МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

Є. Ю. БЕРКУНСЬКИЙ, А. Ю. ПАВЛЕНКО

СТРУКТУРИ ТА ОРГАНІЗАЦІЯ ДАНИХ МОВОЮ KOTLIN

Навчальний посібник

Рекомендовано Вченою радою НУК

ВИДАВНИЦТВО
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КОРАБЛЕБУДУВАННЯ
ІМ. АДМІРАЛА МАКАРОВА

2025

УДК 004.43/004.62
Б48

Автори:
Є. Ю. Беркунський, старш. викладач;
А. Ю. Павленко, старш. викладач

Рецензенти:
М. В. Ушкац, д-р фіз.-мат. наук, проф., завідувач кафедри фізики та математики Національного університету кораблебудування імені адмірала Макарова;
О. Б. Данченко, д-р техн. наук, проф. кафедри комп'ютерних наук та системного аналізу Черкаського державного технологічного університету;
І. М. Журавська, д-р техн. наук, проф., завідувачка кафедри комп'ютерної інженерії Чорноморського національного університету імені Петра Могили

Рекомендовано Вченою радою НУК ім. адмірала Макарова як навчальний посібник
(*протокол № 04 від 29.05.2024 р.*)

Беркунський Є. Ю.
Б48 Структури та організація даних мовою Kotlin : навчальний посібник / Є. Ю. Беркунський, А. Ю. Павленко. – Миколаїв : НУК, 2025. – 262 с.

ISBN 978-966-321-478-8

Вміщено теоретичні відомості та довідкова інформація для вивчення абстрактних структур даних, їхньої реалізації за допомогою мови програмування Kotlin. Розглянуто основні задачі, для розв'язання яких використання класичних структур даних дозволяє отримати ефективні за часом та простором алгоритми.

Призначено для студентів спеціальностей F3 Комп'ютерні науки, F4 Системний аналіз та наука про дані, F6 Інформаційні системи і технології усіх форм навчання. Може бути корисним для самостійного вивчення структур даних.

УДК 004.43/004.62

ISBN 978-966-321-478-8

ЗМІСТ

Вступ

Розробка ефективних алгоритмів є дуже важливою навичкою, яку прагнуть усі компанії, що займаються розробкою програмного забезпечення. Більшість співбесід для компаній-розробників зосереджені на знаннях структур даних і алгоритмів.

Окрім мов програмування, сучасному фахівцю також потрібно добре володіти фундаментальними комп'ютерними знаннями та вміннями, щоб не лише пройти співбесіду, але й досягти успіху у своїй роботі розробника.

Поняття абстрактних структур даних є основним для розуміння більшості сучасних алгоритмів.

Абстрактний тип даних (АТД) – це логічний опис даних та їх операцій. АТД можна розглядати як точку зору користувача щодо даних. АТД описує можливі значення даних та інтерфейсів доступу до них. АТД не описує та не забезпечує фактичну реалізацію структури даних.

Наприклад, користувач хоче зберегти кілька цілих чисел і знайти їх середнє значення. АТД для цієї структури даних матиме дві функції, одну для додавання цілих чисел, а іншу для отримання середнього значення. АТД для цього випадку не говорить про те, як саме вона буде реалізована.

Структури даних – це конкретні представлення даних, визначені з точки зору програміста. Структура даних представляє, яким чином дані можуть зберігатися в пам'яті. Кожна структура даних має свій набір переваг і недоліків. Залежно від типу задачі програміст може обирати структуру даних, що підходить найкраще.

Навчальний посібник складається з дев'яти тем. Перша тема присвячена розгляду реалізацій абстрактних типів даних

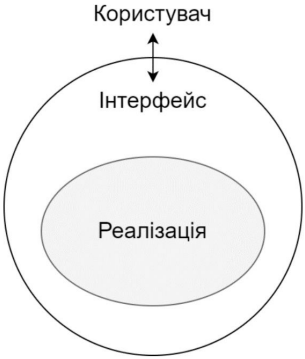


Рис. 1. Абстрактний тип даних

у стандартній бібліотеці мови Kotlin. У другій темі розглядаються основні алгоритми сортування масивів та списків. Третя тема містить опис абстрактної структури даних «Зв'язаний список» та основні концепції реалізації її засобами мови Kotlin. Четверта та п'ята теми присвячені структурам «Стек» та «Черга», які мають багато спільних рис та використовуються у прикладних програмах як інструмент для побудови алгоритмів. У шостій темі розглядаються деревоподібні структури даних та основна увага приділяється бінарним деревам. Розглядаються проблеми балансування дерев та використання бінарних дерев пошуку. Сьома тема містить опис реалізацій пріоритетних черг, основною з яких є двійкова (бінарна) купа. Структура даних «Хеш-таблиця» є змістом восьмої теми. Розглядаються методи організації хеш-таблиць на основі методів відкритого хешування та розділених ланцюжків. У останній, дев'ятій темі розглядається структура «Граф». Ця тема узагальнює матеріал попередніх тем, адже більшість відомих алгоритмів на графах використовує структури даних: стек, чергу, пріоритетну чергу, хеш-таблиці тощо.

Кожна тема складається із теоретичних відомостей про структуру, що розглядається, та розгляду задач, для вирішення яких ефективно її використовувати.

Тема 1. **Kotlin Collections Framework**

Мова програмування Kotlin поставляється зі стандартною бібліотекою Kotlin Collections Framework, яка є набором високоякісних, високопродуктивних та багаторазово використовуваних структур даних та алгоритмів.

Переваги використання Kotlin Collections Framework:

1. Програмістам не потрібно багаторазово впроваджувати базові структури даних і алгоритми. Таким чином, це запобігає повторному винаходу колеса. Програміст може більше зусиль приділяти бізнес-логіці.

2. Код Kotlin Collections Framework є добре протестованим, високоякісним і високопродуктивним кодом. Використання його підвищує якість програм.

3. Вартість розробки зменшується, оскільки базові структури даних і алгоритми реалізовані в Collections Framework.

4. Легко переглядати та розуміти програми, написані іншими розробниками, тому що більшість розробників Kotlin використовують Collections Framework. Крім того, Kotlin Collections Framework добре задокументований.

Масиви

Масиви – це найпростіші структури даних, які зберігають елементи одного типу даних.

Операції з АТД «Масив»:

1. Додати елемент на k-ту позицію. Значення можна зберегти в масиві на k-й позиції за O(1) – константний час. Просто потрібно зберегти значення в arr[k].

2. Зчитування значення, збереженого в k-й позиції. Доступ до значення, яке зберігається за деяким індексом у масиві також є O(1) – константним часом. Знову, просто потрібно прочитати значення, що зберігається в arr[k].

3. Заміна значення, що зберігається в k-й позиції, на нове значення. *Часова складність*: O(1) – *константний час.*

Розглянемо приклад:

```
fun main() {
    val array = IntArray(10)
    for (i in 0..9) {
        array[i] = i
    }
    for (i in array){
        print(" $i")
    }
}
```

Результат:

```
0 1 2 3 4 5 6 7 8 9
```

ArrayList

ArrayList<E> у Kotlin Collections Framework – це структура даних, що реалізує інтерфейс List<E>, таким чином у ньому можуть бути повторювані елементи. ArrayList – це реалізація динамічного масиву, який може зростати або зменшуватися за потреби. (Усередині ArrayList використовується масив, коли він буде заповнений, виділяється новий, більший, масив і старі значення масиву копіюються до нього).

```
fun main() {
    val arrayList = ArrayList<Int>()
    arrayList.add(1) // додати 1 в кінець списку
    arrayList.add(2) // додати 2 в кінець списку
    println("Array : $arrayList")
    println("Array Size : " + arrayList.size)
    println("Array IsEmpty : " + arrayList.isEmpty())
    arrayList.removeAt(arrayList.size - 1) // останній елемент видалено.
    println("Array : $arrayList")
    arrayList.removeLast() // останній елемент масиву видалено.
    println("Array IsEmpty : " + arrayList.isEmpty())
}
```

Linked List

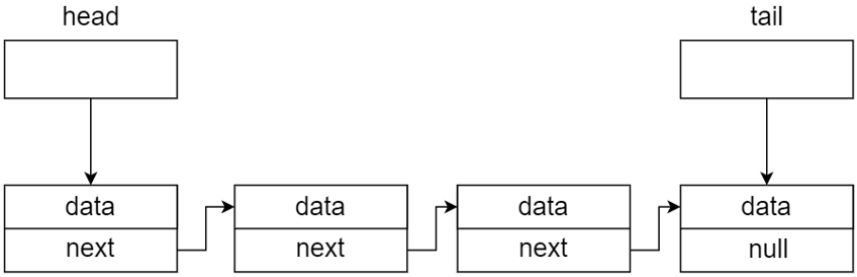


Рис. 1.1. Linked List

Зв'язаний список – це динамічна структура даних, у якій пам'ять виділяється під час виконання. Концепцією зв'язаного списку є не зберігати дані безперервно. Вузли зв'язаного списку містять посилання, які вказують на наступні елементи у списку.

З точки зору продуктивності, зв'язані списки повільніші за масиви, оскільки немає прямого доступу до елементів зв'язаного списку. Зв'язаний список є корисною структурою даних, коли ми завчасно не знаємо кількості елементів, що зберігатимуться. Існує багато типів зв'язаних списків: лінійний, циклічний, подвійний, подвійний круговий тощо.

Операції з LinkedList

Розглянемо API зв'язаного списку:

1. Insert(k) – вставити елемент на початку списку. Просто створюється новий елемент і переміщується покажчик. Таким чином, цей новий елемент стане першим елементом списку. Ця операція займе O(1) – константний час.

2. Delete() – видалити елемент на початку списку. Для цього потрібно лише перемістити один покажчик. Ця операція також займе O(1) – константний час.

3. Print() – відобразити всі елементи списку. Починаючи з першого елемента, а потім слідувати за покажчиками. Ця операція займе O(N) часу.

4. Find(k) – знайти позицію елемента зі значенням k. Починаючи з першого елемента, переміщуватись за покажчиками, поки не буде отримано потрібне значення або не досягнуто кінця списку. Ця операція займе O(N) часу.

5. IsEmpty() – перевірити, чи дорівнює нулю кількість елементів у списку. Просто перевірити головний покажчик списку, якщо він має значення null, тоді список порожній, інакше не порожній. Ця операція займе O(1) – константний час.

Реалізація LinkedList у Kotlin Collections Framework

LinkedList<E> від Java Collections – це структура даних, яка також реалізує інтерфейс List<E>.

```
import java.util.LinkedList
fun main() {
    val list = LinkedList<Int>()
    list.addFirst(2) // 2 додається у початок списку
    list.addLast(10) // 10 додається у кінець списку
    list.addFirst(1) // 1 додається у початок списку
    list.addLast(11) // 11 додається у кінець списку
    println("Contents of Linked List: $list")
    list.removeFirst()
    list.removeLast()
    println("Contents of Linked List: $list")
}
```

Результат:

```
Contents of Linked List: [1, 2, 10, 11]
Contents of Linked List: [2, 10]
```

Аналіз: значення додаються в кінці та на початку пов’язаного списку. Після цього, значення, що зберігаються у зв’язаному списку, друкуються.

Потім значення видаляються зі зв’язаного списку з початку та з кінця та знову вміст списку друкується на екрані.

Стек

Стек – це структура даних, яка дотримується принципу: «Останній прийшов – першим вийшов» (LIFO). Це означає, що елемент, доданий останнім, буде видалено першим.

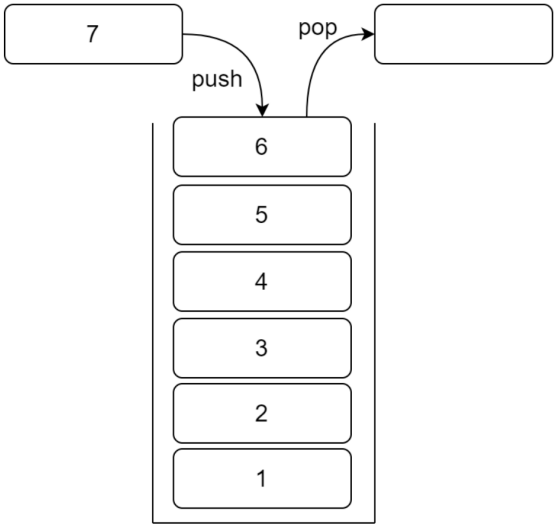


Рис. 1.2. Стек

Операції зі стеком

Розглянемо API стека:

1. Push(k) – додати значення k у верхню частину стека.
2. Pop() – видалити елемент із вершини стека та повернути його значення.
3. Top() – повернути значення елемента на вершині стека.
4. Size() – повернути кількість елементів у стеку.
5. IsEmpty() – повідомити нам, порожній стек чи ні. Він повертає true, якщо стек порожній, інакше повертає false.

Примітка. Усі наведені вище операції зі стеком мають складність O(1) – константний час.

Реалізація стека в Kotlin Collections Framework

Стек реалізується шляхом виклику методів push і pop класу Stack<T>.

```
import java.util.*

fun main() {
    val stack = Stack<Int>()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    println("Stack : $stack")
    println("Stack size : " + stack.size)
    println("Stack pop : " + stack.pop())
    println("Stack top : " + stack.peek())
    println("Stack isEmpty : " + stack.isEmpty())
}
```

Результат:

```
Stack : [1, 2, 3]
Stack size : 3
Stack pop : 3
Stack top : 2
Stack isEmpty : false
```

Стек також можна реалізувати шляхом виклику методів push і pop класу ArrayDeque<T>.

Черга

Черга – це структура даних, яка відповідає принципу: «Першим прийшов – першим вийшов» (FIFO). Перший доданий елемент до черги першим буде видалено, і навпаки.

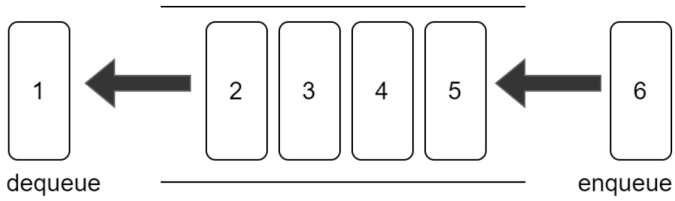


Рис. 1.3. Черга

Операції з чергами

Розглянемо API черги:

1. Add(k) – додати елемент k у кінець черги.
2. Remove() – видалити перший елемент на початку черги та повернути його значення.
3. Front() – повернути значення елемента на початку черги.
4. Size() – повернути кількість елементів у черзі.
5. IsEmpty() – перевірити, порожня черга чи ні. Якщо вона порожня, повертає true, інакше повертає false.

Примітка. Усі наведені вище операції з чергою мають складність O(1) – константний час.

Реалізація черги в Kotlin Collections Framework

ArrayDeque<T> – це реалізація класу двосторонньої черги. Якщо ми використовуємо add(), removeFirst() і first(), він буде поводитися як черга. Більше того, якщо ми використовуємо add(), removeLast() і last(), він поводиться як стек.

```
fun main() {
    val queue = ArrayDeque<Int>()
    queue.add(1)
    queue.add(2)
    queue.add(3)
    println("Queue : $queue")
    println("Queue size : " + queue.size)
    println("Queue peek : " + queue.first())
    println("Queue remove : " + queue.removeFirst())
    println("Queue isEmpty : " + queue.isEmpty())
}
```

Результат:

```
Queue : [1, 2, 3]
Queue size : 3
Queue peek : 1
Queue remove : 1
Queue isEmpty : false
```

Аналіз: значення додаються до черги та друкуються на екрані. Оскільки черга працює за принципом: «Перший прийшов – перший вийшов», значення, які додаються першими, першими виходять із черги.

Дерево

Дерево – це структура даних, організованих в ієрархію. Кожен елемент структури даних дерева називають *вузлом*. Верхній вузол дерева називається *кореневим вузлом*. Кожен вузол у дереві, крім кореня, має батьківський вузол і нуль або більше дочірніх вузлів (нащадків). Для останнього рівня вузлів немає нащадків. Вони називаються *листовими вузлами*. Там, де потрібно зберігати ієрархічні записи, найбільше підходять для використання деревоподібні структури даних.

Бінарне дерево – це тип дерева, у якому кожен вузол має не більше двох дочірніх елементів (0, 1 або 2), які називаються *лівим дочірнім* і *правим дочірнім вузлами.*

Бінарне дерево пошуку – Binary Search Tree (BST)

Двійкове дерево пошуку (BST) – це двійкове дерево, у якому вузли впорядковані наступним чином:

1. Ключ у лівому піддереві менший або дорівнює ключу в його батьківському вузлі.
2. Ключ у правому піддереві більший за ключ у його батьківському вузлі.

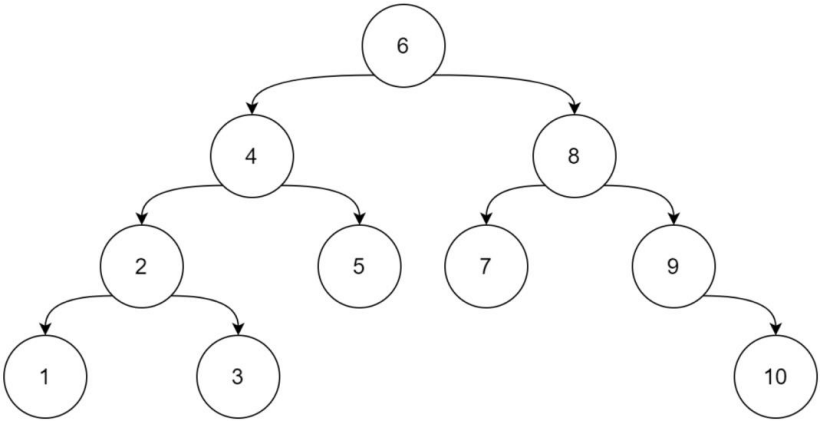


Рис. 1.4. Бінарне дерево пошуку

Операції з BST:

1. Insert(k) – вставити елемент k у дерево.
2. Delete(k) – видалити елемент k з дерева.
3. Search(k) – шукати значення k у дереві, якщо воно присутнє або ні.
4. FindMax() – знайти максимальне значення, яке зберігається в дереві.
5. FindMin() – знайти мінімальне значення, яке зберігається в дереві.

Середня *часова складність* усіх наведених вище операцій у двійковому дереві пошуку становить O(log(n)), але тільки у випадку, коли дерево збалансоване. У найгіршому випадку *часова складність* становить O(n), коли дерево незбалансоване.

Реалізація TreeSet у Kotlin Collections

```
import java.util.*

fun main() {
// Створюємо tree set.
    val treeSet = TreeSet<String>()
// Додаємо елементи в tree set.
    treeSet.add("Banana")
    treeSet.add("Apple")
    treeSet.add("Mango")
    println(treeSet)
    println("Apple present : " + treeSet.contains("Apple"))
    println("Orange present : " + treeSet.contains("Orange"))
    treeSet.remove("Apple")
    println("Apple present : " + treeSet.contains("Apple"))
}
```

Результат:

```
[Apple, Banana, Mango]
Apple present : true
Orange present : false
Apple present : false
```

Примітка. TreeSet реалізований за допомогою бінарного дерева пошуку, тому методи add, remove і contains мають

логарифмічну *часову складність* O(log(n)), де n – кількість елементів у наборі.

Реалізація TreeMap у Kotlin Collections

Map<> – це інтерфейс, який зіставляє ключі зі значеннями. TreeMap<> є реалізацією Map<> і реалізований за допомогою червоно-чорного збалансованого бінарного дерева, тому пари ключ-значення зберігаються у відсортованому порядку.

```
import java.util.*

fun main() {
// Створюємо tree map.
    val treeMap = TreeMap<String, Int>()
// Додаємо елементи в map
    treeMap["Apple"] = 40
    treeMap["Banana"] = 10
    treeMap["Mango"] = 20
    println("Size :: " + treeMap.size)
    for (key in treeMap.keys) {
        println(key + " cost : " + treeMap[key])
    }
    println("Apple present : " + treeMap.containsKey("Apple"))
    println("Orange present : " + treeMap.containsKey("Orange"))
    treeMap.remove("Apple")
    println("Apple present : " + treeMap.containsKey("Apple"))
}
```

Результат:

```
Size :: 3
Apple cost : 40
Banana cost : 10
Mango cost : 20
Apple present : true
Orange present : false
Apple present : false
```

Бінарна купа / Пріоритетна черга – Heap / Priority Queue

Бінарна купа – це структура даних, яка використовується для реалізації пріоритетної черги. У бінарній купі дані логічно організовані як повне бінарне дерево. Повне бінарне дерево – це бінарне дерево, яке заповнюється на всіх можливих рівнях,

крім останнього рівня. Значення кожного вузла в купі більше (або менше), ніж значення у його дочірніх вузлах.

Існує два типи структури даних купи:

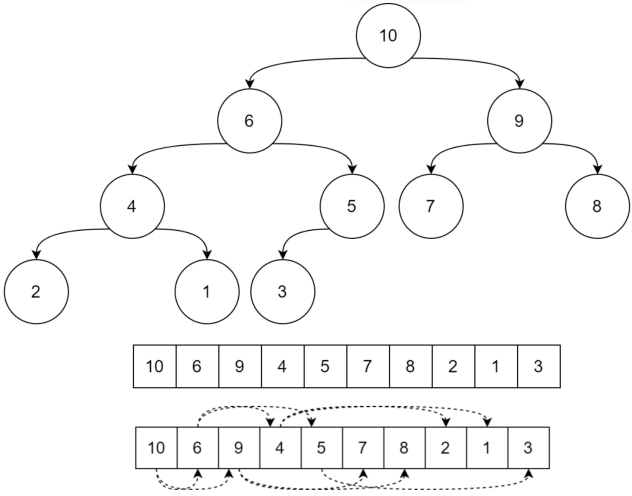


Рис. 1.5. Max-Heap

1. Max-Heap (Максимальна купа): значення кожного вузла має бути більше або дорівнювати значенням його дочірніх вузлів.

2. Min-Heap (Мінімальна купа): значення кожного вузла має бути менше або дорівнювати значенням його дочірніх вузлів.

Операції з бінарною купою

Розглянемо операції з купою:

1. Insert(k) – додати новий елемент k до купи. *Часова складність* цієї операції становить O(log(n)).

2. Remove() – видалити та повернути максимальний елемент для Max-Heap (або мінімальний елемент для випадку Min-Heap). *Часова складність* цієї операції дорівнює O(log(n)).

3. Heapify() – перетворити масив чисел у купу. Ця операція має *часову складність* O(n).

Реалізація пріоритетної черги в Kotlin Collections

Реалізація Priority Queue як Min-Heap:

```
import java.util.*

fun main() {
    val pq = PriorityQueue<Int>()
    val arr = intArrayOf(8, 1, 2, 10, 7, 3, 4, 6, 5, 9)
    for (i in arr) {
        pq.add(i)
    }
    println("Heap : $pq")
    while (!pq.isEmpty()) {
        print(" ${pq.remove()}")
    }
}
```

Результат:

```
Heap : [1, 5, 2, 6, 8, 3, 4, 10, 7, 9]
 1 2 3 4 5 6 7 8 9 10
```

Реалізація Priority Queue як Max-Heap:

```
import java.util.*

fun main() {
    val pq = PriorityQueue<Int>(Collections.reverseOrder())
    val arr = intArrayOf(8, 1, 2, 10, 7, 3, 4, 6, 5, 9)
    for (i in arr) {
        pq.add(i)
    }
    println("Heap : $pq")
    while (!pq.isEmpty()) {
        print(" ${pq.remove()}")
    }
}
```

Результат:

```
Heap : [10, 9, 4, 6, 8, 2, 3, 1, 5, 7]
 10 9 8 7 6 5 4 3 2 1
```

Хеш-таблиця – Hash Table

Хеш-таблиця – це структура даних, яка зіставляє ключі зі значеннями. Хеш-таблиця обчислює індекс даних у масиві за допомогою хеш-функції. Ми використовуємо хеш-таблицю,

коли кількість пар ключ-значення, яка зберігається, невелика відносно кількості можливих ключів.

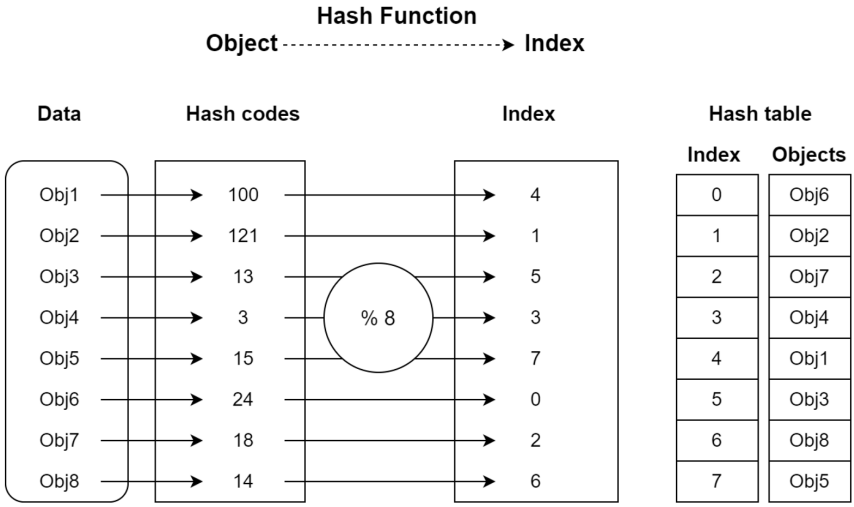


Рис. 1.6. Hash Table

Процес зберігання даних за допомогою хеш-функції виглядає наступним чином:

1. Щоб зберігати дані, створюється масив розміром M, який називається *хеш-таблицею*.

2. Знайти хеш-код ключа за допомогою хеш-функції.

3. Обчислюється остача від ділення хеш-коду на розмір хеш-таблиці, таким чином отримується індекс комірки, до якої потраплять дані.

4. Нарешті, дані записуються в комірку з відповідним індексом.

Процес пошуку даних у хеш-таблиці за допомогою хеш-функції виглядає наступним чином:

1. Використовуючи хеш-функцію, обчислюється хеш-код для ключа, який шукається.

2. Обчислюється остача від ділення хеш-коду на розмір хеш-таблиці, щоб отримати індекс таблиці, де знаходяться дані.

3. Нарешті, повертається значення за обчисленим індексом (якщо воно є).

Операції з хеш-теблицею

Розглянемо операції з хеш-таблицями:

1. Insert(x) – додати x до хеш-таблиці.
2. Delete(x) – видалити x із хеш-таблиці.
3. Search(x) – шукати x у хеш-таблиці.

Хеш-таблиця – це корисна структура даних для створення словника. Середній час пошуку для елемента в хеш-таблиці дорівнює O(1).

Реалізація HashSet в Kotlin Collections

HashSet <> – це клас, який реалізує інтерфейс Set<>, що означає, що він зберігає унікальні елементи. HashSet <> реалізовано за допомогою хеш-таблиці. Оскільки HashSet<> реалізовано за допомогою хеш-таблиці, її елементи не зберігаються в послідовному порядку.

```
fun main() {
// Створюємо hash set.
    val hs = HashSet<String>()
// Додаємо елементи у hash set.
    hs.add("Banana")
    hs.add("Apple")
    hs.add("Mango")
    println(hs)
    println("Apple present: " + hs.contains("Apple"))
    println("Orange present: " + hs.contains("Orange"))
    hs.remove("Apple")
    println("Apple present: " + hs.contains("Apple"))
}
```

Результат:

```
[Apple, Mango, Banana]
Apple present: true
Orange present: false
Apple present: false
```

Реалізація LinkedHashSet у Kotlin Collections

LinkedHashSet<> – це клас, який реалізує інтерфейс Set<>, що означає, що він зберігає унікальні елементи. LinkedHashSet<> реалізовано за допомогою хеш-таблиці та зв'язаного списку. LinkedList використовується для збереження порядку елементів на основі вставки. Обхід елементів хеш-таблиці відбувається в порядку вставки.

```
fun main() {
// Створюємо hash set.
    val hs = HashSet<String>()
// Додаємо елементи в hash set.
    hs.add("Banana")
    hs.add("Apple")
    hs.add("Mango")
    println("HashSet : $hs")
// Створюємо linked hash set.
    val lhs = LinkedHashSet<String>()
// Додаємо елементи в linked hash set.
    lhs.add("Banana")
    lhs.add("Apple")
    lhs.add("Mango")
    println("LinkedHashSet : $lhs")
}
```

Результат:

```
HashSet : [Apple, Mango, Banana]
LinkedHashSet : [Banana, Apple, Mango]
```

Таблиця 1.1. **Порівняння різних класів Set**

	TreeSet	HashSet	LinkedHashSet
Зберігання	Червоно-чорне дерево	Хеш-таблиця	Хеш-таблиця та зв'язаний список
Продуктивність	Повільніше за HashSet, O(log(N))	Найшвидше, константний час	Більш витратний, ніж HashSet, мусить підтримувати зв'язаний список
Послідовність обходу	У порядку зростання	Немає гарантованого порядку	У порядку додавання

Реалізація HashMap у Kotlin Collections

Map<> – це структура даних, яка відображає ключі та значення. HashMap<> є реалізацією Map<> і реалізовано за допомогою хеш-таблиці, тому пари ключ-значення не зберігаються у відсортованому порядку. Map<> не дозволяє дублювати ключі, але значення можуть дублюватися.

```
fun main() {
// Створюємо hash map.
    val map = HashMap<String, Int>()
// Додаємо елементи в map
    map["Apple"] = 40
    map["Banana"] = 10
    map["Mango"] = 20
    println("Size : " + map.size)
    for (key in map.keys) {
        println(key + " cost : " + map[key])
    }
    println("Apple present : " + map.containsKey("Apple"))
    println("Orange present : " + map.containsKey("Orange"))
}
```

Результат:

```
Size : 3
Apple cost : 40
Mango cost : 20
Banana cost : 10
Apple present : true
Orange present : false
```

Таблиця 1.2. **Порівняння різних класів Map**

	TreeMap	HashMap	LinkedHashMap
Зберігання	Червоно-чорне дерево	Хеш-таблиця	Хеш-таблиця та зв'язаний список
Продуктивність	Повільніше за HashMap, O(log(N))	Найшвидше, константний час	Більш витратний, ніж HashMap, мусить підтримувати зв'язаний список
Послідовність обходу	У порядку зростання	Немає гарантованого порядку	У порядку додавання

Висновок

У цьому розділі подано короткий вступ до різних структур даних та їх складності. У наступних розділах детально розглядатимуться всі ці структури даних. Якщо ви знаєте інтерфейс структури даних, ви можете використовувати її для розв'язання різних задач, не знаючи її внутрішньої реалізації.

Тема 2. **Сортування**

Вступ

Сортування – це процес розташування елементів у порядку зростання або спадання. Наприклад, під час гри в карти, карти можна сортувати за значенням, щоб легко знайти потрібну карту.

Інший приклад: у бібліотеці, книги розташовуються за темами (алгоритми, операційні системи, мережі тощо). Сортування впорядковує елементи даних у логічному порядку для спрощення та прискорення здійснення пошуку. Набагато легше знайти потрібну книгу, якщо книги розташовані належним чином у порядку індексації.

Алгоритми сортування, такі як BubbleSort, InsertionSort і SelectionSort, прості в реалізації та добре працюють з невеликими наборами вхідних даних. Однак вони неефективні для великих наборів даних. Алгоритми сортування, такі як MergeSort, QuickSort та HeapSort підходять для сортування великих наборів даних. Однак вони є надмірно складними, якщо ми хочемо відсортувати невеликий набір даних. Деякі інші алгоритми існують для сортування величезних наборів даних, що не можуть зберігатися в пам'яті повністю, для чого і розроблена техніка зовнішнього сортування.

Типи сортування

Внутрішнє сортування: усі елементи можна зчитати в пам'ять одночасно і виконати сортування в пам'яті:

1. SelectionSort.
2. InsertionSort.

3. BubbleSort.
4. QuickSort.
5. MergeSort.

Зовнішнє сортування: у цьому типі сортування набір даних настільки великий, що неможливо завантажити весь набір даних у пам'ять, тому сортування виконується порціями.

Функція порівняння

Перш ніж почати обговорення різних алгоритмів по одному, по-перше, ми повинні розглянути функцію порівняння, яка використовується для порівняння двох значень.

Функція less повертає true, якщо value1 менше за value2, інакше повертає значення false.

Функція greater повертає true, якщо value1 більше за value2, інакше повертає значення false.

```
fun less(value1: Int, value2: Int): Boolean {
    return value1 < value2
}

fun greater(value1: Int, value2: Int): Boolean {
    return value1 > value2
}
```

Значення в різних алгоритмах сортування порівнюються за допомогою однієї з наведених вище функцій, і вони будуть мінятися місцями залежно від значення, що повертається цими функціями. Якщо використовується функція порівняння greater(), тоді масив буде відсортований у порядку зростання. З іншого боку, якщо використовується функція порівняння less(), тоді масив буде відсортовано в порядку спадання.

BubbleSort

BubbleSort – це найповільніший алгоритм сортування, який використовується, коли набір даних невеликий.

Його легко реалізувати та використовувати. У BubbleSort ми порівнюємо кожну пару суміжних значень.

Ми хочемо відсортувати значення в порядку зростання, тож якщо друге значення менше за перше, ми міняємо ці два значення місцями.

Після цього ми переходимо до наступної пари. Таким чином, найбільше значення опиняється в кінці масиву.

Після першого проходу найбільше значення буде у крайньому правому місці. Після *N* проходів отримаємо повне сортування масиву.

5	1	3	4	2	7	6	Обмін
1	5	3	4	2	7	6	Обмін
1	3	5	4	2	7	6	Обмін
1	3	4	5	2	7	6	Обмін
1	3	4	2	5	7	6	Без обміну
1	3	4	2	5	7	6	Обмін
1	3	4	2	5	6	7	

Рис. 2.1. Сортування BubbleSort

```
fun bubbleSort(arr: IntArray) {
    val size = arr.size
    for (i in 0 until size - 1) {
        for (j in 0 until size - i - 1) {
            if (greater(arr[j], arr[j + 1])) {
                /* Swapping */
                arr[j + 1] = arr[j].also { arr[j] = arr[j + 1] }
            }
        }
    }
}
```

Для тестування цього методу використаємо такий код:

```
fun main() {
    val array = intArrayOf(8, 2, 9, 5, 7, 6, 3, 4, 1)
    bubbleSort(array)
    for (i in array.indices) {
        print("${array[i]} ")
    }
}
```

Результат:

```
1 2 3 4 5 6 7 8 9
```

Аналіз:

1. Зовнішній цикл представляє кількість обмінів, які виконуються для порівняння даних.

2. Внутрішній цикл використовується для порівняння даних. У першій ітерації найбільше значення буде переміщено в кінець масиву, у другій ітерації буде переміщено друге за величиною значення, воно опиниться перед найбільшим значенням і т. д.

3. Функція greater() використовується для порівняння, що означає, коли значення першого аргумента більше за значення другого аргумента, тоді виконується обмін. Таким чином, значення у масиві сортуються в порядку зростання. Якщо використати функцію less() замість greater (), тоді масив буде відсортовано в порядку спадання.

Аналіз складності

Внутрішній цикл виконується (n-1), (n-2), (n-3),…, 3, 2, 1 разів, тому в найгіршому випадку час буде (n-1)+(n-2)+(n-3)+…+3+2+1 = $O(n^2)$.

Покращений BubbleSort

Коли за один прохід зовнішнього циклу не було зроблено жодного обміну, масив уже відсортовано. На цьому етапі слід припинити сортування. Це вдосконалення сортування у BubbleSort особливо корисно, коли відомо, що, за винятком кількох елементів, решта масиву вже відсортована.

```
fun bubbleSort2(arr: IntArray) {
    val size = arr.size
    var swapped = true
    for (i in 0 until size - 1) {
        if (! swapped) break
        swapped = false
        for (j in 0 until size - i - 1) {
            if (greater(arr[j], arr[j + 1])) {
                arr[j] = arr[j + 1].also { arr[j + 1] = arr[j] }
                swapped = true
            }
        }
    }
}
```

Застосовуючи це покращення, найкраща продуктивність цього алгоритму покращується, коли масив майже впорядковано. У цьому випадку нам потрібен лише один прохід, а найкраща складність – O(n).

Аналіз складності

Таблиця 2.1. **Характеристики BubbleSort**

Продуктивність у найгіршому випадку	$O(n^2)$
Середня продуктивність	$O(n^2)$
Додаткова пам’ять	O(1)
Продуктивність покращеного алгоритму, коли масив практично відсортований	O(n)
Стабільне сортування	Так

InsertionSort

Сортування вставками працює подібно до того, як ми впорядковуємо колоду карт. Ми зберігаємо відсортований підмасив. Кожне значення розміщується у відсортованому підмасиві у належному місці, щоб підмасив залишався відсортованим.

Складність часу InsertionSort дорівнює $O(n^2)$, що є таким же, як і BubbleSort, але в середньому цей спосіб працює швидше.

5	1	3	4	2	7	6	Вставка 5
1	5	3	4	2	7	6	Вставка 1
1	3	5	4	2	7	6	Вставка 3
1	3	4	5	2	7	6	Вставка 4
1	2	3	4	5	7	6	Вставка 2
1	2	3	4	5	7	6	Вставка 7
1	2	3	4	5	6	7	Вставка 6

Рис. 2.2. Сортування InsertionSort

```
fun insertionSort(arr: IntArray) {
    val size = arr.size
    for (i in 1 until size) {
        val temp = arr[i]
        var j = i
        while (j > 0 && greater(arr[j - 1], temp)) {
            arr[j] = arr[j - 1]
            j--
        }
        arr[j] = temp
    }
}
```

Для тестування цього методу використаємо такий код:

```
fun main() {
    val array = intArrayOf(8, 2, 9, 5, 7, 6, 3, 4, 1)
    insertionSort(array)
    for (i in array.indices) {
        print("${array[i]} ")
    }
}
```

Результат:

```
1 2 3 4 5 6 7 8 9
```

Аналіз:

1. Зовнішній цикл використовується для вибору значення, яке буде вставлене у відсортовану частину масиву ліворуч.

2. Вибране значення, яке ми хочемо вставити, зберігається у змінній temp.

3. Внутрішній цикл виконує порівняння за допомогою функції greater(). Значення зсуваються праворуч, доки ми не знайдемо правильну позицію для значення, збереженого у змінній temp.

4. Нарешті, значення змінної temp установлюється в належне місце. У кожній ітерації зовнішнього циклу довжина відсортованого масиву збільшується на одиницю. Коли ми виходимо із зовнішнього циклу, масив відсортовано.

Таблиця 2.2. **Характеристики InsertionSort**

Продуктивність у найгіршому випадку	$O(n^2)$
Найкраща продуктивність	O(n)
Середня продуктивність	$O(n^2)$
Додаткова пам'ять	O(1)
Стабільне сортування	Так

SelectionSort

Алгоритм сортування вибором обходить несортований масив і розміщує найбільше значення в кінці. Цей процес повторюється n-1 разів. Цей алгоритм також має квадратичну *часову складність*, але він працює краще, ніж BubbleSort та InsertionSort, оскільки в цьому підході потрібно менше обмінів.

```
fun selectionSort(arr: IntArray) {
// сортований масив створюється у зворотному порядку.
    val size = arr.size
    for (i in 0 until size - 1) {
        var max = 0
        for (j in 1 until (size - i)) {
            if (arr[j] > arr[max]) {
                max = j
            }
```

```
        }
        arr[size - 1 - i] = arr[max]. also { arr[max] = arr[size - 1 - i] }
    }
}
```

5	1	3	4	2	7	6	Обмін
5	1	3	4	2	6	7	Без обміну
5	1	3	4	2	6	7	Обмін
2	1	3	4	5	6	7	Без обміну
2	1	3	4	5	6	7	Без обміну
2	1	3	4	5	6	7	Обмін
1	2	3	4	5	6	7	

Рис. 2.3. SelectionSort

Для тестування цього методу використаємо такий код:

```
fun main() {
    val array = intArrayOf(8, 2, 9, 5, 7, 6, 3, 4, 1)
    selectionSort(array)
    for (i in array.indices) {
        print("${array[i]} ")
    }
}
```

Результат:

```
1 2 3 4 5 6 7 8 9
```

Аналіз:

1. Зовнішній цикл визначає кількість ітерацій внутрішнього циклу. Для масиву з *N* елементів, зовнішній цикл повториться *N* разів.

2. Найбільше значення розміщується в кінці масиву після кожної ітерації внутрішнього циклу.

3. Після всіх ітерацій масив сортується. Відсортований масив створюється у зворотному порядку.

Таблиця 2.3. **Характеристики SelectionSort**

Продуктивність у найгіршому випадку	$O(n^2)$
Найкраща продуктивність	$O(n^2)$
Середня продуктивність	$O(n^2)$
Додаткова пам'ять	O(1)
Стабільне сортування	Ні

Ми можемо реалізувати той самий алгоритм, який створюватиме відсортований масив з початку масиву. Давайте подивимося на другий алгоритм, за допомогою якого ми можемо реалізувати сортування вибором.

```
fun selectionSort2(arr: IntArray) {
// сортований масив створюється у прямому порядку
    val size = arr.size
    for (i in 0 until size - 1) {
        var min = i
        for (j in i+1 until size) {
            if (arr[j] < arr[min]) {
                min = j
            }
        }
        arr[i] = arr[min].also { arr[min] = arr[i] }
    }
}
```

MergeSort

MergeSort використовує техніку «Розділяй і володарюй» для сортування вхідного масиву. Сортування злиттям рекурсивно ділить вхідний масив на дві половини. Потім обидві половини сортуються окремо, а результат об'єднується в кінцевий відсортований масив – результат. Рекурсивний крок продовжує ділити вхідні дані на менші частини до тих пір, поки довжина кожної частини стане рівною одному.

Розглянемо схему сортування злиттям.

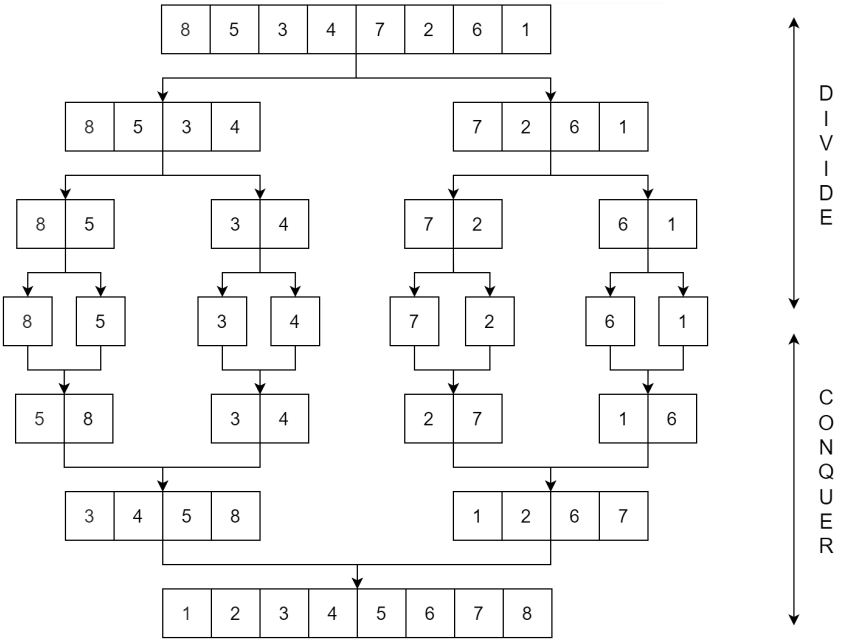


Рис. 2.4. MergeSort

```
fun mergeSort(arr: IntArray) {
    val size = arr.size
    val tempArray = IntArray(size)
    mergeSort(arr, tempArray, 0, size - 1)
}
fun   mergeSort(arr:   IntArray,   tempArray:   IntArray,   lowerIndex:
Int, upperIndex: Int) {
    if (lowerIndex >= upperIndex) {
        return
    }
    val middleIndex = (lowerIndex + upperIndex) / 2
    mergeSort(arr, tempArray, lowerIndex, middleIndex)
    mergeSort(arr, tempArray, middleIndex + 1, upperIndex)
    merge(arr, tempArray, lowerIndex, middleIndex, upperIndex)
}
```

```
fun merge(arr: IntArray, tempArray: IntArray, lowerIndex: Int,
middleIndex: Int, upperIndex: Int) {
    var lowerStart = lowerIndex
    var upperStart = middleIndex + 1
    var count = lowerIndex
    while (lowerStart <= middleIndex && upperStart <= upperIndex) {
        if (arr[lowerStart] < arr[upperStart]) {
            tempArray[count++] = arr[lowerStart++]
        } else {
            tempArray[count++] = arr[upperStart++]
        }
    }
    while (lowerStart <= middleIndex) {
        tempArray[count++] = arr[lowerStart++]
    }
    while (upperStart <= upperIndex) {
        tempArray[count++] = arr[upperStart++]
    }
    for (i in lowerIndex..upperIndex) {
        arr[i] = tempArray[i]
    }
}
```

Для тестування цього методу використаємо такий код:

```
fun main() {
    val array = intArrayOf(8, 2, 9, 5, 7, 6, 3, 4, 1)
    mergeSort(array)
    for (i in array.indices) {
        print("${array[i]} ")
    }
}
```

Результат:

```
1 2 3 4 5 6 7 8 9
```

Аналіз:

1. *Часова складність* сортування злиттям становить O(n.log(n)) у всіх трьох випадках (найкращому, середньому та найгіршому), оскільки він завжди ділить масив на дві половини та об'єднує їх у лінійному часі.

2. Він потребує стільки ж додаткового місця, скільки й несортований масив. Значить, зовсім не рекомендовано для опрацювання великих несортованих масивів.

Таблиця 2.4. **Характеристики MergeSort**

Продуктивність у найгіршому випадку	O(n·log n)
Найкраща продуктивність	O(n·log n)
Середня продуктивність	O(n·log n)
Додаткова пам'ять	O(n)
Стабільне сортування	Так

Таблиця 2.5. **Плюси та мінуси алгоритму MergeSort**

Плюси сортування злиттям	Мінуси сортування злиттям
1. Можна використовувати для сортування масивів великого розміру. 2. Можна використовувати для сортування зв'язаних списків. 3. Використовується у зовнішньому сортуванні. 4. MergeSort – стабільне сортування	1. Використовує додатковий простір. 2. Не є алгоритмом сортування на місці. 3. Навіть якщо список уже відсортований, він завжди буде виконуватися за час O(n·log(n)). 4. Рекурсивні алгоритми повільніші за ітераційні

QuickSort

Давайте подивимося на роботу алгоритму швидкого сортування.

1. Швидке сортування – це рекурсивний алгоритм. На кожному кроці вибирається опорний елемент (скажімо, перший елемент масиву).

2. Масив поділяється на дві частини, причому всі елементи, менші за опорний, переміщуються ліворуч, а всі елементи, більші за опорний, переміщуються у праву частину масиву. Щоб зробити це, треба виконати наступні кроки:

• Вибирається опорний елемент (наприклад, перший елемент у масиві).

• Використовуються дві індексні змінні: нижня та верхня.

• Нижній індекс установлюється на другий елемент у масиві, а верхній індекс – на останній елемент масиву.

• Нижній індекс збільшується, доки значення елемента по нижньому індексу менше за опорне.

• Далі зменшується верхній індекс, доки значення елемента по верхньому індексу більше за опорне значення.

• Потім виконується обмін місцями значень за нижнім та верхнім індексами.

• Повторюються наведені вище 3 кроки, поки верхній індекс більше за нижній.

• Зрештою обмінюються значення за опорним та верхнім індексами.

3. Потім сортуються лівий та правий підмасиви окремо за допомогою рекурсії.

4. Коли алгоритм буде завершено, весь масив буде відсортовано.

