МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

Г. В. Солодовник
О. О. Шаповалова

РОЗРОБКА ТА АНАЛІЗ АЛГОРИТМІВ

Навчальний посібник

Харків
ХНЕУ ім. С. Кузнеця
2023

УДК 004.021(075.034)
С60

Авторський колектив: канд. техн. наук, доцент Г. В. Солодовник – підрозд. 1 – 3; канд. техн. наук, доцент О. О. Шаповалова – підрозд. 4 – 6.

Рецензенти: провідний науковий співробітник відділу математичного моделювання й оптимального проектування Інституту проблем машинобудування ім. А. М. Підгорного НАН України д-р техн. наук, с. н. с. *А. М. Чугай*; доцент кафедри вищої математики та інформатики Харківського національного університету ім. В. Н. Каразіна, канд. фіз.-мат. наук *О. О. Аршава*.

Рекомендовано до видання рішенням ученої ради Харківського національного економічного університету імені Семена Кузнеця.

Протокол № 5 від 26.04.2023 р.

Самостійне електронне текстове мережеве видання

Солодовник Г. В.

С60 Розробка та аналіз алгоритмів [Електронний ресурс] : навчальний посібник / Г. В. Солодовник, О. О. Шаповалова. – Харків : ХНЕУ ім. С. Кузнеця, 2023. – 251 с.

ISBN 978-966-676-867-7

Наведено основні підходи до побудови алгоритмів і визначення їхньої складності. Розглянуто класичні алгоритми роботи зі складними динамічними структурами даних, наведено приклади роботи алгоритмів, контрольні запитання та завдання для самоперевірки, лабораторні роботи та контрольні запитання до них.

Рекомендовано для студентів спеціальності 125 «Кібербезпека» першого (бакалаврського) рівня всіх форм навчання.

УДК 004.021(075.034)

© Солодовник Г. В., Шаповалова О. О., 2023
© Харківський національний економічний університет імені Семена Кузнеця, 2023

ISBN 978-966-676-867-7

Вступ

Події останніх років, випробування, із якими зіткнулося людство загалом і українська спільнота зокрема, ставлять жорсткі вимоги до степеня безпеки передавання, зберігання й опрацювання даних у віртуальному просторі. Перехід бізнесу та банківських операцій до електронного формату дещо змістив напрям вектора подальшого розвитку суспільства в бік інформаційних технологій, підґрунтям яких є саме алгоритмічний підхід.

Фахівець у галузі кібербезпеки має знати основні принципи розроблення й оцінювання алгоритмів, уміти використовувати їх для забезпечення потрібного рівня безпеки комунікації, збереження даних та ведення бізнесу в умовах сьогодення.

Метою навчальної дисципліни «Розробка та аналіз алгоритмів» є надання здобувачам вищої освіти систематизованих знань теоретичних основ розроблення й оцінювання якості алгоритмів, а також формування в них відповідних компетентностей для застосування в інформаційних системах і мережах під час виконання завдань із забезпечення кібербезпеки.

Завданнями навчальної дисципліни «Розробка та аналіз алгоритмів» є такі:

дослідження методів формулювання та виконання завдань із розроблення алгоритмів;

аналіз принципів алгоритмізації та трудомісткості алгоритмів;

аналіз сутності алгоритмічного забезпечення інформаційних систем;

автоматизація виконання завдань з інформаційної безпеки;

побудова та впровадження математичних та обчислювальних моделей процесів опрацювання інформації;

оптимізація та визначення напрямів удосконалення.

Предметом навчальної дисципліни є науково-методичні засади розроблення та вибору алгоритмів для виконання завдань із кібербезпеки, а також сучасні методи побудови й аналізу алгоритмів із використанням ефективних способів зберігання, подання та перетворення інформації.

У результаті вивчення навчальної дисципліни студент має набути таких **компетентностей**:

здатність застосовувати знання в практичних ситуаціях;

здатність знаходити, опрацювати й аналізувати інформацію;

здатність використовувати інформаційно-комунікаційних технології, сучасні методи й моделі інформаційної безпеки та/або кібербезпеки;

здатність використовувати програмні й програмно-апаратні комплекси засобів захисту інформації в інформаційно-телекомунікаційних (автоматизованих) системах;

здатність забезпечувати неперервність бізнесу, згідно зі встановленою політикою інформаційної та/або кібербезпеки;

здатність забезпечувати захист інформації, що опрацьовують в інформаційно-телекомунікаційних (автоматизованих) системах, із метою реалізації встановленої політики інформаційної та/або кібербезпеки;

здатність відновлювати штатне функціонування інформаційних, інформаційно-телекомунікаційних (автоматизованих) систем після реалізації загроз, здійснення кібератак, перебоїв і відмов різних класів та походження;

здатність упроваджувати та забезпечувати функціонування комплексних систем захисту інформації (комплекси нормативно-правових, організаційних та технічних засобів і методів, процедур, практичних прийомів та ін.);

здатність здійснювати процедури управління інцидентами, проводити розслідування, давати їм оцінку;

здатність здійснювати професійну діяльність на основі впровадженої системи управління інформаційною та/або кібербезпекою;

здатність виконувати моніторинг процесів функціонування інформаційних, інформаційно-телекомунікаційних (автоматизованих) систем, згідно зі встановленою політикою інформаційної та/або кібербезпеки;

здатність аналізувати, виявляти й оцінювати можливі загрози, уразливості та дестабілізаційні чинники інформаційному простору й інформаційним ресурсам, згідно зі встановленою політикою інформаційної та/або кібербезпеки.

Результатами вивчення навчальної дисципліни є системні знання та практичні навички у сфері розроблення та застосування алгоритмічних моделей, методів побудови алгоритмів опрацювання даних, визначення складності алгоритмів, їхнього вдосконалення та оптимізації.

Розділ 1
Основи теорії алгоритмів

1. Поняття алгоритму

1.1. Теорія алгоритмів як математична наука

Теорія алгоритмів – це наука, що вивчає загальні властивості та закономірності алгоритмів, а також формальні моделі їхнього подання, завдяки яким стають можливими порівняння алгоритмів за ефективністю, перевірка їхньої еквівалентності, визначення галузей застосовності.

Теорія алгоритмів виникла як розділ математичної логіки й саме в ній набула розвитку та широкого застосування. Поняття алгоритму тісно пов'язано з поняттям числення. До того ж теорія алгоритмів є теоретичним фундаментом програмування, її використовують усюди, де мають місце алгоритмічні проблеми (основи математики, теорія інформації, теорія управління, конструктивний аналіз, обчислювальна математика, теорія ймовірності, лінгвістика, економіка тощо).

1.2. Визначення алгоритму та вимоги до нього

Є багато варіантів вербального визначення алгоритму, деякі з них звучать таким чином:

- Визначення 1: алгоритм – це будь-яка система обчислень, що виконують за строго визначеними правилами й після кінцевої кількості кроків приводить до вирішення поставленого завдання.
- Визначення 2: алгоритм – це точне розпорядження, що визначає обчислювальний процес і приводить від варійованих вихідних даних до шуканого результату.

Розв'язання певної математичної, інженерної, економічної задачі або задачі будь-якого іншого типу з використанням інформаційних технологій може бути подано низкою таких етапів:

1) формулювання умов задачі;
2) надання математичного опису;

3) обґрунтування вибору методу розв'язання задачі;

4) побудова алгоритму відповідного обчислювального процесу;

5) складання програми;

6) налагодження програми;

7) розв'язання задачі за допомогою ЕОМ;

8) аналіз та інтерпретація результатів.

Із позиції цієї роботи найбільш важливим і відповідальним із згаданих етапів є побудова алгоритму. Наведімо визначення для подальшої формалізації поняття «алгоритм».

Алгоритм – це поданий певною мовою кінцевий припис, який визначає скінченну послідовність елементарних операцій, що мають бути виконаними для розв'язання задачі. Такий припис має бути загальним для класу можливих первинних даних.

Припустімо D – множина початкових даних задачі, а R – множина можливих результатів розв'язання задачі, тоді визначають, що алгоритм здійснює відображення $D \rightarrow R$.

Опис послідовності дій алгоритму для ЕОМ потребує подальшої формалізації. Такою формалізацією є застосування синтаксису певної мови програмування. У цьому разі опис алгоритму є програмою.

Незважаючи на ступінь формалізації на метод подання, алгоритм має відповідати певним вимогам, які можна подати як загальні властивості алгоритмів:

1) *зрозумілість* – той, хто реалізує алгоритм, має за описом розібратися, як його виконувати;

2) *дискретність* – алгоритм має відображати процес розв'язання задачі як послідовність простих або раніше визначених кроків (етапів);

3) *результативність* – можливість визначення результату після виконання кінцевої кількості кроків;

4) *визначеність* – збіг визначених результатів у разі виконання алгоритму різними користувачами та за застосування різних технічних засобів;

5) *масовість* – можливість застосування алгоритму до цілого класу однотипних задач, що розрізняють значеннями початкових даних [1; 2; 6].

1.3. Типи алгоритмів та форми їхнього подання

Серед найбільш поширених структур алгоритмів виділяють такі типи:

1) *лінійний алгоритм*, дії якого виконують послідовно одна за одною;

2) *розгалужений алгоритм*, який реалізують різними шляхами, залежно від виконання (або невиконання) певної умови (запитання, на яке можна відповісти тільки «так» або «ні»);

3) *циклічний алгоритм*, дії якого повторюють.

Крім коду програми, до форм подання або засобів опису алгоритмів належать:

1) вербально-формульний опис: алгоритм записують у формі тексту з формулами й подають за пунктами, які визначають послідовність дій;

2) структурний або блок-схемний опис: алгоритм зображують блоками, у яких записують послідовність дій: блоки пов'язані між собою лініями зі стрілками – напрямками потоків;

3) опис із використанням граф-схем: початкова вершина графа-схеми відображає вхідні дані, фінальна – результат виконання, інші вершини відповідають окремим крокам виконання алгоритму;

4) опис із використанням мереж Петрі: складається з вершин (станів) і зв'язків (переходів), що відображають залежності між станами, кожен перехід має назву й опис, що пояснюють його функцію;

5) опис псевдокодом: це текстовий опис алгоритму, що використовує спеціальні команди та ключові слова для позначення кроків алгоритму. Псевдокод не має суворих правил синтаксису, його часто використовують для розроблення алгоритмів на ранніх стадіях;

6) опис за допомогою діаграми прецедентів: це графічний спосіб відображення взаємодії між користувачем і системою, що описує функціональність системи на основі вхідних та вихідних даних;

7) опис за допомогою моделі даних: це спосіб опису алгоритму, який використовує математичні моделі для відображення даних та їхньої взаємодії. Наприклад, ER-діаграми використовують для моделювання відношень між даними.

У подальшому для опису алгоритмів будемо використовувати блок-схемне подання. Табл. 1.1 містить основні види блоків та опис їхнього призначення.

Таблиця 1.1

Призначення елементів блок-схемного опису алгоритмів

№ з/п	Вигляд блоків	Опис призначення блоків
1	2	3
1	1 початок	Блок *початку*. Ініціалізує початок роботи алгоритму, має лише один (вихідний) потік
2	8 кінець	Блок *завершення*. Ініціалізує завершення роботи алгоритма, має лише один (вхідний) потік
3	2 F, K, q	Блок *уведення/виведення* даних. Призначено для введення даних користувачем із клавіатури або з інших файлів та виведення проміжних або кінцевих результатів роботи алгоритму, мусить мати один вхідний та один вихідний потоки
4	5 c = F + 2× a	Блок *операції*. Призначено для виконання арифметичних та інших операцій, або заздалегідь описаної функції чи процедури, мусить мати один вхідний та один вихідний потоки
5	8 x > – 5 ні так	Блок *умови*. Призначено для перевірки логічного виразу, із метою визначення подальшого напряму виконання процесу розв'язання, мусить мати один вхідний та два вихідні потоки: один відповідає значенню TRUE логічного виразу, а інший – значенню FALSE
6	4 i = 1, n 5	Блок *циклу*. Призначено для організації циклів, цей блок містить змінну, яка є лічильником циклу, початкове значення цієї змінної, кінцеве значення та крок зміни значення змінної, мусить мати один вхідний потік (згори), два вихідні: один прямує до блоків тіла циклу (знизу), інший – до блоків, які виконують після завершення циклу (праворуч), а також потік зациклення (ліворуч), який прямує до виконання наступного проходження в разі, якщо виконання циклічного процесу ще не завершено

Закінчення табл. 1.1

1	2	3
7	2 N Уводьмо кількість студентів N	*Коментар.* Призначено для внесення на блок-схему додаткової інформації для пояснень тих або тих дій
8	від 4 5 5 до 6	Позначки переносів блок-схеми. Використовують, якщо блок-схема не вміщується на одному аркуші. Може, крім номера блоку, містити номер сторінки, на якій цей блок розташовано

У теорії алгоритмів виділяють такі цілі та співвіднесені з ними завдання:

1) формалізацію поняття «алгоритм» і дослідження формальних алгоритмічних систем;

2) формальний доказ алгоритмічної нерозв'язуваності задач;

3) класифікацію задач, визначення та дослідження складності класів;

4) дослідження й аналіз рекурсивних алгоритмів;

5) визначення явних функцій трудомісткості, із метою порівняльного аналізу алгоритмів;

6) розроблення критеріїв порівняльного оцінювання якості алгоритмів.

1.4. Принципи побудови алгоритмів

По-перше, слід зазначити, що алгоритм застосовують до вихідних даних і він видає результати. Це означає, що алгоритм мусить мати входи (дані, які вводять) та виходи (дані, які виводять і є проміжними або кінцевими результатами роботи алгоритму). Під час роботи алгоритм генерує проміжні результати, які використовують у подальшій роботі. Тобто кожен алгоритм має три види даних: початкові, проміжні та вихідні.

У теорії алгоритмів фіксують конкретні кінцеві набори початкових елементарних об'єктів, а також кінцевий набір засобів утворення об'єктів

з елементарних даних. Набір елементарних об'єктів становить кінцевий алфавіт початкових символів (цифр, літер тощо), із яких утворюють усі інші об'єкти.

По-друге, дані для зберігання потребують пам'яті, яку зазвичай уважають однорідною та дискретною, тобто вона складається з однакових комірок. Кожну комірку призначено для зберігання одного символу алфавіту даних.

По-третє, алгоритм складається з окремих елементарних кроків, кількість таких кроків є обмеженою.

По-четверте, послідовність виконання кроків алгоритму є чітко визначеною: після кожного кроку вказують, який слід виконувати наступним. Крім того, має бути визначено умову, за виконання якої роботу алгоритму завершують.

По-п'яте, згідно із властивістю результативності, алгоритм після виконання певної кількості кроків має генерувати дані, які можна вважати результатом. Після визначення результату слід обчислити його збіжність, що є не тривіальною задачею, оскільки немає достатньо універсального методу такої перевірки.

По-шосте, різними етапами процесу алгоритмізації є такі:

1) опис алгоритму (наприклад у вигляді коду програми);

2) процес реалізації алгоритму на конкретному технічному забезпеченні, який передбачає такі засоби: запуску та зупинки, виконання елементарних кроків, надання результатів та управління ходом обчислення;

3) застосування алгоритму до конкретних початкових даних, тобто послідовність кроків, яку буде виконано за застосування алгоритму до різних початкових входів.

1.5. Математичні основи аналізу складності алгоритмів

Складність алгоритму визначають низкою різнопланових показників. Так, інтуїтивно можна виділити такі складові, як логічна, статична, часова та ємнісна складність алгоритмів [1].

Логічну складність визначають трудовитратами на розроблення алгоритму і вимірюють кількістю людиномісяців.

Статичну складність пов'язано з довжиною опису алгоритму, її вимірюють кількістю операторів, із яких складається алгоритм.

Часову складність визначають тривалістю виконання та вимірюють секундами, мінутами або годинами.

Ємнісну складність визначають кількістю умовних одиниць пам’яті, необхідних для роботи алгоритму.

Метою аналізу трудомісткості алгоритмів є пошук оптимального варіанта алгоритму для розв’язання конкретної задачі. За критерій оптимальності алгоритму вибирають його трудомісткість, тобто кількість елементарних операцій, які має бути виконано для розв’язання задачі за допомогою цього алгоритму. *Функцією трудомісткості* називають відношення, яке пов’язує вхідні дані алгоритму з кількістю елементарних операцій [7].

Характер залежності трудомісткості алгоритмів від вхідних даних може бути різним. Так, для одних алгоритмів трудомісткість залежить тільки від обсягу даних, для інших – від конкретних значень вхідних даних, навіть порядок надходження може впливати на неї. Здебільшого всі перелічені раніше чинники можуть певною мірою впливати на трудомісткість алгоритму.

Отже, вибір оптимального алгоритму для фахівця в галузі комп’ютерних технологій є не лише науково актуальним, але й фінансово обґрунтованим. Розроблення алгоритму потребує як кваліфікованих трудових ресурсів і часових витрат, так і фінансування з боку замовника розроблення. Одним зі спрощених видів аналізу, що використовують на практиці, є асимптотичний аналіз алгоритмів.

1.5.1. Асимптотичний аналіз

Асимптотичний аналіз здійснюють, із метою порівняння витрат ресурсів, потрібних різним алгоритмам для вирішення того самого завдання за великих обсягів вихідних даних [6]. В асимптотичному аналізі використовують оцінювання функції трудомісткості або складність алгоритму. Таке оцінювання дозволяє визначити, наскільки швидко зростає трудомісткість алгоритму зі збільшенням обсягу вихідних даних.

Наведімо основні оцінки складності алгоритму.

Основною оцінкою функції складності алгоритму $f(n)$ є оцінка Θ, тут n – величина обсягу даних або довжина входу. Уважають, що оцінка складності алгоритму $f(n) = \Theta(g(n))$, якщо для $g > 0$ та $n > 0$ є додатні

константи c_1, c_2, n_0, такі що $c_1g(n) \le f(n) \le c_2g(n)$ за умови, що $n > n_0$, інакше кажучи можна визначити такі c_1 та c_2, що за достатньо великих значень n функція f(n) буде вкладеною між $c_1g(n)$ та $c_2g(n)$.

У цьому разі вважають, що функція g(n) є асимптотично точною оцінкою функції f(n), оскільки за визначенням функція f(n) не відрізняється від функції g(n) із точністю до постійного множника.

Є неочевидним, що $\Theta(g(n))$ є не функцією, а множиною функцій, які визначають характер зростання з точністю до постійного множника.

Оцінка $\Theta(g(n))$ дає одночасно верхню та нижню оцінку зростання функції трудомісткості. Часто виникає потреба розглядати ці оцінки окремо.

Оцінка O(g(n)) є верхньою асимптотичною оцінкою трудомісткості алгоритму, отже вважають, що:

$$f(n) = O(g(n)),$$
$$\text{якщо } \exists\ c > 0,\ n_0 > 0:\ 0 \le f(n) \le c\, g(n),\ \forall\ n > n_0.$$

Запис O(g(n)), означає що f(n) належить класу функцій, які зростають *не швидше* за функцію g(n) із точністю до постійного множника.

Оцінка $\Omega()$ задає нижню асимптотичну оцінку зростання функції f(n) та визначає клас функцій, які зростають *не повільніше* ніж g(n) із точністю до постійного множника:

$$f(n) = \Omega(g(n)).$$
$$\text{якщо } \exists\ c > 0,\ n_0 > 0 : 0 \le cg(n) \le f(n),\ \forall\ n > n_0.$$

Асимптотичний аналіз алгоритмів має не тільки практичне, але й теоретичне значення. Так, наприклад, доведено, що всі алгоритми сортування, засновані на попарному порівнянні елементів, відсортують n елементів за час, не менший за $\Omega(n \log n)$.

Іншим видом аналізу, який використовують на практиці, є інтервальний. Проблема полягає в тому, що не завжди алгоритм, який має асимптотично оптимальну трудомісткість у сенсі оцінок O() або $\Theta()$, має кращі показники для реальної множини початкових даних за суттєвої різниці констант, які переховують за O() або $\Theta()$ асимптотичними оцінками.

Розв'язання цієї проблеми лежить у сфері практичного порівняльного аналізу обчислювальних алгоритмів та пов'язано з виконанням таких етапів:

детального аналізу трудомісткості алгоритмів, тобто визначення у явному вигляді функції трудомісткості (асимптотичної оцінки складності для порівняльного аналізу недостатньо);

порівняльного аналізу функцій трудомісткості претендентних алгоритмів, із метою вибору раціонального алгоритму розв'язання цієї задачі за реальних обмежень множини вихідних даних.

1.5.2. Часова складність алгоритму

Часова складність алгоритму є характеристикою його продуктивності, її визначають кількістю елементарних операцій, потрібних для його реалізації. Під час розрахунків уважають, що всі елементарні операції мають однакову тривалість.

Часову складність оцінюють для найгіршого випадку та визначають як максимальний час, потрібний алгоритму для опрацювання множини з n елементів [9].

Позначмо через f(n) функцію, яка за О-нотацією показує, як буде змінюватися обчислювальна складність алгоритму зі зміною кількості вхідних даних у найгіршому для алгоритму випадку. Часову складність алгоритму зазвичай визначають виразом O(f(n)), який означає, що час виконання алгоритму зростає з тією самою швидкістю, що й функція f(n).

Якщо час роботи алгоритму не залежить від обсягу вхідних даних, то його часову складність позначають як O(1). Якщо потрібно визначити вміст третього елемента масиву, для цього не варто ані запам'ятовувати елемента, ані проходити ними кілька разів, достатньо дочекатися третього елемента в потоці вхідних даних. Це буде результатом, на обчислення якого за довільної кількості даних потрібний один і той самий проміжок часу.

Лінійна складність O(n): якщо подвоєння розмірності задачі подвоює і час, який є потрібним для її розв'язання, мова йде про лінійну складність. Пошук найбільшого елемента в невідсортованому масиві, що потребує перегляду всіх n елементів масиву, або додавання/віднімання чисел, що містять n цифр, належать до алгоритмів лінійної складності.

Квадратична складність $O(n^2)$: якщо час роботи алгоритму зростає пропорційно квадрату кількості елементів, яку має опрацювати цей алгоритм, мова йде про квадратичну складність. Бульбашкове сортування, за якого виконують два вкладені цикли перебору масиву, належить до алгоритмів квадратичної складності.

Кубічна складність $O(n^3)$: якщо час роботи алгоритму зростає пропорційно кубу кількості елементів, яку має опрацювати цей алгоритм, мова йде про кубічну складність. Наприклад для опрацювання n елементів вхідних даних алгоритм виконує $n^3 + 10n$ умовних операцій. За зростання входу n на час роботи алгоритму значно більше буде впливати перший доданок n^3 (піднесення n до куба), ніж другий 10n (добуток розміру входу n на константу 10).

Позначмо через T(n) часову складність алгоритму. Значення функції T(n) залежить від розміру входу n та визначає максимальну кількість елементарних операцій, які виконує комп’ютер під час розв’язання задачі заданого розміру. Наприклад:

$$T(n) = 10n^2 + 5n^3.$$

Швидкість обрахунків різниться, залежно від технічних засобів, які використовують для реалізації алгоритму. Вона залежить від частоти та архітектури процесора, місткості пам’яті тощо. Тому функція часової складності хоча і є функцією часу, що визначають не одиницями часу, а кількістю операцій, що виконують на ідеальному комп’ютері. Одиниці вимірювання для функції часу T(n) точно не визначено.

Точне значення часової складності залежить від визначення елементарних операцій, які можуть бути такими:

а) арифметичними: додавання, віднімання, множення та ділення чисел;

б) порівняння: порівняння двох значень для визначення того, що вони дорівнюють одне одному або одне з них є меншим/більшим за інше;

в) надання: надання значення змінній або виразу;

г) зчитування та запису в пам’ять: зчитування значень із пам’яті або запис значень у пам’ять;

ґ) уведення-виведення: уведення даних із клавіатури або файлу, виведення даних на екран або у файл;

д) управління: умовні оператори, цикли, функції тощо;

е) побітові: побітове І (AND), побітове АБО (OR), побітове виключне АБО (XOR), негація (NOT), зсув бітів;

є) на машині Тьюрінґа:

переходу до іншого стану: машина Тьюрінґа може перейти з поточного стану до іншого стану на основі правил переходу;

запису на стрічку: машина Тьюрінґа може записувати символи на стрічку, змінюючи їхнє значення на позиції, де міститься головка;

читання зі стрічки: машина Тьюрінґа може читати символи зі стрічки на позиції, де міститься головка;

зсуву головки: машина Тьюрінґа може зміщувати головку вліво або вправо на одну позицію;

зупинки: машина Тьюрінґа може зупинитися після виконання певної кількості операцій;

перевірки на зупинку: машина Тьюрінґа може перевіряти, чи є вона в стані зупинки.

1.5.3. Асимптотична складність

Асимптотична складність визначає ступінь (швидкість) зростання часу роботи певного алгоритму. Для опису асимптотичної складності використовують О-нотацію – математичний запис, який дозволяє врахувати найбільш вагомі елементи функції часу [3].

Визначення: функція часу $T(n)$ має ступінь зростання $O(f(n))$, якщо є константи c та n_0, такі, що для всіх $n \geq n_0$ виконано нерівність $T(n) \leq cf(n)$.

Для функції $T(n) = 2n^2 + 3n^3$ за $n_0 = 0$ та $c = 5$ для всіх цілих значень $n \geq 0$ виконано нерівність $2n^2 + 3n^3 \leq 5n^3$. Тому ступінь зростання наведеної функції $T(n)$ буде мати третій порядок $O(n^3)$, а асимптотична складність буде дорівнювати $O(n^3)$:

$$n = 0: 2 \times 0^2 + 3 \times 0^3 \leq 5 \times 0^3;$$
$$n = 1: 2 \times 1^2 + 3 \times 1^3 \leq 5 \times 1^3;$$
$$n = 2: 2 \times 2^2 + 3 \times 2^3 \leq 5 \times 2^3.$$

Тоді $f(n)$ – це верхня межа швидкості зростання алгоритму.

О-нотація дозволяє не брати до уваги елементи, що є незначущими з позиції швидкості зростання складності алгоритму.

Правила аналізу алгоритмів:

1. Оператори, асимптотична складність яких дорівнює O(1):

- надання;
- читання;
- запису.

2. Час виконання умовного оператора складається із:

- часу обрахування логічної умови (зазвичай асимптотична складність дорівнює O(1));
- часу виконання тіла конструкції умови.

3. Час виконання послідовності операторів визначають за правилом суми.

4. Час виконання циклу визначають додаванням часу всіх ітерацій циклу, тобто:

- часу виконання операторів, що становлять тіло циклу;
- часу обрахунку умови припинення циклу (зазвичай асимптотична складність дорівнює O(1)).

Наприклад, асимптотична складність алгоритму становить O(1), отже, час виконання є константою і не залежить від розміру вхідних даних, для такого коду:

```
public int GetCount(int[] items)
{
return items. Length;
}
```

Або, інший приклад, асимптотична складність алгоритму становить O(n), отже, час виконання лінійно залежить від розміру вхідних даних, для наступного коду:

```
public long GetSum(int[] items)
{
long sum = 0;
foreach (int i in items
{
Sum += i;
  }
    return sum;
  }.
```

Розгляньмо кілька прикладів можливих варіантів **О-нотацій визначення асимптотичної складності**:

- $O(\log n)$ – час виконання відображає логарифмічна функція; наприклад, асимптотична складність алгоритмів пошуку на бінарному дереві дорівнює $O(\log n)$;
- $O(n \log n)$ – час виконання відображає логарифмічна функція з лінійно зростальним коефіцієнтом; наприклад, асимптотична складність алгоритму швидкого сортування дорівнює саме $O(n \log n)$;
- $O(n^2)$ – час виконання відображає квадратична функція; наприклад, асимптотична складність алгоритму бульбашкового сортування дорівнює саме $O(n^2)$;
- $O(n^3)$ – час виконання відображає кубічна функція; наприклад, асимптотична складність алгоритму з потрійним ступенем укладеності дорівнює саме $O(n^3)$;
- $O(2^n)$ – час виконання відображає експоненціальна функція; наприклад, асимптотична складність алгоритмів розв'язання задачі комівояжера та повного перебирання дорівнює саме $O(2^n)$.

Зупинімося детальніше на розгляді правил додавання та добутку.

Правило додавання: якщо частини програми мають час виконання $T_1(n)$ та $T_2(n)$ зі ступенем зростання $O(f(n))$ та $O(g(n))$, відповідно, то сума $T_1(n) + T_2(n)$ має ступінь зростання $O(\max(f(n), g(n)))$.

Правило добутку: якщо частини програми мають час виконання $T_1(n)$ та $T_2(n)$ зі ступенем зростання $O(f(n))$ та $O(g(n))$, відповідно, то добуток $T_1(n) \times T_2(n)$, має ступінь зростання $O(f(n) \times g(n))$.

З останнього правила випливає, що $O(c \times f(n))$ еквівалентно $O(f(n))$, де $c = const$. Наприклад, $O(200 \times n^2)$ еквівалентно $O(n^2)$.

Розгляньмо кілька прикладів поданих програмним кодом алгоритмів, із метою визначення ступеня зростання часу (асимптотичної складності).

Приклад коду на правило додавання (суми). Якщо основна програма викликає методи (процедури або функції) програми послідовно, то складність визначають за правилом додавання:

```
static void DoSlowly (int inum, int jnum, int knum)
{
          int  a = 0;
```

```
for  (int  i = 0;  i < inum;  i++)
{
     for  (int  j = 0;  j <  jnum;  j++)
{
for  (int  k = 0;  k < knum;  k++)
{
                              a ++;
}
}
}
Console. WriteLine(«a = {0}», a);
}
```

Ступінь зростання часу (асимптотична складність) методу DoSlowly дорівнює $O(n^3)$:

```
static  void  DoFastly (int inum, int jnum)
{
             int  b = 0;
for  (int  i = 0;  i < inum;  i++)
{
     for (int j = 0; j < jnum; j++)
{
                              b ++;
}
}

}
```

Ступінь зростання часу (асимптотична складність) методу DoFastly дорівнює $O(n^2)$:

```
static void Main (string[] args)
{
             DoSlowly(1, 1, 1);
DoFastly(2, 2);

Console. WriteLine(«Done»);
```

```
Console. ReadKey( );

}
```

Основна програма Main викликає ці методи по черзі, тобто *послідовно*, отже, загальну асимптотичну складність розраховують за правилом додавання:

$$O(n^3) + O(n^2) = O(\max O(n^3), O(n^3)) = O(n^3).$$

Приклад коду на правило добутку (множення) [6]. Якщо метод викликають усередині другого, або, наприклад, усередині циклу, то асимптотичну складність визначають за правилом добутку.

Наведімо приклад. Коди методу DoSlowly та програми Main залишмо без змін, а код методу DoFastly перепишімо таким чином:

```
static void DoFastly (int inum, int jnum)
{
        int  b = 0;
for  (int  i = 0; i < inum;  i++)
{
    for (int j = 0; j < jnum; j++)
{
                    DoSlowly(5, 6, 7);
}
}

}
```

Метод DoFastly викликає метод DoSlowly *всередині циклів*, які є вкладеними один у другий, отже, ступінь зростання часу (асимптотична складність) програми Main будуть визначати за правилом добутку:

$$O(n^3) \times O(n^2) = O(n^5).$$

Приклад коду для аналізу складності. Визначмо розрахунковий час виконання алгоритму бульбашкового сортування. Кількість елементів

масиву array[n], який необхідно відсортувати, визначають розміром масиву вхідних даних. Оператори надання мають сталий час виконання, який не залежить від обсягу вхідної інформації:

```
1    static  int []  Sort (int[] arr )
2    {
3         for  (int  i  =  0;  i  <  arr. Length  -  1;  i++)
4         {
5              for  (int  j  =  0;  j  <  arr. Length  -  i  -  1;  j++)
6              {
7                   if  (arr[j]  >  arr[j  +  1])
8                   {
9                        int  buf  =  arr[j];
10                       arr[j]  =  arr[j  +  1];
11                       arr[j  +  1]  =  buf;
12                  }
13             }
14        }
15        return  arr;
16   }
17
18   static  void  Main(string[]  args)
19   {
20        int[]  array  =  new int[]  ( 5,  4,  3,  2,  1 );
21        for  (int  i  =  0;  i  <  arr. Length;  i++)
22        {
23             Console. WriteLine(array[i]);
24        }
25   }
```

За O-нотацією оператори у рядках (9) – (11) мають сталий час виконання O(1), який дорівнює деякій константі та не залежить від розміру вхідних даних. Тоді за правилом суми асимптотична складність цієї групи операторів становить O(max(1,1,1)) = O(1).

Оскільки оператори if та for є вкладеними один у другий, визначення асимптотичної складності слід здійснювати від внутрішнього оператора до зовнішнього. Послідовно визначаймо складність умовного оператора для кожної ітерації циклу. Для оператора if перевірка логічної умови

займає час порядку O(1). Отже, час виконання рядків (7) – (12) також становить O(1).

Оцінімо асимптотичну складність групи операторів внутрішнього циклу (рядки (5) – (15)), загальний час виконання яких розраховують додаванням часу виконання кожної ітерації циклу. Для операторів (7) – (12) ступінь складності за кожної ітерації становить O(1), цикл повторюють n − i разів. За правилом добутку загальний час виконання циклу становить $O((n - k) \times 1)$ і дорівнює $O(n - k)$.

Час виконання зовнішнього циклу (рядки (3) – (14)), де розміщено всі оператори, що виконують у програмі, розраховують додаванням часу виконання всіх ітерації внутрішнього циклу та їх може бути подано як суму елементів арифметичної прогресії такого виду:

$$(2 \times a_1 + (n - 1) \times d) / 2 \times n, \qquad (1.1)$$

де a_1 – перший член прогресії, $(n - 1)$;

n – кількість елементів;

d – крок прогресії, що дорівнює –1.

Кількість елементів масиву у прикладі дорівнює 5, тоді ступінь зростання складності можна подати прогресією такого виду:

$$(n - 1) + (n - 2) + (n - 3) + (n - 4). \qquad (1.2)$$

З урахуванням (1.1) і (1.2) час виконання алгоритму визначають такою формулою:

$$T(n) = (2 \times (n - 1) + (n - 1) \times (-1)) / 2 \times n = n^2 / 2 - n / 2. \qquad (1.3)$$

Асимптотична складність у цьому разі дорівнює $O(n^2)$. Отже, алгоритм бульбашкового сортування виконують за час, пропорційний квадрату кількості елементів масиву, який слід упорядкувати.

Складність алгоритму оцінюють за допомогою функції часу виконання T(n) за ігнорування констант функції. У загальному випадку алгоритм з асимптотичною складністю $O(n^2)$ буде кращим за алгоритм з аналогічним показником, що дорівнює $O(n^3)$. За однакових комбінацій програма-комп'ютер алгоритм із часом виконання $5n^3$ мс може завершити

виконання швидше за алгоритм із часом виконання 100 n^2 мс, але лише за умови n < 20 (рис. 1.1).

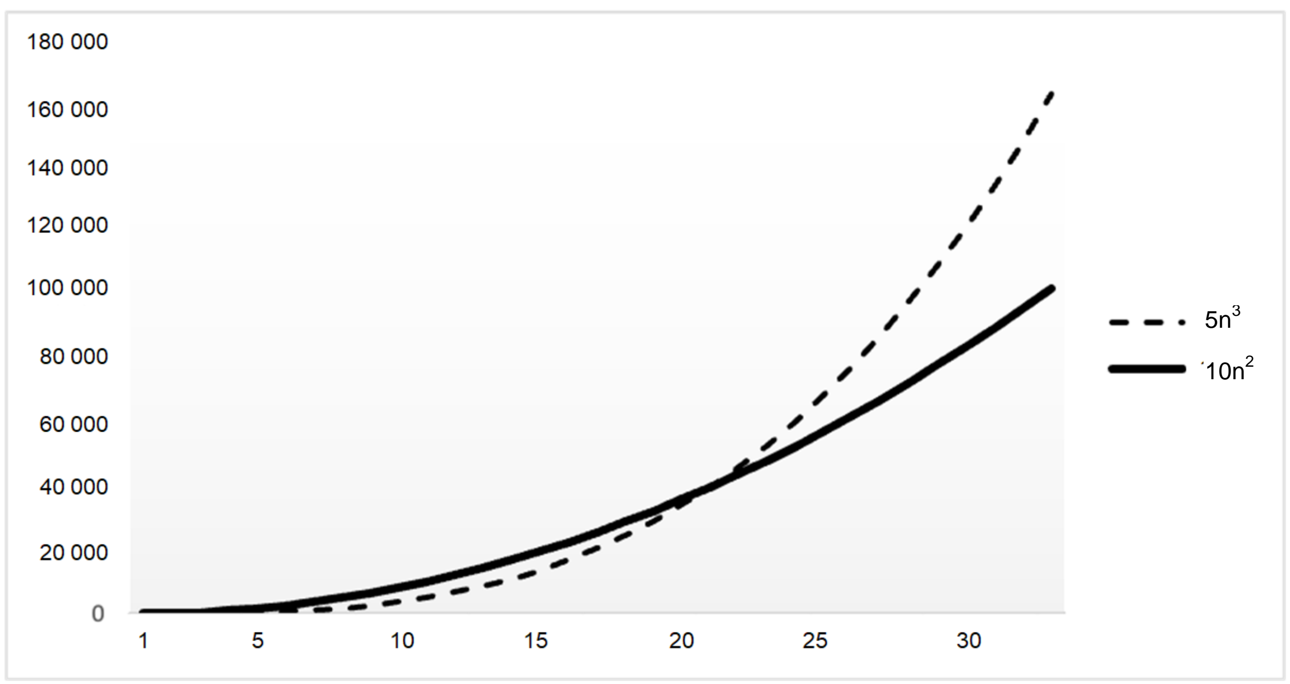


Рис. 1.1. **Залежність часу виконання алгоритмів від розміру вхідних даних**

Контрольні запитання і завдання для самоперевірки

1. Дайте визначення алгоритму.
2. Із яких етапів складається процес розв'язання задач для розробника ПЗ?
3. Які типи структур алгоритмів ви знаєте?
4. Які форми подання алгоритмів ви знаєте? Наведіть різні форми подання на прикладі одного алгоритму.
5. Назвіть блоки структурного опису алгоритмів.
6. Назвіть принципи побудови алгоритмів.
7. Що розуміють під складністю алгоритму і що на неї впливає?
8. Що впливає на трудоміскість алгоритму? Що розуміють під функцією трудомісткості?
9. Які способи оцінювання функції складності алгоритму ви знаєте?
10. Які задачі розв'язують за допомогою асимптотичного аналізу?
11. Яким чином можна визначити верхню та нижню оцінки зростання трудомісткості алгоритму?
12. Яку функцію виконує O-нотація?

13. Дайте визначення часової складності алгоритму.

14. Наведіть приклади алгоритмів, часову складність яких описано як O(1), O(n), O(n^2) або O(n^3).

15. Якщо часова складність алгоритму є квадратичною, то що це означає в разі збільшення розмірності задачі?

16. Що визначає асимптотичну складність алгоритму?

17. Як визначають час виконання послідовності операторів? Умовного оператора? Операторів циклу?

18. Сформулюйте правило додавання.

19. Сформулюйте правило добутку.

Лабораторна робота 1
Побудова та аналіз алгоритмів обчислювальних процесів

Мета. Ознайомитися зі способами алгоритмізації лінійних, розгалужених і циклічних обчислюваних процесів, навчитися будувати відповідні алгоритми розв'язання математичних задач.

Рекомендації з підготовки до виконання. Для успішного виконання лабораторної роботи (ЛР) студент має знати мету виконання роботи, порядок її виконання та загальні теоретичні положення; уміти будувати блок-схеми лінійних, розгалужених та циклічних обчислювальних процесів.

Завдання до виконання. Проаналізуйте формулювання задач за індивідуальними варіантами (додаток А табл. А.1 – А.3), обґрунтуйте вибір типу алгоритму для її розв'язання, формалізуйте обчислювальний процес розв'язання математичної задачі та побудуйте блок-схеми обчислювальних процесів.

Загальні теоретичні положення

Під час складання алгоритмів слід орієнтуватися на виконання таких етапів:

1. Формулювання умов початкової задачі: перед складанням алгоритму необхідно чітко сформулювати умови задачі та визначити, якими є вхідні та вихідні дані.

2. Аналіз вхідних даних: перед тим як розпочати складати алгоритм, потрібно здійснити аналіз вхідних даних та визначити, які дані є необхідними для розв'язання цієї задачі.

3. Декомпозиція задачі на підзадачі: у разі, якщо задача є великою та складною, слід розподілити її на менші підзадачі, виконання яких є більш зрозумілим і простим.

4. Вибір оптимальних алгоритмічних способів: для кожної підзадачі потрібно вибрати оптимальний алгоритмічний спосіб, що дозволить ефективно розв’язати задачу.

5. Створення послідовності дій: на основі вибраних алгоритмічних способів слід скласти послідовність дій, які потрібно виконати для розв’язання кожної підзадачі.

6. Перевірка алгоритму: перед тим як увести алгоритм в експлуатацію, потрібно перевірити його на тестових даних, щоб переконатися у його правильності й ефективності (цей етап зазвичай передбачає використання інформаційних систем).

7. Документування алгоритму: для того щоб інші користувачі зрозуміли алгоритм, необхідно створити документацію, яка описує послідовність дій та пояснює кожен крок.

Розгляньмо алгоритмізацію *лінійного обчислювального процесу*, який характеризується тим, що кроки, на які його розподіляють, виконують послідовно в тому порядку, у якому їх подано.

Лінійна структура – це найпростіший тип алгоритму, у ході виконання якого відтворюють послідовне виконання блоків алгоритму. Об’єктом опису в схемах алгоритмів є константи, змінні величини, арифметичні та логічні вирази.

Обчислювальний процес є лінійним, якщо його операції виконують послідовно, у порядку їхнього запису або подання на блок-схемі. У лінійному обчислюваному процесі кожна операція є самостійною одиницею, виконання якої не залежить від жодної вимоги.

Лінійні обчислювальні процеси застосовують під час обчислення арифметичних виразів. **Арифметичний вираз** – це сукупність констант, змінних і функцій, пов’язаних між собою знаками арифметичних операцій. **Змінна** – це елемент програми або алгоритму, який має ідентифікатор (унікальне ім’я (назву)) та призначений для зберігання, корекції та передавання даних. **Константа** – це незмінна величина визначеного типу. Під час запису арифметичного виразу використовують стандартні математичні функції.

Послідовність виконання операцій в арифметичному виразі така:

1) операції у дужках;

2) обчислення функцій;

3) піднесення до степеня;

4) множення та ділення;

5) додавання та віднімання.

Виконання операцій з однаковим пріоритетом відбувається зліва направо.

Розгляньмо алгоритмізацію *розгалуженого обчислювального процесу*, у якому реалізація операції залежить від виконання певних умов, які ґрунтуються на значеннях вхідних даних або проміжних результатах. За реалізації розгалуженого обчислювального процесу передбачено кілька напрямків, кожен із яких є окремою гілкою. Алгоритми розгалужених обчислювальних процесів є структурами, які мають декілька варіантів обчислень. Кожен із варіантів подано окремою гілкою та вибрано, залежно від виконання (або невиконання) певної умови.

Виконання умови зазвичай перевіряють із застосуванням логічних виразів. **Логічний вираз** – це вираз, який містить логічні значення (true або false) та логічні оператори, що дозволяють обчислювати нові логічні значення на основі наявних. Логічні вирази використовують у програмуванні для ухвалення рішень на основі певних умов або відношень між даними. У логічних виразах використовують такі логічні оператори:

Логічне «І» (AND): повертає значення true, якщо всі операнди мають значення true.

Логічне «АБО» (OR): повертає значення true, якщо хоча б один з операндів має значення true.

Логічне «НЕ» (NOT): повертає значення, протилежне до значення операнда.

Логічне «НЕ І» (NAND): повертає значення false, якщо всі операнди мають значення true.

Логічне «НЕ АБО» (NOR): повертає значення false, якщо хоча б один з операндів має значення true.

Логічне «виключне АБО» (XOR): повертає значення true, якщо тільки один з операндів має значення true.

Логічний вираз може містити також порівняльні оператори (>, <, =, != тощо), які порівнюють значення двох операндів і повертають логічне значення true або false, залежно від результату порівняння.

Якщо розгалужений процес складається лише із двох гілок, то він є простим, якщо з більше ніж двох гілок – складним. Складний розгалужений процес можна подати за допомогою кількох простих.

Є два типи логічних відношень: еквівалентності та порядку. Бінарне відношення перевіряє, чи рівними є два вирази, тобто чи мають вони однакові значення («x == y» поверне true, якщо x і y мають однакові значення). Бінарне відношення порядку використовують для порівняння двох виразів за допомогою таких операторів: більше «>», менше «<», більше або дорівнює «>=», менше або дорівнює «<=».

Блок-схема алгоритму розгалуженого обчислюваного процесу завжди містить хоча б один блок умови, у якому перевіряють виконання логічної умови. Логічні умови можуть об'єднувати в логічні вирази за допомогою наведених раніше логічних операторів.

Розгляньмо алгоритмізацію *циклічного обчислювального процесу*, який передбачає багаторазове виконання однієї операції за різних первинних даних.

Цикл – це фрагмент обчислювального процесу, який багаторазово повторюють за різних первинних даних, доки не буде виконано наперед задану умову.

Тіло циклу – це послідовність операторів, яку повторюють.

Лічильник циклу – це змінна, значення якої змінюють за кожного повторення циклу, та яка визначає момент завершення обчислень.

Параметр циклу – арифметичний вираз усередині циклу, який не містить керівної змінної (лічильника циклу).

Для організації циклу слід передбачити такі значення:

1) початкове та кінцеве значення лічильника циклу;

2) операцію зміни значення лічильника під час кожного повторення циклу та крок зміни;

3) умову завершення циклу, якою може бути:

а) перевищення лічильником циклу свого кінцевого значення;

б) виконання заданої кількості повторень;

в) досягнення завданої точності розрахунків.

Розрізняють арифметичні й ітераційні цикли.

Арифметичні цикли – цикли, у яких кількість повторень є відомою заздалегідь або може бути визначеною на підставі закону зміни лічильника циклу.

Ітераційні цикли – цикли, у яких кількість повторень тіла циклу заздалегідь є невідомою, а цикл повторюють доти, доки не буде виконано умову виходу із циклу.

Алгоритм, тіло циклу якого містить ще один цикл називають *алгоритмом із укладеним циклом*. Цикл, у тіло якого вкладено команди іншого циклу, називають *зовнішнім циклом*, а цикл, який укладено – *укладеним* або *внутрішнім циклом.*

Укладеними в циклічних процесах можуть бути не лише інші цикли, але й розгалуження та фрагменти лінійного типу.

Порядок виконання роботи

Розгляньмо процес побудови алгоритмів на конкретних прикладах.

Завдання 1.1. Складіть блок-схему алгоритму для обчислення та виведення змінних *M, a*.

$$M = (K + 4F) / (c + a),$$
$$c = F + 2a,$$
$$a = K + q + b,$$
$$b = 21{,}4.$$

Виконання завдання 1.1.

Аналіз завдання. Формулювання завдання не передбачає перевірки виконання будь-яких умов або повторення операцій розрахунків, тому обчислюваний процес є лінійним.

Аналіз первинних даних. До того як розпочати будувати блок-схему алгоритму, слід надати відповіді на низку запитань.

1. Значення яких параметрів визначено (тобто є константами в цьому завданні)?

Відповідь: *b* = 21,4.

2. Значення яких змінних має ввести користувач (це мають бути змінні, для розрахунку значень яких у завданні не передбачено формул)?

Відповідь: *F, K, q*.

3. У якому порядку розраховують значення інших змінних?

Відповідь: *1 – a, 2 – c, 3 – M.*

4. Значення яких змінних слід вивести для користувача?

Відповідь: *a, M.*

Побудова блок-схеми. Блок-схему алгоритму лінійного обчислюваного процесу зображено на рис. 1.2. У наведеній схемі блоки 1 та 8 є блоками початку та кінця алгоритму, відповідно, блоки 2 та 7 – уведення/виведення даних, блок 3 – надання, 4 – 6 – розрахунків.

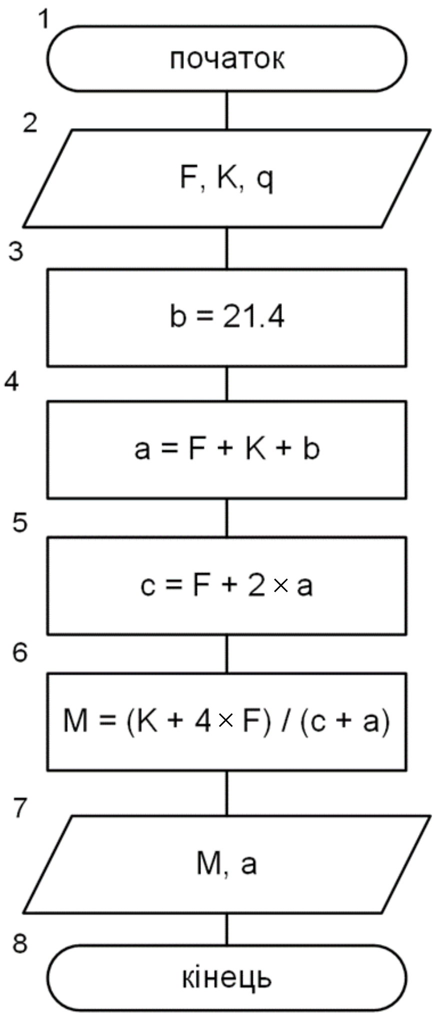


Рис. 1.2. **Алгоритм лінійного обчислюваного процесу**

Завдання 1.2. Складіть схему алгоритму та виведіть значення змінних *y, Z*.

$$y=\begin{cases} Ln(x) + a, \text{ якщо } x > 0 \text{ та } x < 1, \\ x^2, \text{ якщо } 1 \le x, \\ x^3, \text{ якщо } -5 < x < 0, \\ x, \text{ в інших випадках.} \end{cases}$$

$$Z=\begin{cases} 1 + y, \text{ якщо } y > 5, \\ 2, \text{ якщо } y = 0, \\ y^3, \text{ якщо } 0 < y \le 5, \\ 4 + y^4, \text{ якщо } y < 0. \end{cases}$$

Виконання завдання 1.2.

Аналіз завдання. У формулюванні завдання для розрахунку *y* та *Z* передбачено кілька варіантів. Вибір формули, за якою слід обчислювати

значення кожної із цих змінних залежить від того, яку з умов виконано, тому для виконання доцільно вибрати розгалужений алгоритм.

Аналіз первинних даних. До того як розпочати будувати блок-схему потрібно дати відповіді на низку запитань. Крім відповідей на запитання для лінійного алгоритму, слід з’ясувати, які умови пов’язано оператором логічного додавання (логічне АБО), а які – оператором логічного множенням (логічне І).

У разі, якщо умови пов’язано оператором логічного додавання ($x > -5$ АБО $x < 0$), то результат можна подати у такому вигляді:

Умова 1	**Умова 2**	**Результат**
ІСТИНА	ІСТИНА	ІСТИНА
ІСТИНА	ХИБНО	ІСТИНА
ХИБНО	ІСТИНА	ІСТИНА
ХИБНО	ХИБНО	ХИБНО

Побудова блок-схеми. За допомогою блок-схеми це подають таким чином (рис. 1.3).

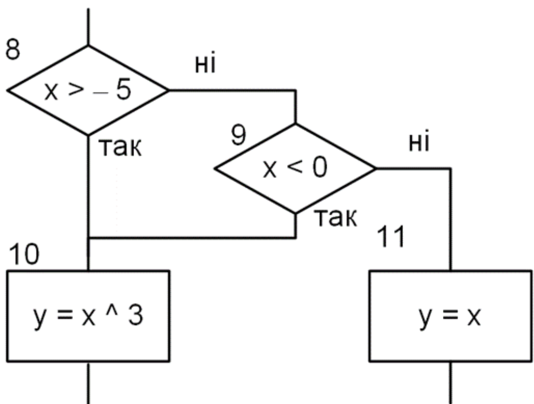


Рис. 1.3. **Фрагмент алгоритму із застосуванням логічного АБО**

За виконання умови в блоці 8 немає потреби перевіряти другу частину логічного виразу (умову в блоці 9), оскільки за виконання хоча б однієї з умов вираз загалом виконано, отже, слід розраховувати *y* за формулою блоку 10.

Якщо умову в блоці 8 не виконано, потрібно перевірити виконання другої частини логічного виразу, яку записано в блоці 9. Лише в разі, якщо її також не виконано, *y* слід розраховувати за формулою, яку вказано в блоці 11. У разі, якщо умову в блоці 8 виконано, *y* слід розраховувати за формулою, яку вказано в блоці 10.

Якщо логічні умови поєднано оператором логічного множення (логічне І) ((*x* > −5 *І x* < *0)* або *(*−*5* < *x* < *0*)), результат можна подати в такому вигляді:

Умова 1	**Умова 2**	**Результат**
ІСТИНА	ІСТИНА	ІСТИНА
ІСТИНА	ХИБНО	ХИБНО
ХИБНО	ІСТИНА	ХИБНО
ХИБНО	ХИБНО	ХИБНО

За допомогою блок-схеми це подано таким чином (рис. 1.4).

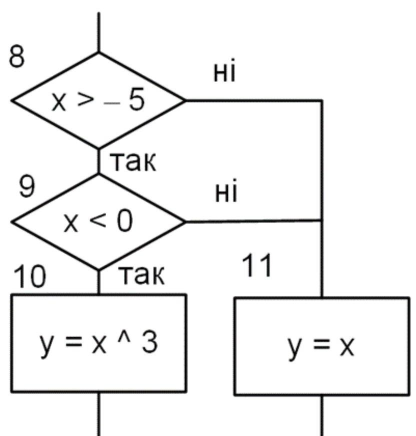


Рис. 1.4. **Фрагмент алгоритму із застосуванням логічного І**

За поєднання логічних умов оператором логічного множення, якщо умову в блоці 8 не виконано, то в разі логічного множення цього достатньо, щоб зробити висновок щодо хибності виразу загалом. Тому, якщо умову в блоці 8 не виконано, *у* розраховують за формулою в блоці 11. У разі виконання умови в блоці 8, слід перевірити другу частину умови (блок 9). Хибність виразу в блоці 9 також призводить до хибності виразу загалом та розрахунку *у* за формулою в блоці 11. Якщо умову в блоці 9 виконано, то виконують і вираз загалом і *у* розраховують за формулою в блоці 10.

Перевірка виконання умов за побудови блок-схеми алгоритму розгалуженого обчислюваного процесу відбувається в блоках перевірки логічних умов, які мусять мати один вхідний потік та два вихідні:

«так» (true або «+») – потік, який відповідає виконанню умови, що міститься в блоці;

«ні» (false або «−») – потік, який відповідає невиконанню умови, що міститься в блоці.

Зауваження. Після побудови блок-схеми розгалуженого алгоритму слід перевірити чи всі вихідні потоки блоків перевірки умов описано: мають логічне завершення та ведуть до кінця алгоритму.

Після розрахунку значення змінної *y* за будь-якою з формул його потрібно вивести для користувача. Потім слід розрахувати значення змінної *Z*, перевіряючи вказані в завданні умови, і також вивести значення цієї змінної для користувача (рис. 1.5).

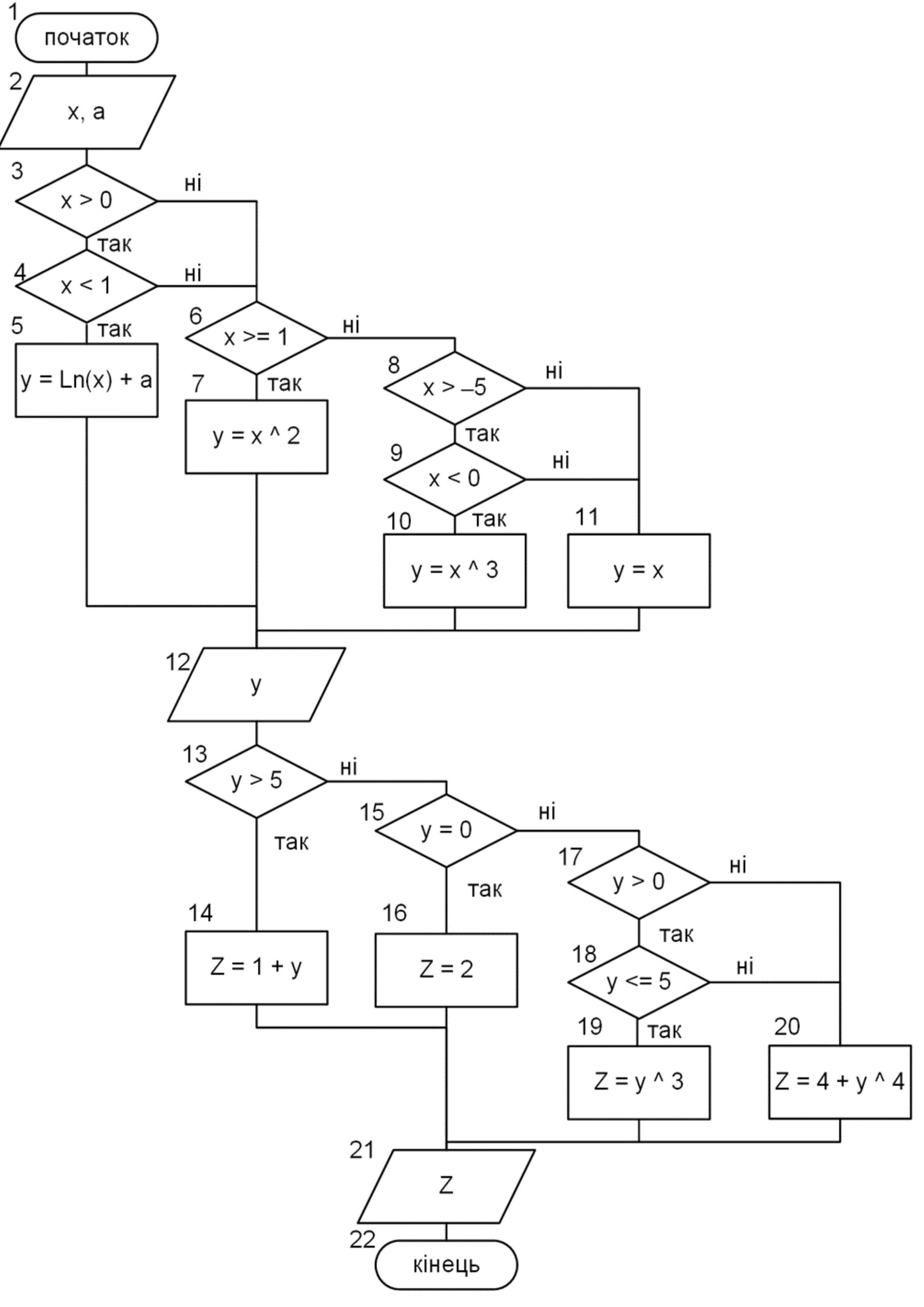


Рис. 1.5. **Розгалужений алгоритм**

Завдання 1.3. Складіть схему алгоритму циклічного обчислюваного процесу для $x \in [1; 5]$, $\Delta x = 0,1$.

$$y = x\left(\frac{\sum_{i=1}^{10} i^2}{k!} + \prod_{j=1}^{k} j^3\right).$$

Виведіть значення змінних *x, y*.

Виконання завдання 1.3.

Аналіз завдання. Для циклічного процесу характерним є повторення набору операцій над різними даними. Наведене завдання потребує виконання чотирьох таких наборів:

1) обчислення суми $\sum_{i=1}^{10} i^2$;

2) обчислення факторіала числа *k!*;

3) обчислення добутку $\prod_{j=1}^{k} j^3$;

4) обчислення значення змінної *y* для кожного зі значень параметра *x*, значення якого змінюють від 1 до 5 з кроком 0,1.

Розгляньмо побудову блок-схем алгоритмів для кожного із зазначених пунктів.

Обчислення суми. За математичною формою подання число, яке вказано знизу знака суми, позначає ліву межу інтервалу зміни значення змінної *i*, а число згори – праву межу цього інтервалу. Так у цьому завданні *i* змінюють від 1 до 10. Крок зміни значення за такого запису завжди дорівнює 1. Позначмо через *S* змінну, у якій буде зберігатися результат обчислень:

$$S = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2.$$

Згідно з формулою, зазначеною під знаком суми, виконують операцію піднесення змінної *i* до другого степеня. Результати виконання операції акумулюють у змінній *S*. Такий процес можна подати виконанням двох кроків, які можна записати в такому вигляді:

$$S = S + i^2,$$
$$i = i + 1.$$

Блок-схема алгоритму розрахунку буде мати вигляд, показаний на рис. 1.6а, водночас параметр *n* дорівнює 10.

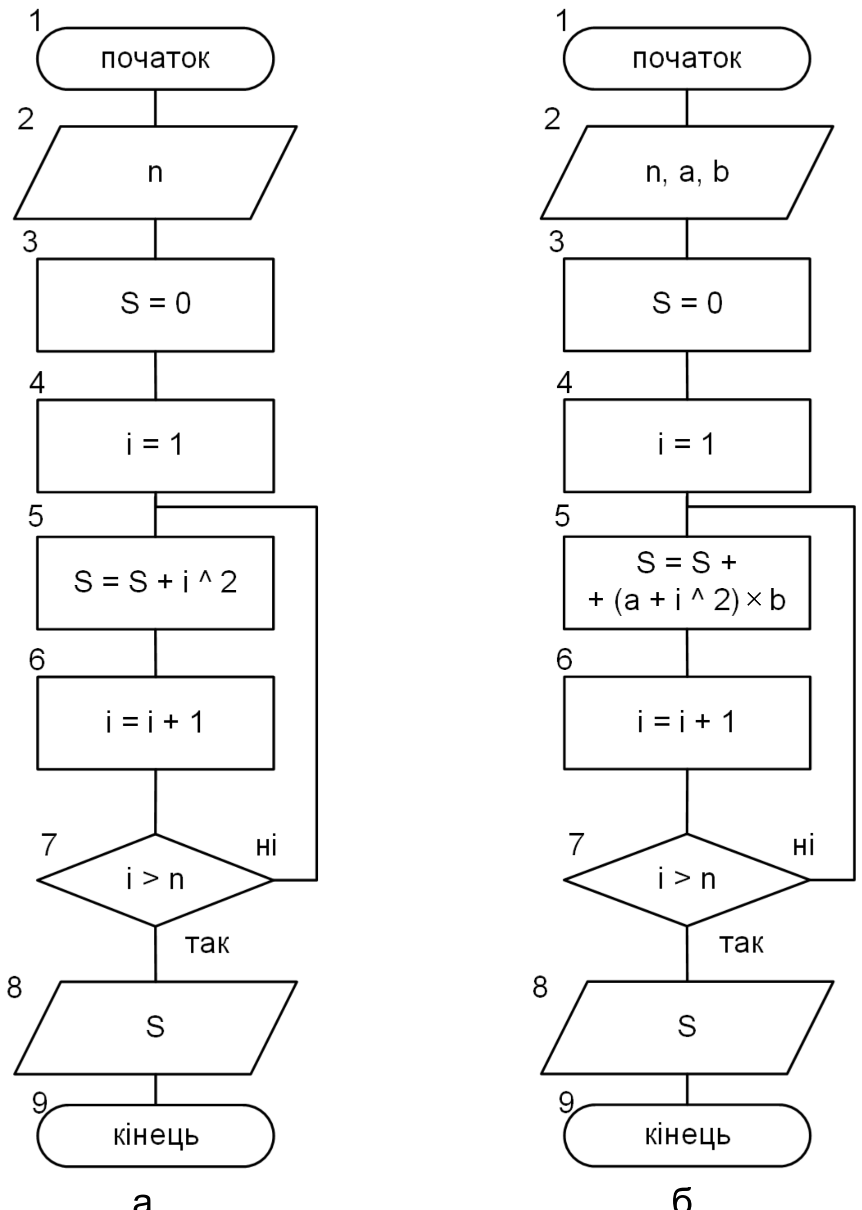


Рис. 1.6. **Блок-схема алгоритму обчислення суми**

Із погляду реалізації на комп'ютері формула $S = S + i^2$ має таку специфіку: за першої ітерації циклу значення змінної *S* буде невідомим. У комірці пам'яті, яку буде відведено для зберігання змінної, може міститися будь-яке значення (наприклад, код символу @). Таке значення спотворить результат розрахунку суми. Тому перш ніж починати цикл, слід надати змінній *S* значення, яке не буде впливати на результат розрахунків (для суми таким значенням є нуль).

Якщо під знаком суми міститься формула для будь-яких інших обчислень, наприклад:

$$\sum_{i=1}^{10} (a + i^2)\, b,$$

структура блок-схеми не змінюється. Змінюють формула в блоці 5 та кількість змінних, значення яких уводяться в блоці 2 (рис. 1.6б).

Наведені в прикладі цикли організовано таким чином:

1) початкове значення лічильника циклу, який позначено змінною i, дорівнює 1 (блок 4), кінцеве значення дорівнює 10;

2) операція зміни значення лічильника міститься в блоці 6, крок зміни лічильника дорівнює 1;

3) умовою завершення циклу є перевищення лічильником свого кінцевого значення ($i > n$), перевірку умови виконують у блоці 7.

Поки умову в блоці 7 не буде виконано, обчислювальний процес буде повертатися до блоку 5, у якому відбувається обчислення тіла циклу (формула в блоці 5) та інкримінація значення лічильника (блок 6).

Обчислення факторіала. Факторіал (*k!*) – це математична функція, що визначають як добуток усіх цілих чисел від 1 до заданого числа *k*. Обчислення факторіала можна здійснити за допомогою ітерації (циклу) або рекурсії:

ітераційний спосіб: реалізувати цикл від 1 до *k* та перемножити всі числа між собою;

рекурсивний спосіб: передбачає визначення базового випадку (яким є факторіал числа 1, що дорівнює 1) та рекурсивний виклик функції зі значенням, що є меншим на одиницю від числа *k*.

Наведімо опис алгоритму рекурсивного способу розрахунку факторіала у вигляді коду:

```
int factorial(int n) {
   if (n == 1) {
      return 1;
   } else {
      return n * factorial(n – 1);
   }
}
```

Розгляньмо ітераційний спосіб розрахунку факторіала. Позначмо через *F* змінну, у якій буде зберігатися значення факторіала, тоді:

$$F = 1 \times 2 \times 3 \times \ldots \times (k - 1) \times k.$$

Якщо розподілити процес розрахунку факторіала на окремі операції, то вони будуть складатися із множення результату, визначеного на попередньому кроці, на нове значення, тобто $F = F \times i$. Значення змінної i змінюється від 1 до k із кроком 1. На рис. 1.7 зображено блок-схему алгоритму ітераційного способу розрахунку факторіала.

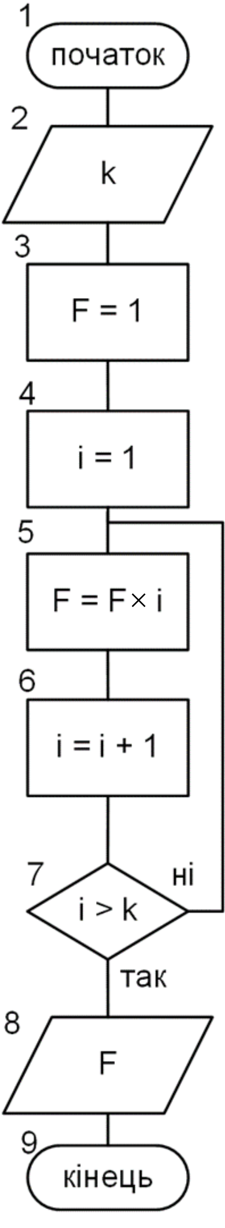


Рис. 1.7. **Блок-схема алгоритму обчислення факторіала *k!***

До початку реалізації циклу, необхідно змінній, яка буде містити результат, надати значення, яке не змінить результат розрахунків. Для операції добутку таке значення має дорівнювати 1 (блок 3).

Наведений у прикладі цикл організовано таким чином:

1) початкове значення лічильника циклу, який позначено змінною i, дорівнює 1 (блок 4), кінцеве значення дорівнює k;

2) операція зміни значення лічильника міститься у блоці 6, крок зміни лічильника дорівнює 1;

3) умовою завершення циклу є перевищення лічильником свого кінцевого значення (i > k), перевірку умови виконують у блоці 7.

Обчислення добутку. Побудови блок-схеми алгоритму обчислення добутку та алгоритму обчислення підрахунку суми є ідентичними. Відмінності полягають у тому, що змінній, у якій буде зберігатися результат (у цьому прикладі змінній P) потрібно надати початкове значення 1, а у тілі циклу застосувати операцію множення (рис. 1.8а).

Якщо під знаком добутку обчислюють більш складну формулу, наприклад:

$$\prod_{j=1}^{k} d\,(j^3 + b),$$

то блок-схема алгоритму буде мати вигляд, зображений на рис. 1.8б.

Вигляд блок-схем алгоритмів циклічних обчислюваних процесів можна спростити шляхом використання блоку циклу (див. п. 6 у табл. 1.1).

Такий блок містить початкове та кінцеве значення лічильника циклу, а також крок зміни лічильника. Якщо крок зміни лічильника дорівнює 1, його не вказують. Блок циклу замінює блоки 4, 6 та 7 та має два вхідні потоки: один – від попереднього блоку, другий – від останнього блоку тіла циклу (якщо умову виходу із циклу ще не виконано) та два вихідні потоки: один – до блоку тіла циклу; другий – до наступних блоків алгоритму (якщо умову виходу із циклу виконано).

