|  |  |  |  |
| --- | --- | --- | --- |
| МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені Тараса Шевченка  ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ  **Кафедра програмних систем і технологій**  Дисципліна  **«Структури данних, аналіз і**  **алгоритми ком’ютерної**  **обробки інформаціїї»**  **Лабораторна робота № 2**  **на тему:**  **Бінарні Дерева** | | | |
| **Виконав:** | Круць Михайло Ярославович | **Перевірила**: | Бичков Олексій Сергійович |
| Група | ІПЗ-12(1) | Дата перевірки |  |
| Форма навчання | денна | Оцінка |  |
| Спеціальність | 121 |
| 2022 | | | |

**Умова задачі:**

Використовуючи структуру даних: бінарне дерево пошуку, червоно-чорне дерево та АВЛ дерево реалізувати такі алгоритми:

1. Вставка вузла в дерево
2. Видалення вузла із дерева
3. Перефарбування двох однакових вузлів братів у протилежний колір
4. Лівий, правий повороти

**Аналіз задачі:**

Дана лабораторна робота підсумовує розробку різних структур даних на основі бінарного дерева та реалізації різних алгоритмів його обробки:

1. Вставка вузла в дерево
2. Видалення вузла з дерева
3. Пошук вузла
4. Лівий Правий повороти
5. Перефарбування двох однакових вузлів братів у протилежний колір
6. Виведення значення дерева у певному вигляді обходу(Pre-Order, Infix-Order, Post-Order)

* **Бінарне дерево пошуку** - це бінарне дерево, що має додаткову властивість: значення лівого менше за значення батька, а значення правого нащадка більше значення батька для кожного вузла дерева. Тобто дані в бінарному дереві пошуку зберігаються у відсортованому вигляді.
  + **Властивості:** Нехай **x** є вузол бінарного дерева пошуку. Якщо **y** є вузлом у лівому під-дереві **x**, то **y.key<x.key**. Якщо **y** є вузлом у правому під-дереві **x**, то **y.key>x.key**.
* **Червоно-чорне дерево** - це бінарне дерево з додатковим атрибутом – колір: червоний або чорний. Таке дерево широко використовується переважно для зберігання впорядкованих даних, його часова складність становить O(lg n) – де “n” висота дерева.
  + **Властивості**:
    - Кожен вузол червоний або чорний
    - Корінь дерева – чорний
    - Кожен лист дерева (NUL) – чорний
    - Якщо вузол червоний, то дочірні від нього – чорні.
    - Для кожного вузла всі прості шляхи від нього до листя, що є потомками даного вузла, мають однакову кількість чорних вузлів.
* **АВЛ дерево -** це бінарне дерево пошуку з додатковою властивістю – воно є збалансованим. Тобто для будь-якого вузла дерева висота його правого під-дерева відрізняється від висоти лівого під-дерева не більше ніж на один.
  + **Властивості:** Нехай **x** є вузол даного дерева. То за властивістю АВЛ дерева маємо: [-1 <= x.left.height – x.right.height <= 1].
    - Висота NULL вузла 0
    - Кожен вузол має атрибут висоти (1 + найбільша висота серед нащадків)

**Структура основних вхідних та вихідних даних:**

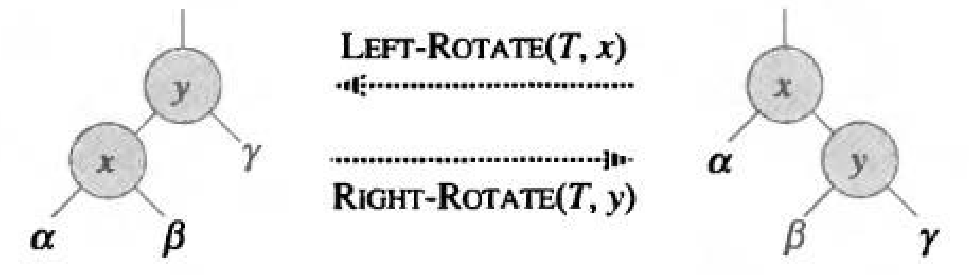
* Вхідні дані – масив чисел
* Вихідні дані – значення вузлів. Якщо дерево червоно-чорне або АВЛ то додатково виводиться або колір або висота

**Алгоритм розв'язання задачі:**

* **Головне меню та клас Вузла:**

У головному меню програми в залежності від обраної структури даних, створюємо екземпляр даної структури. Клас вузла створюється в окремому класі та його екземпляри лише в класах структур даних (Поліморфізм).

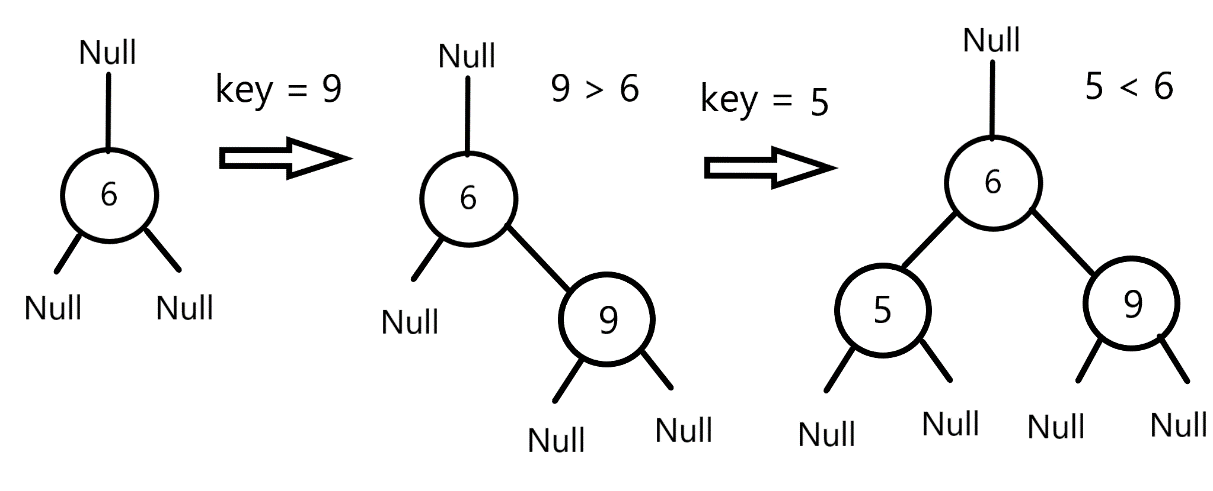
* **Лівий та Правий повороти:**



Для лівого повороту встановлюємо значення Y як правий нащадок вузла X (поточного вузла). Призначаємо ліве під-дерево Y (на малюнку «В»), як праве під-дерево Х. Потім передаємо батька Х до Y. Та в кінці розміщуємо вузол Х, як ліве під-дерево вузла Y.

