Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

С. О. Цибульник

ПРОГРАМНІ АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Курс лекцій

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Комп'ютерно-інтегровані системи та технології в приладобудуванні»
спеціальностей 151 Автоматизація та комп'ютерно-інтегровані технології
та
174 Автоматизація, комп'ютерно-інтегровані технології та робототехніка

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

УДК 004.421.2
Ц11

Рецензент *Приходько С., заступник начальника відділу КП СПБ «Арсенал»*

Відповідальний редактор *Півторак Д.О., канд. техн. наук, доцент кафедри комп'ютерно-інтегрованих оптичних та навігаційних систем КПІ ім. Ігоря Сікорського*

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 3 від 07.12.2023 р.)
за поданням вченої ради приладобудівного факультету
(протокол № 9/23 від 30.10.2023 р.)

Цибульник С.О.

Ц11 Програмні алгоритми та структури даних. Курс лекцій [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Комп'ютерно-інтегровані системи та технології в приладобудуванні» спец. 151 Автоматизація та комп'ютерно-інтегровані технології та 174 Автоматизація, комп'ютерно-інтегровані технології та робототехніка / С. О. Цибульник; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2023. – 126 с.

Даний навчальний посібник містить лекційний матеріал, необхідний для розуміння основних термінів і понять у сфері обробки та маніпулювання даними для виконання конкретних практичних завдань зі створення структур для ефективного пошуку та сортування даних, а також контрольні запитання для самоперевірки. Він буде також корисним для студентів інших спеціальностей та закладів вищої освіти.

У навчальному посібнику основну увагу приділено найбільш розповсюдженим структурам даних, серед яких масиви, списки, стеки, черги, хеш-таблиці, графи та дерева. Також приділено увагу базовим алгоритмам побудови та перетворення згаданих структур даних. Наведено основні алгоритми пошуку та сортування даних, які використовуються в сучасній практиці обробки великих об'ємів даних, з основами оцінювання їх швидкодії.

УДК 004.421.2

Реєстр. № НП 23/24-158. Обсяг 5,5 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП

Навчальний посібник складено відповідно до чинного силабусу навчальної дисципліни «Програмні алгоритми та структури даних» для здобувачів ступеня бакалавра приладобудівного факультету, які навчаються за спеціальностями 151 «Автоматизація та комп’ютерно-інтегровані технології», 174 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» та освітньою програмою «Комп’ютерно-інтегровані системи та технології в приладобудуванні». Даний навчальний посібник також може використовуватися здобувачами та студентами інших спеціальностей або освітніх програм.

Мета даного навчального видання – допомогти студентам отримати необхідні знання та уміння з теорії та практики, а також проектування та реалізації структур та алгоритмів при вирішенні задач обробки великих масивів даних сучасними методами.

Виконання практичних завдань, пов’язаних з освоєнням принципів оцінки часу виконання алгоритмів, методик вибору ефективних структур даних для вирішення задач пошуку та сортування інформації сприятиме закріпленню, поглибленню та узагальненню теоретичних основ курсу, а також розвитку навичок самостійної творчої роботи студентів у процесі їх навчання, зокрема при виконанні контрольних та інших видів робіт з дисципліни «Програмні алгоритми та структури даних».

Навчальний посібник містить лекційний матеріал, необхідний для розуміння основних термінів та понять у сфері обробки та маніпулювання даними, для виконання конкретних практичних завдань зі створення структур для ефективного пошуку та сортування даних, а також контрольні запитання для самоперевірки.

Лекція №1

Основні терміни та поняття

Розробники низки сучасних комерційних програмних продуктів дуже часто не звертають уваги ні на часову ефективність, ні на оптимальне використання оперативної пам’яті. На їх думку, це можна пояснити тим, що розвиток апаратних засобів та елементної бази дозволяє не звертати уваги на більшість недоліків кодової бази, тому що проблеми швидкодії можуть бути легко виправлені закупівлею більш потужного обладнання. Подібні розробники вважають, що якщо їх програмне забезпечення займає занадто багато місця на пристрої користувача, то він придбає додаткову пам’ять, а якщо воно працює занадто повільно, то він може купити швидший процесор.

Однак розширення апаратних можливостей пристроїв не може відбуватися нескінченно. У мобільних телефонах та планшетах за замовчуванням відсутня можливість модифікації комплектуючих. Те ж саме можна сказати про ноутбуки, в яких ця можливість значно обмежена, а саме: для заміни, наприклад, процесора на більш потужний майже завжди необхідно змінювати материнську плату, що рівносильно купівлі нового пристрою. Персональні комп’ютери дають більшу свободу модифікації компонентів, але з урахуванням конкуренції у сфері розроблення програмного забезпечення, кінцевому користувачу на багато простіше знайти аналогічний за функціональними можливостями програмний продукт з необхідними показниками якості, ніж проводити дорогу модифікацію обладнання.

Окрім потужності самих компонентів, ефективність роботи апаратних засобів обмежена також швидкістю: комутації каналів зв'язку комп'ютерів, що беруть участь у передачі даних; поширення світла оптичними кабелями;

переміщення електронів по проводах; тощо. Також є ще одна категорія обмежень: складність вихідної задачі. Існують завдання, для вирішення яких при сучасному рівні технологій не вистачить одного людського життя. Складність таких задач не сильно зміниться навіть при використані найшвидших із відомих алгоритмів. Проте серед цих завдань є життєво необхідні для виживання суспільства. Саме тому необхідно розроблювати швидкі алгоритми для отримання наближених у межах припущень рішень.

На початку бурхливого розвитку комп'ютерної техніки (80-ті роки XX століття) архітектура апаратних засобів та технологія виготовлення серйозно обмежували їхню швидкодію та обсяг оперативної пам'яті. Як правило, у той час загальний розмір програмних продуктів та окремих даних не перевищував 64 кб. Якщо порівнювати з сучасними системами, то ця величина зросла в середньому в сотні тисяч разів. Саме тому програмне забезпечення, яке розробляється сьогодні є набагато складнішим, ніж у 1980 році. Більші апаратні потужності дозволяють вирішувати складніші завдання, але це не привід, щоб ігнорувати питання оптимізації програмного коду в процесі розроблення програмного забезпечення для підвищення його якості та ефективності.

Лише знання певної мови програмування у сучасному світі є недостатнім для того, щоб роботодавці розглядали потенційного кандидата на роль програміста у своїй компанії. Для розроблення програмного забезпечення будь-якої складності недостатньо просто знати мову програмування. Життєвий цикл програмного забезпечення складається з багатьох етапів, починаючи зі збору та формалізації вимог і закінчуючи отриманням якісного програмного забезпечення, яке має оптимальну кодову базу і правильно вирішує поставлені завдання.

Визначення вимог є найважливішим етапом процесу розроблення. У специфікацію вимог більшості сучасних проектів включено (у відповідності

до міжнародного стандарту) обмеження на швидкість виконання операцій та на об’єм використання оперативної пам’яті кінцевим програмним продуктом. Дані обмеження мають спонукати програмістів ефективно розподіляти оперативну пам’ять та збільшувати швидкість виконання операцій при мінімальних апаратних потужностях. Подібні вимоги є актуальними для проектів будь-якого рівня складності.

Окрім визначення вимог на даному етапі будуються математичні моделі окремих підзадач із залученням різних математичних конструкцій, таких як: системи диференціальних чи лінійних алгебраїчних рівнянь, матриці, графи, множини, дерева, тощо. Для кожної математичної моделі проводиться пошук ефективних методів рішення поставлених завдань у вигляді послідовності виконання скінченого набору інструкцій – алгоритму.

Інструкція – скінчений набір операндів та операцій. Інструкція є найменшою автономною складовою частиною мови програмування.

Операція – чітка послідовність дій, які виконуються над аргументом.

Операнд – вираз, який визначає аргумент операції. Іншими словами, **операнд** – це дані, які визначають результат операції. Найчастіше подається у вигляді змінної або константи.

Оператор – спеціальний символ, команда чи ключове слово мови програмування, що відповідають за визначення операції, яку необхідно виконати. Наприклад, +, *, %, **for**, **new**, тощо.

Алгоритм – це скінчена послідовність стандартизованих інструкцій (дій) для виконання поставленого завдання, які мають чіткий зміст у межах предметної області, виконуються за скінчений час, а також зі скінченими витратами обчислювальних ресурсів. Іншими словами, алгоритм є методикою розв’язання вихідної задачі, яка має вихідні дані та очікуваний результат. Вимоги до вихідних даних і кінцевого результату визначаються у специфікації вимог до програмного забезпечення чи програмної системи.

Алгоритм у ході свого виконання оперує наступними категоріями даних:

- вихідними – необхідні для початку роботи алгоритму;
- проміжними – результати виконання окремих підзадач алгоритму (можуть бути результатами виконання інших алгоритмів);
- сформованими самим алгоритмом – необхідні для забезпечення його роботи;
- результуючими – очікувані результати виконання алгоритму.

Усі дані, які необхідні для вирішення алгоритмом поставленого завдання, організовуються особливим чином у так звані структури, вибір яких є одним з важливих чинників розроблення алгоритмів. Це пов’язано з тим, що дуже часто від вибору структур даних залежить ефективність виконання алгоритму. Одне й те ж саме завдання майже завжди можна вирішити з використанням декількох різних алгоритмів, які можуть відрізнятися кількістю необхідних ресурсів: елементарних операцій та об’єктів. Наприклад, під час виконання таких елементарних операцій, як додавання, віднімання, множення, порівняння, ділення, перехід та інших, використовується процесор, а для тимчасового зберігання об’єктів – оперативна пам’ять.

Отже, **структура даних** – один зі способів організації даних, який складається з механізмів їх подання та механізмів взаємодії з ними.

Найбільш поширеними структурами даних у сфері розроблення програмного забезпечення є: масив, зв’язаний список, стек, черга, дерево, хеш-таблиця та інші. Алгоритми неспроможні існувати без подібних структур даних. При цьому неможливо створити ефективний алгоритм, використовуючи структури, які йому не відповідають.

Алгоритми пояснюють, як ефективно сортувати записи, розраховувати результати обчислень (наприклад, прості множники), шукати

елементи, знаходити найкоротший шлях між двома точками, визначати пропускну здатність мережі, тощо.

Для виконання поставлених завдань алгоритм повинен мати наступні властивості:

- корисність – уміння правильно вирішувати поставлене завдання;
- детермінованість – суворе дотримання визначеної послідовності кроків виконання у всіх можливих ситуаціях;
- кінцевість – здатність завершуватися для будь-якого набору вхідних даних;
- масовість – можливість застосування з різноманітними вхідними даними;
- ефективність – здатність використовувати обмежену (мінімальну) кількість ресурсів;
- коректність – здатність отримувати правильні результати при будь-яких допустимих вхідних даних.

Це не вичерпний перелік властивостей якісного та ефективного алгоритму, але на зазначені буде звертатися особлива увага у подальших розділах даного навчального посібника. Щоб забезпечити наведені властивості, при розробленні кожного алгоритму варто задати собі наступні питання:

1) Швидкість алгоритму. Він виконується з достатньою швидкістю для рішення поточної задачі чи ні? Робота алгоритму сповільнюється при будь-яких вхідних даних чи тільки при певних комбінаціях?

2) Вимоги до оперативної пам’яті, яку потребує алгоритм для виконання. Яка кількість обчислювальних ресурсів потрібна алгоритму для ефективної роботи? Чи впорається поточний алгоритм з рішенням задачі з доступною йому кількістю оперативної пам’яті?

3) Поведінка алгоритму. Алгоритм просто вирішує завдання чи знаходить найкраще (оптимальне) рішення? Чи може бути декілька найкращих (оптимальних) рішень поставленого завдання? Чи є сенс віддати перевагу тільки одному з них?

4) Основні методи, які використовуються в алгоритмі. Чи можна задіяти їх повторно для вирішення подібних завдань?

Також варто зазначити, що аналіз існуючих чи розроблюваних алгоритмів не залежить від конкретних пристроїв (персонального комп'ютера, телефона, планшета, тощо) або мов програмування (Java, Python, C#, тощо). Саме тому більшість алгоритмів, які будуть розглядатися у даному навчальному посібнику, будуть записані у вигляді псевдокоду. Ці алгоритми зможе прочитати будь-яка людина, яка знайома з базовими поняттями умовного виразу (наприклад, для конструкцій розгалуження **if-else** чи **swith-case**), циклу (**for** чи **while**) та рекурсії.

Рекурсія – явище, коли метод (функція) прямо чи опосередковано (за допомогою інших методів) викликає сам себе. Більш детально рекурсію буде розглянуто в наступних лекціях.

Псевдокод для запису алгоритмів

Для запису прикладів деяких алгоритмів у даному навчальному посібнику буде використовуватись псевдокод, логіка якого будуватиметься на основі використання інструкцій, конструкцій та операторів мови програмування Java. При цьому, для забезпечення достатнього рівня абстрагування від конкретної мови програмування, а також для більш зручного сприйняття алгоритмів, синтаксис деяких елементів псевдокода відрізняється від синтаксису відповідних елементів мови Java.

Крім інструкцій, конструкцій та операторів у псевдокоді використовуються також такі розповсюджені елементи мов програмування, як: змінні, умови, вирази, методи (функції), тощо. Наведений псевдокод не матиме фіксованого набору типів даних чи деяких структур, тому змінні можуть належати довільному типу (масив, вектор, список, тощо). Описи типів в алгоритмах наводитися не будуть, але тип даних будь-якої змінної та її область використання (глобальна, локальна) стають зрозумілими з контексту псевдокоду.

Умовні вирази або умови (приймають значення *true* або *false*) записуватимуться у вигляді класичної математичної конструкції, а не відповідно до синтаксису обраної мови програмування. В якості умови може виступати також довільна фраза природною мовою, яка задає правило обчислення значення певного виразу, наприклад, фраза «довільний елемент з множини X» є виразом, а фраза «y – просте число» – умовою.

Методи (функції) мають той самий зміст, що й у мові програмування Java. Відмінність полягає в тому, що при визначенні методу не будуть вказуватися: типи формальних параметрів; тип самих методів; поділ параметрів на константи та змінні. Ця інформація має бути зрозумілою з контексту. Методи (функції), які використовуються в тілі інших методів, вважаються локальними, якщо їх належність до глобальних методів не обговорена окремо або не випливає з контексту.

Псевдокод використовує наступні оператори:

1) Оператор присвоєння виду

змінна ← вираз

обчислює значення виразу та присвоює його змінній. У ряді випадків для скорочення запису може використовуватися одночасне присвоєння, наприклад, інструкція «*змінна1 ← змінна2 ← вираз*» є еквівалентною послідовності двох операторів присвоювання: *змінна1 ← вираз* та *змінна2 ← вираз*. Час

виконання оператора присвоєння визначається часом дії присвоєння, а також часом обчислення виразу.

2) Розгалуження виду

if (*умова)* **then** *інструкції* **else** *інструкції*

або

if (*умова)* **then** *інструкції*

має той самий сенс, що й у Java. Час виконання конструкції визначається як сума часу, необхідного для обчислення значення умовного виразу та його перевірки, а також часу виконання інструкції в основній частині або в **else**-частині залежно від того, яка із них виконується.

3) Розгалуження виду

$$\textbf{\textit{switch}}\ (\text{змінна})\ \begin{cases} \textbf{\textit{case}}\ (\text{значення}):\ \text{інструкції};\ \textbf{\textit{break}}; \\ \ldots \\ \textbf{\textit{case}}\ (\text{значення}):\ \text{інструкції};\ \textbf{\textit{break}}; \end{cases}.$$

У даній конструкції умова задається неявно, а саме: значення змінної **switch**-частини порівнюється з кожним значенням, яке записано у «значенні» кожної **case**-частини до першого результату «*true*». Час виконання даної конструкції розгалуження складається з суми часу, необхідного на пошук та перевірку умови, а також часу виконання відповідних інструкцій.

4) Оператори циклу з передумовою виду

while (*умова*) **do** *інструкції*

та постумовою виду

do *інструкції* **while** (*умова*)

мають той самий сенс, що й у Java. Час виконання оператора циклу визначається як сума часу виконання кожної ітерації, який складається з часу перевірки умови виходу з циклу, а також часу виконання інструкцій в тілі циклу. При цьому слід враховувати, що в операторі циклу **while** число перевірок умови виходу з циклу на одиницю більше, ніж число ітерацій. У свою

чергу, в операторі циклу **do-while** число перевірок умови виходу з циклу дорівнює числу ітерацій.

5) Оператор циклу з параметром виду

for (*змінна* ← *вираз1*; *умова*; *вираз2*) **do** *інструкції*

Тут змінна є локальним параметром циклу; вираз1 та вираз2 – відповідно початкове значення змінної та розмір кроку зміни параметра циклу. Час виконання оператора циклу **for** визначається аналогічно до часу виконання оператора циклу **while** з урахуванням операцій, які пов'язані з параметром циклу та проходять у визначенні оператора **for**. У випадках, коли початкове значення, умова та розмір кроку параметра циклу є очевидними, використовується простіша форма оператора. Наприклад, запис

for (x ∈ S) **do** *інструкції*

означає, що інструкції тіла циклу виконуються для кожного елемента множини *S*, а число ітерацій дорівнює її довжині.

6) Блок коду. У псевдокоді для виокремлення послідовності інструкцій, які належать одній і тій же конструкції чи оператору циклу, будуть використовуватися операторні дужки початку «{» та кінця «}» або ключові слова **begin** та **end** (наприклад, **end for**, **end while**, **end if**, тощо), або односторонння фігурна дужка (див. пункт 3) конструкцію **switch-case**). Кожна інструкція блоку коду записується в окремому рядку, тому спеціальний символ (наприклад, у більшості мов програмування – точка з комою) для відокремлення інструкцій одна від одної не вказується. Час виконання блоку коду визначається сумою часів виконання інструкцій, які входять до нього.

7) Оператор повернення результату **return** використовується як останній оператор будь-якого методу (функції). Використання даного оператору у псевдокоді робить зайвим застосування одного з наведених у пункті 6 позначень початку або кінця блоку коду для виділення границь методів. У прикладах використовуватиметься форма оператора повернення виду

return (*вираз*),

де значення обчисленого виразу є значенням, яке повертає метод. Час виконання оператора повернення визначається часом обчислення виразу. Якщо ж вираз подається у вигляді константи, то час виконання оператора також є константою.

8) Оператор виклику методу (функції) виду

function *ім'я_методу_або_функції* (*фактичні_параметри*)

має той самий сенс, що й у Java. Час виконання визначається часом виклику з урахуванням передачі відповідних параметрів та виконання тіла методу (функції).

9) Довільний оператор є фразою природною мовою, що визначає дію. Такі оператори використовують у випадках, коли деталі реалізації несуттєві, очевидні чи розглянуті окремо. Час виконання визначається часом, необхідним реалізації зазначеного дії.

10) Логічні оператори, які є складовою частиною будь-якого логічного виразу, у псевдокоді позначатимуться наступним чином:

- **AND** – логічне множення, логічне «і»;
- **OR** – логічне додавання, логічне «або»;
- **!** – заперечення, логічне «не».

Додатково при написанні псевдокоду будуть використовуватися наступні припущення:

а) якщо спеціально не обумовлено, всі логарифми беруться з основою 2, тобто запис $\log(x)$ означає $log_2(x)$;

б) для більш компактного представлення алгоритмів операція обміну значеннями записуватиметься у вигляді $x \leftrightarrow y$ (значення змінних x та y змінюються місцями). Даний запис є еквівалентним наступній послідовності виконання з трьох операторів присвоювання: $temp \leftarrow x$; $x \leftarrow y$; $y \leftarrow temp$;

в) для обчислення логічних виразів використовуються швидкі оператори. Якщо для логічного виразу (*a* **AND** *b*) (*a* і *b* – логічні операнди) встановлено, що *a* = *false*, то навіть без обчислення логічного виразу можна сказати, що він матиме значення *false*. Аналогічно для (*a* **OR** *b*), якщо *a* = *true*, те значення всього виразу буде також *true*;

г) для включення до алгоритму коментарів природною мовою, які будуть спрямовані на пояснення його окремих частин, використовується символ «//», який починає коментар.

Контрольні запитання.

1. Що таке алгоритм?
2. Чим інструкція відрізняється від оператора?
3. Який зв’язок між операндами та операціями?
4. Чому важливо використовувати оптимальні алгоритми для рішення задач?
5. Що таке псевдокод?
6. Які властивості має алгоритм?

Лекція №2

Масиви, списки, стеки та черги

В основі процесу розроблення алгоритмів лежить робота з різними структурами даних, які часто являються абстрактними типами даних (особливо в об'єктно-орієнтованих мовах програмування). Одним з базових абстрактних типів даних є множина – сукупність деяких елементів. Над множинами можна виконувати наступні операції: пошук (визначення приналежності елемента до даної множини), сортування, додавання чи вилучення елементу, тощо. Наведений перелік операцій над множинами не є вичерпним, але вони є універсальними й використовуються практично у кожному програмному забезпеченні. Для реалізації множин використовується декілька різних структур даних, вибір яких залежить від переліку необхідних операцій. Тому в даному розділі буде розглянуто основні структури даних, які використовуються для створення множин скінченої довжини, та деякі операції над ними.

Розглянемо поняття індексу. Послідовність вигляду

$$x_1, x_2, x_3, \ldots, x_n$$

можна формально визначити як функцію, областю визначення якої є множина об'єктів або чисел. У наведеній послідовності кожен елемент займає певну позицію і має свій унікальний індекс, який вказує на цю позицію. Це означає, що існує однозначна відповідність між елементом послідовності та його індексом. Будь-яка послідовність може мати один або декілька однакових елементів, які при цьому займатимуть різні позиції, тобто матимуть різні індекси. Проте жодна послідовність не може містити навіть два елементи з однаковими індексами.

Масиви

Найбільш простим уявленням множини $X=\{x_1, x_2, x_3, \ldots, x_n\}$ є точне відображення скінченого переліку її елементів у послідовних комірках оперативної пам'яті. Така структура даних називається послідовним розподілом або масивом.

Кожна комірка оперативної пам'яті має свій індекс, який називається адресою. Індекси комірок є цілими числами, починаючи з 0 і закінчуючи 4 Гб (для 32-бітних систем) або приблизно 16 ЕБ (для 64-бітних систем). Припустимо, що апаратний засіб, на якому проходить виконання алгоритму, має байт-адресовану пам'ять. Це означає, що кожна комірка пам'яті відповідає 1 байту інформації, тобто для зберігання одного числа типу даних *integer* (4 байти) потрібно 4 послідовних комірки пам'яті.

Припустимо, що множина складається з елементів, які належать одному і тому ж типу даних. Тоді для зберігання одного елемента потрібно мати l послідовних комірок оперативної пам'яті, а для масиву – $n \cdot l$ послідовних комірок. Таким чином, з даного співвідношення можна вирахувати адресу першої комірки пам'яті кожного елементу множини для отримання безпосереднього доступу до нього. Це означає, що час доступу до будь-якого елементу є фіксованим.

У подальшому послідовності, які можуть змінювати свою максимальну довжину в процесі виконання алгоритму, називатимемо динамічними, а які не можуть, – статичними. Також обмежимося використанням лише двох основних операцій: додавання та віднімання елементів.

При роботі з масивами використання операції додавання елемента k у задану позицію x_i призведе до переміщення елементів $x_i, x_{i+1}, \ldots, x_n$ вправо (рис. 2.1) на одну позицію. Після зсуву усіх необхідних елементів комірка

пам’яті x_i -го елементу стає доступною для запису нових даних, куди й вставляється новий елемент k.

Операція видалення елемента з позиції x_i відбувається аналогічним способом. Єдиною відмінністю є те, що елементи масиву будуть зміщуватись вліво.

Основною перевагою використання масивів є можливість прямого доступу до будь-якого елемента множини, оскільки існує просте співвідношення між його індексом та адресою комірки оперативної пам’яті, в якій він зберігається. Крім того, послідовний розподіл легко реалізується і вимагає невеликих витрат пам’яті для множин фіксованої довжини.

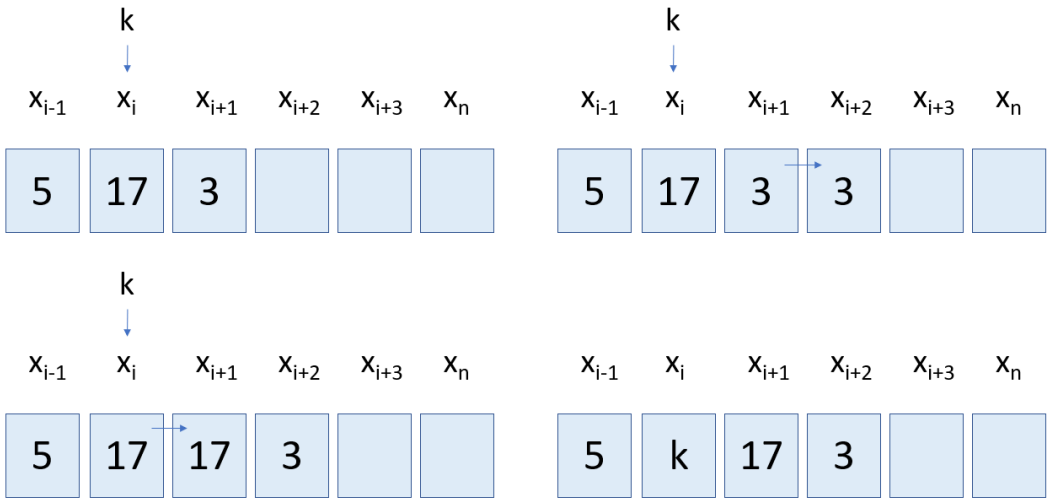


Рисунок 2.1 – Умовний приклад комірок пам’яті з довільно розміщеними даними

Значним недоліком використання масивів є їх низька ефективність у разі необхідності реалізації динамічних структур даних. Цей недолік є наслідком статичної природи масивів, а саме:

- при створенні цієї структури даних доводиться відразу резервувати обсяг пам’яті, виходячи з максимально можливої довжини масиву

незалежно від поточної кількості елементів, які записані в нього, що призводить до підвищеного використання оперативної пам’яті;

- не завжди присутня можливість точного визначення необхідної довжини масиву;
- час виконання операцій додавання та видалення елементів у середині масиву значним чином залежить від його поточного заповнення: чим більше елементів, тим довше виконуються операції.

Для рішення багатьох задач зручно використовувати табличне подання даних. Математичною абстракцією таблиці є матриця, механізми створення якої є в більшості мовах програмування. Як правило, вона реалізується у вигляді двовимірного масиву.

У мові програмування Java (та деяких інших) двовимірний масив представляється як масив масивів. Інакше кажучи, масив з *m* рядків і *n* стовпців насправді являє собою одновимірний масив з *m* елементів, кожен з яких також є одновимірним масивом, але з *n* елементів. Найчастіше подібні структури даних використовуються для зберігання та обробки великих обсягів даних.

Списки

На відміну від масивів (послідовного списку/розподілу) існує інший варіант реалізації множин, який не потребує виділення послідовної неперервної області комірок оперативної пам’яті. Це означає, що різні елементи множини $X=\{x_1, x_2, x_3, \ldots, x_n\}$ можуть зберігатися у будь-яких вільних комірках пам’яті. Але при такій організації зберігання порядок розміщення у пам’яті може бути також довільним, що призводить до виникнення

проблеми, яка зображена на рис. 2.2 (кожен прямокутник відповідає діапазону комірок, в яких міститься певний тип даних).

Рисунок 2.2 – Умовний приклад комірок пам’яті з довільно розміщеними даними

З розкиданих по пам’яті даних певним чином треба відтворити вихідну множину $X=\{x_1, x_2, x_3, \ldots, x_n\}$, зберігаючи правильну послідовність її елементів. Цього можна досягти, якщо кожен елемент окрім значень також буде містити посилання на наступний x_{i+1} елемент списку. Таке рішення проблеми призведе до збільшення необхідного об’єму оперативної пам’яті у порівнянні з послідовним списком і до зменшення швидкодії (робота з послідовними комірками відбувається значно швидше), але є необхідним для рішення інших задач. Таке подання множини називається зв’язним списком.

Реалізація цієї множини має декілька особливостей. По-перше, для доступу до всього зв’язаного списку потрібен додатковий елемент, який містить посилання (яке у Java є лише вказівником на об’єкт) на даний список. Як правило, цей елемент є найпершим і подається у вигляді поля (змінної)

list, яка містить посилання на перший елемент послідовності $X=\{x_1, x_2, x_3, \ldots, x_n\}$.

Для уникнення подальших колізій у термінології будемо вважати поле *list* нульовим елементом списку, а перший елемент множини – першим елементом списку.

Щоб отримати посилання на список, треба використати оператор доступу до членів класу (полів та методів) «.» у вигляді *L.list*, де *L* – ім'я відповідного списку. Кожен його елемент називається вузлом і має два поля: *key* (містить поточне значення відповідного елементу) та *next* (містить посилання на наступний елемент). Здійснити доступ до вузла можна лише в тому випадку, коли на нього присутнє хоча б одне посилання.

По-друге, поле *next* останнього елементу списку містить «порожнє» чи «нульове» значення посилання, тому що наступного елементу в поточному списку не існує. У подальшому таке значення посилання буде позначатися символом «Ω». Покажчиком пустого списку, який не містить жодного елемента множини, є рівність поля *list* значенню Ω, тобто *L.list*= Ω – пустий список.

