністю завдань чи об'єктів:

1. системи обробки запитів;
2. алгоритми обходу графів;
3. системи керування ресурсами;
4. реалізація буфера;
5. системи керування подіями;
6. керування друкарськими завданнями.

Існує кілька способів реалізації черги:

1. масив. Черга може бути реалізована на основі масиву, де елементи додаються в кінець масиву, а видаляються з його початку. Такий підхід може вимагати періодичного переміщення елементів при видаленні, що може призвести до втрати часу;
2. список. Використання зв'язного списку для реалізації черги дозволяє додавати елементи в кінець та видаляти їх з початку.
3. динамічний масив;
4. дек: Може бути реалізовано на основі деку, що підтримує всі існуючі послідовності.

Основні алгоритми додавання та витягування елементів, у випадку реалізації черги вказівником відображено на рис. 3.3 та рис. 3.4.

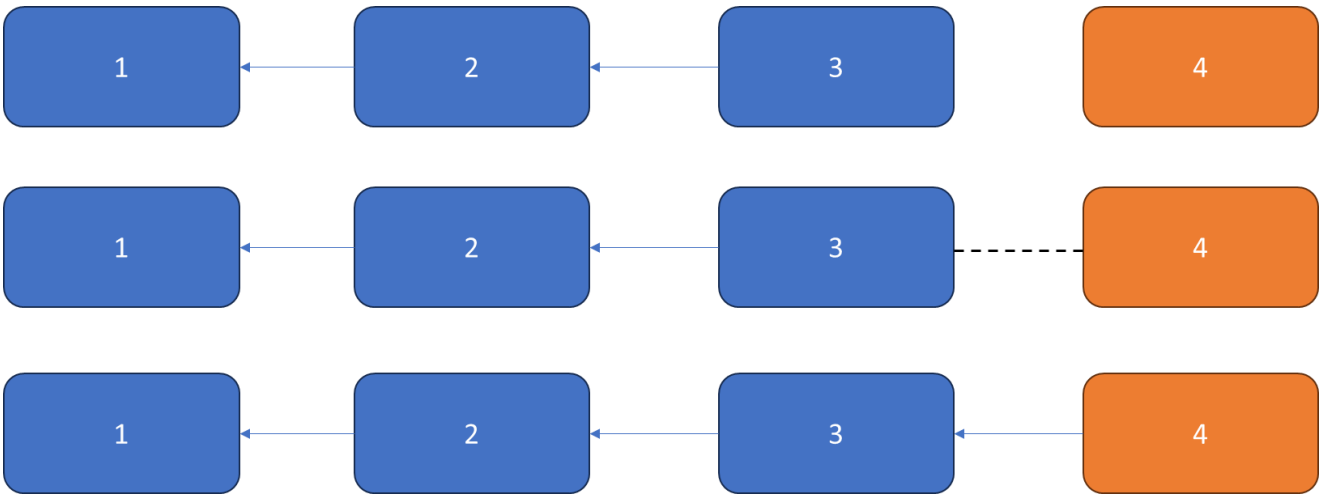


Рис. 3.3 Додавання елементів до черги у випадку реалізації вказівником (ENQUEUE)

При додаванні нового елементу, його загальна структура має відображати, як значення, так і посилання на наступний елемент. При цьому, в коді має міститись посилання на голову списку – перший доданий елемент, та хвіст списку - останній доданий елемент.

Приклад такої структури мовою C++:

```
struct QueueNode {
   int data;
   QueueNode* next; // Вказівник на наступний елемент черги
};
```

Схема відображає ситуацію, коли до 3-х елементів потрібно додати 4-й. Така структура містить значення та посилання на наступний елемент. Для додавання 4-о елементу наступний – потрібно додати його, як(умовно):

3-й елемент.next = 4-й елемент;

Таким чином, новий елемент буде додано в кінець черги. Голова черги при цьому не змінюється, а змінюється лише хвіст черги.

Приклад реалізації черги масивом наступний:

```
class CircularQueue {
   const int MAX_SIZE = 100;
   int  data[MAX_SIZE];
   int  front = 0;
   int  rear  = 0;
   int  size  = 0;
public:
   // Додаємо елемент у кінець черги
   void enqueue(int x) {
      if (rear == MAX_SIZE) throw std::overflow_error("Queue overflow");
      data[rear] = x;
      rear = rear + 1;
      ++size;
   }

   // Вибираємо елемент з початку черги
   int dequeue() {
      if (front == rear) throw std::underflow_error("Queue underflow");
      int val = data[front];
      front = front + 1;
      --size;
      return val;
   }
};
```

Даний приклад відображає логіку реалізації черги статичним масивом data з максимальною кількістю 100 елементів. Для повного функціонування класичної черги методу rear не вистачає виклику метод оберненого зсуву елементів, оскільки при діставанні елементів з черги, її початок буде зміщуватись вправо.

4. Дек

Дек – лінійна структура даних, у якій виконуються такі умови:

- є динамічною;

- один елемент може приєднуватись з обох боків послідовності;
- вибірка елементів можлива з обож боків послідовності.

Основні операції з деком:

- pushback - додавання в кінець дека;
- pushfront - додавання в початок дека;
- popback - вибірка з кінця дека;
- popfront - вибірка з початку дека;

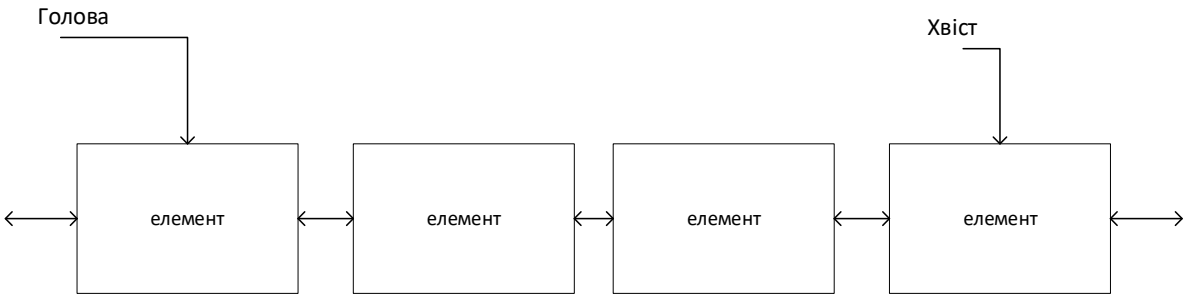


Рис. 3.4 Приклад роботи деку

Існує кілька способів реалізації дека:

1. масив. Використання масиву для реалізації дека. Елементи додаються та видаляються з обох кінців масиву. Такий підхід може вимагати періодичного зміщення елементів, що може бути витратним з точки зору часу.
2. вказівник;
3. двосторонній зв'язаний список;
4. дві черги. Використання двох черг, де одна відповідає за початок, а інша за кінець дека. Цей підхід може забезпечувати зручну реалізацію дека через операції enqueue та dequeue.

5. Складність лінійних структур

Як і інші структури даних, реалізація цих має свою складність виконання. В таблиці 3.1 приведена обчислена складність для визначених структур даних.

Таблиця 3.1

Порівняння складностей для лінійних структур даних

Структура	**Основні операції**	**Часова складність виконання**	**Особливості**
Стек	push, pop	O(1)	Доступ лише до верхнього елемента (LIFO)
Черга	enqueue, dequeue	O(1)+O(n)	Доступ лише до початку/кінця (FIFO). Потрібен обернений зсув при вилученні елемента.
Дек	push_front, push_back, pop_front, pop_back	O(1)+O(n)	Додавання/видалення з обох кінців. Підкоряється принципам FIFO, LIFO. Потрібен обернений зсув при вилученні елемента спочатку.

Всі структури даних містять складність O(1), що означає, що всі основні операції виконуються за сталий час для звичайних реалізацій. Однак, така операція, як обернений зсув елемент потребує зсуву всіх елементів вліво, що є затратною операцією, оскільки вимагає проходу всієї структури. Тому, часова складність буде визначена, як O(n), де n – кількість елементів в структурі.

Практичні завдання

1. **Симулятор браузера на стеку.** Реалізуйте програму, яка імітує історію переходів браузера:
   - Стек «назад» (back), стек «вперед» (forward).
   - Операції: перейти на нову сторінку, повернутись назад, перейти вперед.

   Поясніть, чому стек — оптимальна структура для цієї задачі.

2. **Порівняйте продуктивність реалізацій черги:**
   1. Реалізуйте чергу на масиві (кільцевий буфер) за допомогою std::deque.
   2. Проведіть експеримент: виконайте 1 мільйон операцій додавання та видалення.
   3. Заміряйте час виконання для кожного варіанту, побудуйте порівняльну таблицю або графік.
   4. Зробіть висновок: яка реалізація ефективніша для великих обсягів даних.

3. Реалізуйте дек і протестуйте час виконання всіх основних операцій (push_front, push_back, pop_front, pop_back).

Питання до самоконтролю

1. Що таке лінійні структури даних?
2. Опишіть структуру стек.
3. Опишіть структуру черга.
4. Опишіть структуру дек.

Лекція № 4

Тема лекції: Лінійні списки. Нелінійні структури даних. Таблиці. Хешування.

План лекції

1. Списки
2. Однонапрямлені списки
3. Двонапрямлені списки
4. Циклічні списки
5. Нелінійні структури даних
6. Хешування

1. Лінійні списки

Лінійний список є динамічною лінійною структурою даних, що містить елементи одного типу. Довжина списку визначає кількість елементів, що містяться у послідовності. Оскільки, структура є динамічною, розмір списку може змінюватись. Є гнучкою, оскільки є динамічною та елемента доступні для вставки та видалення. Також, елементи можна отримати за індексом.

Над лінійним списком допустимі наступні операції:

- вставка: вставка елементу в конкретну позицію в списку;
- отримання елементу з списку: повертає елемент, що знаходиться в конкретній позиції списку;
- видалення: видаляє елемента за індексом;
- інші можливі операції над списком.

Найпростіший варіант реалізації списку є масив. Визначивши масив і додавши відповідні операції можна працювати з такою комбінацією, як з повноцінним списком.

При зв’язному представленні кожен елемент списку складається із значення і покажчика, який вказує на наступний елемент у списку.

2. Однозв’язний список

Однозв'язний список – список з елементів, кожен з яких має дані та вказівник на наступний вузол. Основні характеристики однозв'язного списку включають:

- елемент має включати дані (інформацію, яку тримає елемент (наприклад, ціле число, рядок і т. д.).) та вказівник на наступний вузол (показує на наступний елемент в списку). У випадку останнього елемента списку вказівник може бути рівним nullptr або NULL;
- мають бути присутніми такі операції, як:

- вставка елементу: додавання нового елементу в кінець списку, або після визначеного індексу;
- видалення: видалення елементу зі списку;
- звернення до елементу: звертання до конкретного елементу за індексом.

UML діаграма, що відображає елемент такого списку показана на рис. 4.1.

Рис. 4.1 UML діаграма, що відображає вузол списку

Змінна value зберігає дані вузла типу int, next – покажчик на наступний елемент у списку.

Однозв'язний список має просту структуру та може бути використаний у випадках, коли важливо легко додавати та видаляти елементи в середині списку, але доступ до елементів за індексом не є критичною операцією. Схематично, однозв’язних список показано на рис. 4.1

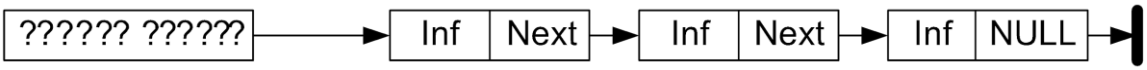


Рис. 4.2 Схема однозв’язного списку

Як видно з рис. 4.2, такий список починається з голови (першого елементу) та містить посилання на наступний. Наступний, в свою чергу, на наступний і т.д. Як зазначалось раніше, якщо елемент не містить посилання на наступний елемент (null) – цей елемент вважається останнім. Приклад вставки елемента в кінець списку мовою C++:

```
struct Node {
    int data;
    Node* next;
    Node(int x) : data(x), next(nullptr) {}
};

void insertAtEnd(Node* current, int value) {
    if (!current) return;
    Node* newNode = new Node(value);
    newNode->next = current->next;
    current->next = newNode;
}
```

Рис. 4.3 відображає блок-схему алгоритму вставки елемента у довільну позицію.

Рис. 4.3 Блок-схема алгоритму вставки елемента у довільну позицію

Розглядаються дві гілки від однієї умови: якщо список порожній, і навпаки. Якщо список порожній (head=null):

1. Створюється єдиний вузол і стає головою списку.

Якщо список має елементи:

2. Відбувається перехід до потрібної позиції, створення нового вузла та перев’язка покажчиків:

new_node->next = current->next; current->next = new_node.

Такі умови є важливими для повного функціонування операцій списку.

3. Двозв’язний список

Двозв'язний список є структурою даних, подібною до однозв'язного списку з відмінністю, що кожен елемент має два вказівники. Один посилається на наступний елемент, інший — на попередній. Основні характеристики двозв'язного списку включають:

- елемент має включати дані (інформацію, яку тримає вузол (наприклад, ціле число, рядок і т. д.).) та вказівник на наступний вузол (показує на наступний елемент в списку). У випадку останнього елемента списку вказівник може бути рівним nullptr або NULL;
- мають бути присутніми такі операції, як:
- вставка елементу: додавання нового елементу в кінець списку, або після визначеного індексу;
- видалення: видалення елементу зі списку;

- звернення до елементу: звертання до конкретного елементу за індексом.

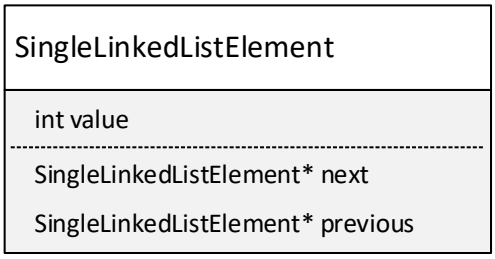


Рис. 4.4 UML діаграма, що відображає вузол списку

Двозв'язний список дозволяє ефективніше виконувати операції вставки та видалення елементів в середині списку, оскільки можна легко пересуватись в обидва напрямки. Втім, це призводить до більшого використання пам'яті на зберігання вказівок. Схематично, двозв’язний список показано на рис. 4.5.

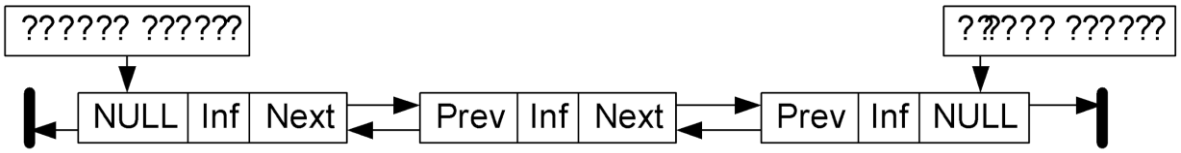


Рис. 4.5 Схема двозв’язного списку

Такий список починається з голови (першого елементу) та закінчується кінцем списку. Містить посилання на наступний та попередній елементи. Як зазначалось раніше, якщо елемент не містить посилання на наступний елемент (null) – цей елемент вважаться останнім, якщо ж не посилається на попередній – цей елемент вважається першим.

4. Циклічний список

Циклічний список - структура даних, з особливістю того, що останній елемент посилається на перший, утворюючи цикл. В останньому елементі циклічного списку вказівник на наступний елемент вказує не на nullptr (або NULL), як в попередніх типах списку, а на початковий елемент списку. Це посилання і є умовою останнього елементу списку.

Циклічні списки можуть мати різні застосування, такі як:

- циклічні черги;
- циклічні списки з обмеженим доступом;
- алгоритми з динамічним розміром буфера.