Розглянемо схему сортування QuickSort.

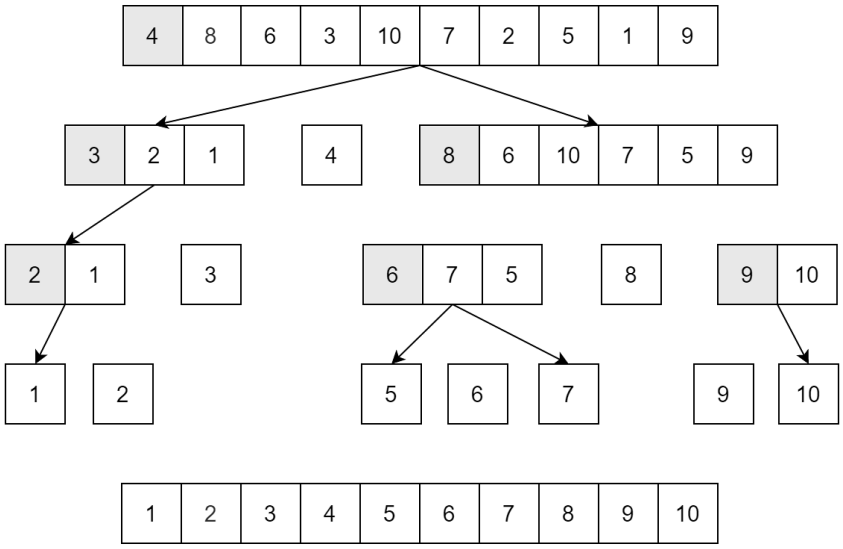


Рис. 2.5. QuickSort

```
fun quickSort(arr: IntArray) {
    sort(arr, 0, arr.size - 1)
}
fun sort(arr: IntArray, start: Int, stop: Int) {
    if (stop <= start) return
    val pivot = arr[start]
    var lower = start
    var upper = stop
    while (lower < upper) {
        while (arr[lower] <= pivot && lower < upper)
            lower++
        while (arr[upper] > pivot && lower <= upper)
            upper--
        if (lower < upper)
            swap(arr, upper, lower)
    }
    swap(arr, upper, start)
    sort(arr, start, upper - 1)
    sort(arr, upper + 1, stop)
}
fun swap(arr: IntArray, first: Int, second: Int) {
    arr[first] = arr[second].also { arr[second] = arr[first] }
}
```

Для тестування цього методу використаємо такий код:

```
fun main() {
    val array = intArrayOf(8, 2, 9, 5, 7, 6, 3, 4, 1)
    quickSort(array)
    for (i in array.indices) {
        print("${array[i]} ")
    }
}
```

Результат:

```
1 2 3 4 5 6 7 8 9
```

Аналіз:

1. QuickSort потребує набагато менше місця, потрібно лише O(n·log(n)) додаткового місця.

2. QuickSort не є стабільною технікою сортування. Він може змінювати порядок елементів з ідентичними ключами.

Аналіз найкращого випадку: найкращий випадок виникає, коли процес розділення вибирає середній елемент як центр, який ділить масив на дві рівні частини.

Таблиця 2.6. **Характеристики QuickSort**

Продуктивність у найгіршому випадку	$O(n^2)$
Найкраща продуктивність	$O(n \cdot \log n)$
Середня продуктивність	$O(n \cdot \log n)$
Додаткова пам'ять	$O(n \cdot \log n)$
Стабільне сортування	Ні

Рекурентне співвідношення для найкращого випадку: $T(n) = 2T(n/2) + O(n)$.

Розв'язуючи наведене вище рекурентне співвідношення, отримаємо: $O(n \cdot \log(n))$.

Аналіз найгіршого випадку: найгірший випадок виникає, коли опорний елемент не може розділити масив і кожна ітерація просто зменшить розмір масиву для сортування на 1. Це відбувається, коли всі елементи в масиві мають значення, більші або менші за опорне значення.

Це може статися в наступних випадках:

1. Масив уже відсортовано в порядку зростання.
2. Масив уже відсортований у порядку спадання.

Рекурентне співвідношення для найкращого випадку: $T(n) = T(n-1) + O(n)$.

Розв'язуючи наведене вище рекурентне співвідношення, отримаємо: $O(n^2)$.

CountingSort

Сортування підрахунком є найпростішим і найефективнішим видом сортування. Сортування підрахунком має сувору вимогу попередньо визначеного діапазону даних. Наприклад, ми хочемо відсортувати кількість людей у кожній віковій групі. Ми знаємо, що вік людей може варіюватися від 1 до 130. Ми можемо безпосередньо зберігати підрахунки в масиві розміром 130. Якщо ми знаємо діапазон введення, тоді сортування можна виконати за допомогою підрахунку сортування в $O(n+k)$, де n – кількість елементів, а k – можливий діапазон у наведеному вище прикладі – 130.

Розглянемо схему сортування CountingSort:

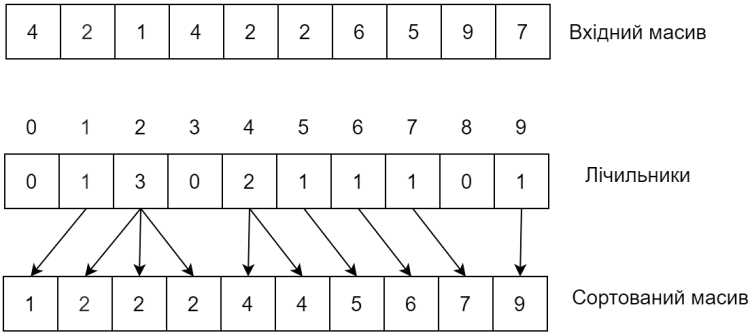


Рис. 2.6. CountingSort

```
fun countSort(arr: IntArray, lowerRange: Int, upperRange: Int) {
    val size = arr.size
    val range = upperRange - lowerRange
    val count = IntArray(range)
    var i = 0
    while (i < size) {
        count[arr[i] - lowerRange]++
        i++
    }
    i = 0
    var j = 0
    while (i < range) {
        while (count[i] > 0) {
            arr[j++] = i + lowerRange
            count[i]--
        }
        i++
    }
}
```

Для тестування цього методу використаємо такий код:

```
fun main() {
    val array = intArrayOf(15, 16, 19, 12, 20, 19, 22, 22, 12, 12)
    countSort(array, 10, 25)
    for (i in array.indices) {
        print("${array[i]} ")
    }
}
```

Результат:

```
12 12 12 15 16 19 19 20 22 22
```

Аналіз:

1. Для зберігання кількостей ми створили масив лічильників.
2. Елементам масиву лічильників установлено значення 0.
3. Індекс, пов'язаний із вхідним масивом, пробігає всі можливі значення, виконуючи підрахунки.
4. Нарешті, інформація, що збережена в масиві лічильників, відображується в результуючому масиві.

Таблиця 2.7. **Характеристики QuickSort**

Структура даних	Масив
Найкраща продуктивність	O(n+k)
Середня продуктивність	O(n+k)
Додаткова пам'ять	O(k)

Примітка. k – кількість різних елементів, n – загальна кількість елементів масиву.

HeapSort

У HeapSort дані зберігаються у структурі даних Heap, яка завжди видає дані в сортованому порядку.

Докладно HeapSort буде розглянуто в розділі, що присвячений реалізації Priority Queue за допомогою структури Heap.

Таблиця 2.8. **Характеристики QuickSort**

Структура даних	Масив
Найгірша продуктивність	O(n·log n)
Середня продуктивність	O(n·log n)
Додаткова пам'ять	O(1)

TreeSort

У TreeSort дані зберігаються у структурі даних «Бінарне дерево пошуку». Обходячи в інфіксному порядку Двійкове

дерево пошуку, отримаємо дані в порядку зростання. Докладно бінарне дерево пошуку буде розглянуто в розділі «Дерева».

Таблиця 2.9. **Характеристики TreeSort**

Найкраща продуктивність	O(n·log n)
Найгірша продуктивність	$O(n^2)$
Середня продуктивність	O(n·log n)
Додаткова пам'ять	O(n)
Стабільне сортування	Так

Висновок

Який же алгоритм сортування – найкращий?

Жоден алгоритм сортування не є ідеальним. Кожен з них має свої плюси і мінуси. Давайте обговоримо їх по черзі:

QuickSort: цей алгоритм використовується, коли стабільне сортування не потрібне, а середня продуктивність важливіша, ніж продуктивність у найгіршому випадку. QuickSort варто вибирати, коли дані є випадковими.

Середня *часова складність* QuickSort становить O(n·log(n)), а найгірша *часова складність* – $O(n^2)$. Складність QuickSort щодо пам'яті становить O(log(n)) допоміжного сховища, яке є простором стека, який використовується в рекурсії.

MergeSort: цей алгоритм використовується, коли потрібне стабільне сортування та *часова складність* O(n·log(n)). MergeSort повільніше, ніж QuickSort, оскільки на етапі злиття бере участь багато копій. Існує два варіанти використання сортування злиттям:

- Об'єднати два відсортовані пов'язані списки.
- Сортування злиттям використовується у зовнішньому сортуванні.

HeapSort: коли не потрібне стабільне сортування, і більш важлива продуктивність у найгіршому випадку, ніж середня

продуктивність. Вона гарантовано дорівнює O(n·log(n)), а використання допоміжного простору O(1) означає, що у вас не виникне непередбачуваного вичерпання пам'яті на дуже великих наборах вхідних даних.

InsertionSort: коли потрібне стабільне сортування та розмір вхідних даних гарантовано малий, наприклад, базові випадки QuickSort або MergeSort. Найгірша часова складність – $O(n^2)$. Він має дуже маленький постійний коефіцієнт множення для обчислення фактично витраченого часу. Тому для невеликих розмірів вхідних даних він працює краще, ніж MergeSort або QuickSort. Це також корисно, коли дані вже відсортовані. У цьому випадку його час роботи дорівнює O(n).

BubbleSort: відомо, що дані майже відсортовані. Скажімо, лише два елементи не на місці. Тоді за один прохід BubbleSort відсортує дані, а за другий прохід побачить усе, що відсортовано, а потім вийти. Потрібно лише два проходи масиву.

SelectionSort: найкращий, найгірший і середній час виконання – $O(n^2)$. Це корисно лише тоді, коли ви хочете швидко щось зробити. Його можна використовувати, коли ви просто створюєте прототип.

CountingSort: коли ви сортуєте дані в межах обмеженого діапазону.

Тема 3. **Зв’язані списки**

Припустимо, що є масив, який містить наступні сім елементів 1, 2, 4, 5, 6, 7, 8. Якщо потрібно вставити новий елемент зі значенням «3» між «2» і «4», то у масиві це не так легко зробити.

Для цього потрібно створити ще один масив, який буде достатньо великим, щоб зберігати поточні значення та ще одне місце для «3». Потім потрібно скопіювати ці елементи в новий масив. Ця операція копіювання неефективна. Для того, щоб усунути цю неефективність, зокрема, може використовуватись зв’язаний список.

Зв’язаний список – Linked List

Зв’язаний список – це список елементів, які називаються *вузлами*. Вузли мають дві частини: частину значення та частину зв’язку. Частина значення використовується для зберігання даних. Частина значення вузла може бути базовим типом даних як ціле число, або це може бути інший тип даних, наприклад, структура. Частина посилання – це покажчик, який зберігає адресу наступного елемента списку.

Рис. 3.1. Вузол Linked List

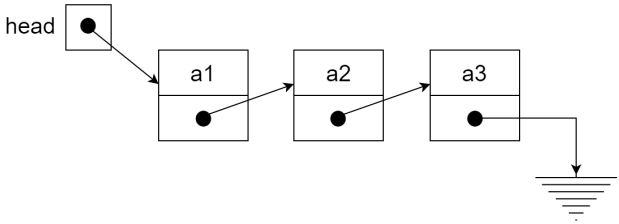


Рис. 3.2. Зв'язаний список – Linked List

Важливі частини зв'язаного списку:

1. **Head:** голова – це вказівник, який містить адресу першого вузла у зв'язаному списку.

2. **Nodes:** елементи у зв'язаному списку називаються *вузлами*.

3. **Value:** дані, які зберігаються в кожному вузлі зв'язаного списку.

4. **Link:** посилальна частина вузла використовується для зберігання посилання на інший вузол. Зазвичай використовуються імена «next» і «prev», щоб зберегти адресу наступного або попереднього вузла.

Типи зв'язаних списків

Існують різні типи зв'язаних списків. Основна відмінність між ними полягає в тому, як з'єднуються їхні вузли. Далі будуть розглянуті однозв'язний список, двозв'язний список, круговий зв'язаний і двозв'язний круговий список.

Однозв'язний список

Однозв'язний список складається з вузлів. Кожен вузол складається з двох частин: перша частина – це дані, а друга частина – це посилання, яке містить посилання на наступний вузол у зв'язаному списку. Посилальна частина останнього вузла містить значення null. Початкове посилання списку називається «head» і вказує на перший вузол зв'язаного списку.

Примітка. У однозв'язному списку можливо рухатися лише в одному напрямку, оскільки кожен вузол має лише посилання на наступний вузол (рис. 3.2).

Давайте подивимося на вузол. Частина значення (value) вузла має тип Int, але це може бути й інший тип даних. Посилальна частина вузла на наведеній нижче схемі називається link.

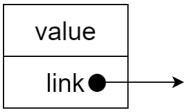


Рис. 3.3. List node

```
class LinkedList {

    class Node(var value: Int, var next: Node? = null )

    private var head: Node? = null
    private var size = 0

    val isEmpty: Boolean
        get() = size == 0

    fun size(): Int {
        return size
    }

    fun peek(): Int {
        if (isEmpty) throw IllegalStateException("EmptyListException")
        return head!!.value
    }
// Other Methods.
}
```

Основні операції, які можна виконувати над зв'язаними списками:

1. Вставити елемент у список. Ця операція використовується для створення зв'язаного списку.

2. Роздрукувати всі елементи списку.
3. Пошук елемента у списку.
4. Видалення елемента зі списку.
5. Інвертування (перевертання) зв'язаного списку.

Для однозв'язного списку завжди треба тестувати такі три випадки:

1. Нуль елементів / порожній зв'язаний список.
2. Випадок з одним елементом / одним вузлом.
3. Загальний випадок.

Випадки з одним вузлом і відсутністю вузлів використовуються для тестування граничних випадків. Завжди треба перевіряти ці випадки перед надсиланням коду до готового застосування.

Додавання елемента у зв'язаний список

Елемент можна вставляти у зв'язаний список у різні місця. Розглянемо деякі приклади найбільш поширених випадків:

1. Вставка елемента на початок зв'язаного списку.
2. Вставка елемента в кінець зв'язаного списку.
3. Вставка елемента на *N*-у позицію у зв'язаному списку.
4. Вставка елемента у відсортований зв'язаний список із збереженням властивості упорядкування.

Вставка елемента на початок зв'язаного списку

Задача: вставити елемент на початку зв'язаного списку.

```
fun addHead(value: Int) {
    head = Node(value, head)
    size++
}
```

Аналіз:

1. Потрібно створити новий вузол зі значенням, переданим у функцію як аргумент.

2. Покажчик next нового вузла вказуватиме на голову пов'язаного списку або null, якщо список порожній.

3. Новостворений вузол стане головою зв'язаного списку.
Часова складність: O(1).

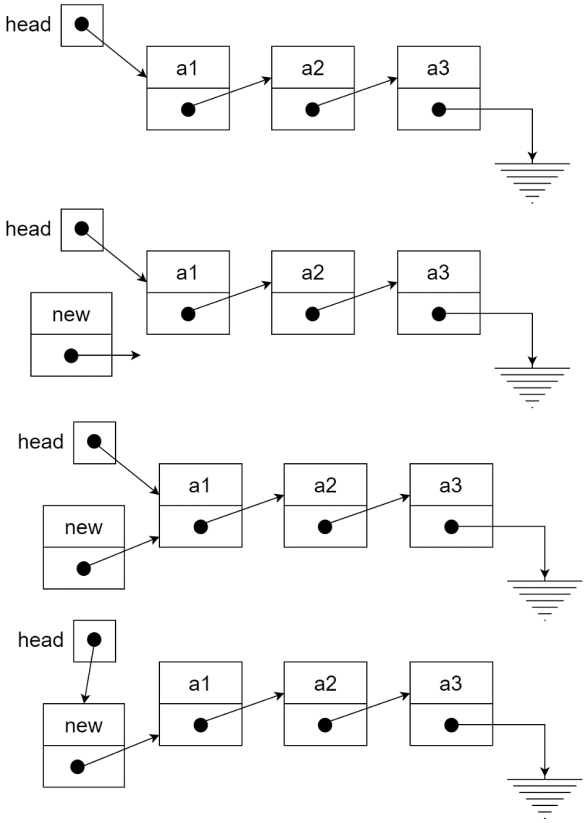


Рис. 3.4. Додавання елемента на початок списку

Вставка елемента у кінець зв'язаного списку

Задача: вставити елемент у кінець зв'язаного списку.

```
fun addTail(value: Int) {
    val newNode = Node(value, null)
    if (head == null) {
        head = newNode
    } else {
        var curr = head
```

```
        while (curr!!.next != null) {
            curr = curr.next
        }
        curr.next = newNode
    }
}
```

Аналіз:

1. Створюється новий вузол, значення зберігається в ньому та встановлюється нульове посилання в покажчик next.

2. Якщо список порожній, то цей новий вузол стане головою зв'язаного списку.

3. Якщо список не порожній, потрібно перейти до кінця списку.

4. Нарешті, новий вузол додається в кінець списку.

Часова складність: O(n).

Примітка. Ця операція неефективна, оскільки кожного разу, коли потрібно вставити елемент, необхідно переходити на кінець списку. Отже, складність створення списку становить $O(n^2)$. Таким чином, щоб зробити цю операцію ефективною, потрібно стежити за останнім елементом, зберігаючи покажчик на хвіст списку. Тому якщо потрібно вставити елемент у кінці зв'язаного списку, треба також відстежувати посилання на хвіст.

Обхід зв'язаного списку

Задача: надрукувати всі елементи зв'язаного списку.

```
fun print() {
    var temp = head
    while (temp != null) {
        print("${temp.value} ")
        temp = temp.next
    }
    println()
}
```

Аналіз:

Обходимо список і друкуємо значення, що зберігаються у вузлах. Список проходить шляхом створення тимчасового

посилання, яке спочатку вказує на голову, переходячи кожного разу на наступний вузол.

Часова складність: O(n).

Вставка елемента у відсортований зв'язаний список

Задача: вставити елемент у відсортованому порядку у зв'язаний список із заданим покажчиком head.

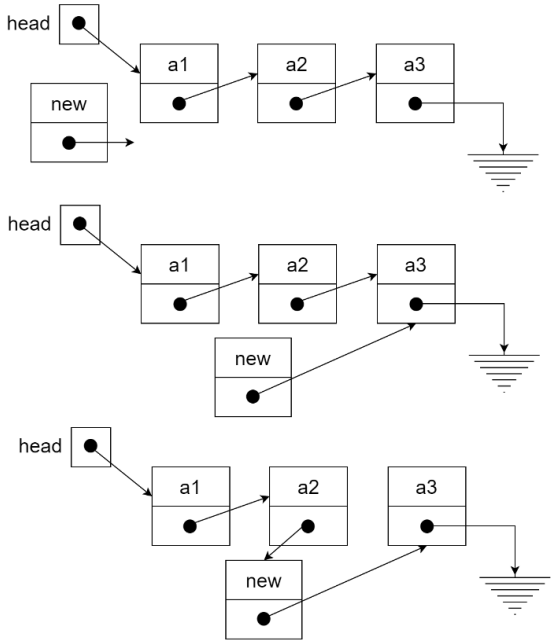


Рис. 3.5. Додавання елемента в сортований список

```
fun sortedInsert(value: Int) {
    val newNode = Node(value, null)
    var curr = head
    if (curr == null || curr.value > value) {
        newNode.next = head
        head = newNode
```

```
            return
        }
        while (curr!!.next != null && curr.next!!.value < value) {
            curr = curr.next
        }
        newNode.next = curr.next
        curr.next = newNode
    }
```

Для тестування виконаємо наступний код:

```
fun main() {
    val list = LinkedList()
    list.sortedInsert(1)
    list.sortedInsert(2)
    list.sortedInsert(3)
    list.sortedInsert(1)
    list.sortedInsert(2)
    list.sortedInsert(3)
    list.print()
}
```

Результат:

```
1 1 2 2 3 3
```

Аналіз:

1. Створюється новий порожній вузол зв'язаного списку. Ініціалізується шляхом збереження значення аргумента в його значення. Поле next цього вузла встановлюється в null.

2. Якщо список порожній, або якщо значення, збережене в першому вузлі, більше, ніж значення у новоствореному вузлі, цей щойно створений вузол, буде додано на початок списку. У такому випадку треба змінити посилання head.

3. У всіх інших випадках ми переглядаємо список, щоб знайти правильну позицію, де може бути вставлено новий вузол так, щоб зберегти список упорядкованим.

4. Нарешті, вузол буде додано до списку.

Часова складність: O(n).

Пошук елемента у зв'язаному списку

Задача: знайти елемент у зв'язаному списку. Дано покажчик head і шукане значення. Повернути true, якщо значення знайдено у списку, інакше повернути false.

```
fun search(data: Int): Boolean {
    var temp = head
    while (temp != null) {
        if (temp.value == data) return true
        temp = temp.next
    }
    return false
}
```

Аналіз:

1. Перебираємо список за допомогою циклу.
2. Значення кожного елемента списку порівнюється із заданим значенням. Якщо значення знайдено, тоді функція поверне true.
3. Якщо значення не знайдено, то в кінці функція повертає false.

Часова складність: O(n).

Примітка. Пошук в однозв'язному списку можна здійснювати лише в одному напрямку, оскільки всі елементи у списку мають посилання на наступний пункт у списку. Тому обхід зв'язаних списків є лінійним.

Видалення першого елемента у зв'язаному списку

Задача: видалити елемент у голові зв'язаного списку.

```
fun removeHead(): Int {
    if (isEmpty) throw IllegalStateException("EmptyListException")
    val value = head!!.value
    head = head!!.next
    size--
    return value
}
```

Аналіз:

Спочатку нам потрібно перевірити, чи список уже порожній. Якщо порожній, то повернути.

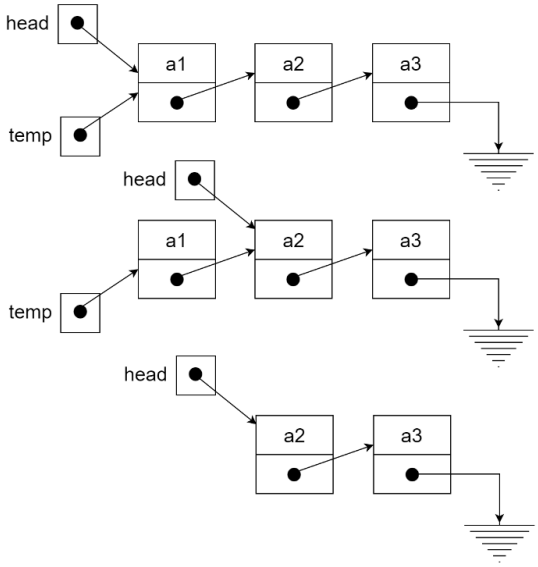


Рис. 3.6. Видалення елемента в голові зв'язаного списку

Якщо список не порожній, то голова списку зберігатиме посилання на наступний вузол.

Часова складність: O(1).

Видалити вузол із заданим значенням зі зв'язаного списку

Задача: видалити перший вузол, значення якого дорівнює заданому значенню.

```
fun deleteNode(delValue: Int): Boolean {
    if (isEmpty) throw IllegalStateException("EmptyListException")
    if (delValue == head!!.value) {
        head = head!!.next
        size--
        return true
    }
    var temp = head
    while (temp!!.next != null) {
        if (temp.next!!.value == delValue) {
```

```
            temp.next = temp.next!!.next
            size--
            return true
        }
        temp = temp.next
    }
    return false
}
```

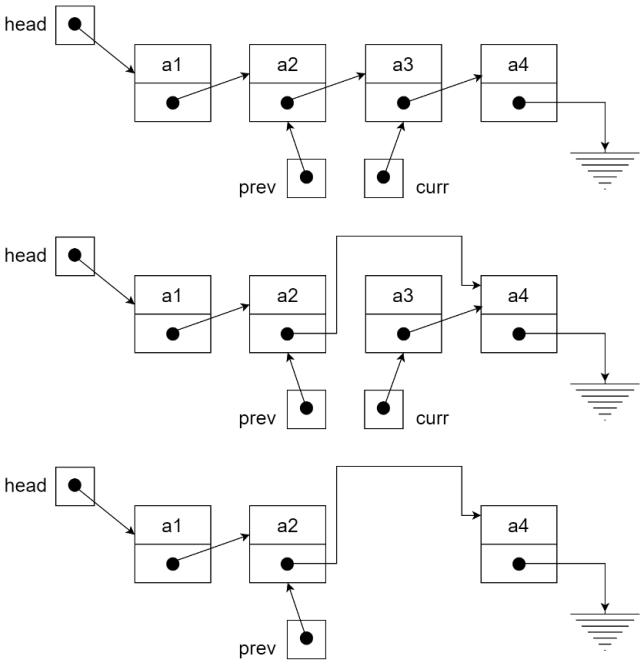


Рис. 3.7. Видалення вузла зі зв’язаного списку

Для тестування цього методу виконаємо наступний код:

```
fun main() {
    val list = LinkedList()
    list.addHead(7)
    list.addHead(6)
    list.addHead(5)
    list.addHead(2)
    list.addHead(5)
```

```
    list.addHead(6)
    list.print()
    println("deleteNode : " + list.deleteNode(5))
    list.print()
}
```

Результат:

```
6 5 2 5 6 7
deleteNode : true
6 2 5 6 7
```

Аналіз:

1. Є два випадки: перший випадок, коли вузол, який потрібно видалити, є першим вузлом і другий випадок, коли вузол, який потрібно видалити, не є першим вузлом у списку. Коли вузол, який потрібно видалити, це перший вузол, тоді заголовок зв'язаного списку буде змінено. У інших випадках – ні.

2. Спочатку ми перевіряємо, чи є перший вузол вузлом із значенням, яке ми шукаємо, а потім посилання head вказуватиме на наступне посилання для поточного посилання head.

3. В інших випадках переглядаємо список посилань за допомогою циклу while і спробуємо знайти вузол, який потрібно видалити. Якщо вузол знайдено, ми вказуємо у його попередньому вузлі посилання next на наступний вузол після вузла, який ми хочемо видалити.

Часова складність: O(n).

Видалити всі вузли із заданим значенням зі зв'язаного списку

Задача: видалити всі вузли, значення яких дорівнюють заданому значенню.

```
fun deleteNodes(delValue: Int): Boolean {
    var currNode = head
    var nextNode: Node?
    var found = false
    while (currNode != null && currNode.value == delValue) {
        head = currNode.next
```

```
            currNode = head
            found = true
        }
        while (currNode != null) {
            nextNode = currNode.next
            if (nextNode != null && nextNode.value == delValue) {
                currNode.next = nextNode.next
                found = true
            } else {
                currNode = nextNode
            }
        }
        return found
}
```

Для тестування цього методу виконаємо наступний код:

```
fun main() {
    val list = LinkedList()
    list.addHead(3)
    list.addHead(2)
    list.addHead(5)
    list.addHead(6)
    list.addHead(7)
    list.addHead(5)
    list.addHead(3)
    list.addHead(5)
    list.addHead(3)
    list.print()
    println("deleteNode : " + list.deleteNode(2))
    list.print()
    println("deleteNodes : " + list.deleteNodes(5))
    list.print()
}
```

Результат:

```
3 5 3 5 7 6 5 2 3
deleteNode : true
3 5 3 5 7 6 5 3
deleteNodes : true
3 3 7 6 3
```

Аналіз:

1. У першому циклі while ми видалимо всі вузли, які знаходяться на початку списку та мають значення, що

дорівнюють значенню, яке ми хочемо видалити. У цьому випадку нам потрібно оновити голову списку.

2. У другому циклі while ми видалимо всі вузли, які не знаходяться на початку списку і мають значення, що дорівнює значенню, яке ми хочемо видалити. Пам'ятайте, що ми не повертаємо нічого та проходимо до кінця списку.

Часова складність: O(n).

Видалення зв'язаного списку

Задача: видалити весь зв'язаний список.

```
fun deleteList() {
    head = null
    size = 0
}
```

Аналіз: просто потрібно позначити початок списку як нульовий. Вузли списку будуть звільнені «збиральником сміття». Також треба встановити розмір списку як 0.

Часова складність: O(n). (За рахунок роботи «збиральника сміття»).

Інвертування зв'язаного списку

Задача: інвертувати (змінити порядок елементів на протилежний) зв'язаний список за допомогою циклу та трьох покажчиків.

```
fun reverse() {
    var curr = head
    var prev: Node? = null
    var next: Node?
    while (curr != null) {
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    }
    head = prev
}
```

Для тестування використаємо такий код:

```
fun main() {
    val list = LinkedList()
    list.addHead(7)
    list.addHead(8)
    list.addHead(42)
    list.addHead(1)
    list.print()
    list.reverse()
    list.print()
}
```

Результат:

```
1 42 8 7
7 8 42 1
```

Аналіз: проходимо по списку. Робимо так, щоб змінна next у наступному вузлі вказувала на поточний вузол. Також установлюємо значення змінної next поточного таким, що вказує на попередній вузол. Змінюємо prev на поточний вузол (curr), а curr – на наступний (next).

Часова складність: O(n).

Рекурсивне інвертування зв'язаного списку

Задача: перевернути однозв'язний список за допомогою рекурсії.

```
fun reverseRecurse() {
    head = reverseRecurse(head, null)
}
fun reverseRecurse(currentNode: Node?, nextNode: Node?): Node? {
    if (currentNode == null) return null
    if (currentNode.next == null) {
        currentNode.next = nextNode
        return currentNode
    }
    val result = reverseRecurse(currentNode.next, currentNode)
    currentNode.next = nextNode
    return result
}
```

Аналіз:

1. Функція reverseRecurse() викличе функцію reverseRecurse() з параметрами, щоб повернути список

і покажчик, що повертає ця функція, буде головою перевернутого списку.

2. Поточний вузол указуватиме на наступний вузол, який є попереднім вузлом старого списку.

Часова складність: O(n).

Примітка. Зв'язаний список можна перевернути за допомогою двох рішень. Перший розв'язок – за допомогою трьох покажчиків. Другий розв'язок використовує рекурсію; обидва є лінійними розв'язками, але вважається, що розв'язок з трьома покажчиками є більш ефективним.

Інвертування зв'язаного списку в інший список

Задача: скопіювати вміст зв'язаного списку в інший зв'язаний список у зворотному порядку. Якщо оригінальний список містить елементи в порядку 1, 2, 3, 4, то новий список має містити елементи в порядку 4, 3, 2, 1.

```
fun copyListReversed(): LinkedList {
    var tempNode: Node? = null
    var tempNode2: Node?
    var curr = head
    while (curr != null) {
        tempNode2 = Node(curr.value, tempNode)
        curr = curr.next
        tempNode = tempNode2
    }
    val result = LinkedList()
    result.head = tempNode
    return result
}
```

Для тестування використаємо такий код:

```
fun main() {
    val list = LinkedList()
    list.addHead(7)
    list.addHead(42)
    list.addHead(3)
    list.print()
    val list2 = list.copyListReversed()
    list2.print()
}
```

Результат:

```
3 42 7
7 42 3
```

Аналіз: переглядаємо список і додаємо значення вузла до початку нового списку. Оскільки список проходить у прямому напрямку, і кожен вузол додається до голови нового списку, то новий список буде сформований у зворотному порядку початкового списку.

Часова складність: O(n).

Копіювання вмісту зв'язаного списку в інший зв'язаний список

Задача: скопіювати вміст заданого зв'язаного списку в інший зв'язаний список. Якщо оригінальний зв'язаний список містить елементи в порядку 1, 2, 3, 4, то новий список має містити елементи в тому ж порядку 1, 2, 3, 4.

```
fun copyList(): LinkedList? {
    val result = LinkedList()
    if (head == null)
        return result
    var curr: Node? = head
    var headNode: Node? = Node(curr!!.value, null)
    var tailNode: Node? = headNode
    curr = curr.next
    while (curr != null) {
        var tempNode = Node(curr.value, null)
        tailNode!!.next = tempNode
        tailNode = tempNode
        curr = curr.next
    }
    result.head = headNode
    return result
}
```

Для тестування використаємо такий код:

```
fun main() {
    val list = LinkedList()
    list.addHead(3)
```

```
    list.addHead(42)
    list.addHead(7)
    list.print()
    val list2 = list.copyList()
    list2!!.print()
}
```

Результат:

```
7 42 3
7 42 3
```

Аналіз: переглядаємо список і додаємо значення вузла до нового списку, але цього разу завжди в кінці списку. Інший покажчик tailNode використовується для відстеження кінця списку. Оскільки список обходиться у напрямку вперед і значення кожного вузла додається в кінець нового списку, то сформований список збігається з початковим списком.

Часова складність: O(n).

Порівняння списків

Задача: порівняти значення двох пов'язаних списків за їхніми головними покажчиками.

Рекурсивний розв'язок:

```
fun compareList(list2: LinkedList): Boolean {
    return compareList(head, list2.head)
}
fun compareList(head1: Node?, head2: Node?): Boolean {
    if (head1 == null && head2 == null)
        return true
    else if (head1 == null || head2 == null || head1.value != head2.value)
        return false
    else
        return compareList(head1.next, head2.next)
}
```

Аналіз:

1. Списки порівнюються рекурсивно. Крім того, якщо ми досягнемо кінця списку, і обидва списки будуть нульовими, тоді обидва списки рівні, тому повертається true.

2. Списки порівнюється рекурсивно. Якщо один зі списків порожній або значення відповідних вузлів є різними, то списки не є рівними, і функція compareList() поверне false.

3. Рекурсивно викликається функція порівняння списків для наступного вузла з поточних вузлів.

Ітераційний розв'язок:

```
fun compareList2(list2: LinkedList?): Boolean {
    var head1 = head
    var head2 = list2!!.head
    while (head1 != null && head2 != null) {
        if (head1.value != head2.value)
            return false
        head1 = head1.next
        head2 = head2.next
    }
    return head1 == null && head2 == null
}
```

Аналіз:

1. Проходимо по спискам у циклі та у будь-якій точці, якщо значення обох вузлів списку не дорівнюють одне одному, повертаємо false, тобто, що вони не є рівними списками.

2. Якщо, зрештою, обидва списки пройдено повністю, тоді повертаємо true, інакше повертаємо false, якщо будь-який із списків має один, чи більше непройдених елементів.

Часова складність: обидва розв'язки мають однакову O(n) часову складність.

Знайти елемент з номером *N* від початку

Задача: знайти *N*-й елемент від початку списку.

```
fun nthNodeFromBeginning(index: Int): Int {
    if (index > size() || index < 1)
        return Int.MAX_VALUE
    var count = 0
    var curr = head
    while (curr != null && count < index) {
        count++
        curr = curr.next
```

```
    }
    return curr!!.value
}
```

Для тестування використаємо такий код:

```
fun main() {
    val list = LinkedList()
    list.addHead(1)
    list.addHead(2)
    list.addHead(3)
    list.addHead(4)
    list.addHead(5)
    list.addHead(6)
    println(list.nthNodeFromBeginning(2))
}
```

Результат:

```
4
```

Висновок

Зв'язані списки дуже широко використовуються сучасними розробниками. Деякі їхні застосування наведені нижче:

1. Для реалізації різних структур даних, таких як стек, черга, хеш-таблиця, графи тощо.

2. Функції «Наступний» і «Попередній» у програмах перегляду фотографій. У програмах для перегляду фотографій використовується двозв'язний список. Ми можемо перейти від поточного зображення до попереднього або наступного.

3. Наступний і попередній треки в музичному плеєрі. У музичному плеєрі використовується циклічний двозв'язний список. Ми можемо відтворити попередню або наступну пісню. І коли буде відтворено весь список пісень, він починається знову з початку.

Тема 4. **Стек**

Вступ

Стек – це базова структура даних, яка організовує дані за принципом: «Останній прийшов – першим вийшов» (LIFO). Останній елемент, вставлений у стек, буде видалений з нього першим.

Реальною аналогією стека є «стос тарілок». Уявіть собі стопку тарілок в обідній зоні. Кожен бере тарілку з верхньої частини стопки. У результаті стає доступною наступна тарілка. Стек дозволяє отримати доступ лише до верхнього елемента. Елемент, який знаходиться внизу стека – це той, який залишається там найдовше.

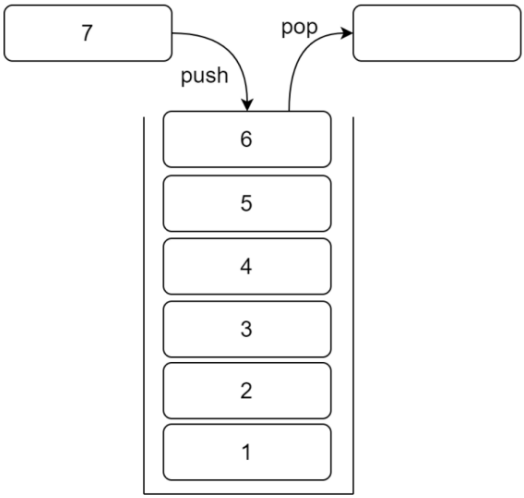


Рис. 4.1. Стек

Інформатика також має загальний приклад стека. Гарним прикладом стека є стек викликів функцій. Функція main() викликає функцію foo(), а потім foo() викликає bar(). Ці виклики функцій реалізовано за допомогою стека. Спочатку завершиться bar(), потім foo() і, нарешті, main().

Коли ми переходимо від вебсторінки до вебсторінки, URL-адреси вебсторінок зберігаються у стеку разом із URL поточної сторінки вгорі. Якщо ми натиснемо кнопку «Назад», кожен запис URL-адреси буде витягнуто один за одним.

Абстрактний тип даних – стек

Абстрактний тип даних – стек визначається як структура даних, яка слідує за LIFO або «Останній прийшов – перший вийшов» для елементів, доданих до нього.

Стек повинен підтримувати такі операції:

Push(): додає один елемент у верхню частину стека.

Pop(): видаляє один елемент із вершини стека.

Top(): читає значення верхнього елемента стека (не видаляє його).

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

Size(): повертає кількість елементів у стеку.

Push: додати значення на вершину стека (рис. 4.2).

Pop: видалити верхній елемент стека та повернути його до функції виклику (рис. 4.3).

Стек може бути реалізований за допомогою масиву або зв'язаного списку.

1. Коли стек реалізований за допомогою масиву, його ємність фіксована.

2. У випадку зв'язаного списку немає обмеження на кількість елементів, які він може містити.

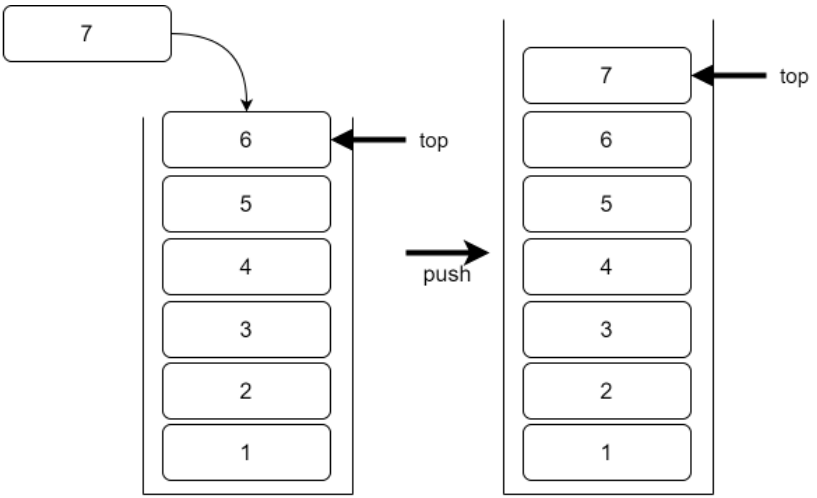


Рис. 4.2. Додавання елемента у стек

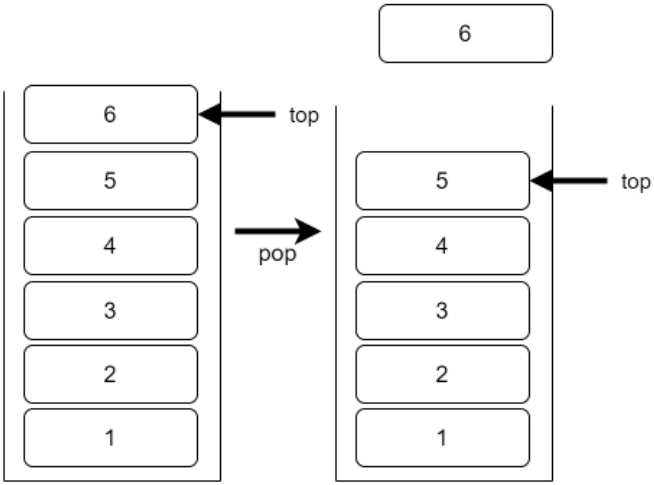


Рис. 4.3. Видалення елемента зі стека

Реалізація стека на основі масиву

Задача: реалізувати стек, використовуючи масив фіксованої довжини.

```
class Stack constructor(s: Int = 1000) {
    private var capacity: Int
    private var data: IntArray
    private var top = -1
    init {
        data = IntArray(s)
        capacity = s
    }
    // інші методи
}
```

isEmpty() повертає значення true, якщо стек порожній, або false у всіх інших випадках.

```
val isEmpty: Boolean
    get() = top == -1
```

Size() повертає кількість елементів у стеку.

```
fun size(): Int {
    return top + 1
}
```

Print() виводить на екран (друкує) елементи стека.

```
fun print() {
    for (i in top downTo 0) {
        print("${data[i]} ")
    }
    println()
}
```

Push() додає значення до даних.

```
fun push(value: Int) {
    if (top + 1 == capacity) {
        throw IllegalStateException("StackOverflowException")
    }
    top++
    data[top] = value
}
```

Pop() спочатку перевірить, чи стек не порожній. Якщо так, тоді витягне значення з масиву даних і поверне його.

```
fun pop(): Int {
    if (isEmpty) {
        throw IllegalStateException("StackEmptyException")
    }
    val topVal = data[top]
    top--
    return topVal
}
```

Top() повертає значення, збережене на вершині стека, але не видаляє його.

```
fun top(): Int {
    if (isEmpty) {
        throw IllegalStateException("StackEmptyException")
    }
    return data[top]
}
```

Протестуємо код для стека:

```
fun main() {
    val s = Stack()
    s.push(7)
    s.push(42)
    s.push(3)
    s.print()
    println(s.pop())
    println(s.pop())
}
```

Результат:

```
3 42 7
3
42
```

Аналіз: об’єкт стека оголошено та ініціалізовано.

Функції push() і pop() використовуються для додавання та видалення значень у стеку.

Часова складність: push(), pop(), top(), size() та isEmpty() – усі ці операції мають часову складність O(1). Функція print() має часову складність O(n).

Реалізація стека на основі масиву з управлінням пам'яттю

Задача: створити стек за допомогою масиву, але потрібно управляти пам'яттю масиву. Коли стек заповнений, його місткість (capacity) подвоюється. Крім того, коли кількість елементів падає нижче capacity/2, місткість стека зменшується вдвічі. Крім того, ми не хочемо, щоб місткість стека була нижче початково виділеного розміру. Ми можемо визначити мінімальну довжину під час створення стека.

Аналіз: структура стека містить чотири поля: «top», який є індексом найвищого елемента; покажчик «data», який вказує на динамічно виділену пам'ять; «size», який використовується для збереження розміру стек; «capacity», яка є місткістю стека.

Далі реалізовані функції push і pop. Усі інші функції, такі як isEmpty(), size() і top() залишаться без змін.

Реалізація push для стека на основі масиву, що може міняти розмір.

```
fun push2(value: Int) {
    if (top + 1 == capacity) {
        println("Size doubled.")
        val newData = IntArray(capacity * 2)
        java.lang.System.arraycopy(data, 0, newData, 0, capacity)
        data = newData
        capacity *= 2
    }
    top++
    data[top] = value
}
```

Аналіз:

1. Ми подвоюємо змінну ємності стека – capacity.
2. Розмір пам'яті перерозподіляється до більшого значення. Усі дані у стеку копіюються на нове місце.
3. Нарешті, дані переносяться у стек збільшеної ємності.

Реалізація pop для стека на основі масиву, що може міняти розмір

Реалізація функції pop() здійснюється таким чином, що коли кількість елементів у стеку падає нижче ємності/2, то місткість стека зменшується.

```
class Stack constructor(s: Int = 1000) {
    private var capacity: Int
    private var data: IntArray
    private var top = -1
    private val minCapacity : Int // for dynamic stack

    init {
        data = IntArray(s)
        capacity = s
        minCapacity = s // for dynamic stack
    }
    // інші методи
}
```

```
fun pop2(): Int {
    if (isEmpty) {
        throw IllegalStateException("StackEmptyException")
    }
    val topVal = data[top]
    top--
    if (top + 1 < capacity / 2 && capacity > minCapacity) {
        println("Size halved.")
        capacity /= 2
        val newData = IntArray(capacity)
        java.lang.System.arraycopy(data, 0, newData, 0, capacity)
        data = newData
    }
    return topVal
}
```

Аналіз: розмір стека перевіряється, якщо він опускається нижче половини ємності та перевищує мінімальний розмір, ємність стека ділиться на 2, а пам’ять перерозподіляється до половинного розміру.

Реалізація стека на основі зв'язаного списку

Задача: стек можна реалізувати за допомогою зв'язаного списку.

```
class StackOnLinkedList {
    private class Node(val value: Int, var next: Node?)
    private var head: Node? = null
    private var length = 0
    fun size(): Int {
        return length
    }
    val isEmpty: Boolean
        get() = length == 0
    fun peek(): Int {
        if (isEmpty) {
            throw IllegalStateException("StackEmptyException")
        }
        return head!!.value
    }
    fun push(value: Int) {
        head = Node(value, head)
        length++
    }
    fun pop(): Int {
        if (isEmpty) {
            throw IllegalStateException("StackEmptyException")
        }
        val value = head!!.value
        head = head!!.next
        length--
        return value
    }
    fun print() {
        var temp = head
        while (temp != null) {
            print(temp.value.toString() + " ")
            temp = temp.next
        }
        println()
    }
}
```

Для тестування цього класу використаємо такий код:

```
fun main() {
    val s = StackOnLinkedList()
    s.push(42)
```

```
    s.push(2)
    s.push(7)
    s.print()
    println(s.pop())
    println(s.pop())
}
```

Результат:

```
7 2 42
7
2
```

Аналіз:

1. Стек, реалізований за допомогою зв'язаного списку, є простим однозв'язним списком із вставкою та видаленням на початку списку.

2. У функції push() пам'ять виділяється для одного вузла. Потім значення зберігається в цьому вузлі. Нарешті, вузол вставляється на початку списку.

3. У функції pop() змінна head зв'язаного списку починає вказувати на другий вузол. Після цього повертається значення першого вузла.

Часова складність: push(), pop() та isEmpty() – усі ці операції мають часову складність O(1). Функція print() має часову складність O(n).