Для правого повороту виконуються усі ті ж самі операції, але з протилежним покажчиками.

* **Додавання:**
  + **Бінарне дерево пошуку:**

Спершу виконується перевірка чи дерево порожнє, тобто вузол кореня **= NULL**. Якщо так, то корінь отримує значення вузла який ми хочемо додати. В іншому випадку запускається цикл, який перевіряє з вузла кореня допоки не буде доданий вузол, перевіряється два випадки: якщо число менше за поточний вузол, то перезаписуємо поточний вузол як лівий, в іншому випадку перезаписуємо поточний на правий вузол. Так продовжуємо поки поточний вузол не стане **NULL**. У такому разі перезаписуємо значення **NULL** на число, яке необхідно додати;

* + **Червоно-чорне дерево:**

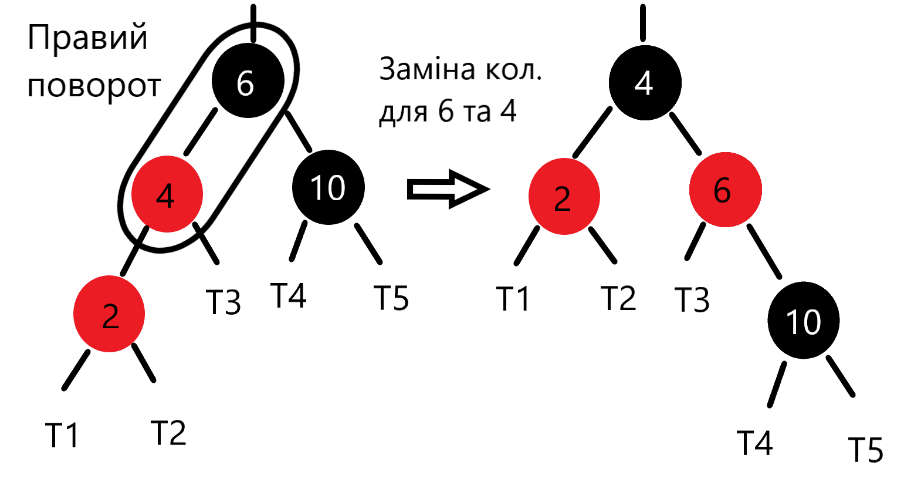
Додавання вузла виконується за тим самим алгоритмом, що і бінарне дерево пошуку, але колір нового доданого вузла є червоний. Потім виконується урегулювання властивостей червоно-чорного дерева за допомогою рекурсії.

Спершу перевірка чи є даний вузол коренем: **так ->** перефарбуємо його у чорний колір, **ні ->** запуск відновлення властивостей.

Алгоритм відновлення властивостей такий:

Допоки поточний вузол не є коренем виконувати такі дії:

* + - Встановлення позиції дядька даного вузла:
    - Якщо дядько **з права** та має чорний колір або є NULL то виконуються такі два випадки:

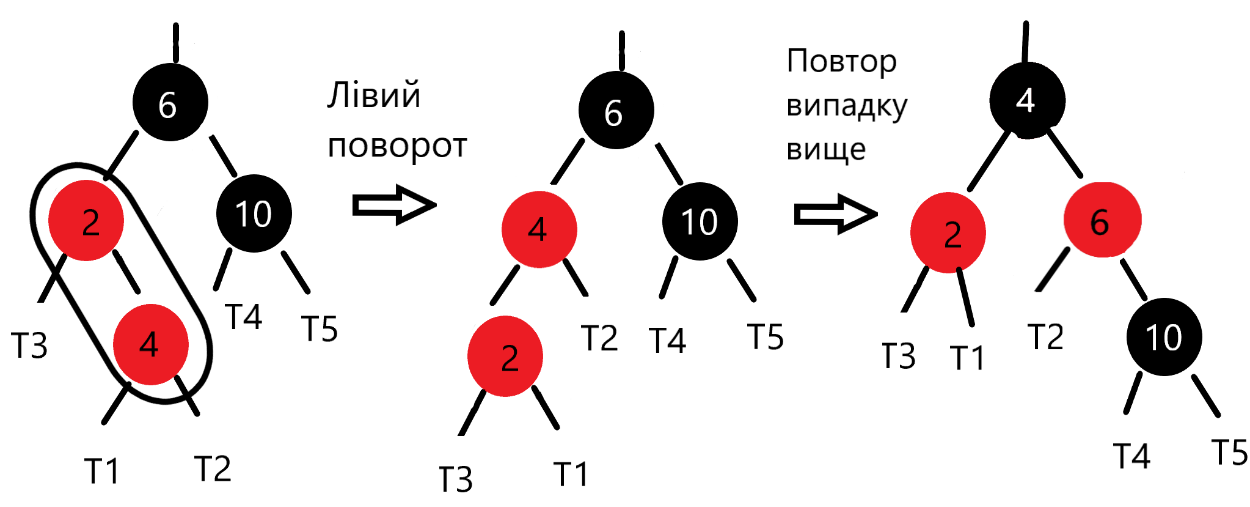
1. ****Якщо поточний вузол є **лівим** нащадком батька(Left-Left):

Виконуємо Правий поворот ;

Перефарбовуємо батька у чорний колір та брата у червоний колір

1. Якщо поточний вузол є **правим** нащадком батька(Left-Right):

Виконуємо Лівий поворот та Правий поворот ;

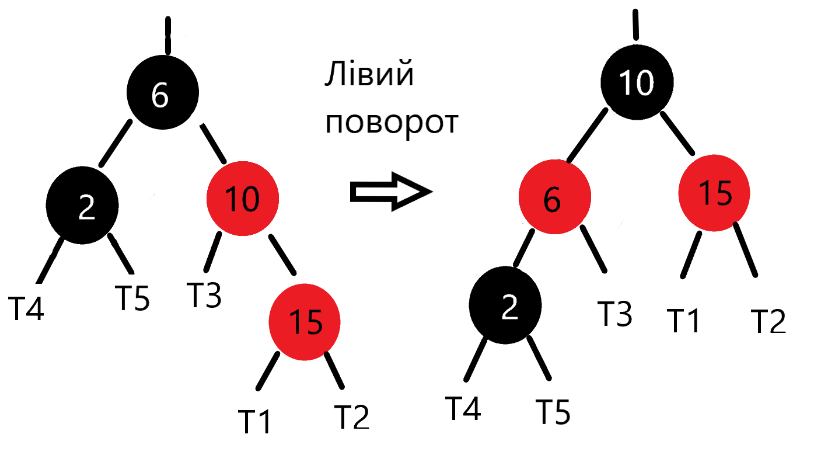
****Перефарбовуємо поточний вузол у чорний колір та правого нащадка у червоний колір;

