О. О. Коваленко, О. М. Ткаченко, Р. Ю. Чехместрук

Алгоритми та структури даних

Міністерство освіти і науки України
Вінницький національний технічний університет

О. О. Коваленко, О. М. Ткаченко, Р. Ю. Чехместрук

Алгоритми та структури даних

Електронний навчальний посібник

Вінниця
ВНТУ
2025

УДК 004.415.2043+004.421
К56

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 2 від 23.08.2024 р.)

Рецензенти:
Р. М. Бабаков, доктор технічних наук, доцент
Н. В. Добровольська, кандидат технічних наук, доцент
Є. А. Паламарчук, кандидат технічних наук, доцент

Коваленко, О. О.

К56 Алгоритми та структури даних : навчальний посібник [Електронний ресурс] / Коваленко О. О., Ткаченко О. М., Чехместрук Р. Ю. – Вінниця : ВНТУ, 2025. – PDF, 113 с.

Посібник присвячений матеріалам лекційного курсу з дисципліни «Алгоритми та структури даних» для здобувачів, які навчаються за спеціальністю 121 «Інженерія програмного забезпечення» денної та заочної форм навчання.

Мета посібника – надати здобувачам можливість більш детально вивчити аудиторний матеріал, опрацювати теми, відведені на самостійну роботу і підготуватися до іспиту, а також застосувати отримані знання для подальшої фахової роботи.

Перелік та зміст тем відповідає програмі вказаної вище дисципліни.

УДК 004.415.2043+004.421

© ВНТУ, 2025

ЗМІСТ

ВСТУП

Основні поняття алгоритмізації, базові алгоритми та структури даних є основою для професійної роботи фахівців з програмної та комп'ютерної інженерії. Розглянуті навчальні ресурси сформовані на основі досвіду авторів та здобувачів спеціальності «Інженерія програмного забезпечення» розробки та впровадження інноваційних і навчальних ІТ-проєктів [1].

Для чого вивчати алгоритми? Це перше питання, на яке ми хочемо відповісти у вступі до навчального посібника. Відповіді на ці питання можна знайти в опитуваннях досвідчених програмістів на багатьох професійних форумах в мережі Інтернет, в шкільних та вузівських підручниках з інформатики, програмування та теорії алгоритмів. Такі дискусії часто піднімаються серед початківців та менеджерів ІТ-проєктів. Але головна відповідь полягає в тому, що алгоритми дозволяють чітко формалізувати задачу, виявити критичні точки зміни стану системи, яку ви реалізуєте, використати існуючі оптимальні рішення реалізації різноманітних підпрограм, оптимізації вхідних і вихідних даних. Алгоритми є життєво потрібними складовими для розв'язання будь-яких задач і особливо в сфері комп'ютерних наук. Алгоритми відіграють ключову роль у сучасному розвитку технологій в таких напрямах, як:

• розв'язання математичних рівнянь різної складності, знаходження добутку матриць, обернених матриць;

• знаходження оптимальних шляхів транспортування товарів і людей;

• знаходження оптимальних варіантів розподілення ресурсів між різними вузлами та системами;

• знаходження послідовностей, які збігаються;

• пошук інформації в глобальній мережі Інтернет;

• прийняття фінансових рішень в електронній комерції;

• обробка та аналіз аудіо- та відеоінформації;

• розробка різноманітних програмних продуктів автоматизації діяльності людини, підприємств та організацій;

• розробка та впровадження Інтернет-речей;

• розробка та впровадження сучасних технологій вимірювання та подання інформації.

Цей список можна ще довго продовжувати, але ми навели його для того, щоб було зрозуміло – все починається з постановки задачі та алгоритмів її реалізації. Крім того, після появи книги Д. Кнута «Мистецтво програмування», вже немає сумнівів, що алгоритмізація та програмування – це захопливий, цікавий процес створення нових програмних продуктів. Може, хтось з читачів нашого посібника буде автором нових алгоритмів. Алгоритми завжди базуються на обмеженнях та цільових настановах. Золоте ж правило розробника відповідає гаслу: «Можливо, найбільш важли-

вим принципом для гарного розробника алгоритмів є відмова від того, щоб бути задоволеним результатом» [2]. Теорія та практика створення й використання алгоритмів – це сучасна дисципліна, яка постійно вдосконалюється, знання з якої активно використовуються в практичній та науково-дослідній діяльності.

І, як останній аргумент щодо потреби у знаннях базових алгоритмів та структур даних – моніторинг питань на співбесідах при відборі фахівців для роботи в престижних ІТ-компаніях показав, що питання з алгоритмів, логічного мислення та математичних залежностей займають третину з усіх питань співбесіди.

Навчальний посібник відповідає програмі з нормативної дисципліни «Алгоритми та структури даних» галузі знань «Інформаційні технології» спеціальності «Інженерія програмного забезпечення. В посібнику розглянуто 14 основних тем, сформовано контрольні питання для перевірки знань. Посібник подано як електронний ресурс. У процесі написання навчальних матеріалів було використано відкриті навчальні ресурси університетів світу, книги та підручники, які стали вже класикою в теорії та практиці алгоритмізації, матеріали відкритих дистанційних курсів, форумів професійних фахівців з програмної інженерії, результати науково-дослідної роботи авторів і здобувачів, навчальні проєкти здобувачів ВНТУ.

1 ОСНОВНІ ПОНЯТТЯ АЛГОРИТМІВ ТА ПРИКЛАД ПЕРШОГО АЛГОРИТМУ

1.1 Поняття алгоритму

Поняття алгоритму є одним з фундаментальних понять і тому не існує єдиного строгого означення, що таке алгоритм. Наприклад, в [3] автори наводять таке неформальне означення алгоритму.

Алгоритм (algorithm) – це будь-яка коректно визначена обчислювальна процедура, на вхід (input) якої подається певна величина чи набір величин, а результатом виконання якої є вихідна (output) величина чи набір значень. Таким чином, алгоритм є послідовністю обчислювальних кроків, що перетворюють вхідні величини у вихідні.

Алгоритм також можна розглядати як інструмент, призначений для вирішення коректно поставленого обчислювального завдання (computational problem). У постановці завдання загалом задаються співвідношення між входом і виходом. В алгоритмі описується конкретна обчислювальна процедура, за допомогою якої вдається домогтися виконання зазначених співвідношень.

Дещо інше означення алгоритму можна зустріти в [4].

Під алгоритмом будемо розуміти скінченну чітко сформульовану сукупність вказівок, які визначають послідовність дій виконавця, спрямованих на досягнення мети або розв’язання задач певного класу за скінченний проміжок часу.

У будь-якому випадку нас більше цікавить не формальне означення алгоритму, а відповідь на питання, чи може деяка обчислювальна процедура вважатися алгоритмом. А для цього потрібно перевірити, чи має згадана процедура властивості, притаманні алгоритмам. Отже розглянемо ці властивості більш детально.

1.2 Властивості та форми подання алгоритмів

По-перше, що потрібно зазначити стосовно будь-якого алгоритму, – це те, що він застосовується до вхідних даних і видає результати. У звичних технічних термінах це означає, що алгоритм має **входи й виходи**. Крім того, в процесі роботи алгоритму з’являються проміжні результати, які використовуються надалі. Таким чином, кожний алгоритм має справи з даними – вхідними, проміжними та вихідними.

По-друге, дані для свого розміщення вимагають **пам’яті**. Пам’ять, зазвичай, вважається однорідною й дискретною, тобто складається з однакових комірок, причому кожна комірка може містити один символ алфавіту даних. Таким чином, одиниці вимірювання обсягу даних і пам’яті узгоджені. При цьому пам’ять може бути нескінченною. Питання про те,

потрібна спільна чи окрема пам'ять для кожного з трьох видів даних (вхідних, вихідних і проміжних), вирішується по-різному.

По-третє, алгоритм складається з окремих елементарних кроків або дій. І кожна така дія має бути виконана ще до виконання наступної дії. Дана властивість алгоритму отримала назву **дискретність**. Типовий приклад множини елементарних дій – система команд комп'ютера. Зазвичай елементарний крок має справу з фіксованим числом символів (це зручно, наприклад, для вимірювання часу роботи алгоритму числом виконаних кроків).

Властивість **зрозумілість** означає, що алгоритм може бути виконаний, якщо цілком зрозуміла кожна дія (команда), і вона може бути виконана строго за її призначенням.

По-четверте, послідовність кроків алгоритму має бути такою, що після кожного кроку або вказується який крок робити далі, або дається команда зупинки, після чого робота алгоритму вважається завершеною. Така властивість алгоритму отримала назву **точність.**

Часто під одночасним виконанням властивостей зрозумілість та точність розуміють таку властивість як **визначеність або детермінованість.**

По-п'яте, логічно від алгоритму чекати **результативності,** тобто зупинки після кінцевого числа кроків (що залежить від даних) з отриманням результату. Дана властивість і отримала назву результативність. Зокрема, будь-хто, хто розробляє алгоритм розв'язання деякого завдання, наприклад, обчислення функції f(x), зобов'язаний показати, що алгоритм зупиняється після кінцевого числа кроків (як кажуть, збігається) для будь-якого x з області завдання. Однак перевірити збіжність набагато складніше. Збіжність, звичайно, не вдається встановити простим переглядданням опису алгоритму, а загального методу перевірки збіжності, придатного для будь-якого алгоритму A і будь-яких даних x, взагалі не існує.

По-шосте, кожен алгоритм має мати властивість **масовості**. Ця властивість передбачає, що такий алгоритм придатний для розв'язання будь-якої задачі з деякого класу задач. Проте цю властивість не потрібно розуміти як можливість розв'язання багатьох задач, оскільки у деяких випадках клас може складатися лише з однієї задачі.

По-сьоме, потрібно розрізняти: а) опис алгоритму (інструкції або програми); б) механізм реалізації алгоритму (наприклад, за допомогою комп'ютера), що містить засіб запуску, зупинки, реалізації елементарних кроків, видачі результатів і забезпечення управління ходом обчислення; в) процес реалізації алгоритму, тобто, послідовність кроків, що буде породжена у разі застосування алгоритму до конкретних даних [5].

На практиці найбільш поширені такі форми подання алгоритмів [6]:

- **словесна** (записи природною мовою);
- **графічна** (зображення у вигляді графічних символів);

- у вигляді **псевдокоду** (напівформалізовані описи алгоритмів умовною алгоритмічною мовою, що містить як елементи мови програмування, так і фрази природної мови, загальноприйняті математичні позначення тощо);

- програмна (тексти мовою програмування)

1.3 Перший алгоритм. Сортування вставкою

Наш перший алгоритм, алгоритм сортування методом вставок, призначений для вирішення задачі сортування (sorting problem). Наведемо її формулювання [7].

Вхід: послідовність із n чисел $a_1, a_2, \ldots, a_n$.

Вихід: перестановка (зміна порядку) $a'_1, a'_2, \ldots, a'_n$ вхідної послідовності таким чином, що для її членів виконується співвідношення $a'_1 <= a'_2 <= \ldots <= a_n$.

Числа, що сортуються, називають ключами.

Наше вивчення алгоритмів починається з розгляду сортування вставкою (insertion sort). Цей алгоритм ефективно працює, коли мова йде про сортування невеликої кількості елементів. Сортування вставкою нагадує спосіб, до якого звертаються гравці, щоб сортувати наявні на руках карти. Нехай спочатку в лівій руці немає жодної карти, і всі вони лежать на столі сорочкою вгору. Далі зі столу береться по одній карті, кожна з яких поміщається у потрібне місце серед карток, що знаходяться у лівій руці. Щоб визначити, куди потрібно помістити чергову карту, її масть та значення порівнюється з мастю та значенням карт у руці. Припустимо, порівняння проводиться в напрямку зліва направо. В будь-який момент часу карти в лівій руці буде відсортовано, та це будуть ті карти, які спочатку лежали в колоді на столі.

Псевдокод сортування методом вставок наведено нижче під назвою INSERTION_SORT. На його вхід подається масив A [1..n], який містить послідовність з n чисел, що сортуються (кількість елементів масиву A позначено в цьому коді як length [A]. Вхідні числа сортуються без використання додаткової пам'яті: їх перестановка проводиться в межах масиву, а обсяг використовуваної при цьому додаткової пам'яті не перевищує деяку постійну величину. Після роботи алгоритму INSERTION_SORT вхідний масив містить відсортовану послідовність:

```
INSERTION_SORT(A)
1 for j ← 2 to length[A]
2       do      key ← A[j]
3       //Вставте елемент A[j] у відсортовану послідовність A[1..j − 1]
4               i ← j − 1
5               while i > 0 та A[i] > key
6                       do      A[i + 1] ← A[i]
7                               i ← i − 1
8               A[i + 1] ← key
```

Індекс j вказує «поточну карту», яка береться в руку. На початку кожної ітерації зовнішнього циклу for з індексом j масив A складається з двох частин. Елементи A [1..j - 1] відповідають відсортованим картам у руці, а елементи A [j + 1..n] – стопці карт, які поки що залишилися на столі. Зауважимо, що елементи A [1..j - 1] спочатку теж знаходилися на позиціях від 1 до j - 1, але в іншому порядку, а тепер вони відсортовані. Назвемо цю властивість елементів A [1..j - 1] **інваріантом** циклу (loop invariant) та сформулюємо його ще раз.

На початку кожної ітерації циклу for з рядків 1–8 підмасив A [1..j − 1] містить ті ж елементи, які були в ньому з самого початку, але розташовані у відсортованому порядку.

Інваріанти циклу дозволяють зрозуміти, чи правильно працює алгоритм. Потрібно показати, що інваріанти циклів мають такі три властивості.

Ініціалізація. Вони справедливі перед першою ініціалізацією циклу.

Збереження. Якщо вони є істинними перед черговою ітерацією циклу, то залишаються істинними і після неї.

Завершення. Після завершення циклу інваріанти дозволяють переконатися в правильності алгоритму.

Якщо виконуються перші дві властивості, інваріанти циклу залишаються дійсними перед кожною черговою ітерацією циклу. Зверніть увагу на схожість з математичною індукцією, коли для доведення певної властивості всіх елементів упорядкованої послідовності потрібно довести її справедливість для початкового елемента цієї послідовності, а потім обґрунтувати крок індукції. У цьому випадку першій частині доведення відповідає обґрунтування того, що інваріант циклу виконується перед першою ітерацією, а другій частині – доведення того, що інваріант циклу виконується після чергової ітерації (крок індукції).

Для наших цілей третя властивість найважливіша, тому що нам потрібно за допомогою інваріант циклу продемонструвати коректність алгоритму. Вона також відрізняється розглянутим нами методом від звичайної математичної індукції, в якій крок індукції використовується в нескінченних послідовностях. У цьому випадку у разі закінчення циклу «індукція» завершується.

Розглянемо, чи виконуються ці властивості для сортування методом вставок.

Ініціалізація. Почнемо з того, що покажемо справедливість інваріанта циклу перед першою ітерацією, тобто при j = 2. Таким чином, підмножина елементів A [1..j - 1] складається тільки з одного елемента A [1], що зберігає початкове значення. Більше того, у цій підмножині елементи відсортовані (тривіальне твердження). Усе сказане вище підтверджує, що інваріант циклу витримується перед першою ітерацією циклу.

Збереження. Далі, обґрунтуємо другу властивість: покажемо, що інваріант циклу зберігається після кожної ітерації. Висловлюючись неформально, можна сказати, що в тілі внутрішнього циклу for відбувається зсув

елементів A [j - 1], A [j - 2], A [j - 3], на одну позицію вправо доти, доки не звільниться відповідне місце для елемента A [j] (рядки 4–7), куди він і поміщається (рядок 8). За більш формального підходу до розгляду другої властивості потрібно було б сформулювати та обґрунтувати інваріант для внутрішнього циклу while. Однак на цьому етапі краще не вдаватися в такі формальні подробиці, тому задовольнятимемося неформальним аналізом, щоб показати, що у зовнішньому циклі дотримується друга властивість.

Завершення. Зрештою, подивимося, що відбувається після завершення роботи циклу. При сортуванні методом вставки зовнішній цикл for завершується, коли j перевищує n, тобто. коли j = n + 1. Підставивши у формулювання інваріанта циклу значення n + 1, отримаємо таке твердження: у підмножині елементів A [1..n] знаходяться ті ж елементи, що були в ньому до початку роботи алгоритму, але розташовані у відсортованому порядку. Однак підмножина A [1..n] і є сам масив A! Таким чином, весь масив відсортовано, що і підтверджує коректність алгоритму.

1.4 Контрольні питання

Дайте означення алгоритму.
Що таке дискретність роботи алгоритму?
Що таке детермінованість алгоритму?
Що таке зрозумілість алгоритму ?
Що таке точність алгоритму?
Що таке результативність алгоритму?
Що таке масовість алгоритму?
Які форми подання алгоритму Ви знаєте?

2 АНАЛІЗ АЛГОРИТМІВ

Аналіз алгоритму полягає в тому, щоб передбачити потрібні для його виконання ресурси. Іноді оцінюється потреба в таких ресурсах, як пам'ять, пропускна спроможність мережі або потрібне апаратне забезпечення, але найчастіше визначається час обчислення. Шляхом аналізу деяких алгоритмів, призначених для розв'язання однієї та тієї самої задачі, можна легко обрати найбільш ефективний з них. В процесі такого аналізу може також виявитись, що декілька алгоритмів приблизно рівноцінні, а всі інші варто відкинути.

З урахуванням того, що алгоритми реалізуються у вигляді комп'ютерних програм, в більшості випадків як технологія аналізу алгоритмів буде використовуватись модель узагальненої однопроцесорної машини з довільним доступом до пам'яті (RAM – Random Access Memory). В цій моделі команди процесора виконуються послідовно; операції, які виконуються одночасно, відсутні. У цю модель входять ті команди, які зазвичай можна знайти в реальних комп'ютерах: арифметичні (додавання, віднімання, добуток, ділення, обчислення остачі від ділення, наближення дійсного числа найближчим більшим чи найближчим меншим цілим числом), операції переміщення даних (завантаження значення в пам'ять, копіювання) та операції управління (умовне та безумовне розгалуження, виклик підпрограми і повернення з неї). Для виконання кожної такої інструкції потрібний певний фіксований проміжок часу.

В моделі RAM є цілочисловий тип даних та тип чисел з плаваючою комою. Також передбачається, що є верхня межа розміру одного слова даних. У моделі RAM, яка розглядається, не моделюється ієрархія пристроїв пам'яті, які сьогодні розповсюджені у звичайних комп'ютерах. Таким чином, кеш та віртуальна пам'ять відсутні у RAM. Моделі, які мають таку ієрархію, набагато складніші моделей RAM, тому вони можуть ускладнити роботу. Окрім цього, аналіз, який заснований на моделі RAM, зазвичай чудово прогнозує продуктивність алгоритмів, які виконуються на реальних машинах [8].

2.1 Аналіз алгоритму, що працює за методом вставок

Час роботи процедури INSERTION_SORT залежить від набору вхідних значень: для сортування тисячі чисел потрібно більше часу, ніж для сортування трьох чисел. Крім того, час сортування за допомогою цієї процедури може бути різним для послідовностей, що складаються з однієї і тієї ж кількості елементів, залежно від ступеня впорядкованості цих послідовностей до початку сортування. Загалом, час роботи алгоритму збільшується зі збільшенням кількості вхідних даних, тому загальноприйнята практика – подавати час роботи програми як функцію, яка залежить від

кількості вхідних елементів. Для цього поняття «час роботи алгоритму» та «розмір вхідних даних» потрібно визначити точніше.

Найбільш адекватне поняття розміру вхідних даних (input size) залежить від розглянутої задачі. У багатьох задачах, таких як сортування або дискретні перетворення Фур'є, – це кількість вхідних елементів, наприклад, розмір n масиву, що сортується.

Час роботи алгоритму для тих чи інших даних вимірюється у кількості елементарних операцій, або «кроків», які потрібно виконати. Тут зручно ввести поняття кроку, щоб міркування були якомога більш машинно незалежними. На даному етапі ми виходитимемо з того, що, для виконання кожного рядка псеводкоду потрібен фіксований час. Час виконання різних рядків може відрізнятися, але ми припустимо, що він є однаковим для будь-якого рядка.

Почнемо з того, що введемо для процедури INSERTION_SORT час виконання кожної інструкції та кількість їх повторень. Для кожного j = 2, 3, ... , n, де n = length [A], позначимо через t_j кількість перевірок умови у циклі **while** (рядок 5). При нормальному завершенні циклів **for** або **while** (тобто коли перестає виконуватися умова, задана в заголовку циклу) умова перевіряється на один раз більше, ніж виконується тіло циклу.

INSERTION_SORT(A)	кількість повторів
1 for j ← 2 to length[A]	n
2 do key ← A[j]	n-1
3 //Вставте елемент A[j] в A[1..j − 1]	
4 i ← j − 1	n-1
5 while i > 0 та A[i] > key	$\sum_{j-2}^{n} t_j$
6 do A[i + 1] ← A[i]	$\sum_{j-2}^{n}(t_j - 1)$
7 i ← i − 1	$\sum_{j-2}^{n}(t_j - 1)$
8 A[i + 1] ← key	n-1

Час роботи алгоритму – це сума проміжків часу, потрібних для виконання кожної інструкції, що входить до його складу. Якщо виконання інструкції триває протягом часу c і вона повторюється в алгоритмі n разів, її внесок у повний час роботи алгоритму дорівнює cn.

Щоб обчислити час роботи алгоритму INSERTION_SORT (позначимо його через T(n)), потрібно підсумувати значення, що стоять у стовпці «кількість повторів», внаслідок чого отримаємо

$$T(n) = c(n + (n - 1) + (n - 1) + \sum_{j-2}^{n} t_j + \sum_{j-2}^{n}(t_j - 1) + \sum_{j-2}^{n}(t_j - 1) + (n - 1)).$$

Навіть якщо розмір вхідних даних є фіксованою величиною, час роботи алгоритму може залежати від ступеня впорядкованості сортованих величин, який вони мали до введення. Наприклад, найсприятливіший випадок для алгоритму INSERTION_SORT – коли всі елементи масиву вже відсортовані. Тоді для кожного j = 2, 3, . . . , n має місце, що A [i] ≤ key в рядку

5, коли і дорівнює своєму початковому значенню j - 1. Таким чином, у разі j = 2, 3, . . . , n кількість перевірок умови $t_j = 1$, і час роботи алгоритму у найсприятливішому випадку обчислюється так:

$$T(n) = c(n + (n - 1) + (n - 1) + (n - 1) + (n - 1)) = c(5n-4).$$

Цей час роботи можна записати як an + b, де a та b — константи, що залежать від величини c; тобто. цей час є **лінійною функцією** від n.

Якщо елементи масиву відсортовані в порядку, оберненому необхідному (в цьому випадку в порядку зменшення), то це найгірший випадок. Кожен елемент A [j] потрібно порівнювати з усіма елементами вже відсортованої підмножини A [1..j - 1], так що для j = 2, 3, . . . n значення $t_j = j$. З урахуванням того, що

$$\sum_{j-2}^{n} j = \frac{n(n+1)}{2} - 1$$

та

$$\sum_{j-2}^{n}(j-1) = \frac{n(n-1)}{2},$$

отримуємо, що час роботи алгоритму INSERTION_SORT у найгіршому випадку визначається співвідношенням

$$T(n) = c(n + (n - 1) + (n - 1) + (\frac{n(n+1)}{2} - 1) + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + (n - 1)) =$$
$$= c(4n-3 + \frac{n(n+1)}{2} - 1 + 2\frac{n(n-1)}{2}).$$

Цей час роботи можна записати як $an^2 + bn + c$, де константи a, b і c' залежать від c. Отже, це **квадратична функція** від n.

2.2 Найгірший і середній час роботи

Аналізуючи алгоритм, який працює за методом вставок, ми розглядали як найкращий, так і найгірший випадки, коли елементи масиву були відсортовані у порядку, оберненому потрібному. Далі ми приділятимемо основну увагу визначенню лише часу роботи у найгіршому разі, тобто, максимальному часу роботи з усіх вхідних даних розміру n. Тому є три причини [9].

• Час роботи алгоритму у найгіршому випадку – це верхня межа цієї величини для будь-яких вхідних даних. Маючи це значення, ми точно знаємо, що для виконання алгоритму не потрібно більше часу. Не треба буде робити якихось складних припущень про час роботи та сподіватися, що насправді ця величина не буде перевищена.

• У деяких алгоритмах найгірший випадок трапляється досить часто. Наприклад, якщо у базі даних відбувається пошук інформації, то найгір-

шому випадку відповідає ситуація, коли потрібна інформація у базі даних відсутня. У деяких пошукових програмах випадок відсутності інформації може відбуватися досить часто.

• Характер поведінки «усередненого» часу роботи часто нічим не кращий за поведінку часу роботи для найгіршого випадку. Наприклад, якщо оцінити середній час роботи алгоритму як середнє арифметичне між найкращим і найгіршим випадком, то характер залежності буде квадратичним, як і в найгіршому випадку.

2.3 Порядок зростання

Для полегшення аналізу процедури INSERTION_SORT було зроблено деякі спрощувальні припущення. По-перше, ми проігнорували фактичний час виконання кожної інструкції, подавши цю величину як деяку константу c, однакову для всіх інструкцій. Далі ми побачили, що врахування цих констант дає зайву інформацію: час роботи алгоритму у найгіршому разі виражається формулою $an^2 + bn + c$. Отже, ми ігноруємо як фактичні вартості команд, так і їх абстрактні вартості c [10].

Тепер введемо ще одне абстрактне поняття, яке спрощує аналіз. Це швидкість зростання (rate of growth), або порядок зростання (order of growth), часу роботи, який і цікавить нас насправді. Отже, до уваги буде братися лише головний член формули (тобто, у нашому випадку an^2), оскільки за великих n членами меншого порядку можна знехтувати. Крім того, постійні множники при головному члені також ігноруватимуться, оскільки для оцінювання обчислювальної ефективності алгоритму з вхідними даними великого обсягу вони менш важливі, ніж порядок зростання. Отже, час роботи алгоритму, який працює за методом вставок, у найгіршому випадку дорівнює $\Theta(n^2)$ (вимовляється «тета від n у квадраті»).

Зазвичай один алгоритм вважається ефективнішим за інший, якщо час його роботи у найгіршому разі має нижчий порядок зростання. Через наявність постійних множників і другорядних членів ця оцінка може бути помилковою, якщо вхідні дані невеликі. Однак якщо обсяг вхідних даних значний, то, наприклад, алгоритм $\Theta(n^2)$ у найгіршому випадку працює швидше, ніж алгоритм $\Theta(n^3)$. Різні позначення, що використовуються для оцінювання складності алгоритмів, детально буде розглянуто у розділі 8.

2.4 Контрольні питання

Які припущення передбачає використання моделі RAM?
Охарактеризуйте поняття «складність алгоритму».
Охарактеризуйте поняття «ефективність алгоритму».
Що таке інваріанти циклу та для чого вони використовуються?
Як оцінюються найгірший та середній час роботи алгоритму?
Чому найчастіше оцінюється найгірший час роботи алгоритму?

3 ОСНОВНІ СТРУКТУРИ ДАНИХ

3.1 Стеки

Стеком називається множина деякої змінної кількості даних, над якою виконуються такі операції:

- поповнення стека новими даними;
- перевірка, яка визначає чи стек порожній;
- перегляд останніх доданих даних;
- знищення останніх доданих даних [11].

На основі такого функціонального опису можна сформувати логічний опис. Стек – це такий послідовний список із змінною довжиною, внесення та вилучення елементів з якого виконуються тільки з одного боку списку. Застосовуються й інші назви стека – магазин, пам'ять що функціонує за принципом LIFO (Last – In – First – Out – «останнім прийшов – першим вийшов»).

«Найверхніший» елемент стека, тобто останній доданий і ще не знищений, відіграє особливу роль: саме його можна модифікувати й знищувати. Цей елемент називають *вершиною стека*. Іншу частину стека називають *тілом стека*. Тіло стека, безумовно, є стеком: якщо вилучити зі стека його вершину, тіло перетворюється в стек.

Основні операції над стеком – внесення нового елемента (push – заштовхувати) і вилучення елемента зі стека (pop – вискакувати).

Корисними можуть бути також допоміжні операції:

- визначення поточної кількості елементів в стекові;
- очищення стека;
- «неруйнуюче» читання елемента з вершини стека, яке може бути реалізоване як комбінація основних операцій – вилучити елемент зі стека та внести його знову в стек.

При поданні стека в статичній пам'яті для нього виділяється пам'ять, як для вектора. В описі цього вектора окрім звичайних для вектора параметрів має знаходитися також покажчик стека – адреса вершини стека. Обмеження даного подання полягає в тому, що розмір стека обмежений розмірами вектора.

Покажчик стека може вказувати або на перший вільний елемент стека, або на останній записаний в стек елемент. Однаково, який з цих двох варіантів вибрати, важливо надалі строго дотримуватися його при обробці стека.

У процесі внесення елемента в стек він записується на місце, яке визначається покажчиком стека, потім покажчик модифікується так, щоб він вказував на наступний вільний елемент (якщо покажчик вказує на останній записаний елемент, то спочатку модифікується покажчик, а потім проводиться запис елемента). Модифікація покажчика полягає в доданні

до нього або у відніманні від нього одиниці (стек росте у бік збільшення адреси).

Операція вилучення елемента полягає в модифікації покажчика стека (в напрямку, зворотному модифікації при внесенні) і вибиранні значення, на яке вказує покажчик стека. Після вибирання комірка, в якій розміщувався вибраний елемент, вважається вільною.

Операція очищення стека зводиться до запису в покажчик стека початкового значення – адреси початку виділеної ділянки пам'яті.

Визначення розміру стека зводиться до обчислення різниці покажчиків: покажчика стека й адреси початку ділянки.

При зв'язному поданні стека кожен його елемент складається зі значення і покажчика, який вказує на попередньо занесений у стек елемент. Зв'язне подання викликає втрату пам'яті, що зумовлено наявністю покажчика в кожному елементі стека, і є цікавим тільки у випадку, коли важко визначити максимальний розмір стека.

Отже, для зв'язного подання стека потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Щоб виконати операції над стеком, потрібен один покажчик на вершину стека. Створення порожнього стека полягатиме у присвоєнні покажчика на вершину нульового значення, що означатиме, що стек порожній.

Послідовність кроків для додавання елемента в стек у випадку зв'язного подання складається з декількох кроків:

1. Виділити пам'ять під новий елемент стека;
2. Занесення значення в інформаційне поле;
3. Встановлення зв'язку між ним і «старою» вершиною стека;
4. Переміщення вершини стека на новий елемент;

Вилучення елемента зі стека також проводять за кілька кроків:

1. Зчитування інформації з інформаційного поля вершини стека;
2. Встановлення на вершину стека допоміжного покажчика;
3. Переміщення покажчика вершини стека на наступний елемент;
4. Звільнення пам'яті, яку займає «стара» вершина стека.

3.2 Черги

Чергою називається множина змінної кількості даних, над якою можна виконувати такі операції:

- поповнення черги новими даними;
- перевірка, яка визначає чи порожня черга;
- перегляд перших доданих даних;
- знищення перших доданих даних.

На основі такого функціонального опису можна сформувати логічний опис. Чергою FIFO (First – In – First – Out – «першим прийшов – першим вийшов») називається такий послідовний список зі змінною довжиною, в

якому внесення елементів виконується тільки з одного боку списку (хвіст черги), а вилучення – з іншого боку (голова черги).

Основні операції над чергою – ті ж, що і над стеком – внесення, вилучення, визначення розміру, очищення, «неруйнуюче» читання.

У разі подання черги вектором в статичній пам’яті на додаток до звичайних для опису вектора параметрів в ньому мають знаходитися два покажчики: на голову і на хвіст черги. У процесі внесення елемента в чергу він записується за адресою, яка визначається покажчиком на хвіст, після чого цей покажчик збільшується на одиницю. У процесі вилучення елемента з черги вибирається елемент, що адресується покажчиком на голову, після чого цей покажчик зменшується на одиницю.

Вочевидь, з часом покажчик на хвіст у процесі чергового внесення елемента досягне верхньої межі тієї ділянки пам’яті, яку виділено для черги. Проте, якщо операції внесення чергувати з операціями вилучення елементів, то в початковій частині відведеної під чергу пам’яті є вільне місце. Для того, щоб місця, які займалися вилученими елементами, могли бути повторно використані, черга замикається в кільце: покажчики (на початок і на кінець), досягнувши кінця виділеної ділянки пам’яті, перемикаються на її початок. Така організація черги в пам’яті називається кільцевою чергою. Можливі, звичайно, й інші варіанти організації: наприклад, кожний раз, коли покажчик кінця досягне верхньої межі пам’яті, зсовувати всі непорожні елементи черги до початку ділянки пам’яті, але, як цей, так і інші варіанти вимагають переміщення в пам’яті елементів черги і менш ефективні, ніж кільцева черга.

У початковому стані покажчики на голову і хвіст вказують на початок ділянки пам’яті. Рівність цих двох покажчиків є ознакою порожньої черги. Якщо в процесі роботи з кільцевою чергою кількість операцій внесення перевищує кількість операцій вилучення, може виникнути ситуація, в якій покажчик кінця «наздожене» покажчик початку. Це ситуація заповненої черги, але якщо в цій ситуації покажчики порівняються, ця ситуація буде така ж, як при порожній черзі. Для розрізнення цих двох ситуацій до кільцевої черги висувається вимога, щоб між покажчиком кінця і покажчиком початку залишався «проміжок» з вільних елементів. Коли цей «проміжок» скорочується до одного елемента, черга вважається заповненою і подальші спроби запису в неї блокуються. Очищення черги зводиться до запису одного і того ж (не обов’язково початкового) значення в обидва покажчики. Визначення розміру полягає в обчисленні різниці покажчиків з урахуванням кільцевої природи черги.

У разі зв’язного подання черги кожен елемент черги складається зі значення і покажчика, який вказує на попередньо занесений у чергу елемент.

Зв’язне подання викликає втрату пам’яті, що зумовлено наявністю покажчика в кожному елементі черги, і є цікавим тільки у випадку, коли важко визначити максимальний розмір черги. Для зв’язного подання черги

потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Щоб виконати операції над чергою, потрібні два покажчики: на голову і хвіст черги. Створення порожньої черги полягатиме у присвоєнні покажчикам на голову і хвіст черги нульових значень, що означатиме, що черга порожня.

*Послідовність кроків для додавання елемен*та в кінець черги складається з декількох кроків:

1. Виділення пам’яті під новий елемент черги;
2. Занесення значення в інформаційне поле;
3. Занесення нульового значення в покажчик;
4. Встановлення зв’язку між ним і останнім елементом черги, і новим, враховуючи випадок пустої черги;
5. Переміщення покажчика кінця черги на новий елемент.

Вилучення елемента з черги також проводять за кілька кроків:

1. Зчитування інформації з інформаційного поля голови черги;
2. Встановлення на голову черги допоміжного покажчика;
3. Переміщення покажчика початку черги на наступний елемент;
4. Звільнення пам’яті, яку займав перший елемент черги.

В реальних задачах іноді виникає потреба у формуванні черг, відмінних від наведених структур. Порядок вибирання елементів з таких черг визначається пріоритетами елементів. Пріоритет в загальному випадку може бути поданий числовим значенням, яке обчислюється або на підставі значень деяких полів елемента, або на підставі зовнішніх чинників. Так, попередньо наведені структури стек і черги можна трактувати як пріоритетні черги, в яких пріоритет елемента залежить від часу його внесення в структуру. У процесі вибирання елемента кожен раз вибирається елемент з щонайбільшим пріоритетом.

Черги з пріоритетами можуть бути реалізовані на лінійних структурах – в суміжному або зв’язному поданні. Можливі черги з пріоритетним внесенням, в яких послідовність елементів черги весь час підтримується впорядкованою, тобто, кожний новий елемент вноситься на те місце в послідовності, яке визначається його пріоритетом, а в процесі вилучення завжди вибирається елемент з голови. Можливі і черги з пріоритетним вилученням – новий елемент вноситься завжди в кінець черги, а в процесі вилучення в черзі шукається (цей пошук може бути тільки лінійним) елемент з максимальним пріоритетом і після вибирання вилучається з послідовності. І в тому, і в іншому варіанті потрібний пошук, а якщо черга розміщується в статичній пам’яті, то ще й переміщення елементів.