Реалізація стека в Kotlin Collections

Стек реалізовано шляхом виклику методів push і pop класу Stack<T>.

```
import java.util.*

fun main() {
    val stack = Stack<Int>()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    println("Stack : $stack")
    println("Stack size : " + stack.size)
    println("Stack pop : " + stack.pop())
    println("Stack top : " + stack.peek())
    println("Stack isEmpty : " + stack.isEmpty())
}
```

Результат:

```
Stack : [1, 2, 3]
Stack size : 3
Stack pop : 3
Stack top : 2
Stack isEmpty : false
```

Аналіз:

1. Значення 1, 2 і 3 додаються до стека за допомогою функції push(). Потім стек друкується.
2. Верх стека повертається за допомогою функції top().
3. Розмір стека повертається за допомогою функції size().
4. Функція isEmpty() повертає true, якщо стек порожній, інакше повертає false.
5. Функція pop() використовується для видалення верхнього елемента стека.

Задачі зі стеком

Інфіксний та постфіксний записи виразів

Інфіксний вираз: коли ми маємо алгебраїчний вираз на зразок A + B, ми знаємо, що змінна A додається до змінної B. Цей тип виразу називається *інфіксним виразом*, оскільки тут є оператор + між операндом A та операндом B.

Тепер розглянемо інший інфіксний вираз: A + B * C. У виразі є проблема пріоритету операторів. Яка операція буде виконана першою: + або *? Спочатку додаються A і B, а потім результат множиться на C, або B і C спочатку перемножуються, а потім результат додається до A. Це робить вираз неоднозначним. Щоб усунути цю неоднозначність, ми визначаємо правила пріоритету або використовуємо дужки для усунення неоднозначності.

Отже, якщо ми хочемо спочатку помножити B і C, а потім додати результат до A, тоді той самий вираз може бути записано однозначно з використанням дужок як A + (B * C). З іншого боку, якщо ми хочемо додати A і B спочатку, а потім

суму помножити на С, ми записуємо це як (А + В) * С. Тому нам потрібні дужки в інфіксному виразі, щоб зробити вираз однозначним.

Таблиця 4.1. **Порівняння інфіксного та постфіксного записів**

Інфіксний запис	Постфіксний запис
A + B	A B +
A + (B * C)	A B C * +
(A + B) * C	A B + C *

Навіщо нам такий неприродний постфіксний запис, коли у нас уже є інфіксний запис, – і це працює для нас?

Відповідь полягає в тому, що інфіксні вирази неоднозначні, і для їх уточнення потрібні дужки. Тоді як постфіксний запис не потребує дужок.

Перетворення інфіксного запису в постфіксний

Задача: написати функцію для перетворення інфіксного виразу в постфіксний вираз.

Розв'язок: «*Алгоритм сортувальної станції*»

1. Операнди виводяться в такому ж порядку, як вони надходять.

2. Якщо стек порожній або містить ліву круглу дужку «(» зверху, ми повинні додати вхідний оператор у стек.

3. Якщо вхідним символом є ліва дужка «(», вона додається у стек.

4. Якщо вхідним символом є права кругла дужка «)», то зі стека виштовхуються та друкуються всі оператори, доки не буде виштовхнута ліва дужка «(». Після того пару дужок треба відкинути.

5. Якщо пріоритет вхідного оператора вищий за пріоритет оператора на вершині стека, треба додати його у стек.

6. Якщо вхідний символ має рівний пріоритет, однаковий з вершиною стека, треба врахувати асоціативність. Якщо

асоціативність зліва направо, треба виштовхнути та надрукувати символ на вершині стека, а потім заштовхніть вхідний оператор. Якщо асоціація справа наліво, то вхідний оператор треба додати у стек.

7. Якщо пріоритет вхідного оператора нижчий за пріоритет оператора не вершині стека, треба виштовхнути і надрукувати верхній оператор. Потім порівняти вхідний оператор із новим оператором на вершині стека.

8. Якщо вираз закінчився, треба витягнути і надрукувати всі оператори, що залишились у стеку.

```
import java.util.*

fun main() {
    val expression = "10-((42))*17/(13+4)"
    val value = infixToPostfix(expression)
    println("Infix Expression: $expression")
    println("Postfix Expression: $value")
}

fun precedence(x: Char): Int {
    return when (x) {
        '(' -> 0
        '+', '-' -> 1
        '*', '/', '%' -> 2
        '^' -> 3
        else -> 4
    }
}
fun infixToPostfix(expression: String): String {
    var output = StringBuilder()
    val out = infixToPostfix(expression.toCharArray())
    for (ch in out) {
        output.append(ch)
    }
    return output.toString()
}

fun infixToPostfix(expression: CharArray): CharArray {
    val stk = Stack<Char>()
    var output = ""
    var out: Char
    for (ch in expression) {
```

```
        if (ch in '0'..'9') {
            output += ch
        } else {
            when (ch) {
                '+', '-', '*', '/', '%', '^' -> {
                    while (!stk.isEmpty() &&
                        precedence(ch) <= precedence(stk.peek())) {
                        out = stk.pop()
                        output = "$output $out"
                    }
                    stk.push(ch)
                    output = "$output "
                }
                '(' -> stk.push(ch)
                ')' -> while (!stk.isEmpty()) {
                    out = stk.pop()
                    if(out != '(')
                        output = "$output $out "
                    else
                        break
                }
            }
        }
    }
    while (!stk.isEmpty()) {
        out = stk.pop()
        output = "$output$out "
    }
    return output.toCharArray()
}
```

Результат:

```
Infix Expression: 10-((42))*17/(13+4)
Postfix Expression: 10 42 17 * 13 4 + / -
```

Обчислення виразу в постфіксному запису

Задача: написати функцію для обчислення постфіксного виразу.

Наприклад, вираз «1 2 + 3 4 + *» дає результат 21.

```
fun postfixEvaluate(expression: String?): Int {
    val stk = Stack<Int>()
    val tokens = Scanner(expression!!)
    while (tokens.hasNext()) {
        if (tokens.hasNextInt()) {
            stk.push(tokens.nextInt())
```

```
        } else {
            val num1: Int = stk.pop()
            val num2: Int = stk.pop()
            when (tokens.next()[0]) {
                '+' -> stk.push(num1 + num2)
                '-' -> stk.push(num1 - num2)
                '*' -> stk.push(num1 * num2)
                '/' -> stk.push(num1 / num2)
            }
        }
    }
    tokens.close()
    return stk.pop()
}
fun main() {
    val expression = "12 4 2 5 + 6 * + 7 + *"
    val value = postfixEvaluate(expression)
    println("Result after evaluation: $value")
}
```

Результат:

```
Result after evaluation: 636
```

Застосування стека для пошуку у глибину

Під час пошуку у глибину ми рухаємося доступними шляхами, доки не потрапляємо у глухий кут; тоді ми повертаємося назад, витягуючи попередню позицію зі стека, щоб отримати альтернативний шлях.

1. Створити стек.
2. Створити початкову точку.
3. Помістити початкову точку у стек.
4. Поки (пошук не закінчено і стек не порожній).
   4.1. Витягнути значення зі стека.
   4.2. Знайти всі можливі точки після тієї, яку щойно дістали.
   4.3. Помістити ці точки на вершину стека.

Докладніше цей алгоритм буде розглянутий у розділі, присвяченому графам.

Висновок

Структура даних «Стек» використовується у багатьох задачах, зокрема:

- Де потрібна функція LIFO.
- Функції скасування та повторення в текстовому редакторі.
- Функції «Назад» і «Вперед» у веббраузері.
- Перевірка граматики, наприклад, зіставлення дужок.
- Перетворення виразів (наприклад, інфікс у постфікс тощо).
- Постфіксне обчислення виразу.
- Виклики функцій реалізовано за допомогою системного стека.
- Пошук у глибину в деревах і графах.

Тема 5. **Черга**

Черга – структура даних, яка організовує елементи за принципом FIFO. Перший елемент, доданий до черги, буде видалений першим. Він також відомий як «Перший прийшов – перший обслужений».

Справжня аналогія черги – це типова черга, у якій ми всі час від часу беремо участь.

1. Ми стояли в черзі біля залізничної каси.
2. Чекаємо в черзі в кафе.
3. Ми чекаємо в черзі, коли телефонуємо у службу підтримки.

Елементи, які знаходяться на початку черги, це ті, які залишалися в черзі найдовше.

Деякі типові приклади черг в інформатиці:

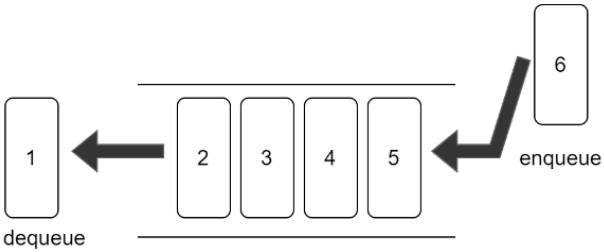


Рис. 5.1. Черга

• Команда друку: коли ми видаємо команду друку з нашого офісу на один принтер, завдання на друк шикуються в чергу принтера. Спочатку буде надруковано перше завдання друку перед наступними завданнями в черзі.

• Операційна система: операційна система використовує різні черги для керування плануванням процесів. Процеси додаються до черги обробки, для якої операційною системою використовуються різні алгоритми планування.

• Обхід графів: обхід графів у ширину використовує структуру даних черги.

Абстрактний тип даних «*Черга*»

Абстрактний тип даних «Черга» визначається як структура даних, екземпляр якої слідує правилу FIFO або «Першим прийшов – першим вийшов» для доданих до нього елементів.

Черга повинна підтримувати такі операції:

• add() додає один елемент у кінці черги;

• remove() видаляє один елемент із початку черги;

• isEmpty() повертає true, якщо черга порожня, або повертає false, якщо черга містить елементи;

• size() повертає кількість елементів у черзі.

Черга на основі масиву

Щоб реалізувати чергу за допомогою масиву, створюємо масив arr розміром N і беремо дві змінні: front та back, обидві ініціалізуються 0. Спочатку розмір черги дорівнює 0. Функція add() використовується для додавання елемента в кінці черги за допомогою змінної back. Для видалення елемента з початку черги використовується функція remove(), це можна виконати за допомогою змінної front. Змінна back збільшується, коли нове значення додається до черги, а змінна front збільшується, коли значення видаляється з черги. Оператор обчислення остачі від ділення використовується для отримання наступних значень змінних back та front. Таким чином, масив веде себе як круговий масив.

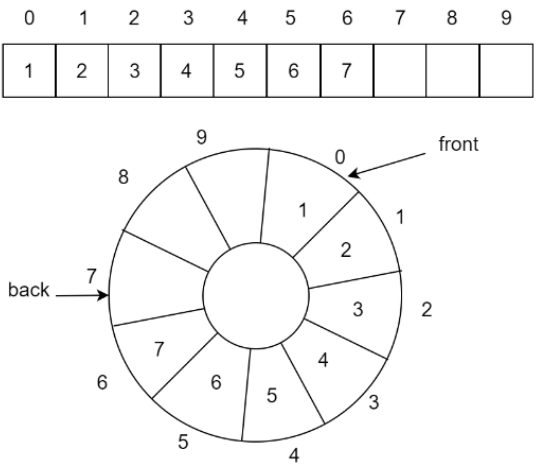


Рис. 5.2. Круговий масив як черга

```
class Queue constructor(private val capacity: Int = 100) {
    private val data = IntArray(capacity)
    private var size = 0
    private var front = 0
    private var back = 0

    val isEmpty: Boolean
        get() = size == 0

    fun size(): Int {
        return size
    }

    fun add(value: Int): Boolean {
        if (size >= capacity) {
            // Queue is full
            return false
        } else {
            size++
            data[back] = value
            back = ++back % capacity
        }
        return true
    }
    fun remove(): Int {
        val value: Int
```

```
        if (size <= 0) {
            // Queue is empty
            return Int.MIN_VALUE
        } else {
            size--
            value = data[front]
            front = ++front % capacity
        }
        return value
    }
    fun print() {
        if (size == 0) {
            println("Queue is empty.")
            return
        }
        var temp = front
        print("Queue is : ")
        for (s in 0 until size) {
            print("${data[temp]} ")
            temp = (temp + 1) % capacity
        }
        println()
    }
}
```

Для тестування створеної черги використаємо такий код:

```
fun main() {
    val queue = Queue()
    queue.add(1)
    queue.add(2)
    queue.add(3)
    println("isEmpty :  ${queue.isEmpty}")
    println("size :  ${queue.size()}")
    println("Queue remove : ${queue.remove()}")
    println("Queue remove : ${queue.remove()}")
}
```

Результат:

```
isEmpty : false
size : 3
Queue remove : 1
Queue remove : 2
```

Черга з використанням кругового зв’язаного списку

Чергу можна реалізувати за допомогою циклічного зв’язаного списку. Перевага використання циклічного зв’язаного

списку полягає в тому, що час, потрібний для отримання значення вузлів head та tail є константним. Коли нам потрібно вставити елемент до черги, треба додати елемент до хвоста (tail). Коли потрібно видалити елемент із черги, він видаляється з голови (head) списку.

```
class QueueLinkedList {
    private class Node(val value: Int, var next: Node?)
    private var tail: Node? = null
    private var size = 0
    val isEmpty: Boolean
        get() = size == 0
    fun size(): Int {
        return size
    }
    fun print() {
        if (size == 0) {
            print("Queue is empty.")
            return
        }
        var temp = tail!!.next
        print("Queue is : ")
        for (i in 0 until size) {
            print(temp!!.value.toString() + " ")
            temp = temp.next
        }
        println()
    }
    fun peek(): Int {
        if (isEmpty) throw IllegalStateException("StackEmptyException")
        val value: Int
        if (tail === tail!!.next)
            value = tail!!.value
        else
            value = tail!!.next!!.value
        return value
    }
}
```

Додавання

Вузли додаються в кінець зв'язаного списку. Схема нижче показує, як новий вузол додається до списку. Хвіст (tail) змінюється щоразу, коли з'являється нове значення, що додається до черги.

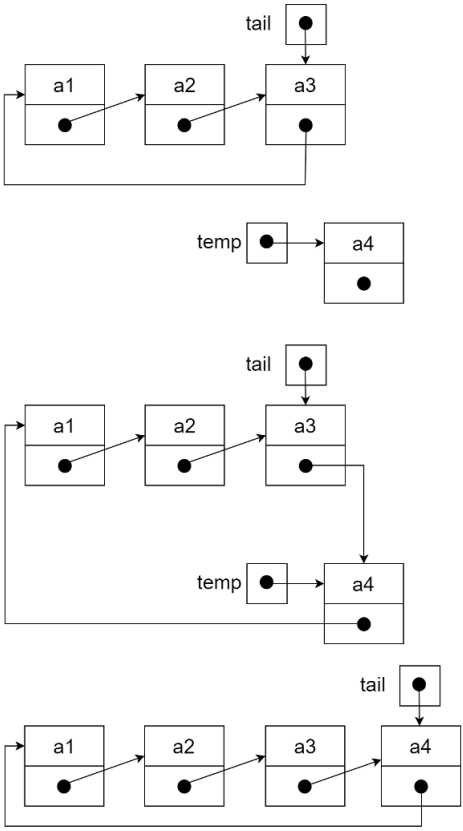


Рис. 5.3. Додавання елемента в чергу

```
fun add(value: Int) {
    val temp = Node(value, null)
    if (tail == null) {
        tail = temp
        tail!!.next = tail
    } else {
        temp.next = tail!!.next
        tail!!.next = temp
        tail = temp
    }
    size++
}
```

Аналіз: операція add() додає один елемент у кінець черги (круговий зв'язаний список).

Видалення

Операція видалення з черги виконується шляхом видалення елемента вузла head. Наступним вузлом для вузла tail є вузол head.

Посилання next вузла tail буде вказувати на вузол, наступний за вузлом head. Звільнюється пам'ять видаленого вузла. Після цього повертається його значення.

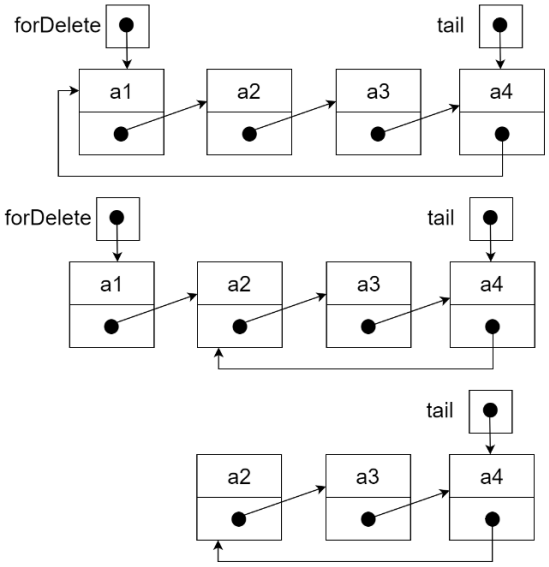


Рис. 5.4. Видалення елемента з черги

```
fun remove(): Int {
    if (size == 0) throw IllegalStateException("StackEmptyException")
    var value : Int
    if (tail === tail!!.next) {
        value = tail!!.value
        tail = null
    } else {
```

```
        value = tail!!.next!!.value
        tail!!.next = tail!!.next!!.next
    }
    size--
    return value
}
```

Результат:

```
isEmpty : false
size : 3
Queue remove : 17
Queue remove : 42
```

Аналіз: операція remove видаляє перший вузол на початку черги (круговий зв'язаний список).

Задачі з чергою

Черга на основі стека

Задача: реалізувати чергу за допомогою стека. Дозволяється використовувати декілька стеків (наприклад, 2).

Розв'язок: для реалізації черги можна використати два стеки.

1. Операція додавання в чергу: нові елементи додаються до верхньої частини першого стека.

2. Операція видалення з черги: елементи витягуються з другого стека. Коли другий стек порожній, тоді всі елементи першого стека виштовхуються один за одним і вставляються у другий стек.

```
class QueueUsingStack {
    private val stk1: Stack<Int> = Stack<Int>()
    private val stk2: Stack<Int> = Stack<Int>()

    fun add(value: Int) {
        stk1.push(value)
    }

    fun remove(): Int {
        var value: Int
        if (!stk2.isEmpty()) {
            return stk2.pop()
        }
```

```
        while (!stk1.isEmpty()) {
            value = stk1.pop()
            stk2.push(value)
        }
        return stk2.pop()
    }
}
```

Для тестування використаємо такий код:

```
fun main() {
    val queue = QueueUsingStack()
    queue.add(17)
    queue.add(42)
    queue.add(13)
    println("Queue remove : " + queue.remove())
    println("Queue remove : " + queue.remove())
}
```

Результат:

```
Queue remove : 17
Queue remove : 42
```

Аналіз: усі операції add() відбуваються зі стеком 1, коли викликається remove(), видалення відбувається зі стека 2.

Якщо стек 2 порожній, увесь зміст стека 1 виштовхується та поміщується у стек 2. Це виштовхування зі стека 1 і додавання у стек 2 змінює порядок вилучення, тим самим створюючи поведінку черги з двох стеків.

Часова складність у середньому O(1), елементи додаються в перший стек лише один раз. Вони переміщуються до другого стека лише один раз і виштовхують за допомогою операції pop().

Стек на основі черги

Задача: реалізувати стек за допомогою черги.

Перший розв'язок: використаємо дві черги.

Push: додати нові елементи до queue1.

Pop: поки розмір queue1 більший за 1. Перемістити всі елементи з черги 1, крім останнього, до черги 2. Змінюємо назви черги 1 і черги 2. Потім повертаємо останній елемент.

Операція push() має *часову складність* O(1), а операція pop() – O(n).

```
class Stack {
    private var queue1 = ArrayDeque<Int>()
    private var queue2 = ArrayDeque<Int>()
    private var size = 0

    fun push(value: Int) {
        queue1.add(value)
        size += 1
    }

    fun pop(): Int {
        var value = 0
        var s = size
        while (s > 0) {
            value = queue1.first()
            queue1.removeFirst()
            if (s > 1) queue2.add(value)
            s--
        }
        val temp = queue1
        queue1 = queue2
        queue2 = temp
        size -= 1
        return value
    }
}
```

Другий розв'язок: те саме можна зробити, використовуючи лише одну чергу.

Push: додати елемент до черги.

Pop: знайти розмір черги. Якщо розмір дорівнює нулю, повернути помилку. Інакше, якщо розмір додатний, то видалити size-1 елементів із черги та знову додати до тієї ж черги. Нарешті, видалити наступний елемент і повернути його.

```
fun push2(value: Int) {
    queue1.add(value)
    size += 1
}

fun pop2(): Int {
    var value = 0
```

```
    var s = size
    while (s > 0) {
        value = queue1.first()
        queue1.removeFirst()
        if (s > 1) queue1.add(value)
        s--
    }
    size -= 1
    return value
}
```

Для тестування можна використати такий код:

```
fun main() {
    val stack = Stack()
    stack.push(17)
    stack.push(42)
    stack.push(31)
    println("Stack pop : " + stack.pop())
    println("Stack pop : " + stack.pop())
}
```

Результат:

```
Stack pop : 31
Stack pop : 42
```

Пошук у ширину з використанням черги

Під час пошуку в ширину спочатку досліджуються всі найближчі вузли, знаходячи до того ж всі можливі наступники, які додаються до черги.

Розв'язок:

1. Створити чергу.
2. Створити початкову точку.
3. Додати початкову точку в чергу.
4. Поки (шукане значення не знайдено, і черга не порожня).

4.1. Дістати елемент з черги.

4.2. Знайти всі можливі елементи, до яких є безпосередні шляхи від елемента, який щойно видалений з черги.

4.3. Додати ці елементи в чергу.

Задача Йосипа Флавія

Задача: *n* людей стоять у черзі в очікуванні страти. Відлік починається спереду черги. На кожному кроці k людей видаляється з черги та знову додаються один за одним у чергу. Потім страчують першу наступну особу в черзі. Ці дії тривають по колу до того, поки залишиться остання людина, яка буде вільна. Треба знайти ту позицію, де буде стояти той, хто здобуде свободу.

Розв'язок:

1. У чергу послідовно додаються цілі числа від 1 до k (відповідає k особам).

2. Визначається функція kPop() так, щоб вона видаляла та додавала чергу k–1 елементів, а потім видаляла один. (Ця людина страчується).

3. Повторюється другий крок, поки розмір черги не стане 1.

4. Виводиться значення останнього елемента. Це і є розв'язок.

```
fun main() {
    println("Position : " + josephus(11, 5))
}

fun josephus(n: Int, k: Int): Int {
    val que = ArrayDeque<Int>()
    for (i in 0 until n) que.add(i + 1)
    while (que.size > 1) {
        for (i in 0 until k - 1) {
            que.add(que.peek())
            que.remove()
        }
        que.remove() // Kth person executed.
    }
    return que.peek()
}
```

Результат:

```
Position : 8
```

Шлях коня

Задача: дано шахівницю та стартову позицію коня. Потрібно знайти мінімальну кількість кроків, необхідних для переміщення коня з початкової в кінцеву позицію.

Розв'язок: BFS – створюємо чергу для збереження початкового розташування коня. Тоді ми виконуємо BFS, взявши один елемент із черги та обробивши його. Під час обробки елемента, який щойно вилучено з черги, усі його зв'язки, тобто не більше восьми різних клітин уставляються в кінець черги. До того ж ті клітини, які знаходяться ближче до початкового розташування коня, обробляються перед більш віддаленими клітинами. Це ефективний алгоритм із *часовою складністю* $O(n^2)$, де n – розмір шахівниці.

```
class Knight(val x: Int, val y: Int, val dist: Int)

fun stepsOfKnight(srcX: Int, srcY: Int, dstX: Int, dstY: Int, size: Int): Int {
    val traversed = Array(size) { IntArray(size) { Int.MAX_VALUE } }
    val dir = arrayOf(
        Pair(-2, -1), Pair(-2, 1), Pair(2, -1), Pair(2, 1),
        Pair(-1, -2), Pair(1, -2), Pair(-1, 2), Pair(1, 2))
    val queue = ArrayDeque<Knight>()

    queue.add(Knight(srcX - 1, srcY - 1, 0))
    traversed[srcX - 1][srcY - 1] = 0

    while (queue.isNotEmpty()) {
        val knight = queue.first()
        queue.removeFirst()
        for (i in 0..7) {
            val x = knight.x + dir[i].first
            val y = knight.y + dir[i].second
            val dist = knight.dist + 1
            if ((x in (0 until size)) && (y in (0 until size)) &&
                (traversed[x][y] > dist)) {
                queue.add(Knight(x, y, dist))
                traversed[x][y] = dist
            }
        }
    }
    return traversed[dstX - 1][dstY - 1]
}

fun main() {
    println("Steps from (1,1) to (3,3) = ${stepsOfKnight(1, 1, 3, 3, 8)}")
}
```

Результат:

```
Steps from (1,1) to (3,3) = 4
```

Висновок

Черги зручно використовувати в таких задачах:

1. Там, де потрібна функція FIFO.
2. Для реалізації черги принтера для планування різних команд друку.
3. Для планування використання спільних ресурсів в операційній системі.
4. Задачі мультипрограмування.
5. Черги повідомлень у службі обміну повідомленнями.
6. Пошук у ширину (BFS) на деревах і графах.

Тема 6. **Дерева**

Дерево – це нелінійна структура даних, яка використовується для представлення ієрархічних зв'язків між батьківськими і дочірніми вузлами. Кожен вузол у дереві з'єднаний з іншим вузлом направленим ребром.

Чому ми використовуємо дерева? Лінійні структури даних, такі як масив, зв'язаний список тощо, мають недолік – лінійний час пошуку елемента.

Рис. 6.1. Дерево – ієрархія співробітників компанії

Термінологія дерев

Нижче наведено базову термінологію дерева:

Вузол (node) – це фундаментальний елемент дерева. Кожен вузол має дані та два покажчики, які можуть указувати на null або на його дочірні вузли.

Ребро (edge) – це також фундаментальний елемент дерева, який використовується для з'єднання двох вузлів.

Корінь (root): Корінь дерева – це єдиний вузол без вхідних ребер. Це верхній вузол дерева.

Лист (leaf): Листовий вузол – це вузол, який не має дочірніх елементів.

Шлях (path): Шлях – це впорядкований список вузлів, з'єднаних ребрами.

Висота вузла (height of node) – це кількість ребер на найдовшому шляху між вузлом і листовим вузлом.

Висота дерева (height of tree) – це кількість ребер на найдовшому шляху між коренем і листовим вузлом.

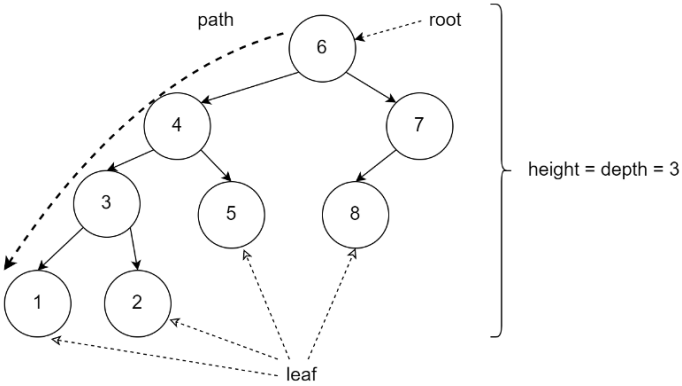


Рис. 6.2. Елементи дерева

Батьківський вузол (parent): вузол називається батьківським вузлом, якщо він має вихідні ребра до інших вузлів.

Дочірній вузол (child): вузол, який має вхідне ребро від іншого вузла, називається дочірнім для цього вузла.

Споріднені вузли (siblings): вузли, які є дочірніми вузлами одного батьківського вузла, називаються спорідненими вузлами.

Предок (ancestor): вузол називається вузлом-предком, якщо він доступний під час переходу від дочірнього до батьківського.

Рівень або глибина вузла (level or depth of a node): рівень вузла – це кількість ребер на шляху від кореня дерева до цього

вузла. Наприклад, рівень кореневого вузла root дорівнює 0, а рівні лівого і правого дочірніх елементів root дорівнюють 1.

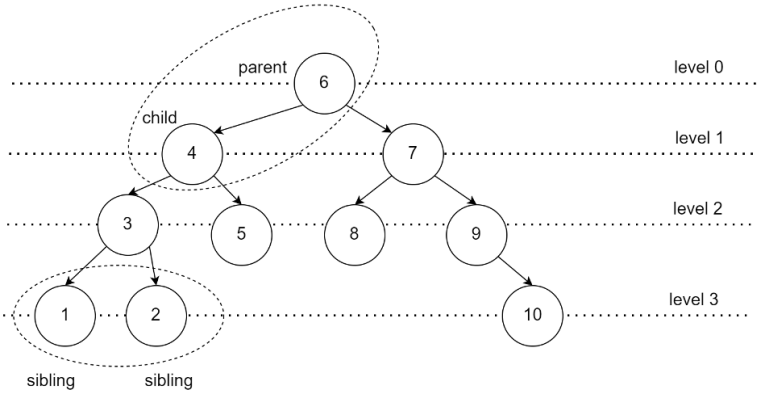


Рис. 6.3. Термінологія дерев

Бінарне дерево

Бінарне дерево – це тип дерева, у якому кожен вузол має не більше двох дочірніх елементів (0, 1 або 2), які називають лівим та правим дочірніми вузлами. Нижче наведено вузол бінарного дерева, у якому «a» – дані, що зберігаються, та в якому лівий дочірній (lchild) і правий дочірній вузли (rchild) обидва вказують на null.

```
class Tree (var root: Node? = null) {
    class Node (var value: Int, var left: Node? = null, var right: Node? = null)
    /* Інші методи */
}
```

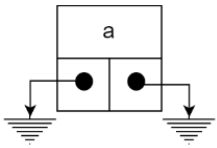


Рис. 6.4. Дерево з одного вузла

Розглянемо приклад бінарного дерева:

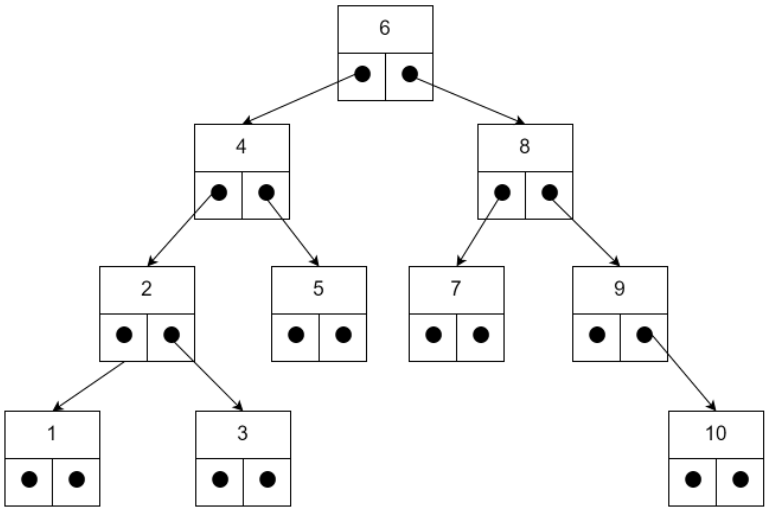


Рис. 6.5. Бінарне дерево, вузли якого містять числа від 1 до 10

Дещо спрощено, та для економії місця, бінарне дерево також може бути зображене, як на наступному рисунку.

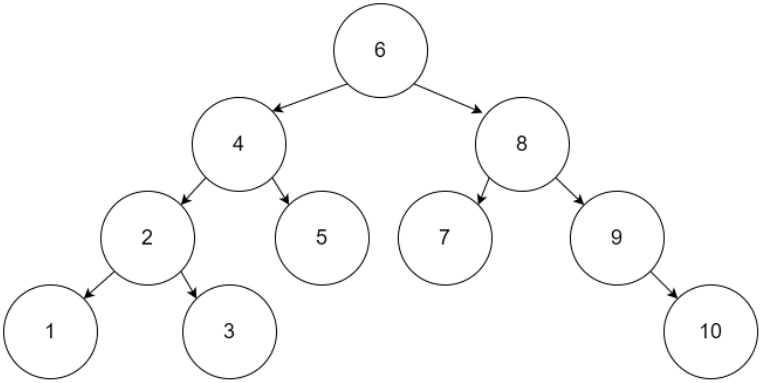


Рис. 6.6. Спрощене зображення бінарного дерева

Бінарне дерево має такі властивості:

1. Максимальна кількість вузлів на рівні L бінарного дерева становить $2L$, де $i \geq 0$.

2. Максимальна кількість вузлів у бінарному дереві висотою k становить $2^{(k+1)} - 1$, де $k \geq 0$.

3. Існує точно один шлях від кореня до будь-якого вузла у дереві.

4. Дерево з N вузлами має рівно N–1 ребер, що з'єднують ці вузли.

5. Висота повного бінарного дерева з N вузлів дорівнює $\log_2 N$.

Типи бінарних дерев

Повне бінарне дерево

У повному бінарному дереві кожен рівень, крім останнього, повністю заповнений. Усі вузли ліворуч заповнюються в першу чергу, потім заповнюються вузли праворуч. Двійкова купа є прикладом повного бінарного дерева.

Правильне (суворе), бінарне дерево

Правильне бінарне дерево – це бінарне дерево, у якому кожен вузол має рівно нуль або два дочірні елемента.

Ідеальне бінарне дерево

Ідеальне бінарне дерево – це тип повного бінарного дерева, у якому кожен нелистовий вузол має рівно два дочірні вузли. Усі листові вузли мають однакову довжину шляху, і всі можливі слоти вузлів зайняті.

Бінарне дерево перекошене праворуч

Бінарне дерево, у якому або кожен вузол має правий дочірній елемент, або не має дочірнього елемента (є листом), називається *перекошеним праворуч бінарним деревом.*

Бінарне дерево перекошене ліворуч

Бінарне дерево, у якому або кожен вузол має лівий дочірній елемент, або не має дочірнього елемента (є листом), називається *перекошеним ліворуч бінарним деревом.*

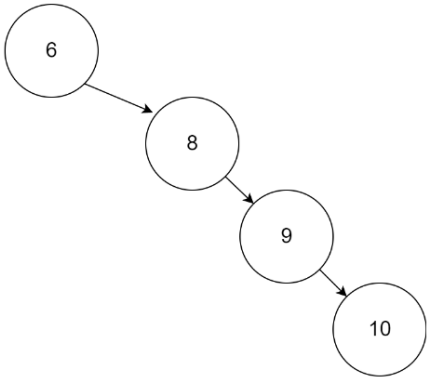


Рис. 6.7. Бінарне дерево, перекошене праворуч

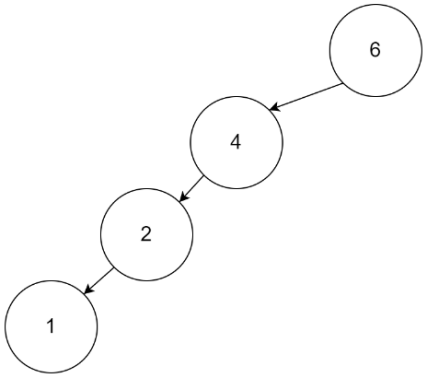


Рис. 6.8. Бінарне дерево, перекошене ліворуч

Збалансоване бінарне дерево

Збалансоване бінарне дерево – це бінарне дерево, у якому ліве та праве піддерева для будь-якого заданого вузла відрізняються висотою максимум на одиницю. AVL-дерево і RB-дерево є прикладами дерев зі збалансованою висотою.

Примітка. Кожне повне бінарне дерево є збалансованим бінарним деревом.

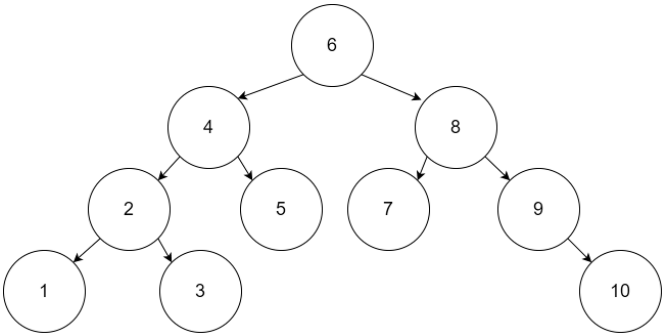


Рис. 6.9. Збалансоване бінарне дерево

Задачі з деревами

Формування бінарного дерева

Задача: створити повне двійкове дерево зі значень, заданих у вигляді масиву.

Розв'язок: щоб створити повне бінарне дерево, ми будемо заповнювати елементи в дереві по рівнях, починаючи з рівня 0.

Етапи створення повного бінарного дерева такі:

• Вставити перший елемент у масиві (індекс i = 0) як кореневий вузол на рівні 0 дерева.

• Обчислити індекс лівого дочірнього елемента як 2 * i + 1 та додати його в ліве піддерево кореня.

• Обчислити індекс правого дочірнього елемента як 2 * i + 2 і додати його у праве піддерево кореня.

• Рекурсивно застосувати ці дії для лівого та правого вузлів піддерева.

```
fun createCompleteBinaryTree(arr: IntArray) {
    root = createCompleteBinaryTree(arr, 0)
}

fun createCompleteBinaryTree(arr: IntArray, start: Int): Node {
    val size = arr.size
    val curr = Node(arr[start])
    val leftIndex = 2 * start + 1
    val rightIndex = 2 * start + 2
```

```
    if (leftIndex < size)
        curr.left = createCompleteBinaryTree(arr, leftIndex)
    if (rightIndex < size)
        curr.right = createCompleteBinaryTree(arr, rightIndex)
    return curr
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printPreOrder()
}
```

Результат:

```
1 2 4 8 9 5 10 3 6 7
```

Аналіз складності: це ефективний алгоритм для створення повного бінарного дерева.

Часова складність: O(n). *Просторова складність*: O(n).

Обхід дерева у префіксному порядку (Pre-Order Traversal)

Задача: виконати обхід дерева у префіксному порядку.

Розв'язок: під час префіксного порядку обходу спочатку відвідується / обходиться сам вузол, потім його лівий дочірній елемент (або ліве піддерево), а потім – його правий дочірній елемент (або праве піддерево). Цей підхід виконується рекурсивно для кожного вузла в дереві, поки ми не досягнемо листкового вузла.

```
fun printPreOrder() {
    printPreOrder(root)
    println()
}

fun printPreOrder(node: Node?) {
    if (node != null) {
        print("${node.value} ")
        printPreOrder(node.left)
        printPreOrder(node.right)
    }
}
```

```
// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printPreOrder()
}
```

Результат:

```
1 2 4 8 9 5 10 3 6 7
```

Часова складність: O(n). *Просторова складність*: O(n).

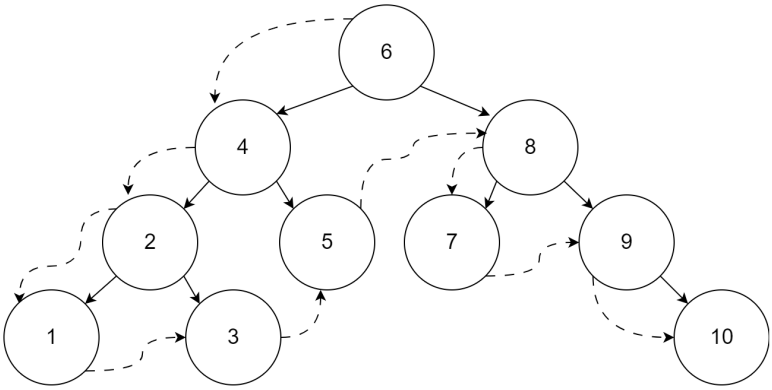


Рис. 6.10. Префіксний порядок обходу дерева

Примітка. Якщо розглядається алгоритм, у якому обходяться всі вузли, то складність не може бути меншою ніж O(n). Але якщо є велика частина дерева, яку не треба обходити, то складність може зменшитись.

Обхід дерева у постфіксному порядку (Post-Order Traversal)

Задача: виконати обхід дерева у постфіксному порядку.

Розв'язок: під час постфіксного порядку обходу спочатку відвідується / обходиться лівий дочірній елемент вузла (або ліве піддерево), потім – його правий дочірній елемент (або праве піддерево) і лише тоді сам вузол. Цей підхід

виконується рекурсивно для кожного вузла в дереві, поки ми не досягнемо листкового вузла.

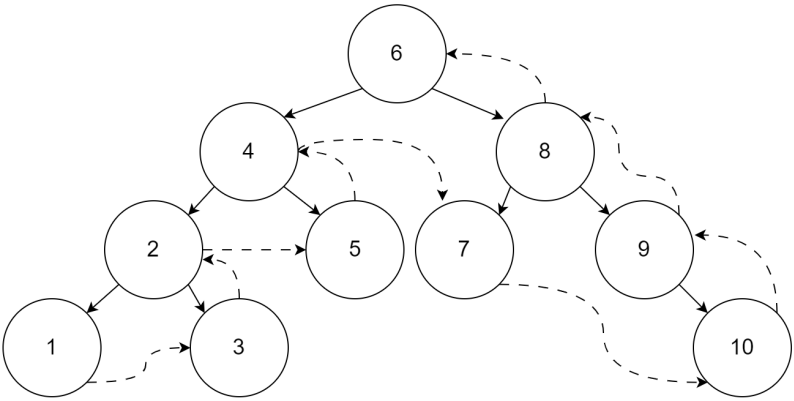


Рис. 6.11. Постфіксний порядок обходу дерева

```
fun printPostOrder() {
    printPostOrder(root)
    println()
}
private fun printPostOrder(node: Node?) { /* post order */
    if (node != null) {
        printPostOrder(node.left)
        printPostOrder(node.right)
        print("${node.value} ")
    }
}

// Testing code.
fun main() {
     val t = Tree()
     val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
     t.createCompleteBinaryTree(arr)
     t.printPostOrder()
}
```

Результат:

```
8 9 4 10 5 2 6 7 3 1
```

Часова складність: O(n). *Просторова складність*: O(n).

Обхід дерева в інфіксному порядку (In-Order Traversal)

Задача: виконати обхід дерева в інфіксному порядку.

Розв'язок: під час інфіксного порядку обходу спочатку відвідується / обходиться лівий дочірній елемент (або ліве піддерево) вузла, потім сам вузол, а наостанок – його правий дочірній елемент (або праве піддерево). Цей підхід виконується рекурсивно для кожного вузла в дереві, поки ми не досягнемо листкового вузла.

Примітка. Результатом обходу бінарного дерева пошуку в інфіксному порядку буде відсортований у порядку зростання список елементів.

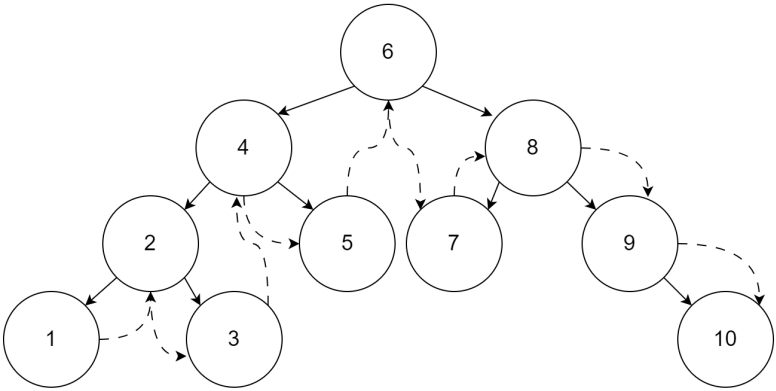


Рис. 6.12. Інфіксний порядок обходу дерева

```
fun printInOrder() {
    printInOrder(root)
    println()
}

private fun printInOrder(node: Node?) { /* In order */
    if (node != null) {
        printInOrder(node.left)
        print("${node.value} ")
        printInOrder(node.right)
    }
}
```

```
// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printInOrder()
}
```

Результат:

```
8 4 9 2 10 5 1 6 3 7
```

Часова складність: O(n). *Просторова складність*: O(n).

Примітка. Префіксний, постфіксний та інфіксний способи обходу призначені для всіх бінарних дерев. Їх можна використовувати для обходження будь-якого типу бінарного дерева.

Обхід дерева по рівнях (обхід дерева в ширину)

Задача: написати код для реалізації обходу дерева по рівнях. Тобто таким чином, щоб вузли на глибині k були опрацьовані перед вузлами на глибині k+1.

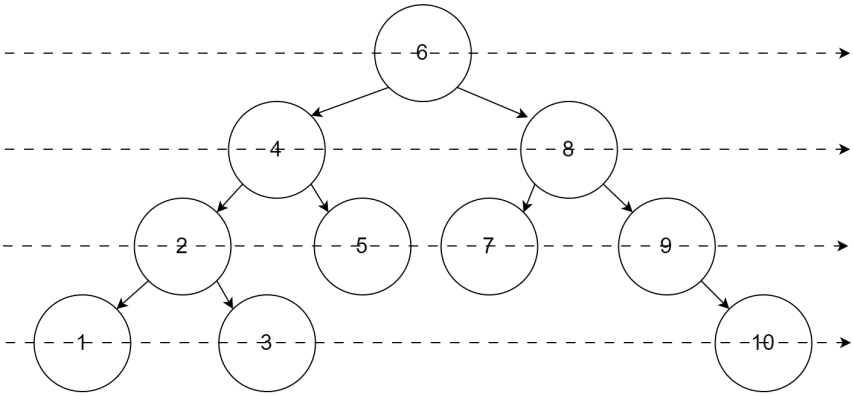


Рис. 6.13. Обхід дерева по рівнях

Розв'язок: обхід дерева по рівнях або обхід дерева в ширину здійснюється за допомогою черги. Спочатку

покажчик кореневого вузла додається до черги. Обхід дерева виконується, поки черга не буде порожньою. Коли ми обходимо дерево, ми спочатку видаляємо елемент із черги, опрацьовуємо значення, що зберігається в цьому вузлі, а потім його лівий і правий дочірній елементи додаються до черги.

```
fun printBreadthFirst() {
    val que = ArrayDeque<Node>()
    var temp: Node
    if (root != null) que.addLast(root!!)
    while (!que.isEmpty()) {
        temp = que.removeFirst()
        print("${temp.value} ")
        if (temp.left != null) que.addLast(temp.left!!)
        if (temp.right != null) que.addLast(temp.right!!)
    }
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printBreadthFirst()
}
```

Результат:

```
1 2 3 4 5 6 7 8 9 10
```

Часова складність: O(n). *Просторова складність*: O(n).

Обхід (друк) дерева у глибину без використання рекурсії

Задача: виконати пошук у бінарному дереві у глибину без використання рекурсії.

Розв'язок: обхід дерева у глибину виконується за допомогою рекурсії (із використанням системного стека). Однак те ж саме можна реалізувати за допомогою структури даних «Стек». На початку посилання кореневий вузол додається до стека. Проходимо все дерево, доки стек не буде порожнім. У кожній ітерації елемент витягується зі стека, його значення опрацьовується (друкується на екрані). Потім правий та лівий дочірні елементи вузла додаються до стека. Наступний код демонструє такий підхід.

```
fun printDepthFirst() {
    val stk = ArrayDeque<Node>()
    var temp: Node
    if (root != null) stk.addLast(root!!)
    while (!stk.isEmpty()) {
        temp = stk.removeLast()
        print("${temp.value} ")
        if (temp.right != null) stk.addLast(temp.right!!)
        if (temp.left != null) stk.addLast(temp.left!!)
    }
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printDepthFirst()
}
```

Результат:

```
1 2 4 8 9 5 10 3 6 7
```

Часова складність: O(n). *Просторова складність*: O(n).

Вивести (надрукувати) дерево по рівнях (рядок за рядком)

Задача: виконати обхід бінарного дерева в порядку рівнів, щоб можна було друкувати рівні рядок за рядком.