* + - Якщо дядько **зліва** та має чорний колір або є NULL то виконуються такі два випадки:

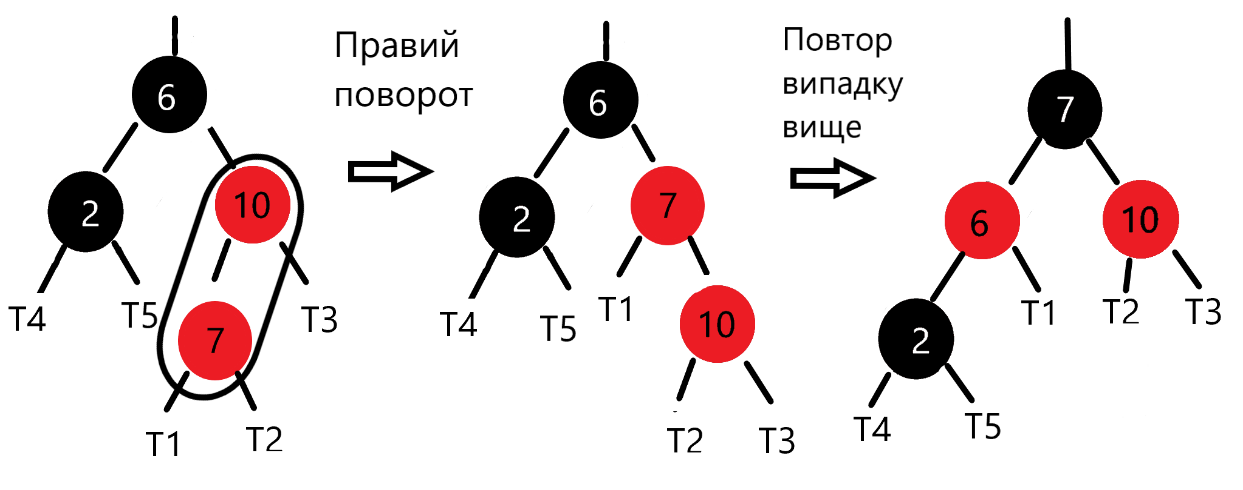
1. Якщо поточний вузол є **правим** нащадком батька(Right-Right):

Виконуємо Лівий поворот ;

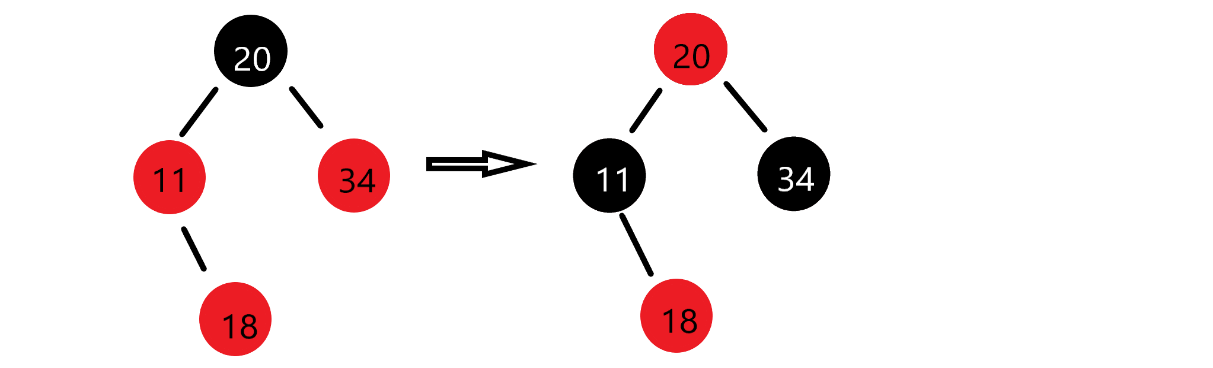
Перефарбовуємо батька у чорний колір та брата у червоний колір



1. Якщо поточний вузол є **лівим** нащадком батька(Right-Left): Виконуємо Правий поворот та Лівий поворот ;

Перефарбовуємо поточний вузол у чорний колір та лівого нащадка у червоний колір;

* + - Якщо дядько має червоний колір то:

 У будь якій ситуації ( не залежить від позиції дядька або позиції поточного вузла ) перефарбовуємо дядька та батька у чорний кольори та дідуся у червоний колір;

* + - Якщо дядько був червоний то запускаємо відновлення властивостей для вузла, що є на позиції дідуся;
    - В іншому випадку запускаємо відновлення властивостей для батька;
  + **АВЛ дерево:**

Додавання вузла виконується за тим самим алгоритмом, що і бінарне дерево пошуку, але встановлюється додаткове поле висоти вузла(за замовчуванням новий доданий вузол має висоту 0). Потім виконується врегулювання збалансованості дерева використовуючи рекурсію. Даний алгоритм використовує трохи змінену версію Лівого та правого поворотів. В кінці методу додається оновлення висот вузлів, що повертались.