3.3 Деки

Дек – особливий вид черги. Дек (deq – double ended queue, тобто черга з двома кінцями) – це такий послідовний список, в якому як внесення, так і

вилучення елементів може здійснюватися з будь-якого з двох кінців списку. Так само можна сформулювати поняття дека, як стек, в якому внесення та вилучення елементів може здійснюватися з обох кінців.

Деки рідко зустрічаються у своєму первісному означенні. Окремий випадок дека – дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури дека аналогічні логічній і фізичній структурі кільцевої черги. Однак, стосовно дека, доцільно говорити не про голову і хвіст, а про лівий і правий кінці.

Над деком доступні такі операції:

- внесення елемента праворуч;
- внесення елемента ліворуч;
- вилучення елемента праворуч;
- вилучення елемента ліворуч;
- визначення розміру;
- очищення.

Фізична структура дека в статичній пам’яті ідентична структурі кільцевої черги.

3.4 Лінійні списки

Лінійні списки є узагальненням попередніх структур; вони дозволяють подати множину так, щоб кожний елемент був доступний і, щоб «доступитися» до нього, не потрібно було б зачіпати деякі інші [12].

Списки є досить гнучкою структурою даних, оскільки їх легко зробити більшими або меншими, а їх елементи доступні для вставляння або вилучення в будь-якій позиції списку. Списки також можна об’єднувати або розділяти на менші списки.

Лінійний список – це скінченна послідовність однотипних елементів (вузлів), можливо, з повторенням. Кількість елементів у послідовності називається *довжиною списку*. Вона в процесі роботи програми може змінюватися. Лінійний список L, що складається з елементів $d_1, d_2. \ldots, d_n$, які мають однаковий тип, записують у вигляді $L = < d_1, d_2. \ldots, d_n >$ або зображають графічно (рис. 3.1).

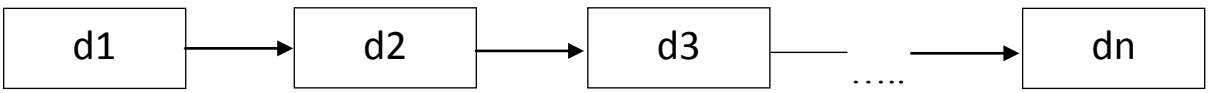

Рисунок 3.1 – Графічне подання лінійного списку

Важливою властивістю лінійного списку є те, що його елементи можна лінійно впорядкувати відповідно до їх позиції в списку.

Для формування абстрактного типу даних на основі математичного визначення списку потрібно задати множину операторів, які виконуються над об’єктами типу список. Проте не існує однієї множини операторів, що

виконуються над списками, які задовольняють відразу всі можливі застосунки.

Найчастіше зі списками доводиться виконувати такі операції:

- знайти елемент із заданою властивістю;
- визначити і-й елемент у лінійному списку;
- внести додатковий елемент до або після вказаного вузла;
- вилучити певний елемент списку;
- впорядкувати вузли лінійного списку в певному порядку.

У реальних мовах програмування не існує якої-небудь структури даних для зображення лінійного списку так, щоб всі операції над ним виконувалися однаково ефективно. Тому. працюючи з лінійними списками, важливе значення має подання лінійних списків, які використовуються в програмі, так, щоб була забезпечена максимальна ефективність і за часом виконання програми, і за обсягом потрібної їй пам'яті.

Лінійний список є послідовність об'єктів. Позиція елемента в списку має інший тип даних, відмінний від типу даних елемента списку, і цей тип залежить від конкретної фізичної реалізації. Над лінійним списком допустимі такі операції.

Операція вставляння – вставляє елемент в конкретну позицію в списку, переміщуючи елементи від цієї позиції і далі в наступну, більш вищу позицію.

Операція локалізації – повертає позицію об'єкта в списку. Якщо в списку об'єкт зустрічається декілька разів, то повертається позиція першого від початку списку об'єкта. Якщо об'єкта немає в списку, то повертається значення, яке дорівнює довжині списку, збільшене на одиницю.

Операція вибирання елемента зі списку – повертає елемент, який знаходиться в конкретній позиції списку. Результат не визначений, якщо в списку немає такої позиції.

Операція вилучення – вилучає елемент в конкретній позиції зі списку. Результат не визначений, якщо в списку немає вказаної позиції.

Операції вибирання попереднього і наступного елементів – повертають, відповідно, наступний і попередній елементи списку відносно конкретної позиці в списку.

Функція очищення списку робить список порожнім. Основні методи зберігання лінійних списків поділяються на методи послідовного і зв'язного зберігання. При виборі способу зберігання в конкретній програмі потрібно враховувати, які операції і з якою частотою будуть виконуватися над лінійними списками, вартість їх виконання та обсяг потрібної пам'яті для зберігання списку.

Найпростіша форма подання лінійного списку — це вектор. Визначивши таким чином список можна почергово звертатися до них в циклі і виконувати потрібні дії. Однак у разі такого подання лінійного списку не вдасться уникнути фізичного переміщення елементів, якщо потрібно додавати нові елементи, або вилучати існуючі. Набагато швидше елементи

можна вилучати за допомогою простої схеми очищення пам'яті. Замість вилучення елементів із списку, їх помічають як невикористані.

Більш складною організацією роботи зі списками є розміщення в масиві декількох списків або розміщення списку без прив'язки його початку до першого елемента масиву.

У разі зв'язного подання лінійного списку кожен його елемент складається зі значення і покажчика, який вказує на наступний елемент у списку.

На рисунку 3.2 наведено структуру однозв'язного списку. Кожний список має мати особливий елемент, який називається покажчиком на початок списку, або головою списку.

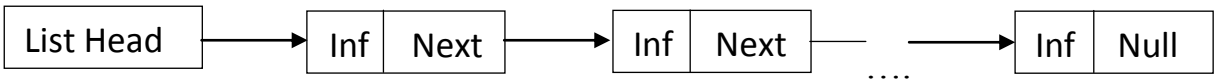

Рисунок 3.2 – Структура однозв'язного списку

Однак обробка однозв'язного списку не завжди зручна, оскільки відсутня можливість просування в протилежний бік. Таку можливість забезпечує двозв'язний список, кожний елемент якого містить два покажчики: на наступний і попередній елементи списку (рис. 3.3).

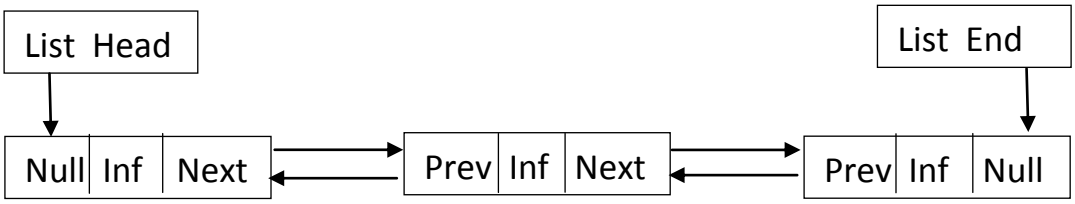

Рисунок 3.3 – Структура двозв'язного списку

Для зручності обробки списку додають ще один особливий елемент – покажчик кінця списку. Наявність двох покажчиків в кожному елементі ускладнює список і призводить до додаткових витрат пам'яті, але, разом з тим, забезпечує більш ефективне виконання деяких операцій над списком.

Різновидом розглянутих видів лінійних списків є кільцевий список, який може бути організований на основі як однозв'язного, так і двозв'язного списків. При цьому в однозв'язному списку покажчик останнього елемента має вказувати на перший елемент; в двозв'язному списку в першому і останньому елементах відповідні покажчики змінюються.

Підсумовуючи, відзначимо *основні переваги зв'язних списків*. А саме:

• легкість додавання та видалення елементів;

• розмір обмежений лише обсягом пам'яті комп'ютера та розрядністю вказівників;

• динамічне додавання та видалення елементів.

До *недоліків цієї технології* належить:

• складність визначення адреси елемента за його індексом (номером) у списку;

• на поля-вказівники (вказівники на наступний та попередній елемент) витрачається додаткова пам’ять (у масивах, наприклад, покажчики не потрібні);

• робота зі списком повільніша, ніж з масивами, оскільки до будь-якого елемента списку можна звернутися, тільки пройшовши всі попередні елементи;

• елементи списку можуть бути розташовані в пам’яті розріджено, що негативно вплине на кешування процесора;

• над зв’язними списками набагато важче (хоча, в принципі, можливо) проводити такі паралельні векторні операції, як обчислення суми;

• кеш-промахи під час обходу списку.

3.5 Контрольні питання

Поясніть, у чому полягають особливості стека.
Як можна реалізувати стек програмно?
Назвіть основні операції, які можна виконувати зі стеком.
Дайте означення черги.
Які різновиди черг Ви знаєте?
З яких кроків складається додавання/вилучення елемента в чергу?
З яких кроків складається вилучення елемента з черги?
Що таке деки?
Дайте означення лінійного списку.
Які операції можна виконувати зі списками?
Які переваги та недоліки списків порівняно з масивами?

4 МЕТОД ДЕКОМПОЗИЦІЇ. РЕКУРЕНТНІ СПІВВІДНОШЕННЯ

4.1 Приклад алгоритму, що реалізує метод декомпозиції

Багато корисних алгоритмів мають рекурсивну структуру: для вирішення даного завдання вони рекурсивно викликають самі себе один або кілька разів, щоб вирішити допоміжне завдання, що безпосередньо стосується поставленого завдання. Такі алгоритми часто розробляються за допомогою **методу декомпозиції**, або **розбиття**: складне завдання розбивається на декілька більш простих, які подібні до вихідної задачі, але мають менший обсяг; далі ці допоміжні завдання вирішуються рекурсивним методом, після чого отримані рішення комбінуються з метою одержання вирішення вихідного завдання [13].

Ця рекурсивна парадигма отримала назву «розділяй та володарюй» (divide and conquer). Сама парадигма «розділяй та володарюй» складається з трьох етапів.

Розділення задачі на декілька підзадач.

Підкорення – рекурсивне розв’язання цих підзадач. Коли обсяг підзадач достатньо малий, вони розв’язуються безпосередньо.

Комбінування розв’язку початкової задачі з розв’язків допоміжних підзадач.

Розгляд цієї парадигми ми почнемо з алгоритму сортування методом злиття (merge sort). В рамках підходу «розділяй та володарюй» цей метод можна описати так.

Розділення: послідовність для сортування, яка складається з n елементів, розбивається на дві менші послідовності, кожна з яких містить n/2 елементів.

Рекурсивне розв’язання: сортування обох створених послідовностей методом злиття у випадку, якщо їх розмір перевищує одиницю.

Комбінування: злиття двох відсортованих послідовностей для отримання кінцевого результату.

Рекурсія досягає своєї нижньої межі, коли довжина послідовності для сортування дорівнює 1. В цьому випадку всю роботу вже зроблено, оскільки будь-яку таку послідовність можна вважати впорядкованою.

Основна операція, яка виконується в процесі сортування за методом злиття, – це об’єднання двох відсортованих послідовностей у процесі комбінування (останній етап). Це робиться за допомогою додаткової процедури Merge(A, p, q, r), де A – це масив, а p, q та r – індекси, які нумерують елементи масиву, такі, що $p<q<r$. В цій процедурі припускається, що елементи підмасивів A[p..q] та A[q+1..r] впорядковані. Вона зливає ці два підмасиви в один відсортований, елементи якого замінюють поточні елементи підмасиву A[p..r]. Як і в попередньому прикладі, вважаємо, що сортуються карти.

З обчислювальної точки зору виконання кожного основного кроку займає однакові проміжки часу, тоді як все зводиться до порівняння двох верхніх карт. Оскільки потрібно виконати принаймні n основних кроків, час роботи процедури Merge дорівнює Θ(n). Приклад виконання процедури наведено на рис. 4.1, а) – і).

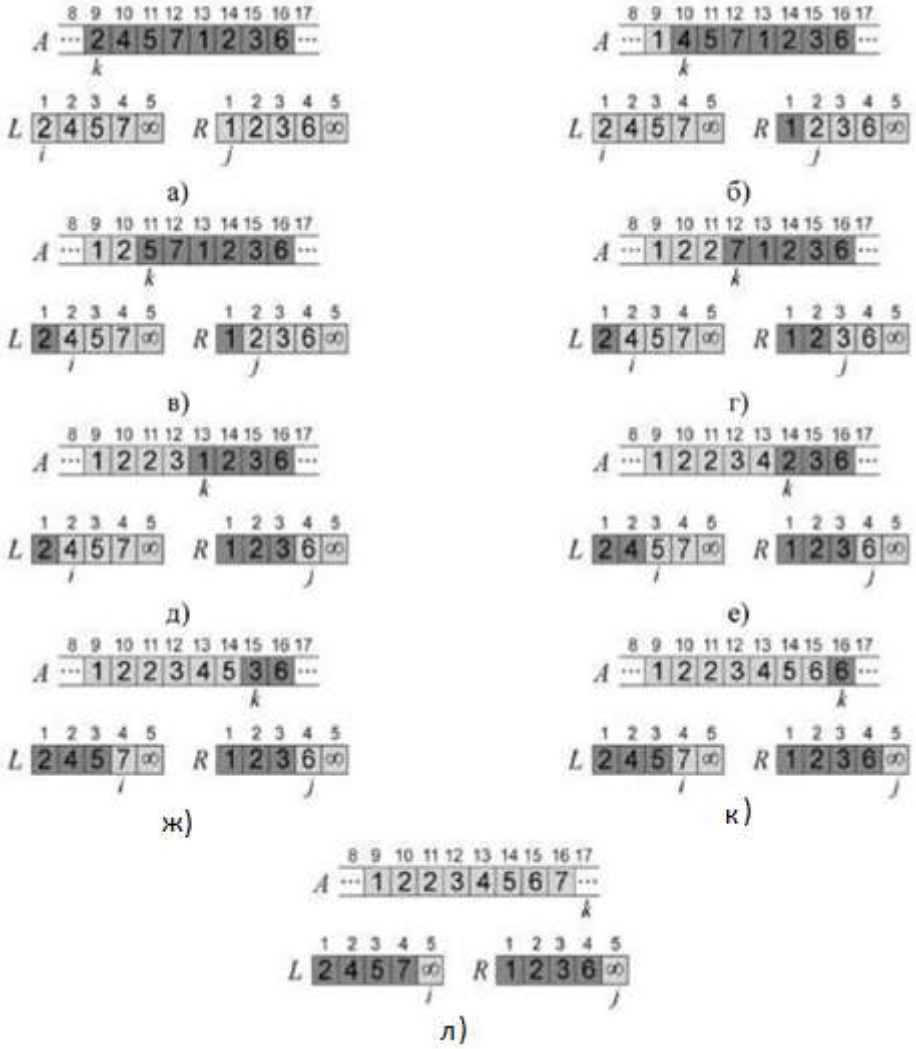

Рисунок 4.1 – Приклад виконання процедури сортування

Описану ідею реалізовано у нижченаведеному псевдокоді, проте в ньому також є додаткові програмістські хитрощі, завдяки яким в ході кожного основного кроку не доводиться перевіряти, чи кожний з двох стосів порожній. Ідея полягає в тому, щоб помістити в низ обох колод, що об’єднуються, так звану сигнальну карту особливого значення, що дозволяє спростити код. Для позначення сигнальної картки використовується символ ∞. У колоді немає карт, вищих за сигнальну карту. Процес триває доти, доки карти, що перевіряються, в обох стосах не виявляться сигнальними. Як тільки це станеться, це означатиме, що всі несигнальні карти вже розміщено у вихідній стос. Оскільки заздалегідь відомо, що у вихідному стосі має бути рівно r - p + 1 карта, виконавши таку кількість основних кроків, можна зупинитися/

```
MERGE(A, p, q, r)
1 n1 ← q − p + 1
2 n2 ← r − q
3 // Створюємо масиви L[1..n1 + 1] и R[1..n2 + 1]
4 for i ← 1 to n1
5       do L[i] ← A[p + i − 1]
6 for j ← 1 to n2
7       do R[j] ← A[q + j]
8 L[n1 + 1] ← ∞
9 R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 for k ← p to r
13     do if L[i] ≤ R[j]
14           then  A[k] ← L[i]
15                 i ← i + 1
16           else  A[k] ← R[j]
17                 j ← j + 1
```

Перед кожною ітерацією циклу for у рядках 12–17 підмасив A[p..k − 1] містить k - p найменших елементів масивів L[1..n1 + 1] та R[1..n2 + 1] в відсортованому порядку. Крім того, елементи L[i] та R[i] є найменшими елементами масивів L та R, які ще не скопійовано в масив A.

Потрібно показати, що цей інваріант циклу витримується перед першою ітерацією аналізованого циклу for, що кожна ітерація циклу не порушує його і що з його допомогою вдається продемонструвати коректність алгоритму, коли цикл закінчує свою роботу.

Ініціалізація. Перед першою ітерацією циклу k = p, тому підмасив A[p..k - 1] порожній. Він містить k - p = 0 найменших елементів масивів L та R. Оскільки i = j = 1, елементи L[i] та R[j] – найменші елементи масивів L та R, не скопійовані назад у масив A.

Збереження. Щоб переконатися, що інваріант циклу зберігається після кожної ітерації, спочатку припустимо, що L[i] ≤ R [j]. Тоді L[i] – найменший елемент, не скопійований до масиву A. Оскільки в підмасиві A [p..k - 1] міститься k - p найменших елементів, після виконання рядка 14, в якому значення елемента L[i] присвоюється елементу A[k], в підмасиві A[p..k] буде міститись k - p + 1 найменший елемент. В результаті збільшення параметра k циклу for і значення змінної i (рядок 15), інваріант циклу відновлюється перед наступною ітерацією. Якщо ж виконується нерівність L[i] > R[j], то в рядках 16 та 17 виконуються відповідні дії, протягом яких також зберігається інваріант циклу.

Завершення. Алгоритм завершується, коли k = r + 1. Відповідно до інваріанта циклу, підмасив A[p..k - 1] (тобто підмасив A [p..r]) містить k - p = r - p + 1 найменших елементів масивів L[1..n1 + 1] та R[1..n2 + 1] у відсортованому порядку. Сумарна кількість елементів у масивах L та R дорівнює n1 + n2 + 2 = r - p + 3. Всіх їх, крім двох найбільших, скопійовано назад в масив A, а два елементи, що залишилися, є сигнальними.

Щоб показати, що час роботи процедури MERGE дорівнює Θ(n), де n = r - p + 1, зауважимо, що кожен із рядків 1–3 та 8–11 виконується протягом фіксованого часу; тривалість циклів for у рядках 4–7 дорівнює Θ(n1 + n2)=Θ(n), а в циклі for у рядках 12–17 виконується n ітерацій, на кожну з яких витрачається фіксований час.

Тепер процедуру MERGE можна використовувати як підпрограму в алгоритмі сортування злиттям. Процедура MERGE_SORT(A, p, r)виконує сортування елементів підмасиву A[p..r]. Якщо справедлива нерівність p ≥ r, то в цьому підмасиві міститься не більше одого елемента, і, отже, він відсортований. Інакше проводиться розбиття, під час якого обчислюється індекс q, що розділяє масив A [p..r] на два підмасиви: A [p..q] з n/2 елементами та A [q..r] з n/2 елементами.

```
MERGE_SORT(A, p, r)
1 if p < r
2 then q ← (p + r)/2
3            MERGE_SORT(A, p, q)
4            MERGE_SORT(A, q + 1, r)
5            MERGE(A, p, q, r)
```

Щоб відсортувати послідовність A = (A [1], A [2], . . . , A [n]) викликається процедура MERGE_SORT(A, 1, length [A]), де length [A] = n. На рис. 4.2 проілюстровано роботу цієї процедури у висхідному напрямку, якщо n – це степінь двійки. У ході роботи алгоритму відбувається попарне об’єднання одноелементних послідовностей у відсортовані послідовності довжини 2, потім попарне об’єднання двоелементних послідовностей у відсортовані послідовності довжини 4 і т. д., поки не буде отримано дві послідовності, що складаються з n/2 елементів, які об’єднуються в кінцеву відсортовану послідовність довжини n.

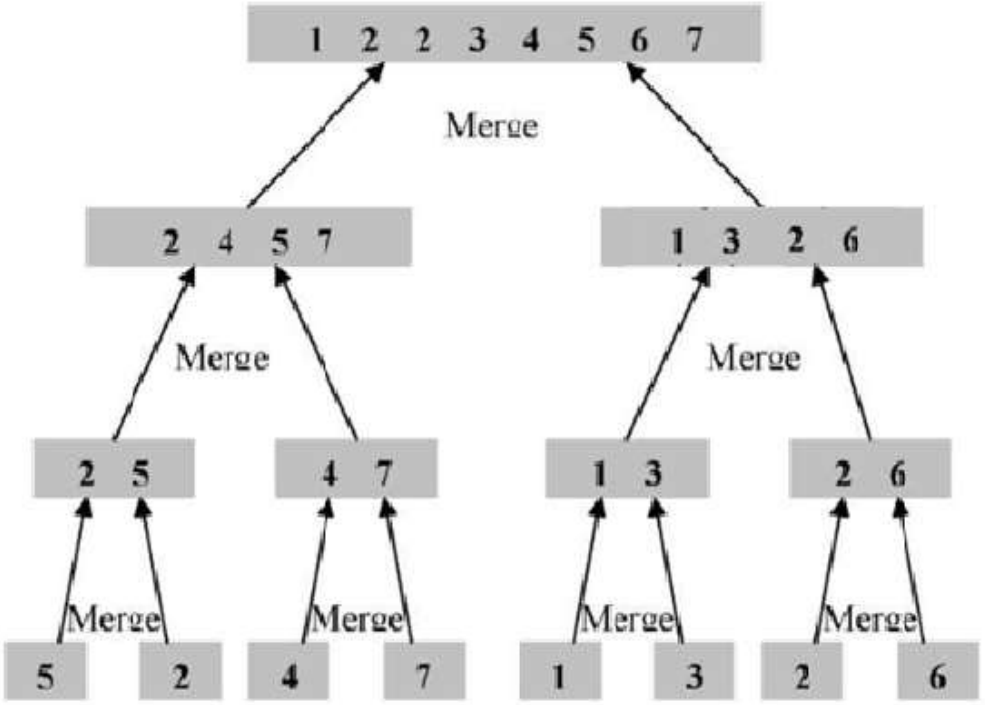

Рисунок 4.2 – Процес сортування масиву A= <5, 2, 4, 7, 1, 3, 2, 6>

4.2 Аналіз алгоритмів, що реалізують метод декомозиції

Якщо алгоритм рекурсивно звертається до себе, час його роботи часто описується за допомогою рекурентного рівняння, або рекурентного співвідношення, в якому повний час, потрібний для вирішення всього завдання з обсягом введення n, виражається через час вирішення допоміжних підзадач. Потім це рекурентне рівняння вирішується за допомогою певних математичних методів, і встановлюються межі продуктивності алгоритму.

Отримання рекурентного співвідношення, щоб визначити час роботи алгоритму, заснованого на принципі «розділяй і володарюй», базується на трьох етапах, що відповідають парадигмі цього принципу. Позначимо через T(n) час розв’язання задачі, розмір якої дорівнює n. Якщо розмір завдання досить малий, скажімо, n <= c де c – деяка заздалегідь відома константа, то завдання вирішується безпосередньо протягом певного фіксованого часу, який ми позначимо через Θ(1). Припустимо, що завдання ділиться на b підзадач, обсяг кожної зокрема дорівнює 1/b від обсягу вихідної задачи. Якщо розбиття завдання на допоміжні підзавдання відбувається протягом часу D(n), а об’єднання вирішень підзавдань у вирішення вихідної задачі – протягом часу C(n), то ми отримаємо таке рекурентне співвідношення:

T (n) = Θ (1) при n <= c,
T (n) = aT (n/b) + D (n) + C (n) в іншому випадку.

4.3 Аналіз алгоритму сортування злиттям

Щоб отримати рекурентне рівняння для верхньої оцінки часу роботи T(n) алгоритму, що виконує сортування n чисел методом злиття, будемо міркувати так. Сортування одного елемента шляхом злиття триває протягом фіксованого часу. Якщо n > 1, час роботи розподіляється в такий спосіб.

Розбиття. У ході розбиття визначається, де знаходиться середина підмасиву. Ця операція триває фіксований час, тому D(n) = Θ(1).

Підкорення. Рекурсивно вирішуються дві підзадачі, обсяг кожної з яких становить n/2. Час розв'язання цих подзадач дорівнює 2T(n/2).

Комбінування. Як уже згадувалося, процедура MERGE у n-елементному підмасиві виконується протягом часу Θ(n), тому C(n) = Θ(n).

Склавши функції D(n) і C(n), отримаємо суму величин Θ(n) та Θ(1), яка є лінійною функцією від n, тобто Θ(n). Додаючи до цієї величини доданок 2T(n/2), що відповідає етапу «підкорення», отримаємо рекурентне співвідношення для часу роботи T(n) алгоритму сортування за методом злиття у найгіршому випадку.

T(n) = Θ(1) при n = 1,
T(n) = 2T(n/2) + Θ(n) при n > 1

В [3] формально показано, що T(n) = Θ(n lg n). Щоправда, можна інтуїтивно зрозуміти, що рішенням попереднього рекурентного співвідношення є вираз T(n) = Θ(n lg n). Перепишемо його у вигляді:

T(n) = c у разі n = 1,
T(n) = 2T (n/2) + cn у разі n > 1,

де константа c позначає час, який потрібен для вирішення задачі, розмір якої дорівнює 1, а також питомий (припадає на один елемент) час, потрібний для поділу та поєднання.

Процес розв'язання рекурентного співвідношення проілюстровано на рис. 4.3, а) – г).

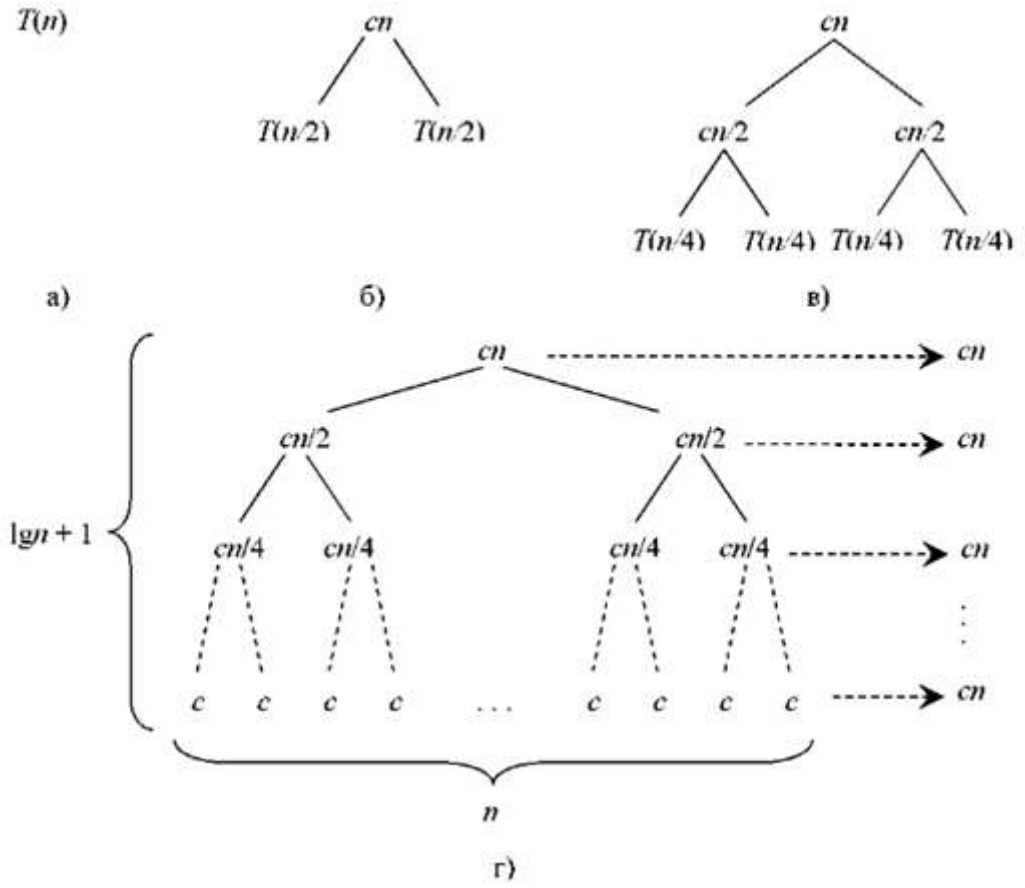

Рисунок 4.3 – побудова дерева рекурсії для рівняння T(n) = 2T(n/2) + cn

Для зручності припустимо, що n дорівнює степеню двійки. У частині а) згаданого рисунка показано час T(n) поданий у частині б) у вигляді еквівалентного дерева, яке уособлює рекурентне рівняння. Коренем цього дерева є доданок cn (вартість верхнього рівня рекурсії), а два піддерева, що беруть початок від кореня, є двома меншими рекурентними послідовностями T(n/2). У частині в) показаний черговий крок рекурсії. Час виконання кожного з двох підвузлів, що знаходяться на другому рівні рекурсії, дорівнює cn/2. Далі триває розкладання кожного вузла, що входить до складу дерева, шляхом розбиття на складові частини, визначені в рекурентній послідовності. Так відбувається доти, доки розмір задачі не стає рівним 1, а час його виконання – константі c. Дерево, що вийшло в результаті, показано в частині г). Дерево складається з lg n + 1 рівнів (тобто, його висота дорівнює lg n), а кожен рівень дає вклад в повний час роботи, рівний cn. Таким чином, повний час роботи алгоритму дорівнює cn lg n + cn, що відповідає Θ(nlg n).

Після того як дерево збудовано, тривалості виконання всіх його вузлів підсумовуються за всіма рівнями. Повний час виконання верхнього рівня дорівнює cn, наступний рівень дає внесок, що дорівнює c (n/2) + c (n/2) = cn. Ту ж величину внеску дають і всі наступні рівні.У загальному випадку рівень i (якщо вести відлік зверху) має 2^i вузлів, кожен з яких дає вклад у

загальний час роботи алгоритму, рівний $c(n/2^i)$, тому повний час виконання всіх вузлів, що належать певному рівню, дорівнює $2^i c(n/2^i) = cn$. На нижньому рівні є n вузлів, кожен із яких дає вклад c, що у сумі дає час, рівний cn.

Повна кількість рівнів дерева дорівнює $\lg n + 1$. Це легко зрозуміти з неформальних індуктивних міркувань. У найпростішому випадку, коли n=1, є лише один рівень. Оскільки $\lg 1 = 0$, вираз $\lg n + 1$ дає правильну кількість рівнів. Тепер як індуктивне припущення приймемо, що кількість рівнів рекурсивного дерева з 2^i вузлами дорівнює $\lg 2^i + 1 = i+1$ (оскільки для будь-якого і виконується співвідношення $\lg 2^i = i$). Оскільки ми припустили, що кількість вхідних елементів дорівнює степеню двійки, тепер потрібно розглянути випадок для 2^{i+1} елементів. Дерево з 2^{i+1} вузлами має на один рівень більше, ніж дерево з 2^i вузлами, тому повна кількість рівнів дорівнює $(i+1) + 1 = \lg 2^{i+1} + 1$.

Щоб знайти повний час, що є розв'язанням рекурентного співвідношення, потрібно просто скласти внески від усіх рівнів. Усього є $\lg n + 1$ рівнів, кожен з яких виконується протягом часу cn, тому повний час дорівнює $cn (\lg n + 1) = cn \lg n + cn$. Нехтуючи членами нижчих порядків та константою c отримуємо $\Theta(n \lg n)$.

4.4 Рекурентні співвідношення

Рекурентні співвідношення мають надзвичайно важливе значення для програмування. Вони використовуються в аналізі алгоритмів, наближених обчисленнях, динамічному програмуванні тощо [14].

Рекурентне співвідношення першого порядку

Нехай $\{a_k : k \geq 0\}$ – деяка послідовність дійсних чисел.

Означення 1. Послідовність $\{a_k : k \geq 0\}$ називається заданою рекурентним співвідношенням першого порядку, якщо явно задано її перший член a_0, а кожен наступний член a_k цієї послідовності визначається деякою залежністю через її попередній член a_{k-1}, тобто,

$$a_0 = u,\ a_k = f(n, p, a_{k-1}),\ k \geq 0,$$

де u – задане (початкове) числове значення,

p – деякий сталий параметр або набір параметрів,

f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій, доступних для виконання за допомогою мови програмування.

У рекурентних співвідношеннях m-го порядку k-й член послідовності визначається через m попередніх.

Рекурсія

Означення 2. Рекурсія – спосіб визначення об'єкта (або методу) попереднім описом одного чи кількох його базових випадків, з наступним описом на їхній основі загального правила побудови об'єкта.

Означення 3. Рекурсивна функція – метод визначення функції, при якому функція прямо або неявно викликає сама себе.

Як відомо, послідовність чисел Фібоначчі визначається рекурентним співвідношенням другого порядку

$F_0 = 1, F_1 = 1,$
$F_n = F_{n-1} + F_{n-2}, n \geq 2.$

Отже, якщо покласти, що функція F_n визначає n-не число послідовності Фібоначчі, то з вищенаведеного рекурентного співвідношення отримаємо, що послідовність Фібоначчі може бути визначена рекурсивною функцією:

$F(0) = 1, F(1) = 1,$
$F(n) = F(n-1) + F(n-2), n \geq 2.$

Початкові значення F(0) = 1, F(1) = 1 дуже важливі у разі визначення рекурсивної функції. Якби їх не було, то рекурсія б стала нескінченною! Тому, описуючи рекурсивну функцію, завжди треба переконуватися, що для всіх припустимих значень аргументів виклик рекурсивної функції завершиться, тобто, що рекурсія буде скінченною.

Можна звернути увагу на те, що, на відміну від обчислення елементів послідовності, заданої рекурентним співвідношенням, де обчислення відбувається від тривіального (початкового) елемента до шуканого, рекурсивна функція починає обчислення від шуканого.

Кількість вкладених викликів функції називається глибиною рекурсії. Наприклад, глибина рекурсії у разі знаходження F(5) буде 4.

Наведемо стандартний підхід до опису рекурсивної функції. Структурно рекурсивна функція на верхньому рівні завжди є розгалуженням, що складається з двох або більше альтернатив, з яких:

• принаймні одна є термінальною (припинення рекурсії);

• принаймні одна є рекурсивною (тобто здійснює виклик себе з іншими аргументами).

4.5 Контрольні питання

У чому полягає основна ідея методу декмопозиції?

Назвіть основні етапи парадигми «розділяй та володарюй».

Що являє собою послідовність $a_0 = 1; a_k = a_{k-1} \cdot k$?

Дайте означення рекурентного співвідношення першого порядку.

Наведіть приклад рекурентного співвідношення другого порядку.

Дайте означення рекурсивної функції.

5 МЕТОД ЗАМІТАЛЬНОЇ ПРЯМОЇ

5.1 Задача пошуку перетину відрізків

Проілюструємо цей метод на прикладі конкретної задачі: задано множину відрізків на площині, треба знайти всі їх перетини. Візьмемо вертикальну пряму *l,* яка розбиває дану площину на ліву та праву півплощини. Припустимо, що кожна з цих півплощин містить кінці заданих відрізків. Зрозуміло, що розв'язком задачі є об'єднання розв'язків для кожної з двох півплощин, тому, якщо припустити, що у нас вже є множина точок перетину ліворуч від *l*, то на цю множину не будуть впливати відрзки, розташовані праворуч від *l* [15].

Тепер зазначимо, що перетин може відбутися тільки між такими двома відрізками, чиї перетини з деякою вертикаллю суміжні; отже, якщо побудувати всі вертикальні перерізи відрізків заданої множини, то будуть знайдені і всі шукані перетини. Але цієї задачі (що немає розв'язку) побудови (континуально) нескінченної множини всік вертикалей, що перерізають множину відрізків, можна уникнути; якщо помітити, що площина розбивається на вертикальні смуги, обмежені або кінцями відрізків, або точками їх перетинів, а також, що вертикальний порядок точок перетину вздовж будь-якого перерізу в такій смузі не змінюється. Таким чином, все, що треба зробити – це стрибок з лівого кінця смуги на її правий кінець з оновленням порядка точок перетину вертикалі та пошуком нових перетинів серед «сусідніх» в цьому порядку відрізків.