Спрощене зображення зв'язного списку показано на рис. 2.3.

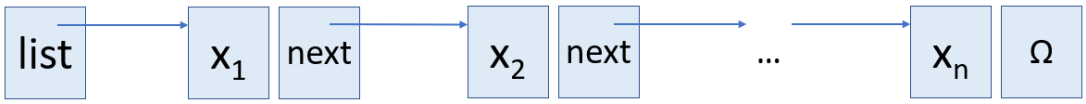


Рисунок 2.3 – Зв'язний список

Отже, зв'язний список є динамічною структурою даних і полегшує виконання деяких операцій. На відміну від масиву, для зберігання такої структури даних заздалегідь не резервуються комірки оперативної пам'яті, – пошук вільного місця відбувається за необхідності при виконанні операцій.

Для забезпечення такої можливості у більшості мов програмування існують механізми, які дозволяють вести взаємодію з диспетчером пам'яті. Наприклад, в Java при створенні об'єкту використовується ключове слово *new*, яке викликає диспетчер пам'яті для пошуку вільного місця під створюваний об'єкт. Диспетчер пам'яті у відповідності до початкового необхідного розміру знаходить безперервну послідовність комірок оперативної пам'яті, резервує її та повертає посилання на початкову комірку. Після цього ключове слово *new* передає отримане посилання в конструктор об'єкта для початкової ініціалізації.

У подальшому будемо використовувати метод *new()* для позначення процесу створення об'єкта або змінної. Це означає, що запис $new(S_i)$ розміщуватиме новий порожній вузол і надає елементу S_i посилання на нього. А для звільнення пам'яті, яка зайнята вузлом, будемо використовувати метод $dispose(S_i)$, після якого $S_i = \Omega$ і посилатися на даний елемент заборонено. Не варто забувати, що під «звільненням пам'яті» мається на увазі надання доступу на перезапис даних, які знаходяться у відповідних комірках, а не їх повне знищення.

Інколи виникає необхідність переходу не тільки до наступних елементів списку, а й до попередніх. У такому випадку зручно використовувати інший вид динамічних списків – двозв'язні (двобічно зв'язані, рис. 2.4).

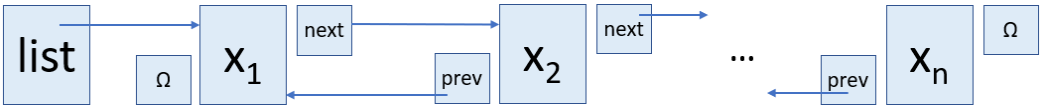


Рисунок 2.4 – Двобічно зв'язаний список

Ключова відмінність у структурі вузла полягає в наявності додаткового поля *prev*, яке містить посилання на попередній (x_{i-1}) елемент списку. Також дане поле першого елементу списку містить значення Ω. Перевага

такої будови полягає у зручності виконання операцій додавання та видалення елемента списку перед заданим без необхідності додаткового маніпулювання посиланнями та даними. Проте простота проведення операцій з елементами двобічно зв'язаного списку призводить до збільшення витрат оперативної пам'яті.

Стеки

Стек – це динамічна послідовність елементів з однією точкою доступу. Точка доступу стеку називається його вершиною, а нижній (перший) елемент – дном. Додавання елементу до стеку або видалення з нього відбувається лише з вершини. Подібний принцип роботи називається «останнім прийшов – першим пішов» (*англ. Last In First Out, LIFO*).

У відповідності до цього принципу вершина стеку є фіксованою (знаходиться на одній і тій же позиції), а його елементи рухаються під час виконання операцій додавання або видалення. Проте програмна реалізація такого алгоритму не є ефективною. На практиці відбувається все навпаки, а саме: положення елементів стеку фіксуються, а його вершина стає рухомою. Для цього додатково вводиться спеціальний вказівник на поточну вершину стеку.

У подальшому у псевдокоді стек буде позначатися символом «*S*», операція додавання елементу x до стеку буде задаватися як $S \Leftarrow x$, а операція видалення – $x \Leftarrow S$. Дані позначення дозволяють не звертати уваги на деталі реалізації стеку, тобто підтримують необхідний рівень інкапсуляції даних, що дає можливість зосередитися на загальній логіці алгоритму.

Як правило, на практиці дані операції реалізовують у вигляді методів (функцій) з іменами *push* для додавання та *pop* для видалення елементів

стеку. Дані методи мають ряд функціональних відмінностей. Додавання елементу у стек потребує двох аргументів (**function** *push(S, x)* або **function** *push(x, S)*), які вказують на елемент та необхідний стек, а також не повертає ніяких значень. На відміну від нього, видалення елементу зі стеку реалізовується з одним аргументом (**function** *pop(S)*), який вказує лише на необхідний стек, і найчастіше (але не обов'язково) повертає значення видаленого елементу. Це означає, що операції $x \Leftarrow S$ відповідає присвоювання $x \leftarrow pop(S)$.

Одним з найбільш розповсюджених способів реалізації стеку є використання масиву (послідовного розподілу), який візуально представляється у вигляді вертикального контейнеру фіксованої висоти. Обмеження висоти (глибини) стеку визначається максимальним розміром масиву, так як він є статичною структурою даних. Наприклад, на рис. 2.5 наведено порожній стек з 6 елементів, яким він уявляється, та його реалізація у вигляді масиву.

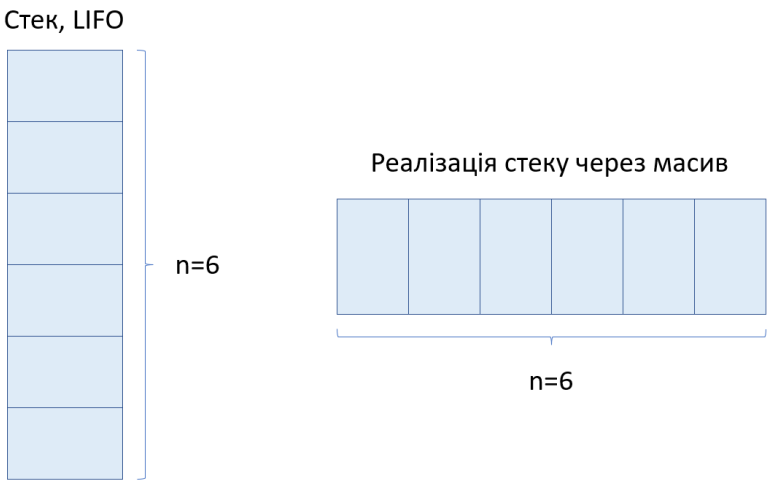


Рисунок 2.5 – Приклад візуального уявлення та реалізації порожнього стеку

У якості вказівника вершини стека будемо використовувати змінну *top*, яка є індексом останнього включеного в стек елементу. При цьому має

завжди виконуватися умова $0 \le top \le n$. У випадку $top=0$ стек являється порожнім, а при $top>n$ – переповненим. Стек складається з наступних елементів: $S_1, S_2, \ldots, S_i$, де S_1 – дно стеку, а S_i – вершина.

Реалізація операцій додавання елементів у стек та їх видалення з нього зображена на рис. 2.6.

$S \Leftarrow x$	$x \Leftarrow S$
$top \leftarrow top + 1$ **if** $top > n$ **then** // переповнення стеку **else** $S_i \leftarrow x$	**if** $top = 0$ **then** // стек порожній **else** { $x \leftarrow S_i$ $top \leftarrow top - 1$ }

Рисунок 2.6 – Алгоритми додавання та видалення елементів стеку

Розглянемо також приклад роботи зі стеком, в якому містяться елементи цілочисельного типу даних (рис. 2.7).

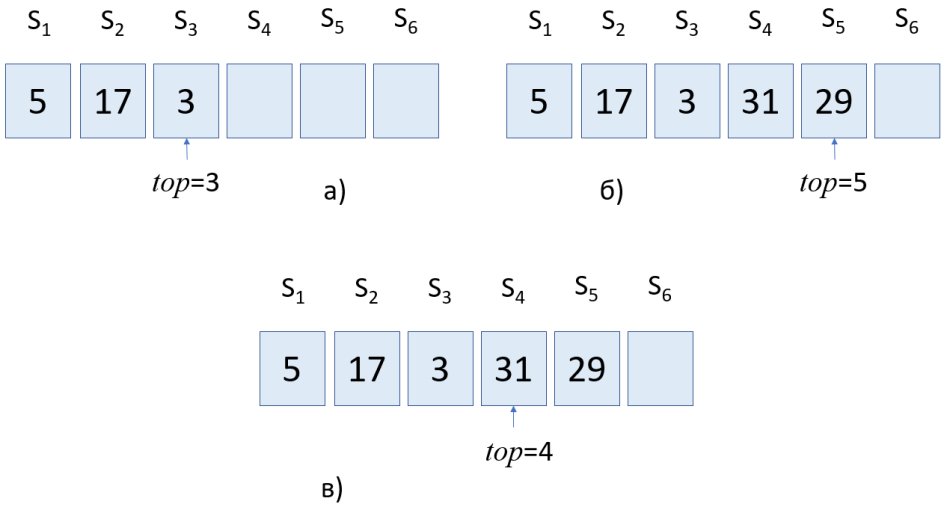


Рисунок 2.7 – Приклад реалізації стеку: а) 3 елементи, вершина *top*=3; б) 5 елементів, вершина *top*=5; в) 5 елементів, вершина *top*=4.

Глибина стеку складає шість елементів, у ньому міститься три значення, вершина знаходиться на елементі з індексом 3 (*top*=3). Даний варіант стеку зображено на рис. 2.7, а. Проведемо дві операції додавання нового елементу до поточного стеку, а саме: $S \Leftarrow 31$ та $S \Leftarrow 29$ (рис. 2.7, б). Вершиною стеку стане елемент *top*=5. У якості останньої операції для прикладу проведемо видалення елементу $29 \Leftarrow S$, що призведе до зміщення вершини стеку в позицію *top*=4 (рис. 2.7, в).

З даного прикладу можна зробити висновок, що операція видалення не стирає елемент зі стеку, а лише зміщує положення його вершини. Це означає, що дані, які були записані в стек не можна видалити безслідно, – вони існують до тих пір, поки існує сам стек. Проте їх можна перезаписати, наприклад, додавши в наведений стек ще один елемент $S \Leftarrow 13$. Подібне обмеження диктується специфікою роботи з комірками оперативної пам'яті. Саме тому необхідно уважно проектувати алгоритми, щоб ненароком не зчитати з пам'яті обривок некоректних даних, особливо при роботі з послідовною реалізацією стеку.

Незважаючи на це, перевагою такого подання стеку є менша кількість операцій, які необхідні для роботи з комірками пам'яті. Це пов'язано з тим, що всі комірки оперативної пам'яті йдуть послідовно і всі операції проходять лише над сусідніми елементами.

З іншого боку, одним з недоліків реалізації стеку у вигляді масиву є його фіксований розмір, вихід за межі якого призведе до переповнення та виникнення помилки у більшості мов програмування. Цей недолік є наслідком протиріччя між динамічною поведінкою стеку та статичною поведінкою масиву. Вирішити дану проблему для створення стеку можна за допомогою використання динамічної структури даних, а саме: зв'язного списку.

У такому випадку поле *next* нижнього елементу стеку має значення Ω, а функцію вершини стеку *top* виконує поле *list* зв'язного списку. При цьому

показником відсутності в стеку елементів є рівність *top*= Ω. Алгоритм реалізації операцій додавання $S \Leftarrow x$ та видалення $x \Leftarrow S$ елементів зі стеку на базі зв'язного списку зображено на рис. 2.8 та рис. 2.9.

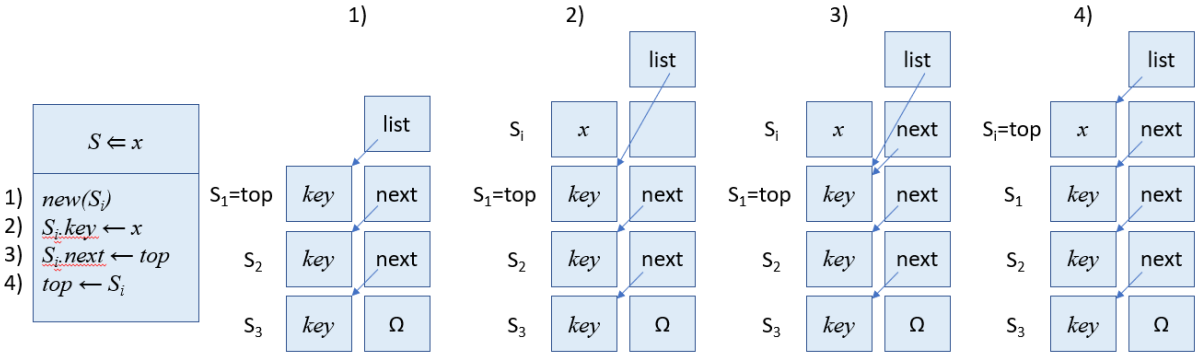


Рисунок 2.8 – Алгоритм додавання елементів динамічного стеку

```
x ⇐ S
if top= Ω
   then // стек порожній
   else { x ← top.key
          Si ← top
          top ← top.next
          dispose(Si) }
```

Рисунок 2.9 – Алгоритм видалення елементів динамічного стеку

Черги

Чергою називається динамічна структура даних з двома точками доступу, а саме: початковою (голова) і кінцевою (хвіст). Новий елемент послідовності завжди додається в кінець черги, видаляється з її початку, а решта елементів черги є недоступними. Це означає, що черга працює за принципом «першим прийшов – останнім пішов» (*англ. First In First Out, FIFO*).

У подальшому у псевдокоді черга буде позначатися символом «Q», операція додавання елементу x до черги буде задаватися як $Q \Leftarrow x$, а операція видалення – $x \Leftarrow Q$.

Реалізація черги у вигляді статичної структури потребує спеціальних прийомів. Це пов'язано з тим, що елементи в черзі з'являються на одному кінці, а зникають на протилежному – рух відбувається в сторону правої межі. У такому випадку елементи черги можуть перейти цю межу, навіть якщо в лівій частині масиву буде достатньо місця для розміщення елементів (рис. 2.10).

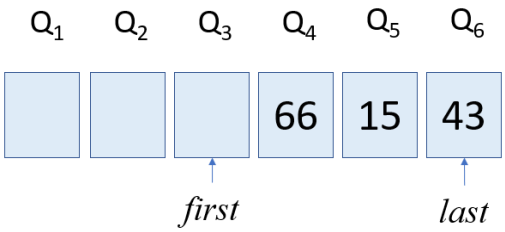


Рисунок 2.10 – Приклад черги

Для вирішення цієї проблеми масив потрібно згорнути в кільце. Для цього будемо використовувати операцію *mod* – обчислення залишку від ділення. У такому випадку для черги $Q = \{Q_1, Q_2, \ldots, Q_n\}$ виділяється масив з n елементів і вважається, що Q_1 йде за Q_n. Якщо використовувати змінну f (*first*) як вказівник позиції, яка знаходиться перед початком черги, а змінну l (*last*) – як вказівник на кінцевий елемент. У такому випадку черга представлятиметься як $Q = \{Q_{f+1}, Q_{f+2}, \ldots, Q_l\}$ (елемент з індексом f містить лише посилання на f+1), а рівність $f = l$ буде свідчити про те, що черга порожня.

Реалізацію операцій додавання та видалення елементів наведено на рис. 2.11. Переповнення черги призводить до виникнення помилки, а відсутність елементів – до закінчення алгоритму. Також варто звернути увагу на причину, з якої відбувається переповнення черги. Як було сказано вище, для

реалізації послідовності елементів Q = {Q_1, Q_2, …, Q_n} потрібно на один елемент більше для зберігання посилання на перший елемент послідовності, тобто Q = {Q_0,Q_1, Q_2, …, Q_n}. Таким чином, фактична довжина масиву складає *n*+1, але робоча – *n*. Хоч в масиві і присутня порожня комірка, але додавання *n*+1-го елемента призводить до переповнення черги.

$Q \Leftarrow x$	$x \Leftarrow Q$
$l \leftarrow (l+1)$ **mod** n **if** $f = l$ **then** // переповнення черги **else** $Q_l \leftarrow x$	**if** $f = l$ **then** // черга порожня **else** { $f \leftarrow (f+1)$ **mod** n $x \leftarrow Q_f$ }

Рисунок 2.11 – Алгоритм додавання та видалення елементів з черги

На рис. 2.12 зображено приклад реалізації черги на основі масиву (довжина черги дорівнює 5) та процес її зміни в часі. У своєму початковому стані черга містить всього два елементи (рис. 2.12, а). Додамо до неї ще два елементи за допомогою операції $Q \Leftarrow x$ (рис. 2.12, б). При цьому голова не зміститься і буде знаходитися в позиції Q_3, а хвіст перейде з позиції Q_5 у позицію Q_1. Після використання операції виключення елемента з черги $x \Leftarrow Q$ (рис. 2.12, в) значення Q_4 все ще залишається в масиві, але в черзі вже не враховується. Елемент Q_4 стає головою черги.

Недоліком реалізації черги у вигляді послідовного розподілу (масиву) є його фіксована довжина. Сформована на базі зв’язного списку черга дає змогу уникнути даної проблеми.

У такому випадку вузол черги буде складатися з поля *key*, в якому міститься значення елементу, і поля *next*, в якому міститься посилання на наступний елемент черги. Перший елемент зв’язного списку відповідатиме за

вказівник голови черги *f* (*first*), але додатково потрібен ще один вказівник на хвіст черги. Черга вважається порожньою при $f = \Omega$.

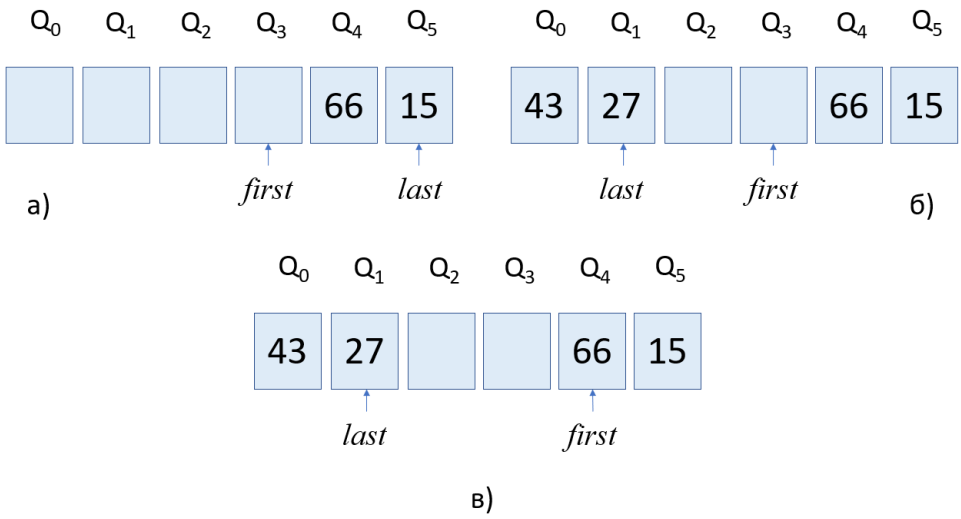


Рисунок 2.12 – Приклад реалізації черги: а) 2 елементи; б) 4 елементи; в) 3 елементи.

Реалізація черги спрощується, якщо використовувати для її створення зв'язний список із заголовком. У даному випадку поле *key* заголовка не використовується, а поле *next* посилається на перший елемент черги. Порожня черга буде складатися лише з одного вузла – заголовка. Критерієм порожнечі черги може виступати або рівність $f.next = \Omega$, або $f = l$. Операції додавання та видалення елементів зображено на рис. 2.13.

$Q \Leftarrow x$	$x \Leftarrow Q$
new(l.next) $l \leftarrow l.next$ $l.key \leftarrow x$ $l.next \leftarrow \Omega$	***if*** $f = l$ ***then*** // *черга порожня* ***else*** { $x \leftarrow f.next.key$ $Q_i \leftarrow f$ $f \leftarrow f.next$ *dispose*(Q_i) }

Рисунок 2.13 – Алгоритм додавання та видалення елементів з черги

Контрольні запитання.

1. У лекції наведено алгоритм включення першого елементу в динамічний стек. Як виглядатиме алгоритм для додавання елементу в будь-яке місце зв’язного списку?

2. Чим масив відрізняється від зв’язного списку?

3. Що таке стек? Як відбувається додавання та віднімання його елементів?

4. Що таке черга? Чим черга відрізняється від стеку?

5. Яким буде алгоритм додавання елементу у двобічно зв’язаний список?

6. Яким буде алгоритм видалення елементу з середини (будь-яке місце, яке не є останнім) масиву?

7. Яким буде алгоритм додавання першого/будь-якого елемента в зв’язаний список? Яким буде алгоритм видалення першого/будь-якого елемента з нього?

Лекція №3

Хеш-таблиці

Однією з суттєвих переваг використання хеш-таблиць є швидкий пошук значень. Ця можливість забезпечується завдяки спеціальній організації збережених даних, яка, на відміну від впорядкованих списків, дозволяє легко розрахувати місцезнаходження необхідного елементу.

Концептуально хеш-таблицю можна розглядати як звичайний масив, тому що їх перевагою є швидкий доступ до випадкового елементу: якщо потрібно дізнатися, наприклад, що знаходиться в комірці з індексом 33, то треба напряму звернутися до даної позиції. Дана операція виконується за фіксований час. Якщо потрібно перезаписати дані в комірці 78, то це також виконується за фіксований час.

Розглянемо наступний приклад: необхідна структура даних для зберігання персональних даних студентів одного факультету. Припустимо кожен студент має неповторюваний порядковий номер. Тоді його дані можна зберегти у вигляді єдиного об’єкту, звернувшись до комірки масиву з відповідним індексом. У такому випадку, навіть якщо студенти змінюються з часом, вимоги до пам’яті залишаються незначними, а операції запису, пошуку та видалення виконуються за фіксований час.

Розглянемо інший варіант: персональні дані студента необхідно зберігати за його прізвищем, а не порядковим номером. У такому випадку, як і раніше, можна використовувати масив, в якому комірки будуть індексуватися за всіма можливими прізвищами. Для доступу до даних студента Петренка треба звернутися до комірки в позиції «Петренко». У даному випадку необхідний для реалізації подібної структури масив є занадто великим через

різноманіття можливих прізвищ. Іншим недоліком є співпадіння прізвищ декількох студентів. Саме тому використання стандартного масиву є недоцільним. Альтернативною структурою даних, яка повторює основні функціональні можливості масивів (з певними припущеннями) з постійними у часі операціями додавання, видалення та пошуку, є хеш-таблиця. Вона також дає змогу використовувати об'єм оперативної пам'яті, пропорційний кількості об'єктів, які необхідно зберегти.

Основи хеш-таблиць

Основна задача хеш-таблиць – з'єднати дані з комірками пам'яті. Дуже часто такий зв'язок утворюється між значенням ключа (англ. *Key*), наприклад, порядковим номером (прізвищем, реєстраційним номером облікової картки платника податків, тощо) та набагато більшим набором даних. Саме тому інколи хеш-таблиці називають словниками або асоціативними масивами. У більшості випадків ключ являє собою натуральне число, але якщо він представляється у вигляді послідовності символів, то, наприклад, у Java існує метод hashCode(), який виконує необхідне перетворення.

Процес перетворення значень відповідних ключів до необхідного вигляду називається **хешуванням**. Результатом цього процесу є хеш-функція, яка для кожного нового елементу визначає натуральне число – індекс комірки базової структури даних. Різні хеш-функції будуть по-різному впливати на швидкодію алгоритму. Це залежить від ситуації та даних, якими треба оперувати. Існує ряд розроблених та відкритих для широкого кола користувачів хеш-функцій, ефективність яких неодноразово доведено, наприклад, SpookyHash, MurmurHash3 та MD5. Розроблення власної хеш-функції є дуже складним процесом, тому не рекомендується.

У деяких випадках значення ключів дуже схожі і погані хеш-функції можуть розподіляти їх у одні і ті ж самі комірки пам’яті. Наприклад, якщо необхідно зробити записи про студентів з прізвищами Петренко та Петренков, то в ідеальному випадку хеш-функція має їх зв’язати з двома різними наборами комірок оперативної пам’яті. Але в реальності може генеруватися декілька безглуздих значень, які наштовхують на думку, що дійсне значення ключа було розділено на частини.

Якщо хеш-таблиця містить дуже велику кількість даних, то рано чи пізно виникне випадок, коли два (чи більше) ключів виявляться зв’язаними з одними і тими ж комірками пам’яті. Дане явище називається **колізією**. У такому випадку потрібно розробити політику прийняття рішень у разі виникнення колізій. Показником ймовірності виникнення колізій у хеш-таблиці є коефіцієнт її заповнення. При додаванні ключа до заповненої на 90% таблиці більше шансів зв’язати другий ключ з комірками пам’яті, які вже використовуються, ніж до заповненої на 10%.

Колізії викликають неоднозначність визначення положення елементу в хеш-таблиці. В ідеальному випадку потрібно розробити хеш-функцію таким чином, щоб ймовірність виникнення колізій прямувала до нуля. Але на практиці колізії неминучі у відповідності до принципу Діріхле: не можна в масиві скінченої довжини записати в окрему комірку кожне з нескінченної множини значень – існуватиме мінімум одна комірка, в яку необхідно буде розмістити два не пов’язаних між собою значення. Іншим аргументом проти існування ідеальної хеш-функції є парадокс днів народження. Таким чином, якою б не була хеш-функція, але на практиці завжди виникатимуть колізії.

Отже, для якісної реалізації хеш-таблиць необхідно виконати наступні умови:

- правильно реалізувати структуру для збереження даних;

- правильно реалізувати хеш-функцію для зв’язування комірок пам’яті з обраною структурою даних;
- розробити політику прийняття рішень у випадку виникнення колізій.

Мінімальні функціональні можливості, які вже роблять хеш-таблицю дуже корисною, – додавання нових елементів та пошук серед збережених. Дуже часто можливість видалення хешованого ключа є відсутньою, але вона також є дуже корисною і рекомендується до реалізації у кожній хеш-таблиці.

Під час виконання алгоритму хеш-таблиця може повністю заповнитися або дійти до того стану, коли ймовірність виникнення колізій буде занадто великою. У такому випадку треба удосконалювати алгоритм для визначення моменту, коли хеш-таблиця переповнюється, щоб збільшити її або навпаки зменшити у разі низького відсотку використання її об’єму (наприклад, використовується 100 записів із 1 000 000).

Найпростішим способом змінити розміри хеш-таблиці є створення нової з бажаними розмірами та розв’язання усіх елементів з початкової структури даних. Цей спосіб є найбільш використовуваним, але також існує альтернатива, а саме: пряме зв’язування.

Пряме зв’язування

Значення ключів у хеш-таблиці з прямим зв’язуванням (рис. 3.1) зберігаються у спеціальних наборах записів – блоках (зображені у вигляді порожніх квадратів на рис. 3.1). Кожен блок є вершиною зв’язного списку, у якому містяться зв’язані з блоком елементи.

Дуже часто блоки розташовуються таким чином, що для визначення їх ключа достатньо простої хеш-функції. Наприклад, якщо є *n* блоків з ключами у вигляді чисел, то ключ *k* можна зв'язати з блоком під номером *k mod n*.

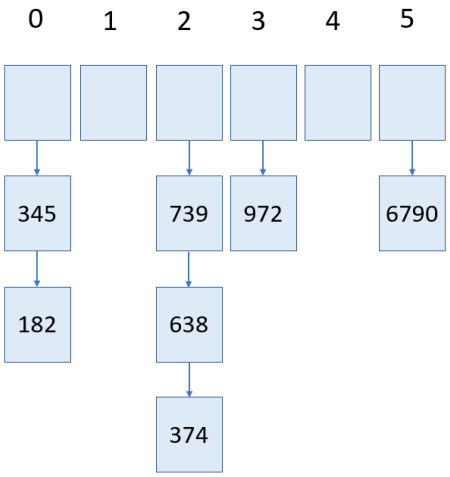


Рисунок 3.1 – Хеш-таблиця з прямим зв'язуванням

Для того, щоб до хеш-таблиці додати ключ, потрібно зв'язати його з блоком через хеш-функцію, а потім вставити нову комірку у верхню частину списку. Кожна з цих двох операцій займає фіксований час O(1) (дану тему більш детально буде розглянуто в наступній лекції), тобто швидко виконується. За замовчуванням хеш-таблиця не має містити двох однакових значень ключа. Якщо є *n* елементів і *b* блоків, які рівномірно розподілені, то зв'язний список кожного блоку матиме приблизно *n/b* елементів. У цьому випадку для перевірки наявності нового елементу у відповідному блоці знадобиться O(*n/b*) кроків, а для додавання елементу в хеш-таблицю – O(1) + O(*n/b*) = O(*n/b*) кроків.

Пришвидшити операцію пошуку елементів у хеш-таблиці можна, якщо хеш-ключі зв'язаних списків будуть послідовно впорядковані. Таким чином, якщо алгоритм пошуку дійде до значення, яке більше за шуканий

ключ, то зробить висновок, що ключ відсутній і не переглядатиме всю структуру даних до кінця.

