Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут комп'ютерних систем
Кафедра інформаційних систем

Лабораторна робота № 7
з дисципліни: «Теорія алгоритмів»
Тема: «Робота з бінарним та червоно-чорним деревами пошуку»

Варіант № 6

Виконав:
Студент групи АІ-243
Гаврилов О. В.
Перевірили:
Смик С. Ю.
Арсірій О.О.

Одеса 2025

Мета роботи:

виконання лабораторної роботи є набуття практичних навичок із проектування, реалізації, тестування та аналізу алгоритмів пошуку, вставки та видалення елементів із деревоподібних структур даних. В якості структур даних для задачі пошуку розглядаються бінарне дерево пошуку та червоно- чорне дерево

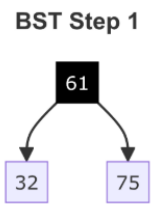
Завдання:

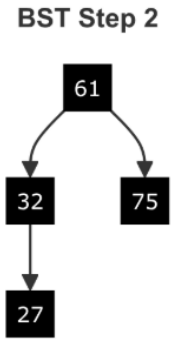
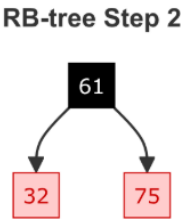
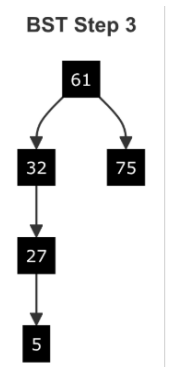
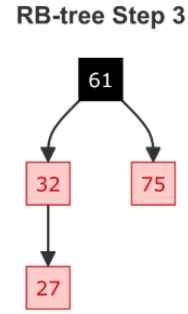
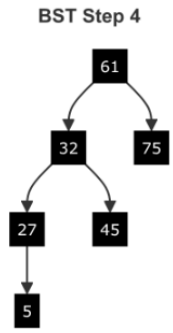
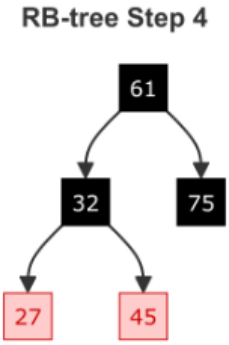
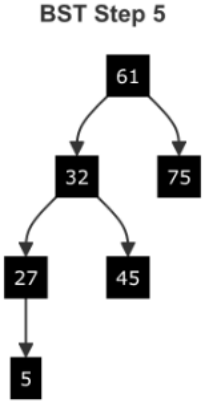
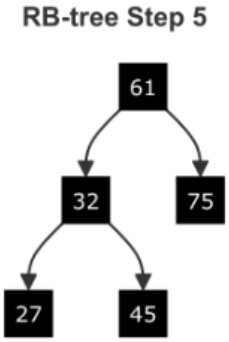
1) Побудувати бінарне дерево пошуку та червоно-чорне дерево за варіантами завдань наводячи покроковий опис (зразок таблиця 7.2)
2) Навести результати симетричного, прямого та зворотного обходів побудованого бінарного дерева пошуку та червоно-чорне дерева. Для симетричного обходу навести стан стека викликів рекурсивної функції inorderTraversal. Порівняйте результати, зробить висновки
3) Видаліть по черзі та опишіть процедуру видалення із побудованого бінарного дерева пошуку трьох кореневих елементів в такому порядку: праве, ліве піддерева, основний кореневий елемент (зразок таблиця 7.4)
4) Видаліть по черзі та опишіть процедуру видалення із побудованого червоно-чорного дерева трьох чорних елементів, які мають переважно чорних синів (зразок таблиця 7.5)