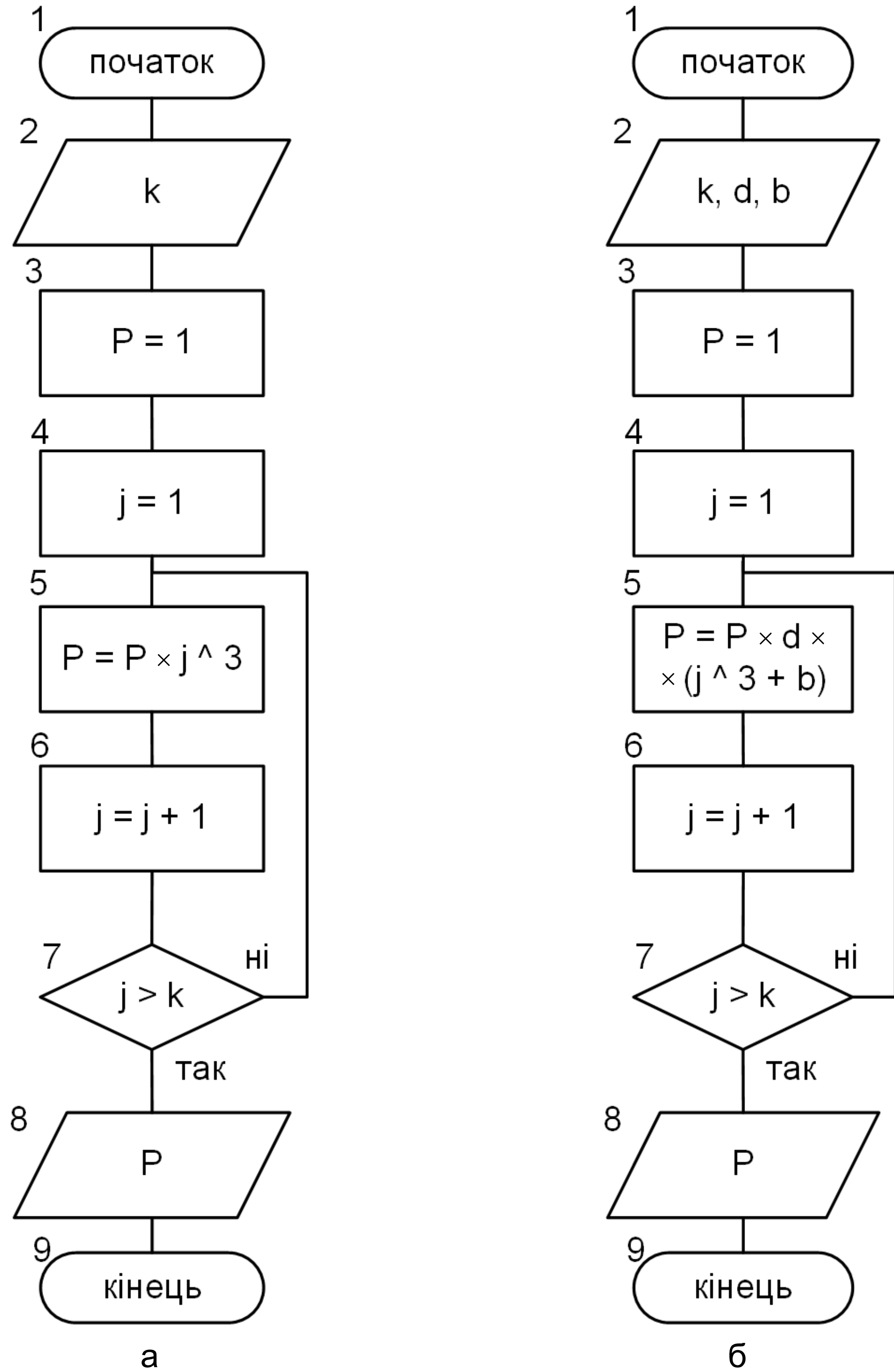
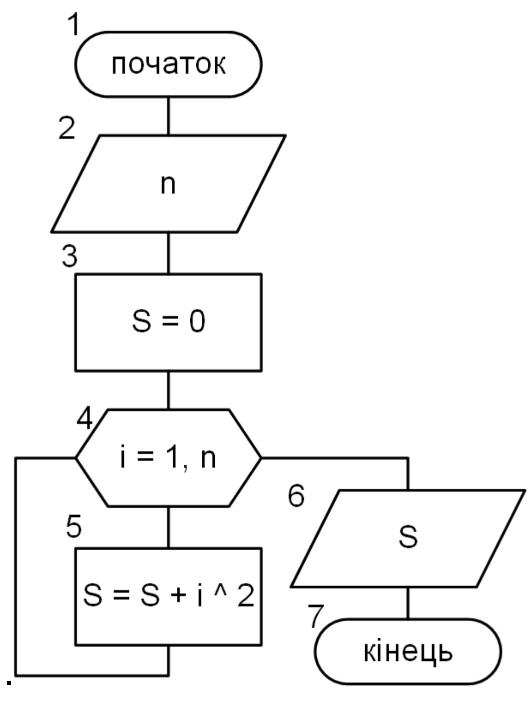
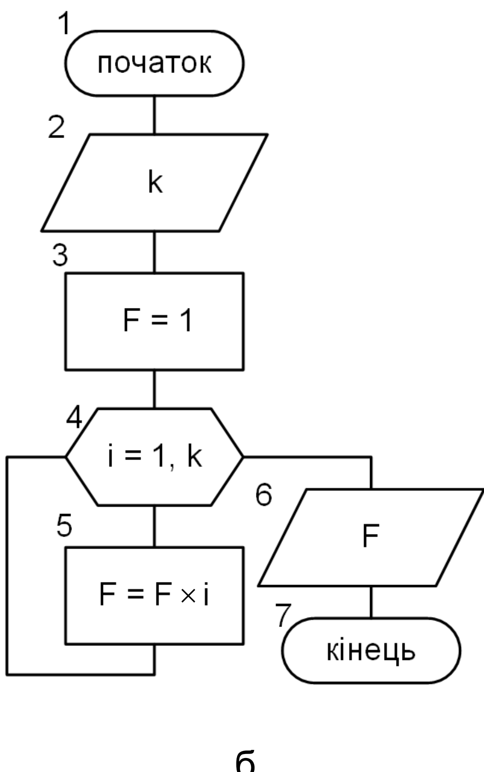


Рис. 1.8. **Блок-схеми для обчислення добутків**

Блок-схеми розрахунків суми, факторіала та добутку за використання блоку циклу зображено на рис. 1.9а, 1.9б та 1.10, відповідно.



а



б

Рис. 1.9. **Блок-схеми обчислення суми (а), факторіала (б) за застосування блоку циклу**

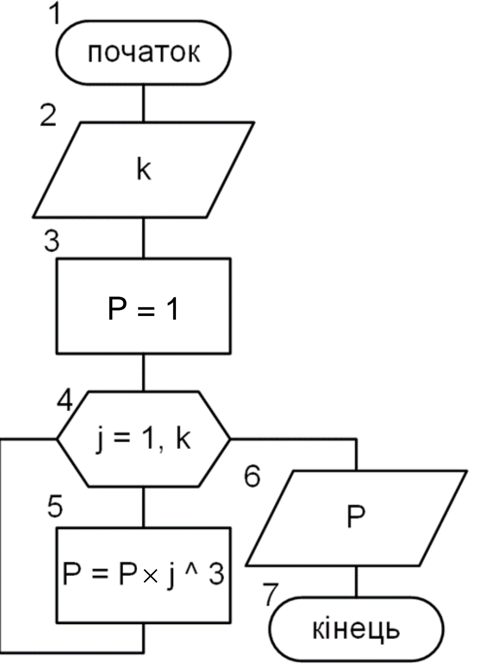


Рис. 1.10. **Блок-схема обчислення добутку за застосування блоку циклу**

Обчислення *y*. Величина *y* залежить від змінної *x*, яка набуває декількох значень (x = 1; 1,1; 1,2; …; 4,8; 5), а тому для обчислення та виведення значення змінної *y* слід організувати цикл за змінною *x*.

Організація такого циклу полягає в такому:

1) початкове значення лічильника циклу, який позначено змінною *x*, дорівнює 1, кінцеве значення дорівнює 5;

2) операція зміни значення лічильника міститься в блоці 16, крок зміни лічильника дорівнює 0,1;

3) умовою завершення циклу є перевищення лічильником свого кінцевого значення (x > 5), перевірку умови виконують у блоці 16.

Тіло циклу будуть становити блоки 17, який містить розрахунок y, та 18, який містить операцію виведення *x* та *y* для користувача.

Значення факторіала, суми та добутку не залежать від *x*, тому вони можуть бути розрахованими ще до організації циклу за змінною *x*.

Остаточний вигляд блок-схеми алгоритму для завдання 1.3 зображено на рис. 1.11.

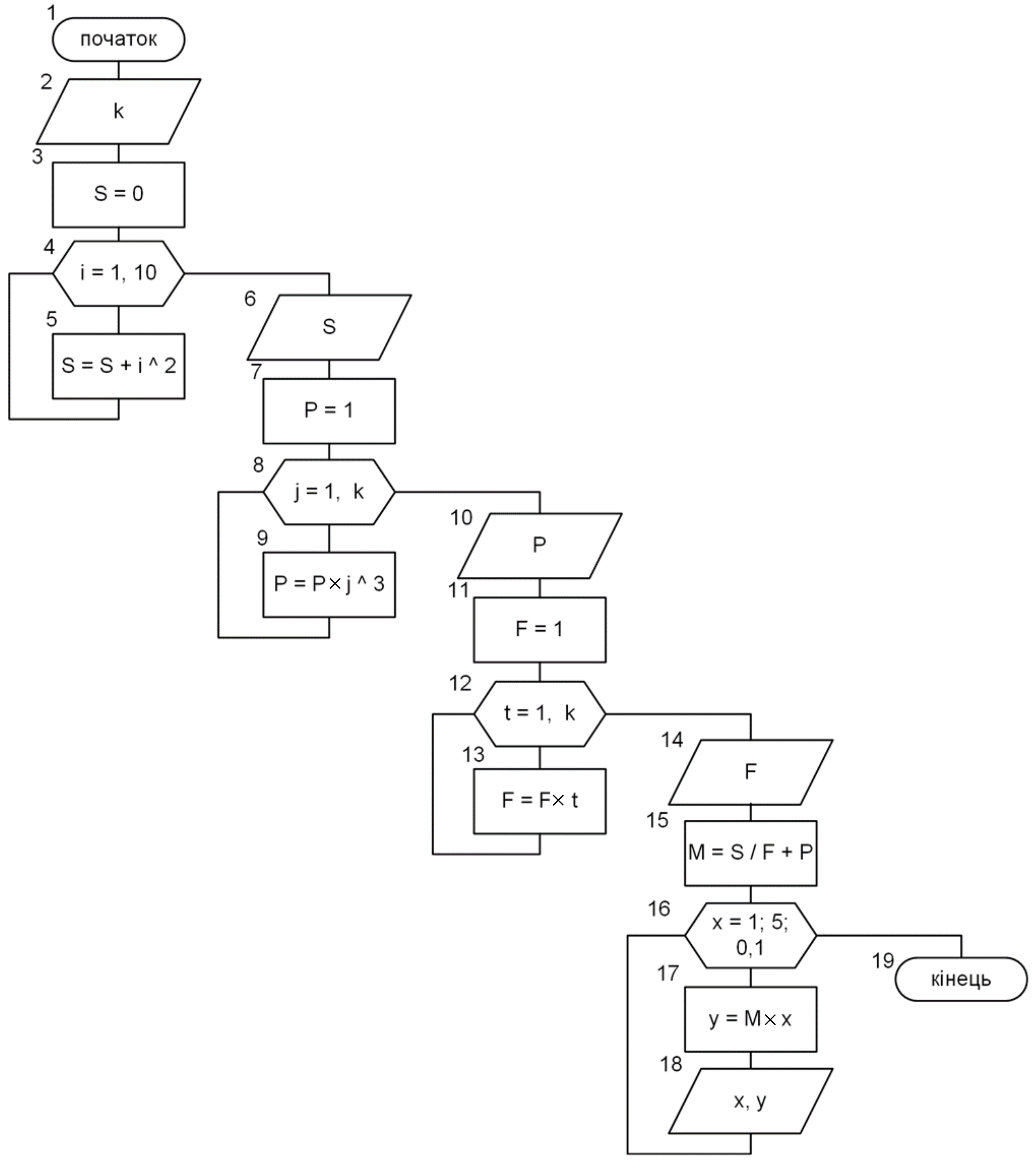


Рис. 1.11. **Алгоритм циклічного обчислювального процесу**

Завдання 1.4. Побудуйте блок-схему алгоритму циклічного обчислюваного процесу із вкладеним циклом та виведіть значення змінної *y*:

$$y = \sum_{x=1}^{K} \left(\frac{1}{x^2} + \frac{\prod_{n=1}^{m} \frac{1}{n^2 x}}{x!} \right).$$

Виконання завдання 1.4.

Аналіз завдання. На початку блок-схеми алгоритму слід увести значення змінних K та m.

На рис. 1.12 зображено блок-схему алгоритму циклічного обчислювального процесу із вкладеними циклами.

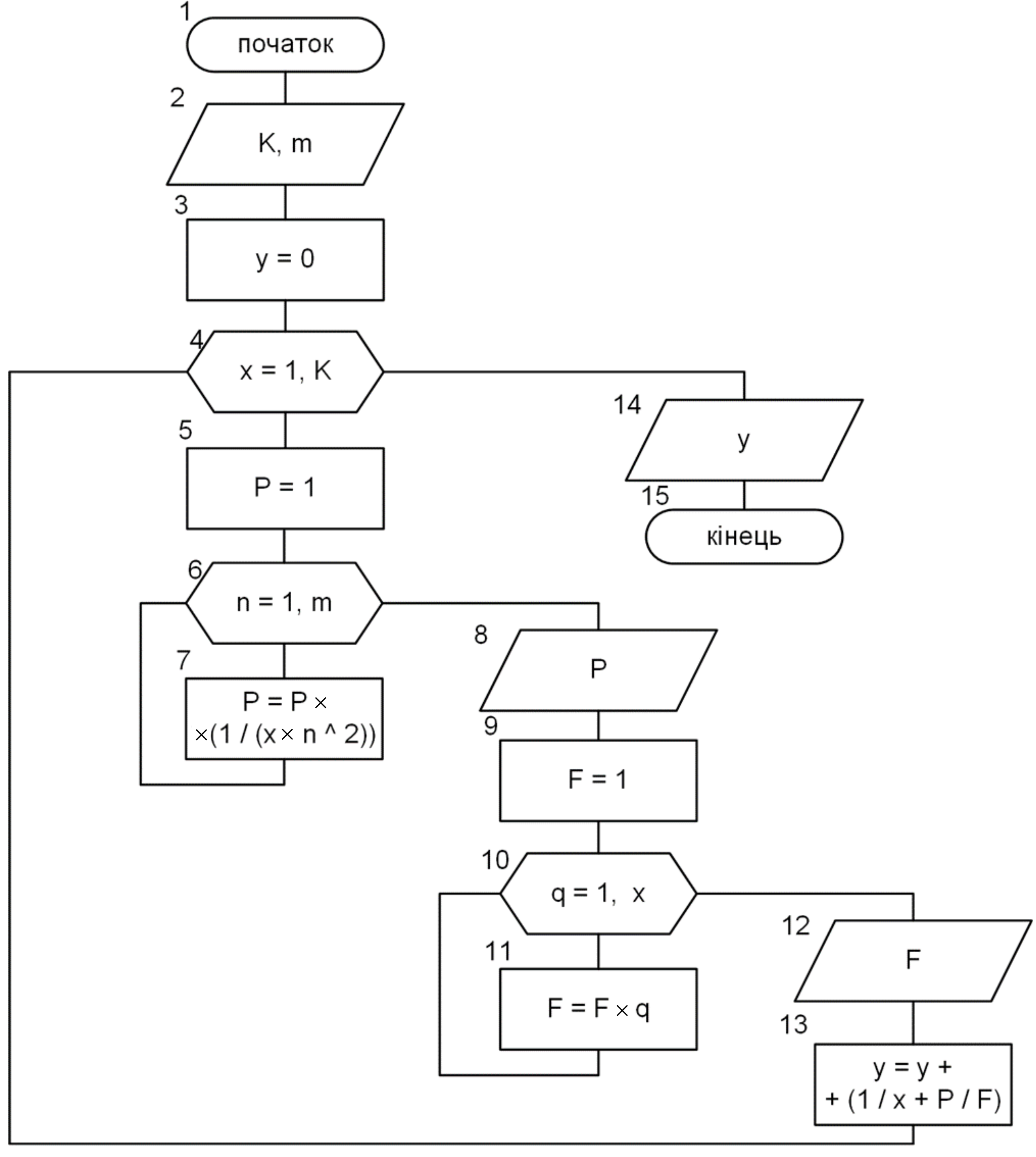


Рис. 1.12. **Блок-схема алгоритму із вкладеними циклами**

Згідно із формулою, зовнішній цикл треба організувати за змінною *x*. Тіло зовнішнього циклу буде містити розрахування таких частин формули:

суми значень для розрахунку *y* (блок 13);

підрахунку факторіала *x* (блоки 9 – 11);

підрахунку добутку, значення якого залежить від змінної *x* (блоки 5 – 7).

Зауваження. Оскільки в змінній *y* буде накопичуватися значення суми, перед початком зовнішнього циклу змінній *y* слід надати значення, яке б не змінювало результату операції додавання (блок 3).

Завдання 1.5. Побудуйте блок-схему алгоритму циклічного обчислюваного процесу із вкладеними циклами та виведіть значення змінної *y*:

$$y = \sum_{x=1}^{m} \left(\frac{1}{\sum_{i=1}^{n} \frac{1}{x^2} + 1} + \prod_{i=1}^{n} \frac{x}{i!} \right).$$

Виконання завдання 1.5.

Аналіз завдання. У цьому завданні до циклу за змінною x укладено цикли розрахунку суми та добутку. Своєю чергою, до циклу розрахунку добутку вкладено цикл підрахунку факторіала значення лічильника циклу. Отже, маємо подвійне вкладення циклів.

На рис. 1.13 зображено блок-схему алгоритму циклічного обчислювального процесу з подвійним укладенням.

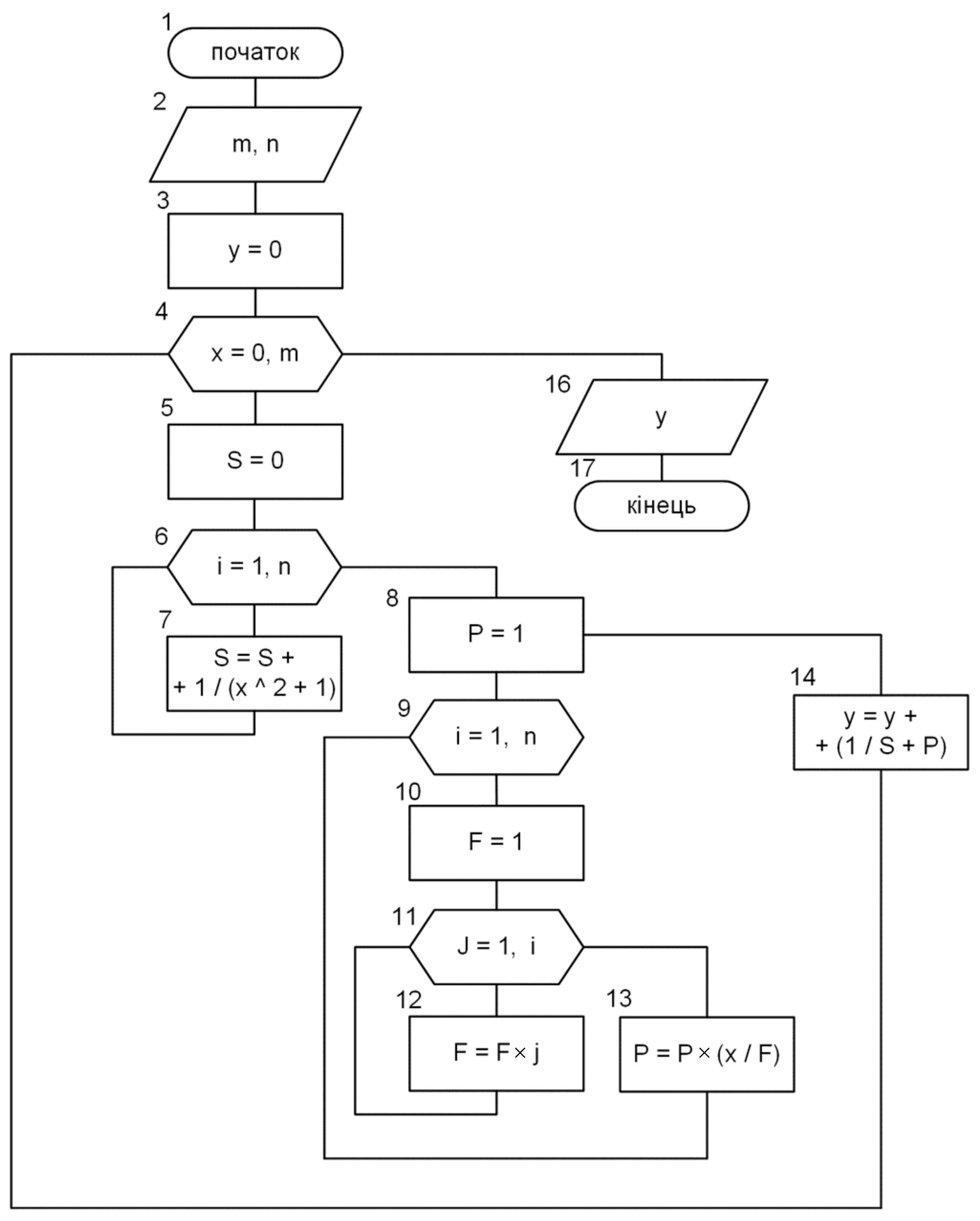


Рис. 1.13. **Блок-схема алгоритму з подвійним укладенням циклів**

Отже, для реалізації алгоритму завдання необхідно:

організувати зовнішній цикл за змінною *x* (блок 4), усередині якого:

організувати цикл для розрахунку суми (блоки 5 – 7);

організувати цикл для розрахунку добутку (блоки 9, 10, 13), усередині якого:

організувати цикл для розрахунку факторіала (блоки 10 – 12).

Завдання 1.6. Складіть схему алгоритму обчислюваного процесу:

$$D=\begin{cases} x^2 + abx, \text{ якщо } 0 \le x \le 2, \\ |x| - a - b, \text{ якщо } 2 < x \le 3 \text{ або } -3 \le x \le -2, \\ abx^2, \text{ в інших випадках.} \end{cases}$$

$$C = D + x,$$

за умови, що змінна *x* набирає значення від −10 до 5 із кроком 0,5.

Виконання завдання 1.6.

Аналіз завдання. Передбачено виконання обчислюваних процесів двох типів: циклічного та розгалуженого.

Шукані значення D та C залежать від x, яка змінюють від −10 до 5 із кроком 0,5, тому спочатку доцільно організувати цикл за x. Лічильником циклу буде сама змінна x із початковим і кінцевим значеннями, відповідно, −10 та 5 та кроком 0,5. Умовою виходу із циклу є перевищення лічильником свого кінцевого значення.

Усередині циклу будуймо блок-схему розгалуженого процесу для визначення формули, за якою слід розраховувати значення змінної D.

Блок-схему алгоритму для наведеного завдання зображено на рис. 1.14.

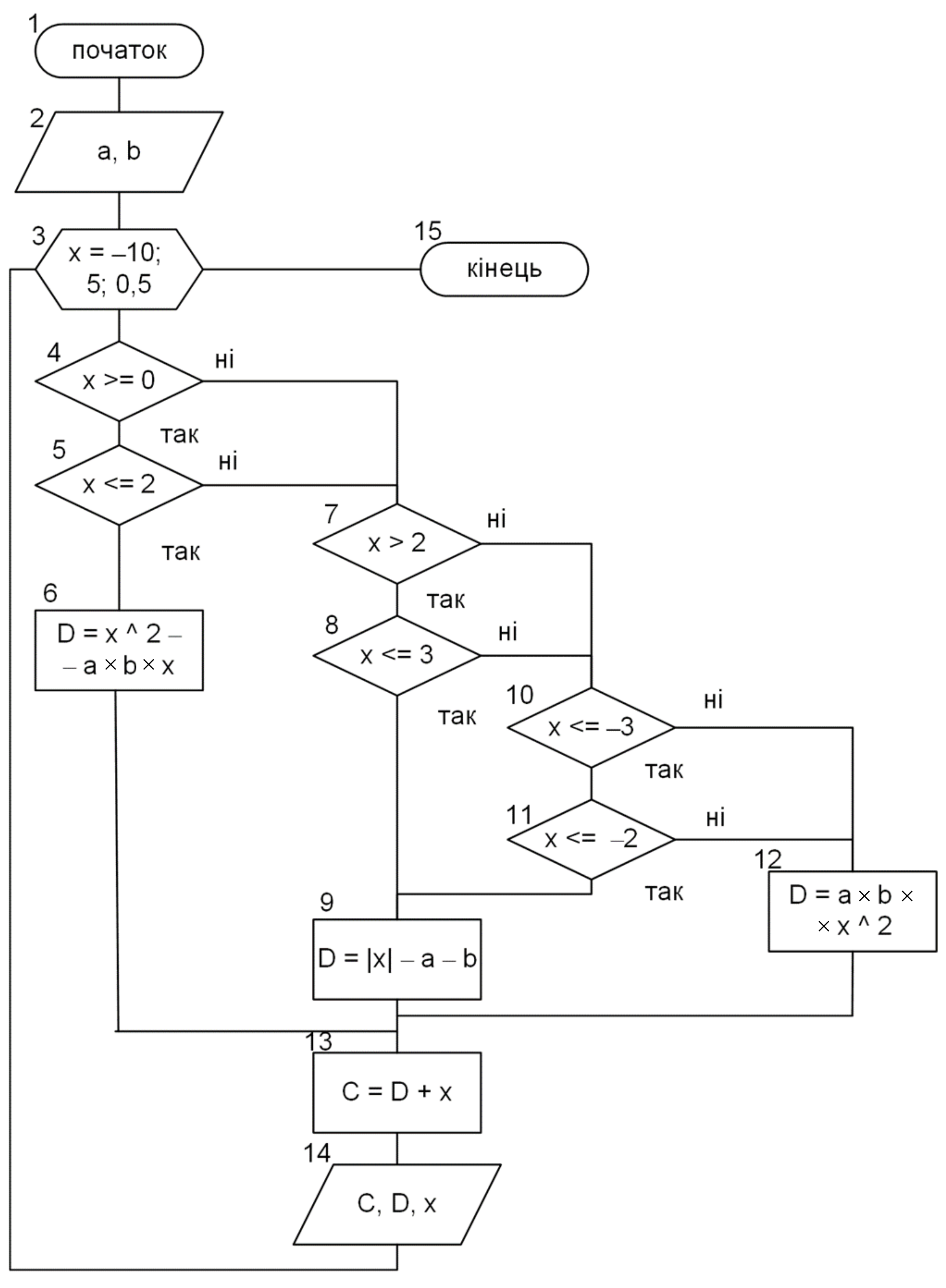


Рис. 1.14. **Блок-схема алгоритму завдання 1.6 (поєднання циклічного та розгалуженого обчислюваного процесу)**

Контрольні запитання до лабораторної роботи 1

1. Який обчислюваний процес називають лінійним?
2. Які блоки використовують у блок-схемах лінійного обчислюваного процесу?
3. На які запитання слід дати відповіді перш ніж почати складати блок-схему алгоритму лінійного обчислюваного процесу?
4. Наведіть приклад лінійного процесу.
5. Охарактеризуйте типи алгоритмів.
6. Який обчислюваний процес називають розгалуженим?
7. Від чого залежить вибір напрямку у розгалуженому обчислювальному процесі?
8. Що таке «логічна умова»?
9. Який розгалужений процес називають простим?
10. Який розгалужений процес називають складним?
11. Які операції в логічних виразах вам відомі? Що є результатом кожної з них?
12. Які типи логічних відношень вам відомі?
13. Дайте визначення відношення еквівалентності.
14. Дайте визначення відношення порядку.
15. Які блоки використовують у блок-схемах розгалуженого обчислюваного процесу?
16. На які запитання слід дати відповіді перш ніж складати блок-схему алгоритму розгалуженого обчислюваного процесу?
17. У чому полягає особливість блоку перевірки логічної умови?
18. У чому полягає різниця між блок-схемою для алгоритмізації логічного додавання та логічного множення?
19. Наведіть результати логічного додавання в табличній формі.
20. Наведіть результати логічного множення в табличній формі.
21. Які особливості побудови блок-схеми алгоритму розгалуженого обчислювального процесу?
22. Наведіть приклад розгалуженого процесу.
23. Який обчислюваний процес називають циклічним?
24. Дайте визначення циклу.
25. Що таке «лічильник циклу»?
26. Що таке «параметр циклу»?

27. Що треба визначити для організації циклу?
28. Що може бути умовою виходу із циклу?
29. Який цикл називають ітераційним?
30. Який цикл називають арифметичним?
31. Що таке «вкладений цикл»?
32. Який цикл називають внутрішнім циклом?
33. Який цикл називають зовнішнім циклом?
34. Наведіть блок-схему для підрахунку суми.
35. Наведіть блок-схему для підрахунку факторіала.
36. Наведіть блок-схему для підрахунку добутку.
37. Скільки вхідних та вихідних потоків має блок циклу? Назвіть їх.
38. Яку інформацію записують у блоці циклу?
39. Які блоки замінює блок циклу?
40. Наведіть приклад розгалуженого процесу.
41. Наведіть приклад лінійного процесу.
42. Наведіть приклад циклічного процесу.
43. У яких випадках доцільно використовувати вкладені цикли й чому?

2. Елементарні структури даних

2.1. Структурування й абстракція під час написання програм

Значні за обсягом і складні програми проєктують шляхом декомпозиції задач – виділення в них характерних структур та відповідних абстракцій. Водночас під абстрагуванням розуміють певного виду перетворення, яке дозволяє, з одного боку, виділити істотні властивості, а з іншого – відокремити та позбутися неістотних і другорядних. Це передбачає ігнорування низки незначних деталей задач або проблем, із метою їхнього спрощення. Задачі абстрагування і декомпозиції є типовими для процесу створення програм:

декомпозицію використовують для розподілу програм на компоненти;

абстрагування передбачає ґрунтовний вибір компонент задачі.

Структурний підхід до даних і алгоритмів надає виконавцям можливість декомпонувати складну програму. У сучасних умовах розроблення програми методом «усе й відразу» є недоцільним. Спочатку програму має

бути подано у вигляді певної структури – складових частин і зв’язків між ними. Створення правильної структури програми на кожному етапі розроблення дає можливість зосередитися на певній компоненті або модулі та в разі потреби долучити до реалізації різних компонент відповідних виконавців.

Під час побудови структури програм застосовують два типи підходів, які ґрунтуються на структуризації алгоритмів:

- «низхідне» проєктування або «програмування згори вниз»;
- «висхідне» проєктування або «програмування знизу вгору».

За *низхідного проєктування* відбувається структуризація алгоритмів, яка передбачає насамперед структурування дій, які має виконувати програма [2]. Водночас велику та складну задачу розподіляють на кілька задач меншого обсягу. Отже, модуль найвищого рівня, який відповідає за розв’язання всієї задачі, виявляється достатньо простим. Модуль вищого рівня забезпечує лише послідовність викликів модулів нижчих рівнів. За тим самим принципом декомпозиції підлягає кожна підзадача. Процес розподілу на підзадачі продовжують доти, поки на черговому етапі декомпозиції реалізація задачі не виявляється достатньо простою. Отже, будь-яка програма водночас є набором функціональних абстракцій (рис. 2.1). Аналіз такого подання програми надає узагальнену абстракцію функції.

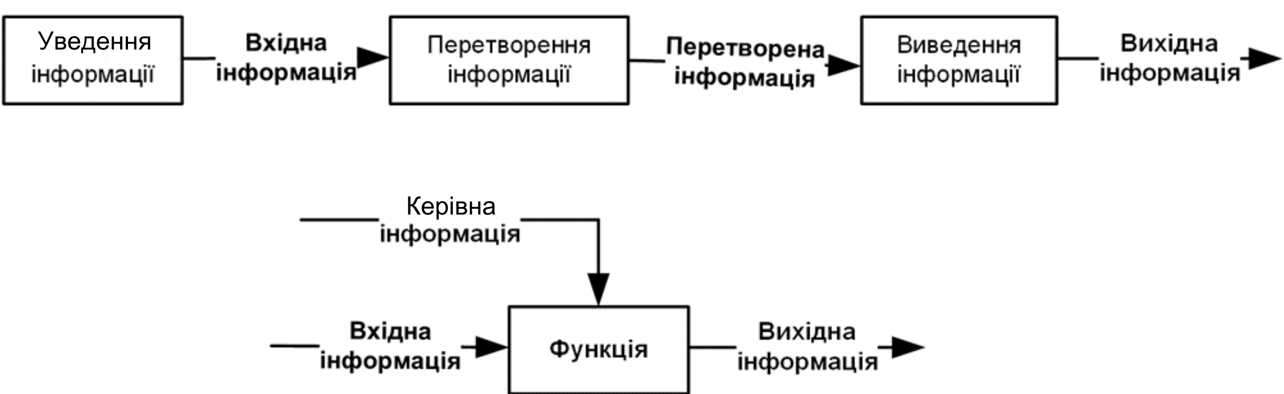


Рис. 2.1. **Подання структуризації алгоритмів**

Висхідне проєктування ґрунтується насамперед на структуризації даних. Такий підхід обумовлено тим, що будь-яку програму призначено для опрацювання даних, тобто здобуття нових даних на підставі наявних.

Під час розроблення програм можна використовувати різні алгоритми, проте зазвичай у програми є замовник, який має певні вхідні дані, та прагне визначати на їхній основі коректні вихідні дані, а яким чином забезпечують здобуття цих даних, не має значення.

Отже, завданням програми є перетворення вхідних даних на вихідні. Тоді мови програмування можна розглядати як набори базових типів даних та операцій над ними.

Узагальнюючи й компонуючи базові типи, розробник програмного забезпечення створює нові складні типи даних і визначає операції над ними. В ідеалі останній крок композиції дає типи даних, які відповідають вхідним і вихідним даним завдання, а операції над ними в повному обсязі реалізують завдання проєкту.

Розглядаючи програму як набір даних, кожне із яких має відповідний набір дозволених функцій, здобувають таке абстрактне подання програми (рис. 2.2).

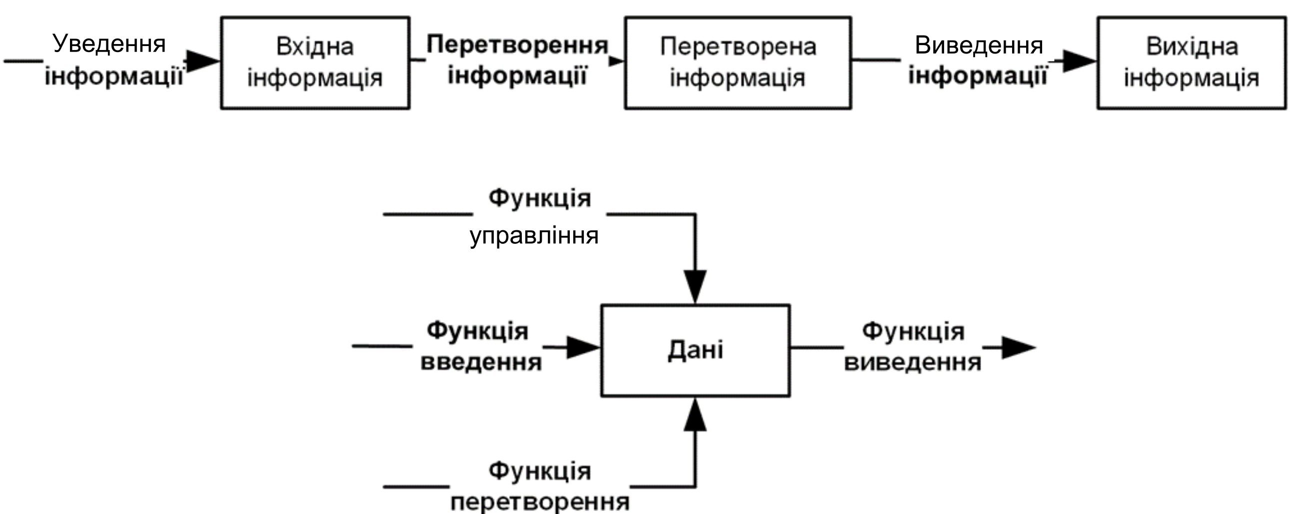


Рис. 2.2. **Подання структуризації даних**

Аналіз такого подання програми дозволяє також визначити абстракцію даних. Визначені абстракції алгоритмів і даних лежать в основі багатьох формалізованих методів розроблення програм.

Не слід протиставляти низхідне проєктування висхідному і весь час дотримуватися одного підходу. Реалізація будь-якого реального проєкту завжди відбувається за використання обох підходів: структури алгоритмів

постійно коригують, згідно з результатами розроблення структур даних, і навпаки.

Іншим продуктивним технологічним методом, пов'язаним зі структуризацією даних, є інкапсуляція [8]. Сутність інкапсуляції полягає в тому, що дані та методи, що належать до класу, об'єднують в один об'єкт або модуль, який можна розглядати як окрему структуру даних. Внутрішня будова такої структури даних є недосяжною для програміста-користувача.

Використовуючи такий тип даних, програміст може управляти даними цього типу лише через методи, які є визначеними для цього типу. Отже, цей тип даних має вигляд «чорної скриньки», входи та виходи якої є відомими, а вміст невідомим і недоступним.

Для сучасних мов програмування притаманним є блоковий тип, завдяки чому вони надають розвинуті засоби розроблення програм із модульною структурою й управлінням доступом модулів до функцій та даних. Об'єктно-орієнтованість сучасних мов програмування зумовлено розширенням можливостей конструювання типів і їхньої інкапсуляції. Розроблені та повністю закриті структури даних утворюють класи, а функції, які працюють із їхнім внутрішнім поданням, – методи роботи із класами. За такого підходу значних змін зазнає й сама концепція програмування: програміст під час роботи із об'єктами зазначає в програмі «що» слід зробити з об'єктом, а не «яким чином».

2.2. Концепція структур даних

Алгоритми та структури даних є засобами для створення програм. Навіть сам комп'ютер складається зі структур даних та алгоритмів. Бінарні величини зберігають у вбудованих структурах даних, які подають у вигляді регістрів та слів пам'яті. Алгоритми, закладені до конструкції апаратури, є реалізацією жорстких правил, за якими дані інтерпретують як команди, які мають бути виконаними. Тому в основі функціонування комп'ютера лежить спроможність оперувати тільки одним видом даних – з окремими бітами. Із даними комп'ютер працює, згідно зі сталим набором алгоритмів. Такі алгоритми визначено системою команд центрального процесора. Проте задачі, які розв'язують за допомогою комп'ютера,

зрідка подають мовою бітів. Зазвичай, дані мають форму чисел, літер, текстів, символів, або складніших структур: масивів, черг, списків, дерев.

Структура даних може бути поданою схемою організації інформації в пам'яті комп'ютера, тому може бути зарахованою до «просторових» понять. Алгоритм описує процес опрацювання даних, тому є процедурним елементом структури програми.

В алгоритмах можуть використовувати досить складні структури даних, тому вибір структури даних є суттєвим етапом ефективного програмування і може впливати на якість програми сильніше за деталі реалізації алгоритму. Вибір подання даних є творчим слабкоформалізованим завданням і значною мірою залежить від вимог, які ставлять до програми в кожному конкретному випадку.

Незалежно від вмісту та складності, дані в пам'яті комп'ютера подано як послідовність бінарних розрядів, значеннями яких є бінарні числа. Дані, які подають послідовністю бітів, мають дуже просту організацію, тобто є слабкоструктурованими. Організація даних, складніших за біт, передбачає застосування поняття «структура даних».

Розрізняють фізичну структуру даних, або структуру зберігання, та абстрактну, або логічну, структуру даних [4]. Фізична структура відображає спосіб фізичного подання (зберігання) даних у пам'яті комп'ютера. Абстрактна структура даних надає подання даних без урахування їхнього подання в пам'яті комп'ютера. Логічний опис даних дозволяє забезпечити однозначне розуміння структури даних між розробниками, тестувальниками та замовником. На скільки сильно відрізняються логічна і фізична структури певних даних залежить від самих даних та особливостей середовища, у якому реалізують роботу із цими даними.

Отже, кожна структура даних може бути описаною на трьох рівнях, як-от:

функціональна специфікація – визначає операції, дозволені для роботи з певним класом імен, а також властивості цих операцій;

логічний опис – визначає типи даних, зв'язки між ними й обмеження, встановлені на ці зв'язки, забезпечує декомпозицію об'єктів та операцій;

фізичне подання – визначає спосіб розміщення в пам'яті комп'ютера величин, які становлять структуру певного типу, а також спосіб кодування операцій визначеною мовою програмування.

Одній і тій самій функціональній специфікації можна надати кілька логічних описів, а кожен логічний опис можна реалізовувати кількома фізичними поданнями. Під час декомпозиції на кожному такому рівні структуру даних мають погоджувати зі структурою вищого рівня.

2.3. Класифікація структур даних

У загальному випадку структуру даних визначають як множину елементів даних і зв'язків між цими елементами. Виокремлюють прості, або базові, та інтегровані, або структуровані, структури даних (рис. 2.3).

За можливістю розподілу

Прості (базові)

- не можуть бути розділеними на складові частини, більші, ніж біти;
- завжди можна наперед сказати, яким буде розмір цього простого типу і яка структура його розміщення в пам'яті;
- є неподільними одиницями з точки зору логіки

Інтегровані (структуровані)

- складовими частинами яких є інші структури даних;
- конструюють програмістом із використанням засобів інтеграції даних, які подають мовами програмування

Рис. 2.3. **Класифікація структур даних за можливістю розподілу**

Простими є структури даних, які не можна розподілити на складові частини, більші за біти. Важливою особливістю простих структур даних є те, що за визначених архітектури комп'ютера й системи програмування завжди можна знати розмір цього простого типу та структуру його розміщення в пам'яті ще на етапі компілювання. За логічного подання прості структури даних є неподільними одиницями.

Інтегрованими є структури даних, які складаються з інших структур даних. Інтегровані структури даних конструюють за допомогою інтеграції даних у спосіб, обумовлений певною мовою програмування.

У порівнянні із простими структурами даних інтегровані структури мають кілька переваг:

1) використання інтегрованих структур даних оптимізує використання пам'яті та підвищує швидкодію виконання програми;

2) інтегровані структури даних забезпечують можливість збереження різних типів даних, що забезпечує гнучкість та масштабованість даних, залежно від потреб програми;

3) інтегровані структури даних забезпечують можливість взаємодії між різними елементами даних: наприклад, дерева та графи можуть бути використаними для подання взаємозв'язків між об'єктами, що дозволяє програмі взаємодіяти зі складними структурами даних і здійснювати аналіз.

На рис. 2.4 наведено класифікації структур даних за різними ознаками. Залежно від наявності явно заданих зв'язків між елементами даних виокремлюють зв'язні та незв'язні структури.

Рис. 2.4. **Класифікація структур даних**

Незв'язні структури даних, як-от масиви, зберігають дані послідовно, а їхні елементи розташовують у пам'яті поряд один із одним. Незв'язні структури даних можуть бути розміщеними в одному місці пам'яті, але вони не взаємодіють між собою. Кожен елемент незв'язної структури даних містить лише саму інформацію.

Зв'язні структури даних, як-от зв'язні списки та дерева, зберігають дані як окремі вузли, кожен із яких містить посилання на інші вузли. Зв'язні структури даних зазвичай забезпечують доступ до даних у будь-якому порядку, а не лише в порядку їхнього збереження. Зв'язні структури даних можуть бути використаними для організації вкладених і складних структур даних, як-от стеки, черги та дерева, а також для зберігання даних у вигляді списків залежностей і зв'язків між різними елементами даних.

Із погляду використання в програмі важливою ознакою структури даних є її динамічність – це властивість змінювати кількість елементів та/або зв'язки між елементами структури під час виконання програми. За цією ознакою розрізняють статичні, напівстатичні та динамічні структури.

Іншою ознакою структури даних, важливою з погляду реалізації алгоритмів, є характер упорядкованості її елементів. Згідно із цією ознакою, виокремлюють лінійні й нелінійні структури даних (див. рис. 2.4).

2.4. Операції над структурами даних

Над будь-якими структурами даних дозволено виконання чотирьох основних операції: створення, знищення, вибір (доступ), оновлення [13].

Операція створення полягає у виділенні пам'яті для зберігання структури даних. Пам'ять може виділятися як під час виконання програми, так і на етапі компіляції. Якщо структуровані дані конструює програміст іноді операція створення потребує ініціалізації, тобто задавання початкових значень параметрів структури.

Незалежно від мови програмування, структури даних мають бути явно або неявно оголошеними операторами створення.

Операція знищення полягає у звільнені пам'яті, яку займала певна структура, із метою подальшого використання цієї пам'яті. Операція знищення забезпечує ефективне використання пам'яті комп'ютера.

Операція вибору полягає в здійсненні доступу до даних у певній структурі. Реалізацію операції доступу значною мірою обумовлено типом структури даних. Метод доступу є однією з найважливіших властивостей структур, яка сильно впливає на вибір розробником конкретної структури даних.

Операція оновлення полягає в зміні значення даних у структурі. Прикладами операції оновлення є операції надання та передавання параметрів.

Крім наведених основних операцій, для кожної структури даних можна визначити специфічні операції, які працюють лише з даними конкретного типу або структури.

2.5. Прості структури даних

2.5.1. Арифметичний тип даних

Стандартні типи даних ще називають *арифметичними* через те, що їх можна використовувати в арифметичних операціях. Опис стандартних типів визначають такими ключовими словами: цілий, символьний, логічний, дійсний.

Цілі числа використовують для підрахунку кількості об’єктів. Ця величина є дискретною: якщо об’єкти утворюють кінцеву множину, то їх можна перелічити кінцевою множиною значень. Внутрішнє подання змінних цілого типу є цілим числом у бінарному коді. Внутрішнє подання змінних дійсного типу утворено із двох частин – мантиси та порядку.

Результатом порівняння даних певних типів є величини *логічного типу,* які можуть набувати значення лише true або false. Внутрішня інтерпретація значення false – 0 (нуль), інакше значення інтерпретують як true.

Величинами *символьного типу* є знаки заздалегідь визначеної множини. У більшості персональних комп’ютерів такою множиною є ASCII, яка містить 256 різних знаків, включно із цифрами, розділовими знаками, символами великих і малих літер, спеціальними керівними символами тощо, упорядкованими за певною ознакою. Величина символьного типу займає в пам’яті один байт. Іншим поширеним стандартом для подання символьних даних є код Unicode, за використання якого на кожний символ виділено два байти.

Як і зо всіма іншими типами даних, з арифметичними типами здійснюють чотири основні операції: створення, знищення, вибір, оновлення, а також специфічні операції, досяжні для числових типів даних, – додавання, віднімання, множення та ділення.

Іншою групою операцій над арифметичними типами є операції порівняння: «дорівнює», «не дорівнює», «більше», «менше» тощо. Щодо операцій порівняння, то слід зауважити специфіку цих операцій із дійсними числами (порівняння на рівність/нерівність). Оскільки ці числа можуть

мати різну точність, як-от 8-бітні, 16-бітні, 32-бітні або 64-бітні числа, їх порівняння не завжди може бути повністю достовірним.

2.5.2. Перелічуваний тип даних

Під час написання програм часто виникає потреба визначити іменовані константи, кожна з яких має унікальне значення. Для цього використовують перелічуваний тип, що є впорядкованим набором даних, який визначає програміст шляхом формування переліку всіх значень, які може набирати змінна цього типу.

Діапазон значень перелічуваного типу визначають кількістю бітів, потрібних для подання всіх значень множини. Будь-яке значення цілого типу можна явно привести до перелічуваного типу, але в разі виходу за межі діапазону результат буде невизначеним. Якщо ініціалізацію не було здійснено під час опису, перша константа набирає нульове значення, а значення кожної наступної константи є на одиницю більшим від значення попередньої.

Над змінними перелічуваного типу визначено операції створення, знищення, вибору, оновлення. Під час реалізації цих операцій здійснюють визначення порядкового номера константи за її значенням і, навпаки, за номером визначають значення.

Під час виконання арифметичних операцій перелік перетворюється на ціле. Перелічуваний тип визначає користувач, тому для роботи з ним можна визначати власні операції [15].

2.5.3. Покажчики

У змінних із типом даних «покажчик» зберігають адреси в пам’яті, які можна використовувати для доступу до даних та об’єктів.

У більшості сучасних інформаційних систем розмір комірки, тобто мінімальної одиниці пам’яті, до якою можна адресувати, дорівнює одному байтові. Під час програмування на низькому рівні робота з адресами комірок пам’яті становить значну частину програм. Під час розв’язання прикладних задач із використанням мов програмування високого рівня використання покажчиків відбувається в таких випадках:

у разі потреби подати одну й ту саму ділянку пам’яті (тобто одні й ті самі фізичні дані) як дані різної логічної структури;

під час роботи з динамічними структурами даних.

Можна розрізняти різні види покажчиків, наприклад, на об'єкт, функцію, порожній тип. Покажчики різних типів відрізняються властивостями та набором дозволених операцій. Отже, покажчик не можна вважати самостійним типом, він завжди є пов'язаним з іншим типом даних.

Під час використання мов програмування високого рівня є можливість роботи як із типізованими покажчиками, так і з нетипізованими. Під час оголошення типізованого покажчика слід визначити тип об'єкта в пам'яті, на адресу якого цей покажчик посилається. Це означає, що типізовані покажчики можуть бути використаними для зберігання адрес тільки певних типів даних. Наприклад, якщо типізований покажчик має тип «ціле число», то він може бути використаним лише для зберігання адрес цілих чисел у пам'яті.

Типізовані покажчики забезпечують безпеку та захист від помилок у програмі. Вони дозволяють компілятору перевірити, що покажчик використовують правильно та він зберігає тільки відповідний тип даних. Це допомагає уникнути небезпечних помилок, як-от використання неправильної адреси чи запис до пам'яті, яка не є призначеною для цього.

Нетипізований покажчик не має визначеного типу даних та може зберігати адреси будь-якого типу даних. Це означає, що нетипізовані покажчики можуть бути використаними для зберігання адрес будь-якого типу даних та об'єктів у пам'яті.

Основними операціями, у яких задіяні покажчики, є надання, визначення адреси об'єкта та вибір. Набір операцій над покажчиками в більшості мов програмування обмежено наведеним переліком, оскільки їх достатньо для розв'язання задач прикладного програмування. Проте системне програмування потребує гнучкішої роботи з адресами, а тому в таких мовах, як C/C++, можна здійснювати також операції адресної арифметики.

2.6. Статичні структури даних

Статичні структури даних належать до класу базових структур, які є множинами елементарних даних. Оскільки статичні структури характеризуються сталістю, пам'ять для їхнього зберігання виділяється автоматично. Зазвичай, це відбувається на етапі компіляції або під час виконання, за активізації програмного блоку, у якому визначені відповідні дані. У деяких мовах програмування можливим є розміщення статичних структур у пам'яті комп'ютера під час виконання програми за явною

вимогою, та навіть у цьому разі розмір пам’яті, виділеної для зберігання даних, залишається незмінним до моменту знищення структури.

Виділення пам’яті під час компіляції є зручною властивістю статичних структур, тому іноді ці структури використовують навіть для опису об’єктів, яким притаманна змінюваність. Наприклад, якщо розмір масиву є наперед невідомим, то під нього виділяється максимально можливий розмір.

Статичні структури в мовах програмування зв’язано зі структурованими типами, які дозволяють створювати структури даних довільної складності. До таких типів належать масиви, структури та похідні з них.

2.6.1. Масиви

Масив – це структура даних, що складається з послідовно розташованих елементів певного типу даних. Для визначення масиву потрібно вказати тип даних, що зберігають у кожному елементі масиву, а також кількість цих елементів. Адресу кожного елемента визначають одним (одновимірний масив) або кількома (багатовимірний масив) індексами [6].

За логічним і фізичним поданням масив є єдиною сукупністю однорідних даних.

За розмірністю масиви розподіляють на одновимірні (вектори), двовимірні (матриці) та багатовимірні (трьох, чотирьох та більше).

До основних характеристик масиву як структури даних належать:

- фіксованість набору елементів певного типу;
- унікальність набору значень індексів для кожного елемента;
- визначення розмірності масиву за кількістю індексів його елементів;
- можливість доступу до елемента за ім’ям масиву та значеннями індексів елемента.

Розмір ділянки пам’яті, яку резервують під кожен елемент масиву, визначено його базовим типом. Розмір ділянки для зберігання всього масиву визначають добутком розміру ділянки для зберігання одного елемента та кількості елементів масиву.

Доступ до заданого елемента масива є важливою операцією, оскільки без неї не може бути реалізовано жодної операції з елементом. Операції, які є дозволеними для елементів певного масиву, визначено типом даних, до якого належать елементи.

Адресою всього масиву є адреса першого байта першого елемента масиву. Індексом першого елемента масива може бути нуль або одиниця. Найчастіше над масивами виконують такі операцій логічного рівня, як сортування, пошук елемента за ключем або статистичні операції, як-от пошук елемента з максимальним (мінімальним) значенням, середньоарифметичне або сума всіх елементів та ін.

2.6.2. Розріджені масиви

Розріджений масив – це масив, більшість елементів якого мають однакові (нульові) значення. Елементи, значення яких є фоновими, називають *нульовими*; елементи, значення яких є відмінними від фонового, – *ненульовими*. Проте фонове значення не завжди дорівнює нулю.

Використання розріджених масивів є актуальним для багатовимірних масивів великого розміру, зберігання яких у повному обсязі потребує значних витрат пам'яті. У цьому разі є доцільним зберігання в пам'яті лише невеликої кількості значень, які є відмінними від нульового (фонового) значення.

Розріджені масиви можуть бути різної розмірності, але для їхнього опрацювання використовують процедури, які є відмінними від процедур роботи із простими масивами.

Розріджені масиви розрізняють за способом розміщенням елементів:

1) масиви, у яких ненульові елементи розміщають за певним математичним законом,

2) масиви з випадковим розміщенням ненульових елементів.

У першому випадку ненульові елементи зберігають зазвичай в одновимірному масиві, а зв'язок між розташуванням у початковому (розрідженому) масиві та новому (одновимірному) може бути описаним математичною формулою, яка перетворює індекси розрідженого масиву на індекси вектора.

Під час роботи з розрідженими масивами розробляють такі функції:

перетворення індексів масиву в індекси вектора;

обчислення значення елемента масиву за його упакованого подання за індексами;

запису значення елемента масиву до його запакованого подання за індексами.

У другому випадку основним способом зберігання розріджених масивів є запам'ятовування ненульових елементів в одновимірному масиві з ідентифікацією кожного елемента масиву за допомогою індексів – метод зберігання ненульових елементів та їхніх координат. Такий спосіб збереження масиву скорочує обсяг пам'яті більш ніж у два рази, проте є незручним для реалізації операцій над елементами, оскільки доступ до заданого елемента потребує повного перебору.

Є інший спосіб – збереження за допомогою зв'язаних структур, за якого для кожного ненульового елементу зберігають його значення, номери рядка та стовпця, а також покажчик на наступний елемент. Такий спосіб потребує більших обсягів пам'яті, проте дозволяє реалізовувати операції над масивами швидше за подання даних у вигляді одновимірного масиву.

Якщо часто виникає потреба в реалізації операцій додавання та вилучення елементів матриці, то доцільніше вибрати метод зв'язаних структур, оскільки за використання послідовного методу зберігання ненульових елементів та їхніх координат ці операції потребують переміщення великої кількості інших елементів.

Слід зауважити, що метод зв'язаних структур переводить масив як структуру даних в інший розділ класифікації: логічна структура даних залишається статичною, а фізична структура набуває динамічного характеру.

2.6.3. Множини

Множини часто використовують у програмуванні, хоча їхня реалізація потребує визначення користувацьких типів через те, що цей тип даних не є стандартним для деяких мов програмування.

Множина – структура даних, для якої є характерними унікальність та однорідність набору даних, а також довільний порядок розташування елементів.

Основною операцією із множинами є визначення того, чи належить цей елемент множині. Множина, яка не містить елементів, є *порожньою*, або *нульовою* множиною.

Множина є *підмножиною* іншої множини, якщо всі її елементи містяться в іншій множині. Множина є *надмножиною* іншої множини, якщо вона містить усі елементи цієї множини.

Якщо елемент входить до складу множини, то він є членом цієї множини.

Над множинами визначено такі специфічні операції (рис. 2.5):

об’єднання множин: результатом є множина, яка містить всі елементи початкових множин;

переріз множин: результатом є множина, яка містить елементи, що є членами обох початкових множин;

різниця множин: результатом є множина, яка містить елементи, що є членами першої множини, але не є членами другої множини;

симетрична різниця: результатом є множина, яка містить елементи, що не є членами відразу двох множин;

перевірка на входження елемента до множини: результатом є значення булевого типу, що вказує, чи є елемент членом множини.

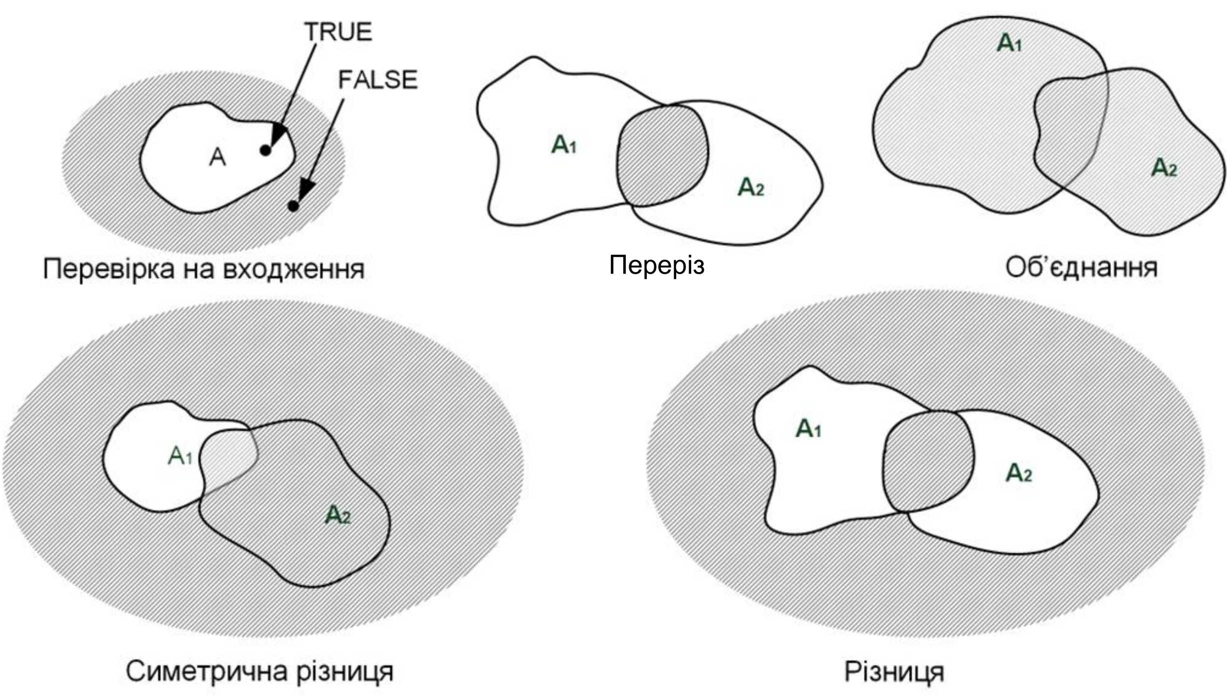


Рис. 2.5. **Подання операцій над множинами діаграмами Ейлера**

2.6.4. Структури

Відмінність структур полягає в тому, що вона може містити елементи різних типів. Елементи структури або її поля можуть мати довільний тип, за винятком типу цієї самої структури, проте можуть бути покажчиками

на неї. Якщо під час опису структури не вказано тип структури, то має бути зазначеним список змінних, покажчиків або масивів цієї структури. На етапі компіляції звернення до окремих полів структури заміщають їхніми адресами.

Основною операцією над структурами є операція доступу до певного поля, або операція кваліфікації. Після кваліфікації поля структури над ним можна реалізовувати будь-які операції, які є допустимими для типів даних цього поля.

У більшості мов програмування операції роботи із структурою розглядають її як єдине ціле, а не набір окремих полів.

Окремим випадком структури є об'єднання – це структура, усі поля якої мають одну адресу [9]. Об'єднання має такий самий формат опису, як і структура. Поля об'єднання зберігають почергово в одному й тому самому просторі пам'яті, тому розмір об'єднання дорівнює розміру найбільшого поля в об'єднанні. Наприклад, якщо об'єднання містить три поля: ціле число, дійсне число з рухомою комою та символ, і розмір цілого числа становить 4 байти, дійсного числа – 8 байтів, а символу – 1 байт, то розмір цього об'єднання буде 8 байтів, оскільки це найбільший розмір із трьох полів. За використання об'єднань у програмуванні важливо пам'ятати про можливість виникнення конфліктів під час доступу до різних полів об'єднання, оскільки їх зберігають в одному просторі пам'яті.

Об'єднання можна використовувати, із метою економії пам'яті, але лише в тих випадках, коли відомо, що більше від одного поля одночасно непотрібно, а також для різної інтерпретації одного й того самого бітового подання.

Іноді використовують комбінацію структурного типу й об'єднання, якщо об'єкти програми належать до одного класу, але відрізняються деталями. За цього способу об'єднання мають сенс полів структури, а до структури додають поле, що визначає, який саме елемент об'єднання застосовують у кожний момент.

2.6.5. Бітові типи

У програмуванні **бітові типи (бітові поля)** – це типи даних, які дозволяють зберігати окремі біти в пам'яті замість цілих байтів. Бітові типи зазвичай використовують для економії пам'яті й оптимізації розміру даних.

Бітові типи можуть мати різний розмір, зазвичай від 1 до 32 бітів. Для оголошення бітового поля можна використовувати ключові слова, як-от bit, bool або flag.

Найчастіше задачі, які потребують операцій над окремими бінарними розрядами, виникають у системному програмуванні. Бітові типи можуть бути корисними також у випадках, якщо потрібно зберігати велику кількість булевих значень або бітових прапорців. Однак під час використання бітових полів слід ураховувати, що доступ до окремих бітів може бути повільнішим і менш ефективним, ніж доступ до цілих байтів.

Дані бітового типу подають набором бітів, запакованих у байти або слова, які є логічно не пов’язаними один із одним.

Наведімо кілька випадків використання бітових полів:

бітові поля можуть бути використаними для збереження бітових прапорців, які вказують на стан об’єктів або подій (наприклад, у програмі можуть бути прапорці, що позначають, чи було здійснено вхід користувачем, чи зареєстровано певну подію тощо);

бітові поля використовують для економії пам’яті, оскільки можна зберігати більше значень в одному байті, тим самим знижуючи обсяг пам’яті, яку потрібно виділити для зберігання даних;

бітові поля можуть бути корисними для оптимізації роботи з пам’яттю, оскільки доступ до окремих бітів може бути менш витратним, ніж доступ до цілих байтів;

бітові поля можуть бути використаними для роботи з масивами булевих значень, де кожен елемент масиву містить один біт, це дозволяє зменшити кількість потрібних операцій для роботи з таким масивом;

бітові поля можуть бути використаними для кодування даних у форматі, за якого значення окремих бітів позначають певні властивості або параметри об’єктів (наприклад, бітові поля можуть бути використаними для кодування зображень, аудіофайлів та інших мультимедійних даних).

Операції з бітовими даними, які можна виконувати в програмуванні, містять:

логічні операції: AND (&), OR (|), XOR (^), NOT (~) – ці операції виконують над окремими бітами двох бітових значень, результатом є нове бітове значення;

зсув бітів: «<<», «>>» – й операції зміщують біти вліво або вправо на задану кількість позицій;

перетворення типів – ці операції дозволяють перетворювати значення із одного типу даних на інший, включно з бітовими типами;

порівняння: «==» (рівність), «!=» (нерівність), «<» (менше), «>» (більше), «<=» (менше або дорівнює), «>=» (більше або дорівнює) – ці операції дозволяють порівнювати значення бітових полів і встановлювати умови для виконання дій на їхній основі;

бітові операції надання: «&=», «|=», «^=», «<<=», «>>=» – ці операції виконують операцію над значенням бітового поля та надають його результат вибраній змінній.

Операції використання бітових полів у структурах даних: бітові поля можуть використовувати у складних структурах даних для економії пам'яті та забезпечення більш ефективної роботи з даними. Ці операції можуть бути корисними під час роботи з бітовими даними, оскільки вони дозволяють виконувати широкий спектр операцій із цими даними.

2.6.6.Таблиці

Іноді елементами векторів або масивів є інтегровані структури. Однією з таких складних структур є таблиця, яка за фізичного подання є вектором, елементи якого – структурами. Для логічного подання таблиць є характерним доступ до елементів за ключем, а не за індексом як у звичайному масиві. Ключ – це властивість об'єкта, яка однозначно ідентифікує певну структуру в множині однотипних структур. Такий ключ є унікальним первинним ключем таблиці. Ключ може бути складовою частиною структури або полем таблиці. Якщо ключ є самостійною одиницею і не входить до складу структури, його значення обчислюють за значеннями атрибутів певної таблиці. Таблиця може мати більше ніж один ключ.

Головна операція під час роботи з таблицями – це операція доступу до структури за значенням ключа, яку реалізують процедурою пошуку. Під час упорядкування таблиць за значеннями ключів процедуру пошуку виконують ефективніше, тому над таблицями доцільним є виконання операції сортування.

Виокремлюють таблиці із фіксованою довжиною структури та змінною довжиною. Таблиці, які об'єднують структури однакових типів, мають фіксовану довжину структур. Варіабельність довжини структури виникає в задачах, де розглядають об'єднання. Проте таблиці, які використовують

для розв'язання таких задач, може бути подано таблицями з фіксованою структурою максимальної довжини.

Потребу в таблицях, які мають дійсно змінну довжину структури, виникає вкрай рідко. Використання таблиць зі змінною довжиною структур економить пам'ять, проте значно звужує можливості роботи з даними: у цьому разі не можна визначити адресу структури за її номером. Таблиці зі змінною довжиною структур можуть опрацьовувати дані лише послідовно, у порядку зростання номерів структур. Операцію доступу до елемента такої таблиці зазвичай здійснюють у два етапи. На першому етапі визначають постійну частину структури, яка явно або неявно містить довжину структури, на другому – за довжиною структури визначають змінну частину. Тоді адресу наступної структури можна визначити додаванням довжини структури до адреси поточної структури.

Оскільки кожен елемент таблиці зі змінною довжиною структури може мати різний розмір, звернення до елемента може бути більш витратним, ніж звернення до елемента таблиці з фіксованою довжиною структури. Тому в деяких випадках можуть використовувати оптимізацію для прискорення доступу до даних у таблиці зі змінною довжиною структури.

2.7. Напівстатичні структури даних

Напівстатичні структури даних характеризуються тим, що їхній розмір може змінюватися, але зміни розміру відбуваються тільки в обмеженому діапазоні [10]. Це означає, що структура даних може збільшуватися або зменшуватися, але тільки до певного розміру.

Напівстатичні структури даних зазвичай використовують для збереження даних, які мають динамічний розмір або потрібно зберігати велику кількість даних, але пам'ять є обмеженою. Водночас важливо, щоб розмір структури даних не перевищував доступний обсяг пам'яті.

Одним із прикладів напівстатичної структури даних є *динамічний масив* – це масив, розміри якого можуть змінюватися під час виконання програми. Іншим прикладом напівстатичної структури даних є *дерево зміни* – це структура даних, яка зберігає в собі зміни, які було внесено в документ, зазвичай, текстовий файл. Дерево зміни дозволяє відновлювати стан документа на будь-який момент часу та виконувати операції перетворення та відновлення.

Використання напівстатичних структур даних передбачає прості процедури змінення розміру даних. Однак слід уважно планувати розмір і механізм їхнього змінення, щоб уникнути перевищення доступного обсягу пам'яті або надто великих часових витрат на змінення розміру таких структур.

За логічного подання напівстатична структура є послідовністю даних, пов'язаних відносинами лінійного списку, доступ до елемента якої здійснюють за порядковим номером.

За фізичного подання напівстатична структура даних є набором комірок пам'яті комп'ютера, у якому кожний наступний елемент розташовують у наступній комірці. Фізичне подання має вигляд однозв'язного списку (ланцюга), елементи якого мають покажчик для посилання на наступний елемент.

2.7.1. Стеки

Стек – це абстрактна структура даних, яка дозволяє зберігати й управляти даними згідно із принципом «останнім прийшов – першим пішов» (LIFO, last-in-first-out). Стек є послідовним списком змінної довжини, додавання та вилучення елементів із якого виконують лише з кінця списку. Елемент, який був доданим останнім, є вершиною стека. Вершина є особливим елементом, лише його можна модифікувати та вилучати. Інша частина стека є тілом стека. Стек є рекурсивною структурою, оскільки тіло стека так само є стеком, у якому можна виокремити вершину та тіло [2].

Стек може бути реалізованим за допомогою масиву або зв'язного списку. У випадку з масивом стек зазвичай має фіксований розмір, і для додавання нового елемента до стека, програма перевіряє, чи є вільне місце в масиві. У випадку зі зв'язним списком стек може мати довільний розмір, і для додавання нового елемента до стека, програма створює новий вузол списку та пов'язує його з попереднім елементом.

Прикладом використання стека є збереження контексту виклику функцій у комп'ютерних програмах. Кожен раз, коли програма викликає нову функцію, стек зберігає поточний стан програми, включно зі значенням змінних, адресу повернення та іншу інформацію. Коли функція завершує свою роботу та повертає значення, стек відновлює попередній стан програми та продовжує виконання.

Стек також може бути використаним для розв'язання різних задач, як-от перетворення виразів з інфіксного на префіксний запис, обхід дерев за допомогою алгоритму DFS (depth-first search) та багатьох інших.

Основними операціями над стеком є додавання елемента (push) і вилучення елемента (pop). Над стеком виконують такі операції:

- додавання нового елемента;
- визначення, чи є стек порожнім;
- перегляд елемента, який було додано останнім;
- вилучення елемента, який було додано останнім;
- визначення поточної кількості елементів у стеку;
- очищення стека;
- зчитування елемента з вершини стека без вилучення – комбінація двох основних операцій: вилучення елемента зі стека та повторне його додавання до стека.

За статичного подання стек є вектором із додатковим параметром – адресою вершини стека. За такого подання розмір стека є обмеженим розміром вектора.