Для пошуку необхідного елементу потрібно провести хешування ключа та визначити в якому з блоків він міститься. Після цього пошук вже проходитиме у відповідному зв'язному списку до тих пір, доки не знайдеться шуканий елемент або буде досягнуто кінець списку. У останньому випадку робиться висновок про відсутність шуканого елемента в хеш-таблиці. Як і у випадку додавання елементу, потрібно виконати приблизно O(*n/b*) кроків.

Видалення елементів з хеш-таблиці проходить аналогічним чином і для виконання операції вимагає O(*n/b*) кроків.

Недоліком створення хеш-таблиці на базі зв'язного списку є те, що у випадку значного розростання розмірів списків таблиці збільшиться також і час виконання розглянутих операцій.

Хоч хеш-таблиця на базі зв'язного списку і може змінювати свої розміри в обидві сторони за необхідності, але якщо її зв'язані списки стануть занадто довгими, то сильно зросте і час виконання основних розглянутих операцій. У цьому випадку доцільно буде збільшити кількість блоків хеш-таблиці, так як вона легко може змінювати свої розміри в обидві сторони за необхідності. При рехешуванні (повторному хешуванні) не потрібно буде проводити пошук дублікатів, тому з цією операцією можна впоратися за час O(*n*).

Відкрита адресація

Основна перевага прямого зв'язування полягає у тому, що загальна кількість значень не залежить від кількості блоків. Проте, як сказано вище,

при збільшенні кількості блоків зростає і час проведення операцій. Також у ході виконання алгоритмів можуть накопичуватися порожні блоки, які не використовуються і займають місце в оперативній пам’яті.

Даних недоліків можна позбутися завдяки використанню відкритої адресації, при якій значення елементів зберігаються в масиві, а хеш-функція подається у вигляді розрахунку. Наприклад, для масиву з довжиною n проста хеш-функція може зв’язати значення ключа k з елементом $k \bmod n$. При цьому для різних варіантів відкритої адресації використовуються різні хеш-функції та політики прийняття рішень у разі виникнення колізій. У загальному випадку задля уникнення колізій для кожного елементу послідовності підбирається декілька комірок масиву. Якщо перша з них зайнята, то алгоритм намагається записати дані в другу і так до тих пір, поки дані запишуться або не залишиться вільного місця.

Цей набір комірок масиву, які алгоритм обирає для спроби запису елемента послідовності даних називається **пробною послідовністю**. При використанні відкритої адресації наповненість хеш-таблиці оцінюється за середньою довжиною пробної послідовності. В ідеальному випадку довжина пробної послідовності не перевищує 2 комірок масиву. Більші числа свідчать про велику наповненість хеш-таблиці.

У деяких випадках при некоректній реалізації алгоритму прийняття рішень при виникненні колізій за допомогою використання пробних послідовностей може не знайтися вільного місця для запису елементу, навіть якщо воно в дійсності є.

При правильній реалізації алгоритму заповнення хеш-таблиці відкрита адресація має високу швидкодію. У випадку, коли пробна послідовність дорівнює 1 або 2 коміркам, операції пошуку та додавання елементів виконуються за фіксований час, але продуктивність знижується при високому відсотку наповненості хеш-таблиці. У останньому (найгіршому)

випадку час виконання операцій пошуку та додавання складатиме O(*n*). Як і у випадку прямого зв'язування, зменшення коефіцієнту наповненості хеш-таблиці може вирішити дану проблему.

Ситуація з видаленням елементів з хеш-таблиці є більш складною, ніж при прямому зв'язуванні: одна і та ж комірка масиву може бути частиною декількох різних пробних послідовностей. Якщо одну з них порушити, то пов'язаний з нею елемент вже не вийде знайти.

Наприклад, у масиві *M* присутні два елементи, а саме: *X* та *Y*. Пробна послідовність $[M_i, M_x]$ дозволяє визначити позицію для запису значення *X*, а $[M_x, M_y]$ – для запису значення *Y*. Припустимо, що елемент *X* видалено з масиву. Якщо алгоритм тепер звернеться до пробної послідовності $[M_x, M_y]$, то побачить, що перша позиція M_x вільна, і помилково вирішить, що елемент Y у масиві відсутній.

Дану проблему можна вирішити, позначивши спеціальним чином видалені та порожні елементи. Це можуть бути, наприклад, границі певного типу даних (-2 147 483 648 та 2 147 483 647 для **int** у Java), які є недосяжними у 99,9% випадків. Такий підхід негативно вплине на пошук елементів, але операція додавання буде виконуватися за той же час, що і раніше.

Інші варіанти прийняття рішень

На практиці використовується декілька різних варіантів політики прийняття рішень при виникненні колізій. Один з них – *лінійне пробування (зондування)*.

Для формування пробної послідовності за даним методом хеш-функція визначає початкову комірку масиву, яку необхідно перевірити на наявність даних, а потім, у разі знаходження заповненої комірки, переходить до

іншої, індекс якої є більшим на одиницю. Послідовний перебір комірок масиву відбувається до того часу, поки буде знайдено порожню комірку або кінець масиву. Можна реалізувати алгоритм таким чином, що у разі досягнення останньої комірки масиву, пошук переходить на комірку з індексом 0, тобто використовується циклічна структура даних (див. циклічні списки).

Розглянемо наступний приклад: хеш-таблиця містить 10 елементів, а операція хешування зв'язує ключ *k* з коміркою масиву з індексом *k mod* 10, де *k mod* 10 – хеш-функція. Тоді пробна послідовність для значення 59 послідовно перевіряє комірки з індексами 9, 0, 1, 2 і так далі.

На рис. 3.2 зображено масив довжиною 10 елементів, в якому вже міститься декілька значень. Для запису в нього значення 65 з використанням лінійної пробної послідовності необхідно зв'язати дане значення з коміркою з індексом 5 (65 *mod* 10), але в ній вже міститься інший елемент. Виникає колізія, тому алгоритм переходить до наступної комірки з індексом 6, яка теж заповнена. Наступною для перевірки є комірка з індексом 7, яка не містить ніяких значень, тому в неї і записується поточне.

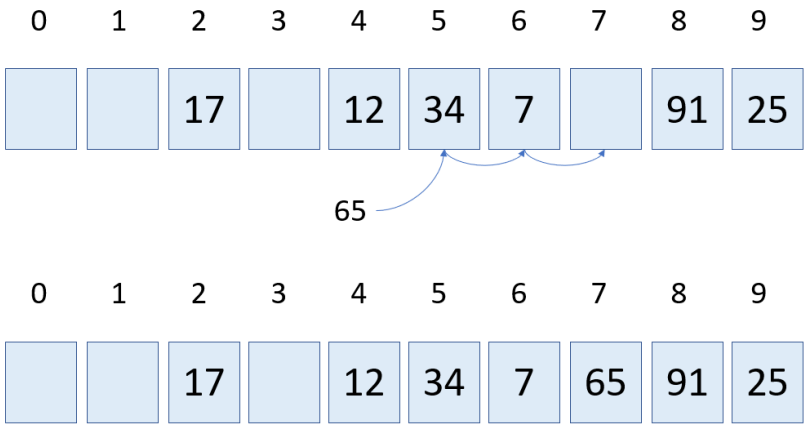


Рисунок 3.2 – Лінійне пробування

Простота – це основна перевага даного методу. Пробна послідовність перевірить кожну комірку і запише значення у вільне місце, якщо воно буде

в масиві. Недоліком лінійного пробування є виникнення первинної кластеризації.

Кластер – це неперервна послідовність даних, яка об'єднана спільною логікою.

Суть первинної кластеризації полягає в тому, що з часом кількість кластерів у хеш-таблиці зростає і вони постійно об'єднуються у більші. Тому велика кількість кластерів призводить до формування довгих пробних послідовностей. У результаті при додаванні нового елементу пробна послідовність має пройти по значенням до кінця кластеру, що знижує швидкодію алгоритму особливо при високому коефіцієнті заповнення.

Отже, великі кластери виникають, коли нові значення записуються в комірки масиву, які знаходяться в кінці кластеру, збільшуючи його. Знизити ймовірність цього явища можна завдяки використанню іншого варіанту прийняття рішень при виникненні колізій – *квадратичного пробування (зондування)*.

Створення пробної послідовності відбувається схожим до попереднього випадку чином з єдиною відмінністю – крок підноситься до квадрату. Наприклад, лінійне пробування проводило пошук за наступним набором індексів {i, i+1, i+2, i+3, …}, а квадратичне – {i, $i+1^2$, $i+2^2$, $i+3^2$, …}. У такому випадку при виникненні колізії, коли перший індекс *i* двох пробних послідовностей співпаде, два елементи будуть зв'язані з різними позиціями, індекси яких значно рознесені між собою. У загальному випадку ці два елементи будуть належати різним кластерам. Також варто зауважити, що у якості кроку не обов'язково використовувати одиницю, як у прикладах вище. У обох розглянутих варіантах пробування початковий крок може бути довільним числом, яке не змінюється під час роботи алгоритму. Наприклад, якщо крок дорівнюватиме 7, то набори індексів виглядатимуть наступним чином:

{i, i+7, i+14, i+21, …} для лінійного пробування та {i, $i+7^2$, $i+14^2$, $i+21^2$, …} для квадратичного.

Розглянемо наступний приклад (рис. 3.3): хеш-таблиця містить 13 елементів і один малий кластер з трьох значень, квадратична пробна послідовність має крок 1 (один), операція хешування зв'язує ключ *k* з коміркою масиву з індексом *k mod* 10. Для додавання в хеш-таблицю значення 65 необхідно записати його в комірку з індексом 65 *mod* 10 (i = 5), яка вже зайнята. Тому алгоритм далі перевіряє комірки за індексами квадратичної пробної послідовності далі і знаходить вільне місце в комірці з індексом 9. Додавання елементу в знайдену комірку призводить до формування другого малого кластеру в хеш-таблиці.

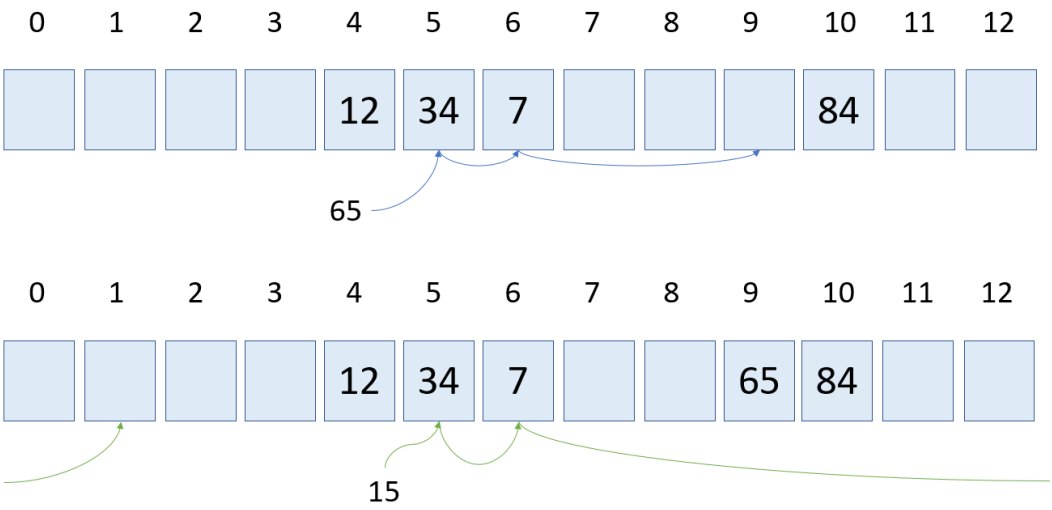


Рисунок 3.3 – Квадратичне пробування з циклічною базовою структурою даних

При спробі розміщення в хеш-таблиці ще одного значення 15 виникає колізія, бо для нього також *i* = 5. Пробна послідовність знаходить для нього вільне місце в комірці з індексом 1 (один), що не призводить до утворення нового кластеру.

Якщо порівнювати послідовне та квадратичне пробування, то друге у більшості випадків матиме в середньому менші пробні послідовності при заповненні хеш-таблиці. Іншою перевагою є те, що воно вирішує проблему первинної кластеризації.

Серед недоліків варто зазначити наявність вторинної кластеризації – явища, при якому значення, що зв'язані з однаковим початковим індексом комірки масиву, отримують одну і ту ж (як у наведеному вище прикладі, але часто в декілька разів довшу) пробну послідовність. Це призводить до появи впорядкованих груп кластерів.

Також недоліком квадратичного пробування є те, що воно може не знайти вільну комірку масиву, навіть якщо така присутня. Це зумовлюється сильним зростанням значень сусідніх індексів у пробній послідовності.

Ще одним варіантом вирішення проблем з колізіями є використання *псевдовипадкового пробування*. Воно є схожим на два розглянутих вище варіанти з відмінністю у шляхах вибору кроку. Псевдовипадкове пробування використовує наступний набір індексів $\{i, i + p \cdot 1, i + p \cdot 2, i + p \cdot 3, \dots\}$, де p – значення, яке генерується за допомогою псевдовипадкової функції.

Псевдовипадкове пробування має ідентичні квадратичному пробуванню переваги та недоліки.

Розглянуті вище політики прийняття рішень при виникненні колізій мають спільний та значний недолік, а саме: наявність вторинної кластеризації. Вона виникає через те, що два окремих елементи з одним і тим самим початковим індексом комірки масиву мають однакові пробні послідовності. Отже, для вирішення цієї проблеми необхідно формувати для таких елементів різні пробні послідовності.

Це можна зробити, наприклад, з використанням операції *подвійного хешування*. Даний метод передбачає використання двох хеш-функцій замість однієї: перша визначає початкову позицію пробної послідовності, а

друга відповідає за визначення окремого кроку для кожного елементу. Як правило, за основу береться псевдовипадкове пробування, але для елементів A та B воно виглядатиме відповідно $\{i, i + p_A \cdot 1, i + p_A \cdot 2, i + p_A \cdot 3, \dots\}$ та $\{\mathrm{i}, \mathrm{i} + \mathrm{p}_B \cdot 1, \mathrm{i} + \mathrm{p}_B \cdot 2, \mathrm{i} + \mathrm{p}_B \cdot 3, \dots\}$, де (у загальному випадку) $p_A \neq p_B$. Як видно з даного прикладу, при однаковому початковому індексі *i* елементи *A* та *B* матимуть різні пробні послідовності.

Отже, подвійне хешування вирішує обидві проблеми, а саме: первинну та вторинну кластеризацію. Проте даний метод також не позбавлений недоліків. Як і у попередніх випадках, він може пропускати порожні комірки масиву, в які можна записати дані.

Упорядковане хешування

Враховуючи велику кількість програмного забезпечення, яке сьогодні розробляється у різних сферах науки та техніки, є широке різноманіття вирішуваних завдань. Одним з них є швидкий пошук необхідних значень у більш-менш статичній структурі даних (мається на увазі, що операції пошуку значно перевищують додавання та видалення). Прикладами таких програм можуть бути словники, адресні книги, каталоги товарів та подібні таблиці даних.

Пришвидшити пошук у хеш-таблиці можна у випадку використання прямого зв’язування, якщо всі дані попередньо впорядкувати за певним принципом, наприклад, від меншого до більшого, від «А» до «Я», тощо. У такому випадку алгоритм буде зупинятися, коли знаходить необхідне значення і не буде перебирати всю структуру до кінця. Сортування можна використовувати також і для безпосередньої побудови хеш-таблиць. Створюючи її подібним чином, можна значно прискорити пошук елементів.

Псевдокод, який зображено на рис. 3.4., демонструє алгоритм пошуку елементів у впорядкованій хеш-таблиці.

```
function findValue(array[], key){
    i ← початковий індекс
    while (true) do
            // перевірка існування елементу
        if (array[i] == key) then
            return i
        end if
            // перевірка того, чи розглядався раніше цей варіант
        if (array[i] > key) then
            return -1
        end if
            // перевірка існування вільного місця
        if (array[i] == EMPTY) then
            return -1
        end if
        i ← наступна позиція пробної послідовності
    end while
}
```

Рисунок 3.4 – Псевдокод пошуку даних у впорядкованій хеш-таблиці

У розглянутому прикладі функція повертає положення ключа в масиві або -1, якщо не знаходить його. Від’ємне значення не належить до множини натуральних, до якої входять усі індекси, тому свідчить про аварійне завершення методу (функції).

Припустимо, що при додаванні потрібно сортувати елементи хеш-таблиці від найменшого до найбільшого. У такому випадку перший та всі наступні індекси пробної послідовності, ключа який додається, не мають бути меншими за відповідні індекси попереднього доданого ключа. Тільки за такої умови можна буде проводити швидкий пошук. Але для реалізації цього випадку необхідно заздалегідь знати необхідні для внесення даних, а також принцип їх організації. На практиці такого досягти можна не часто.

Існує спосіб реалізації впорядкованої хеш-таблиці без необхідності урахування порядку додавання в неї елементів. Спочатку у відповідності до пробної послідовності проводиться пошук вільної комірки. Якщо у першій комірці пробної послідовності ключ виявляється більшим за значення нового елементу, то цей ключ поступається своїм місцем і проходить рехешування. Якщо у ході рехешування він потрапляє на ключ зі ще більшим значенням, то займає його позицію, а більший ключ аналогічно проходить рехешування. Такі порівняння та перестановки будуть продовжуватись до того часу, коли буде знайдено порожню комірку.

Якщо у першій комірці пробної послідовності ключ виявляється меншим за значення нового елементу, то відбувається перехід до наступного значення пробної послідовності. Так відбувається до того часу, коли виникне одна з наступних ситуацій: 1) знайдеться вільна комірка – значення запишеться; 2) знайдеться ключ з більшим значенням – подібну ситуацію описано вище.

Псевдокод, який зображено на рис. 3.5., демонструє розглянутий алгоритм дій.

```
function addValue(array[], key){
    i ← початковий індекс
    while (true) do
            // перевірка існування вільного місця
        if (array[i] == EMPTY) then
            array[i] ← key
            return i
        end if
            // перевірка того, чи новий ключ більше за існуючий
        if (array[i] > key) then
            temp ← array[i]
            array[i] ← key
            key ← temp
        end if
        i ← наступна позиція пробної послідовності
    end while
}
```

Рисунок 3.5 – Псевдокод додавання елементу в хеш-таблицю

Наведений метод реалізовує заміну більших значень ключа меншими. Єдиним сумнівним моментом стає нове значення ключа, тому що воно продовжує слідувати за пробною послідовністю початкового ключа. Саме тому повне сортування хеш-таблиці, у залежності від обраної хеш-функції, може зайняти певний час (певну кількість доданих елементів).

Дане твердження залишається справедливим при використанні лінійного та псевдовипадкового пробування, а також подвійного хешування. Останній метод вимагає додаткових операцій по визначенню кроку для кожного нового значення ключа, яке виявилося більшим за поточне, але він є цілком працездатним. З іншого боку, квадратичне пробування майже неможливо використати для коректної реалізації наведеного методу.

Контрольні запитання.

1. Чим є хеш-таблиця?
2. У чому полягає суть прямого зв’язування?
3. У чому полягає суть відкритої адресації?
4. Що таке лінійне пробування? Які його недоліки?
5. Що таке квадратичне пробування? Які його недоліки?
6. Що таке псевдовипадкове пробування? Які його недоліки?
7. Що таке хешування та як визначається хеш-функція?
8. Як вирішуються проблеми первинної та вторинної кластеризації?
9. Якщо швидкодія є пріоритетом, а обсяг пам’яті не є важливим, що краще обрати: пряме зв’язування чи відкриту адресацію?
10. Що є показником якості хеш-функції?
11*. Чому розглянута в прикладі до рис. 3.3 хеш-функція є поганою? Обґрунтувати відповідь розрахунками.

12*. Які зміни необхідно внести в псевдокод, зображений на рис. 3.4 та рис. 3.5, щоб він відповідав принципам SOLID?

* – питання підвищеної складності для самостійного опрацювання

Лекція №4

Оцінювання часу виконання алгоритму

Будь-яка поставлена перед розробником задача може мати декілька варіантів рішення, тобто різні алгоритми. Обрання серед них найбільш ефективного не є тривіальним завданням: необхідно у кожній окремій ситуації порівнювати характеристики якості різних алгоритмів та на основі результатів робити відповідні висновки.

Вивчення існуючих на сьогодні алгоритмів дозволить правильно їх застосовувати для найбільш оптимального рішення поставлених завдань. Це пов’язано з тим, що алгоритми можуть по-різному реагувати на різні набори даних. Проте, якщо конкретний алгоритм виявиться не ефективним у певній ситуації, він дозволить по-іншому поглянути на поставлену задачу або знайти несподіваний спосіб його застосування.

Асимптотична складність алгоритму

Одними з найбільш важливих критеріїв оцінки якості будь-якого алгоритму є необхідні для його виконання час та об’єм оперативної пам’яті. Складність кожного завдання оцінюється по вхідним даним. Вони пов’язуються з деяким числом, яке виступає у ролі показника складності та називається розміром задачі (завдання). Залежність часу виконання алгоритму від розміру задачі називається часовою складністю алгоритму. Тобто часова складність алгоритму – це оцінка необхідного на виконання алгоритму часу при заданих вхідних даних. При відомій часовій складності алгоритму

можна відразу сказати чи буде він ефективним для вирішення конкретного завдання, не реалізовуючи його.

Об’ємну складність – оцінку необхідної на виконання алгоритму оперативної пам’яті при заданих вхідних даних – можна оцінити подібним чином. У більшості випадків об’ємна складність є очевидною і не потребує додаткової оцінки, тому основну увагу буде приділено саме часовій складності алгоритмів.

Найчастіше для розрахунку часової складності будь-якого алгоритму проводиться оцінка асимптотики зростання часу його роботи при нескінченно великих вхідних даних (якщо розмір вхідних даних позначити за n, то оцінка проходить при $n \to \infty$). Така оцінка називається асимптотичною часовою складністю. Якщо асимптотична часова складність одного алгоритму менша за асимптотичну часову складність іншого, то можна зробити висновок про більшу ефективність першого з них. Виключенням можуть бути випадки з малою довжиною вхідних даних.

Швидкість зростання різних функцій можна описати за допомогою асимптотичних позначень. Оскільки розмір вхідних даних та час роботи алгоритму є додатними (натуральними) числами, функції у всіх асимптотичних співвідношеннях також не можуть бути від’ємними.

Якщо функція $f(n)$ зростає з такою ж швидкістю, як і $p(n)$ ($f(n) \sim p(n)$), то вони асимптотично рівні:

$$\lim_{n\to\infty} \frac{f(n)}{p(n)} = 1.$$

Якщо існують такі константи $c_1>0$, $c_2>0$ та $n_0 \geq 0$, що $c_1 \cdot p(n) \leq f(n) \leq c_2 \cdot p(n)$ для всіх $n \geq n_0$, то функція $p(n)$ є асимптотично точною

оцінкою функції $f(n)$ ($f(n) = \Theta(p(n))$). Дане твердження є симетричним, тобто, якщо $f(n) = \Theta(p(n))$, то і $p(n) = \Theta(f(n))$.

Розглянемо наступний приклад: $f(n)=5n^3+3n^2+n+7$, $p(n) = 3n^3+5$. У відповідності до визначення мають виконуватися наступні нерівності: $c_1 \cdot (3n^3+5) \leq 5n^3+3n^2+n+7 \leq c_2 \cdot (3n^3+5)$ для всіх $n \geq n_0$. Розділимо дану нерівність на дві та розглянемо їх окремо.

Перша частина нерівності – це $c_1 \cdot (3n^3+5) \leq 5n^3+3n^2+n+7$. Підставивши у першу частину $n_0 = 0$, отримаємо $c_1 \cdot (3 \cdot 0^3+5) \leq 5 \cdot 0^3+3 \cdot 0^2+0+7$. Якщо спростити дану нерівність, то в результаті буде $c_1 \leq 1{,}4$.

Для другої частини нерівності ($5n^3+3n^2+n+7 \leq c_2 \cdot (3n^3+5)$) аналогічним чином можна визначити, що при $n_0 = 0$ $c_2 \geq 1{,}4$. Не важко перевірити, що при $c_2=1{,}4$ загальна нерівність не матиме сенсу і не виконуватиметься для більшості n_0. Проте, якщо взяти значення більше (будь-яке, при якому умова завжди виконуватиметься), наприклад, $c_2=5$, то при $c_1=1{,}4$ та всіх $n \geq 0$ ($n_0 = 0$) загальна рівність буде справедливою. Отже, можна сказати, що розглянуті функції є асимптотично точними, тобто $f(n) = \Theta(p(n))$.

Розглянемо аналогічний приклад, але змінимо функцію $p(n)$ $p(n)$ з $3n^3+5$ на $3n^3$, а $f(n)$ залишимо такою ж. У даному випадку можна легко побачити, що при $n_0 = 0$ не виконується умова $c_2 > 0$ ($7 \leq c_2 \cdot 0$). При цьому легко перевірити, що можна знайти додатні коефіцієнти ($c_1 > 0$, $c_2 > 0$), за яких виконується умова $c_1 \cdot p(n) \leq f(n) \leq c_2 \cdot p(n)$ для всіх $n \geq n_0$ лише при будь-яких $n \geq 1$ ($n_0 \geq 1$).

Асимптотично точна оцінка функції $f(n)$ має також дві межі, а саме: верхню та нижню. Зазвичай їх розглядають окремо.

Якщо функція $f(n)$ зростає не швидше, ніж $p(n)$ та існують константи $c > 0$ і $n_0 \geq 0$ такі, що $f(n) \leq c \cdot p(n)$ для всіх $n \geq n_0$, то функція $p(n)$ є верхньою межею швидкості росту функції $f(n)$ ($f(n) = O(p(n))$). Це означає, що зі збільшенням n відношення $f(n)/p(n)$ залишається обмеженим.

Якщо функція $f(n)$ зростає не повільніше, ніж $p(n)$ та існують константи $c > 0$ і $n_0 \geq 0$ такі, що $c \cdot p(n) \leq f(n)$ для всіх $n \geq n_0$, то функція $p(n)$ є нижньою межею швидкості росту функції $f(n)$ ($f(n) = \Omega(p(n))$).

Асимптотичне відношення $f(n) = \Theta(p(n))$ виконується для будь-яких двох функцій $f(n)$ та $p(n)$ тільки тоді, коли $f(n) = O(p(n))$ та $f(n) = \Omega(p(n))$. Наприклад, $f(n) = 3n^2+n+7 = \Theta(n^2) = O(n^2) = \Omega(n^2)$.

Якщо виконується рівність

$$\lim_{n\to\infty} \frac{f(n)}{p(n)} = 0,$$

то функція $f(n)$ зростає повільніше, ніж $p(n)$. Це твердження записується як $f(n) = o(p(n))$. Наприклад, $3n^2+n+7 = o(n^3)$, але при цьому $3n^2+n+7 \neq O(n^3)$.

Аналогічно вводиться ще одне твердження. Якщо виконується рівність

$$\lim_{n\to\infty} \frac{p(n)}{f(n)} = 0,$$

то функція $f(n)$ зростає швидше, ніж $p(n)$. Це твердження записується як $f(n) = w(p(n))$. Наприклад, $3n^2+n+7 = w(n)$, але при цьому $3n^2+n+7 \neq \Omega(n)$.

Одним з найважливіших випадків асимптотичних функцій є варіант, коли $p(n)$=1 (наприклад, $O(1)$). Це означає, що дана функція обмежена якоюсь додатною константою та не залежить від розміру вхідних даних n. Якщо говорити про часову складність алгоритму, то запис $O(1)$ свідчить про те, що час виконання є статичним (константою) і не залежить від розміру вхідних даних.

Варто також пам’ятати, що в асимптотичних формулах права їх частина містить менше інформації, ніж ліва. Наприклад, якщо є дві асимптотичні формули $f(n) = O(p(n))$ та $k(n) = O(p(n))$, то при цьому $f(n) \neq k(n)$ у 99,9 % випадків.

Отже, для оцінки ефективності роботи алгоритмів використовується асимптотична часова складність. У більшості випадків на практиці розглядається верхня межа зростання швидкості часової складності як функції від розміру вхідних даних $O(p(n))$. Це пояснюється тим, що даний варіант є найгіршим, а саме: вище цієї границі час виконання алгоритму не підніметься. Інакше кажучи, верхня межа зростання швидкості часової складності є найбільшим часом, який в залежності від розміру вхідних даних може витратити алгоритм на своє виконання. Тобто твердження про те, що час роботи алгоритму асимптотично обмежений зверху квадратичною функцією ($O(n^2)$ – тобто алгоритм не може працювати ще повільніше) є на багато кориснішим з точки зору оцінки ефективності алгоритму, ніж те, що час роботи алгоритму асимптотично обмежений знизу квадратичною функцією ($\Omega(n^2)$ – алгоритм не може працювати ще швидше).

Саме з цієї причини у переважній більшості літературних джерел для аналізу швидкості виконання також використовується О-символіка. Це означає, що запис $f(n) = O(p(n))$ трактується як верхня межа рівності $f(n) = \Theta(p(n))$.

Аналіз алгоритмів у межах О-символіки може включати в себе додаткові операції, наприклад, додавання чи множення. Розглянемо як ці операції впливають на загальну рівність:

1. Якщо деякій математичній функції $f(n)$ необхідно виконати певні дії n разів, то на це їй потрібно буде витратити $O(f(n))$ часу.