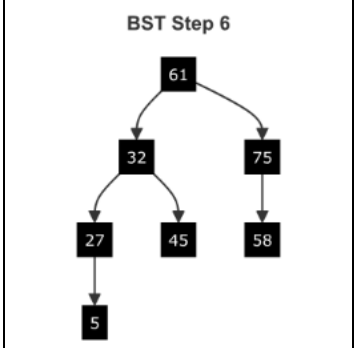
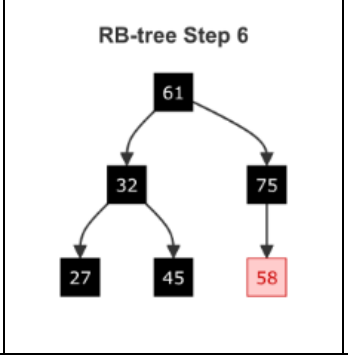
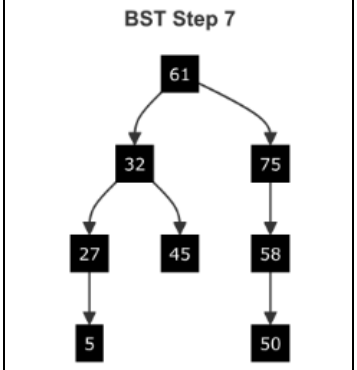
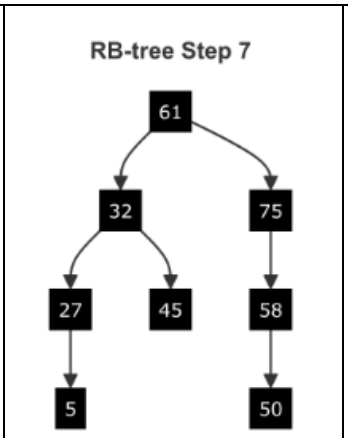
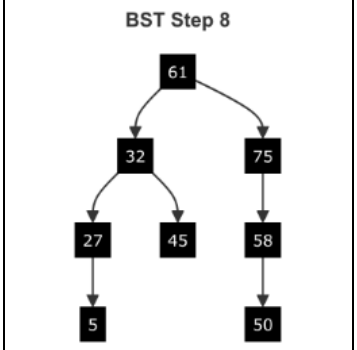
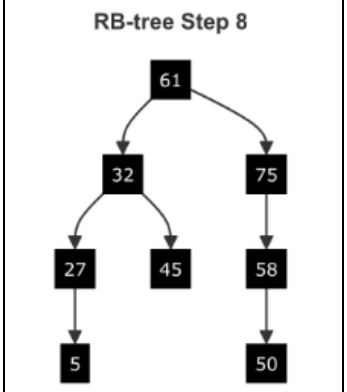
Номер варіанту: 6;
Вхідні дані: 61, 32, 27, 45, 75, 58, 5, 50, 99;

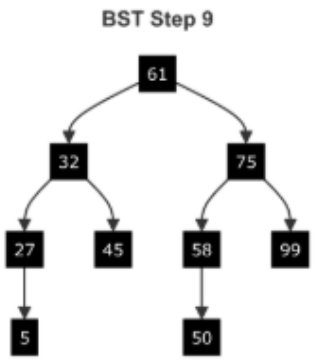
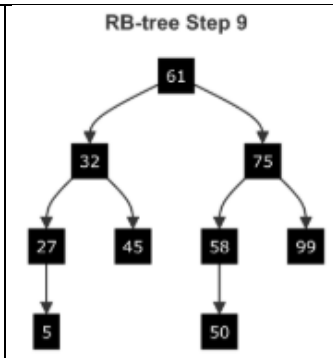
Результати виконання завдання:

Таблиця 7.1 − Побудова бінарного та червоно-чорного дерева пошуку

<table>
<tr><td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td></tr>
<tr><td>Побудова BST
Додаткових пояснень не потребує</td><td colspan="2">Побудова RB tree. Пояснення дій за алгоритмом вставки обов’язкові</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 61.
Корень красно-чорного дерева зафарбовуємо в чорний колір.
Порушень немає.
Вимоги стосовно чорної висоти виконуються.</td></tr>
<tr><td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td></tr>
</table>