Основна властивість циклічного списку — його здатність обходити елементи в нескінченному циклі. Схематично, однозв’язних список показано на рис. 4.6.

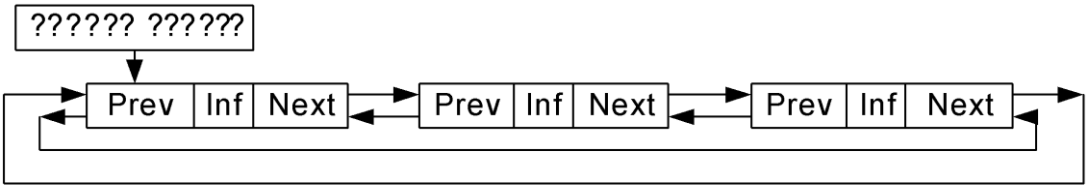


Рис. 4.6 Схема циклічного списку

Такий список починається з голови (першого елементу) та закінчується елементом, що посилається на голову. Такий цикл може спросити проходження черги, і оптимізувати деякі процеси. Однак, його «циклічність» може викликати неправильну роботу програму, якщо таку особливість правильно не обробити.

5. Складність операцій циклічного списку

Усі операції циклічного списку мають визначену складність. Аналізуючи її можна визначити, який тип списку потрібен для конкретних ситуацій. Приклади таких складностей описані в таблиці 3.3.

Таблиця 3.3

Порівняння складностей для різних типів списку

Тип списку	Доступ за індексом	Додавання/видалення в кінці	Перехід вперед	Перехід назад	Складність пошуку	Складність пам’яті
Однозв’язний	O(n)	O(n)	O(1)	Ні	O(n)	Низька
Двозв’язний	O(n)	O(1)	O(1)	O(1)	O(n)	Середня
Кільцевий	O(n)	O(1)	O(1)	(залежить від типу)	O(n)	Середня
Вектор/масив	O(1)	O(n)	O(1)	O(1)	O(n)	Низька

Проаналізувавши таблицю можна побачити, що мають місце стала O(1) і лінійна складності O(n).

6. Нелінійні структури даних

Нелінійні структури даних - структури даних, що не можуть бути організовані у вигляді послідовності або списку. Основна відмінність між лінійними та нелінійними структурами даних полягає в тому, як вони організовують та зберігають дані.

Нелінійні структури даних включають:

- дерева;

- графи;
- матриці;
- таблиці.

Кожна зі згаданих структур даних буде розглянута окремо.

Нелінійні структури даних забезпечують більше гнучкості у представленні складних взаємодій та відносин у даних. Такий підхід може бути важливим для вирішення багатьох завдань, або написання специфічних алгоритмів.

7. Таблиця

Таблиця – структура даних, що представляє собою набір поіменованих записів довільної природи, з кожним з яких пов'язане його ім'я. Ім'я запису називають ключем.

Таблиці є основною структурою запам'ятовування у файлових структурах, організованих на зовнішній пам'яті комп'ютера. Ключ визначає місце кожного запису в таблиці і забезпечує прямий доступ до нього. Кожен запис у таблиці містить один або більше ключів. З ключами пов'язується тип виконуваних дій.

Ключ, який використовуватиметься для ідентифікації записів називають первинним. Приклад схеми таблиці показано на рис. 4.7.

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

Рис. 4.7 Схема структури даних «таблиця»

Таблиці поділяються на:

- впорядковані – таблиці, що впорядковані за атрибутом (зростання коду ключа або частотою звернення до запису);
- невпорядковані – розміщуються один за одним без пропусків.

Операції з таблицями:

- введення. Введення запису до таблиці, що складається з ключа та безпосереднього значення;
- заміна. Знаходження запису за ключем і заміна його;
- вилучення. Вилучення елемента за ключем;
- пошук. Пошук елемента за ключем.

8. Хешування

Хешування - це процес перетворення вхідних даних будь-якого розміру в фіксований розмір за допомогою хеш-функції.

Хеш-функція - функція, що приймає ключ і повертає числовий хеш-код. Хеш-функція часто застосовується для функціонування структури даних: хеш-таблиці.

Хеш-таблиця - структура даних, яка використовується для зберігання та отримання даних з використанням хеш-функції.

Існує безліч методів хешування. Серед них є:

1. Метод ділення. Цей метод використовує операцію ділення для обчислення хеш-коду. Формулою для обчислення хешу є:

$$hash\_code = key \ \% \ table\_size \quad (4.1)$$

, де *hash*_code – обчислена хеш-послідовність, key - це значення ключа, а table_size - розмір хеш-таблиці.

2. Метод множення. Використовується множення для обчислення хеш-коду. Формула має наступний вигляд:

$$hash\_code = table\_size \cdot ((key \cdot A) \ \% \ 1) \quad (4.2)$$

, де A - константа між 0 і 1(зазвичай A=0.618), може підбиратись емпірично.

3. Метод фолдування. Розділяє ключ на різні частини, а потім сумує або застосовує інші операції для отримання хеш-коду. Наприклад, можна складати деяку кількість біт ключа.

4. Метод середнього квадрату. Піднімає ключ до квадрату та використовує середній рядок отриманого значення як хеш-код.

5. Хеш-функції на основі CRC. Використовують циклічні операції для створення хеш-коду.

6. Метод XOR. Використовує операцію XOR для комбінації біт ключа та отримання хеш-коду.

7. Універсальні хеш-функції. Є спеціально розробленими для забезпечення високого рівня випадковості та ефективної роботи з різноманітними видами даних.

8. Хеш-функції на основі блочних алгоритмів: Використовують блочні алгоритми, такі як SHA-256 або MD5, для генерації хеш-коду.

9. Хеш-функції на основі HMAC. Використовують ключ для безпечного обчислення хеш-коду.

Ці, як і інші методи, можуть отримати одне і те саме значення ключа, яке називається колізією.

9. Динамічне хешування.

Динамічне хешування - метод управління динамічно змінюваною колекцією даних, що використовує хеш-таблиці(чи інші подібні) для забезпечення ефективного доступу до елементів за ключем. Основна ідея полягає в тому, щоб автоматично змінювати розмір хеш-таблиці при додаванні нових елементів або видаленні існуючих. Таким чином, це дозволяє уникнути переповнення або використання надмірної пам'яті. Таке часто відбувається при використанні статичних структур даних.

Стратегія полягає у тому, що по мірі зростання бази потрібно періодично замінювати хеш-функцію перераховуючи всі адреси, забираючи додаткові ресурси. Умовою для заміни є втрата продуктивності. Конкретні числові значення ефективності мають бути визначені експертних шляхом. При виборі та налаштуванні хеш-функції потрібно розраховувати запас.

Одним з найпоширеніших методів динамічного хешування є розширення хеш-таблиці. Під час розширення структури даних створюється нова, більша хеш-таблиця, і всі елементи переносяться в нову структуру і рехешуються урахуванням нових параметрів. Аналогічні дії можуть бути виконані і у випадку звуження хеш-таблиці.

Один з ключових викликів динамічного хешування полягає в підборі оптимальних параметрів розширення та звуження для забезпечення ефективності та продуктивності структури даних у різних сценаріях використання, оскільки вимагає експертного підходу для вирішення проблеми.

Властивості хешування відображають властивості хеш-функції. Основними властивості хеш-функцій є:

1. визначеність: Для одних і тих самих вхідних даних хеш-функція повертає завжди одне і те ж значення;
2. швидкість обчислення: Мають бути обчислюваними дуже швидко, оскільки вони часто використовуються в реальному часі;
3. фіксований розмір вихідного значення: Завжди повертає значення фіксованого розміру, незалежно від розміру вхідних даних;
4. ефективність в обчисленнях: Повинні бути ефективними з точки зору витрат обчислювальних ресурсів;
5. стійкість до колізій: зміна навіть одного біта в вхідних даних повинна призводити до істотних змін у хеш-коді. Це важливо для уникнення колізій, коли різні вхідні дані мають однаковий хеш-код.

Методи розв’язання колізій. Як зазначалось, методи хешування можуть отримати одне і те саме значення ключа, тобто одну адресу. Таке явище називають колізією. Для розв’язання цієї проблеми використовуються деякі ефективні методи. Описуються наступною схемою, показаною на рис. 4.8.

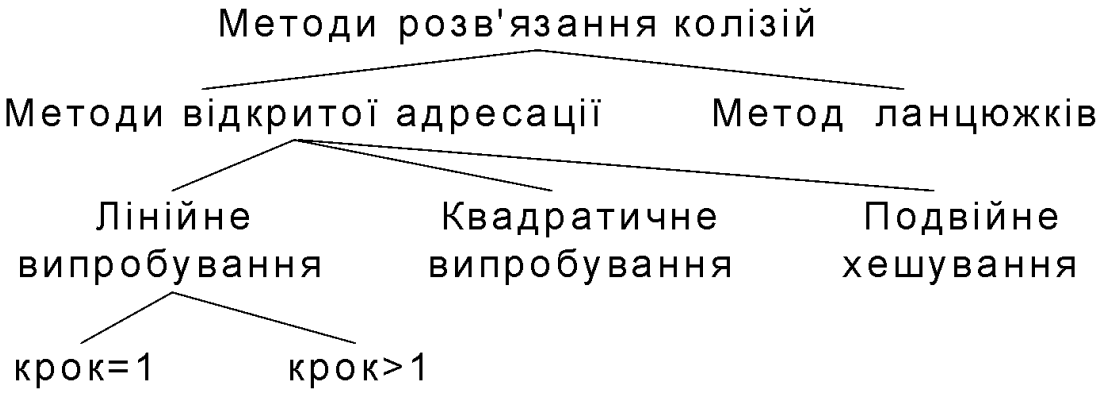


Рис. 4.8 Методи розв’язання колізії

Варто розглянути кожен метод окремо.

Метод ланцюжків - метод, в якому для розв’язання колізій у всі записи (або значення хеш-таблиці) вводяться покажчики, які використовуються для організації списків. Такий ланцюжок покажчиків нагадує однозв’язний список. У випадку виникнення колізій при заповненні таблиці в список для потрібної адреси хеш-таблиці додається ще один елемент. Схему методу ланцюжків показано на рис. 4.9.

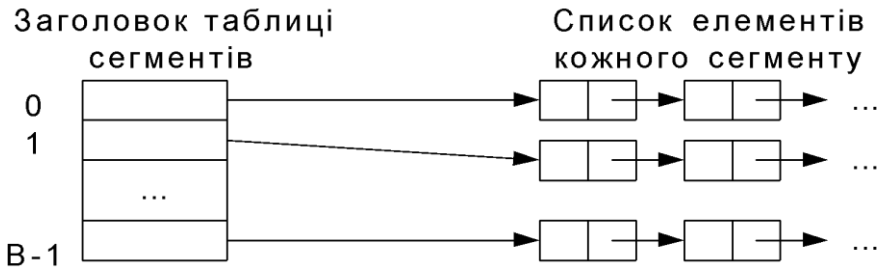


Рис. 4.9 Схема методу ланцюжків

Пошук в хеш-таблиці з ланцюжками працює за алгоритмом:

1. обчислюється адреса за значенням ключа;
2. здійснюється послідовний пошук в списку, який зв’язаний з обчисленим адресом.

Процедура вилучення з таблиці зводиться до пошуку елемента і його вилучення з ланцюжка переповнення.

Приклад простої хеш-таблиці з ланцюжками мовою C++:

```
#include <vector>
#include <list>

class HashTable {
   static const int SIZE = 10;
   std::vector<std::list<int>> table;
   int hash(int key) { return key % SIZE; }
public:
   HashTable() : table(SIZE) {}

   void insert(int key) {
      int idx = hash(key);
      table[idx].push_back(key);
   }

   bool search(int key) {
      int idx = hash(key);
      for (int x : table[idx])
         if (x == key) return true;
      return false;
   }
};
```

Така реалізація передбачає створення динамічного вектору, що містить список значень. Хешем виступає індекс, який обчислюється хеш-функцією. Метод пошуку отримує деякий ключ, хешує його і перебирає ланцюжок з метою його знаходження.

Суть **методів відкритої адресації** полягає в тому, щоб користуючись алгоритмом, що забезпечує перебір елементів таблиці, та переглядає її в пошуках вільного місця для нового запису. До них відносяться наступні методи:

1. Лінійне випробування – метод, що зводиться до послідовного перебору елементів таблиці з деяким фіксованим кроком. Формулу методу відображено у формулі 4.3.

$$a=h(key) + c*i \quad (4.3)$$

, де i – номер спроби розв'язати колізію , c – крок(константа): при кроці рівному одиниці відбувається послідовний перебір усіх елементів після поточного.

2. Квадратичне випробування відрізняється від лінійного тим, що крок перебору елементів залежить від номеру спроби знайти вільний елемент. Формула метода має наступний вигляд:

$$a = h(key2) + c*i + d*i^2 \quad (4.4)$$

, де d – крок (константа).

Таким чином, зменшується кількість спроб при великій кількості ключів-синонімів. Особливістю такої нелінійності є широкий крок, що дозволяє працювати з великими колекціями. Потрібно також враховувати, що відносно невелика кількість спроб може швидко привести до виходу за адресний простір таблиці, якщо вона є невеликою. Причиною цьому є квадратична залежність, що висвітлюється у формулі.

3. Подвійне хешування. Полягає у використанні додаткового методу хешування для обчислення нової хеш-функції. Формула має наступний вигляд.

4.

$$a=h1(key) + i*h2(key) \quad (4.4)$$

, де h1 – виклик хеш-функції основного методу хешування, h2 – виклик хеш-функції додаткового методу хешування.

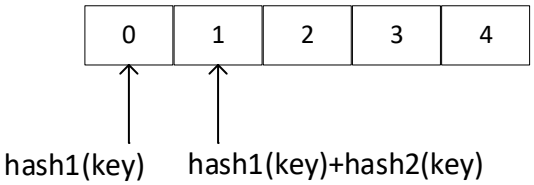


Рис 4.10. Схема подвійного хешування

На рисунку 4.10 відображено схему подвійного хешування. Принцип полягає у тому, що hash1(основна хеш-функція) обчислюється хеш (що є індексом), якщо індекс зайнятий, додатково використовується hash2(допоміжна хеш-функція) помножений на номер спроби, для пошуку нового хеша.

Вимоги хеш-функції. Хороша хеш-функція має задовольняти таким вимогами:

- обчислення має бути дуже швидким;
- має бути мінімальне число колізій.

Перша властивість залежить від особливості машини. Друга – від характеру даних.

Практичні завдання

1. **Реалізуйте двозв'язний список.** Напишіть клас для двозв'язного списку, що підтримує додавання та видалення елементів за індексом. Додайте методи для прямого та зворотного обходу списку. Порівняйте швидкодію з однозв'язним списком на типових операціях.
2. **Моделюйте кільцевий буфер.** Створіть клас або структуру для кільцевого буфера фіксованого розміру (наприклад, 10 елементів). Реалізуйте операції додавання та видалення, забезпечивши циклічність індексів. Проведіть тест: додайте у буфер більше елементів, ніж його розмір, і переконайтеся, що старі значення заміщаються новими.

Питання до самоконтролю

1. Що таке список?
2. Опишіть основні операції списку.
3. Опишіть однонаправлений список.
4. Опишіть двонаправлений список.
5. Опишіть циклічний список.
6. Що таке хешування? Що таке хеш функція?
7. Опишіть структуру даних «таблиця».
8. Які є методи хешування?
9. Які є методи вирішення колізії?
10. Які є властивості хеш-функцій?
11. Які є критерії оцінювання хеш функції?
12. Що таке динамічне хешування?