Відзначимо *суттєві риси методу плоского замітання*. Є вертикаль, яка замітає площину зліва направо, зупиняючись в особливих точках, які назвемо «точками подій». Перетин замітальної прямої з вхідними даними задачі містить всю корисну для продовження пошуку інформацію. Отже, маємо дві основні структури:

1. Список точок подій – послідовність абсцис, впорядованих за неспаданням, які визначають позиції зупинок замітальної прямої;
2. Статус замітальної прямої – адекватний опис перетину цієї замітальної прямої з геометричною структурою, що «замітається». «Адекватний» означає, що цей перетин містить інформацію, корисну для поставленої задачі. Статус замітальної прямої оновлюється в кожній точці подій.

Перед тим, як розглядати задачу пошуку перетину відрізків на площині, розглянемо більш просту (одновимірну) задачу. Нехай задано N інтервалів на дійсній осі. Треба визначити, чи перекриваються буль-які два з них. Інтервали задано значеннями своїх лівого та правого кінців.

Відповідь можна отримати за час $O(N^2)$, якщо перебрати всі пари інтервалів, але є кращий алгоритм, який базується на сортуванні. Якщо впорядкувати 2N кінцевих точок цих інтервалів і позначити їх як лівий (+1) або

правий (-1), то ці інтервали не перекриваються тоді і тільки тоді, коли кінці в послідовності йдуть строго по черзі: Л П Л П … Л П (рис. 5.1).

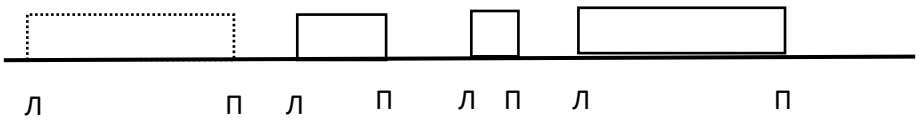

Рисунок 5.1 – Інтервали не перекриваються

Натомість, якщо два одноіменних кінці відрізків будуь йти підряд (Л Л або П П), це означатиме, що інтервали перекриваються (рис. 5.2).

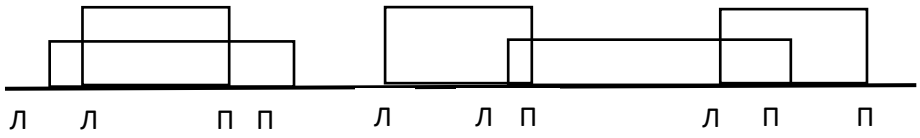

Рисунок 5.2 – Інтервали перекриваються

Цю перевірку можна провести за час $O(n \log n)$, якщо використати замітальну пряму (тобто переглянути масив (вектор) відсортованих інтервалів в якому лівий кінець помічено як +1, а правий – як -1 і підсумовуючи ці значення. Поява в сумі значення 2 означає, що інтервали перетинаються).

Задача пошуку відрізків, що перетинаються, насправді є узагальненням роглянутої задачі пошуку відрізків, що перетинаються, на двовимірний випадок.

5.2 Застосування методу замітальної прямої для перевірки перетину відрізків

Дано n відрізків на площині. Потрібно перевірити, чи перетинаються один з одним хоча б два з них. (Якщо відповідь позитивна – то вивести, що перетинаються; інакше – ні).

Наївний алгоритм рішення – перебрати за $O(n^2)$ всі пари відрізків і перевірити для кожної пари перетинаються вони чи ні. Далі наводиться алгоритм з часом роботи $O(n \log(n))$, який заснований на принципі скануальної (замітальної) прямої (англійською: «sweep line»).

Алгоритм

Проведемо подумки вертикальну пряму $x = -\infty$ і почнемо рухати цю пряму праворуч (рис. 5.3). По ходу свого руху ця пряма буде зустрічатися з відрізками, причому у будь-який момент часу кожен відрізок буде перетинатися з нашою прямою в одній точці (ми поки вважатимемо, що вертикальних відрізків немає).

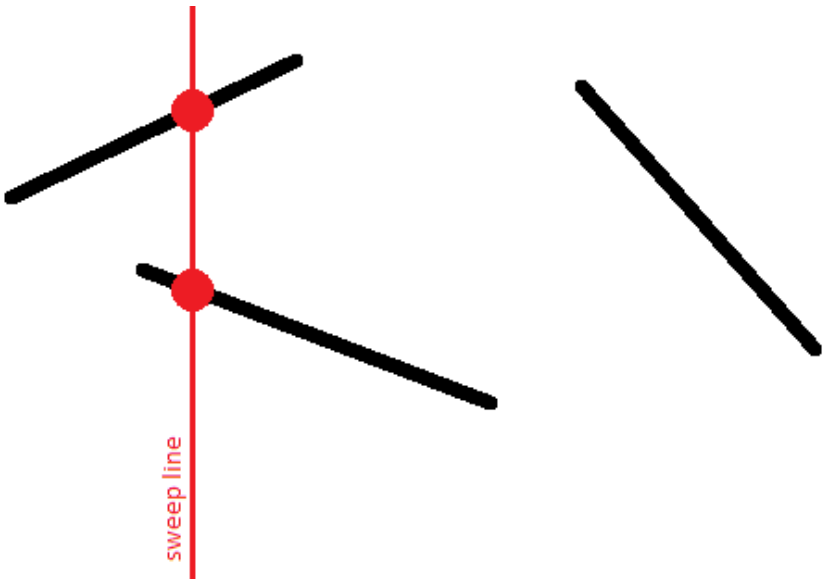

Рисунок 5.3 – Графічна ілюстація методу замітальної прямої

Таким чином, для кожного відрізка в якийсь момент часу його точка з’явиться на сканувальній прямій, потім з рухом прямої буде рухатися і ця точка, і, нарешті, в якийсь момент відрізок зникне з прямої.

Нас цікавить відносний порядок відрізків по вертикалі. А сааме: ми зберігатимемо список відрізків, що перетинають сканувальну пряму в той момент часу, коли відрізки будуть відсортовані за їх y-координатою на сканувальній прямій (рис. 5.4).

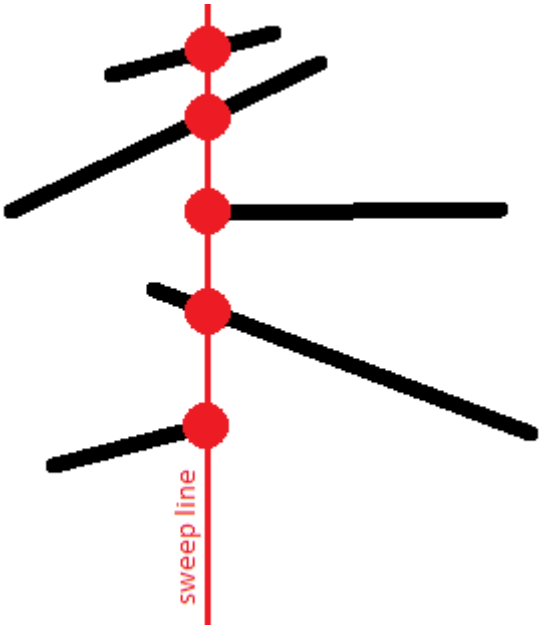

Рисунок 5.4 – Відрізки не перетинаються

Цей порядок цікавий тим, що відрізки, що перетинаються, матимуть однакову y-координату хоча б в один момент часу (рис. 5.5):

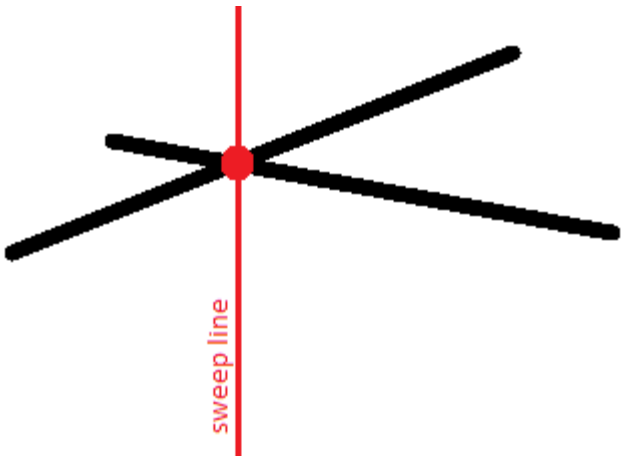

Рисунок 5.5 – Відрізки перетинаються

Сформулюємо *ключові твердження*.

• Для пошуку пари, що перетинається, достатньо розглядати при кожному фіксованому положенні сканувальної прямої **тільки сусідні відрізки**.

• Достатньо розглядати сканувальну пряму не в усіх можливих дійсних позиціях $(-\infty \ldots +\infty)$, а **лише в тих позиціях, коли з'являються нові відрізки або зникають старі**. Інакше кажучи, досить обмежитися лише положеннями, рівними абсцисам точок-кінців відрізків.

• У разі появи нового відрізка достатньо **вставити** його в потрібне місце в список, отриманий для попередньої сканувальної прямої. Перевіряти на перетин треба **тільки відрізок (що додається) з його безпосередніми сусідами в списку зверху і знизу.**

• У разі зникнення відрізка достатньо **видалити** його зі списку. Після цього треба **перевірити на перетин між собою відрізки, які були його верхнім та нижнім сусідами** у списку.

• Інших змін у порядку проходження відрізків у списку, крім описаних, не існує. Інших перевірок на перетин робити не треба.

Для розуміння істинності цих тверджень достатньо *таких зауважень*

• Два відрізки, що не перетинаються, ніколи не змінюють свого відносного порядку.

Справді, якщо один відрізок спочатку був вищим за інший, а потім став нижчим, то між двома цими моментами стався перетин цих двох відрізків.

• Мати однакові y-координати два відрізки, що не перетинаються, також не можуть.

• З цього випливає, що в момент появи відрізка ми можемо знайти в черзі позицію для цього відрізка, і більше цей відрізок переставляти в черзі не доведеться: його **порядок щодо інших відрізків у черзі не змінюватиметься.**

• Два відрізки, що перетинаються, в момент точки свого перетину виявляться **сусідами** один одного в черзі.

• Отже, для знаходження пари відрізків, що перетинаються, достатньо перевірити на перетин лише ті пари відрізків, які коли-небудь за час руху сканувальної прямої хоча б раз **були сусідами один одного.**

Легко помітити, що для цього достатньо лише перевіряти відрізок, що додається, зі своїми верхнім і нижнім сусідами, а також, у разі видалення відрізка, – його верхнього і нижнього сусідів (які після видалення стануть сусідами один одного).

• Слід звернути увагу, що при фіксованому положенні сканувальної прямої ми **спочатку** маємо зробити **додавання** всіх відрізків, що з'являються тут, і лише **потім – видалення** всіх відрізків, що зникають тут.

Тим самим ми не пропустимо перетину відрізків «по вершині»: тобто такі випадки, коли два відрізки мають загальну вершину.

• Зауважимо, що вертикальні відрізки насправді не впливають на коректність алгоритму. Ці відрізки відрізняються тим, що вони з'являються і зникають в той самий момент часу. Однак? за рахунок попереднього зауваження, ми знаємо, що спочатку всі відрізки будуть додані в чергу, і потім будуть видалені. Отже, якщо вертикальний відрізок перетинається з якимось іншим відкритим у цей момент відрізком (зокрема вертикальним), це буде виявлено.

У яке місце черги поміщати вертикальні відрізки? Адже вертикальний відрізок не має однієї певної y-координати, він простягається на цілий відрізок по y координаті. Однак легко зрозуміти, що як y координату можна взяти будь-яку координату з цього відрізка.

Таким чином, весь алгоритм здійснить не більше $2n$ тестів на перетин пари відрізків, і здійснить $O(n)$ операцій з чергою відрізків (по $O(1)$ операцій в моменти появи та зникнення кожного відрізка).

Підсумкова асимптотика алгоритму становить $O(n \log n)$.

5.3 Контрольні питання

Які основні структури даних потребує реалізація алгоритму перевірки перетину відрізків за методом замітальної прямої?

Порівняйте ефективність алгоритму перевірки перетину відрізків за методом замітальної прямої та «наївного» алгоритму.

Придумайте можливі застосування методу замітальної прямої.

6 ХЕШУВАННЯ ДАНИХ

6.1 Загальні поняття про хешування даних

Для прискорення доступу до даних можна використовувати попереднє їх впорядкування згідно зі значеннями ключів. За такої умови можуть використовувати методи пошуку в упорядкованих структурах даних, наприклад, метод дихотомічного пошуку, що суттєво скорочує час пошуку даних за значенням ключа. Однак, додавши новий запис, потрібно дані знову впорядкувати. Втрати часу на повторне впорядкування можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних використовується так зване випадкове впорядкування або хешування. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції h, яка використовується для «обчислення» адреси за значенням ключа, як це показано на рис. 6.1.

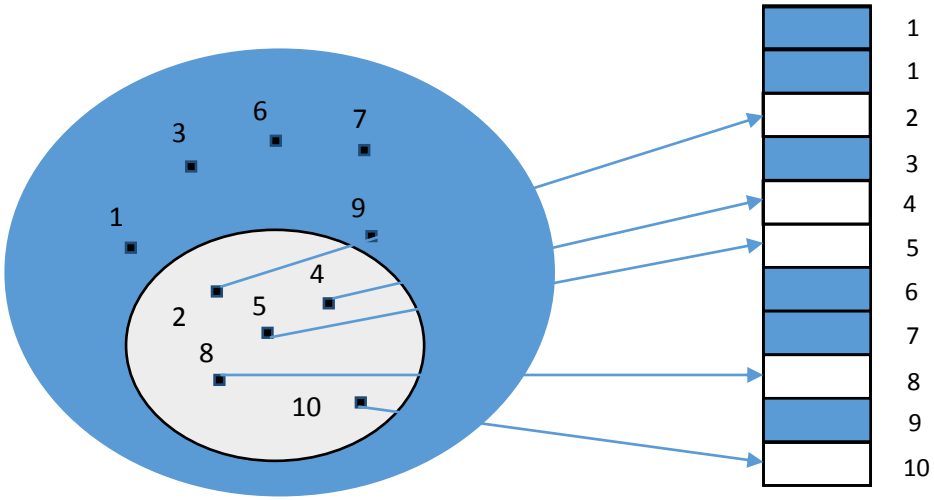

Рисунок 6.1 – Ілюстрація принципу хешування даних

Ідеальною хеш-функцією є така хеш-функція, яка для будь-яких двох неоднакових ключів дає неоднакові адреси. Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів відомі наперед. Така організація даних носить назву «досконале хешування».

У випадку наперед невизначеної множини значень ключів і обмеження розміру таблиці підбір досконалої функції складний. Тому часто використовують хеш-функції, які не гарантують виконання умови [16].

6.2 Колізії та методи їх розв’язання

Вочевидь, що, оскільки існує більше можливих значень ключа, ніж комірок в таблиці, то деякі значення ключів можуть відповідати одним і

тим самим коміркам таблиці. Даний випадок носить назву «колізій», а такі ключі називаються «ключі-синоніми».

Щоб уникнути цієї потенційної проблеми, схема хешування має містити алгоритм розв'язання колізій, який визначає послідовність дій у випадку, якщо ключ відповідає позиції в таблиці, яка вже зайнята іншим записом.

Як працюють методи розв'язання колізій? Спочатку встановлюють відповідність між ключем запису і розміщенням в хеш-таблиці. Якщо ця комірка вже зайнята, вони відображають ключ на іншу комірку таблиці. Якщо вона також вже зайнята, процес повторюється знову до тих пір, поки нарешті алгоритм не знайде незайняту комірку в таблиці.

В результаті, для реалізації хешування потрібні три речі:

- структура даних (хеш-таблиця) для зберігання даних;
- функція хешування, яка встановлює відповідність між значеннями ключа і розміщенням в таблиці;
- алгоритм розв'язання колізій, який визначає послідовність дій, якщо декілька ключів відповідають одній комірці таблиці.

Для розв'язання колізій використовуються різноманітні методи, ми обмежимося описом методів «ланцюжків» і «відкритої адресації».

Методом ланцюжків називається метод, в якому для розв'язання колізій в усі записи вводяться вказівники, які використовуються для організації списків – «ланцюжків переповнення». У випадку виникнення колізій, заповнюючи таблиці, в список для потрібної адреси хеш-таблиці додається ще один елемент (рис. 6.2).

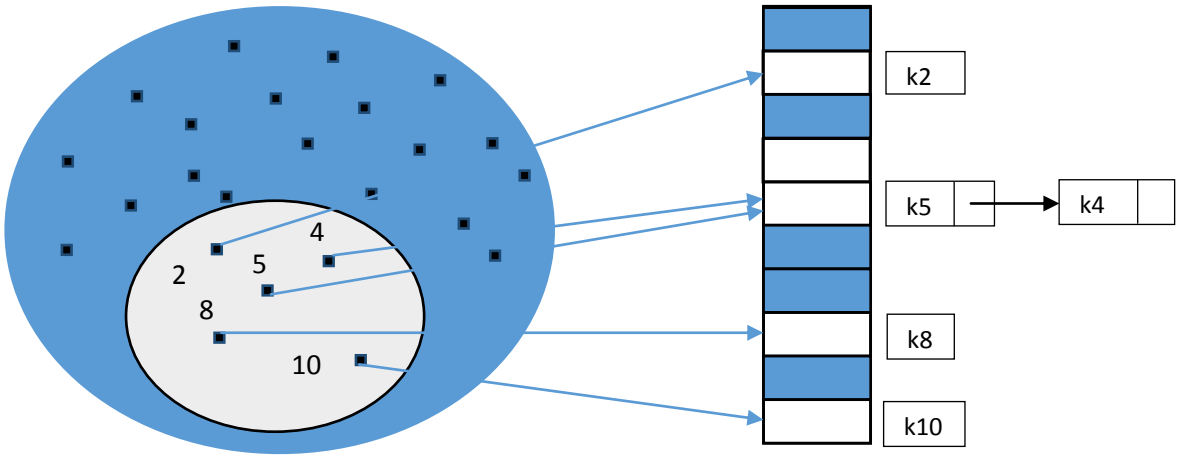

Рисунок 6.2 – Ілюстрація розв'язання колізій методом ланцюжків

Пошук в хеш-таблиці з ланцюжками переповнення здійснюється таким чином. Спочатку обчислюється адреса за значенням ключа. Потім здійснюється послідовний пошук в списку, який зв'язаний з обчисленою адресою.

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

Якщо ланцюжки приблизно однакові за розміром, то у цьому випадку списки усіх ланцюжків мають бути найбільш короткими за цієї кількості ланцюжків. Якщо вихідна множина складається з N елементів, а кількість комірок в таблиці – M, тоді середня довжина списків буде дорівнювати $\alpha = N/M$ елементів. Якщо можна оцінити N і вибрати M якомога ближчим до цієї величини, то в списку буде один–два елементи. Тоді час доступу до елемента множини буде малою постійною величиною, яка залежить від N. В [3] показано, що середній час пошуку елемента в хеш-таблиці складає $\Theta(1 + \alpha)$.

Одна з переваг цього методу хешування полягає в тому, що в разі його використання хеш-таблиці ніколи не переповнюються. За цих обставин вставляння і пошук елементів завжди виконується дуже просто, навіть якщо елементів у таблиці дуже багато. Із хеш-таблиці, яка використовує ланцюжки, також просто вилучати елементи, водночас елемент просто вилучається з відповідного зв'язного списку.

Один із недоліків зв'язування полягає в тому, що якщо кількість зв'язних списків недостатньо велика, то розмір списків може стати великим, за таких умов для вставляння чи пошуку елемента потрібно буде перевірити велику кількість елементів списку.

Метод відкритої адресації полягає в тому, щоб, користуючись якимось алгоритмом, який забезпечує перебір елементів таблиці, переглядати їх в пошуках вільного місця для нового запису.

З використанням методу відкритої адресації всі елементи зберігаються у хеш-таблиці, тобто. кожен запис таблиці містить або елемент динамічної множини, або значення NIL. Шукаючи елемент ми систематично перевіряємо комірки таблиці до того моменту, доки знайдемо шуканий елемент або переконаємося у його відсутності у таблиці. Тут, на відміну від методу ланцюжків, немає ані списків, ані елементів, що зберігаються поза таблицею. Таким чином, у методі відкритої адресації хеш-таблиця може бути заповненою, унеможливлюючи вставляння нових елементів; коефіцієнт заповнення не може перевищувати 1.

Звичайно, при хешуванні з розв'язанням колізій методом ланцюжків можна використовувати вільні місця в хеш-таблиці для зберігання пов'язаних списків, але перевага відкритої адресації полягає в тому, що вона дозволяє повністю відмовитися від вказівників. Замість слідування за вказівниками, ми обчислюємо послідовність комірок, що перевіряються. Додаткова пам'ять, що вивільняється внаслідок відмови від вказівників, дозволяє використовувати хеш-таблиці більшого розміру при тій же загальній кількості пам'яті, потенційно приводячи до меншої кількості колізій та більш швидкій вибірці.

У разі вставляння за відкритої адресації ми послідовно перевіряємо, або досліджуємо (probe), комірки хеш-таблиці до тих пір, поки не знахо-

димо порожню комірку, в яку поміщаємо ключ, що вставляється.. На відміну від фіксованого порядку дослідження комірок 0, 1, . . . , m - 1 (для чого потрібно Θ(n) часу), характерного для методу ланцюжків, за відкритої адресації послідовність досліджуваних комірок залежить від ключа, що вставляється в таблицю. Щоб визначити досліджувані комірки ми розширимо хеш-функцію, внесши до неї як другий аргумент номер дослідження (починається з 0).

В результаті хеш-функція стає такою:

$h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\}$.

У методі відкритої адресації потрібно, щоб для кожного ключа k послідовність досліджень

$$h(k, 0), h(k, 1), \ldots, h(k, m - 1)$$

була б перестановкою множини 0, 1, . . . m – 1, щоб в результаті могли бути переглянуті всі комірки хеш-таблиці.

Для обчислення послідовності досліджень за відкритої адресації зазвичай використовуються три методи: лінійне дослідження, квадратичне дослідження та подвійне хешування. Ці методи гарантують, що для кожного ключа k $h(k, 0), h(k, 1), \ldots, h(k, m - 1)$ є перестановкою $0, 1, \ldots, m - 1$. Однак ці методи не задовольняють припущення про рівномірне хешування, оскільки жоден з них не в змозі згенерувати більше m^2 різних послідовностей досліджень (замість m!, потрібних для рівномірного хешування). Найбільшу кількість послідовностей досліджень дає подвійне хешування і, як і слід очікувати, дає найкращі результати.

Лінійне дослідження зводиться до послідовного перебору елементів таблиці з деяким фіксованим кроком. Якщо крок дорівнює одиниці, відбувається послідовний перебір усіх елементів після поточного.

Квадратичне дослідження відрізняється від лінійного тим, що крок перебору елементів нелінійно залежить від номера спроби знайти вільний елемент. Завдяки нелінійності такої адресації зменшується кількість спроб при великій кількості ключів-синонімів. Однак навіть відносно невелика кількість спроб може швидко привести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номера спроби.

Ще один різновид методу відкритої адресації, яка називається **подвійним хешуванням**, базується на нелінійній адресації, яка досягається за рахунок підсумовування значень основної та додаткової хеш-функцій.

Очевидно, що з заповненням хеш-таблиці будуть відбуватися колізії, і в результаті їх розв’язання методами відкритої адресації чергова адреса може вийти за межі адресного простору таблиці. Щоб це явище відбувалося рідше, можна піти на збільшення розмірів таблиці порівняно з діапазоном адрес, які обчислюються хеш-функцією.

З одного боку, це приведе до скорочення кількості колізій і прискорення роботи з хеш-таблицею, а з іншого – до нераціональних витрат адресного простору. Навіть за збільшення таблиці в два рази, порівняно з областю значень хеш-функції, немає гарантій того, що в результаті колізій адреса не перевищить розмір таблиці. Водночас в початковій частині таблиці може залишатися достатньо вільних елементів. Тому на практиці використовують циклічний перехід на початок таблиці.

Розглядаючи можливість виходу за межі адресного простору таблиці, ми не враховували фактори наповненості таблиці й вдалого вибору хеш-функції. За великої наповненості таблиці виникають часті колізії і циклічні переходи на початок таблиці. За невдалого вибору хеш-функції відбуваються аналогічні явища. В найгіршому випадку за повного заповнення таблиці алгоритми циклічного пошуку вільного місця призведуть до зациклювання. Тому, використовуючи хеш-таблиці, потрібно намагатися уникати дуже щільного заповнення таблиць. Зазвичай довжину таблиці вибирають із розрахунку дворазового перевищення передбачуваної максимальної кількості записів. Не завжди, організовуючи хешування, можна правильно оцінити потрібну довжину таблиці, тому у випадку великої наповненості таблиці може знадобитися рехешування. У цьому випадку збільшують довжину таблиці, змінюють хеш-функцію і впорядковують дані.

Проводити окреме оцінювання щільності заповнення таблиці після кожної операції вставляння недоцільно, тому можна проводити таке оцінювання непрямим способом – за кількістю колізій під час одного вставляння. Достатньо визначити деякий поріг кількості колізій, при перевищенні якого потрібно провести рехешування. Крім того, така перевірка гарантує неможливість зациклювання алгоритму у випадку повторного перегляду елементів таблиці.

6.3 Побудова хеш-функції

Як вже відзначалося, дуже важливим є правильний вибір хеш-функції. При вдалій побудові хеш-функції таблиця заповнюється більш рівномірно, зменшується кількість колізій і зменшується час виконання операцій пошуку, вставляння та вилучення. Для того, щоб попередньо оцінити якість хеш-функції, можна провести імітаційне моделювання [17].

Якісна хеш-функція задовольняє (приблизно) припущення простого рівномірного хешування: для кожного ключа рівноймовірне розміщення в будь-яку з m комірок, незалежно від хешування інших ключів. На жаль, цю умову зазвичай неможливо перевірити, оскільки, як правило, розподіл ймовірностей, відповідно до якого надходять ключі, що вносяться до таб-

лиці, невідомий; крім того, ключі, що вставляються, можуть не бути незалежними.

На практиці при побудові якісних хеш-функцій найчастіше використовуються різні евристичні методики. У процесі побудови велику допомогу надає інформація про розподіл ключів. Розглянемо, наприклад, таблицю символів компілятора, у якій ключами слугують символьні рядки, які мають ідентифікатори у програмі. Найчастіше в одній програмі зустрічаються схожі ідентифікатори, наприклад pt та pts. Хороша хеш-функція має мінімізувати шанси потрапляння цих ідентифікаторів до одної комірки хеш-таблиці.

При побудові хеш-функції хорошим підходом є вибір функції таким чином, щоб вона ніяк не корелювала з закономірностями, яким можуть відповідати наявні дані. Наприклад, метод ділення обчислює хеш-значення як залишок від ділення ключа на деяке просте число. Якщо це просте число не пов'язане з розподілом вихідних даних, метод часто дає хороші результати. На закінчення зауважимо, деякі застосування хеш-функцій можуть накладати додаткові вимоги, крім вимог простого рівномірного хешування. Наприклад, ми можемо вимагати, щоб «близькі», в деякому сенсі, ключі давали далекі хеш-значення.

Побудова хеш-функції методом ділення

Побудова хеш-функції методом ділення полягає у відображенні ключа k в одну з комірок шляхом отримання залишку від ділення k на m, тобто. хеш-функція має вигляд h(k) = k mod m.

Наприклад, якщо хеш-таблиця має розмір m = 12, а значення ключа k = 100 то h(k) = 4. Оскільки для обчислення хеш-функції потрібна лише одна операція ділення, хешування шляхом ділення вважається досить швидким.

Використовуючи цей метод ми, зазвичай, намагаємося уникати деяких значень m. Наприклад, m не має бути степенем 2, оскільки якщо $m = 2^p$, то h(k) є просто p молодших бітів числа k. Якщо тільки заздалегідь невідомо, що всі набори молодших p бітів ключів рівноймовірні, краще будувати хеш-функцію таким чином, щоб її результат залежав від усіх бітів ключа.

Найчастіше хороші результати можна отримати, вибираючи як значення m просте число, досить далеке від степеня двійки. Припустимо, наприклад, що ми хочемо створити хеш-таблицю з розв'язанням колізій методом ланцюжків для зберігання n = 2000 символьних рядків, розмір символів яких дорівнює 8 бітам. Нас влаштовує перевірка в середньому трьох елементів при невдалому пошуку, так що ми вибираємо розмір таблиці рівним m = 701. Число 701 вибрано як просте число, близьке до величини 2000/3 і не є степенем 2. Розглядаючи кожен ключ k як ціле число, ми отримуємо шукану хеш-функцію h(k) = k mod 701.

6.4 Універсальне хешування

Якщо зловмисник буде навмисно вибирати ключі для хешування за допомогою конкретної хеш-функції, то він зможе підібрати n значень, які будуть хешуватися в одну і ту саму комірку таблиці, приводячи до середнього часу вибирання $\Theta(n)$. Таким чином, будь-яка фіксована хеш-функція стає вразливою, і єдиний ефективний вихід із ситуації – випадковий вибір хеш-функції, який залежить від того, з якими саме ключами їй належить працювати. Такий підхід, який називається універсальним хешуванням, гарантує хорошу продуктивність у середньому, незалежно від того, які дані будуть обрані зловмисником.

Головна ідея універсального хешування полягає у випадковому виборі хеш-функції з деякого ретельно відібраного класу функцій на початку роботи програми. Як і у разі швидкого сортування, рандомізація гарантує, що одні й ті самі вхідні дані неспроможні постійно давати найгіршу поведінку алгоритму. В силу рандомізації алгоритм працюватиме щоразу по-різному, навіть для тих самих вхідних даних, що гарантує високу середню продуктивність для будь-яких вхідних даних.

Нехай H – кінцева множина хеш-функцій, що відображають простір ключів U в діапазон $\{0, 1, 2, \ldots, m - 1\}$. Така множина називається універсальною, якщо для кожної пари різних ключів $k, l \in U$ кількість хеш функцій $h \in H$, для яких $h(k) = h(l)$, не перевищує $|H|/m$. Іншими словами, у разі випадкового вибору хеш-функції H ймовірність колізії між різними ключами k і l не перевищує ймовірності збігу двох випадковим чином обраних хеш-значень з множини $\{0, 1, 2, \ldots, m - 1\}$, що дорівнює $1/m$.

Побудова універсальної множини хеш-функцій

Побудувати таку множину досить просто, що випливає з теорії чисел. Почнемо з вибору простого числа p, досить великого, щоб усі можливі ключі перебували в діапазоні від 0 до p - 1 включно. Нехай Z_p позначає множину $\{0, 1, \ldots, p - 1\}$, а Z^*_p – множину $\{1, 2, \ldots, p - 1\}$. Оскільки p – просте число, ми можемо розв’язувати рівняння за модулем p. З припущення, що простір ключів більший, ніж кількість комірок у хеш-таблиці, випливає, що $p > m$.

Тепер визначимо хеш-функцію $h_{a,b}$ для будь-яких $a \in Z^*_p$ і $b \in Z_p$ так:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m.$$

Наприклад, якщо $p = 17$ та $m = 6$ $h_{3,4}(8) = 5$. Сімейство всіх таких функцій утворює множину

$$H_{p,m} = \{h_{a,b} : a \in Z^*_p \text{ і } b \in Z_p\}.$$

Кожна хеш-функція $h_{a,b}$ відображає Z_p на Z_m. Цей клас хеш-функцій має ту властивість, що розмір вихідного діапазону m довільний і не

обов’язково являє собою просте число. Оскільки число a можна вибрати p - 1 способом, і p способами – число b, всього у множині $H_{p,m}$ міститься $p(p-1)$ хеш-функцій.

6.5 Ідеальне хешування

Хоча найчастіше хешування використовується завдяки високій середній продуктивності, можлива ситуація, коли реально отримати високу продуктивність хешування в найгіршому випадку. Такою ситуацією є статична множина ключів, тобто, після того, як всі ключі збережені в таблиці, їх множина ніколи не змінюється. Низка застосунків, у силу своєї природи, працює зі статичними множинами ключів. Як приклад можна навести множину зарезервованих слів мови програмування або множину імен файлів на компакт-диску. Ідеальним хешуванням ми називаємо методику, яка в найгіршому випадку виконує пошук за O(1) звернень до пам’яті.

Основна ідея ідеального хешування досить проста. Ми використовуємо дворівневу схему хешування з універсальним хешуванням на кожному рівні. Перший рівень, насправді, той же, що і у разі хешування з ланцюжками: n ключів хешуються в m комірок з використанням хеш-функції h, ретельно обраної з сімейства універсальних хеш-функцій. Однак замість того, щоб створювати список ключів, хешованих в комірку j, ми використовуємо маленьку вторинну хеш-таблицю S_j зі своєю хеш-функцією h_j. Шляхом точного вибору хеш-функції h_j ми можемо гарантувати відсутність колізій на другому рівні.

6.4 Контрольні питання

Що являє собою ключ у хеш-таблиці?
Дайте означення поняття колізії.
У чому полягає розв’язання колізій методом ланцюжків?
Які різновиди методу відкритої адресації Ви знаєте?
Які вимоги можуть висуватися до хеш-функції?
Охарактеризуйте побулову хеш-функції методом ділення.
Що зумовлює потребу універсального хешування?
Як відбувається побудова універсальної множини хеш-функцій?
Що таке ідеальне хешування?
У яких випадках можливе ідеальне хешування?

7 БІНАРНІ ДЕРЕВА ПОШУКУ

7.1 Поняття про бінарні дерева пошуку

Дерева пошуку являють собою структури даних, які підтримують багато операцій з динамічними множинами, а саме пошук елемента, мінімального та максимального значення, попереднього та наступного елемента, вставляння та видалення. Таким чином, дерево пошуку може використовуватися і як словник, і як черга з пріоритетами.

Основні операції в бінарному дереві пошуку виконуються за час, пропорційний його висоті. Для повного бінарного дерева з n вузлами ці операції виконуються за час $\Theta(\lg n)$, у найгіршому випадку. Математичне очікування висоти побудованого випадковим чином бінарного дерева дорівнює $O(\lg n)$, так що всі основні операції над динамічною множиною в такому дереві виконуються, в середньому, за час $\Theta(\lg n)$.

Як випливає з назви, бінарне дерево пошуку в першу чергу є бінарним деревом, як показано на рис. 7.1. Таке дерево може бути подано за допомогою зв'язної структури даних, в якій кожен вузол є об'єктом. На додаток до полів ключа key та супутніх даних, кожен вузол містить поля left, right та p, які вказують на лівий та правий дочірні вузли та на батьківський вузол відповідно. Якщо дочірній або батьківський вузол відсутні, поле має значення NULL. Єдиний вузол, покажчик p якого дорівнює NULL, – це кореневий вузол дерева. Ключі в бінарному дереві пошуку зберігаються таким чином, щоб будь-якої миті задовольняти нижчевказану властивість бінарного дерева пошуку

> *Якщо x – вузол бінарного дерева пошуку, а вузол y знаходиться у лівому піддереві x, то key [y] ≤ key [x]. Якщо вузол y знаходиться у правому піддереві x, то key [x] ≤ key [y].*

Так, на рис. 7.1, а) ключ кореня дорівнює 5, ключі 2, 3 та 5, які не перевищують значення ключа в корені, знаходяться в його лівому піддереві, а ключі 7 і 8, які не менші, ніж ключ 5, – в його правому піддереві. Та ж властивість, як легко переконатись, виконується для кожного іншого вузла дерева. На рис. 7.1, б) показано дерево з тими ж вузлами і тими ж властивостями, проте менш ефективне в роботі, оскільки його висота дорівнює 4, на відміну від дерева на рис. 7.1, а), висота якого дорівнює 2.