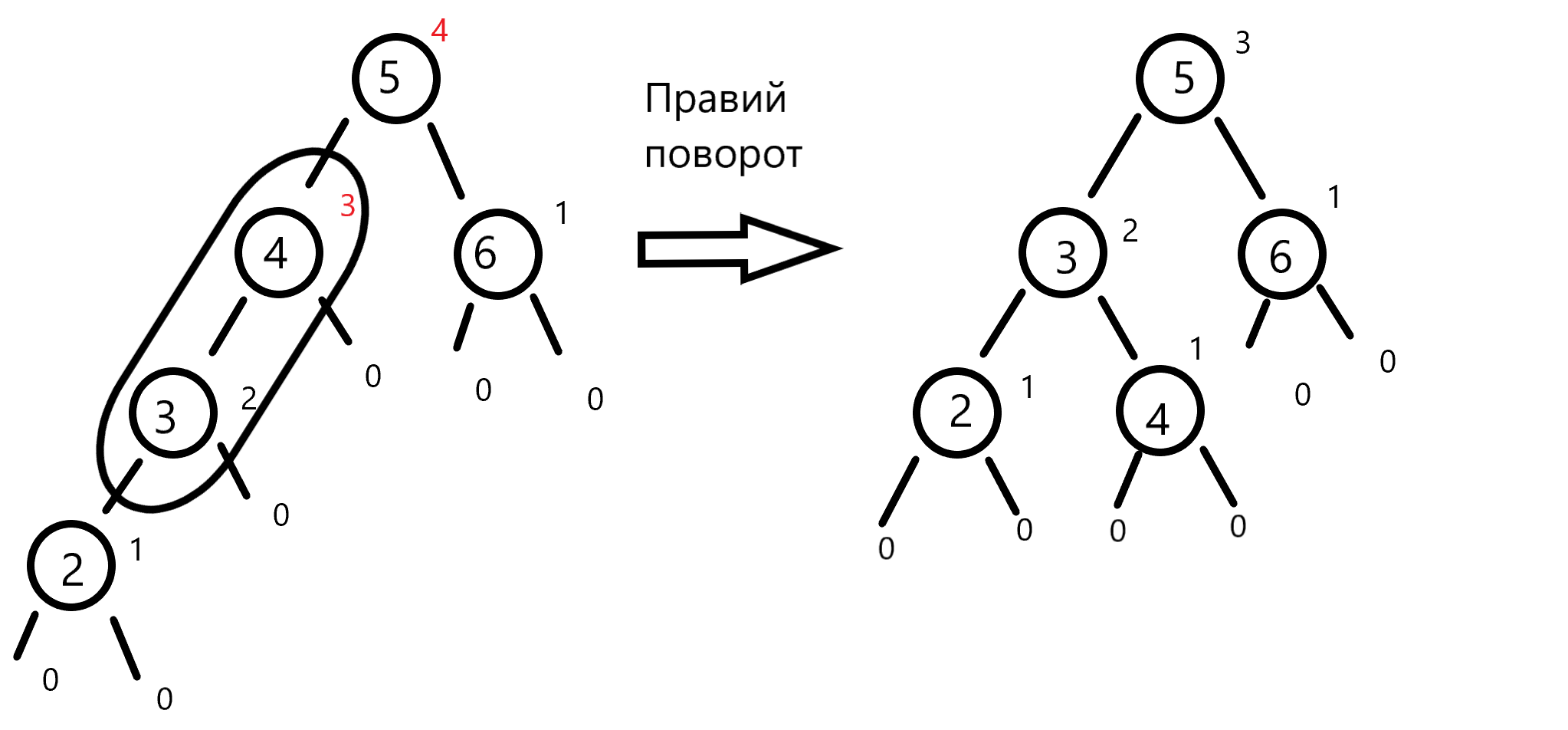
Алгоритм відновлення збалансованості такий:

* + - Якщо поточний вузол не є NULL то починаємо відновлення;
    - Першим кроком є встановлення висоти поточного вузла:

Висота вузла = 1 + висота лівого або правого під-дерева(вибирається найвище з двох)

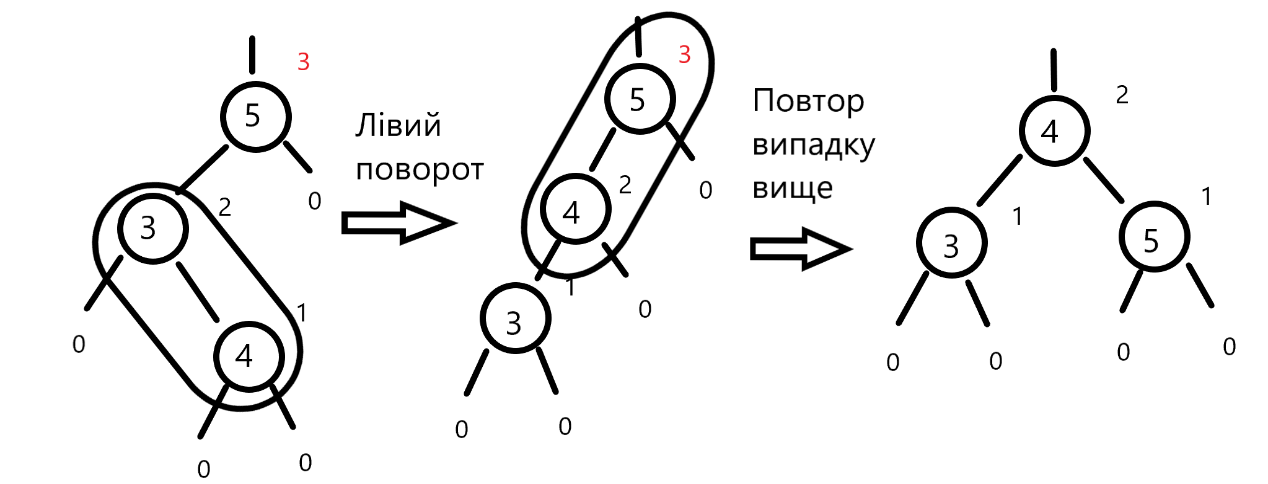
* + - Наступним є перевірка збалансованості вузла (збалансованість визначається різницею висот під-дерев поточного вузла «лівий – правий»);
    - Якщо фактор збалансованості поточного вузла > 1 (тобто висота лівого під-дерева вища за висоту правого більше ніж на 1):
    - Якщо поточний вузол має два послідовних нащадки зліва:

Виконуємо Правий поворот для поточного вузла;