Лекція № 5

Тема лекції: Нелінійні структури даних. Продовження

План лекції

1. Графи
2. Дерева
3. Бінарні дерева
4. Кістякове дерево
5. Купа
6. Червоно-чорне дерево

1. Графи

Граф – нелінійна структура даних, що використовується для моделювання відносин між об'єктами. Є динамічною структурою даних. Бувають орієнтованим (де ребра мають напрямок) або неорієнтованим, де ребра цього напрямку не мають.

Основні компоненти графа включають:

- вершина - є об'єктом або точкою в графі. Вершини можуть мати додаткову інформацію, таку як мітки або атрибути;
- ребро - представляє зв'язок між двома вершинами. Ребра можуть мати напрямок (у орієнтованому графі) або бути ненапрямленими (у неорієнтованому графі).

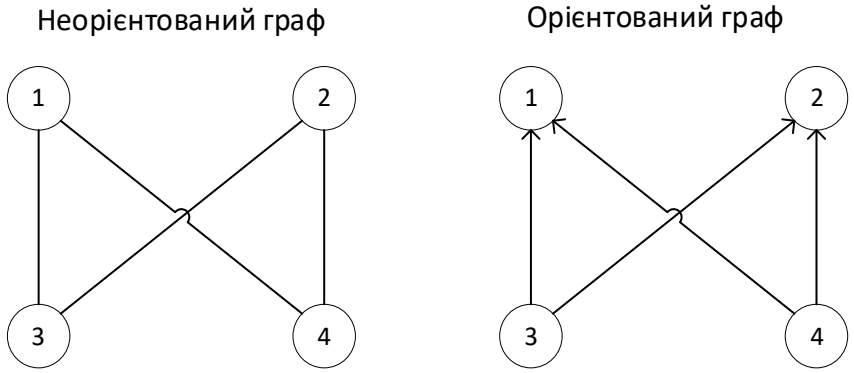


Рис. 5.1 Схеми неорієнтованого та орієнтованого графів

В ребрах графа може бути присутня характеристика ваги, що вказує на числову характеристику. Цією числовою характеристикою є **вартість ребра**, або **вага графа**.

Кількість ребер, що з'єднуються з визначеною вершиною є **ступінню вершини**. В ступені можна враховувати напрямок ребер.

Графи бувають циклічними та ациклічними. Циклічні - ті, що містять замкнений шлях, що повертається до початкової вершини. Ациклічна циклі не мають.

Від графу можна виділити частину. Така частина також є графою. Її називають **підграфом**.

Граф, в якому є шлях між будь-якою парою вершин називають **графом зв'язності**.

Метою графа є моделювання різноманітних сценаріїв, як мережі, транспортні системи, соціальні взаємодії, алгоритми пошуку і т.д.

Структуру даних можна реалізувати **матрицею** та **вказівниками**.

Графи мають поняття обходу. Це означає, що існують деякі алгоритми, що обходять всі його вершини та роблять задані обчислення. Такий обхід здійснюється з метою визначення мінімальної ваги, суми ваги, або здійснення Існує два основних алгоритми обходу - **пошук у глибину** та **пошук у ширину**. При **пошуку в глибину**, алгоритм починає з вибору початкової вершини та переходить якнайглибше вздовж одного з гілок до тих пір, поки не зустріне вершину, яку не було відвідано. При цьому використовується стек для зберігання вершин, які ще потрібно відвідати або взяти знову. При **пошуку у ширину**, алгоритм починає з вибору початкової вершини та відвідує всі вершини на одному рівні, перш ніж перейти до вершин на наступному рівні. При цьому використовує чергу для зберігання вершин, які потрібно відвідати.

Нижче приведено приклад коду обходу графу в ширину мовою C++:

```
#include <iostream>
#include <vector>
#include <queue>

void bfs(const std::vector<std::vector<int>>& adj, int start) {
    int n = adj.size();
    std::vector<bool> visited(n, false);
    std::queue<int> q;
    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int v = q.front(); q.pop();
        std::cout << v << " ";
        for (int u : adj[v]) {
            if (!visited[u]) {
                visited[u] = true;
                q.push(u);
            }
        }
    }
}

// Виклик:
int main() {
    std::vector<std::vector<int>> adj = {
        {1, 2},      // 0
        {0, 3, 4},   // 1
        {0, 5},      // 2
        {1},         // 3
        {1},         // 4
```

```
        {2}         // 5
    };
    bfs(adj, 0); // Обхід з вершини 0
}
```

Код відображає побудову графу «списком у списку» у функції main та його обхід з 0-ї вершини.

2. Дерева

Дерево – є структурою даних, що також є графом. Однак має наступні характеристики:

- такий граф зазвичай орієнтований;
- має містити елемент, на який немає жодного посилання. Його називають **коренем**;
- від кореня по певному ланцюжку є посилання на інші елементи – дочірні;
- на кожний елемент, крім кореня, є єдине посилання.

Гілка - лінія зв'язку між парою вузлів дерева. Вузли, що не мають посилаються на інші вузли дерева є **листям**. Приклад дерева показано на рис. 5.2.

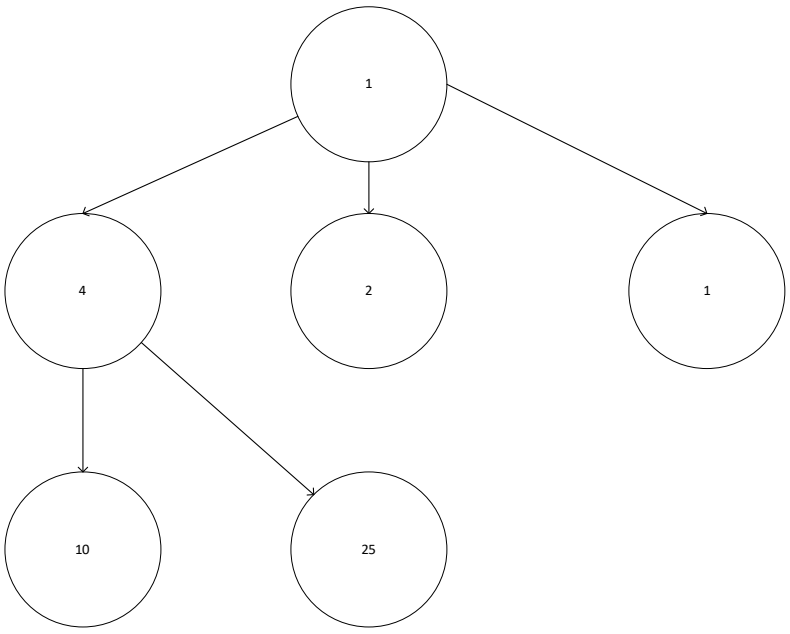


Рис. 5.2 Приклад дерева

Основні операції над дерева виділяють:

1. вставка. Додає новий вузол до дерева;
2. вилучення. Видаляє вузол з дерева;
3. пошук. Знаходить вузол з певним ключем у дереві;
4. оновлення. Змінює значення вузла у дереві.

Реалізація дерева відбувається тільки шляхом створення структури вказівників.

3. Бінарні дерева

Бінарне дерево - є деревом, де кожен вузол може мати не більше двох нащадків, які відомі як лівий та правий нащадок. Це базова структура, що має багато застосувань у програмуванні та обробці даних.

Початковим вузлом дерева, з якого розгалужуються всі інші вузли є **корінь**(або **root**). Вузли, які безпосередньо розгалужуються від даного вузла називаються **лівим і правим нащадками**. Вузли, що не мають нащадків, тобто їхні лівий та правий нащадки є порожніми є **листями**. Вузли, що мають хоча б одного нащадка є **внутрішніми вузлами. Рівнем** називають **в**ідстань від кореня до певного вузла, при цьому рівень кореня дорівнює 0. **Висотою** дерева називають максимальний рівень будь-якого вузла у дереві. Вузол, з якого виходить конкретний вузол називається **батьківським вузлом**.

Бінарне дерево, як і граф, потребує обходу. Однак, у випадку з бінарними деревами, спосіб обходу має відмінності, якщо порівнювати його і звичайним деревом. Прийнятими способами обходу бінарних дерев є:

1. префіксний обхід. Відвідування кореня перед його лівим та правим піддеревами. Порядок: корінь -> ліве піддерево -> праве піддерево;
2. інфіксний обхід. Відвідування лівого піддерева, кореня, правого піддерева. Порядок: ліве піддерево -> корінь -> праве піддерево;
3. постфіксний обхід. Відвідування лівого та правого піддерева перед коренем. Порядок: ліве піддерево -> праве піддерево -> корінь.

Приклад бінарного дерева показаний на рис. 5.3.

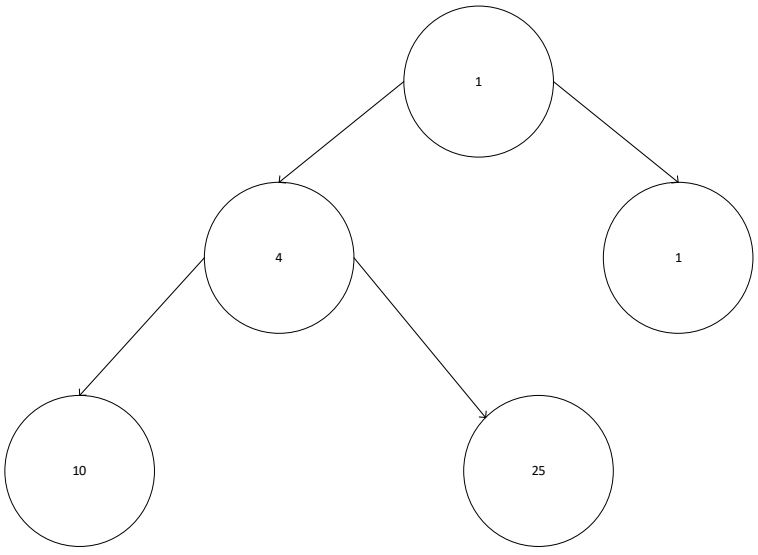


Рис. 5.3 Приклад дерева

Бінарні дерева так само можуть бути реалізовані шляхом створення вказівників, або масивом.

4. Кістякове дерево

Кістякове дерево - є підграф у графі, що містить усі вершини цього графа і є деревом (тобто зв'язний та ациклічний граф). Це означає, що кістякове дерево

включає всі вершини початкового графа, але містить лише стільки ребер, скільки необхідно, щоб зберегти зв'язність, не утворюючи циклів. Приклад кістякового дерева показаний на рис. 5.4.

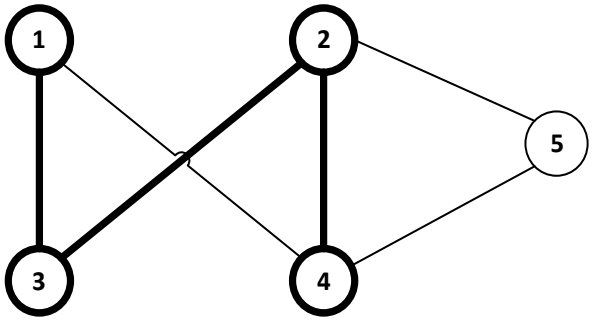


Рис. 5.4 Приклад кістякового дерева

Мінімальне кістякове дерево є кістяковим деревом з мінімальною сумарною вагою ребер у зваженому графі. Тобто, це кістякове дерево, що мінімізує суму ваг ребер. Алгоритмами для пошуку мінімального кістякового дерева є наступні алгоритми:

1. Алгоритм Крускала. Має наступний порядок:

1. сортуємо всі ребра графа за вагою;
2. поступово додаємо ребра до кістякового дерева, починаючи з найменшої ваги, уникаючи циклів;
3. використовуємо структури даних для об'єднання та знаходження наборів.

2. Алгоритм Прима. Має наступний порядок:

1. починаємо з довільної вершини і розширює кістякове дерево, додаючи найменше ребро, яке з'єднує вже побудоване дерево з новою вершиною;
2. повторюємо цей процес, поки всі вершини не будуть включені в дерево.

5. Купа

Купа (або Heap) є частково впорядкованою деревовидною структурою, де кожен вузол має значення, яке менше або рівне значенням його дочірніх вузлів (для максимальної купи) або більше або рівне значенням його дочірніх вузлів (для мінімальної купи). Таким чином, формуються умови для існування купи:

- всі шари купи, окрім останнього повинні бути заповнені;
- верхній елемент бінарного дерева/піддерева повинен бути максимальний(мінімальний);

Таку структуру даних ще часто називають пірамідою.

Одним з найбільш важливих варіантів купи є бінарна купа, де кожний вузол має не більше двох дочірніх вузлів. Бінарні купи досить ефективні та знаходять застосування в різних алгоритмах, таких як сортування купою, алгоритми пошуку найбільшого або найменшого елементу, а також в алгоритмах пошуку шляху. Приклад бінарної купи показаний на рис. 5.5.

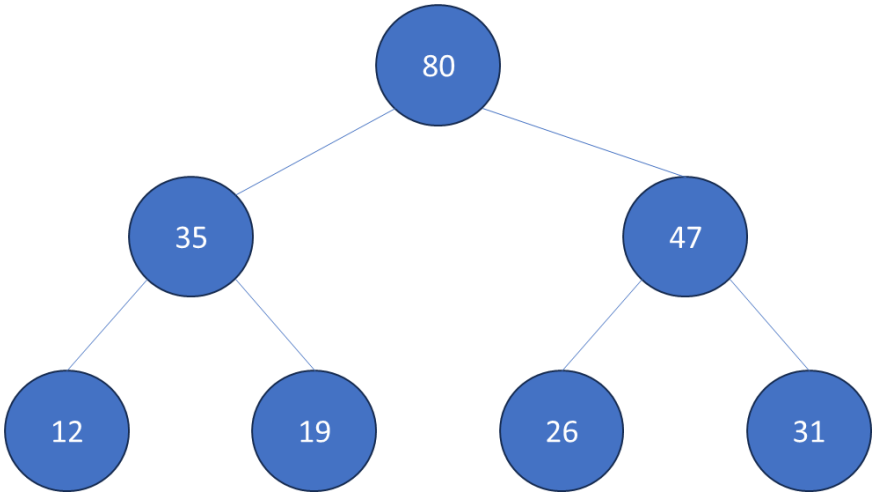


Рис. 5.5 Бінарна купа

Основними операції для реалізації купи є:

- додавання елементу до купи. При цьому мають бути дотримані наступні умови:
- елементи купи додаються зліва-направо знизу-вверх;
- дерево має задовольняти умовам існування купи.
- видалення елементу з купи. Мають бути дотримані наступні умови:
- елементи купи додаються зліва-направо знизу-вверх;
- дерево має задовольняти умовам існування купи.
- побудова купи з набору елементів;
- отримання максимального (або мінімального) елементу купи без видалення його.

Купу можна реалізувати за допомогою масивів та вказівників. Приклад представлення бінарної купи масивом можна побачити на рис. 5.6.

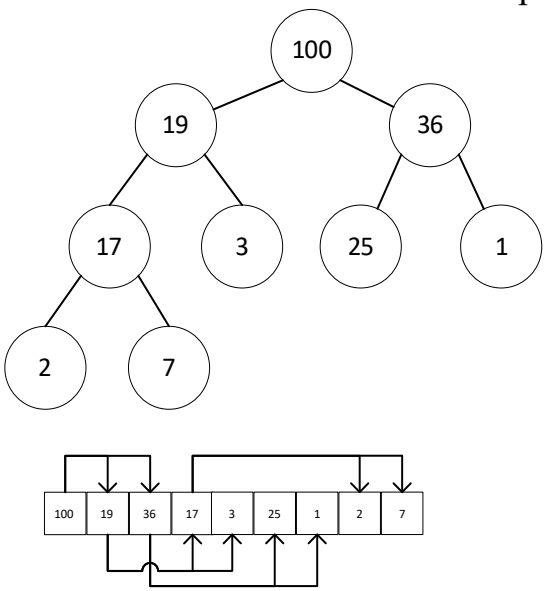


Рис. 5.6 Бінарна купа масивом

Як видно з рис. 5.6, масив сортовано від найбільшого до найменшого, а позиціонування його елементів відображає логіку побудови купи.