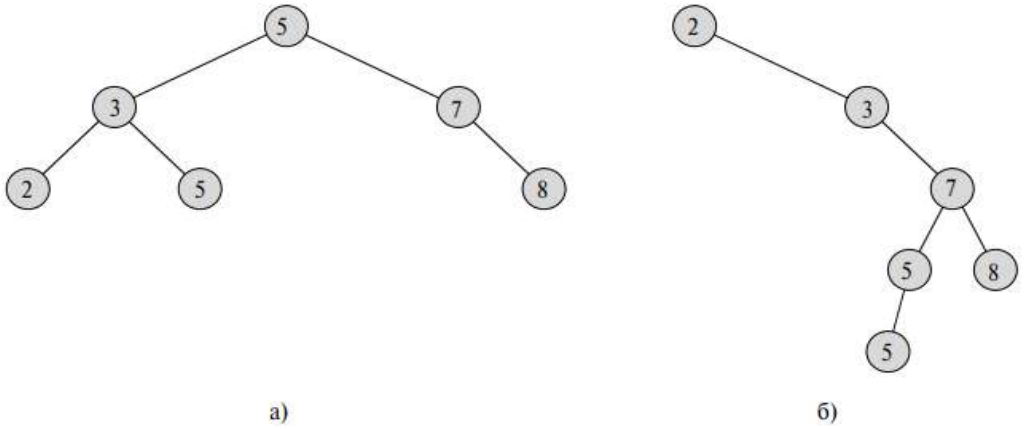

Рисунок 7.1 – Бінарні дерева пошуку

Властивість бінарного дерева пошуку дозволяє нам вивести всі ключі, що знаходяться у дереві, у відсортованому порядку за допомогою простого рекурсивного алгоритму, що називається центрованим (симетричним) обходом дерева (inorder tree walk). Цей алгоритм отримав таку назву у зв'язку з тим, що ключ у корені піддерева виводиться між значеннями ключів лівого піддерева та правого піддерева. Є й інші способи обходу, а саме: обхід у порядку (preorder tree walk), у якому спочатку виводиться корінь, а потім – значення лівого і правого поддерев, і обхід у зворотному порядку (postorder tree walk), коли першими виводяться значення лівого і правого піддерев, а вже потім кореня. Центрований обхід дерева T реалізується процедурою INORDER_TREE_WALK(root [T]):

```
INORDER_TREE_WALK(x)
1  if x != NULL
2     then  INORDER_TREE_WALK(left[x])
3           print key[x]
4           INORDER_TREE_WALK(right[x])
```

7.2 Робота з бінарним деревом пошуку

Найбільш поширеною операцією, що виконується з бінарним деревом пошуку, є пошук у ньому певного ключа. Крім того, бінарні дерева пошуку підтримують такі запити, як пошук мінімального та максимального елемента, а також попереднього та наступного. Всі вони можуть бути виконані в бінарному дереві пошуку заввишки h за час O(h).

Пошук

Для пошуку вузла з заданим ключем у бінарному дереві пошуку використовується процедура TREE_SEARCH, яка отримує як параметр покажчик на корінь бінарного дерева x і ключ k, а повертає покажчик на вузол з цим ключем (якщо такий існує; інакше повертається значення NULL).

```
TREE_SEARCH(x, k)
1  if x == NULL або k == key[x]
2       then return x
3  If k < key[x]
4       then return TREE_SEARCH(left[x], k)
5       else return TREE_SEARCH(right[x], k)
```

Процедура пошуку починається з кореня дерева та проходить вниз по дереву. Для кожного вузла x на шляху вниз його ключ key [x] порівнюється з переданим як параметр ключом k. Якщо ключі однакові, пошук буде завершено. Якщо k менше key [x], пошук продовжується у лівому піддереві x; якщо більше – то пошук переходить у праве піддерево. Так, на рис. 2 для пошуку ключа 13 ми маємо пройти такий шлях від кореня: 15 → 6 → 7 → 13. Вузли, які ми відвідуємо, здійснюючи рекурсивний пошук, утворюють низхідний шлях від кореня дерева, тому час роботи процедури TREE_SEARCH дорівнює O(h), де h – висота дерева. Ту ж процедуру можна записати ітеративно, «розвертаючи» кінцеву рекурсію в цикл while. На більшості комп'ютерів така версія виявляється більш ефективною. Графічну ілюстрацію процедури пошуку у бінарному дереві наведено на рис. 7.2.

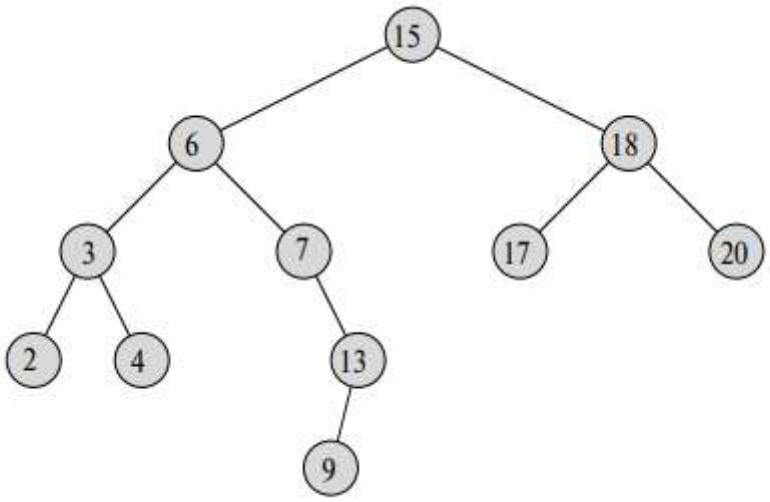

Рисунок 7.2 – Запити в бінарному дереві пошуку

Так виглядає псевдокод рекурсивної процедури пошуку:

```
ITERATIVE_TREE_SEARCH(x, k)
1 while x != NULL та k != key[x]
2       do if k < key[x]
3        then x ← left[x]
4        else x ← right[x]
5 return x
```

Пошук мінімуму та максимуму

Елемент з мінімальним значенням ключа легко знайти, слідуючи за вказівниками left від кореневого вузла до того моменту, доки не зустрінеться значення NULL. Так, на рис. 7.2, слідуючи за вказівниками left, ми пройдемо шлях 15 → 6 → 3 → 2 до мінімального ключа в дереві, що дорівнює 2. Ось як виглядає реалізація описаного алгоритму:

```
TREE_MINIMUM(x)
1 while left[x] != NULL
2       do x ← left[x]
3 return x
```

Властивість бінарного дерева пошуку гарантує правильність процедури TREE_MINIMUM. Якщо у вузла x немає лівого піддерева, то, оскільки всі ключі в правому піддереві x не менше ключа key [x], мінімальний ключ піддерева з коренем у вузлі x знаходиться у цьому вузлі. Якщо ж у вузла є ліве піддерево, то, оскільки в правому піддереві не може бути вузла з ключем, меншим key[x], а всі ключі у вузлах лівого піддерева не перевищують key[x], вузол з мінімальним значенням ключа знаходиться в піддереві, коренем якого є вузол left[x].

Алгоритм пошуку максимального елемента дерева симетричний алгоритму пошуку мінімального елемента:

```
TREE_MAXIMUM(x)
1  while right[x] != NULL
2       do x ← right[x]
3  return x
```

Наведені процедури знаходять мінімальний (максимальний) елемент дерева за час O(h), де h – висота дерева, оскільки, як і в процедурі TREE_SEARCH, послідовність вузлів, що перевіряються, утворює низхідний шлях від кореня дерева.

Вставлення

Щоб вставити нове значення v у бінарне дерево пошуку T, ми скористаємося процедурою TREE_INSERT. Процедура отримує як параметр вузол z, у якого key [z] = v, left [z] = NULL і right [z] = NULL, після чого вона таким чином змінює T та деякі поля z, що z виявляється вставленим у відповідну позицію у дереві.

```
TREE_INSERT(T, z)
1 y ← NULL
2 x ← root[T]
3 while x != NULL
4 do y ← x
5 if key[z] < key[x]
6 then x ← left[x]
7 else x ← right[x]
```

```
8 p[z] ← y
9 if y == NULL
10 then root[T] ← z            // Дерево T — порожнє
11 else if key[z] < key[y]
12 then left[y] ← z
13 else right[y] ← z
```

На рис. 7.3 показано роботу процедури TREE_INSERT. Подібно до процедур TREE_SEARCH і ITERATIVE_TREE_SEARCH, процедура TREE_INSERT починає роботу з кореневого вузла дерева і проходить низхідним шляхом. Покажчик x відзначає пройдений шлях, а покажчик y вказує на батьківський, стосовно x, вузол. Після ініціалізації цикл while у рядках 3–7 переміщує ці вказівники вниз по дереву, переміщаючись ліворуч чи праворуч залежно від результату порівняння ключів key[x] і key[z], доки x не стане рівним NULL. Це значення знаходиться саме у тій позиції, куди потрібно помістити елемент z. У рядках 8–13 виконується встановлення значень покажчиків для вставляння z. Так само, як і інші примітивні операції над бінарним деревом пошуку, процедура TREE_INSERT виконується за час O(h) у дереві заввишки h.

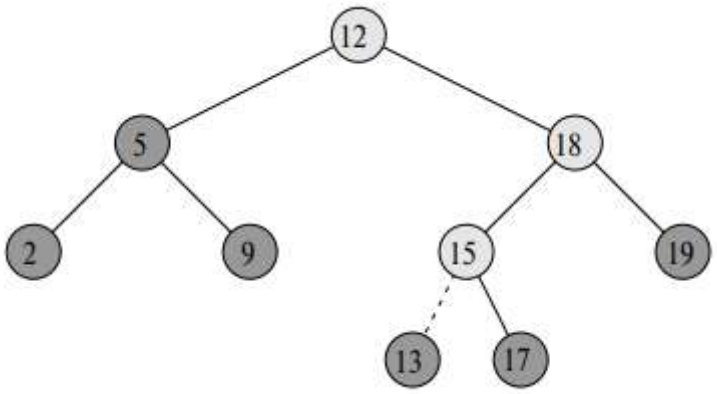

Рисунок 7.3 – Вставляння елемента з ключем 13 в бінарне дерево пошуку

Світлі вузли вказують шлях від кореня до позиції вставляння; пунктиром вказано зв’язок, що додається при вставлянні нового елемента.

Видалення

Процедура видалення певного вузла z з бінарного дерева пошуку отримує як аргумент покажчик на z. Процедура розглядає три можливі ситуації. Якщо у вузла z немає дочірніх вузлів, ми просто змінюємо його батьківський вузол p [z], замінюючи у ньому покажчик на z значенням NULL. Якщо у вузла z лише один дочірній вузол, ми видаляємо вузол z, створюючи новий зв’язок між батьківським і дочірнім вузлами вузла z. І нарешті, якщо у вузла z два дочірні вузли, то ми знаходимо наступний за ним вузол y, у якого немає лівого дочірнього вузла, прибираємо його з по-

зиції, де він знаходився раніше, шляхом створення нового зв'язку між його батьком і нащадком, і замінюємо ним вузол z.

Псевдокод процедури TREE_DELETE реалізує видалення вузла.

```
TREE_DELETE(T, z)
1     if left[z] == NULL або right[z] == NULL
2           then y ← z
3           else y ← TREE_SUCCESSOR(z)
4     if left[y] != NULL
5           then x ← left[y]
6           else x ← right[y]
7     if x != NULL
8           then p[x] ← p[y]
9     if p[y] == NULL
10          then root[T] ← x
11          else if y = left[p[y]]
12                then left[p[y]] ← x
13                else right[p[y]] ← x
14    if y != z
15          then key[z] ← key[y]
16                // Копіювання супутніх даних в z
17    return y
```

У рядках 1–3 алгоритм визначає шляхом «склейки» батька та нащадка вузол y, що видаляється з дерева. Цей вузол є або вузол z (якщо у вузла z не більше одного дочірнього вузла), або вузол, що йде за вузлом z (якщо у z два дочірніх вузла). Потім у рядках 4–6 x надається покажчик на дочірній вузол вузла y або значення NULL, якщо у y немає дочірніх вузлів. Потім вузол y забирається з дерева в рядках 7–13 шляхом зміни покажчиків у p[y] та x. І нарешті, у рядках 14–16, якщо видалений вузол y був наступним за z, ми перезаписуємо ключ z та супутні дані ключем та супутніми даними y. Видалений вузол y повертається в рядок 17, щоб викликальна процедура могла, за потреби, звільнити або використовувати пам'ять, яку він займає. Час роботи описаної процедури з деревом заввишки h становить O(h). Наведемо також без коментарів псевдокод процедури TREE_SUCCESSOR.

```
TREE_SUCCESSOR(x)
1 if right[x] != NULL
2     then return TREE_MINIMUM(right[x])
3 y ← p[x]
4 while y != NULL та x == right[y]
5     do    x ← y
6           y ← p[y]
7 return y
```

Випадкова побудова бінарного дерева пошуку

Всі базові операції з бінарними деревами пошуку мають час виконання O(h), де h – висота дерева. Однак у процесі вставляння та видалення

елементів висота дерева змінюється. Якщо, наприклад, всі елементи вставляються у дерево в послідовності, що строго зростає, то таке дерево вироджується в ланцюжок висоти n - 1. З іншого боку, $h \geq \lg n$. Як і у разі швидкого сортування, можна показати, що поведінка алгоритму в середньому набагато ближча до кращого випадку, ніж найгіршого.

У ситуації, коли, формуючи бінарне дерево пошуку, використовуються і вставляння, і видалення, про середню висоту дерев, що утворюються, відомо мало, тому обмежимося аналізом ситуації, коли дерево будується лише з використанням вставлянь, без видалень. Визначимо випадкове бінарне дерево пошуку (randomly built binary search tree) з n ключами як дерево, що виникає при вставлянні ключів у порожнє дерево у випадковому порядку, коли всі n! перестановок вхідних ключів рівноймовірні. В [3] показано, що математичне очікування висоти випадкового бінарного дерева пошуку з n ключами дорівнює O (log n).

7.3 Порівняння хеш-таблиць і бінарних дерев пошуку

Швидкість хеш-таблиці залежить, в основному, від вибору функції хешування, оскільки колізії сповільнюють доступ до потрібних елементів. Такі операції з хеш-таблицею, як вставляння, видалення та пошук, надзвичайно швидкі та займають у середньому O(1).

Однак, якщо у нас дуже погана функція хешування, яка завжди вказує на одну й ту саму комірку таблиці, найгірше, що ми можемо отримати, є O(n). У цьому випадку ми перебираємо всі елементи, як у масиві або зв'язному списку. Ключові показники порівняння збалансованого бінарного дерева пошуку і хеш-таблиці наведено на рис. 7.4 [18].

Average Time Complexity			
	Search	Insertion	Deletion
Self-balancing BST	$O(log(n))$	$O(log(n))$	$O(log(n))$
Hash Table	$O(1)$	$O(1)$	$O(1)$

Рисунок 7.4 – Порівняння бінарного дерева пошуку і хеш-таблиці

Інша ситуація, коли для хеш-таблиці потрібний лінійний час O(n), якщо розмір таблиці перевищує початково виділений розмір. Тоді рехешування має створити більшу таблицю та переробити кожен елемент у ній. Це робить хеш-таблицю не дуже придатною для роботи з «живими» даними, що постійно зростають.

Часова складність для тих самих операцій вставляння, пошуку та видалення в самобалансувальному бінарному дереві пошуку в середньому складає O(log(n)). Це значно повільніше порівняно з хеш-таблицею, якщо розглядати лише ці три операції.

Тим не менш, якщо програма вимагає таких більш складних завдань, як пошук найближчих менших або більших елементів до числа або виконання запитів діапазону, тоді відсортована структура бінарного дерева пошуку виконає пошук набагато швидше порівняно з хеш-таблицею.

Бінарні дерева пошуку, як правило, ефективно використовують **пам'ять**, оскільки вони не резервують більше пам'яті, ніж потрібно. З іншого боку, хеш-таблиці можуть бути дещо більш вимогливими, якщо ми не знаємо точну кількість елементів, які хочемо зберегти.

Зазвичай хеш-таблиця виділяє масив і уникає колізій шляхом рівномірного хешування за розміром цього масиву. Отже, якщо ми маємо хеш-таблицю з початково зарезервованою пам'яттю для 100 елементів, але використовуємо лише 20, решта пам'яті буде просто витрачено.

7.4 Контрольні питання

Поясніть відмінність між бінарними деревами та бінарними деревами пошуку.

У якому порядку вібувається обхід бінарного дерева пошуку?

Які операції у бінарних деревах пошуку Ви знаєте?

Яка складінсть операції пошуку вузла у бінарному дереві пошуку?

Яка складінсть операції вставляння вузла у бінарне дерево пошуку?

Яка складінсть операції видалення вузла з бінарного дерева пошуку?

У чому полягають переваги бінарних дерев пошуку порівняно з хеш-таблицями?

У чому полягають недоліки бінарних дерев пошуку порівняно з хеш-таблицями?

8 ОЦІНЮВАННЯ СКЛАДНОСТІ АЛГОРИТМІВ. АЛГОРИТМИ СОРТУВАННЯ

Розглядаючи вхідні дані досить великих розмірів для оцінювання тільки такої величини, як порядок зростання часу роботи алгоритму, ми тим самим вивчаємо **асимптотичну** ефективність алгоритмів.

Це означає, що нас цікавить лише те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних у межі, коли цей розмір прагне нескінченності. Зазвичай алгоритм, ефективніший в асимптотичному сенсі, буде більш продуктивним для всіх вхідних даних, крім дуже маленьких (хоча бувають і винятки).

Хоча в багатьох випадках ці позначення використовуються неформально, корисно почати з точних визначень.

8.1 Θ-позначення

Нехай час T(n) роботи деякого алгоритму на входах довжини n є $\Theta(n^2)$. Точний зміст цього твердження такий: знайдуться такі константи c_1, $c_2 > 0$ і таке число n_0, що $c_1n^2 \le T(n) \le c_2n^2$ для всіх $n \ge n_0$. Взагалі, якщо g(n) – деяка функція, то запис $f(n) = \Theta(g(n))$ означає, що знайдуться такі c_1, $c_2>0$ і таке n_0, що $0 \le c_1g(n) \le f(n) \le c_2g(n)$ для всіх $n \ge n_0$.

Функція f(n) належить множині Θ(g(n)), якщо існують позитивні константи c_1 і c_2, що дозволяють укласти цю функцію в межі між функціями $c_1g(n)$ і $c_2g(n)$ для досить великих n. Оскільки Θ(g(n)) – це нескінченність, то можна написати «$f(n) \in \Theta(g(n))$». Це означає, що функція f(n) належить множині Θ(g(n)) (іншими словами, є його елементом). Але найчастіше використовують еквівалентний запис «$f(n) = \Theta(g(n))$». Але таке позначення варто вживати з обережністю: встановивши, що $f_1(n) = \Theta(g(n))$ і $f_2(n) = \Theta(g(n))$, не робити висновок, що $f_1(n) = f_2(n)$.

На рис. 8.1 показано інтуїтивне зображення функцій f(n) і g(n) таких, що $f(n) = \Theta(g(n))$. Для всіх значень n, що лежать праворуч від n_0, функція f(n) більша або дорівнює функції $c_1g(n)$, але не перевищує функції $c_2g(n)$.

Іншими словами, для всіх $n \ge n_0$ функція f(n) дорівнює функції g(n) з точністю до постійного множника. Кажуть, що функція g(n) є асимптотично точною оцінкою функції f(n).

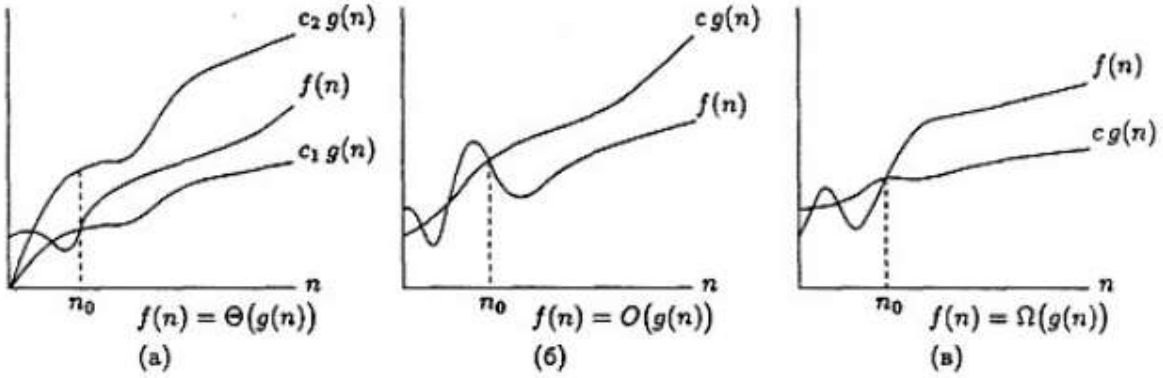

Рисунок 8.1 – Ілюстрація оцінювання складності алгоритмів

Покажемо для прикладу, що $(1/2)n^2 - 3n = \Theta(n^2)$. Відповідно до означення треба вказати додатні константи c_1, c_2 і число n_0 так, щоб нерівності

$$c_1 n^2 \le 1/2\, n^2 - 3n \le c_2 n^2$$

виконувалися для всіх $n \ge n_0$. Розділимо дані нерівності на n^2

$$c_1 \le 1/2 - 3/n \le c_2 .$$

Бачимо, що для виконання другої нерівності досить покласти $c_2 = 1/2$. Перше буде виконано, якщо, наприклад, $n_0 = 7$ і $c_1 = 1/14$. Звичайно, константи можна вибрати по-іншому, проте важливо не те, як їх вибрати, а те, що така можливість існує.

Інший приклад використання формального визначення: покажемо, що $6n^3 \ne \Theta(n^2)$. Справді, нехай існують такі c_2 і n_0, що $6n^3 \le c_2 n^2$ для всіх $n \ge n_0$. Але тоді $n \le c_2/6$ для всіх $n \ge n_0$, що явно не так.

Визначаючи асимптотично точну оцінку для суми, можна відкидати члени меншого порядку, що у разі великих n стають малими порівняно з основним складовим. Зазначимо також, що коефіцієнт при старшому члені ролі не відіграє (він може вплинути тільки на вибір констант c_1 і c_2).

Згадаємо важливий окремий випадок використання Θ-позначень: Θ(1) означає обмежену функцію, відокремлену від нуля деякою додатною константою при досить великих значеннях аргумента.

8.2 O- і Ω- позначення

Запис $f(n) = \Theta(g(n))$ містить дві оцінки: верхню і нижню. Їх можна розділити. Кажуть, що $f(n) = O(g(n))$, якщо знайдеться така константа $c > 0$ і таке число n_0, що $0 \le f(n) \le cg(n)$ для всіх $n \ge n_0$.

O-позначення застосовуються, коли потрібно вказати верхню межу функції з точністю до постійного множника. Інтуїтивне уявлення про O-позначення дозволяє отримати рис. 8.1, б. Для всіх n, що лежать праворуч від n_0, значення функції f(n) не перевищує значення функції cg(n).

Щоб вказати, що функція f(n) належить множині O(g(n)), пишуть f(n) = O(g(n)). Зверніть увагу, що з f(n) = Θ(g(n)) випливає f(n) = O(g(n)), оскільки Θ-позначення сильніші, ніж O-позначення. В значеннях теорії множин $\Theta(g(n)) \subset O(g(n))$.

Щоб записати час роботи алгоритму в O-позначеннях, часто-густо достатньо вивчити його загальну структуру. Наприклад, наявність подвійного вкладеного циклу в структурі алгоритму сортування за методом вставлянь свідчить про те, що верхня межа часу роботи в найгіршому випадку виражається як $O(n^2)$: «вартість» кожної ітерації у внутрішньому циклі обмежена зверху константою O(1), індекси i та j – числом n, а внутрішній цикл виконується щонайбільше один раз для кожної з n^2 пар значень i та j.

Оскільки O-позначення описують верхню межу, то в ході їх використання для обмеження часу роботи алгоритму у найгіршому випадку ми отримуємо верхню межу цієї величини для будь-яких вхідних даних. Таким чином, межа $O(n^2)$ для часу роботи алгоритму в найгіршому випадку може застосовуватися для часу розв'язання задачі з будь-якими вхідними даними, чого не можна сказати про Θ-позначення. Наприклад, оцінка $\Theta(n^2)$ для часу сортування вставлянням у найгіршому випадку не застосовується для довільних вхідних даних. Наприклад, якщо вхідні елементи вже відсортовані, час роботи алгоритму сортування вставлянням оцінюється як Θ(n).

Кажуть, що $f(n)=\Omega(g(n))$, якщо знайдеться така константа $c>0$ і таке число n_0, що $0 \le cg(n) \le f(n)$ для всіх $n \ge n_0$. Інтуїтивне уявлення про Ω-позначення дозволяє отримати рис.1, в. Для всіх n, що лежать праворуч від n_0, значення функції f (n) більші або дорівнюють cg(n).

8.3 Алгоритми сортування

Сортування за квадратичний час. Варіації на тему «сортування бульбашкою»

Сортування вставлянням було розглянуто раніше (див. розділ 1).

Розберемо алгоритм, у якому обмін місцями двох елементів є характерною особливістю процесу. Викладений нижче алгоритм ґрунтується на порівнянні й зміні місць для пари сусідніх елементів і продовженні цього процесу доти, доки не будуть упорядковані всі елементи (тому цей алгоритм отримав ще назву «сортування прямим обміном»). І такі проходи по масиву потрібно повторювати, пересуваючи щоразу найменший елемент послідовності, яка залишилась, до лівого кінця масиву. Якщо розглядати масиви як вертикальні, а не горизонтальні, то елементи можна інтерпретувати як бульбашки у чані з водою, причому вага кожного відповідає його ключу. Тут під час кожного проходу одна бульбашка «піднімається» до рівня, що відповідає її вазі (табл. 8.1).

Таблиця 8.1 – Сортування бульбашкою

прохід	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
Елементи	44	6	6	6	6	6	6	6
	55	44	12	12	12	12	12	12
	32	55	44	18	18	18	18	18
	42	12	55	44	42	42	42	42
	94	42	18	55	44	44	44	44
	18	94	42	42	55	55	55	55
	6	18	94	67	67	67	67	67
	67	67	67	94	94	94	94	94

Псевдокод, що реалізує такий підхід у найпростішому вигляді, наведено нижче.

```
BUBBLESORT(A)
1     for i ← 2 to n
2           do for j ← n downto i
3                 do if A[j-1] > A[j] then
4                       виконати обмін A[j] ↔ A[j-1]
```

Поліпшення цього алгоритму очевидне. На прикладі у табл. 8.1 бачимо, що три останніх проходи не впливають на порядок елементів, оскільки вони вже відсортовані. Отже, потрібно запам’ятовувати, мали місце перестановки під час деякого проходу або ні. І якщо у черговому проході перестановок не було – алгоритм можна завершувати.

Зазначимо, що це поліпшення можна знову ж поліпшити, якщо запам’ятовувати не лише факт обміну, а й індекс k останнього обміну. Цілком зрозуміло, що всі пари сусідніх елементів вище цього індексу вже відсортовані. Тому перегляд можна закінчувати на цьому індексі, а не йти до заздалегідь визначеної нижньої межі для і.

Крім того, тут можна помітити ще деяку своєрідну асиметрію: «спливає» бульбашка на потрібне місце за один прохід, а «тоне» на потрібне місце дуже повільно – лише на один крок за один прохід. Наприклад, масив

12 18 42 44 55 67 94 06

за допомогою удосконаленого бульбашкового сортування можна упорядкувати за один прохід, а для массиву

94 06 12 18 42 44 55 67

потрібно сім проходів. Така асиметрія наводить на думку про третє поліпшення: послідовно чергувати напрямок проходів. Алгоритм, що дозволяє реалізувати таке поліпшення, називають шейкерним сортуванням (ShakerSort). Псевдокод, що дозволяє реалізувати алгоритм шейкерного

сортування, наведено нижче (тут змінні L, R, k позначають ліву та праву межі ділянки сортованого масиву, а також індекс останнього обміну, відповідно). А таблиця 8.2 ілюструє процес сортування тих же (табл. 8.1) восьми ключів.

```
SHAKERSORT(A)
1     L ← 2
2     R ← n
3     k ← n
4     repeat
5           for j ← R downto L
6                 do if A[j-1]>A[j] then
7                       виконати обмін A[j] ↔ A[j-1]
8                       k ← j
9           L ← k+1
10          for j ← L to R
11                do if A[j-1]>A[j] then
12                      виконати обмін A[j] ↔A[j-1]
13                      k ← j
14          R ← k-1
15    until L<R
```

Таблиця 8.2 – Приклад шейкерного сортування

L	2	3	3	4	4
R	8	8	7	7	4
Напрямок проходу	↑	↓	↑	↓	↑
Елементи масиву	44	6	6	6	6
	55	44	44	12	12
	32	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	6	18	67	67	67
	67	67	94	94	94

Сортування за псевдолінійний час. Алгоритм Хоара

Сортування злиттям було розглянуто раніше. Розглянемо, напевно, кращий, з практичного погляду алгоритм, який носить назву швидкого сортування, або алгоритм Хоара.

Алгоритм Хоара – це алгоритм «розділяй і володарюй». Як і всі алгоритми «розділяй і володарюй», він спочатку ділить великий масив на два менші підмасиви, а потім рекурсивно сортує підмасиви. Насправді весь процес має три етапи.

Вибір опорного елемента. Виберіть елемент, що називається опорним, з масиву (зазвичай це найлівіший чи най правіший, чи середній елемент розділу).

Поділ. Переупорядкуйте масив таким чином, щоб усі елементи зі значеннями менше опорного розташовувалися перед опорним. Навпаки, всі елементи зі значеннями більшими, ніж точка опори, йдуть після неї. Рівні значення можуть у будь-якому напрямі. Після цього розбиття опорний елемент займає своє кінцеве положення.

Повторювати. Рекурсивно застосуйте описані вище кроки до підмасиву елементів із меншими значеннями, ніж у опорного, та окремо до підмасиву елементів із більшими значеннями, ніж у опорного.

Базовим випадком рекурсії є масиви розміром 1, які ніколи не потрібно сортувати. На нижченаведеній діаграмі показано, як ми вибираємо крайній правий елемент як опорний на кожному етапі алгоритму Хоара, розбиваємо масив та повторюємо дії у двох підмасивах, які ми отримуємо після процесу поділу. Зверніть увагу, що етапи вибору опори та розбиття можуть бути виконані декількома способами і вибір конкретних схем реалізації істотно впливає на продуктивність алгоритму. Сортування масиву A = <9, -3, 5, 2, 6, 8, -6, 1, 3> за алгоритмом Хоара наведено на рис. 8.2:

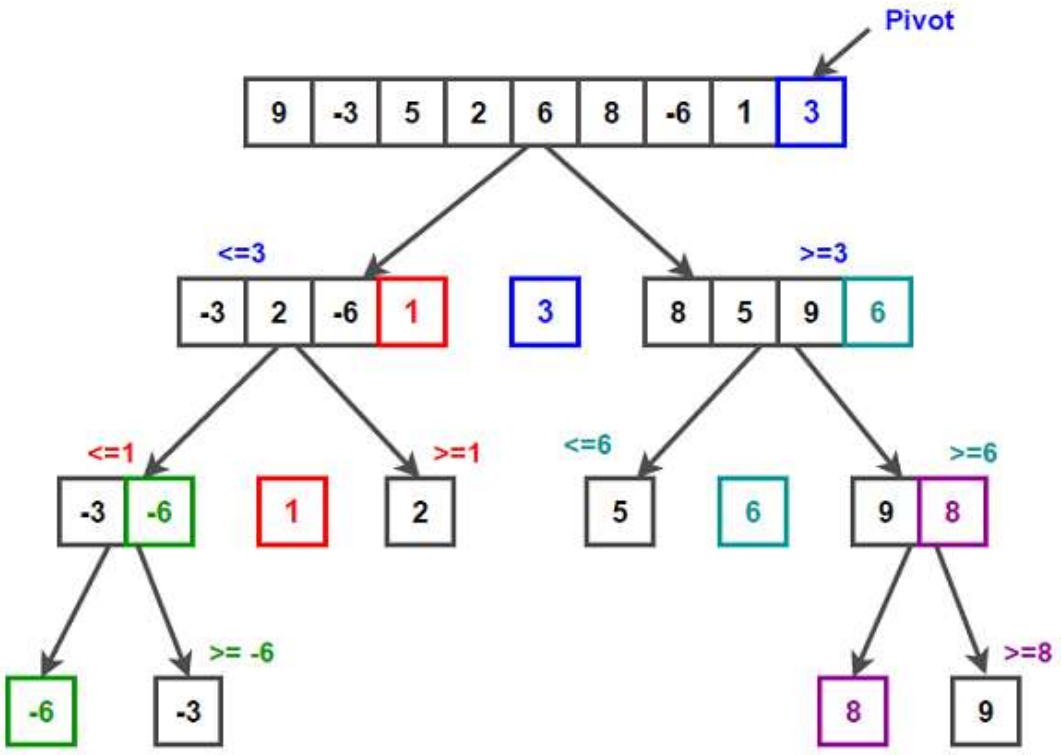

Рисунок 8.2 – Сортування за алгоритмом Хоара

Нижче наведено псевдокод для алгоритму Хоара та основної процедури сортування PARTITION.

```
QUICKSORT (A, p, r)
1. if p < r
2.   then q←PARTITION(A, p, r)
3.        QUICKSORT (A, p, q)
4.        QUICKSORT (A, q + 1, r)

PARTITION (A, p, r)
1. x ⇐ A[p]
2. i ← p − 1
3. j ← p − 1
4. while TRUE
5.     do repeat j ← j − 1
6.        until A[j] ≤ x
7.        repeat i ← i − 1
8.        until A[j] ≤ x
9.        if i < j
10.          then виконати заміну A[i] ↔ A[j]
11.          else return j
```

8.4 Контрольні питання

Поясніть значення терміна «асимптотична ефективність алгоритмів».
У чому різниця між Θ-, O- і Ω-позначеннями?
Які алгоритми сортування за квадратичний час Ви знаєте?
Як можна вдосконалили алгоритм сортування бульбашкою?
Поняніть значення терміна «сортування за псевдолінійний час».
Які алгоритми сортування за псевдолінійний час Ви знаєте?

9 ГРАФИ

9.1 Подання графів

Граф – це складна нелінійна багатозв'язна динамічна структура, що відображає властивості і зв'язки складного об'єкта.

Ця багатозв'язна структура має такі властивості:

- на кожний елемент (вузол, вершину) може бути довільна кількість посилань;
- кожний елемент може мати зв'язок з будь-якою кількістю інших елементів;
- кожний зв'язок (ребро, дуга) може мати напрям і вагу.

У вузлах графу міститься інформація про елементи об'єкта. Зв'язки між вузлами задаються ребрами графу. Ребра графу можуть мати спрямованість, тоді вони називаються орієнтованими, в іншому випадку – неорієнтовані. Граф, усі зв'язки якого орієнтовані, називається орієнтованим графом; граф зі всіма неорієнтованими зв'язками – неорієнтованим графом; граф із зв'язками обох типів – змішаним графом.

Існує два основні методи подання графів в пам'яті комп'ютера: матричний і зв'язними нелінійними списками. Вибір методу подання залежить від природи даних і операцій, що виконуються над ними. Якщо задача вимагає великої кількості внесень і вилучень вузлів, то доцільно подавати граф зв'язними списками; інакше можна застосувати і матричне подання.

У разі використання матриць суміжності їхні елементи подаються в пам'яті комп'ютера елементами масиву. За цих обставин матриця простого графу складається з нулів і одиниць; а мультиграфу – з нулів і цілих чисел, які вказують кратність відповідних ребер; зваженого графу – з нулів і дійсних чисел, які задають вагу кожного ребра.

Орієнтований граф подається зв'язним нелінійним списком, якщо він часто змінюється або якщо півміри входу і виходу його вузлів великі.

Графом називається сукупність двох множин: непорожньої множини V (множини вершин) та множини E неупорядкованих пар різних елементів множини V (E – множина ребер).

На відміну від дерев, графи можуть містити цикли та петлі (ребра, які з'єднують вершину саму з собою).

Для практичного застосування потрібно розробити подання графів у пам'яті комп'ютера, а також забезечити реалізацію таких дій:

- обхід графу;
- пошук довжин найкоротших шляхів від заданої вершини графу (джерела) до всіх інших вершин графу.

Є два стандартних способи подання графу G = (V, E): як набору списків суміжних вершин або як матриці суміжності. Обидва способи подання застосовні як орієнтованих, так неорієнтованих графів. Зазвичай краще по-

дання з допомогою списків суміжності, оскільки воно забезпечує компактне подання розріджених (sparse) графів, тобто таких, у яких |E| набагато менша $|V|^2$.

