МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

Є. Ю. БЕРКУНСЬКИЙ, А. Ю. ПАВЛЕНКО

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт з дисципліни «Структури та організація даних»

Рекомендовано Методичною радою НУК

ВИДАВНИЦТВО
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КОРАБЛЕБУДУВАННЯ
ІМ. АДМІРАЛА МАКАРОВА

2026

УДК 004.451.5(076)
Б48

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

Рецензент О. В. Гайдаєнко, кандидат технічних наук, доцент

Рекомендовано Методичною радою НУК імені адмірала Макарова

Беркунський Є. Ю.

Б48 Методичні вказівки до виконання лабораторних робіт з дисципліни «Структури та організація даних» / Є. Ю. Беркунський, А. Ю. Павленко. – Миколаїв : НУК, 2026. – 61 с.

Наведено короткі теоретичні відомості та довідкові матеріали, необхідні для виконання завдань лабораторних робіт з дисципліни «Структури та організація даних», а також сформульовано завдання й контрольні питання до відповідних лабораторних робіт; ці вказівки є доповненням до навчального посібника «Структури та організація даних мовою Kotlin» авторів Євгена Беркунського та Альони Павленко.

Призначено для підготовки студентів першого (бакалаврського) рівня вищої освіти зі спеціальностей галузі знань F – Інформаційні технології: F3 – Комп'ютерні науки, F4 – Системний аналіз та наука про дані, F6 – Інформаційні системи і технології.

УДК 004.451.5(076)

© Є. Ю. Беркунський, А. Ю. Павленко, 2026
© Національний університет кораблебудування
імені адмірала Макарова, 2026

ЗМІСТ

ВСТУП.....4

Лабораторна робота № 1. Лінійні масиви. Пошук. Сортування.....6

Лабораторна робота № 2. Лінійний зв'язаний список.....11

Лабораторна робота № 3. Стек, Черга, Дек.....17

Лабораторна робота № 4. Дерева. Бінарне дерево пошуку.....29

Лабораторна робота № 5. Хеш-таблиці. Черги з пріоритетами.....39

Лабораторна робота № 6. Графи.....46

СПИСОК ЛІТЕРАТУРИ.....60

ВСТУП

Методичні вказівки призначені для організації виконання лабораторних робіт з дисципліни «Структури та організація даних» студентами першого (бакалаврського) рівня вищої освіти за спеціальностями галузі знань F – Інформаційні технології (зокрема F3 – Комп'ютерні науки, F4 – Системний аналіз та наука про дані, F6 – Інформаційні системи і технології). Вони спрямовані на систематичне опанування фундаментальних понять абстрактних структур даних і практичних навичок реалізації та аналізу, що є необхідною основою для подальшого професійного розвитку в галузі розробки програмного забезпечення та наукових досліджень у сфері обробки даних. Особлива увага приділена формуванню алгоритмічного мислення, умінню обґрунтовано вибирати структури даних для конкретних задач і оцінювати ефективність реалізованих рішень за часовими та просторовими характеристиками.

У методичних вказівках поєднано теоретичні відомості з практичними завданнями, методичними рекомендаціями щодо реалізації та критеріями оцінювання результатів. Навчальний матеріал охоплює організацію лінійних масивів, алгоритми пошуку і сортування, реалізацію зв'язних списків, принципи роботи стеків, черг і двосторонніх черг, побудову деревних структур з акцентом на бінарні дерева пошуку та питання їхнього балансування, організацію хеш-таблиць

і реалізацію пріоритетних черг на основі купи, а також представлення графів і застосування розглянутих структур даних у алгоритмах на графах. Методичні вказівки складаються з шести лабораторних робіт, кожна з яких містить стислий теоретичний огляд теми, завдання для програмної реалізації та тестування, методичні вказівки щодо вибору представлення даних і структури програми, а також вимоги до оформлення звітної документації.

Виконання лабораторних робіт передбачає реалізацію відповідних структур даних, проведення експериментальних досліджень на наборах даних різного обсягу й складності, фіксацію результатів вимірювань продуктивності та аналітичне обґрунтування вибору підходів у звітній частині. Очікується, що опанування матеріалу забезпечить здатність пояснювати відмінності між абстрактними типами даних і їх конкретними реалізаціями, реалізовувати базові структури й алгоритми, оцінювати їхню складність і практичну ефективність, а також інтегрувати кілька структур даних у складні алгоритмічні рішення, що має безпосереднє застосування в навчальній, науковій і професійній діяльності.

Методичні вказівки є доповненням до навчального посібника «Структури та організація даних мовою Kotlin» авторів Євгена Беркунського та Альони Павленко і призначені для лабораторних занять та самостійної роботи студентів.

Лабораторна робота № 1
Лінійні масиви. Пошук. Сортування

Мета роботи: Отримати практичні навички в розробленні програм, що використовують сортування й пошук, та аналізувати їхню ефективність.

Короткі теоретичні відомості

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

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 розроблена техніка зовнішнього сортування.

BubbleSort

BubbleSort – це найповільніший алгоритм сортування, який використовується, коли набір даних невеликий. Його легко реалізувати і використовувати. У BubbleSort ми порівнюємо кожну пару суміжних значень.

Ми хочемо відсортувати значення у 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 в крайньому правому місці. Після N проходів отримаємо повне сортування масиву.

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.

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.

Докладна інформація по темі наведена в посібнику [1, с. 23–41].

Завдання

Завдання 1.1

1. Сформуйте масив з 10^5 випадкових цілих чисел з діапазону [0, 10^5+N*1000) (N – номер варіанта за списком групи).
2. Порахуйте кількість чисел у заданому масиві, що потрапляють у діапазон [N,2*N]

3. Визначте найменший індекс елемента масиву, що дорівнює 1000+N. Якщо масив не містить такого елемента, вивести –1.

4. Згідно з варіантом, розробіть програму, що виконує сортування створеного масиву. Метод сортування обрати згідно з таблицею. Виміряйте час роботи алгоритму сортування.

Варіант	***Метод сортування***
1, 4, 7, 10, 13, 16, 19, 22, 25, 28	Метод вставки
2, 5, 8, 11, 14, 17, 20, 23, 26, 29	Метод обмінів
3, 6, 9, 12, 15, 18, 21, 24, 27, 30	Метод вибору

5. Згенеруйте масив аналогічно до п. 1, збільшивши кількість елементів масиву у 2 рази. Відсортуйте новий масив методом з п. 4. Порівняйте час роботи сортування для двох масивів.

6. Повторіть завдання п. 3 для відсортованого масиву, використовуючи алгоритм бінарного пошуку. Порівняйте кількість операцій порівнянь, які необхідно виконати для пошуку елемента у впорядкованому та невпорядкованому масиві.

7. Порівняйте час, що потрібний для сортування різними способами і час, за який виконується сортування стандартним методом sort класу IntArray.

8. Проаналізуйте метод пошуку максимального елемента методом «Розділяй і володарюй» – яка складність його буде?

Пояснення до виконання

Якщо завдання виконується мовою Java, для генерування масиву випадкових чисел можна скористатись методами генерації випадкових чисел класу Random:

`Random rnd = new Random();` – створення генератору випадкових чисел

`rnd.nextInt(bound);` – випадкове ціле число з діапазону [0, bound)

`rnd.ints(N, origin, bound).toArray();` – масив з N випадкових цілих чисел з діапазону [origin, bound)

Для вимірювання часу роботи програми можна скористатися методом currentTimeMillis() класу System. Це можна зробити за такою схемою:

```
long start = System.currentTimeMillis();
// обчислювальний процес, час роботи якого треба виміряти
long finish = System.currentTimeMillis();
System.out.println(«Time = " + (finish-start));
```

Якщо завдання виконується мовою C++, для генерування випадкових чисел доцільно використовувати описані в бібліотеці <random> механізми. Наприклад, так:

```
// створюємо Генератор Псевдовипадкових Чисел та ініціалізуємо його значенням time(0)
std::mt19937 gen(time(0));
// створюємо розподіл uid та ініціалізуємо його початковими значеннями
std::uniform_int_distribution<int> uid(0, 100_000);
// наша змінна отримує значення випадкового числа з указаного розподілу
int randomNumber = uid(gen)
```

Використання функцій rand(), srand() не рекомендується, оскільки вони дають погану різноманітність даних та невеликий діапазон значень.

Якщо завдання виконується мовою Kotlin, можна скористатись наведеним у цьому проєкті шаблоном програми.