Перший розв'язок: ми будемо використовувати дві черги для виконання обходу по рівнях. Як варіант, ми обробляємо черги та розміщуємо дочірні елементи черги в іншу чергу, щоб, коли кожен рівень оброблено, ми мали змогу надрукувати вихідні дані в різних рядках.

```
fun printLevelOrderLineByLine() {
    val que1 = ArrayDeque<Node>()
    val que2 = ArrayDeque<Node>()

    if (root != null) que1.addLast(root!!)
    while (que1.isNotEmpty() || que2.isNotEmpty()) {
        processQueues(que1, que2)
        processQueues(que2, que1)
    }
}
```

```
fun processQueues(que1: ArrayDeque<Node>, que2: ArrayDeque<Node>) {
    while (que1.isNotEmpty()) {
        val temp = que1.removeFirst()
        print("${temp.value} ")
        if (temp.left != null) que2.addLast(temp.left!!)
        if (temp.right != null) que2.addLast(temp.right!!)
    }
    println()
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printLevelOrderLineByLine()
}
```

Результат:

```
1
2 3
4 5 6 7
8 9 10
```

Другий розв'язок: ми можемо розв'язати цю проблему за допомогою однієї черги. Припустимо, у будь-який час ми маємо всі вузли k-го рівня, присутні в черзі. Очевидно, ми можемо знайти кількість цих елементів. Будемо обробляти елементи з черги, знайдену кількість разів і додавати їх дочірні елементи до черги. Потім ми можемо надрукувати новий рядок. На цей момент матимемо всі вузли на рівні k+1. Починаючи з додавання кореневого вузла, як першого рівня черги, треба виконати описані вище дії.

```
fun printLevelOrderLineByLine2() {
    val que = ArrayDeque<Node>()
    var count: Int
    if (root != null) que.addLast(root!!)
    while (que.isNotEmpty()) {
        count = que.size
        while (count > 0) {
            val temp = que.removeFirst()
            print(temp.value.toString() + " ")
            if (temp.left != null) que.addLast(temp.left!!)
```

```
            if (temp.right != null) que.addLast(temp.right!!)
            count -= 1
        }
        println()
    }
}
```

Вивести всі шляхи в дереві

Задача: дано бінарне дерево, надрукувати всі шляхи від кореня до листа.

Розв'язок: ми будемо використовувати стек і застосуємо пошук у глибину (DFS). Щоразу, коли ми переходимо через вузол, ми додаємо цей вузол до стека. Коли ми досягаємо листа, ми друкуємо весь стек (від кореневого вузла до цього листового вузла). Коли ми повертаємося з функції, ми видаляємо елемент, який був доданий до стека під час входу до неї.

```
fun printAllPath() {
    val stk = ArrayDeque<Int>()
    printAllPathUtil(root, stk)
}

private fun printAllPathUtil(curr: Node?, stk: ArrayDeque<Int>) {
    if (curr == null) return
    stk.add(curr.value)
    if (curr.left == null && curr.right == null) {
        println(stk)
        stk.removeLast()
        return
    }
    printAllPathUtil(curr.right, stk)
    printAllPathUtil(curr.left, stk)
    stk.removeLast()
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.printAllPath()
}
```

Результат:

```
[1, 3, 7]
[1, 3, 6]
[1, 2, 5, 10]
[1, 2, 4, 9]
[1, 2, 4, 8]
```

Часова складність: O(n). *Просторова складність*: O(n).

Кількість вузлів у бінарному дереві

Задача: визначити загальну кількість вузлів у бінарному дереві.

Розв'язок: знаходимо кількість вузлів у бінарному дереві шляхом додавання кількості вузлів у правому піддереві до кількості вузлів у лівому піддереві. А наостанок додаємо до цієї суми 1.

```
fun countNodes(curr: Node? = root): Int {
    return if (curr == null) 0 else 1 + countNodes(curr.right) + countNodes(curr.left)
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.countNodes())
}
```

Результат:

```
10
```

Часова складність: O(n). *Просторова складність*: O(n).

Сума значень у вузлах бінарного дерева

Задача: задано бінарне дерево, знайти суму значень в усіх його вузлах.

Розв'язок: обчислимо суму вузлів рекурсією. Спочатку, використовуючи функцію sumAllBT(), обчислимо суму всіх вузлів лівого піддерева, а потім, таким же чином – суму вузлів правого піддерева. Наостанок додамо значення поточного вузла до цієї суми та повертаємо остаточну суму.

```
fun sumAllBT(curr: Node? = root): Int {
    return if (curr == null) 0
    else curr.value + sumAllBT(curr.left) + sumAllBT(curr.right)
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.sumAllBT())
}
```

Результат:

```
55
```

Часова складність: O(n). *Просторова складність*: O(n).

Кількість листових вузлів

Задача: задано бінарне дерево, знайти кількість листових вузлів у ньому.

Розв'язок: обчислюємо загальну кількість листових вузлів у дереві, додавши кількість листових вузлів у правому піддереві та кількість листових вузлів у лівому піддереві.

```
fun countLeafNodes(curr: Node? = root): Int {
    return if (curr == null) 0
    else if (curr.left == null && curr.right == null) 1
    else countLeafNodes(curr.right) + countLeafNodes(curr.left)
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.countLeafNodes())
}
```

Результат:

```
5
```

Часова складність: O(n). *Просторова складність*: O(n).

Кількість повних вузлів у бінарному дереві

Задача: дано бінарне дерево, знайти кількість повних вузлів у ньому. Повний вузол має ненульові ліве та праве піддерева.

Розв'язок: повний вузол – це вузол, який має і ліве, і праве піддерева. Ми будемо рекурсивно проходити все дерево та збільшувати лічильник кількості повних вузлів, коли ми його знайдемо.

```
fun countFullNodes(curr: Node? = root): Int {
    if (curr == null) return 0
    var count = countFullNodes(curr.right) + countFullNodes(curr.left)
    if (curr.right != null && curr.left != null) count++
    return count
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.countFullNodes())
}
```

Результат:

```
4
```

Часова складність: O(n). *Просторова складність*: O(n).

Пошук значення у двійковому дереві

Задача: пошук певного значення у двійковому дереві.

Розв'язок: щоб дізнатися, чи присутнє значення у двійковому дереві, ми використовуємо алгоритм повного пошуку. По-перше, значення в поточному вузлі порівнюється з нашим ключем, якщо збігається, ми завершуємо пошук, якщо ні, ми рекурсивно шукаємо ключ у лівому та правому піддеревах. Ми зупиняємось, коли знаходимо бажаний ключ у дереві та повертаємо true, або, якщо дерево пройдено повністю без знаходження ключа, повертаємо false.

```
fun searchInBinaryTree(value: Int): Boolean {
    return searchInBinaryTreeeUtil(root, value)
}
private fun searchInBinaryTreeeUtil(curr: Node?, value: Int): Boolean {
```

```
    return if (curr == null) false
    else if (curr.value == value || searchInBinaryTreeeUtil(curr.left, value) ||
        searchInBinaryTreeeUtil(curr.right,value)) true
    else false
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.searchInBinaryTree(9))
}
```

Результат:

```
true
```

Часова складність: O(n). *Просторова складність*: O(n).

Знайти максимальне значення в бінарному дереві

Задача: дано двійкове дерево, знайти в ньому максимальне значення.

Розв'язок: розв'язуємо цю задачу, рекурсивно обходячи вузли бінарного дерева. По-перше, знайдемо відповідні максимальні значення в лівому та правому піддеревах вузла, а потім порівняємо ці значення зі значенням поточного вузла. Нарешті, повертаємо найбільше з цих трьох значень.

```
fun findMaxInBinaryTree(curr: Node? = root): Int {
    if (curr == null) return Int.MIN_VALUE
    var max = curr.value
    val leftMax = findMaxInBinaryTree(curr.left)
    val rightMax = findMaxInBinaryTree(curr.right)
    if (leftMax > max) max = leftMax
    if (rightMax > max) max = rightMax
    return max
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.findMaxInBinaryTree())
}
```

Результат:

```
10
```

Висота (глибина) дерева

Задача: задано бінарне дерево, знайти його глибину.

Розв'язок: ми знаходимо глибину дерева, рекурсивно обходячи лівий і правий дочірній елемент кореня. На кожному рівні обходу розраховується глибина лівого та правого піддерева. Більша глибина серед лівого та правого нащадків додається до 1 (це глибина поточного вузла). Нарешті, це значення повертається.

```
fun treeDepth(curr: Node? = root): Int {
    return if (curr == null) 0 else {
        val lDepth = treeDepth(curr.left)
        val rDepth = treeDepth(curr.right)
        if (lDepth > rDepth) lDepth + 1 else rDepth + 1
    }
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.treeDepth())
}
```

Результат:

```
4
```

Часова складність: O(n). *Просторова складність*: O(n).

Максимальна довжина шляху в бінарному дереві / Діаметр бінарного дерева

Задача: задано бінарне дерево, знайти в ньому шлях максимальної довжини.

Розв'язок: щоб знайти діаметр бінарного дерева, потрібно знайти глибину лівого дочірнього та правого дочірнього елементів. Потім ці два значення глибини додаються та збільшуються на одиницю, щоб отримати

максимальну довжину шляху (діаметр кандидата), який містить поточний вузол. Тоді ми знайдемо шлях максимальної довжини зліва дочірнього піддерева. Потім знайдемо максимальну довжину шляху у правому дочірньому піддереві. Наостанок порівняємо три значення та повертаємо максимальне значення серед них. Це значення буде діаметром бінарного дерева.

```
fun maxLengthPathInBinaryTree(curr: Node? = root): Int { // diameter
    if (curr == null) return 0
    val leftPath = treeDepth(curr.left)
    val rightPath = treeDepth(curr.right)
    var max = leftPath + rightPath + 1
    val leftMax = maxLengthPathInBinaryTree(curr.left)
    val rightMax = maxLengthPathInBinaryTree(curr.right)
    if (leftMax > max) max = leftMax
    if (rightMax > max) max = rightMax
    return max
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.maxLengthPathInBinaryTree())
}
```

Результат:

```
6
```

Копіювати дерево

Задача: задано бінарне дерево, скопіюйте його значення в інше бінарне дерево.

Розв'язок: копія дерева створюється копіюванням вузлів вхідного дерева на кожному рівні обходу до вихідного дерева. На кожному рівні обходу створюється новий вузол і до нього копіюється значення вузла вхідного дерева. Ліве піддерево копіюється рекурсивно, а потім покажчик повертає посилання на нове піддерево, яке потім буде призначено лівому дочірньому посиланню поточного нового вузла. Подібним

чином цей процес виконується для правого піддерева. Таким чином, дерево копіюється повністю.

```
fun copyTree(): Tree {
    val tree2 = Tree()
    tree2.root = copyTree(root)
    return tree2
}
private fun copyTree(curr: Node?): Node? {
    return if (curr == null) null else {
        val temp: Node = Node(curr.value)
        temp.left = copyTree(curr.left)
        temp.right = copyTree(curr.right)
        temp
    }
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    val t2 = t.copyTree()
    t2.printInOrder()
}
```

Результат:

```
8 4 9 2 10 5 1 6 3 7
```

Часова складність: O(n). *Просторова складність*: O(n).

Ідентичність

Задача: два дерева вважаються ідентичними, якщо на кожному рівні значення у відповідних вузлах є рівними.

```
fun isEqual(T2: Tree): Boolean {
    return isEqualUtil(root, T2.root)
}
private fun isEqualUtil(node1: Node?, node2: Node?): Boolean {
    return if (node1 == null && node2 == null) true
    else if (node1 == null || node2 == null) false
    else isEqualUtil(node1.left, node2.left) && isEqualUtil(node1.right,
        node2.right) && node1.value == node2.value
}

// Testing code.
fun main() {
    val t = Tree()
```

```
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    val t2 = t.copyTree()
    println(t.isEqual(t2))
}
```

Результат:

```
true
```

Часова складність: O(n). *Просторова складність*: O(n).

Знищити дерево

Задача: задано бінарне дерево, звільнити всі його вузли.

Розв'язок: кореню дерева присвоюємо значення null. Вузли дерева буде видалено збиральником сміття (garbage collector).

```
fun free() {
    root = null
}
```

Чи є дерево повним?

Задача: задано бінарне дерево, з'ясуйте, чи є воно повним деревом. Дерево повне, якщо воно повністю заповнено на всіх рівнях, крім, можливо, останнього рівня. Останній рівень заповнюється зліва направо.

Перший розв'язок: ми виконаємо обхід дерева в ширину за допомогою черги. Оскільки в повному дереві, якщо ми отримуємо вузол, який не має лівого дочірнього елемента, він також не може мати правого дочірнього елемента. І якщо знайдемо вузол, який не має жодного дочірнього елемента, тоді жоден інший вузол у обході в ширину не може мати дочірнього елемента.

```
fun isCompleteTree(): Boolean {
    val que = ArrayDeque<Node>()
    var temp: Node?
    var noChild = 0
    if (root != null) que.add(root!!)
    while (que.size != 0) {
        temp = que.removeFirst()
        if (temp.left != null) {
```

```
            if (noChild == 1) return false
            que.add(temp.left!!)
        } else noChild = 1
        if (temp.right != null) {
            if (noChild == 1) return false
            que.add(temp.right!!)
        } else noChild = 1
    }
    return true
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.isCompleteTree())
}
```

Результат:

```
true
```

Другий розв'язок: ми можемо розв'язати цю проблему за допомогою рекурсії, розглядаючи дане дерево як купу. Якщо ми вважаємо, що розташування батьківського елемента дорівнює index, то розташування лівого дочірнього елемента має бути 2*index+1 і розташування правого дочірнього елемента має бути 2*index+2. Якщо це правильно для кожного вузла, то ми маємо повне дерево.

```
private fun isCompleteTreeUtil(curr: Node?, index: Int, count: Int): Boolean {
    return if (curr == null) return true
    else if (index > count) false
    else isCompleteTreeUtil(curr.left, index * 2 + 1, count)
            && isCompleteTreeUtil(curr.right, index * 2 + 2, count)
}
fun isCompleteTree2(): Boolean {
    val count = countNodes()
    return isCompleteTreeUtil(root, 0, count)
}
```

Чи є дерево представленням купи?

Задача: дано бінарне дерево, треба з'ясувати, чи воно представляє Min-Heap.

Щоб перевірити, чи є дерево купою, потрібно перевірити дві умови:

1. Це повне дерево.
2. Значення батьківського вузла менше або дорівнює значенням у його лівому та правому дочірньому вузлах.

Розв'язок: потрібно з'ясувати, чи задане дерево є повним і чи дотримується властивість «батьківсько-дочірній» купи (батьківські вузли мають значення, яке менше або дорівнює дочірнім вузлам), тоді це дерево становить мінімальну купу.

Виклик функції isCompleteTree() займає лінійний час, як і функція isHeap(). Тому загальна *часова складність* дорівнює O(n). Усе дерево обходять три рази, спочатку, щоб знайти загальну кількість елементів у дереві, потім ще раз, щоб визначити, чи це повне дерево, і, нарешті, для тестування властивості купи.

```
fun isHeapUtil(curr: Node?, parentValue: Int): Boolean {
    return if (curr == null) true
    else if (curr.value < parentValue) false
    else isHeapUtil(curr.left, curr.value) && isHeapUtil(curr.right,curr.value)
}

fun isHeap(): Boolean {
    val infinite = -9999999
    return isCompleteTree() && isHeapUtil(root, infinite)
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    println(t.isHeap())
}
```

Результат:

```
true
```

Ітераційний обхід у префіксному порядку

Задача: задано бінарне дерево. Виконати обхід дерева у префіксному порядку без використання рекурсії.

Розв'язок: замість використання системного стека в рекурсії ми можемо обійти дерево за допомогою структури даних «стек».

```
fun iterativePreOrder() {
    val stk = ArrayDeque<Node>()
    var curr: Node
    if (root != null) stk.add(root!!)
    while (!stk.isEmpty()) {
        curr = stk.removeLast()
        print(curr.value.toString() + " ")
        if (curr.right != null) stk.add(curr.right!!)
        if (curr.left != null) stk.add(curr.left!!)
    }
    println()
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createCompleteBinaryTree(arr)
    t.iterativePreOrder()
}
```

Результат:

```
1 2 4 8 9 5 10 3 6 7
```

Часова складність: O(n). *Просторова складність*: O(n).

Бінарне дерево пошуку

Двійкове дерево пошуку (BST) – це двійкове дерево, у якому вузли впорядковані таким чином:

- Ключ у лівому піддереві менший за ключ у його батьківському вузлі.
- Ключ у правому піддереві більший за ключ у його батьківському вузлі.
- Не допускається дублювання ключа.

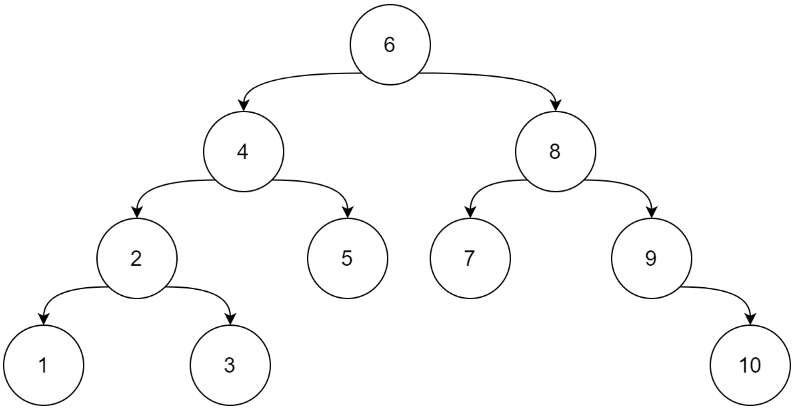


Рис. 6.14. Бінарне дерево пошуку

Розглядаючи наступні питання, варто мати на увазі, що у вузлі дерева може бути два окремих поля ключа та значення (і навіть більше). Однак для простоти ми будемо вважати, що значення у вузлі і є ключем. Усі задачі в бінарному дереві пошуку розв'язуються з урахуванням того, що значення у вузлі є ключами для дерева.

Оскільки бінарне дерево пошуку є двійковим деревом, тому всі алгоритми бінарного дерева, що були розглянуті раніше, можуть бути застосовані до бінарного дерева пошуку.

Задачі з бінарним деревом пошуку

Формування бінарного дерева пошуку з відсортованого масиву

Задача: створити бінарне дерево пошуку з масиву значення, у якому йдуть у відсортованому порядку.

Розв'язок: оскільки елементи в масиві відсортовані, ми хочемо створити бінарне дерево пошуку, у якому ліві вузли піддерева мають значення, менші за поточний вузол, а вузли правого піддерева мають значення більше ніж значення

поточного вузла. Нам доведеться знайти середній вузол, щоб створити поточний вузол і відправити решту масиву для побудови лівого та правого піддерев.

```
fun createBinarySearchTree(arr: IntArray) {
    root = createBinarySearchTree(arr, 0, arr.size - 1)
}

private fun createBinarySearchTree(arr: IntArray, start: Int, end: Int): Node? {
    if (start > end) return null
    val mid = (start + end) / 2
    val curr = Node(arr[mid])
    curr.left = createBinarySearchTree(arr, start, mid - 1)
    curr.right = createBinarySearchTree(arr, mid + 1, end)
    return curr
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    t.printInOrder()
}
```

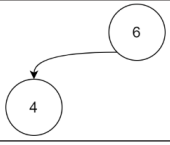
Результат:

```
1 2 3 4 5 6 7 8 9 10
```

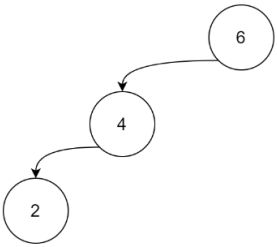
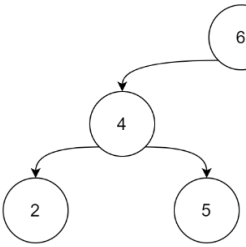
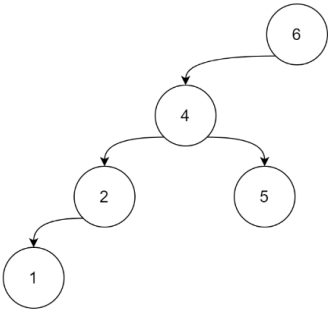
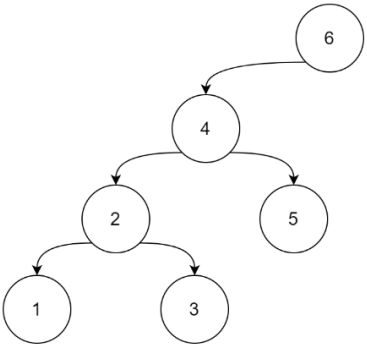
Задача: вставка елемента в бінарне дерево пошуку.

Розглянемо процес додавання елементів у дерево по кроках. Нижче наведено дерево після вставки вузлів один за одним. Вставляємо вузли з ключами 6, 4, 2, 5, 1, 3, 8, 7, 9, 10.

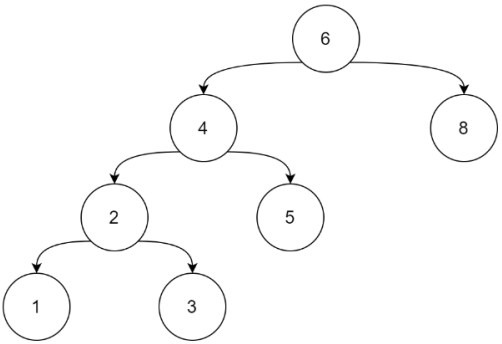
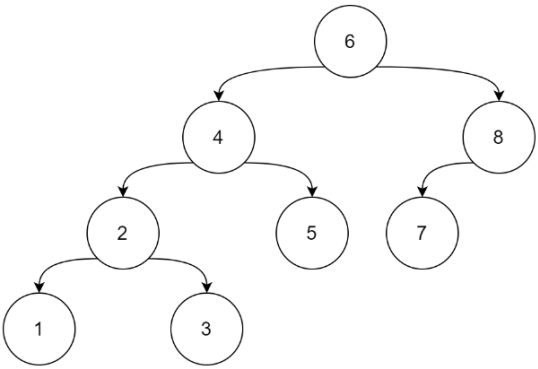
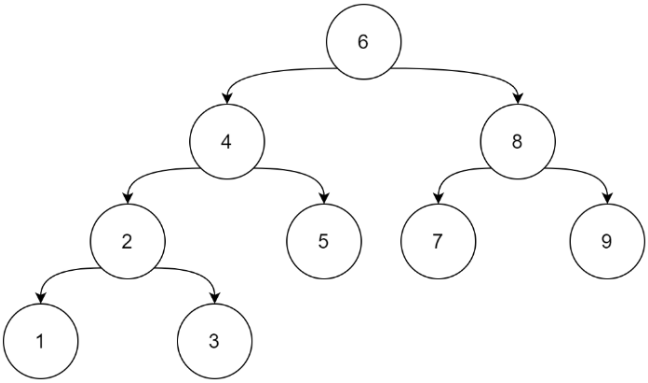
Таблиця 6.1. **Побудова бінарного дерева з відсортованого масиву**

1	
2	

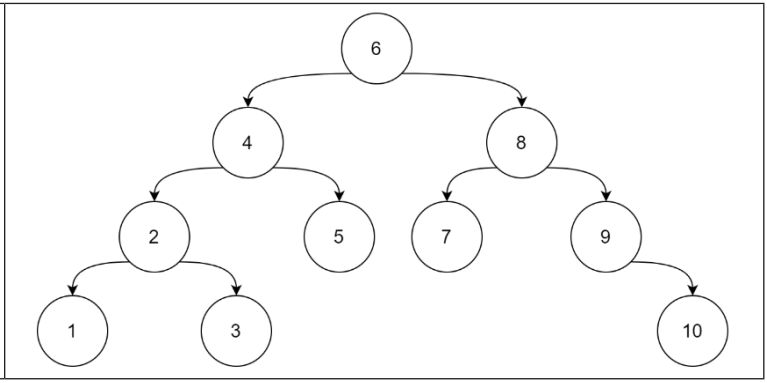
Продовж. табл. 6.1

3	
4	
5	
6	

Продовж. табл. 6.1

7	
8	
9	

Продовж. табл. 6.1

10	

Розв'язок: менші значення додаються до лівого дочірнього піддерева поточного вузла, а більші значення додаються до правого дочірнього піддерева поточного вузла.

```
fun insert(value: Int) {
    root = insert(root, value)
}
private fun insert(node: Node?, value: Int): Node {
    if (node == null) {
        return Node(value, null, null)
    } else if (node.value > value) {
        node.left = insert(node.left, value)
    } else {
        node.right = insert(node.right, value)
    }
    return node
}

// Testing code.
fun main() {
    val t = Tree()
    t.insert(6)
    t.insert(4)
    t.insert(2)
    t.insert(5)
    t.insert(1)
    t.insert(3)
    t.insert(8)
    t.insert(7)
    t.insert(9)
```

```
    t.insert(10)
    t.printInOrder()
}
```

Результат:

```
1 2 3 4 5 6 7 8 9 10
```

Часова складність: O(n). *Просторова складність*: O(n).

Пошук вузла

Задача: знайти вузол, що містить указане значення.

Розв'язок: у бінарному дереві пошуку значення, яке перевищує поточне значення вузла, завжди буде знаходитись у правому піддереві, а значення, яке менше значення у поточному вузлі, повинно бути у лівому піддереві.

Таким чином, ми можемо знайти бажане значення ключа, обходячи ліве та праве піддерева ітеративно.

```
fun find(value: Int): Boolean {
    var curr = root
    while (curr != null) {
        curr = if (curr.value == value) {
            return true
        } else if (curr.value > value) {
            curr.left
        } else {
            curr.right
        }
    }
    return false
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    println(t.find(3))
    println(t.find(16))
}
```

Результат:

```
true
false
```

Часова складність: O(n). *Просторова складність*: O(1).

Пошук мінімального

Задача: знайти вузол з мінімальним значенням.

Розв'язок: крайній лівий дочірній елемент дерева буде вузлом із мінімальним значенням. Ми будемо ітеративно переходити від кореня до його лівого дочірнього елемента, доки ми не досягнемо крайнього лівого вузла.

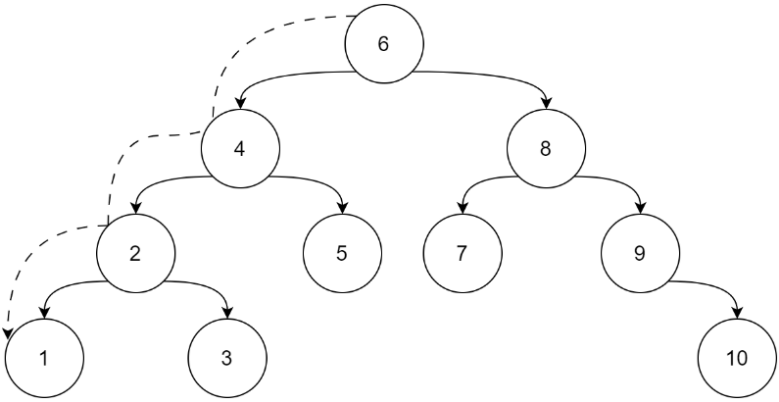


Рис. 6.15. Пошук мінімального елемента в бінарному дереві пошуку

```
fun findMin(): Int {
    var node: Node? = root ?: return Int.MAX_VALUE
    while (node!!.left != null) {
        node = node.left
    }
    return node.value
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    println(t.findMin())
}
```

Результат:

```
1
```

```
fun findMinNode(curr: Node?): Node? {
    var node: Node? = curr ?: return null
    while (node!!.left != null) {
        node = node.left
    }
    return node
}
```

Часова складність: O(n). *Просторова складність*: O(1).

Пошук максимального

Задача: знайти вузол з максимальним значенням.

Розв'язок: крайній правий дочірній елемент дерева буде вузлом із максимальним значенням.

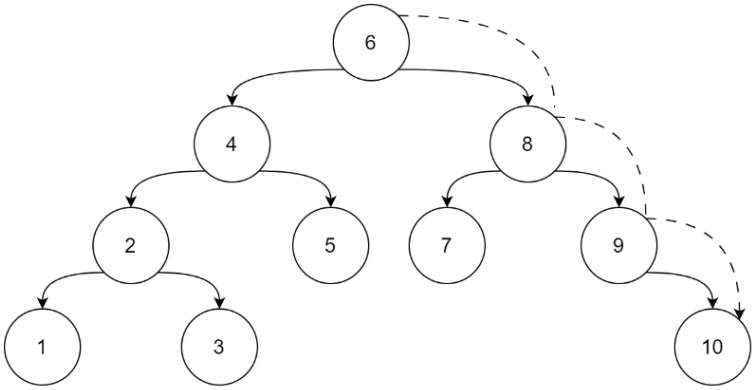


Рис. 6.16. Пошук максимального елемента в бінарному дереві пошуку

```
fun findMax(): Int {
    var node: Node? = root ?: return Int.MIN_VALUE
    while (node!!.right != null) {
        node = node.right
    }
    return node.value
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
    t.createBinarySearchTree(arr)
    println(t.findMax())
}
```

Результат:

```
10
```

```
fun findMaxNode(curr: Node?): Node? {
    var node: Node? = curr ?: return null
    while (node!!.right != null) {
        node = node.right
    }
    return node
}
```

Часова складність: O(n). *Просторова складність*: O(1).

Чи є дерево бінарним деревом пошуку?

Задача: перевірити, чи є задане дерево бінарним деревом пошуку.

Перший розв'язок: на кожному вузлі ми будемо перевіряти, чи максимальне значення лівого піддерева менше ніж значення поточного вузла та мінімальне значення правого піддерева більше ніж поточний вузол. Якщо ці умови виконані, то дане бінарне дерево є BST.

```
fun isBST1(curr: Node? = root): Boolean {
    return
        if (curr == null) true
        else if (curr.left != null &&
                 findMaxNode(curr.left)!!.value > curr.value) false
        else if (curr.right != null &&
                 findMinNode(curr.right)!!.value <= curr.value) false
        else isBST1(curr.left) && isBST1(curr.right)
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    println("Is BST: ${t.isBST1()}")
}
```

Результат:

```
Is BST: true
```

Часова складність: O(n). *Просторова складність*: O(n).

Хоча зазначене вище рішення є правильним, воно неефективне, оскільки одні й ті ж вузли дерева перебираються багато разів.

Другий розв'язок: кращим рішенням буде те, у якому ми будемо перебирати кожен вузол лише один раз. Це можна зробити, звужуючи діапазон. Для цього будемо використовувати функцію IsBST(), яка приймає максимальний та мінімальний діапазон значень вузлів. Початкові значення *min* та *max* буде мінімальним та максимальним значенням цілого числа в мові Kotlin (Int.MIN_VALUE та Int.MAX_VALUE).

```
fun isBST(curr: Node? = root,
          min: Int = Int.MIN_VALUE,
          max: Int = Int.MAX_VALUE): Boolean {
    return if (curr == null) true
    else if (curr.value < min || curr.value > max) false
    else isBST(curr.left, min, curr.value) && isBST(curr.right, curr.value, max)
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    println("Is BST: ${t.isBST()}")
}
```

Результат:

```
Is BST: true
```

Часова складність: O(n). *Просторова складність*: O(n).

Видалення вузла

Задача: видалити вузол *x* з бінарного дерева пошуку, переорганізувати вузли бінарного дерева пошуку, щоб зберегти його необхідні властивості.

Ми маємо три випадки для видалення вузла. Для простоти вузол, який потрібно видалити, позначимо як *x*.

Випадок 1: вузол *x* не має дочірніх вузлів. Просто видаліть його (тобто змініть батьківський вузол, щоб він не вказував на *x*).

Випадок 2: вузол *x* має один дочірній вузол. Виріжте *x*, з'єднавши батьківський вузол *x* з дочірнім *x*.

Випадок 3: вузол *x* має два дочірні вузли. Виріжте наступника *x* та замініть *x* наступником *x*.

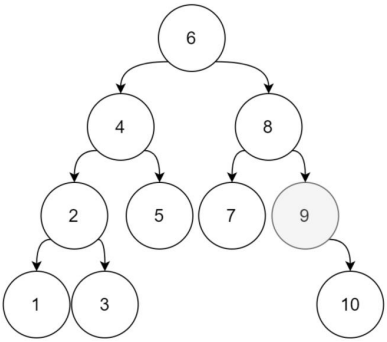
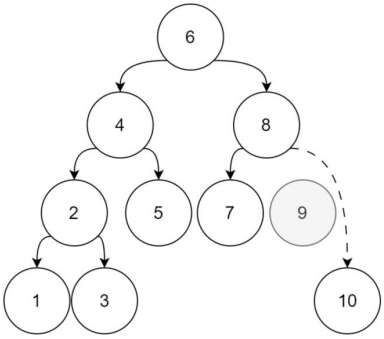
Коли вузол, який потрібно видалити, не має дочірніх вузлів

Це тривіальний випадок, у якому ми безпосередньо видаляємо вузол, повертаючи null.

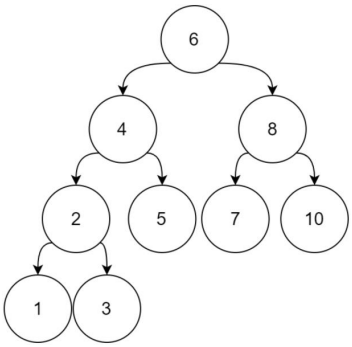
Коли вузол, який потрібно видалити, має лише один дочірній вузол

У цьому випадку ми повертаємо дочірній вузол вузла, що видаляється.

Таблиця 6.2

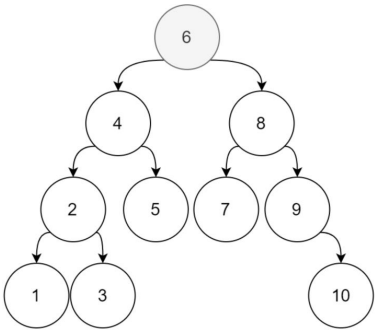
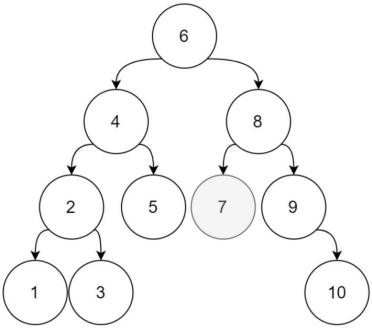
	Ми хочемо видалити вузол зі значенням 9. Цей вузол має лише один дочірній елемент
	Правий дочірній елемент батьківського вузла для вузла зі значенням 9, а саме вузол зі значенням 8 указуватиме на дочірній елемент вузла зі значенням 9, тобто вузол зі значенням 10

Продовж. табл. 6.2

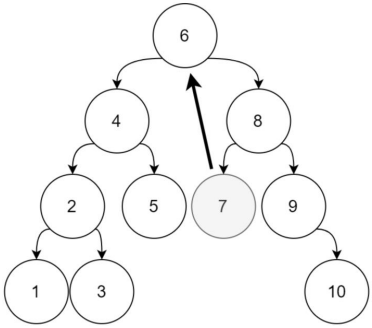
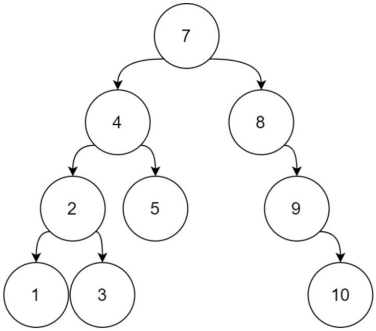
	У результаті, вузол зі значенням 9 видалений з дерева

Коли вузол, який потрібно видалити, має два дочірні вузли

Таблиця 6.3

	Ми збираємось видалити вузол, що містить число 6. Він має два нащадки
	Знаходимо вузол із найменшим значенням у правому піддереві вузла зі значенням 6. Це вузол зі значенням 7

Продовж. табл. 6.3

	Значення із знайденого вузла копіюється у вузол, значення якого видаляється
	Викликається видалення вузла з мінімальним значенням 7 у правому піддереві вузла. Як результат, значення 6 буде видалено з дерева

```
fun deleteNode(value: Int) {
    root = deleteNode(root, value)
}
private fun deleteNode(node: Node?, value: Int): Node? {
    if (node != null) {
        if (node.value == value) {
            if (node.left == null && node.right == null) {
                return null
            } else if (node.left == null) {
                return node.right
            } else if (node.right == null) {
                return node.left
            }
            val minNode = findMinNode(node.right)
            val minValue = minNode!!.value
            node.value = minValue
            node.right = deleteNode(node.right, minValue)
        } else {
            if (node.value > value) {
```

```
                node.left = deleteNode(node.left, value)
            } else {
                node.right = deleteNode(node.right, value)
            }
        }
    }
    return node
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    println("Before delete")
    t.printInOrder()
    println("After delete")
    t.deleteNode(6)
    t.printInOrder()
}
```

Результат:

```
Before delete
1 2 3 4 5 6 7 8 9 10
After delete
1 2 3 4 5 7 8 9 10
```

Часова складність: O(n). *Просторова складність*: O(n).

Надрукувати вузли дерева, які належать до діапазону

Задача: вивести лише ті вузли дерева, значення яких знаходяться в заданому діапазоні.

Розв'язок: ми можемо вивести вузли в заданому діапазоні, обходячи дерево в інфіксному порядку та перевіряючи, чи потрапляє поточне значення в межі заданого діапазону.

```
fun printInRange(min: Int, max: Int) {
    printInRange(root, min, max)
    println()
}
private fun printInRange(root: Node?, min: Int, max: Int) {
    if (root == null) return
    printInRange(root.left, min, max)
    if (root.value in min..max) print(root.value.toString() + " ")
    printInRange(root.right, min, max)
}
```

```
// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    t.printInRange(4, 7)
}
```

Результат:

```
4 5 6 7
```

Знайти значення Ceil і Floor для вказаного ключа у BST

Задача: у даному дереві потрібно знайти найбільше значення в дереві, яке менше за вказане значення, та знайти найменше значення в дереві, яке більше вказаного значення. Таким чином, для заданого значення ми прагнемо знайти найближчі до нього найменше та найбільше значення.

Розв'язок: ми будемо використовувати пошук у BST, щоб знайти Ceil та Floor у BST. Коли ми шукаємо Ceil, якщо ми знаходимо будь-яке значення, яке більше за задане, ми зберігаємо це значення як імовірне значення. Ми звужуємо наш пошук для значення, яке ближче до нашого вхідного значення. Цей алгоритм займає O(log(n)) часу, якщо BST збалансоване. Аналогічно, ми можемо знайти і Floor.

```
fun ceilBST(value: Double): Int {
    var curr = root
    var ceil = Int.MIN_VALUE
    while (curr != null) {
        if (curr.value.toDouble() == value) {
            ceil = curr.value
            break
        } else if (curr.value > value) {
            ceil = curr.value
            curr = curr.left
        } else {
            curr = curr.right
        }
    }
    return ceil
}
```

```
fun floorBST(value: Double): Int {
    var curr = root
    var floor = Int.MAX_VALUE
    while (curr != null) {
        if (curr.value.toDouble() == value) {
            floor = curr.value
            break
        } else if (curr.value > value) {
            curr = curr.left
        } else {
            floor = curr.value
            curr = curr.right
        }
    }
    return floor
}

// Testing code.
fun main() {
    val t = Tree()
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    t.createBinarySearchTree(arr)
    println(t.ceilBST(3.14))
    println(t.floorBST(3.14))
}
```

Результат:

```
4
3
```

Дерево відрізків

Дерево відрізків – це бінарне дерево, яке використовується для виконання декількох запитів до діапазону та оновлення діапазону в масиві.

Приклади задач, для яких дерево відрізків можна використовувати у багатьох задачах. Найбільш відомі серед них такі:

1. Знаходження суми всіх елементів масиву в заданому діапазоні індексів.

2. Знаходження максимального значення масиву в заданому діапазоні індексу.

3. Знаходження мінімального значення масиву в заданому діапазоні індексів (також відоме як задача Range Minimum Query).

Задача: задано масив з *N* чисел. Вам потрібно виконувати наступні операції:

1. Оновити довільний елемент масиву.
2. Знайти максимум у довільному заданому діапазоні (i, j).

Перший розв'язок

Оновлення: просто оновити елемент масиву a[i] = x. Пошук максимуму в діапазоні (i, j), перебираючи елементи: масиву в цьому діапазоні. *Часова складність* оновлення – O(1), а пошуку – O(n).

Другий розв'язок

Створити ще один масив, який буде зберігати префіксну суму. Значення з індексом «i» – це сума перших «i» елементів вхідного масиву.

Основний масив:

1	2	3	4	5	6	7	8	9	10	11

Масив префіксних сум:

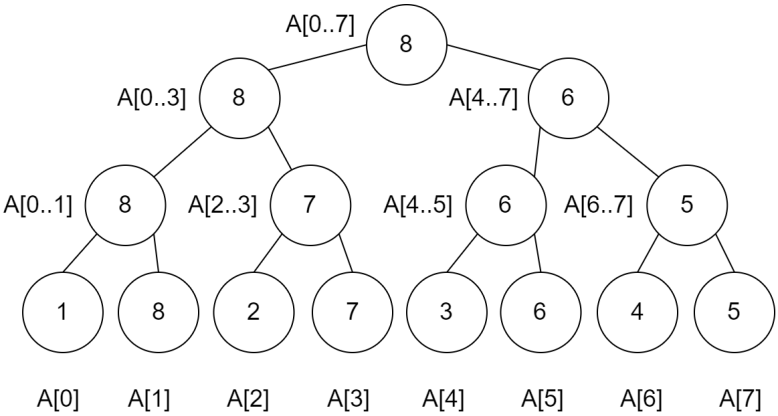
1	3	6	10	15	21	28	36	45	55	66

У разі такого підходу префіксна сума буде обчислюватися за постійний час O(1). Але оновлення одного елемента з індексом «i» призведе до оновлення всіх елементів після цього індексу в масиві префіксних сум. Таким чином, оновлення займе O(n) часу. Цей підхід буде корисним, якщо обчислення префіксної суми відбувається часто, а оновлення – рідко.

Третій розв'язок

Обидва попередні рішення гарні. Однак чи можемо ми отримати хорошу продуктивність для запитів на оновлення та діапазону. Відповідь: «так». Ми можемо виконати обидві операції за O(log(n)), де n – розмір масиву. Це можна зробити за допомогою дерева відрізків.

Припустимо, що нам задано вхідний масив A {1, 8, 2, 7, 3, 6, 4, 5}. На рисунку нижче зображено дерево відрізків, сформоване відповідно до вхідного масиву A.



Початковий масив: A {1, 8, 2, 7, 3, 6, 4, 5}

Рис. 6.17. Дерево відрізків

```
class RangeMaxSegmentTree(input: IntArray) {
    private var segmentArr: IntArray
    private var n: Int = input.size // Висота дерева відрізків.
    init {
        val x: Int = ceil(log2(n.toDouble())).toInt()
        // Максимальний розмір дерева відрізків
        val maxSize: Int = 2 * 2.0.pow(x.toDouble()).toInt() - 1
        // Створення масиву для розміщення дерева відрізків
        segmentArr = IntArray(maxSize)
        buildSegmentTree(input, 0, n - 1, 0)
    }

    private fun buildSegmentTree(input: IntArray, start: Int, end: Int, index:
Int): Int {
        // Якщо це один елемент, зберегти його в поточному вузлі
        // дерева відрізків та повернутись
        if (start == end) {
            segmentArr[index] = input[start]
            return input[start]
        }
```

```
        // Якщо вже маємо більше одного елемента,
        // тоді обходимо ліве та праве піддерева
        // та зберігаємо мінімальне значення в поточному вузлі.
        val mid = (start + end) / 2
        segmentArr[index] = max(buildSegmentTree(input, start, mid, index *
2 + 1),
            buildSegmentTree(input, mid + 1, end, index * 2 + 2)
        )
        return segmentArr[index]
    }
}
```

Аналіз:

• Дерево відрізків створюється на основі масиву. Кількість вузлів, необхідних для створення (2*n – 1), якщо n є степенем 2, інакше буде 2*(степінь 2 більший за n) – 1. Це можна обчислити таким чином:

```
val x: Int = ceil(log2(n.toDouble())).toInt()
val maxSize: Int = 2 * 2.0.pow(x.toDouble()).toInt() - 1
```

• Функція buildSegmentTree використовується для заповнення різних елементів дерева відрізків.

```
fun getMax(start: Int, end: Int): Int {
    // Перевірка коректності діапазону
    if (start > end || start < 0 || end > n - 1) {
        println("Invalid Input.")
        return Int.MIN_VALUE
    }
    return getMaxUtil(0, n - 1, start, end, 0)
}

private fun getMaxUtil(segStart: Int, segEnd: Int,
                       queryStart: Int, queryEnd: Int, index:Int): Int {
    // випадок повного перекриття.
    if (queryStart <= segStart && segEnd <= queryEnd)
        return segmentArr[index]
    // випадок, коли немає перекриття.
    if (segEnd < queryStart || queryEnd < segStart)
        return Int.MIN_VALUE
    // Відрізок дерева частково перекривається з діапазоном запиту.
    val mid = (segStart + segEnd) / 2
    return max(getMaxUtil(segStart, mid, queryStart, queryEnd, 2 * index + 1),
        getMaxUtil(mid + 1, segEnd, queryStart, queryEnd, 2 * index + 2)
    )
}
```

Аналіз:

Функція getMax() використовується для виконання запиту по діапазону. Якщо діапазон вузлів segStart і segEnd знаходиться всередині діапазону запиту, то повертається максимальне значення вузла. У випадку відсутності перекриття повертається MIN_VALUE, щоб не впливати на остаточну відповідь. У випадку часткового перекриття виконуються запити до лівого і правого піддерев, а їх результати об'єднуються для отримання остаточного результату.

```
fun update(ind: Int, value: Int) {
    // Перевірка діапазону
    if (ind < 0 || ind > n - 1) {
        println("Invalid Input.")
        return
    }
    // Оновити значення у дереві відрізків
    updateUtil(0, n - 1, ind, value, 0)
}
// Завжди повертається мінімум із допустимого діапазону
fun updateUtil(segStart: Int, segEnd: Int, ind: Int, value: Int, index: Int): Int {
    // Оновлюваний index знаходиться ззовні діапазону поточного відрізка
    // У такому випадку, мінімум не зміниться
    if (ind < segStart || ind > segEnd) return segmentArr[index]
    // Якщо вхідний index у діапазоні цього вузла, тоді оновлюємо
    // значення поточного вузла і його нащадків
    if (segStart == segEnd) {
        return if (segStart == ind) { // значення index потрібно оновити
            segmentArr[index] = value
            value
        } else {
            segmentArr[index] // значення index не змінилось
        }
    }
    val mid = (segStart + segEnd) / 2
    // Значення поточного вузла оновлено (змінено) на min
    segmentArr[index] = max(updateUtil(segStart, mid, ind, value, 2*index + 1),
    updateUtil(mid + 1, segEnd, ind, value, 2 * index + 2))
    // Значення різниці просувається на батьківський вузол
    return segmentArr[index]
}
```

Аналіз:

1. Якщо індекс, який потрібно змінити, знаходиться за межами діапазону segStart і segEnd, то він не змінить відрізок. Буде повернуто мінімальне значення відрізка.

2. Якщо індекс, який потрібно змінити, знайдено, то значення цього індексу оновлюється.

3. У всіх інших випадках для модифікації бінарного дерева обходиться як ліве, так і праве піддерева.

```
// Testing code.
fun main() {
    val arr = intArrayOf(1, 8, 2, 7, 3, 6, 4, 5)
    val tree = RangeMaxSegmentTree(arr)
    println("Max value in the range(1, 5): " + tree.getMax(1, 5))
    println("Max value in the range(2, 7): " + tree.getMax(2, 7))
    println("Max value of all the elements: " + tree.getMax(0, arr.size - 1))
    tree.update(2, 9)
    println("Max value in the range(1, 5): " + tree.getMax(1, 5))
    println("Max value of all the elements: " + tree.getMax(0, arr.size - 1))
}
```

Результат:

```
Max value in the range(1, 5): 8
Max value in the range(2, 7): 7
Max value of all the elements: 8
Max value in the range(1, 5): 9
Max value of all the elements: 9
```

Аналіз:

У даному масиві максимальне значення між індексами 1 і 5 дорівнює 8. Аналогічно, максимальне значення між індексами 2 і 7 дорівнює 7. Значення при індексі 2 змінено на 9, таким чином, максимальне значення в діапазоні 1 і 5 стає рівним 9.