Адресу вершини стека можна визначати або адресою першого вільного елемента стека, або адресою останнього записаного в стек елемента. Різниця між цими варіантами є несуттєвою, проте після вибору одного з варіантів слід чітко дотримуватися його під час опрацювання стека. Під час додавання елемента в стек його записують за адресою, яку визначають покажчиком, а потім покажчик змінюють таким чином, щоб він указував на наступне вільне місце. Якщо покажчик має адресу останнього доданого елементу, то спочатку змінюють покажчик, а потім відбувається запис елемента. Зміна покажчика полягає у його збільшенні або зменшенні на одиницю.

Операція вилучення елемента полягає в зміненні покажчика стека в напрямі, зворотному від зміни за додавання та зчитування значення, на яке вказує покажчик. Потім комірку, у якій містився вилучений елемент, уважають вільною.

Операція очищення стека полягає у вилученні всіх елементів зі стека, водночас стек стає порожнім. Для очищення стека програма може виконати послідовне вилучення кожного елемента зі стека, доки стек не стане порожнім. Операція очищення стека передбачає запис у покажчик початкового значення, тобто адреси ділянки пам'яті, яку виділено

під стек. Розмір стека визначають шляхом обчислення різниці покажчиків: покажчика стека й адреси початку ділянки пам'яті, відведеної під стек.

За зв'язного подання стека кожен його елемент складається зі значення (ключа) та покажчика – адреси останнього доданого до стека елемента. Наявність покажчиків призводить до зайвих витрат пам'яті під час використання зв'язаного подання стека. Тому таке подання є доцільним лише тоді, коли неможливо заздалегідь визначити максимальний розмір стека.

Виконання операцій зі стеком ґрунтується на покажчику стека. Створення порожнього стека передбачає надання покажчику нульового значення.

Алгоритм операції **додавання елемента в стек** такий:

виділення пам'яті для збереження нового елемента;

занесення значення (ключа) в інформаційне поле нового елемента;

установлення зв'язку між новим елементом та попередньою вершиною стека;

переміщення вершини стека на новий елемент (зміна покажчика стека).

Алгоритм операції **вилучення елемента зі стека** такий:

зчитування інформації з інформаційного поля вершини стека;

установлення на вершину стека допоміжного покажчика;

переміщення покажчика вершини стека на наступний елемент (зміна покажчика стека);

звільнення пам'яті, яку займав вилучений елемент.

2.7.2. Черги

Черга – це абстрактна структура даних, яка працює за принципом «першим прийшов – першим вийшов» (FIFO, first-in-first-out). Додавання нового елемента до черги (enqueue) та вилучення елемента (dequeue) відбувається з різних боків: додавання елемента здійснюють лише із хвоста черги, а вилучення – із голови.

Черга може бути реалізованою за допомогою масиву або зв'язного списку. У випадку з масивом черга зазвичай має фіксований розмір, і для додавання нового елемента до черги програма перевіряє, чи є вільне місце в масиві. У випадку зі зв'язним списком черга може мати довільний

розмір, і для додавання нового елемента до черги програма створює новий вузол списку та зв'язує його із попереднім елементом.

Чергу використовують у різних сферах, наприклад, в операційних системах для управління процесами, інтернет-протоколах для збереження та передавання повідомлень, комп'ютерних мережах для організації передавання даних тощо.

Одним із прикладів використання черги є операція опрацювання даних у порядку їхнього надходження. Наприклад, якщо потрібно опрацювати багато завдань, які приходять від користувачів, можна використовувати чергу, щоб зберігати ці завдання в порядку їхнього надходження. Водночас програма може виконувати перше завдання із черги та переходити до наступного завдання. Це дозволяє забезпечувати справедливе опрацювання завдань у порядку їхнього надходження та уникати багатозадачності.

Над чергою можна виконувати такі операції:

додавання нових даних;

визначення чи містить черга елементи;

зчитування даних, які були доданими першими;

вилучення даних, які були доданими першими;

визначення розміру черги;

очищення черги;

зчитування елемента без вилучення.

За статистичного подання черга є вектором із двома додатковими параметрами: покажчиком на голову та покажчиком на хвіст черги. Під час додавання елемента в чергу його записують за адресою, яка є покажчиком на хвіст. Після цього покажчик збільшують на одиницю. Під час вилучення елемента із черги зчитується елемент, на який указує покажчик голови черги. Після цього покажчик голови зменшують на одиницю.

Кільцева черга (англ. circular queue, circular buffer) – це варіант черги, у якому кінці черги є з'єднаними з утворенням кола, або кільця. Це означає, що якщо черга є заповненою, наступний елемент буде доданим на початок черги.

У кільцевій черзі використовують два покажчики: початковий і кінцевий. Початковий покажчик указує на перший елемент черги, а кінцевий покажчик – на місце, де потрібно додати новий елемент. Коли елемент

вилучають із черги, початковий покажчик зсувається вправо, щоб указувати на наступний елемент.

Кільцева черга може бути корисною в тих випадках, якщо потрібно зберігати обмежену кількість елементів у пам'яті або якщо слід опрацьовувати дані в порядку їхнього надходження. Також вона може бути використаною для збереження потоків даних у реальному часі, якщо важливо забезпечити неперервний потік даних.

Кільцева черга має деякі переваги над звичайною чергою, як-от більш ефективне використання пам'яті та швидкий доступ до даних, оскільки вона забезпечує константний час доступу до будь-якого елемента черги. Однак, на відміну від звичайної черги, кільцева черга має фіксований розмір.

Якщо покажчик на голову черги дорівнює покажчику на хвіст, це свідчить про те, що черга є порожньою. Якщо під час роботи з кільцевою чергою кількість операцій додавання елементів є більшою за кількість операцій вилучення, то може виникнути ситуація, за якої покажчик кінця зрівняється з покажчиком початку, тобто черга буде заповненою. Проте покажчики на кінець і початок дорівнюють один одному й за порожньої черги. Щоб розрізняти ці випадки, за роботи з кільцевою чергою використовують наявність вільних елементів між покажчиками кінця та початку. Якщо кількість вільних елементів зменшується до одного, то чергу вважають заповненою та подальші спроби запису блокують. Операція очищення черги передбачає запис одного й того самого значення до обох покажчиків. Розмір черги визначають різницею покажчиків з урахуванням кільцевої природи черги.

За зв'язного подання кожен елемент черги складається із двох полів: інформаційного поля (ключа) та покажчика на попередній елемент черги. Через зайву витрату пам'яті для збереження покажчиків зв'язне подання є доцільним лише у випадках, коли складно заздалегідь визначити максимальний розмір черги.

Виконання операцій із чергою ґрунтується на двох покажчиках: покажчик на голову та покажчик на хвіст черги. Створення порожньої черги полягає в наданні цим покажчикам нульових значень.

Алгоритм **додавання елемента в** чергу такий:

виділення пам'яті під новий елемент;

занесення значення до інформаційного поля елемента;

для нового елемента занесення нульового значення до покажчика на наступний елемент;

установлення зв'язку між новим елементом і попереднім останнім елементом черги, з урахуванням можливості порожньої черги;

переміщення покажчика кінця черги на новий елемент.

Алгоритм **вилучення елемента** із черги такий:

зчитування інформації (ключа) з інформаційного поля елемента, який є початком черги;

установлення допоміжного покажчика на початок черги;

переміщення покажчика початку черги на наступний елемент;

звільнення пам'яті, яку займав вилучений елемент черги.

Виконання деяких завдань потребує формування черг, порядок вибору елементів у яких залежить від пріоритету елемента. Такий пріоритет зазвичай є числом, яке обчислюють за значенням ключа елемента або зовнішніми параметрами. Так, наприклад стеки та черги можна розглядати як структури, пріоритет елементів яких визначають часом додавання елемента до структури. Тоді під час роботи визначають елемент із максимальним пріоритетом. Пріоритет елемента можна враховувати або під час додавання, тоді вилучають елемент лише з голови черги, або під час вилучення, тоді під час додавання елемент завжди записують у кінець черги. За обох варіантів потрібна певна організація доступу до певного елемента. У разі статичної пам'яті потрібним є ще й переміщення елементів.

Реалізація черг із пріоритетами на лінійних структурах може бути поданою послідовними або зв'язними структурами.

2.7.3. Деки

Дек (від англ. *deq – double ended queue*) можна розглядати як особливий вид черги, який може бути подовженим або скороченим із двох боків. Отже, дек є таким послідовним списком, де й додавання, і вилучення елементів можна здійснювати з довільного краю списку. Інакше кажучи, дек є стеком, де дозволено додавання та вилучення елементів з обох боків. Тоді як деки в класичному розумінні використовують не часто, окремі їхні випадки з обмеженим входом і виходом є достатньо затребуваними.

Логічна та фізична структура дека є подібною замкнутій у кільце черзі, проте щодо дека краще називати його боки не головою та хвостом, а лівим і правим краєм.

Під час роботи з деком виділяють такі операції, як додавання та вилучення елементів справа та зліва, визначення розміру та очищення. У статичній пам’яті фізична структура дека є повністю ідентичною структурі кільцевої черги.

2.7.4. Лінійні списки

Узагальненням усіх раніше розглянутих структур є **лінійні списки**, які дозволяють подавати множини в такий спосіб, щоб забезпечити досяжність кожного елемента. Водночас, щоб зберегти послідовність розташування, немає потреби переміщувати будь-які елементи.

Списки є достатньо гнучким елементом структур даних, який може динамічно змінювати свою довжину шляхом вилучення та вставляння елементів у будь-яку позицію. До того ж над списками можна виконувати операції об’єднання або розподілу.

Лінійний список становить кінцеву послідовність елементів (вузлів) одного типу, не обов’язково унікальних. Отже, лінійний список передбачає наявність повторень. Під довжиною списку мають на увазі кількість елементів послідовності, яка в загальному випадку не є сталою та може змінюватися в ході виконання програми. Лінійний список L, до складу якого входять елементи одного типу d1, d2,…, dn, можна записати як L = [d1, d2,…, dn] або подати графічно (рис. 2.6).

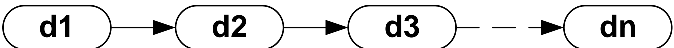


Рис. 2.6. **Графічне подання лінійного списку**

Важливою рисою лінійного списку є той факт, що його елементи можуть бути лінійно впорядкованими за їхньою позицією в послідовності.

Для формування абстрактного типу даних на основі математичного визначення списку потрібно задати множину операторів, які виконують над об’єктами типу список. Проте немає вичерпного списку операторів, які виконують над списками та можуть задовольнити всі можливі запити.

До операцій, що найчастіше доводиться виконувати зі списками, можна зарахувати такі:

- пошук елемента за заданою властивістю;
- визначення в лінійному списку *i*-го елемента;
- унесення додаткового елементу до списку в наперед указане місце (до або після певного вузла);
- вилучення певного елемента списку;
- упорядкування вузлів лінійного списку.

Сучасні мови програмування не мають якої-небудь спеціальної структури для подання лінійних списків таким чином, щоб усі операції над ними виконували однаково ефективно [11]. Тому під час виконання певного завдання із застосуванням можливостей лінійних списків, важливо вибирати саме таке їхнє подання, що забезпечить максимальну ефективність як за часом виконання програми, так і за обсягом пам’яті, що має бути використаною. Лінійний список є послідовністю об’єктів певного типу, тоді як тип даних позиції елемента відрізняється від типу даних об’єктів і залежить від конкретної фізичної реалізації.

Визначмо дію операцій, які є припустимими під час роботи з лінійним списком.

Операція вставляння додає елемент у конкретну позицію списку. Водночас елемент, що раніше займав цю позицію, переміщується до більш високої позиції.

Операція локалізації повертає позицію об’єкта в списку. За умови наявності в списку кількох однакових об’єктів операцією локалізації буде повернено позицію найближчого до початку списку об’єкта. За умови відсутності об’єкта в списку операцією локалізації буде повернено значення, що дорівнює збільшеній на одиницю довжині списку.

Операція вибору елемента зі списку повертає елемент, що відповідає шуканій позиції списку. За умови відсутності в списку такої позиції результат не буде визначеним.

Операція вилучення видаляє елемент, що відповідає шуканій позиції списку. За умови відсутності в списку такої позиції результат не буде визначеним.

Операції вибору попереднього й наступного елемента повертають, відповідно, попередній і наступний щодо конкретної позиції елементи списку.

Операція очищення вилучає всі елементи списку та робить його порожнім.

Важливим моментом організації ефективного процесу опрацювання та збереження даних є вибір методів зберігання лінійних списків, які загалом розподіляють на методи послідовного та зв'язного зберігання. Під час вибору методу зберігання для конкретної реалізації варто враховувати характер операцій, які будуть виконувати над лінійними списками, їхню частоту, вартість та обсяги пам'яті, потрібної для зберігання списку.

Подання лінійного списку у вигляді вектора є одним із найпростіших варіантів. Така форма опису списку дозволяє із застосуванням циклу по черзі звертатися до його елементів і за потреби виконувати операції над ними. Однак векторна форма подання не дозволить уникнути фізичного переміщення елементів, зокрема, у разі додавання нових об'єктів або вилучання наявних. Важливою опцією швидкого вилучення елементів є застосування простої схеми очищення пам'яті, коли замість того, щоб вилучати елементи зі списку, їх просто позначають як невикористані.

Дещо складнішим варіантом організації та зберігання списків є розміщення їх у вигляді масиву або формування списку, початок якого не є прив'язаним до першого елемента масиву.

У разі однозв'язного подання лінійного списку кожен елемент складається власне зі значення та покажчика на наступний елемент списку. До того ж список мусить мати спеціальний елемент, який є покажчиком початку, або головою списку (рис. 2.7).

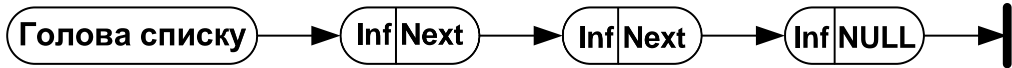


Рис. 2.7. **Структура однозв'язного списку**

Через неможливість просування у зворотному напрямку опрацювання даних, організованих як однозв'язний список, може виявитися незручним. Якщо організувати список таким чином, щоб кожний елемент містив покажчики на сусідні елементи (наступний і попередній), то цей недолік буде усунено. Саме так і побудовано двозв'язний список (рис. 2.8).

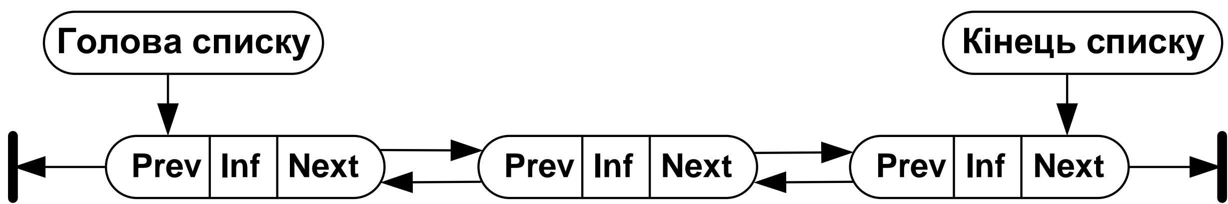


Рис. 2.8. **Структура двозв’язного списку**

Інколи для зручності опрацювання до списку вносять додатковий елемент, що є покажчиком кінця списку. З одного боку, за наявності в кожного елемента двох покажчиків спрощується виконання обчислювальних операцій над списком, з іншого – це призводить до ускладнення списку та зайвих витрат пам’яті. До того ж, щоб уникнути потрапляння в замкнений цикл, під час перегляду елементів такого списку слід уживати запобіжних заходів.

На основі розглянутих раніше лінійних як одно-, так і двозв’язних списків може бути побудовано кільцеву структуру. У випадку кільцевої однозв’язної структури покажчик останнього елемента списку вказує на перший елемент, у випадку двозв’язної структури списку покажчики першого й останнього елементів змінюються (рис. 2.9).

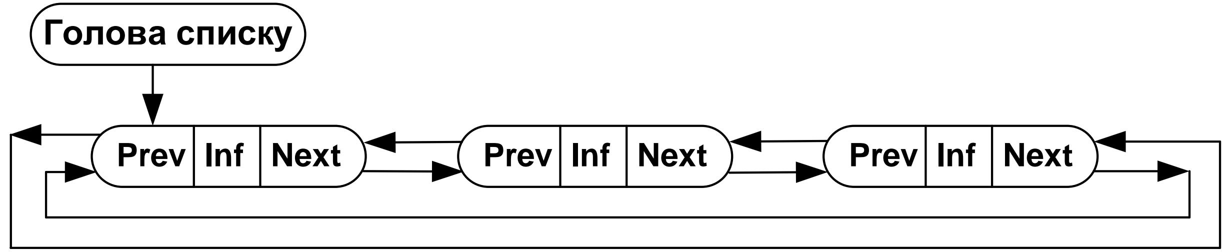


Рис. 2.9. **Структура кільцевих списків**

Із застосуванням списку описують сукупність структур, що мають однаковий розмір та формат і містяться в певній ділянці пам’яті. Ці структури є лінійно зв’язаними та впорядкованими із застосуванням покажчиків. Структура характеризується інформаційними полями й полями покажчиків на сусідні елементи. Інколи як поля інформаційної частини можуть бути покажчики на блоки пам’яті, до яких додають інформацію про елементи списку.

Серед базових операцій як з однозв’язними, так і двозв’язними лінійними списками виділяють уставляння (додавання) та вилучення елементів (рис. 2.10 – 2.14).

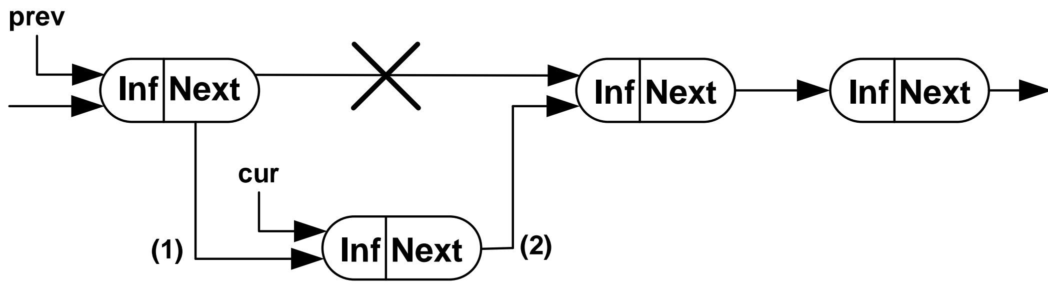


Рис. 2.10. **Уставляння елемента всередину однозв’язного списку**

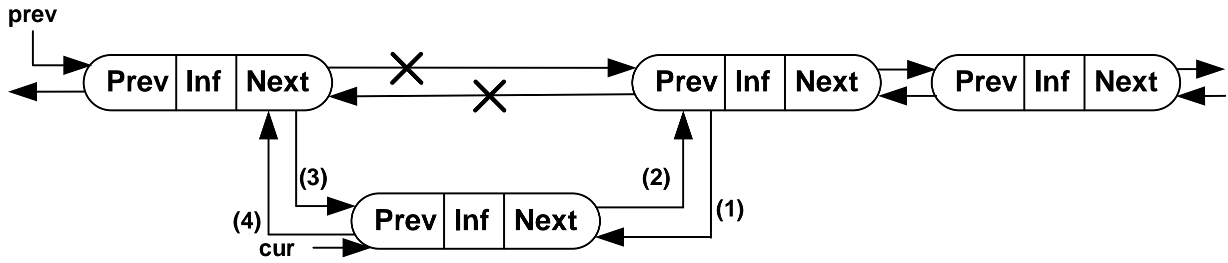


Рис. 2.11. **Уставляння елемента у двозв’язний список**

У разі спроби вставляння елемента на початку списку наведені раніше схеми не можуть бути застосованими, оскільки є потрібною модифікація покажчика на початок списку (див. рис. 2.12).

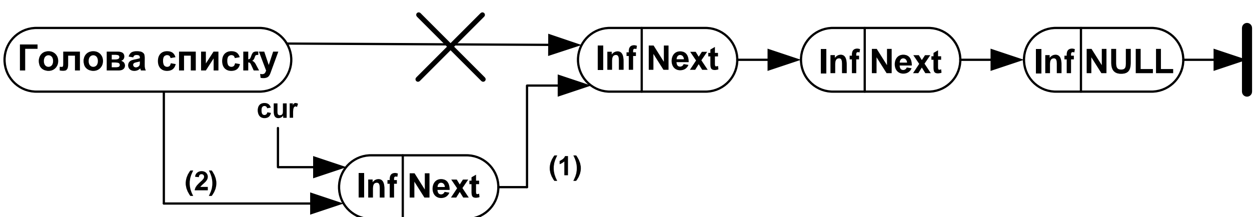


Рис. 2.12. **Модифікація покажчика на початок списку**

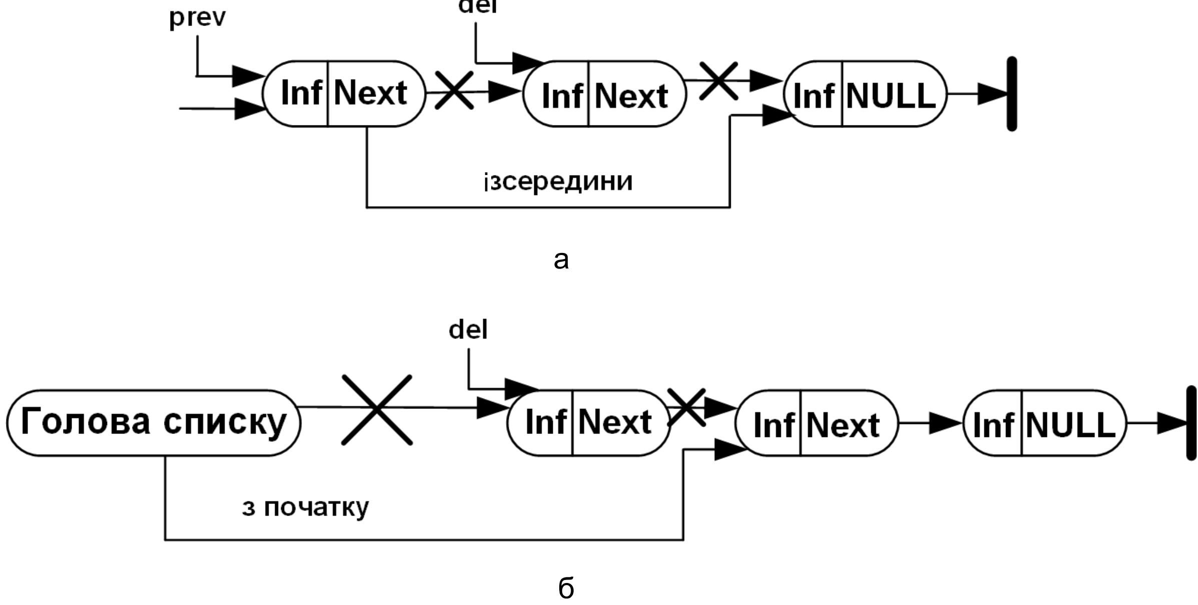


Рис. 2.13. **Вилучення елемента однозв’язного списку (ізсередини, із початку)**

Що стосується операції вилучення елемента із двозв’язного списку, то вона потребує модифікації більшої кількості покажчиків, ніж в однозв’язному списку, але виявляється простішою, оскільки дозволяє уникнути пошуку попереднього елемента, який за замовчуванням вибирають просто за покажчиком (див. рис. 2.14).

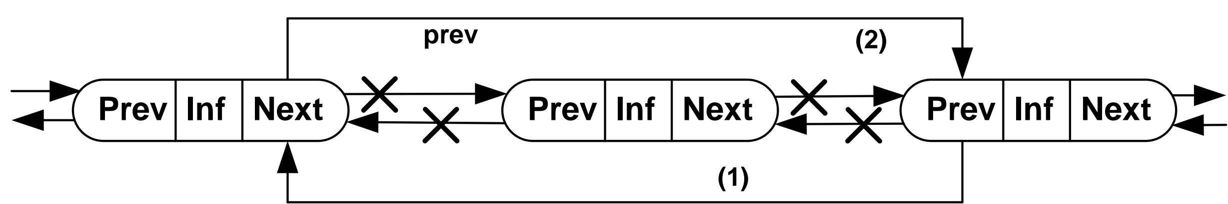


Рис. 2.14. **Вилучення елемента із двозв’язного списку**

Якщо для динамічних структур даних змінення як розмірів структури, так і зв’язків між елементами є цілком природними, то у зв’язних структурах досягти зміни зв’язків можна шляхом корекції покажчиків. Процес обміну місцями двох сусідніх елементів списку показано на рис. 2.15. У випадку однозв’язного списку організація процесу переставлення відбувається через те, що наперед відома адреса елемента, який стоїть

лівіше пари елементів, що мають бути переставленими. Випадок переставлення елементів на початку списку водночас не взято до розгляду. Що стосується двозв'язного списку, то організація переставлення на початку списку в цьому разі не спричиняє проблем і може бути реалізованою в межах стандартного алгоритму.

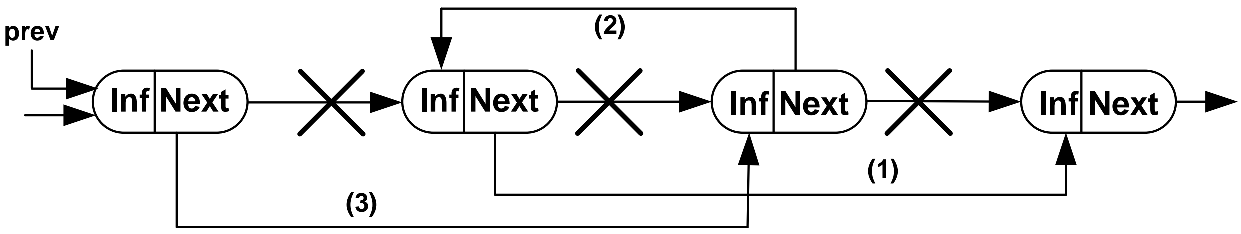


Рис. 2.15. **Процес переставлення двох сусідніх елементів списку**

2.7.5. Мультисписки

Під час опрацювання об'єктів складної структури виникають завдання, пов'язані із трансформацією даних, що належать до окремих підмножин. Для спрощення процесу виокремлення підмножин та усунення повного перебору з відсіванням записів, що не належать шуканим підмножинам, до кожного запису вносять додаткові поля з посиланнями, які зв'язують елементи окремих підмножин у відповідні лінійні списки. У результаті такої організації формують багатозв'язний список, або **мультисписок**. Елементи мультисписку одночасно можуть належати кільком різним однозв'язним спискам.

Мультисписки дозволяють таке:

економити пам'ять (інформаційна частина списків, об'єднаних у мультисписок, фізично є єдиною, яку використовують усі списки-складові);

забезпечувати цілісність даних (усі підзавдання звертаються до єдиної версії інформаційної частини, отже, результати виконання кожного підзавдання в реальному масштабі часу є досяжними всім іншим підзавданням);

кожному підзавданню за використання полів зв'язку працювати зі своєю підмножиною як із лінійним списком.

Відмінність роботи з мультисписком спостерігають тільки під час виконання операції вилучення елемента списку. У разі вилучення елемента з окремого списку не обов'язково вилучати його з пам'яті, він може входити до складу інших списків. Вилучення елемента з пам'яті є обов'язковими тільки, якщо він не є елементом жодного списку мультисписку. За умови наявності головного списку, який містить усі елементи мультисписку, завдання вилучення значно спрощується. Щоб вилучити елементи з неголовних списків, достатньо змінити покажчики, не залучаючи для цього пам'ять. У разі вилучення елементів із головного списку потрібним є як очищення пам'яті, так і зміна покажчиків у кожному списку, до якого елемент, що вилучають, належав.

2.7.6. Рядки (Strings)

Рядок є лінійною впорядкованою послідовністю символів, які належать до кінцевої множини символів, що утворюють алфавіт.

Рядкам є притаманними такі ключові властивості:

- довжина, що є, у загальному випадку, змінною за фіксованого алфавіту;
- звернення до елементів рядка відбувається з одного боку послідовності (значення має впорядкованість послідовності, не індексація);
- метою доступу до рядка є ланцюжок символів, а не окремий елемент.

Коли мова йде про рядки за замовчуванням це рядки тексту, що складаються із символів алфавіту певної мови, цифр, пунктуаційних знаків та службових символів. *Текстовий рядок* є універсальною формою подання інформації.

Рядки належать до напівстатичних структур даних, водночас їхня динамічність може варіюватися від повної відсутності до практично необмежених можливостей мінливості. Орієнтація на певну міру мінливості рядків визначає їхнє фізичне подання в пам'яті та специфіку виконуваних над ними операцій. У більшості мов програмування рядки вважають напівстатичними структурами.

Список базових операцій над рядками містить такі позиції:

- визначення довжини рядка;
- надання;

- порівняння;
- конкатенація, або зчеплення;
- виділення підрядка;
- пошук входження.

Операція визначення довжини рядка є функцією, що повертає поточну кількість (ціле число) символів у рядку.

Операції надання та **порівняння рядків** мають такий самий сенс, що й для інших типів даних.

У разі **порівняння** спочатку відбувається порівняння перших символів двох рядків. Якщо символи не є тотожними, рядок із символом, місце якого буде ближчим до початку алфавіту, визнають меншим. У разі тотожності символів переходять до порівняння других символів, потім третіх тощо. За умови різної довжини рядків коротший рядок визнають меншим. За умови однакової довжини рядків та попарної тотожності всіх символів рядки визнають однаковими.

Операція зчеплення двох рядків повертає рядок, довжина якого дорівнює сумі довжин рядків-операндів, а значення містять усі символи першого операнда, безпосередньо за яким без змін слідують символи другого операнда.

Операція виділення підрядка повертає послідовність символів первинного рядка із заданої позиції та заданою довжиною.

Операція пошуку входження повертає покажчик на місце першого входження підрядка-еталона в первинний рядок. У результаті операції має бути повернуто номер позиції первинного рядка, де вперше спостерігають входження еталона (покажчик початку входження). За відсутності входження операцією буде повернуто певне спеціальне значення, яким може бути порожній покажчик або від’ємний номер позиції.

У разі подання рядка вектором постійної довжини в пам’яті для запису символів рядка виділяють наперед фіксовану кількість байтів. Якщо відведений під рядок вектор є більшим за рядок, зайві місця позначають пропусками; якщо вектор не вміщає рядок повністю, зайві (справа) символи примусово відкидають. Щоб уникнути такої ситуації, є можливість використати вектор зі змінною довжиною та ознакою завершення.

Ознакою завершення є спеціальний символ, який належить алфавіту й має таку саму кількість розрядів, як і інші символи. Отже, у результаті кількість робочих символів алфавіту зменшено на один. Щодо витрат пам’яті, то вони становлять один символ на рядок.

Одним із методів забезпечення можливості змінювати довжину рядків є використання лічильника символів, що є цілим числом, для якого відведено більш ніж достатню для рядка довільної довжини кількість бітів. Використання лічильника символів дозволяє організувати в межах рядка вільний доступ до символів.

Гнучкість виконання операцій над рядками, зокрема, додавання та вилучення окремих символів і їхніх ланцюжків, може бути забезпечено за поданням рядків у пам'яті списком. Водночас для надання рядку потрібних обсягів пам'яті використовують системні засоби управління нею, що призводить до додаткових її витрат. Ще одним недоліком цього подання рядків є той факт, що елементи-сусіди логічної схеми рядка не є сусідами у фізичній схемі пам'яті. Така розбіжність ускладнює доступ до ланцюжків елементів, якщо порівнювати, наприклад, із векторним поданням.

Під час подання рядка однозв'язним лінійним списком кожен символ рядка є елементом зв'язного списку, який містить код символу та покажчик на наступний елемент. Одностороннє зчеплення дозволяє виконувати операції, рухаючись тільки в одному напрямі вздовж рядка. За використання двозв'язних лінійних списків до кожного елемента списку додають ще покажчик на попередній елемент, що дозволяє організувати двосторонній рух уздовж списку, що приводить до значного підвищення ефективності виконання низки операцій із рядками.

Метод блоково-зв'язного подання рядків для більшості операцій дає можливість уникнути зайвих витрат, пов'язаних з управлінням динамічною пам'яттю. До того ж блоково-зв'язне подання рядків забезпечує високу ефективність використання пам'яті з рядками змінної довжини.

2.8. Динамічні структури даних

Характерною рисою динамічних структур є можливість зміни як самої структури, так і її розміру в процесі роботи програми. Це істотно підвищує гнучкість програм, а розмір структури обмежено тільки розміром пам'яті обчислювальної системи. На жаль, компенсувати цю перевагу доводиться витратами пам'яті на зберігання покажчиків та власне структури, а також на її опрацювання. Динамічні структури фізично не є суміжними з елементами структури в пам'яті, вони характеризуються мінливістю як складу, так і розміру (кількості елементів) під час опрацювання.

Оскільки елементи динамічної структури не є прив'язаними до певних адрес пам'яті, їхні координати неможливо обчислити за адресами первинного або попереднього елементів. Для встановлення явного зв'язку між елементами в динамічних структурах використовують покажчики. У пам'яті таке подання даних дістало назву *зв'язного*.

Кожен елемент динамічної структури має два поля:

- інформаційне, або даних, де, власне, і міститься та інформація, заради якої створювали структуру;
- зв'язку, де локалізовані один або декілька покажчиків, які відповідають за зв'язок цього елемента із іншими частинами структури.

За використання зв'язного подання даних під час розв'язання прикладних задач, користувач працює тільки зі вмістом інформаційного поля, тоді як програміст-розробник користується полями зв'язку.

Серед переваг зв'язного подання даних можна виділити такі:

- гнучкість (мінливість) структури;
- майже необмежений розмір структури (обмежений лише обсягами пам'яті обчислювальної системи);
- зміни логічної послідовності елементів структури досягають корекцією покажчиків, а не переміщенням даних у пам'яті.

З іншого боку, серед недоліків зв'язного подання можна виділити такі:

- більш високі вимоги до кваліфікації програміста у сфері роботи з покажчиками;
- потребу в додатковій пам'яті для роботи з полями зв'язку;
- значні часові витрати для здобуття доступу до елементів зв'язної структури.

Саме недостатньо висока ефективність за часом є найбільш вагомим аргументом, що обмежує застосування зв'язного подання даних. Якщо за суміжного подання даних обчислення адреси будь-якого елемента відбувається за номером елемента та інформацією в описі структури, то за зв'язного подання на основі первинних даних адресу елемента не обчислюють.

В описі зв'язної структури міститься один або декілька покажчиків, за якими дозволено увійти до структури та, рухаючись послідовно від елемента до елемента ланцюжком покажчиків, здійснити пошук потрібного елемента. Саме тому в задачах з векторами або масивами з доступом

за номерами елементів як логічну структуру даних зв'язне подання майже ніколи не застосовують. Воно є доцільним під час роботи з таблицями, списками, деревами тощо, саме там, де логічна структура потребує первинної інформації іншого виду.

2.9. Нелінійні структури даних

2.9.1. Графи

Граф є складною нелінійною багатозв'язною динамічною структурою, яка подає об'єкти, їхні властивості та зв'язки й характеризується таким:

- довільною кількістю посилань на кожний елемент (вузол, вершину);
- кожний елемент є зв'язаним із довільною кількістю інших елементів;
- кожний зв'язок (ребро, дуга) може мати напрям і вагу.

Вузли графу містять інформацію про елементи об'єкта. Зв'язки між вузлами задають ребрами графу. Якщо ребра графу мають напрям, їх називають *орієнтованими*, якщо ні – *неорієнтованими*. Граф, який містить виключно орієнтовані зв'язки, називають *орієнтованим;* граф із неорієнтованими зв'язками – *неорієнтованим;* граф, який містить зв'язки обох типів, – *змішаним*.

Для подання графу в комп'ютерній пам'яті зазвичай використовують один із двох основних методів: матричний або зв'язними нелінійними списками. Природа даних та операції, які над ними виконують, визначають вибір методу подання графу. За потреби у виконанні значної кількості додавань і вилучень вузлів, є сенс подавати граф зв'язними списками; під час розв'язання задач, що цього не потребують, доцільно використати матричне подання.

У разі використання матриць суміжності в пам'яті комп'ютера їхні елементи подають як елементи масиву. Зазначмо, що для простого графу матриця містить тільки нулі та одиниці; для мультиграфу – нулі та цілі числа, що відповідають кратності відповідних ребер; для зваженого графу – нулі та дійсні числа, що характеризують ваги кожного з ребер.

У разі, коли орієнтований граф часто змінюють або півміри входу та виходу його вузлів є достатньо великими, його подають зв'язним нелінійним списком. Як багатозв'язні структури графи широко застосовують під час створення банків даних та управління базами даних, у системах

імітаційного моделювання складних комплексів, штучного інтелекту, планування тощо.

2.9.2. Дерева

Формально **дерево** можна визначити як граф Т з кінцевою множиною вузлів, який характеризується такими властивостями:

- дерево Т має єдиний корінь, тобто елемент (вузол чи вершину), на який жоден інший елемент не посилається;
- усі вузли, крім кореня, є розподіленими на множини Т1, Т2, …, Тm, що не перетинаються, причому кожна із цих множин також є деревом; дерева Т1, Т2, …, Тm ще називають *піддеревами кореня Т*;
- просуваючись за ланцюжком покажчиків від кореня вгору, за кінцевий проміжок часу можна досягти будь-якого елемента структури;
- усі елементи, за винятком кореня, адресовано єдиним покажчиком, тобто кожний елемент дерева має одне посилання.

Дерево дістало свою назву на основі асоціації з ієрархічною деревоподібною логічною структурою з теорії графів. Зв'язок між парою вузлів дерева за тією самою аналогією називають *гілкою*. Вузли, що не посилають на жоден інший вузол дерева, називають *листям*. Усі вузли, крім листя та кореня, називають *проміжними,* або *вузлами галуження*.

Зазвичай, послідовність проходження вершин у кожному ярусі за замовчуванням є заданою. Під час організації розміщення дерева в комп'ютерній пам'яті така послідовність формується автоматично. Послідовність проходження вершин на певному ярусі можна примусово змінити, позначаючи вершини одну за одною як першу, другу тощо. У разі потреби можна задавати послідовність проходження за ребрами замість вершин. За умови встановленої на кожному ярусі орієнтованого дерева послідовності проходження вершин, таке дерево вважають упорядкованим.

2.10. Алгоритми сортування

Алгоритми сортування виконують завдання сортування, тобто здійснюють упорядкування масиву елементів. Сам термін «сортування» (*sorting*) означає розподіл елементів за певною ознакою, що не зовсім коректно описує сутність завдання. Точніше було б назвати це завдання *алгоритмом упорядкування* (*ordering*), проте через перевантаженість

слова «порядок» *(order)* різноманітними значеннями, закріпився термін «сортування».

Основними характеристиками алгоритму сортування є такі:

1) час, потрібний на впорядкування масиву з n елементів. Для значної кількості алгоритмів середній і найгірший час упорядкування масиву з n елементів є $O(n^2)$, оскільки передбачено переставляння елементів, що стоять поряд (різниця між індексами елементів не перевищує деякого заданого числа). Такі алгоритми зазвичай є стабільними, хоч і неефективними для великих масивів. Є клас алгоритмів, які здійснюють упорядкування за час O(n logn). Такі алгоритми використовують можливість обміну місцями елементів, які містяться на певній відстані один від одного;

2) потреба в додатковій пам’яті для сортування;

3) стабільність сортування: сортування, за якого взаємне розташування елементів з однаковими значеннями не змінюють.

Процес сортування модифікує масив, тобто замість первинного повертає масив, елементи якого розташовано в порядку, що задовольняє умови сортування.

Застосовують три методи сортування:

уставляннями, або простими додаваннями;

вибором;

обміном [2].

2.10.1. Сортування вставлянням (простим додаванням)

Реалізація алгоритму простими додаваннями, або сортування вставляннями передбачає інкрементний підхід, за якого елементи умовно розподіляють на вже відсортовану послідовність a1, a2, …., ai − 1 та початкову послідовність ai, …. an. На кожному кроці алгоритму вибирають один з елементів початкових даних і вставляють на потрібну позицію у вже відсортованому наборі даних. Ця процедура відбувається доки в послідовності початкових даних ще є невідсортовані елементи. Спосіб вибору чергового елемента з початкового набору даних є довільним. Найчастіше елементи вибирають за порядком, у якому їх розташовано в початковому масиві.

Метод сортування вставляннями ґрунтується на тому, що всі записи, які передують R[j], а саме R[1], R[2], . . . , R[j − 1], уже є впорядкованими, а, отже, елемент R[j] вставляють у потрібне місце. Сортування

починають із другого елемента, значення якого порівнюють зі значенням першого елемента. Якщо впорядкованості між елементами R[1] та R[2] немає, їх міняють місцями. Потім значення запису R[3] порівнюють зі значеннями елементів R[2] та R[1]. Щойно виявляють, що (j + 1)-й елемент масиву є меншим за j-й (у разі сортування за зростанням), значення цього елемента копіюють у додаткову змінну. Потім із початку масиву до j-го елемента відбувається порівняння доти, доки значення додаткової змінної не буде меншим за значення будь-якого елемента x. Після чого частину масиву, починаючи з елемента x та закінчуючи *j*-м елементом, зміщують на одну позицію в бік зростання. На вільну позицію, яка утворилася, записують значення елемента, який переміщують. Потім відбувається переміщення основної частини масиву до елемента n − 1.

У книзі Т. Кормена «Алгоритми: побудова та аналіз» [4; 5] наведено візуальне подання роботи алгоритму вставляннями (рис. 2.16).

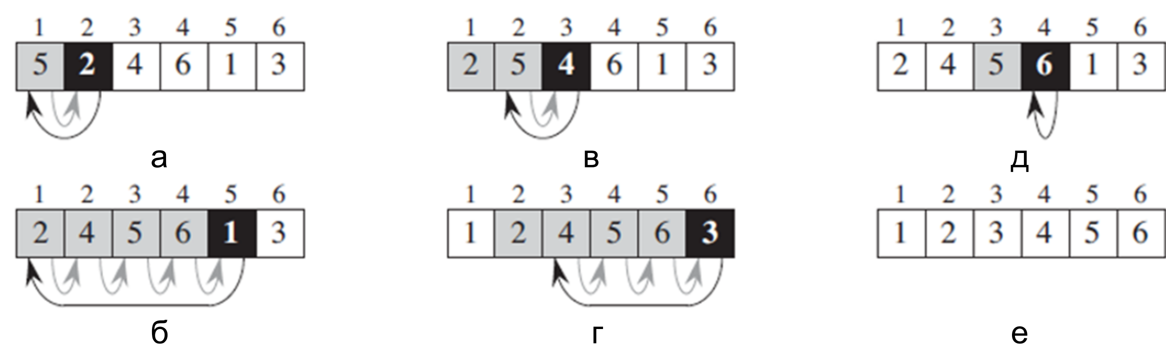


Рис. 2.16. **Сортування Insertion-Sort для масиву з елементами 5, 2, 4, 6, 1, 3**

Переваги методу сортування вставляннями (рис. 2.17) такі:

1) ефективність для невеликих наборів даних (чим менший набір, тим вища ефективність методу);

2) ефективність для наборів даних, які вже є частково відсортованими;

3) стійкість алгоритму (не змінює порядок елементів, які вже є відсортованими);

4) може виконувати сортування списку по мірі його складання;

5) може працювати значно швидше шляхом бінарного пошуку.

Головним недоліком сортування вставляннями є занадто висока обчислювальна складність (за використання стандартного алгоритму).

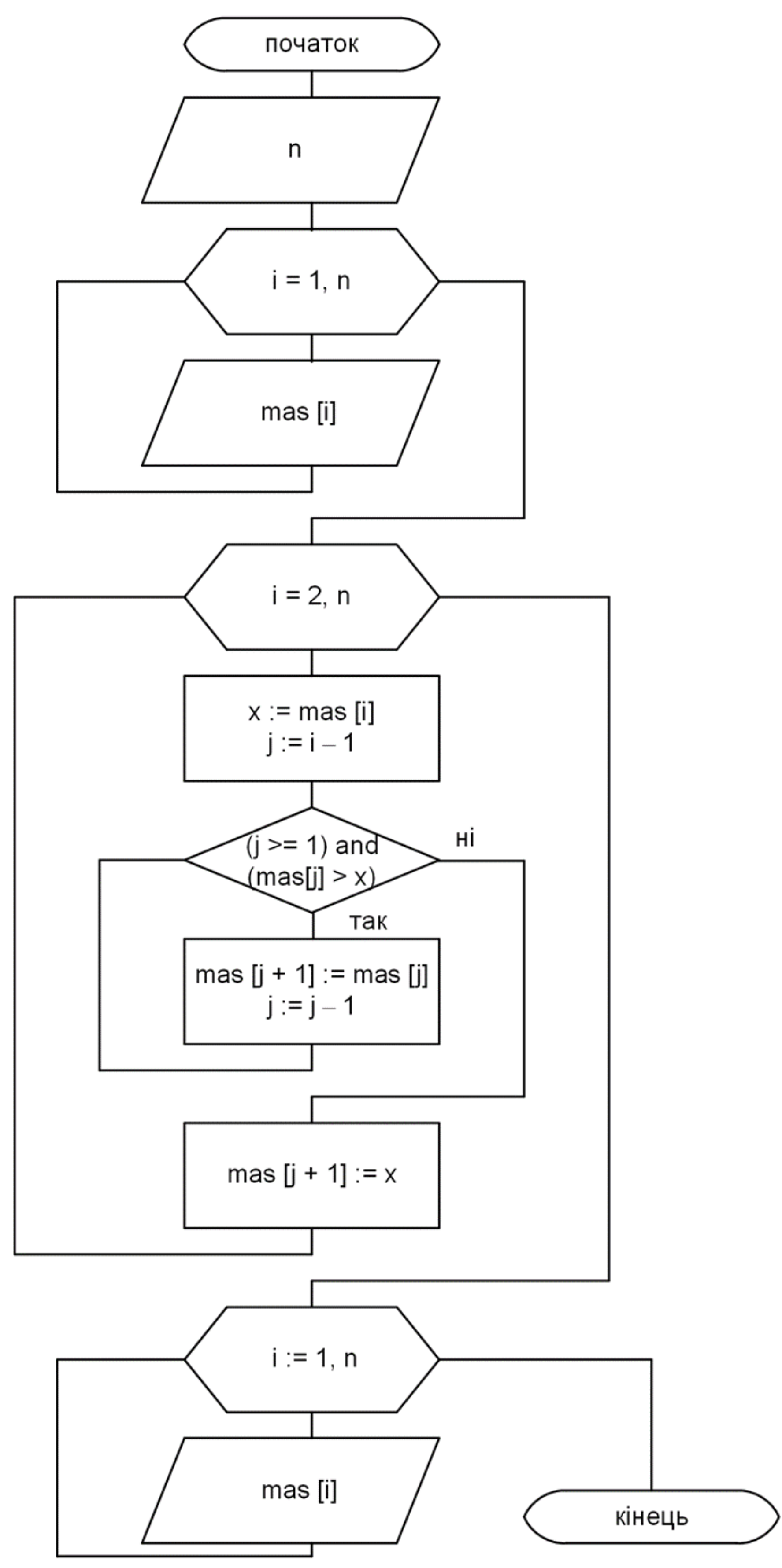


Рис. 2.17. **Алгоритм сортування вставляннями**

Проаналізуймо складність алгоритму сортування вставляннями. Повернімося до процедури сортування вставляннями та зазначмо біля кожного рядка його вартість (кількість операцій) і кількість разів виконання цього рядка (табл. 2.1).

Таблиця 2.1

Часові та вартісні витрати алгоритму сортування вставляннями

№ з/п	Оператори сортування вставляннями	Вартість	Кількість повторів
1	for j ← 2 to length[A]	c_1	n
2	do key ← A[j]	c_2	n – 1
3	додати A[j] до відсортованої частини A[1. . ., j – 1]	0	n – 1
4	I ← j – 1	c_4	n – 1
5	while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6	do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i + 1] ← key	c_8	n – 1

Для кожного j від 2 до n (тут n = length|A| розмір масиву) підрахуймо, скільки разів буде виконано рядок 5, і позначмо цю кількість через t_j. (Зауважмо, що рядки всередині циклу виконують на один раз менше, ніж перевірку, оскільки після останньої перевірки замість обчислення відбувається вихід із циклу).

Рядок вартістю c, повторений m разів, дає внесок $c \times m$ у загальну кількість операцій. (Для обсягу використаної пам'яті цього сказати не можна!). Додавши внески всіх рядків, маємо:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n}(t_j) +$$
$$+ c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1).$$

Час роботи процедури залежить не тільки від n, а й від того, який саме масив розміру n подано на вхід. Для процедури сортування вставляннями найбільш сприятливим є випадок, коли масив уже є відсортованим. Тоді цикл у рядку 5 завершується після першої перевірки (оскільки $A[i] \le key$ за $i = j - 1$), оскільки всі $t_j = 1$ і загальний час є таким:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Отже, у найбільш сприятливому випадку час T(n), необхідний для сортування масиву розміру n, є лінійною функцією від n, тобто має вигляд $T(n) = an + b$ для деяких констант a і b (ці константи визначають вибраними значеннями $c_1, \dots, c_8$).

Якщо масив розташовано у зворотному порядку (від більшого до меншого), час роботи процедури буде максимальним: кожен елемент A[j] доведеться порівняти із усіма елементами A[1],. . . , A[j – 1]. До того ж $t_j = j$. Оскільки

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

маємо, що в гіршому випадку час роботи процедури дорівнює

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5\left(\frac{n(n+1)}{2}-1\right)+c_6\left(\frac{n(n-1)}{2}\right)+$$

$$+c_7\left(\frac{n(n-1)}{2}\right)+c_8(n-1)=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+$$

$$+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n-(c_2+c_4+c_5+c_8).$$

Тепер функція T(n) є квадратичною, тобто має вигляд $T(n) = an^2 + bn + c$ для деяких констант a, b і c (ці константи визначають вибраними значеннями $c_1, \ldots, c_8$).

Отже, бачимо, що часи роботи алгоритму в найгіршому та найкращому випадках значно відрізняються. Зазвичай здебільшого цікавить час роботи в найгіршому випадку, який визначають як максимальний час роботи для входів цього розміру. Знаючи час роботи в найгіршому випадку, можна гарантувати, що виконання алгоритму закінчиться за деякий час, навіть не знаючи, що саме буде на вході.

На практиці «погані» вхідні дані (для яких час роботи близький до максимуму) трапляються часто. Наприклад, для бази даних «поганим» запитом може виявитися пошук відсутнього елемента (досить часта ситуація).

Час роботи в середньому (про який мова йде далі) може бути досить близьким до часу роботи в найгіршому випадку. Нехай, наприклад, сортуймо випадково розташовані числа за допомогою процедури сортування вставляннями. Скільки разів доведеться виконати цикл у рядках 5 – 8? У середньому близько половини елементів масиву A[1…, j − 1] є більшими за A[j], оскільки t_j в середньому можна вважати таким, що дорівнює j / 2, і час T(n) квадратично залежить від n.

У деяких випадках цікавить також середній час роботи алгоритму на входах цієї довжини. Звичайно ця величина залежить від вибраного розподілу ймовірностей, і на практиці реальний розподіл входів може відрізнятися від передбачуваного, яке зазвичай уважають рівномірним, іноді можна досягти рівномірності розподілу, використовуючи датчик випадкових чисел.

2.10.2. Алгоритм сортування Шелла

Алгоритм сортування Шелла є удосконаленим варіантом сортування простими додаваннями. Розгляньмо сортування масиву з таких елементів: 44, 55, 12, 42, 94, 18, 06, 67. За цим методом сортування на першому проході об'єднують усі елементи, віддалені один від одного на чотири позиції. Такий процес має назву 4-сортування (рис. 2.18).

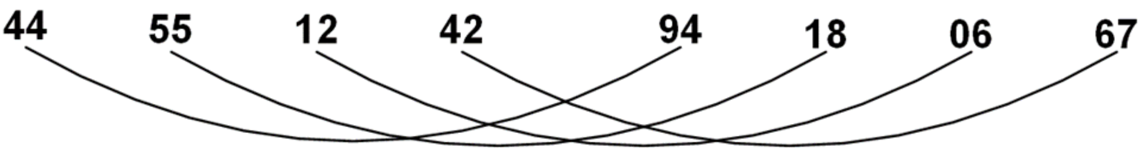


Рис. 2.18. **Сортування Шелла (перший прохід)**

На наступному етапі елементи об'єднують з елементами, розташованими один від одного на відстані двох позицій, і знову сортують. Такий процес має назву 2-сортування (рис. 2.19).

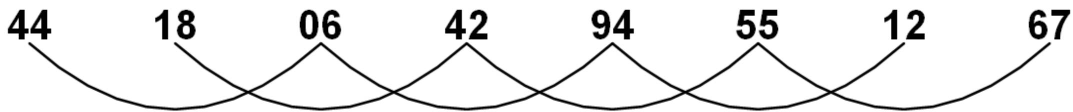


Рис. 2.19. **Сортування Шелла (другий прохід)**

На третьому етапі елементи сортують звичайним сортуванням (рис. 2.20).

Рис. 2.20. **Сортування Шелла**

На кожному етапі сортування Шелла (рис. 2.21) одночасно сортують невелику кількість елементів, тому що елементи вже є достатньо впорядкованими та не потребують значної кількості переставлянь.

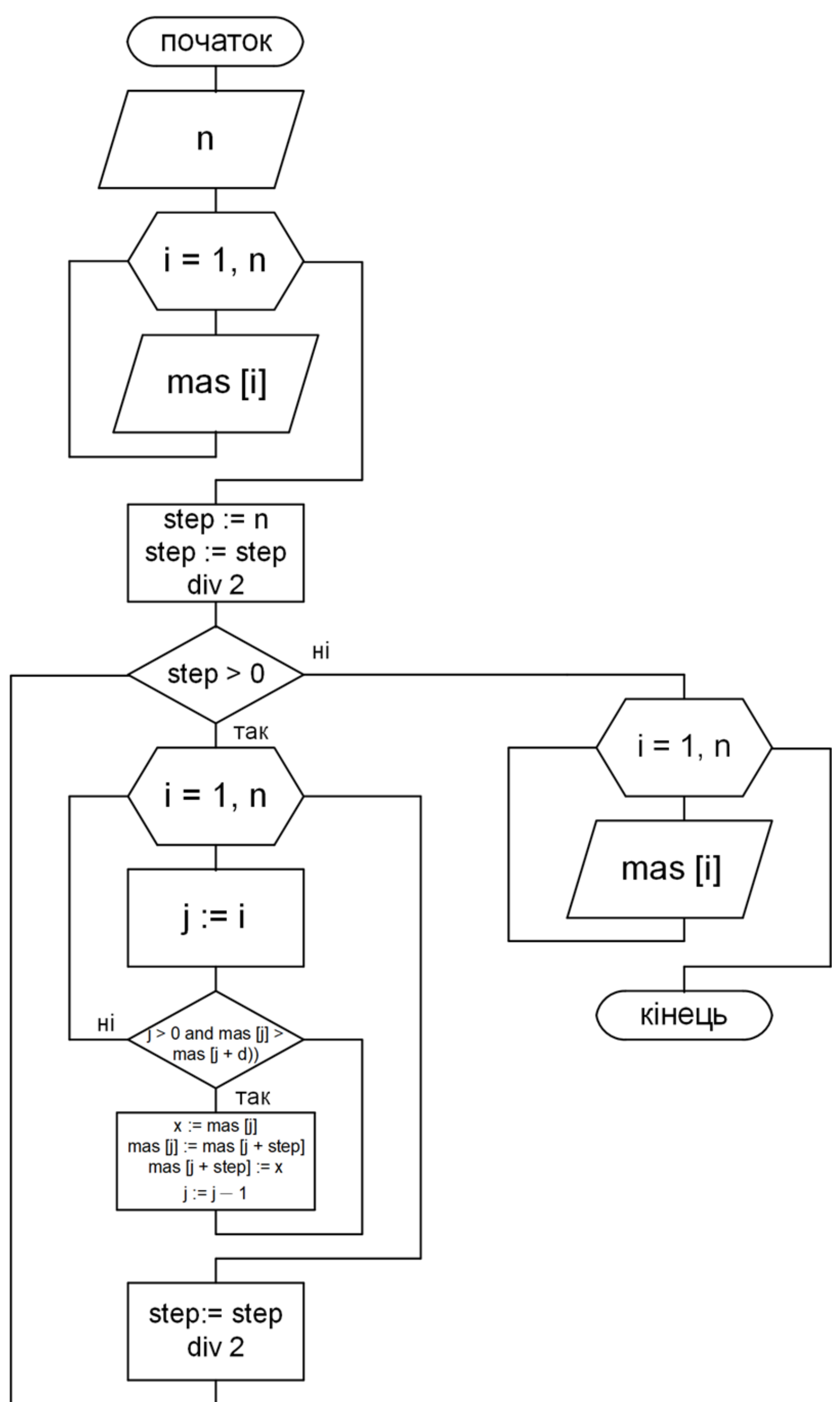


Рис. 2.21. **Алгоритм сортування Шелла**

Наведімо псевдокод алгоритму сортування Шелла:

1 **for** step := length[A] / 2 **to 1**
2 **for** i := step **to** length[A]
3 **for** j := i – step **to** 1

```
4 if A[j]  >  A[j + step]
5 x := A[j]
6 A[j] := A[j + step]
7 A[j + step] :=x
            End
8 j := j – 1
      End
9 i := i + 1
End
10 step := step / 2
```

Реалізація алгоритму **методу сортування простим вибором** є протилежною до реалізації алгоритму сортування простими додаваннями. Так, за сортування простими додаваннями на кожному етапі розглядають лише один черговий елемент початкової послідовності та всі елементи вже відсортованого масиву для визначення місця додавання. Тоді як за сортування простим вибором розглядають усі елементи початкового масиву для визначення елемента з мінімальним значенням, і цей один елемент додають до відсортованого вже масиву. Цей метод реалізують за таким алгоритмом:

1) вибрати елемент із мінімальним значенням;
2) поміняти місцями вибраний елемент та перший елемент a_1;
3) повторити кроки 1 та 2 з n – 1 елементами, які залишилися;
4) повторити кроки 1 та 2 з n – 2 елементами та ін.

Кроки 1 та 2 повторюють доти, доки не залишиться один елемент із найбільшим значенням.

Сутність методу сортування простим вибором полягає у створенні відсортованого масиву шляхом послідовного додавання до нього елементів у визначеному порядку.

Наприклад, для побудови впорядкованого набору, починаючи з лівого кінця масиву, застосовують алгоритм із n послідовних кроків, починаючи із нульового елемента до (n – 1)-го. На i-му кроці обирають найменший із a[i] . . . a[n], який міняють місцем з елементом a[i].

Найпростішим методом сортування є поелементний перебір із послідовним порівнянням кожного елемента зі всіма іншими за певною ознакою (умовою сортування). Такий процес є не дуже ефективним із погляду використання обчислювальних ресурсів, але може бути прийнятним у разі сортування масивів із невеликою кількістю елементів.

2.10.3. Алгоритм бульбашкового сортування

Алгоритм бульбашкового сортування (Bubble sort) є одним із найпростіших алгоритмів сортування, сутність якого полягає в послідовному порівнянні сусідніх елементів масиву та їхньому обміні, якщо вони стоять у неправильному порядку.

Алгоритм працює таким чином:

1. Починаючи із першого елементу масиву, порівнюймо його з наступним.
2. Якщо наступний елемент є меншим за поточний, міняймо їх місцями.
3. Переходьмо до наступної пари елементів та повторюймо кроки 1 – 2.
4. Повторюймо кроки 1 – 3, доки масив не буде повністю відсортованим.

Через свою простоту алгоритм бульбашкового сортування є дуже простим для застосування, але водночас його ефективність зазвичай є нижчою, порівняно з іншими алгоритмами сортування. У найгіршому випадку, якщо масив є повністю не відсортованим, час сортування складається з n^2 порівнянь та n^2 обмінів, де n – кількість елементів початкового масиву. Тому бульбашкове сортування часто використовують для невеликих масивів або як основу для реалізації інших алгоритмів сортування, наприклад, для алгоритму швидкого сортування, який має більшу ефективність для великих масивів (рис. 2.22).

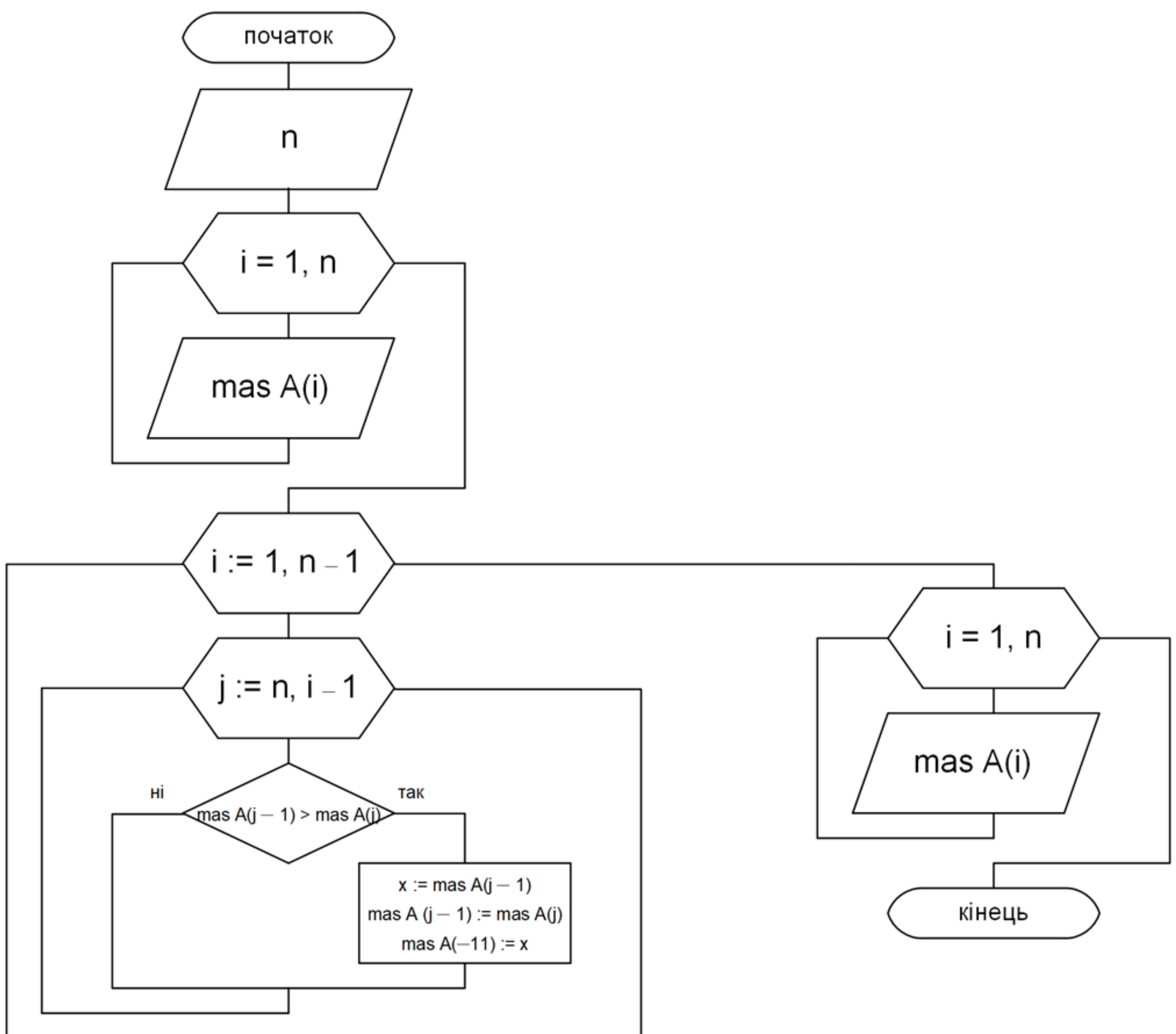


Рис. 2.22. **Алгоритм бульбашкового сортування**

За кожного проходження алгоритму найбільший елемент масиву із внутрішнього циклу ставлять на своє місце в кінець масиву поряд із попереднім «найбільшим елементом», а найменший елемент зміщують на одну позицію вгору (до початку) («спливає» до потрібної позиції, як бульбашка на воді, звідси й походить назва алгоритму).

Псевдокод алгоритму бульбашкового сортування набуває такого вигляду:

1 **for** i := 1 **to** length[A]
2 **for** j := length[A] **to** i
3 **if** A[j – 1] > A [j]

```
4 x := A[j – 1]
5 A[j – 1] := A[j]
6 A[j] := x
        End
7 j := j –1
End
8 i := i + 1
```

Контрольні запитання і завдання для самоперевірки

1. Які типи підходів застосовують під час структурування програм?

2. Що таке «декомпозиція задачі» та для чого її використовують?

3. Що мають на увазі під узагальненою абстракцією функції в програмуванні?

4. Охарактеризуйте поняття «фізична структура даних», наведіть приклади.

5. Охарактеризуйте рівні опису структури даних (фізичний, логічний, функціональний).

6. За якими ознаками здійснюють класифікацію структур даних?

7. Охарактеризуйте загальні операції над структурами даних, наведіть приклади кожної з них.

8. Охарактеризуйте арифметичний тип даних та операції над ним.

9. Які операції здійснюють із покажчиками?

10. Які типи даних належать до класу статичних структур?

11. Які типи даних належать до класу напівстатичних структур?

12. Охарактеризуйте масив на трьох рівнях опису структури.

13. Які властивості масиву ви знаєте?

14. Чим розріджені масиви відрізняються від звичайних? У чому їхні переваги й недоліки?

15. Які типи розріджених масивів ви знаєте?

16. Які операції, зокрема, специфічні застосовують під час роботи із множинами?

17. Які операції використовують під час роботи з бітовими типами?

18. Які інтегровані типи структур ви знаєте?

19. Які операції використовують під час роботи з таблицями?

20. Дайте визначення стека й опишіть принципи роботи із ним.

21. Які операції використовують під час роботи зі стеками?

22. Дайте визначення черги й опишіть принципи роботи з нею.

23. Дайте визначення дека й опишіть принципи роботи з ним.

24. Порівняйте між собою стек, чергу та дек. Наведіть приклади їхнього використання.

25. Дайте визначення лінійних списків, назвіть їхні типи.

26. Наведіть структури однозв'язного та двозв'язного списків та опишіть принципи роботи з ними.

27. У чому полягає специфіка побудови кільцевих списків?

28. Що таке «мультисписки» та які переваги вони надають розробнику?

29. Назвіть ключові властивості рядків (String), які операції над ними виконують?

30. Охарактеризуйте зв'язне подання даних у пам'яті.

31. Які нелінійні структури даних ви знаєте? Наведіть приклади їхнього використання.

32. Які властивості є притаманними графу як складній нелінійній багатозв'язній динамічній структурі?

33. Опишіть алгоритм сортування вставляннями. Які недоліки й переваги йому є притаманними?

34. Опишіть алгоритм сортування Шелла.

35. У чому полягає сутність бульбашкового алгоритму сортування?

36. Опишіть сутність методу сортування простим вибором.

Лабораторна робота 2
Програмування елементарних структур даних

Мета. Ознайомитися з алгоритмами роботи з елементарними структурами даних, набути навичок в організації обчислювальних процесів із застосуванням елементарних структур даних.

Рекомендації з підготовки до виконання. Для успішного виконання лабораторної роботи студент має знати мету, порядок виконання

роботи та загальні теоретичні положення; уміти організовувати обчислювальний процес опрацювання елементарних структур різних типів.

Завдання до виконання. Проаналізуйте формулювання завдання за індивідуальним варіантом (додаток Б табл. Б.1), номер варіанта відповідає номеру студента в журналі; формалізуйте обчислювальний процес розв’язання математичної задачі та побудуйте його блок-схему.

Загальні теоретичні положення

Алгоритмізація роботи із одновимірними масивами. Під час опрацювання значних обсягів даних для організації ефективної роботи використання лише базових типів змінних є недостатнім. У мовах програмування є структура, яка дає можливість зручно розташовувати в пам’яті значний обсяг інформації, і називають її **масив**.

Масив становить набір змінних, які мають одне базове ім’я, але відрізняються одна від одної числовою ознакою (індексом). Елементи масиву мають ті самі типи даних, як ординарні змінні. Ім’я масиву вказують одразу за ключовим словом, яке визначає тип його елементів. Квадратні дужки ([ ]) після імені вказують на те, що це масив, а число у дужках – кількість елементів масиву. Відлік елементів масиву починають із 0, а не з 1. Окремий елемент масиву визначають за допомогою його номера, який називають **індексом**.

Для того щоб звернутися до елемента масиву, використовують константи, змінні та вирази цілого типу. Елементи масиву розміщують у пам’яті послідовно один за одним. У пам’яті кожному елементові відведено стільки місця, скільки й простій змінній такого самого типу.

Елементам оголошеного масиву можна надати початкові значення. Водночас список значень потрібно подати у фігурних дужках, а самі значення – відділити комами: int a[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}.

У процесі компіляції під масив відведемо пам’ять в обсягу, який дорівнює доданку кількості елементів на розмір пам’яті для одного елемента.

Алгоритмізація роботи із двовимірними масивами. Двовимірний масив є масивом, елементами якого є масиви. Розмір двовимірного масиву задають після імені масиву у квадратних дужках, наприклад: int a[2] [2].

Уводити числові значення елементів масиву можна так:

безпосередньо із клавіатури;

записом числових значень після опису масиву: Int a[2] [2] = 1, 2, 4, 8.

Запис масиву із двох рядків та чотирьох стовпчиків має такий вигляд:

int a [2] [4] = 11, 12, 13, 14,
21, 22, 23, 24.

Уведення елементів одновимірного масиву. За фізичного подання масив є набором елементів, які мають один тип даних і розташовані в пам'яті один за одним. Важливою ознакою масиву є те, що доступ до його елементів відбувається за номером елемента, тобто його індексом. Кількість індексів, якої достатньо для доступу до елемента масиву, визначає його розмірність. Наприклад, в одновимірному масиві-векторі – достатньо лише одного індексу для однозначної ідентифікації певного елемента.

Виокремлюють динамічні та фіксовані масиви. Якщо кількість елементів масиву може змінюватися під час виконання програми, то це *динамічний*, інакше – *фіксований* масив.