2. Якщо алгоритм виконує один код з часом $O(f(n))$, а після нього інший код з часом $O(p(n))$, то загальна продуктивність для функцій $f(n)$ та $p(n)$

буде складати $O(f(n)+p(n))$. Дану асимптотичну часову складність можна спростити до виразу $O(max(f(n),p(n)))$. Правило додавання використовується у тому випадку, коли необхідно визначити порядок зростання часу виконання послідовних фрагментів. Наприклад, $O(n^2+n)=O(n^2)$ або $O(1+n^3)=O(n^3)$.

3. Якщо алгоритм під час виконання однієї ітерації (одного кроку) функції $f(n)$ з часом $O(f(n))$ має витратити $O(p(n))$ часу на виконання функції $p(n)$, то загальна продуктивність для функцій $f(n)$ та $p(n)$ буде складати $O(f(n)\cdot p(n))$. Якщо асимптотична часова складність функції $p(n)$ складає $O(p(n)) = O(1) = const$, то $O(f(n)\cdot p(n)) = O(f(n)\cdot const) = O(f(n)\cdot 1) = O(f(n))$. Це означає, що при використанні правила множення можна знехтувати постійними додатними множниками, тобто константами.

Розповсюджені оцінки часової складності алгоритмів

Функція асимптотичної складності алгоритму показує наскільки погіршується робота алгоритму при збільшенні розміру вхідних даних. Наприклад, $O(n^2)$ означає, що при збільшенні розміру вхідних даних n у два рази час роботи алгоритму збільшиться в чотири. Існує ряд інших асимптотичних функцій, які охоплюють більшість можливих операцій та їх комбінацій при виконанні будь-якого алгоритму.

Розглянемо асимптотичні функції часової складності алгоритмів, які найчастіше використовуються на практиці:

- $O(1)$ – час виконання алгоритму не залежить від розміру вхідних даних і є постійним (константою). Наприклад, формула розрахунку, оголошення змінної, тощо;

- *O(n)* – це найкраща часова складність алгоритму за виключенням випадку *O(1)*. Подібна складність має місце, коли алгоритм лінійно виконує операції з вхідними даними постійне число разів. У даному випадку для отримання відповіді треба хоча б один раз звернутися до кожного елементу послідовності;
- $O(n^2)$ – обмежений квадратичною функцією час виконання. Алгоритм з такою складністю найчастіше зустрічається у вигляді двох вкладених циклів. При цьому усі пари вхідних даних перебираються не більше, ніж за $O(n^2)$;
- $O(n^3)$ – обмежений кубічною функцією час виконання. Алгоритм з такою складністю найчастіше зустрічається у вигляді трьох вкладених циклів. При цьому усі трійки вхідних даних перебираються не більше, ніж за $O(n^3)$;
- $O(\sqrt{n})$ – алгоритм з даною часовою складністю працює повільніше, ніж *O(log(n))*, але швидше, ніж *O(n)*. Його властивість полягає в тому, що $\sqrt{n} = \frac{n}{\sqrt{n}}$, тобто дані довжиною *n* можна розбити на $\sqrt{n}$ частин, кожна з яких виконуватиметься за час $O(\sqrt{n})$;
- *O(log(n))* – обмеження часу виконання алгоритму логарифмічною функцією означає, що розмір вхідних даних на кожній ітерації зменшуватиметься вдвічі, оскільки $log_2 n$ показує скільки разів необхідно поділити *n* на 2, щоб отримати 1;
- *O(n·log(n))* – така асимптотична часова складність може відповідати або алгоритму сортування, або свідчити про наявність структури даних, в якій кожна операція займає час *O(log(n))*;
- $O(2^n)$ – така асимптотична часова складність відповідає алгоритмам, які перебирають всі можливі підмножини певної множини. Наприклад, є множина {5, 16, 34, 89}, яка має наступні

підмножини: {5}, {16}, {34}, {89}, {5, 16}, {16, 34}, {34, 89}, {5, 89}, {5, 16, 89}, {16, 34, 89}, {5, 16, 34}, тощо;

- *O(n!)* – подібна часова складність відповідає випадку, коли алгоритм перебирає всі перестановки вхідних елементів. Наприклад, є множина {5, 16, 34, 89}, яка має наступні перестановки: {5, 34, 16, 89}, {5, 34, 89, 16}, {16, 5, 34, 89}, {16, 34, 5, 89}, {16, 34, 89, 5}, {89, 16, 34, 5}, тощо.

Якщо алгоритм має асимптотичну часову складність не вище, ніж $O(n^k)$ (k – константа), то він називається поліноміальним. Зазвичай константа k дуже мала. Це означає, що алгоритм з поліноміальною часовою складністю може обробляти вхідні дані великого розміру.

Варто також зауважити, що всі наведені вище асимптотичні часові складності, крім $O(n!)$ та $O(2^n)$, є поліноміальними. Проте в світі існує велика кількість задач, для яких поліноміальний алгоритм ще не знайдено, тому не існує їх ефективного рішення. Прикладом таких завдань можуть бути NP-важкі завдання.

Визначення часу роботи алгоритмів

Для обробки вхідних даних з розміром n будь-якому алгоритму потрібен певний час, який залежить від виду та кількості операцій. Якщо час виконання кожної операції відомий, то визначити загальний час виконання алгоритму можна, наприклад, за наведеними вище правилами.

Розглянемо приклад підрахунку числа одиниць, які знаходяться у множині $X=\{x_1, x_2, x_3, \ldots, x_n\}$ з розміром n. Перший варіант алгоритму зображено на рис. 4.1.

Алгоритм	*Час виконання операції*	*Кількість повторювань*
count ← *0* ***for*** *(i* ← *1; i* ≤ *n; i++)* ***if*** x_i = *1* ***then*** *count* ← *count + 1* ***end if*** ***end for***	t_1 t_2; t_3; t_4 t_5 t_6	*1* *1; n+1; n* *n* *x*

Рисунок 4.1 – Перший варіант алгоритму

Для кожного рядка алгоритму вказано також час виконання t_i, а також кількість повторювань цієї інструкції. При цьому t_i є деякою фіксованою величиною, яка не залежить від розміру вхідних даних n, тобто $t_i = O(1)$, а x – невідоме число повторень, яке залежить від кількості одиниць у початковій послідовності X. Очевидним є те, що значення x може варіюватися від 0 (жодної одиниці в множині) до n (усі елементи множини є одиницями), тобто $0 \le x \le n$.

Якщо додати час виконання окремих операцій наведеного алгоритму з урахуванням їх повторень, можна отримати загальний час виконання алгоритму:

$$t(n)=t_1+t_2+t_3\cdot(n+1)+n\cdot(t_4+t_5)+x\cdot t_6=$$
$$t_1+t_2+t_3+n\cdot(t_3+t_4+t_5)+x\cdot t_6$$

На практиці подібний аналіз алгоритмів важко застосовувати. Значення t_i показує лише те, що час виконання i-тої операції є константою. Конкретне ж значення часу залежить від великої кількості факторів, а саме: апаратного забезпечення (характеристик «заліза»), набору машинних команд, якості компіляції, тощо. Саме тому у більшості випадків визначити точні значення неможливо і використовуються менш точні оцінки, які, наприклад,

базуються на константах пропорційності (вказують як змінюється один об’єкт при зміні одного чи декількох інших) або асимптотичних функціях.

Якщо у даному алгоритмі припустити, що час виконання кожної операції дорівнює одиниці, то загальний час роботи буде складати $t(n)=3n+x+3$. Враховуючи той факт, що $0 \le x \le n$, тобто $x = O(n)$, можна зробити висновок, що асимптотична оцінка часу виконання алгоритму становитиме $t(n)=3n+O(n)+3= O(n)$.

Наведений приклад оцінювання часу виконання алгоритму базувався на використанні констант пропорційності, але досить часто їх визначення викликає проблеми або взагалі не розглядається. У такому випадку доцільно використовувати правила додавання та множення асимптотичних функцій. Розглянемо їх на прикладі того ж самого алгоритму.

Операції присвоювання мають постійний час виконання t_1 та t_2. Якщо звернути увагу на відповідні інструкції, то не важко помітити, що вони не залежать від розміру вхідних даних n. Це означає, що їх асимптотичний час виконання складає $O(1)$. Аналогічний час виконання у кожної інструкції, яка міститься в тілі циклу. У наведеному прикладі цикл **for** виконується n разів, що за правилом множення асимптотичних функцій приводить до виразу $O(1 \cdot n) = O(n)$. У результаті, якщо додати наведені вище асимптотичні часові складності послідовних фрагментів алгоритму, то отримаємо наступну залежність:

$$O(1) + O(n) = O(n).$$

Розмір вхідних даних часто є не єдиним фактором, який впливає на час роботи алгоритмів. Для таких алгоритмів розглядають декілька випадків часової складності, а саме: найгірший, середній, найкращий. Найкращий варіант часу розглядається для визначення найбільш ефективних вхідних

даних алгоритму. Визначення середнього часу дуже часто є складною математичною задачею, тому що він залежить від розподілу ймовірностей вхідних даних, який може відрізнятися від реального (на практиці здебільшого використовується рівномірний розподіл). Найбільший інтерес при аналізі алгоритмів становить саме найгірший випадок, що визначає максимальний час роботи.

Наприклад, у наведеному вище випадку величина *x* має як мінімальне, так і максимальне значення – 0 та n відповідно. Середнє значення цієї величини при рівномірному розподілі складає *n/2*. Таким чином, можна розраховувати час роботи алгоритму:

$$t_{макс}(n)=4n+3=O(n),$$

$$t_{сер}(n)=3,5n+3=O(n),$$

$$t_{мін}(n)=3n+3=O(n).$$

Отже, асимптотична часова складність алгоритму складає *O(n)* у всіх випадках.

Припустимо, що множина $X=\{x_1, x_2, x_3, \ldots, x_n\}$ містить двійковий набір даних, тобто тільки нулі та одиниці. У межах першого алгоритму не потрібно вносити ніяких змін, він буде працювати коректно. Розглянемо другий спосіб підрахунку числа одиниць у двійковому наборі даних (рис. 4.2).

Він базується на тому, що значення розряду містить інформацію про кількість одиниць у ньому. Для того, щоб порівняти наведені алгоритми, однакові операції залишилися з одними і тими ж позначеннями часу виконання. Час роботи другого алгоритму складає

$$t(n)=t_1+t_2+t_3+n\cdot(t_3+t_4+t_7)=3n+3=O(n)$$

і залежить лише від розміру вхідних даних.

Алгоритм	*Час виконання операції*	*Кількість повторювань*
count ← *0* ***for*** *(i ← 1; i ≤ n; i++)* *count ← count + x_i* ***end for***	t_1 t_2; t_3; t_4 t_7	*1* *1; n+1; n* *n*

Рисунок 4.2 – Другий варіант алгоритму

Як видно з наведених рівностей, як перший, так і другий алгоритми мають ідентичну часову складність *O(n)*. Якщо звернути увагу на константи пропорційності, які були визначені при умові рівності часу кожної операції одиниці, то можна зробити висновок, що другий алгоритм є більш ефективним. Проте для близьких за часом виконання алгоритмів подібні спрощення призводять до грубих результатів оцінювання.

Розглянемо, в яких ситуаціях перший алгоритм у середньому є ефективнішим за другий. Нехай $t_1+t_2+t_3=a$, $t_3+t_4+t_5+t_6/2=b_1$, $t_3+t_4+t_7=b_2$. У цьому випадку середній час роботи першого алгоритму складатиме $t_1(n)=a+b_1\cdot n$, а другого – $t_2(n)=a+b_2\cdot n$. Оскільки розглядається варіант ефективності першого алгоритму, то $t1(n)\leq t2(n)$. З цієї нерівності можна зробити висновок, що $b_1\leq b_2$, тобто $t_3+t_4+t_5+t_6/2 \leq t_3+t_4+t_7$. Якщо спростити цю нерівність, то отримаємо $t_5+t_6/2\leq t_7$. Час виконання інструкцій *count* ← *count* + *1* та count ← count + x_i можна пов’язати співвідношенням $t_7=t_6+const$. Підставивши дану рівність в попередній вираз, можна отримати, що $t_5\leq t_6/2+const$ або навіть у найгіршому випадку $t_5\leq const$. Існування даної константи залежить

від архітектури процесора, результатів трансляції коду, тощо. Не виключено випадок, коли *const = 0* і перший алгоритм не буде більш ефективним, ніж другий.

Основою третього алгоритму (рис. 4.3) для підрахунку числа одиниць у двійкових вхідних даних є операція *X^(X-1)*, яка змінює праву одиницю в *X* нулем. При цьому *X* інтерпретується як двійковий код при виконанні кон'юнкції та як двійкове уявлення числа при виконанні операції віднімання. Саме тому цикл буде повторюватися до того часу, коли *X* стане дорівнювати нулю, тобто двійкове уявлення буде складатися лише з нулів. Кількість пройдених ітерацій і буде результатом виконання алгоритму.

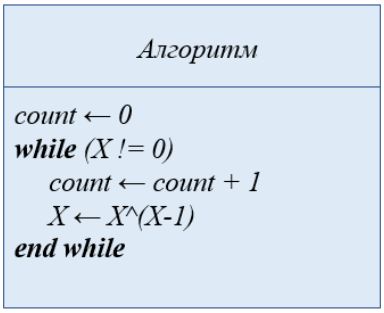


Рисунок 4.3 – Третій варіант алгоритму

Якщо зробити припущення, що операція присвоювання $X \leftarrow X^{(X-1)}$ відповідає двом одиницям часу, а числу одиниць у множині *X* відповідає величина *x*, то час виконання алгоритму буде дорівнювати наступному виразу:

$$t(n)=4x+2.$$

Враховуючи, що $0 \le x \le n$, можна прийти до наступних залежностей:

$$t_{макс}(n)=4n+2=O(n),$$

$$t_{сер}(n)=2n+2=O(n),$$

$$t_{мін}(n)=2=O(1).$$

Очевидно, що для пошуку одиниць у двійковому наборі даних найефективнішим буде третій алгоритм (особливо у випадку вхідної множини малої довжини).

Отже, можна зробити висновок, що для порівняння ефективності декількох алгоритмів необхідно в першу чергу порівняти їх асимптотичну часову складність. У тому разі, якщо часові складності виявляться однаковими, треба розглядати константи пропорційності або інші аспекти алгоритмів.

Варто також пам’ятати, що великий порядок зростання часу виконання алгоритму може мати меншу константу пропорційності при використанні з множиною вхідних даних правильної довжини. Тому не буває хороших чи поганих алгоритмів – все залежить від конкретного завдання та розмірів вхідної множини даних.

Контрольні запитання.

1. Що таке константа пропорційності?

2. Чому при оцінюванні часу виконання алгоритму розглядається саме асимптотична складність?

3. Яка функція є асимптотично точною?

4. Що таке часова складність алгоритму? Чому вона є більш інформативною, ніж об’ємна складність?

5. Яка різниця між нижньою межею швидкості росту функції та верхньою?

6. Які є основні правила проведення операцій при аналізі алгоритмів у межах О-символіки?

7. Які є розповсюджені асимптотичні оцінки складності часу виконання алгоритмів? Яка між ними різниця?

8. Що потрібно враховувати при визначенні часової складності алгоритму?

Лекція №5

Алгоритми пошуку

Завдання пошуку даних

Кожне завдання, яке включає обробку інформації в межах будь-якої сфери діяльності людини, у тому чи іншому вигляді потребує використання пошуку даних у скінченій множині значень.

У загальному випадку дана множина $X=\{x_1, x_2, x_3, \ldots, x_n\}$, яка складається з *n* елементів, може мати складну структуру та є підмножиною множини NS (англ. *Name Space*, простір імен), яка може бути скінченою або нескінченою, тобто $X \in NS$. При цьому з кожним елементом X пов'язаний ключ (англ. *Key*, рис. 5.1), який може іноді називатися ідентифікатором, заголовком, назвою, ім'ям, тощо. Незважаючи на численні варіанти терміну, основне завдання ключа – відрізнити різні елементи послідовності X один від одного.

У подальших прикладах буде розглянуто варіант, який зображено на рис. 5.1, б, тому що з точки зору процесу пошуку важливим фактором є факт виявлення, а не конкретні дані. Тому будемо вважати, що елементами множини X є ключі, тобто $X=\{k_1, k_2, k_3, \ldots, k_n\}$.

Пошук є методом (функцією), який для існуючої скінченої множини даних X та деякої величини *arg*, яка являється аргументом пошуку, повертає вказівник на ключ з множини, який дорівнює аргументу, у разі його існування. З даного визначення можна зробити висновок, що пошук може бути вдалим (коли знайдено ключ, який дорівнює аргументу) або невдалим (коли відповідний ключ не знайдено).

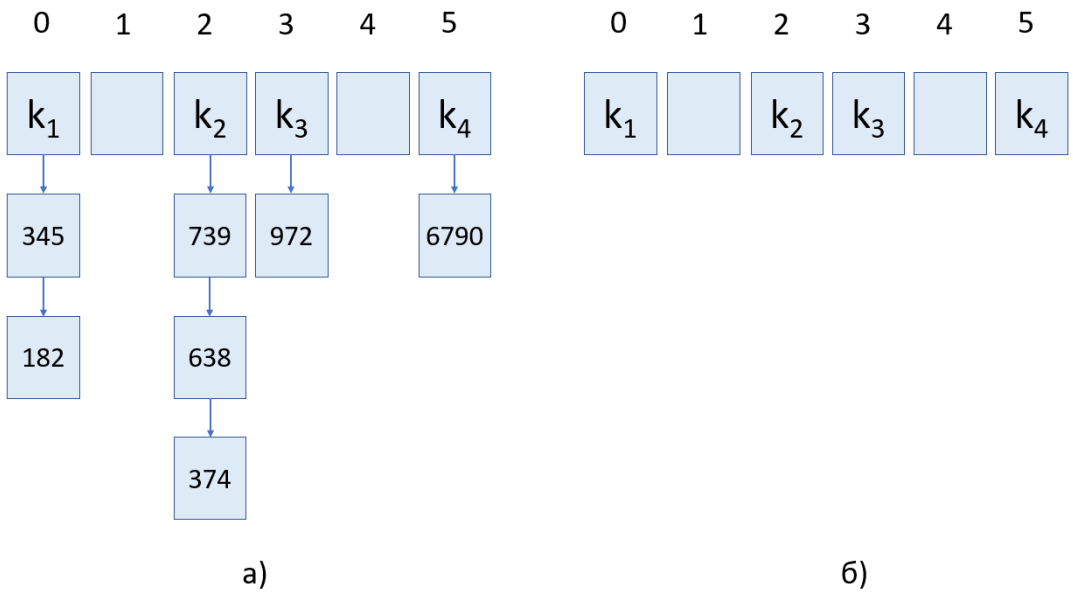


Рисунок 5.1 – Множина X: а) у вигляді зв’язаного списку, який містить ключі та дані; б) у вигляді масиву ключів

Розглянемо декілька умов, які мають виконуватися для виконання пошуку:

1. Якщо множина X ∈ NS та множина Y ∈ NS, то x_i, y_j ∈ NS (x_i ∈ X, y_j ∈ Y).

2. Будь-які два елементи, які належать множині NS (x_i, y_j ∈ NS) можна порівняти між собою, тобто має виконуватися одне з наступних відношень порядку: $x_i < y_j$, $x_i > y_j$, $x_i = y_j$. Простір імен NS характеризується лінійним або природним порядком, коли всі елементи розміщено за зростанням.

3. Для кожного з зазначених у пункті 2 відношень має виконуватися властивість транзитивності. Наприклад, якщо $x_i < y_j$, а $y_j < z_m$ (z_m ∈ Z ∈ NS), то $x_i < z_m$ для будь-яких елементів x_i, y_j, z_m ∈ NS

4. Час порівняння елементів x_i, y_j ∈ NS не залежить від розмірів множин X, Y та NS.

Як відомо, множину можна представити у вигляді статичної (максимальні розміри ніколи не змінюються) або динамічної (максимальні розміри

можуть змінюватися під час виконання алгоритму) структури даних. Для статичних структур мінімізація часу пошуку досягається завдяки правильній організації даних. У динамічних структурах загальна ефективність роботи може залежати від сукупності різноманітних операцій, наприклад, пошуку, сортування, об'єднання, додавання, видалення, тощо. Як правило, не можна оптимізувати (мінімізувати) час виконання одночасно всіх можливих операцій. Саме тому в динамічних структурах обираються операції, які є пріоритетними (наприклад, виконуються найчастіше) і для них оптимізується час виконання. Усе це залежить від конкретних задач.

Найчастіше зустрічаються наступні операції, які потребують використання пошуку:

1. Пошук елементу в множині значень. Якщо *arg* ∈ X, то повертається або значення true, або вказівник на необхідні дані, або номер комірки, в якій ці дані розміщено.

2. Додавання елементу. Якщо *arg* ∉ X, то необхідно знайти вільне місце й записати ключ з необхідними даними.

3. Видалення елементу. Якщо *arg* ∈ X, то видалити ключ з відповідними даними.

Розглянемо основні види пошукових алгоритмів, які використовуються на практиці.

Лінійний пошук

Лінійний або послідовний пошук полягає в послідовному порівнянні аргументу *arg* з кожним елементом множини $X=\{k_1, k_2, k_3, \ldots, k_n\}$ (рис. 5.2). З даного визначення можна зробити висновок, що найгіршим випадком буде той, при якому необхідно здійснити перегляд всієї множини, тобто

порівняти *n* елементів. Саме через таку ймовірність лінійний пошук доцільно використовувати для множин з невеликою кількістю елементів.

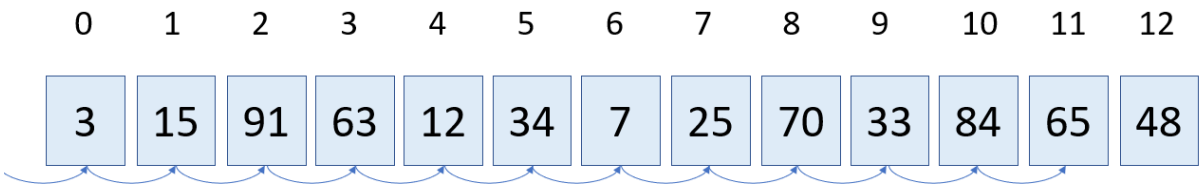


Рисунок 5.2 – Приклад лінійного пошуку числа 65

Для обробки великих структур даних лінійний пошук, як правило, не використовується, тому що навіть при осередненні він використовує число операцій, які пропорційні розміру *n* множини. Розглянемо один з прикладів реалізації лінійного пошуку, який зображено на рис. 5.3.

```
function linearSearch(array[], arg){
    for (i ← 0; i<n; i++)
        if (array[i] == arg) then
            return true
        end if
    end for
    // якщо виконання коду дійшло до наступної інструкції, то
    // шуканий елемент відсутній у множині значень
    return false
}
```

Рисунок 5.3 – Перший варіант алгоритму лінійного пошуку

Даний варіант алгоритму використовується для випадку невпорядкованої множини значень та повертає інформацію про наявність (*true*) або відсутність (*false*) необхідного елементу в ній.

Не важко зрозуміти, що при такій реалізації алгоритму буде існувати два найгірші з точки зору часу виконання варіанти: 1) шуканий аргумент

знаходиться на останній позиції в множині; 2) шуканий елемент відсутній в множині. Вважатимемо, що внесені в послідовність ключі є унікальними і не повторюються, як при використанні хеш-таблиць. Це дає змогу зробити висновок, що усі попередні значення (у разі збігу або його відсутності) були невдалими, тобто алгоритм зробив *n* порівнянь (де *n* – довжина множини).

Оскільки множина значень не є впорядкованою, пропускати перевірку її елементів не можна ні в якому разі, тому що шуканий аргумент міг дорівнювати одному з пропущених значень. Це пояснення виступає у ролі додаткового аргументу щодо необхідності виконання *n* порівнянь.

Отже, асимптотична часова складність даного алгоритму складає *O(n)*.

Якщо елементи в початковій множині даних попередньо було впорядковано за зростанням, то наведений вище алгоритм можна удосконалити (рис. 5.4).

```
function linearSearch(sortedArray[], arg){
    for (i ← 0; i<n; i++)
        if (sortedArray[i] > arg) then
            return false
        end if
        if (sortedArray[i] == arg) then
            return true
        end if
    end for
    return false
}
```

Рисунок 5.4 – Другий варіант алгоритму лінійного пошуку

Даний варіант алгоритму дозволяє завершити виконання коду, якщо поточне значення, яке порівнюється, є більшим, ніж значення аргументу. Тому, коли алгоритм зустрічає більше число у впорядкованій за зростанням

значень послідовності, стає зрозумілим той факт, що шуканий аргумент відсутній. Кількість операцій, які буде виконано у даному випадку, є значно меншою (окрім найгірших варіантів, які описані вище), ніж при використанні першого алгоритму. Проте в межах О-символіки дане прискорення не впливатиме на загальну часову складність алгоритму.

Доведемо останнє твердження. Для того, щоб здійснити пошук, алгоритм має порівняти між собою два елементи: аргумент та елемент множини. Для першого елементу множини кількість порівнянь складатиме одне, для другого – два, для третього – три, для n-го – n. Враховуючи, що наведені алгоритми та використані структури даних мають початковий індекс i=0, можна зробити висновок, що кількість порівнянь j дорівнює i+1 для кожного елементу. Цю особливість варто враховувати під час проектування алгоритму.

Розрахуємо загальну кількість порівнянь, яку треба виконати в найгіршому випадку:

$$f(n) = 1 + 2 + \cdots + n = \sum_{j=1}^{n} j\,. \tag{5.1}$$

Якщо замість послідовного додавання кожного числа проводити дану операцію парами з першого та останнього елементу доступного діапазону (використані при попередній операції елементи виключаються з подальшого розгляду, тобто 1+ n, 2+ n-1, 3+ n-2, тощо), то можна привести формулу (5.1) до наступного вигляду:

$$f(n) = \sum_{j=1}^{n} j = \frac{n(n+1)}{2}, \tag{5.2}$$

де $(n+1)$ – сума кожної пари чисел; $\frac{n}{2}$ – кількість таких пар.

Якщо припустити, що аргумент може займати будь-яке з *n* положень у структурі даних і ймовірність співпадіння рівноймовірна для кожного з них (1/*n*), то час виконання алгоритму буде дорівнювати

$$t(n) = \frac{1}{n} f(n) = \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} = \frac{1}{2}(n+1) = O(n). \quad (5.3)$$

Отже, використання сортування множини значень перед пошуком не призводить до зміни верхньої межі зростання функції і складає *O(n)*. Для інших варіантів алгоритмів пошуку подібні твердження будуть наводитися без доведення через їх подібність.

Незважаючи на цей недолік, лінійний пошук є мабуть єдиним методом, який застосовується до структур даних, в яких не можна легко перестрибнути через декілька елементів, наприклад, зв'язних списків. Алгоритм лінійного пошуку з використанням зв'язного списку наведено на рис. 5.5.

```
function linearSearch(array[], arg){
    for (i ← 0; i<n; i ← i.next)
        if (array[i] == arg) then
            return true
        end if
    end for
    return false
}
```

Рисунок 5.5 – Алгоритм лінійного пошуку в зв'язному списку

У разі необхідності повернення індексу знайденого елементу наведені алгоритми не важко модифікувати. Також на практиці прийнято

резервувати значення 0 (нуль) для випадку, коли шуканий елемент не знайдено в заданій множині значень, що впливає на умовний вираз циклу.

Бінарний пошук

Як було сказано вище, лінійний пошук повільно працює через необхідність послідовної перевірки кожного елементу множини. Існує декілька алгоритмів, які дозволяють зменшити загальний час операції пошуку. Одним з них є бінарний або двійковий пошук, суть якого полягає у зменшенні об'єму задачі в два рази на кожній ітерації.

Бінарний пошук відслідковує індекси мінімального (*min*) та максимального (*max*) елементів заданої послідовності, після чого визначає індекс центрального елементу (*middle*), який розділяє множину значень на дві частини: праву та ліву. Шуканий аргумент порівнюється з центральним елементом і на основі порівняння робиться один з наступних висновків:

1) Якщо *arg* < *middle*, то аргумент має знаходитися в лівій частині, а права відкидається. У подальшому в новій послідовності визначаються ті ж самі ключові елементи (*min*, *max*, *middle*) і проходить наступна ітерація.

2) Якщо *arg* > *middle*, то аргумент має знаходитися в правій частині, а ліва відкидається. У подальшому в новій послідовності визначаються ті ж самі ключові елементи (*min*, *max*, *middle*) і проходить наступна ітерація.

3) Якщо *arg* = *middle*, то алгоритм завершує роботу.

Для перших двох випадків алгоритм закінчить свою роботу або у разі *arg* = *middle* під час однієї з ітерацій, або у разі рівності усіх ключових елементів, тобто *arg* = *middle* = *min* = *max*. Очевидно, що шуканий аргумент буде відсутнім у послідовності, якщо буде виконуватися одна з наступних умов: *max*<*min, max*<*middle, min*>*max*, *min*>*middle*.

Приклад бінарного пошуку зображено на рис. 5.6, а відповідний алгоритм на рис.5.7.