Пам’ять програми в купі

Після запуску будь-якого процесу операційною системою виділяє деяка квота пам'яті з розміщення даних, що відображають логіку купи. Таким чином, виділяється безперервна область пам'яті, поділена на зайняті і вільні області різного розміру. Вся необхідна інформація розміщується в пам’яті за наступними принципам купи:

- дані, що потребують більшого розміру розміщуються на початку;
- ті, що меншого розміру – в кінці.

Надалі пам'ять для купи може виділятися динамічно. Для цього існують такі операції:

- виділення пам’яті для даних методами malloc, calloc та new, що приблизно виконують такі дії:
  1. перегляд списку областей пам'яті, розміщених в купі, в пошуках вільної області відповідного розміру;
  2. якщо не вистачає вільної пам'яті процес запитує додаткової пам'яті;
  3. додавання знайденої області в список зайнятих областей (або позначення області як зайнятої);
  4. повернення посилання на початок області пам'яті;
  5. якщо виділити пам'ять не вдалося, повідомляємо про помилку.
- звільнення пам’яті. Для цього існують функції/ключові слова free, або delete. Виконують такі дії:
  1. перегляд списку областей пам'яті, розміщених в купі, в пошуках зазначеної області;
  2. видалення зі списку зайнятих областей пам'яті знайденої області;
  3. додавання знайденої області в список вільних областей (або позначення області як вільної).

Для таких мов програмування, як C#, Java і інших, впорядковувач купи буде балансувати розміром вільної та зайнятої пам’яті. У випадку з C++ основними є механізми malloc(), calloc(), free(), new, delete та ін.

Масиви розміщуються в пам’яті за принципом купи, однак принцип розміщення елементів у самому масиві у пам’яті є послідовним.

6. Червоно-чорні дерева

Червоно-чорне дерево - бінарне дерево, основною можливістю якого є самобалансування. Приклад червоно-чорного дерева показано на рис. 5.7.

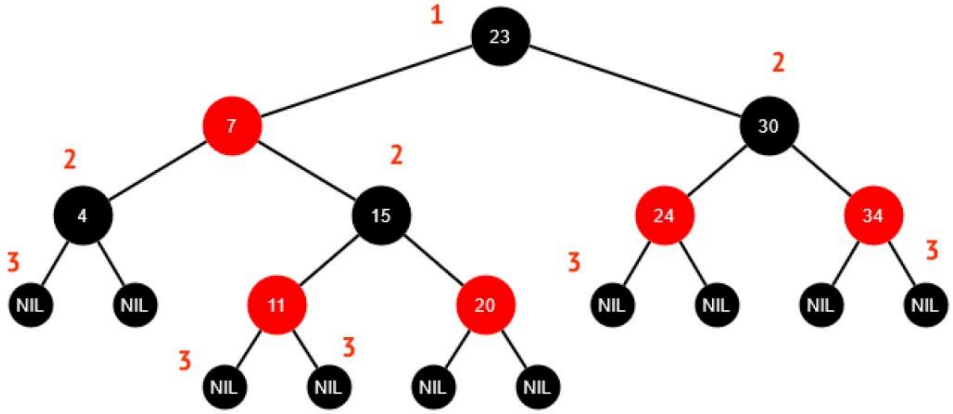


Рис. 5.7 Приклад червоно-чорного дерева

Рисунок містить червоні та чорні вузли, значення цих вузлів та чорну висоту - кількість чорних вузлів. Такій структурі властиві наступні принципи:

- кожен вузол може бути червоним, або чорним;
- корінь дерева завжди є чорним;
- всі листя, що не містять даних - чорні;
- обидва нащадки кожного червоного вузла – чорні;
- глибина в чорних вузлах однакова для будь-якого піддерева.

Дерево підтримує наступні операції:

1. пошук;
2. вставка елемента;
3. видалення елемента.

У випадку порушення властивостей червоно-чорного дерева внаслідок вставки, видалення, чи іншої допоміжної операції, необхідно виконати **перебалансування**.

Для цього є два простих підходи - перефарбування та повороти. Існують 4 види основних види розбалансування:

- лівий-лівий - left-left imbalance (LL);
- правий-правий - right-right imbalance (RR);
- лівий-правий - left-right imbalance (LR);
- правий-лівий - right-left imbalance (RL).

Їх можна побачити на рис. 5.8.

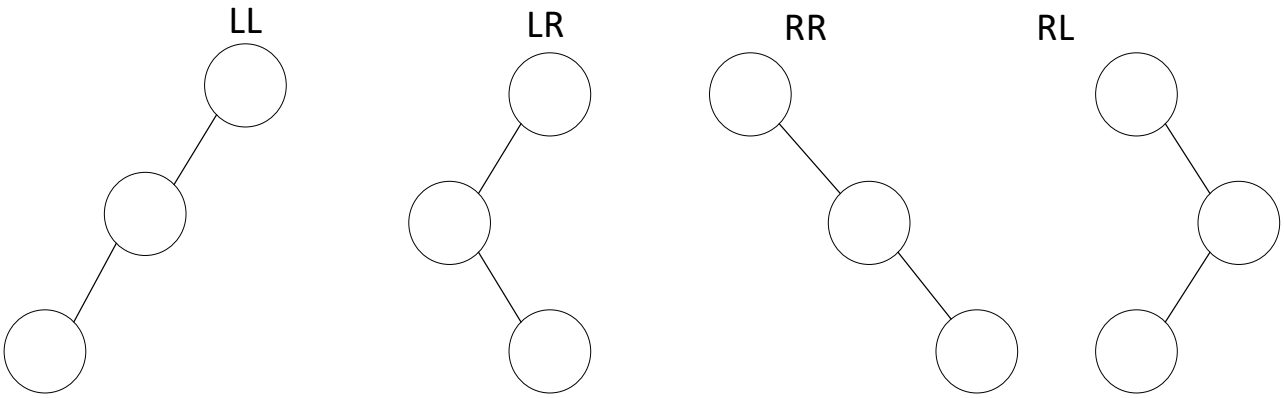


Рис. 5.8 Види розбалансування

RR, LL – лінійні стани. LR, RL – трикутні стани.

Для такого балансування 2 види ротацій:

1. У випадку з лінійними розбалансуваннями(просте балансування): потягнути перший вузол вниз, так щоб середина встала нагорі. Приклад показано на рис. 5.9.

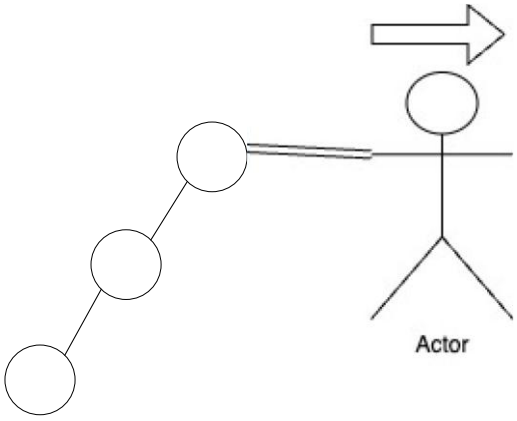


Рис. 5.9 Приклад простого балансування

При ускладненому лівому повороті, коли вузли також мають нащадків, принцип залишається тим же, однак правий нащадок вузла, який стає батьком стає лівим/правим нащадком вузла, за який ”тягнуть”. Приклад відображено на рис. 5.10.

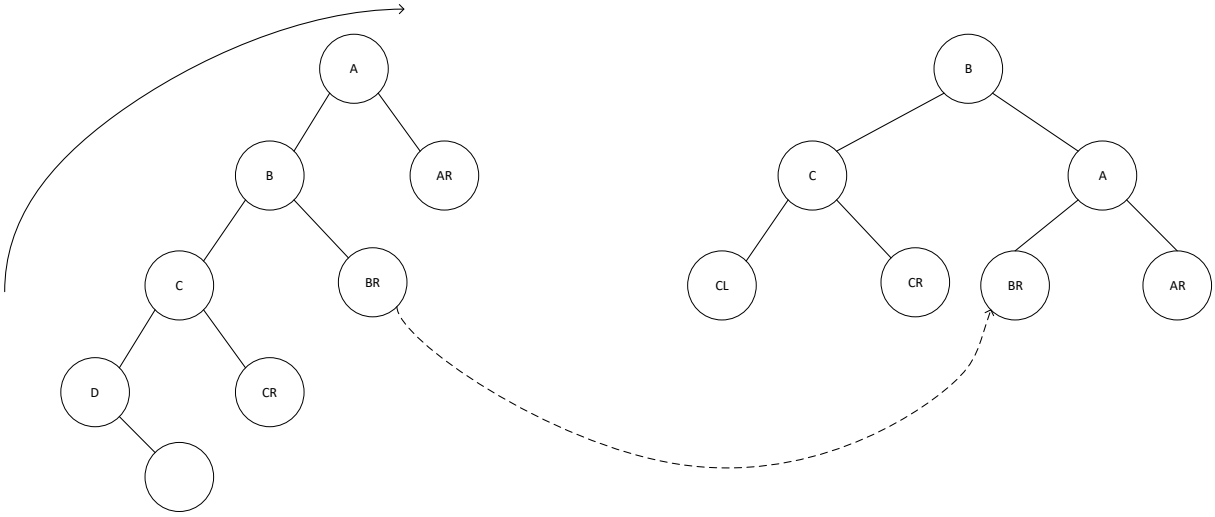


Рис. 5.10 Приклад ускладненого балансування

2. LR,RL балансування полягає в тому, щоб спочатку привести дерево до першого стану і вирішувати його, як лінійне розбалансування. У випадку ускладненої трикутної ротації(коли вузли також мають нащадків), принцип залишається тим самим, однак колишні нащадки ”нового” батька стають на місця, що звільняються, нащадків дочірніх вузлів.

Очевидно, що така ситуація має особливу стратегію вставки, що складається з двох етапів:

1. Вставити вузол і пофарбувати його в червоний.
2. Якщо вставка порушила властивості, то перефарбувати відповідні вузли і зробити повороти вузлів.

Загалом, вставка зводиться до 4 наступних сценаріїв. Для зручності, введемо наступні визначення:

- Z – елемент, що вставляється;
- Header – корінь;
- батько (або parent) – батьківський вузол;
- брат – дочірній вузол батьківського
- дід (або grand) – батьківський вузол батьківського вузла;
- дядько (або uncle) – дочірній вузол діда.

Сценарії наступні:

1. Відбувається вставка вузла Z від кореня. В такому випадку просто фарбуємо корінь у чорний. Приклад показаний на рис. 5.11.

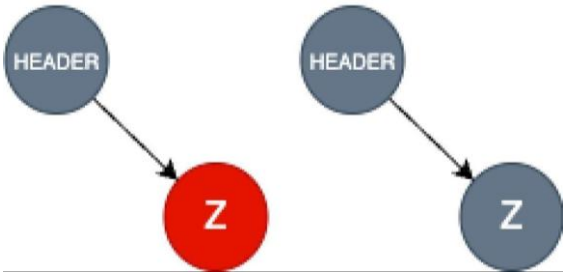


Рис. 5.11 Відображення випадку 1

На рис. Видно вузол Z, що вставляється батьківським від кореня.

2. Вставляється вузол Z, що має батька A, діда B, дядька C. При цьому вузол дядька пофарбований в червоний колір, а вузол B – не є кореневим. В такому випадку фарбуємо діда у червоний , а дядька у чорний. Якщо дід є коренем, то знову його фарбуємо у чорний. Приклад показано на рис. 5.12.

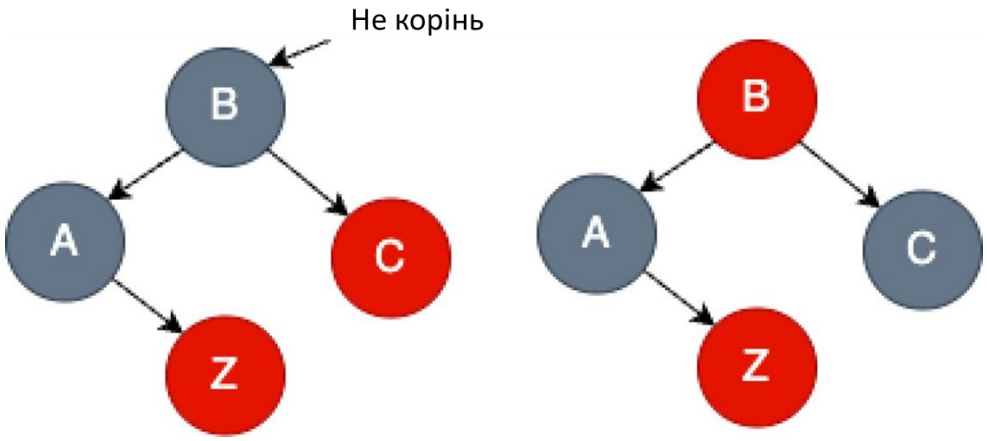


Рис. 5.12 Відображення випадку 2

3. Вставляється вузол Z, до батька A, та брата D. Також має діда B і дядька C. Червоними є Z і A. В такому випадку здійснюємо ротацію, тобто тягнемо вузол дідуся вниз. A займає місце B, а потім перефарбовуємо вузол ”колишнього” дідуся до червоного , а батька до чорного. Приклад такої послідовності дій показано на рис. 5.14.

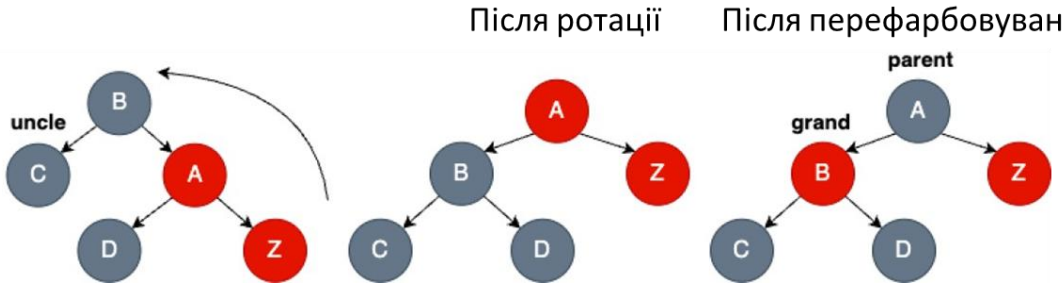


Рис. 5.14 Відображення випадку 3

4. Вставляємо елемент Z, що є лівим сином від батька A і діда B. Також маємо дядька C. Червоними є вузли Z і A. Тоді, спочатку здійснюємо ротацію за видом № 2, що призводить до стану, описаного у випадку № 3, тобто зробимо ротацію, проте дерево, як і раніше, у стані розбалансування, оскільки нащадок вузла ”Z” (вузол ”A”) є червоним . Тобто зараз вузол ”A” порушує властивості дерева і ротація проводитиметься щодо його батьківських вузлів, якими є: дід – вузол ”B”, батько – вузол ”Z”. Знову робимо ротацію, а потім перефарбовуємо ”колишнього” дідуся в червоний , а батька в **чорний** . Відображення стану показано на рис. 5.15.