Для вимірювання часу роботи програми можна скористатися методом currentTimeMillis() класу System. Це можна зробити за такою схемою:

```
val start = System.currentTimeMillis()
// обчислювальний процес, час роботи якого треба виміряти
val finish = System.currentTimeMillis()
println(«Time = ${finish - start}")
```

Пошук максимуму методом «Розділяй і володарюй»:

```
fun findMaxBinary(array: IntArray) : Int? {
    return if (array.isEmpty()) {
        null
    } else findMax(array, 0, array.size - 1)
}

fun findMax(array: IntArray, low: Int, high: Int): Int {
    if (low == high) {
        return array[low]
    } else {
        val mid = (low + high) / 2
        val leftMax = findMax(array, low, mid)
        val rightMax = findMax(array, mid + 1, high)
        return maxOf(leftMax, rightMax)
    }
}
```

Контрольні питання

1. Що таке сортування та яку роль воно відіграє у пошуку даних?
2. Чим відрізняється внутрішнє сортування від зовнішнього?
3. У чому полягає принцип роботи алгоритму BubbleSort?
4. Яка часова складність BubbleSort у найгіршому випадку?
5. Як працює алгоритм InsertionSort і чому його порівнюють із впорядкуванням карт?
6. Яка різниця між найкращим і найгіршим випадком для InsertionSort?
7. Чому SelectionSort виконує менше обмінів порівняно з BubbleSort?
8. Чи є SelectionSort стабільним алгоритмом сортування?
9. Як працює QuickSort і яку роль відіграє вибір опорного елемента?
10. Чому QuickSort може мати часову складність $O(n^2)$ у найгіршому випадку?

Лабораторна робота № 2
Лінійний зв'язаний список

Мета роботи: Отримати практичні навички в розробленні програм, що використовують лінійні зв'язані списки.

Короткі теоретичні відомості

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. 2.1. Вузол LinkedList

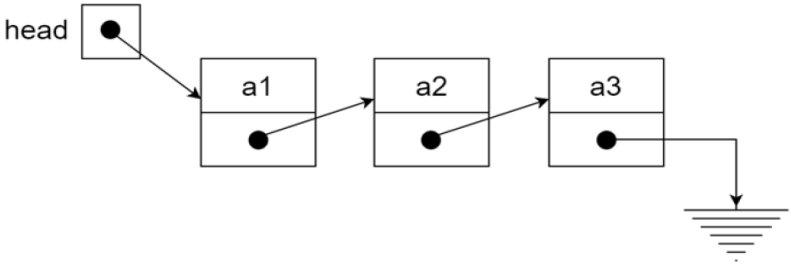


Рис. 2.2. Зв'язаний список – LinkedList

Важливі складові частини зв'язаного списку

- Head: голова – це покажчик, який містить адресу першого вузла у зв'язаному списку.
- Nodes: елементи у зв'язаному списку називаються вузлами.
- Value: дані, які зберігаються в кожному вузлі пов'язаного списку.
- 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. Найчастіше використовуються такі види списків: однозв'язний список, двозв'язний список, круговий зв'язаний список 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**. В однозв'язному списку можливо рухатися лише в одному напрямку, оскільки кожен вузол має лише посилання на наступний вузол, як показано на рисунку вище.*

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

1. Додати елемент у список.
2. Виконати перегляд усіх елементів списку.
3. Пошук елемента в списку.
4. Видалення елемента зі списку.
5. Інвертування (перевертання) зв'язаного списку.

Додавання елемента в зв'язаний список

Елемент можна вставляти у зв'язаний список у різні місця. Приклади найбільш поширених випадків:

1. Додавання елемента в початок зв'язаного списку;
2. Додавання елемента в кінець зв'язаного списку;
3. Додавання елемента на N-у позицію у зв'язаному списку;
4. Додавання елемента у відсортований зв'язаний список зі збереженням властивості впорядкування.

Перегляд усіх елементів списку та пошук елемента

Перегляд елементів та пошук у зв'язаному списку зручно виконувати за допомогою циклічних операцій з вказівниками.

Додавання елемента у зв'язаний список

Додавання елемента на N-у позицію у зв'язаному списку чи вставка у відсортований список зі збереженням упорядкування виконується у декілька кроків: спочатку треба знайти місце для вставки, а потім, залежно від знайденої позиції, виконати вставку. Важливо врахувати часткові випадки: знайдена позиція – на початку списку, знайдена позиція в середині списку та знайдена позиція у кінці списку. Для кожного з них треба реалізувати відповідні алгоритми.

Видалення елемента зі списку

Як і для вставки, для видалення необхідно спочатку знайти потрібний елемент (за номером, або за значенням) – отримати посилання на нього. А після цього врахувати один з трьох варіантів: знайдений елемент – перший у списку, знайдений елемент – останній, та загальний випадок – знайдений елемент у середині списку. Для кожного з них треба реалізувати відповідні алгоритми.

Двозв'язний список

Двозв'язний список також складається з вузлів. Але тепер кожен вузол складається з трьох частин: перша частина – це дані, а друга та третя частини – це посилання на наступний та попередній вузли у зв'язаному списку. Посилальна частина першого та останнього вузлів містить значення null (для першого вузла немає попереднього, а в останнього немає наступного). Початкове посилання списку називається «head» і вказує на перший вузол зв'язаного списку, кінцеве посилання називається «tail» і містить посилання на останній вузол списку. Робота з таким списком відбувається аналогічно до однозв'язного списку, але при реалізації кожної операції додавання чи видалення елемента треба враховувати наявність двох посилань (на попередній та наступний вузли).

Примітка. *Завдяки наявності двох посилань у кожному вузлі у двозв'язному списку можливо рухатися по ньому в обох напрямках.*

Докладна інформація по темі наведена в посібнику [1, с. 42–61].

Завдання

Завдання 2.1

Опишіть структури даних, необхідних для реалізації списку для виконання операцій згідно з варіантом. Реалізуйте вказані операції. Списки заповнювати тестовими даними, що читаються з текстових файлів. Файли та їхню структуру розробити самостійно.

Варіант	***Завдання***
1	Реалізувати однозв'язний список з базовими операціями: додавання в голову, додавання в кінець, видалення елемента, пошук за значенням.
2	Реалізувати функцію реверсу списку: змінити порядок елементів на протилежний.
3	Знайти середній елемент списку: без використання додаткових структур.
4	Об'єднати два однозв'язні списки в один: спочатку всі елементи першого списку, потім – усі елементи другого списку.
5	Об'єднати два однозв'язні списки в один: спочатку елементи обох списків по черзі, після закінчення одного списку – усі елементи іншого, що залишились.
6	Видалити всі вузли з певним указаним значенням: видалити всі вузли зі значенням x, що вводиться з клавіатури.
7	Реалізувати сортування списку: наприклад, методом бульбашки (BubbleSort).
8	Реалізувати множинне додавання елементів у список: з масиву або іншого списку.
9	Реалізувати функцію пошуку *n*-го елемента з кінця списку.
10	Реалізувати злиття двох відсортованих списків в один відсортований.
11	Реалізувати двозв'язаний список з операціями вставки, видалення, пошуку.
12	Реалізувати двосторонню ітерацію: можливість проходити список вперед і назад.

Варіант	*Завдання*
13	Реалізувати вставку перед/після заданого вузла.
14	Реалізувати видалення всіх вузлів з парними значеннями.
15	Реалізувати функцію реверсу двозв'язного списку.
16	Реалізувати структуру deque (двостороння черга) на основі двозв'язного списку.
17	Реалізувати сортування двозв'язного списку методом вставки.
18	Реалізувати функцію обміну двох вузлів за індексами.
19	Реалізувати функцію розділення списку на дві частини: за серединою або за умовою.
20	Реалізувати перетворення двозв'язного списку в кільцевий: останній вузол указує на перший і навпаки.

Контрольні питання

1. Що таке вузол зв'язаного списку та які його основні частини?
2. Чим однозв'язний список відрізняється від двозв'язного?
3. Які основні операції можна виконувати над зв'язаним списком?
4. Яка складність вставки елемента на початок списку?
5. Чому вставка елемента в кінець списку має часову складність O(n)?
6. Як можна зробити вставку в кінець списку більш ефективною?
7. У чому полягає алгоритм обходу зв'язаного списку?
8. Як здійснюється вставка елемента у відсортований зв'язаний список?
9. Яка складність пошуку елемента у зв'язаному списку?
10. Як відбувається видалення першого елемента у зв'язаному списку?

Лабораторна робота № 3
Стек, Черга, Дек

Мета роботи: Отримати практичні навички використання структур даних стек, черга та дек для розв'язання прикладних задач та організації обробки даних.

Короткі теоретичні відомості

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.