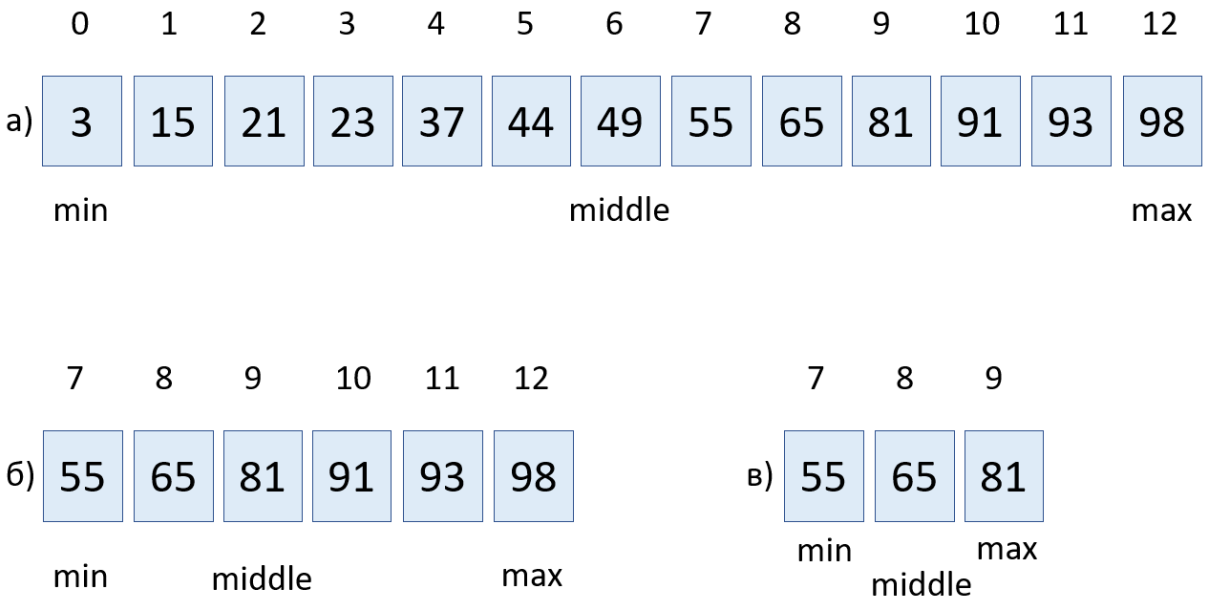


Рисунок 5.6 – Приклад бінарного пошуку числа 65: а) перша ітерація; б) друга ітерація; 3) остання ітерація

```
function binarySearch(array[], arg){
    min ← 0
    max ← n-1
    while (min <= max) do
        middle ← (min + max)/2
        if (arg>array[middle]) then
            min ← middle+1
        else if (arg>array[middle]) then
            max ← middle-1
        else
            return true
        end if
    end while
    return false
}
```

Рисунок 5.7 – Алгоритм бінарного пошуку

З наведеного алгоритму видно, що при відкидуванні однієї з частин послідовності змінюється також відповідна межа: *min* ← *middle* + 1 при видаленні лівої частини, *max* ← *middle* - 1при видаленні правої частини. В обох розглянутих випадках друга використовувана межа зберігається. Також варто зазначити, що додавання та віднімання одиниці при зміні меж відповідає за виключення з подальшого розгляду центрального елементу, з яким проходило порівняння в поточній ітерації, – він не співпадає з аргументом, тож далі не потрібен.

Враховуючи, що бінарний пошук кожної ітерації ділить діапазон значень на дві половини, можна привести загальну довжину множини до вигляду $n = 2^m$-1. Якщо розв'язати дане рівняння відносно значення m, то можна отримати, що $m = log_2(n+1)$. З цього вже можна зробити висновок, що час виконання алгоритму в найгіршому випадку буде складати $O(log(n+1))$.

Якщо шуканий елемент буде відсутнім у послідовності значень, в якій проходить пошук, то всього при обчисленні буде $2n + 1$ ймовірностей, а часову складність алгоритму можна розрахувати за наступною формулою:

$$t(n) = \frac{1}{2n+1}\left(\sum_{j=1}^{m} j2^{j-1} + m(n+1)\right). \qquad (5.4)$$

Якщо провести необхідні математичні операції, то формула (5.4) прийме наступний вигляд:

$$t(n) = \frac{m \cdot 2^{m+1} - 2^m + 1}{2^{m+1}} \approx m - \frac{1}{2} = |m = \log_2(n+1)| =$$

$$= \log_2(n+1) - \frac{1}{2} = O(\log(n+1)). \qquad (5.5)$$

Отже, асимптотична часова складність бінарного пошуку при наявності чи відсутності елементу в множині складає *O(log(n+1))*.

Інтерполяційний пошук

Для ефективної роботи інтерполяційного пошуку у загальному випадку мають виконуватися дві умови: вхідна множина значень $X=\{k_1, k_2, k_3, \ldots, k_n\}$ має бути обов'язково упорядкована по зростанню елементів, а також мати рівномірний розподіл значень. Якщо задовольнити наведені вимоги, можна значно прискорити виконання пошуку.

Це досягається завдяки тому, що алгоритм інтерполяційного пошуку намагається передбачити положення аргументу в межах всієї послідовності значень. При цьому основна частина алгоритму ідентична бінарному пошуку (рис. 5.8). Ці алгоритми відрізняються лише методом визначення центрального значення, яке порівнюється з аргументом методу (функції).

```
function interpolationSearch(array[], arg){
    min ← 0
    max ← n-1
    while (min <= max) do
        middle ← min + (max-min)*(arg-array[min])/(array[max]-array[min])
        if (arg>array[middle]) then
            min ← middle+1
        else if (arg>array[middle]) then
            max ← middle-1
        else
            return true
        end if
    end while
    return false
}
```

Рисунок 5.8 – Алгоритм інтерполяційного пошуку

З рис. 5.8 видно, що поняття «центральний елемент» в інтерполяційному пошуку носить лише умовний характер, тому що індекс, з яким буде проходити порівняння у більшості випадків буде знаходитися ближче до одного з країв. Розрахунок цього елементу включає в себе масштабування відстані від *min* до *max* (*min + (max - min)*) за очікуваною часткою відстані на проміжку між *array[min]* та *array[max]* (0 < *(arg - array[min])/(array[max] - array[min])* < 1), де має знаходитися шуканий аргумент.

Наведений варіант алгоритму інтерполяційного пошуку також має ряд проблем, наприклад, інколи розрахункове значення центрального елементу може виходити за межі мінімального або максимального. Існують варіанти рішення, але в межах даного посібника вони розглядатися не будуть.

Приклад інтерполяційного пошуку зображено на рис. 5.6.

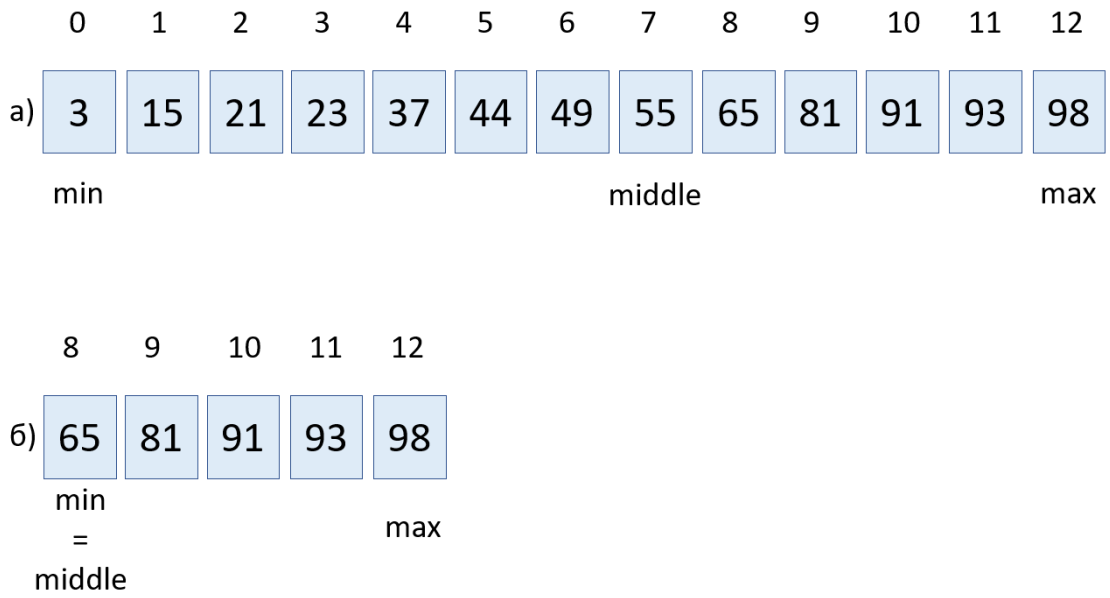


Рисунок 5.9 – Приклад інтерполяційного пошуку числа 65: а) перша ітерація; б) остання ітерація

Асимптотично складність бінарного пошуку вища, ніж інтерполяційного, який виконується за *O(log(log(n)))*. Проте, якщо множина має

нерівномірний розподіл значень, то асимптотична часова складність інтерполяційного пошуку може зростати до значення *O(n)*.

Контрольні запитання.

1. У чому полягає суть лінійного пошуку?

2. Яка різниця між лінійним пошуком у впорядкованому та невпорядкованому масиві?

3. Чому значення ключа має бути унікальним?

4. Як вплине на алгоритми лінійного пошуку, які наведено вище, заміна початкового індексу з 0 на 1? Що треба буде змінити?

5. Чи можна застосовувати бінарний пошук для невпорядкованих даних? Чому?

6. У чому полягає суть бінарного пошуку?

7. Які переваги та недоліки лінійного, бінарного та інтерполяційного пошуку?

8. Чому бінарний та інтерполяційний пошуки не можуть проводитися в зв’язних списках? Пояснити відповідь.

9. Чому бінарний та інтерполяційний пошуки не можуть проводитися в невпорядкованих послідовностях? Пояснити відповідь.

Лекція №6

Алгоритми сортування. Рекурсія

Завдання сортування даних

Сортування є методом (функцією), який для існуючої скінченої множини даних $X=\{x_1, x_2, x_3, \ldots, x_n\}$ повертає її значення, розміщені у природному порядку (за зростанням) або у порядку спадання.

Як і у минулих лекціях, будемо вважати, що вхідна множина може мати складну внутрішню структуру, але з кожним її елементом асоційований унікальний ключ. Його основна задача – запобігання виникнення невизначеності або помилок ідентифікації при виконанні операцій з даними. Отже, у подальшому вважається, що множина X містить саме значення ключів, тобто $X=\{k_1, k_2, k_3, \ldots, k_n\}$. Це означає, що будь-які $k_i, k_j \in X$ при $i \neq j$ мають бути або $k_i < k_j$, або $k_i > k_j$. У цьому випадку процес сортування має знайти перестановку П, яка відображає вхідну множину даних X у вигляді зростаючої (спадаючої) послідовності ключів.

Перестановка – будь-яка послідовність унікальних (без повторів) елементів множини X, яка враховує їх певний порядок. Різні перестановки з n елементів множини X відрізняються лише порядком елементів. Наприклад, для перших чотирьох натуральних чисел можна утворити наступні перестановки: {1, 2, 3, 4}, {3, 1, 2, 4}, {4, 3, 1, 2}, тощо.

Алгоритми сортування поділяються на стійкі та нестійкі. У деяких випадках множина значень містить як первинні, так і вторинні ключі. Вона може бути попередньо впорядкованою за вторинними ключами. При стійкому сортуванні за первинним ключем у випадку наявності двох однакових

імен зберігається порядок попереднього сортування за вторинним ключем. Якщо ж дана умова не виконується, алгоритм вважається нестійким. У даному випадку завдання сортування необхідно вирішувати з використанням комбінованого ключа, який об'єднує первинний та вторинний.

Алгоритми сортування також поділяються на внутрішні та зовнішні. Якщо початкова множина даних має малі розміри, то вона може бути повністю розміщеною в оперативній пам'яті. Сортування такої множини називається внутрішнім. Якщо початкова множина даних має великі розміри та не може повністю розміститися в оперативній пам'яті, то її сортування найчастіше проходить частинами, а сама множина зберігається на зовнішньому пристрої (наприклад, HDD, SSD, CD, тощо). Сортування подібних даних називається зовнішнім.

Дані, які містяться у вхідній множині, визначають час, необхідний на її сортування. Саме тому необхідно визначити критерій , який дозволяє оцінити ступінь відхилення даних від їх природного порядку. Таким критерієм може бути, наприклад, інверсія перестановки.

Нехай існує деяка перестановка $П=\{k_1, k_2, k_3, \ldots, k_n\}$. Інверсією перестановки П називається пара елементів (k_i, k_j), для яких виконуються наступні умови: $i < j$ та $k_i > k_j$. Послідовність цілих чисел $D=\{d_1, d_2, d_3, \ldots, d_n\}$ називається вектором інверсій перестановки П. Елемент вектору інверсій перестановки d_j показує кількість пар (k_i, k_j), які є інверсіями перестановки П. Якщо перефразувати, то d_j – це число елементів множини, які є більшими за елемент k_j та знаходяться зліва від нього. З даного визначення можна зробити висновок, що $0 \le d_j \le j$ для випадку, коли індекси починаються з нуля (при i, j – цілі невід'ємні числа, $i, j = 0, 1, 2, \ldots, n\text{-}1$), або $0 \le d_j < j$ для випадку, коли індекси починаються з одиниці (при i, j – натуральні числа, $i, j = 1, 2, \ldots, n\text{-}1$). Приклад визначення вектору інверсій перестановки П у випадку початкового індексу структури даних $i = 0$ зображено на рис. 6. 1.

Рисунок 6.1 – Приклад визначення вектору інверсій перестановки П: а) перестановка П; б) вектор інверсій перестановки П; в) визначення інверсій

Вектор інверсій однозначно визначає перестановку. Якщо ж додати усі отримані інверсії, то можна отримати їх загальну кількість (для наведеного прикладу 8), яка і характеризує ступінь відхилення даних від природного порядку.

Існує декілька різних принципів, які використовуються для сортування даних. Їх приблизний перелік наведено в табл. 6.1.

Таблиця 6.1 – Алгоритми сортування

Сортування	Час	Об’єм даних
Вибором	$O(n^2)$	Множини малого розміру
Вставкою	$O(n^2)$	Множини малого розміру
Бульбашкове	$O(n^2)$	Частково впорядковані множини малого розміру

Продовження таблиці 6.1

Швидке	*O(n·log(n))*	Множини великого розміру без великої кількості дублікатів
Пірамідальне	*O(n·log(n))*	Множини великого розміру з невідомим законом розподілу
Злиттям	*O(n·log(n))*	Множини великого розміру з невідомим законом розподілу

Алгоритми з часом виконання $O(n^2)$

Алгоритми даної категорії працюють повільно, але їх реалізація є досить простою. Саме тому при сортуванні множин малих розмірів ці алгоритми можуть працювати ефективніше, ніж їх більш швидкі, але і більш складні, аналоги.

Сортування вставкою. Алгоритм такого сортування окрім масивів працює також зі зв'язними списками, стеками та чергами. Основу цього методу складає вибір елементу з початкової множини та його вставка в відповідне положення після зміщення вправо елементів, які більші за нього. Після завершення всіх операцій вставки отримана перестановка відповідатиме впорядкованій початковій множині. Одну з варіацій алгоритму сортування вставкою та приклад його використання зображено на рис. 6.2 та рис 6.3 відповідно.

Як видно з прикладу, алгоритм має циклічно змістити всі елементи, які знаходяться між індексами *i* та *j*. Це означає, що при роботі з великими масивами даних алгоритм має виконувати дуже великий об'єм операцій на кожній ітерації циклу *for*, а загальний час його роботи становитиме $\frac{n^2-n}{2}$, тобто $O(n^2)$.

```
function sortByInsertion(array[]){
    for (i ← 1; i < n; i++) do
        temp ← array[i]
        j ← i-1
        while (j ≥ 0 && array[j] > temp) do
            array[j+1] ← array[j]
            j--
        end while
        array[j+1] ← temp
    end for
}
```

Рисунок 6.2 – Алгоритм сортування вставкою

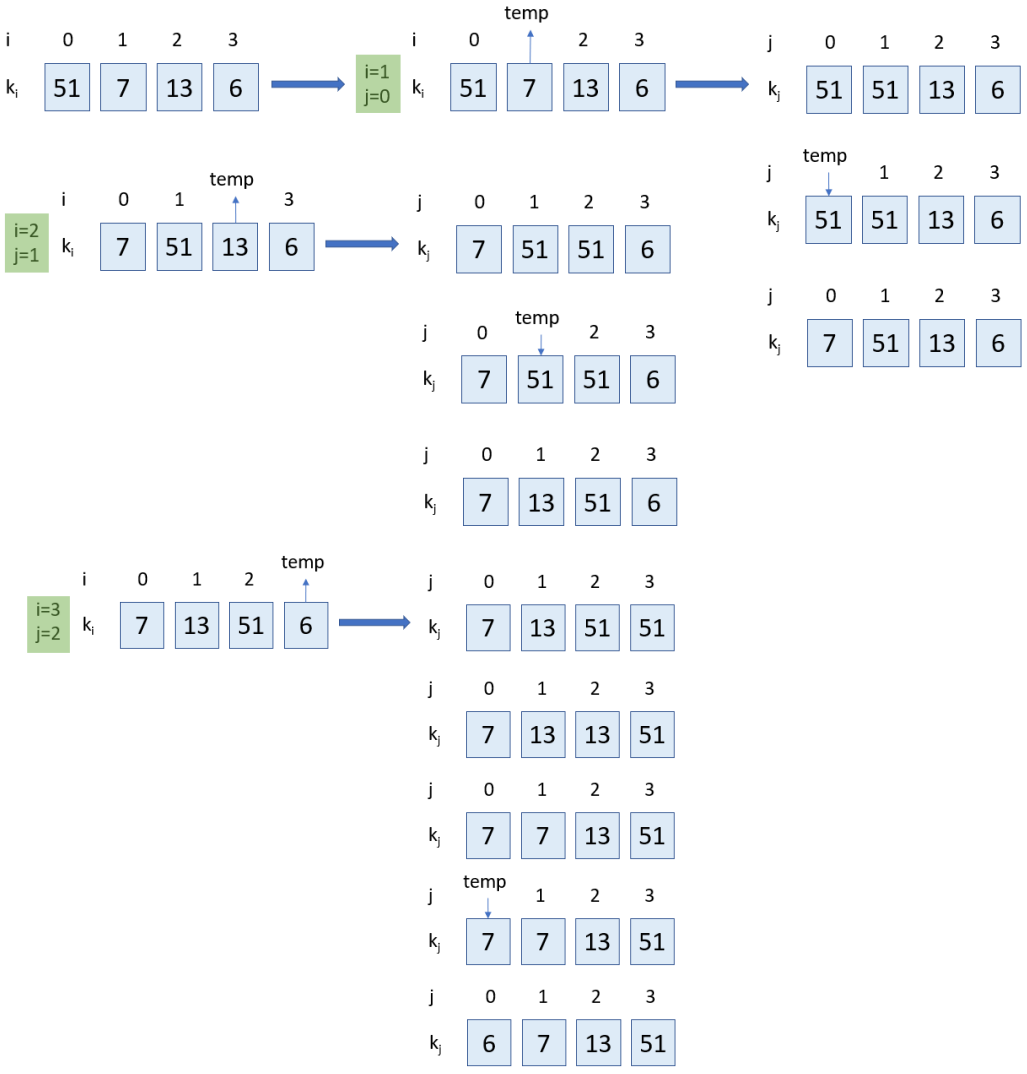


Рисунок 6.3 – Приклад використання алгоритму сортування вставкою

Сортування вибором. Даний алгоритм сортування полягає в пошуку найменшого (найбільшого) елементу множини X, який переміщується на свою остаточну позицію. Інакше кажучи, множина розбивається на дві частини: впорядковану та невпорядковану. На кожній ітерації елементи множини переміщуються з невпорядкованої частини в впорядковану і навпаки, займаючи місце один одного. Так відбувається до того часу, коли буде отримано перестановку, яка відповідатиме впорядкованій початковій множині.

Одну з варіацій алгоритму сортування вибором від найменшого до найбільшого та приклад його використання зображено на рис. 6.4 та рис 6.5 відповідно.

```
function sortBySelection(array[]){
    for (i ← 0; i < n; i++) do
        oldMinIndex ← i
        min ← array[i]
        for (j ← i+1; j < n; j++) do
            if (array[j] < min) then
                oldMinIndex ← j
                min ← array[j]
        end for
        array[oldMinIndex] ← array[i]
        array[i] ← min
    end for
}
```

Рисунок 6.4 – Алгоритм сортування вибором

Так само, як і в минулому методі, сортування вибором включає два вкладених цикли для пошуку та переміщення значень елементів між індексами *i* та *oldMinIndex,* а загальний час його роботи також становитиме $\frac{n^2-n}{2}$, тобто $O(n^2)$.

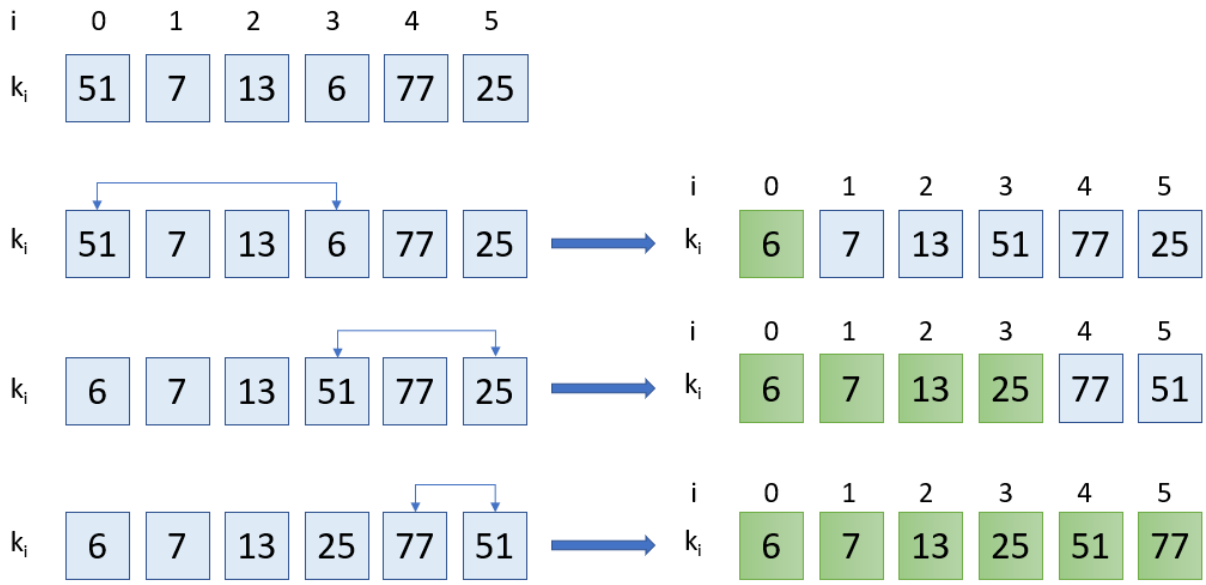


Рисунок 6.5 – Приклад використання алгоритму сортування вибором

Бульбашкове сортування. Даний алгоритм сортування в своїй основі має припущення, що будь-яких два суміжних елементи невпорядкованої множини перебувають у неправильному положенні. Через це алгоритм має декілька разів проходити по множині, замінюючи місцями неправильні пари.

Свою назву даний алгоритм отримав через особливість уявлення вхідної множини, а саме: не горизонтально, як у більшості випадків, а вертикально. При цьому індекси елементів збільшуються згори до низу, а сортування починається з самого нижнього елемента. Таким чином, під час виконання алгоритму елементи наче спливають.

Якщо порівняти алгоритми попередніх методів з алгоритмом бульбашкового сортування (рис. 6.6), можна побачити, що їх особливістю є наявність вкладених циклів. Саме через необхідність багатократного проходження по множині даних для кожного її елементу асимптотична часова складність розглянутих алгоритмів обмежується значенням $O(n^2)$. Незважаючи на це, реалізація алгоритму бульбашкового сортування є більш

простою, тому на практиці використовується частіше інших алгоритмів з часом $O(n^2)$.

```
function sortByBubble(array[]){
    for (i ← 0; i < n; i++) do
        for (j ← n-1; j > i; j--) do
            if (array[j-1] > array[j]) then
                temp ← array[j-1]
                array[j-1] ← array[j]
                array[j] ← temp
        end for
    end for
}
```

Рисунок 6.6 – Алгоритм бульбашкового сортування

Приклад двох ітерацій бульбашкового сортування множини зображено на рис. 6.7.

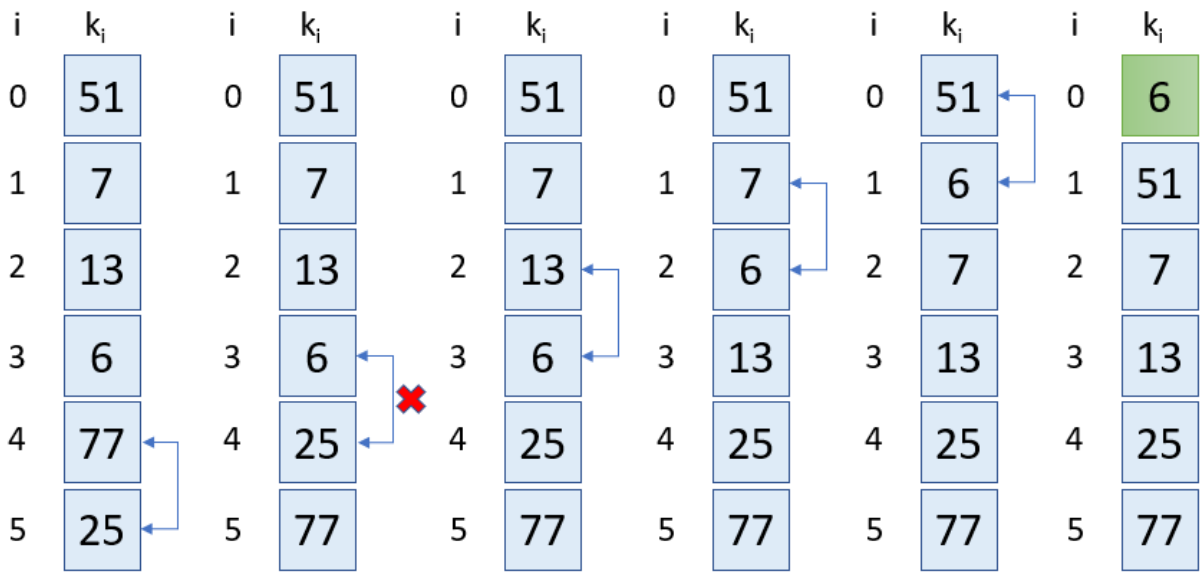


Рисунок 6.7 – Приклад використання алгоритму бульбашкового сортування

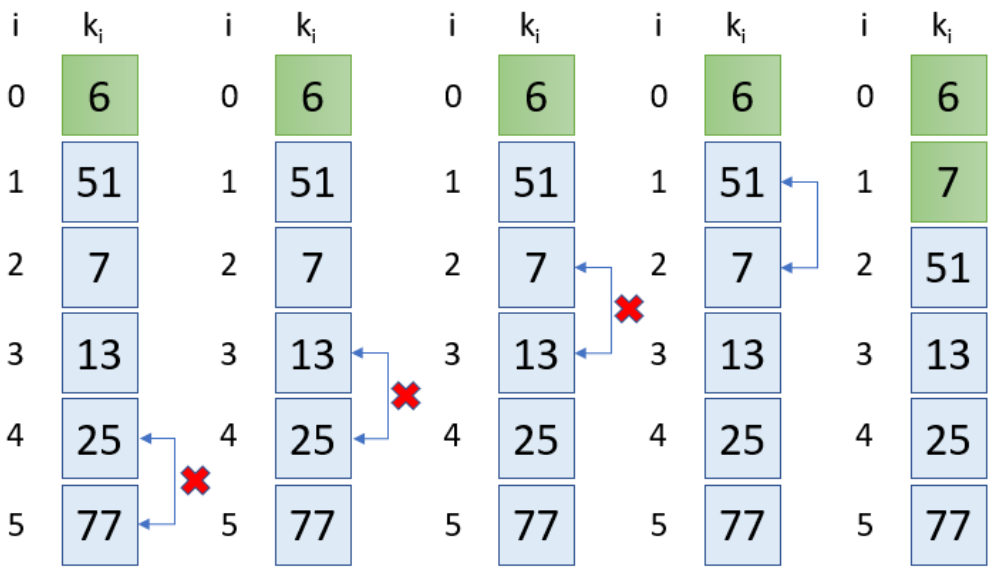


Рисунок 6.7, аркуш 2

З прикладу можна побачити, що ітерація зовнішнього циклу не завершується при невдалому результаті порівняння елементів – алгоритм обов'язково перевіряє усі пари невпорядкованих елементів.

Рекурсія

Явище рекурсії полягає у багаторазовому виклику методом (функцією) самого себе. Якщо виклик відбувається безпосередньо, то така рекурсія називається простою, якщо через інші методи, – складною.

Людині важко мислити рекурсивно, але в природі існують завдання, які для свого рішення потребують багаторазового використання однієї і тієї ж послідовності дій. Ключовою особливістю таких завдань є те, що для рішення поточної ітерації необхідні дані з наступної, наступній – ще з наступної і так далі. Це означає, що знайти кінцеве рішення можна тільки у тому випадку, коли алгоритм знайде рішення в межах останньої ітерації, що дозволить йому повертатися назад допоки він не розрахує кінцеве рішення.