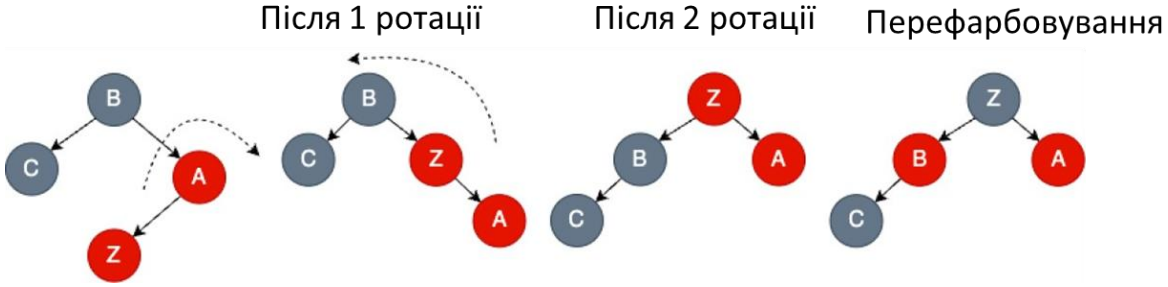


Рис. 5.15 Відображення випадку 4

Приклад реалізації спрощеної структури червоно-чорного дерева мовою C++.

```
enum Color { RED, BLACK };

struct Node {
   int data;
   Color color;
   Node *left, *right, *parent;
   Node(int val) : data(val), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}
};

class RBTree {
   Node* root;

   // Додайте функції для балансування і вставки тут...

public:
   RBTree() : root(nullptr) {}

   // Спрощена вставка без балансування
   void insert(int val) {
      Node* newNode = new Node(val);
      if (!root) {
```

```
            newNode->color = BLACK;
            root = newNode;
            return;
        }
        Node* parent = nullptr;
        Node* curr = root;
        while (curr) {
            parent = curr;
            if (val < curr->data) curr = curr->left;
            else curr = curr->right;
        }
        newNode->parent = parent;
        if (val < parent->data) parent->left = newNode;
        else parent->right = newNode;
    }
};
```

Для повноцінного червоно-чорного дерева треба реалізувати балансування та перефарбування, однак спрощена вставка елемента відображає основну суть роботи червоно-чорного дерева.

Складності нелінійних структур даних

Складності нелінійних структур даних грають не менш важливу роль у їх використання. Порівняльна таблиця, що відображає складності найбільш складних в цьому курсі нелінійних структур відображена у табл. 5.1.

Таблиця 5.1

Порівняльна таблиця складностей нелінійних структур

Тип дерева	Висота в найгіршому випадку	Час вставки/видалення
Червоно-чорне дерево	O(log n)	O(log n)
Бінарна купа (Heap)	O(log n)	O(log n)

В таблиці 5.1 приводяться оцінки складності за такими характеристиками, як збільшення складності операції при збільшення висоти дерева і час вставки, або видалення значення дерева.

Як видно, вони мають складність O(log n), що означає, що зі збільшенням кількості даних, вони будуть працювати оптимізовано.

Практичні завдання

1. Реалізуйте обхід графа в ширину і в глибину на списках суміжності. Реалізуйте обидва алгоритми для одного графа (можна неорієнтований).

Виведіть порядок відвідування вершин для кожного обходу. Поясніть, у чому різниця між алгоритмами на конкретному прикладі.

2. Дослідіть глибину бінарного дерева пошуку при різних послідовностях вставки. Згенеруйте кілька послідовностей чисел. Побудуйте дерево для кожної послідовності, виміряйте його висоту (глибину). Порівняйте результати. Зробіть висновок: як порядок вставки впливає на ефективність дерева?

Питання до самоконтролю

1. Що таке нелінійна структура даних? Як структури даних Ви знаєте?
2. Опишіть структуру граф.
3. Які способи обходу графів Ви знаєте?
4. Опишіть структуру дерево.
5. Опишіть структуру бінарне дерево.
6. Опишіть варіанти обходу бінарного дерева.
7. Що таке кістякове дерево? Що таке мінімальне кістякове дерево?
8. Які алгоритми є для пошуку мінімального кістякового дерева? Опишіть їх.
9. Що таке купа?
10. Що таке червоно-чорне дерево? Які операції є о червоно-чорному дереві?
11. Як розміщаються змінні в пам’яті програми? Які є функції для управління пам’яттю?
12. Як розміщаються елементи масиву у пам’яті програми?

Лекція № 6

Тема лекції: Алгоритми сортування. Рекурсивні алгоритми. Евристичні алгоритми.

План лекції

1. Алгоритми сортування.
2. Рекурсивні алгоритми.
3. Евристичні алгоритми.

1. Алгоритми сортування

Основна суть алгоритмів сортування полягають у організації елементів в певному порядку. Основним цілями використання алгоритмів сортування є:

- пошук;
- оптимізація операцій та ефективності алгоритмів;
- попередня підготовка даних для аналізу;
- покращення ефективності взаємодії користувача з даними.

Методи сортування діляться на наступні групи:

1. Сортування лінійних структур:
   - 1.1. вставкою, тобто послідовного вставлення елементів у відсортовану частину масиву:
     - 1.1.1. сортування включенням;
     - 1.1.2. сортування Шелла;
   - 1.2. вибором, тобто такий, що базується на ідеї послідовного вибору найменшого (або найбільшого) елемента із несортованої частини масиву та обміну його з першим елементом у цій частині:
     - 1.2.1. метод простої вибірки;
     - 1.2.2. швидкий метод сортування;
   - 1.3. обміном, тобто в такому випадку елементи масиву порівнюються між собою і обмінюються місцями, якщо вони не відповідають потрібному порядку сортування:
     - 1.3.1. метод бульбашки;
   - 1.4. інші:
     - 1.4.1. порозрядне сортування.
2. Сортування з використанням нелінійних структур:
   - турнірне сортування;
   - пірамідальне сортування.

Кожен з цих алгоритмів сортування має бути розглянутий окремо.

Метод простої вибірки

Метод простої вибірки (англ. Selection Sort) - алгоритм сортування вибором. Основним принцип полягає у тому, що він послідовно обирає мінімальний елемент із залишених невідсортованих елементів і обмінює його з першим елементом у відсортованій частині масиву. Процес повторюється до тих пір, поки весь масив не стане відсортованим.

Основні кроки алгоритму простої вибірки є наступними:

1. Визначаємо вхідний масив і позначаємо початок невідсортованої частини як усі елементи;
2. Проходимо невідсортовану частину масиву і знаходимо мінімальний елемент.
3. Міняємо знайдений мінімальний елемент з першим елементом у невідсортованій частині.
4. Зменшуємо розмір невідсортованої частини, виключаючи вже відсортований елемент.
5. Повторюємо цей процес, поки не залишиться лише один елемент у невідсортованій частині.

Алгоритм завершується, коли весь масив відсортований.

Часова складність - $O(n^2)$, де n - кількість елементів у масиві. Такий метод є неефективним для великих масивів, однак простий для реалізації та може бути ефективним для невеликих обсягів даних або вже відсортованих масивів.

Приклад роботи алгоритму показаний на рис. 6.1

6	15	4	20	7	13	0
6	15	4	20	7	13	0
0	15	4	20	7	13	6
0	4	15	20	7	13	6
0	4	6	20	7	13	15
0	4	6	7	20	13	15
0	4	6	7	13	20	15
0	4	6	7	13	15	20

Рис. 6.1 Приклад роботи алгоритму

На схемі показаний наочний приклад сортування. Синім виділено вибраний елемент, червоним – елемент який потрібно поміняти з синім. Зелений елемент є вже відсортованим.

Метод бульбашкового сортування

Метод бульбашкового сортування (англ. Bubble Sort) - алгоритм сортування обміном. Основний принцип полягає в порівнянні сусідніх елементів і обмінюванні їх, у випадку неправильного порядку. Процес повторюється до тих пір, поки не буде відсортований весь масив.

Основні кроки алгоритму бульбашкового сортування наступні:

1. Визначаємо вхідний масив і позначаємо його як невідсортований.
2. Починаємо порівнювати сусідні елементи починаючи з першого і до останнього.

3. Якщо два сусідні елементи розташовані у неправильному порядку, то міняємо їх місцями.

4. Повторюємо цей процес для всього масиву, працюючи з початку до кінця.

Ітерації продовжуються, поки не буде діяти умова, що жоден елемент не потребує обміну під час повної ітерації.

Алгоритм має часову складність $O(n^2)$, де n - кількість елементів у масиві. Такий метод не є ефективним для великих масивів, але він простий для реалізації та може бути ефективним для невеликих обсягів даних або вже відсортованих масивів.

Приклад роботи масиву показано на рис. 6.2.

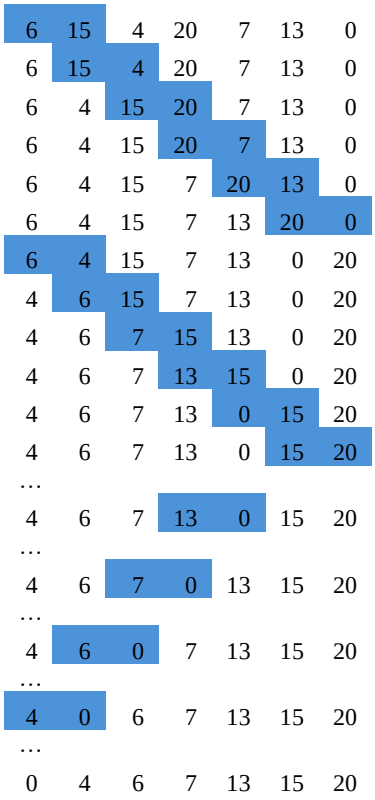


Рис. 6.2 Приклад роботи алгоритму

На схемі показаний наочний приклад сортування. Синім виділяються ті елементи, що аналізуються.

Швидкісне сортування

Швидкісне сортування (або алгоритм Хоара) (англ. Quicksort) - алгоритм сортування, що використовує стратегію "розділяй і володарюй". Основний принцип полягає в розбитті задачі на підзадачі і вирішенні окремої з них.

Основна ідея алгоритму полягає в тому, щоб обрати один елемент масиву, відомий як "опорний елемент", і розмістити його на відповідному місці так, щоб всі елементи, які менше за опорний, знаходилися ліворуч від нього, а всі, що більше - праворуч. Після цього опорний елемент вже не бере участі в подальших

порівняннях. Процес розподілу (розділення) повторюється для кожної з отриманих підмасивів.

Основні етапи швидкісного сортування:

1. З масиву обирається опорний елемент.
2. Масив розбивається так, що елементи менше опорного розміщуються ліворуч, більше - праворуч.
3. Застосовується швидке сортування рекурсивно до лівого та правого підмасивів.
4. Опорний елемент маркується, як відсортований, а тому не включається до подальших порівнянь.

Алгоритм продовжується, поки не буде відсортований весь масив.

Часова складність - O(n*log(n)), де n - розмір вхідного масиву. Через використання рекурсії та розділення масиву на алгоритм є ефективним для великих масивів. Особливо, якщо опорний елемент обирається оптимально. Однак, при невдалому виборі опорного елемента, ефективність знижується і алгоритм починає працювати неефективно у порівнянні з іншими.

Приклад роботи алгоритму показано на рис. 6.3.

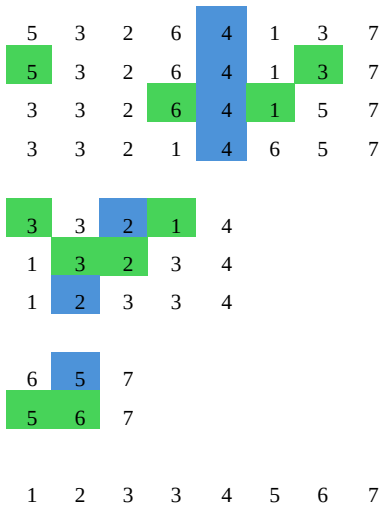


Рис. 6.3 Приклад роботи алгоритму

Синім вибрано опорний елемент, зеленим – ті елементи, що обмінюються.

Сортування включенням

Сортування включенням (англ. Insertion Sort) - простий алгоритм сортування вставками. Подібний до того, як люди впорядковують гральні карти в руках. Основний принцип полягає у поступовому включенні елементів в вже відсортовану частину масиву.

Основні кроки алгоритму наступні:

1. перший елемент масиву маркується відсортованим;
2. наступний елемент включається в відсортовану частину, порівнюючи його з кожним елементом у відсортованій частині та переміщуючи його на відповідне місце.

Процес повторюється для кожного елементу, поки не пройдемо весь масив.

Приклад показано на рис. 6.4.

44	55	12	42	94	18	6	47
12	44	55	42	94	18	6	47
12	42	44	55	94	18	6	47
12	42	44	55	18	6	47	94
12	18	42	44	55	6	47	94
6	12	18	42	44	55	47	94
6	12	18	42	44	47	55	94
6	12	18	42	44	47	55	94

Рис. 6.4 Приклад роботи алгоритму

Різними кольорами позначені ті елементи, як такі, що потрібно перемістити.

Сортування Шелла

Сортування Шелла (англ. Shell Sort) є вдосконаленням сортування включенням. Використовує "інкременти" або "кроки", для поетапногл сортувати елементи масиву. Суть полягає в тому, щоб зменшувати відстань між елементами, що порівнюються і використовуати алгоритм вставок для кожної групи елементів. Такий підхід пришвидшує вставку менших елементів на правильні позиції.

Алгоритм має наступні кроки:

1. Вибір кроків, які будуть визначати, які елементи масиву порівнюються між собою. Поширеними послідовностями кроків є n/2,n/4,n/8,…,1*n*/2,*n*/4,*n*/8,…,1.
2. Цей крок використовується для сортування вставками для окремих груп елементів масиву. Кожен елемент в своїй групі порівнюється з іншими та вставляється на відповідне місце.
3. Крок зменшується, і процес повторюється до тих пір, поки крок не стане рівним 1.

Приклад роботи алгоритму показано на рис. 6.5.

	44	55	12	42	94	18	6	47
h=4	44	55	12	42	94	18	6	47
	44	18	6	42	94	55	12	47
h=2	44	18	6	42	94	55	12	47
	6	18	12	42	44	47	94	55
Сортування вставками:								
h=1	6	18	12	42	44	47	94	55
	6	12	18	42	44	47	55	94

Рис. 6.5 Приклад роботи алгоритму

Різними кольорами позначені пари, що вибрані для обміну.

Сортування злиттям

Сортування злиттям - алгоритм сортування, в основі якого лежить принцип "розділу і володарювання". Основна принцип полягає в поділі масиву

на дві рівні частини, рекурсивному сортуванні кожної з них, з подальшим об'єднанням.

Основні кроки:

1. розділення. Масив розбивається на дві частини навпіл, поки не досягться масиву з одним елементом;
2. злиття. Відсортовані підмасиви об'єднуються так, щоб отримати один відсортований масив шляхом порівняння елементів двох підмасивів та їх об'єднання в новий масив.

Складність даного алгоритму - O(n*log(n)), де n - розмір масиву.

Приклад роботи алгоритму показано на рис. 6.6.

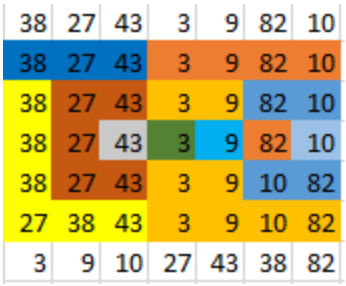


Рис. 6.6 Приклад роботи алгоритму

Різні кольори показують покрокове розділення масиву на частини.

Порозрядне сортування

Порозрядне сортування - алгоритм сортування, що сортує елементи за розрядами.

Процес сортування включає наступні кроки:

1. Визначається найбільша кількість розрядів у найбільшого числа масиву. Це визначає кількість ітерацій сортування.
2. Створюється алфавіт за символами/цифрами, що присутні в масиві.
3. Починаючи з менш значущого розряду і закінчуючи найбільш значущим, елементи сортуються за їхніми розрядами.
4. Після сортування за всіма розрядами результати складаються послідовно, утворюючи відсортований масив.