Операція введення елементів одновимірного масиву полягає у зчитуванні та записі їх у пам'ять комп'ютера один за одним. Отже, маємо операцію, яку повторюють для кожного елемента, а тому її може бути організовано у вигляді такого циклу:

початкове значення лічильника циклу *i*, за який у разі фіксованого масиву можна взяти індекс елемента, дорівнює 1 (за синтаксису деяких мов програмування 0), кінцеве значення дорівнює *n* – кількості елементів масиву;

крок зміни лічильника дорівнює 1;

умовою завершення циклу є перевищення лічильником свого кінцевого значення ($i > n$).

Тілом такого циклу є операція введення елемента. Для виведення елементів масиву або реалізації інших операцій над масивом організовують такий самий цикл з іншими операціями тіла циклу.

Алгоритмізація процесу сортування масивів. Під **сортуванням**, зазвичай, розуміють процес переставляння об'єктів цієї множини для досягнення визначеного порядку. Мета процесу сортування полягає в полегшенні визначення елементів у наборі даних, який сортують. У такому

розумінні елементи, які підлягають сортуванню, наявні майже в усіх задачах.

Велика кількість фундаментальних способів, які застосовують під час побудови алгоритмів, ґрунтуються на процедурі сортування. Розгляньмо детальніше сортування масивів [3; 4].

Припустімо маємо набір елементів: $a_1, a_2, \ldots, a_n$. Процес сортування передбачає виконання переставляння елементів у такому порядку: $a_{k1}, a_{k2}, \ldots, a_{kn}$.

Тобто за заданої функції впорядкування f справедливим є таке відношення: $f(a_{k1}) \leq f(a_{k2}) \leq \ldots \leq f(a_{kn})$.

Зазвичай, визначення функції впорядкування не потребує додаткових обчислень, її значення міститься в інформаційному полі елемента масиву, значення якого називають **ключем елемента**.

Під час вибору методу сортування найперше слід зважати на обсяг пам'яті, потрібної для його реалізації. Зазвичай за критерії класифікації алгоритмів сортування використовують обсяг необхідної пам'яті та кількість часу на виконання алгоритму.

Ефективність алгоритму сортування можна оцінити за кількістю потрібних порівнянь C і кількістю пересилань елементів M. Ці значення розраховують за певними залежностями від кількості елементів n, які підлягають сортуванню. Більш ефективні алгоритми сортування потребують кількості порівнянь порядку $n \times \log n$, тоді як прості методи сортування – порядку n^2.

Порядок виконання роботи

Розгляньмо процес побудови алгоритмів на конкретних прикладах.

Завдання 2.1. Наведіть блок-схемний опис алгоритму підрахунку добутку ненульових елементів одновимірного масиву.

Виконання завдання 2.1. Блок-схему алгоритму підрахунку добутку ненульових елементів одновимірного масиву показано на рис. 2.23.

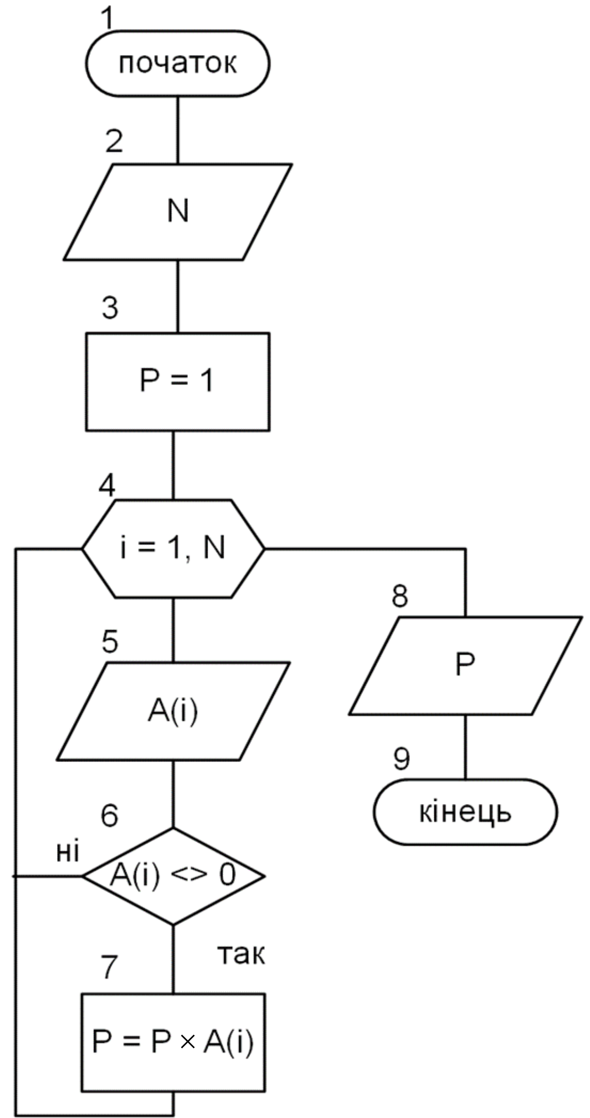


Рис. 2.23. **Блок-схема алгоритму підрахунку добутку ненульових елементів одновимірного масиву**

Розроблення цього алгоритму передбачає:

уведення кількості елементів масиву *n* (блок 2);

задавання змінній, яка буде містити результат добутку, початкового значення, яке б не змінювало кінцевого результату (блок 3);

організацію циклу за змінною *i*, яка відповідає індексу елемента масиву (блок 4).

Тіло масиву становлять такі операції:

уведення елементів масиву (блок 5);

перевірка елемента на рівність нулю (блок 6);

зміна значення добутку (блок 7), якщо умову блоку 6 виконано.

Результат підрахунку добутку виводять після завершення циклу (блок 8).

Завдання 2.2. Розробіть блок-схему алгоритму визначення максимального елемента одновимірного масиву.

Виконання завдання 2.2. Блок-схему алгоритму пошуку максимального елемента одновимірного масиву показано на рис. 2.24.

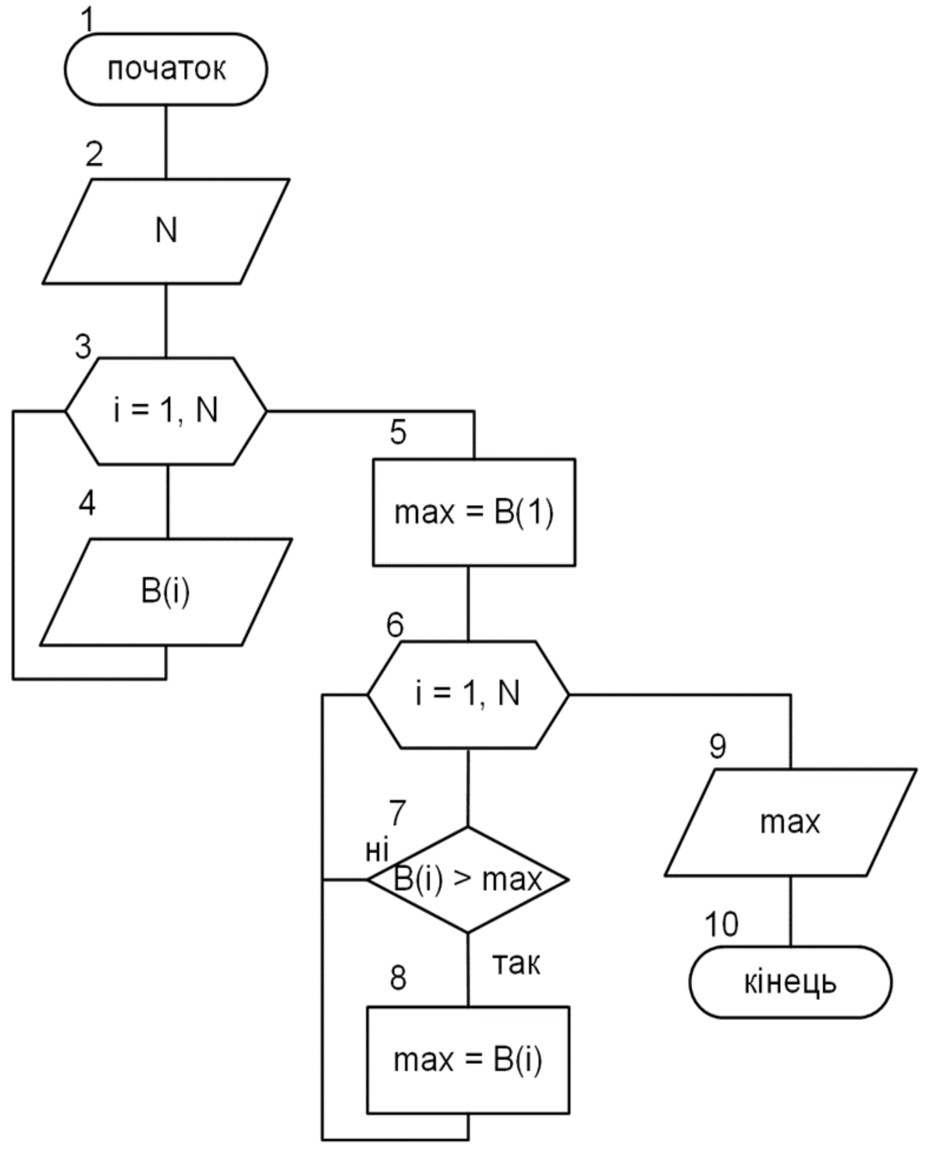


Рис. 2.24. **Блок-схема алгоритму визначення максимального елемента одновимірного масиву**

Організація обчислювального процесу в цьому завданні передбачає побудову двох окремих циклів: один цикл для введення елементів

масиву, другий – для визначення елемента з максимальним значенням.

Результат, тобто елемент із максимальним значенням, буде міститися у змінній *max*. Перш ніж організовувати цикл для його визначення, потрібно надати змінній *max* певне значення, із яким будуть порівнювати кожен елемент масиву. Зазвичай таким значенням є значення першого елемента масиву (блок 5).

Зауваження. Можлива інша організація цього обчислювального процесу, яка передбачає побудову одного циклу, тіло якого містить одночасно й операцію введення елементу масиву, й операцію його порівняння зі змінною *max*. У цьому разі перед організацією циклу слід в окремому блоці надати змінній *max* значення першого елемента масиву. Таке спрощення робить алгоритм ефективнішим із погляду швидкодії.

Тіло цього циклу становлять такі операції (див. рис. 2.24):

перевірка, чи є поточний елемент більшим за змінну *max* (блок 7);

надання змінній *max* нового найбільшого значення в разі виконання умови блоку 7 (блок 8).

Поза циклом виводьмо остаточне значення змінної *max*, яке містить максимальне значення елементів цього масиву (блок 9).

Завдання 2.3. Розробіть блок-схему алгоритму підрахунку суми від'ємних елементів одновимірного масиву.

Виконання завдання 2.3. Блок-схему алгоритму підрахунку суми від'ємних елементів одновимірного масиву показано на рис. 2.25. Ця блок-схема є аналогічною блок-схемі алгоритму підрахунку добутку ненульових елементів масиву (див. рис. 2.23). Блок-схема алгоритму підрахунку суми від'ємних елементів потребує змінення значення, яке надають змінній результату (блок 3), а також формули тіла циклу (блок 7).

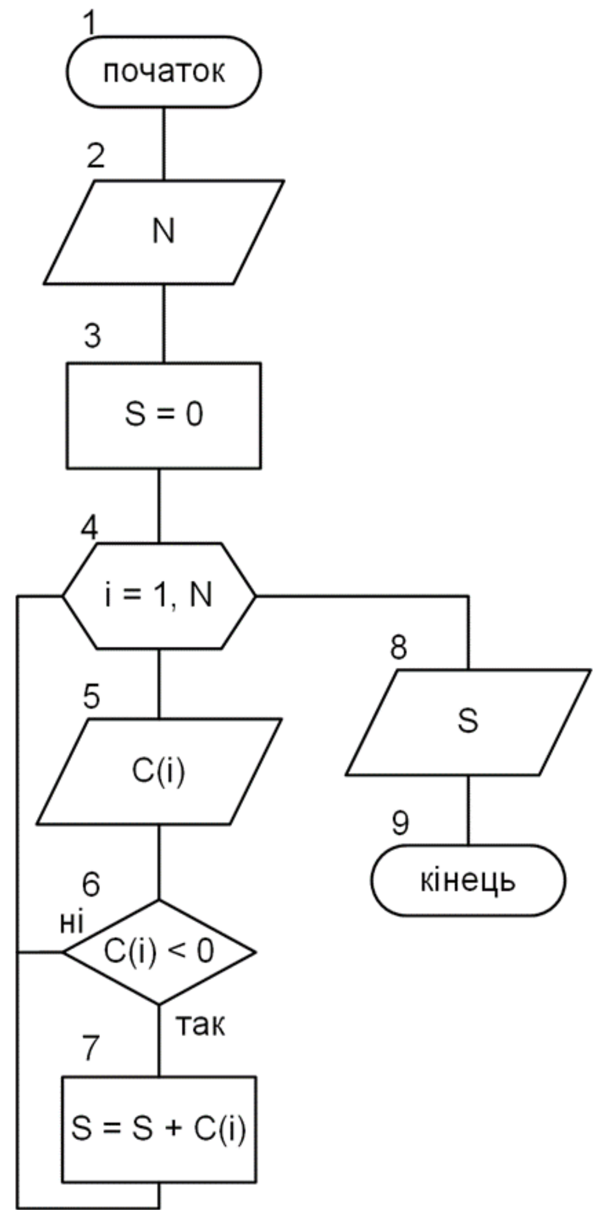


Рис. 2.25. **Блок-схема алгоритму підрахунку суми від'ємних елементів одновимірного масиву**

Завдання 2.4. Розробіть блок-схему алгоритму підрахунку кількості нульових елементів одновимірного масиву.

Виконання завдання 2.4. Блок-схему алгоритму підрахунку кількості нульових елементів масиву показано на рис. 2.26.

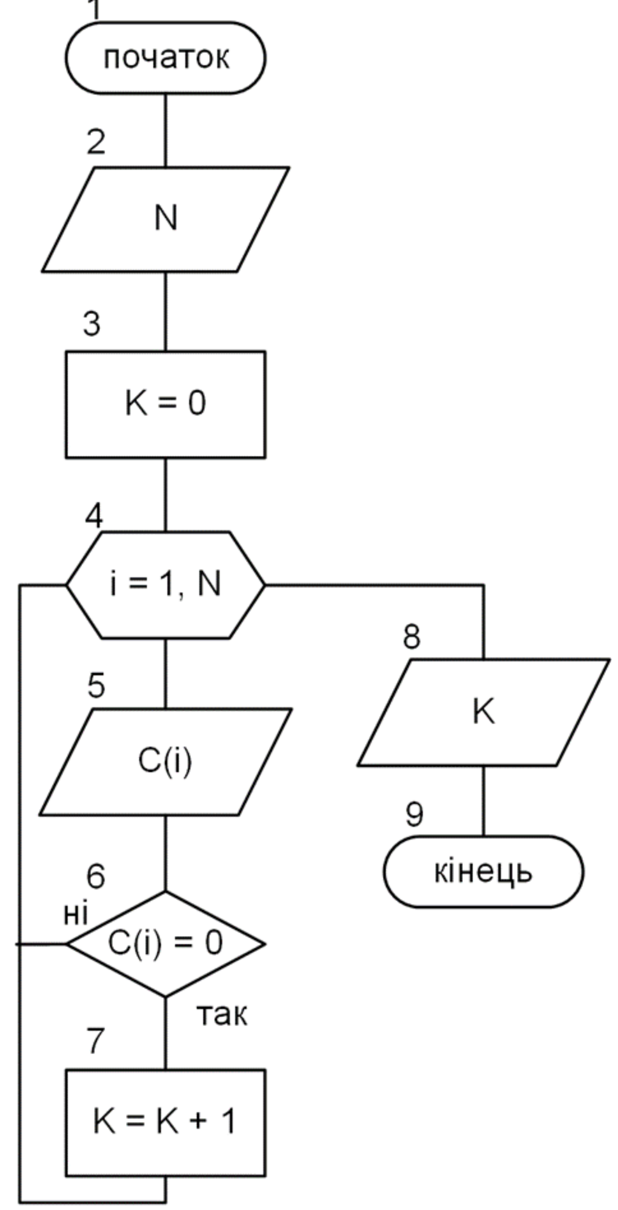


Рис. 2.26. **Блок-схема алгоритму підрахунку кількості нульових елементів одновимірного масиву**

Контрольні запитання до лабораторної роботи 2

1. Дайте визначення масиву.
2. У чому полягає різниця між розмірністю та розміром масиву?
3. У якому блоці доцільно виконувати введення числових значень елементів масиву?
4. Охарактеризуйте поняття «одновимірний масив».

5. Наведіть приклад одновимірного масиву та проведіть паралель із математикою. Як у математиці називають одновимірний масив?

6. Як називають масив, у якому кількість елементів можна змінювати під час виконання програми?

3. Двійкові дерева пошуку

3.1. Поняття дерева. Елементи дерев

Дерево – це абстрактна структура даних, яка складається з вузлів та ребер і має логічне подання, подібне до реального дерева з гілками та листям. Кожен вузол дерева містить деяку інформацію, яку потрібно зберігати, і може мати посилання на декілька нащадків (дітей), які також є вузлами дерева. Вузли без нащадків називають *листям*.

Дерева зазвичай використовують для зберігання й організації даних у структуру з відносно швидким доступом до них. Їх широко застосовують у комп’ютерних науках, наприклад, у базах даних, компіляторах, операційних системах, мережах та ін.

Найпростіша форма дерева – це *бінарне дерево*, у якому кожен вузол має не більше за двох нащадків. Інші форми дерева можуть мати більше за двох нащадків, наприклад, n-арне дерево, у якому кожен вузол має не більше ніж n нащадків.

Дерева можуть бути використаними для реалізації різноманітних алгоритмів, наприклад, пошук вузла, додавання та вилучення вузла, сортування, обходу дерева тощо. Також дерева використовують в алгоритмах шифрування, як-от RSA, де їх використовують для зберігання й управління ключами шифрування.

Розгляньмо деякі поняття, які використовують під час роботи із деревами. Вузол X називають *предком*, або *батьком*, для вузлів Y і Z, якщо він є розташованим на верхньому рівні ієрархії дерева щодо вузлів Y і Z. Тоді вузли Y і Z є *нащадками*, або *синами*, а між собою такі вузли є *братами*. Слід зважати, що лівий син є старшим, а правий – молодшим. Кількість піддерев, які виходять із певного вузла (вершини) називають *мірою* цього *вузла* (цієї *вершини*). Структурну схему дерева показано на рис. 3.1.

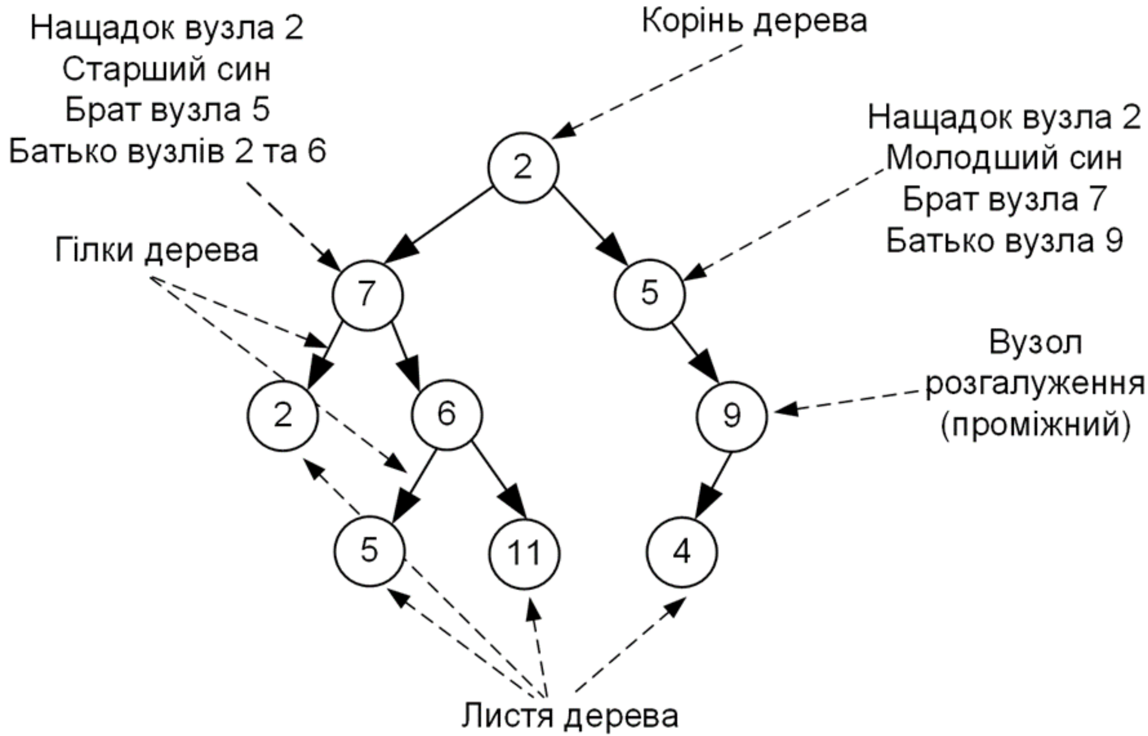


Рис. 3.1. **Структура дерева та його елементи**

Бінарне дерево пошуку – це бінарне дерево, у якому значення ключа лівого вузла-нащадка є меншим або дорівнює ключу батьківського вузла, а ключ правого вузла-нащадка є більшим або дорівнює ключу батьківського вузла [7]. *Ключ* – це значення, для збереження якого створено цей вузол, і яке записано у його інформаційному полі.

Якщо зі структури дерева прибрати корінь та гілки, які з'єднують корінь із вузлами першого рівня ієрархії, то маємо множину незв'язаних дерев, яку позначають терміном «ліс».

Для графічного зображення дерев використовують кілька способів:

на основі діаграм Венна, які застосовують для зображення піддерев як підмножин вузлів дерева;

із застосуванням дужок, які вкладають одна в одну;

подібний до складання змісту книг: кожному вузлові надають номер, який має бути меншим за номери, надані кореневим вузлам піддерев, пов'язаних із цим вузлом.

Є кілька способів подання дерев. Розгляньмо основні з них.

Масив. Дерево може бути подано у вигляді масиву, де індекс елемента масиву відповідає номеру вузла дерева, а значення відповідає

даним, які зберігають у цьому вузлі. У бінарному дереві лівий і правий нащадки вузла можуть бути визначеними з використанням формул: лівий нащадок = (2 × номер вузла) + 1, правий нащадок = (2 × номер вузла) + 2. У n-арному дереві формула буде залежати від кількості нащадків.

Посилання. Наприклад, у бінарному дереві кожен вузол може мати три поля: інформаційне (для збереження даних), поле посилання на правого нащадка, поле посилання на лівого нащадка. Цей підхід може займати більше пам'яті, порівняно із масивним підходом, але надає більшу гнучкість під час додавання та вилучення вузлів із дерева.

Потоки. Дерево може бути подано у формі послідовності даних (потоку) зі збережених вузлів. У такому разі кожен вузол дерева може мати посилання на нащадків та батьківський вузол. Цей підхід є корисним для опрацювання дерева за допомогою потокових алгоритмів, як-от алгоритми потокового обходу дерева.

Матриця суміжності. Для деяких видів дерев можна використовувати матрицю суміжності, де значення елементів матриці відповідають наявності ребер між двома вузлами дерева. Наприклад, у бінарному дереві матриця містить 1 на перетині рядка та стовпця, якщо між ними є ребро, та 0 в іншому випадку.

Виокремлюють *повні дерева*, які містять максимально можливу кількість вузлів на кожному рівні ієрархії, крім нижнього. Таке дерево є найкоротшим варіантом дерева, яке містить визначену кількість вузлів. Так, наприклад, якщо розглянути повне бінарне дерево, то кожен його вузол, крім листя, буде мати два нащадки.

Кожен елемент n-арного дерева містить стільки покажчиків, скільки гілок із нього виходить. Фізичне подання таких дерев потребує значних витрат пам'яті та різноманітності початкових елементів, що ускладнює алгоритми опрацювання таких структур даних. Із метою економії пам'яті та спрощення алгоритмів n-арні дерева подають бінарними деревами, усі вузли яких є однотипними елементами й мають інформаційне поле та два покажчики. Алгоритми роботи із бінарними деревами є добре вивченими та мають ефективну трудомісткість.

Алгоритм перетворення n-арного дерева на бінарне містить такі кроки:

залишити в кожному вузлі лише ребро до старшого нащадка (сина);

з'єднати горизонтальними ребрами всіх братів (нащадків одного рівня) одного батька;

перебудувати дерево за правилом: лівий син – вузол, розташований під цим; правий син – вузол, розташований праворуч від цього (тобто на одному рівні);

розгорнути дерево так, щоб усі вертикальні ребра відображали лівих синів, а горизонтальні – правих.

Результатом такого перетворення є ліве піддерево, зв'язане з коренем. За такого перетворення покажчик на правого сусіда кожного вузла бінарного дерева буде вказувати на вузол того самого рівня або мати значення NULL. Покажчик на лівого сусіда буде вказувати на вузол нижнього рівня ієрархії або мати значення NULL.

Описаний метод подання довільних упорядкованих n-арних дерев як бінарних можна узагальнити для довільного впорядкованого лісу.

Алгоритм перетворення впорядкованого лісу на бінарне дерево містить такі кроки:

з'єднати горизонтальними зв'язками корені всіх піддерев лісу;

трансформувати визначене довільне дерево за раніше розглянутим алгоритмом.

Результатом перетворення впорядкованого лісу є повне бінарне дерево з лівим та правим піддеревами.

У пам'яті дерева доцільно подавати за допомогою зв'язних списків і масивів. Зв'язне подання дерев за структурою подібно до логічного подання, саме тому його використовують найчастіше. Зв'язне зберігання передбачає задавання зв'язків між батьківським вузлом і вузлами-синами. Елемент бінарного дерева має два покажчики та є структурою із трьома полями: left – це поле, яке містить покажчик на ліве піддерево; right – це поле, яке містить покажчик на праве піддерево; inf – це поле, яке містить інформацію, для зберігання якої створено цю структуру та містить дані певного типу.

3.2. Основні операції з деревами

Наведімо основні операції, які можна виконувати над деревами:

додавання нового вузла;

вилучення вузла (піддерева);

пошук вузла із заданим ключем;

обхід дерева в певному порядку (низхідний, висхідний, змішаний).

Пошук вузла (вершини) у бінарному дереві здійснюють за ключем, тобто за значенням інформаційного поля. Розгляньмо реалізацію цього процесу. Припустімо, побудовано бінарне дерево та потрібно визначити вузол (вершину) із ключем X, алгоритм пошуку за ключем передбачає виконання таких кроків:

1) порівняймо ключ, який міститься в корені дерева зі значенням X ключ: якщо значення збігаються, то пошук закінчено, а покажчик результату вказує на корінь, інакше переходьмо до кроку 2;

2) якщо значення X є меншим за значення ключа щойно розглянутого вузла, то X порівнюють зі значенням ключа вузла, який міститься на нижньому рівні ліворуч, інакше – зі значенням ключа вузла, який міститься на нижньому рівні праворуч.

Роботу алгоритму завершують за виконання однієї з умов:

визначено вузол, ключ якого містить значення, яке дорівнює значенню X;

у дереві немає вузла, до якої слід перейти для реалізації наступного кроку пошуку (тобто дерево не містить вузла із завданим значенням ключа).

За виконання першої умови повертається покажчик на визначений вузол, за виконання другої – покажчик на вузол, на якому зупинено пошук (це зручно для побудови дерева, тому що саме після цього вузла можна додавати вузол із ключем X).

Для додавання нового вузла до бінарного дерева найперше слід визначити вузол, до якого можна приєднати новий. Визначення такого вузла має забезпечувати збереження впорядкованості ключів вузлів дерева. Алгоритм пошуку вузла, після якого слід додати новий вузол є подібним до алгоритму пошуку вузла за заданим ключем.

Розв'язання багатьох задач, у яких використовують дерева, передбачають здійснення систематичного перегляду всіх вузлів дерева в певному порядку, який називають *обходом*, або *проходженням*, дерева.

Алгоритм обходу бінарного дерева *за низхідного способу* має такий вигляд:

1) вибрати корінь дерева за початковий вузол, перейти до кроку 2;

2) опрацювати вузол, згідно з умовами завдання, перейти до пункту 3;

3а) якщо вузол має два ребра, за новий вузол вибрати лівий вузол, а правий занести в стек; перейти до пункту 2;

3б) якщо вибраний вузол є кінцевим (листям дерева), то, якщо стек не є порожнім, вибрати як новий вузол зі стека й перейти до пункту 2; якщо ж стек є порожнім, обхід дерева завершено, перейти до пункту 4;

3в) якщо вибраний вузол має одне ребро, то за новий вибрати вузол, на який указує це ребро, перейти до пункту 2;

4) завершити роботу алгоритму.

Використання рекурсії значно спрощує реалізацію цього алгоритму:

1) опрацювання кореневого вузла;

2) низхідний обхід лівого піддерева;

3) низхідний обхід правого піддерева.

Алгоритм змішаного обходу має такий вигляд:

1) спуститися за лівим ребром із занесенням вузлів до стека;

2) у разі порожнього стека перейти до кроку 5;

3) вибрати вузол зі стека й опрацювати його;

4) якщо вузол має правого сина, перейти до нього; перейти до кроку 1;

5) завершити роботу алгоритму.

Змішаний обхід за використання рекурсії має такий вигляд:

1) змішаний обхід лівого піддерева;

2) опрацювання кореневого вузла;

3) змішаний обхід правого піддерева.

Складність *висхідного обходу* полягає в тому, що кожний вузол заносять у стек двічі: уперше під час обходження лівого піддерева, удруге під час обходження правого піддерева. Отже, в алгоритмі є потреба виокремлювати два види стекових записів: перший означає, що відбувається обходження лівого піддерева; другий – обходження правого, тому в стек, крім покажчика, слід заносити певну ознаку зі значеннями: 1 або 2, відповідно до номера запису.

Алгоритм реалізації висхідного обходу описують таким чином:

1) спуститися за лівим ребром із записом вузла в стек (1-й вид запису);

2) якщо стек є порожнім, перейти до кроку 5;

3) вибрати вузол зі стека:

3а) якщо ознака стекового запису елемента 1, то повернути вузол до стека, а ознаку змінити на 2; перейти до правого сина цього вузла та перейти до кроку 1;

3б) якщо ознака стекового запису елемента 2, перейти до кроку 4;

4) опрацювати інформаційне поле вузла, згідно з умовами завдання, та перейти до кроку 2;

5) кінець алгоритму.

Рекурсивний висхідний обхід за використання рекурсії має такий вигляд:

1) висхідний обхід лівого піддерева;

2) висхідний обхід правого піддерева;

3) опрацювання кореневого вузла.

Якщо в наведених алгоритмах змінити місцями поля покажчиків на ліве та праве піддерева, то маємо зворотні процедури низхідного, змішаного та висхідного обходів.

3.3. Бінарні дерева пошуку

Бінарне дерево пошуку (Binary Search Tree, BST) – це структура даних, що є деревом, у якому кожен вузол містить ключ і може мати максимум двох синів: лівого та правого. Ключі вузлів дерева впорядковано таким чином, що для кожного вузла всі ключі лівого сина є меншими за його ключ, а всі ключі правого сина – більшими. Ця властивість дозволяє здійснювати ефективний пошук, уставляння та вилучення елементів у дереві.

Бінарне дерево пошуку має такі властивості:

лівий син вузла має менший ключ за його власний, а правий син – більший;

дерево не містить дублікатів ключів;

дерево може бути порожнім.

Бінарні дерева дозволяють виконувати операції пошуку, уставляння та вилучення елементів за час O(log n), де n – кількість елементів дерева. Однак у найгіршому випадку, якщо дерево має нелінійну структуру, час виконання може досягати O(n), що робить алгоритм менш ефективним, порівняно з іншими алгоритмами сортування та пошуку.

Бінарні дерева застосовують у багатьох сферах, де потрібен швидкий пошук та вставляння елементів. Наприклад, бінарні дерева використовують в базах даних для швидкого пошуку записів за ключем, у компіляторах – для зберігання символів та ключових слів мов програмування, а також в операційних системах – для зберігання інформації про процеси та ресурси. Бінарні дерева пошуку є структурами даних, які підтримують операції пошуку елемента за ключем, пошуку елемента з мінімальним

та максимальним значенням ключа, пошуку батька та сина, уставляння нового елемента та вилучення.

Під час роботи з бінарними деревами пошуку виокремлюють дерева з додатковою ознакою кольору вузлів, так звані червоно-чорні дерева.

Червоно-чорне дерево (Red-Black Tree) – це бінарне дерево пошуку, у якому кожен вузол має додаткове поле – колір, який може бути червоним або чорним. Це дерево є самобалансувальним, тобто таким, що забезпечує балансування дерева (кількість вузлів у кожній гілці дерева є приблизно однаковою) під час уставляння та вилучення елементів, що дозволяє забезпечити оптимальний час виконання операцій.

Червоно-чорні дерева мають такі властивості:

1) кожен вузол має кольорове позначення (червоний або чорний);
2) корінь завжди має чорний колір;
3) усі листи дерева є чорними вузлами та не мають значення ключів;
4) кожен червоний вузол має двох чорних синів.

Усі шляхи від кореня до листя мають однакову кількість чорних вузлів (ця властивість гарантує балансування дерева).

Додавання та вилучення вузлів у червоно-чорному дереві може призвести до порушення властивостей дерева. Для забезпечення балансування дерева в таких випадках використовують різні операції перебудови дерева, як-от ротації та зміна кольорів вузлів.

Червоно-чорні дерева використовують у багатьох алгоритмах, у яких потрібно забезпечити оптимальний час виконання операцій із деревом, наприклад, у базах даних, операційних системах, компіляторах та інших системах.

3.3.1. Структура бінарного дерева

Бінарне дерево може бути подано зв’язаною структурою даних. За такого подання кожен вузол дерева є об’єктом, який, крім полів ключа *key* та супутніх даних, містить:

поле *left*, яке вказує на сина ліворуч;

поле *right*, яке вказує на сина праворуч;

поле *p*, яке вказує на батьківський вузол.

Якщо сина або батьківського вузла немає, відповідне поле буде містити значення NULL. Якщо покажчик *p* вузла має значення NULL, то такий вузол є *коренем дерева*. У бінарному дереві пошуку ключі зберігають

таким чином, щоб завжди було виконано **властивості бінарного дерева пошуку:**

якщо x є вузлом бінарного дерева пошуку, а вузол y міститься в лівому піддереві вузла x, то key[y] ≤ key[x];

якщо вузол y міститься в правому піддереві вузла x, то key[x] ≤ ≤ key[y] (рис. 3.2).

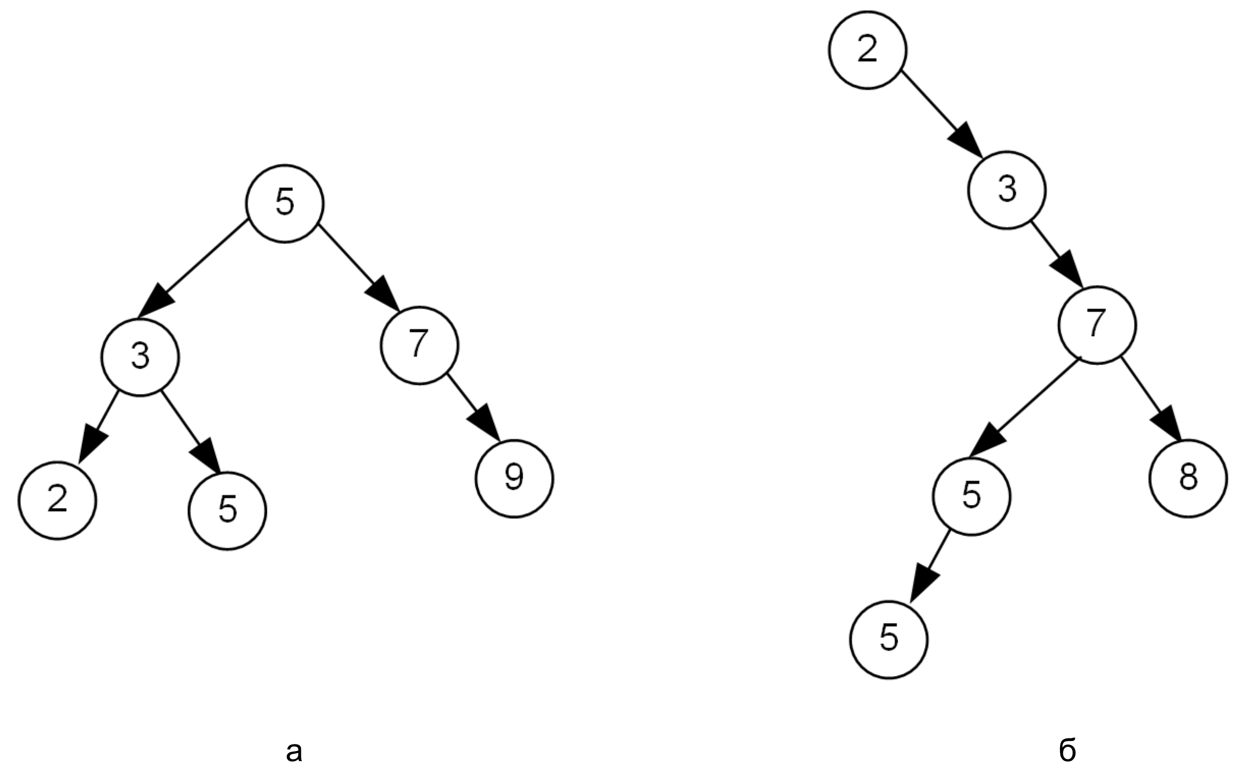


Рис. 3.2. **Приклади бінарних дерев пошуку**

На рис. 3.2а показано приклад бінарного дерева пошуку, ключ кореня якого дорівнює 5. Ключі 2, 3 та 5, які не перевищують значення ключа кореня, розташовано в лівому піддереві, а ключі 7 та 8, які є не меншими за ключ 5, – у правому піддереві. Таку саме властивість виконано для кожного вузла дерева. На рис. 3.2б показано дерево з такими самими вузлами, яке має ту саму властивість, проте є менш ефективним у роботі, тому що його висота становить 4, на відміну від дерева на рис. 3.2а, висота якого становить лише 2.

Описана властивість бінарного дерева пошуку дозволяє вивести всі ключі, які зберігають у дереві, у відсортованому порядку. Для цього використовують рекурсивний алгоритм, який має назву симетричного обходу дерева.

Симетричний обхід дерева (inorder tree walk) – це алгоритм обходу бінарного дерева пошуку, за якого відбувається рекурсивний обхід дерева в такому порядку:

1) обходьмо лівого сина поточного вузла, викликавши рекурсивно симетричний обхід для лівого піддерева;

2) опрацьовуймо поточний вузол, згідно з умовами завдання;

3) обходьмо правого сина поточного вузла, викликавши рекурсивно симетричний обхід для правого піддерева.

Під час симетричного обходу дерева, вузли опрацьовують у порядку зростання їхніх ключів, що забезпечує сортування елементів у дереві.

Симетричний обхід дерева має ряд важливих властивостей:

гарантує відсутність дублікатів ключів у вузлах;

забезпечує сортування елементів у дереві;

дозволяє відтворювати дерево з допомогою послідовності його елементів.

Симетричний обхід дерева застосовують у багатьох алгоритмах та програмах, де потрібно сортування елементів за ключами, а також для визначення певних властивостей дерева, як-от підрахунок кількості елементів у дереві, пошук мінімального та максимального елементів та ін.

Алгоритм симетричного обходу дерева пошуку можна подати псевдокодом процедури 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])
```

За застосування наведеного алгоритму до обходу прикладів дерев, показаних на рис. 3.2, маємо в обох випадках той самий порядок ключів: 2, 3, 5, 5, 7, 10. Коректність описаного алгоритму випливає безпосередньо із властивості бінарного дерева пошуку.

Складність алгоритму симетричного обходу визначають функцією O(n), оскільки процедуру повторюють двічі для кожного вузла дерева: один раз для його лівого сина, другий – для правого. У разі незбалансованого дерева, складність може досягати O(n^2), що може бути неприйнятним для опрацювання великих обсягів даних.

Теорема 3.1. Якщо *x* – корінь піддерева, у якому є n вузлів, то процедуру Inorder_Tree_Walk(x) виконують за час Θ(*n*).

3.3.2. Пошук у бінарних деревах

Пошук вузла за заданим ключем у бінарному дереві пошуку відбувається за допомогою порівняння ключів вузлів та містить такі кроки:

1) порівнюють шукане значення ключа із ключем кореневого вузла дерева;

2) якщо шуканий ключ є меншим за ключ кореневого вузла, продовжують пошук у лівому піддереві шляхом повторення кроку 1 для лівого нащадка кореневого вузла;

3) якщо шуканий ключ є більшим за ключ кореневого вузла, продовжують пошук у правому піддереві шляхом повторення кроку 1 для правого нащадка кореневого вузла;

4) якщо шуканий ключ дорівнює ключу кореневого вузла, то вузол визначено;

5) якщо ключ не визначено, процедура повертає порожній покажчик.

Пошук вузла в бінарному дереві пошуку має логарифмічний час виконання в середньому випадку й найгіршому випадку, що робить його ефективним для опрацювання великих обсягів даних. Використання бінарного дерева пошуку застосовують у багатьох сферах, де потрібен швидкий пошук та вставляння елементів.

Алгоритм пошуку вузла за ключем можна подати псевдокодом процедури Tree_Search, яка має два параметри: покажчик на корінь бінарного дерева *x* та значення ключа *k*:

```
Tree_Search(x,k)
1  if x = NULL or 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 повертає покажчик на вузол із шуканим ключем, якщо дерево містить такий вузол, або NULL, якщо ні.

Алгоритм пошуку починають із кореня дерева та просувають деревом покроково вниз. На цьому шляху ключ *key*[*x*] кожного вузла x порівнюють із параметром *k* (значенням шуканого ключа). Якщо значення ключів збігаються, то пошук завершують. Інакше перевіряють умову: якщо *k* < *key*[*x*], то пошук продовжують у лівому піддереві вузла *x*; інакше – у правому піддереві.

Наприклад, пошук вузла із ключем 13 відбувається за проходження такого шляху: 15 → 6 → 7 → 13 (рис. 3.3). Вузли, які відвідують під час рекурсивного пошуку, утворюють спадний шлях, починаючи від кореня, тому складність алгоритму Tree_Search становить O(*h*), де *h* – висота дерева.

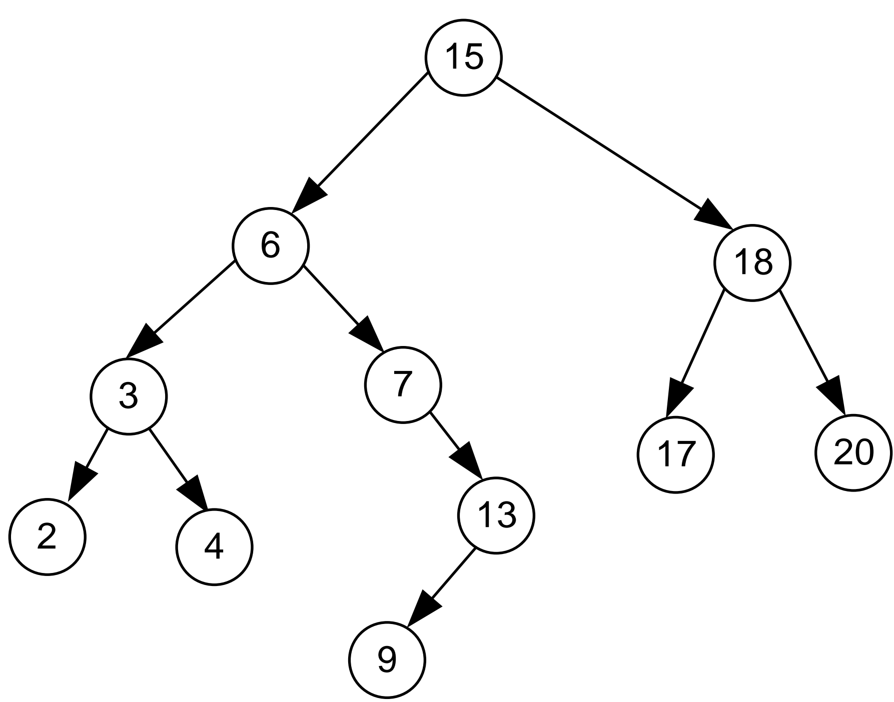


Рис. 3.3. **Запити в бінарному дереві пошуку**

Ту саму процедуру можна записати ітеративно, замінивши рекурсію циклом while. Iterative_Tree_Search(x, k):

```
1 while x ≠ NULL and k ≠ key[x]
2       do if k < key[x]
3             then x ← left[x]
4             else x ← right[x]
5  return x
```

3.3.3. Пошук мінімуму й максимуму

З огляду на основну властивість бінарних дерев пошуку, елемент із мінімальним ключем слід шукати, рухаючись від кореня за покажчиками *left* доти*,* доки не зустрінеться значення NULL, тобто буде досягнуто листя дерева. Для дерева, показаного на рис. 3.3, просуваючись за покажчиками *left*, маємо шлях 15 → 6 → 3 → 2 та мінімальне значення ключа, яке дорівнює 2.

Алгоритм пошуку мінімального ключа можна подати псевдокодом процедури Tree_Minimum(x):

```
Tree_Minimum(x)
1       while left[x] ≠ NULL
2               do x ← left[x]
3       return x
```

Коректність процедури Tree_Minimum випливає з основної властивості бінарного дерева пошуку, згідно з якою всі ключі в правому піддереві *x* є не меншими від ключа *key*[*x*], тому якщо вузол *x* не має лівого піддерева, то мінімальний ключ піддерева з коренем у вузлі *x* міститься у вузлі *x*. Якщо вузол *x* має ліве піддерево, то вузол із мінімальним ключем міститься в піддереві, коренем якого є вузол *left*[*x*].

За тією самою логікою можна визначити, що алгоритм пошуку максимального ключа є симетричним щодо алгоритму пошуку мінімального ключа:

```
Tree_Maximum(x)
1       while right[x] ≠ NULL
2               do x ← right[x]
3       return x
```

Складність процедур пошуку мінімального та максимального ключів становить O(*h*), де *h* – висота дерева.

3.3.4. Пошук попереднього та наступного елементів

Іноді для певного вузла бінарного дерева пошуку потрібно визначити, який вузол є наступним за ним у відсортованій послідовності симетричного обходу, а який вузол є попереднім. Принцип побудови бінарного дерева пошуку дозволяє визначити наступний вузол без порівняння ключів.

Алгоритм пошуку наступного вузла можна подати псевдокодом процедури Tree_Successor(x):

```
Tree_Successor(x)
1  if right[x] ≠ NULL
2        then return Tree_Minimum(right[x])
3  y ← p[x]
4  while y ≠ NULL and x = right[y]
5         do x ← y
6              y ← p[y]
7  return y
```

Наведений алгоритм повертає вузол, який є наступним за вузлом *x*, або NULL, якщо *x* має ключ із найбільшим значенням.

Складність алгоритму Tree_Successor становить O(h), оскільки рух виконують або шляхом униз від вихідного вузла, або шляхом угору. Процедура пошуку наступного вузла в дереві Tree_Predecessor є симетричною до процедури Tree_Successor, її складність також O(h).

За наявності в дереві вузлів, які мають ключі з однаковими значеннями, наступний та попередній вузли визначають процедурами Tree_Successor і Tree_Predecessor, відповідно.

Теорема 3.2. Процедури пошуку мінімального, максимального, попереднього та наступного елементів у бінарному дереві пошуку виконують за час O(h), де h – висота дерева.

3.3.5. Уставляння та вилучення в бінарному дереві

Процедури вставляння та вилучення елементів змінюють динамічну множину, подану за допомогою бінарного дерева пошуку. Проте такі

зміни структури даних мають бути реалізованими таким чином, щоб не змінити властивостей бінарних дерев пошуку.

Алгоритм уставляння нового елемента v у бінарне дерево пошуку T можна подати псевдокодом процедури Tree_Insert:

```
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  // Tree T is empty
11        else If key[z] < key[y]
12                then left[y] ← z
13                else right[y] ← z
```

Як параметр процедура має вузол *z*, у якого *key*[*z*] = *v*, а покажчики на ліве та праве піддерева містять значення NULL (*left*[*z*] = NULL та *right*[*z*] = NULL). Процедура змінює дерево *T* і покажчикам вузла *z* надають значення, за яких *z* виявляють уставленим у таку позицію в дереві, за якої властивості дерева не змінюються.

Роботу процедури Tree_Insert показано на рис. 3.4. Світлі вузли вказують шлях від кореня до позиції вставляння нового елемента 13; пунктиром позначено зв'язок, який додають за вставляння нового елемента.

Так само як процедури Tree_Search та Iterative_Tree_Search, процедура Tree_Insert починає роботу з кореня дерева та рухається в спадному напрямку [5]. Покажчик вузла *x* зазначає пройдений шлях, а покажчик вузла *y* указує на вузол, який є батьком вузла *x*. Після ініціалізації цикл *while* (рядки 3 – 7) переміщує ці покажчики деревом униз та ліворуч або праворуч, відповідно до результату порівняння ключів *key*[*x*] та *key*[*z*] доти, доки покажчик вузла *x* не набере значення NULL.

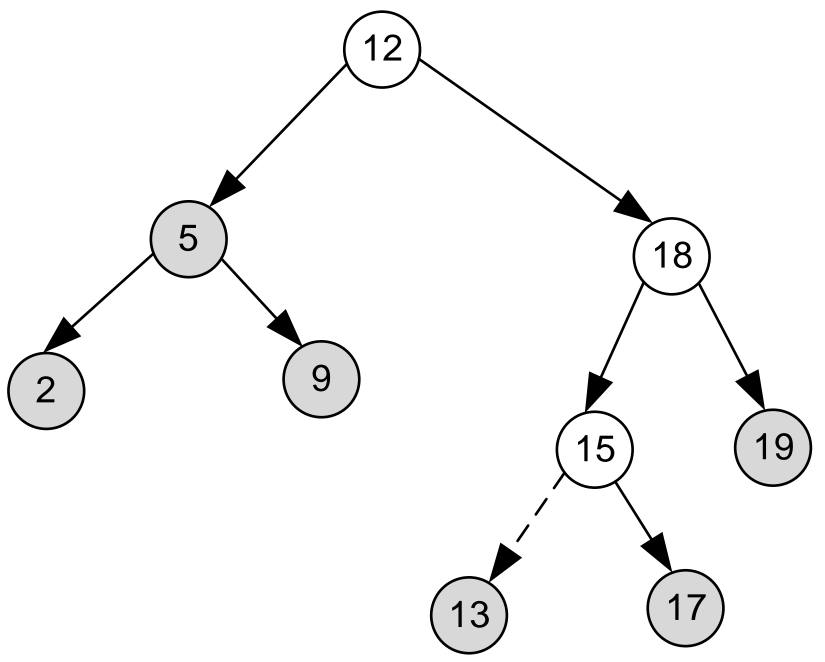


Рис. 3.4. **Уставляння елемента із ключем 13 у бінарне дерево пошуку**

Таке значення міститься саме в тій позиції, у яку слід розмістити елемент *z*. Після чого покажчики вузла *z* набирають відповідних значень (рядки 8 – 13). Процедура Tree_ Insert має складність O(*h*), де *h* – висота дерева.

Процедура вилучення цього вузла *z* із бінарного дерева пошуку має як аргумент покажчик на *z*. Процедура розглядає три варіанти розвитку подій (рис. 3.5):

1) якщо вузол *z* не має синів (рис. 3.5а), тоді змінюють покажчик на батьківський вузол *p*[*z*]: замість покажчика на вузол *z* записують значення NULL;

2) якщо вузол *z* має лише одного сина (рис. 3.5б), тоді вилучають вузол *z*, утворюючи новий зв'язок між батьківським вузлом та сином вузла *z*;

3) якщо вузол *z* має двох синів (рис. 3.5в), тоді визначають наступний за ним вузол *y*, який не має лівого сина, вузол *y* вилучають із позиції, у якій він міститься, через створення нового зв'язку між його батьком та сином, а на його позицію вставляють вузол *z*.

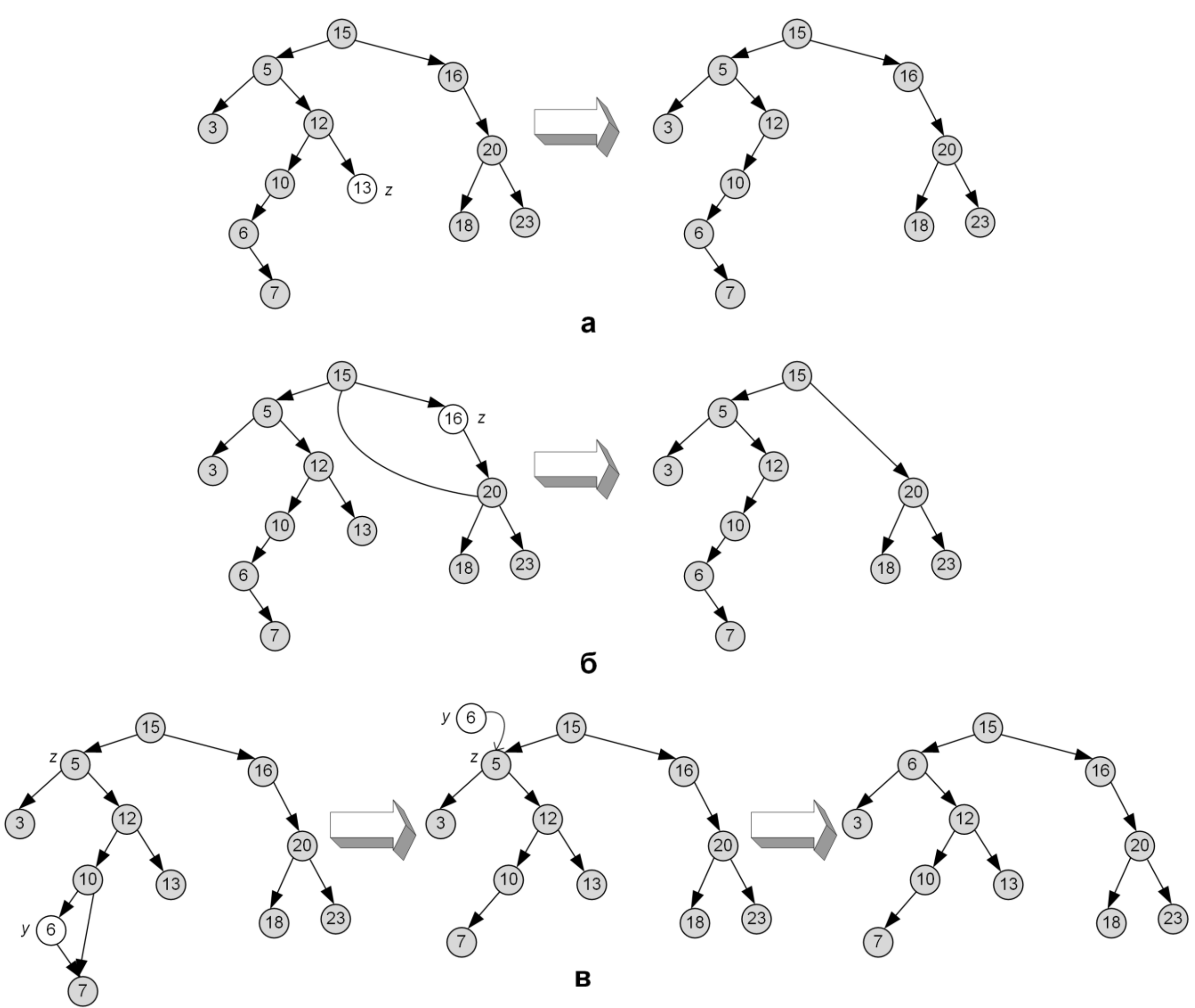


Рис. 3.5. **Вилучення вузла z із бінарного дерева пошуку**

Алгоритм вилучення вузла можна подати псевдокодом процедури Tree_Delete:

```
Tree_Delete(T, z)
1  if left[z] = NULL or 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        thenp[x] ← p[y]
```

```
9  if p[y] = NULL
10         then root[T]
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 даних з y до z
17  return y
```

В алгоритмі визначають вузол *y*, який має бути переміщеним шляхом склеювання батька та сина (рядки 1 – 3). Таким вузлом буде або вузол *z* (якщо вузол *z* має не більше за одного сина), або вузол, який іде наступним за вузлом *z* (якщо вузол *z* має двох синів). Далі покажчику вузла *x* надають покажчик на сина вузла *y* або значення NULL, якщо вузол *y* не має синів (рядки 4 – 6). Наступним кроком вузол *y* вилучають із дерева зміною покажчиків *p*[*y*] на *x* (рядки 7 – 13). Таке вилучення вузла потребує урахування граничних умов: якщо покажчик *x* дорівнює NULL або якщо *y* є коренем дерева.

Якщо вузол *y* був наступним за вузлом *z*, то змінюймо ключ вузла *z* на ключ вузла *y* (рядки 14 – 16). Вузол *y* повертають, із метою звільнення та повторного використання пам'яті, яку займав елемент *y* (рядок 17).

Процедура Tree_Delete має складність O(*h*), де *h* – висота дерева.

Теорема 3.3. Операції вставляння і вилучення в бінарному дереві пошуку висоти h можуть бути виконаними за час O(*h*).

Контрольні запитання і завдання для самоперевірки

1. Дайте визначення *n*-арного дерева.
2. Опишіть алгоритм перетворення *n*-арного дерева на бінарне.
3. Опишіть алгоритм перетворення впорядкованого лісу на бінарне дерево.
4. Перелічіть основні операції з деревами.
5. Яким чином за ключем здійснюють пошук вузла в бінарному дереві?
6. Опишіть алгоритм низхідного обходу бінарного дерева в разі застосування рекурсії.

7. Опишіть алгоритм висхідного обходу бінарного дерева.

8. Опишіть алгоритм змішаного обходу бінарного дерева в разі застосування рекурсії.

9. Що називають червоно-чорним бінарним деревом і які властивості є йому притаманними?

10. Яку структуру мають бінарні дерева? Наведіть кілька прикладів дерев.

11. У чому полягає сутність симетричного обходу дерева?

12. Опишіть дію процедури Tree_Search. Що є її вхідним параметром і що вона повертає наприкінці?

13. У чому полягає сутність алгоритму визначення мінімального (максимального) елемента на бінарному дереві?

14. Який функціонал закладено в процедури Tree_Successor і Tree_Predecessor?

15. Яким чином відбувається вставляння нового значення в бінарне дерево?

16. Яким чином відбувається вилучення елемента з бінарного дерева?

17. Чим визначено час виконання операцій уставляння та вилучення елементів із бінарного дерева?

Лабораторна робота 3
Бінарні дерева пошуку

Мета. Ознайомитися зі структурою, термінологією та призначенням бінарних дерев, набути навичок у побудові бінарних дерев та організації пошуку в них даних.

Рекомендації з підготовки до виконання. Для успішного виконання лабораторної роботи студент має знати мету, порядок виконання роботи та загальні теоретичні положення; уміти будувати блок-схеми процесів опрацювання бінарних дерев пошуку.

Завдання до виконання:

1. Упорядкуйте за допомогою бінарного дерева задану послідовність чисел за індивідуальними варіантами (табл. 3.1 і 3.2).

2. За даними табл. 3.2 позначте бінарне дерево, що задано трійками (корінь, лівий син, правий син).

Примітка. Знак ^ означає відсутність відповідного вузла.

Для обох завдань зобразіть дерево та запишіть послідовності вузлів, які утворюються під час обходу цього дерева: а) у префіксному порядку; б) в інфіксному порядку; в) у постфіксному порядку.

Таблиця 3.1

Первинні дані до лабораторної роботи, завд. 1

№ з/п	Первинні дані для упорядкування
1	{40, 3, 18, 44, 77, 61, 58, 50, 40, 80, 60, 63, 6, 66, 32}
2	{24, 50, 92, 56, 3, 52, 28, 83, 16, 46, 69, 81, 73, 80, 11}
3	{41, 13, 32, 85, 91, 3, 70, 96, 44, 80, 13, 60, 65, 92, 8}
4	{83, 10, 46, 41, 61, 68, 67, 39, 16, 41, 11, 76, 56, 60, 12}
5	{52, 47, 38, 59, 46, 67, 50, 32, 12, 63, 24, 28, 11, 71, 21}
6	{75, 14, 81, 20, 21, 96, 85, 64, 66, 15, 46, 34, 7, 96, 91}
7	{92, 82, 81, 25, 18, 51, 42, 99, 50, 71, 36, 23, 48, 35, 96}
8	{4, 2, 52, 49, 95, 88, 0, 3, 23, 99, 40, 86, 1, 40, 53}
9	{53, 4, 81, 86, 15, 5, 11, 55, 77, 51, 56, 31, 84, 8, 86}
10	{52, 30, 80, 24, 85, 23, 44, 6, 21, 66, 14, 89, 44, 56, 47}
11	{83, 19, 82, 6, 36, 84, 84, 60, 34, 43, 44, 71, 37, 59, 85}
12	{33, 58, 77, 7, 65, 95, 89, 22, 75, 56, 99, 73, 28, 30, 57}
13	{80, 77, 44, 99, 66, 61, 3, 17, 43, 51, 44, 39, 64, 16, 77}
14	{54, 14, 83, 84, 55, 40, 21, 94, 79, 77, 92, 97, 98, 86, 68}
15	{70, 59, 3, 70, 95, 23, 16, 20, 95, 40, 90, 28, 56, 97, 59}
16	{70, 45, 51, 64, 41, 61, 99, 67, 25, 70, 0, 82, 78, 90, 0}
17	{93, 12, 95, 87, 44, 26, 14, 54, 22, 44, 25, 4, 39, 26, 95}
18	{76, 88, 25, 27, 32, 78, 33, 30, 82, 90, 49, 80, 40, 33, 64}
19	{95, 33, 97, 75, 36, 8, 58, 82, 74, 57, 4, 87, 59, 31, 4}
20	{89, 79, 46, 81, 48, 51, 60, 43, 33, 95, 39, 54, 74, 89, 43}
21	{39, 58, 92, 35, 14, 11, 50, 11, 57, 0, 52, 23, 8, 24, 91}
22	{14, 19, 80, 98, 60, 21, 97, 67, 48, 30, 59, 47, 89, 33, 80}
23	{55, 70, 44, 34, 86, 18, 56, 17, 17, 76, 50, 79, 77, 58, 80}
24	{47, 3, 46, 22, 43, 43, 88, 76, 37, 42, 29, 0, 94, 17, 61}
25	{70, 62, 36, 63, 72, 24, 24, 4, 4, 60, 55, 43, 4, 14, 92}

Таблиця 3.2

Первинні дані до лабораторної роботи, завд. 2

№ з/п	Первинні дані
1	2
1	(A, B, C); (B, D, E); (C, F, G); (E, H, ^); (F, I, J); (G, K, L); (H, ^, M); (I, N, ^); (K, P, ^)
2	(A, B, C); (B, ^, D); (C, E, F); (D, G, H); (E, I, J); (F, K, L); (H, ^, M); (I, N, ^); (J, P, ^)
3	(P, M, N); (M, I, J); (N, K, L); (I, ^, D); (J, E, F); (L, G, H); (E, A, B); (G, C, ^)
4	(P, M, N); (M, J, K); (N, ^, L); (J, D, E), (K, F, G); (L, H, I); (E, ^, A); (H, B, C)
5	(A, B, C); (B, D, E); (C, F, G); (D, H, I), (E, J, ^); (F, K, L); (G, M, ^); (H, ^, N); (M, P, ^)
6	(A, B, C); (B, D, ^); (C, E, F); (D, G, H), (E, ^, I); (F, J, K); (H, L, ^); (I, M, ^); (K, N, P)
7	(P, M, N); (M, I, J); (N, K, L); (I, C, D); (J, ^, E); (K, F, ^); (L, G, H); (E, A, ^); (F, ^, B)
8	(P, M, N); (M, J, K); (N, L, ^); (J, ^, E), (K, F, G); (L, H, I); (E, A, B); (G, ^, C); (I, D, ^)
9	(A, B, C); (B, D, E); (C, F, G); (D, H, ^), (E, ^, I); (F, ^, J); (G, K, ^); (I, L, ^); (J, M, N); (K, ^, P)
10	(A, B, C); (B, ^, D); (C, E, F); (D, G, H); (E, ^, I); (F, J, ^); (G, ^, K); (H, L, ^); (I, ^, M); (J, N, P)
11	(P, M, N); (M, I, J); (N, K, L); (I, ^, E), (J, F, ^); (L, G, H); (E, A, B); (F, ^, C); (H, D, ^)
12	(P, M, N); (M, J, K); (N, L, ^); (J, F, ^); (K, G, H); (L, ^, I); (F, ^, A); (G, ^, B); (H, C, D); (I, ^, E)
13	(A, B, C); (B, D, E); (C, F, G); (E, H, ^), (F, ^, I); (G, J, ^); (H, K, L); (I, M, N); (J, ^, P)
14	(A, B, C); (C, D, E); (D, F, G); (E, H, I), (F, ^, J); (G, K, L); (H, M, N); (I, P, ^)
15	(P, M, N); (M, I, J); (N, K, L); (I, ^, F); (J, ^, G); (L, H, ^); (F, ^, A); (G, B, C); (H, D, E)
16	(P, M, N); (M, ^, J); (N, K, L); (J, ^, E); (K, F, G); (L, H, I); (E, ^, A); (G, B, ^); (H, ^, C); (I, ^, D)
17	(A, B, C); (B, D, E); (C, F, ^); (D, ^, G); (E, H, I); (F, ^, J); (G, ^, K); (H, L, M); (I, N, P)
18	(A, B, C); (B, D, E); (C, F, G); (D, ^, H), (E, I, J); (F, ^, K); (I, L, M); (J, N, P)
19	(P, M, N); (M, ^, J); (N, K, L); (J, ^, G); (K, ^, H); (L, ^, I); (G, A, B); (H, C, D); (I, E, F)

1	2
20	(P, M, N); (M, I, J); (N, K, L); (I, ^, F), (J, ^, G); (K, ^, H); (F, ^, A); (G, B, C); (H, D, E)
21	(A, B, C); (B, ^, D); (C, E, F); (D, ^, G); (E, H, I); (F, J, ^); (G, ^, K); (H, ^, L); (I, ^, M); (J, N, P)
22	(A, B, C); (B, D, E); (C, F, G); (D, ^, H), (E, ^, I); (F, ^, J); (G, ^, K); (H, L, ^); (I, M, N); (K, P, ^)
23	(P, M, N); (M, ^, J); (N, K, L); (J, E, F); (K, G, H); (L, I, ^); (F, A, B); (I, C, D)
24	(P, M, N); (M, I, J); (N, K, L); (I, ^, D); (J, ^, E); (K, F, ^); (L, G, H); (D, ^, A); (F, ^, B); (G, ^, C)
25	(A, B, C); (B, ^, D); (C, E, F); (D, G, H), (E, I, J); (F, K, ^); (G, ^, L); (I, M, ^); (K, N, P)

Загальні теоретичні положення

Розгляньмо терміни, які використовують під час роботи з бінарними деревами.

Дерево є впорядкованою (отже, відсортованою) динамічною структурою даних, зв'язним графом без циклів (зокрема, петель і кратних ребер).

Бінарне дерево (рис. 3.6) є таким деревом, кожен вузол якого має лише двох синів (ліве та праве піддерево або гілку).

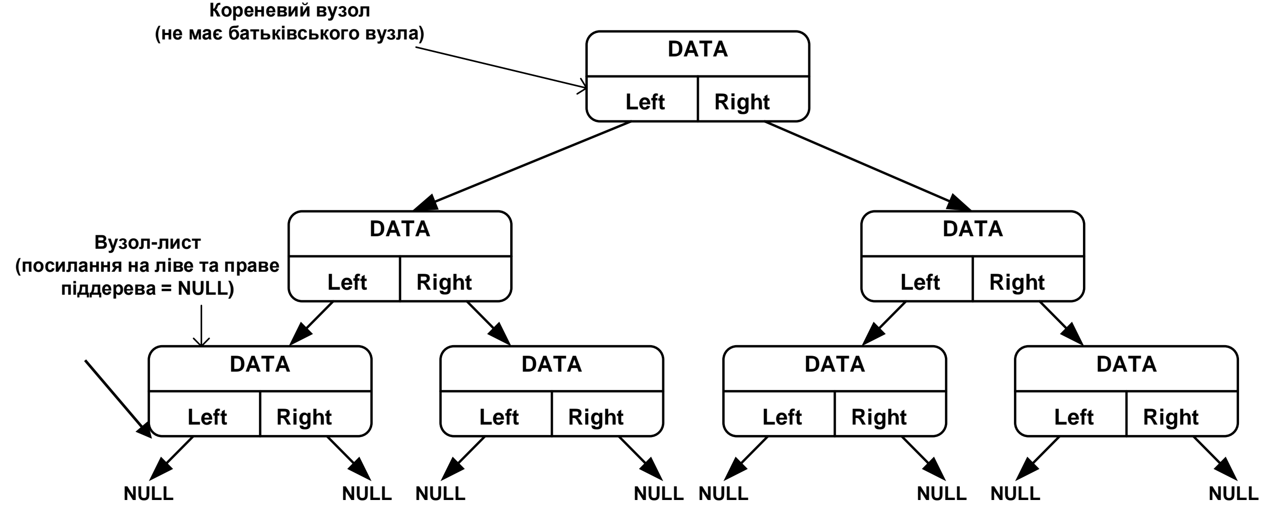


Рис. 3.6. **Структура бінарного дерева**

Кореневий вузол є найвищим вузлом в ієрархії дерева, який не має батьківського елемента. На наведеній далі схемі (рис. 3.7) це вузол A.