<table>
<tr>
<td> 
</td>
<td> 
</td>
<td>Додавання елемента 32.
Новий вузол зафарбовуємо в червоний колір.
Порушень немає. Батько (61)
чорний, тому червоно-чорне
порушення відсутнє.
Вимоги стосовно чорної висоти
виконуються.</td>
</tr>
<tr>
<td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
<td>Додавання елемента 27.
Новий вузол зафарбовуємо в червоний колір.
Червоне-червоне порушення.
Дядько 61 чорний.
Випадок 3 утворення кута.
Перефарбовуємо батька (32) в чорний
колір, робимо правий поворот
навколо вузла 32.
Порушень немає.</td>
</tr>
<tr>
<td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
<td>Додавання елемента 45.
Новий вузол зафарбовуємо в червоний колір.
Червоне-червоне порушення.
Дядько 27 червоний.
Випадок 2 подвійного порушення.
Перефарбовуємо батька (32) та
дядька (27) в чорний колір,
а дідуся (61) в червоний.
Тепер батько 61 червоний,
але його батько відсутній,
тому зафарбовуємо 61 в чорний колір.
Порушень немає.</td>
</tr>
<tr>
<td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
<td>Додавання елемента 75.
Новий вузол зафарбовуємо в червоний колір.
Порушень немає. Батько (61)
чорний, тому червоно-чорне
порушення відсутнє.
Вимоги стосовно чорної висоти
виконуються.</td>
</tr>
</table>

<table>
<tr><td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 58.
Новий вузол зафарбовуємо в червоний колір.
Червоне-червоне порушення.
Дядько 61 чорний.
Випадок 3 утворення кута.
Робимо лівий поворот
навколо вузла 75.
Порушень немає.</td></tr>
<tr><td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 5.
Новий вузол зафарбовуємо в червоний колір.
Червоне-червоне порушення.
Дядька немає.
Випадок 3 утворення кута.
Робимо правий поворот
навколо вузла 27.
Порушень немає.</td></tr>
<tr><td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td></tr>
<tr><td> 
</td><td> 
</td><td>Додавання елемента 50.
Новий вузол зафарбовуємо в червоний колір.
Червоне-червоне порушення.
Дядька немає.
Випадок 3 утворення кута.
Робимо правий поворот
навколо вузла 58, а потім
лівий поворот навколо вузла 45.
Порушень немає.</td></tr>
<tr><td colspan="3">61, 32, 27, 45, 75, 58, 5, 50, 99</td></tr>
</table>

		Додавання елемента 99. Новий вузол зафарбовуємо в червоний колір. Порушень немає. Батько (75) чорний, тому червоно-чорне порушення відсутнє. Завершуємо побудову. Вимоги стосовно чорної висоти виконуються.

1. Симетричний обхід (inorderTraversal)

Лістинг 7.1 – Симетричний обхід (inorderTraversal)

```
func inorderTraversal(x: Node)
    if x != null
        inorderTraversal(x.left)
        print x.key
        inorderTraversal(x.right)
```

Відповідно до inorderTraversal алгоритму порядок обходу елементів бінарного дерева пошуку у симетричному порядку наступний:

5, 27, 32, 45, 50, 58, 61, 75, 99.

Відповідно до inorderTraversal алгоритму порядок обходу елементів червоно-чорного дерева у симетричному порядку наступний:

5, 27, 32, 45, 50, 58, 61, 75, 99.

Стан стека викликів функції inorderTraversal

При цьому стек викликів функції inorderTraversal на кожній ітерації буде виглядати наступним чином:

Таблиця 7.2 − Стан стека викликів функції inorderTraversal

Ітерація	Вміст стека
1	61

2	61, 32
3	61, 32, 27
4	61, 32, 27, 5
5	61, 32, 27
6	61, 32
7	61, 32, 45
8	61, 32
9	61
10	61, 75
11	61, 75, 58
12	61, 75, 58, 50
13	61, 75, 58
14	61, 75
15	61
16	61, 99
17	61
18	-

Як бачимо в першому і другому випадках кількість ітерацій при симетричному обході однакова. Але в першому випадку в стеку викликів на 4 та 12 ітераціях міститься по 4 виклики відповідно на відміну від другого випадку, де 4 виклики - максимальна кількість викликів. Тому для реалізації inorderTraversal для BST знадобилося більше витрат пам'яті ніж для RB tree.