Більшість алгоритмів припускає, що вхідний граф подається у вигляді списку суміжності. Подання за допомогою матриці суміжності краще у разі щільних (dense) графів, тобто коли значення |E| близько до $|V|^2$, або коли нам треба мати можливість швидко визначити, чи існує ребро, що з'єднує дві ці вершини. Подання графу G = (V, E) як списку суміжності (adjacency-list representation) використовує масив Adj з |V| списків, по одному для кожної вершини з V. Для кожної вершини u ∈ V список Adj[u] містить усі вершини v такі, що (u, v) ∈ E, тобто Adj [u] складається з усіх вершин, суміжних з u у графі G (список може містити не самі вершини, а покажчики на них). Вершини у кожному списку зазвичай зберігаються у довільному порядку. На рис. 9.1, б) показано таке подання графу, зображеного на рис. 9.1, а) (на рис. 9.1, в) наведено його матрицю суміжності). Аналогічно, на рис. 9.2, б) та 9.2, в) показано список і матрицю суміжності орієнтованого графу, зображеного на рис. 9.2, а).

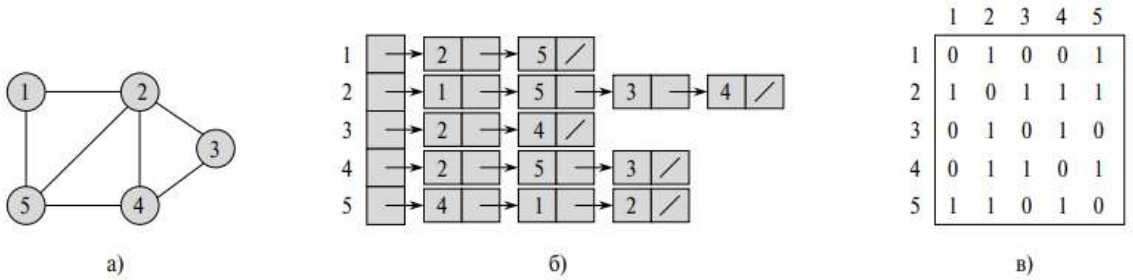

Рисунок 9.1 – Два подання неорієнтованого графу

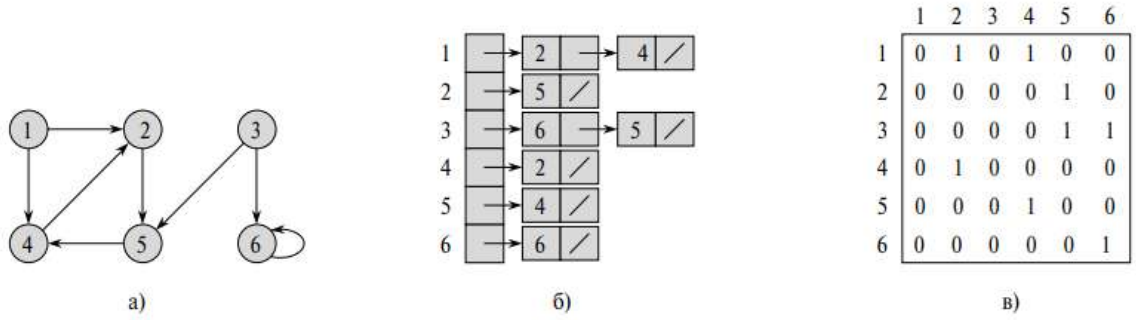

Рисунок 9.2 – Два подання орієнтованого графу

Якщо G – орієнтований граф, то сума довжин всіх списків суміжності дорівнює |E|, оскільки ребру (u, v) однозначно відповідає елемент v списку Adj[u]. Якщо G – неорієнтований граф, то сума довжин усіх списків суміжності дорівнює 2 |E|, оскільки ребро (u, v), бувши неорієнтованим, з'являється у списку Adj[v] як u, та у списку Adj[u] – як v. Як для орієнтованих, так і для неорієнтованих графів подання у вигляді списків вимагає обсягу пам'яті, що дорівнює Θ(V + E).

Списки суміжності легко адаптуються для подання зважених графів (weighted graph), тобто графів, з кожним ребром яких пов'язана певна вага (weight), зазвичай визначається ваговою функцією (weight function) w: E→R.

Наприклад, нехай G = (V, E) – зважений граф з ваговою функцією w. Вага w(u, v) ребра (u, v) ∈ E просто зберігається разом з вершиною v у списку суміжності u. Подання за допомогою списків суміжності досить гнучко у тому сенсі, що легко адаптується для підтримки багатьох інших варіантів графів.

Потенційний недолік подання за допомогою списків суміжності полягає в тому, що за таких обставин немає швидшого способу визначити, чи є це ребро (u, v) у графі, ніж пошук v у списку Adj [u]. Цей недолік можна усунути ціною використання асимптотично більшої кількості пам'яті та подання графа за допомогою матриці суміжності.

Подання графу G = (V, E) за допомогою матриці суміжності (adjacency-matrix representation) передбачає, що вершини перенумеровані в деякому порядку числами 1, 2, . . . , | V |. У цьому випадку подання графу G з використанням матриці суміжності є матрицею A = (a_{ij}) розміром |V | × |V | такою, що

$$a_{ij} = \begin{cases} 1, \text{якщо } (i,j) \in E \\ 0 \text{ в протилежному випадку.} \end{cases}$$

На рис. 9.1, в) і 9.2, в) показано подання з використанням матриці суміжності неорієнтованого та орієнтованого графів, наведених на рис. 9.1, а) та 9.2, а), відповідно.Матриця суміжності графу потребує обсяг пам'яті, що дорівнює Θ (V^2), незалежно від кількості ребер графу.

Хоча список суміжності асимптотично, як мінімум, настільки ж ефективний в плані пам'яті, як і матриця суміжності, простота останньої робить її кращою при роботі з невеликими графами. Крім того, у випадку незважених графів для подання одного ребра достатньо одного біта, що дозволяє суттєво заощадити потрібну для подання пам'ять.

9.2 Пошук в ширину

Пошук в ширину (breadth-first search, BFS) є одним із найпростіших алгоритмів для обходу графу і є основою багатьох важливих алгоритмів для роботи з графами. Наприклад, алгоритм Прима (Prim) пошуку мінімального кістяка або алгоритм Дейкстри (Dijkstra) пошуку найкоротшого шляху з однієї вершини використовують ідеї, подібні до ідей, що використовуються при пошуку в ширину.

Нехай заданий граф G = (V, E) та виділена вихідна (source) вершина s. Алгоритм пошуку в ширину систематично обходить усі ребра G для «відкриття» всіх вершин, досяжних з s, обчислюючи при цьому відстань (мінімальна кількість ребер) від s до кожної досяжної s вершини. Крім того, в

процесі обходу будується «дерево пошуку в ширину» з коренем s, що містить усі досяжні вершини. Для кожної досяжної з s вершини v шлях у дереві пошуку в ширину відповідає найкоротшому (тобто, містить найменшу кількість ребер) шляху від s до v G. Алгоритм працює як для орієнтованих, так і для неорієнтованих графів.

Пошук в ширину має таку назву тому, що в процесі обходу ми йдемо вшир, тобто перед тим як розпочати пошук вершин на відстані k + 1, виконується обхід всіх вершин з відстанню k.

Для відстеження роботи алгоритму пошук в ширину розфарбовує вершини графу в білий, сірий і чорний кольори. Спочатку всі вершини білі, і вони можуть стати сірими, а потім чорними. Коли вершина відкривається (discovered) у процесі пошуку, вона фарбується. Таким чином, сірі та чорні вершини – це вершини, які вже були відкриті, але алгоритм пошуку в ширину по-різному працює з ними, щоб забезпечити оголошений порядок обходу. Якщо (u, v) ∈ E і вершина u чорного кольору, то вершина v або сіра, або чорна, тобто всі вершини, суміжні із чорною, вже відкриті. Сірі вершини можуть мати білих сусідів, являючи собою межу між відкритими та невідкритими вершинами.

Пошук в ширину будує дерево пошуку в ширину, яке спочатку складається з одного кореня, яким є вихідна вершина s. Якщо в процесі сканування списку суміжності відкритої вершини u відкривається біла вершина v, то вершина v і ребро (u, v) додаються в дерево. Ми говоримо, що u є попередником (predecessor), або батьком (parent), v у дереві пошуку. Оскільки вершина може бути відкрита не більше одного разу, вона має не більше одного батька. Взаємини предків і нащадків визначаються в дереві пошуку в ширину як завжди, а саме: якщо u перебуває в шляху від кореня s до вершини v, то u є предком v, а v – нащадком u.

Нижче наведено процедуру пошуку в ширину BFS, яка передбачає, що вхідний граф G = (V, E) поданий за допомогою списків суміжності. Крім того, підтримуються додаткові структури даних у кожній вершині графу. Колір кожної вершини u ∈ V зберігається в змінній color [u], а попередник – у змінній π [u]. Якщо попередника у u немає (наприклад, якщо u = s чи u не відкрита), то π [u] = NIL. Відстань від s до вершини u, що обчислюється алгоритмом, зберігається в полі d[u]. Алгоритм використовує чергу Q для роботи з множиною сірих вершин.

```
BFS(G, s)
1 for (для) кожної вершини u ∈ V [G] − s
2     do color [u] ← WHITE
3           d[u] ← ∞
4           п[u] ← NIL
5     color [s] ← GRAY
6 d[s] ← 0
7 п[s] ← NIL
```

```
8 Q ← ∅
9 ENQUEUE(Q, s)
10 while Q ≠ ∅
11      do u ← DEQUEUE(Q)
12           for (для) кожної v ∈ Adj[u]
13                do if color [v] = WHITE
14                     then  color [v] ← GRAY
15                           d[v] ← d[u] + 1
16                           п[v] ← u
17                           ENQUEUE(Q, v)
18           color [u] ← BLACK
```

Для кращого розуміння роботи BFS проілюструємо його роботу покроково (рис. 9.3–9.9) [19].

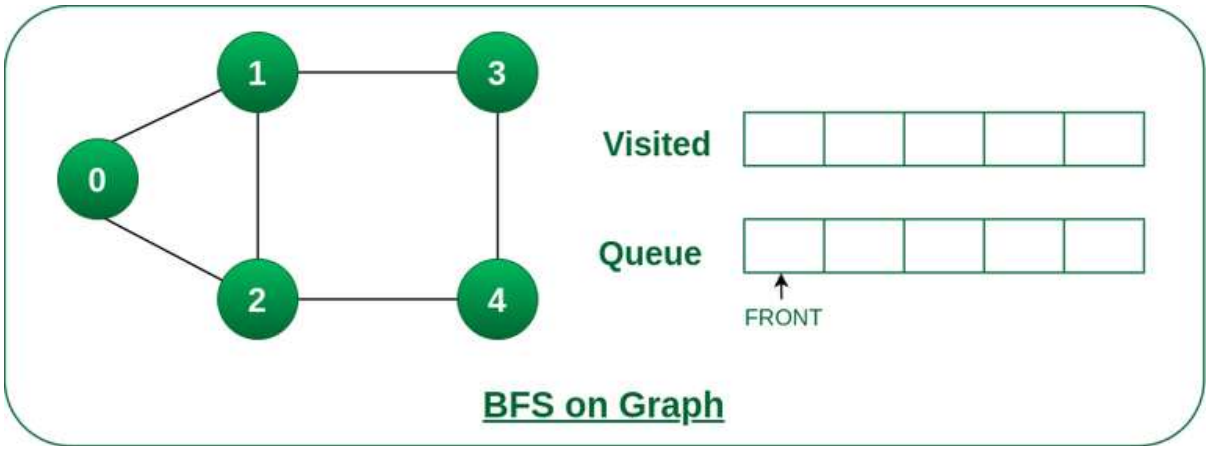

Рисунок 9.3 – Крок 1: черга та масив відвіданих вершин спочатку порожні

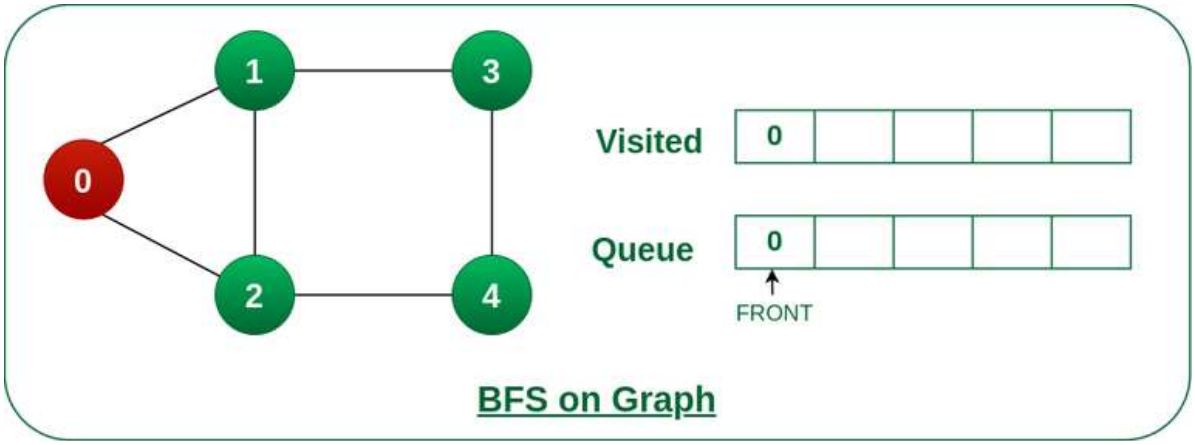

Рисунок 9.4 – Крок 2: занести вершину 0 в чергу та відмітити її як відвідану

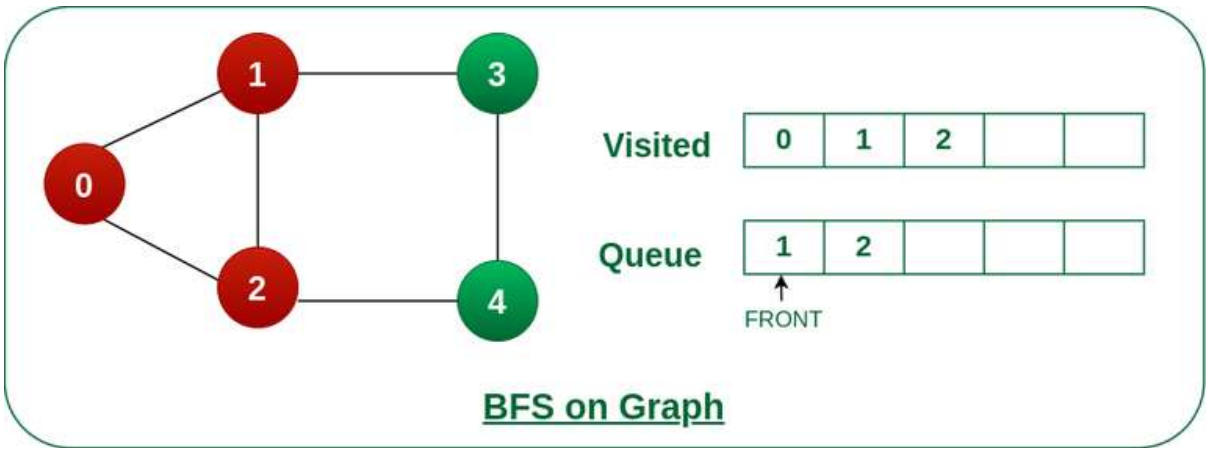

Рисунок 9.5 – Крок 3: видалити вершину 0 з черги;
її невідвіданих сусідів 1 і 2 занести в чергу і відмітити як відвіданих

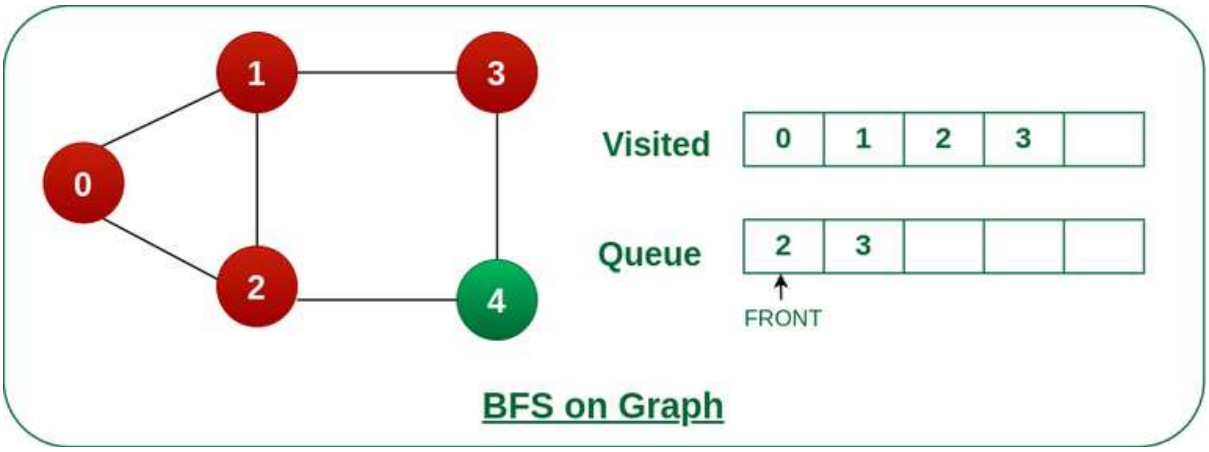

Рисунок 9.6 – Крок 4: видалити вершину 1 з черги;
її невідвіданого сусіда 3 занести в чергу і відмітити як відвіданого

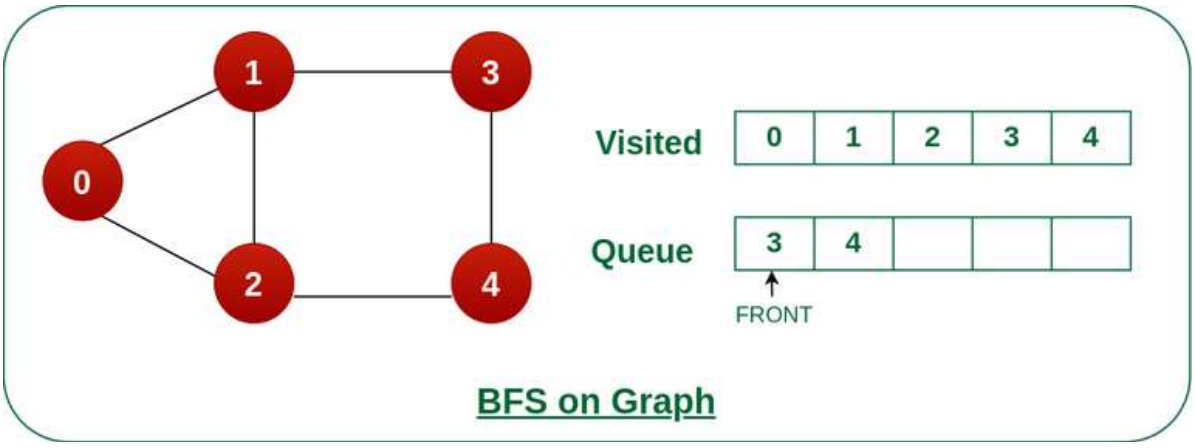

Рисунок 9.7 – Крок 5: видалити вершину 2 з черги;
її невідвіданого сусіда 4 занести в чергу і відмітити як відвіданого

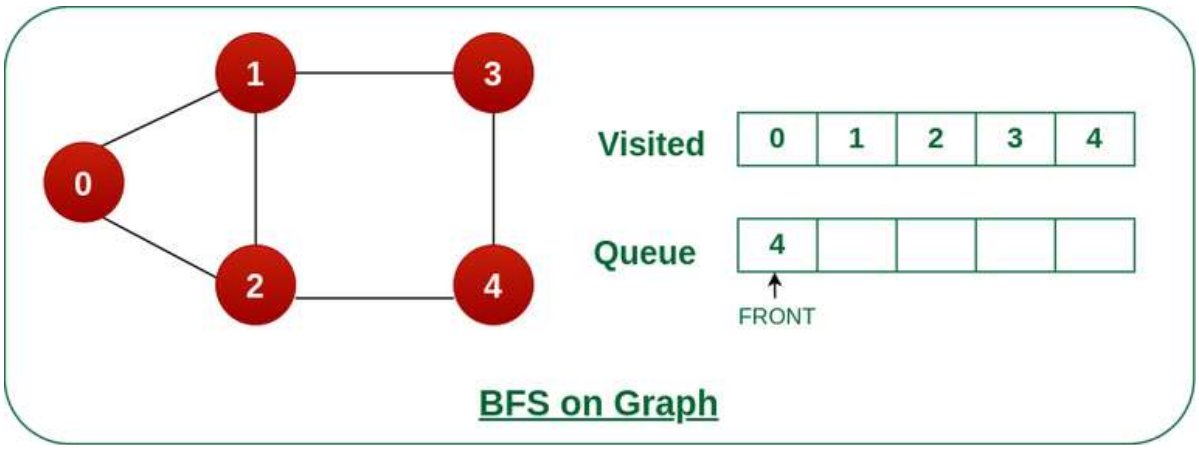

Рисунок 9.8 – Крок 6: видалити вершину 3 з черги; оскільки у неї немає невідвіданих сусідів, у чергу нічого не заноситься

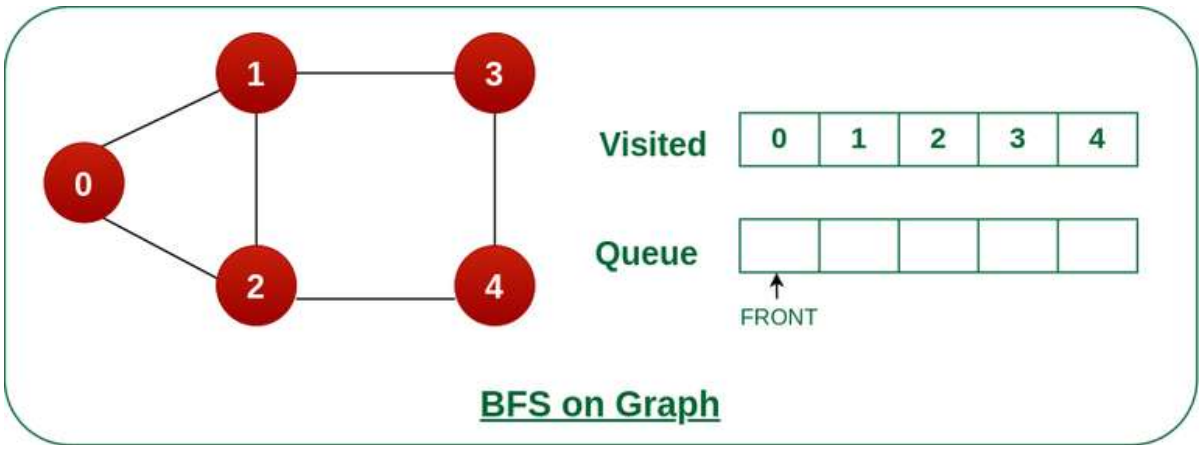

Рисунок 9.9 – Крок 7: видалити вершину 4 з черги; оскільки у неї немає невідвіданих сусідів, у чергу нічого не заноситься

Тепер черга порожня, отже, ітераційний процес завершено.

9.3 Пошук в глибину

Стратегія пошуку в глибину (depth first search, DFS), як випливає з її назви, полягає в тому, щоб йти «вглиб» графу, наскільки це можливо. Під час пошуку в глибину досліджуються всі ребра, що виходять з вершини, відкритої останньою, і залишає вершину, тільки коли не залишається недосліджених ребер – водночас відбувається повернення у вершину, з якої була відкрита вершина v. Цей процес триває доти, доки не будуть відкриті всі вершини, досяжні з вихідної. Якщо при цьому залишаються невідкриті вершини, то одна з них вибирається як нова вихідна вершина, і пошук повторюється вже з неї. Цей процес повторюється доти, доки не будуть відкриті всі вершини.

Як і у разі пошуку в ширину, коли вершина v відкривається в процесі сканування списку суміжності вже відкритої вершини u, процедура пошу-

ку записує цю подію, встановлюючи поле попередника v π[v] рівним u. На відміну від пошуку в ширину, де підграф передування утворює дерево, у разі пошуку в глибину підграф передування може складатися з кількох дерев, оскільки пошук може виконуватися з кількох вихідних вершин.

Підграф передування (predecessor subgraph) пошуку в глибину, таким чином, дещо відрізняється від такого для пошуку в ширину. Ми визначаємо його як граф

$$G\pi = (V, E\pi),$$

де $E\pi = \{(\pi [v] , v) : v \in V \text{ і } \pi [v] \neq NIL\}$.

Підграф передування пошуку в глибину утворює ліс пошуку в глибину (depth first forest), який складається з кількох дерев пошуку в глибину (depth-first trees). Ребра в Eπ називаються ребрами дерева (tree edges).

Як і в процесі пошуку в ширину, вершини графу розфарбовуються в різні кольори, що свідчать про їх стан. Кожна вершина спочатку біла, потім, під час відкриття (discover) у процесі пошуку вона забарвлюється в сірий колір, і після завершення (finish), коли її список суміжності повністю сканований, вона стає чорною. Така методика гарантує, що кожна вершина зрештою знаходиться тільки в одному дереві пошуку в глибину, так що дерева не перетинаються.

Крім побудови лісу пошуку у глибину, пошук у глибину також проставляє у вершинах мітки часу (timestamp). Кожна вершина має дві такі мітки - першу d[v], в якій вказується, коли вершина v відкривається (і забарвлюється в сірий колір), і друга – f[v], яка фіксує момент, коли пошук завершує сканування списку суміжності вершини v і вона стає чорною. Ці мітки використовуються багатьма алгоритмами та корисні при розгляді поведінки пошуку у глибину.

Наведена нижче процедура DFS записує в полі d[u] момент, коли вершина u відкривається, а в полі f[u] – момент завершення роботи з вершиною u. Ці мітки часу є цілими числами в діапазоні від 1 до 2|V|, оскільки у кожної з |V| вершин є тільки одна подія відкриття і одна – завершення. Для кожної вершини u

$$d [u] < f [u].$$

До моменту часу d[u] вершина має колір WHITE, між d[u] та f[u] – колір GRAY, а після f [u] – колір BLACK.

Далі наведено псевдокод алгоритму пошуку в глибину. Вхідний граф G може бути як орієнтованим, так і неорієнтованим. Змінна time глобальна та використовується для міток часу.

```
DFS(G)
1     for (для) кожної вершини ∈ V [G]
2           do color [u] ← WHITE
3                 π[u] ← NIL
4     time ← 0
```

```
5       for (для) кожної вершини u ∈ V [G]
6               do if color [u] = WHITE
7                       then DFS_VISIT(u)

DFS_VISIT(u)
1       color [u] ← GRAY                        // Відкрито білу вершину u
2       time ← time +1
3       d[u] ← time
4       for (для) кожної вершини v ∈ Adj[u] // Дослідження ребра (u, v).
5               do if color [v] = WHITE
6                       then π[v] ← u
7                               DFS_VISIT(v)
8       color [u] ← BLACK                       // Завершення
9       f[u] ← time ← time +1
```

Для кращого розуміння роботи DFS проілюструємо його роботу покроково (рис. 9.10–9.15) [20].

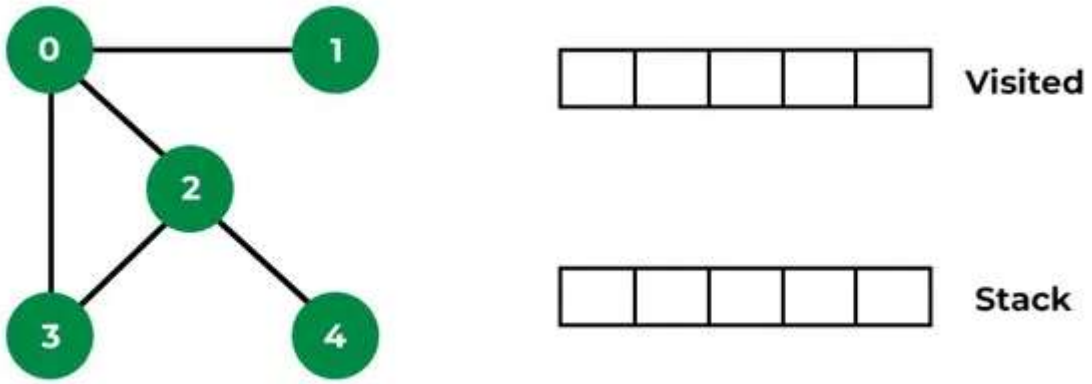

Рисунок 9.10 – Крок 1: спочатку стек і масив відвіданих вершин порожні

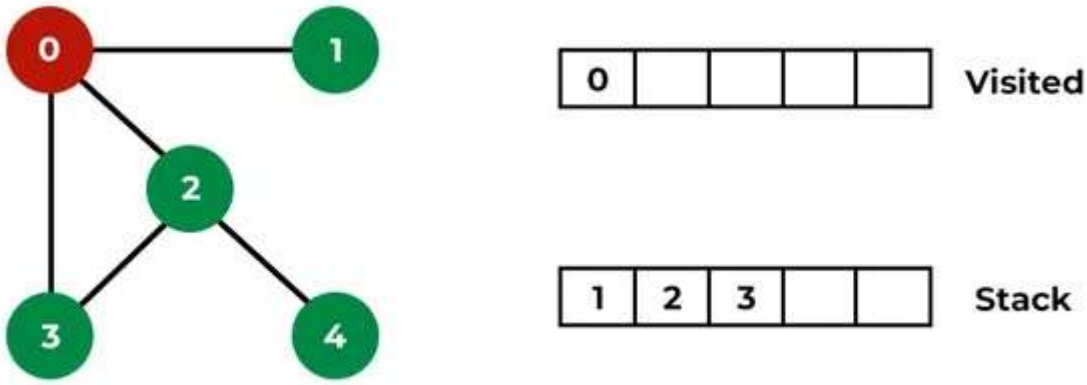

Рисунок 9.11 – Крок 2: відвідати вершину 0 і занести в стек невідвіданих сусідів

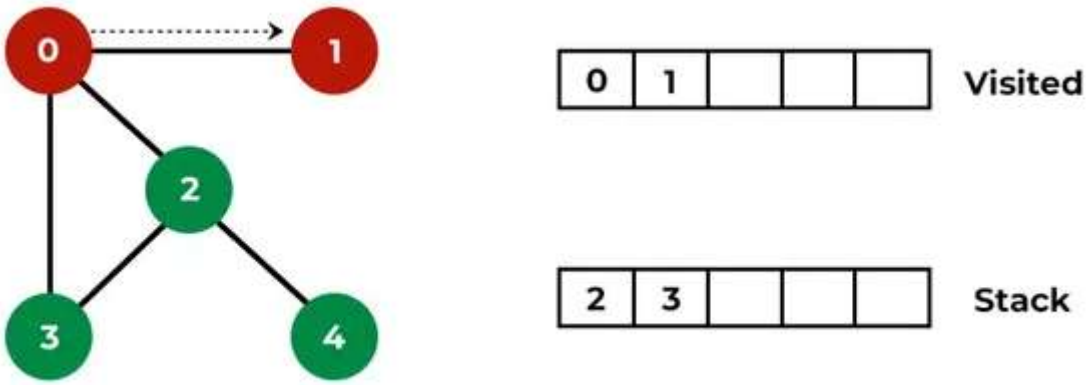

Рисунок 9.12 – Крок 3: виштовхнути зі стека вершину 1 і помітити її як відвідану; занести в стек її невідвіданих сусідів (яких немає)

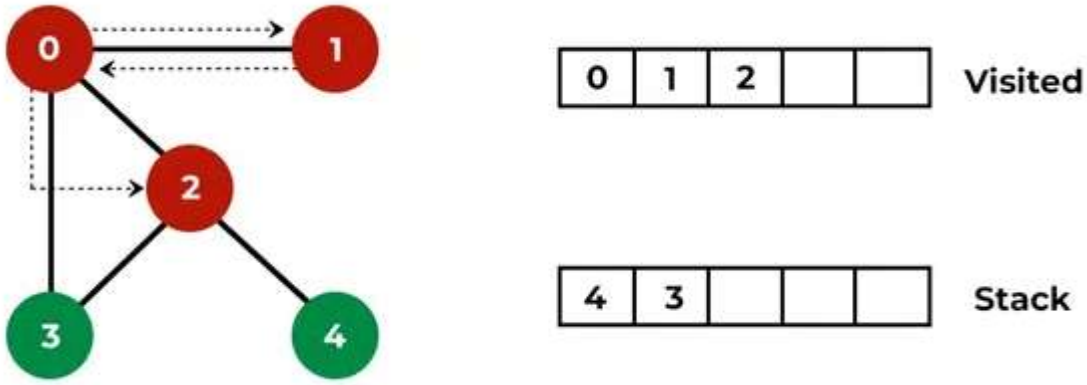

Рисунок 9.13 – Крок 4: виштовхнути зі стека вершину 2 і помітити її як відвідану; занести в стек її невідвіданих сусідів (тобто 4)

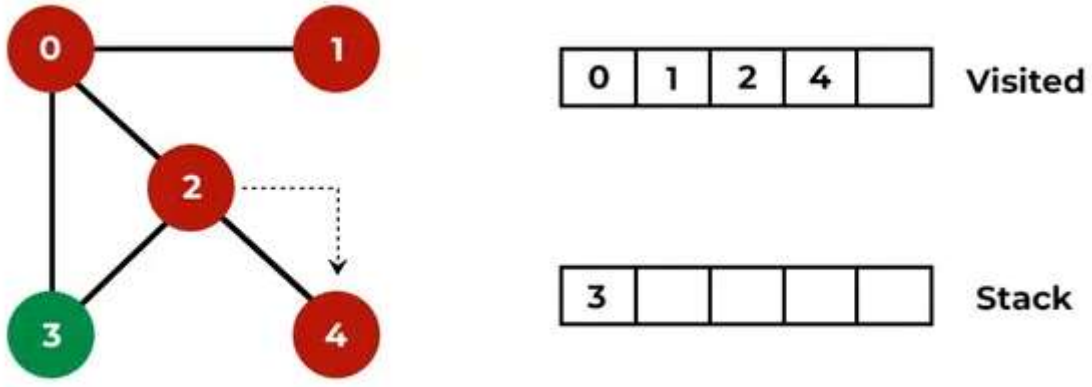

Рисунок 9.14 – Крок 5: виштовхнути зі стека вершину 4 і помітити її як відвідану; занести в стек її невідвіданих сусідів (яких немає)

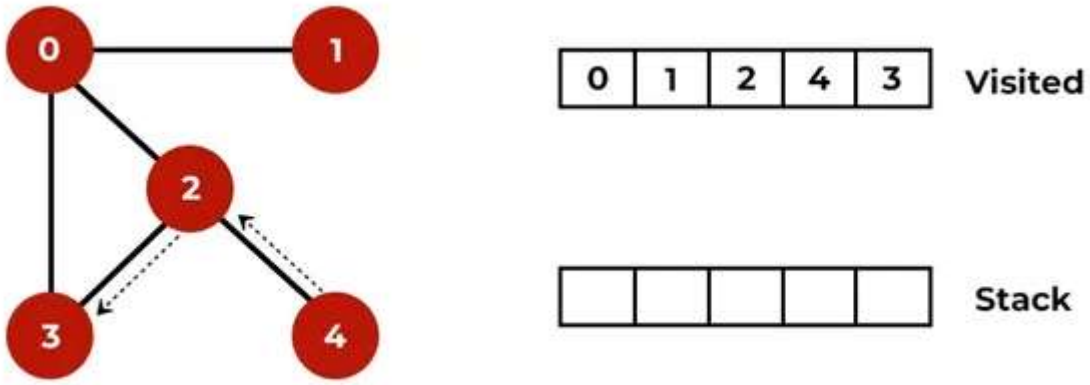

Рисунок 9.15 – Крок 6: виштовхнути зі стека вершину 3 і помітити її як відвідану; занести в стек її невідвіданих сусідів (яких немає)

Тепер стек порожній, отже, ітераційний процес завершено.

9.4 Вибір між BFS та DFS

Важливе практичне питання, яке нас цікавить, полягає в тому, як зробити вибір між DFS і BFS.

Якщо ми знаємо, що цільова вершина розташована глибоко, ми маємо використовувати DFS, оскільки він досягає глибоких вузлів швидше, ніж BFS. Хорошим прикладом є задоволення обмежень. У цих задачах ми маємо призначити значення кільком змінним таким чином, щоб обмеження на змінні та їхні значення були задоволені. Кожен вузол у отриманому графі пошуку є списком призначень. Оскільки ми зацікавлені в повних рішеннях, має більше сенсу досягти повних вузлів призначення якомога швидше за допомогою DFS, ніж повільно шукати неповні призначення за допомогою BFS.