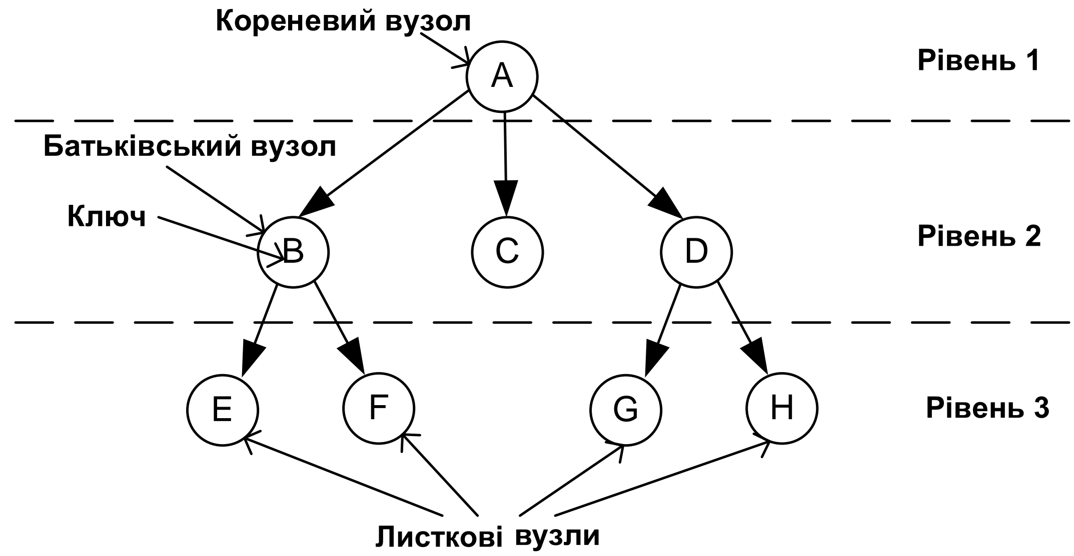


Рис. 3.7. **Схема зв'язків основних елементів дерев**

Листковий вузол є найнижчим вузлом у ієрархії дерева, який немає синів. Листкові вузли також називають *зовнішніми*. На рис. 3.7 вузли E, F, G, H і C є листковими (зовнішніми) вузлами.

Піддерево є множиною всіх нащадків вузла. Дерево складається з кореня та одного або кількох піддерев. На рис. 3.7 (B – E, B – F) та (D – G, D – H) є піддеревами.

Батьківський вузол – це будь-який вузол, крім кореневого, який є безпосереднім попередником вузла, який розглядають.

Вузлами предків є всі вузли-попередники на шляху від кореня до вузла, який розглядають. Зверніть увагу, що корінь не має предків. На рис. 3.7 вузли A і B є предками вузла E.

Ключем називають значення, яке записано у інформаційному полі вузла.

Рівень (**висота**) подає генерацію вузла. Корінь дерева утворює рівень 0. Сини кореня утворюють рівень 1, онуки кореня – рівень 2 тощо. Загалом кожен вузол розміщено на рівні вищому, ніж його батьківський вузол. Висотою дерева також називають довжину найдовшого шляху від кореня до будь-якого листка, висотою (або рівнем) вузла v – довжину шляху від кореня до цього вузла.

Шлях – це послідовність ребер, які з'єднують певні вузли. На рис. 3.7 шлях від кореня до вузла E становить послідовність A => B => E.

Степінь вузла визначено кількістю синів, яких має. На показаній схемі (див. рис. 3.7) степінь вузлів B і D дорівнює 2, степінь вузла C дорівнює 0.

Нащадок *v* вузла *u* – це вузол *v*, до якого веде шлях із вузла *u*; тоді *u* є предком для вузла *v*. Якщо довжина такого шляху дорівнює 1, то *u* є батьком для *v*, а *v* – сином для *u*.

Бінарне (двійкове) дерево *T* – це орієнтоване дерево, від кожного вузла якого може відходити не більше за двох ребер. Вузли бінарного дерева можуть мати:

двох синів (рис. 3.8): вузол *b* має лівого *a* та правого *c* синів, а корінь *g* – лівого *d* та правого *j* синів;

одного сина: вузол *i* є правим сином вузла *h*, вузол *h* є лівим сином вузла *j*;

жодного сина: вузли *a*, *c*, *e*, *i* є листям дерева.

Вузли *a*, *b* та *c* є нащадками вузла *d*, тоді вузол *d* є предком для цих вузлів.

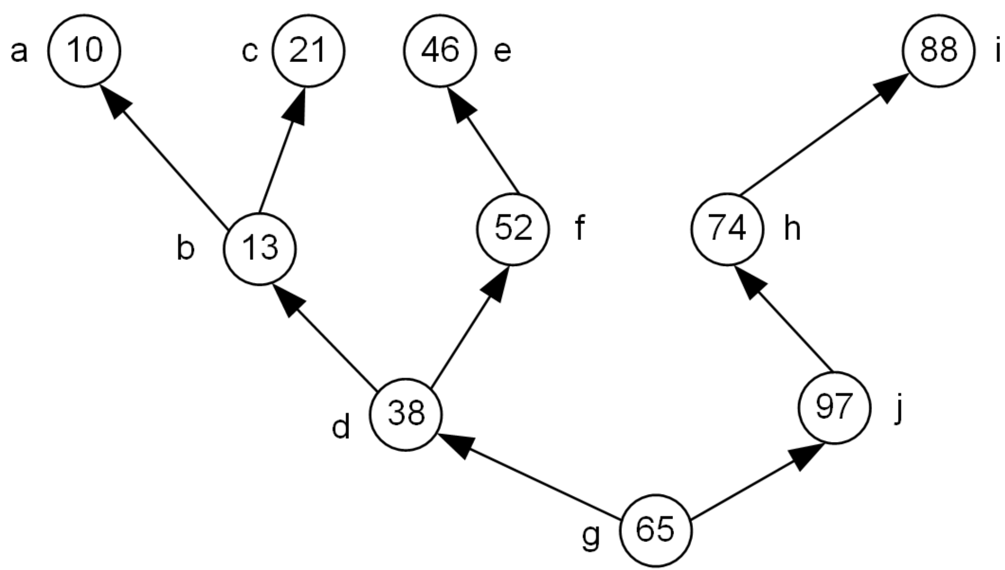


Рис. 3.8. **Упорядковане бінарне дерево**

Ліве піддерево вузла *v* – це максимальне піддерево Tl(vl) ⊂ T, коренем якого є лівий син *vl* вузла *v*.

Праве піддерево Tr(vr) ⊂ T вузла *v* – це максимальне піддерево із коренем *vr* – правим сином вузла *v*.

Для вузлів *v* бінарного дерева на кожному рівні організовано відношення строгого порядку: якщо *vl* – лівий син вузла *v*, а *vr* – правий син вузла *v*, то k[vl] < k[v] < k[vr], де k[.] – ключ відповідного вузла. Таке відношення обумовлює використання бінарних дерев для зберігання довільної інформації в упорядкованому стані.

Прикладом використання бінарних дерев є створення асоціативних масивів (рис. 3.9), елементами яких є ключі – посилання на дані (аналогічно

індексам у масивах). Ключі не обов'язково мають бути числами, проте вони мають бути унікальними й такими, щоб їх можна було порівнювати для визначення більшого або меншого за значенням.

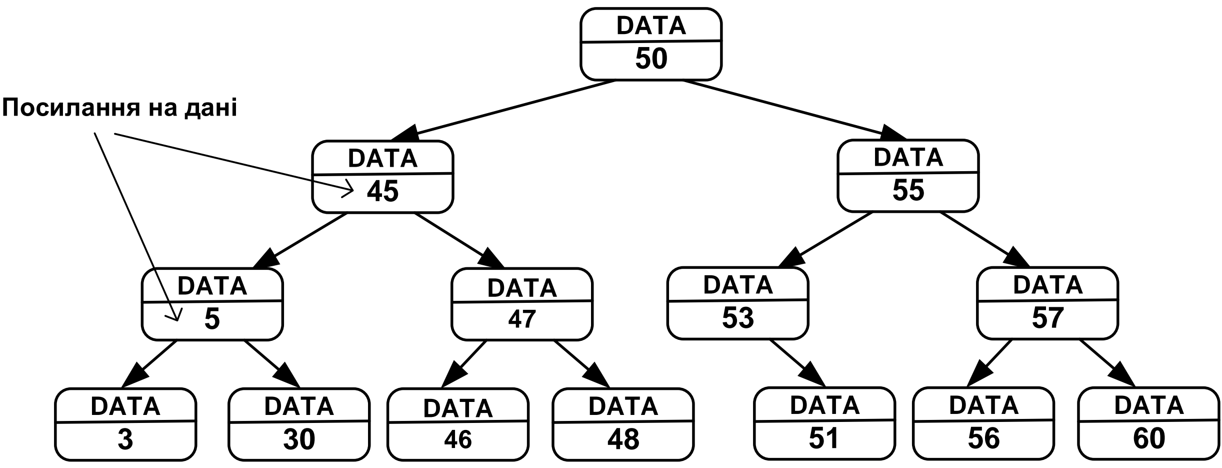


Рис. 3.9. **Асоціативна структура даних**

Бінарне дерево є рекурсивною структурою, тому що будь-який його вузол є коренем для свого піддерева. Тому під час організації обходу бінарного дерева або реалізації інших операцій доцільно використовувати рекурсивні процедури.

Будь-які два вузли дерева можуть бути з'єднаними єдиним шляхом. Кількість вузлів дерева є завжди на одиницю більшою за кількість ребер: дерево із n вершинами (n > 1) завжди містить m = n − 1 ребер.

Формулюють такі властивості дерев:

1) дерева не містять циклів, але додавання ребра між двома будь-якими несуміжними вузлами приводить до появи одного циклу;

2) дерево є зв'язною структурою, яке втрачає цю властивість за вилучення будь-якого ребра.

Вузол *v* дерева *T* називають *кінцевим (висячим),* або *листком*, якщо його степінь *ρ(v)* дорівнює 1. Безпосередньо зв'язане із кінцевим вузлом ребро також є кінцевим.

Якщо кінцеве дерево складається більш ніж з одного вузла, воно має хоча б два кінцеві вузли й хоча б одне кінцеве ребро.

Дерево з кореневим вузлом v_0 можна природним чином орієнтувати, тобто спрямувати ребра в напрямку від кореня. Таке дерево називають *орієнтованим*. За зміни вказаного напрямку ребер на протилежний, маємо також орієнтоване дерево з напрямком ребер до кореня.

Дерева можна порівнювати, але для цього, спочатку слід визначити їхні центри та привести до центрально-кореневого вигляду. Це може бути досягнуто шляхом послідовного відсікання зовнішніх (листкових) вузлів: спочатку відсікають вузли 1-го порядку, потім 2-го тощо. Процес не припиняють доти, доки не залишиться ребро із двома вузлами максимального порядку або два чи один вузол (рис. 3.10).

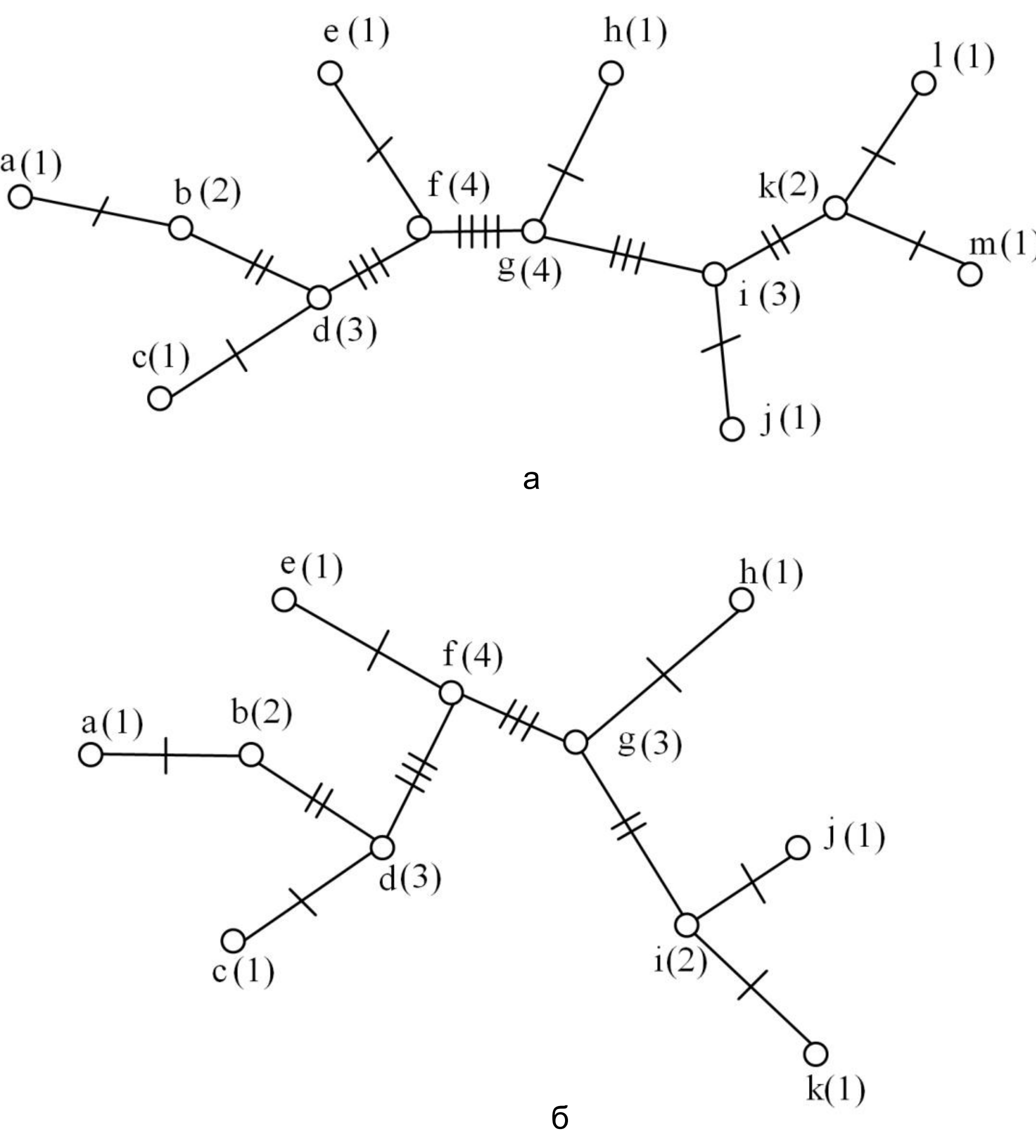


Рис. 3.10. **Схема визначення центра дерев (підготовка до центрально-кореневого подання)**

На рис. 3.10а дерево має біцентр. Із двох вузлів коренем зазвичай вибирають той, який зв'язаний із меншою кількістю зовнішніх вузлів. На рис. 3.10б залишається лише один центр, який і беруть за корінь дерева.

На рис. 3.11 висота дерева дорівнює 4, вузол k має рівень 3, листок h – рівень 2, корінь f – рівень 0 тощо (див. рис. 3.11а). Центрально-кореневе подання дерева найзручніше зображувати з урахуванням рівня відповідних вузлів, починаючи з кореня, який уважають розташованим на нульовому рівні.

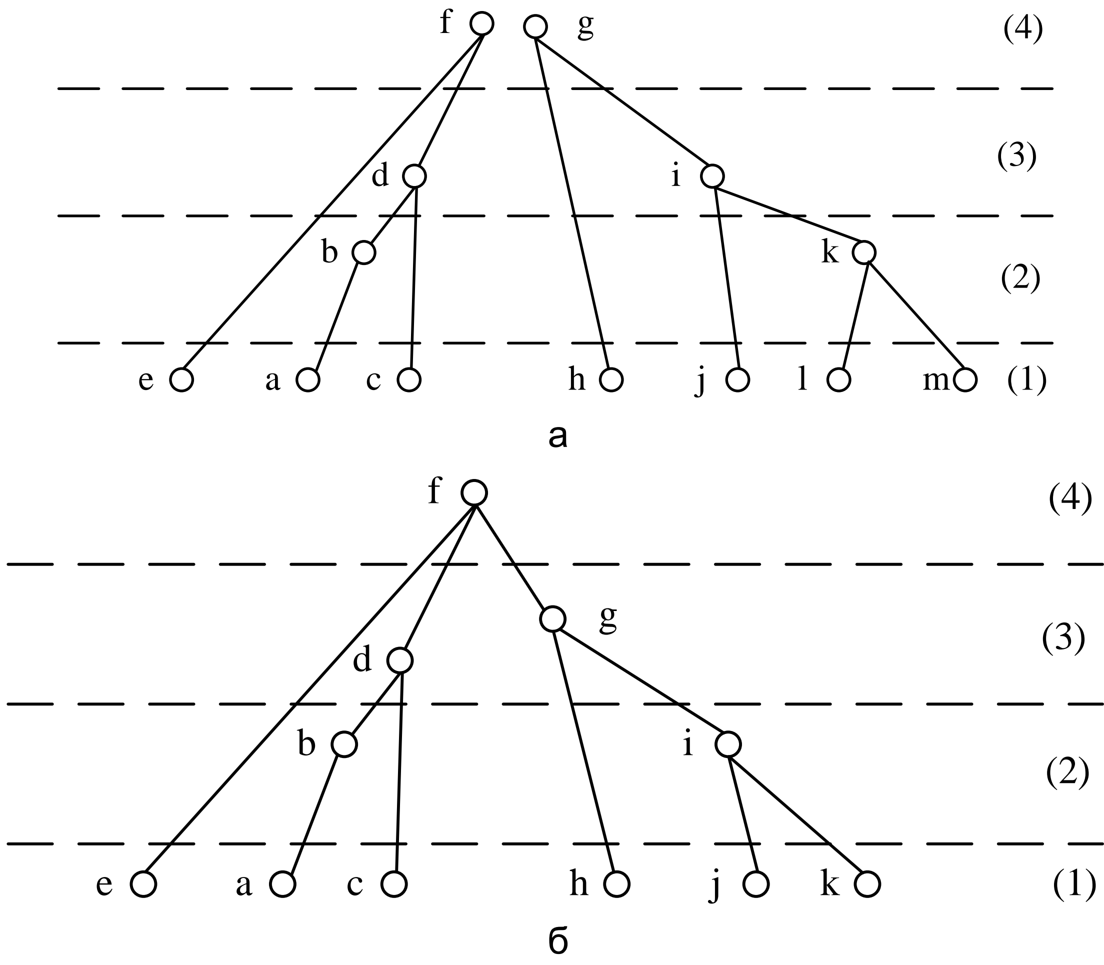


Рис. 3.11. **Центрально-кореневе подання дерев**

Порядок виконання роботи

Розгляньмо процес виконання роботи на прикладі.

Завдання 3.1. Поступово вводьте послідовність із десяти чисел: 65, 97, 38, 52, 74, 13, 88, 21, 46, 10. Потрібно забезпечити впорядкування даної послідовності, розташовуючи числа у вузлах бінарного дерева. Після завершення побудови дерева складіть алгоритм відновлення з нього послідовності чисел в упорядкованому вигляді.

Виконання завдання 3.1. Побудову бінарного дерева починають із розміщення першого числа послідовності (65) у корені дерева

(див. рис. 3.11). Наступним надходить значення 97, яке розміщують у правому сині (оскільки 97 > 65). Число 38 розміщують у лівому сині (оскільки 38 < 65). Отже, у дереві починають формування лівого та правого піддерев. Розміщення наступного числа *a* здійснюють, згідно з рекурсивним алгоритмом:

- число *a* послідовно порівнюють із значеннями вузлів u_i, які воно проходить; за результатами порівняння *a* прямує або до лівого піддерева (якщо $a < u_i$), або до правого піддерева (якщо $a > u_i$);
- крок 1 повторюють доти, доки не буде досягнуто зовнішнього вузла (листка); за досягнення листка до нього додають лівого або правого сина (за наведеним правилом порівняння), у якому розміщено *a*.

У табл. 3.3 зведено кроки руху чисел 88 і 21 до потрібної позиції в бінарному дереві.

Таблиця 3.3

Покроковий процес пошуку місця для розміщення *a в* бінарному дереві

a = 88				a = 21		
Кроки	Визначальні порівняння	Напрямки руху		Кроки	Визначальні порівняння	Напрямки руху
1	88 > 65	Праве піддерево		1	21 < 65	Ліве піддерево
2	88 < 97	Ліве піддерево		2	21 < 38	Ліве піддерево
3	88 > 74	Праве піддерево		3	21 > 10	Праве піддерево

Друга частина завдання передбачає встановлення такого порядку обходу вузлів дерева, який забезпечить виведення збережених числових значень у порядку зростання. Для цього слід скористатися інфіксним (зворотним) порядком обходу вузлів, який складається з таких етапів:

1) перевірки, чи не є поточний вузол порожнім: якщо ні – перейти до кроку 2, інакше – закінчити процедуру;

2) рекурсивного виклику інфіксного обходу для лівого піддерева;

3) опрацювання поточного вузла (кореня);

4) рекурсивного виклику інфіксного обходу для правого піддерева.

Для наведеного прикладу результатом інфіксного обходу є впорядкована числова послідовність: 10, 13, 21, 38, 46, 52, 65, 74, 88, 97.

Крім інфіксного порядку обходу вершин дерева, є префіксний (прямий) та постфіксний (кінцевий) обходи.

Процедура постфіксного обходу складається з таких етапів:

1) перевірки, чи не є поточний вузол порожнім: якщо ні – перейти до кроку 2, інакше – закінчити процедуру;

2) рекурсивного виклику постфіксного обходу для лівого піддерева;

3) рекурсивного виклику постфіксного обходу для правого піддерева;

4) опрацювання поточного вузла (кореня).

Процедура префіксного обходу складається з таких етапів:

1) перевірки, чи не є поточний вузол порожнім: якщо ні – перейти до кроку 2, інакше – закінчити процедуру;

2) опрацювання поточного вузла (кореня);

3) рекурсивного виклику префіксного обходу для лівого піддерева;

4) рекурсивного виклику префіксного обходу для правого піддерева.

Зауважмо, що всі три процедури обходу дерева мають лінійну складність O(n), де n – кількість вузлів у дереві.

Контрольні запитання до лабораторної роботи 3

1. Опишіть алгоритм приведення бінарного дерева до центрально-кореневого подання.
2. Дайте визначення бінарних дерев. Які умови мають бути виконаними для бінарного дерева?
3. Опишіть алгоритм інфіксного обходу дерева.
4. Опишіть алгоритм префіксного обходу дерева.
5. Опишіть алгоритм постфіксного обходу дерева.
6. Що таке «кореневий вузол»?
7. Дайте визначення шляху в бінарному дереві.
8. Опишіть співвідношення між вузлами в бінарному дереві.
9. Що таке «орієнтоване дерево»?
10. Що розуміють під асоціативними масивами?

Розділ 2
Алгоритмізація розв’язання прикладних задач

4. Геш-таблиці

Значну кількість програм налаштовано на роботу з динамічними множинами (set), які активно використовують операції INSERT, SEARCH і DELETE. Наприклад, компілятор, який здійснює перетворення операцій мови програмування на комп’ютерний код, потребує взаємодії з таблицею символів, де ключі елементів є довільними рядками символів, що відповідають ідентифікаторам мови.

Геш-таблиця є ефективною структурою даних для впровадження бібліотек елементів. Хоча пошук елемента в геш-таблиці може зайняти стільки ж часу, скільки й пошук елемента у зв’язаному списку ($\Theta(n)$ у гіршому разі), на практиці гешування працює надзвичайно добре. За певних припущень, середній час пошуку елемента в геш-таблиці становить $O(1)$.

Геш-таблиця узагальнює поняття звичайного масиву. За наявності достатнього обсягу пам’яті для зберігання масиву, кількість елементів якого дорівнює кількості ключів, для кожного елемента відведено комірку масиву, що надає можливість дістатися до будь-якого запису за час $O(1)$. Проте, якщо кількість комірок масиву є значно меншою за кількість можливих ключів, ефективніше використовувати гешування, сутність якого є розрахунок позиції запису в масиві на підставі ключа.

До переваг гешування зараховують його ефективність і практичність, адже за його використання виконання основних операцій у середньому потребує час $O(1)$.

4.1. Пряма адресація

Припустімо, що програмі потрібен динамічний набір, у якому кожен елемент має ключ із множини $U = \{0; 1; \ldots; m - 1\}$, де m – не надто велике число. Будемо вважати, що всі ключі є унікальними. Пряма адресація добре працює, якщо множина всіх можливих ключів U не прагне до нескінченності.

Для подання динамічного набору використаймо масив або таблицю прямих адрес T[0, . . . m − 1], де кожна позиція або слот відповідає ключу із множини U, а T[k] є позицією, яка вказує елемент у наборі із ключем k (рис. 4.1). Якщо множина не містить елемента із ключем k, то T[k] = NULL.

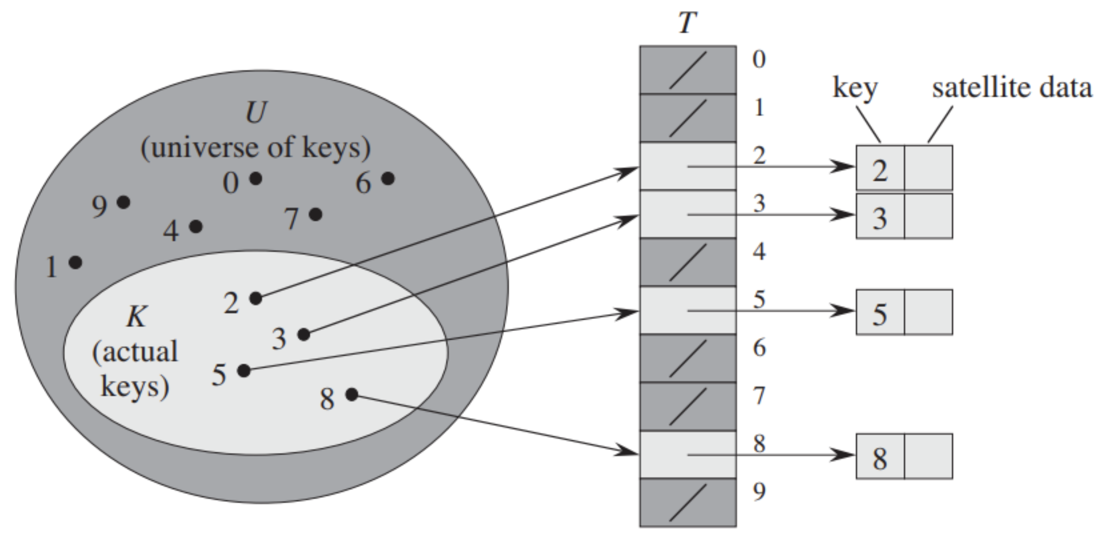


Рис. 4.1. **Реалізація динамічного набору даних за допомогою таблиці прямих адрес T**

Кожен ключ із множини U = {0; 1; …; 9} відповідає індексу в таблиці. Множина K = {2; 3; 5; 8} справжніх ключів заповнює слоти таблиці, які мають відповідні покажчики на елементи. Темно-сірі комірки таблиці T містять значення NULL.

У цьому разі основні операції реалізують тривіально й час виконання для кожної становить O(1):

```
Direct-Address-Search(T, k)
1 return T[k]
Direct-Address-Insert(T, x)
1 T[x, key] = x
Direct-Address-Delete(T, x)
1 T[x, key] = NULL
```

Для деяких застосунків таблиця прямих адрес може містити елементи із динамічного набору даних. Тобто замість того, щоб зберігати посилання та супутні дані елемента, у слоті зберігають об’єкт (тобто самі

дані), заощаджуючи таким чином місце. Щоб позначити порожній слот, усередині об'єкта використовують спеціальний ключ. Навіть більше, якщо маємо індекс об'єкта в таблиці, маємо й ключ, отже, можна не зберігати ключ об'єкта окремо. Однак у разі, коли ключі не зберігають, необхідно мати спосіб визначити, чи порожнім є слот.

4.2. Поняття геш-таблиці

Недоліки прямої адресації є очевидними: якщо множина всіх можливих ключів U є значною, зберігати таблицю T розміром $|U|$ є недоцільним або навіть неможливим, з огляду на обсяг пам'яті на персональному комп'ютері. Крім того, набір ключів K, який фактично зберігають, може бути настільки малим, порівняно з U, що більша частина пам'яті, відведеної для таблиці T, буде витраченою даремно.

Коли набір ключів K, що зберігають у таблиці, є набагато меншим, ніж множина всіх можливих ключів U, геш-таблиця потребує значно менше пам'яті, ніж таблиця прямих адрес, зокрема, можна зберігати геш-таблицю в пам'яті місткістю до $\Theta(|K|)$, де K – множина записів. Водночас зберігають виграш у часі: у геш-таблиці для пошуку елемента потрібно часу O(1). Зауважмо, що це оцінка середнього часу, тоді як для прямої адресації така оцінка в найгіршому випадку.

За прямої адресації елемент із ключем k зберігають у слоті з номером k. За використання гешування, цей елемент зберігають у слоті з номером h(k), тобто геш-функцію h використовують для обчислення номера слота від ключа k у геш-таблиці T[0, …, m − 1]:

$$h: U \to \{0, 1, \ldots, m - 1\},$$

де h – деяка функція, яка відображає множину всіх можливих ключів U у слоти геш-таблиці T[0, …, m − 1];

m – розмір геш-таблиці, що зазвичай є набагато меншим за $|U|$.

Число h(k) є геш-значенням елемента із ключем k.

Основна ідея методу гешування полягає в тому, що замість масиву розміром $|U|$ використовують масив m меншого розміру (рис. 4.2).

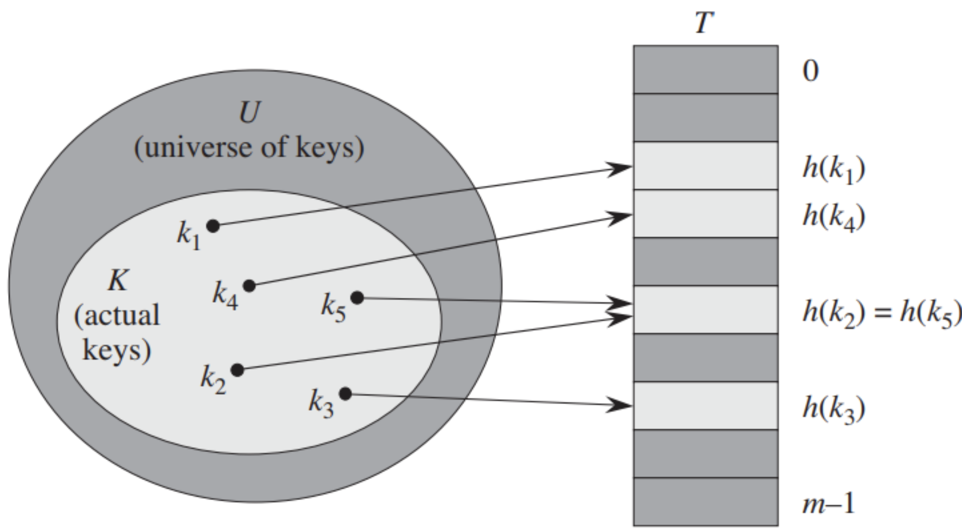


Рис. 4.2. **Використання геш-функції h для відображення ключів у слоти геш-таблиці**

Недоліком такого методу є виникнення колізій (зіткнень), тобто випадків, за яких два або більше ключів можуть гешуватися в один слот. На рис. 4.2 ключі k_2 і k_5 потрапляють в один слот, отже, виникає колізія.

Звичайно, ідеальним рішенням було б узагалі уникати таких ситуацій шляхом вибору відповідної геш-функції h. Наприклад, за умови випадковості h, колізій можна позбутися зовсім або звести їх до мінімуму. Сам термін «гешувати» (від англ. *hash* – змішувати, відтинати) асоціюється із мішаниною та відтинанням, що передає дух цього підходу. Звичайно, геш-функція h має бути детермінованою в тому сенсі, що за умови подання на вхід k на виході завжди має бути h(k).

Оскільки |U| > m, має бути принаймні два ключі, які мають однакове геш-значення, тому уникнути колізій узагалі неможливо. Отже, хоча геш-функція випадкового вигляду може мінімізувати кількість зіткнень, питання винаходу універсального методу уникнення колізій узагалі є відкритим.

Найпростішими технологіями розв'язання колізій є методи розв'язання цієї проблеми за допомогою ланцюгів та альтернативний – гешування з відкритою адресацією [3].

4.3. Розв'язання колізій за допомогою ланцюгів

За розв'язання колізій за допомогою ланцюгів розміщуймо всі елементи, які гешують, в один слот у зв'язаний список, як це показано на рис. 4.3.

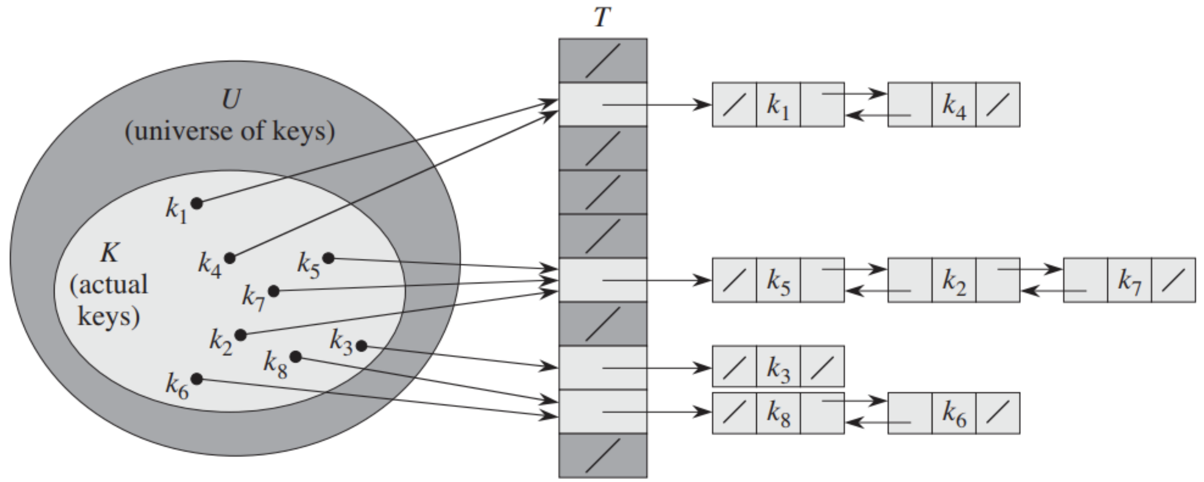


Рис. 4.3. **Розв’язання колізій за допомогою ланцюгів**

Кожен слот геш-таблиці T[j] містить зв’язаний список усіх ключів, геш-значення яких дорівнюють j. Наприклад, $h(k_1) = h(k_4)$, $h(k_5) = h(k_2) = h(k_7)$ і $h(k_8) = h(k_6)$. На рис. 4.3 показано двозв’язаний список, оскільки операцію вилучення в цьому разі виконують швидше. У загальному випадку пов’язаний список може бути як одно- так і двозв’язаним.

Слот j містить покажчик на початок списку. Усі збережені елементи, які гешують до j-го слота, є елементами цього списку; якщо таких елементів немає, слот j містить NULL.

Основні операції з геш-таблицею T легко реалізувати під час розв’язання колізії за допомогою ланцюга:

- **Уставляння** – CHAINED-HASH-INSERT

1 insert x at the head of list T[h(x. key)];

- **Пошук** – CHAINED-HASH-SEARCH

1 search for an element with key k in list T[h(k)];

- **Вилучення** – CHAINED-HASH-DELETE

1 delete x from the list T[h(x. key)].

Найдовший час операції вставляння дорівнює O(1). Ця процедура передбачає, що елемент *x*, який уставляють, ще не наявний у таблиці, отже, спочатку слід перевірити, так це чи ні (що займає час). Водночас перевіряють наявність у таблиці елементу, ключем якого є x. key.

У найгіршому випадку час виконання операції пошуку є пропорційним довжині списку. Операція вилучення елемента, як і вставляння, у разі використання двозв’язаних списків потребує часу O(1) (див. рис. 4.3).

Зауважмо, що процедура вилучення CHAINED-HASH-DELETE бере як вхідний елемент даних *x*, а не його ключ *k*, отже, пошук х виконувати не потрібно.

Якщо використання геш-таблиці потребує частого вилучення елементів, для прискорення пошуку елемента *x* для вилучення доцільно використовувати двозв'язні списки. Використання із цією метою однозв'язних списків буде потребувати здійснення покрокового пошуку всіх його попередників у списку T[h(x. key)].

У разі використання однозв'язаних списків операції вилучення та пошуку мають однакові асимптотичні часи виконання.

4.4. Аналіз гешування з ланцюгом

Оцінімо часові витрати алгоритму гешування з ланцюгом. Припустімо, є геш-таблиця T з m слотами, яка зберігає n елементів. Визначмо коефіцієнт заповнення таблиці $\alpha = n / m$, тобто середню кількість елементів, що зберігають у ланцюжку (це число може бути більшим або меншим за 1).

Найгірший випадок гешування з ланцюгом: усі n ключів гешують в один слот, створюючи список довжиною n. Отже, час для пошуку буде дорівнювати $\Theta(n)$, як-от для пошуку в однозв'язному списку плюс час для обчислення геш-функції. Очевидно, що в цьому разі гешування не має сенсу.

Середня продуктивність гешування залежить від того, наскільки добре геш-функція h у середньому розподіляє набір ключів за слотами геш-таблиці.

Будемо вважати, що будь-який елемент з однаковою ймовірністю гешують у довільний слот, незалежно від того, куди гешували інші елементи. Це припущення називають гіпотезою простого рівномірного гешування.

Для $j = 0, 1, \dots, m - 1$, позначмо довжину списку T[j] через n_j, оскільки

$$n_j = n_0 + n_1 + \dots + n_{m-1} \quad (4.1)$$

очікуване значення n_j $E[nj] = \alpha = n / m$.

Будемо вважати, що часу O(1) достатньо для обчислення геш-значення h(k), оскільки час, необхідний для пошуку елемента із ключем k,

залежить лінійно від довжини $n_{h(k)}$ списку T[h(k)]. Відкидаючи час O(1), потрібний для обчислення геш-функції та доступу до слота h(k), час пошуку елемента із ключем k лінійно залежить від кількості елементів, перевірених алгоритмом пошуку, тобто кількості елементів у списку T[h(k)].

Розгляньмо два випадки:

- пошук є невдалим: жоден елемент таблиці не має ключа k;
- пошук успішно визначає елемент із ключем k.

Теорема 4.1. У геш-таблиці, де колізії розв'язують за допомогою ланцюга, за умови простого рівномірного гешування невдалий пошук займає в середньому $\Theta(\alpha + 1)$ часу.

Отже, очікувана кількість елементів, перевірених під час невдалого пошуку, дорівнює α, а загальний необхідний час з урахуванням обчислення h(k) становить $\Theta(\alpha + 1)$.

Ситуація в разі успішного пошуку є дещо іншою, оскільки ймовірність визначення елемента є пропорційною кількості елементів, що містить список. Однак очікуваний час пошуку все одно дорівнює $\Theta(\alpha + 1)$.

Теорема 4.2. У геш-таблиці, де колізії розв'язують за допомогою ланцюга, за умови простого рівномірного гешування успішний пошук займає в середньому $\Theta(\alpha + 1)$ часу.

Із наведених теорем можна зробити такий висновок: якщо кількість слотів геш-таблиці принаймні є пропорційною кількості елементів у таблиці, n = O(m), то

$$\alpha = n / m = O(m) / m = O(1).$$

Отже, пошук у середньому займає постійний час O(1). Оскільки в найгіршому випадку вставляння та вилучення займають O(1) часу, у разі використання двозв'язаних списків час виконання всіх основних операції в середньому становить O(1).

4.5. Побудова ефективних геш-функцій

Визначмося з параметрами ефективної геш-функції. Ефективна геш-функція має (хоча б приблизно) задовольняти припущення простого

рівномірного гешування: усі ключі мають однакову ймовірність гешування до кожного з m слотів, незалежно від того, куди гешують інші ключі, або для наступного ключа всі m геш-значень мають бути рівноймовірними.

На жаль, зазвичай не завжди є можливість контролювати виконання цієї умови, якщо немає відомостей щодо розподілу ймовірностей вибору ключів. До того ж ключі не можуть бути вибраними довільно. Іноді розподіл є відомим, наприклад, якщо відомо, що ключі є випадковими дійсними числами k, рівномірно розподіленими в діапазоні $0 \leq k < 1$, то геш-функція $h(k) = [km]$ задовольняє умову простого рівномірного гешування.

На практиці часто для створення ефективної геш-функції використовують евристичні методи. Якісна інформація про розподіл ключів може бути корисною в цьому процесі проєктування. Розгляньмо таблицю символів компілятора, де ключі є рядками символів, які подають програмні ідентифікатори. Схожі набори символів, як-от pt і pts, часто можна зустріти в одній програмі, отже, ефективна геш-функція має мінімізувати ймовірність того, що такі варіанти гешують в один слот. За ефективного підходу геш-значення мають бути незалежними від будь-яких шаблонів або збігів, які можуть бути в початкових даних.

Розгляньмо способи побудови ефективних геш-функцій: ділення із залишком, множення та універсальне гешування.

Ділення із залишком обчислює геш-значення як залишок від ділення, коли ключ ділять на задане просте число. Цей метод дає гарні результати за умови вибору простого числа, яке не є пов’язаним із жодними шаблонами в розподілі ключів.

Зазначмо, що в деяких випадках на геш-функції можуть накладати більш жорсткі умови, ніж забезпечення простого рівномірного гешування. Наприклад, ключі, які в певному сенсі є близькими або схожими, мусять мати геш-значення далекі одне від одного. Універсальне гешування надає можливість набуття геш-функціями таких властивостей.

Більшість геш-функцій припускають, що множина всіх можливих ключів (область визначення геш-функції) є множиною натуральних чисел $N = \{0; 1; 2; \ldots\}$. Отже, якщо ключі не є натуральними числами, їх довільним способом інтерпретують як натуральні числа. Наприклад, можна інтерпретувати рядок символів як ціле число, записане у відповідній системі числення. Аналогічно ідентифікатор pt можна інтерпретувати як пару

десяткових цілих чисел (112; 116), оскільки p = 112 і t = 116 у наборі символів за таблицею кодування ASCII; тоді поданий у вигляді цілого числа з основою 128 оператор pt набуде значення $112 \times 128 + 116 = 14\ 452$. Отже, будемо вважати, що ключі є натуральними числами.

4.5.1. Гешування методом ділення із залишком

Для створення геш-функцій із застосуванням методу ділення із залишком ключ k відображають в один із m слотів шляхом визначення залишку від ділення k на m. Тобто геш-функція має такий вигляд:

$$h(k) = k \bmod m,$$

де m – кількість можливих геш-значень.

Наприклад, якщо геш-таблиця має розмір m = 12, а ключ k = 100, то h(k) = 100 mod 12 = 4. Оскільки для цього є потрібною лише одна операція ділення, гешування за методом ділення із залишком є досить швидким.

Використовуючи метод ділення із залишком, зазвичай уникають певних значень m. Так наприклад, m не має бути степенем 2, оскільки якщо $m = 2^p$, то h(k) є просто p молодшими бітами числа k. Якщо немає впевненості, що всі p-бітові шаблони нижчого порядку зустрічають з однаковою ймовірністю, краще розробити геш-функцію так, щоб вона залежала від усіх бітів ключа.

Не доцільно також вибирати за m степінь десятки, якщо ключі є десятковими числами. У цьому разі частина цифр ключа повністю визначає геш-значення. Якщо ключі виникають як числа в системі числення з основою 2^p, то не варто вибирати $m = 2^p - 1$, оскільки в цьому разі однакові геш-значення мають ключі, що відрізняються переставленням 2^p-х цифр.

Ефективних результатів можна досягти, вибираючи за m просте число, не дуже близьке до степеня 2. Припустімо, необхідно виділити геш-таблицю з розв'язання колізій за допомогою ланцюга, щоб зберігати приблизно n = 2 000 рядків символів, де символ має 8 бітів. У разі невдалого пошуку перебір комбінації трьох елементів у середньому є прийнятним, отже, виділяймо геш-таблицю розміру m = 701. Такий вибір здійснено

тому, що $m = 701$ є простим числом, до того ж $701 \approx 2\ 000 / 3$ і водночас 701 є далеким від будь-якого степеня 2. Тобто за геш-функцію вибирають $h(k) = k \bmod 701$.

4.5.2. Гешування методом множення

Метод множення для створення геш-функцій працює у два етапи.

1. Множмо ключ k на константу A, яка лежить у діапазоні від 0 до 1 ($0 < A < 1$) і берімо дробову частину виразу $kA \bmod 1$.

2. Множмо обчислене значення на m (це і є результат). Геш-функція в цьому разі має такий вигляд: $h(k) = m\ (kA \bmod 1)$, де $kA \bmod 1$ – дробова частина kA.

Перевагою методу множення є те, що значення m не є критичним, тобто ефективність геш-функції незначно залежить від вибору m.

Зазвичай за m вибирають степінь 2 ($m = 2^p$ для деякого цілого p), оскільки на більшості комп'ютерів множення на таке значення m реалізовано як зсув слова.

Припустімо, що розмір машинного слова дорівнює w бітів, а k уписано в одне слово. Тоді за $m = 2^p$ розрахунок геш-функції здійснюють множенням k на w-бітове ціле число $A \times 2^w$ (будемо вважати, що це число є цілим) і визначенням 2w-бітового числа. Нехай r_0 – число, що утворено молодшими w розрядами, тоді геш-значення утворено старшими p розрядами числа r_0.

Метод множення працює за будь-якого значення константи A, проте певні її значення роблять метод більш ефективним. Так, оптимальний вибір залежить від характеристик гешованих даних. Згідно із Дональдом Кнутом, за A = 0,618 033 988 7 результати роботи метода є ефективними:

$$A \approx (5 \wedge (1 / 2) - 1) = 0{,}618\ 033\ 988\ 7.$$

W-бітове подання ключа k множать на w-бітове число $s = A \times 2^w$ (рис. 4.4). Тоді виокремлюють молодші w розрядів результату множення, а з них виокремлюють p бітів старшого порядку і таким чином формують потрібне геш-значення h(k).

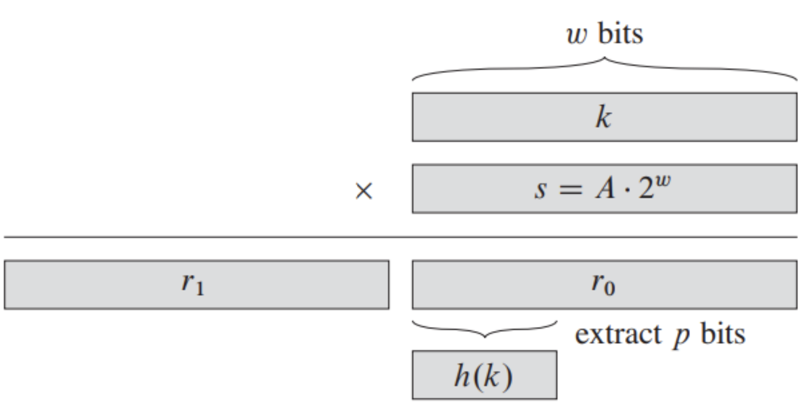


Рис. 4.4. **Метод множення**

Як приклад, припустімо, що маємо k = 123 456, p = 14, $m = 2^{14}$ = = 16 384, і w = 32. Вибираймо за A = 0,618 033 988 7, оскільки $A = 2\,654\,435\,769 / 2^{32}$. Тоді:

$$h(k) = [10\,000 \times (123\,456 \times 0{,}618\,03\ldots \bmod 1)] =$$
$$= [10\,000 \times (76\,300{,}004\,115\,1\ldots \bmod 1)] =$$
$$= [10\,000 \times 0{,}004\,115\,1\ldots] = [41{,}15\ldots] = 41.$$

4.5.3. Універсальне гешування

Якщо зловмисник вибирає ключі для гешування за допомогою фіксованої геш-функції, він може вибрати n ключів, які гешують в один слот, за середнього часу пошуку Θ(n). У такий спосіб будь-яка фіксована геш-функція може бути дискредитованою. Єдиний ефективний спосіб поліпшити ситуацію – вибирати геш-функції випадковим чином, незалежно від фактичних ключів, що зберігають. Саме універсальне гешування дозволяє реалізувати такий підхід. Будь-які дії зловмисника не досягнуть успіху за прийнятний час, якщо вибір геш-функції є невідомим.

За універсального гешування: під час виконання програми випадковим чином вибираймо геш-функцію з ретельно розробленого класу функцій. У разі повторного запуску програми з тими самими даними алгоритм буде працювати іншим чином. Як і в разі швидкого сортування, рандомізація гарантує, що за жодних вхідних даних не виникне найгіршого сценарію роботи алгоритму.

Нехай H – кінцевий набір геш-функцій, які відображають множину всіх можливих ключів U в множину {0; 1; …; m − 1}. Такий набір називають *універсальним*, якщо для кожної пари різних ключів k, l ∈ U, кількість

геш-функцій $h \in H$, для яких $h(k) = h(l)$ не більше за $|H| / m$. Інакше кажучи, із випадково вибраної з H геш-функцією ймовірність виникнення колізій між ключами k і l не більша за $1 / m$ імовірності збігу двох випадково вибраних геш-значень.

Наступна теорема свідчить на користь того, що універсальний набір геш-функцій у середньому є високоефективним.

Теорема 4.3. Припустімо, що геш-функцію h вибирають випадковим чином з універсального набору та використовують для гешування n ключів у таблицю T розміру m за використання для розв'язання колізій методу з ланцюгом. Якщо ключа k в таблиці немає, очікувана довжина $E[n_{h(k)}]$ списку, до якого його гешують, не перевищує коефіцієнта заповнення таблиці $\alpha = n / m$; за наявності ключа k в таблиці, очікувана довжина $E[n_{h(k)}]$ списку, до якого його гешують, не перевищує $1 + \alpha$.

Отже, універсальне гешування забезпечує бажаний результат, а тому унеможливлює для зловмисника вибір послідовності операцій, які визначають час роботи в найгіршому випадку. Грамотно рандомізуючи вибір геш-функції під час виконання програми, можна гарантувати виконання послідовності операцій за прийнятний час.

Наслідок із теореми 4.3. Універсальне гешування з розв'язанням колізій за допомогою ланцюга для таблиці з m слотів потребує очікуваного часу $\Theta(n)$ для опрацювання довільної послідовності з n елементів для операцій INSERT, а також для SEARCH і DELETE, що містять $O(m)$ операції INSERT.

Спроєктувати універсальний набір геш-функцій можна на підставі елементарної теорії чисел. Нехай кількість можливих геш-значень m є простим числом, а кожен ключ є послідовністю $r + 1$ байтів.

Виберіть просте число p, достатньо велике для того, щоб відобразити довільний ключ k в діапазоні від 0 до $p - 1$ включно. Нехай Z_p позначає множину $\{0; 1; \ldots; p - 1\}$, і $Z \times p$ – множину $\{1; 2; \ldots ; p - 1\}$. За припущення перевищення множиною всіх можливих ключів кількість слотів у геш-таблиці, маємо $p > m$.

Із застосуванням лінійного перетворення визначаймо геш-функцію h_{ab} для довільних $a \in Z \times p$ і $b \in Z_p$ із подальшим скороченням спочатку за модулем p, а потім за модулем m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m. \quad (4.2)$$

Так, для $p = 17$ і $m = 6$ маємо $h_{3;4}(8) = 5$.

Набір всіх таких геш-функції визначають за такою формулою:

$$h_{pm} = \{ h_{ab} : a \in Z \times p \text{ and } b \in Z_p \}. \quad (4.3)$$

Кожна геш-функція h_{ab} відображає множину Z_p в множину Z_m. Цей набір геш-функцій має перевагою ту властивість, що розмір m вихідного діапазону є довільним, тобто не обов'язково простим. Оскільки в нас є $p - 1$ варіантів для вибору значення a та p варіантів вибору для значення b, набір функцій H_{pm} містить $p(p - 1)$ геш-функції.

Теорема 4.4. Набір H_{pm} геш-функцій, визначених рівняннями (4.2) і (4.3), є універсальним.

4.6. Відкрита адресація

За відкритої адресації всі елементи займають слоти геш-таблиці. Тобто кожен запис таблиці містить елемент динамічного набору даних або NULL. Під час пошуку елемента, слоти таблиці систематично переглядають, доки не буде визначено потрібний елемент або доведено, що його немає в таблиці. За такого підходу, на відміну від методу з використанням ланцюга, списки не використовують та елементи за межами таблиці не зберігають. Отже, за відкритої адресації кількість даних, що зберігають, не може перевищувати кількість слотів таблиці, а коефіцієнт заповнення таблиці – бути більшим за одиницю ($\alpha \le 1$).

Якщо за гешування з ланцюгами можна для збереження даних використовувати вільні місця в геш-таблиці, то за використання відкритої адресації покажчики не використовують узагалі, а послідовність комірок, що переглядають, розраховують. Збереження пам'яті за допомогою покажчиків дозволяє збільшити кількість позицій у таблиці, а отже, змінити кількість колізій та прискорити пошук.

Щоб із застосуванням відкритої адресації вставити елемент у таблицю, комірки якої занумеровано цілими числами від 0 до $m – 1$, геш-таблицю послідовно переглядають до визначення порожнього слота, куди можна помістити ключ. Якщо кожного разу переглядати всі слоти (від 0-го до $m – 1$-го) час виконання буде становити $\Theta(n)$. Ідея полягає в тому, що порядок перегляду таблиці має залежати від ключа! Тобто до геш-функції

додають другий аргумент – номер спроби (нумерацію починають із 0). У цьому разі геш-функція буде мати такий вигляд:

$$h: U\text{x}\{0, 1, \dots, m - 1\} \to \{0, 1, \dots, m - 1\}.$$

Послідовність випробуваних місць або послідовність спроб (probe sequence), для певного ключа k має такий вигляд:

$$(h(k, 0), h(k, 1), \dots, h(k, m - 1)).$$

Функція h має задовольняти таку умову: кожне число від 0 до m − 1 мають зустрічати в послідовності один раз, тобто для кожного ключа всі слоти таблиці мають бути досяжними.

Розгляньмо текст процедури HASH-INSERT додавання елемента в таблицю T за використання відкритої адресації. Під час побудови процедури взято припущення, що записи, крім ключа, не містять додаткової інформації. Якщо слот таблиці є порожнім, у ньому записано значення NULL (фіксоване значення, яке є відмінним від значень усіх можливих ключів):

```
HASH-INSERT(T, k)
1 i = 0
2 repeat
3        j = h(k, i)
4       if T[j] == NIL
5          T[j] == k
6           return j
7        else i = i + 1
8  until i == m
9  error «hash table overflow»
```

Під час пошуку в таблиці з відкритою адресацією елемента із ключем k слоти переглядають у тому самому порядку, що й під час процедури додавання елемента із ключем k. Якщо визначено слот зі значенням NULL, то шуканого елемента в таблиці немає (інакше він був би записаним у цей слот). Проте такий підхід може бути застосованим лише за умови, що елементи з таблиці не вилучають.

Розгляньмо текст процедури HASH-SEARCH для виконання пошуку елементу за заданим ключем. Якщо шуканий елемент визначено в позиції j таблиці T, то процедура повертає номер цієї позиції, інакше повертає NULL.

Організація вилучення елемента з геш-таблиці з відкритою адресацією є не таким уже й простим завданням. Під час вилучення ключа зі слота не можна просто позначити цей слот як порожній (записати у нього значення NULL), тому що в майбутньому неможливо буде визначити елементи, у момент додавання яких в таблицю цей слот був зайнятим (і через це був вибраним більш віддалений слот з послідовності випробуваних місць).

Можливе рішення – записати на місце вилученого елемента замість NULL спеціальне значення DELETED. Тоді процедурою додавання елементів у таблицю мають розглядати слот зі значенням DELETED як порожній, а процедурою пошуку – як зайнятий, і продовжувати пошуки потрібного елементу далі. Недоліком такого підходу є значний час пошуку навіть за низького коефіцієнта заповнення. Тому в разі потреби вилучення записів із геш-таблиці перевагу віддають гешуванню з ланцюгом.

Під час розгляду методу відкритої адресації в подальшому будемо вважати, що має місце рівномірне гешування в тому сенсі, що всі $m!$ переставлень множини $\{0, 1, \dots, m - 1\}$ є рівноймовірними. На практиці це припущення не завжди виконують, хоча б тому, що в разі його виконання кількість ключів була б як мінімум $\leq m!$, де m – кількість геш-значень.

Для обчислення послідовностей проб, потрібних для відкритої адресації, зазвичай використовують лінійний та квадратичний методи або подвійне гешування. У кожному із цих методів послідовність $(h(k, 0), h(k, 1), \dots , h(k, m - 1))$ є переставленням множини $\{0, 1, \dots, m - 1\}$ для кожного ключа k. Однак жоден із цих методів не відповідає умові рівномірного гешування, оскільки не здатен генерувати більше ніж m^2 різних переставлень із $m!$ можливих. Подвійне гешування має найбільшу кількість переставлень для пробних послідовностей, тому дає найкращі результати.

4.6.1. Лінійний метод обчислення послідовностей проб

Нехай є звичайна геш-функція $h': U \to \{0; 1; \dots ; m - 1\}$, яку будемо називати *допоміжною.* Метод лінійного обчислення послідовностей проб використовує геш-функцію $h(k, i) = (h'(k) + i) \bmod m$, $i = 0, 1, \dots , m - 1$.

Під час роботи із ключем k спочатку досліджуймо слот T[h'(k)], тобто слот, визначений за допомогою допоміжної геш-функції. Далі перевіряймо слот T[h'(k) + 1], T[h'(k) + 2] і так далі до слота T[m − 1]. Потім переходьмо до слотів T[0], T[1], поки нарешті не доходьмо до перевірки слота T[h'(k) − 1]. Оскільки початковий слот визначає всю послідовність спроб, є лише m окремих послідовностей.

Лінійний метод обчислення послідовностей проб легко реалізують, але він має один недолік, що дістав назву *первинної кластеризації*. Первинна кластеризація (або кластеризація) полягає в утворенні довгої серії зайнятих слотів, що збільшує середнє значення часу пошуку. Якщо в таблиці з m слотами всі слоти з парними номерами є зайнятими, а з непарними – вільними, то середня кількість спроб під час пошуку відсутнього в таблиці елемента дорівнює 1,5. Якщо за такою самою кількістю зайнятих слотів m / 2 йдуть поспіль, то середня кількість спроб приблизно дорівнює m / 8 = n / 4, де n – кількість зайнятих місць у таблиці.

Процес утворення кластерів полягає в такому: якщо *i* заповнених слотів ідуть поспіль, імовірність того, що під час наступного додавання елемента до таблиці буде використаним слот, який є наступним у цій послідовності (i + 1) / m, тоді як для вільного слота, попередник якого теж є вільним, ця ймовірність становить лише 1 / m. Такий факт свідчить про те, що лінійний метод обчислення послідовностей проб є далеким від рівномірного гешування.

4.6.2. Квадратичний метод обчислення послідовностей проб

Квадратичний метод обчислення послідовностей проб використовує геш-функцію такого виду:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \qquad (4.4)$$

де h' – допоміжна геш-функція;

c_1 і c_2 – додатні допоміжні константи.

Спроби починають зі слота з номером T[h'(k)] так само як і за лінійного методу, але далі перевіряють не один за одним, а за таким принципом: номер слота, який перевіряють, квадратично залежить від номера спроби. Цей метод працює набагато краще, ніж лінійний, але для повного використання геш-таблиці, значення c_1, c_2 та m слід вибирати ґрунтовно.

Як і за лінійного методу, початковий слот визначає всю послідовність спроб, тому використовують лише m окремих послідовностей спроб, тобто m різних переставлень. За використання цього методу зникає тенденція утворення первинних кластерів, але в більш м'якій формі виникає ефект утворення вторинних кластерів.

4.6.3. Подвійне гешування

Подвійне гешування є одним із найкращих доступних методів відкритої адресації, оскільки переставлення, що генерують за цього методу, мають характеристики, близькі до випадково вибраних переставлень, а тому сам метод є найближчим до рівномірного гешування. Подвійне гешування використовує геш-функцію такого виду:

$$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m, \tag{4.5}$$

де h_1 і h_2 є допоміжними геш-функціями.

Послідовність спроб для роботи із ключем k є арифметичною прогресією (за модулем m), перший член якої є значення $h_1(k)$, а крок значенням $h_2(k)$. Приклад подвійного гешування показано на рис. 4.5.

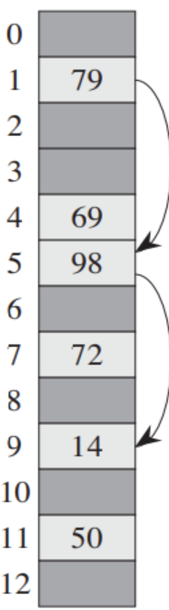


Рис. 4.5. У**ставляння елемента за допомогою подвійного гешування за відкритої адресації**

Маємо геш-таблицю розміру $m = 13$ із $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$. Якщо $k = 14$, то $14 \equiv 1 \pmod{13}$ і $14 \equiv 3 \pmod{11}$, ключ 14 ставмо в порожній слот 9, оскільки слоти 1 і 5 вже є зайнятими [9].

Значення $h_2(k)$ має бути взаємно простим до розміру геш-таблиці m, щоб покрити всю геш-таблицю. Зручний спосіб задовольнити цю умову – за m вибрати число, яке є степенем 2, а h_2 має бути таким, щоб завжди генерувалося непарне число. Інший спосіб – за m вибрати просте число, а h_2 вибрати так, щоб завжди поверталося додатне ціле число, менше за m. Наприклад, виберімо m простим і визначмо геш-функції таким чином:

$$h_1(k) = k \bmod m; \quad (4.6)$$

$$h_2(k) = 1 + (k \bmod m'), \quad (4.7)$$

де m' – число, незначно менше за m (скажімо, $m' = m - 1$ або $m' = m - 2$).

Наприклад, якщо $k = 123\,456$, $m = 701$ і $m' = 700$, маємо $h_1(k) = 80$ і $h_2(k) = 257$, тому спочатку досліджуймо позицію 80, а потім кожен 257-й слот (модуль m), поки не визначмо ключ або не завершімо огляд (кожний слот пройдено).

Коли m є простим числом або степенем 2, подвійне гешування поліпшує результат, порівняно з лінійним або квадратичним методами, оскільки за раціонального вибору функцій $h_1(k)$ та $h_2(k)$ генерують не m послідовностей, а $\Theta(m^2)$, тому що кожній можливій парі $h_1(k)$ та $h_2(k)$ відповідає власна послідовність спроб. У результаті за таких значень m продуктивність подвійного гешування здається дуже близькою до виконання ідеальної схеми рівномірного гешування.

4.6.4. Аналіз гешування з відкритою адресацією

Як і в аналізі гешування з ланцюгами, будемо здійснювати аналіз відкритої адресації в термінах коефіцієнта заповнення геш-таблиці $\alpha = n / m$. Зазвичай за відкритої адресації кожному слотові таблиці відповідає лише один запис, тому $\alpha \leq 1$.

За умови рівномірного гешування послідовність спроб генерують таким чином: ключі вибирають так, щоб усі m! можливих послідовностей спроб були рівноймовірними. Оскільки така ідеальна ситуація є далекою

від реальної, наведені далі теореми слід розглядати як евристичні викладки, а не як математичні закони, що описують роботу реальних алгоритмів.

Теорема 4.5. Математичне очікування кількості спроб під час пошуку в таблиці з відкритою адресацією елемента, якого немає, не перевищує ($\alpha - 1$) за припущення, що гешування є рівномірним.

Тобто, якщо коефіцієнт заповнення є відмінним від одиниці, то за теоремою 4.5 пошук відсутнього елемента займе в середньому час O(1).

Наприклад, якщо таблицю заповнено навпіл, то середня кількість спроб буде не більшою за 1 / (1 − 0,5) = 2, а якщо на 90 %, то не більшою за 1 / (1 − 0,9) = 10.

Наслідок із теореми 4.5. За припущення рівномірного гешування математичне очікування кількості спроб за додавання нового елемента до таблиці з відкритою адресацією не перевищує $1 / (1 - \alpha)$, де $\alpha < 1$ – коефіцієнт заповнення.

Теорема 4.6. Розгляньмо таблицю з відкритою адресацією, коефіцієнт заповнення якої не перевищує 1, $\alpha < 1$. Припустімо, що гешування є рівномірним. Тоді математичне очікування кількості спроб за успішного пошуку елемента в таблиці не перевищує $1 / \alpha \times \ln (1 / (1 - \alpha))$, якщо вважати, що ключ для успішного пошуку в таблиці вибирають випадковим чином, і всі такі вибори є рівноймовірними.

Наприклад, якщо таблицю заповнено навпіл, середня кількість спроб для успішного пошуку не перевищує 1,387, а якщо на 90 %, то є меншою за 2,559.

4.6.5. Ідеальне гешування

Хоча гешування здебільшого є гарним вибором через свою високу середню продуктивність, воно також може забезпечити непогані результати в найгіршому випадку, коли набір ключів є статичним (ключі зберігають у таблиці й ніколи не змінюють).

Деякі програми мають статичні набори ключів, наприклад, зарезервовані слова певної мови програмування або набір імен файлів на CD-ROM. Техніку гешування вважають ідеальною, якщо для виконання пошуку

в найгіршому випадку є потрібними звернення до пам'яті O(1). Щоб створити ідеальну схему гешування, використаймо два рівні з універсальним варіантом на кожному рівні.

Зовнішньою геш-функцією є $h(k) = ((a \times k + b) \bmod p) \bmod m$, де $a = 3$, $b = 42$, $p = 101$ і $m = 9$. Наприклад, $h(75) = 2$, і тому ключ 75 гешують до слота 2 таблиці T. Вторинна геш-таблиця S_j зберігає всі ключі, що гешують у слот j. Розмір геш-таблиці S_j дорівнює $m_j = n^2_j$, а пов'язану із цією таблицею геш-функція визначають як $h_j(k) = ((a_j^k + b_j) \bmod p) \bmod m_j$. Починаючи з $h_2(75) = 7$, ключ 75 зберігають у слоті 7 вторинної геш-таблиці S_2. У жодній із вторинних геш-таблиць не відбувається колізій, тому пошук у найгіршому випадку займає постійний час (рис. 4.6) [12].

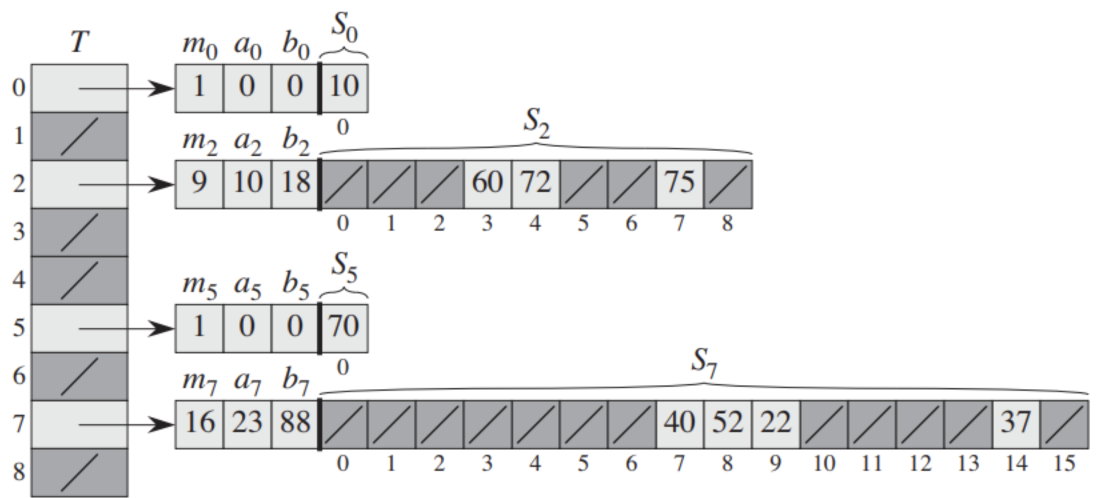


Рис. 4.6. **Використання ідеального гешування для зберігання набору K = {10; 22; 37; 40; 52; 60; 70; 72; 75}**

Перший крок ідеального гешування за сутністю повторює перший крок гешування з ланцюгом: розміщуймо n ключів у m слотів, використовуючи геш-функцію h, ретельно відібрану із сімейства універсальних геш-функцій. Замість того щоб створювати зв'язний список ключів, за гешування у слот j використовуймо невелику вторинну геш-таблицю S_j із зв'язною геш-функцією h_j (див. рис. 4.6). Ретельно вибираючи геш-функції h_j, забезпечуймо відсутність колізій на вторинному рівні. Для цього визначаймо розмір m_j геш-таблиці S_j як квадрат n_j – кількості ключів, що гешують у слот j. Зауважмо, що у загальному випадку квадратична залежність m_j від n_j може спричинити надмірні витрати пам'яті, але за ґрунтовного вибору геш-функції першого рівня можна обмежити обсяг пам'яті, яку використовують.

Теорема 4.7. За умови зберігання n ключів у геш-таблиці розміру $m = n^2$ за допомогою геш-функції h, випадковим чином вибраної із сімейства універсальних геш-функцій, імовірність виникнення колізій є меншою за 1 / 2.

Контрольні запитання і завдання для самоперевірки

1. Поясніть поняття прямої адресації, назвіть її недоліки та переваги. Наведіть приклади її застосування під час гешування.
2. Що таке «геш-таблиці»? Для чого їх застосовують?
3. У чому полягає сутність колізій у гешуванні? Які методи боротьби з ними ви знаєте?
4. У чому полягає сутність розв’язання колізій із застосуванням ланцюгів?
5. Оцініть витрати часу на пошук під час гешування з ланцюгом.
6. Що таке «геш-функції» та які способи їхньої побудови ви знаєте?
7. Опишіть сутність методу ділення із залишком під час формування геш-функцій.
8. Опишіть сутність методу множення під час формування геш-функцій.
9. У чому полягає сутність універсального гешування?
10. Які часові витрати на виконання основних операцій у разі універсального гешування з розв’язанням колізій за допомогою ланцюга?
11. Як математично описують набір універсальних геш-функцій?
12. Який розділ математики використовують під час формування набору універсальних геш-функцій?
13. Порівняйте гешування з ланцюгами з іншими видами гешування.
14. Опишіть алгоритм гешування за використання відкритої адресації.
15. Опишіть процедуру HASH-SEARCH для виконання пошуку елемента за заданим ключем у разі використання відкритої адресації.
16. Опишіть процедуру HASH-INSERT додавання елемента в таблицю у разі використання відкритої адресації.
17. Опишіть алгоритм лінійного методу обчислення послідовностей проб під час гешування за використання відкритої адресації, яку геш-функцію в цьому разі використовують?

18. Опишіть алгоритм квадратичного методу обчислення послідовностей проб під час гешування за використання відкритої адресації, яку геш-функцію в цьому разі використовують?