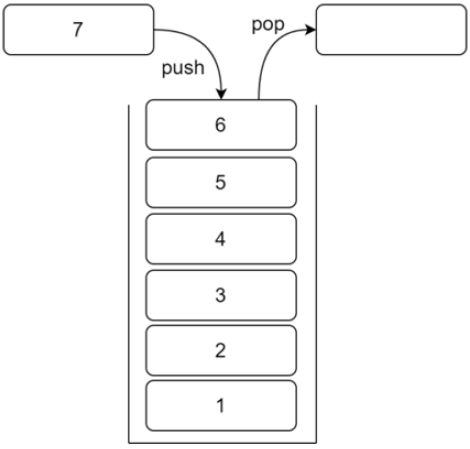


Рис. 3.1. Структура даних «Стек»

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 стеку.
- Пошук у глибину в деревах і графах.

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, в 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 в службу підтримки.

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

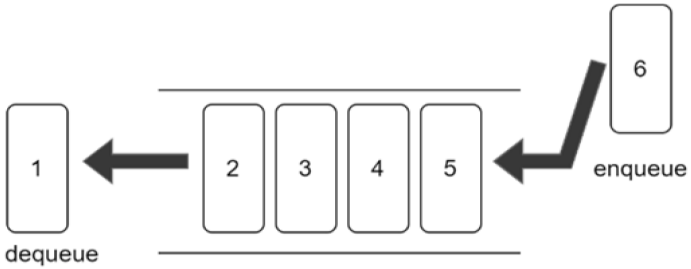


Рис. 3.2. Структура даних «Черга»

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.

Дек (двостороння черга) – це структура даних, яка дозволяє додавати і видаляти елементи з обох кінців.

Дек – скорочення від двосторонньої черги (англ. deque, від double-ended queue). На відміну від звичайної черги, де елементи додаються в один кінець і видаляються з іншого, дек дозволяє операції з обох кінців.

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()` – додати елемент у верхню частину стека
- `pop()` – видалити елемент із вершини стека.
- `top()` – читати значення верхнього елемента стека (не видаляючи його).
- `isEmpty()` – повернути true, якщо стек порожній, інакше повернути false.
- `size()` – повернути кількість елементів у стеку.

Push: Додати значення на вершину стека

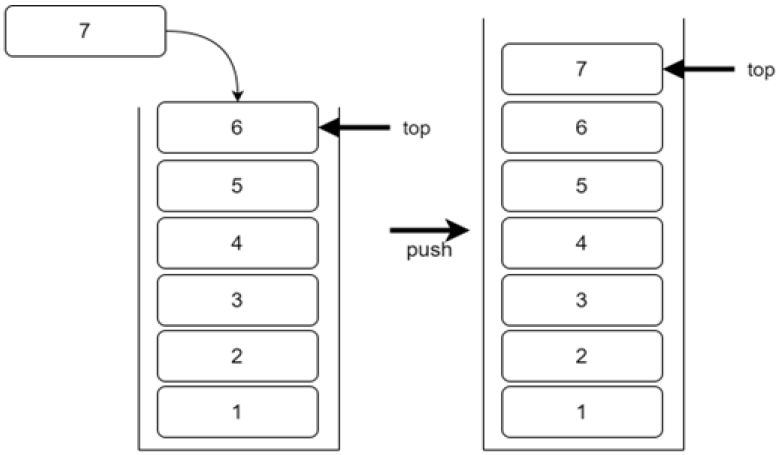


Рис. 3.3. Додавання елемента в стек – push()

Pop: Видалити верхній елемент стека та повернути його до функції виклику

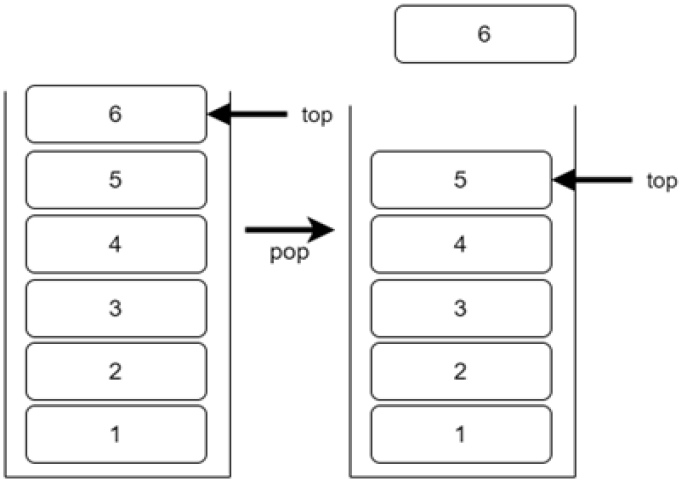


Рис. 3.4. Видалення елемента зі стека – pop()

1. Коли стек реалізований за допомогою масиву, його місткість може бути фіксованою.

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

Абстрактний тип даних «Черга»

Абстрактний тип даних «Черга» визначається як структура даних, екземпляр якої слідує правилу FIFO або «Першим прийшов, першим вийшов2 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 із черги. Оператор обчислення остачі від ділення використовується для отримання наступних значень змінних `back` та `front`. Таким чином, масив поводиться як круговий масив.

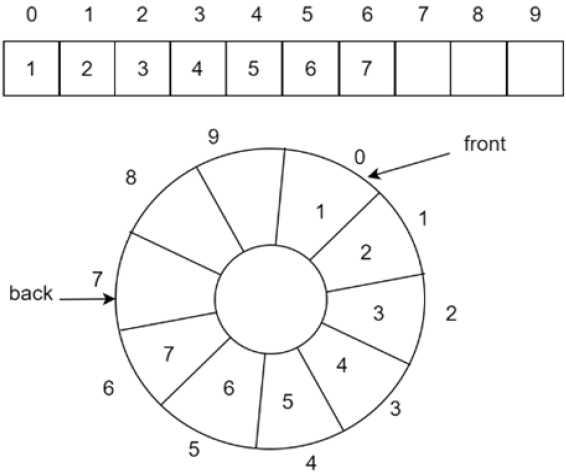


Рис. 3.5. Круговий масив як черга

Абстрактний тип даних «Дек»

Абстрактний тип даних «Дек» (двостороння черга) – це логічна модель структури даних, яка дозволяє додавати та видаляти елементи з обох кінців.

Дек повинен підтримувати такі операції:

- `addFirst()` додати елемент на початок
- `addLast()` додати елемент в кінець
- `removeFirst()` видалити та повернути елемент з початку
- `removeLast()` видалити та повернути елемент з кінця
- `peekFirst()` повернути елемент з початку без видалення
- `peekLast()` повернути елемент з кінця без видалення
- `isEmpty()` перевірити, чи дек порожній
- `size()` повернути кількість елементів

Дек, так само як Стек та Черга, може бути реалізований за допомогою масиву чи лінійного динамічного списку.

Докладна інформація по темі наведена в посібнику [1, с. 62–90].

Завдання

Завдання 3.1

Виконати завдання свого варіанта для кожного з розділів (Стек, Черга та Дек).

1. Рекомендується також виконати одне з трьох (на вибір і необов'язково) додаткових завдань для кожної структури.

2. Для парних варіантів моделювання виконувати на основі масиву (відповідна структура може бути обмежена за розміром). Для непарних варіантів моделювання виконувати на основі лінійного динамічного списку.

3. Реалізувати діалогову програму, що зберігає структуру даних у пам'яті та через меню пропонує користувачеві виконати основні операції роботи з відповідною структурою та відобразити поточний стан структури.

Стек

Для виконання всіх варіантів завдань спочатку створіть реалізацію структури Стек у вигляді класу (використовуйте можливості зручної для вас мови програмування). Обов'язково треба реалізувати такі операції:

- `push` (Додати елемент у стек)
- `pop` (Видалити елемент зі стека)
- `isEmpty` (Перевірити, чи стек порожній)

Також рекомендується реалізувати операції:

- `size` (Отримати кількість елементів у стеку)
- `top` (Отримати, але не вилучити елемент на вершині стека).

Варіант	*Завдання*
1	Знайти максимальний елемент у стеку (простим перебором).
2	Перевірити, чи є стек паліндромом.
3	Розвернути стек, використовуючи допоміжний стек.
4	Перевірити, чи містить стек певне значення.
5	Додати елементи 1..N у стек і вивести тільки парні.
6	Знайти суму всіх елементів у стеку.
7	Порахувати, скільки разів зустрічається певне число.
8	Видалити зі стека всі елементи, менші за задане число.
9	Перемістити всі елементи зі стека в інший стек.
10	Порівняти два стеки на рівність.
11	Перевірити, чи є стек відсортованим за зростанням.
12	Знайти другий елемент зверху, не змінюючи стек.
13	Об'єднати два стеки в один (другий додається до низу першого).
14	Знайти середнє арифметичне елементів у стеку.
15	Зробити копію стека без використання масиву (через допоміжний стек).