З іншого боку, якщо ми знаємо, що ціль може відображатися на низькому рівні в дереві пошуку, BFS є кращим вибором. Те саме стосується випадків, коли ми не можемо прийняти неоптимальний шлях і готові витрачати більше пам’яті, щоб отримати теоретичні гарантії. Але застереження полягає в тому, що BFS може вимагати багато пам’яті та виявиться неефективним, якщо коефіцієнт розгалуження занадто великий або ціль знаходиться глибоко.

Саме витрати пам’яті роблять BFS непрактичним, навіть якщо він є повним і оптимальним у випадках, коли DFS не є таким. З цієї причини DFS став «робочою конячкою» у системах штучного інтелекту. Ми навіть можемо допустити проблеми його незавершеності та неоптимальних рішень. Ітераційне поглиблення, яке обмежує глибину DFS і запускає його з поступовими глибинами обмеження, є повним, оптимальним і має таку ж часову складність, як і BFS. Однак його витрати пам’яті такі ж, як і у DFS.

9.5 Контрольні питання

Дайте означення графу.
Як називаються і як зображуються основні елементи графу?
Які різновиди графів Ви знаєте?
Як подаються графи у пам’яті комп’ютера?
Які структури даних використовуються в процесі пошуку в ширину?
Які структури даних використовуються в процесі пошуку в глибину?
В яких випадках доцільно використовувати пошук в ширину?
В яких випадках доцільно використовувати пошук в глибину?

10 АЛГОРИТМИ ПОШУКУ НАЙКОРОТШОГО ШЛЯХУ У ГРАФІ

10.1 Алгоритм Дейкстри

Алгоритм нідерландського вченого Едсгера Дейкстри знаходить всі найкоротші шляхи з однієї наперед заданої вершини графу до всіх інших. З його допомогою, за наявності всієї необхідної інформації, можна, наприклад, дізнатися яку послідовність доріг краще використовувати, щоб дістатися з одного міста до кожного з багатьох інших, або в які країни вигідніше експортувати нафту тощо.

Мінусом даного підходу є неможливість обробки графів, в яких є ребра з від'ємною вагою, тобто, якщо, наприклад, деяка система передбачає збиткові для Вашої фірми маршрути, то для роботи з нею бажано скористатися відмінним від алгоритму Дейкстри методом.

Для програмної реалізації алгоритму використовуємо два масиви: логічний visited – для зберігання інформації про відвідані вершини та числовий distance, в який будуть заноситися знайдені найкоротші шляхи. Отже, є граф G = (V, E). Кожна з вершин, що входять в множину n, спочатку позначена як невідвідана, тобто елементам масиву visited присвоєно значення false.

Оскільки найвигідніші шляхи тільки належить знайти, в кожен елемент вектора distance записується таке число, яке точно більше будь-якого потенційного шляху (зазвичай, це число називають нескінченністю, але в програмі використовують, наприклад, максимальне значення конкретного типу даних). За вихідний пункт вибирається вершина s, і їй приписується нульовий шлях: distance [s] = 0 , тобто, немає ребра з s в s (метод не передбачає петель).

Далі, знаходяться всі сусідні вершини (в які є ребро з s) (нехай такими будуть t і u) і по черзі досліджуються, а саме: обчислюється вартість маршруту з s в кожну з них:

distance [t] = distance [s] + вага_інцидентного s і t ребра;

distance [u] = distance [s] + вага_інцидентного s і u ребра.

Але цілком ймовірно, що в ту чи іншу вершину з s існує кілька шляхів, тому вартість шляху в таку вершину в масиві distance доведеться переглядати, тоді найбільше (неоптимальне) значення ігнорується, а найменше ставиться у відповідність вершині.

Після обробки суміжних з s вершин вона позначається як відвідана: visited [s] = true, і активною стає та вершина, шлях з s в яку мінімальний. Припустимо, шлях з s в u коротший, ніж з s в t, отже, вершина u стане активною і вищеописаним способом досліджуються її сусіди, за винятком вершини s. Далі, u позначається як пройдена: visited [u] = true, активною стає вершина t, і вся процедура повторюється для неї. Алгоритм Дейкстри триває доти, поки всі доступні з s вершини не будуть досліджені.

Тепер на конкретному графі простежимо роботу алгоритму, знайдемо всі найкоротші шляхи між початковою та всіма іншими вершинами. Розмір (кількість ребер) зображеного на рис. 10.1 графу дорівнює 7 ($|E| = 7$) , а порядок (кількість вершин) – 6 ($|n| = 6$) . Це зважений граф, кожному з його ребер поставлено у відповідність деяке числове значення, тому цінність маршруту необов'язково визначається числом ребер, що лежать між парою вершин.

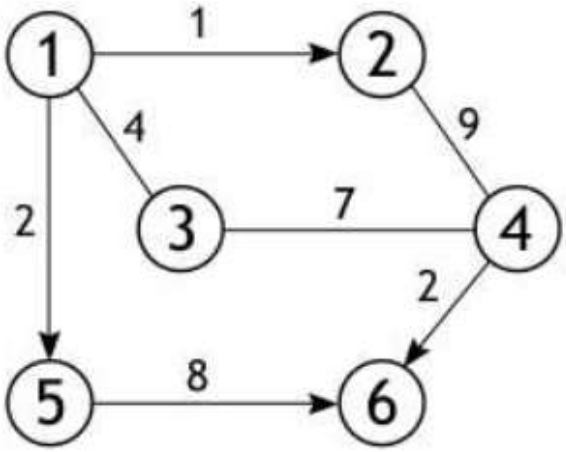

Рисунок 10.1 – Початковий граф для пошуку найкоротшого шляху за алгоритмом Дейкстри

З усіх вершин, що належать до множини n, виберемо одну, ту, від якої потрібно знайти найкоротші шляхи до інших доступних вершин. Нехай це буде вершина 1. Довжина шляху до всіх вершин, крім першої, спочатку дорівнює нескінченності, а до неї – 0, тобто, граф не має петель.

У вершини 1 рівно 3 сусідніх (вершини 2, 3, 5), і щоб обчислити довжину шляху до них потрібно додати вагу дуг, що лежать між вершинами: 1 і 2, 1 і 3, 1 і 5 до значення першої вершини (з нулем):

$2 \leftarrow 1 + 0$

$3 \leftarrow 4 + 0$

$5 \leftarrow 2 + 0$

Як уже зазначалося, отримані значення присвоюються вершинам лише в тому випадку, якщо вони «кращі» (мають менше значення), ніж інші на даний момент. А оскільки кожне з трьох чисел менше нескінченності, вони стають новими величинами, що визначають довжину шляху з вершини 1 до вершин 2, 3 і 5 (рис. 10.2).

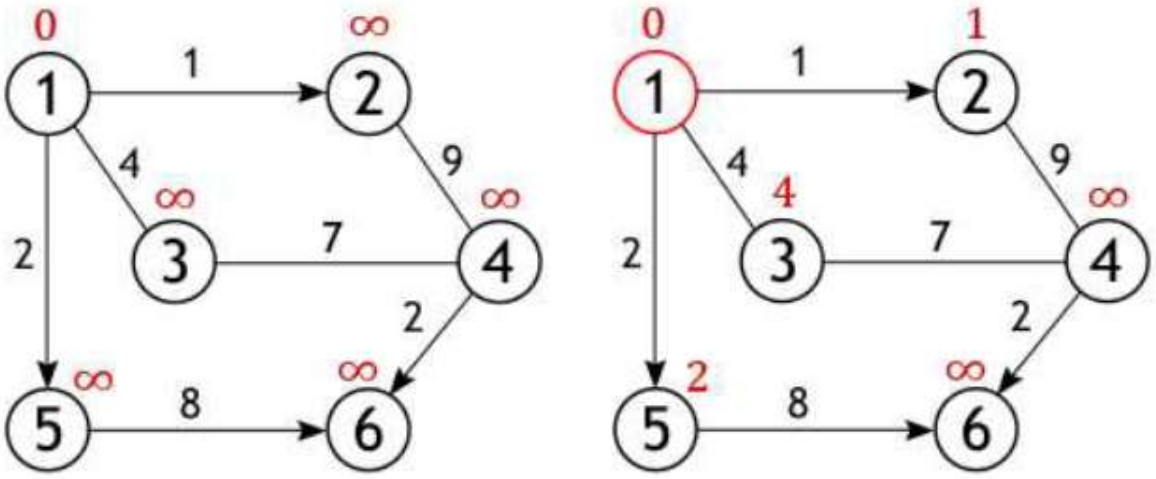

Рисунок 10.2 – Алгоритм Дейкстри: обробка вершини 1

Далі, активна вершина позначається як відвідана, статус «активної» (червоне коло) переходить до однієї з її сусідніх, а саме: до вершини 2, оскільки вона найближча до раніше активної вершини.

У вершини 2 всього один нерозглянутий сусід (вершина 1 позначена як відвідана), відстань до якого з неї дорівнює 9, але нам потрібно обчислити довжину шляху з початкової вершини, для чого потрібно додати величину, розраховану для вершини 2, до ваги дуги з неї до вершини 4

$$4 \leftarrow 1 + 9.$$

Умова ($10 < \infty$) виконується, отже, вершина 4 отримує нове значення довжини шляху (рис. 10.3).

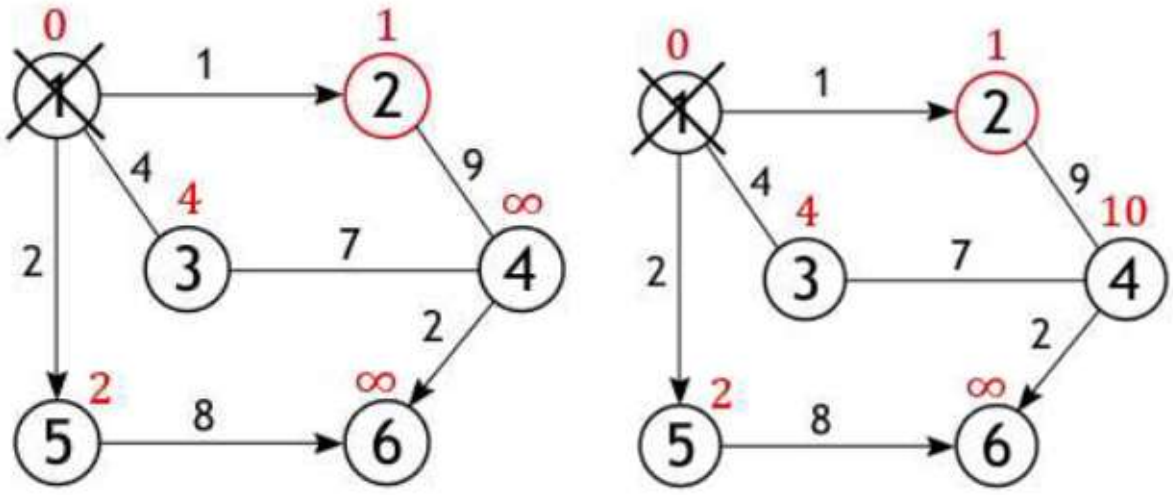

Рисунок 10.3 – Алгоритм Дейкстри: обробка вершини 2

Вершина 2 перестає бути активною так, як і вершина 1, і видаляється зі списку невідвіданих. Тепер тим самим способом досліджуються сусіди вершини 5, і обчислюються відстані до них (рис. 10.4).

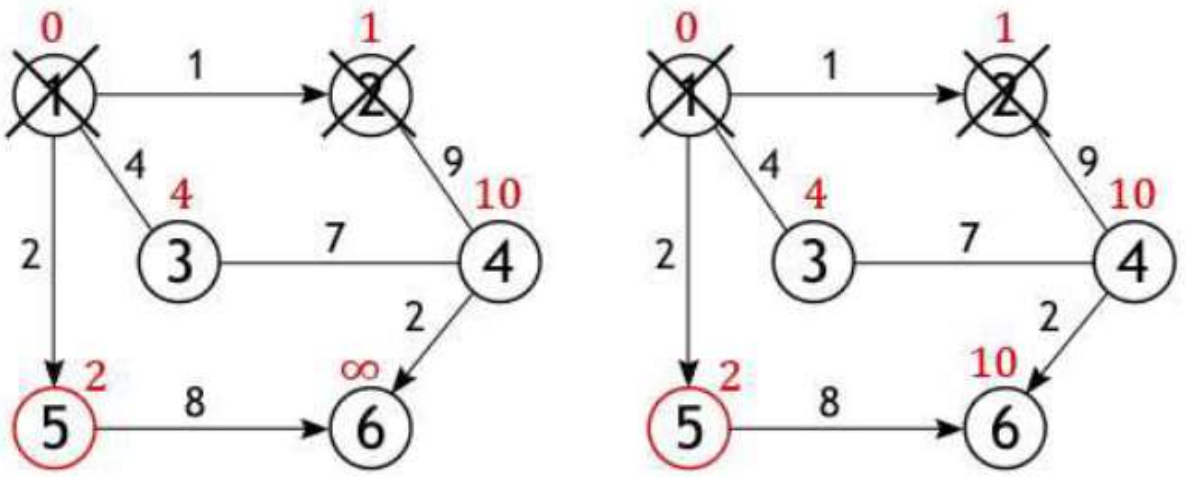

Рисунок 10.4 – Алгоритм Дейкстри: обробка вершини 5

Коли справа доходить до огляду сусідів вершини 3, то тут важливо не помилитися, оскільки вершину 4 вже була досліджено і довжина одного з можливих шляхів з початкової точки до неї обчислено. Якщо рухатися до неї через вершину 3, то шлях становитиме 4 + 7 = 11, а 11 > 10, тому нове значення ігнорується, старе залишається (рис. 10.5).

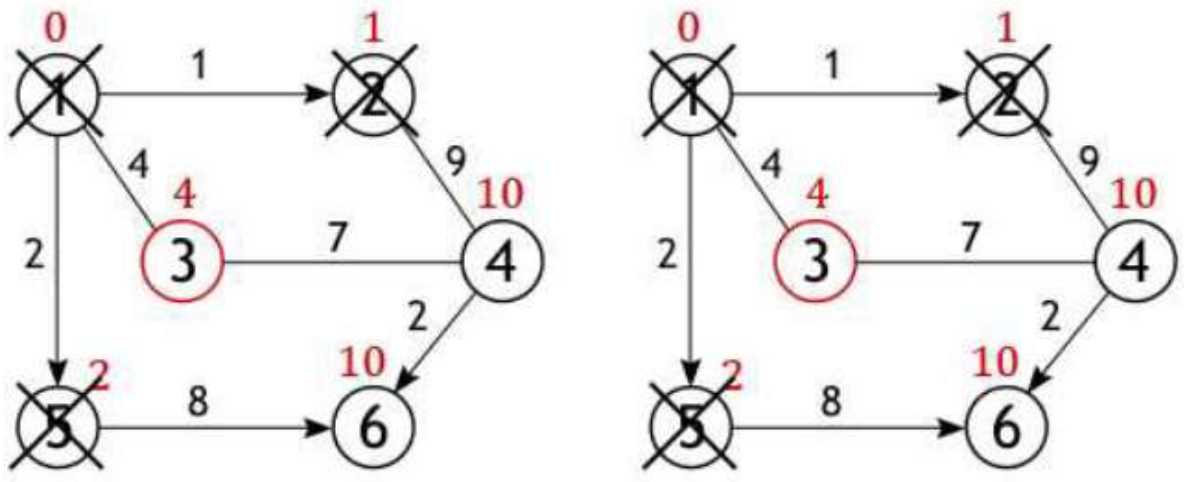

Рисунок 10.5 – Алгоритм Дейкстри: обробка вершини 3

Аналогічна ситуація з вершиною 6. Значення найближчого шляху до неї з вершини 1 дорівнює 10, а виходить це тільки в тому випадку, якщо йти через вершину 5 (рис. 10.6).

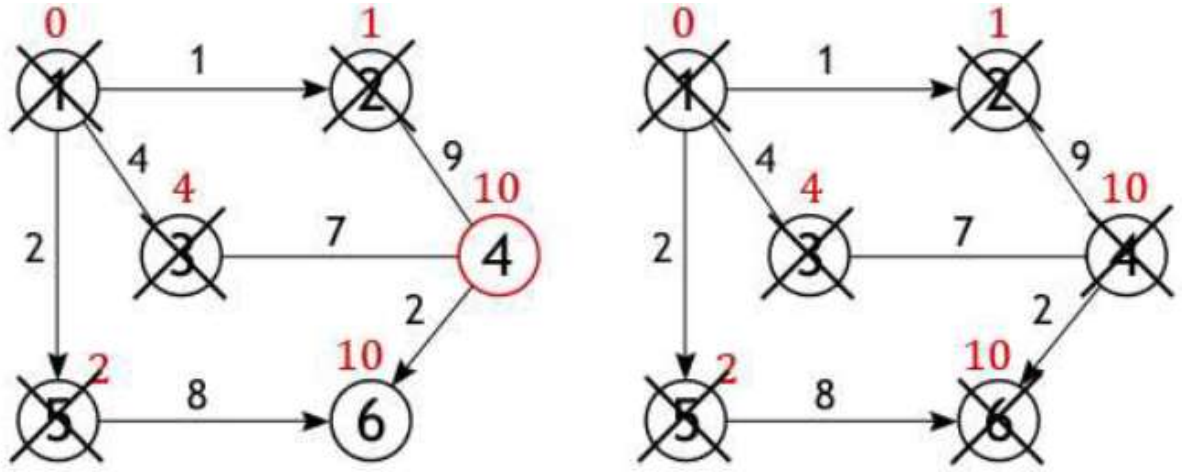

Рисунок 10.6 – Алгоритм Дейкстри: обробка вершин 4 і 6

Коли всі вершини графу, або всі ті, що доступні з початкової точки, будуть позначені як відвідані, тоді робота алгоритму Дейкстри завершиться, і всі знайдені шляхи будуть найкоротшими. Так, наприклад, буде виглядати список найоптимальніших відстаней для шляхів, що лежать між вершиною 1 і всіма іншими вершинами розглянутого графу:

1 → 1 = 0
1 → 2 = 1
1 → 3 = 4
1 → 4 = 10
1 → 5 = 2
1 → 6 = 10

10.2 Алгоритм Беллмана-Форда

Завдання: Дано граф і початкова вершина src у графі. Потрібно знайти найкоротші шляхи від src до всіх вершин у цьому графі. У графі можуть бути ребра з від'ємними вагами.

Ми вже обговорювали алгоритм Дейкстри як спосіб вирішення цього завдання. Алгоритм Дейкстри є «жадібним» алгоритмом, складність якого дорівнює O(VlogV) (з використанням купи Фібоначчі). Однак алгоритм Дейкстри не працює для графів з від'ємними вагами ребер, тоді як алгоритм Беллмана-Форда припускає існування у графі таких ребер. Алгоритм Беллмана-Форда навіть простіший, ніж алгоритм Дейкстри, і добре працює у розподілених системах. У той самий час складність його дорівнює O(VE), що більше, ніж показник алгоритму Дейкстри.

Алгоритм
Нижче наведено докладно розписані кроки алгоритму.
Вхідні дані: граф та початкова вершина src.

Вихідні дані: найкоротша відстань до всіх вершин src. Якщо трапляється цикл від'ємної ваги, то короткі відстані не обчислюються, виводиться повідомлення про наявність такого циклу.

1. На цьому кроці ініціалізуються відстані від вихідної вершини до решти вершин, як нескінченні, а відстань до самого src приймається рівною 0. Створюється масив dist [] розміру |V| з усіма значеннями, рівними нескінченності, крім елемента dist[src], де src – вихідна вершина.

2. Другим кроком обчислюються найкоротші відстані. Наступні кроки потрібно виконувати |V|-1 раз, де |V| – число вершин в даному графі. Виконати таку дію для кожного ребра u-v:

якщо dist[v] > dist[u] + вага ребра uv, то оновити dist[v]:
dist [v] = dist [u] + вага ребра uv

На цьому кроці повідомляється, чи є у графі цикл від'ємної ваги. Для кожного ребра u-v потрібно виконати таке:

якщо dist[v] > dist[u] + вага ребра uv,
то у графі присутній цикл від'ємної ваги.

3. Ідея кроку 3 полягає в тому, що крок 2 гарантує якнайкоротшу відстань, якщо граф не містить циклу від'ємної ваги. Якщо ми знову переберемо всі ребра і отримаємо коротший шлях для будь-якої з вершин, це буде сигналом присутності циклу від'ємної ваги.

Як це працює? Як і в інших завданнях динамічного програмування, алгоритм обчислює найкоротші шляхи знизу вгору. Спочатку він обчислює найкоротші відстані, тобто шляхи довжиною не більше, ніж одне ребро. Потім він обчислює найкоротші шляхи завдовжки не більше двох ребер тощо. Після i-ї ітерації зовнішнього циклу обчислюються найкоротші шляхи довжиною не більше i ребер. У кожному простому шляху може бути максимум |V|-1 ребер, тому зовнішній цикл виконується саме |V|-1 раз. Ідея полягає в тому, що якщо ми вирахували найкоротший шлях з не більш ніж i ребрами, то ітерація по всіх ребрах гарантує отримання найкоротшого шляху з не більше ніж i + 1 ребрами.

Приклад

Давайте розберемося в алгоритмі на прикладі графу, зображеного на рис. 10.7.

Нехай початкова вершина дорівнює 0. Візьміть всі відстані за нескінченні, крім відстані до самої src. Загальна кількість вершин у графі дорівнює 5, тому всі ребра потрібно пройти 4 рази.

A	B	C	D	E
0	∞	∞	∞	∞

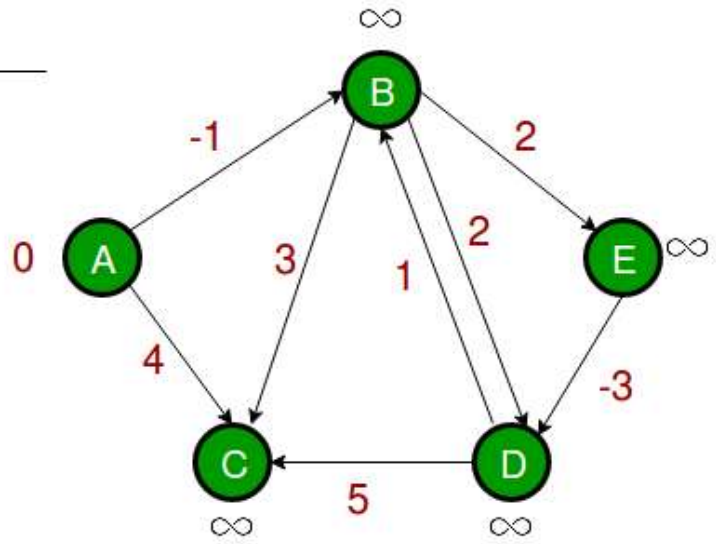

Рисунок 10.7 – Початковий граф для пошуку алгоритму Беллмана-Форда

Нехай ребра відпрацьовуються в такому порядку: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). Ми отримуємо такі відстані, коли прохід по ребрах був здійснений вперше. Перший рядок показує початкові відстані, другий рядок показує відстані, коли ребра (B, E), (D, B), (B, D) та (A, B) обробляються. Третій рядок показує відстань у процесі обробки (A, C). Четвертий рядок показує, що відбувається, коли обробляються (D, C), (B, C) та (E, D) (рис. 10.8).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

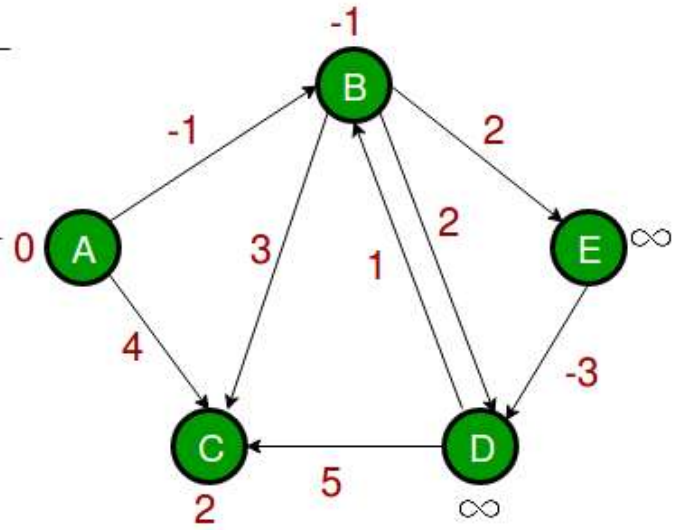

Рисунок 10.8 – Алгоритм Беллмана-Форда: перша ітерація

Перша ітерація гарантує, що всі найкоротші шляхи будуть не довшими за шлях в 1 ребро. Ми отримуємо такі відстані, коли буде здійснено другий прохід по всіх ребрах (в останньому рядку показані кінцеві значення).

На рис. 10.9 показано результат роботи алгоритму після другої ітерації.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

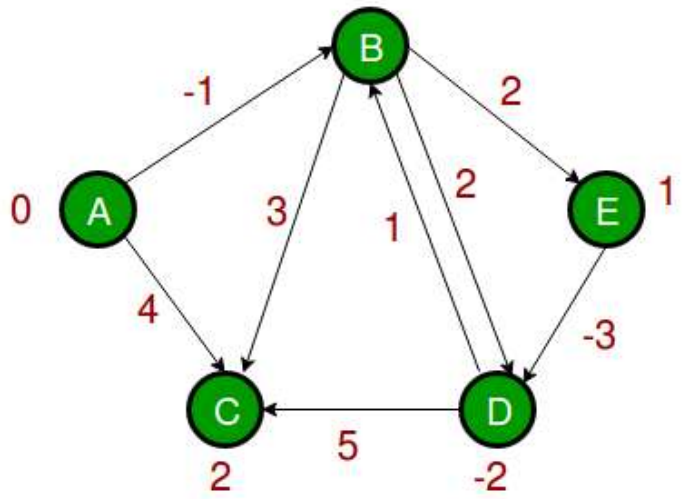

Рисунок 10.9 – Алгоритм Беллмана-Форда: друга ітерація

Друга ітерація гарантує, що всі найкоротші шляхи матимуть довжину трохи більше 2 ребер. Алгоритм проходить по всіх ребрах ще 2 рази. Відстань мінімізується після другої ітерації, тому третя і четверта ітерації не оновлюють значення відстаней.

10.3 Контрольні питання

Яка часова складність алгоритму Дейкстри?

Які обмеження має застосування алгоритму Дейкстри?

Які структури даних використовуються для реалізації алгоритму Дейкстри?

У чому переваги алгоритму Беллмана-Форда порівняно з алгоритмом Дейкстри?

Сформулюйте основні кроки алгоритму Беллмана-Форда.

11 ДИНАМІЧНЕ ПРОГРАМУВАННЯ

11.1 Загальне поняття про динамічне програмування

Динамічне програмування – це особливий підхід до розв'язання завдань. Немає єдиного означення динамічного програмування, але спробуємо його сформулювати. Ідея полягає в тому, що оптимальне рішення найчастіше можна знайти, розглянувши всі можливі шляхи вирішення завдання, і вибрати серед них найкраще.

Робота динамічного програмування дуже нагадує рекурсію з запам'ятовуванням проміжних рішень. Таку рекурсію ще називають **мемоізацією**. Рекурсивні алгоритми, зазвичай, розбивають велике завдання на більш дрібні подзадачі і розв'язують їх. Динамічні алгоритми ділять завдання на окремі частини та обчислюють їх по черзі, крок за кроком нарощуючи розв'язки. Тому динамічні алгоритми можна рохглядати як рекурсію, що працює знизу вверх (рис. 11.1).

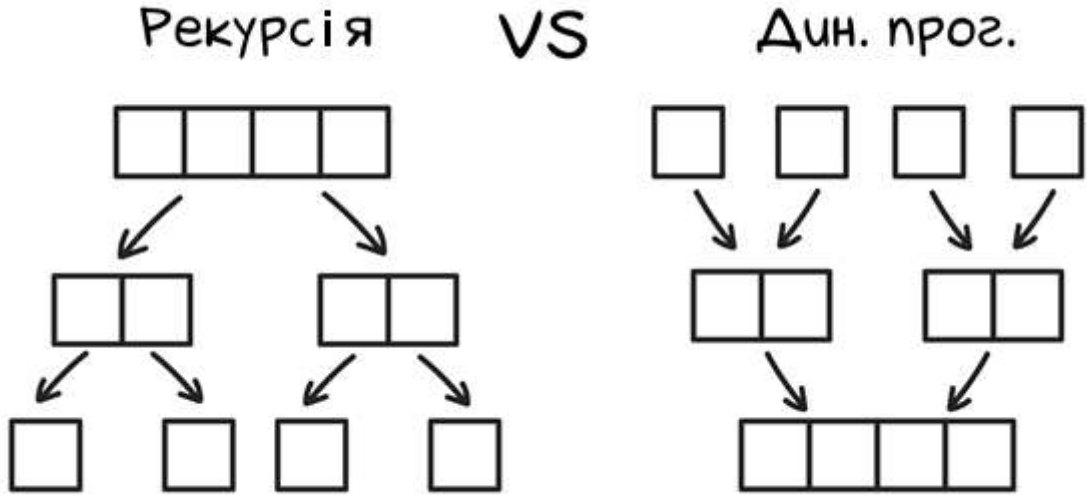

Рисунок 11.1 – Порівняння методів рекурсії та динамічного програмування

Магія динамічного програмування полягає у розумному поводженні з рішеннями підзавдань. «Розумний» у цьому контексті означає «що не вирішує те саме підзавдання двічі». Для цього рішення дрібних підзавдань мають десь зберігатися. Для багатьох реальних алгоритмів динамічного програмування такою структурою даних є таблиця.

У найпростіших випадках ця таблиця складатиметься лише з одного рядка – аналогічно звичайному масиву. Ці випадки будуть називатися одновимірним динамічним програмуванням і використовувати O(n) пам'яті. Наприклад, алгоритм ефективного обчислення чисел Фібоначчі використовує звичайний масив для запам'ятовування проміжних обчислених результатів. Класичний рекурсивний алгоритм робить дуже багато

безглуздих робіт – він по мільйонному разу розраховує те, що вже було розраховано у сусідніх гілках рекурсії.

У найпоширеніших випадках ця таблиця виглядатиме звично і складатиметься з рядків та стовпців. Звичайна таблиця, схожа на таблиці Excel. Це називається двовимірним динамічним програмуванням, яке при n рядках та n стовпцях таблиці потребує $O(n \cdot n) = O(n^2)$ пам'яті. Наприклад, квадратна таблиця з 10 рядків та 10 стовпців міститиме 100 комірок. Трохи нижче буде детально розібрано саме таке завдання.

Бувають і більш заплутані задачі, що використовують для розв'язання тривимірні таблиці, але це рідкість – розв'язання задачі з використанням тривимірної таблиці часто просто не можна собі дозволити. Невелика двовимірна таблиця на 1024 рядки і 1024 стовпці може вимагати кілька мегабайтів пам'яті. Тривимірна таблиця з такими ж параметрами займатиме вже кілька гігабайтів.

Що потрібно, щоб вирішити задачу динамічно, крім її вихідних даних? Усього три речі.

- Таблиця, до якої вноситимуться проміжні результати. Один з них буде обраний наприкінці роботи алгоритму як відповідь.
- Декілька простих правил щодо заповнення порожніх комірок таблиці, базуючись на значеннях у вже заповнених комірках. Універсального рецепта тут немає і до кожного завдання потрібен свій підхід.
- Правило вибору фінального розв'язку після заповнення таблиці.

Розберемо ці принципи на прикладі.

11.2 Приклад розв'язання задачі

Демонстраційним піддослідним виступить класичне завдання динамічного програмування – відстань Левенштейна. Незважаючи на уявну складну назву, насправді це завдання про трансформацію одного слова в інше шляхом додавання, видалення та заміни букв з мінімальною кількістю опера

Це завдання може бути сформульовано так: знайти мінімальну «відстань» між двома словами. Відстанню в цьому випадку буде мінімальна кількість операцій, які потрібно застосувати до першого слова, щоб отримати друге (або навпаки).

Доступних операцій у нас три:

- insert – додати одну літеру в будь-яке місце в слові, зокрема на початок і в кінець;
- delete – видалити одну літеру з будь-якого місця у слові;
- replace – замінити одну літеру в певному місці на іншу літеру.

Всі ці операції мають рівну вартість: +1 до відстані між словами.

Візьмемо, як приклад, два простих слова, MONEY та MONKEY. Яка мінімальна кількість операцій потрібна, щоб перетворити MONEY на

MONKEY? Винахідливе людське око швидко зрозуміє, що одна: додати літеру K після між третьою та четвертою літерами. Візьмемо випадок складніше. Спробуємо перетворити слово SUNDAY на слово SATURDAY, і побачимо, що кількість комбінацій, які потрібно перебрати, потенційно дуже велика. Якщо вирішувати завдання перебором, можна розглянути всі можливі комбінації, як у прикладі зі зламом пароля. Замість можливих 94 символів-кандидатів ми маємо три операції –insert, delete, replace. Три комбінації для першої літери, 3*3 для двох букв, 3*3*3 для трьох букв, 3^N – для N букв. Комбінаторний вибух.

Динамічне рішення

Приступимо до динамічного рішення. На початку створимо таблицю та розмістимо вихідні слова на її краях, залишивши трохи вільного місця (табл. 11.1). Другий стовпець і другий рядок використовуватимемо для порожніх рядків (їх часто позначають символом ε, читається epsilon).

Таблиця 11.1 – Початкова таблиця

	ε	**S**	**A**	**T**	**U**	**R**	**D**	**A**	**Y**
ε									
S									
U									
N									
D									
A									
Y									

Тепер заповнимо другий стовпець і другий рядок, керуючись абсолютно інтуїтивними міркуваннями: як перетворити порожній рядок на якийсь рядок? Звичайно ж, додати до нього потрібні символи! Наприклад, щоб перевести ε до SATU, потрібно додати букву S, букву A, букву T та букву U. Чотири операції. Що робити з перетворенням ε на ε (другий рядок, другий стовпець)? Елементарно нічого робити не потрібно, нуль дій (табл. 11.2).

Таблиця 11.2 – Перший крок

	ε	**S**	**A**	**T**	**U**	**R**	**D**	**A**	**Y**
ε	0	1	2	3	4	5	6	7	8
S	1								
U	2								
N	3								
D	4								
A	5								
Y	6								

Тепер потрібна система простих правил, за допомогою якої зможемо заповнити таблицю. Таблиця буде називатися D, а перший рядок і стовпець залишаться на її полях. Працювати з таблицею ми будемо як із двовимірним масивом, тобто символами D[0, 2] позначається комірка на перетині нульового рядка та другого стовпця. У прикладі D[0, 2] = 2.

Також назвемо слово по вертикалі A, а слово по горизонталі – B. Ця пара нам потрібна, щоб мати доступ до оригінальних слів на полях. Через додаткові колонки для ε індекси в A і B відрізняються від індексів в таблиці. Якщо точніше, вони зсунуті на одиницю. A[0] = S, A[1] = U, A[2] = N, B[7] = Y, тощо.

Нарешті створимо наше правило заповнення таблиці. Для кожної нової комірки ми перевіряємо верхню, ліву або ліву верхню по діагоналі сусідні комірки. З трьох чисел буде обрано найменше та записано до нової комірки.

```
D[i, j] = minimum(
    D[i-1, j] + 1,                          // delete
    D[i, j-1] + 1,                          // insert
    D[i-1, j-1] + (A[i-1] = B[j-1] ? 0 : 1)  // replace
)
```

Це важливий момент у динамічному програмуванні: правила здаються безглуздими, а зібрати загальну картину того, що відбувається в голові, складно. Давайте подивимося на маленький фрагмент таблиці, можливо, він проллє світло на деякі деталі (табл. 11.3).

Таблиця 11.3 – Другий крок

	ε	**S**	**A**
E	0	1	2
S	1		
U	2		

Що потрібно записати в комірку D[1,1] як результат переходу з S до S? Інтуїтивно зрозуміло, що для цього нічого робити і не потрібно, нуль операцій. Запишемо в комірку нуль. На що схоже це значення з огляду на те, що віднімати нічого не можна? Серед сусідів нуль є лише з діагоналі (табл. 11.4).

Таблиця 11.4 – Третій крок

	ε	**S**	**A**
ε	0	1	2
S	1	0	
U	2		

Що записати в комірку D[2,1] як результат переходу з SU до S? Потрібно видалити букву U, це одна операція. Насправді, вартість переходу з SU в S дорівнює вартості видалення літери U і переходу з S в S (чия вартість вже була порахована і лежить в комірці D[1,1]) (табл. 11.5).