19. У чому полягає сутність подвійного гешування? Яку геш-функцію в цьому разі використовують?

20. Оцініть часові витрати основних операцій під час гешування з відкритою адресацією.

21. У чому полягає сутність проєктування ідеальної схеми гешування?

5. Основні алгоритми на графах

5.1. Подання графів

Графи широко застосовують під час розв'язання різних задач, й алгоритми для їхнього опрацювання є невід'ємною складовою освіти фахівця в галузі комп'ютерних наук. Час опрацювання заданого графу G = (V; E) залежить від кількості його вузлів (вершин) (|V|), тобто потужності графу, та ребер (|E|). У подальшому для спрощення будемо записувати V та E замість (|V|) та (|E|), а у програмах множину вузлів графу G – як G.V та множину ребер – як G.E.

Є два стандартні способи подання графу G = (V; E) (рис. 5.1):

набором списків суміжних вузлів;

матрицею суміжності.

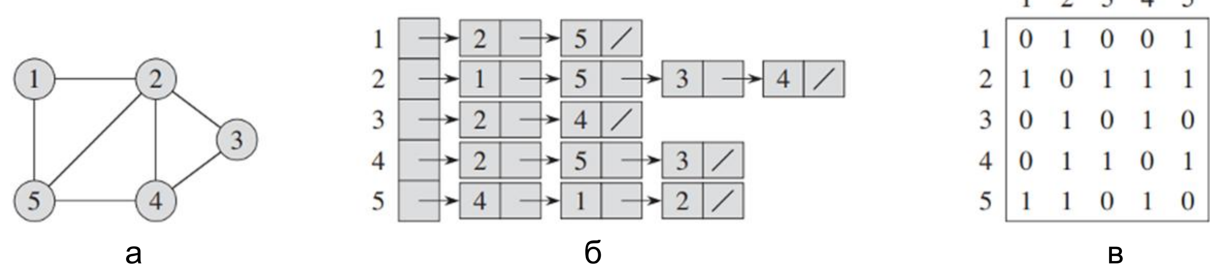


Рис. 5.1. **Способи подання неорієнтованого графу**

Перевагою першого способу є більш компактне подання розряджених графів, тобто таких, де кількість ребер є значно меншою за квадрат кількості вузлів. Для подання щільних графів, у яких |E| є порівняним

із $|V|^2$, зручніше використовувати матрицю суміжності, яка дозволяє швидко визначати, чи кожні два вузли з'єднано ребрами.

Подання графу G = (V; E) у вигляді списку суміжних вузлів використовує масив Ad_j із |V| списків – по одному на вузол. Для кожного вузла $u \in V$ список суміжних вузлів $Ad_j[u]$ містить у довільному порядку показники на всі суміжні з ним вузли (усі вузли ν, для яких $(u, \nu) \in E$).

Рис. 5.1б відповідає поданню неорієнтованого графу рис. 5.1а за допомогою набору списку суміжних вузлів, а рис. 5.1в – за допомогою матриці суміжності.

Аналогічно подання для орієнтованого графу рис. 5.2а показано на рис. 5.2б і 5.2в.

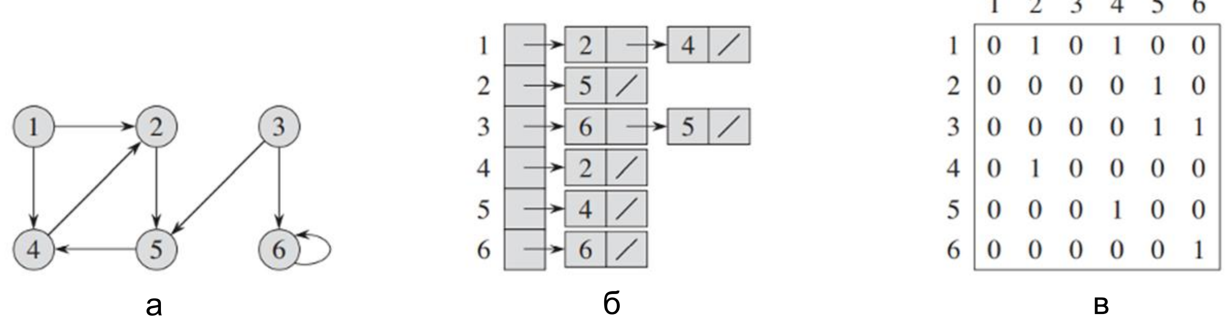


Рис. 5.2. **Способи подання орієнтованого графу**

Для орієнтованого графу сума довжин усіх списків суміжних вузлів дорівнює загальній кількості ребер: ребро (u, ν) відповідає елементу ν списку $Ad_j[u]$. Для неорієнтованого графу ця сума дорівнює подвоєній кількості ребер через те, що ребро (u, ν) породжує елемент списку суміжних вузлів як для вузла u, так і для ν. В обох випадках потреби пам'яті сягають O(max(V, E)) = O(V + E).

Списки суміжних вузлів є зручними для збереження графів із вагами, де кожному ребру відповідає вагова функція $w: E \rightarrow R$. У цьому разі зберігати вагу $w(u, \nu)$ ребра $(u, \nu) \in E$ доцільно разом із вузлом ν у списку вузлів, суміжних із u. Так само можна зберігати й іншу пов'язану із графом інформацію.

Недоліком такого подання є те, що в разі потреби у визначенні наявності в графі ребра з u у ν потрібно у пошуках ν переглядати весь список $Ad_j[u]$. Цього можна уникнути за подання графу у вигляді матриці суміжності, але це буде потребувати більше пам'яті.

За використання матриці суміжності вузли графу (V, E) нумерують числами 1, 2, …, |V| та до розгляду беруть матрицю $A = (a_{ij})$ розміром $|V| \times \times |V|$, де:

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

На рис. 5.1в та 5.2в подано матриці суміжності неорієнтованого та орієнтованого графів, зображених на рис. 5.1а та 5.2а, відповідно. Матриця суміжності потребує $\Theta(V^2)$ пам'яті, незалежно від кількості ребер у графі.

Для неорієнтованого графу матриця суміжності є симетричною відносно головної діагоналі, оскільки (u, ν) = (ν, u). Інакше кажучи, матриця суміжності неорієнтованого графу не змінюється під час транспонування, тобто переходу від матриці $A = (a_{ij})$ до матриці $A^T = (a^T_{ij})$, для якої $a^T_{ij} = a_{ji}$. Через симетричність матриці відносно діагоналі достатньо зберігати лише елементи головної діагоналі та ті, які є вищими за неї, що вдвічі скорочує обсяги пам'яті для зберігання задачі.

За подання графів матрицею суміжності вагу w(u, ν) ребра (u, ν) зберігають на перетині u-го рядка та ν-го стовпчика. Ребра, що є відсутніми, позначають NULL (0 або ∞).

Для великих графів за достатнього обсягу пам'яті краще використовувати матрицю суміжності, оскільки її легше опрацьовувати. Крім того, за відсутності ваг кожен елемент матриці суміжності займає 1 біт, тому можна зберігати кілька елементів в одному машинному слові, що дає помітну економію пам'яті.

5.2. Пошук у ширину

Пошук у ширину належить до базових алгоритмів, які становлять основу багатьох інших алгоритмів (алгоритму Дейкстри – пошуку найкоротших шляхів з одного вузла до інших вузлів графу; алгоритму Пріма – пошуку мінімального покривального дерева, який можна розглядати як узагальнення алгоритму пошуку в ширину).

Нехай задано граф G = (V, E) та зафіксовано початковий вузол s. Алгоритм пошуку в ширину перелічує всі вузли, яких можна досягти з s (якщо рухатися по ребрах) у порядку зростання відстані від s. У процесі пошуку із графу виокремлюють частину, що має назву «дерево пошуку в ширину» з коренем s. Таке дерево містить усі вузли, досяжні з вузла s і тільки їх. Для кожного із цих вузлів шлях від кореня в дереві пошуку буде одним із найкоротших шляхів із початкового вузла у графі. Цей алгоритм може бути застосованим як на орієнтованих так і на неорієнтованих графах.

Назву методу пояснено тим, що пошук рухається в ширину: спочатку переглядають усі сусідні вузли; потім вузли, що є сусідніми із сусідніми тощо.

Для наочності опису роботи методу будемо вважати, що в процесі виконання алгоритму вузли графу можуть бути білими, сірими або чорними. На початку роботи всі вузли є білими, але в процесі виконання алгоритму білий вузол може стати сірим, а сірий – чорним (але не навпаки). Натрапивши на новий вузол алгоритм фарбує його таким чином, що пофарбовані (сірі або чорні) вузли – це саме ті вузли, які вже було переглянуто. Різницю між сірими та чорними вузлами використовують алгоритмом для управління порядком обходу графу: сірі вузли утворюють першу лінію пошуку, а чорні – проміжні. Точніше, виконують таку властивість: якщо $(u, \nu) \in E$ та u чорний, то ν – сірий або чорний вузол. Отже, лише сірі вузли можуть мати суміжними ще не оглянуті вузли.

На початку дерево пошуку складається лише з кореня – початкового вузла s. Щойно алгоритм визначає новий білий вузол ν, який є суміжним із раніше визначеним вузлом u, вузол ν (разом із ребром (u, ν)) додають до дерева пошуку і він стає сином вузла u, а u стає батьком ν. Кожен вузол алгоритм визначає лише один раз, тому двох батьків він мати не може. Є також поняття предків і нащадків, що визначають таким чином: нащадки – це діти, діти дітей тощо. Рухаючись від вузлів до кореня проходьмо всіх предків. На рис. 5.3 наведено приклад виконання процедури BFS.

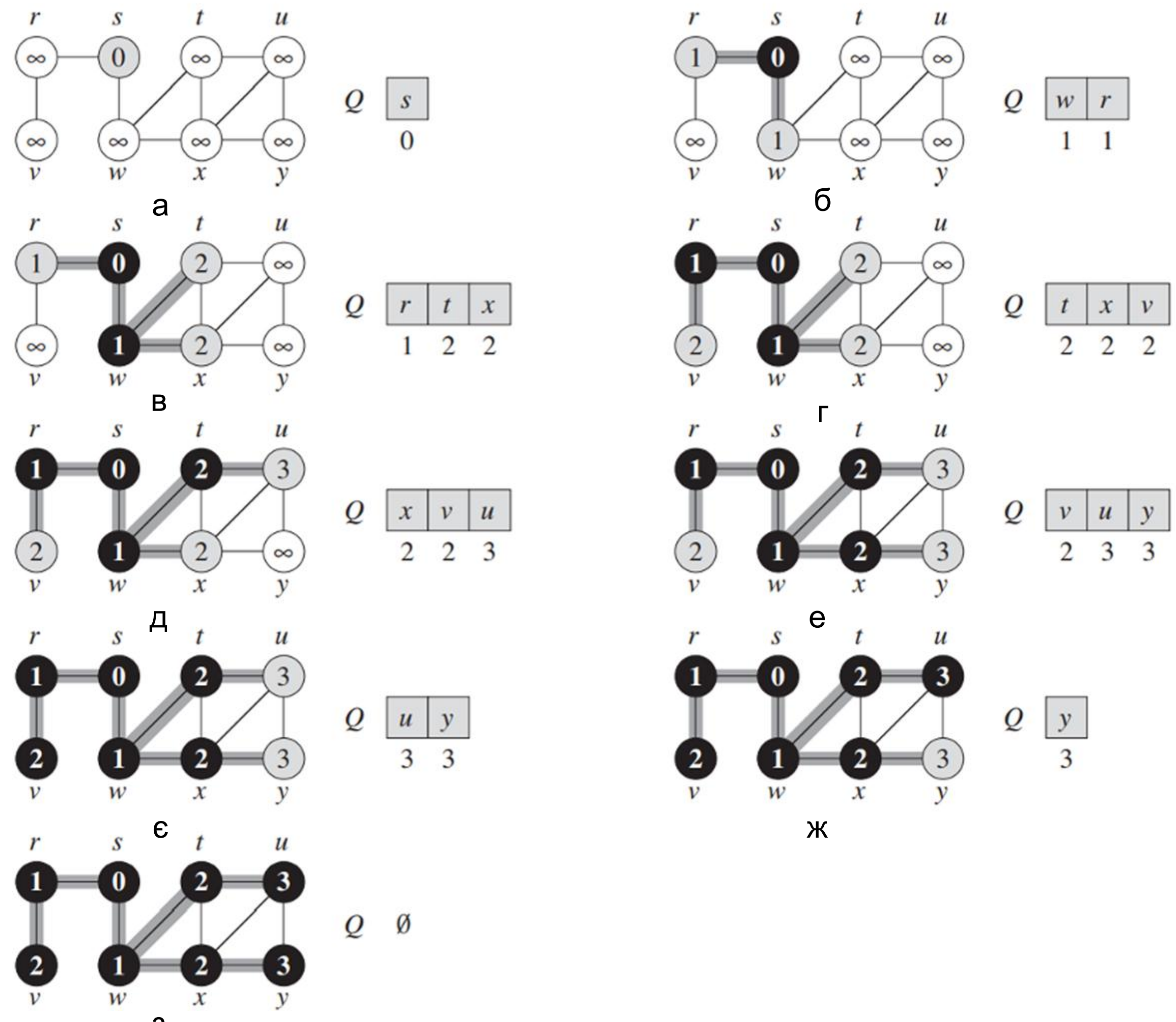


Рис. 5.3. **Приклад виконання процедури BFS**

Наведена далі процедура BFS (breadth-first search – пошук у ширину) використовує подання графу G = (V, E) у вигляді списків суміжних вузлів. Для кожного вузла u графу додатково зберігають його колір u.color та його попередній вузол u.π. Якщо попереднього вузла немає (наприклад, якщо u = s або u ще не визначено), u.π = NULL. Крім того, відстань від s до u записують у поле u.d. Процедура використовує також чергу Q(FIFO) для збереження множини сірих вузлів [13].

```
BFS(G, s)
1 for each vertex u є G. V – {s}
2       u.color = white
3       u.d = ∝
4       u. π = NIL
5  s.color = gray
```

```
6  s.d = 0
7  s.π = NIL
8  Q = 0
9  Enqueue(Q, s)
10  while Q ≠0
11         u = Dequeue(Q)
12         for each v є G. Adj[u]
13             if v.color == white
14                v.color = gray
15                v.d = v. d + 1
16                 v.π = u
17                 Enqueue(Q, v)
18          u.color = black
```

У рядках 1 – 4 наведеної процедури всі вузли набувають білого кольору, усі значення d набирають значення нескінченності, а батькам усіх вузлів надають значення NULL. У рядках 5 – 7 вузол s перефарбовують у сірий колір та виконують пов’язані із цим дії: у рядку 6 відстань s.d набирає значення 0, а в рядку 7 оголошують, що батька у вузла s немає. У рядках 8 – 9 вузол s розміщують у черзі Q та із цього моменту черга буде містити всі сірі вузли й лише їх.

Основний цикл програми (рядки 10 – 18) виконують, доки черга є непорожньою, тобто наявні сірі вузли (ті, що вже визначено, але списки суміжності яких ще не переглянуто). У рядку 11 перший такий вузол розміщено в u. Цикл for у рядках 12 – 17 переглядає всі суміжні з ним вузли. Якщо серед них виявлено білий вузол, його перефарбовують у сірий колір (рядок 14), u оголошують його батьком (рядок 16) і встановлюють відстань d.u + 1 (рядок 15). І, нарешті, цей вузол додають у хвіст черги Q (рядок 17). Після цього можна вилучити вузол u із черги Q, попередньо надавши йому чорного кольору (рядок 18).

Процедури додавання елемента у хвіст черги та вилучення елемента з голови черги, які використовують у наведеній процедурі, подано далі.

```
Enqueue(Q, s)
1 Q[Q.tail] =x
2 if Q.tail == Q.length
```

```
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
         Dequeue(Q)
1 x = Q[Q.head]
2 if Q.head == Q.length
3          Q.head = 1
4 else Q.head = Q.head + 1
5 return x
```

5.2.1. Аналіз алгоритму пошуку в ширину

Час опрацювання заданого графу G = (V, E) залежить від кількості його вузлів (вершин) (|V|), тобто потужності графу, і ребер (|E|). У подальшому для спрощення будемо записувати V та E замість (|V|) та (|E|), а в програмах множину вузлів графу G – як G.V і множину ребер – як G.E.

Оцінімо час роботи алгоритму. У процесі виконання процедури вузли лише темнішають, завдяки перевірці в рядку 13, тому кожен вузол розміщено в черзі лише один раз. Відповідно, вилучити його із черги можна також лише один раз. Кожна операція із чергою потребує O(1) кроків, тому загальний час, який витрачають на ці операції, становить O(V). Беручи до уваги, що список суміжних вузлів переглядають лише тоді, коли вузол вилучають із черги, це відбувається лише один раз. Сума довжин цих списків дорівнює |E| для орієнтованого графу та 2|E| для неорієнтованого, тому загальний час опрацювання буде становити O(E). Ініціалізація потребує O(V) кроків, відповідно, загальний час буде становити O(V + E). Отже, час роботи процедури BFS є пропорційним розміру подання графу G у вигляді списку суміжних вузлів.

5.2.2. Найкоротші шляхи

Пошук у ширину визначає відстань від початкового вузла s до кожного досяжного вузла графу G = (V, E). Під *відстанню* розуміють довжину найкоротшого шляху $\delta(s, \nu)$, яку визначають як мінімальну довжину шляху, що веде з s до ν (*довжина шляху* – це кількість ребер, за якими можна потрапити з s до V). Якщо такого шляху немає (із вузла s неможливо потрапити до вузли ν), відстань дорівнює нескінченності.

Наведімо кілька властивостей визначеної таким чином відстані.

Теорема 5.1. Припустімо s – довільний вузол графу (орієнтованого або ні), $\delta(u, \nu)$ – його ребро. Тоді: $\delta(s, \nu) \le \delta(s, u) + 1$.

Теорема 5.2. Якщо $\delta(s, \nu) > 0$, то є вузол u, відстань до якого є на одиницю меншою за $\delta(s, \nu) = \delta(s, u) + 1$ і для якого ν є суміжним вузлом.

В умовах теореми найкоротший шлях з s до ν можна визначити таким чином: додати до найкоротшого шляху з s до u ребро (u, ν).

Робота процедури BFS складається з початкового етапу (рядки 1 – 9) та повторень циклу в рядках 10 – 18. Визначмося зі станом змінних після кількох повторень тіла циклу.

Теорема 5.3. Для будь-якого цілого $k \ge 0$ є момент після кількох повторень тіла циклу (рядки 11 – 18), коли виконують такі твердження:

вузли, для яких відстань від початкового вузла є меншою за k, – чорні; дорівнює – сірі, більшою – білі;

черга Q містить виключно сірі вузли;

поля d *для чорних* та *сірих вузлів* зберігають істинні значення відстаней від початкового вузла, *для білих вузлів* – значення нескінченності;

якщо ν – це сірий або чорний вузол, то $\delta(s, \nu.\pi) = \delta(s, \nu) - 1$ та в графі є ребро $(\nu.\pi, \nu)$, для білих вузлів значенням π є NULL.

Дійсно, після виконання рядків 1 – 7 для k = 0 всі пункти теореми виконано: на відстані k міститься єдиний вузол (початковий) і він є сірим, усі інші є білими, сірі вузли розташовано в черзі, для білих вузлів d є нескінченністю, а π дорівнює NULL, для сірого вузла значення d та π є істинними.

Припустімо, для деякого k твердження теореми виконано, і після кількох ітерації циклу виконують усі твердження теореми. Далі з черги Q будуть вилучати розташовані в ній вузли (сірі). Для суміжних із ними білих вузлів будуть виконувати рядки 14 – 17, а в рядку 16 їх будуть додавати у хвіст черги. У певний момент часу із черги буде вилучено всі вузли, що містилися в ній на початку (тобто всі вузли, які розташовано на відстані k), і залишаться вузли, які були доданими останніми. Переконаймося, що всі твердження теореми наразі є виконаними для k + 1.

Вузли, що переглядають, є вузлами, які наявні в черзі й містяться на відстані k. Вузли, що додають, розташовано на відстані k + 1: вони

є суміжними з вузлами, що переглядають, і містяться на відстані k, а тому за теоремою 5.1 відстань до них не буде перевищувати $k + 1$. З іншого боку, додають лише білі вузли, а тому за правилами індукції відстань до них є більшою за k. Отже, усі вузли, що розташовано на відстані $k + 1$, будуть доданими. Дійсно, якщо вузол v розташовано на відстані $k + 1$, то за теоремою 5.2 є вузол u на відстані k, для якої він є суміжним. Вузол u має бути серед вузлів, що переглядають, і під час його опрацювання вузол v буде доданим. Слід зазначити, що вузлів u може бути декілька, проте це не змінить хід процедури, тому що під час перегляду першого з них вузол v буде доданим до черги (після чого він стане сірим та умову в рядку 13 не будуть виконувати, а тому наступного разу його вже до черги не будуть додавати).

Після завершення цього етапу процедури черга буде містити всі вузли, розташовані на відстані $k + 1$. Оскільки під час додавання до черги вони стають сірими, а після перегляду – чорними, умова кольорів буде виконаною.

Рядки 15 і 16 гарантують виконання тверджень теореми щодо відстані d та поля π.

Для доведення правильності роботи процедури BFS достатньо взяти велике значення (тобто таке k, що є більшим за максимально віддаленого від початку вузла графу). Тоді вузлів на відстані k (сірих вузлів, що вже розташовано в черзі) не буде, отже, алгоритм завершить роботу, і всі поля будуть коректно заповненими.

Теорема 5.4 (щодо коректності алгоритму BFS). Під час роботи на графі $G = (V, E)$ (орієнтованому або неорієнтованому) із початковим вузлом s процедура BFS визначить (зробить чорними) усі вузли, які є досяжними з вузла s, і для всіх $v \in V$ буде виконано рівність $v.d = \delta(s, v)$. Крім того, для будь-якого вузла $v \neq s$, досяжного з s, один із найкоротших шляхів з s до v можна визначити додаванням ребра $(v.\pi, v)$ до будь-якого найкоротшого шляху з s до v. Для вузлів, які є недосяжними з s, значення $s.\pi$ дорівнює NULL [11].

5.2.3. Дерева пошуку в ширину

Під час роботи процедури BFS виокремлюють певний підграф, дерево пошуку в ширину, яке задають полями $v.\pi$. Застосуймо процедуру

BFS до графу G = (V, E) із початковим вузлом s. Розгляньмо підграф, вузлами якого є такі, що є досяжними з вузла s, а ребрами є ребра ($v.\pi$, v) для всіх v вузлів, досяжних з s, крім s.

Теорема 5.5. Побудований таким чином підграф G_π графу G є деревом, у якому для кожного вузла є єдиний простий шлях з s до v. Такий шлях є найкоротшим шляхом з s до v в графі G.

Дерево G_π має назву *підграф-попередник* (predecessor subgraph) або *дерево пошуку в ширину* (breadth-first tree) для цього графу і початкового вузла. Зауважмо, що побудоване таким чином дерево залежить від порядку перегляду вузлів у їхніх суміжних списках.

Якщо значення полів π уже визначено за процедурою BFS, найкоротші шляхи з s можна вивести на друк за допомогою такої процедури:

```
PRINT-PATH(G, s, ν)
1  if ν == s
2       print s
3  elseif ν, π == NIL
4        print «no path from» s «to» ν «exist»
5  else PRINT-PATH(G, s, ν)
6        print ν
```

Час виконання процедури є пропорційним до шляху, який роздруковують (кожний рекурсивний виклик зменшує відстань від s на одиницю).

5.3. Пошук у глибину

Стратегія пошуку в глибину полягає в такому: слід просуватися в глиб графу, доки це можливо (є ще не пройдені ребра). Якщо таких ребер немає, слід повернутися та шукати інший шлях. Таку процедуру повторюють, доки не буде визначено всі вузли, які можуть бути досягнутими з початкової точки. Якщо залишаться незнайдені вузли, можна вибрати один із них за початковий та повторити процедуру. Пошук завершують, коли всі вузли графу визначено.

Як і під час пошуку в ширину, після першого визначення вузла ν, суміжного з u, процедура розміщує значення u в поле $\pi.\nu$. Якщо пошук генерують із декількох вузлів, маємо дерево або кілька дерев. Визначений підграф-попередник (predecessor subgraph) буде $G_\pi = (V, E_\pi)$, де $E_\pi = \{(\nu.\pi, \nu): \nu \in V$ та $\nu.\pi \neq NULL\}$. Підграф-попередник є лісом пошуку в глибину (depth-first forest), який складається з дерев пошуку в глибину (deep-first trees).

Алгоритм пошуку в глибину для розмічання вузлів також використовує кольори. На початку кожен вузол є білим. Після того як його визначено, вузол стає сірим. Коли його опрацювання завершено, він стає чорним. Опрацювання вважають завершеним, коли список суміжних із ним вузлів повністю переглянуто. Кожен вузол потрапляє лише до одного дерева в разі пошуку в глибину, отже, дерева не перетинаються (не мають однакових вузлів).

Пошук у глибину надає вузлам мітки умовного часу (подібного до номера ітерації). Кожен вузол має дві мітки:

1) у $\nu.d$ записано час, коли вузол було визначено і він набув сірого кольору;

2) у $\nu.f$ записано час, коли було завершено опрацювання списку суміжних із ν вузлів і ν набув чорного кольору.

Ці мітки часу використовують у багатьох алгоритмах на графах і вони є корисними для аналізу властивостей пошуку в глибину. Розгляньмо процедуру DFS (Depth-First Search – пошук у глибину). Мітки часу $\nu.d$ та $\nu.f$ є цілими числами від 1 до 2|V|, для будь-якого вузла u виконано таку нерівність:

$$\nu.d < \nu.f. \quad (5.1)$$

Вузол ν є білим до моменту часу $\nu.d$, сірим – між моментами часу $\nu.d$ та $\nu.f$ і чорним – після $\nu.f$.

Первинний граф може бути як орієнтованим, так і неорієнтованим. Змінна time є глобальною змінною поточного часу, який використовують для визначення міток часу вузлів. На рис. 5.4 показано роботу процедури DFS на графі [10].

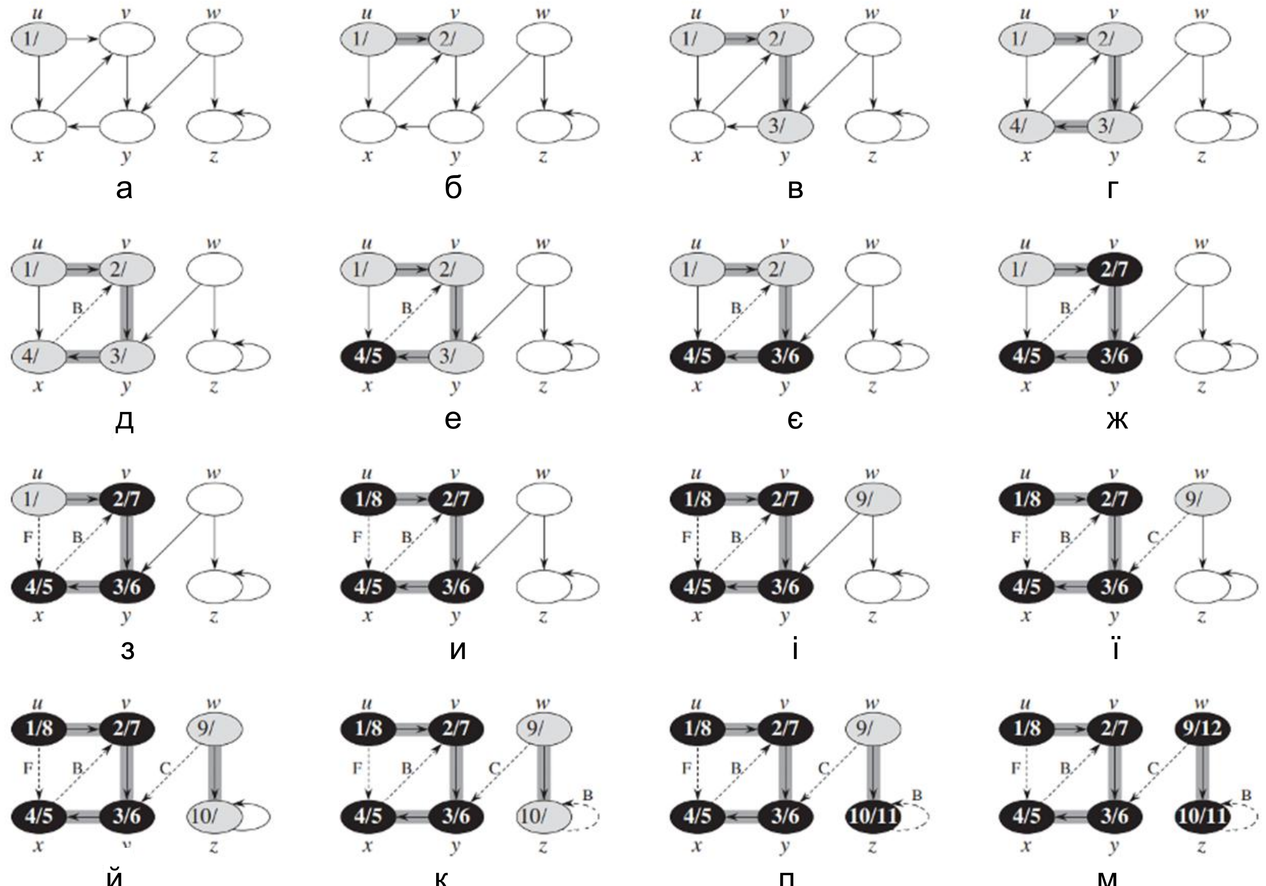


Рис. 5.4. **Приклад виконання алгоритму DFS на орієнтованому графі**

Наведімо опис алгоритму пошуку в глибину у вигляді псевдокоду процедури DFS(G):

```
DFS(G)
1 for each vertex u є G.V
2       u.color = white
3       u.π = NULL
4  time = 0
5   for each vertex u є G. V
6        if u.color = white
7               DFS-VISIT(G, u)
```

Наведімо опис алгоритму відвідування вузлів у вигляді псевдокоду процедури DFS-VISIT(G, u):

```
DFS-VISIT(G, u)
1      time = time + 1          // white vertex u has just been discovered
2      ν.d = time
3  u.color = gray
```

```
4  for each ν є G. Adj[u]   // explore edge(u, ν)
5            if v.color == white
6        ν.π = u
7            DFS-VISIT(G, ν)
8  u.color = black           // blacken u; it is finished
9  time = time + 1
10   u.f = time
```

У рядках 1 – 3 всі вузли набувають білого кольору, а в полі π розміщено значення NULL. У рядку 4 встановлено початковий (нульовий) час. У рядках 5 – 7 викликають процедуру DFS-VISIT(G, u) для всіх вузлів, які залишилися білими на момент виклику (попередні виклики процедури могли зробити їх чорними). Такі вузли стають коренями дерев пошуку в глибину.

У момент виклику DFS-VISIT(G, u) вузол u є білим. У рядку 3 він набуває сірого кольору. У рядку 2 час його визначення заносять до змінної u.d (до цього в рядку 1 лічильник часу збільшувався на 1). У рядках 4 – 7 переглядають суміжні з u вузли. Процедуру DFS-VISIT(G, u) застосовують до тих вузлів, що є білими на момент виклику (перевірка умови в рядку 5). Після перегляду всіх суміжних із u вузлів вузол u набуває чорного кольору (рядок 8), а в змінну u.f записують час цієї події (рядок 10).

Підрахуймо загальну кількість операцій, що входять до складу процедури DFS. Цикли в рядках 1 – 3 та 5 – 7 потребують часу $\Theta(V)$ (окрім викликів DFS-VISIT). Процедуру DFS-VISIT викликають по одному разу для кожного вузла: у процедуру передають білий вузол, який одразу набуває сірого кольору. Під час виконання DFS-VISIT(G, ν) цикл у рядках 4 – 7 виконано $|Adj[\nu]|$ разів. Оскільки $\sum_{v \in V} |Adj[v]| = \Theta(E)$, загальний час виконання рядків 4 – 7 процедури DFS-VISIT становить $\Theta(E)$. У підсумку маємо час $\Theta(V + E)$.

5.3.1. Властивості пошуку в глибину

Підграф-попередник, складений із дерев пошуку в глибину, повністю відповідає структурі рекурсивних викликів процедури DFS-VISIT, а саме, $u = \pi[\nu]$ тоді й тільки тоді, коли відбувається виклик DFS-VISIT(G, ν) під час перегляду списку вузлів, суміжних із u.

Інша важлива властивість полягає в тому, що моменти часу визначення та завершення опрацювання вузлів утворюють коректну (правильну)

структуру дужок (parenthesis structure) [12]. Якщо помічати визначення вузла u дужкою, що розкривається міткою u, а закінчення її опрацювання – дужкою, що закривається з такою самою міткою, то перелік подій буде становити коректно побудований вираз із дужок (парні дужки мають однакові мітки). Наприклад, пошуку в глибину на рис. 5.5а відповідає розташування дужок, зображене на рис. 5.5б.

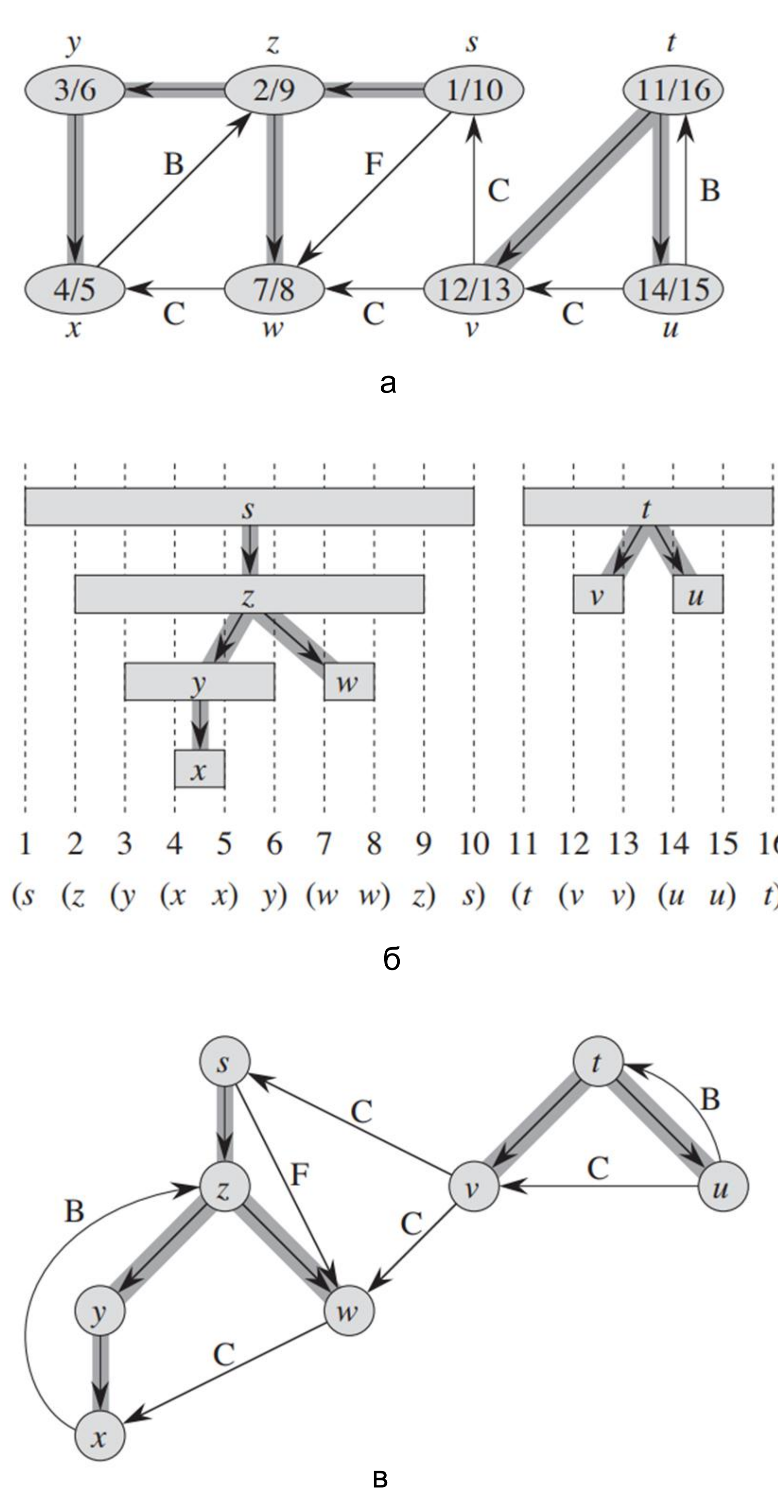


Рис. 5.5. **Властивості пошуку в глибину**

Для доведення цих та інших властивостей доцільно скористатися індукцією. Передбачено, що рекурсивним викликам процедури DFS-VISIT ця властивість є притаманною.

Доведімо, що виклик DFS-VISIT(G, u) для білого вузла u робить цей вузол чорним, а всі білі вузли, які є досяжними з нього за білими шляхами (де всі проміжні вузли також є білими) – сірими, та залишає чорні й сірі вузли без змін.

Дійсно, рекурсивні виклики в рядку 6 виконують лише для білих вузлів (перевірка в рядку 5), які є суміжними з вузлом u. Якщо вузол w був пофарбованим під час цих викликів, то за умовами індукції він був досяжним за білим шляхом з одного із суміжних із u білих вузлів, а тому є досяжним білим шляхом із u.

І навпаки, якщо вузол w є досяжним за білим шляхом із u, то він є також досяжним за білим шляхом із деякого суміжного з u білого вузла ν. Будемо вважати, що ν є першим із таких вузлів (за порядком перегляду в рядку 4). У цьому разі всі вузли білого шляху з ν до w залишаться білими на момент виклику процедури DFS-VISIT(G, ν), оскільки вони є недосяжними білими шляхами з вузлів, які передують ν, бо інакше w також був би недосяжним. За умовами індукції w набуде чорного кольору після виклику DFS-VISIT(G, ν).

Крім того, вузол u набуде спочатку сірого, а потім чорного кольору. Кольори сірих і чорних вузлів залишаються незмінними, що є правильним для рекурсивних викликів за виконання умови індукції.

Розмірковуючи аналогічним чином та дотримуючись умов індукції, можна дійти висновку, що виклик DFS-VISIT(G, u) змінює поля $\nu.\pi$ для всіх вузлів ν, що перефарбовуються, та є відмінними від u. Отже, формують дерево з коренем у u, а також до описаної раніше структури додають дужки з мітками. У результаті формують коректний вираз, зовнішні дужки якого мають мітку u, а всередині містяться дужки з мітками, які відповідають вузлам, що змінюють колір.

Узагальнення наведеного раніше дозволяє сформулювати такі теореми та наслідки:

Теорема 5.6 (*про структуру з дужок*). Під час пошуку в глибину на графі G = (V, E) (орієнтованому або неорієнтованому) для будь-яких двох вузлів u та ν виконують лише одне з таких тверджень:

1) відрізки [u.d, u.f] та [ν.d, ν.f] не перетинаються;

2) відрізок [u.d, u.f] цілком лежить усередині відрізка [ν.d, ν.f] та u є нащадком ν в дереві пошуку в глибині;

3) відрізок [ν. d, ν. f] цілком лежить усередині відрізка [u.d, u.f] та ν є нащадком u в дереві пошуку в глибину.

Наслідок із теореми 5.6 (*укладення інтервалів для нащадків*). Вузол ν є відмінним від u нащадком вузла u в лісі пошуку в глибину для орієнтованого або неорієнтованого графу G тоді й тільки тоді, коли u.d < ν.d < ν.f < u.f.

Теорема 5.7 (*про білий шлях*). Вузол ν є нащадком вузла u в лісі пошуку в глибину для орієнтованого або неорієнтованого графу G = (V, E) лише за умови, що в момент часу u.d, коли вузол було визначено, був шлях із u до ν, який складався лише з білих вузлів.

5.3.2. Класифікація ребер

Залежно від ролі, яку відіграють ребра під час пошуку в глибину, їх розподіляють на категорії. За виконання пошуку в глибину на графі G та визначення лісу G_π можна виокремити кілька категорій ребер.

Ребро (u, ν) є *ребром дерева* (tree edge) G_π, якщо вузол ν було визначено під час опрацювання цього ребра.

Зворотне ребро (back edge) (u, ν) з'єднує вузол u з його попередником ν в дереві пошуку в глибину; *ребра-цикли*, які можуть мати місце в орієнтованих графах, також уважають зворотними ребрами.

Пряме ребро (forward edge) з'єднує вузол із його нащадком, але не входить до дерева пошуку в глибину.

Перехресними ребрами (cross edges) називають усі інші ребра графу; такі ребра можуть з'єднувати два вузли одного дерева пошуку в глибину, якщо жоден із вузлів не є предком іншого, або вузли містяться в різних деревах.

На рис. 5.5. в показано граф, де прямі ребра та ребра дерев ведуть донизу, а зворотні – угору.

Алгоритм DFS може бути доповнено класифікацією ребер за їхніми категоріями. Так, категорію ребра (u, ν) можна визначити за кольором вузла ν в той момент, коли ребро досліджують уперше: білий колір свідчить, що це ребро дерева, сірий – зворотне ребро, чорний – пряме або перехресне. Слід зауважити, що за такого підходу прямі та перехресні ребра не розрізняють. Щоб усе ж таки розрізнити їх, слід скористатися таким правилом: якщо u.d < ν.d, то ребро (u, ν) є прямим, якщо u.d > ν.d – перехресним.

Окремим випадком класифікації ребер є неорієнтований граф, оскільки кожне ребро (u, ν) = (ν, u) у ньому опрацьовують двічі із двох різних боків і воно може потрапити в різні категорії. У цьому разі прийнято зараховувати ребро до тієї категорій, яка стоїть раніше в наведеному переліку категорій. Якщо вважати, що категорію ребра визначено з першого разу та вона не змінюється пізніше, прямих і перехресних ребер у графі виявлено не буде.

Теорема 5.8. Під час пошуку в глибину в неорієнтованому графі G будь-яке ребро є ребром дерева або зворотним ребром.

Нехай (u, ν) є довільним ребром графу G і u.d < ν.d. Тоді вузол ν має бути визначеним та опрацьованим, перш ніж буде закінчено опрацювання вузла u, оскільки ν міститься в списку вузлів, суміжних із u. Якщо ребро (u, ν) уперше опрацьовують в напрямку від u до ν, то ребро (u, ν) є ребром дерева. Якщо ребро (u, ν) опрацьовують уперше в напрямку від ν до u, то ребро (u, ν) є зворотним ребром (на момент дослідження, вузол u є сірим) [10].

5.3.3. Топологічне сортування

Завдання топологічного сортування (topological sort) на орієнтованому графі без циклів (dag – directed acyclic graph) полягає у визначенні такого лінійного порядку на його вузлах, за якого будь-яке ребро веде від меншого вузла до більшого в сенсі цього порядку. Якщо граф містить цикли, такого порядку немає.

Завдання топологічного сортування формулюють таким чином: розташувати на горизонтальній прямій вузли графу так, щоб усі ребра вели зліва направо.

Наведімо приклад ситуації, за якої виникає потреба в топологічному сортуванні. Професор одягається зранку. Водночас йому слід дотримуватися певного алгоритму, щоб одні речі було одягнено попереду інших. Так, шкарпетки мають бути одягненими до того, як він узується. З іншого боку, деякі речі можна одягати в довільному порядку. Наприклад, не має значення, що одягти першим, шкарпетки чи штани. На рис. 5.6а у вигляді орієнтованого графу наведено вимоги до порядку одягання вбрання. Ребро (u, v) означає, що предмет u має бути одягненим перш, ніж предмет v. Топологічне сортування такого графу є описом можливого порядку одягання. Один із варіантів наведено на рис. 5.6б.

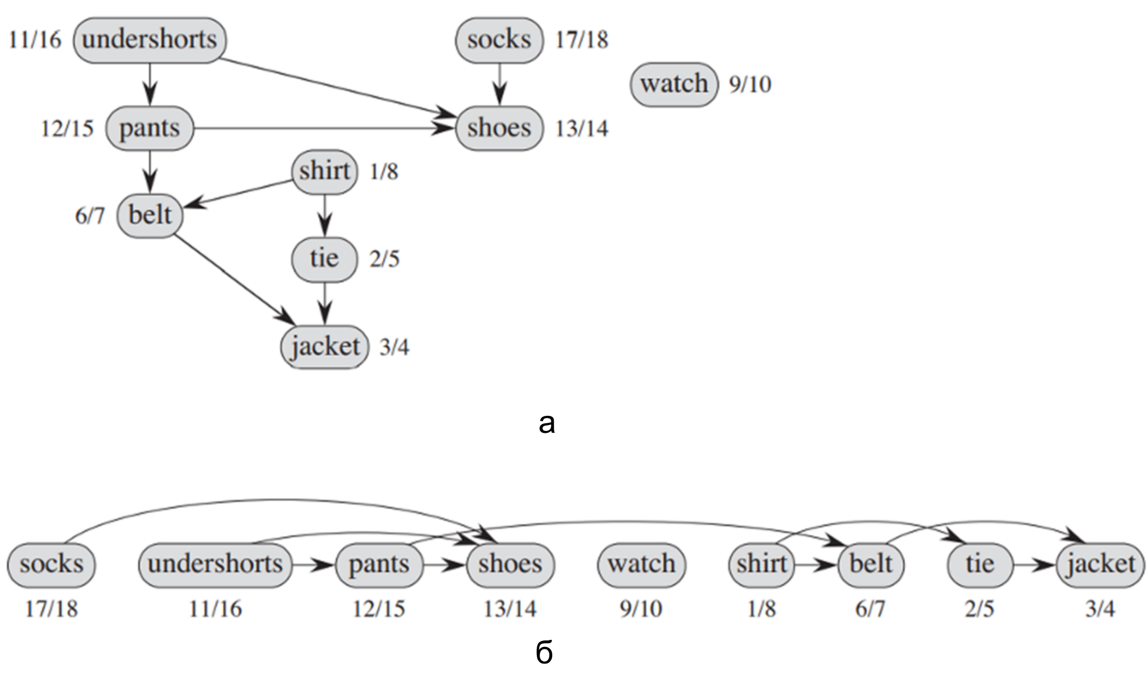


Рис. 5.6. **Топологічне сортування орієнтованого ациклічного графу**

Наведімо алгоритм топологічного сортування орієнтованого ациклічного графу. На рис. 5.6б наведено результат застосування такого алгоритму: значення $v.f$ спадають зліва направо.

```
TOPOLPGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Топологічне сортування відбувається за час $\Theta(V + E)$, тому що саме стільки часу займає процедура пошуку в глибину, а додавання кожного

з |V| вузлів до списку здійснюють за час O(1). Коректність роботи алгоритму може бути доведеною за допомогою наступної теореми.

Теорема 5.9. Орієнтований граф не має циклів тоді й тільки тоді, коли пошук в глибину не виявляє зворотних ребер.

Зворотне ребро з'єднує нащадка з попередником і замикає цикл, що утворено ребрами дерева. Нехай в графі є цикл c. Доведімо, що в цьому разі під час пошуку в глибину обов'язково буде виявленим зворотне ребро.

Виберімо в циклі вузол ν, якого буде визначено першим. Нехай (u, ν) є ребром циклу, яке виходить із цього вузла. Тоді в момент часу ν.d з ν до u веде шлях із білих вузлів. За теоремою 5.8 про білий шлях u є нащадком ν в лісі пошуку в глибину, отже, (u, ν) є зворотним ребром.

Теорема 5.10. Процедура TOPOLOGICAL-SORT(G) виконує топологічне сортування орієнтованого графу G без циклів коректно.

Слід довести, що для будь-якого ребра (u, ν) виконано умову ν.f < u.f. У момент опрацювання цього ребра вузол ν не може бути сірим, тому що це означало б, що він є предком u, і (u, ν) є зворотним ребром, а це суперечить теоремі 5.10. Тому ν в цей момент часу має бути білим або чорним. Якщо ν є білим, він є нащадком u, і тоді ν.f < u.f; якщо він є чорним, умова ν.f < u.f теж є справедливою.

5.3.4. Компоненти сильної зв'язності

Класичним прикладом застосування пошуку в глибину є розв'язання задачі розкладання графу на компоненти сильної зв'язності (Strongly-Connected-Components). Значну кількість алгоритмів, що працює на орієнтованих графах, починають із пошуку компонента сильної зв'язності, після чого розв'язок визначають для кожної з компонент окремо [14]. Наприкінці ж розв'язки комбінують із урахуванням сили зв'язків між компонентами.

Сильно пов'язаною компонентною орієнтованого графу G = (V, E) називають максимальну множину вузлів U ⊂ V, яким є притаманною така властивість: будь-які два вузли u та ν з U є досяжними один з одного (u ~> ν та ν ~> u). Приклад із виокремленими компонентами сильної зв'язності наведено на рис. 5.7.

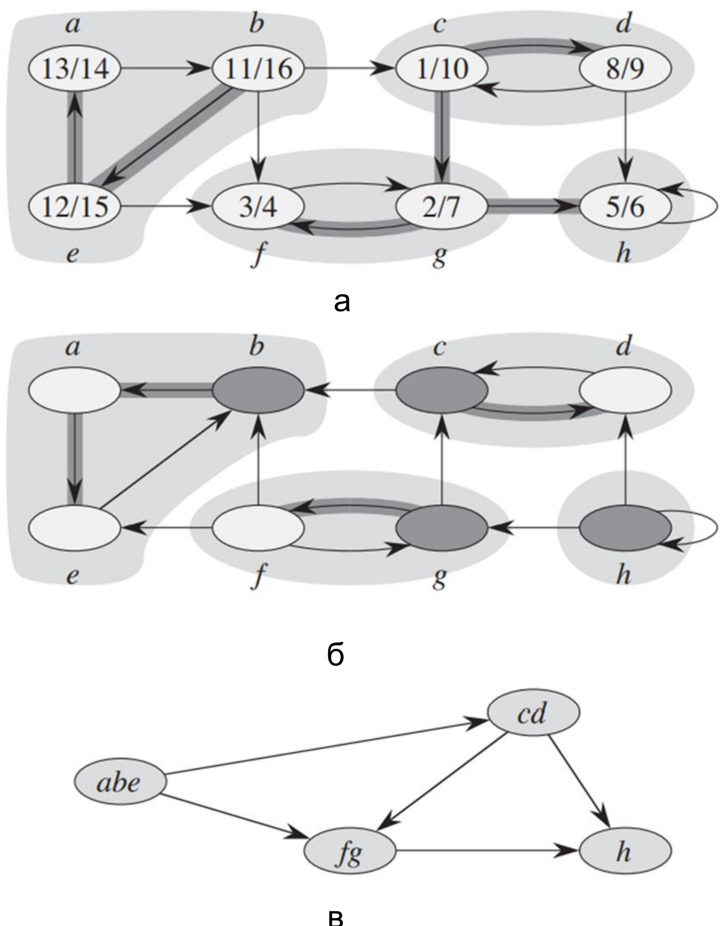


Рис. 5.7. **Граф із компонентами сильної зв’язності**

Алгоритм пошуку компонента сильної зв’язності графу $G = (V, E)$ використовує транспонований граф $G^T = (V, E^T)$, який визначають із початкового графу зміною напрямків стрілок на ребрах $E^T = \{(u, v) : (v, u) \in E\}$. Час побудови такого графу становить $O(V + E)$, якщо вважати, що початковий і транспонований графи задано списками суміжних вузлів. G та G^T мають одні й ті самі компоненти сильної зв’язності, оскільки v є досяжною з u в графі G^T, тільки якщо u є досяжною з v в графі G^T. На рис. 5.7б показано результат транспонування графу рис. 5.7а.

Наступний алгоритм визначає компоненти сильної зв’язності орієнтованого графу $G = (V, E)$, виконуючи двічі пошук в глибину – для G та G^T, витрачаючи на роботу $O(V + E)$ часу.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
```

3 call DFS(G^T), but in the main loop of DFS consider the vertices in order of decreasing u.f (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.

Аналіз наведеного алгоритму починають із таких тверджень щодо довільного орієнтованого графу:

Теорема 5.11. Якщо два вузли належать одній компоненті сильної зв'язності, жоден шлях, що їх зв'язує, не виходить за межі цієї компоненти.

Припустімо, вузол w, що лежить на шляху з u до ν, належить компоненті сильної зв'язності. Тоді u ~> w, w ~> ν та ν ~> u, звідки випливає наведене твердження.

Теорема 5.12. У процесі пошуку в глибину вузли однієї компоненти сильної зв'язності потрапляють в одне й те саме дерево.

Розгляньмо вузол довільної компоненти сильної зв'язності, якого було визначено першою та позначмо її через r. У цей момент усі інші вузли компоненти є білими, а тому досяжними з вузла r білими шляхами, оскільки за теоремою 5.12 шляхи не виходять за межі компоненти. Тому за теоремою 5.8 про білий шлях вони будуть пофарбованими в білий колір під час виклику DFS-VISIT(r) та розташованими в тому самому дереві пошуку.

Продовжуючи аналіз алгоритму STRONGLY-CONNECTED-COMPONENTS, слід узяти, що u.d та u.f будуть означати час визначення та закінчення опрацювання вузла u, визначених у рядку 1 алгоритму, а запис u ~> ν буде означати наявність шляху у G (але не в G^T).

Для кожного вузла u графу визначмо його предків (forefather) φ(u) як ті з вузлів w, які є досяжними з u, та для яких опрацювання було завершено пізніше за інші вузли: φ(u) = w: u~>w, w. f = max.

Може виявитися, що φ(u) = u.

Оскільки будь-який вузол u є досяжним сам із себе,

$$u.f \leq \varphi(u).f. \qquad (5.2)$$

Доведімо, що $\varphi(\varphi(u)) = \varphi(u)$. Дійсно, для будь-яких двох вузлів u, $\nu \in V$

$$u \rightsquigarrow \nu \Rightarrow \varphi(\nu)f \leq \varphi(u).f \quad (5.3)$$

через те, що $\{w: \nu \rightsquigarrow w\} \subseteq \{w: u \rightsquigarrow w\}$, а також через те, що предок будь-якого вузла має максимальний час завершення опрацювання серед вузлів, які є досяжними. Оскільки вузол $\varphi(u)$ є досяжним із вузла u, із ф-ли (5.3) маємо, що $\varphi(\varphi(u)).f \leq \varphi(u).f$. Крім того, з умови (5.2) маємо $\varphi(u).f \leq \varphi(\varphi(u)).f$. Тоді $\varphi(\varphi(u)).f = \varphi(u).f$ і $\varphi(\varphi(u)) = \varphi(u)$, оскільки два вузли з однаковим часом завершення опрацювання збігаються.

У кожній компоненті сильної зв'язності є один вузол, що є предком усіх вузлів цієї компоненти. Під час пошуку в глибину в графі G цей вузол буде визначено першим, а його опрацювання буде завершено останнім серед вузлів цієї компоненти. Під час пошуку у G^T він стає коренем дерева пошуку в глибину.

Для доведення цих властивостей зафіксуймо деяку компоненту S сильної зв'язності. Розгляньмо вузол ν цієї компоненти, який буде визначено першим під час пошуку в глибину, тобто буде мати мінімальне значення $\nu.d$ серед усіх $\nu \in S$. Розгляньмо ситуацію, яка виникає безпосередньо перед викликом DFS-VISIT(ν):

1. У цей момент усі вузли S є білими (інакше ν не є вузлом з S, який було визначено першим).

2. Усі вузли компоненти S є досяжними з ν білими шляхами (насправді, згідно з теоремою 5.12, ці шляхи не виходять за межі S).

3. Жоден сірий вузол не є досяжним із ν. У момент виклику DFS-VISIT(ν) сірі вузли утворюють шлях із кореня дерева пошуку до ν, тому ν є досяжним із будь-якого сірого вузла. Якщо б деякий сірий вузол був би досяжним із ν, він лежав би в одній компоненті з ν, проте всі такі вузли є білими.

4. Будь-який білий вузол w, який є досяжним із ν, є досяжним білим шляхом. На шляху з ν до w не може бути сірих вузлів, тому всі вузли на цьому шляху є білими або чорними. З іншого боку, під час пошуку в глибину ніколи не трапляються ребра, які б вели із чорного вузла в білий, отже, на цьому шляху не може бути чорних вузлів.

5. Усі вузли компоненти S будуть пофарбованими під час виклику DFS-VISIT(ν) і є нащадками ν в дереві пошуку. Таке положення випливає з теореми про білий шлях.

6. Усі вузли, які є досяжними з вузла ν, мають менший час опрацювання, ніж ν. Для чорних вузлів це є очевидним, оскільки вони вже є опрацьованими на момент початку опрацювання ν. Для білих це також є справедливим, оскільки вони будуть опрацьованими під час виклику DFS-VISIT(ν) до закінчення опрацювання ν.

7. Вузол ν є власним предком: $\varphi(\nu) = \nu$.

8. Вузол ν є предком будь-якого вузла u компоненти S. Із u є досяжними всі ті вузли, що й із ν, тому вузол із максимальним часом завершення опрацювання буде тим самим.

9. У кожній компоненті сильної зв'язності є вузол, який є предком усіх вузлів цієї компоненти, його було визначено першим, а опрацьовано останнім.

Теорема 5.13. В орієнтованому графі G = (V, E) предок φ(u) будь-якого вузла u $\in$ V є його предком у дереві пошуку в глибину.

Наслідок із теореми 5.13. Під час будь-якого пошуку в глибину на орієнтованому графі G = (V, E) вузол u та φ(u) розташовано в одній компоненті сильної зв'язності для будь-якого u $\in$ V.

Теорема 5.14. В орієнтованому графі G = (V, E) два вузли розташовано в одній компоненті сильної зв'язності тоді й тільки тоді, коли мають одного й того самого предка під час пошуку в глибину.

Із наведених викладок випливає, що завдання визначення компонент сильної зв'язності зведено до завдання визначення предків усіх вузлів графу. Саме із цією метою виконують пошук в глибину в рядку 3 алгоритму STRONGLY-CONNECTED-COMPONENTS.

Для пояснення цієї процедури розгляньмо вузол r із максимальним значенням r.f серед усіх вузлів графу G. Це ті значення, які розраховують у рядку 1 алгоритму. Цей вузол буде предком для будь-якого вузла, із якого він є досяжним (жоден вузол ν графу не має значення, більшого за значення ν.f). Отже, одну компоненту сильної зв'язності вже визначено – це вузли, із яких вузол r є досяжним. Інакше кажучи, це вузли, які є досяжними з вузла r у транспонованому графі.

Відкинувши всі вузли визначеної компоненти, виберімо серед тих, що залишилися, вузол r’ із максимальним значенням r’.f. Будь-який вузол u, який залишився і з якого вузол r’ є досяжним, буде мати предком вузол r’. Зауважмо, що жоден із відкинутих вузлів не є досяжним із u, інакше r був би досяжним із u й тоді вузол й потрапив би до відкинутих. Отже, визначено другий компонент сильної зв’язності, до якої входять ті з вузлів, що залишилися і за яких вузол r’ є досяжним. Інакше кажучи, це ті з вузлів, що залишилися і які є досяжними з вузла r’ у транспонованому графі.

Призначення рядка 3 алгоритму STRONGLY-CONNECTED-COMPONENTS: пошук в глибину в транспонованому графі, за допомогою якого відбувається почергове виявлення сильно пов’язаних компонент.

Теорема 5.15. Алгоритм STRONGLY-CONNECTED-COMPONENTS дійсно визначає компоненти сильної зв’язності орієнтованого графу.

У рядку 3 алгоритму відбувається виклик алгоритму DFS у транспонованому графі. Цей алгоритм розглядає вузли в порядку зменшення параметра v.f, що розраховують у рядку 1. Водночас будують дерева пошуку, які і є компонентами сильної зв’язності.

На кожному кроці циклу розглядають наступний (у порядку зменшення параметра f) вузол v. На цей час вузли, що входять до побудованих дерев пошуку, уже є чорними, а інші вузли, що залишилися, – білими, сірих вузлів на цей час немає. Зауважмо, що водночас будь-який вузол, що є досяжним у графі G^T із чорного вузла буде саме чорним.

Якщо черговий вузол u є чорним, то алгоритм DFS не робить нічого. Якщо ж він буде білим, то виклик процедури DFS-VISIT(u) у рядку 7 алгоритму DFS робить його та всі досяжні із нього в графі G^T вузли чорними (згідно з властивостями процедури DFS-VISIT). Слід довести, що ці вузли утворюють сильно пов’язану компоненту.

Якщо довільний білий вузол v є досяжним із вузла u в графі G^T, то u є предком v, оскільки u є досяжним із v в графі G^T. Жодні чорні вузли не є досяжними з v у G^T, і u має максимальне значення параметра f серед усіх білих вузлів. З іншого боку, якщо білий вузол v є не досяжним із u в графі G^T, то він не може мати u своїм предком. Чорні вузли також не можуть мати u предком (їхніх предків було визначено на попередніх кроках). Тому множина білих вузлів, які є досяжними з u в графі G^T, збігається з множиною вузлів, що мають u предком, тобто утворюють компоненту сильної зв’язності.

5.4. Алгоритм Дейкстри

Алгоритм Дейкстри (Dijkstra's algorithm) було розроблено 1959 р. нідерландським ученим Едсґером Вібе Дейкстрою для пошуку найкоротшого шляху від одного вузла графу до всіх інших.

Алгоритм є застосовним лише до графів, що не мають ребер із від'ємними значеннями ваг, який широко застосовують у програмуванні та технологіях, наприклад, для усунення кільцевих маршрутів його використовує протокол OSPF. Відомий також під назвою «Спочатку найкоротший шлях» (Shortest Path First).

Алгоритм Дейкстри виконує завдання пошуку найкоротших шляхів для виваженого орієнтованого графу G = (V, E) із вихідним вузлом s, коли ваги всіх ребер є невід'ємними ((u, v) >= 0 для всіх (u, v) $\rightarrow$ E). Його використання виправдано в разі, коли ребра графу мають різні значення ваг.

Формулювання завдання. Для графу G = (V, E), який має деякий вузол позначено як 1, необхідно визначити мінімальні шляхи від цього вузла до решти вузлів графу. Мінімальним шляхом є такий, що має найменшу суму цін або ваг ребер. Під ціною розуміють невід'ємне число, що є вагою ребра.

Ідея алгоритму. Нехай мінімальний шлях із вузла *a* до вузла *b* побудовано, до того ж вузол b пов'язано з певною кількістю вузлів *i*. Позначмо через c_i ціни шляхів із b до i та виберімо з них таке c_i, що є мінімальним. Тоді мінімальний шлях із вузла b має проходити через вибраний вузол i.

Це твердження є цілком логічним і не потребує доведення. До того ж із нього випливає таке: нехай є множина вузлів, через які вже проходять мінімальні шляхи, і вона гарантовано містить вузол 1. Сформульоване раніше твердження дозволяє додавати до вже наявної множини вузлів (далі будемо називати їх *виділеними* вузлами) ще один, а оскільки кількість вузлів графу є скінченною, то за обмежену кількість кроків усі вузли графу виявляться виділеними, а розв'язок визначеним.

Сутність алгоритму Дейкстри полягає в процедурі додавання ще одного вузла до множини вже виділених, і складається із двох кроків.

1. Формуймо множину вузлів, суміжних із виділеними, і визначаймо серед них вузол із найменшою ціною з подальшим додаванням до вже виділених.

2. Формуймо множину вузлів, суміжних із виділеними, і визначаймо для них нові ціни за мінімальними цінами шляхів від виділених вузлів до суміжних. Нову ціну будують за таким правилом:

а) для невиділеного вузла в множині виділених визначають підмножину вузлів, суміжних із заданою;

б) для кожного вузла виділеної підмножини визначають вартість шляху до заданого вузла;

в) визначають мінімальну вартість, що й призначають ціною вузла. Алгоритм працює із двома типами цін: ребер і вузлів.

Ціна ребра – це ціна переходу між з'єднаним ним вузлами. Ціни ребер є *незмінною* величиною.

Ціна вузла – це ціна мінімального шляху від вузла 1 до заданого. Ціни вузлів постійно перераховують.

Зауваження: перерахунку цін потребують тільки ті вузли, що пов'язані з вузлом, який останнім був доданим до множини виділених (на останньому кроці).

Ціни ребер, що відображають, наприклад, прокладання трубопроводу, шляху або проїзду, мають невід'ємні значення, і потрібно визначити найменшу вартість шляху з вузла 1 до вузла та для всіх i = 1, …, n за час $O(n^2)$. У процесі роботи алгоритму деякі вузли будуть виділеними (на початку – тільки вузол 1, до моменту завершення роботи алгоритму – усі). Для кожного виділеного вузла зберігають найменшу вартість шляху $1 \rightarrow i$; до того ж відомо, що мінімуму досягають на шляху, що проходить лише через виділені вершини.

Множину виділених вузлів розширюють, з огляду на таке правило: якщо серед усіх невиділених вузлів узяти такий, у якому збережена ціна є мінімальною, це значення дійсно є найменшою ціною шляху $1 \rightarrow i$. Справді, нехай є коротший шлях. Розгляньмо перший невиділений вузол на цьому шляху – уже до нього шлях є довшим! (Тут істотною є невід'ємність цін). Додавши вибраний вузол до виділених, маємо скоригувати інформацію, що зберігають для невиділених вузлів. Водночас достатньо враховувати лише шляхи, де новий вузол є останнім пунктом «пересадки», а це легко зробити, оскільки мінімальна вартість шляху до нового міста є вже відомою.

Інакше кажучи, кожному вузлу із множини V зіставимо у відповідність мітку – мінімальну відстань від цього вузла до початкового (1). Алгоритм працює покроково, до того ж на кожному кроці «відвідує» один вузол

і намагається зменшити мітку. Роботу алгоритму завершено, коли всі вузли відвідано.

Ініціалізація. Початковому вузлу 1 надано мітку 0, іншим вузлам – мітку нескінченності, оскільки відстані від початкового до інших вузлів поки що є невідомими. Усі вузли графу позначають як невідвідані.

Стислий опис алгоритму. Якщо всі вузли відвідано, алгоритм завершено; якщо ні, то з невідвіданих вузлів вибирають вузол u, що має мінімальну мітку, і розглядають шляхи, де u є передостаннім вузлом.

Вузли, з’єднані з вузлом u ребрами, назвімо його *сусідами*. Для кожного сусіда u розгляньмо нову довжину шляху, що дорівнює сумі поточної мітки u та довжини ребра, що з’єднує u із цим сусідом. Якщо визначена довжина є меншою від мітки сусіда, замінюймо мітку цією довжиною. Розглянувши всіх сусідів, помітьмо вузол u як відвіданий і повторімо крок.

Оскільки алгоритм Дейкстри щоразу вибирає вузли з найменшою вартістю шляху, можна сказати, що він належить до жадібних алгоритмів.

5.5. Алгоритм Флойда – Воршелла

Алгоритм Флойда – Воршелла (Robert Floyd, Stephen Warshall, 1962, Bernard Roy, 1959, тому ще одна назва Roy – Warshall algorithm) використовують для пошуку найкоротшого шляху між усіма парами вузлів у зваженому графі за $O(N^3)$.

В основі алгоритму лежить досить проста ідея. Позначмо через a [i] [j] [k] довжину найкоротшого шляху між вузлами i та j, що проходить через проміжні вузли {0,1, …, k} (i та j не враховують). Нехай слід поліпшити (скоротити) шлях між i та j за збільшення k на 1. Є два варіанти процедури перелічення шляхів за збільшення k:

1) не використовувати вузол k + 1, тоді a [i] [j][k + 1] = a [i] [j] [k];

2) використовувати вузол k + 1, тобто провести шлях через нього.

Стверджують, що найкоротший шлях із i до j за використання k + 1 можна досягти об’єднанням найкоротших шляхів $i \to k + 1$ та $k + 1 \to j$ (використовуючи проміжні вузли до k, оскільки k + 1 уже не є проміжним). Тоді:

$$a [i] [j] [k + 1] = a [i] [k + 1] [k] + a [k + 1] [j] [k].$$

Відразу спробуймо скоротити обсяг пам’яті, потрібний для роботи алгоритму. Зауважмо, що для скорочення шляху є потрібними лише попередні значення довжин шляхів для k − 1. Тому можна відкинути дані

останніх визначень (k) у масиві та взяти за поточну мінімальну довжину шляху із i до j, яку маємо скорочувати, a [i] [j].

Початкові значення для найкоротших шляхів визначмо таким чином:

найкоротший шлях кожного вузла в себе дорівнює нулю: a [i] [i] = 0;

найкоротший шлях між двома вузлами, безпосередньо зв'язаними ребром, дорівнює його довжині: a [i] [j] = w(i, j) (початковій вазі ребра);

найкоротший шлях між рештою пар вузлів дорівнює нескінченності: a [i] [j] = ∞.

Початкові значення не використовують проміжних вузлів, отже, починати роботу алгоритму потрібно з k = 0.

Під час виведення формули було зазначено, що проходження через один вузол двічі не скорочує шлях. Наведімо приклад (рис. 5.8), який доводить це припущення.

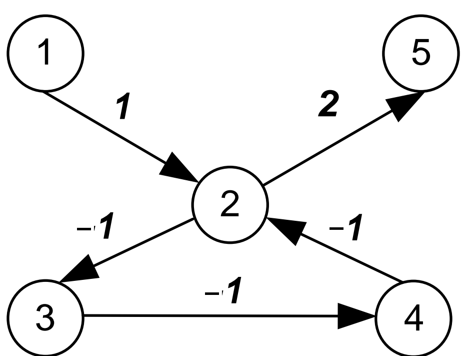


Рис. 5.8. **Приклад графу з від'ємним циклом**

Розгляньмо найкоротший шлях від вузла 1 до вузла 5:

шлях 1 → 2 → 5 має довжину 1 + 2 = 3;

шлях 1 → 2 → 3 → 4 → 2 → 5 має довжину 1 + (−1) + (−1) + (−1) + 2 = 0;

шлях 1 → 2 → 3 → 4 → 2 → 3 → 4 → 2 → 5 має довжину 1 + (−1) + + (−1) + (−1) +(−1) + (−1) + (−1) + 2 = −3.

Отже, проходячи цей від'ємний цикл нескінченну кількість разів можна зменшити довжину шляху до −∞. Наведений граф не має найкоротших шляхів.