Черга

Для виконання всіх варіантів завдань спочатку створіть реалізацію структури Черга у вигляді класу (використовуйте можливості зручної для вас мови програмування). Обов'язково треба реалізувати такі операції:

- `enqueue` (Додати елементи в чергу)
- `dequeue` (Видалити елемент з черги)
- `isEmpty` (Перевірити чи стек порожній)

Також рекомендується реалізувати операції:

- `front` (Отримати, але не вилучити перший елемент)
- `size` (Отримати кількість елементів у черзі)

Варіант	*Завдання*
1	Знайти максимальний елемент у черзі.
2	Перевірити, чи містить черга певне число.
3	Вивести тільки непарні елементи черги.
4	Знайти суму всіх елементів у черзі.
5	Порахувати кількість входжень певного значення.
6	Перемістити всі елементи із черги в іншу чергу.
7	Перевірити, чи є черга відсортованою за зростанням.
8	Об'єднати дві черги в одну.
9	Видалити із черги всі елементи, менші за задане число.
10	Знайти другий елемент у черзі, не видаляючи перший.
11	Створити копію черги.
12	Додати елемент у кінець черги, тільки якщо його там ще немає.
13	Вивести елементи черги у форматі списку через кому.
14	Порівняти дві черги на рівність.
15	Перемістити перші K елементів черги в її кінець.

Дек

Для виконання всіх варіантів завдань спочатку створіть реалізацію структури Дек у вигляді класу (використовуйте можливості зручної для вас мови програмування). Обов'язково треба реалізувати такі операції:

- `pushFront` (Додати елемент у початку дека)
- `pushBack` (Додати елемент у кінці дека)
- `popFront` (Видалити елемент із початку дека)
- `popBack` (Видалити елемент із кінця дека)

Також рекомендується реалізувати операції:

- `front` (Отримати, але не вилучити перший елемент)
- `back` (Отримати, але не вилучити останній елемент)
- `size` (Отримати кількість елементів у черзі)

Варіант	*Завдання*
1	Знайти мінімальний елемент у деці.
2	Перевірити, чи містить дек певне число.
3	Вивести елементи дека зліва направо.
4	Вивести елементи справа наліво.
5	Знайти суму всіх елементів у деці.
6	Видалити всі елементи, більші за задане число.
7	Перемістити всі елементи з одного дека в інший.
8	Перевірити, чи є дек паліндромом.
9	Об'єднати два деки (другий додається справа).
10	Знайти другий елемент зліва, не видаляючи перший.
11	Знайти другий елемент справа.
12	Розвернути дек без використання масиву.
13	Видалити всі дублікати з дека (простий варіант).
14	Порівняти два деки на рівність.
15	Вставити елемент у початок дека, якщо його там ще немає.
16	Вставити елемент у кінець дека, якщо його там ще немає.

Завдання 3.2

Стек

1. **Перетворення інфіксного виразу в постфіксний.** Реалізуйте алгоритм Шунтового двору для виразів з дужками.

2. **Розбиття стека на два.** Розділіть стек на два: один містить парні числа, інший – непарні.

3. **Перевірка коректності вкладених HTML-тегів.** Використовуючи стек, перевірте, чи правильно вкладені теги , , <p> тощо.

Черга

1. **Перевірка, чи можна отримати другу чергу з першої перестановками dequeue/enqueue.** Моделюйте процес.

2. **Фільтрація черги.** Видаліть усі елементи, що діляться на 3, зберігши порядок інших.

3. **Видалення дублікатів, зберігаючи порядок.** Залиште лише перші входження елементів.

Дек

1. **Видалення всіх елементів у діапазоні [L, R].** Збережіть порядок інших.

2. **Видалення сусідніх елементів, що дублюються.** Наприклад, з 1 1 2 2 2 3 → 1 2 3.

3. **Перевірка симетричності послідовності.** Використовуючи дек, визначте, чи є масив симетричним.

Контрольні питання

1. Що таке стек і який принцип його роботи?
2. Як стек використовується для реалізації викликів функцій у програмуванні?
3. Чим стек відрізняється від інших структур даних?
4. Що робить операція push() і яка її складність?
5. Що робить операція pop() і яка її складність?
6. Чим відрізняється операція top() від pop()?
7. Яка різниця між реалізацією стека на масиві та на зв'язаному списку?
8. Які обмеження має стек, реалізований на масиві?
9. У чому полягає перевага стека на зв'язаному списку?
10. Чому функція print() має часову складність O(n)?
11. Що таке інфіксний вираз і чому він може бути неоднозначним?
12. Чим постфіксний запис виразів відрізняється від інфіксного?

13. Який алгоритм використовується для перетворення інфіксного виразу у постфіксний?

14. Як стек застосовується для обчислення постфіксних виразів?

15. Які практичні застосування стека в інформатиці та реальних системах?

Лабораторна робота № 4
Дерева. Бінарне дерево пошуку

Мета роботи: Отримати практичні навички використання деревоподібних структур даних для ефективного зберігання, пошуку та обробки інформації.

Короткі теоретичні відомості

Термінологія дерев

- **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.

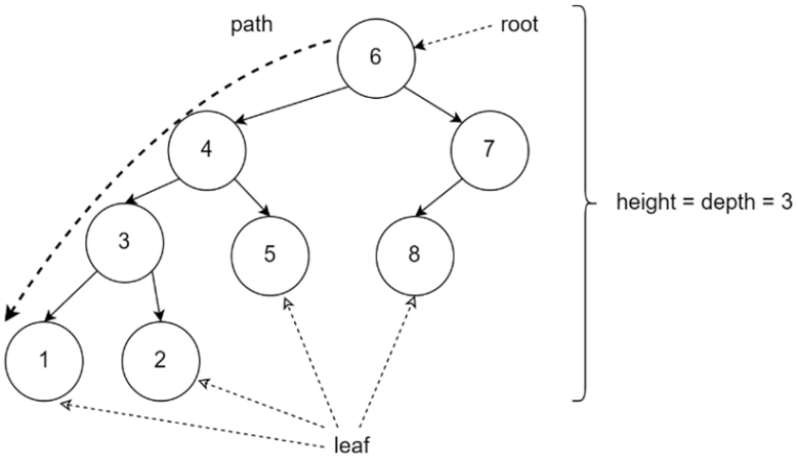


Рис. 4.1. Елементи дерева

- **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)**: висота вузла – це кількість ребер на найдовшому шляху між вузлом і листовим вузлом.
- **Висота дерева (height of tree)**: висота дерева – це кількість ребер на найдовшому шляху між коренем і листовим вузлом.
- **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 та правого дочірніх елементів root дорівнюють 1.

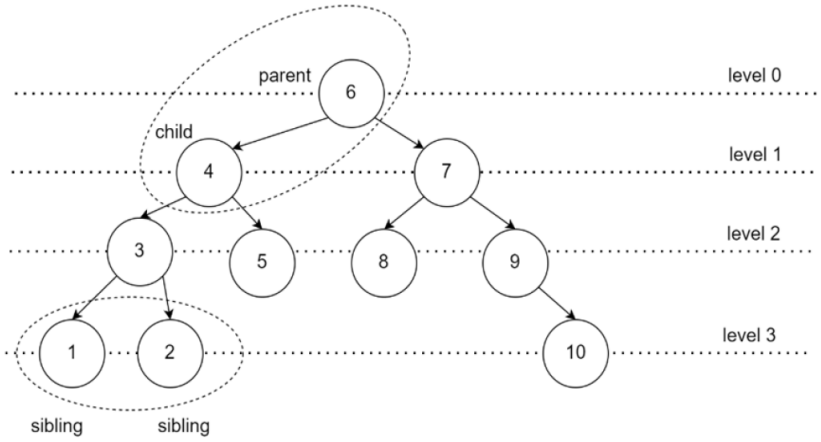


Рис. 4.2. Термінологія дерев

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. На рис. 4.3 зображено вузол бінарного дерева (), у якому «a» – це дані, що зберігаються, а лівий (lchild) і правий дочірні вузли (rchild) вказують на null.