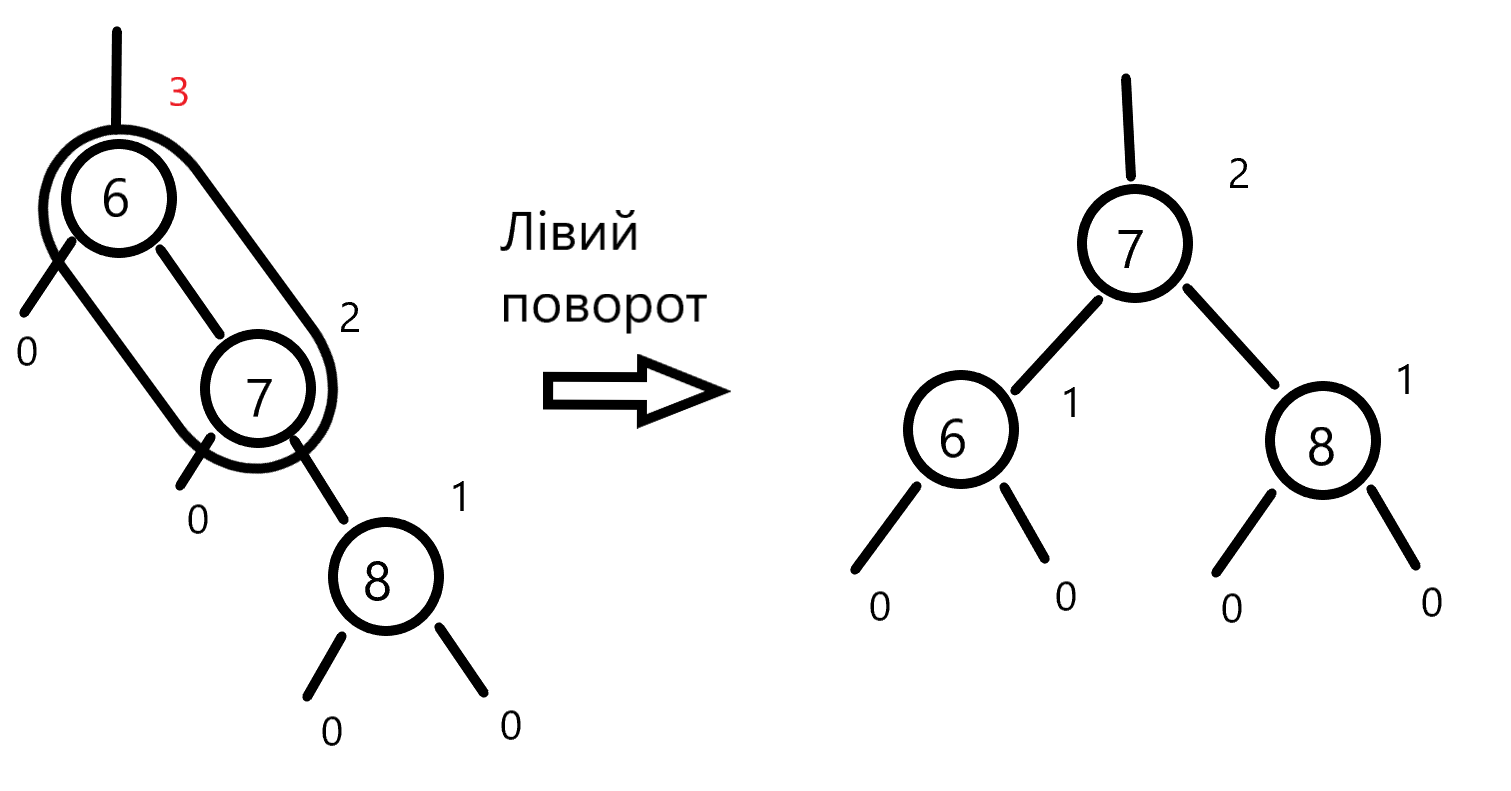
* + - В іншому випадку:

Виконуємо Лівий та Правий повороти для лівого під-дерева;

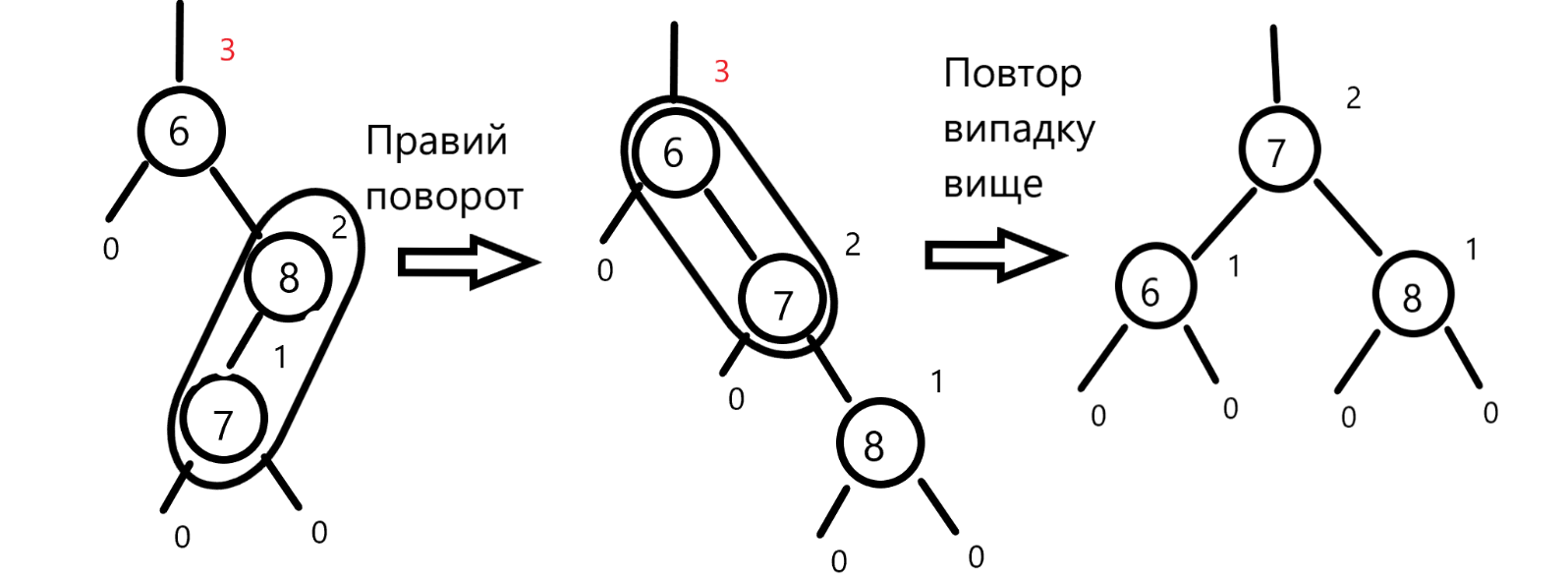


* + - Якщо фактор збалансованості поточного вузла < -1 (тобто висота правого під-дерева вища за висоту лівого під-дерева більше ніж на 1) :
    - Якщо поточний вузол має два послідовних нащадки з права:

Виконуємо Лівий поворот для поточного вузла;

****

* + - В іншому випадку:

Виконуємо Правий та Лівий повороти для правого нащадка;

* + - Перевірка збалансованості для батька поточного вузла;
* **Видалення**
  + **Бінарне дерево пошуку:**

Спершу виконується пошук потрібного вузла. Якщо вузол знайдено то розпочинається перевірка на кілька випадків:

* + - Якщо вузол не має нащадків:
      * Якщо вузол є лівим нащадком батька, то замінюємо лівий покажчик батька на NULL;
      * Якщо вузол є правим нащадком батька, то замінюємо правий нащадок батька на NULL;
      * Якщо вузол є коренем то замінюємо його значення на NULL
    - Якщо вузол має тільки лівого нащадка:
      * Якщо вузол є лівим нащадком батька:

То призначаємо батька лівого під-дерева до батька шуканого вузла і лівий покажчик батька до лівого під-дерева;

* + - * Якщо вузол є правим нащадком батька:

Виконуємо ті ж самі операції, що у випадку вище, але у цьому разі ліве під-дерево призначаємо як правий покажчик батька;

* + - Якщо вузол має тільки правого нащадка:
      * Виконуються ті ж самі операції що у випадку з лівим нащадком, але інвертуються усі позиції призначення;
    - Якщо вузол має двоє нащадків:
      * Призначаємо значення правого під-дерева до тимчасового вузла;
      * Якщо у правого під-дерева немає нащадків, то змінюємо число вузла який ми хочемо видалити на число тимчасового вузла та покажчик на правий нащадок шуканого вузла змінюємо на NULL;
      * В іншому випадку перевіряємо які є нащадки в правому під-дереві;
      * Якщо є тільки праві нащадки, то змінюємо значення поточного вузла на значення правого нащадка, та виконуємо його видалення(те ж саме, що у третьому випадку);
      * Якщо є ліві нащадки в правому під-дереві то знаходимо найменше значення у правому під-дереві;
      * Пошук виконується за допомогою рекурсії, де найменше значення у дереві є вузол який є самим лівим;
      * Перевіряємо чи є у найменшого вузла правий нащадок
      * Якщо так то призначаємо значення найменшого до поточного. Та видаляємо цей найменший вузол(те ж саме, що у третьому випадку);
      * В іншому випадку знов призначаємо значення найменшого до шуканого та виконуємо видалення найменшого за першим випадком.
  + **Червоно-чорне дерево:**

Видалення у даному дереві дещо відрізняється від інших бінарних дерев. Для видалення необхідно знайти поточний вузол, створити тимчасові вузли які прийматимуть значення вузлів(Y & X де Y прийматиме або значення найменшого з ліва нащадка або значення поточного та Х прийматиме значення або NULL або правого нащадка від Y), та в залежності від кількості та позиції нащадків виконати такі дії:

* + - Якщо вузол має одного нащадка:
      * Вузол Y буде покажчиком на вузол який ми хочемо видалити;
    - В іншому випадку знаходимо наступника поточного вузла(тобто найменшого у лівому під-дереві)
    - Встановлюємо покажчик X;
    - Відрізаємо покажчик на Y через встановлення батьківського покажчика Y на Х;
    - Встановлюємо значення поточного вузла на значення найменшого вузла Y;
    - Перевіряємо чи був вузол Y чорним(аби відновити властивість червоно-чорного дерева):
    - Якщо так то відновимо властивості червоно-чорного дерева :
    - Допоки поточний вузол не є коренем та є чорним:
    - Визначаємо позицію поточного вузла по відношенню до батька:
    - Якщо вузол є лівим нащадком батька:
      * Позиція брата буде правим нащадком батька
      * Якщо колір брата червоний, змінюємо його колір на чорний, колір батька на червоний
      * Виконуємо Лівий поворот для батька
      * Якщо колір обох нащадків брата чорний, то перефарбовуємо брата у червоний колір та переміщуємо поточний вузол до його батька;
      * В іншому випадку то перевіряємо чи є правий нащадок брата чорним.
      * Якщо так то перефарбовуємо лівий нащадок брата у чорний колір та брата у червоний
      * Виконуємо Правий поворот для брата
      * Потім перефарбовуємо брата у червоний колір, а батька і правого нащадка брата у чорний
      * Виконуємо Лівий поворот для батька
    - Якщо вузол є правим нащадком батька то виконуємо всі ті самі операції але замінюємо Лівий на Правий та Правий на Лівий повороти і інвертуємо усі значення напрямку покажчиків вузлів;

* + - Після обходу всього дерева перефарбовуємо поточний вузол у чорний:

* + **АВЛ дерево**

Видалення виконується за тим самим принципом що і видалення в бінарному дереві пошуку із додатковою перевіркою знайденого вузла. Алгоритм перевірки для відновлення властивості АВЛ дерева розглянуто у алгоритмі для додавання вузлів до АВЛ дерева.

* **Пошук:**

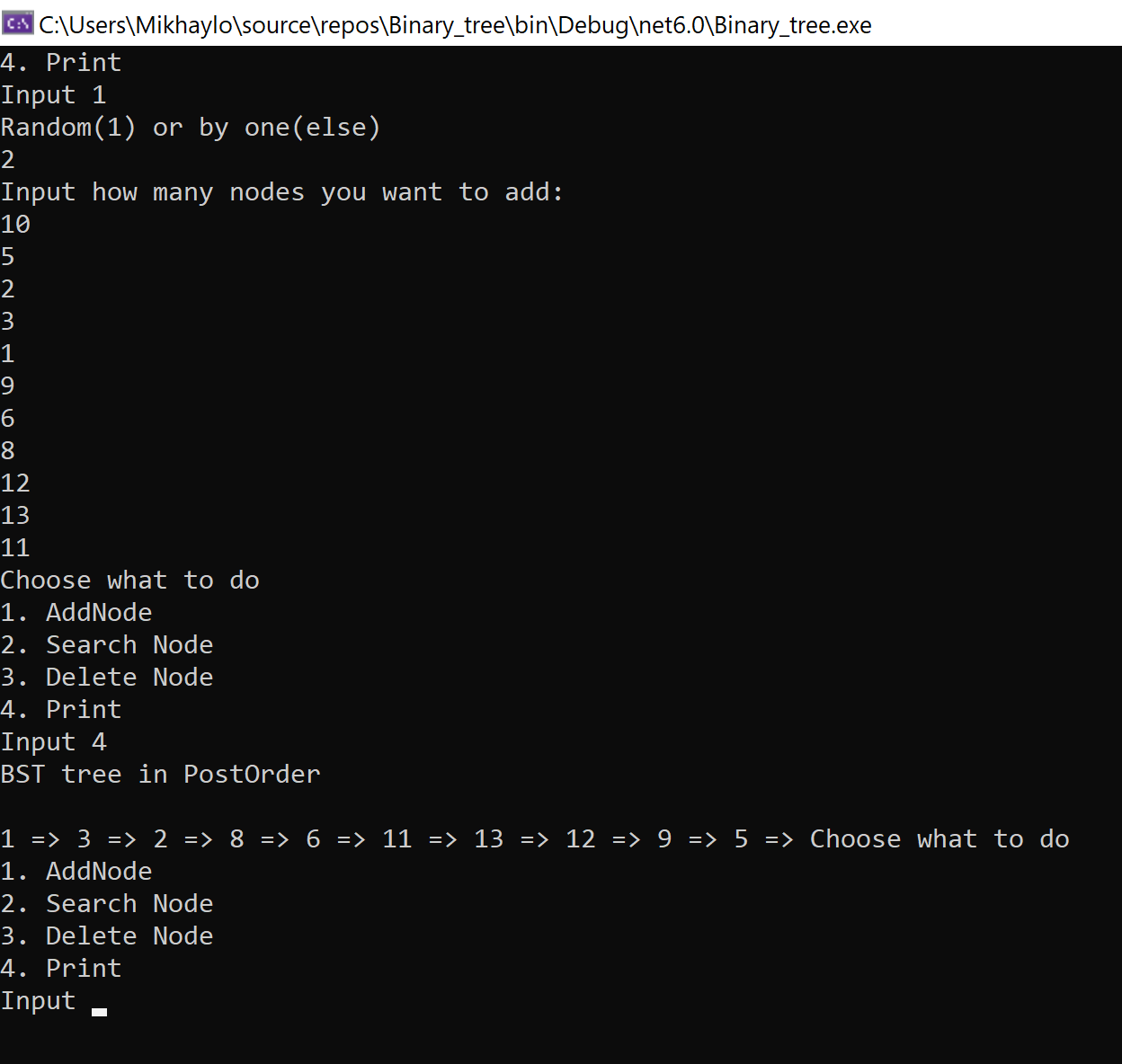
Пошук у будь-якому бінарному дереві виконується однаково. Викликається метод пошуку який у свою чергу повертає метод з атрибутами числа та поточного вузла. При кожному порівнянні числа із значенням поточного вузла знову викликається поточний метод але у аргумент з поточним вузлом записується наступний вузол(лівий або правий нащадок). Якщо значення співпадають із шуканим числом то рекурсія закінчується та повертається знайдений вузол.