Має часову складність O(n*k), де n - кількість елементів у масиві, k - кількість розрядів.

Сортування двійковим деревом

Сортування двійковим деревом - алгоритм сортування, що використовує бінарне дерево пошуку для впорядкування елементів.

Процес сортування має наступні кроки:

1. Створення дерева.
2. Кожен елемент у вхідному масиві вставляємо у відповідне місце за допомогою стандартного алгоритму вставки для двійкового дерева пошуку. Після вставки кожен вузол дерева забезпечується властивістю: значення всіх вузлів у лівому піддереві менше за значення даного вузла, а значення всіх вузлів у правому піддереві більше або рівне значенню даного вузла.

3. Після вставки всіх елементів виконується обхід дерева.

Часова складність - O(n*log(n)), що робить його менш ефективним за інші алгоритми.

Турнірне сортування

Турнірне сортування (англ. Tournament Sort) - алгоритм сортування, що базується на ідеї змагальних турнірів. Основний принцип полягає у використанні структури даних "турнірнє дерево" для визначення найменшого елемента у несортованому масиві. Суть полягає у «проведенні змагань», виведенні найменшого елемента і обміні його з першим елементом масиву. Цей повторюється процес поки всі елементи не будуть проведені через дерево.

Процес сортування можна описати наступним чином:

1. кожен елемент масиву розглядається як листок турнірного дерева. У вершині кожного рівня обирається менший з двох синів і переміщується вище по дереву. Процес повторюється, поки не буде визначено переможця - найменшого елемента у несортованому масиві.
2. Переможець обмінюється з першим елементом масиву. Таким чином, найменший елемент масиву опиняється на своєму місці.
3. Залишок масиву розглядається як новий масив, і процес побудови турнірного дерева та обміну з коренем повторюється до тих пір, поки всі елементи не будуть відсортовані.

Часова складність - O(n*log(n)), де n - кількість елементів у масиві. Цей алгоритм є ефективним, якщо масив вже відсортований або майже відсортований.

Пірамідальне сортування

Пірамідальне сортування (англ. Heap Sort) - алгоритм сортування, що використовує структуру даних купи для впорядкування елементів у вихідному масиві.

Основний принцип полягає у побудові максимальної купи з вихідного масиву. Далі, найбільший елемент (корінь купи) переноситься в кінець масиву. На його місце ставиться правий нижній елемент. Ці кроки повторюються до моменту, поки купа не буде пустою.

Основні кроки:

1. з масиву створюється максимальна купа;
2. найбільший елемент розміщується в кінці відсортованої частини масиву. Після цього купа знову балансується;
3. Кроки 1 і 2 повторюються до моменту, поки купа не буде пустою.

Складність алгоритму - O(n*log(n)). Це робить його ефективним алгоритмом для сортування великих наборів даних та менш ефективним за інші алгоритми.

2. Рекурсивні алгоритми

Рекурсивний алгоритм - алгоритм, що викликає сам себе для вирішення задачі. Зазвичай використовується для вирішення проблем, що можуть бути розкладені на менші аналогічні проблеми.

Прикладом рекурсивного алгоритму є:

1. факторіал числа. Наприклад: 5! = 5 * 4 * 3 * 2 * 1 = 120;
2. сума чисел у масиві;
3. ряд Фібоначчі: кожне наступне число – сума попередніх чисел: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34;
4. обчислення суми цифр числа: сума=сума(n%10 * сума(n/10));
5. обернення рядка;
6. обчислення ступеню числа;
7. перевірка на паліндром;
8. пошук найбільшого спільного дільника двох чисел;
9. обчислення суми елементів бінарного дерева.

Існує багато прикладів використання рекурсивних алгоритмів, в тому числі і для вирішення реальних задач. Зазвичай, використовуються внаслідок такої декомпозиції задач, що передбачає собою виконання, коли більші значення зменшуються до мінімальних у якості вхідних даних.

Приклад коду, що використовує та не використовує рекурсію мовою C++:

```
// Ітеративна реалізація
int factorialIter(int n) {
   int res = 1;
   for(int i = 2; i <= n; ++i)
      res *= i;
   return res;
}

// Рекурсивна реалізація
int factorialRec(int n) {
   if (n <= 1) return 1;
   return n * factorialRec(n - 1);
}
```

Код демонструє приклад видобутку факторіалу числа шляхом двох підходів – ітеративний та рекурсивний. У випадку ітеративного підходу застосовується цикл. Рекурсивний підхід передбачає виклик функції сам себе. Такий підхід є більш оптимізованим, оскільки не вимагає додаткових змінних, однак більш небезпечним (у випадку, якщо умова виходу з рекурсії буде невірною) та складним у реалізації.

3. Евристичні алгоритми

Евристичні алгоритми є окремим видом алгоритмів, що використовують спрощені стратегії або правила, щоб наблизитися до оптимального рішення в області, де не завжди можливо знайти точне рішення за прийнятним часом. Зазвичай, є менш точними, але ефективніші за допомогою обмежених ресурсів.

Основними рисами евристичних алгоритмів:

- швидкість: спрощують задачу, щоб прискорити процес прийняття рішення;
- недетермінованість: можуть отримувати різні результати для одних і тих же вхідних даних при різних умовах;
- не гарантована оптимальність: можуть наближатися до оптимального рішення, але не можуть гарантувати його досягнення.
- використання досвіду: часто використовують інтуїцію, експертний досвід або емпіричні спостереження для прийняття рішень.

Прикладами евристичних алгоритмів є:

1. **жадібні алгоритм** – тип алгоритмів, що вирішує проблему шляхом вибору кожного кроку так, щоб забезпечити найбільший моментальний виграш, не беручи до уваги майбутні наслідки;
2. методи випадкового пошуку - вирішують проблему, шукаючи рішення випадковим чином у просторі можливих рішень;
3. методи вибору найближчого сусіда - вирішує проблему, вибираючи найближчого сусіда на кожному кроці;
4. прогнозуючі алгоритми.

Є доволі розповсюдженим видом алгоритмів для вирішення задач.

Практичні завдання

1. Реалізуйте метод простої вибірки.
2. Реалізуйте метод бульбашки.
3. Реалізуйте швидкий метод сортування.
4. Реалізуйте сортування включенням.
5. Реалізуйте сортування Шелла.
6. Реалізуйте сортування злиттям.
7. Реалізуйте порозрядне сортування.
8. Поясніть різницю між ітеративною та рекурсивною реалізацією алгоритмів на прикладі обчислення факторіалу. Вкажіть переваги й недоліки кожного підходу щодо часу виконання, використання пам’яті й зручності кодування.

Контрольні запитання

1. Які цілі у алгоритмів сортування? На які групи діляться алгоритми сортування?
2. Що таке рекурсивні алгоритми?
3. Опишіть алгоритм роботи метод простої вибірки.
4. Опишіть алгоритм роботи методу бульбашки.
5. Опишіть алгоритм роботи швидкого методу сортування.
6. Опишіть алгоритм роботи включенням.
7. Опишіть алгоритм роботи Шелла.
8. Опишіть алгоритм роботи злиттям.
9. Опишіть алгоритм роботи порозрядного сортування.
10. Опишіть алгоритм роботи турнірного сортування.
11. Опишіть алгоритм роботи пірамідального сортування.

Лекція № 7

Тема лекції: Алгоритми сортування. Рекурсивні алгоритми. Евристичні алгоритми.

План лекції

1. Алгоритми пошуку.
2. Динамічне програмування.

1. Алгоритми пошуку

Метою алгоритмів пошуку є знаходження конкретного елементу (частини тексту) в наборі даних або тексті. Цілі алгоритмів пошуку можуть бути різноманітними, залежно від контексту використання. Основні цілі включають знаходження першого входження текст в рядок, знаходження всіх входжень тексту в рядок, знаходження кількості входжень тексту в рядок, пошук за умовою, пошук найменшого/найбільшого елемента, пошук елементів, задовольняючи певну властивість, пошук найближчого значення, тощо. Загалом, застосовується великою кількість різних інструментів, і операційної системи, і прикладних програм.

Лінійний пошук

Лінійний пошук (або послідовний пошук) (англ. Linear Search) — найпростіший алгоритм пошуку, суть якого полягає у перегляданні кожного елементу до знаходження шуканого значення (чи до кінця масиву).

Алгоритм лінійного пошуку можна описати наступним чином:

1. Починаючи з початку масиву, порівнюємо значення кожного елемента з шуканим значенням.
2. Якщо поточний елемент рівний значенню шуканого, алгоритм завершує свою роботу, повідомляючи про знайдений елемент.
3. Якщо досягнуто кінець масиву і шукане значення не знайдено, алгоритм завершується, повідомляючи, що елемент не знайдено.

Складність алгоритму - O(n), де n – кількість елементів масиву. Застосовується у випадках, коли не важливо, чи список відсортований, або коли відомо, що шуканий елемент розташований близько до початку списку.
Приклад роботи алгоритму показаний на рис. 7.1

Рис. 7.1 Приклад роботи алгоритму

Зеленим відображається поточний елемент, що порівнюється з шуканим. Послідовним перебором знаходимо шукане значення.

Бінарний пошук

Бінарний пошук є алгоритмом пошуку, основна ідея якого полягає у порівнянні шуканого значення з середнім елементом масиву та виключенням тієї половини, в якій значення гарантовано не знайдеться. Виходячи з такого трактування, очевидним є обов'язкова передумова – масив має бути відсортований.

Алгоритм бінарного пошуку можна описати наступним чином:

1. Визначаємо середній елемент масиву.
2. Порівнюємо шукане значення з середнім елементом.
3. Якщо шукане значення співпадає з середнім елементом, то пошук завершено.
4. Якщо шукане значення менше, ніж середній елемент, то виключаємо праву половину масиву і повторюємо пошук в лівій половині.
5. Якщо шукане значення більше, ніж середній елемент, то виключаємо ліву половину масиву і повторюємо пошук в правій половині.
6. Продовжуємо цей процес, ділячи масив навпіл, доки шукане значення не буде знайдено або поки довжина підмасиву не стане нульовою.

Приклад роботи алгоритму показано на рис. 7.2.

Рис. 7.2 Приклад роботи алгоритму

Зелений відображається поточний елемент, що порівнюється з шуканим. Ділячи масив по середині знаходимо шукане значення.

Приклад такого коду мовою C++:

```
int binarySearch(const int* arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid; // знайдено, повертаємо індекс
        if (arr[mid] < target)
            left = mid + 1;
        else
```

```
            right = mid - 1;
        }
        return -1; // не знайдено
    }

    // Приклад використання:
    // int* v = new int[]{1, 4, 7, 10, 14, 19, 24};
    // int pos = binarySearch(v, 14); // поверне 4
```

Наведений приклад відображає метод, що ділить масив на половини і шукає значення/половину масиву, за рахунок якої може «звужувати пошук».

Складність алгоритму - O(log n). Це означає, що він є дуже ефективним для відсортованих масивів, оскільки може швидко знаходити шуканий елемент, роблячи логарифмічну кількість порівнянь.

Інтерполяційний пошук

Інтерполяційний пошук - алгоритм пошуку, що застосовується для знаходження елемента в відсортованому масиві даних. Основною суттю інтерполяційного пошуку є використання методу інтерполяції для знаходження позиції елемента. Передумово для пошуку є відсортований масив.

Складність алгоритму – O(n), де n – кількість елементів в масиві

Основні кроки інтерполяційного пошуку:

1. Спочатку визначається приблизне місцезнаходження елемента за допомогою формули інтерполяції. Формула лінійної інтерполяції:

$$i(K)=((K-Kl)*(l-r))/(K[l]-K[r])+l,$$

де $i(K)$– шуканий індекс елементу K, l – індекс початкового елементу масиву, r – індекс кінцевого елементу масиву, $K[l]$ - значення елементу за індексом l, $K[r]$ - значення елементу за індексом r.

2. Перевіряється значення елемента в зазначеній позиції. Якщо значення елемента співпадає з шуканим, то пошук завершується успішно. Якщо значення менше за шукане, то пошук продовжується в правій половині масиву, інакше - в лівій половині.

3. Процес пошуку в підмасиві повторюється до тих пір, поки не буде знайдений шуканий елемент, або поки масив не буде повністю просканований.

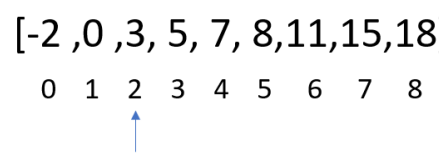
Задача:

шукаємо елемент зі значенням 5, по всьому масиву, відповідно:
$l = 0, r = 8, K[l] = -2, K[r] = 18, K = 5$

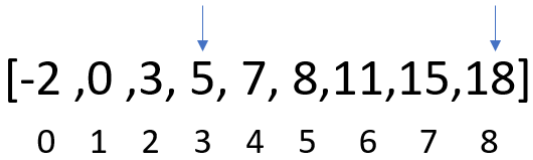
Підставляємо в формулу значення:

$i(5) = \frac{(5-(-2))*(0-8)}{(-2)-18} + 0,$

$i(5) = 2.8 \sim 2;$



K[2]=3, 3<5 – відповідно звужуємо зону пошуку зліва, а отже $l = 2 + 1 = 3.$

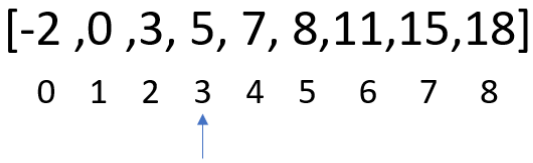


шукаємо елемент зі значенням 5, по всьому масиву, відповідно:
$l = 3, r = 8, K[l] = 5, K[r] = 18, K = 5$

Підставляємо в формулу значення:

$i(5) = \frac{(5-5)*(3-8)}{5-18} + 3,$

$i(5) = 3;$



K[3]=5, 5=5 – шуканий елемент знайдено.

Описана задача наочно показує роботу методу сортування інтерполяцією.

Прямий пошук рядка

Прямий пошук рядка - алгоритм пошуку, основною суттю якого є послідовний перегляд кожного символу у вихідному рядку з метою порівняння з кожним символом у шуканому рядку.

Основні кроки алгоритму прямого пошуку рядка:

1. Починаємо із першого символу вихідного рядка і порівнюємо його з першим символом шуканого рядка.
2. Якщо символи співпадають, переходимо до наступного символу в обох рядках і продовжуємо порівняння.
3. Якщо знайдено часткове співпадіння, але рядок не знайдено повністю, переходимо до наступного символу вихідного рядка і повторюємо пошук.

4. Якщо співпадіння знайдено для кожного символу шуканого рядка, оголошуємо, що рядок знайдено.

Складність алгоритму - O(m * n), де m - довжина шуканого рядка, а n - довжина тексту, в якому відбувається пошук. Найгірший випадок для алгоритму виникає, коли шуканий рядок знаходиться в кінці тексту або взагалі відсутній у тексті, і тоді кількість порівнянь стає максимальною. Приклад показано на рис. 7.3.

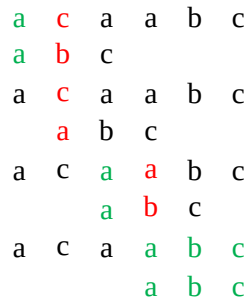


Рис. 7.3 Приклад алгоритму прямого пошуку рядка

Зеленим кольором показані співпадіння еталонного символу з шуканим. Червоним – перше не співпадіння.

Алгоритм Кнута-Морріса-Прата

Алгоритм Кнута-Морріса-Прата - алгоритм для пошуку, основною ідеєю якого є уникнення повторних порівнянь у випадку неспівпадіння між символами підрядка.