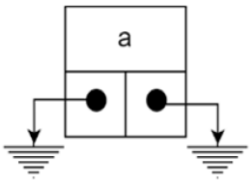


Рис. 4.3. Дерево з одного вузла

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

Розглянемо приклад бінарного дерева (рис. 4.4):

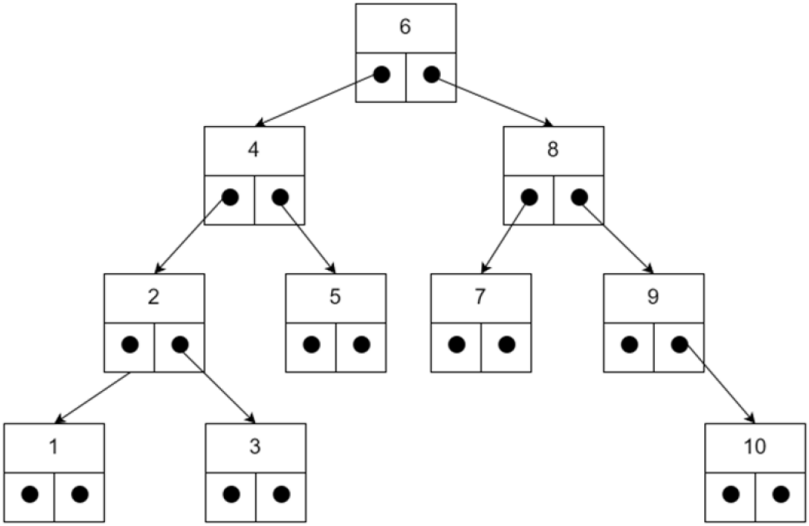


Рис. 4.4. Бінарне дерево, вузли якого містять числа від 1 до 10

Для економії місця бінарне дерево можна зобразити у спрощеному вигляді, як на рисунку 4.5.

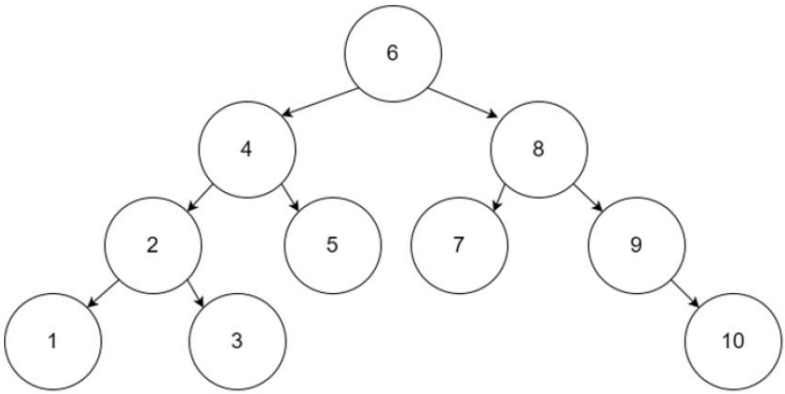


Рис. 4.5. Спрощене зображення бінарного дерева

Бінарне дерево має такі властивості:

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

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 в 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 $\log_2 N$.

Бінарне дерево пошуку

Двійкове дерево пошуку (BST) – це двійкове дерево, в 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.

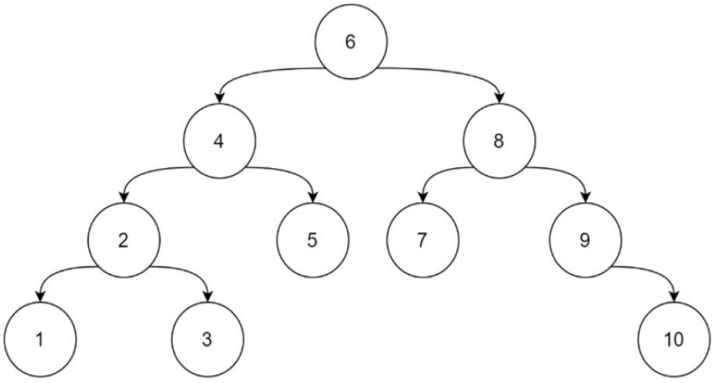


Рис. 4.6. Бінарне дерево пошуку

Розглядаючи наступні питання, варто мати на увазі, що у вузлі дерева може бути два окремі поля ключа та значення (і навіть більше). Однак для простоти будемо вважати, що значення у вузлі і є ключем. Усі задачі в бінарному дереві пошуку вирішуються з урахуванням того, що значення у вузлі є ключами для дерева. Оскільки бінарне дерево пошуку є двійковим деревом, то всі алгоритми бінарного дерева, що були розглянуті раніше, можуть бути застосовані до бінарного дерева пошуку (рис. 4.6).

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. Докладна інформація про особливості та алгоритми роботи з AVL-деревом наведена в посібнику [1, с. 138–146].

Червоно-чорне дерево (RB-Tree)

Червоно-чорне дерево (рис. 4.7) 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 й 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 відносини батько-нащадок.
- Кожен шлях від вершини до листка-нащадка містить однакову кількість чорних вершин.

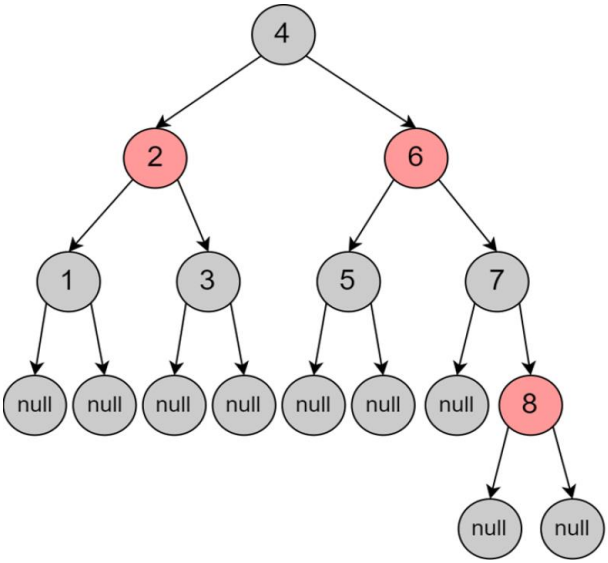


Рис. 4.7. Червоно-чорне дерево

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 рисунку 4.7 кількість чорних вузлів на шляху від кореня до будь-якого листка (NIL-вузла) завжди дорівнює трьом. Як і AVL-дерево, червоно-чорні дерева також є самобалансованими 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 із 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)). Докладна інформація про особливості та алгоритми роботи із червоно-чорними деревами наведена в посібнику [1, с. 147–158].

Завдання

Завдання 4.1

Згідно з варіантом, розробити програму, що реалізує бінарне дерево пошуку для зберігання рядків. Критерій упорядкування вказаний у таблиці.

Варіант	***Критерій упорядкування***
1, 6, 11, 16, 21, 26	У лексикографічному порядку без урахування регістру.
2, 7, 12, 17, 22, 27	У порядку спадання довжини, а якщо довжина однакова – у лексикографічному порядку без урахування регістру.
3, 8, 13, 18, 23, 28	У порядку зростання довжини, а якщо довжина однакова – у лексикографічному порядку з урахуванням регістру
4, 9, 14, 19, 24, 29	У порядку спадання довжини, а якщо довжина однакова – у лексикографічному порядку з урахуванням регістру
5, 10, 15, 20, 25, 30	У лексикографічному порядку з урахуванням регістру

1. Бінарне дерево реалізувати на основі вузлів такої структури:

```
Java
class Node {
    String key;
    Node left;
    Node right;
}
C++
struct Node {
    std::string key;
    Node *left;
    Node *right;
}
Kotlin
class Node (
    var value: String,
    var left: Node? = null,
    var right: Node? = null
)
```

2. Додати у створене дерево 10^5 випадкових чисел. Виміряти час додавання всіх чисел. Повторити для 10^6 чисел.

3. Реалізувати алгоритм обчислення висоти дерева (кількості рівнів у дереві).

Завдання 4.2

Написати функцію перевірки збалансованості бінарного дерева пошуку.

Примітка. *Дерево називається збалансованим, якщо для кожної його вершини висоти її двох піддерев відрізняються не більше ніж на 1.*

Завдання 4.3

Реалізувати структуру AVL-дерева або RB-дерева.

Контрольні питання

1. Що таке дерево як структура даних і чим воно відрізняється від лінійних структур?
2. Які основні елементи дерева (вузол, ребро, корінь, лист)?
3. Що таке шлях у дереві та як він визначається?
4. Як визначається висота вузла та висота дерева?
5. Що означає рівень (глибина) вузла?
6. Хто такі батьківські та дочірні вузли?
7. Як визначаються споріднені вузли?
8. Які властивості має бінарне дерево?
9. Скільки ребер має дерево з N вузлами?
10. Яка максимальна кількість вузлів на рівні L у бінарному дереві?
11. Що таке повне бінарне дерево?
12. Що означає збалансоване бінарне дерево?
13. У чому полягає різниця між AVL-деревом та RB-деревом?
14. У чому полягають принципи обходу дерева у префіксному, інфіксному та постфіксному порядках і чим вони відрізняються?
15. Які практичні застосування мають обходи дерев (префіксний, постфіксний, інфіксний, по рівнях)?