AVL Tree

AVL-дерево – це бінарне дерево пошуку (BST) з додатковою властивістю – воно є збалансованим деревом. Різниця між висотою лівого та правого піддерева кожного вузла відрізняється за висотою не більше ніж на одиницю.

Іншими словами, різниця між висотою лівого та правого піддерев кожного вузла дерева дорівнює –1, 0 або +1. У AVL-дереві кожен вузол зберігає додаткову інформацію, відому як висота.

```
class AVLTree(var root: Node? = null) {
    class Node(var data: Int, var left: Node?, var right: Node?) {
        var height = 0
    }

    fun height(n: Node?): Int {
        return n?.height ?: -1
    }

    fun getBalance(node: Node?): Int {
        return if (node == null) 0
        else height(node.left) - height(node.right)
    }

    fun printTree() {
        printTree(root, "", false)
        println()
    }

    private fun printTree(node: Node?, startIndent: String, isLeft: Boolean) {
        var indent = startIndent
        if (node == null) return
        if (isLeft) {
            print("${indent}L:")
            indent += "| "
        } else {
            print("${indent}R:")
            indent += " "
        }
        println("${node.data}(${node.height})")
        printTree(node.left, indent, true)
        printTree(node.right, indent, false)
    }
}
```

Додавання або видалення вузла з AVL-дерева може призвести до розбалансування AVL-дерева. Такі порушення властивості збалансованості AVL-дерева виправляються за допомогою обертань. Припустимо, що додавання нового вузла

перетворює раніше збалансоване AVL-дерево на незбалансоване. Оскільки дерево раніше було збалансованим, а до нього додано один новий вузол, то незбалансована максимальна різниця у висоті буде дорівнювати 2.

Таким чином, у найнижчому незбалансованому вузлі є лише чотири випадки:

1. Left-Left: новий вузол є лівим нащадком свого батька, який є лівим нащадком дідуся.

2. Left-Right: новий вузол є правим нащадком свого батька, який є лівим нащадком дідуся.

3. Right-Left: новий вузол є лівим нащадком свого батька, який є правим нащадком дідуся.

4. Right-Right: новий вузол є правим нащадком свого батька, який є правим нащадком дідуся.

Обертання ліворуч для виправлення Right-Right

Обертання ліворуч виконується обертанням вузлів проти годинникової стрілки.

Розглянемо наступну вставку, щоб зрозуміти, що таке обертання ліворуч. Вставимо в дерево 1, 2 та 3.

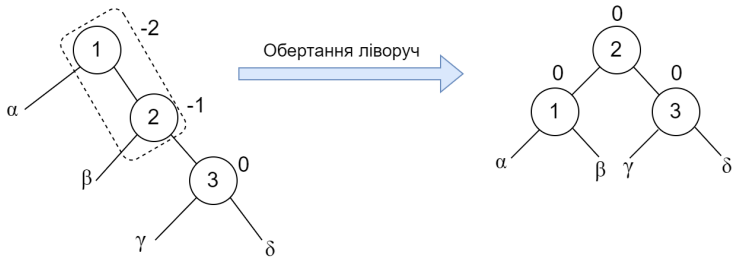


Рис. 6.18. Обертання ліворуч

```
fun leftRotate(x: Node?): Node {
    val y = x!!.right
    val t = y!!.left
    // Обертання
```

```
    y.left = x
    x.right = t
    // Оновити висоти
    x.height = max(height(x.left), height(x.right)) + 1
    y.height = max(height(y.left), height(y.right)) + 1
    // Повернути новий корінь
    return y
}
```

Обертання праворуч для виправлення Left-Left

Обертання праворуч виконується обертанням вузлів за годинниковою стрілкою.

Розглянемо наступну вставку, щоб зрозуміти, що таке обертання праворуч. Вставимо в дерево 3, 2 та 1.

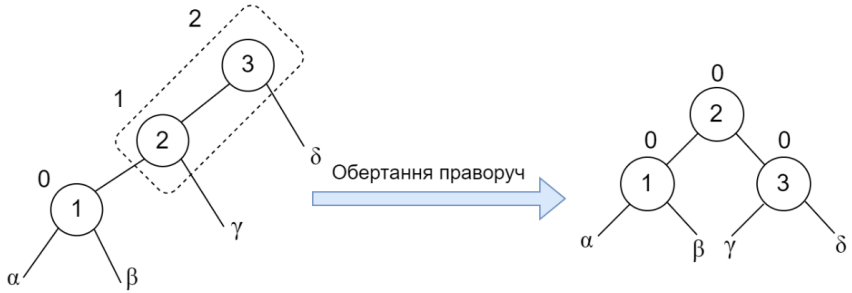


Рис. 6.19. Обертання праворуч

```
fun rightRotate(x: Node?): Node {
    val y = x!!.left
    val t = y!!.right
    // Обертання
    y.right = x
    x.left = t
    // Оновити висоти
    x.height = max(height(x.left), height(x.right)) + 1
    y.height = max(height(y.left), height(y.right)) + 1
    // Повернути новий корінь
    return y
}
```

Обертання ліворуч-праворуч для виправлення Left-Right

Обертання ліворуч-праворуч – це обертання ліворуч, за яким слідує обертання праворуч. У цьому обертанні два нижні вузли спочатку виконують поворот ліворуч проти годинникової стрілки, а потім поворот праворуч за годинниковою стрілкою для двох верхніх вузлів.

Розглянемо наступну вставку, щоб зрозуміти таке обертання. Вставка 3, 1 та 2.

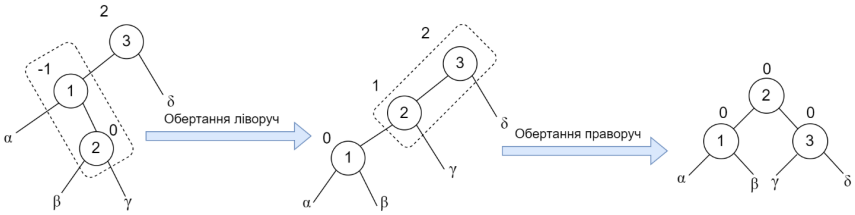


Рис. 6.20. Обертання ліворуч-праворуч

```
fun leftRightRotate(x: Node): Node {
    x.left = leftRotate(x.left)
    return rightRotate(x)
}
```

Обертання праворуч-ліворуч для виправлення Right-Left

Обертання праворуч-ліворуч – це обертання праворуч, за яким слідує обертання ліворуч. У цьому обертанні два нижні вузли спочатку виконують поворот праворуч за годинниковою стрілкою, а потім поворот ліворуч проти годинникової стрілки для двох верхніх вузлів.

Розглянемо наступну вставку, щоб зрозуміти таке обертання. Вставка 1, 3 та 2.

```
fun rightLeftRotate(x: Node): Node {
    x.right = rightRotate(x.right)
    return leftRotate(x)
}
```

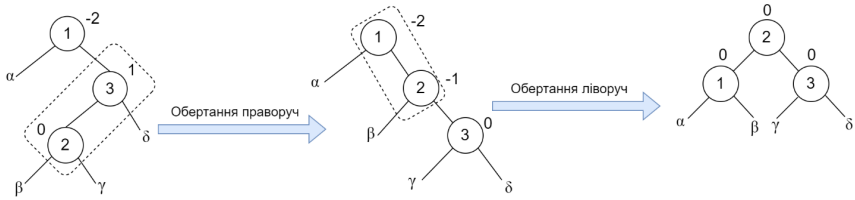


Рис. 6.21. Обертання праворуч-ліворуч

Операції на AVL-дереві

AVL-дерево – це бінарне дерево пошуку. Тому всі операції обходу та пошуку бінарних дерев пошуку застосовні до AVL-дерева; єдина відмінність полягає в операціях insert() та delete(), які можуть зробити дерево незбалансованим.

Вставка в AVL-дерево

У AVL-дереві операція вставки виконується таким чином:

1. Вставити новий елемент у дерево за допомогою стандартного алгоритму вставки у Binary Search Tree.

2. Після вставки перевірити коефіцієнт збалансованості кожного вузла.

3. Якщо коефіцієнт балансу кожного вузла дорівнює 0, 1 або –1, то все готово, інакше переходимо до наступного кроку.

4. Якщо коефіцієнт збалансованості будь-якого вузла відмінний від 0, 1 або −1, то дерево є незбалансованим. У цьому випадку виконайте відповідне обертання, щоб збалансувати дерево. Існує чотири випадки (LL-випадок, LR-випадок, RL і RR), які потрібно врахувати, щоб зробити дерево збалансованим.

У AVL-дереві операція вставки має O(log(n)) часової складності.

```
fun insert(data: Int) {
    root = insert(root, data)
}
private fun insert(node: Node?, data: Int): Node {
    if (node == null) return Node(data, null, null)
```

```
    if (node.data > data) {
        node.left = insert(node.left, data)
    } else if (node.data < data) {
        node.right = insert(node.right, data)
    } else { // Дублювання даних не дозволено
        return node
    }
    node.height = max(height(node.left), height(node.right)) + 1
    val balance = getBalance(node)
    if (balance > 1) {
        if (data < node.left!!.data) // випадок Left Left
        {
            return rightRotate(node)
        }
        if (data > node.left!!.data) // випадок Left Right
        {
            return leftRightRotate(node)
        }
    }
    if (balance < -1) {
        if (data > node.right!!.data) // випадок Right Right
        {
            return leftRotate(node)
        }
        if (data < node.right!!.data) // випадок Right Left
        {
            return rightLeftRotate(node)
        }
    }
    return node
}
```

Видалення з AVL-дерева

У AVL-дереві операція видалення виконується наступним чином:

1. Видалення елемента з дерева виконується за допомогою стандартного алгоритму видалення з Binary Search Tree.

2. Після видалення треба пройтись угору і перевірити коефіцієнт балансу кожного вузла. Якщо знайдено якийсь незбалансований вузол, переходимо до наступного кроку.

3. Якщо коефіцієнт збалансованості будь-якого вузла відмінний від 0 або 1 або –1, то це дерево є незбалансованим.

У цьому випадку треба виконати відповідне обертання, щоб зробити його збалансованим. Існує чотири випадки (LL-випадок, LR-випадок, RL та RR), які потрібно врахувати, щоб зробити дерево збалансованим.

```
fun delete(data: Int) {
    root = delete(root, data)
}
private fun delete(node: Node?, data: Int): Node? {
    if (node == null) return null
    if (node.data == data) {
        if (node.left == null && node.right == null) {
            return null
        } else if (node.left == null) {
            return node.right // no need to modify height
        } else if (node.right == null) {
            return node.left // no need to modify height
        } else {
            val minNode = findMin(node.right)
            node.data = minNode!!.data
            node.right = delete(node.right, minNode.data)
        }
    } else {
        if (node.data > data) {
            node.left = delete(node.left, data)
        } else {
            node.right = delete(node.right, data)
        }
    }
    node.height = max(height(node.left), height(node.right)) + 1
    val balance = getBalance(node)
    if (balance > 1) {
        if (data >= node.left!!.data) { // Left Left Case
            return rightRotate(node)
        }
        if (data < node.left!!.data) { // Left Right Case
            return leftRightRotate(node)
        }
    }
    if (balance < -1) {
        if (data <= node.right!!.data) // Right Right Case
        {
            return leftRotate(node)
        }
        if (data > node.right!!.data) // Right Left Case
        {
            return rightLeftRotate(node)
```

```
        }
    }
    return node
}

private fun findMin(curr: Node?): Node? {
    var node: Node? = curr ?: return null
    while (node!!.left != null) {
        node = node.left
    }
    return node
}
```

Часова складність видалення буде O(log(n)).

Розглянемо виконання такого коду:

```
fun main() {
    val t = AVLTree()
    t.insert(1)
    t.insert(2)
    t.insert(3)
    t.insert(4)
    t.insert(5)
    t.insert(6)
    t.insert(7)
    t.insert(8)
    t.printTree()
    t.delete(5)
    t.printTree()
}
```

Результат:

```
R:4 (3)
 L:2 (1)
 | L:1 (0)
 | R:3 (0)
 R:6 (2)
 L:5 (0)
 R:7 (1)
 R:8 (0)

R:4 (2)
 L:2 (1)
 | L:1 (0)
 | R:3 (0)
 R:7 (1)
 L:6 (0)
 R:8 (0)
```

Червоно-чорне дерево (RB-Tree)

Червоно-чорне дерево містить дані, лівий та правий дочірні елементи, як і будь-яке інше бінарне дерево. На додаток до цього, його вершина також містить додатковий біт інформації, що представляє колір, який може бути або червоним, або чорним. Червоно-чорне дерево також містить спеціальний тип вузлів, які називаються *нульовими.* Нульові вузли – це псевдовузли, які існують на листі дерева і мають чорний колір. Усі внутрішні вузли мають свої дані, пов'язані з ними.

Червоно-чорне дерево має наступні властивості:

- Корінь дерева – чорний.
- Кожен листовий вузол (нульовий вузол) – чорний.
- Дві червоні вершини не можуть мати відношення батько–нащадок.
- Кожен шлях від вершини до листка-нащадка містить однакову кількість чорних вершин.

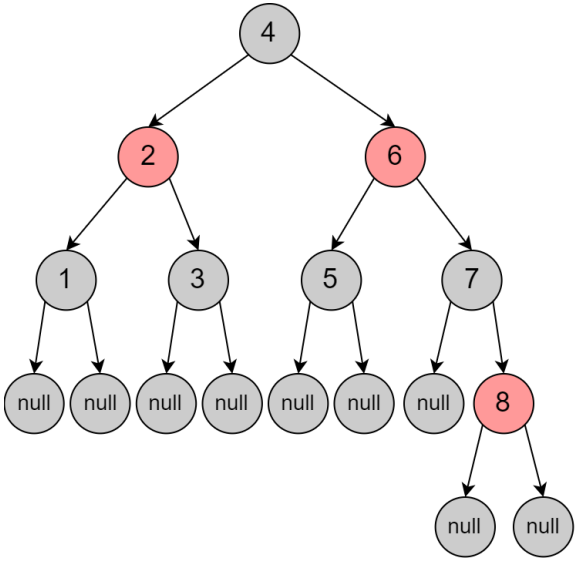


Рис. 6.22. Червоно-чорне дерево

Перші три властивості не потребують пояснень. Четверта властивість стверджує, що від будь-якої вершини дерева до будь-якого листка (нульової вершини) кількість чорних вершин має бути однаковою. На наведеному вище рисунку, від кореневого вузла до листка (нульового вузла) кількість чорних вузлів завжди дорівнює трьом вузлам.

Як і AVL-дерево, червоно-чорні дерева також є самобалансуючими бінарними деревами пошуку. Проте відмінність полягає в тому, що властивість збалансованості AVL-дерева має пряму залежність між висотами лівого та правого піддерев кожного вузла, а в червоно-чорних деревах властивість балансування регулюється чотирма правилами, наведеними вище. Додавання або видалення вузла з червоно-чорного дерева може порушити властивості червоно-чорного дерева. Властивості червоно-чорного дерева відновлюються за допомогою перефарбовування та обертання. Операції вставки, видалення та пошуку мають *часову складність* O(log(n)).

Вставка в RB-Tree

У червоно-чорному дереві кожен новий вузол має бути позначений червоним кольором. Операція вставки у червоно-чорному дереві подібна до операції вставки у BST. Після кожної операції вставки потрібно перевірити всі властивості червоно-чорного дерева. Якщо всі властивості не задовольняються, ми виконуємо обертання і перефарбовування, щоб відновити властивості червоно-чорного дерева. Новий вузол, що вставляється, завжди є червоним, тому властивість, яка найчастіше порушується, – це два червоних підряд.

Операція вставки в червоно-чорному дереві виконується за допомогою наступних кроків:

КРОК 1: додається новий вузол відповідно до вставки BST з червоним кольором нового вузла.

КРОК 2: якщо новий вузол є кореневим вузлом, то пофарбувати його в чорний колір і вийти.

КРОК 3: якщо батьком нової вершини є чорний, то вийти.

КРОК 4: якщо батько нового вузла має червоний колір і дядько (рідний брат батька) має червоний колір, то виконати перефарбування. Зробити дідуся (батька батька) червоними, а батька і дядька – чорними.

КРОК 5: якщо батько нового вузла має червоний колір, а дядько – чорний або нульовий, то зробити відповідні обертання і перефарбувати його.

На рисунку нижче показано КРОК 4, перефарбування у випадку, коли батько і дядько нової вершини мають червоний колір.

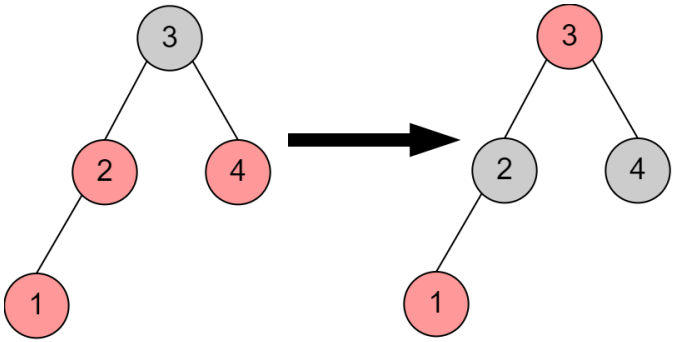


Рис. 6.23. Перефарбування у RB-Tree

Обертання ліворуч для виправлення випадку RR

Обертання ліворуч здійснюється шляхом обертання вузлів у напрямку проти годинникової стрілки.

```
fun leftRotate(x: Node?): Node {
    val y = x!!.right
    val z = y!!.left
    // обертання
    y.parent = x.parent
    y.left = x
    x.parent = y
    x.right = z
    if (z !== nullNode) z!!.parent = x
```

```
    if (x === root) {
        root = y
        return y
    }
    if (y.parent!!.left === x) y.parent!!.left = y
    else y.parent!!.right = y
    // повертаємо новий корінь
    return y
}
```

Обертання праворуч для виправлення випадку LL

Обертання праворуч здійснюється шляхом обертання вузлів у напрямку годинникової стрілки.

```
fun rightRotate(x: Node?): Node {
    val y = x!!.left
    val z = y!!.right
    // обертання
    y.parent = x.parent
    y.right = x
    x.parent = y
    x.left = z
    if (z !== nullNode) z!!.parent = x
    if (x === root) {
        root = y
        return y
    }
    if (y.parent!!.left === x) y.parent!!.left = y
    else y.parent!!.right = y
    // повертаємо новий корінь
    return y
}
```

Обертання ліворуч-праворуч для виправлення випадку LR

Обертання ліворуч-праворуч – це обертання ліворуч, за яким слідує обертання праворуч. У цьому обертанні два нижні вузли спочатку виконують обертання проти годинникової стрілки ліворуч, а потім обертання за годинниковою стрілкою праворуч для двох верхніх вузлів.

```
fun leftRightRotate(node: Node?): Node {
    node!!.left = leftRotate(node.left)
    return rightRotate(node)
}
```

Обертання праворуч-ліворуч для виправлення випадку RL

Обертання праворуч-ліворуч – це обертання праворуч, за яким слідує обертання ліворуч. У цьому обертанні два нижні вузли спочатку виконують обертання за годинниковою стрілкою праворуч, а потім обертання проти годинникової стрілки ліворуч для двох верхніх вузлів.

```
fun rightLeftRotate(node: Node?): Node {
    node!!.right = rightRotate(node.right)
    return leftRotate(node)
}
```

Нижче наведено рисунок, що демонструє КРОК 5, де виконується обертання з подальшим перефарбуванням.

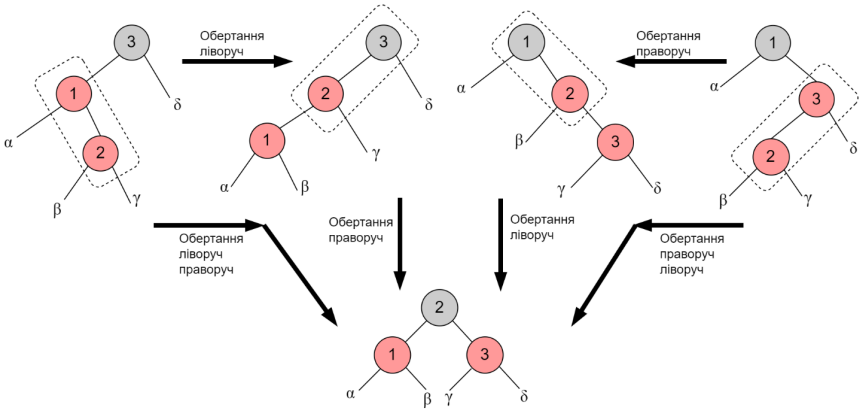


Рис. 6.24. Перетворення у RB-Tree для корекції умови двох червоних поспіль

```
fun insert(data: Int) {
    root = insert(root, data)
    val temp = find(data)
    fixRedRed(temp)
}
```

```
private fun find(data: Int): Node? {
    var curr = root
    while (curr != null) {
        curr = if (curr.data == data) {
            return curr
        } else if (curr.data > data) {
            curr.left
        } else {
            curr.right
        }
    }
    return null
}

private fun insert(nd: Node?, data: Int): Node {
    var node = nd
    if (node === nullNode) {
        node = Node(data, nullNode)
    } else if (node!!.data > data) {
        node.left = insert(node.left, data)
        node.left!!.parent = node
    } else if (node.data < data) {
        node.right = insert(node.right, data)
        node.right!!.parent = node
    }
    return node
}

fun fixRedRed(x: Node?) {
    // якщо x - корінь, фарбуємо його у чорний і виходимо
    if (x === root) {
        x!!.color = false
        return
    }
    if (x!!.parent === nullNode || x!!.parent!!.parent === nullNode) {
        return
    }
    // Визначаємо батька, дідуся, дядю
    val parent = x!!.parent
    val grandparent = parent!!.parent
    val uncle = uncle(x)
    var mid: Node? = null
    if (!parent.color) {
        return
    }
    // колір батька  - червоний, колір дідуся - чорний.
    if (uncle !== nullNode && uncle!!.color) {
```

```
        // пофарбувати батька і дядю у червоний.
        parent.color = false
        uncle.color = false
        grandparent!!.color = true
        fixRedRed(grandparent)
        return
    }
    // колір батька - червоний, дяді і дідуся - чорний.
    // Виконати LR, LL, RL, RR
    if (parent === grandparent!!.left && x === parent.left) { // LL
        mid = rightRotate(grandparent)
    } else if (parent === grandparent!!.left && x === parent.right) { // LR
        mid = leftRightRotate(grandparent)
    } else if (parent === grandparent!!.right && x === parent.left) { // RL
        mid = rightLeftRotate(grandparent)
    } else if (parent === grandparent!!.right && x === parent.right) { // RR
        mid = leftRotate(grandparent)
    }
    mid!!.color = false
    mid.left!!.color = true
    mid.right!!.color = true
}

private fun uncle(node: Node?): Node? {
    // Якщо немає батька, або дідуся, то немає і дяді
         if (node!!.parent === nullNode || node!!.parent!!.parent ===
nullNode) return null
         return if (node!!.parent === node!!.parent!!.parent!!.left) //
дядя праворуч
        node!!.parent!!.parent!!.right else // дядя ліворуч
        node!!.parent!!.parent!!.left
}
```

Видалення у RB-Tree

У RB-Tree операція видалення виконується наступним чином:

1. Видалити елемент з дерева за допомогою стандартного видалення з BST. Під час видалення з BST завжди видаляється листова вершина або така, що містить одну дочірню вершину. Якщо у вузла два дочірні вузли, то до нього копіюються дані наступника під час обходу в інфіксному порядку, і далі рекурсивно викликається його видалення.

2. Якщо видаляється вершина червоного кольору, то властивість Red-Black не порушується.

3. Якщо видалена вершина чорна. Тоді видалені вершини можуть призвести до зменшення кількості чорних вершин на одному із шляхів від кореня до листа.

4. Для виправлення порушених властивостей RB-Tree виконується відповідне обертання та перефарбування.

Кроки для виправлення RB-дерева після видалення чорного вузла:

1. Назвемо видалену вершину «y». Нащадка цієї вершини – «x». Рідного брата видаленої вершини – «s».

2. Випадок 1: якщо нащадок видаленої вершини має червоний колір, то треба зробити колір дочірньої вершини чорним і кількість чорних вершин буде відновлена.

3. Випадок 2: коли брат видаленого вузла чорного кольору і принаймні один з дочірніх вузлів чорного кольору червоний. У цьому випадку можливі чотири різні комбінації:

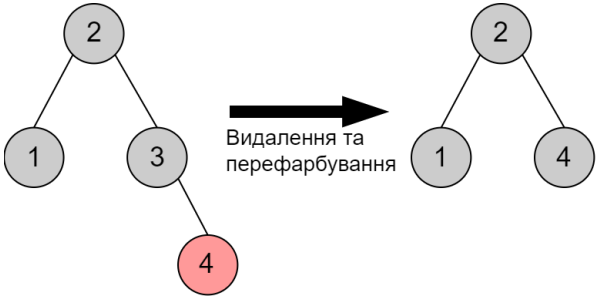


Рис. 6.25. Випадок 1: колір дочірнього вузла – червоний

• Випадок 2(*a*): коли брат «s» видаленої вершини «y» має чорний колір. Рідний брат «s» є лівим нащадком батька, а лівий нащадок його – червоний. Це конфігурація «Left-Left», тому треба виконати обертання вправо. Потім змінити колір нащадка на чорний. Цей випадок показано на рис. 6.26.

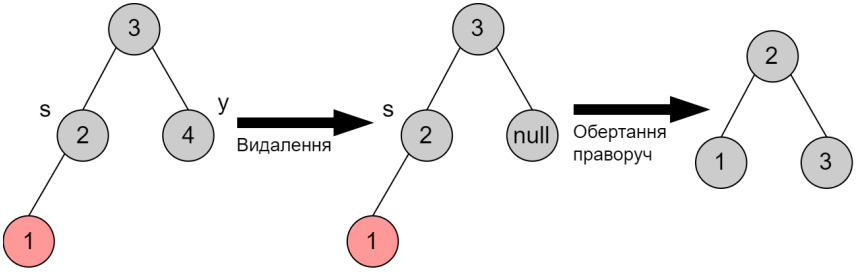


Рис. 6.26. Випадок 2(*a*)

• Випадок 2(*b*): дзеркальне відображення попереднього випадку. Рідний брат «s» є правою дитиною свого батька і його правий нащадок – червоний. Конфігурація «Right-Right», тому її виправить ліве обертання. Після якої змінюється колір дочірнього елемента на чорний. Властивість кількості чорних вершин буде відновлено.

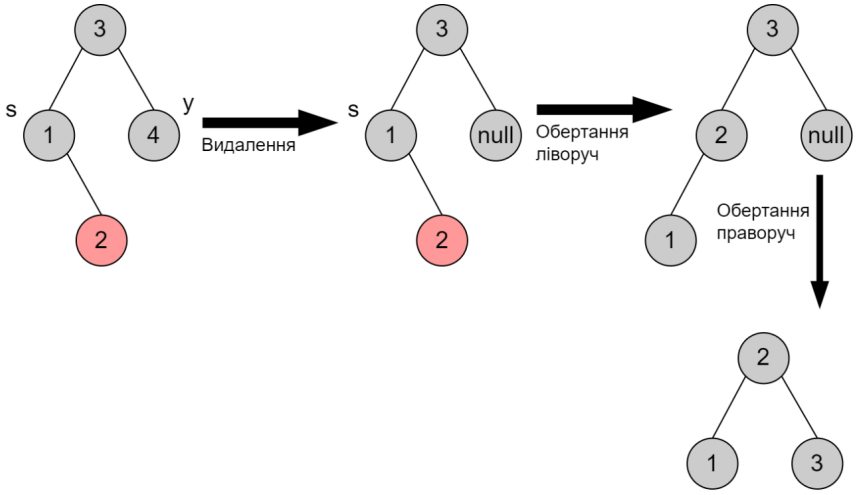


Рис. 6.27. Випадок 2(*c*)

• Випадок 2(*c*): коли брат «s» видаленої вершини «y» має чорний колір. Брат «s» є лівим нащадком свого батька, а його правий нащадок – червоний. Це конфігурація «Left-Right», тому виконується обертання вліво, а потім вправо. Дочірній вузол зафарбовано чорним кольором.

• Випадок 2(*d*): коли брат «s» видаленої вершини «y» – чорний. Брат «s» є правим нащадком батька, а його лівий нащадок – червоний. Це конфігурація «Right-Left», тому виконується обертання праворуч, а потім обертання ліворуч. Колір дочірньої вершини змінюється на чорний.

4. Випадок 3: коли брат видаленої вершини є чорним, а обидва її дочірні вузли є чорними або нульовими (який також чорний). Треба пофарбувати нащадків червоним і рекурсивно додати чорний колір до батька. Якщо батько був червоним, то він стане чорним. Якщо батько був чорним, то він стане подвійно чорним. Якщо батько є кореневою вершиною, то ми робимо кореневу вершину чорною і виходимо.

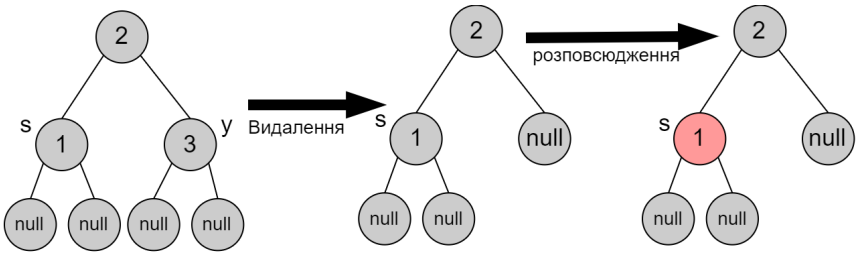


Рис. 6.28. Випадок 3

5. Випадок 4: коли нащадок «s» видаленого вузла «y» є червоним. Обертання виконується для того, щоб підняти нащадка вгору. Потім перефарбовується брат батька.

• Брат «s» є лівим нащадком батька. Це конфігурація «Left-Left», тому треба виконати обертання праворуч.

• Нащадок «s» є правою дитиною батька. Це конфігурація «Right-Right», тому виконайте обертання ліворуч.

• Коли буде виконано обертання, кольори батьків і нащадків поміняються місцями. Після обертання буде отримано один з наступних варіантів: Випадок 2(*a*), 2(*b*), 2(*c*), 2(*d*) або випадок 3.

• На рисунку 6.28 брат є правим нащадком батька, тому виконується обертання ліворуч. Кольори батьків і нащадків міняються місцями. Після обертання і перефарбовування застосовується випадок 3.

```
fun delete(data: Int) {
    delete(root, data)
}

private fun delete(nd: Node?, key: Int) {
    var node = nd
    var z: Node? = nullNode
    val x: Node?
    var y: Node?
    while (node !== nullNode) {
        if (node!!.data == key) {
            z = node
            break
        } else if (node.data <= key) {
            node = node.right
        } else {
            node = node.left
        }
    }
    if (z === nullNode) {
        println(«Значення не знайдено у дереві»)
        return
    }
    y = z
    var yColour = y!!.color
    if (z!!.left === nullNode) {
        x = z!!.right
        joinParentChild(z, z.right)
    } else if (z!!.right === nullNode) {
        x = z!!.left
        joinParentChild(z, z.left)
    } else {
        y = minimum(z!!.right)
        yColour = y!!.color
        z.data = y.data
        joinParentChild(y, y.right)
```

```
            x = y.right
        }
        if (!yColour) {
            if (x!!.color) {
                x.color = false
                return
            } else {
                fixDoubleBlack(x)
            }
        }
    }
    private fun fixDoubleBlack(x: Node?) {
        if (x === root) // корінь
            return
        val sib = sibling(x)
        val parent = x!!.parent
        if (sib === nullNode) {
            // немає брата «двічі чорний» вузол зсувається до батька
            fixDoubleBlack(parent)
        } else {
            if (sib!!.color) {
                // Колір брата – червоний
                parent!!.color = true
                sib.color = false
                if (sib.parent!!.left === sib) {
                    // брат – лівий нащадок
                    rightRotate(parent)
                } else {
                    // брат – правий нащадок
                    leftRotate(parent)
                }
                fixDoubleBlack(x)
            } else {
                // Колір брата – чорний
                // Як мінімум, один нащадок – червоний
                if (sib.left!!.color || sib.right!!.color) {
                    if (sib.parent!!.left === sib) {
                        // брат – лівий нащадок
                        if (sib.left !== nullNode && sib.left!!.color) {
                            // випадок left-left
                            sib.left!!.color = sib.color
                            sib.color = parent!!.color
                            rightRotate(parent)
                        } else {
                            // випадок left-right
                            sib.right!!.color = parent!!.color
                            leftRotate(sib)
```

```
                        rightRotate(parent)
                    }
                } else {
                    // брат – правий нащадок
                    if (sib.left !== nullNode && sib.left!!.color) {
                        // випадок right-left
                        sib.left!!.color = parent!!.color
                        rightRotate(sib)
                        leftRotate(parent)
                    } else {
                        // випадок right-right
                        sib.right!!.color = sib.color
                        sib.color = parent!!.color
                        leftRotate(parent)
                    }
                }
                parent.color = false
            } else {
                // Обидва нащадки – чорні
                sib.color = true
                if (!parent!!.color) fixDoubleBlack(parent) else
                    parent.color = false
            }
        }
    }
}
private fun sibling(node: Node?): Node? {
    // якщо немає батька, то немає і брата
    if (node!!.parent === nullNode) return null
    if (node!!.parent!!.left === node)
        return node!!.parent!!.right
    else
        return node!!.parent!!.left
}
private fun joinParentChild(u: Node?, v: Node?) {
    if (u!!.parent === nullNode) {
        root = v
    } else if (u === u!!.parent!!.left) {
        u!!.parent!!.left = v
    } else {
        u!!.parent!!.right = v
    }
    v!!.parent = u!!.parent
}

private fun minimum(nd: Node?): Node? {
    var node = nd
```

```
    while (node!!.left !== nullNode) {
        node = node!!.left
    }
    return node
}

// Testing code.
fun main() {
    val tree = RBTree()
    tree.insert(1)
    tree.insert(2)
    tree.insert(3)
    tree.insert(4)
    tree.insert(5)
    tree.insert(7)
    tree.insert(6)
    tree.insert(8)
    tree.insert(9)
    tree.printTree()
    tree.delete(4)
    tree.printTree()
}
```

Результат:

```
R:4(false)
 L:2(true)
 | L:1(false)
 | R:3(false)
 R:6(true)
 L:5(false)
 R:8(false)
 L:7(true)
 R:9(true)

R:5(false)
 L:2(true)
 | L:1(false)
 | R:3(false)
 R:7(true)
 L:6(false)
 R:8(false)
 R:9(true)
```

Тема 7. **Пріоритетні черги**

Пріоритетна черга також відома як купа, є різновидом черги. Об'єкти видаляються з початку черги. Однак у пріоритетній черзі логічний порядок об'єктів визначається їхнім пріоритетом. Елемент з найвищим пріоритетом знаходиться на початку черги. Коли ви додаєте об'єкт до пріоритетної черги, новий об'єкт переміщується на позицію відповідну до його пріоритету. Пріоритетна черга є дуже важливою структурою даних. Пріоритетна черга використовується в різних алгоритмах на графах, таких як алгоритм Прима та алгоритм Дейкстри. Пріоритетна черга також використовується в реалізації таймерів тощо.

Пріоритетна черга реалізується за допомогою купи (Heap). Структура даних Heap – це масив елементів, який можна розглядати як повне бінарне дерево.

Купою називається бінарне дерево, яке задовольняє наступним властивостям:

1. Дерево є повним бінарним деревом. Купа є повним бінарним деревом, тому висота дерева з N вершинами завжди дорівнює O(log(n)).

2. Купа задовольняє властивості впорядкованості купи. У Max-Heap значення батька більше або дорівнює значенню дочірнього елемента. У Min-Heap значення батька менше або дорівнює значенню дочірніх елементів.

Купа не є відсортованою структурою даних і може розглядатися як частково впорядкована. Як ви можете бачити на рисунку, між вузлами на будь-якому рівні немає зв'язку, навіть між «братами».

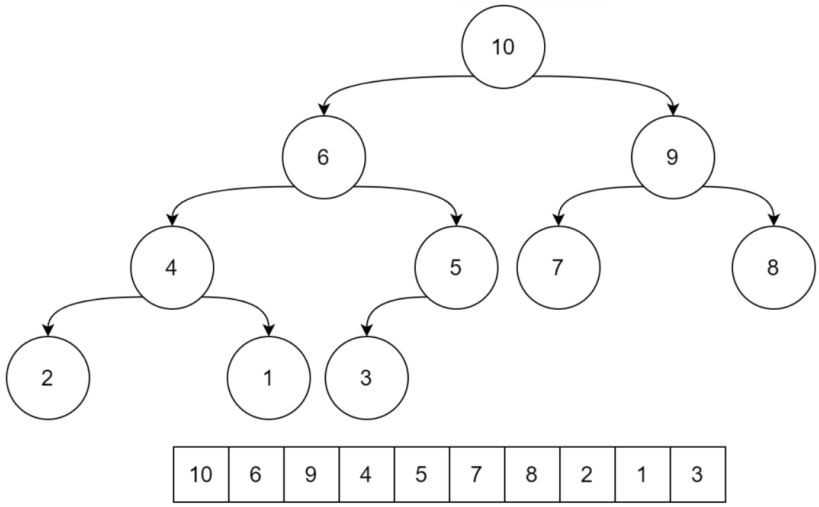


Рис. 7.1. Max-Heap

Купу реалізовано за допомогою масиву. Більше того, оскільки купа є повним бінарним деревом, лівим нащадком батька (у позиції n) є вузол, який знаходиться у позиції (2n + 1) у масиві. Аналогічно, правий нащадок батька знаходиться на позиції (2n+2) у масиві. Щоб знайти батька будь-якого вузла в купі, ми можемо просто поділити. При заданому індексі k вершини, індекс батька буде (k – 1) / 2.

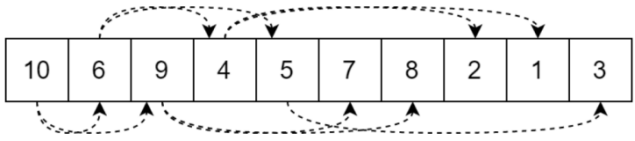


Рис. 7.2. Відношення батько–дитина у масиві купи

Типи купи

Існує два типи купи, і тип залежить від впорядкування елементів. Упорядкування може бути виконано двома способами: Min-Heap та Max-Heap.

Max-Heap

Max-Heap: значення кожного вузла менше або дорівнює значенню його батька, з елементом з найбільшим значенням у корені (рис. 7.1).

Min-Heap

Min-Heap: значення кожного вузла більше або дорівнює значенню його батька, з елементом з найбільшим значенням у корені.

Min-Heap варто використовувати, коли потрібен швидкий доступ до найменшого елемента, оскільки цей елемент завжди буде в корні дерева або першим елементом масиву. Однак решта масиву залишається частково відсортованою. Таким чином, миттєвий доступ можливий лише до найменшого елемента.

У наступній таблиці наведено порівняння часової складності виконання основних операцій у Max-Heap та Min-Heap.

Таблиця 7.1. **Часові складності операцій у купі**

Max-Heap	
Вставка	O(log(n))
Видалення максимального	O(log(n))
Видалення елемента	O(log(n))
Пошук максимального	O(1)
Min-Heap	
Вставка	O(log(n))
Видалення мінімального	O(log(n))
Видалення елемента	O(log(n))
Пошук мінімального	O(1)

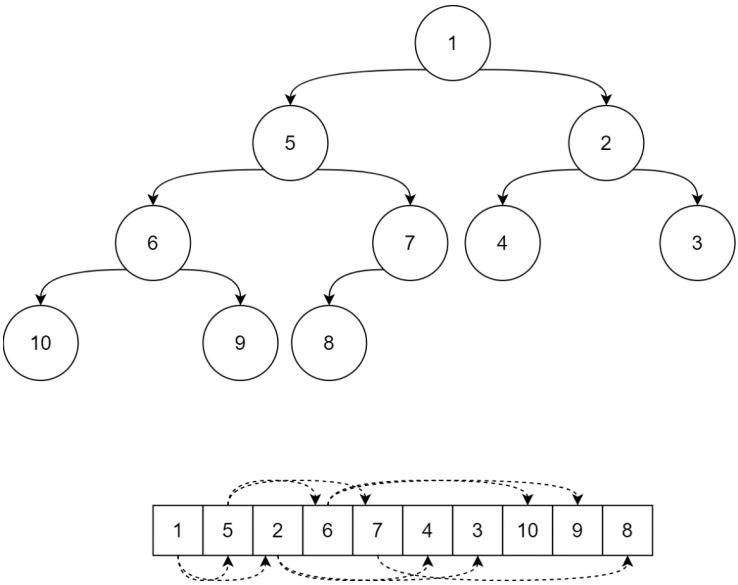


Рис. 7.3. Min-Heap

Операції з абстрактною структурою даних «Купа» (Heap)

Розглянемо основні операції з абстрактною структурою даних «Купа».

Таблиця 7.2. **Операції з абстрактною структурою даних «Купа»**

Binary Heap	Створити нову порожню бінарну купу	O(1)
Insert	Додати новий елемент у купу	O(log(n))
DeleteMax	Видалити максимальний елемент з купи	O(log(n))
findMax	Знайти максимальний елемент у купі	O(1)
isEmpty	Перевірка, чи містить купа елементи	O(1)
Size	Визначити кількість елементів у купі	O(1)
BuildHeap	Створити нову купу з елементів масиву	O(log(n))

Операції з купою

Перш ніж детально розглянути, як ініціалізувати купу, додавати або видаляти елементи до неї, потрібно зрозуміти дві важливі операції, пов'язані з купою. Ці операції використовуються для відновлення властивості впорядкованості купи, коли окремий елемент знаходиться не у своїй позиції.

Є два випадки:

1. Коли батьківський вузол не дотримується властивості купи зі своїми дочірніми вузлами. Це вирішується операцією percolateDown(), у якій значення батьківського вузла міняється місцями зі значенням одного з дочірніх і рекурсивно відновлюється властивість купи.

2. Коли дочірній вузол не наслідує властивість купи у свого батька. Це вирішується операцією percolateUp(), у якій значення дочірнього вузла міняється місцями з батьківським і рекурсивно відновлюється властивість купи.

Представимо купу у вигляді класу. Нижче наведено елементи класу купи:

1) «arr» використовується для зберігання купи;

2) «size» – кількість елементів у купі;

3) «compare» використовується для створення Min-Heap або Max-Heap.

```
class Heap(isMin: Boolean) {
    private var size: Int = 0 // Кількість елементів у купі
    private var arr: IntArray = IntArray(32) // The Heap array – Початковий
розмір 32
    private var isMinHeap: Boolean = isMin
    fun compare(arr: IntArray, first: Int, second: Int): Boolean {
        return if (isMinHeap) arr[first] - arr[second] > 0 // Порівняння в Min-Heap
        else arr[first] - arr[second] < 0 // Порівняння в Max-Heap
    }

    fun isEmpty() = size == 0

    fun length() = size

    fun peek(): Int {
        if (isEmpty()) {
```

```
            throw IllegalStateException()
        }
        return arr[0]
    }

    private fun percolateDown(parent: Int) {
        val lChild = 2 * parent + 1
        val rChild = lChild + 1
        var child = -1
        if (lChild < size) {
            child = lChild
        }
        if (rChild < size && compare(arr, lChild, rChild)) {
            child = rChild
        }
        if (child != -1 && compare(arr, parent, child)) {
            val temp = arr[parent]
            arr[parent] = arr[child]
            arr[child] = temp
            percolateDown(child)
        }
    }

    private fun percolateUp(child: Int) {
        val parent = (child - 1) / 2
        if (parent >= 0 && compare(arr, parent, child)) {
            val temp = arr[child]
            arr[child] = arr[parent]
            arr[parent] = temp
            percolateUp(parent)
        }
    }
}
```

Аналіз: порожня купа створюється шляхом створення порожнього масиву. Оскільки купа порожня, не потрібно виконувати жодних операцій над нею.

1) isEmpty() повертає true, якщо купа порожня, інакше повертає false;

2) size() повертає кількість елементів у купі;

3) peek() повертає найбільш пріоритетний елемент з купи, не видаляючи його.

Створення купи з масиву

Heapify – важливий алгоритм для перетворення масиву в купу.

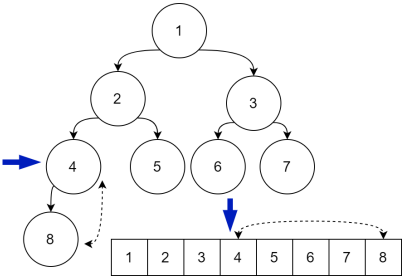
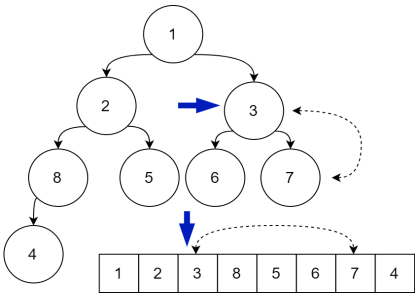
Цей алгоритм можна представити у такі кроки:

1. Значення присутні в масиві.

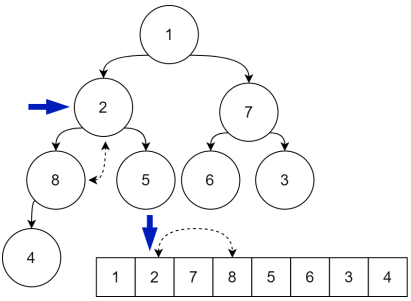
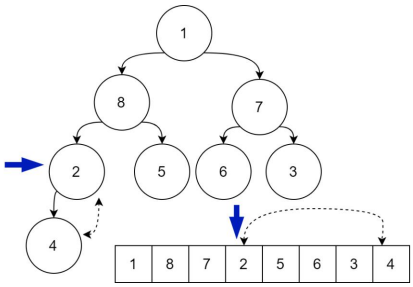
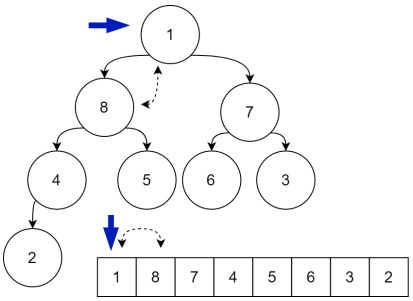
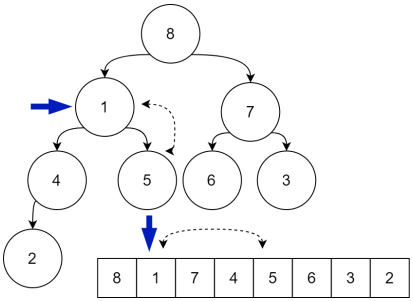
2. Починаючи з середини масиву, рухайтесь вниз до початку масиву. На кожному кроці порівняти батьківське значення з його лівим нащадком та правим нащадком. Крім того, відновлювати властивість купи обміном батьківського значення з його найбільшим дочірнім значенням. Таким чином, батьківське значення завжди буде більшим або рівним для лівого та правого дочірніх елементів.

3. Для всіх елементів від середини масиву до початку масиву виконуємо порівняння і робимо обміни, поки не дійдемо до листових вузлів купи.