Враховуючи вищесказане, можна зробити висновок, що рекурсія не є найкращим варіантом рішення поставлених завдань через значне зниження швидкості виконання програми. Також недоліком рекурсії є ймовірність виникнення нескінченного циклу, що призведе до «зависання» програми. При цьому не можна сказати, що у рекурсії немає переваг. Деякі алгоритми пошуку та сортування використовують властивість рекурсії для підвищення їх ефективності.

Для прикладу розглянемо класичний випадок використання рекурсії – факторіал числа. Він визначається, наприклад, за допомогою алгоритму, який зображено на рис. 6.8.

```
function factorialOf (x){
    if (x == 0) then
        return 1
    end if
    return x·factorialOf (x-1)
}
```

Рисунок 6.8 – Алгоритм розрахунку факторіалу числа

З наведеного алгоритму видно, що час його виконання лінійний і залежить від числа *x*, тобто асимптотична часова складність дорівнює *O(n)*, де *n* – довжина послідовності значень 0, 1, 2, …, *x*, які використовуються для розрахунку факторіалу. Також можна визначити дві характеристики, якими повинні володіти рекурсивні алгоритми:

1. Кожного разу в ході виконання метод знижує складність завдання, а потім викликає себе ще раз, щоб знайти вирішення спрощеного завдання.

2. Час виконання рекурсивних алгоритмів має бути скінченим. Якщо у якості аргументу наведеного алгоритму ввести будь-яке від'ємне ціле число, то алгоритм увійде в нескінченний цикл. Модифікація алгоритму для

вирішення даної проблеми в посібнику розглядатися не буде і виноситься на самостійне опрацювання.

Алгоритми з часом виконання O(n log(n))

Алгоритми даної категорії виконуються в декілька разів швидше, ніж розглянуті вище алгоритми з асимптотичною часовою складністю $O(n^2)$, особливо для вхідних даних великого розміру. Наприклад, якщо довжина множини значень n=1000, то при складності $O(n^2)$ час виконання складатиме 10^6, а при $O(n\ log(n))$ – 10^4, що менше в 100 разів.

Розглянемо особливості реалізації декількох алгоритмів.

Пірамідальне сортування. Основна ідея пірамідального сортування полягає у побудові піраміди, яка складається з елементів вхідної множини, з подальшим сортуванням. Пірамідою зазвичай називається бінарне (двійкове) дерево. Саме тому даний алгоритм сортування буде розглянуто у лекції, яка присвячена зазначеній структурі даних.

Швидке сортування. У розглянутих вище алгоритмах сортування вставкою та бульбашкового сортування основна причина їх низької ефективності полягає в тому, що елементи множини переміщуються лише на одну позицію. Такі алгоритми завжди будуть обмежуватися часом $O(n^2)$ як в найгіршому, так і в середньому випадках.

На відміну від них, алгоритм швидкого сортування полягає у виборі одного з ключів множині, який буде її розділяти на дві частини: з меншими та більшими значеннями. Цей ключ називається базовим або опорним. При цьому сортування отриманих підмножин відбувається рекурсивно. Якщо виділити описані кроки в окремий список, то отримаємо наступне:

1) Визначити базовий елемент множини.

2) Провести сортування інших елементів множини у дві підмножини: а) елементи менше базового; б) елементи більше базового.

3) Рекурсивно повторити пункти 1) та 2) для кожної підмножини.

Алгоритм сортування зображено на рис. 6.9. Як видно даного рисунку, алгоритм швидкого сортування має значно складнішу структуру, ніж алгоритми з часом $O(n^2)$. Це призводить до збільшення необхідних апаратних потужностей при його виконанні, але максимальний час сортування в середньому все одно буде значно меншим.

```
function sortQuick(array[], firstIndex, lastIndex){
    if (lastIndex <= firstIndex) then
        return
    end if
    base ← array[lastIndex]
    i ← firstIndex
    j ← lastIndex
    while (i ≤ j) do
        while (array[i] < base) do
            i++
        end while
        while (array[j] > base) do
            j--
        end while
        if (i ≤ j) then
            temp ← array[i]
            array[i] ← array[j]
            array[j] ← temp
            i++
            j--
        end if
    end while
    if (firstIndex < j) then
        sortQuick(array[], firstIndex, j)
    end if
    if (lastIndex > i) then
        sortQuick(array[], i, lastIndex)
    end if
}
```

Рисунок 6.9 – Алгоритм швидкого сортування

На рис. 6.10 зображено приклад роботи алгоритму швидкого сортування в теорії. З даного прикладу стає зрозумілим, що для повного сортування вхідної послідовності буде розглянуто п’ять підмножин у межах чотирьох рекурсивних викликів. Але, якщо використати наведений вище код, то буде отримано результат, який зображено на рис. 6.11.

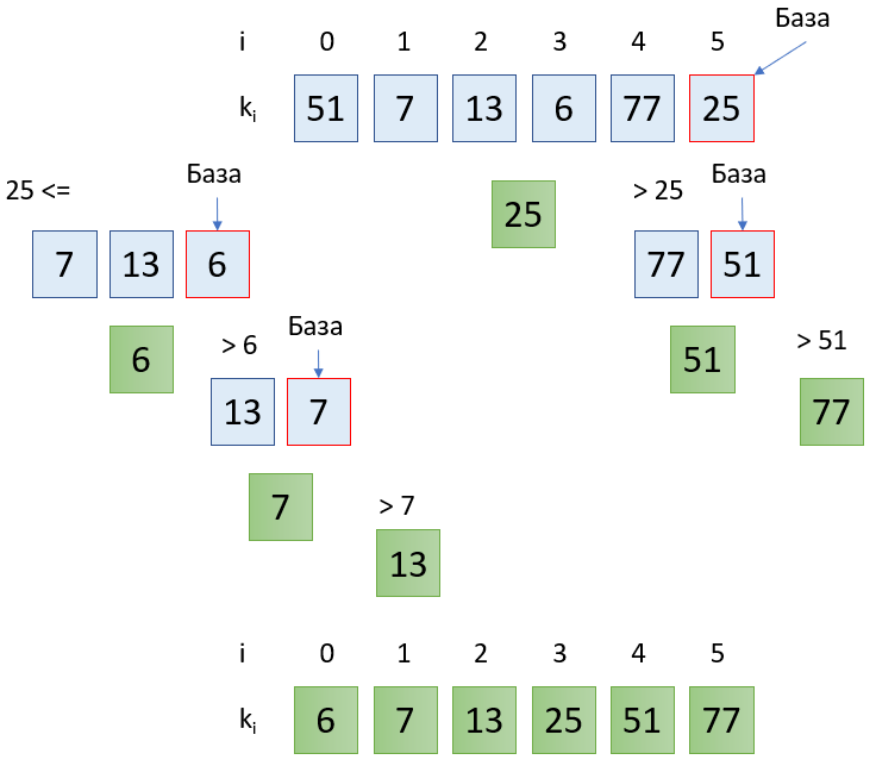


Рисунок 6.10 – Приклад швидкого сортування множини в теорії

```
New algorithm call
[25, 7, 13, 6, 77, 51]
New algorithm call
[6, 7, 13, 25, 77, 51]
New algorithm call
[6, 7, 13, 25, 77, 51]
New algorithm call
[6, 7, 13, 25, 77, 51]
New algorithm call
[6, 7, 13, 25, 51, 77]
```

Рисунок 6.11 – Приклад швидкого сортування множини на практиці

З результатів роботи алгоритму видно, що відбувається п’ять рекурсивних викликів, а результати трьох центральних взагалі співпадають. Саме тому в якості самостійної роботи для закріплення матеріалу та формування навичок розроблення алгоритмів пропонується модифікувати метод, який зображено на рис. 6.10 таким чином, щоб результати його роботи співпадали з теоретичними та не перевищували час $O(n \log(n))$.

Варто також зазначити, що у найгіршому випадку алгоритм швидкого сортування має асимптотичну часову складність $O(n^2)$. Це може відбуватися, наприклад, у випадку, коли множину даних вже було впорядковано до використання швидкого сортування. Тоді у випадку вибору першого або останнього елементу цієї множини як базового призведе до найгіршого випадку.

Для вирішення даної проблеми можна, наприклад, рандомізувати порядок значень у множині, але для дуже великих n такий варіант застосовуватися не може з тих же причин. Іншим варіантом може бути використання елементу, який знаходиться між будь-якими двома з наступних трьох: першим, центральним, останнім. І останнім з можливих рішень є вибір випадкового елементу як базового. У цьому випадку шанс того, що кожен вибір буде найгіршим – мінімальний.

Якщо алгоритм швидкого сортування використовує просту стратегію вибору базового елементу (наприклад, перший чи останній), він може стати причиною вразливості до так званих DoS-атак. Знаючи про дану вразливість, зловмисник може спеціально формувати потік вхідних даних таким чином, щоб час їх обробки був найгіршим.

Сортування злиттям. Даний метод полягає у злитті двох чи більше впорядкованих множин в одну загальну. Це досягається завдяки рекурсивному розділенню початкової множини на підмножини до того часу, коли кожна окрема підмножина буде містити всього один елемент. Очевидно, що

один елемент є впорядкованим за замовчуванням. Після цього сусідні підмножини, які складаються спочатку зі всього одного елементу, рекурсивно зливаються. В якості першого елементу новоутвореної множини на кожному етапі злиття обирається найменший елемент з відповідних підмножин.

Приклад сортування злиттям наведено на рис. 6.12. Темно-синім кольором виділено межі підмножин, на які розбивається основна на кожному етапі.

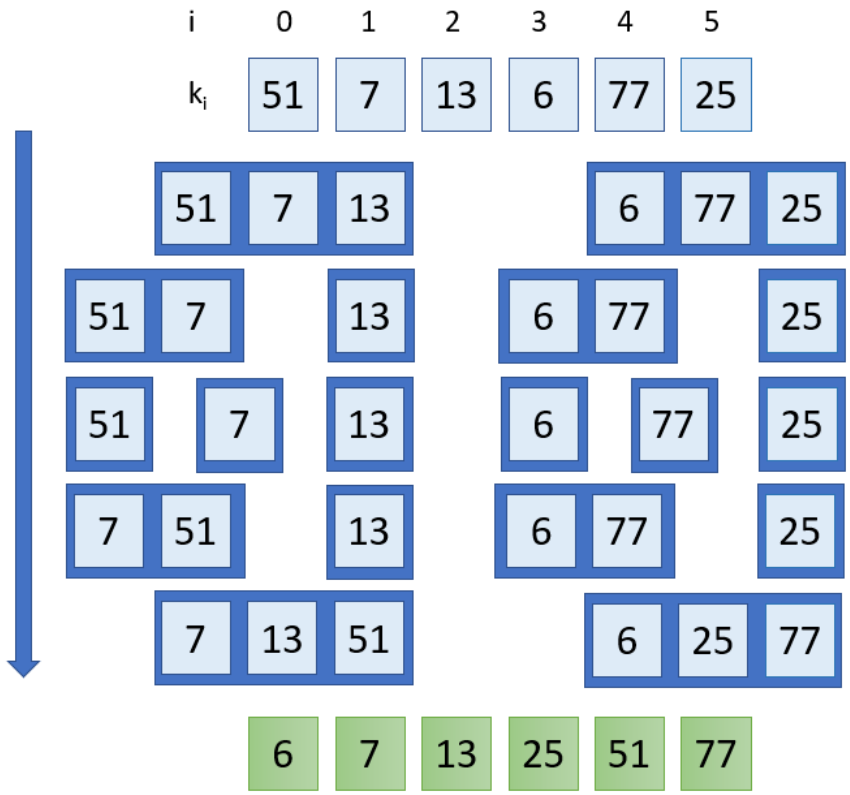


Рисунок 6.12 – Приклад сортування множини злиттям

Так само, як і пірамідальне сортування, сортування злиттям не залежить від порядку елементів у вхідній множині, тому його асимптотична часова складність у найгіршому, середньому та найкращому випадках залишатиметься сталою та буде дорівнювати $O(n \log(n))$. Ця властивість досягається за рахунок того, що кожну підмножину при об'єднанні вже

впорядковано за зростанням, тому не потрібно порівнювати усі елементи між собою для визначення найменшого, – достатньо порівняти перші.

Алгоритм сортування злиттям зображено на рис. 6.13.

```
function sortByMerge(array[], tempArray[], firstIndex, lastIndex){
    if (lastIndex <= firstIndex) then
        return
    end if
    middleIndex ← (firstIndex+lastIndex)/2
    sortByMerge(array, tempArray, firstIndex, middleIndex)
    sortByMerge(array, tempArray, middleIndex, lastIndex)
    leftIndex ← firstIndex
    rightIndex ← middleIndex+1
    tempIndex ← leftIndex
    while ((leftIndex ≤ middleIndex) && (rightIndex ≤ lastIndex)) do
        if (array[leftIndex] ≤ array[rightIndex]) then
            tempArray[tempIndex] ← array[leftIndex]
            leftIndex++
        else
            tempArray[tempIndex] ← array[rightIndex]
            rightIndex++
        end if
        tempIndex++
    end while
    for (i ← leftIndex; i ≤ middleIndex; i++) do
        tempArray[tempIndex] ← array[i]
        tempIndex ++
    end for
    for (i ← rightIndex; i ≤ lastIndex; i++) do
        tempArray[tempIndex] ← array[i]
        tempIndex ++
    end for
    for (i ← firstIndex; i ≤ lastIndex; i++) do
        array[i] ← tempArray[tempIndex]
    end for
}
```

Рисунок 6.13 – Алгоритм сортування злиттям

Існують також алгоритми, які виконуються швидше, ніж *O(n log(n))*, але в межах даного навчального посібника вони розглядатися не будуть.

Контрольні запитання.

1. Що таке сортування? У чому його суть?

2. Що таке перестановка та інверсія перестановки? Для чого вони використовуються?

3. Які є алгоритми з часом $O(n^2)$? Яке обмеження не дає їм виконуватися швидше?

4. Чим відрізняються між собою алгоритми сортування з часом $O(n^2)$?

5. У чому полягає суть сортування вставкою? Пояснити на прикладі.

6. У чому полягає суть сортування вибором? Пояснити на прикладі.

7. У чому полягає суть бульбашкового сортування? Пояснити на прикладі.

8. Що таке рекурсія? Для чого вона використовується?

9. Як потрібно змінити алгоритм, який наведено на рис. 6.8, щоб він коректно реагував на від'ємний аргумент? Пояснити відповідь.

10. Як потрібно змінити алгоритм, який наведено на рис. 6.10, щоб результат його роботи співпадав з теорією, а час роботи не перевищував $O(n\ log(n))$? Пояснити відповідь.

11. У яких випадках алгоритм швидкого сортування працює з часом $O(n^2)$?

12. Як працює алгоритм сортування злиттям? Пояснити на прикладі.

13. Як потрібно змінити алгоритм, який наведено на рис. 6.13, щоб не потрібно було використовувати додатковий порожній масив?

Лекція №7

Графи

Основні терміни та поняття

Велику кількість завдань зі створення алгоритмів можна вирішити простіше (у порівнянні з іншими структурами даних) завдяки використанню графів.

Граф – структура даних, яка складається з *n* вершин, об'єднаних між собою *m* ребрами. На рис. 7.1 зображено граф, який містить 6 вершин у вигляді кіл та 7 ребер у вигляді ліній між деякими вершинами.

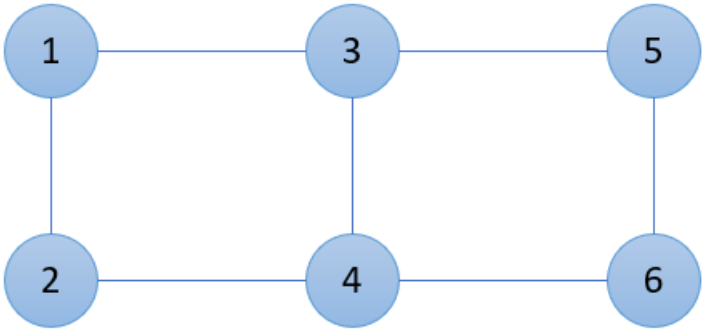


Рисунок 7.1 – Приклад графу

Шлях – відстань від однієї вершини до іншої. Шлях прокладається по ребрам, які з'єднують вершини.

Довжина шляху – кількість ребер, які входять у відповідний шлях.

Цикл – це шлях, в якому остання вершина співпадає з першою.

Ациклічний граф – це граф, який не містить жодного циклу.

Приклади шляху з довжиною 4 (1→4→3→5→6) та циклу 4→6→5→3→4 зображено на рис. 8.2.

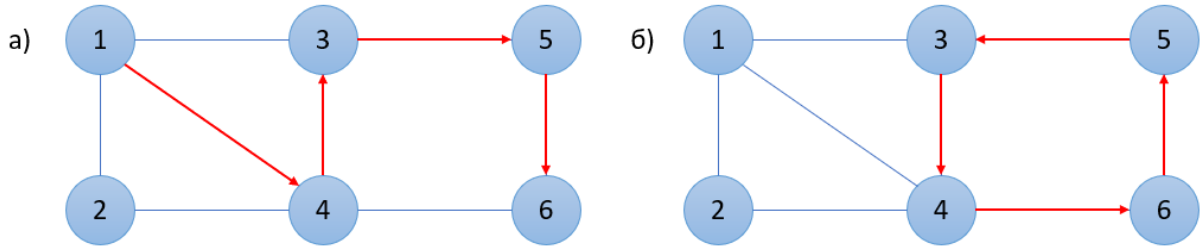


Рисунок 7.2 – Приклад: а) шляху з довжиною 4; б) циклу

Зв'язний граф – це граф, у якого між будь-якими двома вершинами існує хоча б один шлях.

Компоненти зв'язності – зв'язані між собою частини графа.

Граф, який зображено на рис 7.2, є зв'язним, на рис. 7.3 – ні, тому що з вершини «2» не можна перейти до будь-якої іншої через відсутність ребер. Перший граф складається з однієї компоненти зв'язності, а другий – з трьох: {1, 3, 4}, {2}, {5, 6}.

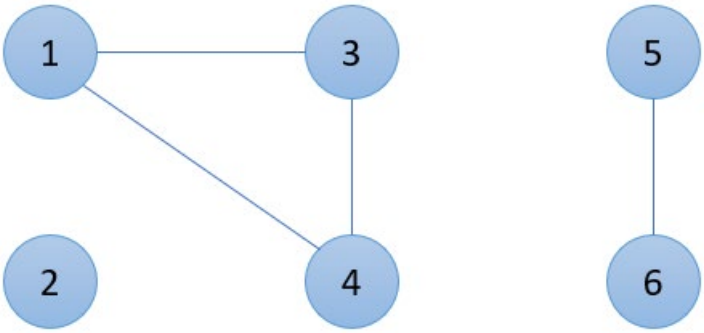


Рисунок 7.3 – Приклад незв'язного графу

Орієнтований граф – це граф, в якому по кожному ребру можна просуватися лише у чітко визначеному напрямку.

На рис. 7.4, а зображено приклад орієнтованого графу, в якому не існує шляху, наприклад, з вершини «6» у вершину «1».

Зважений граф – це граф, у якому кожне ребро має свою вагу.

Вага – найчастіше є довжиною ребра. У такому випадку довжина шляху визначається як сума ваги ребер, з яких він складається.

На рис. 7.4, б зображено зважений граф, довжина найкоротшого шляху від вершини «1» до вершини «6» якого складає 3+3+2+1=9 (1→4→3→5→6).

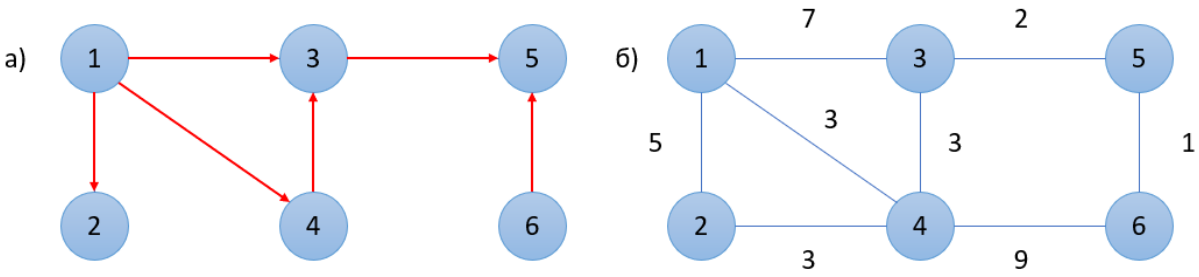


Рисунок 7.4 – Граф: а) орієнтований; б) зважений

Сусідні (суміжні) вершини – дві вершини, які з'єднані одним ребром.

Степінь вершини – число суміжних до поточної вершин.

Сума степенів усіх вершин будь-якого графа буде дорівнювати *2m*, тобто завжди буде парною. Ця властивість випливає з того, що кожне ребро збільшує степені тільки двох вершин. У цьому легко переконатися, якщо розглянути зображений на рис. 7.5 приклад (7 ребер, 14 степінь) або провести підрахунки для графу з рис. 7.4, б (8 ребер, 16 степінь).

Регулярний граф – це граф, степені всіх вершин якого однакові.

Повний граф – це граф, в якому степінь кожної вершини дорівнює *n - 1*, тобто він містить усі можливі ребра.

Напівстепінь заходження вершини – число ребер орієнтованого графа, які закінчуються у поточній вершині.

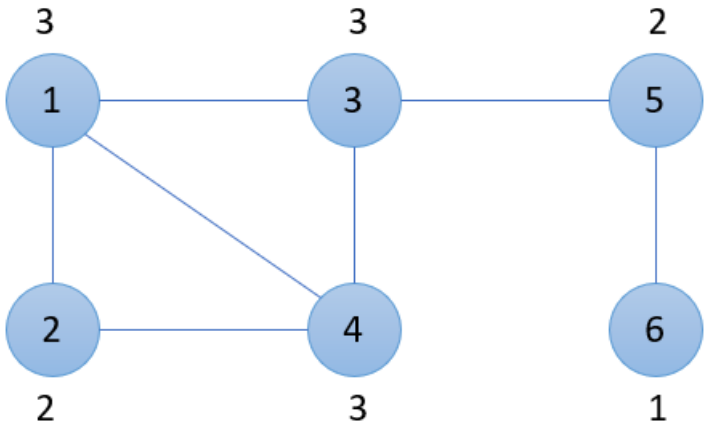


Рисунок 7.5 – Граф зі вказаними степенями його вершин

Напівстепінь виходу вершини – число ребер орієнтованого графа, які починаються у поточній вершині.

На рис. 7.6 зображено всі напівстепені для кожної вершини графа у наступному вигляді: напівстепінь заходження/напівстепінь виходу.

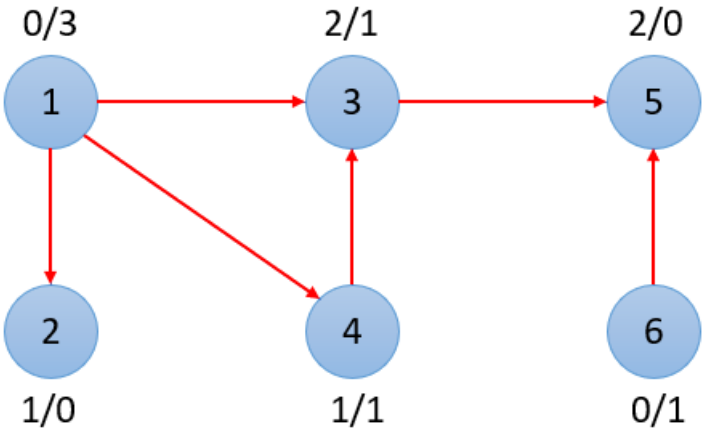


Рисунок 7.6 – Граф зі вказаними напівстепенями його вершин

Граф наступника – це орієнтований граф, в якому кількість напівстепенів виходу кожної вершини дорівнює одиниці, тобто кожна вершина має лише одного наступника. Він складається з одного чи декількох

компонентів зв’язності. Кожна з компонент зв’язності графа наступника має лише один цикл, в який можуть вести декілька різних шляхів.

Приклад графу наступника зображено на рис. 7.7.

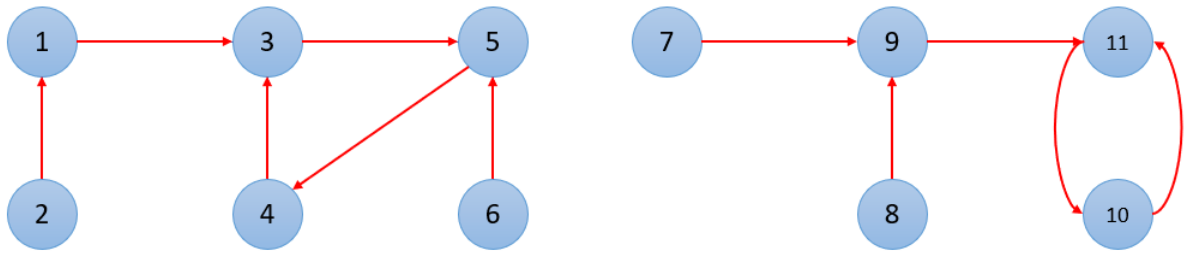


Рисунок 7.7 – Граф наступника

Дводольний граф – це граф, вершини якого можна покрасити в два кольори таким чином, щоб кольори будь-яких сусідніх вершин відрізнялися один від одного. Як правило, графи є дводольними за умови відсутності циклів з непарним числом вершин.

Приклад дводольних графів наведено на рис. 7.8.

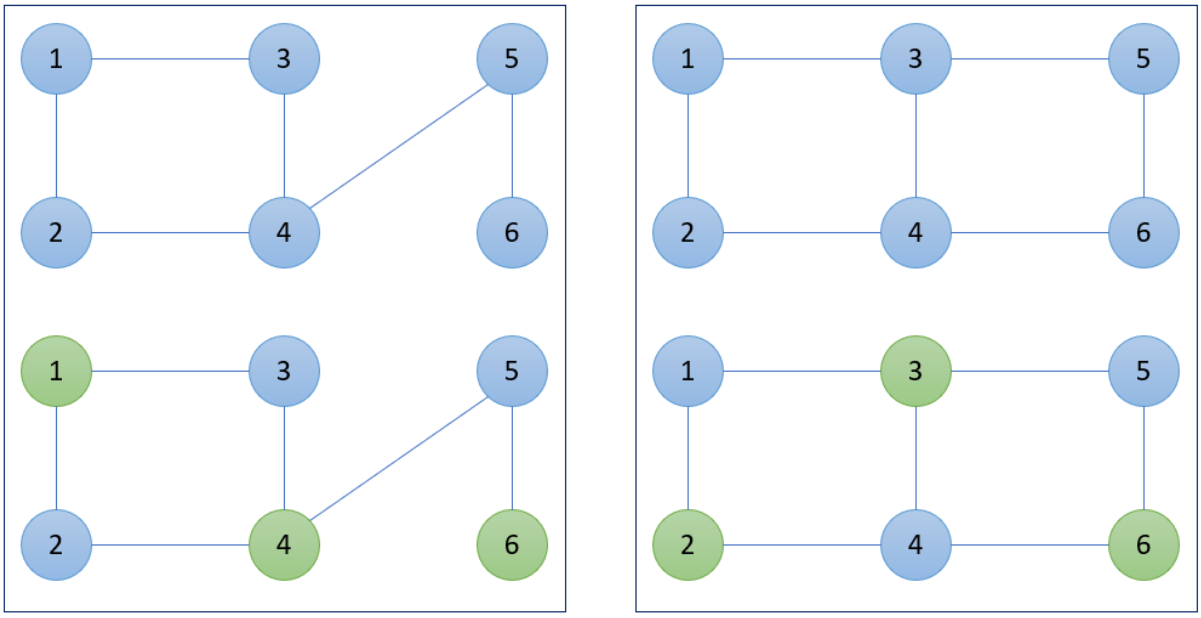


Рисунок 7.8 – Приклади дводольних графів

Наведені вище різновиди графів використовуються у різних задачах для оптимізації пошуку або сортування елементів. Наприклад, пошук найкоротшого шляху в графі допоможе спланувати шлях на автомобілі або відправляти повідомлення в комп'ютерній мережі.