Лабораторна робота № 5
Хеш-таблиці. Черги з пріоритетами

Мета роботи: Отримати практичні навички використання хеш-таблиць і черг з пріоритетами для ефективної організації пошуку, зберігання та обробки даних.

Короткі теоретичні відомості

Хеш-таблиця

Хеш-таблиця (рис. 5.1) – це структура даних, яка відображає ключі на відповідні значення. Кожна позиція хеш-таблиці називається корзиною (bucket, slot). Хеш-таблиця використовує хеш-функцію для обчислення індексу масиву. Хеш-таблицю доцільно використовувати тоді, коли кількість фактично збережених ключів є малою порівняно з обсягом простору всіх можливих ключових значень.

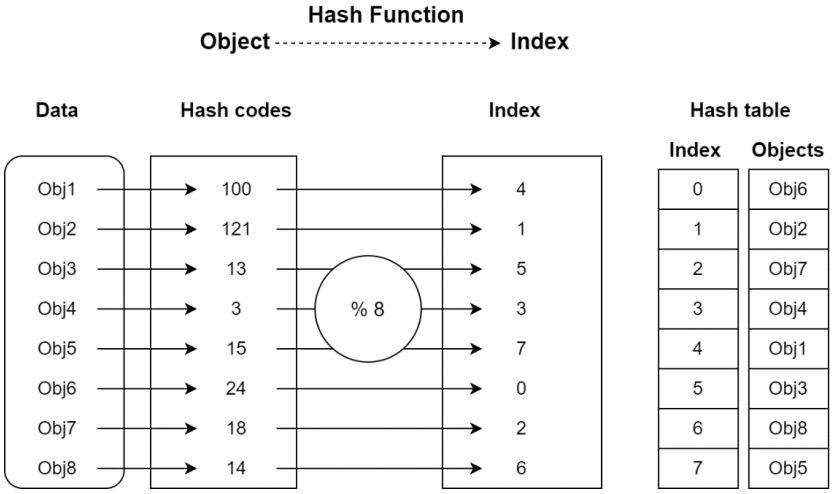


Рис. 5.1. Хеш-таблиця

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.
3. Обчислити остачу від ділення хеш-коду на розмір хеш-таблиці, щоб отримати індекс у таблиці, в 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 за допомогою хеш-функції.
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 є дуже простою хеш-функцією. До цієї функції буде додано різну логіку генерації хешу для отримання кращого хешу.

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. 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)

Відповідно до наведеного 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 при цьому подвоюється. Наприклад, якщо коефіцієнт завантаження дорівнює 3/4, а ємність хеш-таблиці 1024, то ми можемо додати 768 елементів до хеш-таблиці без її перехешування.

Методи розв'язку колізій

Колізії хешу практично неминучі при хешуванні 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 при хешуванні.

Найбільш поширеними і широко використовуваними методами є наступні:

1. Відкрита адресація
2. Метод ланцюжків

Докладна інформація по темі наведена в посібнику [1, с. 182–196].

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.

Докладна інформація по темі наведена в посібнику [1, с. 161–181].

Завдання

Завдання 5.1

Згідно з варіантом розробити програму, що реалізує хеш-таблицю зі списками для зберігання рядків. Виміряти час додавання 10^6 елементів та час пошуку випадкових значень. Порівняти зі стандартною реалізацією у використаній мові програмування (для Java – HashSet).

Варіант	***Коефіцієнт завантаження***
1,4,7,10,13,16,19,22,25,28	loadFactor = 0.5
2,5,8,11,14,17,20,23,26,29	loadFactor = 0.75
3,6,9,12,15,18,21,24,27,30	loadFactor = 1.0

Завдання 5.2

Створіть пріоритетну чергу. Виконайте завдання згідно з варіантом. Реалізуйте дві версії програми: із власною реалізацією пріоритетної черги та зі стандартною PriorityQueue.

Варіант	***Завдання***
1, 1, 21	Знайти k-й найменший елемент у масиві за допомогою min-heap
2, 12, 22	Знайти k-й найбільший елемент у масиві за допомогою max-heap
3, 13, 23	Знайти топ-k найбільших елементів у потоці чисел
4, 14, 24	Знайти топ-k найменших елементів у потоці чисел
5, 15, 25	Підтримувати динамічний набір чисел і швидко повертати мінімум
6, 16, 26	Реалізувати heap sort через пріоритетну чергу
7, 17, 27	Відсортувати масив рядків за довжиною, використовуючи пріоритетну чергу
8, 18, 28	Відсортувати масив пар за пріоритетом
9, 19, 29	Відсортувати масив за модулем числа, використовуючи custom-компаратор у heap
10, 20, 30	Реалізувати чергу задач, де кожна задача має пріоритет

Контрольні питання

1. Що таке хеш-таблиця і яка її основна ідея?
2. Яка роль хеш-функції у хеш-таблиці?
3. Що таке колізія у хеш-таблиці?
4. Які основні методи розв’язання колізій існують?
5. У чому полягає метод лінійного пробування (linear probing)?
6. Як працює метод ланцюжків (separate chaining)?
7. Яка часова складність пошуку елемента у хеш-таблиці в середньому випадку?

8. Що таке коефіцієнт заповнення (load factor) і чому він важливий?

9. Як відбувається перехешування (rehashing) у хеш-таблицях?

10. Які прикладні задачі ефективно розв'язуються за допомогою хеш-таблиць?

11. Що таке пріоритетна черга і чим вона відрізняється від звичайної черги?

12. Як реалізується пріоритетна черга за допомогою купи (Heap)?

13. У чому полягає різниця між Max-Heap та Min-Heap?

14. Чому купа вважається повним бінарним деревом?

15. Як у масиві визначаються індекси дочірніх та батьківських вузлів?

16. Яка часова складність операції вставки у купу?

17. Яка часова складність видалення максимального/мінімального елемента?

18. Чому пошук максимального (у Max-Heap) або мінімального (у Min-Heap) має складність O(1)?

19. Що таке операції percolateUp() та percolateDown() (siftUp() та siftDown())?

20. Як працює алгоритм Heapify для перетворення масиву у купу?

Лабораторна робота № 6
Графи

Мета роботи: Отримати практичні навички використання структур даних для розв'язання задач на графах.

Короткі теоретичні відомості

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 ∈ 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.

Термінологія графів

Неорієнтовний граф: Неорієнтовний граф (рис. 6.1) – це граф, у якому ребра не мають напрямків. Іншими словами, ребра графа мають два напрямки. Ребро (x, y) ідентичне ребру (y, x).

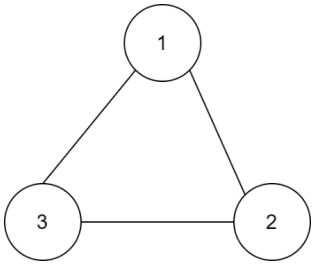


Рис. 6.1. Неорієнтований граф

Орієнтовний граф або **оргграф:** Орієнтовний граф (рис. 6.2) – це граф, у якого ребра мають напрямок. Тут ребра графа є односторонніми. Ребро (x, y) не тотожне ребру (y, x).

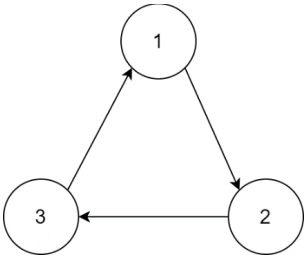


Рис. 6.2. Орієнтований граф

Зважений граф: Граф є зваженим, якщо його ребра мають певне значення або вагу, пов'язану з ними (рис. 6.3).

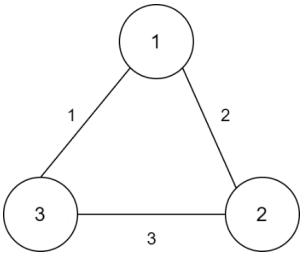


Рис. 6.3. Зважений граф

Незважений граф: Граф, у якому ребра не мають ваги (рис. 6.4).

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.

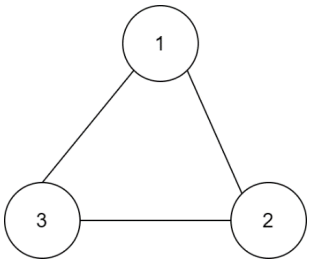


Рис. 6.4. Незважений граф

Суміжні вершини: Дві вершини u та v є суміжними, якщо існує ребро, кінцями якого є u та v.