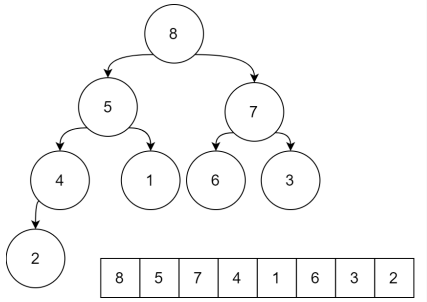
Таблиця 7.3

	На вході задано масив, який потрібно перетворити в купу. Значення індексу порівнюється зі значенням його дочірніх вузлів з індексами (i*2+1) та (i*2+2). Середина списку N/2, тобто індекс 3, порівнюється з індексом 7. Якщо значення дочірньої вершини більше за значення батьківської, то вони міняються місцями
	Аналогічно, значення з індексом 2 порівнюється зі значеннями з індексами 5 і 6. Найбільше з усіх значень дорівнює 7 і буде поміняно місцями зі значенням по індексу 2

Продовж. табл. 7.3

	Далі, значення з індексом 1 порівнюється зі значеннями з індексами 3 та 4. Найбільшим з усіх значень є 8, яке буде поміняно місцями зі значенням за індексом 1
	Функція percolateDown() використовується для подальшого коригування значення, заміненого на попередньому кроці, шляхом порівняння його з дочірніми вузлами
	Тепер значення по індексу 0 порівнюється зі значеннями індексами 1 і 2. 8 є найбільшим значенням, тому воно міняється місцями зі значенням по індексу 0
	Функція percolateDown() використовується для подальшого порівняння значення з індексом 1 з його дочірніми вузлами з індексами 3 і 4

Продовж. табл. 7.3

	Урешті-решт, створюється Max-Heap

```
constructor(array: IntArray, isMin: Boolean) : this(isMin) {
    size = array.size
    arr = array
    isMinHeap = isMin
    // Створити купу з масиву
    for (i in size / 2 downTo 0) {
        percolateDown(i)
    }
}
```

Аналіз: купу можна ініціалізувати, передавши їй масив. Над масивом викликається функція Heapify(), яка робить його купою.

Часова складність побудови купи складає O(N).

Кількість порівнянь залежить від висоти масиву. Загальна кількість порівнянь буде представлена як:

$$(1*n/4) + (2*n/8) + (3*n/16)\ldots\ldots((h-1)*1) = N,$$

де «h» – висота купи.

Додавання – Enqueue

1. Новий елемент додається в кінець масиву. Це збереже всі значення у вигляді повного бінарного дерева, але воно може більше не бути купою, оскільки новий елемент може мати значення більше ніж значення його батька.

2. Якщо так сталось, треба обмінювати місцями новий елемент з його батьком, доки його значення не стане більшим за значення батька.

3. Крок 2 буде завершено, коли новий елемент досягне кореня або коли батько нового елемента буде мати значення, яке більше або дорівнює значенню нового елемента.

Розглянемо приклад Max-Heap, створеного в попередньому прикладі.

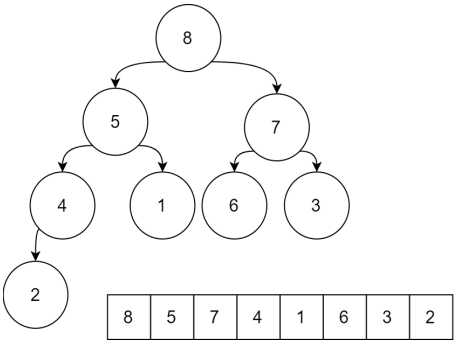
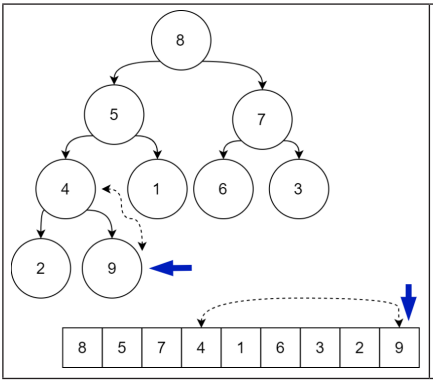


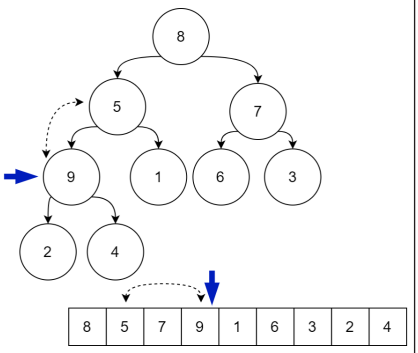
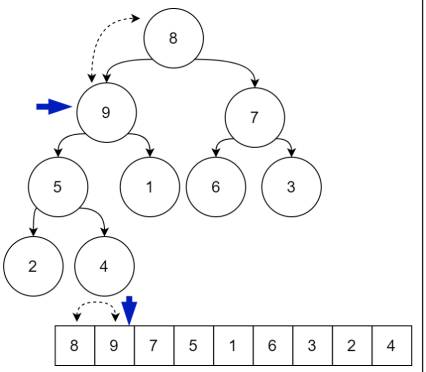
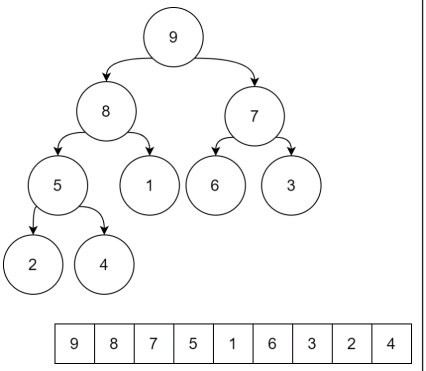
Рис. 7.4. Max-Heap

Розглянемо приклад додавання до купи елемента зі значенням 9. Елемент додається в кінець масиву Heap. Тепер значення буде підніматися вгору шляхом порівняння з батьком. Значення буде додано до індексу 8, а його батьком буде (N–1)/2 = індекс 3.

Таблиця 7.4

	У кінець масиву було додано нове значення 9, оскільки значення 9 більше за 4, то воно буде обміняно місцями з ним

Продовж. табл. 7.4

	Використовується метод percolateUp(), і значення переміщується вгору, доки не буде задоволено властивість heap
	Тепер значення за індексом 1 порівнюється зі значенням з індексом 0 і, щоб задовольнити властивості купи, вони міняються місцями
	Тепер новий вузол було додано до створеного раніше Max-Heap

```
fun add(value: Int) {
    if (size == arr.size) {
        doubleSize()
    }
    arr[size++] = value
    percolateUp(size - 1)
}

private fun doubleSize() {
    val old = arr
    arr = IntArray(arr.size * 2)
    System.arraycopy(old, 0, arr, 0, size)
}
```

Аналіз: перевіряємо, чи не переповнена купа. Якщо купа вже переповнена, то створюємо ще один масив удвічі більшої ємності. Копіюємо в нього вміст купи. Потім додаємо новий елемент до купи, а потім використовуємо операцію percolateUp() для відновлення властивості купи.

Часова складність вставки в купу складає O(log(n)).

Видалення – Dequeue

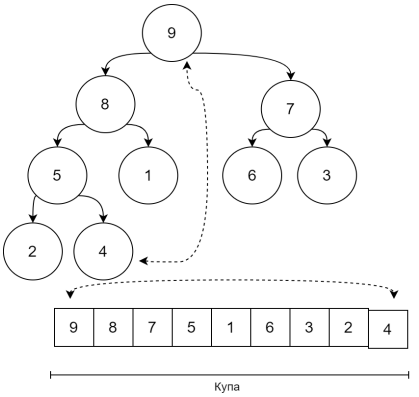
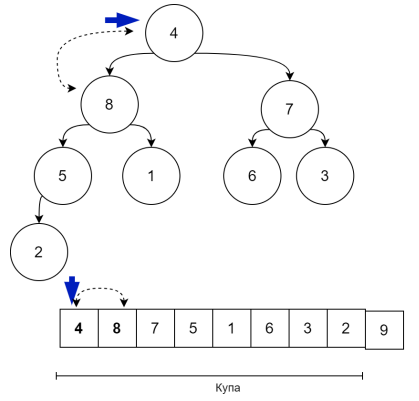
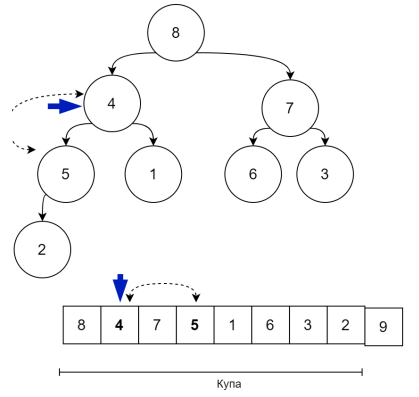
1. Скопіювати значення в корені купи у змінну, яка буде використовуватися для повернення цього значення.

2. Скопіювати останній елемент купи в корінь, а потім зменшити розмір купи на 1. Назвемо цей переміщений елемент «не на своєму місці».

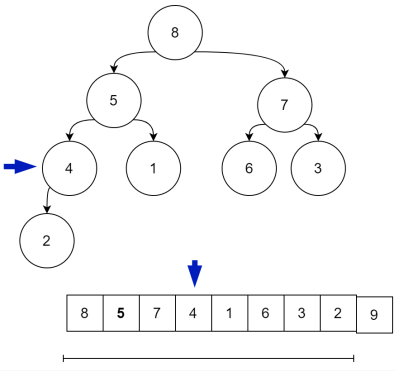
3. Відновити властивість купи, помінявши місцями елемент «не на своєму місці» з його дочірнім елементом з найбільшим значенням. Повторювати цей процес до тих пір, поки елемент «не на своєму місці» не досягне листка, або поки його значення не буде більшим або рівним значенню всіх його дочірніх елементів.

4. Повернути значення, яке було збережено на кроці 1.

Таблиця 7.5

	Значення на початку та в кінці купи міняються місцями. Розмір купи зменшується на одиницю. Властивість купи порушено, тому потрібно виконати просочування вниз, порівнюючи вершину з її дочірніми вершинами і відновлюючи властивість купи
	percolateDown() продовжується шляхом порівняння з дочірніми вузлами
	percolateDown()

Продовж. табл. 7.5

	percolateDown() завершено

```
fun remove(): Int {
    if (isEmpty()) {
        throw IllegalStateException()
    }
    val value = arr[0]
    arr[0] = arr[size - 1]
    size--
    percolateDown(0)
    return value
}
```

Аналіз: значення на вершині купи міняється місцями зі значенням у кінці масиву купи. Розмір купи зменшується на одиницю і викликається percolateDown() для відновлення властивості купи.

Часова складність видалення елементів з купи становить O(log(n)).

Обхід купи

Купи не призначені для обходу (пошуку якогось елемента) або друку вмісту купи. Вони створюються для швидкого пошуку мінімального або максимального елемента. Проте якщо потрібно обійти купу, просто обходьте масив послідовно. *Часова складність* обходу – лінійна O(n).

```
fun print() {
    print("Heap : ")
    for (i in 0..<size) {
        print("${arr[i]} ")
    }
    println()
}

// Testing code.
fun main() {
    val hp = Heap(true)
    hp.add(1)
    hp.add(6)
    hp.add(5)
    hp.add(7)
    hp.add(3)
    hp.add(4)
    hp.add(2)
    hp.print()
    while (!hp.isEmpty())
        print("${hp.remove()} ")
}
```

Результат:

```
Heap : 1 3 2 7 6 5 4
1 2 3 4 5 6 7
```

Реалізація PriorityQueue в Kotlin Collection

Реалізація PriorityQueue у варіанті Min-Heap:

```
import java.util.PriorityQueue
fun main() {
    val pq = PriorityQueue<Int>()
    val arr = intArrayOf(1, 2, 10, 8, 7, 3, 4, 6, 5, 9)
    for (i in arr) {
        pq.add(i)
    }
    println("Heap : $pq")
    while (pq.isNotEmpty()) {
        print(" ${pq.remove()}")
    }
}
```

Результат:

```
Heap : [1, 2, 3, 5, 7, 10, 4, 8, 6, 9]
1 2 3 4 5 6 7 8 9 10
```

Для реалізації Max-Heap треба створити PriorityQueue таким чином:

```
val pq = PriorityQueue<Int>(Collections.reverseOrder());
```

HeapSort

1. Використовуючи функцію createHeap(), створити Max-Heap із заданого списку елементів. Ця операція займе O(N) часу.

2. Поміняти місцями максимальне значення в корені купи і значення в кінці купи за адресою arr[size-1]. Зменшити розмір купи на 1.

3. Відновлення, елемент у корені купи є «не на своєму місці». Відновити властивість heap, помінявши місцями елемент «не на своєму місці» з його дочірнім елементом з найбільшим значенням. Повторювати цей процес доки елемент «не на своєму місці» не досягне листка або не матиме значення, що є більшим або рівним значенню всіх його дочірніх елементів.

4. Повторювати операції 2 і 3 доти, поки в купі не залишиться лише один елемент.

```
fun heapSort(array: IntArray, inc: Boolean) {
// Створити max heap для сортування за зростанням
    val heap = Heap(array, !inc)
    for (i in array.indices) {
        array[array.size - i - 1] = heap.remove()
    }
}
// Testing code.
fun main() {
    val a2 = intArrayOf(1, 9, 6, 7, 8, 2, 4, 5, 3)
    heapSort(a2, true)
    for (i in a2.indices) {
        print(a2[i].toString() + " ")
    }
    println()
    val a3 = intArrayOf(1, 9, 6, 7, 8, 2, 4, 5, 3)
    heapSort(a3, false)
    for (i in a3.indices) {
        print(a3[i].toString() + " ")
    }
}
```

Результат:

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
```

Задачі з Heap

K-й найменший

Задача: у вказаному масиві знайти K-те найменше значення.

Перший розв'язок: відсортуйте заданий масив, а потім виведіть значення з індексом K–1. Часова складність складає O(n*log(n)).

```
fun kthSmallest(arr: IntArray, k: Int): Int {
    Arrays.sort(arr)
    return arr[k - 1]
}
// Testing code.
fun main() {
    val arr = intArrayOf(8, 7, 6, 5, 7, 5, 2, 1)
    println("Kth Smallest : " + kthSmallest(arr, 3))
}
```

Результат:

```
Kth Smallest : 5
```

Другий розв'язок: створити Min-Heap з масиву, викликати операцію deleteMin() K разів і остання операція дасть K-те найменше значення.

Часова складність: O(n*log(n)).

```
fun kthSmallest2(arr: IntArray, size: Int, k: Int): Int {
    val pq = PriorityQueue<Int>()
    for (i in 0..<size) pq.add(arr[i])
    for (i in 0..<k - 1) pq.remove()
    return pq.peek()
}
```

Третій розв'язок: Max-Heap. Створити Max-Heap з перших K елементів масиву. Пройтися по решті елементів масиву. Якщо верхнє значення купи більше за елемент масиву, то видалити верхнє значення і додати елемент масиву до купи. Якщо верхнє значення менше за елемент масиву, то ігноруйте

елемент масиву. Виконуйте цей крок до тих пір, поки не пройдете всі елементи масиву. K-й найменший елемент знаходиться на вершині купи.

Загальна часова складність складає O(n*log(k)).

```
fun kthSmallest3(arr: IntArray, size: Int, k: Int): Int {
    val pq = PriorityQueue<Int>(Collections.reverseOrder())
    for (i in 0..<size) {
        if (i < k) pq.add(arr[i]) else {
            if (pq.peek() > arr[i]) {
                pq.add(arr[i])
                pq.remove()
            }
        }
    }
    return pq.peek()
}
```

Добуток K найменших елементів

Задача: задано масив додатних елементів. Знайти добутки K мінімальних елементів масиву.

Перший розв'язок: сортування, відсортуйте масив і знайдіть добутки перших K елементів. Сортування займе O(n*log(n)) часу. А потім знаходження добутку займе O(n) часу, тому *загальна часова складність* становить O(n*log(n)).

```
fun kSmallestProduct(arr: IntArray, k: Int): Int {
    Arrays.sort(arr)
    var product = 1
    for (i in 0..<k) product *= arr[i]
    return product
}

// Testing code.
fun main() {
    val arr = intArrayOf(8, 7, 6, 5, 7, 5, 2, 1)
    println("Kth Smallest product: " + kSmallestProduct(arr, 3))
}
```

Результат:

```
Kth Smallest product: 10
```

Другий розв'язок: Min-Heap. Створіть Min-Heap з масиву. Потім витягніть K елементів з цієї купи і знайдіть їхній

добуток. Створення купи займе O(n) часу. Витяг елементів з купи займає O(log(n)) часу. Витягується K елементів з купи. Таким чином, *загальна часова складність* складає O(k*log(n)).

```
fun kSmallestProduct2(arr: IntArray, size: Int, k: Int): Int {
    val pq = PriorityQueue<Int>()
    var i = 0
    var product = 1
    while (i < size) {
        pq.add(arr[i])
        i++
    }
    i = 0
    while (i < size && i < k) {
        product *= pq.remove()
        i += 1
    }
    return product
}
```

Min-Heap?

Задача: задано список. Визначте, чи є він Binary Min-Heap.

Розв'язок: властивість Min-Heap, яку ми перевіримо, полягає в тому, що значення елемента батька завжди менше або дорівнює значенню дочірнього елемента. *Часова складність*: O(n).

```
fun isMinHeap(arr: IntArray, size: Int): Boolean {
    var lchild: Int
    var rchild: Int
// індекс останнього елемента = size - 1
    for (parent in 0..<size / 2 + 1) {
        lchild = parent * 2 + 1
        rchild = parent * 2 + 2
// перевірка властивості купи.
        if (lchild < size && arr[parent] > arr[lchild] ||
            rchild < size && arr[parent] > arr[rchild]) return false
    }
    return true
}
// Testing code.
fun main() {
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8)
    println("isMinHeap : " + isMinHeap(arr, arr.size))
}
```

Max-Heap?

Задача: задано список. Визначте, чи є він Binary Max-Heap.

Розв'язок: властивість Max-Heap, яку ми перевіримо, полягає в тому, що значення елемента батька завжди більше або дорівнює індексу дочірнього елемента. *Часова складність*: O(n).

```
fun isMaxHeap(arr: IntArray, size: Int): Boolean {
    var lchild: Int
    var rchild: Int
// індекс останнього елемента = size - 1
    for (parent in 0..<size / 2 + 1) {
        lchild = parent * 2 + 1
        rchild = parent * 2 + 2
// перевірка властивості купи.
        if (lchild < size && arr[parent] < arr[lchild] ||
            rchild < size && arr[parent] < arr[rchild]) return false
    }
    return true
}
// Testing code.
fun main() {
    val arr = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8)
    println("isMaxHeap : " + isMaxHeap(arr, arr.size))
}
```

Використання структури Heap у прикладних задачах

1. **HeapSort:** один з найкращих методів сортування на місці та має складність у часі N*log(N) у всіх сценаріях.

2. **Алгоритми вибору:** знаходження мінімального, максимального, як мінімального, так і максимального, медіани або навіть K-го найбільшого елемента може бути зроблено за лінійний час (часто постійний час) з використанням купи.

3. **Пріоритетні черги:** реалізація пріоритетної черги використовується у графових алгоритмах, таких як алгоритм Прима та алгоритм Дейкстри. Купа є корисною структурою даних, коли вам потрібно видалити об'єкт з найвищим (або найнижчим) пріоритетом. Планувальники, таймери.

4. **Графові алгоритми (BFS):** використання купи як внутрішньої структури даних для обходу зменшує час виконання на поліноміальний порядок. Прикладами таких задач є мінімакс Прима.

5. Через відсутність указівників операції виконуються швидше, ніж з бінарним деревом. Крім того, деякі більш складні купи (наприклад, біноміальні) можна ефективно об'єднувати, що не так просто зробити для бінарного дерева.

Тема 8. **Хеш-таблиці**

Розглянемо задачу пошуку значення в масиві. Якщо масив не відсортовано, то у нас немає іншого варіанта, окрім як переглядати кожен елемент по черзі, тому складність пошуку буде O(n). Якщо масив відсортовано, то ми можемо знайти значення за O(log(n)) – за логарифмічний час, використовуючи бінарний пошук.

А чи можна отримати розташування / індекс значення, яке ми шукаємо в масиві, за допомогою магічної функції, яка повертає індекс за константний час? Ми можемо безпосередньо перейти в це місце і побачити, чи є там значення, яке ми шукаємо, за O(1) константного часу. Хеш-функція працює так само, звичайно, без жодної магії.

Хеш-таблиця

Хеш-таблиця – це структура даних, яка відображає ключі, значення. Кожна позиція хеш-таблиці називається *корзиною* (*bucket*, *slot*). Хеш-таблиця використовує хеш-функцію для обчислення індексу масиву. Використовувати хеш-таблицю є сенс, коли кількість ключів невелика порівняно з кількістю можливих ключів.

Процес зберігання даних у хеш-таблиці можна представити таким чином:

1. Створити масив розміром *M* для зберігання даних, який називається хеш-таблицею.

2. Знайти хеш-код даних, застосувавши до них через хеш-функцію.

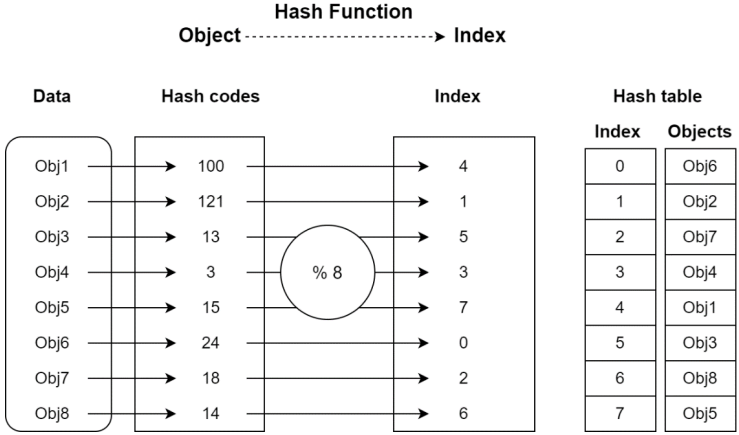


Рис. 8.1. Хеш-таблиця

3. Обчислити остачу від ділення хеш-коду на розмір хеш-таблиці, щоб отримати індекс у таблиці, у якій будуть зберігатися дані.

4. Нарешті, зберегти ці дані за знайденим індексом.

Процес пошуку значення в хеш-таблиці за допомогою хеш-функції відбувається наступним чином:

1. Знайти хеш-код шуканого ключа, застосувавши до нього хеш-функцію.

2. Знайти остачу від ділення хеш-коду на розмір хеш-таблиці, щоб отримати індекс таблиці, за яким зберігається шукане значення.

3. Нарешті, отримати значення за вказаним індексом.

Абстрактний тип даних «хеш-таблиця»

Абстрактний тип даних «хеш-таблиця» містить наступні функції:

1. Insert(x), додати x до набору даних.
2. Delete(x), видалити x з набору даних.
3. Search(x) – пошук x у наборі даних.

Хеш-функція

Хеш-функція – це функція, яка генерує індекс у таблиці для заданого ключа. Хеш-функція, яка генерує унікальний індекс для кожного ключа, називається *ідеальною або досконалою* (*perfect*) *хеш-функцією*.

Приклад найпростішої хеш-функції:

```
fun computeHash(key: Int): Int {
    return key % tableSize
}
```

Існує багато хеш-функцій. Наведена вище функція є дуже простою хеш-функцією. До цієї функції буде додано різну логіку генерації хеша для отримання кращого хеша.

Колізії

Коли хеш-функція генерує однаковий індекс для двох або більше різних ключів, виникає проблема, відома як колізія. В ідеалі хеш-функція повинна повертати унікальну адресу для кожного ключа, але на практиці це неможливо.

Властивості хорошої хеш-функції:

1. Вона повинна забезпечувати рівномірний розподіл хеш-значень. Нерівномірний розподіл збільшує кількість колізій і вартість їх вирішення.

2. Хеш-функція повинна швидко обчислюватися і повертати значення в межах діапазону хеш-таблиці.

3. Бажано обирати хеш-функцію з хорошим алгоритмом вирішення колізій, яку можна використовувати для обчислення альтернативних індексів у разі виникнення колізії.

4. Рекомендується обирати хеш-функцію, яка використовує необхідну інформацію, надану в ключі.

5. Вона повинна мати високий коефіцієнт завантаження (Load Factor) для даного набору ключів.

Коефіцієнт завантаження – Load Factor

Коефіцієнт завантаження = (максимально допустима кількість елементів у хеш-таблиці) / (розмір хеш-таблиці).

Виходячи з наведеного вище визначення, коефіцієнт завантаження – це характеристика того, скільки елементів може містити хеш-таблиця до того, як її ємність буде збільшена. Коли кількість елементів перевищує добуток коефіцієнта завантаження на поточну ємність, хеш-таблиця перехешовується. Зазвичай розмір хеш-таблиці до того ж подвоюється. Наприклад, якщо коефіцієнт завантаження дорівнює 0,75, а ємність хеш-таблиці 1024, то ми можемо додати 768 елементів до хеш-таблиці без її перехешування.

Методи розв'язку колізій

Колізії хеша практично неминучі під час хешування великої кількості ключів. Методи, які використовуються для пошуку альтернативного розташування в хеш-таблиці, називаються *обробкою колізій*. Існує декілька методів розв'язку колізій для обробки колізій під час хешування.

Найбільш поширеними і широко використовуваними методами є наступні:

1. Відкрита адресація.
2. Метод ланцюжків.

Хешування з відкритою адресацією

У разі використання лінійної відкритої адресації хеш-таблиця представляється одновимірним масивом з індексами від 0 до tableSize-1.

Одним з методів вирішення колізій є перегляд хеш-таблиці та пошук іншого вільного слоту для зберігання значення, яке спричинило колізію. Простий спосіб – переходити від одного слоту до іншого у певному порядку, поки не знайдеться вільне місце. Цей процес вирішення колізій називається *відкритою адресацією*.

Лінійне зондування

У лінійному зондуванні ми намагаємося вирішити колізію індексу хеш-таблиці шляхом послідовного перебору вільних місць у хеш-таблиці. Припустимо, що k – це індекс, отриманий з хеш-функції. Якщо k-й індекс уже заповнений, то ми будемо шукати (k+1)%M, потім (k+2)%M і т. д. Коли ми знайдемо вільний слот, ми вставимо дані у цей вільний слот.

Розв'язувальна функція лінійного зондування:

```
fun resolverFun(index: Int): Int {
    return index
}
```

Реалізація лінійного зондування

Структура хеш-таблиці складається з масиву, який використовується для зберігання даних, і ще одного прапора масиву, який використовується для відстеження того, чи зайнятий якийсь слот чи ні. Він також містить розмір масиву.

Клас HashTable

```
class HashTable (private val tableSize: Int = 16) {
    var keys: IntArray = IntArray(tableSize + 1)
    var values: IntArray = IntArray(tableSize + 1)
    var flags: IntArray = IntArray(tableSize + 1){ EMPTY_VALUE }

    companion object {
        private const val EMPTY_VALUE = -1
        private const val DELETED_VALUE = -2
        private const val FILLED_VALUE = 0
    }

    fun computeHash(key: Int): Int {
        return key % tableSize
    }

    fun resolverFun(index: Int): Int {
        return index
    }
    //...
}
```

Якщо хеш-індекс уже зайнято, для отримання нового індексу використовується функція resolver:

```
fun add(key: Int, value: Int): Boolean {
    var hashValue = computeHash(key)
    for (i in 0 until tableSize) {
        if (flags[hashValue] == EMPTY_VALUE ||
            flags[hashValue] == DELETED_VALUE) {
            keys[hashValue] = key
            values[hashValue] = value
            flags[hashValue] = FILLED_VALUE
            return true
        }
        hashValue += resolverFun(i)
        hashValue %= tableSize
    }
    return false
}

fun add(value: Int): Boolean {
    return add(value, value)
}
```

Функція add() використовується для додавання значень до хеш-таблиці. Обчислюється перший хеш. Потім ми намагаємося помістити це значення в хеш-таблицю. Шукаємо порожні вузли або ліниво видалені вузли для вставки значень. Якщо вставка не вдалася, ми шукаємо нове місце за допомогою функції resolver.

```
fun find(key: Int): Boolean {
    var hashValue = computeHash(key)
    for (i in 0 until tableSize) {
        if (flags[hashValue] == EMPTY_VALUE) {
            return false
        }
        if (flags[hashValue] == FILLED_VALUE && keys[hashValue] == key) {
            return true
        }
        hashValue += resolverFun(i)
        hashValue %= tableSize
    }
    return false
}
```

```
fun get(key: Int): Int {
    var hashValue = computeHash(key)
    for (i in 0 until tableSize) {
        if (flags[hashValue] == EMPTY_VALUE) {
            return -1
        }
        if (flags[hashValue] == FILLED_VALUE && keys[hashValue] == key) {
            return values[hashValue]
        }
        hashValue += resolverFun(i)
        hashValue %= tableSize
    }
    return -1
}
```

Функція find() використовується для пошуку ключа в хеш-таблиці. А функція get() використовується для знаходження значення, що відповідає заданому ключу. Обчислюється перший хеш. Потім ми намагаємося знайти це значення в хеш-таблиці. Шукаємо серед слотів, де можливо розташовано це значення або в порожніх слотах. Якщо ми знаходимо шукане значення, то повертаємо його, а якщо не знаходимо, то повертаємо –1. Ми використовуємо функцію resolver, щоб знайти наступний можливий індекс для пошуку.

```
fun remove(key: Int): Boolean {
    var hashValue = computeHash(key)
    for (i in 0 until tableSize) {
        if (flags[hashValue] == EMPTY_VALUE) {
            return false
        }
        if (flags[hashValue] == FILLED_VALUE && keys[hashValue] == key) {
            flags[hashValue] = DELETED_VALUE
            return true
        }
        hashValue += resolverFun(i)
        hashValue %= tableSize
    }
    return false
}
```

Функція remove() використовується для видалення значень з хеш-таблиці. Ми не видаляємо значення, а лише

позначаємо його як DELETED_NODE. Так само, як і під час вставки та пошуку, ми використовуємо функцію resolverFun, щоб знайти наступне ймовірне місцезнаходження ключа.

```
fun print() {
    print("Hash Table contains ::")
    for (i in 0 until tableSize) {
        if (flags[i] == FILLED_VALUE) {
            print("(${keys[i]} -> ${values[i]}) ")
        }
    }
    println()
}

// Testing code.
fun main() {
    val ht = HashTable(1024)
    ht.add(1, 10)
    ht.add(2, 20)
    ht.add(3, 30)
    ht.print()
    println("Find key 2 : " + ht.find(2))
    println("Value at key 2 : " + ht.get(2))
    ht.remove(2)
    println("Find key 2 : " + ht.find(2))
}
```

Результат:

```
Hash Table contains ::(1 -> 10) (2 -> 20) (3 -> 30)
Find key 2 : true
Value at key 2 : 20
Find key 2 : false
```

Функція print виводить уміст хеш-таблиці. Функція main демонструє використання хеш-таблиці.

Хешування з використанням методу у ланцюжків

Інший метод вирішення колізій базується на ідеї розміщення ключів, що зіткнулися, у зв'язаному списку. Цей метод називається *методом ланцюжків*. Для прискорення пошуку варто зберігати цей список відсортованим.

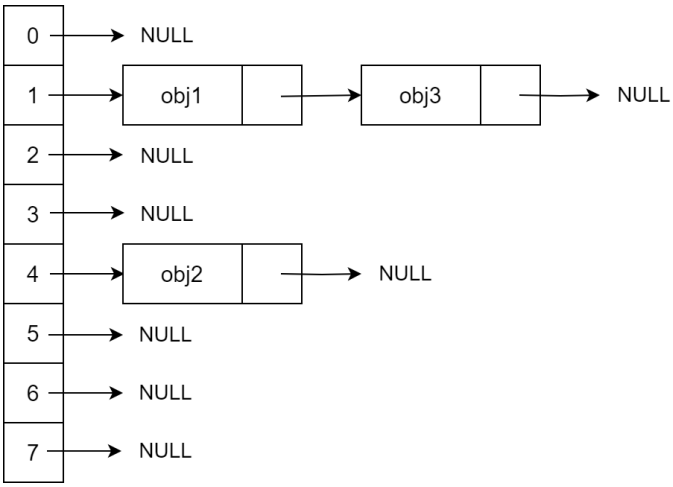


Рис. 8.2. Хеш-таблиця з ланцюжками

Реалізація методу у ланцюжків

```
class HashTableSC {

    inner class Node( val key: Int, val value: Int, var next: Node?)

    private val tableSize = 512
    private var listArray = arrayOfNulls<Node?>(tableSize)

    private fun computeHash(key: Int): Int { // division method
        return key % tableSize
    }

    fun add(key: Int, value: Int) {
        val index = computeHash(key)
        listArray[index] = Node(key, value, listArray[index])
    }

    fun add(value: Int) {
        add(value, value)
    }

    fun remove(key: Int): Boolean {
        val index = computeHash(key)
        var nextNode: Node?
        var head: Node? = listArray[index]
```

```
        if (head != null && head.key == key) {
            listArray[index] = head.next
            return true
        }
        while (head != null) {
            nextNode = head.next
            if (nextNode != null && nextNode.key == key) {
                head.next = nextNode.next
                return true
            } else {
                head = nextNode
            }
        }
        return false
    }

    fun find(key: Int): Boolean {
        val index = computeHash(key)
        var head = listArray[index]
        while (head != null) {
            if (head.key == key) {
                return true
            }
            head = head.next
        }
        return false
    }

    fun get(key: Int): Int {
        val index = computeHash(key)
        var head = listArray[index]
        while (head != null) {
            if (head.key == key) {
                return head.value
            }
            head = head.next
        }
        return -1
    }

    fun print() {
        print("Hash Table contains ::")
        for (i in 0 until tableSize) {
            var head: Node? = listArray[i]
            while (head != null) {
                print("(${head.key} -> ${head.value}) ")
                head = head.next
```

```
            }
        }
        println()
    }
}
```

Результат:

```
Hash Table contains ::(1 -> 10) (2 -> 20) (3 -> 30)
Find key 2 : true
Value at key 2 : 20
Find key 2 : false
```

Примітка. Важливо відзначити, що розмір хеш-таблиці повинен бути таким, щоб усі слоти в ньому були зайняті. В іншому випадку частина таблиці буде невикористаною.

Реалізація множин(sets) у Kotlin Collections

Set (множина) використовується для зберігання унікальних елементів. Set реалізовано на основі хеш-таблиці. Клас множини HashSet<> реалізований із використанням хеш-таблиці, його елементи зберігаються не в послідовному порядку.

```
// Testing code.
fun main() {
// Створюємо set
    val hs = HashSet<String>()
// Додаємо елементи
    hs.add("Pineapple")
    hs.add("Apple")
    hs.add("Apricot")
    println(hs)
    println("Apple present: " + hs.contains("Apple"))
    println("Grapes present: " + hs.contains("Grapes"))
    hs.remove("Apple")
    println("Apple present: " + hs.contains("Apple"))
}
```

Результат:

```
[Apple, Apricot, Pineapple]
Apple present: true
Grapes present: false
Apple present: false
```

Реалізація словника (Dictionary) у Kotlin Collections

HashMap<> – це структура даних, яка відображає ключі у значення. Вона використовує хеш-таблицю, тому пари ключ-значення не зберігаються у відсортованому порядку. Словники не допускають дублювання ключів, але значення можуть дублюватися.

```
// Testing code.
fun main() {
// Створюємо map
    val hm = HashMap<String, Int>()
// Додаємо елементи
    hm["Pineapple"] = 40 // або hm.put("Pineapple", 40)
    hm["Apple"] = 10
    hm["Apricot"] = 20

    println("Size : " + hm.size)
    for (key in hm.keys) {
        println(key + " cost : " + hm[key])
    }

    println("Apple present: " + hm.containsKey("Apple"))
    println("Grapes present: " + hm.containsKey("Grapes"))
    hm.remove("Apple")
    println("Apple present: " + hm.contains("Apple"))
}
```

Результат:

```
Size : 3
Apple cost : 10
Apricot cost : 20
Pineapple cost : 40
Apple present: true
Grapes present: false
Apple present: false
```

Задачі з хешуванням

Анаграми

Задача: визначити, чи є два рядки анаграмами. Анаграма – це слово або словосполучення, утворене перестановкою літер іншого слова або словосполучення.

Розв'язок: два слова є анаграмами, якщо вони мають однаковий розмір і однакові символи. Спочатку треба перевірити,

чи обидва рядки мають однаковий розмір. Потім пройтися по першому рядку і додати кількості символів до хеш-таблиці. Потім пройтися по другому рядку і віднімати кількості символів з хеш-таблиці. Якщо під час проходу другого рядка не знайдено символ у хеш-таблиці або якщо відповідна кількість дорівнює нулю, то рядки не є анаграмами. Якщо другий рядок пройдено повністю, але не знайдено жодної невідповідності, то маємо анаграми.

Часова складність: O(n), n – розмір рядка.

```
fun isAnagram(str1: CharArray, str2: CharArray): Boolean {
    if (str1.size != str2.size) return false
    val hm = HashMap<Char, Int>()
    for (ch in str1) {
        hm.merge(ch, 1, Int::plus)
    }
    for (ch in str2) {
        val value = hm.getOrDefault(ch, 0)
        if (value == 0)
            return false
        else
            hm[ch] = value - 1
    }
    return true
}

//Testing code.
fun main() {
    val first: CharArray = "hello".toCharArray()
    val second: CharArray = "elloh".toCharArray()
    val third: CharArray = "world".toCharArray()
    println("isAnagram : " + isAnagram(first, second))
    println("isAnagram : " + isAnagram(first, third))
}
```

Результат:

```
isAnagram : true
isAnagram : false
```

Видалити дублікати

Задача: видалити дублікати у списку чисел.

Розв'язок: створюється новий список. Треба пройти вхідний список і його елементи додаються до хеш-таблиці.

Якщо якогось числа немає в хеш-таблиці, то воно додається до хеш-таблиці та вихідного списку. Якщо деяке число вже є у хеш-таблиці, то воно ігнорується.

Часова складність: O(n).

```
fun removeDuplicate(list: List<Int>): List<Int> {
    val hs = HashSet<Int>()
    val out = mutableListOf<Int>()
    for (num in list) {
        if (num !in hs) {
            out += num
            hs.add(num)
        }
    }
    return out
}

//Testing code.
fun main() {
    val first = listOf(1, 3, 2, 1, 2, 3, 5)
    println(removeDuplicate(first))
}
```

Результат:

```
[1, 3, 2, 5]
```

Знайти пропущене число

Задача: у заданому масиві цілих чисел потрібно знайти число із вказаного діапазону, яке в ньому відсутнє.

Розв'язок: усі елементи масиву додаються до хеш-таблиці. У хеш-таблиці шукаються всі цілі числа від початку до кінця заданого діапазону. Якщо пропущений елемент знайдено, то повертається його значення.

Часова складність: O(n).

```
fun findMissing(arr: IntArray, start: Int, end: Int): Int {
    val hs = HashSet<Int>()
    for (i in arr) {
        hs.add(i)
    }
    for (curr in start..end) {
        if (curr !in hs) return curr
    }
    return Int.MAX_VALUE
}
```

```
//Testing code.
fun main() {
    val arr = intArrayOf(1, 2, 3, 5, 6, 7, 8, 9, 10)
    println(findMissing(arr, 1, 10))
}
```

Результат:

```
4
```

Висновки

Найчастіші застосування хеш-таблиці наступні:

1. Хеш-таблиці зазвичай використовуються в алгоритмах, де потрібен швидкий пошук даних.

2. Хеш-таблиці використовуються в деревах символів для компіляторів.

3. У деяких базах даних дані зберігаються в хеш-таблицях у вигляді пар ключ-значення.

Тема 9. **Графи**

Граф представляється впорядкованою парою G, де $G = (V, E)$, де V – скінченна множина точок, які називаються *вершинами*, а E – скінченна множина ребер. Кожне ребро є парою (u, v), де $u, v \in V$.

Структура графа складається з наступних двох компонентів:

1. Скінченої множини вузлів, які називаються *вершинами*.
2. Скінченої множини пар вершин, які називаються *ребрами*. Ребра з'єднують дві вершини.

Деякі реальні приклади графів:

1. Карти Google: різні локації можуть бути представлені вершинами графа, а шляхи між ними – ребрами графа. Алгоритми теорії графів використовуються, щоб запропонувати найкоротший та найшвидший шлях між двома місцями.

2. Визначення друзів у Facebook: кожен профіль користувача представлений у вигляді вершин графа, а їхні дружні стосунки представлені ребрами графа. У теорії графів існують алгоритми для пошуку та рекомендації друзів.

3. Топологічне сортування: топологічне сортування – це метод пошуку послідовностей, у яких деякі події потрібно виконати для завершення певного завдання, досліджуючи залежність різних подій одна від одної. Наприклад, послідовність вивчення дисциплін, які треба відвідувати, щоб отримати диплом з комп'ютерних наук. Або послідовність кроків, які ми робимо щодня, коли готуємось до роботи.

4. Транспортна мережа: карта авіаліній представлена у вигляді графа. Різні аеропорти представлені як вершини

графа, і, якщо існує безпосадковий рейс з аеропорту *u* до аеропорту *v*, то на графі існує ребро з вершини *u* до вершини *v*. Ви можете захотіти дістатися з одного місця в інше, за допомогою алгоритмів теорії графів ми можемо обчислити найкоротший, найшвидший або найдешевший шлях від початкового місця до пункту призначення.

Авіасполучення між великими містами Європи також можна зобразити за допомогою графа, наведеного нижче: кожне місто зображено у вигляді вершин, а авіарейси між містами – у вигляді ребер.

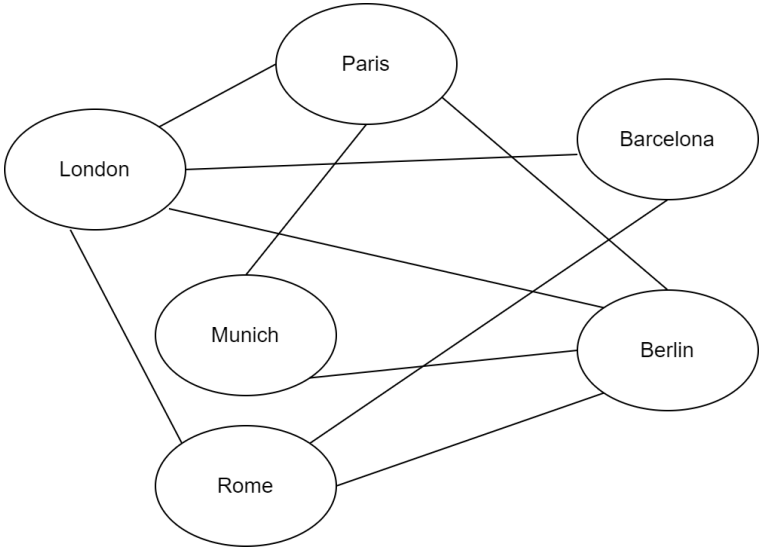


Рис. 9.1. Приклад графа – авіасполучення

Термінологія графів

Неорієнтований граф. Неорієнтований граф – це граф, у якому ребра не мають напрямків. Іншими словами, ребра графа мають два напрямки.

Ребро (x, y) ідентичне ребру (y, x).

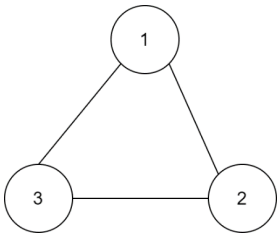


Рис. 9.2. Неорієнтований граф

Орієнтований граф або **оргграф.** Орієнтований граф – це граф, у якого ребра мають напрямок. Тут ребра графа є односторонніми. Ребро (x, y) не тотожне ребру (y, x).

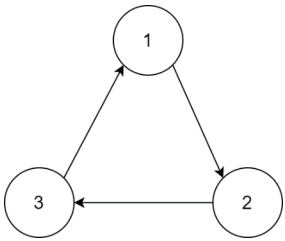


Рис. 9.3. Орієнтований граф

Зважений граф: граф є зваженим, якщо його ребра мають певне значення або вагу, пов'язану з ними.

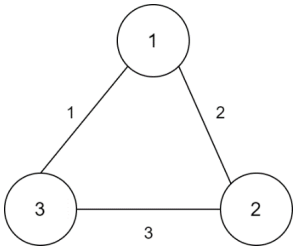


Рис. 9.4. Зважений граф

Незважений граф: граф, у якому ребра не мають ваги.

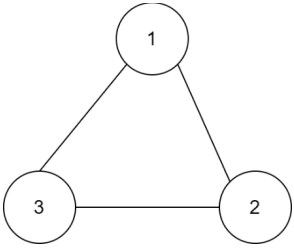


Рис. 9.5. Незважений граф

Шлях: шлях – це послідовність ребер між двома вершинами. Довжина шляху визначається як сума ваг усіх ребер на шляху.

Простий шлях: шлях з різними вершинами називається *простим шляхом*.

Суміжні вершини: дві вершини *u* та *v* є суміжними, якщо існує ребро, кінцями якого є *u* та *v*.

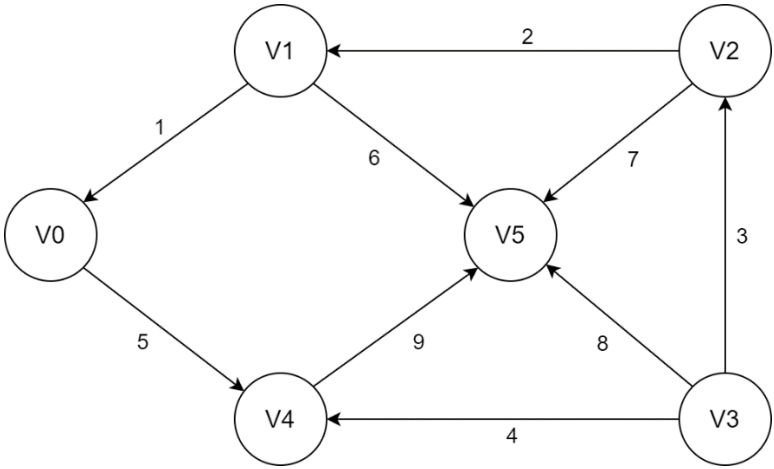


Рис. 9.6. Зважений орієнтований граф

У графі на рис. 9.6:

V = {V1, V2, V3, V4, V5} – множина вершин.

E = { (v1, v0, 1), (v2, v1, 2), (v3, v2, 3), (v3, v4, 4), (v5, v4, 5), (v1, v5, 6), (v2, v5, 7), (v3, v5, 8), (v4, v5, 9) } – множина зважених ребер.

Вхідний степінь вершини v, позначається indeg(v) – це кількість вхідних ребер у вершину v. Вихідний степінь вершини v, позначається outdeg(v) – це кількість вихідних ребер з вершини v.

Степінь вершини v, позначається deg(v) – це загальна кількість ребер, які мають одну кінцеву точку у вершині v. deg(v) = indeg(v) + outdeg(v).

У наведеному вище графі: deg(V2) = 3, indeg(V2) = 1 і outdeg(V2) = 2.

Цикл – це шлях, який починається і закінчується в одній і тій же вершині та включає в себе принаймні одну вершину. Ребро є самопетлею (також називають *петлею*), якщо дві його кінцеві точки збігаються. Це форма циклу.

Циклічний граф: граф, який має один або більше циклів, називається *циклічним.*

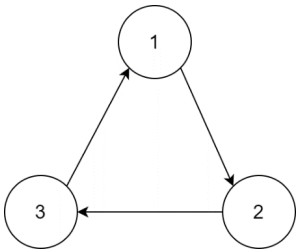


Рис. 9.7. Циклічний граф

Ациклічний граф: граф, який не має циклу, називається *ациклічним графом*. **Орієнтований ациклічний граф:** орієнтований граф, який не має жодного циклу, називається *орієнтованим ациклічним графом.*

Досяжність: вершина *v* досяжна з вершини *u*, або *u* досяжна з *v*, якщо існує шлях з *u* до *v*. У неорієнтованому графі, якщо *v* досяжна з *u*, то *u* досяжна з *v*. Однак у орієнтованому графі можлива ситуація, коли *u* досяжна з *v*, але не існує шляху з *v* до *u*.

Зв'язний граф: граф називається *зв'язним*, якщо для будь-яких двох вершин існує шлях між ними.

Сильнозв'язний граф: орієнтований граф називається *сильнозв'язним*, якщо для кожної пари вершин *u* і *v* існує шлях з *u* в *v* і шлях з *v* в *u*.

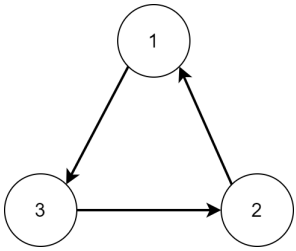


Рис. 9.8. Сильнозв'язний граф

Сильнозв'язні компоненти: орієнтований граф може містити різні підграфи, які є сильнозв'язними. Ці підграфи називаються *сильнозв'язними компонентами.*

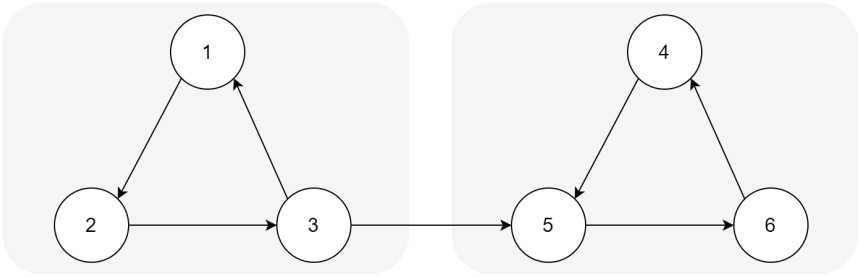


Рис. 9.9. Сильнозв'язні компоненти

Слабозв'язний граф: орієнтований граф називається *слабозв'язним*, якщо для кожної пари вершин *u* і *v* існує шлях з *u* в *v* або з *v* в *u*.

Повний граф: граф називається *повним*, якщо будь-яка його вершина з'єднана ребром з усіма іншими вершинами. Повний граф має максимальну кількість ребер. Для неорієнтованого графа з n вершин загальна кількість ребер буде n*(n–1)/2, а для орієнтованого графа загальна кількість ребер буде n*(n–1).

Підграфом графа *G* називається граф, вершини та ребра якого є підмножиною вершин та ребер графа *G*.

Остовний підграф *G* – це граф, який з'єднує всі вершини *G*.

Ліс – це граф без циклів.

Деревом називається зв'язний граф без циклів. Між будь-якими двома вершинами *u* та *v* існує лише один простий шлях. Якщо ми видалимо будь-яке ребро з дерева, то воно стане лісом.

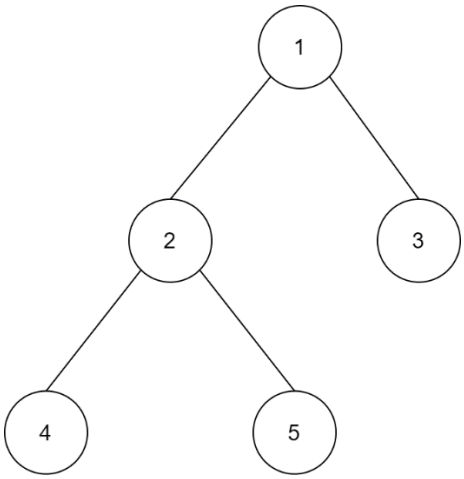


Рис. 9.10. Дерево

Остовне дерево графа – це дерево, яке з'єднує всі вершини графа. Оскільки остовне дерево є деревом, воно не повинно мати циклів.

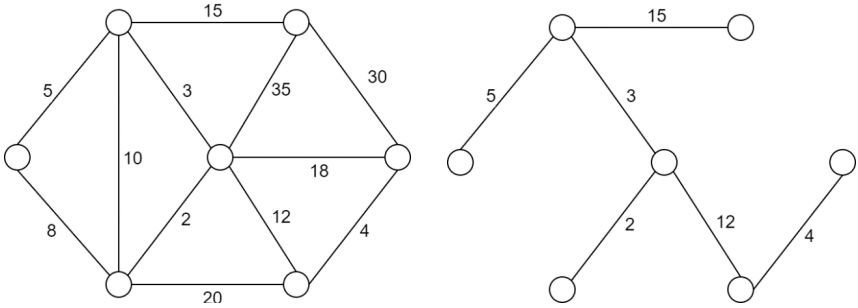


Рис. 9.11. Зважений граф та його мінімальне остовне дерево

Гамільтонів шлях – це шлях, у якому кожна вершина відвідується рівно один раз без повторень, він не повинен починатися і закінчуватися в одній і тій же вершині.

Гамільтоновим циклом називається такий гамільтонів шлях, що містить ребро від його останньої вершини до його першої вершини. **Гамільтонів цикл** – це цикл, який відвідує кожну вершину рівно один раз, і він повинен починатися та закінчуватися в одній і тій же вершині.

Ейлерів шлях – це шлях у графі, який відвідує кожне ребро рівно один раз.

Ейлерів цикл – це ейлерів шлях, який починається і закінчується в одній і тій же вершині. Або *Ейлеровим циклом* називається шлях у графі, який відвідує кожне ребро рівно один раз, і починається, і закінчується в одній і тій же вершині.

Представлення графів

Існують два найпоширеніших способи представлення графа:

1. Матриця суміжності.
2. Список суміжності.

Матриця суміжності

Матриця суміжності – це двовимірна матриця розміром V рядків і V стовпців, де V – кількість вершин у графі. Матриця суміжності представляється за допомогою двовимірного масиву розміром V * V.

Припустимо, що масив має назву «adj», тобто вузол adj[i][j] відповідає ребру з вершини і до вершини j. Якщо adj[i][j] не дорівнює нулю, то це означає, що існує шлях з вершини і до вершини j. Для незваженого графа значення в масиві дорівнюють або 0, якщо шляху немає, або 1, якщо шлях є. Але у випадку зваженого графа значення в матриці вказує на вартість / вагу шляху з вершини і до вершини j.

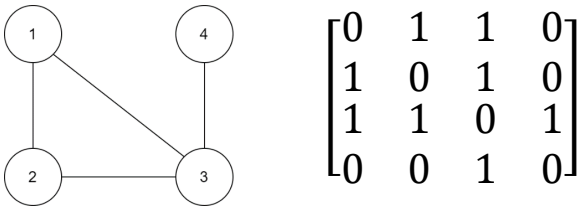


Рис. 9.12. Граф і його матриця зв’язності

У наведеному вище графі кожна вершина має вагу 1, тому матриця суміжності містить лише 1 або 0. Якщо б ребра мали різну вагу, то в матриці була б заповнена ця вага.

Наступний приклад демонструє використання матриці зв’язності для представлення графів.