Таблиця 11.5 – Четвертий крок

	ε	**S**	**A**
E	0	1	2
S	1	0	
U	2	1	

Тепер подивимося на комірку D[1,2], перехід із S до SA. Так, саме вартість переходу дорівнюватиме вартості додавання літери A і переходу з S в S, разом одиниця (табл. 11.6).

Таблиця 11.6 – П’ятий крок

	E	**S**	**A**
ε	0	1	2
S	1	0	1
U	2	1	

Остання комірка D[2,2], перехід із SU SA. Найоптимальнішим рішенням було б замінити букву U на букву A плюс ціна безкоштовного переходу з S в S (табл. 11.7).

Таблиця 11.7 – Шостий крок

	ε	**S**	**A**
ε	0	1	2
S	1	0	1
U	2	1	1

У правій нижній комірці міститься фінальна вартість переходу зі слова SU в слово SA. Так можна заповнити всю таблицю. З комірки D[6,8] ми дізналися, що перехід із слова SUNDAY у слово SATURDAY коштує щонайменше три операції. Жирним шрифтом виділимо оптимальний шлях.

Давайте простежимо його за кроками. Перехід з S до S нічого не вартий. Перехід з S до SA коштує одну операцію. Перехід з S до SAT коштує дві операції. Перехід із SU в SATU коштує дві операції. Перехід із SUN в SATUR коштує три операції. Перехід із SUND до SATURD коштує три операції (вартість попереднього переходу плюс нульова ціна переходу з D до D). Перехід із SUNDA до SATURDA коштує три операції. Нарешті, перехід із SUNDAY до SATURDAY вимагає тих самих трьох операцій.

До речі, якщо придивитися до таблиці, можна побачити, що оптимальних рішень кілька: з D[1,2] можна перейти і у D[1,3], і у D[2,2]. У цій постановці завдання нас цікавить лише мінімальна кількість, а не список усіх можливих шляхів вирішення, тож це несуттєво.

Таблиця 11.8 – Підсумкова таблиця

	ε	**S**	**A**	**T**	**U**	**R**	**D**	**A**	**Y**
ε	0	1	2	3	4	5	6	7	8
S	1	**0**	**1**	**2**	3	4	5	6	7
U	2	1	1	**2**	**2**	3	4	5	6
N	3	2	2	2	3	**3**	4	5	6
D	4	3	3	3	3	4	**3**	4	5
A	5	4	3	4	4	4	4	**3**	4
Y	6	5	4	4	5	5	5	4	**3**

Ось так, власне, і виглядає більшість рішень зі світу динамічного програмування. До речі, це рішення має назву алгоритм Вагнера-Фішера. Цікавий факт: цей алгоритм практично паралельно опублікували різні групи незнайомих вчених з різних кінців планети в епоху, коли ще не було інтернету. Товариші Вагнер та Фішер, до речі, були далеко не першими.

Давайте тепер розглянемо, у чому різниця застосування цього алгоритму від рішення перебором.

Аналіз розв'язання

Як було зазначено, рішення перебором цього завдання простою рекурсією має часову складність $O(3^n)$, але не вимагає зайвої пам'яті, отже, O(1) операцій у пам'яті.

Які витрати динамічного рішення? Уявімо, що порівнюються слова однакової довжини, по n символів у слові. Все рішення зводиться до заповнення таблиці з n + 1 рядками (окрема для порожнього рядка ε), і n + 1 стовпцями. Отже, використовується $(n + 1)^2$ комірок. Не рахуватимемо копійки, і округлимо кількість комірок до n^2. Для кожної комірки ми перевірятимемо трьох її сусідів, що вимагає константного часу O(1). Отже, для заповнення всієї таблиці потрібно $O(n^2)$ операцій.

Якою буде витрата пам'яті? Для таблиці з n^2 комірок нам потрібно $O(n^2)$ пам'яті.

Якщо слова різної довжини, то можна або дивитися за найдовшим, або трохи підвищити ступінь складності формули. Наприклад, якщо перше слово має довжину n, а друге – m, то буде потрібно O(nm) часу та O(nm) пам'яті.

11.3 Підсумки

Основна ідея динамічного програмування має простежуватися у наведеному прикладі: ми жертвуємо солідною кількістю пам'яті (O(nm) замість

O(1)), але отримуємо просто божевільний виграш у часі (O(nm) проти $O(3^n)$).

Перелічимо усі ключові особливості динамічного програмування.

Переваги

Швидкість. Головна перевага динамічного програмування. Нерозв'язні завдання стають розв'язуваними, здебільшого за квадратичний час! Одна операція на заповнення кожної комірки таблиці – і питання закрите.

Універсальність. One Ring to rule them all – створення компактної системи з кількох правил заповнення таблиці гарантує вирішення завдання будь-яких даних. Ані винятків, ані межових випадків, кілька рядків – і складну проблему вирішено.

Точність. Оскільки метод динамічного програмування розглядає абсолютно всі можливі варіанти та сценарії, він гарантовано виявить оптимальне рішення. Жодної втрати точності, ніяких приблизних відповідей. Якщо розв'язок існує, він буде знайдений.

Недоліки

Пам'ять. У більшості випадків алгоритми динамічного програмування вимагають часу на побудову та заповнення таблиць. Таблиці споживають пам'ять. Це може стати проблемою: якщо самі таблиці дуже великі, або якщо для вирішення якогось завдання потрібно побудувати дуже багато таких таблиць і тримати їх усіх у пам'яті.

Когнітивне навантаження. Розв'язання заплутаного завдання за допомогою компактної системи правил – дуже приваблива ідея. Але тут є один нюанс: для складання чи хоч би для розуміння цих систем правил необхідно навчитися «думати у стилі динамічного програмування». Це причина досить суперечливої репутації динамічного програмування.

Сфери застосування

Динамічне програмування – не теоретична конструкція, яка не виходить за межі наукових праць. Воно користується популярністю у багатьох прикладних областях. Їх чимало: прикладна математика, машинобудування, теорії управління чи прогнозування фінансових даних. Але ми зупинимося на одній – біоінформатиці.

Вчені в цій галузі займаються «оцифровуванням» біологічного матеріалу, а також зберіганням та аналізом отриманої інформації. У цій науці сотні захопливих аспектів, і вона ставить перед розробниками дуже серйозні завдання, адже даних неймовірно багато. Наприклад, у геномі людини близько трьох мільярдів пар нуклеотидів (цеглинок ДНК). Одна пара зазвичай кодується одним байтом, у результаті виходить близько трьох мільярдів байтів інформації на один-єдиний геном – три гігабайти даних на одну людину.

Один геном не створює серйозних проблем, але геноми самі собою малоцікаві: щоб виявити мутації в геномі конкретної людини, потрібно

спочатку «вирівняти» його з іншими, референсними геномами (вирівняними і розміченими заздалегідь). Можливих варіантів цього вирівнювання може бути величезна кількість, але потрібно знайти найправдоподібніший з них. Тобто варіант, який має максимальну ймовірність виникнення. Наприклад, варіант із найменшою кількістю мутацій. Якщо взяти до уваги, що генетичний код зазвичай зберігаються у вигляді дуже довгих рядків, що складаються з різних букв, то приклад з відстанню Левенштейна починає грати новими фарбами. Це завдання, що потенційно приводить до комбінаторного вибуху (як перебір всіх комбінацій символів для зламу пароля), чудово вирішується методами динамічного програмування!

Приклади застосування динамічного програмування для розв’язання деяких задач можна знайти в [3].

11.4 Контрольні питання

У чому полягає спільність і відмінність динамічного програмування та рекурсії?

Чому вважається, що алгоритми, які реалізують метод динамічного програмування, працюють швидше рекурсивних?

Яка структура даних асоціюється з методом динамічного програмування?

Наведіть приклади практичного застосування методу динамічного програмування.

12 ЖАДІБНІ АЛГОРИТМИ ТА NP-ПОВНІ ЗАДАЧІ

Жадібними називають клас алгоритмів, які полягають у прийнятті локально оптимальних рішень на кожному етапі. Оскільки локально оптимальне рішення обчислити набагато простіше, ніж глобально оптимальне, такі алгоритми мають хорошу асимптотику.

У деяких випадках жадібні алгоритми приводять до оптимальних кінцевих рішень, а в інших – ні. Придумати і довести коректність жадібних алгоритмів часто буває дуже складно.

12.1 Задача про видачу суми мінімальною кількістю монет

Монетна система деякої держави складається з монет номіналом

$$a_1=1<a_2<a_3<\ldots<a_n,$$

причому кожен наступний номінал ділиться на попередній. Потрібно видати суму S найменшою можливою кількістю монет.

Жадібний алгоритм розв'язання цієї задачі полягає в тому, що потрібно спочатку взяти найбільшу кількість монет найбільшого номіналу a_n, а саме: S / a_n, потім для набору залишку взяти найбільшу кількість монет номіналом a_{n-1} і так далі. Оскільки $a_1=1$, наприкінці ми завжди наберемо потрібну суму.

Зауважимо, що без припущення, що номінали монет діляться один на одного, алгоритм не завжди виводить оптимальне рішення, і в цьому випадку завдання вирішуватиметься динамічним програмуванням. Наприклад, суму в 24 гривні монетами в 1, 5 і 7 гривен жадібний алгоритм розмінює як (7×3+1×3), тоді як оптимальным буде (7×2+5×2).

Коректність. Доведімо корректність жадібного алгоритму від протилежного: нехай максимальна монета має номінал $a_n<S$, але в оптимальній відповіді її немає.

Нехай в оптимальній відповіді монета номиналом a_i зустрілась b_i разів:

$$S=\sum a_i b_i.$$

Якщо $b_i \geq a_{i+1}/a_i$, то a_{i+1}/a_i монет можна замінити на одну монету номіналом a_{i+1}, а це означає, що це не оптимальна відповідь. Тому $b_i \leq a_{i+1}/a_i - 1$.

Тепер порахуємо, якою може бути максимальна сума всіх номіналів всіх монет в оптимальній відповіді, якщо ми не брали максимальні монети.

$$\begin{gathered} S= a_1\cdot b_1+a_2\cdot b_2+\ldots+a_{n-1}\cdot b_{n-1}\leq \\ \leq a_1\cdot(a_2/a_1-1)+a_2\cdot(a_3/a_2-1)+\ldots+a_{n-1}\cdot(a_n/a_{n-1}-1) = \\ = (a_2-a_1)+(a_3-a_2)+\ldots+(a_n-a_{n-1}) = a_n-a_1 < a_n \end{gathered}$$

Звідси робимо висновок, що якщо $S \geq a_n$, то в оптимальну відповідь завжди доведеться взяти максимальну монету розміру a_n, тому що всі менші монети просто не зможуть в оптимальній відповіді давати так багато. Аналогічно для наборів з менших монет отримуємо той самий результат для всіх $S < a_n$, звідки випливає коректність жадібного алгоритму.

Також цей алгоритм працює не тільки для номіналів, що діляться один на одного, але і для деяких інших. Такі монетні системи, де жадібний алгоритм працює, називають канонічними.

Вправа. Чи правильно, що алгоритм працює коректно для нашої грошової системи (1, 2, 5…)?

12.2 Задача про «чорну п'ятницю» в магазині

Уявіть собі, що у чорну п'ятницю магазин одягу та взуття проводить акцію, і кожному 25-му покупцю дозволяється придбати безкоштовно по одному товару кожного типу, але так, щоб їх сумарна вага склала не більше
3 кг. Уявіть, що Ви – той самий 25-й покупець, і Вам потрібно підібрати набір товарів максимальної вартості з такого списку:

Шкіряна куртка 3,5 кг – 5000 грн.

Куртка зимова 2,5 кг – 3500 грн.

Кросівки 1 кг – 2500 грн.

Джинси 1 кг – 1500 грн.

Спортивний костюм 1 кг – 1000 грн.

Футболка 0,5 кг – 500 грн.

Жадібна стратегія говорить, що треба брати товар максимальної вартості, але «Шкіряна куртка» виявляється недоступною. Якщо взяти наступний за вартістю товар «Куртка зимова», то разом з нею можна буде взяти тільки футболку, і сумарна вартість складе 3500 + 500 = 4000 грн. Але можна було б взяти кросівки (2500), джинси (1500) та спортивний костюм (1000) і наповнити свій кошик товаром на суму 5000 грн. Як можна побачити, в цьому випадку жадібний алгоритм трохи не дотягує до оптимального розв'язку, хоча результат виглядає зовсім непогано.

Другий приклад приводить нас до такого висновку: іноді ідеальне – ворог хорошого. У деяких випадках достатньо алгоритму, здатного вирішити завдання досить добре. І в таких галузях жадібні алгоритми працюють просто чудово, оскільки вони легко реалізуються, а отримане рішення зазвичай близько до оптимуму.

Якщо б схожу акцію проводили у фруктовій лавці (звичайно, без обмежень щодо вибору товару, але з обмеженням на загальну вагу 3 кг), як мала б виглядати жадібна стратегія в цьому випадку?

Пропонуєтья такий набір товарів з Сільпо (ціни станом на 26.11.23 р.) (рис. 12.1).

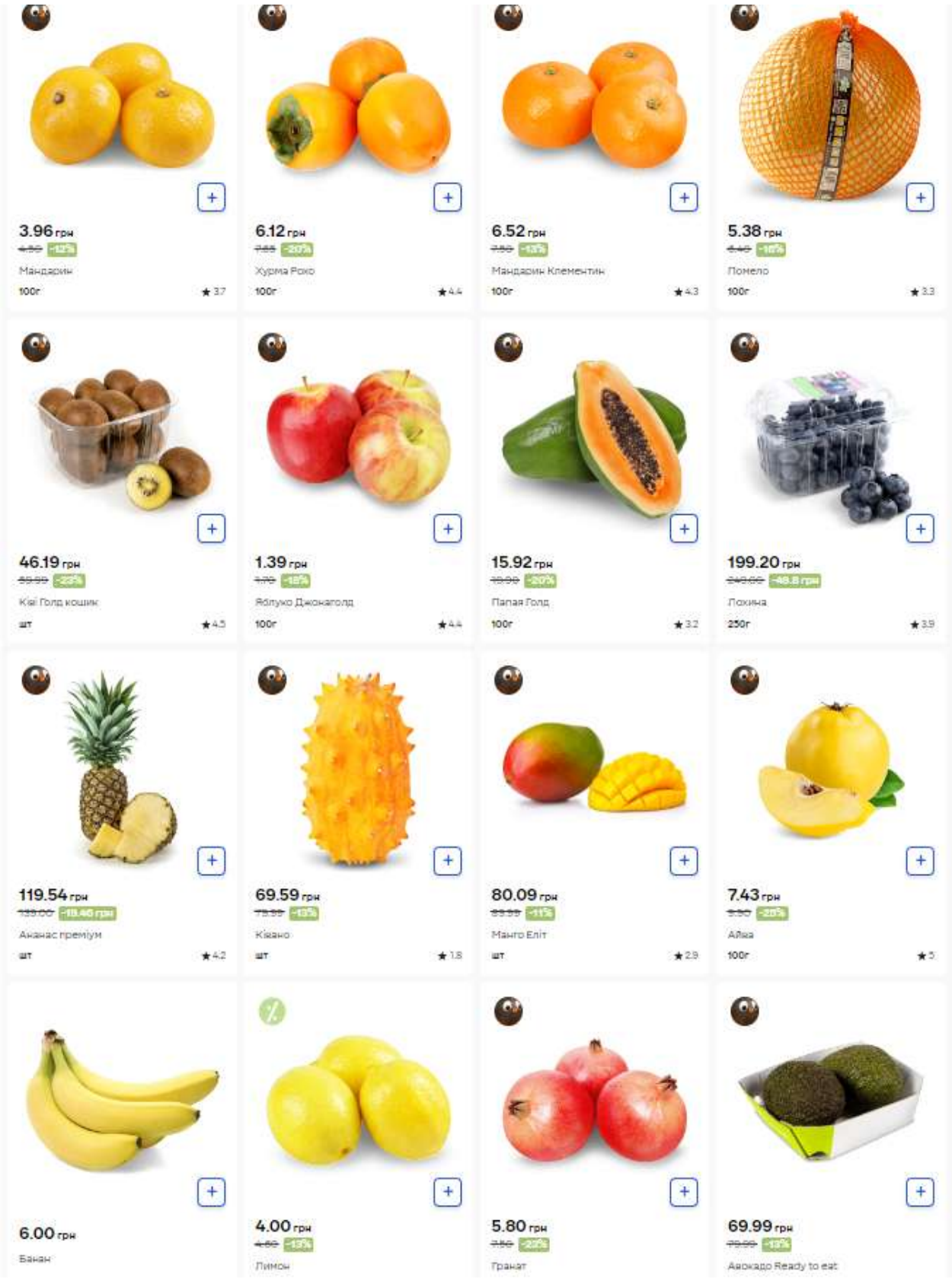

Рисунок 12.1 – Ціни на фрукти та овочі в магазині Сільпо

Жадібний алгоритм від Марії Шклярук

1. Обрахувати ціну за 100 г

2. Відсортувати масив за спаданням цни

3. Набрати скільки можна найдорожчого товару, потім наступного і т. д. поки не закінчаться гроші.

Забув сказати: дозволяється відрізати або відкусити шматочок від будь-якого фрукта.

Тоді жадібний алгоритм Олександра Ткаченка

Відсортуємо товари за спаданням ціни за грам і братимемо їх жадібно в такому порядку: від останнього товара, що не влізе повністю, відкусимо частину, решту покладемо в кошик.

12.3 Задача про розклад занять в одній аудиторії

Подано заявки на проведення занять у деякій аудиторії. У кожній заявці вказано початок l_i та кінець r_i заняття. Потрібно з усіх заявок залишити якнайбільше таким чином, щоб вони не перетиналися.

Тут жадібність стає не такою вже очевидною, бо незрозуміло, в якому порядку розглядати заявки та як їх «жадібно» набирати. Подивимося на першу за часом закінчення заявку. Зауважимо, що нам завжди вигідно внести її в оптимальну відповідь – вона закінчується раніше всіх інших, а тому, якщо в оптимальній відповіді перша заявка якась інша, ми можемо безболісно замінити її на першу за часом кінця, і наразі нових перетинів не з'явиться, тому що ми просто зрушили найпершу заявку ще лівіше.

Після того, як ми вибрали першу за часом кінця, приберемо з розгляду всі, що з нею перетинаються, і так само виберемо з тих, що залишилися, першу за часом кінця, знову приберемо всі, що перетинаються і так далі, поки заявки не закінчаться.

При реалізації зручно не видаляти відрізки явно, а просто відсортувати відрізки за часом і підтримувати час, коли закінчується останнє взяте заняття. Яка асимптотична складність такого алгоритму?

```
struct activity {
    int l, r;
};

int solve(vector<activity> activities) {
    sort(activities.begin(), activities.end(), [](activity a, activity b) {
        return a.r < b.r;
    });
    int cnt = 0, last = 0;
    for (activity a : activities)
        if (a.l >= last)
            last = a.r, cnt++;
    return cnt;
}
```

12.4 Задача про покриття множин

Ви відкриваєте власну авторську програму на радіо і хочете, щоб вас слухали в усіх 50 американських штатах. Потрібно вирішити, на яких радіостанціях має транслюватись ваша передача. Кожна станція коштує грошей, тож кількість станцій потрібно звести до мінімуму. Список станцій наведено на рис. 12.2.

KONE	ID,NV,UT
KTWO	WA, ID, MT
KTHREE	OR, NV, CA
KFOUR	NV, UT
KFIVE	CA, AZ

Рисунок 12.2 – Список станцій та штатів, де вони працюють

Кожна станція покриває певний набір штатів, ці набори перекриваються (рис. 12.3).

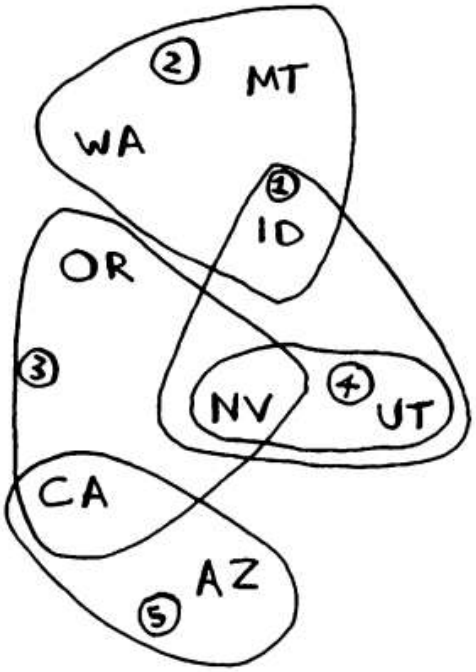

Рисунок 12.3 – Покриття деяких штатів окремими станціями

Як знайти мінімальний набір станцій, який би покривав усі 50 штатів? Начебто просте завдання, правда? Виявляється, воно надзвичайно складне. Ось як це робиться.

1. Скласти список всіх можливих підмножин станцій – так звану степеневу множину. У ньому міститься 2^n можливих підмножин.

2. З цього списку вибирається множина з найменшим набором станцій, що покривають усі 50 штатів (рис. 12.4).

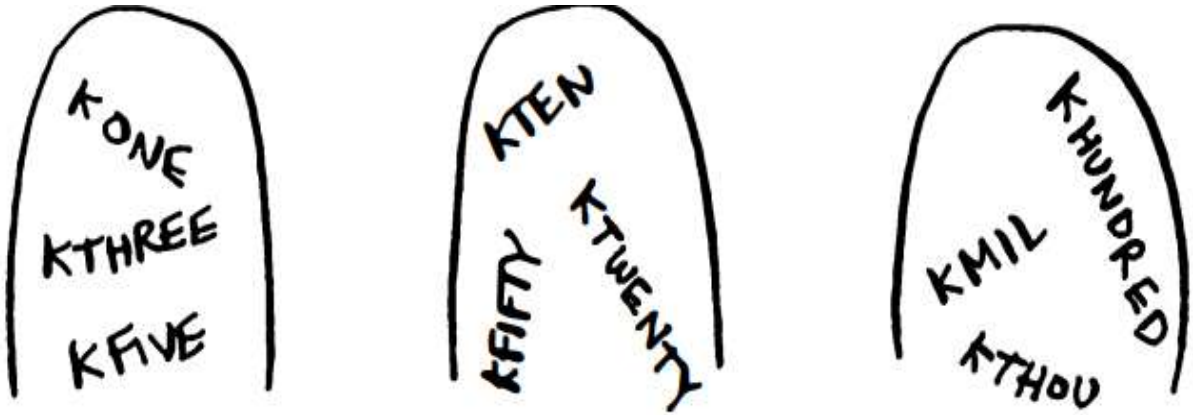

Рисунок 12.4 – Варіант розв’язання задачі

Проблема в тому, що обчислення всіх можливих підмножин станцій займе надто багато часу. Для станцій воно вимагатиме часу $O(2^n)$. Якщо станцій небагато, скажімо від 5 до 10 – це припустимо. Але подумайте, що станеться в усіх розглянутих прикладах при великій кількості елементів. Припустимо, ви можете обчислювати по 1000 підмножин на секунду. Обчислення потребуватиме років…

Не існує алгоритму, який буде обчислювати підмножини з прийнятною швидкістю! Що ж робити?

Наближені алгоритми

На допомогу приходять жадібні алгоритми! Ось як виглядає жадібний алгоритм, який видає результат, досить близький до оптимум.

1. Вибрати станцію, що покриває найбільшу кількість штатів, які ще не входять у покриття. Якщо станція покриватиме деякі штати, що вже входять у покриття, – це нормально.

2. Повторювати, доки залишаються штати, які не входять до покриття.

Цей алгоритм є наближеним. Коли обчислення точного рішення займає дуже багато часу, застосовується наближений алгоритм. Ефективність наближеного алгоритму оцінюється за:

- часом виконання;
- близькістю отриманого рішення до оптимального.

Жадібні алгоритми хороші і тим, що вони, зазвичай, легко формулюються, і тим, що простота, зазвичай, обертається швидкістю виконання. Наразі жадібний алгоритм виконується за час $O(n^2)$, де n – кількість радіостанцій.

12.5 NP-повні задачі

Класичний приклад такої задачі – задача про комівояжера. В цій задачі комівояжеру потрібно відвідати n різних міст. Нехай, задля конкретики, n = 5. Комівояжер намагається знайти найкоротший шлях, який охопить усі п’ять міст. Щоб знайти найкоротший шлях, спочатку потрібно обчислити всі можливі шляхи. Для цього обраховуються такі варіанти: 1, 2, 3, 4, 5 1, 2, 3, 5, 4 1, 2, 4, 3, 5 … 5, 4, 3, 2, 1. Скільки всього можливо варіантів?

Така залежність називається факторіальною. Отже, 5! = 120. Припустимо, є 10 міст. Скільки можливих маршрутів? 10! = 3 628 800. Вже для 10 міст доводиться обчислювати більше 3 мільйонів можливих маршрутів. Як бачите, кількість можливих маршрутів стрімко зростає! Ось чому неможливо знайти правильний розв’язок задачі про комівояжера для дуже великої кількості міст.

У задачі про комівояжера і задачі покриття множин є дещо спільне: ви обчислюєте кожне можливе рішення та вибираєте найкоротше/мінімальне. Обидві ці задачі є NP-повними.

Наближене рішення

Як виглядає хороший наближений алгоритм для завдання про комівояжера? Це має бути простий алгоритм, що знаходить короткий шлях. Спробуйте самостійно знайти відповідь.

Можна зробити приблизно так: початкове місто вибирається довільно, після чого щоразу, коли комівояжер вибирає наступне місто, він переміщається до найближчого міста з тих, що він ще не відвідував.

Коротке пояснення NP-повноти: деякі завдання уславилися складністю свого рішення. Завдання про комівояжера і завдання про покриття множини – два класичні приклади. Багато експертів вважають, що написати швидкий алгоритм для вирішення таких завдань неможливо.

Як визначити, що задача є NP-повною?

NP-повні задачі зустрічаються дуже часто. І було б корисно, якби ви могли зрозуміти, що вирішуване завдання є NP-повним. В цей момент можна припинити пошуки ідеального рішення та перейти до застосування наближеного алгоритму. Але визначити, чи є ваша задача NP-повною, непросто. Зазвичай різницю між легко розв’язуваними і NP-повними задачами дуже незначні. Наприклад, у темі графи йшлося про найкоротші шляхи. Оригінальнй алгоритм Дейкстри обчислює найкоротший шлях з точки A до точки B і робить це за $O(n^2)$.

Але якщо ви хочете знайти найкоротший шлях, що з’єднує кілька точок, то це вже задача про комівояжера, яка є NP-повною. Коротше кажучи, немає простого способу визначити, чи є задача, з якою ви працюєте, NP-повною. *Декілька характерних ознак*

- Ваш алгоритм швидко працює при малій кількості елементів, але суттєво уповільнюється зі збільшенням їх числа.

- Формулювання «всі комбінації x» часто вказує на NP-повноту задачі.

- Вам доводиться обчислювати всі можливі варіанти x, тому що задачу неможливо розбити на менші підзадачі? Така задача може виявитися NP-повною.

- Якщо в задачі зустрічається деяка послідовність (наприклад, послідовність міст, як у задачі про комівояжера) і задача не має простого рішення, вона може виявитися NP-повною.

- Якщо в задачі зустрічається деяка множина (наприклад, множина радіостанцій) і задача не має простого рішення, воно може виявитися NP-повною.

- Чи можна переформулювати задачу в термінах задачі покриття множин чи задачі про комівояжера? У такому разі ваша задача, безперечно, є NP-повною.

12.6 Підсумки

- Жадібні алгоритми прагнуть локальної оптимізації в розрахунку на те, що в результаті буде досягнуто глобального оптимуму.

- У NP-повних завдань немає відомих швидких рішень.

- Якщо у вас є NP-повне завдання, найкраще скористатися наближеним алгоритмом.

- Жадібні алгоритми легко реалізуються і швидко виконуються, тому з них виходять хороші наближені алгоритми.

12.7 Контрольні питання

Чому, на Вашу думку, жадібні алгоритми отримали таку непривабливу назву?

У чому переваги жадібних алгоритмів над алгоритмами, що реалізують метод динамічного програмування?

У чому, на Вашу думку, полягає основний недолік жадібних алгоритмів?

Які з відомих Вам алгоритмів можна віднести до жадібних?

Як можна «розпізнати» NP-повну задачу?

Якщо за всіма ознаками задача є NP-повною, але її все ж таки треба розв’язувати, як би Ви порадили це робити?

13 КЛАСТЕРИЗАЦІЯ ЗА МЕТОДОМ К-СЕРЕДНІХ

13.1 Поняття про кластеризацію

Кластеризація – це розбиття множини об’єктів на групи (кластери, підмножини) так, щоб елементи, що їх відносять до однієї підмножини, відрізнялися між собою значно менше, ніж елементи з різних підмножин.

Кластеризація – один із етапів аналізу даних. Мета – виділити групи подібних об’єктів, вивчити їх особливості та побудувати для кожної групи окрему модель.

На рис. 13.1 наведено набір двовимірних даних, які потрібно кластеризувати.

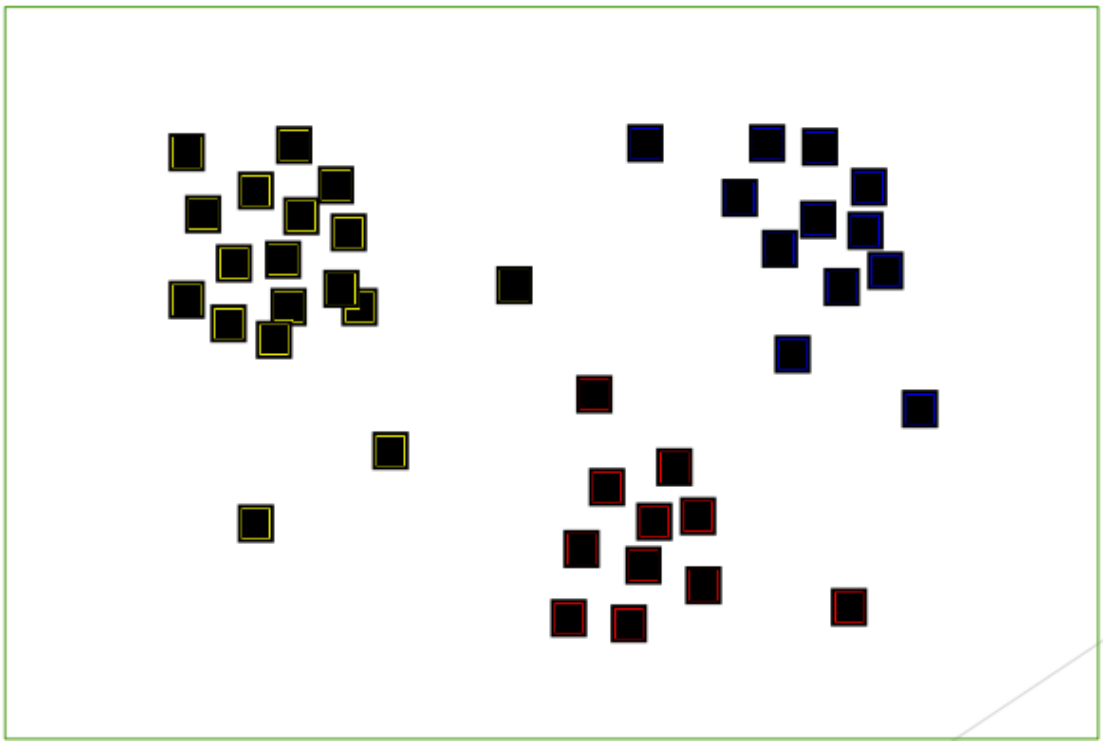

Рисунок 13.1 – Початкові дані для задачі кластеризації

Результати розв’язання задачі кластеризації наведено на рис. 13.2. Елементи, що належать одним і тим самим кластерам показано однаковии кольорами; елементи, що належать різним кластерам – різними кольорами.

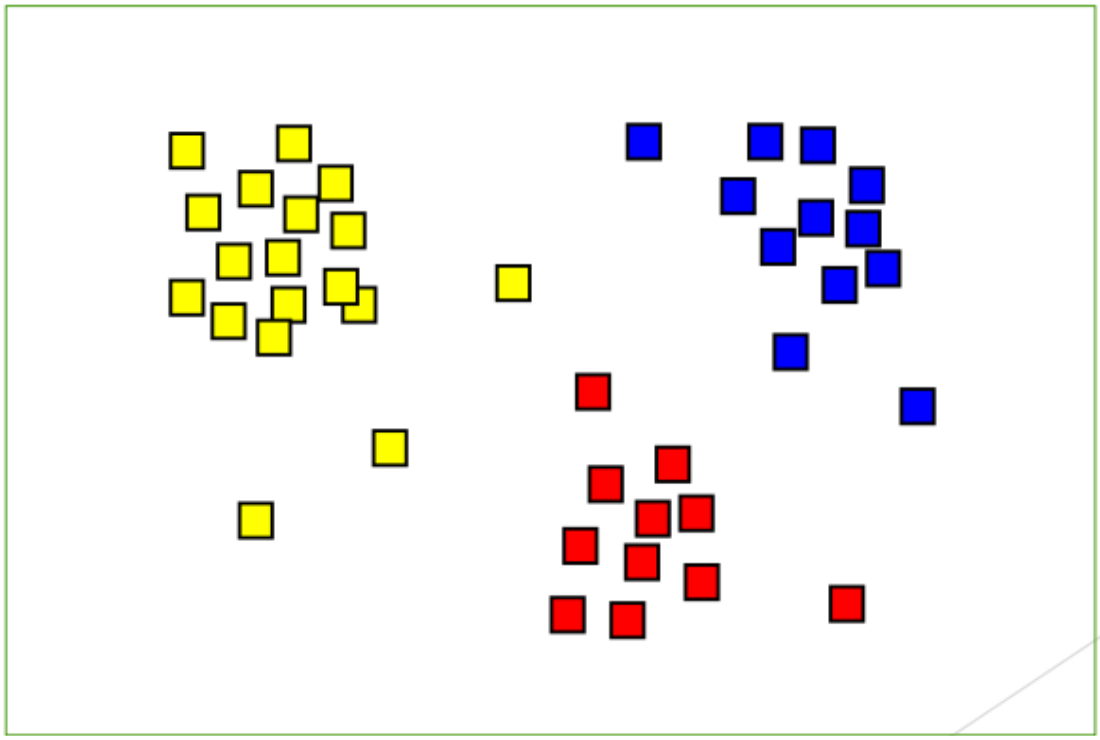

Рисунок 13.2 – Результат розв’язання задачі кластеризації

Відмінність від класифікації: перелік груп (та їх характеристики) не задано заздалегідь і визначається у процесі роботи алгоритму.

Відмінності від статистичних методів:

➢ відсутність попередніх припущень;

➢ не потрібна інформація про закон розподілу;

➢ можна використовувати для об’єктів з кількісними та якісними ознаками.

Етапи кластерного аналізу:

➢ визначення множини змінних, за якими оцінюватимуться об’єкти;

➢ нормалізація значень змінних;

➢ вибір методу кластеризації та виду метрики для обчислення значень міри подібності між об’єктами;

➢ створення груп подібних об’єктів (кластерів);

➢ аналіз результатів, коригування обраної метрики та методу кластеризації до отримання бажаного результату.

Щоб порівнювати два об’єкти, потрібно мати критерій, на підставі якого відбуватиметься порівняння. Критерій – відстань між об’єктами (метрика).

Найбільш поширені метрики

- Евклідова відстань.
- Манхеттенська відстань;
- Відстань Чебишова.

Бізнес-застосування:

1. Виявлення аномалій або ботів

Відокремлення ботів від корисних груп діяльності.

Алгоритм допомагає очистити дійсну групову активність від виявлення викидів.

2. Класифікація сенсорних вимірювань

Виявлення групових фотографій.

Визначення груп моніторингу здоров'я.

Розділення звуку.

Виявлення різноманітних дій у датчиках руху.

3. Категоризація запасів

Згрупувати запаси з показниками виробництва.

Згрупувати запаси з діяльністю збуту.

4. Поведінкова сегментація

Визначення персон на основі інтересів.

Створення профілів на основі моніторингу активності.

Сегментація за допомогою історії покупок.

Створення сегментів шляхом оцінювання дій на платформах, вебсайтах і в додатках.