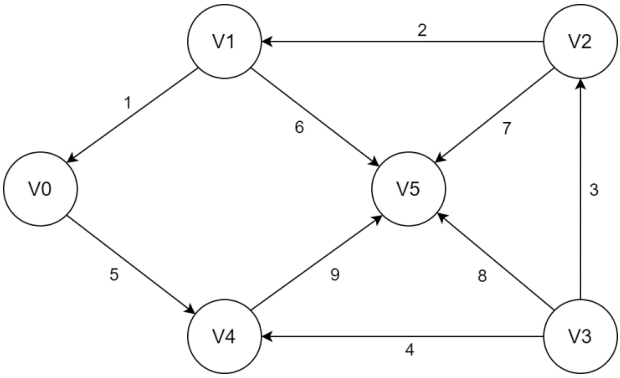


Рис. 6.5. Зважений орієнтований граф

У графі на рисунку 6.5:

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 як indeg(v) – це кількість вхідних ребер у вершину v. Вихідний степінь

вершини v, позначається як outdeg(v) – це кількість вихідних ребер з вершини v.

Степінь вершини v, позначається як 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 і включає 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. 6.6).

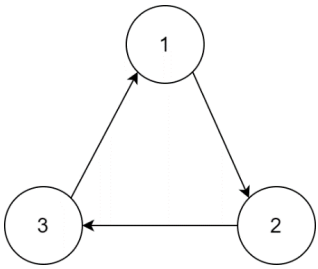


Рис. 6.6. Циклічний граф

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. 6.7).

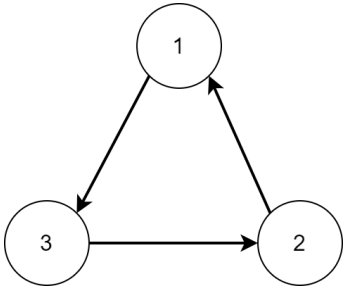


Рис. 6.7. Сильнозв'язний граф

Сильнозв'язні компоненти: Орієнтовний граф може містити різні підграфи, які є сильнозв'язними. Ці підграфи називаються сильнозв'язними компонентами (рис. 6.8).

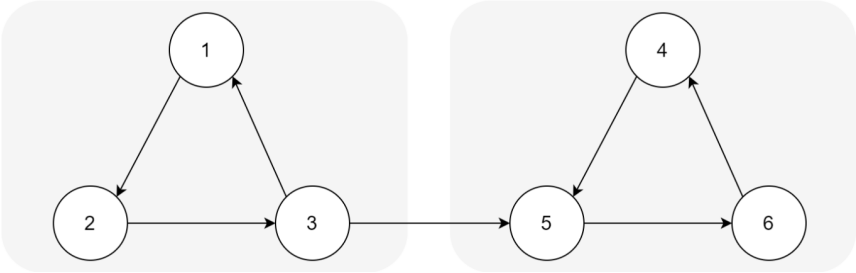


Рис. 6.8. Сильнозв'язні компоненти

Слабозв'язний граф: Орієнтовний 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 неорієнтовного графа з n вершин загальна кількість ребер буде n*(n–1)/2, а для орієнтовного 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. 6.9).

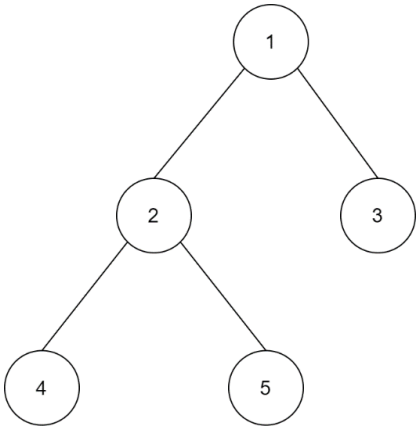


Рис. 6.9. Дерево

Остовне дерево графа – це дерево, яке з'єднує всі вершини графа. Оскільки остовне дерево є деревом, воно не повинно мати циклів (рис. 6.10).

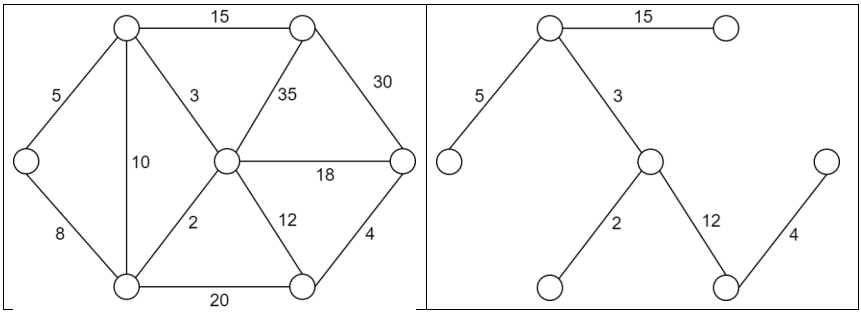


Рис. 6.10. Зважений граф та його мінімальне остовне дерево

Гамільтонів шлях – це шлях, в 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, у якому існує ребро від його останньої вершини до його першої вершини. **Гамільтонів цикл** – це цикл, який відвідує кожну вершину рівно один раз, і він повинен починатися і 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.

Докладна інформація по темі наведена в посібнику [1, с. 197–259].

Завдання

Завдання 6.1

Варіант	***Завдання***
1	Реалізація BFS Опис: Реалізувати пошук в ширину на неорієнтованому графі, знайти відстані від заданої вершини до всіх інших. Вхід: (n, m), список ребер, стартова вершина (s). Вихід: Масив відстаней (−1 для недосяжних). ***Підказка:*** *Використати чергу; складність (O(n+m)).*
2	**Реалізація DFS і топологічне сортування** **Опис:** Реалізувати DFS для орієнтованого ациклічного графа і повернути топологічний порядок. **Вхід:** (n, m), список орієнтованих ребер. **Вихід:** Топологічна перестановка або повідомлення про цикл. ***Підказка:*** *Використати стан вершини (0/1/2) для виявлення циклів; складність (O(n+m)).*
3	Дейкстра для невід'ємних ваг Опис: Реалізувати алгоритм Дейкстри для орієнтованого / неорієнтованого графа з невід'ємними вагами. Вхід: (n, m), список ребер ((u,v,w)), старт (s). Вихід: Масив найкоротших відстаней. ***Підказка:*** *Використати мін-кучу; складність (O((n+m)log n)).*
4	Bellman-Ford з виявленням негативних циклів Опис: Реалізувати Bellman-Ford для знаходження найкоротших шляхів у графі з можливими від'ємними вагами та виявити вершини, до яких ведуть негативні цикли. Вхід: (n, m), ребра ((u,v,w)), старт (s). Вихід: Відстані або позначка для вершин, уражених негативним циклом. ***Підказка:*** *Після (n-1) ітерацій зробити ще одну й відмітити релаксації.*

Варіант	***Завдання***
5	Краскал для мінімального остовного дерева Опис: Реалізувати алгоритм Краскала для знаходження MST у неорієнтованому зваженому графі. Вхід: (n, m), список ребер ((u,v,w)). Вихід: Сума ваг MST або повідомлення, якщо граф не зв'язний. ***Підказка:*** *Використати DSU; складність (O(mlog m)).*
6	Прим для MST (матриця суміжності або список) Опис: Реалізувати алгоритм Прима з використанням мін-купи. Вхід: (n, m), список ребер. Вихід: Сума ваг MST. ***Підказка:*** *Складність (O((n+m)log n)).*
7	**Знаходження сильно зв'язних компонент Косараджу** **Опис:** Реалізувати алгоритм Косараджу для розбиття орієнтованого графа на SCC. **Вхід:** (n, m), орієнтовані ребра. **Вихід:** Нумерація компонент для кожної вершини. ***Підказка:*** *Два проходи DFS; складність (O(n+m)).*
8	**Пошук мостів і артикуляційних вершин** **Опис:** Реалізувати алгоритм Тар'яна для знаходження мостів і точок артикуляції в неорієнтованому графі. **Вхід:** (n, m), список ребер. **Вихід:** Списки мостів і артикуляційних вершин. ***Підказка:*** *Один DFS з tin/low; складність (O(n+m)).*
9	**Алгоритм Форда-Фалкерсона (максимальний потік)** **Опис:** Реалізувати пошук збільшувальних шляхів (DFS/BFS) для знаходження максимального потоку у мережі. **Вхід:** (n, m), орієнтовані ребра з пропускною здатністю, джерело (s), стік (t). **Вихід:** Значення максимального потоку. ***Підказка:*** *Для кращої ефективності використати Edmonds-Karp (BFS) або Dinic.*