Є два випадки, за яких алгоритм Флойда – Воршелла не працює:

1) граф містить від'ємні цикли (найкоротших шляхів із вузла i до вузла j немає);

2) граф не містить ребра i → j (шляхів із вузла i до вузла j немає), тому елемент a_{ij} = ∞.

Алгоритм Флойда – Воршелла визначає найкоротший шлях від кожного вузла графу до кожного, його складність становить n^3. Складність алгоритму Дейкстри становить n^2, проте цей алгоритм визначає найкоротший шлях від одного вузла (джерела) до всіх інших. Тому щоб визначити довжини всіх найкоротших шляхів кожного вузла з кожним, алгоритм Дейкстри доведеться виконати n разів, а це буде дорівнювати складності n^3.

Контрольні запитання і завдання для самоперевірки

1. Які способи подання графів ви знаєте? Які переваги й недоліки кожного з них?
2. У чому полягає сутність організації пошуку в ширину?
3. Наведіть приклади алгоритмів пошуку в ширину.
4. Що мають на увазі під білими, сірими та чорними вузлами в алгоритмах пошуку в ширину?
5. Опишіть сутність процедури BFS.
6. Які показники оцінюють під час аналізу алгоритму пошуку в ширину?
7. Які властивості довжин шляхів використовують для пошуку найкоротшого шляху між вузлами?
8. Дайте визначення дерева пошуку в ширину.
9. Опишіть сутність пошуку в глибину?
10. Які мітки надають вузлам під час пошуку в глибину?
11. Опишіть процедуру DFS.
12. Які властивості пошуку в глибину ви знаєте?
13. Які типи ребер ви знаєте? Наведіть приклади кожного з типів.
14. У чому полягає завдання топологічного сортування? Наведіть приклад завдання такого роду.
15. У чому полягає завдання розкладання графу на сильно пов’язані компоненти?
16. Опишіть алгоритм пошуку сильно пов’язаних компонент.

Лабораторна робота 4
Алгоритми Дейкстри та Флойда – Воршелла

Мета. Ознайомитися з основними підходами до побудови найкоротших шляхів у зважених графах, набути навичок у побудові найкоротших шляхів за алгоритмами Дейкстри та Флойда – Воршелла.

Рекомендації з підготовки до виконання. Для успішного виконання лабораторної роботи студент має знати мету, порядок виконання роботи та загальні теоретичні положення; уміти розв’язувати задачі пошуку найкоротших шляхів на графах із застосуванням алгоритмів Дейкстри та Флойда – Воршелла.

Завдання до виконання. Вигляд можливої схеми маршрутизації інформаційних пакетів від користувача до сервера показано на рис. 5.9. Ваги ребер визначають середній час пересилання інформаційного пакета.

1. Мінімізуйте середній час відповіді.

2. Ускладніть наведену схему, додавши ще три проміжні вузли, ваги ребер задайте на власний розсуд та визначте мінімальний середній час відповіді.

3. Побудуйте блок-схему алгоритму Дейкстри.

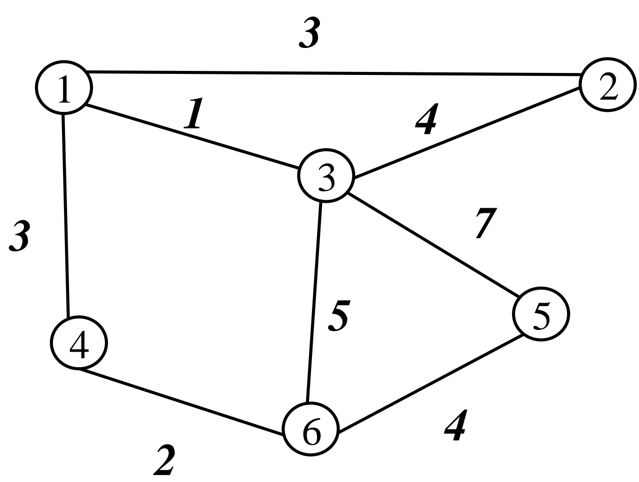


Рис. 5.9. **Граф-схема маршрутизації**

Загальні теоретичні положення

Опишімо схему роботи алгоритму Дейкстри в деталях.

Алгоритм Дейкстри використовує три масиви розмірності N (кількість вузлів графу):

1) масив A містить мітки із двома значеннями: 0 (вузол ще не розглядали) і 1 (вузол вже розглянуто);

2) масив B містить поточні найкоротші відстані від вузла 1 до кожного вузла;

3) масив C містить номери вузлів – k-й елемент C[k] є номером передостаннього вузла на поточному найкоротшому шляху з початкового вузла до k.

Матриця відстаней D[i, k] задає довжини ребер D[i, k]; якщо ребра немає, то D[i, k] надають велике число Б, що дорівнює «машинній нескінченності» або ∞.

Роботу алгоритму можна розподілити на три етапи.

Етап 1 (ініціалізація). У циклі від 1 до N заповніть масив A нулями; заповніть масив C числами i; перенесіть i-й рядок матриці D до масиву B, A[i] : = 1; C[i] : = 0 (i – номер стартового вузла).

Етап 2 (виконання алгоритму). Визначте мінімум серед непозначених вузлів (тобто тих k, для яких A[k] = 0); нехай мінімуму досягають на вузлі з індексом j, тобто B[j] <= B[k]. Далі виконують такі операції: A[j] : = 1; якщо B[k] > B[j] + D[j, k], то (B[k] : = B[j] + D[j, k]; C[k] : = j). (Умова означає, що шлях $i \rightarrow k$ довший, ніж шлях $i \rightarrow j \rightarrow k$.) Якщо всі A[k] позначено, то довжина шляху від вузла i до вузла k дорівнює B[k]. Далі слід визначити вузли, що входять до найкоротшого шляху.

Етап 3 (виведення відповіді). Шлях від вузла i до вузла k виводять у зворотному порядку такою процедурою:

1. z : = C[k];
2. Видати z;
3. z : = C[z]. Якщо z = 0, то кінець, інакше перейти до 2.

На виконання алгоритму потрібно N раз переглянути масив B з N елементів, тобто алгоритм Дейкстри має квадратичну складність $O(n^2)$.

На початку поданого алгоритму відстань для початкового вузла дорівнює нулю, проте інші відстані заповнюють великим позитивним числом (більшим за максимально можливий шлях у графі). Масив прапорців заповнюють нулями та запускають основний цикл.

На кожному кроці циклу шукаймо вузол із мінімальною відстанню та прапорцем, що дорівнює нулю. Потім установлюймо прапорець в 1 і перевіряймо всі сусідні вузли. Якщо відстань є більшою, ніж сума відстані до поточного вузла та довжини ребра, зменшуймо її. Цикл завершують, коли прапорці всіх вузлів дорівнюють 1.

Алгоритм Флойда – Воршелла будують на основі розрахування чисел a_{ijk} – довжини найкоротшого шляху з вузла i до вузла j з використанням вузлів від 1 до k. Наприклад, запис a_{235} означає найкоротший шлях із вузла 2 до вузла 3 з можливими проміжними вузлами 1, …, 5.

Матриця $A^1 = \{a_{ij1}\}_{n \times n}$ – матриця найкоротших шляхів від кожного вузла до кожного з можливістю відвідання вузла 1. Розмірність матриці $n \times n$, де n – кількість вузлів графу, тобто його потужність |V|.

Матриця $A^2 = \{a_{ij2}\}_{n \times n}$ – матриця найкоротших шляхів від кожного вузла до кожного з можливістю відвідання вузлів 1 та 2.

Тоді $A^0 = \{a_{ij0}\}_{n \times n}$ – матриця найкоротших шляхів від кожного вузла до кожного, тобто матриця суміжності графу, елементи якої a_{ij0} дорівнюють довжинам ребер між вузлами i та j, якщо вони є, або нескінченності (∞), якщо немає.

За викладеною логікою кінцевий результат роботи алгоритму (відповідь) має міститися в матриці $A^n = \{a_{ijn}\}_{n \times n}$.

Розгляньмо проміжний k-й крок алгоритму (рис. 5.10) – розрахування елемента a_{ijk}, за якого можливі два варіанти:

1) найкоротший шлях $i \to j$ не проходить через вузол *k*, тоді всі вузли, через які він буде проходити, будуть мати номер, не більший за k − 1, а елемент a_{ijk} залишиться без змін, тому що $a_{ijk} = a_{ijk-1}$;

2) найкоротший шлях $i \to j$ проходить через вузол *k*, тоді його можна умовно розподілити на дві частини $a_{ijk} = a_{ikk-1} + a_{kjk-1}$ (третій індекс у цій формулі k − 1, оскільки нема сенсу відвідувати вузол k двічі).

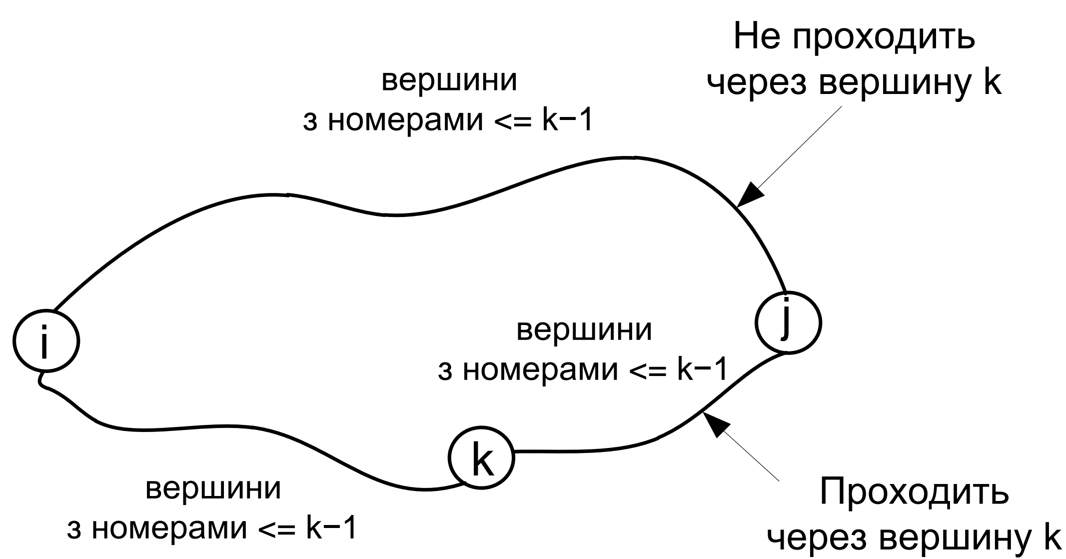


Рис. 5.10. **Вибір найкоротшого шляху**

Постає питання, який із двох варіантів вибрати? Оскільки визначаємо найкоротший шлях, то той, що є мінімальним: $a_{ijk} = \min\{a_{ijk-1}, a_{ikk-1} + a_{kjk-1}\}$.

Із наведеної формули випливає, що значення елементів k-ї матриці залежать лише від значень елементів попередньої (k − 1)-ї матриці, отже,

визначено формулу динамічного програмування, що належить до класу жадібних алгоритмів.

Зауваження. На відміну від алгоритму Дейкстри, алгоритм Флойда – Воршелла працює з від'ємними вагами ребер.

Порядок виконання роботи

Завдання 5.1. Необхідно на неорієнтованому графі (рис. 5.11) визначити найкоротший шлях від вузла 1 до всіх інших із застосуванням методу Дейкстри.

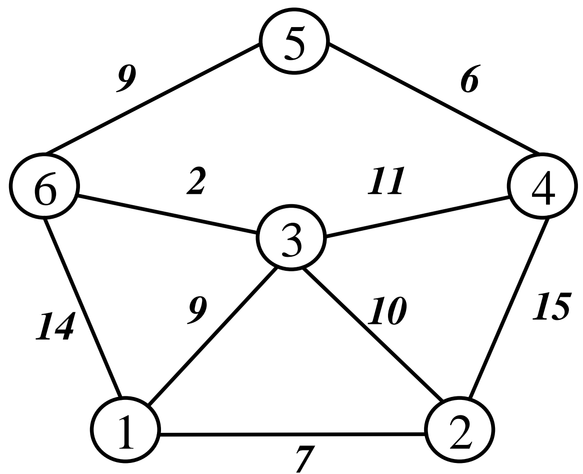


Рис. 5.11. **Первинний граф**

Виконання завдання 5.1. Вигляд графу після ініціалізації показано на рис. 5.12.

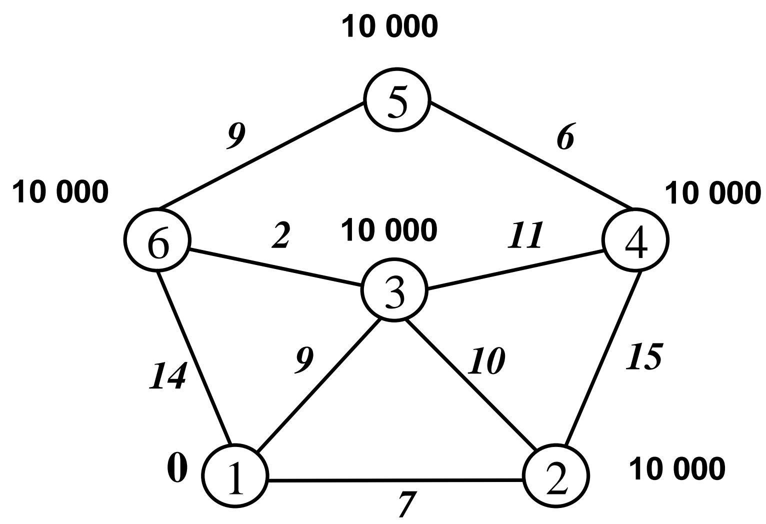


Рис. 5.12. **Розмічання графу після ініціалізації**

Значенню стартового вузла 1 надамо мітку (позначку) 0, іншим вузлам – мітки, значення яких є значно більшими за всі можливі ваги ребер в цьому графі (в ідеалі – нескінченність). Усі вузли графу позначають як невідвідані.

Крок 1. Мінімальну мітку має вузол 1. Його сусідами є вузли 2, 3 та 6. Обходьмо сусідів вузла 1 по черзі.

Оскільки шлях до вузла 2 є найкоротшим, першим розглядаймо його. Довжина шляху через вузол 1 дорівнює сумі найкоротшої відстані до вузла 1 (мітки вузла 1) і довжини ребра між вузлами 1 та 2, тобто 0 + 7 = 7. Це менше за поточну мітку вузла 2 (10 000), отже, нова мітка 2-го вузла дорівнює 7.

Аналогічно визначмо довжину шляхів між вузлами 1 – 3 та 1 – 6. Після чого мінімальна відстань від вузла 1 до вузла 1, визначена поточним значенням його мітки, є остаточною й перегляду на підлягає, а сам вузол 1 уважають переглянутим та позначеним (рис. 5.13).

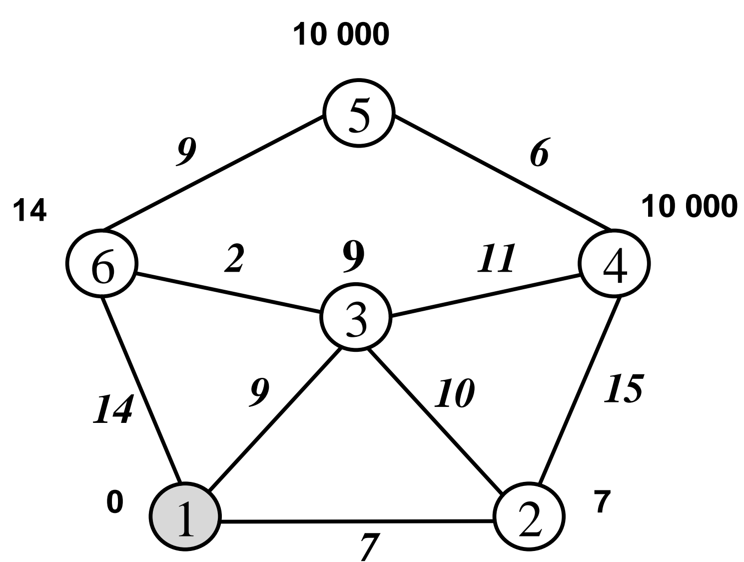


Рис. 5.13. **Розмічання графу після завершення першого кроку**

Крок 2. Розгляньмо вузол 2, що є найближчим сусідом вузла 1, і повторімо для нього дії, що виконували з вузлом 1 на першому кроці, намагаючись знизити вартість шляхів (мітки вузлів) від вузла 1 до сусідів вузла 2.

Вузол 1 уже є відвіданим, вузол 3, наступний сусід вузла 2, має найменшу мітку з усіх вузлів, позначених як невідвідані. Якщо йти до нього

через 2, довжина шляху буде становити 17 (7 + 10 = 17), але поточна мітка третього вузла дорівнює 9, а 9 < 17, тому мітку не змінюють.

Наступний сусід 2 – вузол 4. Якщо йти до нього через 2, то довжина шляху буде становити 22 (7 + 15 = 22). Оскільки 22 < 10 000, змінюймо мітку вузла 4 на 22. Усі сусіди вузла 2 переглянуто, отже, позначаймо його як відвіданий (рис. 5.14).

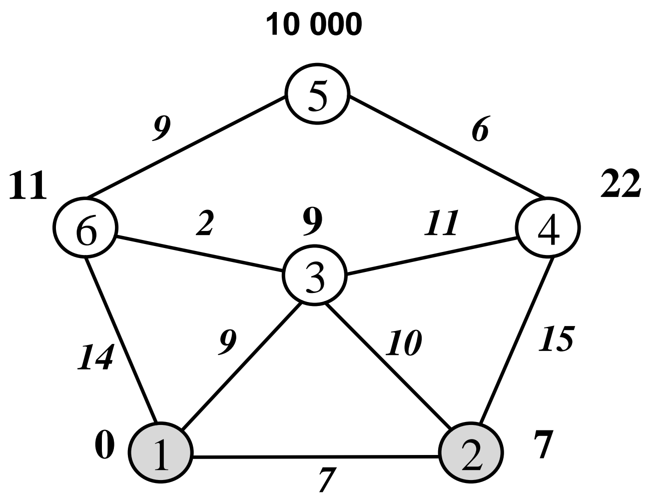


Рис. 5.14. **Розмічання графу після завершення другого кроку**

Аналогічно виконуймо наступні кроки алгоритму (рис. 5.15 – 5.18).

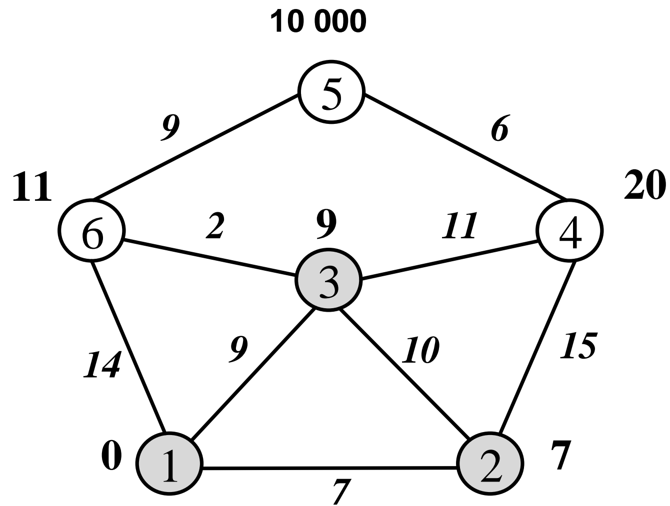


Рис. 5.15. **Розмічання графу після завершення третього кроку**

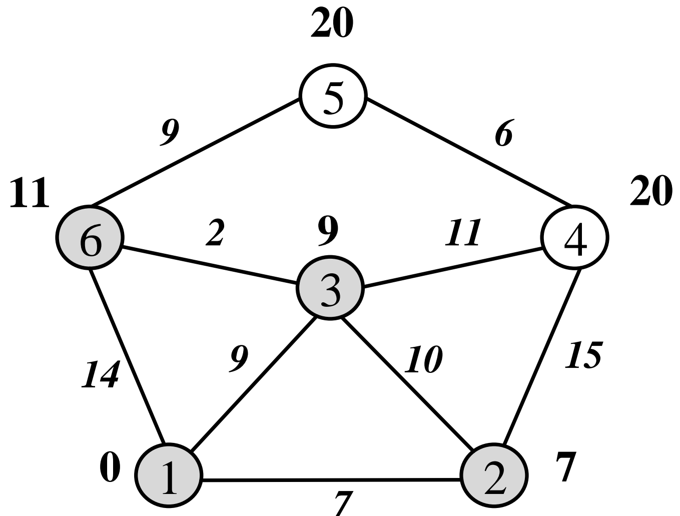


Рис. 5.16. **Розмічання графу після завершення четвертого кроку**

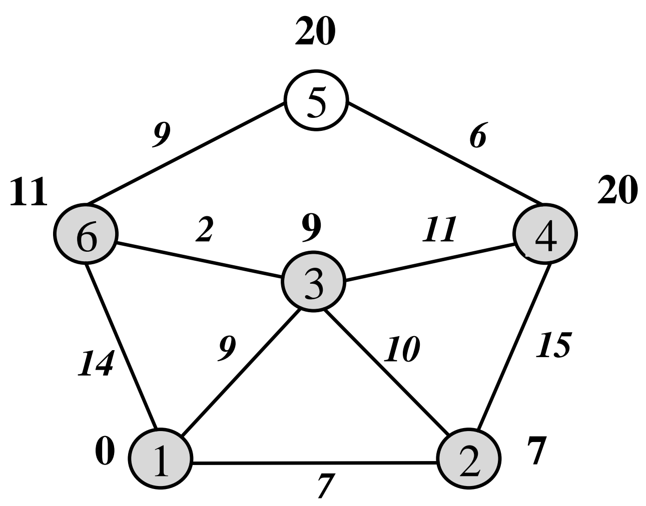


Рис. 5.17. **Розмічання графу після завершення п’ятого кроку**

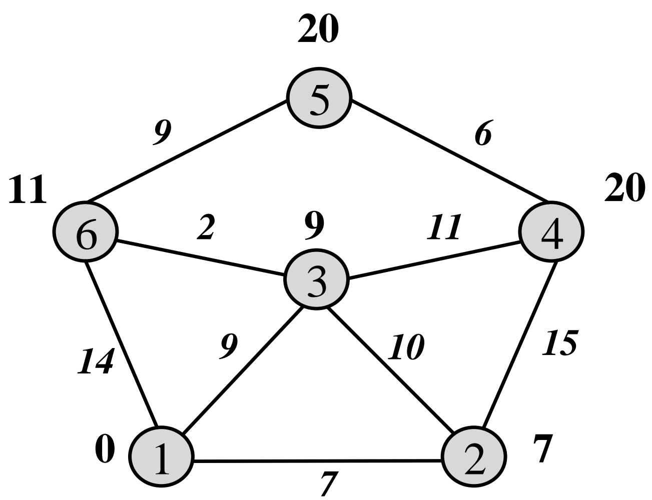


Рис. 5.18. **Розмічання графу після завершення шостого кроку**

Отже, найкоротшим шляхом між вузлами 1 – 5 є шлях через вузли 1 – 3 – 6 – 5, оскільки його вартість, або вага, є мінімальною і дорівнює 20.

Перейдімо до етапу виведення вузлів, що становлять найкоротший шлях. Відомі довжини шляху для кожного вузла й тепер будемо розглядати вузли з кінця. Розглядаймо кінцевий вузол (у цьому разі вузол 5), і для всіх вузлів, із якими його пов'язано ребрами, визначмо довжину шляху як різницю між значенням ваги відповідного ребра та довжиною шляху кінцевого вузла.

Так, вузол 5 має довжину шляху 20 і пов'язаний ребрами з вузлами 6 та 4. Для вузла 6 визначмо вагу 20 − 9 = 11 (значення збігається зі значенням мітки вузла 6). Для вузла 4 визначмо вагу 20 − 6 = 14 (значення не збігається зі значенням мітки вузла 4).

Якщо визначене значення збігається з довжиною шляху вузла (у цьому разі – вузла 6), то саме через нього здійснено перехід у кінцевий вузол. Позначаймо цей вузол як такий, що належить найкоротшому шляху між вузлами 1 – 5.

Повторюймо наведені раніше дії з вузлом 6 тощо, поки не опинімося на початку, тобто у вузлі 1. Якщо на якомусь кроці обходу значення декількох вузлів збігаються, це означає, що декілька шляхів будуть мати однакову довжину та можна вибрати будь-який із них.

Завдання 5.2. Визначте найкоротші шляхи від стартового вузла 1 до всіх вузлів графу (рис. 5.19) за допомогою методу Дейкстри.

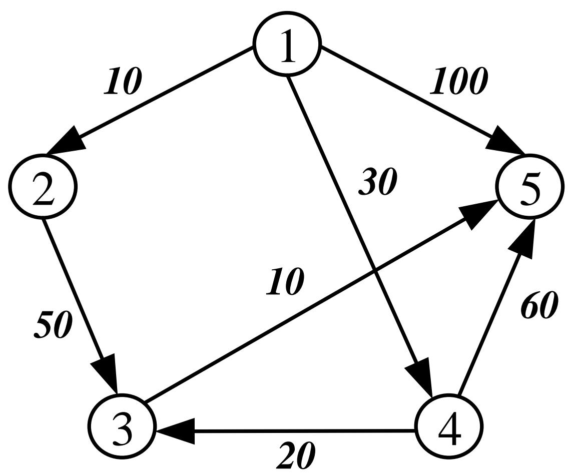


Рис. 5.19. **Первинний орієнтований граф**

Виконання завдання 5.2. Алгоритм Дейкстри формує множину вузлів S, для яких найкоротший шлях від стартового вузла вже є відомим. На кожному кроці до множини S додають той вузол, що залишився нерозглянутим і відстань до якого від вузла 1 є найменшою. Можна стверджувати, що найкоротший шлях між вузлами проходить лише через вузли множини S і має назву *особливого* шляху.

Алгоритм Дейкстри на кожному кроці використовує масив D, куди заносять довжини особливих шляхів вузлів. Коли множина S буде містити всі вузли орграфу, оскільки для них буде визначено особливі шляхи, масив D буде містити довжини найкоротших шляхів від вузла 1 до кожного з вузлів графу.

У початковий момент роботи алгоритму множина S містить лише стартовий вузол – вузол 1, а масив D – значення початкових ваг ребер між вузлом 1 і вузлами з номерами, що відповідають номерам елементів масиву (табл. 5.1). У нашому випадку граф не містить ребра 1 – 3, тому $D[3] = \infty$.

Таблиця 5.1

Уміст масиву D на етапі ініціалізації

Ітерація	S	w	D[2]	D[3]	D[4]	D[5]
Початок	{1}	–	10	∞	30	100

Крок 1. Вибираймо мінімальний елемент масиву D (D[2] = 10). Мінімальне значення відповідає вузлу 2, тому додаймо його до множини S. Позначаймо вузол 2 як вузол w та перераховуймо довжини найкоротших шляхів за такою формулою:

$$D[v] = \min\{D[v], D[w] + C[w, v]\}. \quad (5.4)$$

Тобто у відповідну комірку масиву записано найменше із двох значень: попереднього або суми значень елемента з номером w (на цьому кроці, це D[2] = 10) і ваги ребра між w та вузлом, що розглядають (табл. 5.2).

Таблиця 5.2

Уміст масиву D на першому кроці

Ітерація	S	w	D[2]	D[3]	D[4]	D[5]
Початок	{1}	–	10	∞	30	100
1	{1, 2}	2	–	60	30	100

Обчислення, результати яких занесено до табл. 5.2:

D[3] = min{D [3], D [2] + C [2, 3]} = min{∞, 10 + 50} = min{∞, 60} = 60;

D [4] = min{D[4], D[2] + C[2, 4]} = min{30, 10 + ∞} = min{30, ∞} = 30;

D[5] = min{D[5], D[2] + C[2, 5]} = min{100, 10 + ∞} = min{100, ∞} = 100.

Крок 2. Вибираймо мінімальне значення серед елементів масиву D (D[4] = 30).

Зауваження. До розгляду беруть лише ті вузли, що не входять до множини S, отже, вузол 2 на цьому кроці вже не розглядають.

Мінімальне значення відповідає вузлу 4, його й додаймо до множини S. Позначаймо вузол 4 як w, перераховуймо довжини найкоротших шляхів для вузлів, що не належать множині S за ф-лою (5.4) та заносьмо результати в табл. 5.3.

Таблиця 5.3

Уміст масиву D на другому кроці

Ітерація	S	w	D[2]	D[3]	D[4]	D[5]
Початок	{1}	–	10	∞	30	100
1	{1, 2}	2	–	60	30	100
2	{1, 2, 4}	4	–	50	–	90

D[3] = min{D[3], D[4] + C[4, 3]} = min{60, 30 + 20} = min{60, 50} = 50;

D[5] = min{D[5], D[4] + C[4, 5]} = min{100, 30 + 60} = min{100,90} = 90.

Крок 3. Вибираймо мінімальний серед елементів масиву D (D[3] = 50). Мінімальне значення відповідає вузлу 3, отже, додаймо його до множини S та позначаймо вузол 3 як w. Перераховуймо довжини найкоротших шляхів для вузлів, що не належать множині S за ф-лою (5.4) та заносьмо результати в табл. 5.4.

Таблиця 5.4

Уміст масиву D на третьому кроці

Ітерація	S	w	D[2]	D[3]	D[4]	D[5]
Початок	{1}	–	10	∞	30	100
1	{1, 2}	2	–	60	30	100
2	{1, 2, 4}	4	–	50	–	90
3	{1, 2, 4, 3}	3	–	–	–	60

$$D[5] = \min\{D[5], D[3] + C[3, 5]\} = \min\{90, 50 + 10\} = \min\{90, 60\} = 60.$$

Крок 4. Залишився останній вузол (5), додаймо його до множини S. Отже, усі вузли є переглянутими, а остаточно вміст масиву D наведено в табл. 5.5, останній рядок якої містить довжини найкоротших особливих шляхів від джерела (вузла 1) до кожного з вузлів графу.

Таблиця 5.5

Уміст масиву D на четвертому кроці

Ітерація	S	w	D[2]	D[3]	D[4]	D[5]
Початок	{1}	–	10	∞	30	100
1	{1, 2}	2	–	60	30	100
2	{1, 2, 4}	4	–	50	–	90
3	{1, 2, 4, 3}	3	–	–	–	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

Далі наведено лістинг псевдокоду виконання завдання, у якому підкреслено команди ініціалізації та формування масиву P, що містить вузли, які становлять найкоротший шлях.

```
1    procedure Dejkstra
2    begin
3        S := {1};
4        for i := 2 to n do
5            D[i] := C[1, i];          // ініціалізація масиву D
6            P[i] := 1;                // ініціалізація масиву P
7        end;
8        for i := 1 to n – 1 do begin
9            вибір із підмножини ще не розглянутих вузлів V \ S такого вузла w, у якого значення D[w] є мінімальним;
10           додавання вузла w до підмножини S;
11       end;
12       for кожного вузла v із підмножини V \ S do
13           D[v] := min(D[v], D[w] + C[w, v])
14           if D[w] + C[w, v] < D[v] then P[v] := w;
15       end;
16   end;
```

У табл. 5.6 подано масив P для наведеного в завданні графу.

Таблиця 5.6

Масив вузлів найкоротшого шляху

P[2]	P[3]	P[4]	P[5]
1	4	1	3

Якщо потрібно визначити шлях від 1 до 5 вузла, слід рухатися у зворотному напрямку, починаючи з кінця, вузла 5. Елемент P[5] містить значення 3, яке означає, що до вузла 5 потрапили з вузла 3. Елемент P[3] містить значення 4 – до вузла 3 потрапили з вузла 4; елемент P[4] містить

значення 1 – до вузла 4 потрапили з вузла 1. Отже, визначено зворотний шлях 5 → 3 → 4 → 1. Часова складність алгоритму Дейкстри становить:

$O(|V|^2)$ за подання графу матрицею суміжності, тобто визначено квадратом потужності графу (кількості вузлів) за О-нотацією;

$O(|E| \times Log|V|)$ за подання графу списками суміжності, тобто залежить від добутку кількості ребер графу та логарифма кількості його вузлів.

Завдання 5.3. Визначте найкоротші шляхи в графі на рис. 5.20 від кожного вузла до кожного.

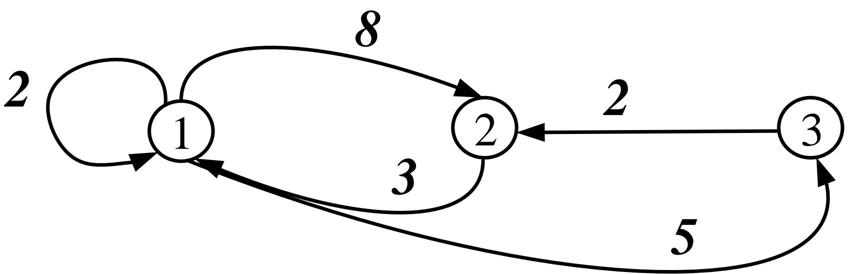


Рис. 5.20. **Первинний зважений граф**

Виконання завдання 5.3. Ребра графу (див. рис. 5.20) мають напрям, тому граф є орієнтованим. Для кожного ребра надано вагу або ціну, отже, граф є зваженим. Він також має петлю з вузла 1.

Матриця суміжності $A^0 = \{a_{ij0}\}_{n \times n}$ в цьому разі буде мати такий вигляд:

$A^0 =$

	1	*2*	*3*
1	2	8	5
2	3	0	∞
3	∞	2	0

Після *першої* ітерації (для **k = 1**) маємо наступну матрицю:

$A^1 =$

	1	*2*	*3*
1	2	8	5
2	3	0	8
3	∞	2	0

Перерахунок елементів такий (визначення елементів a_{1i1} наведено для наочності роботи алгоритму з петлями):

a_{111} = min{2, 2} = 2, a_{121} = min{2, 2 + 8} = 2, a_{131} = min{5, 5 + 2} = 5,

a_{211} = min{3, 3} = 3, a_{221} = min{0, 3 + 8} = 0, **a_{231} = min{∞, 3 + 5} = 8.**

Для елемента a_{211} значення можна не вираховувати, оскільки граф містить лише одне ребро (2, 1) вагою 3. Рядок для вузла 3 також не розглядаймо, оскільки граф не містить ребра (3, 1), а тому перерахунок будь-якого шляху від вузла 3 через вузол 1 буде містити доданок ∞.

Отже, змінилося значення лише одного елемента a_{23}.

Після *другої* ітерації (для **k = 2**) маємо наступну матрицю:

A^2 =

	1	*2*	*3*
1	**2**	**8**	**5**
2	**3**	**0**	**8**
3	**5**	**2**	**0**

Наведімо розрахунки для елемента, значення якого змінилося: a_{312} = min{∞, 2 + 3} = 5.

Після *третьої* ітерації (для **k = 3**) маємо наступну матрицю:

A^3 =

	1	*2*	*3*
1	**2**	**7**	**5**
2	**3**	**0**	**8**
3	**5**	**2**	**0**

Наведімо розрахунки для елемента, значення якого змінилося: a_{123} = min{8, 5 + 2} = 7.

Остання матриця містить найкоротші шляхи від кожного вузла до кожного. Кількість ітерацій дорівнює кількості вузлів графу. Далі наведімо псевдокод програмної реалізації цього алгоритму.

```
1 procedure Floyd (var A : array[1… n, 1… n] of real;
                   var C : array[1… n, 1… n] of real;
                   var P : array[1… n, 1… n] of integer;)
     var i, j, k: integer;
2    begin
```

```
3                    for i := 1 to n do
4                          for j := 1 to n do
5                                A[i, j] := A[i, j];
6                    for k := 1 to n do
7                          for i := 1 to n do
8                                for j := 1 to n do
9                                      if A[i, k] + A[k, j] < A[i, j] then begin
10                                           A[i, j] := A[i, k] + A[k, j];
11                                           P[i, j] := k
12                                     end;
13    end;
```

Підкресленням виокремлено команди, які формують матрицю вузлів найкоротшого шляху. Для наведеного прикладу ця матриця має такий вигляд:

P =

	1	*2*	*3*
1	0	3	0
2	0	0	1
3	2	0	0

Визначмо вузли, через які слід пройти, щоб дістатися найкоротшим шляхом від вузла 1 до вузла 2 у зворотному порядку:

у стовпчику 2 міститься 3, тобто до вузла 2 потрапляймо з вузла 3 ($3 \rightarrow 2$);

у стовпчику 3 міститься 1, тобто до вузла 3 потрапляймо з вузла 1 ($1 \rightarrow 3 \rightarrow 2$).

Далі наведено псевдокод, який дозволяє виводити вузли найкоротших шляхів.

```
1  procedure path(i, j: integer);
            var k: integer;
2     begin
3     k := P[i, j];
4           if k = 0 then return;
5     path(i, k);
```

```
6    writeln(k);
7    path(k, j);
8    end
```

Контрольні запитання до лабораторної роботи 4

1. У чому полягає сутність алгоритму Дейкстри?
2. Опишіть ідею, покладену в основу алгоритму Дейкстри.
3. У чому полягає ініціалізація за застосування алгоритму Дейкстри? Яку функцію вона виконує в загальному випадку?
4. До якого класу має сенс зарахувати алгоритм Дейкстри? Аргументуйте відповідь.
5. Із яких етапів складається алгоритм Дейкстри? Охарактеризуйте кожен із них.
6. У чому полягає сутність алгоритму Флойда – Воршелла?
7. Опишіть ідею, покладену в основу алгоритму Флойда – Воршелла.
8. Із яких етапів складається алгоритм Флойда – Воршелла? Охарактеризуйте кожен із них.
9. Порівняйте алгоритми Дейкстри та Флойда – Воршелла.
10. Які умови накладено на граф, щоб застосування до нього алгоритму Флойда – Воршелла мало сенс?

6. Потоки в мережах

6.1. Поняття мережі. Основні визначення

Мережа – це структура даних, яка складається із множини вузлів та множини дуг, які з'єднують вузли між собою. У мережі зв'язки між вершинами (дуги) можуть мати напрямок, така мережа є *орієнтованою*, або не мати напрямку, тоді мережа є *неорієнтованою*, а зв'язки між вузлами мають назву *ребер*. Крім того, зв'язки можуть мати *вагу*, яка відображає важливість або відстань між вершинами.

Мережі використовують для моделювання різноманітних об'єктів та взаємодії між ними, наприклад:

мережі доріг, залізниць та авіаліній для планування маршрутів;

мережі соціальних зв'язків для аналізу взаємодії між користувачами соціальних мереж;

мережі комп'ютерних систем для побудови структури мережі та оптимізації шляхів передавання даних;

мережі взаємодії біомолекул для дослідження біологічних процесів.

У програмуванні мережі може бути подано за допомогою списку суміжності або матриці суміжності. Список суміжності містить інформацію про кожен вузол і його зв'язки з іншими вершинами, тоді як матриця суміжності є квадратною матрицею, яка містить 1, якщо між двома вершинами є зв'язок, і 0, інакше, в разі зваженої мережі замість 1 на відповідних місцях розташовано значення ваг дуг.

На рис. 6.1 подано мережу, яка складається із чотирьох вузлів та шести орієнтованих дуг.

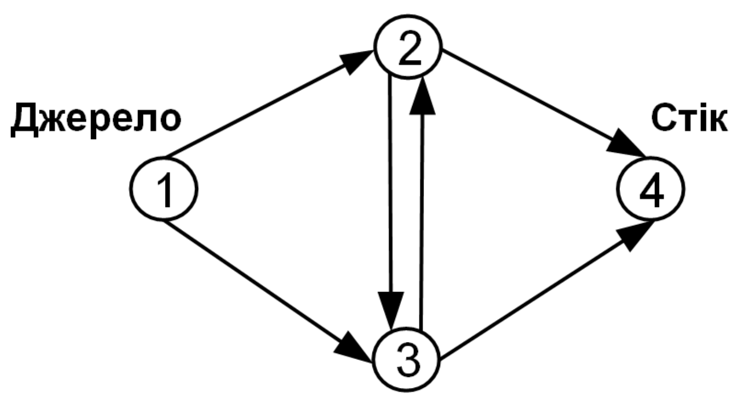


Рис. 6.1. **Мережа з орієнтованими дугами**

Позначмо через N_i i-й вузол, а через A_{ij} – дугу, яка з'єднує вузол N_i та N_j. Якщо орієнтацію в мережі не задано і вузли з'єднано ребрами, то їх можна позначати або A_{ij}, або A_{ji}, незважаючи на порядок індексів.

Мережа є *зв'язною,* якщо за будь-якого розподілу множини вузлів на підмножини X та X' буде дуга A_{ij}, для якої $N_i \in X$ і $N_j \in X'$. Тобто мережа залишається зв'язною, поки є шлях між будь-якими двома вершинами. Два вузли є сусідніми вузлами, якщо їх поєднано дугою.

Набір вузлів і дуг мережі $N_1, A_{12}, N_2, A_{23}, N_3,\ldots, N_{k-1}, A_{k-1\,k}, N_k$ з початком у вузлі N_1, а закінченням – у N_k, є *ланцюгом* або *орієнтованим ланцюгом.* Якщо $N_1 = N_k$, такий набір дуг і вузлів є *орієнтованим циклом.* Якщо ланцюг не має циклів, то такий ланцюг є *простим*.

Так, показаний на рис. 6.1 набір N_1, A_{12}, N_2, A_{24}, N_4 є ланцюгом. Набір N_1, A_{12}, N_2, A_{23}, N_3, A_{34}, N_4 так само є ланцюгом. Набір N_2, A_{23}, N_3, A_{32}, N_2 є *циклом*.

Крім ланцюгів, у мережах виокремлюють *шляхи*. На відміну від ланцюгів, шляхами можна рухатися в напрямку, протилежному до орієнтації дуг. Відповідно, у неорієнтованих мережах поняття шляху та ланцюга збігаються. Набір N_3, A_{31}, N_1, A_{12}, N_2 є шляхом (див. рис. 6.1).

Виокремлюють два специфічні вузли мережі: *джерело* **(source)**, яке позначають N_s або просто s, та *стік* **(flow)**, який позначають N_f або просто f. У мережах дугам можуть відповідати додатні числа x_{ij}, які означають *пропускну спроможність дуги*.

6.2. Задача про максимальний потік у мережі

Перш ніж перейти до розгляду задачі про максимальний потік, уведімо поняття мережевого потоку. *Потоком* із джерела мережі до її стоку є множина невід'ємних чисел, поставлених у відповідність певним дугам. Такі числа мають задовольняти такі обмеження:

$$\sum_i x_{ij} - \sum_k x_{ik} = \begin{cases} -v, \text{ якщо } j = s, \\ 0, \text{ якщо } j \neq s, f, \\ v, \text{ якщо } j = f, \end{cases} \tag{6.1}$$

$$\nu \geq 0, \tag{6.2}$$

$$0 \leq x_{ij} \leq b_{ij}, \text{ для всіх } A_{ij}.$$

У ф-лі (6.1) першу суму розраховують за дугами, які ведуть до N_j, а другу – за дугами, які виходять з N_j. За виконання наведених умов число ν є *величиною потоку,* а число x_{ij} – потоком дуги A_{ij}.

Обмеження (6.1) гарантує умову збереження потоків, яка полягає в тому, що в кожний вузол (окрім стоку і джерела) має втікати стільки, скільки з нього витікає. Графічне подання цієї умови проілюстровано на рис. 6.2. Обмеження (6.2) означає, що потік по дузі є обмеженим її пропускною спроможністю b_{ij}.

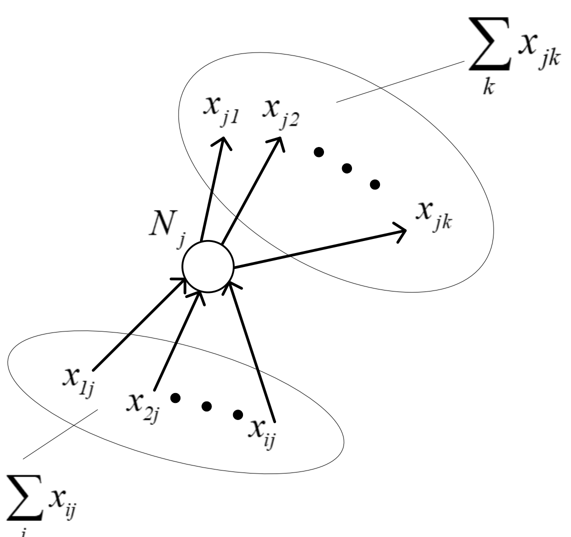


Рис. 6.2. **Умова збереження потоків у вузлі**

Математичне формулювання задачі пошуку оптимальних потоків мережі може бути подано такою цільовою функцією:

$$V = \sum_j x_{sj} \to \max, \tag{6.3}$$

за умови виконання обмежень (6.1), (6.2). У такому формулюванні задача пошуку оптимальних потоків мережі належить до класу лінійних умовних оптимізаційних задач на максимум.

6.2.1. Теореми про оптимальні потоки в мережах

Якщо мережа є ланцюгом N_s, A_{s2}, N_2, A_{23}, N_3, …, N_{f-1}, $A_{f-1\ f}$, N_k із джерелом N_s та стоком N_f, то максимальний потік, який можна пропустити цією мережею, є обмеженим найменшою із пропускних спроможностей дуг мережі. Дуга з найменшою пропускною спроможністю є вузьким місцем мережі.

Позначмо через X підмножину вузлів мережі, а через X’ доповнення цієї підмножини, за якого $X \cup X' = R$, $X \cap X' = \varnothing$, $N_s \in X$, $N_f \in X'$, тобто:

об’єднання підмножин X та X’ утворює всю множину вузлів мережі;

переріз підмножин X та X’ є порожньою множиною;

вузол-джерело належить підмножині X;

вузол-джерело належить підмножині X’.

Тоді *перерізом* (X, X’) називають мінімальну кількість дуг, вилучення яких порушує зв’язаність мережі. У цьому разі переріз відокремлює джерело від стоку ($N_s \in X$, $N_f \in X'$).

Пропускною спроможністю (величиною) перерізу називають величину

$$c(X, X') = \sum_{x \in X} \sum_{x \in \acute{X}} c(x, y), \quad \text{або} \quad c(X, X') = \sum_{i} \sum_{j} b_{ij},$$

для якої суму розраховують за всіма орієнтованими дугами, які ведуть із вузла N_i до вузла N_j, і за всіма неорієнтованими дугами (ребрами), які поєднують підмножини X та X’. Тобто пропускна спроможність перерізу є сумою пропускних спроможностей дуг, початки яких належать підмножині X, а кінці – підмножині X’.

Під час визначення перерізу враховують усі дуги між множинами X та X’, а під час визначення пропускної спроможності перерізу враховують лише пропускні спроможності дуг з N_i до N_j та не враховують орієнтовані дуги з N_j до N_i. Отже, у загальному випадку $c(X, X') \neq c(X', X)$.

Теорема 6.1 *(про максимальний потік)*. Величина максимального потоку мережі дорівнює пропускній спроможності мінімального перерізу, який роз’єднує джерело та стік цієї мережі.

Нехай F_{sf} – множина невід’ємних чисел x_{ij}, які задовольняють обмеження (6.1), (6.2). Припустімо шлях з N_i до N_j збільшує потік x_{ij}, якщо $x_{ij} < b_{ij}$ на всіх прямих дугах та $x_{ij} > 0$ на всіх зворотних дугах цього шляху.

Наслідок із теореми про максимальний потік. Потік є максимальним тоді й тільки тоді, коли немає шляху, який збільшує x_{ij}.

Величина максимального потоку певної мережі має одне значення, проте можуть бути кілька різних шляхів, за яких досягають максимального потоку. Крім того, у мережі може бути кілька мінімальних перерізів.

На рис. 6.3 наведено приклад мережі.

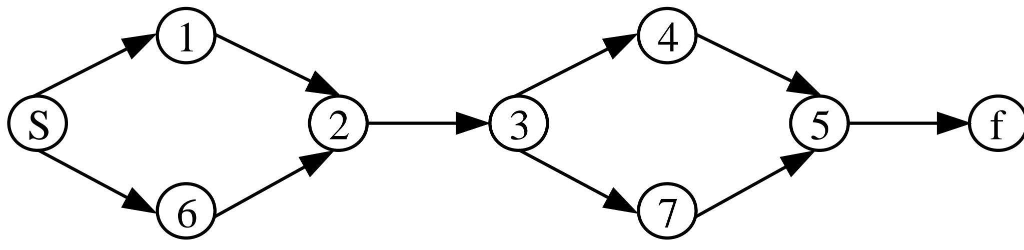


Рис. 6.3. **Мережа із двома мінімальними перерізами**

Припустімо, пропускні спроможності всіх дуг дорівнюють одиниці, отже, величина мінімального потоку також дорівнює одиниці, але максимальних потоків є кілька:

$$x_{s1} = x_{12} = x_{23} = x_{34} = x_{45} = x_{5f} = 1,\ x_{ij} = 0,$$

або

$$x_{s6} = x_{62} = x_{23} = x_{37} = x_{75} = x_{5f} = 1,\ x_{ij} = 0,$$

де x_{ij} – потоки, які не ввійшли до шляху максимального потоку.

Теорема 6.2 *(про мінімальні перерізи)*. Якщо (Y, Y') – мінімальні перерізи, які розділяють s та f, тоді $(X \cup Y, X' \cup Y')$ і $(X \cap Y, X' \cap Y')$ також є мінімальними перерізами, які розділяють s та f.

6.2.2. Метод розставляння міток для визначення максимального потоку

Метод розставляння міток ґрунтується на теоремах про максимальні потоки та мінімальні перерізи. Роботу методу починають із довільного потоку (за початковий можна взяти нульовий потік). Потім намагаються визначити потік, який має більше значення. Робота методу припиняють за досягнення максимального потоку (коли визначено, що цей потік неможливо збільшити).

Метод полягає в систематичному пошуку всіх можливих шляхів, за яких потік збільшується. Під час пошуку використовують процедуру розставляння міток: вузлам приписують мітки, які вказують напрямок можливого збільшення потоку. Після визначення такого шляху обчислюють величину його максимальної пропускної спроможності. Потік збільшують на цю величину, а мітки вузлів вилучають. Наступним етапом є розставляння нових міток, відповідно до визначеного потоку, та продовження пошуку.

Алгоритм методу можна подати такими кроками:

1. Розставляння міток. На цьому кроці вузол може бути в одному із трьох станів: «позначений і переглянутий», «позначений і непереглянутий» та «непозначений». За ініціалізації мережі всі вузли набувають стану «непозначені».

Мітка вузла складається із двох частин:

індексу, який указує на можливість спрямування потоку через цей вузол;

числа, що визначає максимальну величину потоку, який може бути спрямовано із джерела через вузол, за збереження обмежень на пропускні спроможності дуг; таке число називають *резервом вузла*.

Спочатку джерелу приписують мітку $[s^+, \varepsilon(s) = \infty]$, де s^+ означає, що можна збільшити потік, спрямований у вузол; символ нескінченності означає, що величина потоку є необмеженою згори. Вузол s набуває стану «позначений і непереглянутий», усі інші вузли – стану «непозначені».

Розгляньмо будь-який вузол зі станом «позначений і непереглянутий». Припустімо вузол має мітку $[i^+, \varepsilon(j)]$, або $[i^-, \varepsilon(j)]$. З усіх сусідніх з N_j вузлів виділімо непозначені вузли N_k, для яких $x_{jk} < b_{jk}$. Кожному із цих вузлів припишімо мітку $[j^+, \varepsilon(k)]$, де $\varepsilon(k) = \min[\varepsilon(j), b_{jk} - x_{jk}]$. Ці вузли набули стану «позначені й переглянуті».

Усім сусіднім з N_j вузлам, які не мають міток і для яких $x_{kj} > 0$, приписано мітку $[j^-, \varepsilon(k)]$, де $\varepsilon(k) = \min[\varepsilon(j), x_{kj}]$. Тепер ці вузли набули стану «позначені й переглянуті», та всі сусідні з N_j вузли мають мітки. Після цього вузол N_j набуває стану «позначений і переглянутий», а тому на цьому кроці він уже не потребує розгляду.

Якщо деякі сусідні з N_j вузли вже набули стану «позначені й непереглянуті», а інші вже не можуть бути позначеними, то вузол також набуває стану «позначений і переглянутий». Знаки «+» і «–» у мітках визначають, як слід змінити потік на кроці 2.

Мітки вузлам, які є сусідніми з вузлом, який набув стану «позначений і непереглянутий», приписують доти, доки не буде виконано одну із двох умов:

вузол набуде стану «позначений і переглянутий» (тобто всі сусідні з ним вузли не набудуть стану «позначені й непереглянуті»);

не можна позначити жодного вузла, а стік залишається в стані «непозначений».

Наведені умови є умовами припинення роботи алгоритму. Якщо вузол k не може бути позначеним, то немає шляху, за якого можна збільшити потік з N_j до N_k, тому побудований на цьому кроці потік є максимальним. Якщо вузол k набув стану «позначений», то на кроці 2 можна визначити шлях, у якого можна збільшити потік.

2. Зміна потоку. Нехай стік має мітку [k^+, ε(f)]. Замінімо x_{kf} на x_{kf} + ε(f). Якщо стік має мітку [k^-, ε(f)], то x_{fk} замінімо на x_{fk} − ε(f). Потім переходьмо до вузла f.

На другому кроці потоки змінюють за правилом: якщо вузол має мітку [j^+, ε(f)], то потік x_{jk} збільшують x_{jk} + ε(f) і здійснюють перехід до вузла k; якщо вузол має мітку [j^-, ε(f)], то потік x_{kj} зменшують x_{kj} − ε(f) і здійснюють перехід до вузла j. Процедуру продовжують доти, доки не буде досягнуто джерела.

Кроки 1 і 2 повторюють доти, доки є можливість збільшення потоку.

Зауваження. Якщо на початку обчислень значення потоків є цілими числами, то під час роботи цього алгоритму значення потоків також будуть цілими числами. Значення максимального потоку теж буде цілим числом. Це твердження є відомим як теорема про цілочисельність.

Теорема 6.3. *(про цілочисельність)*. Якщо значення пропускних спроможностей дуг є цілими числами, то є максимальний потік, значення якого теж є цілим числом. У наведеному алгоритмі цілочисельність забезпечує збіжність за виконання обмеженої кількості кроків.

Завдання 6.1. Визначте максимальний потік у мережі, у якій кожній орієнтованій дузі поставлено у відповідність два числа (рис. 6.4): перше число визначає пропускну спроможність дуги, друге – початковий потік за дугами.

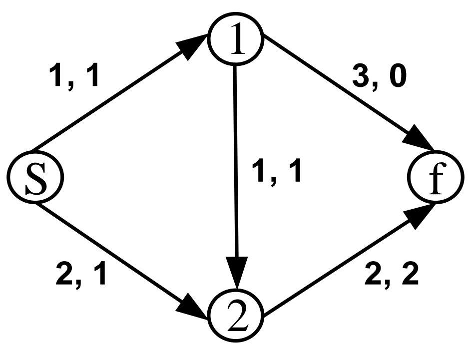


Рис. 6.4. **Первинний вигляд мережі**

Примітка. Початковим може бути будь-який потік, який задовольняє умови (6.1) – (6.2), наприклад потік, який дорівнює одиниці.

Виконання завдання 6.1.

Крок 1. Позначмо вузол s міткою [s^+, ∞]. Вузол s має два сусідні вузли 1 та 2, але позначити їх не можна, оскільки $x_{s1} = b_{s1} = 1$ та $b_{s1} - x_{s1} = 0$, тобто дугового потоку з $x_{s1} > 0$ немає. Вузлу 2 приписуймо мітку [s^+, $b_{s2} - x_{s2}$]. Оскільки $b_{s2} - x_{s2} = 2 - 1 = 1$, то вузол 2 позначаймо міткою [s^+, 1]. Визначаймо резерв вузла 2: $\varepsilon(2) = \min[\varepsilon(s),\ b_{s2} - x_{s2}] = \min[\infty,\ 1] = 1$. Тепер вузол s набуває стану «позначений і переглянутий», а вузол 2 – «позначений і непереглянутий» (рис. 6.5).

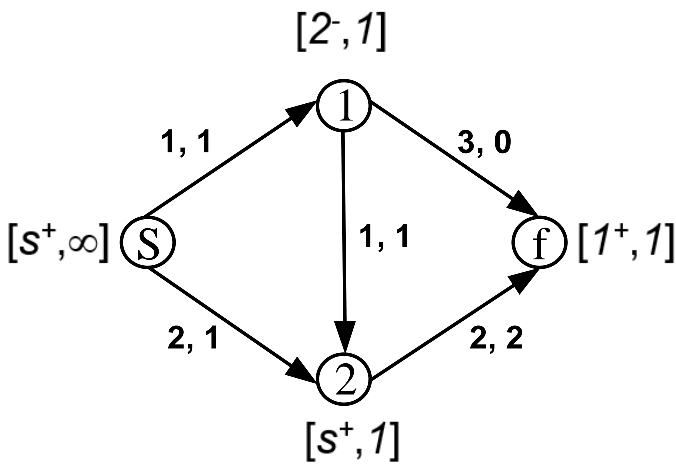


Рис. 6.5. **Результат розставляння міток на першому кроці**

Вузол 2 має два сусідні вузли, які є в стані «непозначений» – 1 і f. Наразі вузол f не може бути позначеним, оскільки $x_{2f} = b_{2f} = 2$ та $b_{2f} - x_{2f} = 0$, а вузлу 1 приписуймо мітку [2^-, 1], оскільки $x_{12} > 0$, а його резерв визначено як $\varepsilon(1) = \min[\varepsilon(2), x_{12}] = \min[1, 1] = 1$. Тепер вузол 2 набуває стану «позначений і переглянутий», а 1 – «позначений і непереглянутий». Вузол 1 має лише одного сусіда, який є в стані «непозначений» – це вузол f, якому приписуймо мітку [1^+, 1], оскільки його резерв визначено як $\varepsilon(f) = \min[\varepsilon(1), b_{1f} - x_{1f}] = \min[1, 3 - 0] = 1$.

На рис. 6.5 подано результат приписування міток на першому кроці. Тепер усі вузли набули стану «позначений», тому переходьмо до другого кроку.

Крок 2. Оскільки мітка вузла f – 1^+, то збільшуймо потік x_{1f} на 1: $x_{1f}^{(1)} = x_{1f} + 1 = 0 + 1 = 1$. Переходьмо до вузла 1, якому приписано мітку [2^-, 1], та зменшуймо потік x_{12} на 1: $x_{12}^{(1)} = x_{12} - 1 = 1 - 1 = 0$. Переходьмо

до вузла 2, якому приписано мітку $[s^+, 1]$ та збільшуймо потік x_{s2} на 1: $x_{s2}^{(1)} = x_{s2} + 1 = 1 + 1 = 2$.

На рис. 6.6 подано результат зміни потоків на другому кроці.

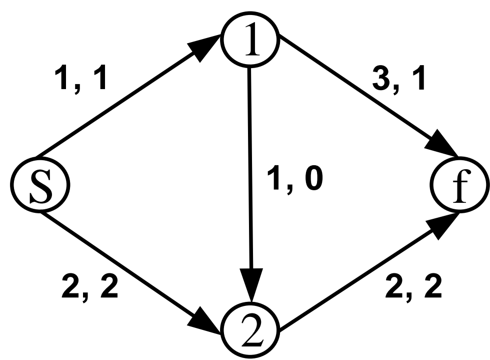


Рис. 6.6. **Кінцевий результат операції зміни потоків**

Крок 3. Надаймо мітку вузлу s. Тепер вузли 1 і 2 не можуть бути позначеними, оскільки $x_{s1} = b_{s1} = 1$ та $b_{s1} - x_{s1} = 0$, а також $x_{s2} = b_{s2} = 2$ та $b_{s2} - x_{s2} = 0$. Вузол f також залишається в стані «непозначений». Це означає, що визначений потік є максимальним, тому алгоритм зупиняє роботу.

Якщо значення пропускних спроможностей не є цілими числами, алгоритм може не збігатися до максимального потоку, а отже, не мати розв'язку.

6.2.3. Модифікований метод розставляння міток для визначення максимального потоку

Модифікацію методу розставляння міток призначено для неорієнтованих мереж, значення пропускних спроможностей яких є дійсними числами. Алгоритм модифікованого методу може бути поданим такими кроками:

Крок 1. Вилучаймо з мережі дуги, у яких потік досягає пропускної спроможності $x_{ij} = b_{ij}$; такі дуги є дугами з *насиченим потоком.*

Крок 2. Використовуючи дуги, які залишилися, визначаймо довільний шлях за допомогою методу розставляння міток та спрямовуймо за ним максимально можливий потік. Якщо такий шлях визначено, переходьмо до кроку 1, інакше – до кроку 3.

Крок 3. Відновлюймо раніше вилучені дуги (дуги з насиченими потоками). Визначаймо шлях із вузла N_i до вузла N_j, за яким потік можна збільшити, та спрямовуймо цим шляхом максимально можливий потік. Якщо такий шлях визначено, переходьмо до кроку 1, інакше алгоритм завершує свою роботу, а визначений шлях є оптимальним.

Для зручності будемо вважати, що мережа, яка вміщує n вузлів, має n(n – 1) дуг, хоч серед них можуть бути дуги, пропускна спроможність яких дорівнює 0. Позначмо через F потік мережі, а через N(F) мережу із цим потоком.

Дуга мережі може бути насиченою $x_{ij} = b_{ij}$, тоді можна вважати, що цієї дуги в мережі немає, оскільки неможливо спрямувати потік із вузла i до вузла j дугою (i, j).

Під мережею N() будемо мати на увазі мережу, яка відрізняється від початкової та у якій ще не було потоку. Метод розставляння міток генерує набір мереж: $N(F_1)$, $N(F_2)$, …, $N(F_m)$, де F_m – максимальний потік. Потік F_{k+1} утворено з потоку F_k додаванням до мережі $N(F_k)$ шляху, який збільшує потік.

Метод розставляння міток розглядає позначені вузли в такому порядку: першим був позначеним – першим буде переглянутим. Отже, кожного разу визначаймо шлях зі зростальним потоком. Такий шлях уміщує мінімальну кількість дуг, за умови що потік мережі ще не досяг максимального значення.

Зауваження. Припустімо, що A_{s1}, $A_{12,}$ A_{23}, …, A_{kf} – це шлях, який збільшує потік, тоді резерв вузла ε(j + 1) визначають за такою формулою:

$$\varepsilon(j + 1) = \min[\varepsilon(j), b_{j,j+1} - x_{j,j+1} + \ldots + x_{j+1,j}].$$

За такої модифікації визначення міток хоча б одна дуга на шляху, який збільшує потік, буде дугою із насиченим потоком.

6.2.4. Алгоритм Форда – Фалкерсона визначення максимального потоку

1954 р. Форд (Ford) і Фалкерсон (Fulkerson) довели теорему про максимальний потік і мінімальний розріз для неорієнтованих графів. Запропонований ними 1955 р. алгоритм складається з таких кроків, які

повторюють, доки умови припинення роботи алгоритму (див. пп. 6.2.2) не буде виконано.

Крок 1. Позначаймо батька та резерв кожного вузла дефісом, що означає стан «непозначений». Резерву вузла s надаймо значення ∞ для того, щоб він не обмежував резерви інших вузлів.

Формуймо множину вузлів S, які є у стані «позначений і непереглянутий». На першому кроці ця множина містить лише джерело мережі S = [s].

Крок 2. Якщо S є порожньою множиною, потік є максимальним, якщо ні, то вилучаймо та опрацьовуймо такий елемент.

Крок 3. Якщо вузол *i* є у стані «позначений і непереглянутий», а (i, j) – дуга мережі, слід визначити резерв вузла *j* за такою формулою:

$$\varepsilon(j) = \min[\varepsilon(i), b_{ij} - x_{ij}]$$

та встановити *i* за батьківський вузол для вузла *j*. Якщо процедура ще не досягла стоку, тобто $j \neq f$, додати вузол *j* до множини S.

Крок 4. Якщо вузол *j* в стані «непозначений» та $x_{ij} > 0$, визначмо резерв вузла *j* як $\varepsilon(j) = \min[\varepsilon(i), x_{ij}]$ та встановімо *i* за батьківський вузол для вузла *j*, додаймо вузол *j* до множини *S*.

Крок 5. Після того як вузол f набуде стану «позначений» слід за допомогою функції передування повернутися до вузла *j* та додати резерв вузла *f* до потоку кожної правильно орієнтованої дуги, яка становить шлях від *j* до *f*, та вирахувати резерв вузла *f* із потоку кожної неправильно орієнтованої дуги. Повернімося до кроку 1.

Завдання 6.2. Визначте максимальний потік за допомогою алгоритму Форда – Фолкерсона в мережі, яка показана на рис. 6.7. Пропускну спроможність дуг позначено числами поруч із дугами.

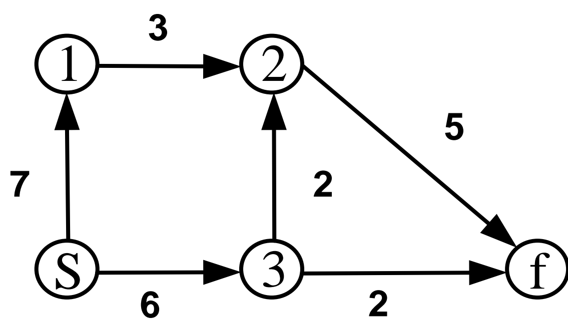


Рис. 6.7. **Початковий стан мережі**

Виконання завдання 6.2. На **кроці** 1 батьківській вузол та резерв кожного вузла, крім джерела, позначаймо дефісом. Резерв вузла s позначаймо ∞ та додаймо його до множини S: S = [s]. На рис. 6.8 подано результат виконання першого кроку.

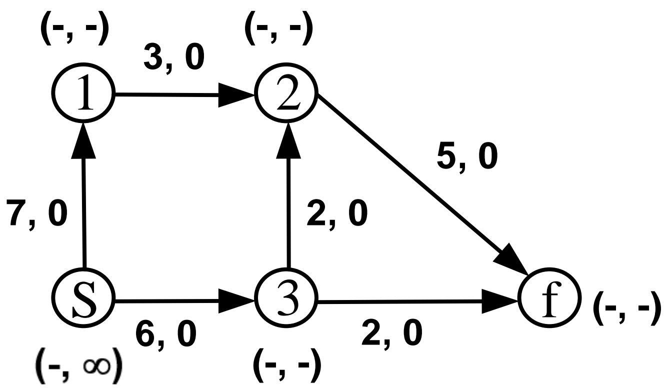


Рис. 6.8. **Результат виконання першого циклу роботи алгоритму**

На **кроці** 2 перевіряймо умову $S \neq \varnothing$. Умову виконано, тому з множини S вилучаймо та опрацьовуймо вузол s.

На **кроці** 3 визначаймо резерв вузла 1 за такою формулою:

$$\varepsilon(1) = \min[\varepsilon(s), x_{s1}],$$

тобто $\varepsilon(1) = \min[\infty, 7] = 7$.

За батьківський вузол вузла 1 визначаймо s. Уносімо вузол 1 до множини S: S = [1]. Визначаймо резерв вузла 3 як $\varepsilon(3) = \min[6, \infty] = 6$, а його батьківський вузол – s. Уносімо вузол 3 до множини S: S = [1, 3].

Оскільки на цьому етапі процедура ще не дійшла до розгляду стоку, то кроки 4 і 5 не виконуймо, а переходьмо до кроку 2.

Крок 2 починаймо з перевірки умови $S \neq \varnothing$. Множина S містить вузли 1 та 2 S = [1, 3], тому з S вилучаймо вузол, наприклад, 1, і залишаймо вузол 3, тепер S = [3].

На **кроці** 3 визначаймо резерв вузла 2 як

$$\varepsilon(2) = \min[\varepsilon(1), b_{12} - x_{12}] = \min[7, 3 - 0] = 3,$$

а вузол 1 як батьківський вузол для вузла 2. Додаймо вузол 2 до множини S, тепер S = [2, 3].

Процедура ще не досягла стоку, тому кроки 4 і 5 не виконуймо, а переходьмо до кроку 2.

Оскільки $S \neq \varnothing$ ($S = [2, 3]$), то із множини S вилучаймо вузол 2, тепер $S = [3]$.

На **кроці 3** визначаймо резерв вузла f як

$$\varepsilon(f) = \min[\varepsilon(2), b_{2f} - x_{2f}] = \min[3, 7 - 0] = 3,$$

вузол 2 як батьківський вузол для вузла f. Вузол f до множини S не вносять.

Оскільки процедура досягла стоку, переходьмо до **кроку 4**. На **кроці 4** потрібно позначити вузол 3, але він уже є в стані «позначений і непереглянутий» ($S = [3]$), тому переходьмо до кроку 5.

На **кроці 5** формуймо функцію передування з вузлів, які були вилученими із множини на попередніх кроках реалізації алгоритму – s; 1; 2. Стік не був унесеним до множини S, тому його слід додати до функції передування окремо. Маємо такий шлях збільшення потоку:

$$s \xrightarrow{(7,\,0)} 1 \xrightarrow{(3,\,0)} 2 \xrightarrow{(5,\,0)} f.$$

Додаючи резерв стоку $\varepsilon(f) = 3$ до потоку кожної дуги шляху, маємо:

$$s \xrightarrow{(7,\,3)} 1 \xrightarrow{(3,\,3)} 2 \xrightarrow{(5,\,3)} f.$$

На рис. 6.9 подано вигляд мережі після зміни потоків.

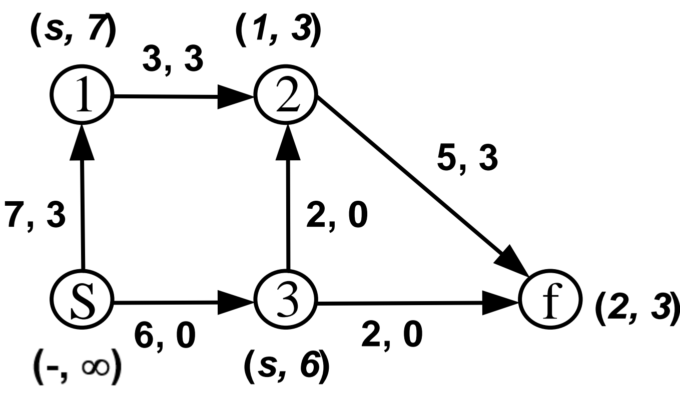


Рис. 6.9. **Результат виконання другого циклу роботи алгоритму**

Умову $S \neq \varnothing$ виконано ($S = [3]$), тому переходьмо до кроку 1.