Алгоритм має наступні кроки:

1. Побудова префікс-функції. Префікс-функція визначається для кожного підрядка та вказує, на якому місці слід продовжити порівняння при виявленні неспівпадіння.
2. Створення таблиці префіксів. Для кожного елемента підрядка обчислюється значення префікс-функції. Це допомагає алгоритму "скочити" через частину тексту, яка вже була порівняна, якщо виявляється неспівпадіння.
3. Пошук підрядка в тексті за допомогою префікс-функції шляхом проходження по тексту, використовуючи відповідні значення префікс-функції для визначення місця, з якого слід продовжити порівняння при неспівпадінні.

Префікс-функція має наступний алгоритм:

1. Перший елемент таблиці префіксів завжди рівний 0.
2. Для кожного елемента таблиці префіксів обчислюється, яка довжина максимального збігу префікса та суфікса для підрядка, що закінчується на даному положенні.
3. Значення оновлюються, якщо виявляється новий максимальний збіг префікса та суфікса.

Складність алгоритму - O(m + n), де m - довжина шуканого рядка, а n - довжина тексту, в якому відбувається пошук. Є доволі ефективним у випадку великої кількості даних.

Приклад алгоритму показано на рис. 7.4.

Рис. 7.4 Приклад роботи алгоритму

Дана схема відображає логіку визначення префіксів: обчислюється кількість попередніх входжень вибраного символу (виділено жовтим), та безпосередній пошук строки: кількість входжень відображає кількість індексів, через які можна «перестрибнути» при пошуку.

Алгоритм Робіна-Карпа

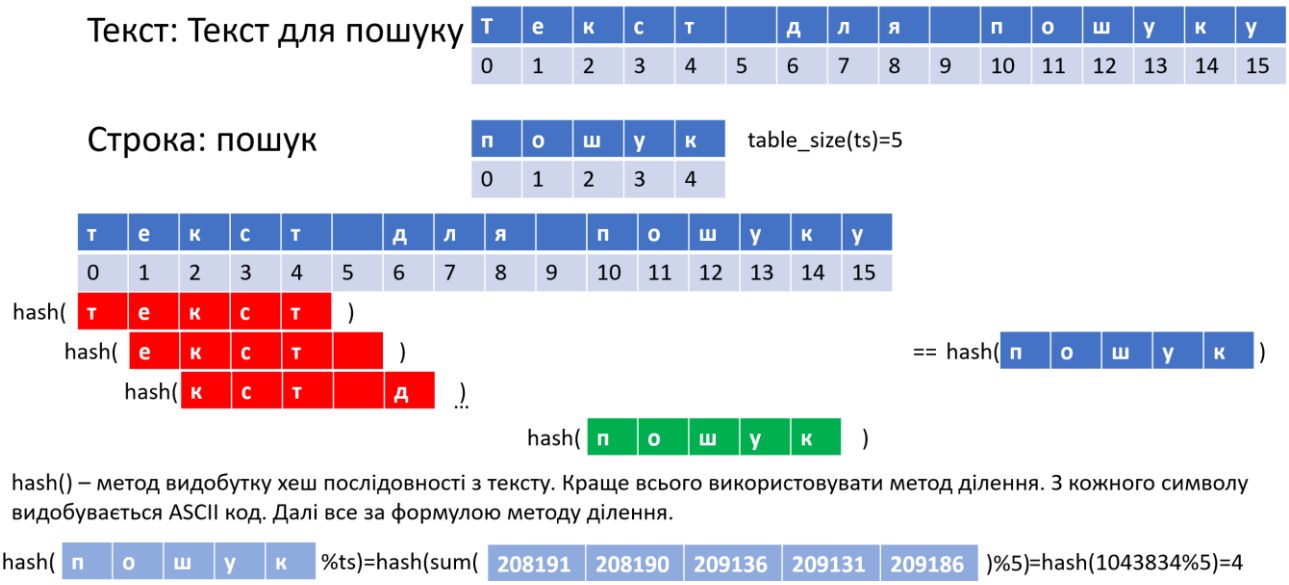
Алгоритм Робіна-Карпа - це алгоритм для пошуку шаблону в тексті, основним принципом якого є обчислення хеш-функції для кожної можливої частини тексту в послідовному порядку та порівнянні його з хеш-значенням шаблону.

Основні кроки алгоритму Робіна-Карпа:

1. Обчислення хеш-значення шаблону і першого «вікна» тексту.
2. Порівняння хеш-значень.
3. Якщо хеш-значення співпадають, перевірка фактичного збігу шаблону з текстом.
4. Якщо не співпадають, обчислення хеш-значення для наступного вікна тексту та порівняння з хеш-значенням шаблону.
5. Повторення кроків 3-4 до досягнення кінця тексту або знаходження входження шаблону.

Складність - O(n * m) часу, де n - розмір тексту, а m - розмір шаблону, що робить його ефективним для пошуку великих текстових даних

Задача:



Описана задача наочно показує роботу методу Робіна-Карпа.

Алгоритм Бойера-Мура

Алгоритм Бойера-Мура — алгоритм для пошуку підрядка у рядку, основна ідея якого полягає в тому, щоб якомога швидше пересувати позицію пошуку, пропускаючи зайві порівняння на основі інформації про підрядок.

Основні кроки алгоритму Бойера-Мура наступні:

1. для кожного символу алфавіту визначається максимальний можливий зсув, що може бути застосований при неспівпадінні символа підрядка і тексту за наступним алгоритмом:

1.1. ініціалізуємо таблицю для всіх можливих значенням довжини підрядка;

1.2. проходимо підрядок справа наліво, починають з передостаннього символу. Для кожного символа запам’ятовуємо значення +1, починаючи з 1. Чим ближче символ до кінця підрядка, тим менший зсув. Якщо символ зустрічається в рядку двічі – береться перше значення символу. Якщо символ не зустрічається в підрядку, то зсув може бути дорівнювати довжині підрядка;

2. починаємо з кінця підрядка і порівнюємо його з останнім символом тексту:

2.1. якщо збіг відбувається - продовжуємо порівнювати символи від кінця до початку підрядка.

2.2. якщо виникає неспівпадіння - використовуємо таблицю зсувів, щоб визначити, на скільки символів можна "скочити" в тексті;

3. повторюємо процес до завершення пошуку або знаходження входження підрядка.

Складність алгоритму - O(n/m), де n - довжина тексту, а m - довжина шуканого рядка. Є ефективним для великих текстових даних і виявляється особливо швидким, коли шуканий рядок має довжину m, близьку до середньої довжини слова в тексті.

Приклад показано на рис. 7.5.

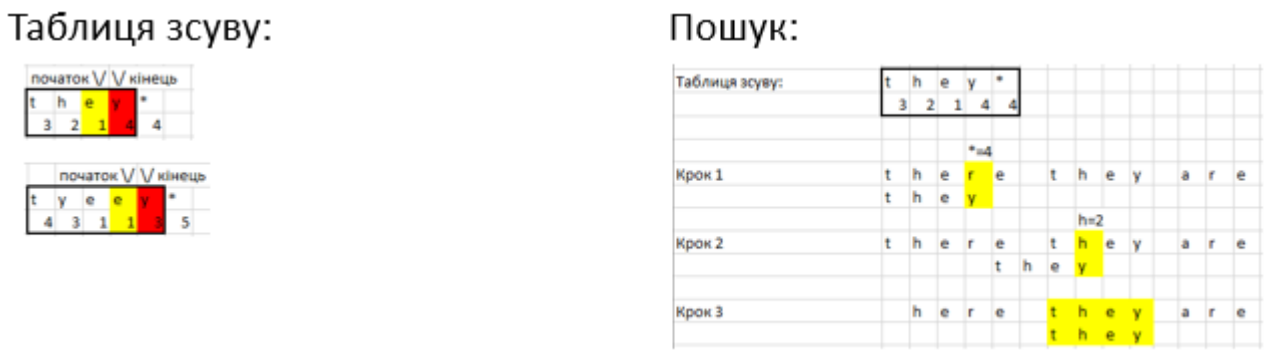


Рис. 7.5 Приклад роботи алгоритму Бойера-Мура

Приклад показує логіку створення таблиці зсуву за принципом описаним за кроком 1. У випадку зі значенням “tyeey”. Визначаємо передостанню букву – e, як зсув – 1. Так само, зсув 2-й букві e визначаємо, як 1. Букви y мають значення 3, оскільки 3-і кроком знаходимо її першою. Проходимось до початку підрядку і захоплюємо останню букву підрядка останнім кроком. Таким чином, формуємо нашу таблицю зсуву. Далі, алгоритм порівнює підрядок з рядком з кінця, зсуваючи підрядок зліва направо. В залежності від індексу рядка визначаємо максимальний його зсув – ті кроки, що гарантовано дадуть неуспішний результат. Виконуємо алгоритм до знаходження рядка. Якщо зсув підрядка досяг кінця строки і значення знайдене не було – робимо висновок, що підрядок в строці відсутній.

Алгоритм Ахо-Корасика

Алгоритм Ахо-Корасика - це алгоритм пошуку підрядків у рядку, основною ідеєю якого є використання структури даних «БОР» та кінцевого автомату.

Алгоритм має широкий спектр застосувань, включаючи пошук підрядків у тексті, перевірку правопису, обробку лексичних аналізаторів і багато іншого. До популярних застосувань даного алгоритму відносяться команда grep і алгоритм перевірки нецензурщини на форумах.

Кінцевий автомат. Кінцевий автомат - це математична модель, що використовується для опису поведінки системи з обмеженим числом станів. Модель є графом(циклічним і ациклічним), в якому вузли відображають стани, а ребра - переходи між станами.

Кінцеві автомати використовуються для моделювання різних систем та процесів, що мають скінченну кількість можливих станів і дискретний характер.

Приклад кінцевого автомату показано на рис. 7.6.

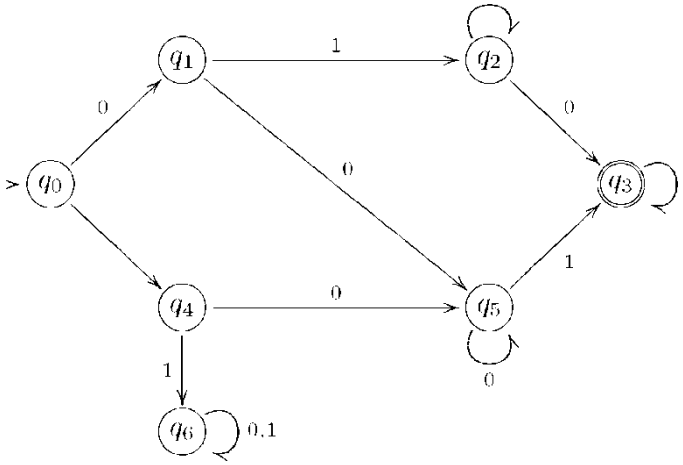


Рис. 7.9 Приклад кінцевого автомату

На прикладі показаний граф, що відображає 7 станів – q0-q6. Зв'язки між ребрами відображають логіку переходу між станами. Стани можуть посилатись самі на себе. Ребра можуть мати вагу – вагу переходу між станами.

Класичними прикладами застосування схеми кінцевого автомату може бути світлофор, регулярний вираз, дверний замок, турнікет, тощо.

Бор. Бор - деревоподібна структура даних, що використовується для зберігання і пошуку множини рядків або послідовностей символів. За своєю суттю, є направленим ациклічним графом, де кожен вузол представляє собою символ, а ребра відповідають переходам між символами.

Основними операціями, що можна виконати з бором є:

- вставка: додавання нового слова або рядка в бор;
- пошук: пошук наявності певного слова або рядка в борі;
- видалення: видалення слова або рядка з бору.

Приклад бор показано на рис. 7.5.

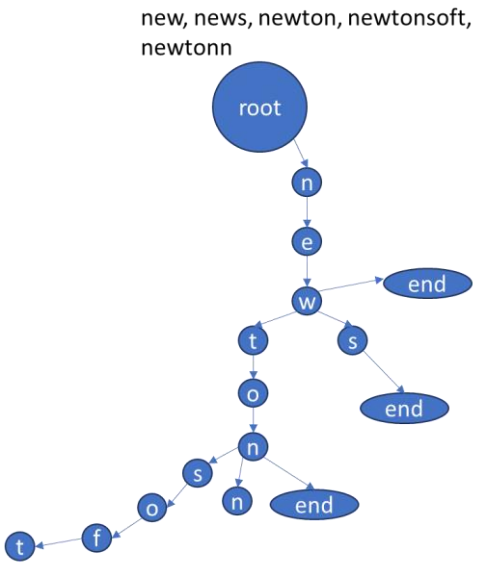


Рис. 7.5 Приклад «БОР»

Як видно з рис. 7.5. для входжень new, news, neton, newtonsoft, newton побудовано деревоподібну структуру, що побудована на принципі об'єднання однакових символів.

Суфіксні вказівники - частина структури даних, що використовується в алгоритмі Ахо-Корасик для підтримки ефективного пошуку зразків у тексті. Метою є організація переходу в кінцевому автоматі. Кожен вузол може мати один або кілька суфіксних вказівників, що вказують на інший вузол у борі. Ці вказівники дозволяють швидше переходити до вузлів, що відповідають можливим суфіксам поточного рядка.

Приклад бору з суфіксними вказівниками показано на рис 7.6.

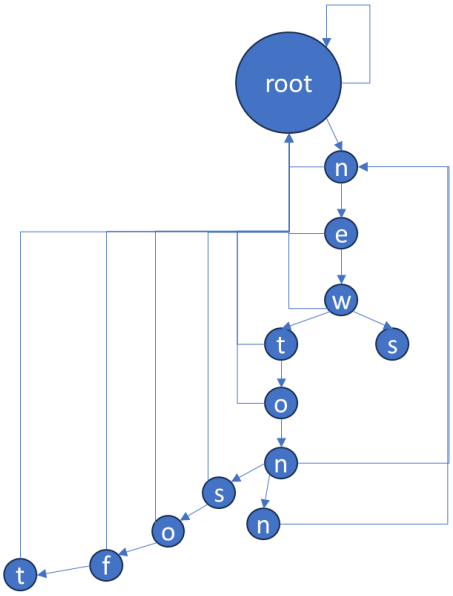


Рис. 7.6 Приклад бору з суфіксними вказівниками

Як видно з прикладу, кожен символ вказує на деякий вузол, на який алгоритм може бути переведений у випадку для подальшого пошуку співпадінь рядка у підрядку. До прикладу, букви n мають посилання на першу букву n, оскільки це означає, що у випадку співпадіння входження підрядка з подальшими символами підрядка може бути знайдений додатковий збіг.

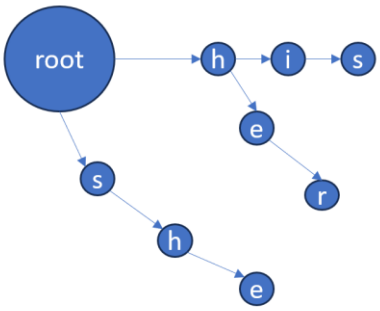
Базуючись на короткому описі кінцевого автомату та структури даних «БОР» можна детально висвітлити кроки алгоритму Ахо-Корасика:

1. побудова бору на основі входжень тексту;
2. побудова суфіксних вказівників;
3. побудова функції переходу;
4. пошук: починаючи з кореня бору, виконується пошук за допомогою функції переходу, яка дозволяє ефективно відстежувати збіги зразків у вхідному тексті.
5. обробка збігів: при кожному збігу виконуються певні дії, такі як позначення місця збігу, підрахунок кількості збігів тощо.