Варіант	*Завдання*
10	**Перевірка двочастинності і побудова парування (DFS/BFS)** **Опис:** Перевірити, чи є неорієнтований граф двочастинним; якщо так – побудувати будь-яке максимальне парування (жадібно або Hopcroft-Karp для двочастинних графів). **Вхід:** (n, m), список ребер. **Вихід:** Так/Ні; якщо Так – розмір парування і пари. ***Підказка:*** *Для великого графа використовувати Hopcroft-Karp (O(sqrt{V}E)).*

Завдання 6.2

Варіант	*Завдання*
1	**Короткі шляхи з обмеженням кількості ребер** **Опис:** Для орієнтованого зваженого графа знайти найкоротші відстані від (s) до всіх вершин за умови, що шлях може містити не більше (k) ребер. **Вхід:** (n, m), список орієнтованих ребер ((u,v,w)), (s), (k). **Вихід:** Найкоротша відстань або INF, якщо немає шляху з ≤ (k) ребер. ***Підказка:*** *Динаміка по кількості ребер або модифікований Bellman-Ford; складність (O(kcdot m)).*
2	**Планування з дедлайнами і пріоритетами** **Опис:** Є набір завдань з пріоритетом і залежностями (орієнтований граф). Потрібно знайти допустимий порядок виконання, який мінімізує суму пріоритетів помножених на час виконання (або вказати, що неможливо). **Вхід:** (n, m), ребра залежностей, масив пріоритетів/ваг, час виконання кожного завдання. **Вихід:** Порядок виконання або NO. ***Підказка:*** *Комбінувати топологічне сортування з чергою пріоритетів; greedy + DAG.*

Варіант	*Завдання*
3	**Найкоротший шлях з обмеженням сумарної вартості ресурсів** **Опис:** Кожне ребро має вагу та споживання ресурсу. Знайти найкоротший шлях від (s) до (t), такий що сумарне споживання ≤ (R). **Вхід:** (n, m), ребра ((u,v,w,c)), (s,t,R). **Вихід:** Мінімальна вага шляху або NO. ***Підказка:*** *Динаміка на станах (вершина, витрачений ресурс) або модифікований Дейкстра; складність (O(nRlog(nR)+mR)).*
4	**Мінімальна вартість з обмеженням кількості зупинок (авіарейси)** **Опис:** Мережа рейсів як орієнтований граф з вартістю; знайти мінімальну вартість з (s) до (t) з не більше ніж (K) пересадками. **Вхід:** (n, m), ребра ((u,v,price)), (s,t,K). **Вихід:** Мінімальна ціна або NO. ***Підказка:*** *Модифікація Bellman-Ford або BFS по рівнях з пріоритетною чергою; складність (O(Kcdot m)).*
5	**MST з обмеженням на максимальне ребро** **Опис:** Знайти MST, у якому максимальна вага ребра мінімізована (мінімізувати bottleneck). Повернути суму ваг і максимальну вагу ребра. **Вхід:** (n, m), ребра ((u,v,w)). **Вихід:** Сума MST і bottleneck або NO. ***Підказка:*** *Краскал природно дає мінімальний bottleneck; перевірити зв'язність.*
	Побудова двох незалежних MST **Опис:** Знайти два остовні дерева з мінімальною сумою їх ваг, що не мають спільних ребер (якщо можливо). **Вхід:** (n, m), ребра ((u,v,w)). **Вихід:** Суми двох MST або повідомлення, якщо неможливо.

Варіант	*Завдання*
6	***Підказка:*** *Комбінаторна задача; можна шукати перше MST, потім для кожного ребра MST шукати MST без нього (перебір + Краскал), оптимізувати.*
7	**Аналіз впливу вузлів у мережі** **Опис:** Для орієнтованого графа знайти всі вершини, видалення яких збільшує кількість SCC (тобто критичні вузли для сильної зв'язності). **Вхід:** (n, m), орієнтовані ребра. **Вихід:** Список таких вершин. ***Підказка:*** *Поєднати SCC + аналіз конденсованого графа; або використовувати алгоритми для артикуляційних точок на неорієнтованому графі після перетворення.*
8	Відновлення зв'язності після видалення ребер **Опис:** Дані запити: видалити ребро, або запитати, чи залишаються дві вершини зв'язаними. Потрібно обробити послідовність запитів офлайн. **Вхід:** Початковий граф, послідовність запитів (видалення / перевірка). **Вихід:** Відповіді на запити перевірки. ***Підказка:*** *Використати DSU з відкатом або обробку запитів у зворотному порядку (offline dynamic connectivity).*
9	**Потік з нижніми та верхніми межами** **Опис:** Кожне ребро має нижню і верхню пропускну здатність. Перевірити, чи існує допустимий потік від (s) до (t) заданого значення (F), і знайти такий потік. **Вхід:** (n, m), ребра ((u,v,low,cap)), (s,t,F). **Вихід:** Так/Ні і приклад потоку, якщо існує. ***Підказка:*** *Перетворити задачу на класичний потік з додатковою суперджерелом/суперстоком та перевіркою балансу.*

Варіант	*Завдання*
10	**Оптимальне розміщення ресурсів через парування** **Опис:** Є дві множини об'єктів (наприклад, працівники і проєкти) з додатковими обмеженнями (кожен працівник може виконати не більше (k_i) проєктів, кожен проєкт потребує (r_j) працівників). Побудувати розподіл, що задовольняє вимоги або повідомити, що неможливо. **Вхід:** Розміри множин, матриця сумісності, вектори (k_i) і (r_j). **Вихід:** Матчинг/розподіл або NO. ***Підказка:*** *Модель як мережевий потік з джерелом, стоком і пропускними здатностями на вершинах.*

Контрольні питання

1. Що таке граф як структура даних і які його основні компоненти?
2. Чим відрізняється орієнтований граф від неорієнтованого?
3. Що таке зважений граф і які задачі він дозволяє розв'язувати?
4. Як визначається шлях у графі та його довжина (вага)?
5. Що таке простий шлях?
6. Як визначаються суміжні вершини?
7. Що таке ступінь вершини та як він обчислюється?
8. Чим відрізняється цикл від самопетлі?
9. Що таке циклічний та ациклічний граф?
10. Що означає поняття досяжності вершини у графі?
11. Чим відрізняється зв'язний граф від сильно зв'язного?
12. Що таке сильнозв'язні компоненти?
13. Як визначається слабка зв'язність графа?
14. Що таке повний граф і скільки ребер він має?

15. Чим відрізняється дерево від лісу у теорії графів?

16. Що таке остовне дерево графа і для чого воно використовується?

17. Що таке гамільтонів шлях і гамільтонів цикл?

18. Що таке ейлерів шлях та ейлерів цикл?

19. Які основні способи представлення графів існують (матриця суміжності, список суміжності)?

20. У чому полягають переваги та недоліки матриці суміжності порівняно зі списком суміжності?

СПИСОК ЛІТЕРАТУРИ

1. Беркунський Є. Ю. Структури та організація даних мовою Kotlin : навч. посіб. / Є. Ю. Беркунський, А. Ю. Павленко. – Миколаїв : НУК, 2025. – 262 с.
2. Беркунський Є. Ю. Алгоритмізація та програмування мовами Kotlin, C/C++ : навч. посіб. / Є. Ю. Беркунський, А. Ю. Павленко. – Миколаїв : НУК, 2022. – 256 с.
3. Алгоритми та структури даних (навчальний посібник): [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології» / укладач: Ю. Є. Грудзинський; КПІ ім. Ігоря Сікорського. – Київ : КПІ ім. Ігоря Сікорського, 2022. – 215 с.
4. Алгоритми та структури даних (комп'ютерний практикум): [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології» / укладач: Ю. Є. Грудзинський; КПІ ім. Ігоря Сікорського. – Київ : КПІ ім. Ігоря Сікорського, 2022. – 100 с.
5. Коротєєва Т. О. Алгоритми та структури даних. Навчальний посібник. – Львів : Вид-во Львівської політехніки, 2014. – 280 с.
6. Campesato O. 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.

Навчальне видання

БЕРКУНСЬКИЙ Євген Юрійович
ПАВЛЕНКО Альона Юріївна

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт з дисципліни «Структури та організація даних»

Комп'ютерна правка й коректура *О. Є. Вакула*
Комп'ютерне верстання *Ю. С. Семенченко*

Формат 60×84/16. Ум. друк. арк. 3,55. Вид. № 48. Зам. № 2110-95.
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.