На **кроці 1** батьківській вузол і резерв кожного вузла, крім джерела ($\varepsilon(s) = \infty$), позначаймо дефісом та додаймо вузол s до множини S, тепер $S = [s, 3]$.

На **кроці 2** перевіряймо умову $S \neq \varnothing$ та вилучаймо вузол s із множини S.

На **кроці 3** визначаймо резерв вузла 1 як

$$\varepsilon(1) = \min[\varepsilon(s), b_{s1} - x_{s1}] = \min[\infty, 7 - 3] = 4,$$

вузол s як батьківський вузол для вузла 1. Додаймо вузол 1 до множини S, тепер $S = [1, 3]$.

Оскільки на цьому етапі процедура ще не дійшла до розгляду стоку, то кроки 4 і 5 не виконуймо, а переходьмо до кроку 2.

На **кроці 2** вилучаймо вершину 1 із множини S (тепер $S = [3]$). Дуга x_{12} має насичений потік ($x_{12} = b_{12}$), тому вершину 2 не позначаймо та не додаймо до множини S. Вилучаймо вузол 3 із множини S, переходьмо до кроку 3 та розглядаймо вузли, які є сусідами вузла 3.

На **кроці 3** визначаймо резерв стоку f як

$$\varepsilon(f) = \min[\varepsilon(3), b_{3f} - x_{3f}] = \min[6, 2 - 0] = 2,$$

а резерв вузла 2 як

$$\varepsilon(2) = \min[\varepsilon(3), b_{32} - x_{32}] = \min[6, 2 - 0] = 2.$$

Процедура досягла стоку, тому переходьмо до кроку 4: визначаймо вузол 3 як батьківський вузол для вузла 2. Вилучаймо вузол 3 із множини S (тепер $S = \varnothing$). Переходьмо до кроку 5.

На кроці 5 вузол 3 буде вже в стані «позначений і переглянутий». Формуймо функцію передування з вузлів, які були вилученими із множини S на попередніх кроках алгоритму та маємо такий шлях збільшення потоку:

$$s \xrightarrow{(6,\ 0)} 3 \xrightarrow{(2,\ 0)} f.$$

Додаймо резерв стоку (ε(f) = 2) до потоку кожної дуги та маємо наступний шлях:

$$s \xrightarrow{(6,\,2)} 3 \xrightarrow{(2,\,2)} f.$$

На рис. 6.10 подано вигляд мережі після зміни потоків.

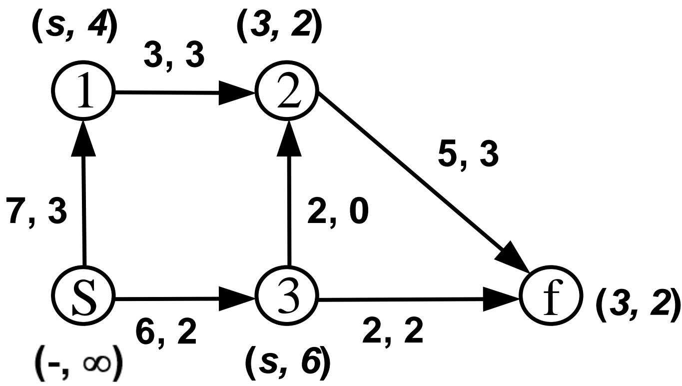


Рис. 6.10. **Результат виконання третього циклу роботи алгоритму**

Переходьмо до **кроку 1**. Позначаймо мітки резервів та батьківських вузлів дефісами (рис. 6.11), а вузол s додаймо до множини S (S = [s]).

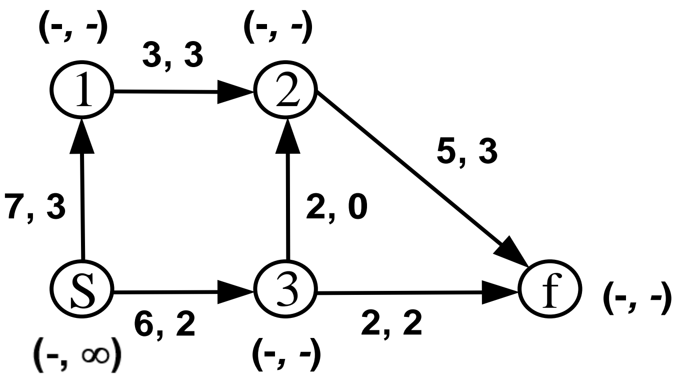


Рис. 6.11. **Нове розмічання мережі після завершення трьох циклів обчислень**

Крок 2. Умову $S \neq \varnothing$ виконано, тому вузол s вилучаймо із множини S.

Крок 3. Значення резерву вузла 1 залишається незмінним ($\varepsilon(1) = 4$), а його батьківським вузлом буде вузол s. Потік x_{12} є насиченим потоком ($b_{12} = x_{12}$), тому вузол 1 до множини S не додаймо.

Визначаймо резерв вузла 3 як

$$\varepsilon(3) = \min[\varepsilon(s), b_{s3} - x_{s3}] = \min[\infty, 6 - 2] = 4.$$

Додаймо вузол s до множини S (S = [s]). Визначаймо вузол s за батьківський вузол для вузла 3. Вузол 3 додаймо до множини S (тепер S = [3]).

Стоку ще не досягнуто, тому кроки 4 і 5 не виконуймо, а переходьмо до кроку 2.

Крок 2. Умову $S \neq \varnothing$ виконано, тому вузол 3 вилучаймо із множини S, визначаймо резерв вершини 2 як

$$\varepsilon(2) = \min[\varepsilon(3), b_{32} - x_{32}] = \min[4, 20] = 2,$$

вузол 3 батьківським вузлом для вузла 2. Додаймо вузол 2 до множини S (S = [2]).

Крок 3. Визначаймо резерв вузла f як:

$$\varepsilon(f) = \min[\varepsilon(2), b_{2f} - x_{2f}] = \min[2, 5 - 3] = 2.$$

Вузол f є стоком, тому його не додають до множини S. Процедура сягнула стоку, тому переходьмо до кроків 4 та 5.

Крок 4. Визначаймо вузол 2 батьківським вузлом для вузла f та вилучаймо його із множини S ($S = \varnothing$).

Крок 5. Формуймо функцію передування з вузлів, які були вилученими із множини S на попередніх кроках алгоритму та маємо такий шлях збільшення потоку:

$$s \xrightarrow{(6,\,2)} 3 \xrightarrow{(2,\,0)} 2 \xrightarrow{(5,\,3)} f.$$

Додаймо резерв стоку $\varepsilon(f) = 2$ до потоку кожної дуги та маємо:

$$s \xrightarrow{(6,\,4)} 3 \xrightarrow{(2,\,2)} 2 \xrightarrow{(5,\,5)} f.$$

На рис. 6.12 подано вигляд мережі після зміни потоків.

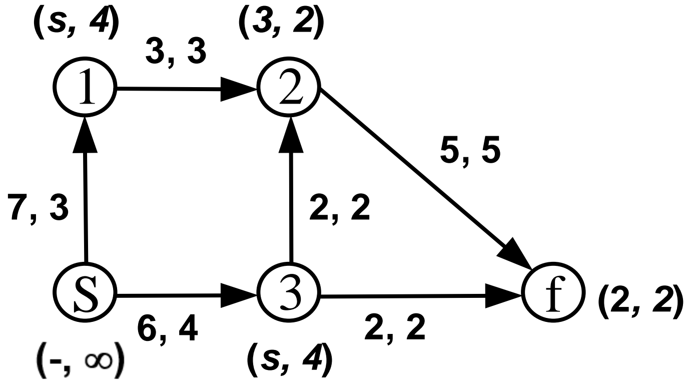


Рис. 6.12. **Стан мережі після четвертого циклу обчислень**

Переходьмо до **кроку 1**. Установлюймо мітки резервів і батьківських вузлів та додаймо вузол s до множини S.

На **кроці 2** вилучаймо вузол s із множини S ($S = \varnothing$).

На **кроці 3** визначаймо резерв вузла 1 $\varepsilon(1) = 4$, а батьківським вузлом вузла 1 вузол s. Додаймо вузол 1 до множини S (S = [1]).

Визначаймо резерв вузла 3 $\varepsilon(3) = 2$ та додаймо вузол 3 до множини S (S = [1, 3]). Переходьмо до кроку 2.

На **кроці 2** вилучаймо вузол 1 із множини S (S = [3]). Потік x_{12} є насиченим $x_{12} = b_{12}$, тому позначити вузол 2 неможливо. Переходьмо до кроку 2.

На **кроці 2** вузол 3 вилучаймо із множини S ($S = \varnothing$). Потоки x_{32} та x_{3f} є насиченими ($x_{32} = b_{32}$ та $x_{3f} = b_{3f}$), тому неможливо позначити вузли 2 та f. Переходьмо до кроку 2.

На **кроці 2** перевіряймо умову $S \neq \varnothing$ але її не виконано, оскільки множина S є порожньою. Роботу алгоритму завершено.

На рис. 6.13 подано максимальний потік мережі.

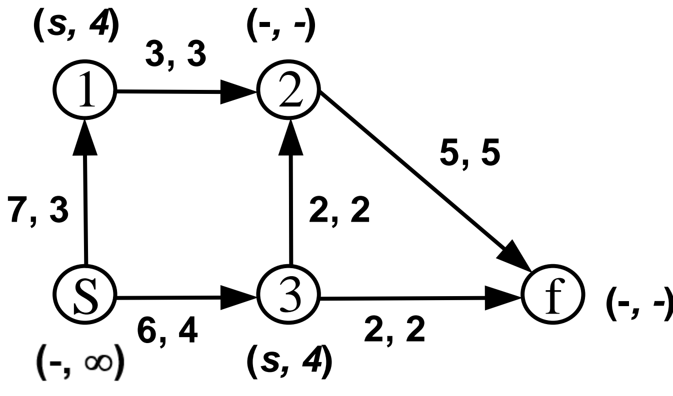


Рис. 6.13. **Кінцевий стан мережі**

Завдання 6.3. Визначте максимальний потік за допомогою алгоритму Форда – Фолкерсона в мережі, яка показана на рис. 6.14. Пропускну спроможність дуг позначено числами поруч із дугами.

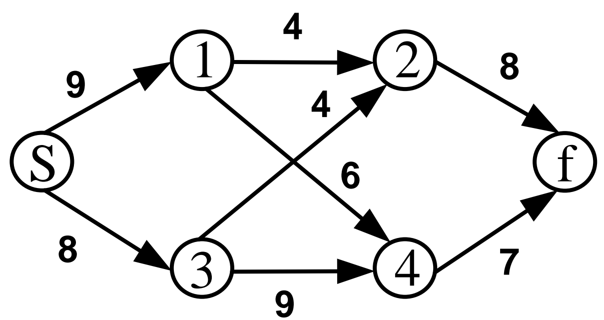


Рис. 6.14. **Первинний стан мережі**

Виконання завдання 6.3. Перші три ітерації алгоритму виконують аналогічно до прикладу 6.2, тому наведімо лише їхній стислий опис.

За *першої ітерації* визначаймо такі мітки вузлів: s(−, ∞), 1(s, 9), 2(1, 4), 3(s, 8), 4(1, 6) та f(2, 4) – та маємо такий шлях:

$$s \xrightarrow{(9,0)} 1 \xrightarrow{(4,0)} 2 \xrightarrow{(8,0)} f.$$

Резерв стоку дорівнює 4(f(2, 4)), тому маємо:

$$s \xrightarrow{(9,4)} 1 \xrightarrow{(4,4)} 2 \xrightarrow{(8,4)} f.$$

На рис. 6.15 подано вигляд мережі після зміни потоків.

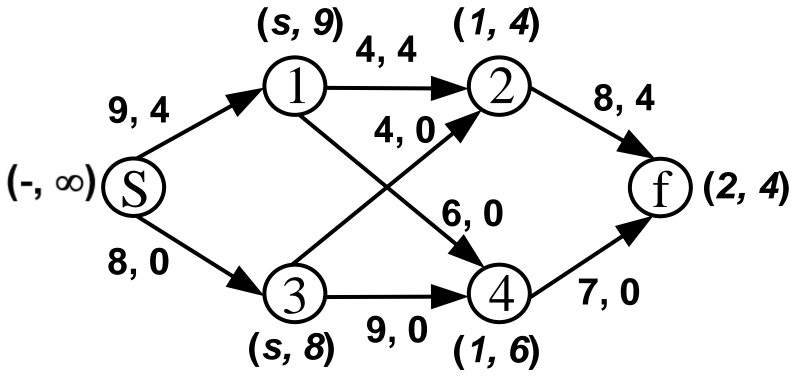


Рис. 6.15. **Стан мережі після першого циклу**

За *другої ітерації* визначаймо такі мітки вузлів: s(−, ∞), 1(s, 5), 2(1, 4), 3(s, 8), 4(3, 8) і f(4, 7) – та маємо шлях:

$$s \xrightarrow{(8,\,0)} 3 \xrightarrow{(9,\,0)} 4 \xrightarrow{(7,\,0)} f.$$

Резерв стоку дорівнює 7(f(2, 7)), тому маємо:

$$s \xrightarrow{(8,\,7)} 3 \xrightarrow{(9,\,7)} 4 \xrightarrow{(7,\,7)} f.$$

На рис. 6.16 подано вигляд мережі після зміни потоків.

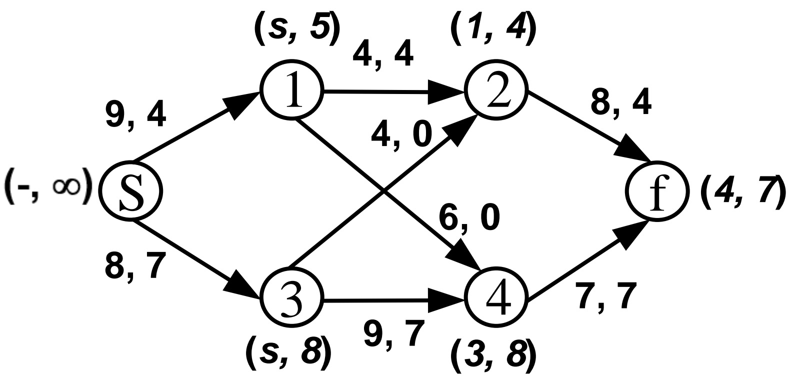


Рис. 6.16. **Стан мережі після другого циклу**

За *третьої ітерації* визначаймо такі мітки вузлів: s(−, ∞), 1(s, 5), 2(3, 1), 3(s, 1), 4(3, 1) і f(2, 1) – та маємо шлях:

$$s \xrightarrow{(8,\,7)} 3 \xrightarrow{(4,\,0)} 2 \xrightarrow{(8,\,4)} f.$$

Резерв стоку дорівнює 1(f(2, 1)), тому маємо:

$$s \xrightarrow{(8,\,8)} 3 \xrightarrow{(4,\,1)} 2 \xrightarrow{(8,\,5)} f.$$

На рис. 6.17 подано вигляд мережі після зміни потоків.

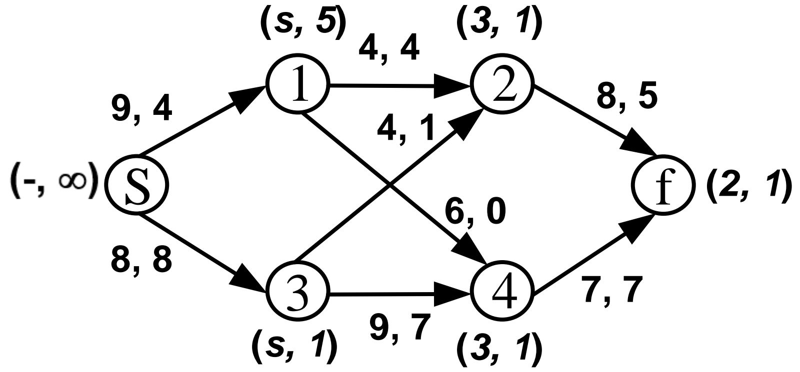


Рис. 6.17. **Стан мережі після третього циклу**

Подальша реалізація алгоритму передбачає деякі особливості, тому розгляньмо її детальніше.

Припишімо вузлу 1 мітку (s, 5) та додаймо вузол 1 до множини S (S = [1]). Вузол 3 позначити неможливо: потік x_{s3} є насиченим ($x_{s3} = b_{s3} = 8$). Вилучаймо вузол 1 із множини S (S = ∅) та розглядаймо сусідні з ним вузли:

вузол 2 неможливо позначити, оскільки потік x_{12} є насиченим ($x_{12} = b_{12} = 4$);

вузлу 4 можна приписати мітку (1, 5) та додати його до множини S (S = [4]).

Вилучаймо вузол 4 із множини S (S = ∅) та розглядаймо сусідні з ним вузли:

вузол f неможливо позначити: потік x_{4f} є насиченим ($x_{4f} = b_{4f} = 7$);

вузол 3 можна позначити, тому що він ще немає мітки.

Зауважмо, що дуга (4, 3) є орієнтованою неправильно, тому на кроці 4 її резерв визначають, згідно з такою формулою:

$$\varepsilon(3) = \min[\varepsilon(4), x_{34}].$$

Ураховуючи, що $\varepsilon(4) = 5$ та $x_{34} = 7$, визначаймо $\varepsilon(3) = \min[5, x] = 5$, вузол 4 – як батьківський вузол для вузла 3.

Вузол 3 додаймо до множини S (S = [3]).

За розгляду сусідів вузла 3 можна приписати вузлу 2 мітку (3, 3) та додати його до множини S (S = [3, 2]). Вилучаймо вузол 2 із множини S (S = [3]) та приписуймо його сусідові, вузлу f, мітку (2, 3). Вузли, сусідні з вузлом 3, розглянуто, тому цей вузол вилучають із множини S (S = ∅).

На цьому поточну ітерацію алгоритму завершено.

За реалізації *четвертої ітерації* маємо такий шлях:

$$s \xrightarrow{(9,4)} 1 \xrightarrow{(6,0)} 4 \xleftarrow{(9,7)} 3 \xrightarrow{(4,1)} 2 \xrightarrow{(8,5)} f.$$

Визначені значення потоків збільшуймо на резерв стоку f $\varepsilon(3) = 3$, окрім дуги (3, 4). Потік цієї дуги має протилежний напрямок, тому його значення зменшуймо на резерв стоку, та маємо такий шлях:

$$s \xrightarrow{(9,7)} 1 \xrightarrow{(6,3)} 4 \xleftarrow{(9,4)} 3 \xrightarrow{(4,4)} 2 \xrightarrow{(8,8)} f.$$

На рис. 6.18 подано вигляд мережі після зміни потоків.

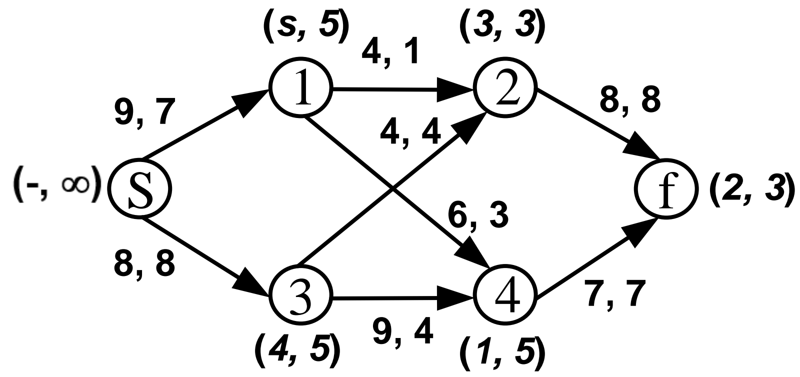


Рис. 6.18. **Результат поточних обчислень**

Розгляньмо *останню ітерацію* алгоритму. Починаймо з S = [s], вилучаймо вузол s із множини S, приписуймо вузлу 1 мітку (s, 5) та додаймо вузол 1 до множини S (S = [1]). Вилучаймо вузол 1 із множини S (S = ∅), приписуймо вузлу 4 мітку (1, 2), а вузлу 2 – мітку (1, 2) та додаймо вузли 2, 4 до множини S (S = [2, 4]).

Вилучаймо вузол 4 із множини S (S = [2]). Визначаймо резерв вузла 3 $\varepsilon(3) = \min[\varepsilon(4), x_{34}] = \min[2, 4] = 2$ та приписуймо вузлу 3 мітку (4, 2). Додаймо вузол 3 до S (S = [2, 3]).

На наступному кроці вилучаймо вузол 3 із множини S (S = [2]) та розглядаймо сусідні з ним вузли: до вузла 2 веде дуга з насиченим потоком, а вузол 4 вже є в стані «позначений і переглянутий», тому переходьмо до кроку 2.

Вилучаймо вузол 2 із множини S (S = ∅) та розглядаймо сусідні з ним вузли: до вузла f веде дуга з насиченим потоком, тому він не може бути позначеним, тому переходьмо до кроку 2.

На кроці 2 перевіряймо умову $S \neq \varnothing$, але множина S є порожньою, тому роботу алгоритму завершено, а визначений потік є максимальним.

На рис. 6.19 подано максимальний потік мережі.

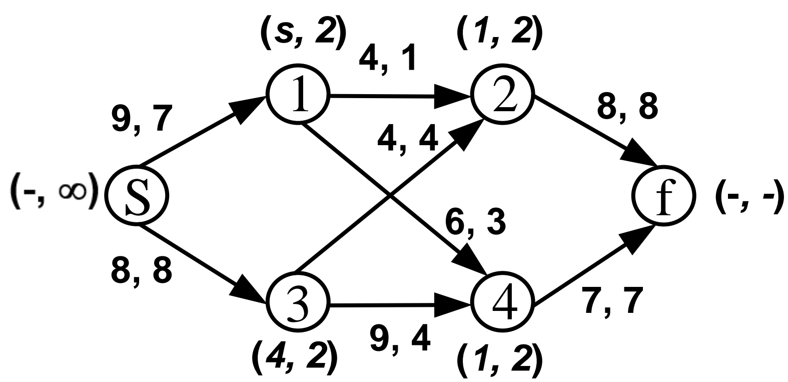


Рис. 6.19. **Кінцевий стан мережі**

Контрольні запитання і завдання для самоперевірки

1. Дайте визначення зв’язної мережі.
2. Що розуміють під орієнтованим ланцюгом? Які ланцюги називають простими?
3. Що розуміють під орієнтованими циклами?
4. Порівняйте поняття шляху й ланцюга в мережі.
5. Дайте визначення пропускної спроможності дуги (ребра).
6. Наведіть формулювання задачі про максимальний потік у мережі.
7. Дайте визначення перерізу мережі.
8. Сформулюйте теорему про максимальний потік.
9. Сформулюйте теорему про мінімальні перерізи.
10. У чому полягає сутність алгоритму розставляння міток для визначення максимального потоку?
11. Яку інформацію містять мітки вузлів мережі в процесі пошуку максимального потоку?
12. Сформулюйте теорему про цілочисельність.
13. Охарактеризуйте кроки алгоритму розставляння міток для визначення максимального потоку.
14. У чому полягає сутність модифікації методу розставляння міток для визначення максимального потоку?

Використана
та рекомендована література

1. Бородкіна І. Л. Теорія алгоритмів / Г. О. Бородкін, І. Л. Бородкіна. – Київ : Центр навчальної літератури, 2020. – 184 с.

2. Васильєв О. Алгоритми / О. Васильєв. – Київ : Вид-во «Ліра-К», 2022. – 422 с.

3. Ковалюк Т. В. Алгоритмізація та програмування / Т. В. Ковалюк. – Львів : Магнолія 2006, 2018. – 400 с.

4. Креневич А. П. Алгоритми і структури даних : підручник / А. П. Креневич. – Київ : ВПЦ «Київський Університет», 2021. – 200 с.

5. Матвієнко М. П. Теорія алгоритмів / М. П. Матвієнко. – Київ : Вид-во «Ліра-К», 2019. – 344 с.

6. Петрова О. О. Алгоритмічні задачі та їх вирішення : навч. посіб. / О. О. Петрова, Г. В. Солодовник ; Харків. нац. ун-т міськ. гос-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2021. – 105 с.

7. Угрин Д. І. Структури даних та алгоритми : підручник / Д. І. Угрин, Ю. О. Ушенко, М. Л. Ковальчук. – Чернівці : Чернівецький національний університет ім. Ю. Федьковича, 2022. – 357 с.

8. Шаховська Н. Б. Алгоритми та структури даних / Н. Б. Шаховська, Р. О. Голощук. – Львів: Магнолія 2006, 2020. – 216 с.

9. Gosnell D. The Practitioner’s Guide to Graph Data: Applying Graph Thinking and Graph Technologies to Solve Complex Problems / D. Gosnell, M. Broecheler. – Sebastopol, CA : O’Reilly Media, 2020. – 420 p.

10. Lewis H. Essential Discrete Mathematics for Computer Science / H. Lewis, R. Zax. – Prinston : Prinston University Press, 2019. – 408 p.

11. Jain H. Data Structures & Algorithms In Go. Second Edition / H. Jain. – S. l. : s. n., 2019. – 519 p.

12. Karumanchi N. Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles / N. Karumanchi. – 5th ed. – S. l. : Career Monk, 2018. – 415 p.

13. Introduction to Algorithms / T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein. – 4-th ed. – Cambridge, MaY : MIT Press, 2022. – 1296 p.

14. Zverovich V. Research Topics in Graph Theory and Its Applications / V. Zverovich. – S. l. : Cambridge Scholars Publishing, 2019. – 309 p.

15. Реалізації алгоритмів сортування у Python / TechUkraine.net [Електронний ресурс]. – Режим доступу : https://techukraine.net/%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%97-%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%96%D0%B2-%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B2-python/.

Додатки

Додаток А

Варіанти індивідуальних завдань

Таблиця А.1

Завдання до лінійного обчислювального процесу

Номери варіантів	Теоретичні запитання	Практичні завдання: складіть схеми алгоритмів
1	2	3
1	Назвіть способи опису алгоритмів	$y = \frac{cx^2 + mb^3 + \ln x + 7}{b + m + \sqrt{c}}$, $m = 2x + \sqrt{x}$, $b = ac^2 - m$. Виведіть значення змінних Y, m, b
2	Дайте визначення поняття «змінна»	$b = a + \sin c$, $a = c + 2$, $s = \sqrt{a + \cos(a + b) - b^2}$. Виведіть значення змінних S, a, b
3	Назвіть етапи розв'язання задач за допомогою ЕОМ	$a = b + c$, $s = (a + b)\sqrt{a^2} + f^3 + b$, $b = c + f^3$, $c = 11$. Виведіть значення змінних a, s, b
4	Дайте визначення поняття «програма»	$a = y$, $z = \frac{1 - y}{1 + y} 2fy + a^2$, $f = 2y + \sin a$. Виведіть значення змінних a, z, f

Продовження табл. А.1

1	2	3
5	Охарактеризуйте типи алгоритмів	$s = (a + b)\sqrt{a^2 + f^2} + b,$ $f = 2a + b,$ $b = 3{,}33.$ Виведіть значення змінних s, f, b
6	Визначте сутність властивості алгоритму «масовість»	$Q = \frac{(p + m)^2}{c + a},$ $p = m^2 + 2 + c_1,$ $c_1 = c + a,$ де a – довільне значення. Виведіть значення змінної Q
7	Визначте сутність властивості алгоритму «дискретність»	$x = 3y + 2b,$ $y = b^2 + 3,$ $m = 2 - \frac{3x}{3 + b} - \frac{y}{0{,}9 - b},$ $b = 9.$ Виведіть значення змінних x, y, m
8	Дайте визначення поняття «алгоритм»	$y = ax^2 + bx + c,$ $c = \sqrt{ax} + d,$ $x = d + b,$ $a = 0{,}24,$ де b – довільне число. Виведіть значення змінних y, c, x

Продовження додатка А

Продовження табл. А.1

1	2	3
9	Визначте сутність властивості алгоритму «скінченність»	$p\frac{a+\ln b+l^{x}}{d+k}$, $d=a^{2}+k1$, $b=2$, $K=(d+a+b)^{3}$. Виведіть значення змінної P
10	Опишіть типи алгоритмів	$M=\frac{K^{2}+4\times F}{c+a}$, $c=F+2a$, $a=K+q+b$, $b=21{,}4$. Виведіть значення змінних M, a
11	Назвіть способи опису алгоритмів	$D=\frac{B+A^{2}}{K+P}$, $P=A+B^{3}$, $K=\ln\lvert A+B^{3}\rvert$, $B=-0{,}1$. Виведіть значення змінних K, P, D
12	Назвіть етапи розв’язання задачі за допомогою ЕОМ	$F=\frac{3\times(m+a)+2\times x^{2}+\sin x}{\ln x}$, $m=\cos x+\sin x$, $x=b$, $b=4{,}4$. Виведіть значення змінних m, F

Продовження табл. А.1

1	2	3
13	Визначте операцію надання значення змінній	$p = \frac{3{,}2d}{a + 2x} + \frac{mt}{5} + \frac{cx^2}{2m}$, $m = C + 2x^2 + d$, $c = \sqrt{b + 2a}$. Виведіть значення змінних P, m, c
14	Дайте визначення поняття «константа»	$T = \frac{t_1 + t_2}{a + c^2}$, $t_1 = x + b$, $t_2 = x - b$, $c = a + b$. Виведіть значення змінної T
15	Визначте сутність властивості алгоритму «масовість»	$I = \frac{j + k^3}{p + n}$, $P = j^3 + n^3$, $K = p^3 + a$, $a = 4{,}4$. Виведіть значення змінної I
16	Перелічіть властивості алгоритмів	$V1 = \frac{v}{\sqrt{1 - \frac{a^2}{c^2}}}$, $X = V1$, $c = 3 \sin x$. Виведіть значення змінних V1, X

Продовження табл. А.1

1	2	3
17	Перелічіть способи опису алгоритмів	$S = \frac{S1 + T + dc}{\sin K}$, $K = d + c$, $d = e^x + \sin y$, $c = y^2 + x^2$ $z = 2S + y_1$, $y_1 = c + d + k^3$. Виведіть значення змінних s, k, d, c, z, y_1
18	Дайте стислу характеристику лінійного обчислювального процесу	$F = \frac{t^2 + t_1^3 + t_2^4}{\ln x + y^3}$, $X = a + \frac{d}{2}$, $t_1 = (t_2 + t_3)^3$, $t_2 = \frac{z + x}{y}$, $y = 2x + k$. Виведіть значення змінних F, x, t_1, t_2, y
19	Визначте сутність властивості алгоритму «результативність»	$N = \frac{k + t}{t_2}$, $y = a + b_1$, $b = y^2$, $c = y + b$, $t = \frac{c + b + y}{2k}$, $k = \frac{t_1 + Q}{3}$. Виведіть значення змінних N, y, e, t, k

Продовження табл. А.1

1	2	3
20	Дайте визначення поняття «алгоритм»	$A = \frac{f + c^2 + p}{2d}$, $d = q + x$, $t = \frac{a + b + n}{\sin p}$. Виведіть значення змінних A, d, p, t
21	Охарактеризуйте типи алгоритмів	$y = x^2 + k$, $x = z + y_1$, $k = z^2 + y_1^2$, $t = \frac{y}{f + m}$, $m = k + 0{,}2$. Виведіть значення змінних m, x, y, t
22	Назвіть способи опису алгоритмів	$Z = \frac{k^2 + p^3 + n}{\sin y + y}$, $y = 0{,}2x$, $k = y + t^2$, $n = (k + y)^2 + a$. Виведіть значення змінних Z, y, k, n
23	Назвіть етапи підготовки та розв’язання задач за допомогою ЕОМ	$t = \frac{2s + m^2}{a - x} + \sqrt{s + c^3} - x^2$, $s = \frac{a + 2c}{m} + q$, $m = 2\sqrt{a^2 + c}$, $a = 5{,}6$. Виведіть значення змінних t, s, m

Продовження табл. А.1

1	2	3
24	Дайте визначення поняття «лінійний обчислювальний процес»	$Z = \frac{(a+b)(a+c)}{c+d+c_1}$, $d = a + c$, $c_1 = a + b$, $a = 0{,}24$, $c = -4$. Виведіть значення змінних Z, d, c_1
25	Дайте визначення поняття «алгоритм»	$x = y + k$, $k = \frac{d+c}{m+n}$, $m = c^2 + z$, $z = p_1 + p_2$, $c = 0{,}13$, $y = 21{,}2$. Виведіть значення змінних K, m, z
26	Дайте визначення поняття «циклічний алгоритм»	$P = \frac{a+b}{t}$, $t = \frac{x+y}{z}$, $d = P + t + a$, $b = k + m$, $m = -12{,}1$. Виведіть значення змінних P, d, t

Продовження додатка А

Закінчення табл. А.1

1	2	3
27	Дайте визначення поняття «розгалужений алгоритм»	$a = k + b,$ $b = c + d,$ $d = t + c,$ $m = a + b + \frac{c}{d},$ $t = 2m + d,\ c = 3{,}3.$ Виведіть значення змінних a, b, m, t
28	Дайте визначення поняття «лінійний алгоритм»	$Z = e^{\frac{x}{t}} + k,$ $F = z + d,$ $d = x + k,$ $k = a + b,$ $x = 0{,}1.$ Виведіть значення змінних Z, F
29	Дайте стислу характеристику лінійного обчислювального процесу	$S = \frac{7}{m + 2t} + \frac{a}{b + cx} + m^2 - 4z,$ $z = 3m + at^2,$ $m = \sqrt{a + 2cx^2},$ $b = 3{,}25.$ Виведіть значення змінних S, z, m
30	Охарактеризуйте властивості алгоритмів	$V_1 = \frac{v}{v_1 - \frac{a^3}{c^2}},$ $x = v\,t,$ $d = V_1 + a + m + z,$ $m = k + v,$ $z = x + c,\ c = 3.$ Виведіть значення змінних V_1, x

Продовження додатка А

Таблиця А.2

Завдання для побудови розгалуженого обчислювального процесу

Номери варіантів	Теоретичні запитання	Практичні завдання: складіть схеми алгоритмів
1	2	3
1	Який алгоритм називають розгалуженим?	$Q = \begin{cases} 2a - z^x, \text{ якщо } 0 < x < 5 \text{ та } z \neq 0, \\ \sqrt{x + z + a}, \text{ якщо } 0 \leq x < 9 \text{ та } a = 0, \\ a + x + z, \text{ в інших випадках.} \end{cases}$ $S = \begin{cases} 2Q, \text{ якщо } Q > 15, \\ 3Q, \text{ якщо } Q \leq 15. \end{cases}$ Виведіть значення змінних Q, S
2	Які блоки використовують під час побудови розгалуженого алгоритму?	$Z = \begin{cases} ax^3 + b\ln 2x^2, \text{ якщо } x > a, \\ \sqrt[6]{\lvert b + cx^2 - a \rvert}, \text{ якщо } x < a, \\ \lg\left(2x + 3b^2\right), \text{ якщо } x = a. \end{cases}$ $Q = \begin{cases} 2Z + \ln a, \text{ якщо } Z > 153, \\ 2Z + l^x, \text{ якщо } Z \leq 153. \end{cases}$ Виведіть значення змінних Z, Q
3	Що є результатом операції логічного множення?	$Y = \begin{cases} x^2 + a, \text{ якщо } 0 < x < 1{,}5, \\ 2\ln x, \text{ якщо } 1{,}5 < x < 3, \\ \sqrt{a + x}, \text{ якщо } 3 \leq x < 5, \\ 12x^2, \text{ в інших випадках.} \end{cases}$ $Z = \begin{cases} 3Y, \text{ якщо } Y > 3{,}8, \\ 7Y, \text{ якщо } Y \leq 3{,}8. \end{cases}$ Виведіть значення змінних Y, Z
4	Які є типи логічних відносин?	$M = \begin{cases} bx^2 + \ln 2x + x, \text{ якщо } x \geq 5, \\ \lg\lvert x - a \rvert + b\dfrac{c - x}{a - m}, \text{ якщо } x < 5. \end{cases}$ $Q = \begin{cases} e^{x2} + \sqrt{\lvert M - a \rvert}, \text{ якщо } M > 56{,}4, \\ \sin^2 2x + b\sqrt{cx}, \text{ якщо } M < 56{,}4, \\ M + \mathrm{arctg}x, \text{ якщо } M = 56{,}4. \end{cases}$ Виведіть значення змінних M, Q

Продовження табл. А.2

1	2	3
5	У чому полягає сутність операції логічного додавання?	$S = \begin{cases} 2x^m - \ln c^2, \text{ якщо } x^m = 396, \\ x^5 - \lg m, \text{ якщо } x^m = 458, \\ c^5 - \sqrt{x^m}, \text{ якщо } x^m > 458, \\ b + 3mx^m, \text{ якщо } x^m < 458. \end{cases}$ $Q = \ln\frac{s}{m} + x^{m+1},$ $A = \begin{cases} Q^2, \text{ якщо } Q > 0, \\ Q + 1, \text{ якщо } Q \leq 0. \end{cases}$ Виведіть значення змінних Q, S, X^m, A
6	Які логічні операцій можна використовувати в логічних виразах?	$T = \begin{cases} a + 2x^b, \text{ якщо } x + c = 2{,}9, \\ m - 3b^2, \text{ якщо } x + c > 2{,}9, \\ c + \ln x, \text{ якщо } x + c < 2{,}9. \end{cases}$ $Q = \frac{X}{T} + \frac{T}{2M} + \sqrt{a + T},$ $S = \begin{cases} a - t, \text{ якщо } T > Q, \\ b + t, \text{ якщо } T \leq Q. \end{cases}$ Виведіть значення змінних T, Q, S
7	Який вигляд має блок-схема алгоритму логічного множення?	$T = \begin{cases} x^2 - am, \text{ якщо } c > x > a, \\ cx^5 + 2bx, \text{ якщо } x \leq a, \\ \sqrt{b + cx^2}, \text{ якщо } x = c, \\ 0, \text{ в інших випадках}. \end{cases}$ $Z = 3{,}5\ T,$ $Q = \begin{cases} Z + 2T, \text{ якщо } Z \leq 29, \\ \ln(Z + T), \text{ якщо } Z > 29. \end{cases}$ Виведіть значення змінних T, Z, Q
8	Який вигляд має блок-схема алгоритму логічного додавання?	$R = \begin{cases} cx^2 + \sqrt{b} - c, \text{ якщо } 2 < x < 5, \\ \sqrt{cx} + a, \text{ якщо } x \leq 2, \\ b - 2ac, \text{ в інших випадках}. \end{cases}$ $Q = 3R + 2bx.$ $T = \begin{cases} 2Q, \text{ якщо } Q < 12{,}8, \\ Q - 3A, \text{ якщо } Q \geq 12{,}8. \end{cases}$ Виведіть значення змінних R, Q, T

Продовження табл. А.2

1	2	3
9	Що називають логічною умовою?	$R = \begin{cases} a + 2x, \text{ якщо } x < 0 \text{ та } a > 2, \\ b - cx, \text{ якщо } x > 0 \text{ або } a = 0, \\ a^2 - 2\sqrt{x}, \text{ якщо } x = 4 \text{ та } c = 7, \\ \sqrt{b + x}, \text{ в інших випадках.} \end{cases}$ $S = \begin{cases} 2R, \text{ якщо } R < 12, \\ R - C, \text{ якщо } R \geq 12. \end{cases}$ Виведіть значення змінних R, S
10	Які особливості має блок логічної умови?	$R = \begin{cases} cx^2 + \lg x, \text{ якщо } 2 < x < 4, \\ b - cx, \text{ якщо } 4 \leq x < 6, \\ a^3 + \lg x, \text{ якщо } 6 \leq x < 8, \\ cx + b^x, \text{ якщо } x \leq 2 \text{ або } x \geq 8. \end{cases}$ $S = \begin{cases} 2R, \text{ якщо } R = 14{,}2, \\ 3R, \text{ якщо } R \neq 14{,}2. \end{cases}$ Виведіть значення змінних R, S
11	У чому полягає відмінність розгалуженого алгоритму від лінійного?	$Y = \begin{cases} x + 2a^m, \text{ якщо } a + m = 4{,}6, \\ \sqrt{x + 3a}, \text{ якщо } a + m \geq 5{,}2, \\ \ln(a + m), \text{ в інших випадках.} \end{cases}$ $Z = \begin{cases} 2Y, \text{ якщо } Y < 16{,}5, \\ 5Y - C, \text{ якщо } Y \geq 16{,}5. \end{cases}$ $Q = a + 2Y - 3Z.$ Виведіть значення змінних Y, Z, Q
12	Які є типи логічних відношень?	$Y = \begin{cases} 3x, \text{ якщо } 2 < x < 4, \\ 5x, \text{ якщо } 4 \leq x < 9, \\ 7x, \text{ якщо } 9 \leq x \text{ або } x \leq 2. \end{cases}$ $Z = 3Y + 2ax - \sqrt[a]{Y},$ $R = \begin{cases} Z + Y, \text{ якщо } Z \geq 25, \\ Y - Z, \text{ якщо } Z < 25. \end{cases}$ Виведіть значення змінних Y, Z

Продовження додатка А

Продовження табл. А.2

1	2	3
13	Який вигляд має блок-схема алгоритму логічного додавання?	$$Y = \begin{cases} \lvert x \rvert + x^2, \text{ якщо } x < 0, \\ kx + b, \text{ якщо } x = 0, k = 5{,}35 \text{ та } b = 10, \\ \ln^2 x, \text{ якщо } x > 0. \end{cases}$$ $$S = \begin{cases} Y^2, \text{ якщо } Y \le 0, \\ \sqrt{Y}, \text{ якщо } Y > 0. \end{cases}$$ Виведіть значення змінних Y, S
14	Охарактеризуйте таку властивість алгоритму, як «детермінованість»	$$Z = \begin{cases} b + am^x, \text{ якщо } x = 125, \\ c - b + x, \text{ якщо } x = 200, \\ e^x + a, \text{ якщо } 0 < x < 125, \\ 10, \text{ в інших випадках.} \end{cases}$$ $$Q = \frac{a}{Z} + \frac{b}{2Z} - c,$$ $$A = \begin{cases} Q + 3, \text{ якщо } Q > 0, \\ Q - 4, \text{ якщо } Q \le 0. \end{cases}$$ Виведіть значення змінних Z, Q, A
15	Охарактеризуйте таку властивість алгоритму, як «дискретність»	$$Z = \begin{cases} ax^3 + b\ln 2x^2, \text{ якщо } x > a, \\ c\sqrt{\lvert b + cx^2 - a \rvert}, \text{ якщо } x = a, \\ \lg\left(2x + 3b^2\right), \text{ якщо } x = a. \end{cases}$$ $$Q = \begin{cases} 2Z^2 + \ln a, \text{ якщо } Z > 153, \\ 2Z^2 + e^2, \text{ якщо } Z \le 153. \end{cases}$$ Виведіть значення змінних Z, Q
16	Який вигляд має блок-схема алгоритму логічного добутку?	$$Y = \begin{cases} x\dfrac{b + a}{\sin x}, \text{ якщо } 0 < x < 0{,}5, a = 3 \text{ та } b = 7, \\ b + \dfrac{x^2 + 1}{5}, \text{ якщо } x = 0 \text{ або } 0 < b < 5, \\ \dfrac{x - \dfrac{x + 1}{2}}{2x}, \text{ в інших випадках.} \end{cases}$$ $$Q = \begin{cases} Y - a, \text{ якщо } Y \ge 10, \\ Y + b, \text{ якщо } Y < 10. \end{cases}$$ Виведіть значення змінних Y, Q

Продовження табл. А.2

1	2	3
17	Охарактеризуйте таку властивість алгоритму як «масовість»	$Z = \begin{cases} \sqrt{ab + \ln x}, \text{ якщо } bx = 6 \text{ та } a < 2, \\ \sqrt[a]{cx + 5a - c}, \text{ якщо } 6 < bx < 9 \text{ та } a > 2, \\ \ln\lvert b + ax^2\rvert - b, \text{ в інших випадках.} \end{cases}$ $R = \begin{cases} 2Z, \text{ якщо } Z \geq 12, \\ 3Z, \text{ якщо } Z < 12. \end{cases}$ Виведіть значення змінних Z, R
18	Який алгоритм називають розгалуженим?	$Y = \begin{cases} x^{e^{x+a}} + e^{2+a}, \text{ якщо } 1 \leq x < 2 \text{ та } a > 0, \\ \sin x, \text{ якщо } -1 \leq x < 1 \text{ або } 0 < b < 3, \\ \ln\lvert b + ax^2\rvert - b, \text{ якщо } -5 \leq x < 1. \end{cases}$ $P = \begin{cases} Y + b^2, \text{ якщо } Y \leq 0, \\ Y + a^2, \text{ якщо } Y > 0. \end{cases}$ Виведіть значення змінних Y, P
19	Який алгоритм називають циклічним?	$Y = \begin{cases} \ln x^3 + \sqrt{xc}, \text{ якщо } 3 < x < 6 \text{ та } c > 0, \\ \frac{x^2}{a + x} + \sqrt{ax}, \text{ якщо } x > 10 \text{ або } 0 < a < 6, \\ x\ln\lvert x\rvert, \text{ в інших випадках.} \end{cases}$ $Z = \begin{cases} \sqrt{Y}, \text{ якщо } Y > 0, \\ \sqrt[3]{Y}, \text{ якщо } Y \leq 0. \end{cases}$ Виведіть значення змінних Y, Z
20	У чому полягає сутність операції логічного додавання?	$Y = \begin{cases} \sin^2(x + a), \text{ якщо } 0 < x < 1 \text{ або } 1 < a < 2, \\ x^2 + \sqrt{\lvert xab\rvert}, \text{ якщо } -1 \leq x, a < 0, b < 0, \\ \tan x, \text{ в інших випадках.} \end{cases}$ $Z = \begin{cases} 2a + Y, \text{ якщо } Y > 0 \text{ та } a > 0, \\ 2b - Y, \text{ якщо } Y < 0 \text{ або } b < 0. \end{cases}$ Виведіть значення змінних Y, Z
21	У чому полягає особливість способу опису алгоритму за допомогою діаграм прецедентів?	$Y = \begin{cases} \cos^2\lvert xa\rvert, \text{ якщо } 2 < x < 11 \text{ та } 3 < a < 14, \\ \sqrt[3]{b + \sin^2 x}, \text{ якщо } x \geq 11 \text{ або } 2 < b < 13, \\ \text{ctg}x, \text{ якщо } x = -5. \end{cases}$ $P = \begin{cases} \ln Y, \text{ якщо } Y > 0, \\ \ln^2\lvert Y\rvert, \text{ якщо } Y \leq 0. \end{cases}$ Виведіть значення змінних Y, P

Продовження табл. А.2

1	2	3
22	У чому полягає особливість способу опису алгоритму за допомогою псевдокоду?	$Y = \begin{cases} \sqrt[3]{xe^x}, \text{ якщо } 7 < x < 9 \text{ або } a < 1 \text{ та } b > 3, \\ tg\sqrt{x}, \text{ якщо } x = 3, \\ \ln\lvert xab \rvert, \text{ якщо } x = 1 \text{ та } a = 2 \text{ або } b = -1. \end{cases}$ $T = \begin{cases} 2Y, \text{ якщо } Y > 5, \\ 3\lvert Y \rvert, \text{ якщо } Y \le -5. \end{cases}$ Виведіть значення змінних Y, T
23	У чому полягає особливість способу опису алгоритму за допомогою мереж?	$Y = \begin{cases} \frac{a\sqrt{x}}{\sqrt{10b}}, \text{ якщо } 1 < x < 2 \text{ та } a < 7 \text{ або } b = 3, \\ \lvert x \rvert, \text{ якщо } x < -1, \\ \ln x^3, \text{ якщо } x = 0{,}75. \end{cases}$ $P = \begin{cases} Y + \sqrt{\lvert a \rvert}, \text{ якщо } Y > 0 \text{ або } a < 0, \\ Y, \text{ якщо } Y = 0. \end{cases}$ Виведіть значення змінних Y, P
24	У чому полягає особливість способу опису алгоритму за допомогою граф-схем?	$Y = \begin{cases} \frac{2^{-x}\sqrt{x + \lvert y_1 \rvert}}{2}, \text{ якщо } 1 < x < 10, \\ \sin z + \sqrt{e^x}, \text{ якщо } 0 < x \le 1, \\ x^2 + 4, \text{ в інших випадках.} \end{cases}$ $S = \begin{cases} 2Y + \ln a, \text{ якщо } Y > 2{,}2, \\ 2Y^2, \text{ якщо } Y \le 2{,}2. \end{cases}$ Виведіть значення змінних Y, S
25	У чому полягає особливість вербально-формульного способу опису алгоритму?	$Q = \begin{cases} ctg^2 x, \text{ якщо } 0 \le x < 1, \\ tgx, \text{ якщо } 1 \le x < 2 \text{ або } x = 3, \\ \ln x, \text{ якщо } 4 < x < 5 \text{ та } 9 < x < 10. \end{cases}$ $T = QZ,$ $A = \begin{cases} T^2, \text{ якщо } T > 0, \\ \lvert T + 1 \rvert, \text{ якщо } T \le 0. \end{cases}$ Виведіть значення змінних Q, T, A
26	Який алгоритм називають розгалуженим алгоритмом?	$F = \begin{cases} e^x + a, \text{ якщо } -3 \le x < 3, \\ \ln x^2 + b, \text{ якщо } 5 \le x < 7, \\ \lvert x \rvert, \text{ якщо } x = -10 \text{ або } -20 < x < -11. \end{cases}$ $P = F + T,$ $Z = \begin{cases} \cos P, \text{ якщо } P < 0, \\ \ln(P), \text{ якщо } P > 0, \\ 5, \text{ якщо } P = 0. \end{cases}$ Виведіть значення змінних F, P, Z

Закінчення табл. А.2

1	2	3
27	У чому полягає особливість блок-схемного способу опису алгоритму?	$Q = \begin{cases} zx + ax, \text{ якщо } 1 < x < 3 \text{ та } 0 < z < 1, \\ \frac{z}{4x} + ax, \text{ якщо } 3 < x < 12, \\ a + z^2, \text{ в інших випадках.} \end{cases}$ $S = \begin{cases} Q^3, \text{ якщо } Q > 17, \\ Q^2, \text{ якщо } Q \le 17. \end{cases}$ Виведіть значення змінних Q, S
28	Які є способи опису алгоритмів?	$T = \begin{cases} \sqrt{bxa}, \text{ якщо } x < 1 \text{ та } 2 < b < 3 \text{ та } a < 5, \\ \cos\frac{bx}{a}, \text{ якщо } 1 < x \text{ або } 0 < b \le 2 \text{ та } a \ne 0, \\ a + b + x, \text{ в інших випадках.} \end{cases}$ $D = \begin{cases} 2T, \text{ якщо } T \ge 10, \\ 3\sqrt{\lvert T \rvert}, \text{ якщо } T < 10. \end{cases}$ Виведіть значення змінних T, D
29	Який алгоритм називають циклічним алгоритмом?	$Q = \begin{cases} 10a - zx, \text{ якщо } 0 < x < 1 \text{ та } 0 < z < 1, \\ x^2 + \sqrt{x}, \text{ якщо } -2 < x < 1 \text{ або } 3 < x < 4, \\ a + x - z^2, \text{ якщо } -4 < z < 3 \text{ та } x > 10. \end{cases}$ $T = \begin{cases} Q^2, \text{ якщо } Q > 0, \\ \cos(Q), \text{ якщо } Q = 0, \\ \lvert Q \rvert, \text{ якщо } Q < 0. \end{cases}$ Виведіть значення змінних Q, T
30	Які є типи структур алгоритмів?	$R = \begin{cases} \sqrt{bx + c^2}, \text{ якщо } 1 < x < 2 \text{ та } 1 < b < 2, \\ \sqrt{b + xb}, \text{ якщо } -1 < x < 0 \text{ та } 3 < b < 4, \\ c\lvert x \rvert + b, \text{ якщо } x < -10 \text{ та } 7 < b < 8. \end{cases}$ $Q = \frac{r}{5} + 10\,bx,$ $A = \begin{cases} 10\,Q, \text{ якщо } Q > 15, \\ 20\,Q, \text{ якщо } Q \le 15. \end{cases}$ Виведіть значення змінних R, Q, A

Продовження додатка А

Таблиця А.3

Завдання для побудови циклічного обчислювального процесу

Номери варіантів	Теоретичні запитання	Практичні завдання: складіть схеми алгоритмів
1	2	3
1	Який алгоритм називають циклічним?	$R=\frac{k!+1}{t+2k}+\prod_{i=1}^{5} i^3+\frac{t^k}{2\sum_{i=1}^{3}(i+a)^2}$. Виведіть значення змінних R, i
2	Який цикл є вкладеним циклом?	$Z=\frac{2m!+3k!+\sqrt{m!+k!}}{\prod_{i=1}^{m}(i+m)^2+5\sum_{i=3}^{7}(m+k+i)^3}$. Виведіть значення змінних Z, Q
3	Який цикл є зовнішнім циклом?	$K=\begin{cases}\ln x_1, \text{ якщо } 0<x_1\le 3,\\ \cos x_1+a, \text{ якщо } -3<x_1\le 0 \text{ або } x_1=-3{,}5,\\ \tan x_1+b, \text{ якщо } x_1=4 \text{ або } x_1=5.\end{cases}$ $Q=Kx_1$, якщо $x_1 \in -10{,}1 \ldots 10{,}1$ із кроком 0,05. Виведіть значення змінних K, Q
4	У чому полягає особливість ітераційного циклу?	$L=\frac{1}{2x}+\frac{2}{m!}\frac{(m+2)!}{2+a+x}$, $S=x\prod_{i=10}^{20}(i-m)^2+3{,}45\,L$, $x\in[0{,}1;\,10]$, $\Delta x=0{,}4$. Виведіть значення змінних L, S, x
5	Опишіть особливість блок-схемного опису алгоритму?	$Q=\frac{a+x^5}{(m+2)!}+\sqrt{\frac{m!}{x+3}}+\mathrm{Ln}(m!)$, $x\in[1{,}5]$, $\Delta x=0{,}2$. Виведіть значення змінних Q, m

Продовження табл. А.3

1	2	3
6	Яка змінна є параметром циклу?	$R = \frac{x^2 + \lg(m! + 3k!)}{\sqrt[3]{x + m! + k! + a^x}}$, $x \in [3,2; 6,1]$, $\Delta x = 0,1$. Виведіть значення змінних R, x
7	Які є етапи організації циклів?	$S = \frac{2,31\ y\left(\prod_{i=1}^{11}(i + b)\right)^2 + 3x}{\sum_{j=1}^{3}(j + mc) + 2,5x^2}$, $x \in [3,12]$, $\Delta x = 0,5$. Виведіть значення змінних x, S
8	У чому полягає особливість арифметичного циклу?	$Z = \frac{(m! + 2k!) - x\sqrt{2k + 1}}{\prod_{i=1}^{3}(i + 2)^2 + \ln(m! + 4) + e^x}$, $x \in [m,k]$, $\Delta x = 0,2\ m$. Виведіть значення змінних x, Z
9	Які існують типи обчислювальних процесів?	$Q = \frac{mb^2 + 2x}{\prod_{i=1}^{7}(i + 2)} - \frac{\sqrt{k! + m!}}{m + 5x}$, $Z = \frac{k + m + 2q}{m + c + k} + 2$, $x \in [1,5]$, $\Delta x = 0,2$. Виведіть значення змінних Q, Z, x
10	У чому полягає особливість способу опису алгоритму за допомогою граф-схем?	$F = \frac{\sqrt[3]{(m + a)!} + 2,2\ x^2 + \sin x}{\lg 2x + \lg 3x}$, $Z = \sqrt{f} + \sum_{i=3}^{9}\left(i + a^2\right) + \sqrt{x}$, $R = a + z - 13,5x$ $x \in [2,9]$, $\Delta x = 0,1$. Виведіть значення змінних F, R, Z, x

Продовження табл. А.3

1	2	3
11	У чому полягає особливість способу опису алгоритму за допомогою моделі даних?	$y = \frac{\sqrt{m! + 2k} + \prod_{i=2}^{5}(i + 2k)^2}{3\sum_{i=1}^{7}(a^i + 2a + k!) + 2x}$, $x \in [3, k]$, $\Delta x = 0{,}5k$. Виведіть значення змінних x, y
12	У яких випадках використовують укладені цикли?	$Q = \frac{mb^2 + 2x}{\prod_{i=1}^{7}(i + 2)} - \frac{\sqrt{k! + m!}}{m + 5}$, $Z = \frac{k + m + 2q}{m + c + k!} + 2^x$, $x \in [a,c]$, $\Delta x = 0{,}15a$. Виведіть значення змінних Q, Z, x
13	У яких випадках використовують циклічні алгоритми?	$R = \frac{k! + 1}{t + 2k} + \prod_{t_1=1}^{5} t_1 3 + \frac{t^k}{2\sum_{i=1}^{3}(i + a)^2}$, $t \in [12,15]$, $\Delta t = 0{,}5$. Виведіть значення змінних R, t
14	Яку змінну називають лічильником циклу?	$Q = \frac{x\sqrt{m + k!} + a\ln(m! + k)}{2x + \sin x^2 + \cos m}$, $x \in [7,17]$, $\Delta x = 0{,}7$. Виведіть значення змінних x, Q
15	Які є умови виходу із циклу?	$R = \sqrt{\prod_{i=2}^{9}\frac{i + a}{2} + 3f^3}$, $Z = \ln(a + R) + \log\left(\sum_{i=3}^{5} i^3\right)$, $Q = R + fZ - 5a$, $f \in [1,10]$, $\Delta f = 0{,}2a$. Виведіть значення змінних R, Z, Q, f

Продовження табл. А.3

1	2	3
16	Які є типи циклів?	$y=\frac{l!+n!}{\prod_{i=1}^{3} i^4+\sum_{i=1}^{n}\frac{i^2+a}{i+10}}.$ Виведіть значення змінної y
17	У яких випадках використовують укладені цикли?	$A=\frac{k!+\sum_{i=1}^{10}(i+a)}{k+3}+\frac{\prod_{j=1}^{3} j^3}{m!}+\frac{k}{2\sum_{i=1}^{4}(i+4)^2}.$ Виведіть значення змінної A
18	Якими є етапи організації циклу?	$y=\frac{k^{x+1}}{\prod_{i=1}^{3} i^2+x}+\frac{m!+x^5+(m+2)!}{k!+2x+3\sum_{i=2}^{7} i^5},$ $x\in[4,7]$, $\Delta x=0{,}25$. Виведіть значення змінних x, y
19	Які типи алгоритмів вам відомі?	$P=\frac{(t+1)!}{\sum_{i=1}^{k} i^2}+\prod_{i=1}^{m} i^3+\frac{k!}{m!}.$ Виведіть значення змінної P
20	У чому полягає особливість способу опису алгоритму за допомогою моделі даних?	$D=\frac{\sum_{k=1}^{5}\left(k^2+a\right)}{t+k}+\sum_{i=1}^{n}\frac{k!i^2}{n}.$ Виведіть значення змінної D
21	У чому полягає особливість циклічного обчислюваного процесу?	$S=\frac{m!}{\sum_{i=1}^{m}\frac{i}{a+i}}+d!+\frac{\prod_{j=1}^{k!} j^2}{k!}.$ Виведіть значення змінної S
22	У чому полягає особливість способу опису алгоритму за допомогою діаграми прецедентів?	$D=\frac{\sum_{i=15}^{25}\left(i^2+a\right)}{\prod_{j=1}^{30}\left(j^3+b\right)}.$ Виведіть значення змінної D

Продовження табл. А.3

1	2	3
23	У чому полягає особливість вербально-формульного способу опису алгоритму?	$P = \frac{1}{k!} + \frac{3}{x} + \frac{(m+k)!}{x+a}$, $S = x\sum_{i=1}^{13}(i-m)^2 + 17{,}5P$, $x \in [1{,}11]$, $\Delta x = 0{,}7$. Виведіть значення змінних P, S, x
24	Що таке «логічна складність» алгоритму?	$D = \frac{(a+b)!}{P_{10}+a} + \frac{\sum_{i=1}^{7} i^4}{\prod_{i=1}^{5}(i+a)^2}$. Виведіть значення змінної D
25	Які є принципи побудови алгоритмів?	$D = \frac{b!+2}{2b+2} + \sum_{i=1}^{5}(i+a) + \frac{\prod_{i=1}^{6} j^4}{m!}$. Виведіть значення змінної D
26	Що є метою аналізу трудомісткості алгоритмів?	$A = \frac{x \times m!}{\sum_{i=5}^{15}(i+m)} + \frac{x^2}{k!}$, $x \in [0{,}1;\ 3{,}1]$. Виведіть значення змінних A, x
27	Що називають функцією трудомісткості алгоритмів?	$P = \frac{m!}{k!} + \frac{(a+b)!}{\sum_{i=1}^{5}(i^2+a+b)} + \frac{\sum_{i=1}^{k!}(i+1)}{k!}$. Виведіть значення змінної P
28	У чому полягає особливість вербально-формульного способу опису алгоритму?	$Q = \frac{t!}{k!} + \frac{k!}{\sum_{i=1}^{n}\left(i^2+k^2\right)} + \frac{\prod_{j=1}^{n}(j+t)}{t!}$, Виведіть значення змінної Q

Закінчення табл. А.3

1	2	3
29	Що є метою асимптотичного аналізу трудомісткості алгоритмів?	$C = \frac{(a+b)! + \sum_{i=1}^{a!} i^2}{\prod_{i=1}^{k} j^3 + k!}$, Виведіть значення змінної C
30	Які є оцінки складності алгоритмів?	$M = \begin{cases} a + bx, \text{ якщо } 0 < x \leq 5{,}5, \\ \log x, \text{ якщо } 6 < x \leq 7, \\ a - bx, \text{ якщо } x < 0 \text{ або } x = 10, \\ abx, \text{ в інших випадках.} \end{cases}$ $P = mx$, $x \in [-0{,}8; 10]$, $\Delta x = 0{,}25$. Виведіть значення змінних M, P, x

Додаток Б

Варіанти індивідуальних завдань для роботи з масивами

Таблиця Б.1

Завдання для опрацювання одновимірного масиву

Номери варіантів	Зміст завдань
1	2
1	У масиві B розмірністю n визначте кількість елементів, які не перебільшують числа A
2	У масиві C розмірністю n визначте кількість елементів, які є не меншими від числа A
3	У масиві A розмірністю n визначте номер елемента із максимальним значенням
4	У масиві D розмірністю n визначте номер елемента із мінімальним значенням
5	У масиві K розмірністю n визначте елемент із мінімальним значенням
6	У масиві M розмірністю n визначте елемент із максимальним значенням
7	У масиві K розмірністю n визначте суму усіх елементів
8	У масиві Z розмірністю n визначте добуток усіх елементів
9	У масиві L розмірністю n визначте добуток усіх елементів, які є більшими за середньоарифметичне значення всіх елементів масиву
10	У масиві G розмірністю n визначте добуток усіх елементів, які є не більшими за середньоарифметичне значення всіх елементів масиву
11	У масиві S розмірністю n визначте кількість елементів, які є більшими за середньоарифметичне значення всіх елементів масиву
12	У масиві W розмірністю n визначте кількість елементів, які є меншими за середньоарифметичне значення всіх елементів масиву.
13	У масиві P розмірністю n визначте кількість від'ємних елементів
14	У масиві Q розмірністю n визначте кількість додатних елементів
15	Виведіть номера, за якими розташовано від'ємні елементи в масиві R розмірністю n
16	У масиві Q розмірністю n визначте суму всіх елементів, розміщених за парними номерами
17	У масиві R розмірністю n визначте суму всіх елементів, розміщених за непарними номерами
18	У масиві V розмірністю n визначте добуток всіх елементів, розміщених за номерами, що не є кратними числу a
19	У масиві P розмірністю n визначте добуток всіх елементів, розміщених за номерами, що є кратними числу a
20	У масиві R розмірністю n визначте різницю між мінімальним і максимальним елементом
21	Виведіть номера, за якими розташовано додатні елементи в масиві R розмірністю n

Закінчення додатка Б

Закінчення табл. Б.1

1	2
22	У масиві S розмірністю n визначте номери елементів, які є більшими за середньоарифметичне значення всіх елементів масиву
23	Визначте різницю між добутками від'ємних і додатних елементів. Масив C(16)
24	У масиві S розмірністю n визначте номери елементів, які є меншими за середньоарифметичне значення всіх елементів масиву
25	У масиві B розмірністю n визначте номери елементів, які не перебільшують числа A
26	У масиві D розмірністю n визначте номери додатних елементів, які не перебільшують числа A
27	У масиві D розмірністю n визначте добуток елементів, значення яких належать інтервалу (l, k)
28	У масиві M розмірністю n визначте номери елементів, значення яких не належать інтервалу (l, k)
29	У масиві F розмірністю n визначте номери елементів, які мають парні значення
30	У масиві H розмірністю n визначте добуток усіх елементів, які мають непарні значення

Зміст

Вступ ... 3
Розділ 1. Основи теорії алгоритмів ... 5
1. Поняття алгоритму ... 5
1.1. Теорія алгоритмів як математична наука ... 5
1.2. Визначення алгоритму та вимоги до нього ... 5
1.3. Типи алгоритмів та форми їхнього подання ... 7
1.4. Принципи побудови алгоритмів ... 9
1.5. Математичні основи аналізу складності алгоритмів ... 10
1.5.1. Асимптотичний аналіз ... 11
1.5.2. Часова складність алгоритму ... 13
1.5.3. Асимптотична складність ... 15
Контрольні запитання і завдання для самоперевірки ... 22
Лабораторна робота 1. Побудова та аналіз алгоритмів обчислювальних процесів ... 23
Контрольні запитання до лабораторної роботи 1 ... 46
2. Елементарні структури даних ... 47
2.1. Структурування й абстракція під час написання програм ... 47
2.2. Концепція структур даних ... 50
2.3. Класифікація структур даних ... 52
2.4. Операції над структурами даних ... 54
2.5. Прості структури даних ... 55
2.5.1. Арифметичний тип даних ... 55
2.5.2. Перелічуваний тип даних ... 56
2.5.3. Покажчики ... 56
2.6. Статичні структури даних ... 57
2.6.1. Масиви ... 58
2.6.2. Розріджені масиви ... 59
2.6.3. Множини ... 60
2.6.4. Структури ... 61
2.6.5. Бітові типи ... 62
2.6.6.Таблиці ... 64
2.7. Напівстатичні структури даних ... 65
2.7.1. Стеки ... 66
2.7.2. Черги ... 68
2.7.3. Деки ... 71
2.7.4. Лінійні списки ... 72

2.7.5. Мультисписки 78
2.7.6. Рядки (Strings) 79
2.8. Динамічні структури даних 81
2.9. Нелінійні структури даних 83
2.9.1. Графи 83
2.9.2. Дерева 84
2.10. Алгоритми сортування 84
2.10.1. Сортування вставлянням (простим додаванням).... 85
2.10.2. Алгоритм сортування Шелла 91
2.10.3. Алгоритм бульбашкового сортування.................... 94
Контрольні запитання і завдання для самоперевірки 96
Лабораторна робота 2. Програмування елементарних структур даних 97
Контрольні запитання до лабораторної роботи 2 105
3. Двійкові дерева пошуку 106
3.1. Поняття дерева. Елементи дерев 106
3.2. Основні операції з деревами 109
3.3. Бінарні дерева пошуку 112
3.3.1. Структура бінарного дерева 113
3.3.2. Пошук у бінарних деревах 116
3.3.3. Пошук мінімуму й максимуму 118
3.3.4. Пошук попереднього та наступного елементів 119
3.3.5. Уставляння та вилучення в бінарному дереві........ 119
Контрольні запитання і завдання для самоперевірки 123
Лабораторна робота 3. Бінарні дерева пошуку 124
Контрольні запитання до лабораторної роботи 3 134
Розділ 2. Алгоритмізація розв’язання прикладних задач 135
4. Геш-таблиці 135
4.1. Пряма адресація 135
4.2. Поняття геш-таблиці 137
4.3. Розв’язання колізій за допомогою ланцюгів 138
4.4. Аналіз гешування з ланцюгом 140
4.5. Побудова ефективних геш-функцій 141
4.5.1. Гешування методом ділення із залишком 143
4.5.2. Гешування методом множення 144
4.5.3. Універсальне гешування 145
4.6. Відкрита адресація 147
4.6.1. Лінійний метод обчислення послідовностей проб 149

4.6.2. Квадратичний метод обчислення послідовностей проб 150
4.6.3. Подвійне гешування 151
4.6.4. Аналіз гешування з відкритою адресацією 152
4.6.5. Ідеальне гешування 153
Контрольні запитання і завдання для самоперевірки 155
5. Основні алгоритми на графах 156
5.1. Подання графів 156
5.2. Пошук у ширину 158
5.2.1. Аналіз алгоритму пошуку в ширину 162
5.2.2. Найкоротші шляхи 162
5.2.3. Дерева пошуку в ширину 164
5.3. Пошук у глибину 165
5.3.1. Властивості пошуку в глибину 168
5.3.2. Класифікація ребер 171
5.3.3. Топологічне сортування 172
5.3.4. Компоненти сильної зв’язності 174
5.4. Алгоритм Дейкстри 180
5.5. Алгоритм Флойда – Воршелла 182
Контрольні запитання і завдання для самоперевірки 184
Лабораторна робота 4. Алгоритми Дейкстри та Флойда – Воршелла 184
Контрольні запитання до лабораторної роботи 4 200
6. Потоки в мережах 200
6.1. Поняття мережі. Основні визначення 200
6.2. Задача про максимальний потік у мережі 202
6.2.1. Теореми про оптимальні потоки в мережах 203
6.2.2. Метод розставляння міток для визначення максимального потоку 205
6.2.3. Модифікований метод розставляння міток для визначення максимального потоку 209
6.2.4. Алгоритм Форда – Фалкерсона визначення максимального потоку 210
Контрольні запитання і завдання для самоперевірки 222
Використана та рекомендована література 223
Додатки 225

НАВЧАЛЬНЕ ВИДАННЯ

Солодовник Ганна Валеріївна
Шаповалова Олена Олександрівна

РОЗРОБКА ТА АНАЛІЗ АЛГОРИТМІВ

Навчальний посібник

Самостійне електронне текстове мережеве видання

Відповідальний за видання *О. В. Старкова*

Відповідальний редактор *О. С. Вяткіна*

Редактор *О. Г. Доценко*

Коректор *О. Г. Доценко*

План 2023 р. Поз. № 12-ЕНП. Обсяг 251 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.