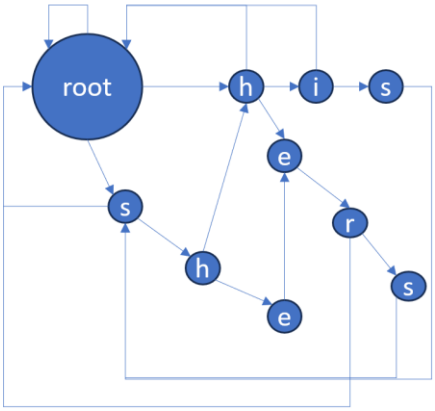
Приклад роботи алгоритму показано за ілюстрованою зв’язкою рис. 7.7.

Задача: знайти в строці shers підстроки she, her, his

1. Будуємо БОР:

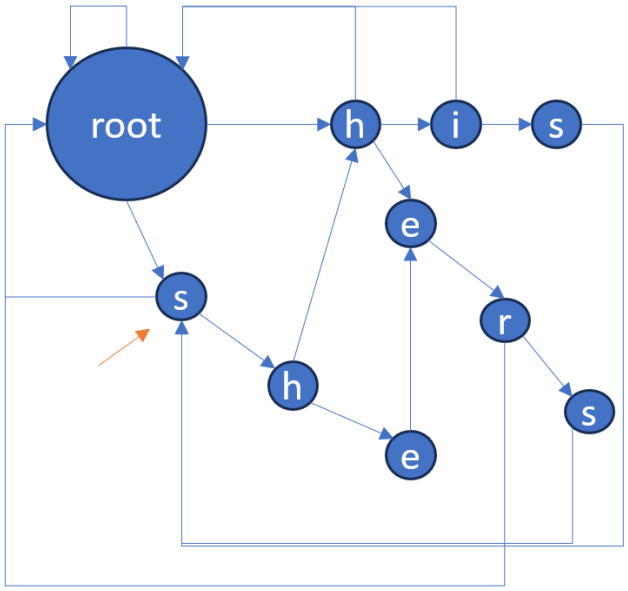


2. Будуємо суфіксні вказівники та переходи



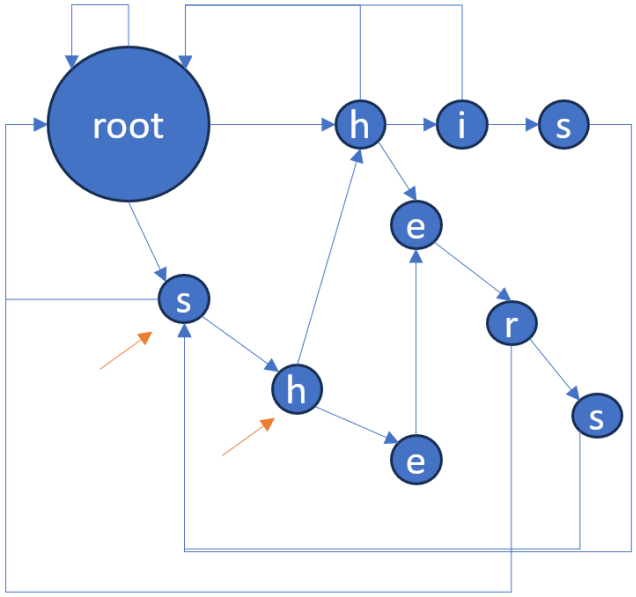
3. Пошук алгоритмом при використанні підходу скінченого автомату

shers



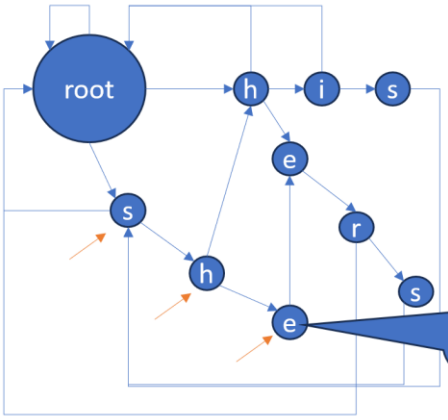
4. Пошук алгоритмом при використанні підходу скінченого автомату

shers



5. Пошук алгоритмом при використанні підходу скінченого автомату

shers

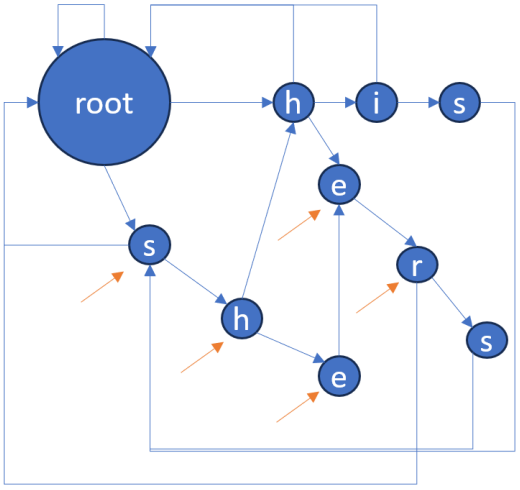


Знайдені слова:
1. she

Бачимо кінець входження і що є суфіксне посилання. Записуємо співпадіння по цьому слову, передимо по суфіксному посиланню поки текст не закінчився

6. Пошук алгоритмом при використанні підходу скінченого автомату

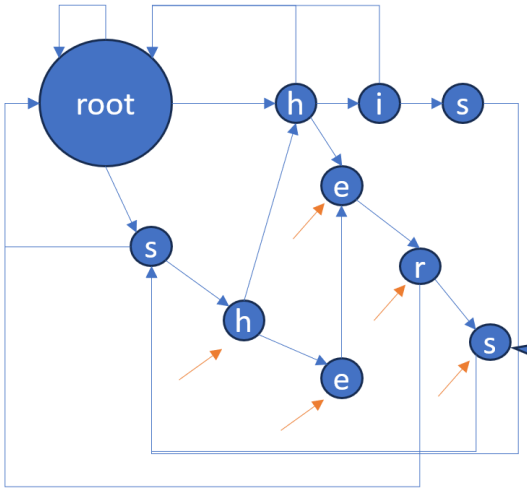
shers



Знайдені слова:
1. she

7. Пошук алгоритмом при використанні підходу скінченого автомату

shers



Бачимо кінець входження і кінець строки. Записуємо входження і закінчуємо пошук, т.я. досягли кінцевого стану

Знайдені слова:
1. she
2. hers

8. Обробляємо збіги:

Шукані слова	Знайдені слова
she	she
his	hers
hers	

Слово his не було знайдено.

Рис. 7.7. Ілюстрований приклад алгоритму Ахо-Корасик

Приклад наочно показує роботу «БОР», кінцевого автомату і роботу самого алгоритму.

2. Динамічне програмування

Динамічним програмуванням є метод розв'язання складних задач, що полягає в розбитті задачі на більш прості підзадачі і збереженні результатів цих підзадач для подальшого використання. Зазвичай використовується тоді, коли задача має ознаки оптимальності, тобто може бути розв'язана за допомогою комбінації оптимальних рішень її підзадач.

Основні етапи застосування динамічного програмування:

1. Визначення підзадач. Складна задача розбивається на більш прості підзадачі. Ці підзадачі повинні бути невеликими і розв'язуватися ефективно.
2. Визначення рекурентних відношень. Для кожної підзадачі визначається відношення, яке описує залежність розв'язку даної підзадачі від розв'язків інших підзадач.
3. Вибір стратегії розв'язання. Існують дві основні стратегії розв'язання підзадач - знизу вгору та зверху вниз.

У випадку застосування стратегії знизу вгору ми спочатку розв'язуємо найменші підзадачі, а потім поступово вирішуємо більші, зберігаючи проміжні результати. У стратегії зверху вниз ми спочатку розв'язуємо найбільші підзадачі, а потім використовуємо ці результати для розв'язання менших підзадач.

4. Обчислення значень. Здійснюється обчислення значень для всіх підзадач згідно визначених рекурентних відношень.
5. Побудова розв'язку. Після обчислення значень всіх підзадач отримуємо розв'язок вихідної задачі.

Таким чином, базуючись на цих етапах, маємо можливість розбити задачу на підзадачі, і вирішуючи кожну з них досягти бажаного результату простішим шляхом.

Практичні завдання

1. Реалізувати та продемонструвати лінійний пошук.
2. Реалізувати та продемонструвати бінарний пошук.
3. Реалізувати та продемонструвати інтерполяційний пошук.
4. Реалізувати та продемонструвати прямий пошук рядка.
5. Реалізувати та продемонструвати алгоритм Кнута-Морріса-Пратта.
6. Реалізувати та продемонструвати алгоритм Бойера-Мура.
7. Реалізувати та продемонструвати алгоритм Бойера-Мура-Хорспула.
8. Реалізувати та продемонструвати алгоритм Робіна-Карпа.
9. Реалізувати та продемонструвати алгоритм Ахо-Корасик.

Контрольні питання

1. Опишіть роботу алгоритму лінійного пошуку з прикладом.
2. Опишіть роботу алгоритму бінарного пошуку з прикладом.
3. Опишіть роботу алгоритму інтерполяційного пошуку з прикладом.
4. Опишіть роботу алгоритму прямого пошуку рядка з прикладом.
5. Опишіть роботу алгоритму Кнута-Морріса-Пратта з прикладом.
6. Опишіть роботу алгоритму Бойера-Мура з прикладом.
7. Опишіть роботу алгоритму Бойера-Мура-Хорспула з прикладом.
8. Опишіть роботу алгоритму Робіна-Карпа з прикладом.
9. Опишіть роботу алгоритму Ахо-Корасик з прикладом.

Список використаних джерел

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.
2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in Java (6th ed.). Wiley.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
4. Skiena, S. S. (2020). The algorithm design manual (3rd ed.). Springer.
5. Weiss, M. A. (2013). Data structures and algorithm analysis in C++ (4th ed.). Pearson.
6. Bhargava, A. (2016). Grokking algorithms: An illustrated guide for programmers and other curious people. Manning Publications.
7. Бондаренко, М. Ф., & Білик, А. І. (2019). Алгоритми та структури даних. Харків: НТУ "ХПІ".
8. Гнатенко, Г. М. (2020). Структури даних та алгоритми. Київ: КПІ ім. Ігоря Сікорського.
9. GeeksforGeeks. (2024). Introduction to red-black tree. Доступно за: https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/.
10. Bhargava, A. (2016). Grokking algorithms: An illustrated guide for programmers and other curious people. Manning Publications.
11. Бондаренко, М. Ф., & Білик, А. І. (2019). Алгоритми та структури даних. Харків: НТУ "ХПІ".
12. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). Algorithms. McGraw-Hill Education.
13. GeeksforGeeks. (2024). Introduction to red-black tree. Доступно за: https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/.
14. Гнатенко, Г. М. (2020). Структури даних та алгоритми. Київ: КПІ ім. Ігоря Сікорського.
15. Aho, A. V., & Hopcroft, J. E. (2008). The design and analysis of computer algorithms (Updated ed.). Addison-Wesley.
16. Brassard, G., & Bratley, P. (2006). Fundamentals of algorithmics. Pearson.
17. Carrano, F. M. (2012). Data structures and abstractions with Java (3rd ed.). Prentice Hall.
18. Cormen, T. H. (2013). Algorithms unlocked. MIT Press.
19. Drozdek, A. (2012). Data structures and algorithms in C++ (4th ed.). Cengage Learning.
20. Erickson, J. (2019). Algorithms. Доступно за: https://jeffe.cs.illinois.edu/teaching/algorithms/.
21. Gusfield, D. (2009). Algorithms on strings, trees, and sequences: Computer science and computational biology. Cambridge University Press.
22. Kleinberg, J., & Tardos, É. (2005). Algorithm design. Addison-Wesley.
23. Lafore, R. (2007). Data structures and algorithms in Java (2nd ed.). Sams Publishing.
24. Levitin, A. (2012). Introduction to the design and analysis of algorithms (3rd ed.). Pearson.

25. Manber, U. (2008). Introduction to algorithms: A creative approach. Addison-Wesley.
26. McConnell, J. J. (2007). Analysis of algorithms: An active learning approach (2nd ed.). Jones & Bartlett Learning.
27. Mehlhorn, K., & Sanders, P. (2008). Algorithms and data structures: The basic toolbox. Springer.
28. Okasaki, C. (2008). Purely functional data structures. Cambridge University Press.
29. Sedgewick, R., & Flajolet, P. (2013). An introduction to the analysis of algorithms (2nd ed.). Addison-Wesley.
30. Shaffer, C. A. (2011). Data structures and algorithm analysis (3rd ed.). Dover Publications.
31. Standish, T. A. (2009). Data structures, algorithms, and software principles in C. Pearson.
32. Tamassia, R., & Goodrich, M. T. (2006). Algorithm design: Foundations, analysis, and internet examples. Wiley.
33. Бабичев, С. А. (2018). Основи програмування та алгоритми. Львів: ЛНУ ім. Івана Франка.
34. Герасименко, О. В. (2021). Алгоритми та структури даних: Практичний курс. Київ: Видавництво "Техніка".
35. Дорошенко, А. Ю. (2016). Алгоритмізація та програмування. Київ: НТУУ "КПІ".
36. Cormen, T. H., & Leiserson, C. E. (2009). Introduction to algorithms (3rd ed.). MIT Press.
37. Eppstein, D. (2018). Dynamic graph algorithms. In Handbook of computational geometry (pp. 231–262). Chapman and Hall/CRC.
38. GeeksforGeeks. (2023). Introduction to graph algorithms. Доступно за: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/.
39. GeeksforGeeks. (2024). Dynamic programming. Доступно за: https://www.geeksforgeeks.org/dynamic-programming/.
40. Klein, P. N. (2010). Coding the matrix: Linear algebra through computer science applications. Newtonian Press.
41. Miller, B. N., & Ranum, D. L. (2011). Problem solving with algorithms and data structures using Python. Franklin, Beedle & Associates.
42. Roughgarden, T. (2020). Algorithms illuminated (Part 1): The basics. Soundlikeyourself Publishing.
43. Skiena, S. S., & Revilla, M. A. (2005). Programming challenges: The programming contest training manual. Springer.
44. Wagner, D., & Willhalm, T. (2007). Speed-up techniques for shortest-path algorithms. Journal of Experimental Algorithmics, 12, 1–26.
45. Weiss, M. A. (2021). Data structures and algorithm analysis in Java (4th ed.). Pearson.
46. Baase, S., & Van Gelder, A. (2008). Computer algorithms: Introduction to design and analysis (3rd ed.). Addison-Wesley.
47. Бойко, В. В., & Савчук, М. М. (2017). Алгоритми та структури даних у програмуванні. Львів: Видавництво "Магнолія".

48. Demaine, E. D., & Goldwasser, S. (2020). Introduction to algorithms and data structures. MIT OpenCourseWare. Доступно за: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/.
49. Heineman, G. T., Pollice, G., & Selkow, S. (2008). Algorithms in a nutshell. O'Reilly Media.
50. Karumanchi, N. (2016). Data structures and algorithms made easy: Data structures and algorithmic puzzles (5th ed.). CareerMonk Publications.
51. Кравець, П. О. (2022). Основи алгоритмізації та програмування. Київ: Видавництво "Ліра-К".
52. Neapolitan, R. E. (2014). Foundations of algorithms (5th ed.). Jones & Bartlett Learning.
53. Roughgarden, T. (2021). Algorithms illuminated (Part 2): Graph algorithms and data structures. Soundlikeyourself Publishing.
54. Skiena, S. S. (2008). The algorithm design manual (2nd ed.). Springer.
55. Wagner, R. A., & Fischer, M. J. (2010). The string-to-string correction problem and related algorithms. Journal of the ACM, 57(3), 1–28.