2. Прямий обхід (preorderTraversal)

Лістинг 7.3 – Прямий обхід (preorderTraversal)

```
func preorderTraversal(x: Node)
    if x != null
        print x.key
        preorderTraversal(x.left)
        preorderTraversal(x.right)
```

Відповідно до preorderTraversal алгоритму порядок обходу елементів бінарного дерева пошуку у прямому порядку наступній:

61, 32, 27, 5, 45, 75, 58, 50, 99

Відповідно до preorderTraversal алгоритму порядок обходу елементів червоно-чорного дерева у прямому порядку наступній:

61, 32, 27, 5, 45, 75, 58, 50, 99

3. Зворотний обхід (postorderTraversal)

Обхід вузлів у зворотному порядку

Лістинг 7.4 – Зворотний обхід (postorderTraversal)

```
func postorderTraversal(x: Node)
    if x != null
        postorderTraversal(x.left)
        postorderTraversal(x.right)
        print x.key
```

Відповідно до postorderTraversal алгоритму порядок обходу елементів бінарного дерева пошуку у зворотному порядку наступній:

5, 27, 45, 32, 50, 58, 99, 75, 61

Відповідно до postorderTraversal алгоритму порядок обходу елементів червоно-чорного дерева у зворотному порядку наступній:

5, 27, 45, 32, 50, 58, 99, 75, 61

Таблиця 7.3 − Порівняльна таблиця результатів

Тип обходу	**BST**	**RB-tree**	**Відмінності**
Симетричний (inorder)	5, 27, 32, 45, 50, 58, 61, 75, 99	5, 27, 32, 45, 50, 58, 61, 75, 99	Немає
Прямий (preorder)	61, 32, 27, 5, 45, 75, 58, 50, 99	61, 32, 27, 5, 45, 75, 58, 50, 99	Немає
Зворотний (postorder)	5, 27, 45, 32, 50, 58, 99, 75, 61	5, 27, 45, 32, 50, 58, 99, 75, 61	Немає

Ідентичність результатів: Для даної послідовності елементів обидва типи дерев (BST та RB-tree) дають ідентичні результати при всіх типах обходів.

– Властивість симетричного обходу: Симетричний обхід для обох дерев дає відсортовану послідовність елементів **(5, 27, 32, 45, 50, 58, 61, 75, 99**), що підтверджує правильність побудови бінарного дерева пошуку.

– Використання пам'яті: Максимальна глибина стека при симетричному обході склала 4 виклики, що відповідає висоті дерева. Для даного прикладу різниця у використанні пам'яті між BST та RB-tree не виявляється, оскільки обидва дерева мають однакову структуру.

– Часова складність: Всі алгоритми обходів мають часову складність O(n), де n - кількість вузлів у дереві (у нашому випадку n = 9).

– Переваги RB-tree: Хоча в даному конкретному випадку результати ідентичні, RB-tree гарантує збалансованість (висота O(log n)), що у великих деревах забезпечить меншу глибину стека порівняно з несбалансованим BST.

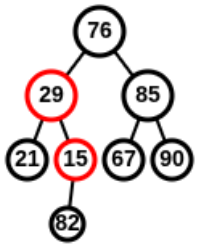
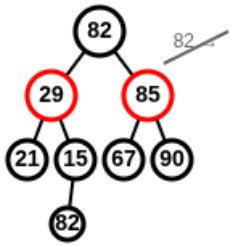
Загальний висновок: Для даної послідовності елементів обидва дерева мають однакову структуру і дають ідентичні результати при всіх типах обходів. Різниця між BST і RB-tree проявляється в гарантіях балансування та додаткових властивостях RB-tree,

Таблиця 7.4 − Видалення вузлів із бінарного дерева пошуку

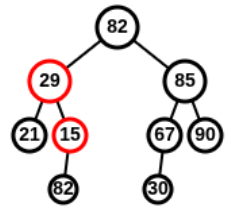
<table>
<tr><td colspan="2">Видаляємо правий кореневий елемент 75</td></tr>
<tr><td>До видалення:
61 / \ 32 75 / \ / \ 27 45 58 99 / 5 \ 50
↓
75 → 99
Після видалення:
61 / \ 32 99 / \ 27 45 / 5 \ 50</td><td>Кореневий елемент правого піддерева 75, що видаляється, має два дочірніх вузла (елементи 58 та 99). Шукаємо крайній лівий елемент правого піддерева – це елемент 99. Тому елемент 99 стає на місце елемента 75.</td></tr>
<tr><td colspan="2">Видаляємо лівий кореневий елемент 32</td></tr>
</table>