Структури даних для подання графів

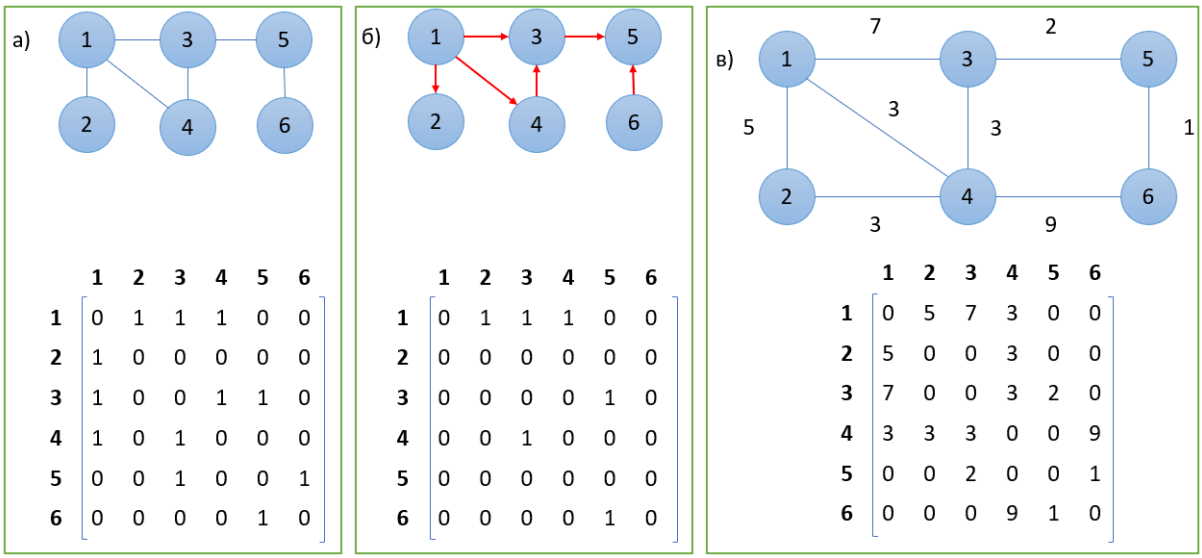
В алгоритми графи подаються декількома способами. Вибір конкретного способу залежить від особливостей алгоритму (наприклад, пошуку), розмірів графу, його типу та інших параметрів. Як правило, використовуються три наступні подання: список ребер, матриця суміжності та списки суміжності.

Список ребер. Він містить перелік усіх ребер графу в певному порядку. Дане подання є зручним у використанні, коли алгоритм обробляє всі ребра і не потрібно знаходити всі з них, що починаються у заданій вершині.

Список ребер зручно зберігати у векторі (масиві, зв'язному списку). У якості параметрів збереження ребра можна використовувати пару значень (a, b), які відповідають за початкову (a) та кінцеву (b) вершини. Для зважених графів необхідно ввести додаткове значення *weight*, яке відповідає за вагу відповідного ребра.

Матриця суміжності. Дане подання показує у вигляді квадратної матриці, які ребра містить граф. У кожній комірці такої матриці записується значення нуль. Якщо з вершини графу B_i до вершини B_j веде ребро, то в комірку $[i, j]$ матриці записується одиниця. У випадку використання зважених графів у комірку записується вага ребра.

Приклади складання матриць суміжності для зв'язного, орієнтовного та зваженого графів наведено на рис. 7.9.



	1	2	3	4	5	6
1	0	1	1	1	0	0
2	1	0	0	0	0	0
3	1	0	0	1	1	0
4	1	0	1	0	0	0
5	0	0	1	0	0	1
6	0	0	0	0	1	0

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	0	0	0	0	0	0
3	0	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	1	0

	1	2	3	4	5	6
1	0	5	7	3	0	0
2	5	0	0	3	0	0
3	7	0	0	3	2	0
4	3	3	3	0	0	9
5	0	0	2	0	0	1
6	0	0	0	9	1	0

Рисунок 7.9 – Матриці суміжності: а) зв’язного графу; б) зв’язного орієнтованого графу; в) зв’язного зваженого графу

Списки суміжності. У цьому випадку для кожної вершини *B* складається список, який містить інформацію про суміжні вершини. Такий варіант подання графів на сьогоднішній день є найбільш популярним, тому що дозволяє ефективно реалізувати більшість алгоритмів, які засновані на використанні графів.

Списки суміжності зручно зберігати в масиві векторів (масиві масивів). У випадку використання орієнтованих графів кожне ребро буде знаходитися лише в одному списку, для неорієнтованих графів – у декількох списках одночасно. Для використання списків суміжності зі зваженими графами необхідно вводити додаткове значення *weight*, яке відповідає за вагу відповідного ребра.

Списки суміжності дозволяють ефективно знаходити вершини, в які можна перейти з поточної по ребру. Програмна реалізація таких списків не є складною, що також впливає на їх популярність серед розробників алгоритмів.

Базові операції пошуку в графах

У даному розділі буде розглянуто два фундаментальних алгоритми на графах: пошук в глибину та пошук в ширину. Суть обох алгоритмів полягає в переміщені з поточної вершини до всіх інших, які мають спільний з нею шлях. Основна відмінність цих алгоритмів полягає лише у визначенні порядку відвідування вершин.

Пошук в глибину. Цей спосіб переміщення по графу є прямолінійним. Алгоритм починає з вхідної вершини та послідовно перебирає всі інші, до яких можна дістатися по ребрам графу.

Даний алгоритм запам'ятовує вже відвідані вершини, тому кожна з них обробляється лише один раз. При цьому пошук в глибину завжди прямує по одному маршруту до того часу, коли на ньому закінчяться невідвідані вершини. Після цього алгоритм повертається назад і шукає інші маршрути, які ведуть до нових вершин.

На рис. 7.10 наведено приклад обробки вершин графа при пошуку в глибину. Пошук може починатися з довільної вершини. У наведеному прикладі початок пошуку лежить у вершині «5». Спочатку алгоритм досліджує шлях 5→3→4→2→1, після чого повертається до вершини «5» і відвідує останню вершину «6».

Пошук в глибину найчастіше реалізовується з використанням рекурсії. Якщо граф задається списками суміжності, то в початковий момент часу усі елементи мають значення *false*. Метод (функція) починає пошук з заданої вершини, і коли алгоритм заходить у нову вершину, то змінює значення на *true*.

Часова складність такого алгоритму складає $O(n+m)$, де n – число вершин графу, а m – число його ребер.

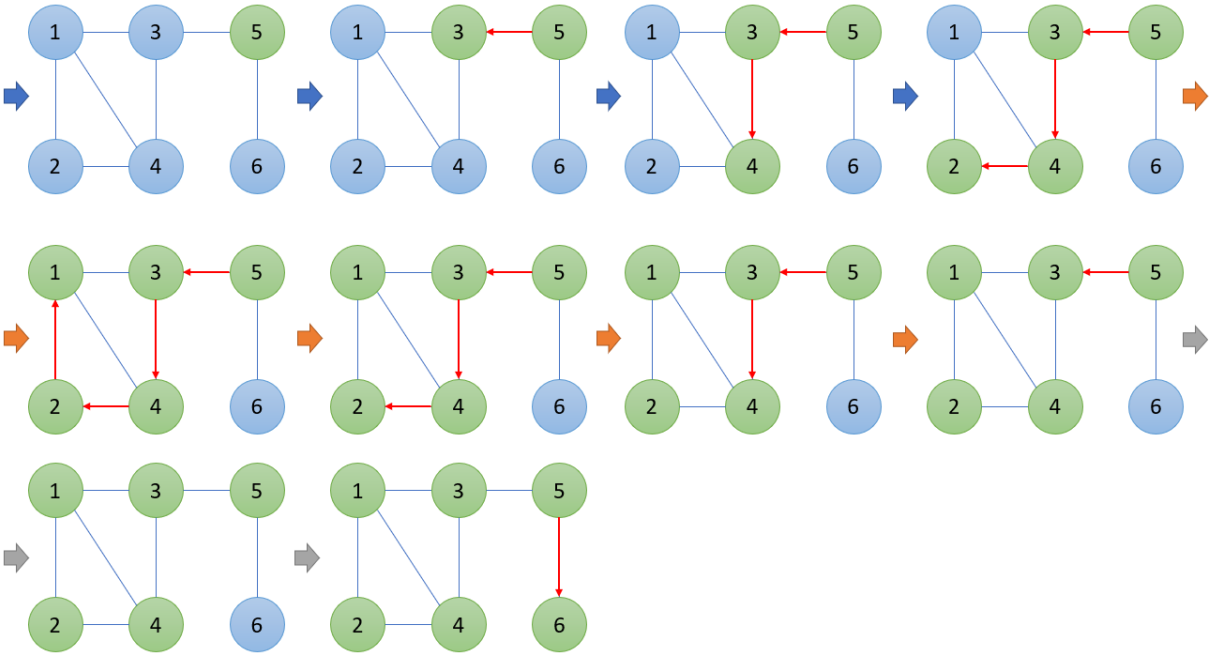


Рисунок 7.10 – Ітерації пошуку в глибину

Пошук в ширину. Цей спосіб полягає у відвідуванні декількох сусідніх до поточної вершин. Алгоритмічно такий варіант реалізувати складніше, ніж пошук в глибину, тим більше, що його асимптотична часова складність також дорівнюватиме $O(n+m)$.

У процесі пошуку в ширину вершини проходяться рівень за рівнем. Спочатку будуть відвідані вершини, які знаходяться на відстані одного ребра від початкової, потім – на відстані два ребра і так далі. Алгоритм продовжить виконання до моменту, коли не залишиться вершин, які не були відвідані.

На рис. 7.11 зображено процес просування вершинами графа при пошуку в ширину. Пошук можу починатися з будь-якої вершини, але в наведеному прикладі початковою вершиною є «5». На першій ітерації відвідуються вершини «3» та «6», які знаходяться на відстані одного ребра, потім – вершини «1» та «4», які віддалені на два ребра від початкової вершини.

Закінчує свою роботу алгоритм після відвідування вершини «2», яка знаходиться на відстані трьох ребер від початкової вершини.

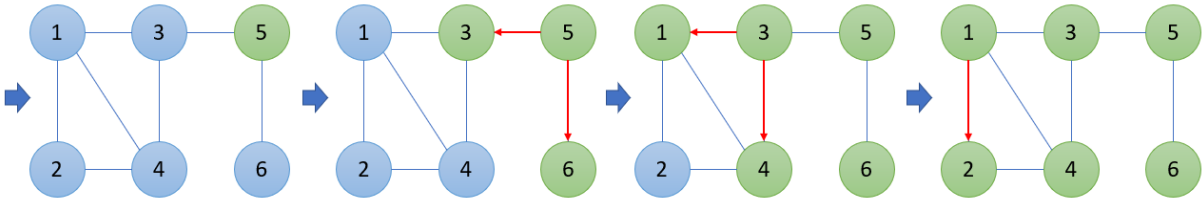


Рисунок 7.11 – Ітерації пошуку в ширину

Складність реалізації даного алгоритму полягає в тому, що необхідно відвідувати вершини, які знаходяться в різних частинах графу. Типова реалізація базується на черзі вершин, де на кожному кроці опрацьовується наступний вузол з черги.

За допомогою алгоритмів пошуку в ширину та глибину можна перевірити основні властивості будь-якого графу. Для рішення подібних задач можна використовувати обидва алгоритми, але на практиці частіше розглядають пошук в глибину через простоту його реалізації.

Серед варіантів використання даних алгоритмів можна назвати наступні:

- перевірка зв’язності: якщо пошук не відвідує всі вершини, то граф не зв’язний;
- пошук циклів: граф буде містити цикл, якщо у процесі пошуку буде знайдено таку вершину, що одну з її сусідніх (крім попередньої на поточному шляху) уже було відвідано; іншим варіантом може бути розрахунок загальної кількості вершин та ребер в кожній компоненті зв’язності: якщо компонента містить x вершин, а кількість ребер, які до неї входять, дорівнює x або більше, то у даній компоненті обов’язково присутній цикл;

- перевірка на дводольність: якщо в якийсь момент алгоритм помічає, що сусідні вершини пофарбовані в один колір, то поточний граф не є дводольним.

Найкоротший шлях в графах

Пошук найкоротшого шляху між двома вершинами графу є однією з найбільш важливих задач, які мають практичне використання. Наприклад, якщо відома карта автомобільних доріг, то можна знайти найкоротший шлях з міста А в місто Б.

Найчастіше використовуються три наступних алгоритми у зважених графах: Белмана-Форда, Дейкстри та Флойда-Уоршела. Розглянемо їх основні особливості.

Алгоритм Белмана-Форда. Він знаходить найкоротші шляхи з початкової вершини до всіх інших. Його можна застосовувати до будь-яких графів, які не містять циклів з від'ємною довжиною шляху.

Даний алгоритм запам'ятовує відстань від початкової вершини до усіх інших. На першому етапі відстань до поточної вершини дорівнює нулю, а до всіх інших – нескінченність. Потім на кожному кроці алгоритм зменшує відстані, шукаючи допустимі ребра з меншою вагою, а зупиняється тоді, коли жодну відстань вже не можна зменшити.

На рис. 7.12 зображено як алгоритм Белмана-Форда проходить граф. Спочатку алгоритм зменшує відстані, використовуючи ребра 1→2, 1→3 та 1→4. Після цього він зменшує відстані за рахунок ребер 2→4 та 3→5, на наступній ітерації – 4→3 та 5→6. Враховуючи, що на попередній ітерації за рахунок кроку 4→3 довжина шляху зменшилася у порівнянні з іншим

шляхом, який проходить через вузол «3», останніми кроками будуть 3→5 та 5→6, що в результаті дасть довжину шляху 10 умовних одиниць.

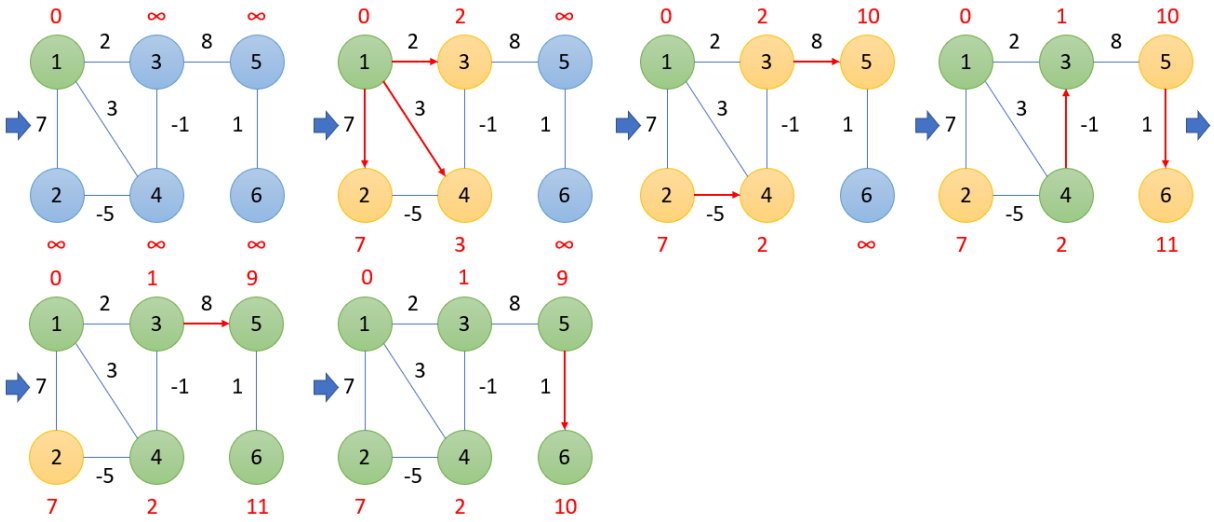


Рисунок 7.12 – Ітерації алгоритму Белмана-Форда

Часова складність такого алгоритму складає *O(nm)*, оскільки на кожній з *n-1* ітерації перебираються всі *m* ребер графу. Якщо граф не має від'ємних циклів, які призводять до нескінченного зменшення довжини шляху, то після *n-1* ітерацій довжини шляхів не можуть змінитися через те, що будь-який найкоротший шлях не може містити більше *n-1* ребер.

Існує також модифікація даного алгоритму, яка виконується швидше, але в межах даного посібника вона розглядатися не буде.

Алгоритм Дейкстри. Аналогічно до попереднього алгоритму він знаходить найкоротші шляхи з початкової вершини до всіх інших. Проте даний алгоритм показує більшу ефективність при використанні з графами великих розмірів, якщо вони не мають ребер з від'ємною вагою.

Алгоритм Дейкстри так само зберігає відстані до вершин графу та зменшує довжину шляху в процесі пошуку. На кожній ітерації він обирає з неопрацьованих вершин ту, відстань до якої є мінімальною. Після цього

алгоритм переглядає всі ребра, які починаються в поточній вершині, та зменшує відповідні шляхи. Ефективність даного алгоритму полягає в тому, що він обробляє кожне ребро тільки один раз. Ця властивість є наслідком відсутності в графі ребер з від'ємною вагою

На рис. 7.13 зображено приклад виконання алгоритму Дейкстри. Аналогічно до попереднього випадку, початкові відстані до всіх вершин (окрім початкової) дорівнюють нескінченності. Алгоритм проходить вершини в порядку 6→5→3→4→2→1. Після обробки будь-якої вершини відстань до неї більше змінюватися не буде.

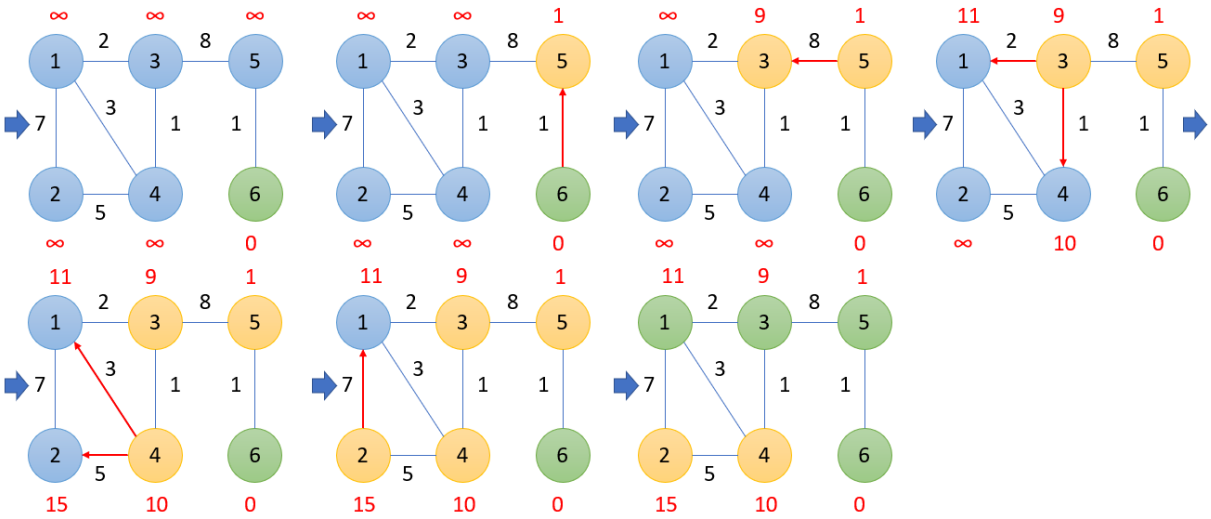


Рисунок 7.13 – Ітерації алгоритму Дейкстри

Часова складність такого алгоритму складає $O(n+m \cdot log(m))$ через те, що алгоритм перебирає всі вершини графу і для кожного ребра додає в чергу тільки одну відстань.

Алгоритм Флойда-Уоршела. Даний алгоритм пропонує дещо інший підхід до пошуку найкоротшого шляху в графі. На відміну від інших алгоритмів, він знаходить найкоротші шляхи між всіма парами вершин за один прохід.

Алгоритм Флойда-Уоршела використовує матрицю, яка містить відстані між парами вершин. У початковий момент часу матриця ініціалізується на базі матриці суміжності зваженого графу. Після цього алгоритм виконує декілька ітерацій, на кожній з яких обирає нову вершину, яка може в подальшому бути проміжною в шляхах, і зменшує відстані з використанням даної вершини.

Приклад роботи алгоритму зображено на рис. 7.14.

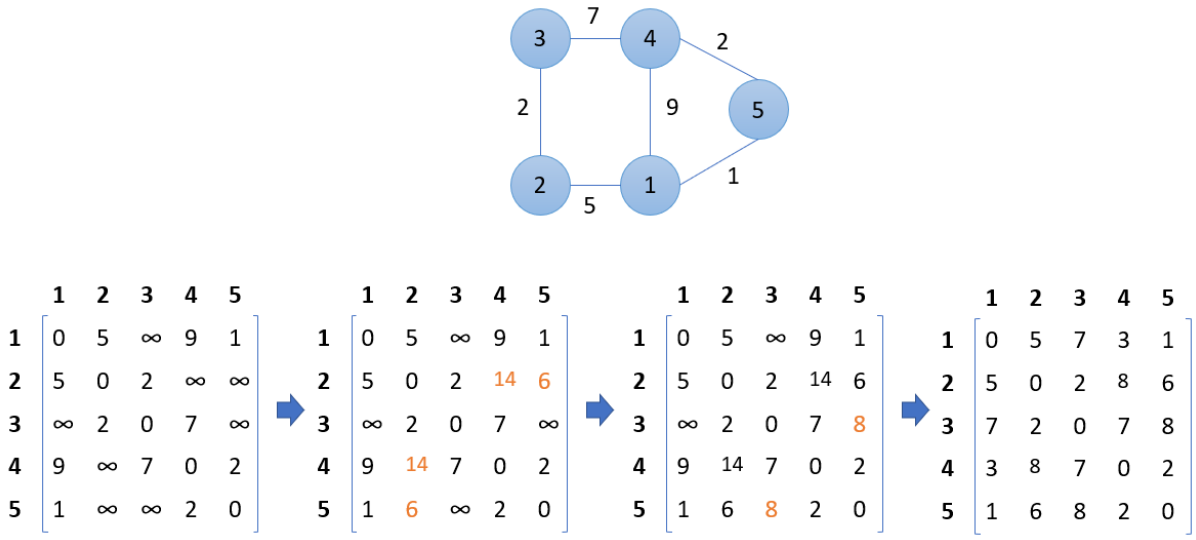


Рисунок 7.14 – Ітерації алгоритму Флойда-Уоршела

На першій ітерації алгоритму новою проміжною вершиною стає «1». Між вершинами «2» та «4» через вершину «1» утворюється новий шлях з довжиною 14 умовних одиниць. Також новий шлях довжиною 6 одиниць утворюється між вершинами «2» та «5».

У межах другої ітерації проміжною вершиною стане вершина «2», що призводить до створення нових шляхів між вершинами «1», «3» та «3», «5». Алгоритм буде продовжувати свою роботу до того часу, коли всі вершини графу побувають у ролі проміжних. Саме тоді матриця суміжності буде містити мінімальні відстані між всіма парами вершин.

Часова складність алгоритму Флойда-Уоршела складає $O(n^3)$, оскільки для перебирання вершин графу знадобиться три вкладених цикли. Перевагою даного алгоритму є те, що його реалізація є простою, а недоліком – можна використовувати тільки для графів малого розміру через кубічну часову складність.

Контрольні запитання.

1. Що таке граф? У чому його суть?
2. Які є види графів? Чим вони відрізняються?
3. Що таке матриця суміжності?
4. Що таке списки суміжності?
5. Який алгоритм пошуку в глибину?
6. Який алгоритм пошуку в ширину?
7. Пояснити як працює алгоритм пошуку найкоротшого шляху Белмана-Форда.
8. Пояснити як працює алгоритм пошуку найкоротшого шляху Дейкстри.
9. Пояснити як працює алгоритм пошуку найкоротшого шляху Флойда-Уоршела.

Лекція №8

Дерева прийняття рішень

Основні терміни та поняття

При роботі зі структурами даних та програмуванні алгоритмів для опису дерев прийняття рішень часто використовуються терміни, які запозичені з садівництва, генеалогії та обчислювальної техніки. Більша частина з них будуть вживатися у тому ж розумінні, як і оригінальні поняття, але в іншому контексті.

Отже, **дерево** – це ациклічний зв'язний граф з ієрархічною структурою, в якому вершини одного рівня ієрархії не можуть бути суміжними. Такий граф у загальному випадку містить *n* вершин та *n-1* ребер.

Приклад дерева у вигляді графа зображено на рис 8.1. Проте такий варіант зображення дерев майже не використовується через складність його сприйняття. Для полегшення розуміння алгоритмів, які використовуються в деревах, класичний та найбільш широко використовуваний вигляд цієї структури даних зображено на рис. 8.2.

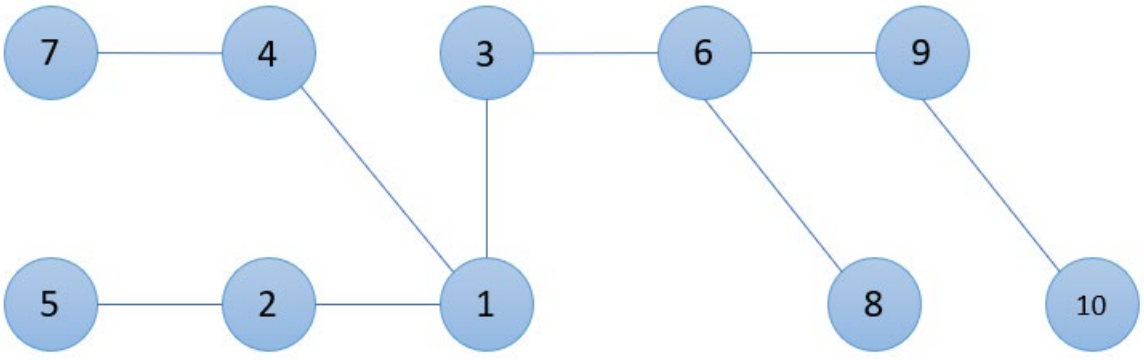


Рисунок 8.1 – Приклад дерева

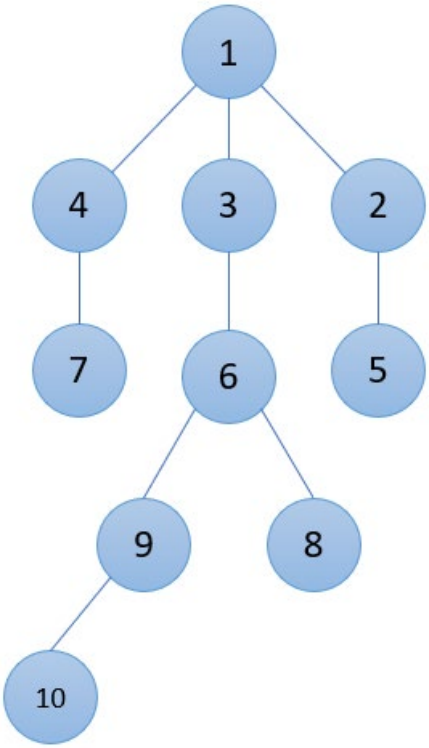


Рисунок 8.2 – Класичне зображення дерева

Подібний вигляд дерева показує ієрархічну (спадкову) структуру та дає початкове розуміння зв’язків між елементами різних рівнів ієрархії: елемент, який знаходиться вище називається **батьківським**; елемент, який має спільні зв’язки з батьківським та знаходяться нижче називається **дочірнім**. Два дочірніх елементи, які мають спільний батьківський, інколи називають **вершинами-сестрами**. Дочірні елементи по відношенню до їх батьківських також називаються **нащадками**, а батьківські елементи батьківських елементів – **предками**.

Будь-яке дерево складається з **вершин (вузлів)** – елементи, які містять дані та з’єднані між собою ребрами.

Гілки (лінії, посилання) – ребра, які з’єднують вершини дерева. Видалення будь-якої гілки розбиває дерево на два компоненти зв’язності.

Листя – вершини дерева, які мають лише одну сусідню (суміжну) вершину.

Корінь – вершина дерева, яка є найвищою у ієрархії, тобто не має предка (батьківської вершини). Кореневе дерево має рекурсивну структуру: кожна вершина може виступати у ролі кореня відповідного піддерева, яке містить дану вершину та всі вершини піддерев її дочірніх вершин. Наприклад, піддерево вершини «6» (рис. 8.2) складається з вершин «6», «8», «9» та «10».

Отже, **піддерево** – частина дерева, яка бере початок у вершині B_i (корінь піддерева) та включає всіх її нащадків.

Найменший (перший) спільний предок – для будь-яких двох вершин це вузол-предок, який розташований найближче до них. Варто звернути увагу, що між будь-якими двома вершинами існує унікальний шлях, який не проходить двічі через одну і туж саму гілку. Він бере початок з першої вершини, проходить вгору до найменшого спільного предка та спускається до другої вершини.

Листова (термінальна) вершина – вершина, яка не має дочірніх вузлів.

Внутрішня вершина – вершина, яка має предка та хоча б одного нащадка.

Глибина (рівень) вершини – відстань від вершини до кореня. Глибина кореневої вершини дорівнює нулю.

Висота вершини – відстань шляху від поточної вершини до листової, тобто до низу дерева.

Степінь вершини – кількість нащадків поточної вершини, яка залежить від типу дерева.

Степінь дерева – найбільше значення степені вершини в поточному дереві.

Повне дерево – це дерево, в якому кожна окрема вершина або не має дочірніх вершин, або їх кількість дорівнює степені дерева (рис. 8.3). Прикладом може бути бінарне дерево.

Бінарне дерево – дерево другої степені. Воно характеризується тим, що у кожної його вершини може бути або жодного, або не більше двох дочірніх вузлів.

Упорядковане дерево – дерево, в якому важливе значення має порядок усіх дочірніх вузлів. Це актуально, тому що більшість алгоритмів можуть по-різному сприймати праву та ліву дочірні вершини. Як правило, на практиці дерева упорядковуються зліва направо незалежно від того, чи вимагає цього певний алгоритм.

Завершене дерево – це дерево, у якого всі рівні ієрархії є повними крім, можливо, але не обов'язково, найнижчого, на якому усі листові вершини зміщуються вліво (рис. 8.3). Тобто завершені дерева можуть бути повними або неповними.

Ідеальне дерево – повне дерево, всі листя якого знаходяться на одному рівні ієрархії (рис. 8.3).