```
class GraphAM (var vertexCount: Int){
    var adj = Array(vertexCount) { IntArray(vertexCount) }

    fun addDirectedEdge(src: Int, dst: Int, cost: Int = 1) {
        adj[src][dst] = cost
    }

    fun addUndirectedEdge(src: Int, dst: Int, cost: Int = 1) {
        addDirectedEdge(src, dst, cost)
        addDirectedEdge(dst, src, cost)
    }
```

```
    fun print() {
        for (i in adj.indices) {
            print("Vertex $i is connected to : ")
            for (j in adj.indices) {
                if (adj[i][j] != 0) print("$j(cost: ${adj[i][j]})")
            }
            println()
        }
    }

}

// Testing code.
fun main() {
    val gph = GraphAM(4)
    gph.addUndirectedEdge(0, 1)
    gph.addUndirectedEdge(0, 2)
    gph.addUndirectedEdge(1, 2)
    gph.addUndirectedEdge(2, 3)
    gph.print()
}
```

Результат:

```
Vertex 0 is connected to : 1(cost: 1)2(cost: 1)
Vertex 1 is connected to : 0(cost: 1)2(cost: 1)
Vertex 2 is connected to : 0(cost: 1)1(cost: 1)3(cost: 1)
Vertex 3 is connected to : 2(cost: 1)
```

Пояснення до прикладу:

1. У конструкторі графа з N вершин створюється двовимірний масив adj розміром NxN.

2. Функція addDirectedEdge() використовується для додавання спрямованого ребра з вершини *u* до вершини *v* шляхом встановлення поля adj[u][v] масиву adj в одиницю.

3. Функція addUndirectedEdge() використовується для додавання неорієнтованого ребра. Вона викликає функцію addDirectedEdge() двічі, щоб з'єднати вказані вершини в обох напрямках.

Аналіз складності:

Просторова складність представлення матриці суміжності становить $O(n^2)$, оскільки створюється двовимірний масив.

Часова складність пошуку складає O(1).

Визначити, чи існує ребро з вершини *u* у вершину *v* можна за постійний час.

Переваги і недоліки матриці суміжності

Переваги матриці суміжності:

- Запит на наявність ребра з вершини *u* до вершини *v* займає O(1).
- Видалення ребра займає O(1) час.

Недоліки матриці суміжності:

- *Просторова складність* становить $O(n^2)$.
- Навіть якщо граф розріджений (з невеликою кількістю ребер), *просторова складність* залишається незмінною.
- Додавання нової вершини займає $O(n^2)$ часу.

Розріджена матриця: у величезному графі кожна вершина з'єднана з декількома вершинами. Таким чином, більшість місць у матриці суміжності залишаються порожніми. Така матриця називається *розрідженою*. У більшості реальних задач розріджена матриця суміжності не є гарним вибором для складних графових даних.

Список суміжності

Більш компактним способом зберігання графів є список суміжності. Список суміжності – це масив зв'язних списків. Кожен елемент списку відповідає ребрам графа. Розмір масиву дорівнює кількості вершин у графі. Нехай масив

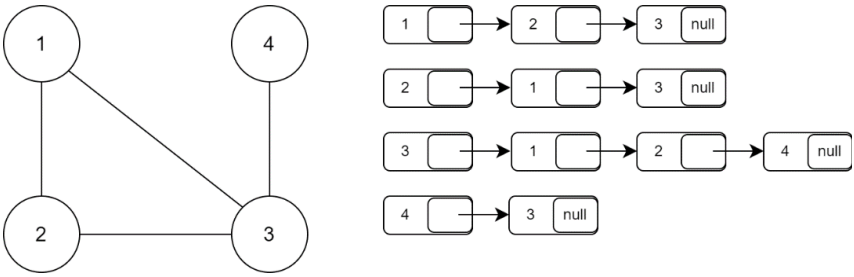


Рис. 9.13. Список зв'язності

називається arr[]. Елемент arr[k] відповідає k-й вершині і є списком вершин, безпосередньо з'єднаних з k-ю вершиною. Ваги ребер можна зберігати у вузлах зв'язаних списків.

Список суміжності допомагає представити розріджений граф. Представлення у вигляді списку суміжності також дозволяє знаходити всі вершини, які безпосередньо з'єднані з будь-якою вершиною, лише одним скануванням списку зв'язків.

Розглянемо таке представлення:

```
class Graph(var count: Int) {
    private class Edge(var u: Int, var v: Int, var cost: Int) :
Comparable<Edge> {
        override operator fun compareTo(other: Edge): Int {
            return cost - other.cost
        }
    }

    private var INFINITY = 999999

    private val adj = MutableList(count) { LinkedList<Edge>() }

    fun addDirectedEdge(u: Int, v: Int, cost: Int = 1) {
        adj[u].add(Edge(u, v, cost))
    }

    fun addUndirectedEdge(u: Int, v: Int, cost: Int = 1) {
        addDirectedEdge(u, v, cost)
        addDirectedEdge(v, u, cost)
    }

    fun print() {
        for (i in adj.indices) {
            print("Vertex $i is connected to : ")
            for (adn in adj[i]) {
                print("${adn.u}(cost: ${adn.cost}) ")
            }
            println()
        }
    }
}


// Testing code.
fun main() {
    val g = Graph(4)
```

```
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(0, 2)
    g.addUndirectedEdge(1, 2)
    g.addUndirectedEdge(2, 3)
    g.print()
}
```

Результат:

```
Vertex 0 is connected to : 0(cost: 1) 0(cost: 1)
Vertex 1 is connected to : 1(cost: 1) 1(cost: 1)
Vertex 2 is connected to : 2(cost: 1) 2(cost: 1) 2(cost: 1)
Vertex 3 is connected to : 3(cost: 1)
```

Аналіз:

1. У конструкторі графів з N вершин створюється масив списків розміром N.

2. Функція addDirectedEdge() використовується для додавання орієнтованого ребра з вершини *u* до вершини *v* шляхом додавання кортежу (*u*, *v, cost*) до відповідного списку вершин.

3. Функція addUndirectedEdge() використовується для додавання до графа неорієнтованого ребра. Вона викликає функцію addDirectedEdge() двічі, щоб з'єднати вершини *u* та *v* в обох напрямках.

Аналіз складності:

Просторова складність списку суміжності становить O(E+V), для створення масиву вершин та зберігання ребер з кожної вершини. *Часова складність* пошуку ребра з вершини *u* до вершини *v* становить inoutdegree(u). Нам потрібно пройтися по списку сусідів вершини *u*. У найгіршому випадку це може зайняти O(E).

Обхід графів

Обхід – це процес дослідження графа шляхом перегляду всіх його ребер і вершин. Для обходу графа використовуються два алгоритми: пошук у глибину (DFS) та пошук у ширину (BFS). Ці ж алгоритми можуть бути використані для пошуку деякої вершини у графі, перевірки досяжності вершини і т. д.

Наведемо перелік деяких задач, які розв'язуються за допомогою обходу графа:

1. Визначення шляху з вершини *u* до вершини *v*, або повідомлення про помилку, якщо такого шляху не існує.
2. За заданою початковою вершиною *s* знайти мінімальну кількість ребер з вершини *s* до всіх інших вершин графа.
3. Перевірити, чи є граф *G* зв'язним.
4. Знаходження остовного дерева графа.
5. З'ясувати, чи існує у графі деякий цикл.

Пошук у глибину – Depth First Traversal (DFS)

Ми запускаємо алгоритм DFS з початкової точки і йдемо у глибину графа, поки не дійдемо до кінцевої вершини, а потім рухаємося до батьківської вершини (Backtrack). У DFS ми використовуємо стек, щоб отримати наступну вершину для початку пошуку. Крім того, ми можемо використовувати рекурсію (системний стек) для того ж самого.

Обхід у глибину графа подібний до обходу у глибину для дерева. Єдина відмінність полягає в тому, що графи можуть містити цикли (дерева не мають циклів). Отже, під час обходу ми можемо повернутися до тієї самої вершини. Щоб уникнути повторної обробки однієї і тієї ж вершини, ми використовуємо булевий масив відвіданих вершин.

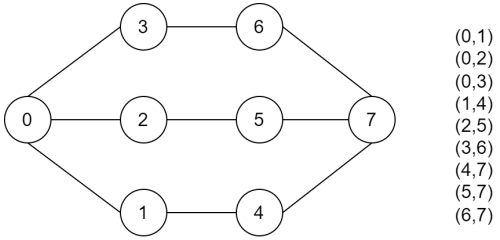
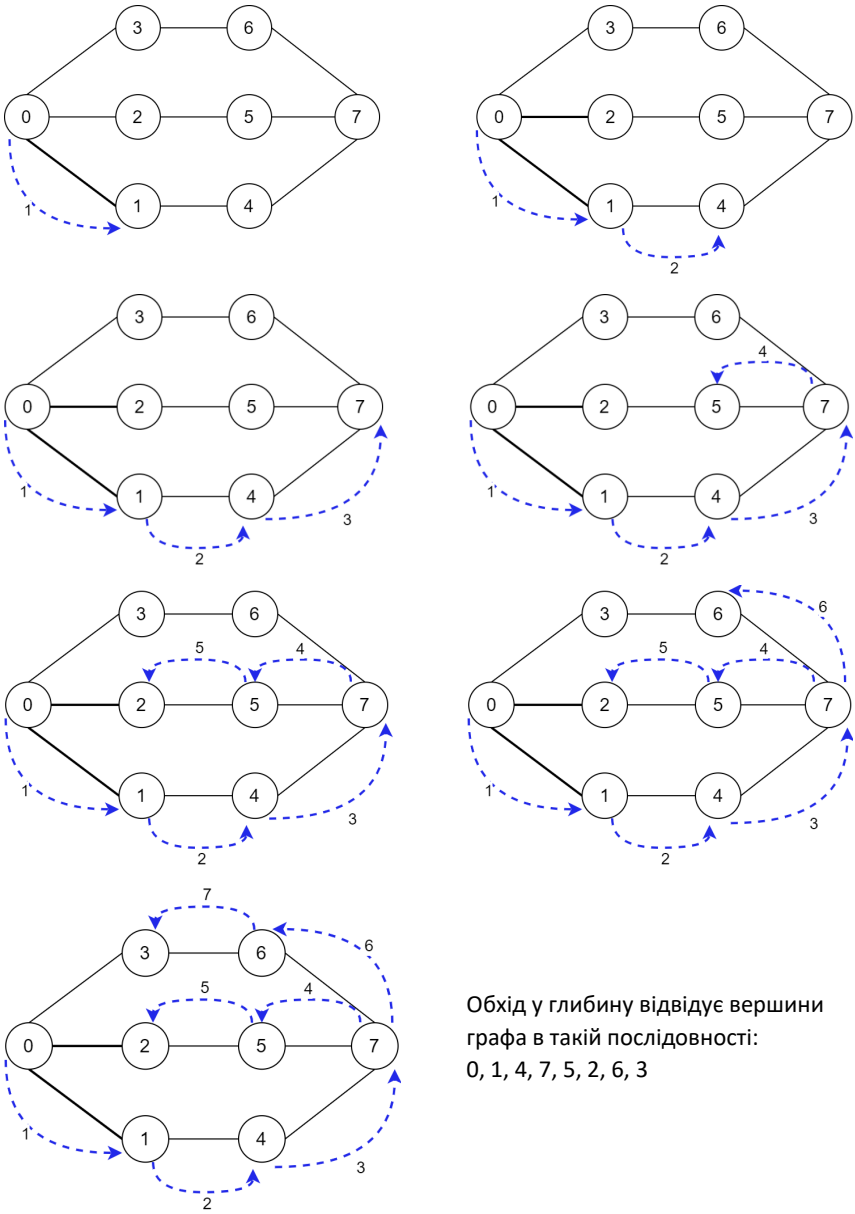


Рис. 9.14. Граф для обходу

На наступних рисунках показано обхід цього графа у глибину (DFS).



Обхід у глибину відвідує вершини графа в такій послідовності:
0, 1, 4, 7, 5, 2, 6, 3

Рис. 9.15. Обхід у глибину (DFS)

Реалізація DFS на основі стека

Алгоритм дій для DFS:

1. Додати початковий вузол у стек та відмітити цей вузол, як такий, що вже опрацьовано.

2. Повторювати кроки 3–5 до тих пір, поки стек не стане порожнім.

3. Витягнути вузол зі стека і назвати цей вузол поточним.

4. Обробити поточний вузол. Вивести на друк і т. д.

5. Пройти всі дочірні вузли поточного вузла і, якщо вони ще не опрацьовані, додати їх у стек.

6. Коли стек став порожнім, обхід завершено.

Приклад коду – DFS на основі стека:

```
fun dfsStack(source: Int, target: Int): Boolean {
    val visited = BooleanArray(count)
    val stack = ArrayDeque<Int>()
    stack.push(source)
    visited[source] = true
    while (stack.isNotEmpty()) {
        val curr = stack.pop()
        val adl = adj[curr]
        for (adn in adl) {
            if (!visited[adn.v]) {
                visited[adn.v] = true
                stack.push(adn.v)
            }
        }
    }
    return visited[target]
}

// Testing code.
fun main() {
    val g = Graph(8)
    g.addUndirectedEdge(0, 3)
    g.addUndirectedEdge(0, 2)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(1, 4)
    g.addUndirectedEdge(2, 5)
    g.addUndirectedEdge(3, 6)
    g.addUndirectedEdge(6, 7)
    g.addUndirectedEdge(5, 7)
    g.addUndirectedEdge(4, 7)
```

```
    println("Path from 0 to 6 : " + g.dfsStack(0, 6))
    g.print()
}
```

Результат:

```
Path from 0 to 6 : true
```

Аналіз складності: *часова складність* алгоритму DFS, коли граф представлено у вигляді списку суміжності, становить O(V+E), де V – загальна кількість вершин, а E – загальна кількість ребер у графі.

У разі подання графа у вигляді матриці суміжності часова складність алгоритмів становить $O(V^2)$. Нам потрібно обійти сусідні вершини, що є ефективним у графі, коли він представлений у вигляді списку суміжності.

Рекурсивна реалізація DFS

Кроки алгоритму для DFS:

1. У функції dfs() створити масив visited для відстеження відвіданих вузлів.

2. У функції dfs() у рекурсивну функцію dfsRecursive() як поточний вузол передається вузол-джерело функції dfsRecursive().

3. Усі вузли, які були відвідані з індексних вузлів, далі передаються у функцію dfsRecursive() через рекурсивні виклики.

4. Ця рекурсія повернеться у функцію dfs(), коли будуть відвідані всі вузли, які доступні для відвідання з джерела. Нарешті, ми можемо дізнатися, чи відвідано цільову вершину чи ні, переглянувши масив visited.

Приклад коду – DFS на основі рекурсії:

```
fun dfs(source: Int, target: Int): Boolean {
    val visited = BooleanArray(count)
    dfsRecursive(source, visited)
    return visited[target]
}

fun dfsRecursive(index: Int, visited: BooleanArray) {
    visited[index] = true
```

```
    val adl = adj[index]
    for (adn in adl) {
        if (!visited[adn.v]) dfsRecursive(adn.v, visited)
    }
}
```

Аналіз складності:

1. *Часова складність* алгоритму DFS, коли граф представлено у вигляді списку суміжності, становить O(V+E), де V – загальна кількість вершин, а E – загальна кількість ребер у графі.

2. У разі подання графа у вигляді матриці суміжності *часова складність* алгоритмів стає $O(V^2)$.

Пошук у ширину – Breadth First Traversal (BFS)

В алгоритмі BFS граф обходить шар за шаром. Спочатку обходяться ті вершини графа, що є найближчими до початкової точки. Для реалізації BFS використовується черга.

Обхід графа в ширину подібний до обходу дерева в ширину. Єдина відмінність полягає в тому, що граф може містити цикли (дерева не мають циклів). Отже, під час обходу ми можемо повернутися до тієї самої вершини знову. Щоб уникнути повторної обробки одного й того ж вузла, ми використовуємо булевий масив відвідувань.

Алгоритм дій для BFS:

1. Додати початковий вузол у чергу та відмітити цей вузол як такий, що вже опрацьовано.

2. Повторювати кроки 3–5 до тих пір, поки черга не стане порожньою.

3. Витягнути вузол з черги і назвати цей вузол поточним.

4. Обробити поточний вузол (вивести на друк і т. д.).

5. Пройти всі дочірні вузли поточного вузла і, якщо вони ще не опрацьовані, додати їх у чергу.

6. Коли черга стала порожньою, обхід завершено

На наступних рисунках показано обхід графа (рис. 9.16) у ширину (BFS).

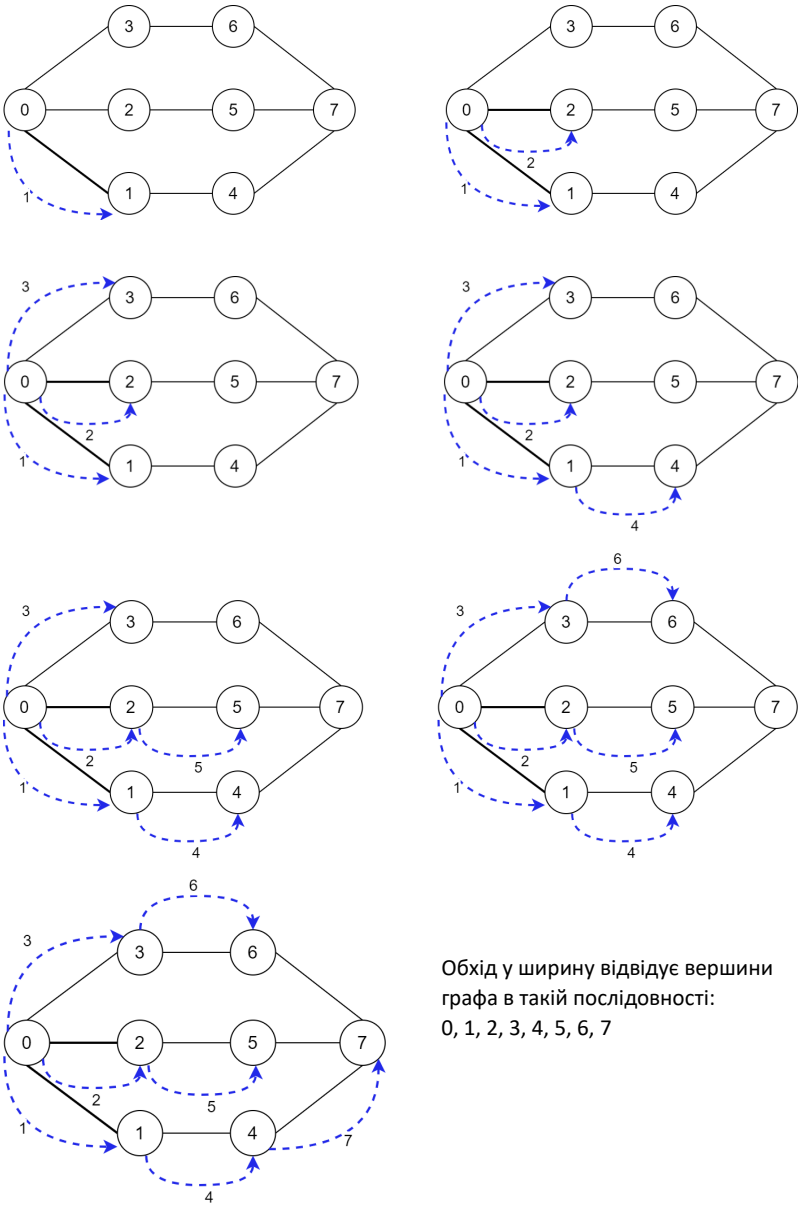


Рис. 9.16. Обхід у ширину (BFS)

```
fun bfs(source: Int, target: Int): Boolean {
    val visited = BooleanArray(count)
    val que = LinkedList<Int>()
    que.add(source)
    visited[source] = true
    while (que.isNotEmpty()) {
        val curr = que.remove()
        val adl = adj[curr]
        for (adn in adl) {
            if (!visited[adn.v]) {
                visited[adn.v] = true
                que.add(adn.v)
            }
        }
    }
    return visited[target]
}

// Testing code.
fun main() {
    val g = Graph(8)
    g.addUndirectedEdge(0, 3)
    g.addUndirectedEdge(0, 2)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(1, 4)
    g.addUndirectedEdge(2, 5)
    g.addUndirectedEdge(3, 6)
    g.addUndirectedEdge(6, 7)
    g.addUndirectedEdge(5, 7)зз
    g.addUndirectedEdge(4, 7)
    println("Path from 0 to 6 : " + g.bfs(0, 6))
}
```

Результат:

```
Path from 0 to 6 : true
```

Аналіз складності: виконання обходу DFS та BFS займає O(V+E) часу, де V – кількість ребер, досяжних з початкової вершини, а E – кількість ребер у графі.

Задачі на основі DFS та BFS

Орієнтований ациклічний граф і топологічне сортування

Орієнтований ациклічний граф – це орієнтований граф без циклу. Орієнтований ациклічний граф представляє

відношення, яке є більш загальним, ніж дерево. Нижче наведено приклад, як може використовуватися такий граф. Існує *N* курсів, які потрібно вивчити в університеті, щоб стати випускником, причому деякі курси є обов'язковими до вивчення перед початком вивчення інших.

Топологічне сортування – це метод упорядкування вузлів орієнтованого графа, у якому вузли представляють завдання, а ребра – залежність між цими завданнями. Щоб топологічне сортування працювало, потрібно, щоб граф був орієнтованим ациклічним, тобто не містив циклів. Просто використовуйте DFS для виконання топологічного сортування.

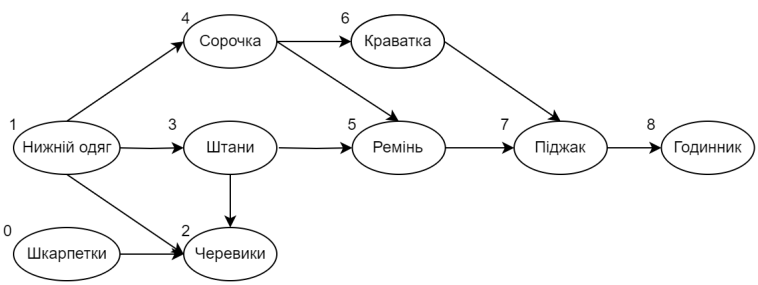


Рис. 9.17. Приклад орієнтованого ациклічного графа

Розв'язок: топологічне сортування використовує DFS-обхід графа. Ми використовуємо стек для виконання топологічного сортування. У топологічному графі дочірні вузли залежать від батьківського вузла, тому спочатку до стека додаються всі дочірні вузли, а потім лише батьківський вузол. До того ж батьківська вершина знаходиться вище дочірньої вершини, або залежні вузли знаходяться нижче у стеку, ніж батьківські. Приклад, наведений нижче, демонструє цей алгоритм.

```
fun dfsUtil2(index: Int, visited: BooleanArray, stk: Deque<Int>) {
    visited[index] = true
    val adl  = adj[index]
    for (adn in adl) {
```

```
        if (!visited[adn.v]) dfsUtil2(adn.v, visited, stk)
    }
    stk.push(index)
}

fun topologicalSort() {
    val stk = ArrayDeque<Int>()
    val visited = BooleanArray(count)
    for (i in 0 until count) {
        if (!visited[i]) {
            dfsUtil2(i, visited, stk)
        }
    }
    print("Topological Sort:")
    while (stk.isNotEmpty()) {
        print(" " + stk.pop())
    }
}

// Testing code.
fun main() {
    val g = Graph(9)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(1, 2)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(1, 4)
    g.addDirectedEdge(3, 2)
    g.addDirectedEdge(3, 5)
    g.addDirectedEdge(4, 5)
    g.addDirectedEdge(4, 6)
    g.addDirectedEdge(5, 7)
    g.addDirectedEdge(6, 7)
    g.addDirectedEdge(7, 8)
    g.topologicalSort()
}
```

Результат:

```
Topological Sort: 1 4 6 3 5 7 8 0 2
```

Аналіз складності: *часова складність* – O(V+E), де V – кількість вершин, а E – кількість ребер. Алгоритм сортування топології є алгоритмом DFS зі стеком. Отже, *часова складність* така ж, як і у DFS.

Визначення шляху з вершини *u* у вершину *v*

Задача: визначити, чи існує шлях з вершини *u* у вершину *v*.

Розв'язок: якщо існує шлях з вершини *u* у вершину *v*, то під час виконання DFS з вершини *u* ми відвідаємо вершину *v*.

```
fun pathExist(source: Int, dest: Int): Boolean {
    val visited = BooleanArray(count)
    dfsRecursive(source, visited)
    return visited[dest]
}
// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(3, 4)
    g.addDirectedEdge(1, 4)
    println("PathExist: " + g.pathExist(0, 4))
}
```

Аналіз складності: *часова складність* така ж, як і у DFS для реалізації на основі списку суміжності графа – O(V+E), а для реалізації на основі матриці зв'язності – $O(V^2)$.

Підрахувати всі шляхи DFS

Задача: за заданими початковою та кінцевою вершинами знайти всі можливі шляхи з початкової вершини до кінцевої.

Розв'язок: виконаємо DFS-обхід графа, починаючи з початкової вершини, і порахуємо всі шляхи, які ведуть до кінцевої вершини. Ми будемо зберігати масив visited[], щоб переконатися, що одна і та ж вершина не буде розглянута двічі.

```
fun countAllPathDFS(visited: BooleanArray, source: Int, dest: Int): Int {
    if (source == dest) {
        return 1
    }
    var count = 0
    visited[source] = true
    val adl = adj[source]
```

```
    for (adn in adl) {
        if (!visited[adn.v]) {
            count += countAllPathDFS(visited, adn.v, dest)
        }
    }
    visited[source] = false
    return count
}
fun countAllPath(src: Int, dest: Int): Int {
    val visited = BooleanArray(count)
    return countAllPathDFS(visited, src, dest)
}
// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(3, 4)
    g.addDirectedEdge(1, 4)
    println("Path Count : " + g.countAllPath(0, 4))
}
```

Результат:

```
Path Count : 3
```

Аналіз складності: якщо граф повністю зв'язний, то в цьому випадку в DFS можливе проходження N! різних шляхів. Таким чином, *часова складність* буде O(N!).

Роздрукувати всі шляхи

Задача: вивести всі шляхи від початкової вершини до кінцевої.

Розв'язок: виконаємо DFS-обхід графа, починаючи з початкової вершини, і порахуємо всі шляхи, що ведуть до кінцевої вершини. Ми будемо зберігати масив visited[] для того, щоб переконатися, що одна і та ж вершина не буде розглянута двічі. Шлях зберігається у стеку path. Коли пункт призначення знайдено, шлях виводиться на екран.

```
fun printAllPathDFS(visited: BooleanArray, source: Int, dest: Int,
                    path: Stack<Int>) {
    path.push(source)
    if (source == dest) {
        println(path)
        path.pop()
        return
    }
    visited[source] = true
    val adl = adj[source]
    for (adn in adl) {
        if (!visited[adn.v]) printAllPathDFS(visited, adn.v, dest, path)
    }
    visited[source] = false
    path.pop()
}

fun printAllPath(src: Int, dest: Int) {
    val visited = BooleanArray(count)
    val path = Stack<Int>()
    printAllPathDFS(visited, src, dest, path)
}

// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(3, 4)
    g.addDirectedEdge(1, 4)
    g.printAllPath(0, 4)
}
```

Результат:

```
[0, 1, 3, 4]
[0, 1, 4]
[0, 2, 3, 4]
```

Кореневa вершина

Задача: знайти кореневу вершину у графі. *Кореневою вершиною* називається вершина, з якої існує шлях у всі інші вершини графа. Якщо існує декілька кореневих вершин, то поверніть будь-яку з них.

Розв'язок: потрібно знайти верхню вершину стека в топологічному сортуванні. Коли DFS-обхід викликається для деякої вершини, то встановлюється індекс відвідуваності цієї вершини і всі інші вершини, які є залежними від неї або мають шлях від поточної вершини, також будуть позначені як набір індексів відвідуваності цієї вершини. Якщо ми знайдемо вершину, індекс відвідуваності якої не встановлено, це означає, що попередня вершина не є кореневою, а нова вершина може бути кореневою. Ми пройдемо через ці вершини і спробуємо знайти кореневу вершину.

```
fun rootVertex(): Int {
    val visited = BooleanArray(count)
    var result = -1
    for (i in 0 until count) {
        if (!visited[i]) {
            dfsRecursive(i, visited)
            result = i
        }
    }
    print("Root vertex is: $result")
    return result
}
// Testing code.
fun main() {
    val g = Graph(7)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(4, 1)
    g.addDirectedEdge(6, 4)
    g.addDirectedEdge(5, 6)
    g.addDirectedEdge(5, 2)
    g.addDirectedEdge(6, 0)
    g.rootVertex()
}
```

Результат:

```
Root vertex is: 5
```

Аналіз складності: *часова складність* – O(V+E), де V – кількість вершин, а E – кількість ребер. Після того, як вершина пройдена, вона не може бути пройдена знову.

Транзитивне замикання

Задача: за заданим орієнтованим графом побудуйте матрицю транзитивного замикання або матрицю досяжності. Вершина *v* досяжна з вершини *u*, якщо існує шлях з *u* у *v*. Транзитивне замикання графа *G* – це граф *G*', який містить ту саму множину вершин, що і *G*, і кожен раз, коли існує шлях з вершини *u* у вершину *v* в *G*, існує ребро з *u* у *v* в *G'*.

Розв'язок: створимо двовимірний масив tc[][] і встановимо всі його елементи у нульове значення. Виконуємо DFS-обхід графа для всіх вершин, починаючи з початкової вершини. Якщо існує шлях з вершини *u* до деякої вершини *v*, то позначимо tc[u][v] як 1. Обмеженням у цій задачі є те, що якщо ми вже розглядали деяку вершину призначення *v*, то ми не будемо розглядати її знову. Подальший обхід деяких вершин DFS виконується тільки тоді, коли tc[u][v] дорівнює 0.

```
fun transitiveClosureRecursive(source: Int, dest: Int, tc: Array<IntArray>) {
    tc[source][dest] = 1
    val adl = adj[dest]
    for (adn in adl) {
        if (tc[source][adn.v] == 0) transitiveClosureRecursive(source, adn.v, tc)
    }
}
fun transitiveClosure(): Array<IntArray> {
    val tc = Array(count) { IntArray(count) }
    for (i in 0 until count) {
        transitiveClosureRecursive(i, i, tc)
    }
    return tc
}

// Testing code.
fun main() {
    val g = Graph(4)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(1, 2)
    g.addDirectedEdge(2, 0)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(3, 3)
    val tc = g.transitiveClosure()
    for (i in 0..3) {
```

```
        for (j in 0..3) print("${tc[i][j]} ")
        println()
    }
}
```

Результат:

```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

Аналіз складності: обхід DFS викликається для всіх V вершин. У DFS вершина обробляється лише один раз, коли tc[u][v] дорівнює 0. Таким чином, загальна *часова складність* буде O(V*(V+E)).

Рівень вузла при BFS

Задача: виконати BFS-обхід графа. Разом з вершинами вивести їх відстань від початкової вершини.

Розв'язок: ми хочемо знайти відстань від початкової вершини, тому використовуємо BFS: спочатку обходимо ближні вершини, а потім дальні. Ми створюємо чергу і зберігаємо в ній початкову вершину. Ми також хочемо відслідковувати відстань, тому можемо зберігати відстань у черзі. Або ми можемо зберігати масив level[], який буде відслідковувати відстань від початкової вершини. Кожного разу, коли вершина додається до черги, її відповідна відстань від вихідної вершини також оновлюється в масиві level[]. *Часова складність* буде O(V+E).

```
fun bfsLevelNode(source: Int) {
    val visited = BooleanArray(count)
    val level = IntArray(count)
    visited[source] = true
    val que = ArrayDeque<Int>()
    que.add(source)
    level[source] = 0
    println("Node - Level")
    while (que.isNotEmpty()) {
        val curr = que.remove()
        val depth = level[curr]
```

```
        val adl = adj[curr]
        println("$curr - $depth")
        for (adn in adl) {
            if (!visited[adn.v]) {
                visited[adn.v] = true
                que.add(adn.v)
                level[adn.v] = depth + 1
            }
        }
    }
}
// Testing code.
fun main() {
    val g = Graph(7)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(0, 2)
    g.addUndirectedEdge(0, 4)
    g.addUndirectedEdge(1, 2)
    g.addUndirectedEdge(2, 5)
    g.addUndirectedEdge(3, 4)
    g.addUndirectedEdge(4, 5)
    g.addUndirectedEdge(4, 6)
    g.bfsLevelNode(1)
}
```

Результат:

```
Node – Level
1 – 0
0 – 1
2 – 1
4 – 2
5 – 2
3 – 3
6 – 3
```

Відстань BFS

Задача: знайти відстань між початковою та кінцевою вершинами графа.

Розв'язок: ми хочемо знайти відстань від початкової вершини, тому використовуємо BFS: спочатку обходимо ближні вершини, а потім дальні. Виконаємо BFS-обхід графа, починаючи з початкової вершини, і відстежуватимемо відстань до сусідніх вершин, перш ніж додавати їх до черги. Якщо ми

знаходимо вершину призначення, то повернути її відстань від початкової вершини. А якщо вершина призначення недосяжна, то повернути –1.

Часова складність буде O(V+E).

```
fun bfsDistance(source: Int, dest: Int): Int {
    val visited = BooleanArray(count)
    val que = LinkedList<Int>()
    que.add(source)
    visited[source] = true
    val level = IntArray(count)
    level[source] = 0
    while (que.isNotEmpty()) {
        val curr = que.remove()
        val depth = level[curr]
        val adl = adj[curr]
        for (adn in adl) {
            if (adn.v == dest) {
                return depth + 1
            }
            if (!visited[adn.v]) {
                visited[adn.v] = true
                que.add(adn.v)
                level[adn.v] = depth + 1
            }
        }
    }
    return -1
}

// Testing code.
fun main() {
    val g = Graph(7)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(0, 2)
    g.addUndirectedEdge(0, 4)
    g.addUndirectedEdge(1, 2)
    g.addUndirectedEdge(2, 5)
    g.addUndirectedEdge(3, 4)
    g.addUndirectedEdge(4, 5)
    g.addUndirectedEdge(4, 6)
    println("BfsDistance(1 - 6) : " + g.bfsDistance(1, 6))
}
```

Результат:

```
BfsDistance(1-6) : 3
```

Пошук циклів у орієнтованому графі

Задача: за заданим орієнтованим графом визначте, чи існує в ньому цикл. Якщо за один обхід деяка вершина проходиться двічі, то цикл існує.

Перший розв'язок: нам потрібно виконати DFS-обхід для всіх вершин. Ті вершини, які вже пройдено, не повинні бути пройдені знову. Для знаходження циклів в орієнтованих графах достатньо відстежувати відвідані вершини.

```
fun isCyclePresentDFS(index: Int, visited: BooleanArray, marked: IntArray):
Boolean {
    visited[index] = true
    marked[index] = 1
    val adl = adj[index]
    for (adn in adl) {
        val dest = adn.v
        if (marked[dest] == 1)
            return true
        if (!visited[dest])
            if (isCyclePresentDFS(dest, visited, marked))
                return true
    }
    marked[index] = 0
    return false
}
fun isCyclePresent() : Boolean {
    val visited = BooleanArray(count)
    val marked = IntArray(count)
    for (index in 0 until count) {
        if (!visited[index])
            if (isCyclePresentDFS(index, visited, marked))
                return true
    }
    return false
}

// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(3, 4)
    println("isCyclePresent : " + g.isCyclePresent())
```

```
    g.addDirectedEdge(4, 1)
    println("isCyclePresent : " + g.isCyclePresent())
}
```

Результат:

```
isCyclePresent : false
isCyclePresent : true
```

Аналіз складності: *часова складність* така ж, як і у DFS. Під час реалізації на основі списку суміжності графа – O(V+E), а для реалізації матриці зв'язності – $O(V^2)$.

Другий розв'язок: визначити, чи є цикл у графі, можна за допомогою кольорового методу. У кольоровому методі початково відвіданому масиву присвоюється значення «білий», це означає, що вершини не були відвідані. Коли ми відвідуємо вершину, ми позначаємо її колір як «сірий». Вузли, що знаходяться на шляху, який ми відвідали, залишаються «сірими», доки не будуть пройдені всі пов'язані з ними вузли, а потім їх колір змінюється на «чорний». Якщо вершина, позначена сірим кольором, відвідується знову, це означає, що на цьому шляху існує цикл.

```
fun isCyclePresentDFSColour(index: Int, visited: IntArray): Boolean {
    visited[index] = 1 // 1 = gray
    var dest: Int
    val adl = adj[index]
    for (adn in adl) {
        dest = adn.v
        if (visited[dest] == 1) // "Gray":
            return true
        if (visited[dest] == 0) // "White":
            if (isCyclePresentDFSColour(dest, visited)) return true
    }
    visited[index] = 2 // "Black"
    return false
}
fun isCyclePresentColour(): Boolean {
    val visited = IntArray(count)
    for (i in 0 until count) {
        if (visited[i] == 0) // "White"
            if (isCyclePresentDFSColour(i, visited)) return true
```

```
    }
    return false
}
```

Аналіз складності: *часова складність* така ж, як і у DFS. Під час реалізації на основі списку суміжності графа – O(V+E), а для реалізації матриці зв'язності – $O(V^2)$.

Пошук цикла у неорієнтованому графі

Задача: визначити, чи існує цикл у неорієнтованому графі.

Перший розв'язок: потрібно виконати DFS-обхід для всіх вершин. Ті вершини, які вже пройдено, не повинні бути пройдені знову. Але відстеження відвіданих вершин достатньо для пошуку циклів у орієнтованому графі. Але в неорієнтованому графі ребра є двонаправленими, тому завжди існує зворотний шлях від дочірньої до батьківської вершини. Тому цей зворотний шлях також потрібно ігнорувати.

```
fun isCyclePresentUndirectedDFS(index: Int, parentIndex: Int, visited:
                                BooleanArray): Boolean {
    visited[index] = true
    var dest: Int
    val adl = adj[index]
    for (adn in adl) {
        dest = adn.v
        if (!visited[dest]) {
            if (isCyclePresentUndirectedDFS(dest, index, visited))
                return true
        } else if (parentIndex != dest)
            return true
    }
    return false
}

fun isCyclePresentUndirected(): Boolean {
    val visited = BooleanArray(count)
    for (i in 0 until count) {
        if (!visited[i] && isCyclePresentUndirectedDFS(i, -1, visited))
            return true
    }
    return false
}
```

```
// Testing code.
fun main() {
    val g = Graph(6)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(1, 2)
    g.addUndirectedEdge(3, 4)
    g.addUndirectedEdge(4, 2)
    g.addUndirectedEdge(2, 5)
    println("Cycle Present : " + g.isCyclePresentUndirected())
    g.addUndirectedEdge(4, 1)
    println("Cycle Present : " + g.isCyclePresentUndirected())
}
```

Результат:

```
Cycle Present : false
Cycle Present : true
```

Аналіз складності: *часова складність* така ж, як і у DFS. Під час реалізації на основі списку суміжності графа – O(V+E), а для реалізації матриці зв'язності – $O(V^2)$.

Другий розв'язок: структура даних «Розділені множини» – це структура даних, яка відстежує ряд розділених або таких, що не перетинаються, множин.

В алгоритмі пошуку об'єднань ми маємо дві операції для підтримки розділених множин:

1. *Find*: знайти множину, у якій присутній певний елемент. Це може бути використано для перевірки того, чи елементи знаходяться в одній множині.

2. *Union*: об'єднати дві множини в одну.

Створіть список ребер вхідного графа. Пройдіть ребрами і перевірте, чи знаходяться дві кінцеві точки ребра в одній або різних множинах. Якщо вони знаходяться у різних множинах, то це ребро не створює циклу, і тоді ці множини повинні бути об'єднані (що є аналогом того, як дві різні групи вершин з'єднуються за допомогою поточного ребра). Якщо ми бачимо, що обидва кінці ребра знаходяться в одній множині, то це означає, що якщо це ребро буде включено, то воно створить цикл. Функція find() знайде кореневу вершину

або перший елемент множини, у якій присутній даний елемент.