13.2 K-середніх кластеризація: реальні застосування

K-середніх кластеризація стає все більш популярною в різних галузях. Ось кілька популярних реальних застосувань цього революційного алгоритму.

Рекомендаційні движки.

Кластеризація дуже корисна для механізмів рекомендацій. Ви можете скористатися цим алгоритмом і рекомендувати друзям пісні чи фільми на основі їхніх уподобань.

Сегментація зображення.

K-середніх кластеризація чудово підходить для сегментації фотографій. Програми для ілюстрації та редагування можуть скористатися атрибутами сегментації зображень цього алгоритму.

Кластеризація документів.

Кластеризація може допомогти вам згрупувати численні документи за короткий час. Це особливо корисно для людей, які мають кілька документів, що містять різну інформацію.

Сегментація клієнтів.

У багатьох галузях промисловості для оптимізації процесів кластеризація клієнтів використовує кластери. Продажі, реклама, спорт, електронна комерція, банківська справа та телекомунікації – це деякі галузі, які використовують переваги цього алгоритму.

13.3 Алгоритм k-середніх

Алгоритм прагне мінімізувати середньоквадратичне відхилення від центроїда для елементів кожного кластера.

$$R = \sum_{i=1}^{k} \sum_{x \in C_i} (x - c_i)^2,$$

де k – число кластерів;

C_i – отримані кластери;

c_i – центроїд i-го кластера.

Алгоритм k-середніх

1. Довільно вибираються центри кластерів k точок.
2. Для кожного об'єкта визначається, до якого кластера він належить (за замовчуванням Евклідова відстань між кожною точкою та центром кластера).
3. Перераховується центр c_i кожного кластера (як середнє всіх елементів кластера).
4. Кроки 2 і 3 повторюються поки відбувається зміна складу кластерів.

На рис. 13.3 наведено схему кластеризації за алгоритмом k-середніх.

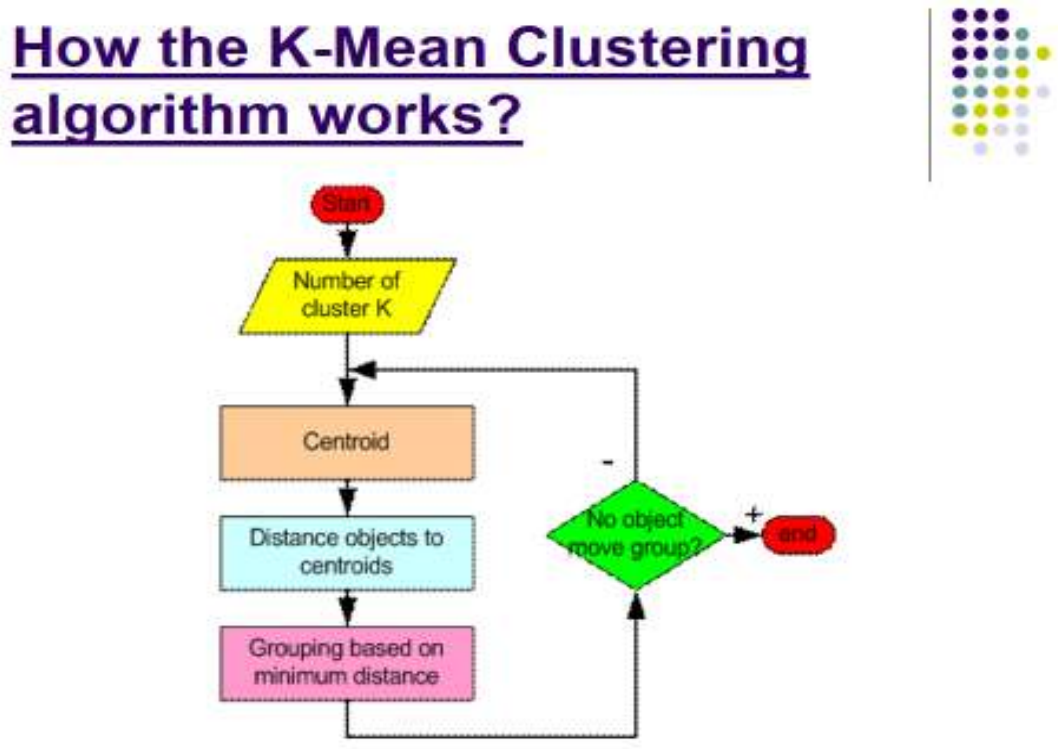

Рисунок 13.3 – Схема кластеризації за алгоритмом k-середніх

Опишемо кроки алгоритму, що наведено на рисунку, більш детально.

Крок 1. Почніть із прийняття рішення щодо значення k = кількість кластерів.

Крок 2. Розмістіть будь-який початковий розподіл, який класифікує дані, на k кластерів. Ви можете призначати навчальні зразки випадковим чином або систематично так:

1. Візьміть першу k навчальну вибірку як одноелементні кластери.

2. Призначте кожну з решти (N - k) навчальних вибірок кластера з найближчим центроїдом. Після кожного призначення повторно обчислюйте центроїд кластера нарощування.

Крок 3. Візьміть кожну точку послідовно та обчисліть її відстань від центроїда кожного з кластерів. Якщо точка наразі не знаходиться в кластері з найближчим центроїдом, перенесіть цю точку на цей кластер і оновіть центроїд кластера, який отримує нову точку, і кластер, який втрачає точку.

Крок 4. Повторюйте крок 3, поки не буде досягнуто збіжності, тобто поки проходження навчальної вибірки не спричинить нових призначень.

На рис. 13. 4 показано, як відбувається міграція центроїдів і пов’язаних з ними окремих елементів даних після кожної ітерації. Квадратиками позначено елементи набору вхідних даних, колами – центроїди.

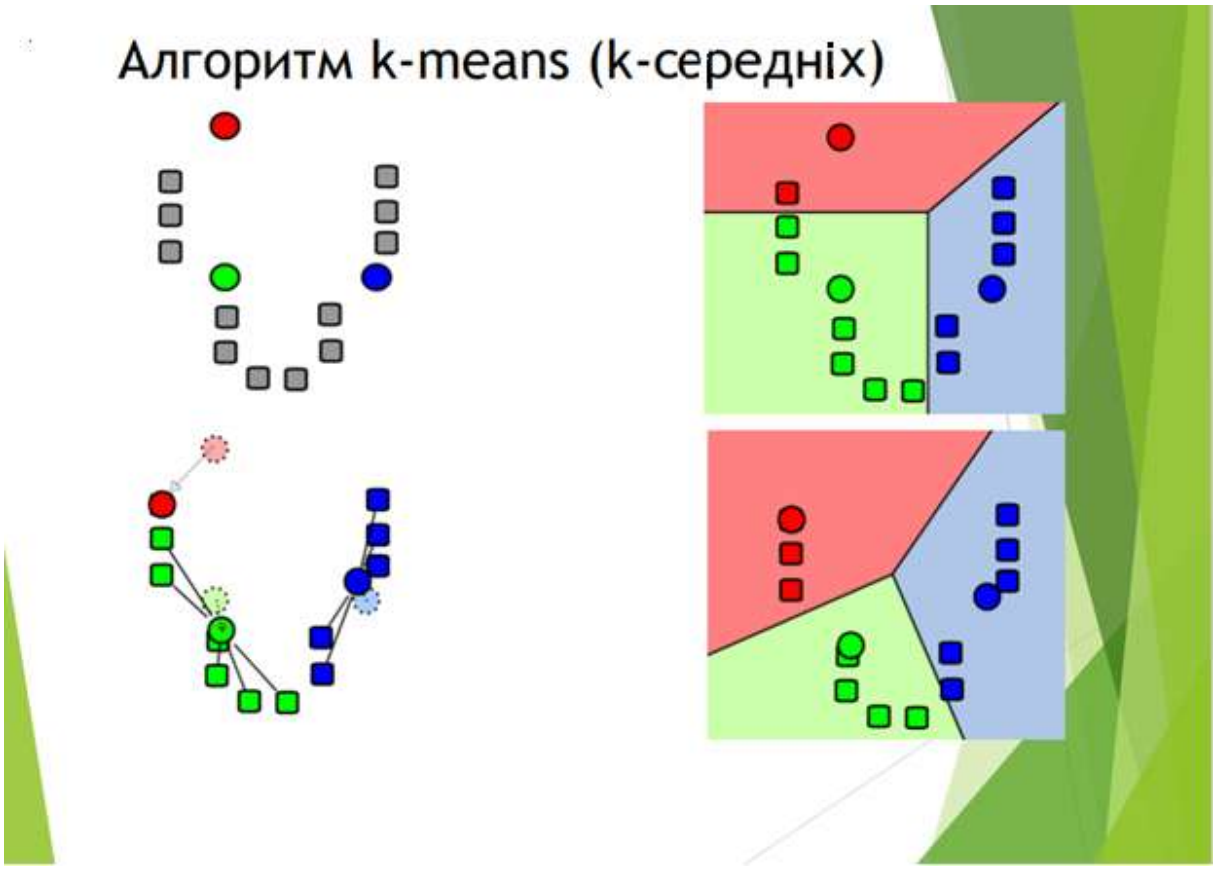

Рисунок 13.4 – Результати роботи алгоритму k-середніх після окремих ітерацій

Насамкінець вкажемо деякі особливості алгоритму k-середніх, які також можно вважати його недоліками:

- потрібно заздалегідь знати (задати) кількість кластерів;
- алгоритм чутливий до вибору початкових центрів кластерів;

- робота алгоритму може завершуватися після потрапляння в локальні оптимуми; отже, алгоритм не гарантує знаходження оптимального рішення (іншими словами, належить до субоптимальних алгоритмів).

Але вказані недоліки не є приводом для того, щоб відмовитися від алгоритму k-середніх. Навпаки, вони стимулювали пошук різноманітних модифікацій, здатних тією чи іншою мірою позбутися згаданих вище вад [21].

13.4 Контрольні питання

Поясніть значення терміна кластеризація.

З якою метою виконують кластеризацію даних?

Яке практичне застосування має кластеризація?

Які метрики використовуються для обчислення відстані в процесі роботи алгоритму k-середніх?

Який з варіантів обчислення відстані в алгоритмі k-середніх використовується найчастіше?

Як відбувається кластеризація за методом k-середніх?

Які недоліки притаманні алгоритму k-середніх?

14 ПОШУК НАЙБЛИЖЧИХ СУСІДІВ

14.1 Загальна характеристика задачі пошуку найближчого сусіда

Пошук найближчого сусіда є фундаментальною обчислювальною задачею і застосовується в багатьох сферах науки, зокрема, інтелектуальний аналіз даних (data mining), розпізнавання і класифікація образів (pattern recognition and classification), машинне навчання (machine learning), ущільнення даних (data compression), мультимедійні бази даних (multimedia databases), пошук документів і статистика. Завдання пошуку найближчого сусіда формулюється так: для заданого x потрібно знайти найбільш близький об'єкт в колекції об'єктів, або k ($k << n$, n – число всіх об'єктів в колекції) найбільш близьких об'єктів; або є множина точок в багатовимірному просторі, і потрібно попередньо сформувати з них таку структуру даних, яка б дозволила для заданого вектора під час запиту виконати пошук найближчого якомога ефективніше.

Для дуже високих розмірностей (d > 20) може бути використано LSH-хешування (locality sensitive hashing LSH), оскільки вищезгадані алгоритми в цьому випадку можуть виявитись навіть гіршими, ніж лінійний пошук даних з погляду обчислювальної ефективності. LSH-хешування – це наближений (імовірнісний) метод пошуку найближчого сусіда з лінійною залежністю від кількості елементів і ймовірнісною гарантією точності. Теоретично LSH забезпечує гарантію виконання пошуку за сублінійний час, на практиці це може бути складним завданням. Оскільки характеристики мають бути ретельно налаштовані: кількість комірок кожного хешу, кількість хешів, співвідношення ближньої і дальньої дистанцій та зіткнення ймовірностей, в результаті структура індексу може стати досить великою через багаторазове хешування кожної точки даних.

Ефективними підходами пошуку для помірно великих розмірностей (d < 20) та стандартних метрик є такі дерева бінарного розбиття простору (BSP), як KD-дерева (розбиття простору на гіперпрямокутні області) або VP-дерева (розбиття простору на гіперсферичні області) тощо. KD-дерево (англ. kd-tree, скорочення від k-вимірне дерево) – це бінарне дерево, в якому кожна вершина задає розбиття простору на два підпростори деякою площиною, що проходить через неї.

Виходячи з оцінювання верхньої межі складності обчислень доцільним виявляється створення структури даних розміром O(N) за O(NlogN) операцій і виконання пошуку у цій структурі за час, пропорційний O(logN). Для розмірності d = 1 це можливо шляхом сортування, а потім застосування бінарного пошуку для визначення найближчого сусіда. При d = 2 це також можливо шляхом застосування діаграми Вороного. Однак

для розмірностей d > 2 складність обчислень діаграми Вороного в найгіршому випадку зростає як $O(N^{[d/2]})$.

Friedman, Bentley, Finkel були першими, хто показав, що можна досягти того, щоб пошук на основі kd-дерев вимагав лише O(N) комірок пам'яті та O(logN) операцій під час пошуку найближчого сусіда. Для розмірностей $d < 20$ середній час пошуку найближчого вектора в корпусі, впорядкованому на основі kd-дерева, в середньому зростає пропорційно $O(h) = O(d \cdot \log(N+1))$, де h – висота дерева. Для більших розмірностей обчислювальна складність може досягати $O(d \cdot N)$. Зберігання kd-дерева вимагає $O(dN)$ пам'яті, а час, потрібний для його побудови, становить $O(N(d + \log(N))$.

Для збалансованих бінарних дерев час пошуку завжди зростає пропорційно $O(\log_2 n)$. Проте сама процедура балансування дерева має високу обчислювальну складність.

Задача наближеного пошуку сусідів активно досліджувалася. Ефективність алгоритмів оцінювалась за витратами пам'яті, часом, потрібним для побудови відповідної структури даних, і часом виконання пошуку. Показано, що kd-дерево є найефективнішим методом сортування та пошуку для багатовимірного простору, хоча і потребує дещо більшого часу на побудову, ніж, наприклад, швидке сортування.

14.2 Впорядкування векторів-центроїдів на основі kd-дерева

Ілюстрацію kd-дерева, кожна вершина якого задає розбиття простору на два підпростори деякою площиною, що проходить через неї вздовж осей (розмірностей) даних, для двовимірного простору показано на рис. 14.1.

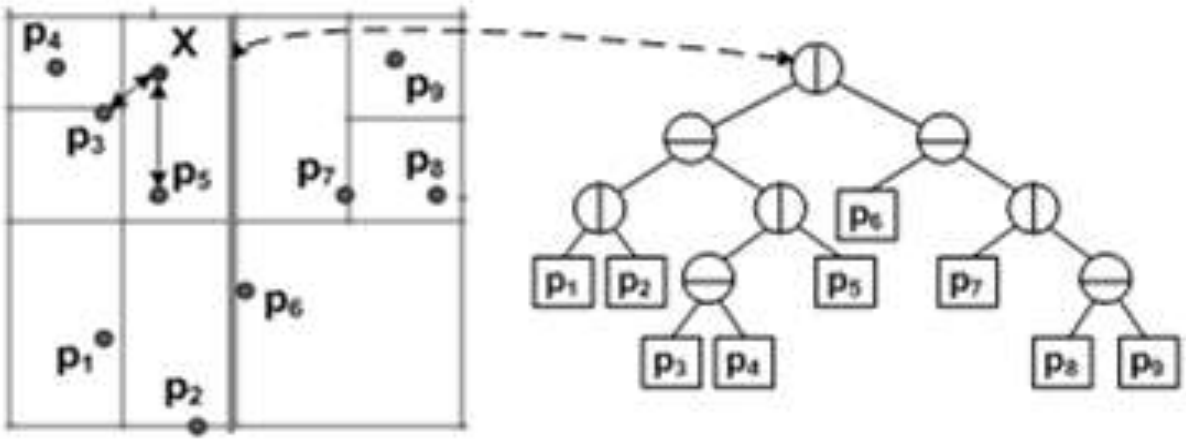

Рисунок 14.1 – Ілюстрація впорядкування центроїдів на основі kd-дерева

Зазначимо, що з кожним вузлом kd-дерева асоціюється d-вимірний прямокутник, який називається коміркою, а також ті точки (вектори), що лежать в його межах. Кореневому вузлу відповідає гіперпрямокутник, що охоплює всі точки (вектори). Точки, що лежать на межі гіперпрямокутників, можуть бути віднесені до будь-якого з них.

Отже, впорядкування векторів-центроїдів в базі параметрів на основі kd-дерева виконується дуже швидко, оскільки розбиття простору відбувається лише вздовж осей даних, відповідно, в такому разі не потрібно обчислювати d-вимірні відстані.

14.3 Пряма та зворотна фаза пошуку

Впорядкування бази параметрів на основі kd-дерева дозволяє суттєво зменшити кількість вимірювань, потрібних для пошуку найближчого центроїда. Однак пошук за kd-деревом не гарантує знаходження дійсно найближчого вектора-центроїда згідно з рисунком 14.1, оскільки математично задача пошуку найближчого елемента в структурі kd-дерева формулюється так: дано деякий багатовимірний вектор $\mathbf{x}$; потрібно знайти вершину kd-дерева $\mathbf{v}'$, щоб виконувалась умова

$$d(\mathbf{v}',\mathbf{x}) = \min\{d(\mathbf{v}_i,\mathbf{x})\}, i = (\overline{1,n}),$$

де $\mathbf{x}$ – вектор, для якого виконується пошук найближчого центроїда-вектора в корпусі;

$\mathbf{v}'$ – вершина дерева, відстань до якої найменша.

Тому для виконання пошуку найближчого сусіда було запроповано вдосконалену процедуру, яка, окрім прямої фази пошуку, має ще й зворотну фазу.

Такий пошук для двовимірного випадку показано на рис. 14.2.

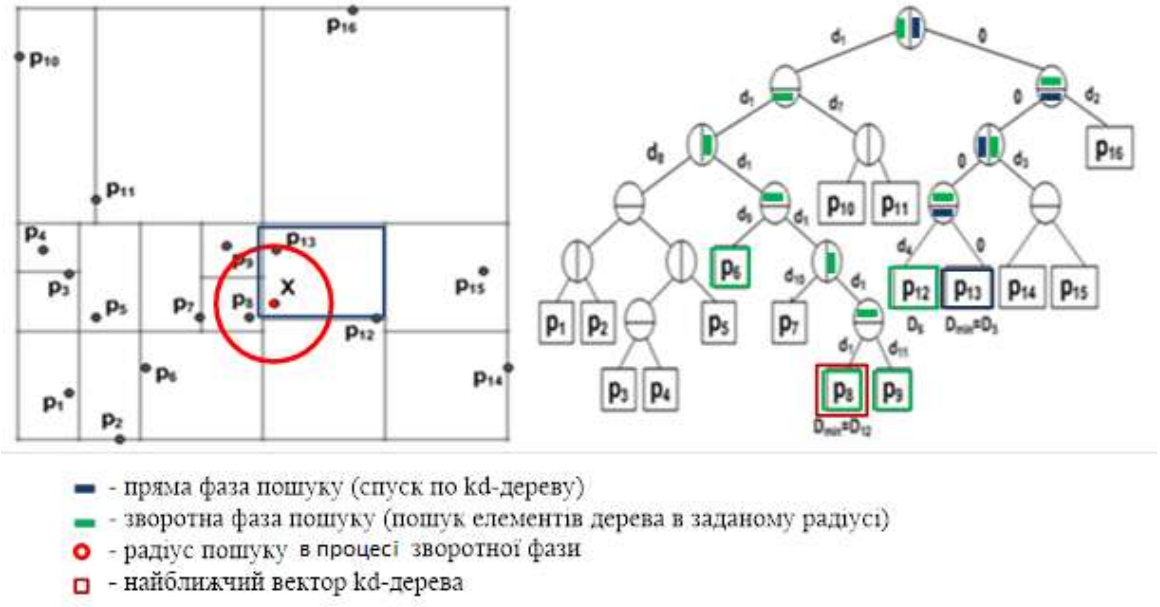

Рисунок 14.2 – Ілюстрація пошуку найближчого сусіда по kd-дереву для двовимірного випадку

Щоб забезпечити знаходження найближчого вектора, пошук, крім прямої фази пошуку (спуску по дереву), яка вимагає $O(\log_2 N)$ операцій, має мати також зворотну (пошук елементів дерева в заданому діапазоні (радіусі)). В процесі прямого пошуку фіксуються всі відстані до вузлів d_i та запам'ятовуються у порядку їхнього зростання, за рахунок чого під час зворотної фази здійснюється їхній обхід у порядку зростання відстані до **x**. Пряма фаза завершується обчисленням відстані $D_{\min} = D_k$ до відповідної термінальної вершини (на рисунку 14.2 – прямокутника), в якій лежить невідомий вектор, а також потенційно найближчий елемент дерева (центроїд в корпусі), що задає радіус пошуку під час зворотної фази. Після цього починається зворотна фаза пошуку, яка виконується рекурсивно, а саме: перш за все здійснюється повернення до батьківського вузла відносно поточного та виконується перевірка чи його інший дочірній вузол лежить у межах радіусу пошуку. Якщо відстань до дочірнього вузла перевищує радіус пошуку – відбувається перехід ще на один рівень вище. Інакше рекурсивно здійснюється перевірка його дочірніх вузлів.

Протягом зворотної фази здійснюється обхід вузлів у порядку зростання відстані d_i до них, причому лише тих, для яких $d_i < D_{\min}$. Якщо для відповідного листа виконується умова $D_k < D_{\min}$, радіус пошуку коригують $D_{\min} = D_k$. Таким чином буде здійснено обхід кожного листа дерева, відстань до якого менша, ніж до найближчого поточного вектора.

Розглянута процедура пошуку гарантує знаходження найближчого вектора, проте однозначно потребує більшої кількості вимірювань відстані, ніж $\log_2 N$.

14.4 Аналіз чинників впливу на продуктивність пошуку на основі kd-дерева

Швидкість пошуку можна оцінювати як число здійснених в процесі пошуку операцій порівняння (обчислення відстані) або ж як кількість перевірок комірок дерева, щоб знайти найближчий вектор.

Метод пошуку на основі kd-дерева має дві стадії пошуку (що було розглянуто раніше): пряму (спуск по дереву до листа) та зворотну, тому загальний час пошуку найближчого вектора також має дві складові

$$t_{search} = t_{down} + t_{up}.$$

Як було зазначено, в середньому потрібно виконати принаймні O(logN) перевірок, тому що будь-який пошук найближчого сусіда в kd-дереві вимагає спуску до термінальної вершини, в яку потрапляє невідомий вектор. Також зрозуміло, що потрібно не більше ніж N операцій порівняння (O(N)), оскільки кожен вузол може бути відвіданий не більше

одного разу. Точне визначення теоретичної кількості операцій є складним завданням, оскільки продуктивність пошуку залежить від кількох чинників, що є взаємозалежними та взаємовпливовими:

- відношення сторін комірок;
- розмір дерева – N;
- розмірність векторів параметрів – d ;
- коефіцієнт розповсюдження зворотної фази пошуку – *r*.

Продуктивність пошуку на основі kd-дерева залежить від способу розбиття простору на гіперпрямокутні ділянки, застосування того чи іншого способу формуваня комірок впливає на кількість обчислень відстані, якою фактично і визначається обсяг здійснених операцій у процесі пошуку. Наразі не існує такого правила розбиття простору на комірки, що забезпечив би всі бажані властивості: лінійний розмір структури даних, логарифмічну висоту дерева, обмеження на відношення сторін комірок, опуклість, константну складність комірок.

Згідно зі стандартним способом розбиття у процесі побудови kd-дерева комірка ділиться площиною перпендикулярно до осі координат, де точки мають найбільший розкид, точка поділу є медіаною по цій осі. Такий підхід, з одного боку, дозволяє побудувати збалансоване kd-дерево висотою *O(logN)*, однак призводить до формування великої кількості комірок з високими значеннями відношення найдовшої сторони до найкоротшої (the aspect ratio). Застосування такого підходу у процесі пошуку найближчого вектора призведе до зростання кількості вузлів, відстані до яких треба обчислити (тобто, зросте час пошуку).

Отже, з одного боку важливо забезпечити обмеження на відношення сторін (співвідношення максимальної та мінімальної сторін комірки) або на максимальну відстань між двома точками комірки, щоб зменшити кількість комірок, які треба обійти під час зворотної фази пошуку. З огляду на це найкращим варіантом розбиття слід було б вважати такий, коли всі комірки мають кубічну форму (відношення сторін 1:1), проте при такому способі побудови дерево може містити багато тривіальних листів і, відповідно, матиме занадто великий розмір. Це також зумовлює незбалансований розподіл векторів, що може призводити до виконання зайвих переходів по kd-дереву. З іншого боку, важливо забезпечити збалансоване розбиття точок в просторі, оскільки це дозволяє визначати висоту дерева та, відповідно, час доступу до листа (або час спуску по дереву під час прямої стадії пошуку) як $\log_2 n$. Проте це не вирішує проблему коливань часу пошуку на зворотній стадії.

Як приємний бонус від застосування зворотної фази пошуку – можливість за «невеличку додаткову платню» отримати не одного, а задану кількість n найближчих сусідів.

14.5 Діаграма Вороного

Потреба застосування зворотної фази пошуку дещо применшує враження (естетичне) від застосування kd-дерев для пошуку найближчого сусіда і наводить на думку: чи не існує іншого методу для розв'язання цієї задачі? Відповідь – так, це можна зробити за допомогою діаграми Вороного.

Діаграма Вороного складається з так званих локусів – регіонів, в яких присутні всі точки, які знаходяться ближче до заданої точки, ніж до всіх інших. У діаграмі Вороного локуси є опуклими багатокутниками.

Як будувати локус? За означенням він будуватиметься так: нехай дано множину з n точок, для якої ми будуємо діаграму. Візьмемо конкретну точку p, на яку будуємо локус, і ще одну точку з цієї множини – q (не рівну p) (рис. 14.3). Проведемо відрізок, що з'єднує ці дві точки, і проведемо пряму, яка буде серединним перпендикуляром цього відрізка. Ця пряма ділить площину на дві півплощини, а саме: в одній лежить точка p, в іншій – точка q. Наразі локусами цих двох точок є отримані півплощини. Тобто для того, щоб побудувати локус точки p, потрібно отримати перетин всіх таких півплощин (тобто на місці q побувають всі точки даної множини, крім p).

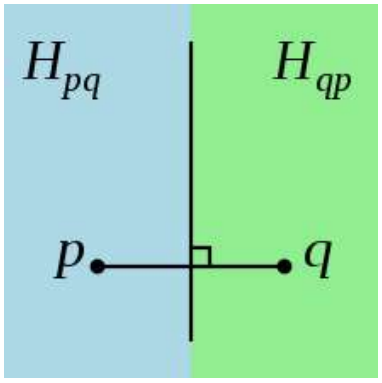

Рисунок 14.3 – Розбиття площини

Точку, для якої будується локус, називають **сайтом (site)**. На рис. 14.4 локуси помічено різними кольорами.

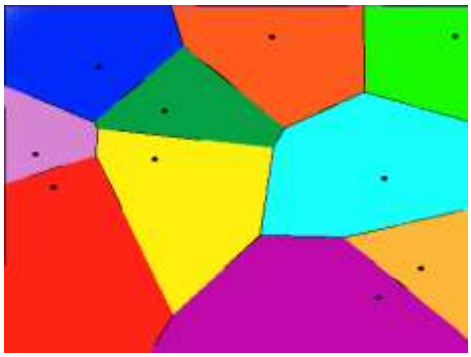

Рисунок 14.4 – Локуси діаграми Вороного

Алгоритми побудови діаграми якраз і є ні що інше, як алгоритми побудови цих локусів для всіх точок із заданого набору. Локуси у цій задачі також називають багатокутниками Вороного або комірками Вороного.

Зрештою, сформулюємо означення (дефініцію) діаграми Вороного n точок на площині (n – натуральне) – це розбиття площини, що складається з n локусів (для кожної точки по локусу).

На цьому сайті можна знайти цікавий візуалізатор діаграми Вороного: https://alexbeutel.com/webgl/voronoi.html. А почитати про великого українця Георгія Вороного можна тут: https://www.ukrinform.ua/rubric-culture/3226969-georgij-voronij-alhimia-u-diagrami-voronogo.html.

14.6 Алгоритм Форчуна побудови діаграми Вороного

Алгоритм базується на застосуванні методу замітання прямою, який було розглянуто у розділі 5. Замітальна пряма є горизонтальною прямою лінією, яка рухається зверзу донизу. На кожному кроці алгоритму діаграма Вороного побудована для множини, що складається з замітальної прямої та точок, розташованих над нею. Водночас межа між областю Вороного, прямою та областями точок складається з відрізків парабол (оскільки геометричне місце точок, рівновіддалених від заданої точки та прямої, – це парабола). Щоразу, коли пряма проходить через чергову точку, ця точка додається до вже побудованої ділянки діаграми. Додавання точки до діаграми при використанні бінарного дерева пошуку має складність lg(n), всього точок n, а сортування точок по x-координаті може бути виконано за n·lg(n), тому обчислювальна складність алгоритму Форчуна дорівнює O(n·lg(n)).

Цікаву візуалізацію та детальний опис алгоритму Форчуна наведено тут: https://blog.ivank.net/fortunes-algorithm-and-implementation.html.

14.7 Приклади застосування діаграми Вороного

В програмуванні, моделюванні та картографії

Існує важливий зв’язок діаграми Вороного з тріангуляцією Делоне, яка дозволяє будувати одне по іншому за O(n), тому що вони є двоїстими один до одного (з’єднуємо ребрами сусідні сайти, в результаті отримаємо тріангуляцію Делоне).

3D-сканування (і комп’ютерний зір) різних об’єктів теж може використовувати діаграму Вороного і тріангуляцію Делоне, також це тісно пов’язане з робототехнікою – рух робота з урахуванням перешкод на шляху.

Використовуються діаграми Вороного в геймдеві – система навігації в ігровому двигуні заснована на діаграмі.

Різний геолокаційний софт використовує діаграми Вороного. Геолокаційні рекомендаційні системи можуть використовувати діаграму Вороного для визначення, наприклад, найближчого до Вас продуктового магазину, для пошуку та аналізу розташування.

Тут можна згадати й застосування діаграми в картографії – для окреслення меж регіонів та подальшого аналізу на цій основі. Та й взагалі, будь-які географічні діаграми, що показують розподіл чогось, можна наочно проілюструвати за допомогою розфарбованих діаграм Вороного, і там буде видно перехід потрібного нам показника (наприклад, температури).

В архітектурі та дизайні

Дуже логічно, що людям на думку спала ідея використовувати діаграму Вороного в архітектурі та дизайні, оскільки вона сама по собі є гарним малюнком, свого роду «геометричним павутинням», так що є багато випадків застосування її як одного з основних елементів композиції.

В археології й екології

Тут багатокутники Вороного застосовуються для нанесення на карту ареалу застосування знаряддя праці у стародавніх культурах й вивчення впливу центрів торгівлі, що конкурують.

В екології можливості організму на виживання залежать від числа сусідів, з якими він має боротися за їжу та світло, що цілком логічно, адже, зазвичай, за будь-яке «виживання» борються саме сусідні регіони.

У біології та хімії

Спільний вплив електричних і близьких сил, для вивчення яких будуються складні діаграми Вороного, допомагає визначати структуру молекул.

Іноді природа також любить пожартувати (рис. 14.5).

Рисунок 14.5 – Граціозний жираф зробив татуювання
у вигляді діаграм Вороного

14.8 Контрольні питання

Яке практичне застосування має задача пошуку найближчого сусіда?

Які структури даних використовуються для пошуку найближчого сусіда?

Чому kd-дерева отримали таку назву?

Для яких розмірностей доцільно використовувати kd-дерева для пошуку найближчого сусіда?

Що таке гіперпрямокутник?

Яку форму мають мати гіперпрямокутники для ефективного пошуку найближчого сусіда?

Яке призначення має зворотна фаза в процесі пошуку найближчого сусіда?

Як забезпечується пошук не одного, а кількох найбличжчих сусідів і для чого це може знадобитися?

Що являє собою діаграма Вороного?

Поясніть значення терміна «локуси діаграми Вороного».

Як працює алгоритм Форчуна побулови діаграми Вороного?

Наведіть приклади практичного застосування діаграми Вороного.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kovalenko O., Palamarchuk Y. Algorithms of Blended Learning in IT Education // Conference: 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT) proceedings : 11-14 September 2018, Lviv, Ukraine. https://www.researchgate.net/publication/328815267_Algorithms_of_Blended_Learning_in_IT_Education.
2. Aho A., Hopcroft J., Ulman J. The Design and Analysis of Computer Algorithms. Pearson India, 2002. 450 p.
3. Вступ до алгоритмів / Кормен Т., Лейзерсон Ч., Рівест Р., Стайн К. К. : К. І. С., 2019. 1288 с. ISBN 978-617-684-239-2.
4. Гришанович Т. О. Курс лекцій з дисципліни «Алгоритми та структури даних» : курс лекцій. Луцьк : ВНУ імені Лесі Українки, 2021. 110 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. Арсенюк І. Р., Колодний В. В., Яровий А. А. Теорія алгоритмів : навч. посіб. Вінниця : ВНТУ, 2006. 150 с.
7. Богач І. В., Довгалець С. М., Дубовой В, М. Алгоритми розв’язання задач з програмування. Вінниця : ВНТУ, 2017. 119 с.
8. Мелешко Є. В., Якименко М. С., Поліщук Л. І. Алгоритми та структури даних : навчальний посібник для студентів технічних спеціальностей денної та заочної форм навчання. Кропивницький : Видавець Лисенко В. Ф., 2019. 156 с.
9. Махровська Н. А., Погромська Г. С. Алгоритми і структури даних : навчально-методичний посібник. Миколаїв : МНУ ім. В. О. Сухомлинського, 2019. 279 с
10. Прийма С. М. Теорія алгоритмів : навчальний посібник. Мелітополь : ФОП Однорог Т. В., 2018. 116 с.
11. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 2 : навчальний посібник. Львів : Видавництво «Новий Світ-2000», 2020. 320 с.
12. Сергієнко А. М., Марченко О. І. Конспект лекцій по курсу «Алгоритми і структури даних» для напряму підготовки 123 Комп’ютерна інженерія. К : Національний технічний Університет України «Київський Політехнічний Інститут імені Ігоря Сікорського», 2017. 74 с.
13. Спірінцева О. В., Литвинов О. А., Герасимов В. В. Java-технології та мобільні пристрої. Алгоритми і структури даних : навч. посіб. Дніпропетровськ : Вид-во ДНУ ім. О. Гончара, 2016. 140 с.
14. Ткачук В. М. Алгоритми та структура даних : навчальний посібник. Івано-Франківськ : Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016. 286 с.

15. C++. Алгоритмізація та програмування : підручник / Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. Одеса : Фенікс, 2019. 477 с.
16. Шаховська Н. Б., Голощук Р. О. Алгоритми і структури даних : навчальний посібник. Львів : Магнолія, 2018. 216 с.
17. Шевчук І. Б. Конспект лекцій з навчальної дисципліни «Алгоритмізація та програмування». Львів : Львівський національний університет ім. Івана Франка, 2018. 30 с.
18. Aibin M. Hash Table vs/ Balanced Binary Tree. URL: https://www.baeldung.com/cs/hash-table-vs-balanced-binary-tree (дата звернення: 15.05.2024).
19. Breadth First Search or BFS for a Graph. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph (дата звернення: 15.05.2024).
20. Depth First Search or DFS for a Graph. URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph (дата звернення: 15.05.2024).
21. Ткаченко О. М., Біліченко Н. О., Грійо-Тукало О. Ф., Дзісь О. В. Метод кластеризації на основі послідовного запуску k-середніх з обчисленням відстаней до активних центроїдів. *Реєстрація, зберігання і обробка даних*. 2012. Т. 14, № 1. С. 25–34.

Електронне навчальне видання

Олена Олексіївна Коваленко

Олександр Миколайович Ткаченко

Роман Юрійович Чехместрук

Алгоритми та структури даних

Навчальний посібник

Рукопис оформив *О. Ткаченко*

Редактор *В. Дружиніна*

Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 05.03.2025 р.
Гарнітура Times New Roman.
Зам. № Р2025-046.

Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
press.vntu.edu.ua;
Email: rvv.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.