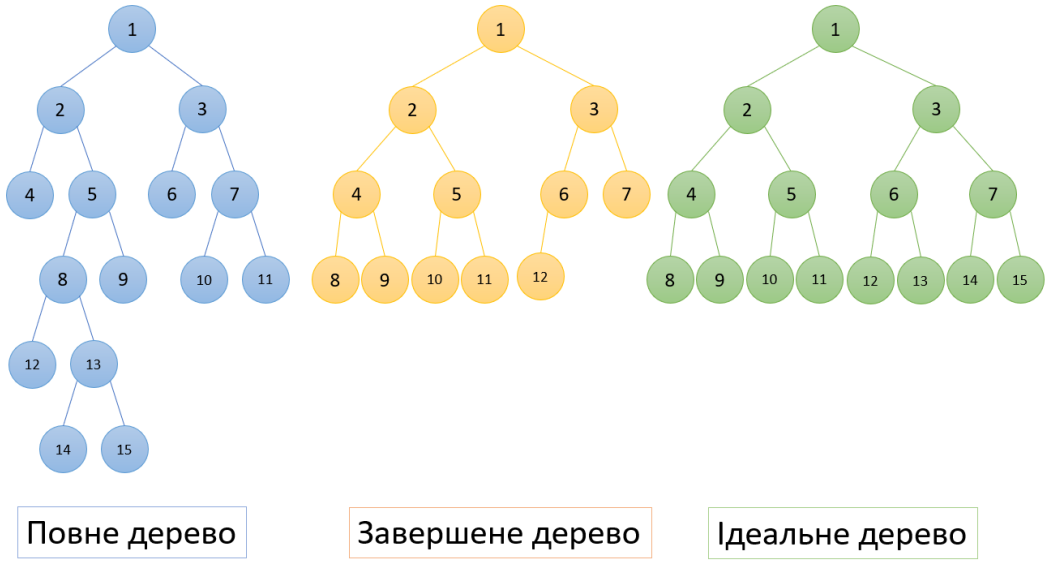


Рисунок 8.3 – Приклад дерев

Ліс – граф, який складається з декількох дерев у вигляді окремих компонентів зв’язності (рис. 8.4).

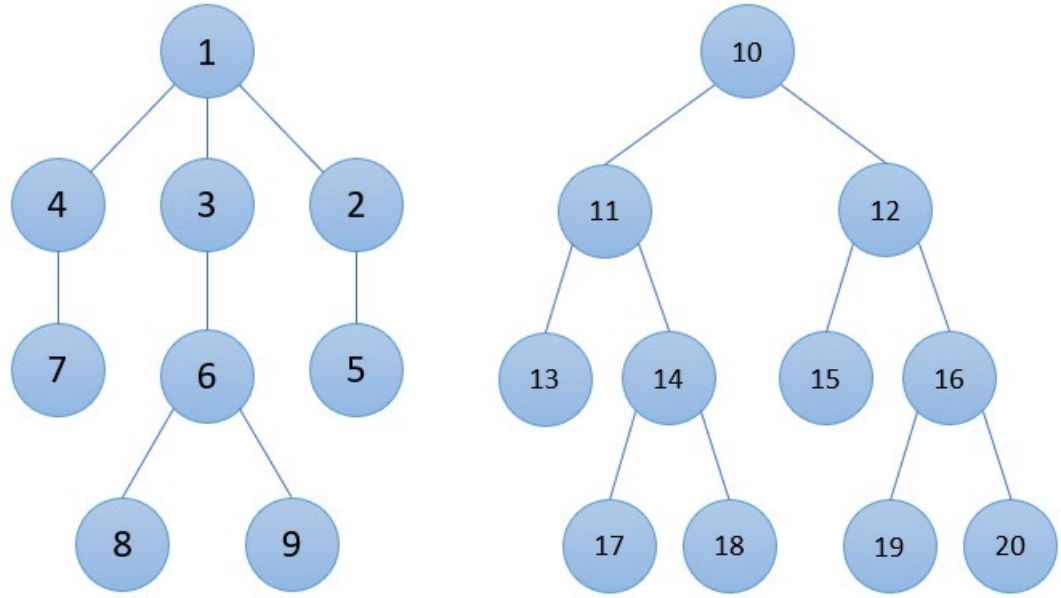


Рисунок 8.4 – Приклад лісу

Збалансоване дерево – це дерево, у якого для кожного вузла висоти лівого та правого піддерев відрізняються не більше, ніж на одиницю.

Бінарні дерева

Бінарне (двійкове) дерево є одним з важливих видів кореневих дерев. Як було сказано вище, воно може бути або порожнім, або містити корінь та не більше двох піддерев: ліве та праве (рис. 8.5).

Для бінарного дерева цей поділ є дуже важливим на відміну від інших видів дерев. Саме тому піддерева, які зображені на рис.8.6, є двома різними. Для інших видів дерев цей поділ може бути менш важливим або взагалі не мати ніякого впливу на алгоритми роботи з ними.

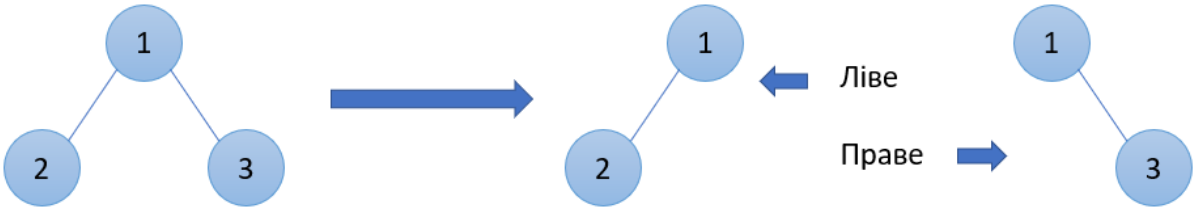


Рисунок 8.5 – Бінарне дерево

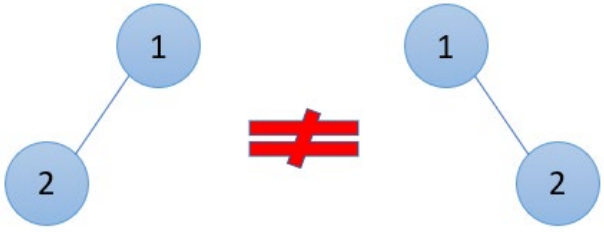


Рисунок 8.6 – Праве та ліве піддерева

Через легкість розуміння та відображення внутрішньої структури бінарні дерева часто використовуються на практиці в різних алгоритмах, тому що більшість задач у своїй основі містить двійковий вибір. Основні властивості бінарних дерев наступні:

1. Кількість гілок g бінарного дерева, яке складається з n вершин, становить $g=n-1$.

2. Кількість вершин n ідеального бінарного дерева з висотою h складає $n=2^{h+1}-1$.

3. Якщо ідеальне бінарне дерево має n вершин, то його висота визначається як $h=log_2(n+1)-1$.

4. Кількість листових (термінальних) вершин l ідеального бінарного дерева з висотою h можна розрахувати за формулою $l=2^h$.

5. Кількість внутрішніх вершин in ідеального бінарного дерева з висотою h та l листовими вершинами можна розрахувати за формулою $in=n-l=2^{h+1}-1-2^h=2\cdot2^h-1-2^h=2^h\cdot(2-1)-1=2^h-1$. Це означає, що кількість листових та внутрішніх вершин майже однакова, тобто $in=l-1$.

6. Кількість місць *m*, в які можна додати дочірню вершину в бінарному дереві з *n* вершинами, розраховується як $m=n+1$.

7. Якщо в бінарному дереві *l* листових вершин та n_2 вершин другого степеня, то термінальних вершин завжди буде на одну більше, тобто $l=n_2+1$.

Завдяки цим властивостям розрахунок часу виконання алгоритмів спрощується. Якщо алгоритму треба буде пройти по всьому ідеальному бінарному дереву, то його асимптотична часова складність складатиме $O(\log(n))$.

У залежності від конфігурації бінарного дерева основа логарифму при розрахунку часової складності виконання алгоритмів може змінюватися, що в межах теорії не впливає на вказану вище оцінку. Проте на практиці можуть виникати ситуації, коли алгоритми з однаковим асимптотичним часом виконуються на одному і тому ж обладнанні за різний час.

У більшості випадків на практиці дерева реалізовуються на основі зв'язних списків. Для подання бінарних дерев, як правило, використовуються поля *info, left ma right*, які присутні у кожної вершини та містять відповідно елемент даних (ключ) або значення, посилання на ліву дочірню вершину, посилання на праву дочірню вершину. Так само, як і в зв'язних списках, «порожнє» чи «нульове» значення посилання, якщо наступна вершина не існує, у подальшому буде позначатися символом «Ω». З бінарним деревом також має бути зв'язане поле *root*, яке містить посилання на його корінь. Якщо $root=\Omega$, то дерево вважається порожнім.

Приклад зображення дерева у вигляді зв'язного списку наведено на рис. 8.7.

Подібна організація даних дозволяє збільшити ефективність виконання операцій додавання та видалення елементів, проте не до всіх бінарних дерев можна застосувати бінарний пошук. У деяких випадках асимптотична

часова складність бінарного пошуку може змінюватися з логарифмічної *O(log(n))* до лінійної *O(n)*.

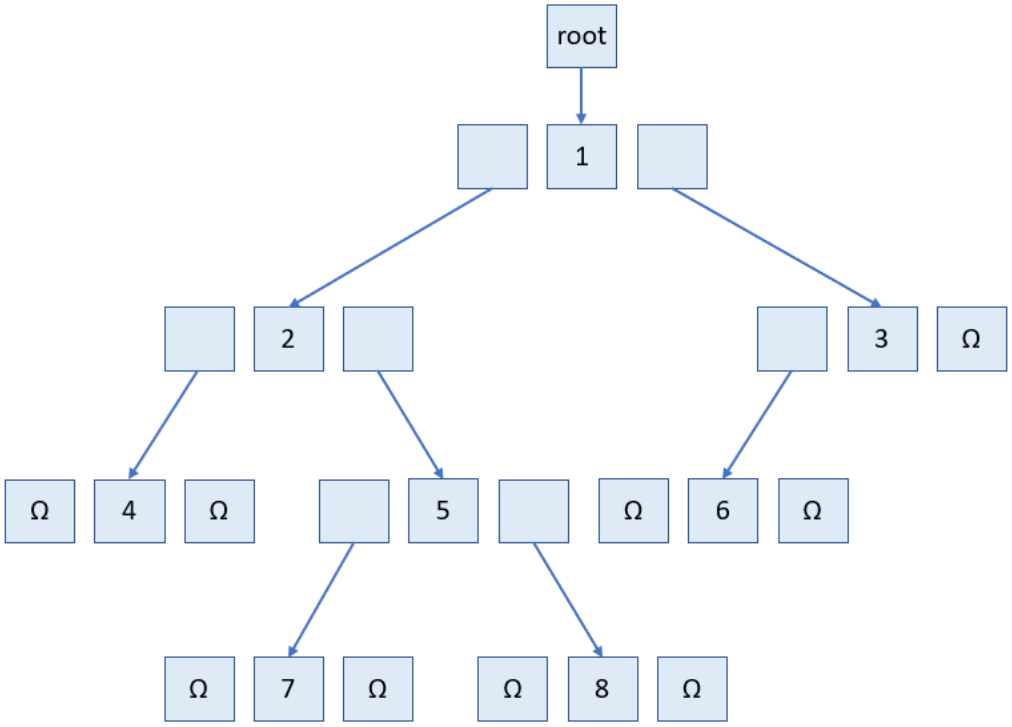


Рисунок 8.7 – Подання зв’язного бінарного дерева

Дерева бінарного пошуку

Дерева бінарного пошуку є структурами даних, які пристосовані до ефективного виконання операцій додавання, видалення та бінарного пошуку елементів множини.

Дерево бінарного пошуку – розширене упорядковане бінарне дерево, усі внутрішні вершини якого відповідають елементам множини $X=\{x_1, x_2, x_3, \ldots, x_n\}$ таким чином, що симетричний порядок проходження вершин співпадає з природним (усі елементи розміщено за зростанням).

Симетричне проходження вершин полягає в наступному:

1) Кожна вершина відвідується лише один раз, але дозволяє проходити повз неї необмежено.

2)Алгоритм спускається по лівому піддереву до лівої термінальної вершини.

3) Після відвідування термінальної вершини відвідується її батьківська.

4) З батьківської вершини йде перехід до правої дочірньої.

5) Пункти 2)-4) повторюються до відвідування кореня дерева.

6) Пункти 2)-4) повторюються, але вже не для лівого, а для правого піддерева до того часу, коли не залишиться жодної невідвіданої вершини.

Для зображеного на рис. 8.8 прикладу симетричний обхід буде відбуватися в наступному порядку: 4→2→5→8→1→6→3→7→9. Враховуючи, що порядок відрізняється від природного, дане дерево не може вважатися деревом бінарного пошуку. Для виправлення цієї проблеми необхідно упорядкувати елементи дерева до вигляду, який зображено на рис. 8.9.

Якщо для впорядкованого дерева провести симетричний обхід вершин, то він буде відбуватися в наступному порядку: 1→2→3→4→5→6→7→8→9, тобто дерево є деревом бінарного пошуку.

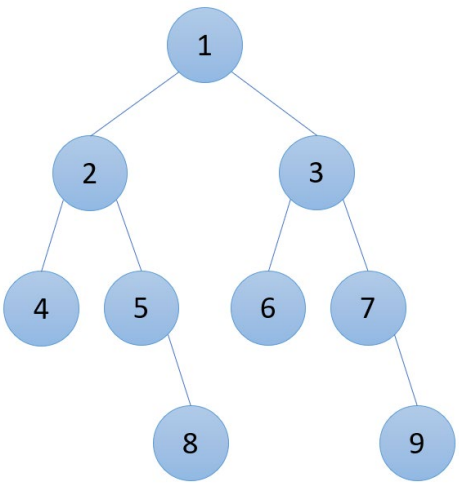


Рисунок 8.8 – Дерево

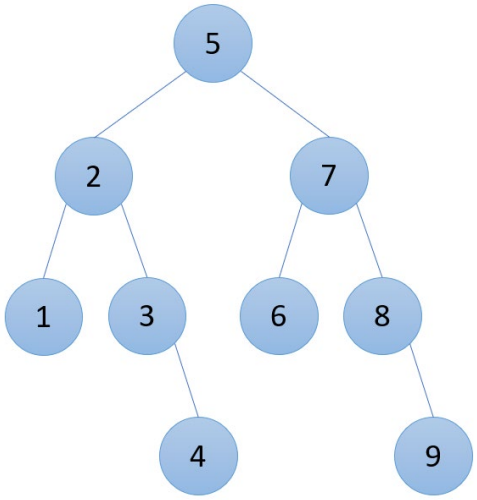


Рисунок 8.9 – Дерево бінарного пошуку

Порожнє дерево також можна віднести до типу дерев бінарного пошуку.

АВЛ-дерева

При роботі з динамічними множинами, в яких часто використовуються операції додавання та видалення елементів, не можна заздалегідь визначити в який момент часу зміниться поточна множина значень. Випадковий характер внесення змін призводить до створення випадкових дерев бінарного пошуку.

Середній час пошуку в таких деревах приблизно в півтора рази більший, ніж середній час пошуку в збалансованих деревах. На практиці для рішення багатьох задач такий приріст часу виконання пошуку є цілком допустимим, але в найгіршому випадку можуть будуватися дерева бінарного пошуку у вигляді лінійних списків з лінійним часом. Саме тому потрібно використовувати додаткові методи для балансування дерев, які будуть гарантувати логарифмічний час пошуку навіть в найгірших випадках, коли

присутня велика кількість операцій додавання та видалення елементів послідовності, а їх послідовність носить випадковий характер.

Якщо при додаванні нової вершини перебудовувати дерево для підтримки його балансу (рис. 8.10), то це дерево, як правило, змінюється повністю (практично жодне відношення між батьківською та дочірньою вершинами не залишається без змін). Такі зміни вимагають часу, який є пропорційним числу вершин дерева. Для рішення даної проблеми використовуються методи, які вносять лише локальні зміни в дерево таким чином, що відхилення від повністю збалансованого бінарного дерева є незначними.

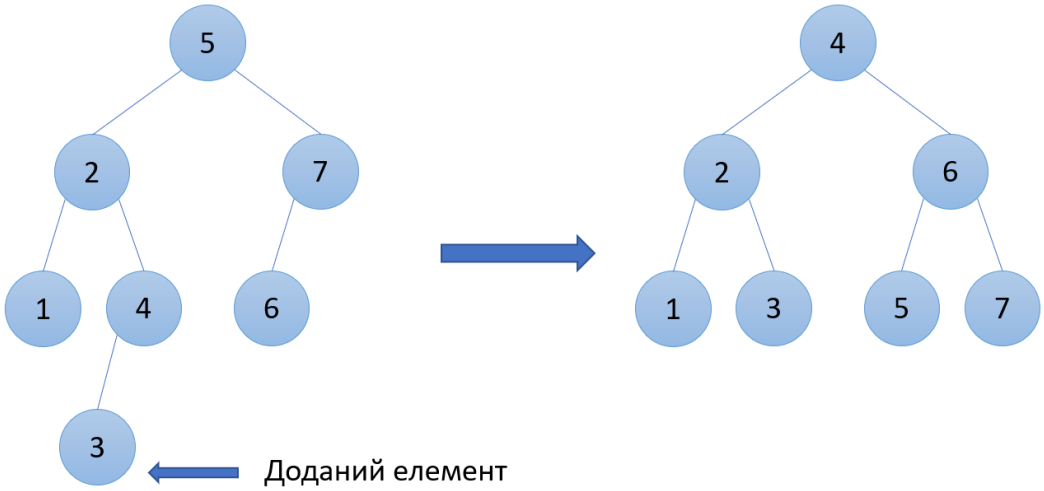


Рисунок 8.10 – Перебудова до повністю збалансованого бінарного дерева

АВЛ-дерево (в честь авторів Г.М. Адельсона-Вельського та Є.М. Ландіса) – це дерево бінарного пошуку, яке є збалансованим по висоті, тобто два його піддерева (T_l – ліве; T_r – праве) задовольняють наступні умови:

- T_l та T_r є збалансованими по висоті дерева h;
- $|h(T_l) - h(T_r)| \leq 1$.

Отже, в АВЛ-деревах для кожної вершини висоти правого та лівого піддерев не більше, ніж на одиницю. Дерево, яке містить єдину вершину, також являється АВЛ-деревом.

Середній час пошуку в асиметричних АВЛ-деревах всього на 5%-10% більше часу пошуку в повністю збалансованих деревах і зберігає логарифмічний характер зміни. Проте на практиці хоч і з дуже низькою ймовірністю, але все одно можуть зустрічатися сильно асиметричні АВЛ-дерева, для яких час пошуку може бути в півтора рази більшим. Таким чином, використання АВЛ-дерев є одним з варіантів підтримання логарифмічного часу виконання алгоритмів при випадковому порядку операцій додавання та видалення елементів множини.

Основною операцію для відновлення балансу (часто характеризується певним коефіцієнтом) в АВЛ-дереві є обертання. Вона являється локальною і забезпечує виконання вимоги впорядкованості вершин при симетричному їх проходженні. У деяких випадках для відновлення балансу в дереві необхідно послідовно виконати два обертання, які інколи об’єднують в одну операцію подвійного обертання.

Червоно-чорні дерева

Червоно-чорне дерево – збалансоване дерево бінарного пошуку, вершини якого розділені на червоні та чорні, а рівні будь-яких двох листових вершин відрізняються не більше, ніж в два рази.

Так само, як і в АВЛ-деревах, спеціальні операції балансування при роботі з червоно-чорними деревами дозволяють гарантувати, що виконання основних операцій з динамічною множиною не перевищить логарифмічного часу.

Оскільки червоно-чорне дерево є розширеною версією дерева бінарного пошуку, воно також будується на базі зв'язних списків з тим же набором полів. Проте для червоно-чорних дерев додатково треба використовувати поле *color* для кожної вершини, яке буде містити інформацію про відповідний колір.

На практиці для зручності часто приймається припущення, що кожна вершина, яка має ім'я, також має два дочірні вузли і вважається внутрішньою. У цьому випадку, якщо поля *right* та/або *left* вершини містять порожні посилання Ω, то це означає, що вершині надаються фіктивні листові вузли, які називаються Ω-синами (дочками). Ці фіктивні вузли мають таку ж структуру, як і звичайні вершини дерева. Дуже часто на практиці використовують єдиний фіктивний вузол, який відповідає усім Ω-синам, для спрощення реалізації алгоритмів.

Отже, дерево бінарного пошуку називається червоно-чорним, якщо виконуються наступні умови:

1. Кожна вершина має бути або червоною, або чорною.

2. Кожна листова вершина – чорна.

3. Якщо вершина червона, то обидва її дочірні вузли будуть чорними.

4. Усі шляхи, які ведуть від кореня до термінальних вузлів, мають містити однакову кількість чорних вершин.

Операції додавання та видалення елементів змінюватимуть дерево таким чином, що результат може не задовольняти наведених вище умов. Для відновлення властивостей червоно-чорного дерева необхідно змінити структуру дерева та перефарбувати деякі вершини. Структура дерева зазвичай змінюється за допомогою виконання локальних операцій правого та лівого обертання за аналогією до АВЛ-дерев.

Вважатимемо, що корінь червоно-чорного дерева завжди є чорним (рис. 8.11). Також необхідно підтримувати виконання цієї умови під час перебудови структури дерева.

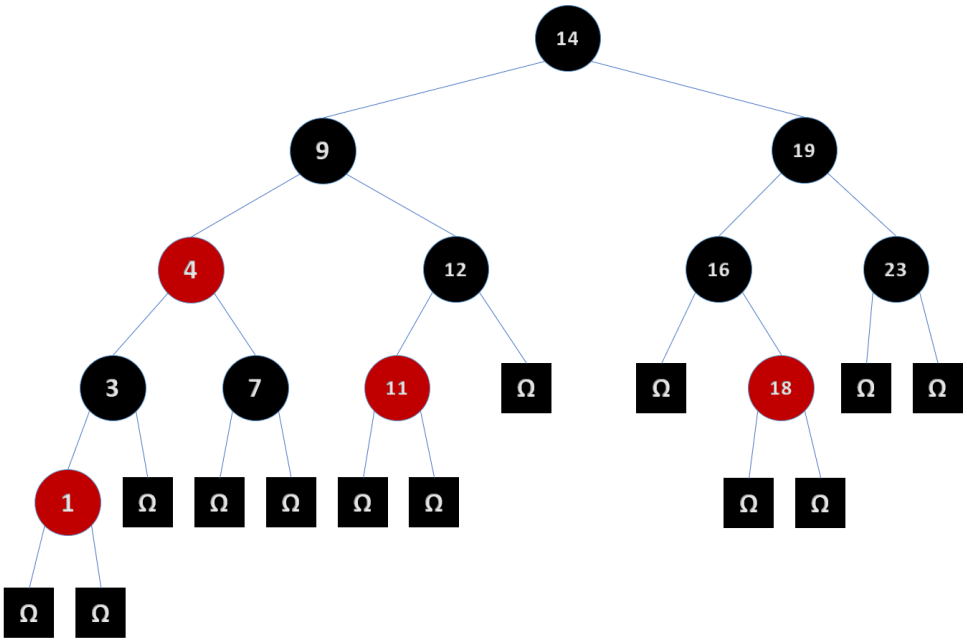


Рисунок 8.11 – Червоно-чорне дерево

Якщо уважно ознайомитися з наведеним на рис. 8.11 деревом, то можна побачити, що четверта умова для нього виконується, тобто всі шляхи від кореня містять однакову кількість вершин чорного кольору.

Чорна висота вершини – кількість чорних вершин від поточної (не включаючи її) вершини до будь-якої термінальної. Чорна висота (з урахуванням четвертої умови) буде однаковою для всіх піддерев, які утворюються з коренем у поточній вершині. Наприклад, якщо позначити чорну висоту як *black_h*, то *black_h(4)=2*, *black_h(9)=2*, *black_h(14)=3*, тощо.

Чорна висота дерева дорівнює чорній висоті його кореня. Якщо червоно-чорне дерево має *n* внутрішніх вершин, то його висота складатиме не

більше, ніж $2 \cdot log(n+1)$. Саме тому асимптотична часова складність алгоритму пошуку для дерева з висотою h буде дорівнювати $O(h)$, тобто $O(log(n))$.

Пірамідальне сортування

Бінарна (двійкова) купа – це повне (інколи майже повне) бінарне дерево, в якому значення кожної вершини є більшим за значення її дочірніх вузлів.

За допомогою бінарних куп часто створюють черги з пріоритетом. Це стає можливим через те, що чим більший елемент послідовності, тим ближче він до кореня дерева. Проте при видаленні елемента буде руйнуватися пірамідальна структура купи. Її можна відновити, якщо перемістити останній елемент в корінь і перебудувати дерево шляхом порівняння батьківської вершини з дочірніми: якщо значення дочірньої вершини є більшим за значення батьківської, то їх необхідно поміняти місцями. Таку процедуру потрібно виконати з усіма вершинами вниз по дереву.

Бінарну купу можна формувати на основі масиву (рис. 8.12).

```
function arrayToHeap(array[]){
    for (i ← 0; i < array.length; i++) do
        index ← i
        while (index != 0) do
            parent  ← (index-1)/2
            if (array[index] ≤ array[parent]) then
                // вийти з циклу
            end if
            temp ← array[index]
            array[index] ← array[parent]
            array[parent] ← temp
            index ← parent
        end while
    end for
}
```

Рисунок 8.12 – Псевдокод для створення купи на базі масиву

Алгоритм пірамідального сортування (рис. 8.13) починає свою роботу з перебудови масиву в бінарну купу. Після цього він декілька разів має видалити її верхній (найбільший) елемент і перемістити його в кінець. Це дозволяє досягти правильної організації елементів всередині купи.

```
function sortByPyramid(array[]){
    arrayToHeap(array)
    for (i ← array.length-1; i == 0; i--) do
        temp ← array[0]
        array[0] ← array[i]
        array[i] ← temp
        // визначити елемент в положенні і, який треба видалити з купи
        // тепер купа буде містити і-1 елементів
        // спустити значення нового вузла вниз для відновлення властивостей купи
    end for
}
```

Рисунок 8.13 – Псевдокод пірамідального сортування

Для даного алгоритму дуже легко розрахувати асимптотичну об'ємну складність, яка складає $O(n)$ через те, що алгоритм зберігає всі дані всередині масиву і використовує обмежену кількість додаткових змінних для перестановки значень.

Асимптотичну часову складність пірамідального сортування визначити складніше. Враховуючи той факт, що дерево є бінарним, кількість часу для пересування елементу вгору по дереву займатиме $O(log(n))$. Відновити властивість після додавання елементу купи необхідно n разів, тому час побудови купи складатиме $O(n \cdot log(n))$.

Для завершення сортування алгоритму потрібно буде видалити кожен елемент з купи і відновити її властивості. На це йому знадобиться також $O(n \cdot log(n))$ часу. Отже, в сумі асимптотична часова складність пірамідального сортування складатиме $O(n \cdot log(n))+O(n \cdot log(n))=O(n \cdot log(n))$.

Контрольні запитання.

1. Чим дерево відрізняється від графа? Навести приклади.

2. Назвати основні елементи дерев та пояснити пов’язані з ними визначення.

3. Які існують види дерев? Навести приклади.

4. Які основні властивості бінарних дерев?

5. Як будується бінарне дерево у вигляді зв’язного списку? Навести приклади.

6. Чим АВЛ-дерево відрізняється від бінарного? Навести приклади.

7. Чим червоно-чорне дерево відрізняється від бінарного? Навести приклади.

8. Які умови необхідно виконати для побудови червоно-чорного дерева?

9. У чому полягає суть пірамідального сортування?

10*. Що таке фіктивний вузол та які основні функції він виконує в деревах?

11*. Доповнити алгоритм пірамідального сортування, зображений на рис. 8.12.

12*. Яким чином можна видалити елемент з бінарної купи та відновити її базові властивості? Навести та пояснити код алгоритму.

* – питання підвищеної складності для самостійного опрацювання

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Креневич А.П. Алгоритми і структури даних. Підручник. К.: ВПЦ "Київський Університет", 2021, 200 с.

2. Коваль В.С., Струбицький П.Р. Алгоритми і структури даних: навчальний посібник. Тернопіль: ФОП Шпак В. Б, 2017, 74 с.

3. Махровська Н.А., Погромська Г. С. Алгоритми і структури даних: навчально-методичний посібник. Миколаїв: МНУ ім. В.О.Сухомлинського, 2019, 279 с.

4. Мелешко Є.В., Якименко М.С., Поліщук Л.І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. Кропивницький: ФОП Лисенко В.Ф., 2019, 156 с.

5. Ільман В.М., Іванов О.П., Панік Л.О. Алгоритми, дані і структури: навч. посіб. Дніпро: Дніпропет. нац. ун-т залізн. трансп.ім. акад. В. Лазаряна, 2019, 134 с.

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

7. Stephens R. Essential Algorithms. Chichester: John Wiley &Sons, 2013, 624 pages.

8. Louridas P. Real-World Algorithms: A Beginner's Guide. Cambridge, Massachusetts: The MIT Press, 2017, 528 pages.