Часова складність: O(VE), функція find() може займати лінійний час.

```
fun find(parent: IntArray, idx: Int): Int {
    var index = idx
    var p = parent[index]
    while (p != -1) {
        index = p
        p = parent[index]
    }
    return index
}
fun union(parent: IntArray, x: Int, y: Int) {
    parent[y] = x
}
// Використовуємо масив flags[][], якщо розглядається ребро (x -> y),
// тоді ігноруємо ребро (y -> x).
fun isCyclePresentUndirected2(): Boolean {
    val parent = IntArray(count) { -1 }
    val edge = LinkedList<Edge>()
    val flags = Array(count) { BooleanArray(count) }
    for (i in 0 until count) {
        val ad = adj[i]
        for (adn in ad) {
// У масиві flags, якщо вже розглянуто ребро (x -> y), тоді ребро (y -> x) ігноруємо.
            if (!flags[adn.v][adn.u]) {
                edge.add(adn)
                flags[adn.u][adn.v] = true
            }
        }
    }
    for (e in edge) {
        val x = find(parent, e.u)
        val y = find(parent, e.v)
        if (x == y) {
            return true
        }
        union(parent, x, y)
    }
    return false
}
```

Транспонування графа

Задача: транспонуванням орієнтованого графа *G* називається орієнтований граф *G'*, який має ту ж саму множину вершин, але напрямок ребер змінено на протилежний.

Розв'язок: створіть порожній граф *G'* з тією ж кількістю вершин, що і вхідний граф *G*. Пройдіться по різним вершинам вхідного графа *G*. Якщо у вхідному графі *G* існує ребро з *v* в *u*, то додайте ребро з *u* в *v* у новому графі *G'*. Повернути новий граф *G'*.

```
fun transposeGraph(): Graph {
    val g = Graph(count)
    for (i in 0 until count) {
        val adl = adj[i]
        for (adn in adl) {
            val dest = adn.v
            g.addDirectedEdge(dest, i)
        }
    }
    return g
}
// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(1, 3)
    g.addDirectedEdge(3, 4)
    g.addDirectedEdge(4, 1)
    val gTransposed = g.transposeGraph()
    gTransposed.print()
}
```

Результат:

```
Vertex 0 is connected to :
Vertex 1 is connected to : 1(cost: 1) 1(cost: 1)
Vertex 2 is connected to : 2(cost: 1)
Vertex 3 is connected to : 3(cost: 1) 3(cost: 1)
Vertex 4 is connected to : 4(cost: 1)
```

Аналіз складності: *часова складність* під час реалізації на основі списку суміжності графа становить O(V+E), а для реалізації на основі матриці зв'язності – $O(V^2)$.

Перевірити, чи є неорієнтований граф зв'язним

Задача: дано неорієнтований граф. Визначити, чи є він зв'язним.

Розв'язок: почнемо з будь-якої вершини, якщо ми можемо відвідати всі інші вершини за допомогою DFS або BFS, то граф є зв'язним.

```
fun isConnectedUndirected(): Boolean {
    val visited = BooleanArray(count)
    dfsRecursive(0, visited)
    for (i in 0 until count) {
        if (!visited[i]) return false
    }
    return true
}
// Testing code.
fun main() {
    val g = Graph(6)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(1, 2)
    g.addUndirectedEdge(3, 4)
    g.addUndirectedEdge(4, 2)
    g.addUndirectedEdge(2, 5)
    println("isConnectedUndirected: " + g.isConnectedUndirected())
}
```

Результат:

```
isConnectedUndirected: true
```

Аналіз складності: *часова складність* така ж, як для DFS або BFS, і під час реалізації на основі списку суміжності графа становить O(V+E), а для реалізації на основі матриці зв'язності – $O(V^2)$.

Сильнозв'язний граф

Орієнтований граф називається сильнозв'язним, якщо для кожної пари вершин *u* та *v* існує шлях з *u* в *v* та шлях з *v* в *u*.

Щоб довести, що граф є зв'язним, потрібно перевірити ці дві умови, що повинні виконуватися для довільної вершини:

1. У кожну вершину можна потрапити з деякої вершини *u*, або кожна вершина досяжна з вершини *u*.

2. Вершина *u* досяжна з будь-якої іншої вершини.

Першу умову можна перевірити, виконавши DFS з деякої вершини *u*. Потрібно перевірити, що всі вершини графа відвідані. Другу умову можна перевірити, створивши спочатку транспонований граф *G'*. Потім виконати DFS над *G'* з вершини *u*. Якщо з вершини *u* у графі *G'* відвідані всі інші вершини, то це означає, що з вершини *u* можна потрапити у всі інші вершини графа *G'*. Це означає, що вершина *u* досяжна з усіх вершин вихідного графа *G*.

Алгоритм Косараджу

Знаходження зв'язності графа на основі DFS:

1. Створіть масив відвіданих вершин розміром *V* та ініціалізуйте всі вершини цього масиву значенням false.

2. Виберіть довільну вершину і виконайте DFS-обхід графа. Для всіх відвіданих вершин позначте їх відвіданими, установивши їх значення в масиві відвіданих як True.

3. Якщо в результаті DFS-обходу не всі вершини позначені як True, то повернути false.

4. Знайти транспонування або реверс графа.

5. Повторити кроки 1, 2 і 3 для оберненого графа.

6. Якщо DFS-обхід позначає всі вершини як True, то повернути true.

```
fun isStronglyConnected(): Boolean {
    val visited = BooleanArray(count)
    dfsRecursive(0, visited)
    for (i in 0..< count) {
        if (!visited[i]) {
            return false
        }
    }
    val gReversed = transposeGraph()
    for (i in 0..< count) {
        visited[i] = false
    }
    gReversed.dfsRecursive(0, visited)
    for (i in 0..< count) {
        if (!visited[i]) {
```

```
            return false
        }
    }
    return true
}

// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(1, 2)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(3, 0)
    g.addDirectedEdge(2, 4)
    g.addDirectedEdge(4, 2)
    println("IsStronglyConnected: " + g.isStronglyConnected())
}
```

Результат:

```
IsStronglyConnected: true
```

Часова складність: O(V+E) для дворазового обходу DFS.

Сильнозв'язні компоненти

Орієнтований граф може мати різні підграфи, які є сильнозв'язними. Такі підграфи називаються *сильнозв'язними компонентами*. У наведеному нижче графі весь граф не є сильнозв'язним, але два його підграфи є сильнозв'язними компонентами.

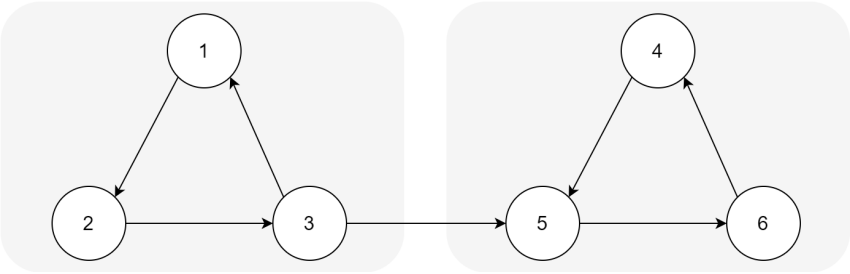


Рис. 9.18. Сильнозв'язні компоненти

Алгоритм пошуку сильнозв'язного компонента:

1. Створіть порожній стек, виконайте DFS-обхід графа *G*. Викличте DFS для суміжних вершин і виштовхніть їх у стек.

2. Транспонуйте граф *G*, щоб отримати новий граф *G'*.

3. Виконайте DFS-обхід графа *G'*, вибираючи вершини з вершини стека.

4. Кожен сильнозв'язний компонент обходиться за одну ітерацію.

```
fun stronglyConnectedComponent() {
    val visited = BooleanArray(count)
    val stk = Stack<Int>()
    for (i in 0..< count) {
        if (!visited[i])
            dfsUtil2(i, visited, stk)
    }
    val gReversed = transposeGraph()
    for (i in 0 until count) {
        visited[i] = false
    }
    val stk2 = Stack<Int>()
    while (!stk.isEmpty()) {
        val index = stk.pop()
        if (!visited[index]) {
            stk2.clear()
            gReversed.dfsUtil2(index, visited, stk2)
            println(stk2)
        }
    }
}
// Testing code.
fun main() {
    val g = Graph(7)
    g.addDirectedEdge(0, 1)
    g.addDirectedEdge(1, 2)
    g.addDirectedEdge(2, 0)
    g.addDirectedEdge(2, 3)
    g.addDirectedEdge(3, 4)
    g.addDirectedEdge(4, 5)
    g.addDirectedEdge(5, 3)
    g.addDirectedEdge(5, 6)
    g.stronglyConnectedComponent()
}
```

Результат:

```
[1, 2, 0]
[4, 5, 3]
[6]
```

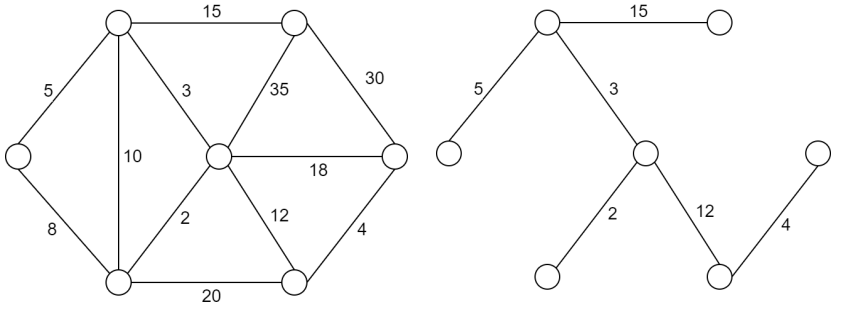
Часова складність: O(V+E) для дворазового обходу DFS.

Мінімальне остовне дерево (MST)

Остовним деревом графа *G* називається дерево, яке містить усі вершини графа.

Мінімальне остовне дерево – це остовне дерево, сума довжин / вагів ребер якого є мінімально можливою.

Наприклад, якщо ви хочете встановити зв'язок між кількома містами, то ви можете захотіти використати найменшу кількість дротів. MST можна використовувати для знаходження маршруту мережі та оцінки вартості кабелю.



Граф Мінімальне остовне дерево

Рис. 9.19. Граф і його мінімальне остовне дерево

Алгоритм Прима для MST

Алгоритм Прима вирощує дерево *T*, додаючи по одному ребру за раз, поки воно не стане остовним деревом. Ми ініціалізуємо *T* з нулем ребер, а *U* – з однією вершиною. Де *T* – множина ребер остовного дерева, а *U* – множина вершин остовного дерева.

На кожному кроці алгоритм Прима додає ребро з найменшим значенням, одна кінцева точка якого знаходиться в U, а інша – не в U. Оскільки кожне ребро додає одну нову вершину до U, то після n–1 додавання, U містить усі вершини остовного дерева і T стає остовним деревом.

Алгоритм Прима використовує пріоритетну чергу для отримання найближчої наступної вершини, алгоритм починається з додавання фіктивної вершини джерела до черги. Вершини з черги беруться по одній за раз. Потім ця вершина обробляється і вершини, які вона відвідала, додаються до черги. Черга є пріоритетною, тому вона завжди буде віддавати вершину з найменшою вагою. Якщо вершина, яка вискакує з черги, містить деяку нову вершину, то ця вершина позначається як відвідана, інакше ребро ігнорується.

```
fun primsMST() {
    val previous = IntArray(count){ -1 }
    val dist = IntArray(count){ Int.MAX_VALUE } // infinite
    val visited = BooleanArray(count)
    var source = 0
    dist[source] = 0
    previous[source] = source
    val queue = PriorityQueue<Edge>(100)
    var node = Edge(source, source, 0)
    queue.add(node)
    while (queue.isNotEmpty()) {
        node = queue.peek()
        queue.remove()
        visited[source] = true
        source = node.v
        val adl = adj[source]
        for (adn in adl) {
            val dest = adn.v
            val alt = adn.cost
            if (dist[dest] > alt && !visited[dest]) {
                dist[dest] = alt
                previous[dest] = source
                node = Edge(source, dest, alt)
                queue.add(node)
            }
        }
    }
    // printing result.
```

```
    var sum = 0
    var isMst = true
    var output = "Edges are "
    for (i in 0 until count) {
        if (dist[i] == Int.MAX_VALUE) {
            output += "($i, Unreachable) "
            isMst = false
        } else if (previous[i] != i) {
            output += "(" + previous[i] + "->" + i + " : " + dist[i] + ") "
            sum += dist[i]
        }
    }
    if (isMst) {
        println(output)
        println("Total MST cost: $sum")
    } else println("Can't get a Spanning Tree")
}

// Testing code.
fun main() {
    val g = Graph(9)
    g.addUndirectedEdge(0, 1, 4)
    g.addUndirectedEdge(0, 7, 8)
    g.addUndirectedEdge(1, 2, 8)
    g.addUndirectedEdge(1, 7, 11)
    g.addUndirectedEdge(2, 3, 7)
    g.addUndirectedEdge(2, 8, 2)
    g.addUndirectedEdge(2, 5, 4)
    g.addUndirectedEdge(3, 4, 9)
    g.addUndirectedEdge(3, 5, 14)
    g.addUndirectedEdge(4, 5, 10)
    g.addUndirectedEdge(5, 6, 2)
    g.addUndirectedEdge(6, 7, 1)
    g.addUndirectedEdge(6, 8, 6)
    g.addUndirectedEdge(7, 8, 7)
    g.primsMST()
}
```

Результат:

```
Edges are (0->1 : 4) (5->2 : 4) (2->3 : 7) (3->4 : 9) (6-
>5 : 2) (7->6 : 1) (0->7 : 8) (2->8 : 2)
Total MST cost: 37
```

Часова складність дорівнює O((E + V) log V), де V вершин та E ребер графа.

Алгоритм Крускала

Алгоритм Крускала багаторазово вибирає ребро з найменшою вагою, яке не утворює цикл. Відсортуйте ребра у неспадаючому порядку вартості: c (e1) ≤ c (e2) ≤ … ≤ c (em).

Установити *T* як порожнє дерево. Додавайте ребра до дерева по одному, так, щоб вони не утворювали цикл.

```
    class Sets internal constructor(var parent: Int, var rank: Int)
    // root element of set
    fun find(sets: Array<Sets?>, idx: Int): Int {
        var index = idx
        var p = sets[index]!!.parent
        while (p != index) {
            index = p
            p = sets[index]!!.parent
        }
        return index
    }
    // consider x and y are roots of sets.
    fun union(sets: Array<Sets?>, x: Int, y: Int) {
        if (sets[x]!!.rank < sets[y]!!.rank) sets[x]!!.parent = y
        else if (sets[y]!!.rank < sets[x]!!.rank) sets[y]!!.parent = x
        else {
            sets[x]!!.parent = y
            sets[y]!!.rank++
        }
    }
    fun kruskalMST() {
//Different subsets are created.
        val sets = arrayOfNulls<Sets>(count)
        for (i in 0 until count) sets[i] = Sets(i, 0)
// Edges are added to array and sorted.
        var E = 0
        val edge = arrayOfNulls<Edge>(100)
        for (i in 0 until count) {
            val ad = adj[i]
            for (adn in ad) {
                edge[E++] = adn
            }
        }
        Arrays.sort(edge, 0, E - 1)
        var sum = 0
        var output = "Edges are "
        for (i in 0 until E) {
            val x = find(sets, edge[i]!!.u)
            val y = find(sets, edge[i]!!.v)
```

```
            if (x != y) {
                output += "(" + edge[i]!!.u + "->" + edge[i]!!.v + " : " +
edge[i]!!.cost + ") "
                sum += edge[i]!!.cost
                union(sets, x, y)
            }
        }
        println(output)
        println("Total MST cost: $sum")
    }
}
```

Часова складність складає O(ElogV). Цикл виконується E разів, а функція find() займає log(V) часу для V вершин.

Ейлерів шлях і цикл

Ейлерів шлях – це шлях у графі, який відвідує кожне ребро рівно один раз.

Ейлерів цикл – це ейлерів шлях, який починається і закінчується в одній і тій же вершині. Або *Ейлеровим циклом* називається шлях у графі, який відвідує кожне ребро рівно один раз, і починається, і закінчується в одній і тій же вершині.

Задача: граф називається *Ейлеровим*, якщо в ньому існує Ейлерів цикл. Граф називається *напівейлеровим*, якщо у ньому існує Ейлерів шлях. Якщо у графі не існує жодного Ейлерова шляху, то він називається *неейлеровим.* Перевірити, чи є граф ейлеровим, напівейлеровим або неейлеровим.

Розв'язок: граф називається *Ейлеровим*, якщо всі його ребра мають парну кількість вершин. Граф називається *напівейлеровим*, якщо він має рівно дві вершини з непарною кількістю ребер або непарним степенем. У всіх інших випадках граф не є Ейлеровим.

```
fun isEulerian(): Int {

    var odd = 0 // Count odd degree
    val inDegree = IntArray(count)
    val outDegree = IntArray(count)
    var adl: LinkedList<Edge>
```

```
    for (i in 0..<count) {
        adl = adj[i]
        for (adn in adl) {
            outDegree[i] += 1
            inDegree[adn.v] += 1
        }
    }
    for (i in 0..<count) {
        if ((inDegree[i] + outDegree[i]) % 2 != 0) {
            odd += 1
        }
    }
    return when (odd) {
        0 -> {
            println("graph is Eulerian")
            2
        }
        2 -> {
            println("graph is Semi-Eulerian")
            1
        }
        else -> {
            println("graph is not Eulerian")
            0
        }
    }
}
// Testing code.
fun main() {
    val g = Graph(5)
    g.addDirectedEdge(1, 0)
    g.addDirectedEdge(0, 2)
    g.addDirectedEdge(2, 1)
    g.addDirectedEdge(0, 3)
    g.addDirectedEdge(3, 4)
    g.isEulerian()
    g.addDirectedEdge(4, 0)
    g.isEulerian()
}
```

Результат:

```
graph is Semi-Eulerian
graph is Eulerian
```

Алгоритми пошуку найкоротшого шляху у графі

Найкоротший шлях з одним джерелом для незваженого графа

Задача: дано незважений граф G = (V, E), де V – множина вершин, а E – множина ребер. Знайти найкоротший шлях від заданої початкової вершини ’s’ до всіх вершин V.

Розв’язок:

1. Спочатку початкова вершина додається в чергу.
2. Виконується обхід у ширину.
3. Вузли, які знаходяться ближче до джерела, обходяться першими й обробляються.
4. *Часова складність* буде O(V + E), загальна кількість разів, коли буде виконано внутрішній цикл, буде E, що дорівнює загальній кількості ребер у графі.

```
fun shortestPath(source: Int) {
    var curr: Int
    val distance = IntArray(count)
    val path = IntArray(count)
    for (i in 0..<count) {
        distance[i] = -1
        path[i] = -1
    }
    val que = LinkedList<Int>()
    que.add(source)
    distance[source] = 0
    path[source] = source
    while (!que.isEmpty()) {
        curr = que.remove()
        val adl = adj[curr]
        for (adn in adl) {
            if (distance[adn.v] == -1) {
                distance[adn.v] = distance[curr] + 1
                path[adn.v] = curr
                que.add(adn.v)
            }
        }
    }
    printPath(path, distance, count, source)
}
fun printPathRecursive(previous: IntArray, source: Int, dest: Int): String {
```

```
    var path = ""
    if (dest == source) path += source else {
        path += printPathRecursive(previous, source, previous[dest])
        path += "->$dest"
    }
    return path
}
fun printPath(previous: IntArray, dist: IntArray, count: Int, source: Int) {
    var output = "Shortest Paths: "
    for (i in 0..<count) {
        if (dist[i] == 99999) output += "($source->$i @ Unreachable) "
else if (i != previous[i]) {
            output += "("
            output += printPathRecursive(previous, source, i)
            output += " @ " + dist[i] + ") "
        }
    }
    println(output)
}
// Testing code.
fun main() {
    val g = Graph(9)
    g.addUndirectedEdge(0, 1)
    g.addUndirectedEdge(0, 7)
    g.addUndirectedEdge(1, 2)
    g.addUndirectedEdge(1, 7)
    g.addUndirectedEdge(2, 3)
    g.addUndirectedEdge(2, 8)
    g.addUndirectedEdge(2, 5)
    g.addUndirectedEdge(3, 4)
    g.addUndirectedEdge(3, 5)
    g.addUndirectedEdge(4, 5)
    g.addUndirectedEdge(5, 6)
    g.addUndirectedEdge(6, 7)
    g.addUndirectedEdge(6, 8)
    g.addUndirectedEdge(7, 8)
    g.shortestPath(0)
}
```

Результат:

```
Shortest Paths: (0->1 @ 1) (0->1->2 @ 2) (0->1->2->3 @ 3)
(0->1->2->3->4 @ 4) (0->1->2->5 @ 3) (0->7->6 @ 2) (0->7 @
1) (0->7->8 @ 2)
```

Алгоритм Дейкстри / Найкоротший шлях з єдиним джерелом за умови, що всі ребра графа мають додатну вагу

Задача: дано зважений граф G = (V, E), де V – множина вершин, а E – множина ребер, причому всі ребра E графа невід'ємні. Знайти найкоротший шлях із заданої початкової вершини у всі вершини V.

Розв'язок: алгоритм Дейкстри використовується для задачі про найкоротший шлях з одним джерелом для зважених ребер з невід'ємною вагою. Алгоритм Дейкстри схожий на алгоритм Прима. Він підтримує множину вершин, для яких відомий найкоротший шлях.

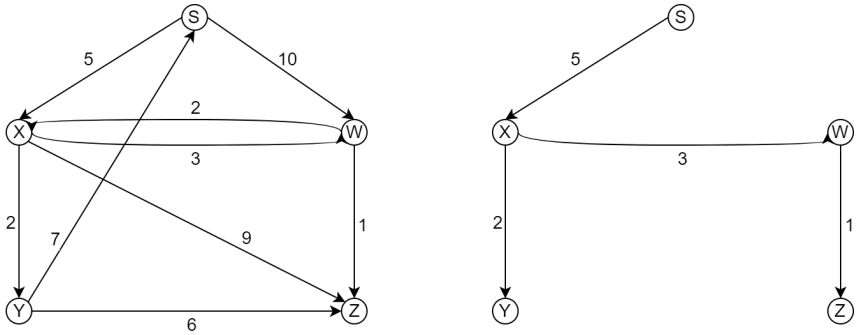


Рис. 9.20. Найкоротший шлях з єдиного джерела

Алгоритм починається з відстеження відстані між кожною вершиною та її батьками. На початку всі відстані дорівнюють нескінченності, оскільки ми не знаємо дійсного шляху до вершин, то батьки всіх вершин установлюються в нульове значення. Усі вершини додаються до черги з пріоритетом (реалізація Min-Heap).

На кожному кроці алгоритм бере одну вершину з черги пріоритетів (яка буде початковою вершиною на початку). Потім оновлюється список відстаней, що відповідає всім

суміжним вершинам. Коли черга спорожніє, ми матимемо відстань та повністю заповнений список батьківських вершин.

Примітка. Алгоритм Дейкстри не працює для графів з від'ємною вагою ребер. Алгоритм Дейкстри застосовується як до неорієнтованих, так і до орієнтованих графів.

Реалізація алгоритму Дейкстри на основі списків суміжності графа

```
fun dijkstra(source: Int) {
    val previous = IntArray(count)
    val dist = IntArray(count)
    val visited = BooleanArray(count)
    Arrays.fill(previous, -1)
    Arrays.fill(dist, 99999) // infinite
    dist[source] = 0
    previous[source] = source
    val queue = PriorityQueue<Edge>(100)
    var edge = Edge(source, source, 0)
    queue.add(edge)
    var curr: Int
    while (queue.isNotEmpty()) {
        edge = queue.peek()
        queue.remove()
        curr = edge.v
        visited[curr] = true
        val adl = adj[curr]
        for (adn in adl) {
            val dest = adn.v
            val alt = adn.cost + dist[curr]
            if (alt < dist[dest] && !visited[dest]) {
                dist[dest] = alt
                previous[dest] = curr
                edge = Edge(curr, dest, alt)
                queue.add(edge)
            }
        }
    }
    printPath(previous, dist, count, source)
}
// Testing code.
fun main() {
    val g = Graph(9)
    g.addUndirectedEdge(0, 1, 4)
    g.addUndirectedEdge(0, 7, 8)
```

```
    g.addUndirectedEdge(1, 2, 8)
    g.addUndirectedEdge(1, 7, 11)
    g.addUndirectedEdge(2, 3, 7)
    g.addUndirectedEdge(2, 8, 2)
    g.addUndirectedEdge(2, 5, 4)
    g.addUndirectedEdge(3, 4, 9)
    g.addUndirectedEdge(3, 5, 14)
    g.addUndirectedEdge(4, 5, 10)
    g.addUndirectedEdge(5, 6, 2)
    g.addUndirectedEdge(6, 7, 1)
    g.addUndirectedEdge(6, 8, 6)
    g.addUndirectedEdge(7, 8, 7)
    g.dijkstra(0)
}
```

Результат:

```
Shortest Paths: (0->1 @ 4) (0->1->2 @ 12) (0->1->2->3 @
19) (0->7->6->5->4 @ 21) (0->7->6->5 @ 11) (0->7->6 @ 9)
(0->7 @ 8) (0->1->2->8 @ 14)
```

Часова складність дорівнює O((E + V) log V), де V – кількість вершин та E – кількість ребер графа.

Реалізація алгоритму Дейкстри для матриці зв'язності графа

```
fun dijkstra2(source: Int) {
    val previous = IntArray(count){-1}
    val dist = IntArray(count){Int.MAX_VALUE}
    val visited = BooleanArray(count)
    dist[source] = 0
    previous[source] = source
    val queue = PriorityQueue<Edge>(100)
    var node = Edge(source, source, 0)
    queue.add(node)
    while (queue.isNotEmpty()) {
        node = queue.peek()
        queue.remove()
        val src = node.v
        visited[src] = true
        for (dest in 0..<count) {
            val cost = adj[src][dest].cost
            if (cost != 0) {
                val alt = cost + dist[src]
                if (dist[dest] > alt && !visited[dest]) {
                    dist[dest] = alt
```

```
                    previous[dest] = src
                    node = Edge(src, dest, alt)
                    queue.add(node)
                }
            }
        }
    }
    printPath(previous, dist, count, source)
}
```

Часова складність дорівнює $O(V^2)$, де V – кількість вершин графа.

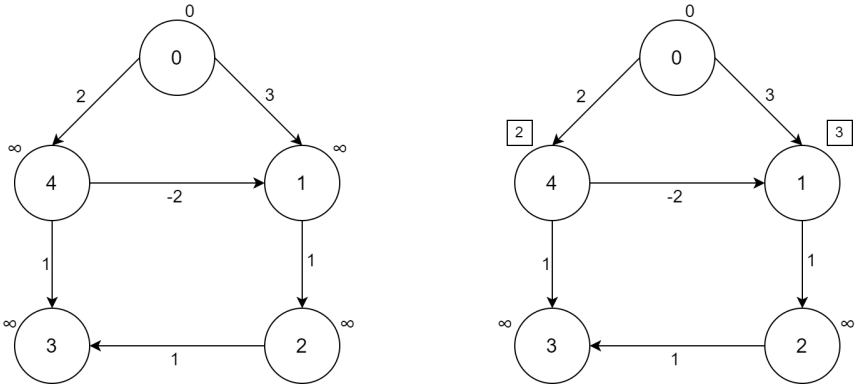
Алгоритм Беллмана-Форда / Найкоротший шлях з одного джерела у графі з ребрами, що можуть мати від'ємну вагу

Задача: дано зважений граф G = (V, E), де V – множина вершин, а E – множина ребер, причому ребра E графа можуть бути від'ємними, але граф не містить циклів від'ємної ваги. Знайти найкоротший шлях від заданої початкової вершини 's' до всіх вершин V.

Розв'язок: алгоритм Беллмана-Форда використовується для знаходження найкоротшого шляху з єдиного джерела до всіх інших вершин графа, що може містити від'ємне ребро, але без циклу з від'ємною вагою. Цикл з від'ємною вагою – це цикл, загальна вага якого від'ємна.

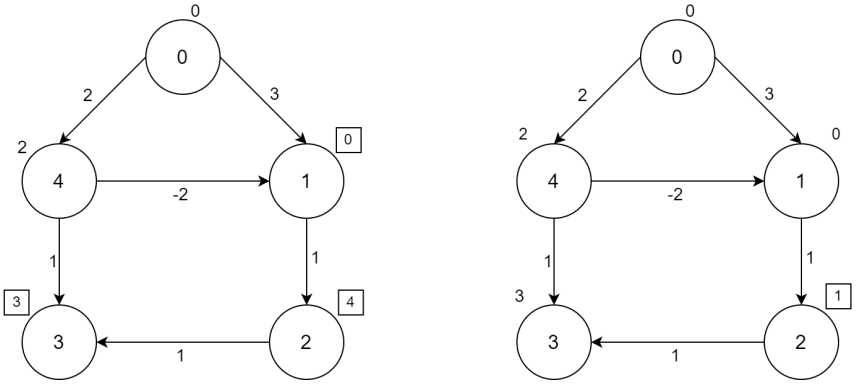
У цьому алгоритмі відстань до всіх вершин дорівнює ∞, а відстань до початкової вершини дорівнює 0. Потім виконується прохід V–1 (прохід, який також називається *релаксацією*) по всіх ребрах, і для кожної вершини оновлюється відстань до пункту призначення. Ми можемо зупинити алгоритм, якщо прохід / релаксація не змінює відстань до жодної з вершин. Якщо після V–1 проходу (релаксації) відбувається зміна ваги відстані, то у графі виникає від'ємний цикл.

Часова складність складає O(V*E), де V – кількість вершин, а E – загальна кількість ребер.



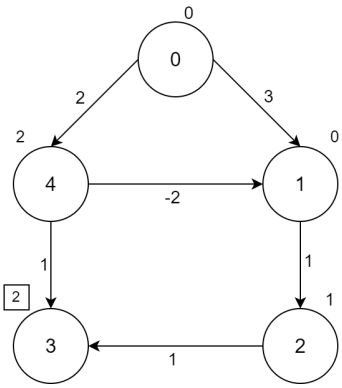
Довжина шляху дорівнює 0 Довжина шляху не більше –1

Рис. 9.21. Початкові кроки алгоритму Беллмана-Форда



Довжина шляху не перевищує 2 Довжина шляху не перевищує 3

Рис. 9.22. Початкові кроки алгоритму Беллмана-Форда



Довжина шляху не перевищує 4

Рис. 9.23. Початкові кроки алгоритму Беллмана-Форда

```
fun bellmanFordShortestPath(source: Int) {
    val distance = IntArray(count)
    Arrays.fill(distance, 99999) // infinite
    val path = IntArray(count)
    Arrays.fill(path, -1)
    distance[source] = 0
    path[source] = source
    // Зовнішній цикл буде виконуватись (V-1) разів.
    // Внутрішній цикл for та цикл while разом будуть
    // виконуватись Edges разів.
    // Що дає в результаті загальну складність як O(V*E)
    for (i in 0..<count - 1) {
        for (j in 0..<count) {
            val adl = adj[j]
            for (adn in adl) {
                val newDistance = distance[j] + adn.cost
                if (distance[adn.v] > newDistance) {
                    distance[adn.v] = newDistance
                    path[adn.v] = j
                }
            }
        }
    }
    printPath(path, distance, count, source)
}
// Testing code.
fun main() {
```

```
    val g = Graph(5)
    g.addDirectedEdge(0, 1, 3)
    g.addDirectedEdge(0, 4, 2)
    g.addDirectedEdge(1, 2, 1)
    g.addDirectedEdge(2, 3, 1)
    g.addDirectedEdge(4, 1, -2)
    g.addDirectedEdge(4, 3, 1)
    g.bellmanFordShortestPath(0)
}
```

Результат:

```
Shortest Paths: (0->4->1 @ 0) (0->4->1->2 @ 1) (0->4->1-
>2->3 @ 2) (0->4 @ 2)
```

Найкоротші шляхи для всіх пар вершин / Алгоритм Флойда-Уоршалла

Задача: дано зважений граф G = (V, E), де V – множина вершин, а E – множина ребер, причому ребра E графа можуть бути від'ємними. Знайти найкоротший шлях між усіма парами вершин *u*, *v* є V.

Розв'язок: алгоритм Флойда-Уоршалла можна використовувати для знаходження найкоротшого шляху між усіма парами вершин графа, якщо граф не містить циклу з від'ємною вагою. Якщо граф містить цикл з від'ємною вагою, то він також повідомляє про від'ємний цикл. Програма порівнює всі можливі шляхи між парами вершин графа. Створіть вихідну матрицю, аналогічну заданій матриці вартості графа. Після цього вихідна матриця буде доповнена всіма вершинами як проміжною вершиною. Коли ми вибрали вершину k як проміжну, ми вже вибрали всі попередні вершини від 0 до k–1 як проміжні вершини.

$Dist^{K}[i, j] = minimum(Dist^{K-1}[i, j], Dist^{K-1}[i, k] + Dist^{K-1}[k, j])$. Усі діагоналі матриці відстаней dist[i][i] на початку дорівнюють 0. Якщо ми знайдемо кращий шлях від i до i, це буде означати дві речі: що існує цикл, і цикл має від'ємну вагу.

Часова складність цього алгоритму становить $O(V^3)$, де V – кількість вершин у графі.

```
    fun floydWarshall() {
        val V = count
        val dist = Array(V) { IntArray(V) }
        val path = Array(V) { IntArray(V) }
        val INF = 99999
        for (i in 0..<V) {
            for (j in 0..<V) {
                dist[i][j] = INF
                if (i == j) path[i][j] = 0 else path[i][j] = -1
            }
        }
        for (i in 0..<V) {
            val adl = adj[i]
            for (adn in adl) {
                path[adn.u][adn.v] = adn.u
                dist[adn.u][adn.v] = adn.cost
            }
        }
        for (k in 0..<V) { // Pick intermediate vertices.
            for (i in 0..<V) { // Pick source vertices one by one.
                for (j in 0..<V) { // Pick destination vertices.
// If we have a shorter path from i to j via k.
// then update dist[i][j] and and path[i][j]
                    if (dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j]
                        path[i][j] = path[k][j]
                    }
                }
// dist[i][i] is 0 in the start.
// If there is a better path from i to i and is better path then
// we have a -ve cycle.
                if (dist[i][i] < 0) {
                    println("Negative-weight cycle found.")
                    return
                }
            }
        }
        printSolution(dist, path, V)
    }
    private fun printSolution(cost: Array<IntArray>, path: Array<IntArray>, V: Int) {
        print("Shortest Paths : ")
        for (u in 0..<V) {
            for (v in 0..<V) {
                if (u != v && path[u][v] != -1) {
                    print("(")
                    printPath2(path, u, v)
                    print(" @ " + cost[u][v] + ") ")
```

```
                }
            }
        }
        println()
    }
    private fun printPath2(path: Array<IntArray>, u: Int, v: Int) {
        if (path[u][v] == u) {
            print("$u->$v")
            return
        }
        printPath2(path, u, path[u][v])
        print("->$v")
    }
// Testing code.
fun main() {
    val g = Graph(4)
    g.addDirectedEdge(0, 0, 0)
    g.addDirectedEdge(1, 1, 0)
    g.addDirectedEdge(2, 2, 0)
    g.addDirectedEdge(3, 3, 0)
    g.addDirectedEdge(0, 1, 5)
    g.addDirectedEdge(0, 3, 10)
    g.addDirectedEdge(1, 2, 3)
    g.addDirectedEdge(2, 3, 1)
    g.floydWarshall()
}
```

Результат:

```
Shortest Paths : (0->1 @ 5) (0->1->2 @ 8) (0->1->2->3 @ 9)
(1->2 @ 3) (1->2->3 @ 4) (2->3 @ 1)
```

Гамільтонів шлях

Гамільтонів шлях – це шлях, на якому кожна вершина відвідується рівно один раз без повторень, він не обов'язково повинен починатися і закінчуватися в одній і тій же вершині.

Гамільтонів шлях є *NP*-повною задачею, тому єдиний розв'язок можливий за допомогою зворотного ходу (Backtracking), який починається з вершини *s* і рекурсивно перебирає всі сусідні вершини. Якщо ми не знаходимо шлях, то повертаємося назад і пробуємо інші вершини.

Розв'язок: підхід Backtracking, знаходження всіх можливих шляхів, які задовольняють умовам гамільтонового шляху.

У гамільтоновому шляху вершина, яка додається, не повинна додаватися знову. Для зберігання path[] буде створено масив path[]. А масив added[] використовується для відстеження елементів, уже доданих до шляху. Якщо ми знайдемо шлях, довжина якого дорівнює кількості вершин, то отримаємо розв'язок.

Складність за часом: O(N!), треба пройти всі можливі шляхи, тому загальна кількість шляхів буде N!

```
fun hamiltonianPathRecursive(path: IntArray, pSize: Int, added: IntArray): Boolean {
    // Вихід з рекурсії, якщо повний шлях знайдено
    if (pSize == vertexCount) {
        return true
    }
    for (vertex in 0..<vertexCount) {
        // Існує ребро від останнього елемента шляху до наступної вершини
        // та наступна вершина поки не входить до шляху.
        if ((pSize == 0 || adj[path[pSize - 1]][vertex] == 1) && added[vertex] == 0) {
            path[pSize] = vertex
            added[vertex] = 1
            if (hamiltonianPathRecursive(path, pSize + 1, added))
                return true // повернення – backtracking
            added[vertex] = 0
        }
    }
    return false
}

fun hamiltonianPath(): Boolean {
    val path = IntArray(vertexCount)
    val added = IntArray(vertexCount)
    if (hamiltonianPathRecursive(path, 0, added)) {
        print("Hamiltonian Path found : ")
        for (i in 0..<vertexCount) print(" " + path[i])
        println()
        return true
    }
    println("Hamiltonian Path not found")
    return false
}
// Testing code.
fun main() {
    val count = 5
    val g = GraphAM(count)
    val adj = arrayOf(
```

```
        intArrayOf(0, 1, 0, 1, 0),
        intArrayOf(1, 0, 1, 1, 0),
        intArrayOf(0, 1, 0, 0, 1),
        intArrayOf(1, 1, 0, 0, 1),
        intArrayOf(0, 1, 1, 1, 0)
    )
    for (i in 0 until count) for (j in 0 until count) if (adj[i][j] == 1)
        g.addDirectedEdge(i, j, 1)
    println("Hamiltonian Path : " + g.hamiltonianPath())
    val g2 = GraphAM(count)
    val adj2 = arrayOf(
        intArrayOf(0, 1, 0, 1, 0),
        intArrayOf(1, 0, 1, 1, 0),
        intArrayOf(0, 1, 0, 0, 1),
        intArrayOf(1, 1, 0, 0, 0),
        intArrayOf(0, 1, 1, 0, 0)
    )
    for (i in 0 until count) for (j in 0 until count) if (adj2[i][j] == 1)
        g2.addDirectedEdge(i, j, 1)
    println("Hamiltonian Path : " + g2.hamiltonianPath())
}
```

Результат:

```
Hamiltonian Path found : 0 1 2 4 3
Hamiltonian Path : true
Hamiltonian Path found : 0 3 1 2 4
Hamiltonian Path : true
```

Гамільтонів цикл (Гамільтонів ланцюг)

Гамільтоновим циклом називається такий гамільтонів шлях, який має ребро, що веде з його останньої вершини до першої.

Гамільтонів цикл – це цикл, який відвідує кожну вершину рівно один раз, і він повинен починатись і закінчуватись в одній і тій же вершині.

Розв'язок: методом зворотного перебору знайдіть усі можливі шляхи, які задовольняють умовам гамільтонового шляху, а також шлях повинен задовольняти ще одній умові – з останньої вершини гамільтонового шляху повинен існувати шлях до першого елемента. У гамільтоновому шляху не можна додавати вершину, яку було додано, повторно. Для зберігання

path[] буде створено масив path[]. А масив added[] використовується для відстеження елементів, уже доданих до шляху. Якщо ми знайдемо шлях, довжина якого дорівнює кількості вершин, і існує шлях від останньої вершини до першої, то ми маємо розв'язок.

Часова складність: O(n!).

```
fun hamiltonianCycleRecursive(path: IntArray, pSize: Int, added: IntArray): Boolean {
    if (pSize == vertexCount) {
        return if (adj[path[pSize - 1]][path[0]] == 1) {
            path[pSize] = path[0]
            true
        } else false
    }
    for (vertex in 0..< vertexCount) {

        if (pSize == 0 || adj[path[pSize - 1]][vertex] == 1 &&
            added[vertex] == 0)
        {
            path[pSize] = vertex
            added[vertex] = 1
            if (hamiltonianCycleRecursive(path, pSize + 1, added)) return true
            // backtracking
            added[vertex] = 0
        }
    }
    return false
}
fun hamiltonianCycle(): Boolean {
    val path = IntArray(vertexCount + 1)
    val added = IntArray(vertexCount)
    if (hamiltonianCycleRecursive(path, 0, added)) {
        print("Hamiltonian Cycle found : ")
        for (i in 0..vertexCount) print(" " + path[i])
        println("")
        return true
    }
    println("Hamiltonian Cycle not found")
    return false
}

// Testing code.
fun main() {
    val count = 5
    val g = GraphAM(count)
    val adj = arrayOf(
```

```
        intArrayOf(0, 1, 0, 1, 0),
        intArrayOf(1, 0, 1, 1, 0),
        intArrayOf(0, 1, 0, 0, 1),
        intArrayOf(1, 1, 0, 0, 1),
        intArrayOf(0, 1, 1, 1, 0)
    )
    for (i in 0..< count)
        for (j in 0..< count)
            if (adj[i][j] == 1)
                g.addDirectedEdge(i, j, 1)
    println("HamiltonianCycle : " + g.hamiltonianCycle())
    val g2 = GraphAM(count)
    val adj2 = arrayOf(
        intArrayOf(0, 1, 0, 1, 0),
        intArrayOf(1, 0, 1, 1, 0),
        intArrayOf(0, 1, 0, 0, 1),
        intArrayOf(1, 1, 0, 0, 0),
        intArrayOf(0, 1, 1, 0, 0)
    )
    for (i in 0..< count)
        for (j in 0..< count)
            if (adj2[i][j] == 1)
                g2.addDirectedEdge(i, j, 1)
    println("HamiltonianCycle : " + g2.hamiltonianCycle())
}
```

Результат:

```
Hamiltonian Cycle found : 0 1 2 4 3 0
HamiltonianCycle : true
Hamiltonian Cycle not found
HamiltonianCycle : false
```

Задача комівояжера (**TSP**)

Задача: у задачі комівояжера намагаються знайти найкоротший маршрут через задану множину з *n* міст, який відвідує кожне місто рівно один раз, а потім повертається до міста, з якого він почав.

Або дано зважений зв'язний граф G = (V, E), де V – це вершини, а E – ребра. Оскільки граф повністю зв'язний, то існує декілька гамільтонових циклів. Нам потрібно знайти найкоротший гамільтонів цикл у графі. Цикл, який проходить через усі вершини графа рівно один раз.

Це *NP*-повна задача, і не існує ефективного алгоритму для її розв'язку. Навіть якщо деякий шлях є розв'язком, так само важко перевірити, чи це правильний розв'язок, чи ні.

```
fun tspRecursive(graph: Array<IntArray>, n: Int, path: IntArray, pSize: Int,
                 pCost: Int, visited: BooleanArray, ansIn: Int, ansPath:
IntArray): Int {
    var ans = ansIn
    if (pCost > ans) return ans
    val curr = path[pSize - 1]
    if (pSize == n) {
        if (graph[curr][0] > 0 && ans > pCost + graph[curr][0]) {
            ans = pCost + graph[curr][0]
            for (i in 0..n) ansPath[i] = path[i % n]
        }
        return ans
    }
    for (i in 0 until n) {
        if (!visited[i] && graph[curr][i] > 0) {
            visited[i] = true
            path[pSize] = i
            ans = tspRecursive(graph, n, path, pSize + 1, pCost + graph[curr][i],
                visited, ans, ansPath)
            visited[i] = false
        }
    }
    return ans
}
fun tsp(graph: Array<IntArray>, n: Int): Int {
    val visited = BooleanArray(n)
    val path = IntArray(n)
    val ansPath = IntArray(n + 1)
    path[0] = 0
    visited[0] = true
    var ans = Int.MAX_VALUE
    ans = tspRecursive(graph, n, path, 1, 0, visited, ans, ansPath)
    println("Path length : $ans")
    print("Path : ")
    for (i in 0..n) print(ansPath[i].toString() + " ")
    return ans
}
// Testing code.
fun main() {
    val n = 4
    val graph = arrayOf(
        intArrayOf(0, 10, 15, 20),
        intArrayOf(10, 0, 35, 25),
```

```
        intArrayOf(15, 35, 0, 30),
        intArrayOf(20, 25, 30, 0))
    tsp(graph, n)
}
```

Результат:

```
Path length : 80
Path : 0 1 3 2 0
```

Використання графових алгоритмів

Найбільш відомі такі застосування графових алгоритмів:

1. Представлення карт за допомогою графів.

2. *GPS*-навігація використовує алгоритми пошуку найкоротшого шляху, такі як алгоритм Дейкстри.

3. Соціальні мережі, такі як *Facebook*, *Instagram* тощо, використовують графи для представлення зв'язків між профілями.

4. Алгоритми пошуку шляхів використовуються в робототехніці, відеоіграх тощо.

5. Алгоритми остовних дерев використовуються для створення мереж зв'язку на основі графів вузлів.

6. Пошук шляху між двома вершинами може бути виконаний за допомогою BFS або DFS.

7. За заданою початковою вершиною *s* знаходження мінімальної кількості ребер з вершини *s* до всіх інших вершин графа здійснюється за допомогою BFS.

8. Перевірка того, чи є граф *G* зв'язним, здійснюється за допомогою BFS або DFS.

9. Пошук циклу у графі або перевірка того, чи є заданий граф деревом, виконується за допомогою DFS.

10. Топологічне сортування здійснюється за допомогою DFS.

11. Сильнозв'язані компоненти або підграфи в орієнтованих графах виконуються за допомогою DFS.

СПИСОК ЛІТЕРАТУРИ

1. Беркунський Є. Ю., Павленко А. Ю. Алгоритмізація та програмування мовами Kotlin, C/C++ : навч. посіб. Миколаїв : НУК, 2022. 256 с.

2. Грудзинський Ю. Є. Алгоритми та структури даних : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології». Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.

3. Грудзинський Ю. Є. Алгоритми та структури даних (комп'ютерний практикум) : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології». Київ : КПІ ім. Ігоря Сікорського, 2022. 100 с.

4. Коротєєва Т. О. Алгоритми та структури даних : навч. посіб. Львів : Видавництво Львівської політехніки, 2014. 280 с.

5. Михелєв І. Л., Слободян С. О. Основи інформаційних технологій : навч. посіб. Миколаїв : НУК, 2014. 164 с.

6. O. Campesato. Data Structures in Java. Mercury Learning and Information LLC, 2023. 231 p.

7. Marcin Moskała. Kotlin Essentials. Leanpub, 2022. 309 p.

8. Ted Hagos. Beginning Kotlin: Build Applications with Better Code, Productivity, and Performance. Apress, 2023. 230 p.

9. Márton Braun, Irina Galata, Matei Suica. Data Structures & Algorithms in Kotlin. Razeware LLC, 2021. 422 p.

ДЛЯ НОТАТОК

Навчальне видання

БЕРКУНСЬКИЙ Євген Юрійович
ПАВЛЕНКО Альона Юріївна

СТРУКТУРИ ТА ОРГАНІЗАЦІЯ ДАНИХ МОВОЮ KOTLIN

Навчальний посібник

Комп'ютерна правка й коректура *О. Г. Костенко*
Комп'ютерне верстання *Ю. С. Семенченко*

Формат 60×84/16. Ум. друк. арк. 15,23. Тираж 100 прим. Вид. № 7. Зам. № 1003-13.
Видавець і виготівник Національний університет кораблебудування
імені адмірала Макарова
просп. Героїв України, 9, м. Миколаїв, 54007
E-mail : publishing@nuos.edu.ua
Свідоцтво суб'єкта видавничої справи ДК № 6402 від 19.09.2018 р.