До видалення: 61 / \ 32 99 / \ 27 45 / 5 \ 50 ↓ 32 → 45 Після видалення: 61 / \ 45 99 / 27 / 5 \ 50	Кореневий елемент лівого піддерева 32, що видаляється, має два дочірніх вузла (елементи 27 та 45). Шукаємо крайній лівий елемент правого піддерева – це елемент 45. Тому елемент 45 стає на місце елемента 32.
Видаляємо кореневий елемент 61	
До видалення: 61 / \ 45 99 / 27 / 5 \ 50 ↓ 61 → 99 Після видалення: 45 / \ 27 99 / 5 \ 50	Кореневий елемент дерева 61, що видаляється, має тільки один лівий дочірній вузол (елемент 45). Тому він займає місце кореневого елемента 61.

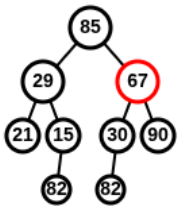
Таблиця 7.5 − Видалення вузлів із червоно-чорного дерева

Видаляємо кореневий елемент 76	
До:  Після:  	Алгоритм видалення: 1.Знаходимо крайній лівий елемент правого піддерева кореня 76. Це елемент 82 (червоний). 2.Заміняємо корінь 76 на елемент 82. 3.Видаляємо вузол 82 з його колишньої позиції (залишається двічі чорний). 4.Чорну висоту (вл. 4) не порушено, оскільки вузол 82 був червоним. 5.Вузол 85 стає червоним, а всі інші вузли зберігають свої кольори. 6.Властивості красно-чорного дерева відновлено. Результат: Нове дерево з коренем 82 (чорним), ліве піддерево містить вузли 29 (червоний), 21 і 15 (чорні), праве піддерево містить вузол 85 (червоний) з піддеревом, що містить 67, 90, 82 (всі чорні).
Видаляємо кореневий елемент 82	

До:



Після:



Алгоритм видалення:

1.Знаходимо крайній лівий елемент правого піддерева кореня 82. Це елемент 85 (червоний).

2.Заміняємо корінь 82 на елемент 85. Узол 85 стає «двічі» чорним (порушення чорної висоти).

3.Виконуємо праворуч оберт навколо вузла 29, де 29 стає новим коренем лівого піддерева, а 85 його правим нащадком.

4.Перефарбовуємо 29 і 85 в чорний колір.

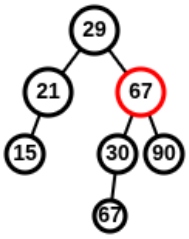
5.Виконуємо ліворуч оберт навколо 67 і 30, та праворуч оберт навколо 67 і 85 для відновлення балансу.

6.Всі властивості красно-чорного дерева відновлено.

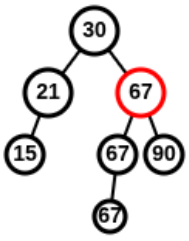
Результат: Нове дерево з коренем 85 (чорним), збалансованою структурою та правильним розподілом кольорів вузлів.

Видаляємо кореневий елемент 29

До:



Після:



Алгоритм видалення:

1.Знаходимо крайній лівий елемент правого піддерева кореня 29. Це елемент 30 (червоний).

2.Заміняємо корінь 29 на елемент 30.

3.Видаляємо вузол 30 з його колишньої позиції.

4.Оскільки вузол 30 був червоним, чорна висота не порушується.

5.Порушень немає - всі властивості красно-чорного дерева збережено.

6.Дерево залишається збалансованим без додаткових операцій.

Результат: Нове дерево з коренем 30 (чорним), збереженою структурою та правильним кольоровим кодуванням всіх вузлів. Трансформація пройшла без порушення властивостей дерева.

Посилання на GitHub:

https://github.com/AlexKim71/Theory-of-Algorithms/tree/main