* **Обхід дерева:**
  + Префіксний:
    - Перевіряємо, чи не є поточний вузол порожнім або null. Якщо так – то кінець обходу
    - Показуємо поле даних кореня (або поточного вузла).
    - Обходимо ліве піддерево рекурсивно, викликавши функцію прямого обходу
    - Обходимо праве піддерево рекурсивно, викликавши функцію прямого обходу
  + Постфіксний:
    - Перевіряємо, чи не є поточний вузол порожнім або null. Якщо так – то кінець обходу.
    - Обходимо ліве піддерево рекурсивно, викликавши функцію прямого обходу
    - Обходимо праве піддерево рекурсивно, викликавши функцію прямого обходу
    - Показуємо поле даних кореня (або поточного вузла)

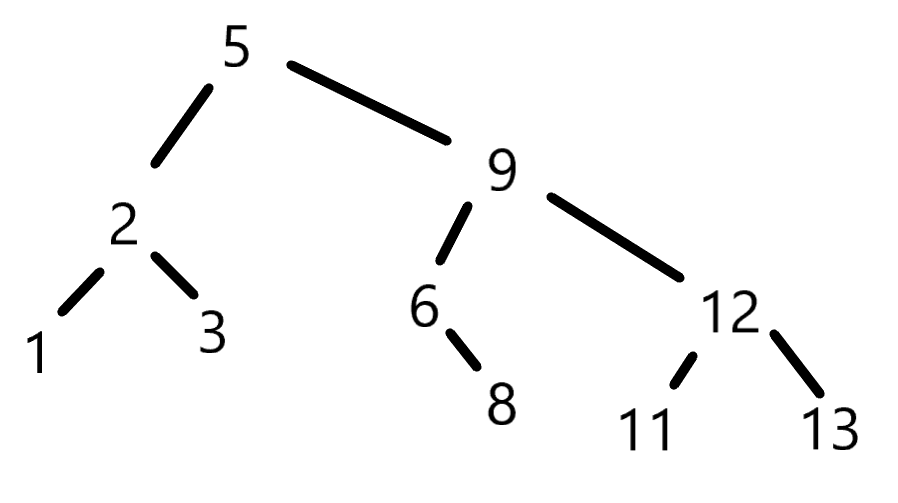
**Набір тестів:**

10 елементів

1. Бінарне дерево пошуку:

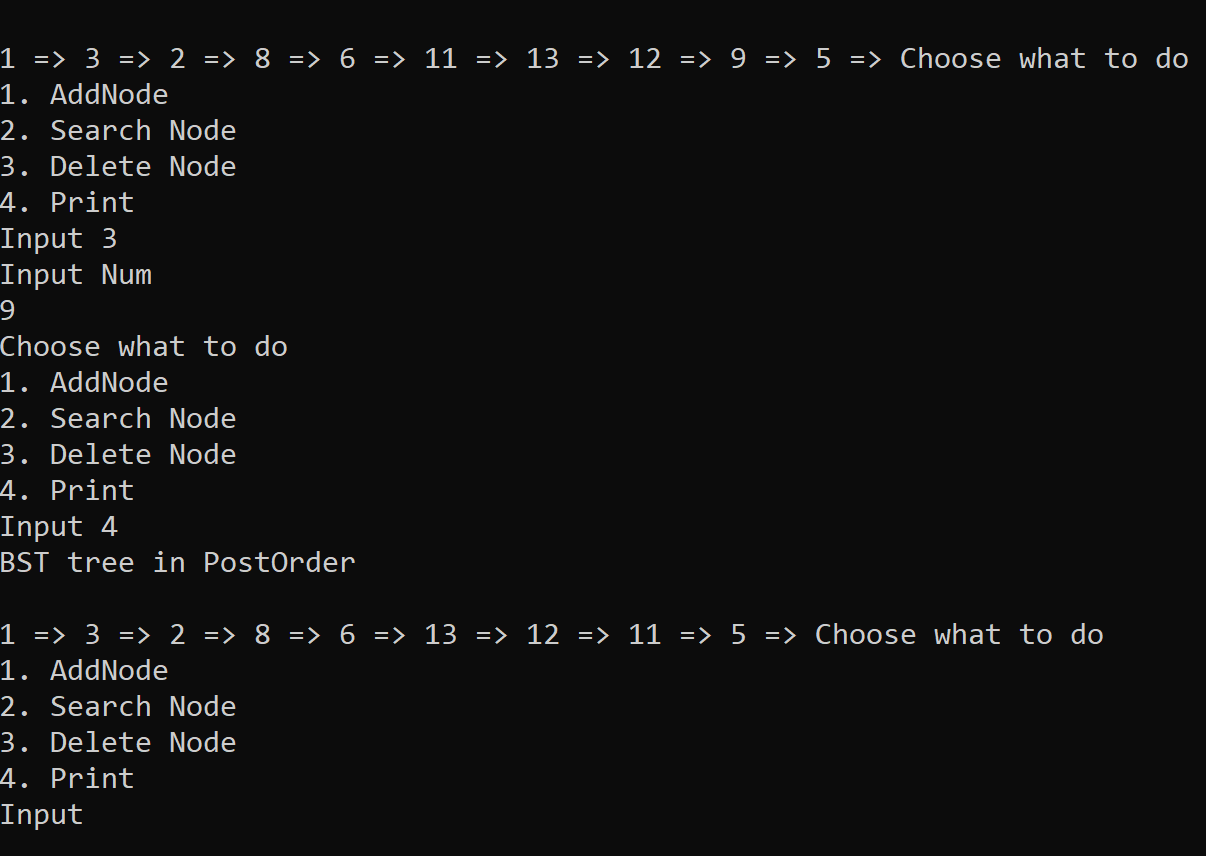


У результаті отримаємо таке дерево

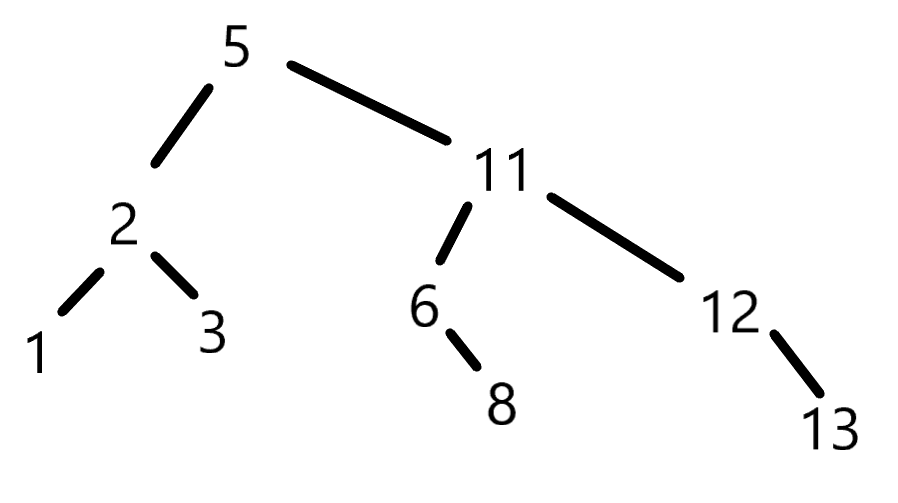


Як видно усі властивості бінарного дерева пошуку збереглись;

Тому спробуємо видалити вузол із значенням 9;



Після видалення вузла 9 отримуємо дерево такого вигляду

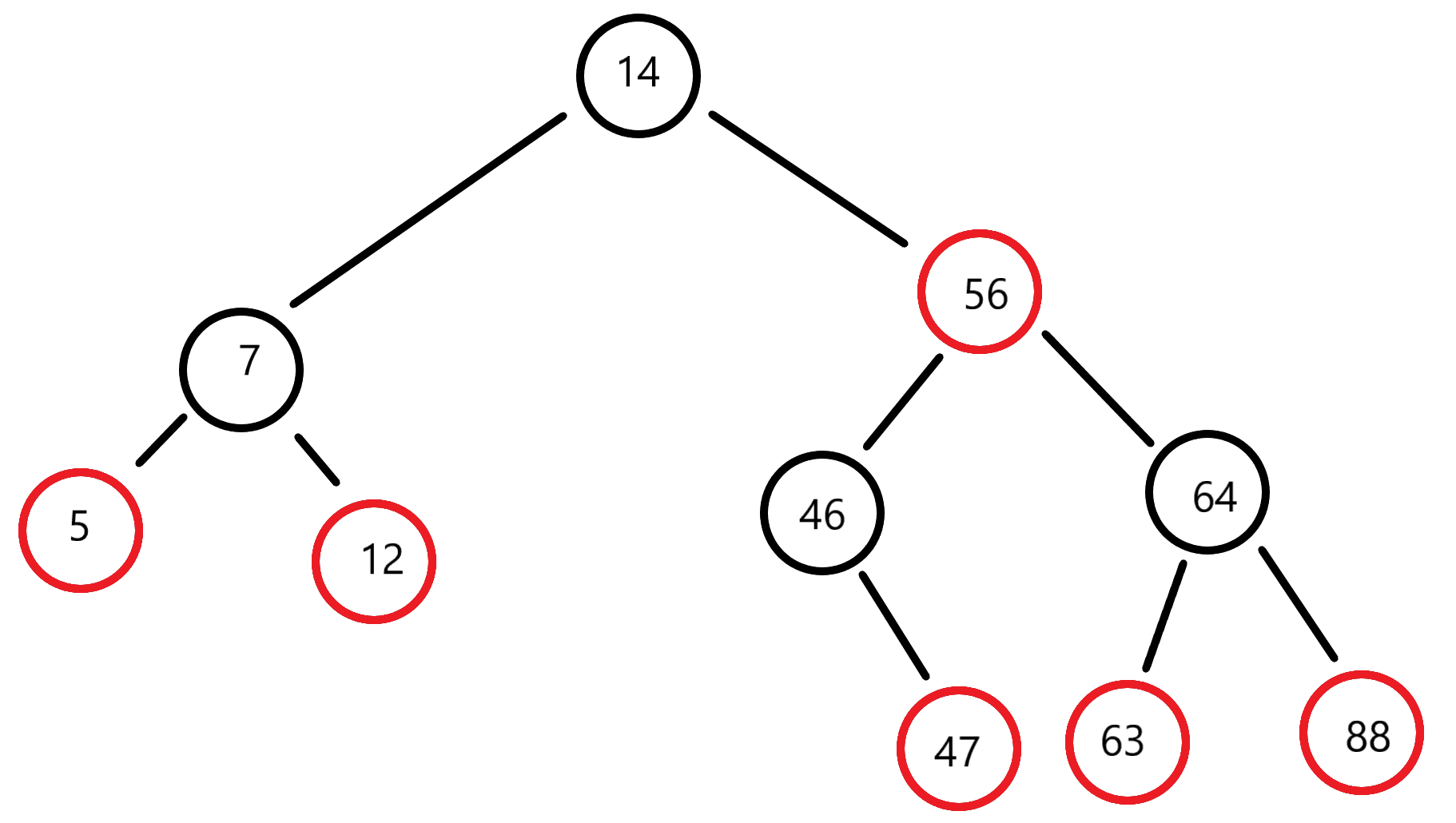


Також усі властивості збережені;

1. Червоно-чорне дерево:

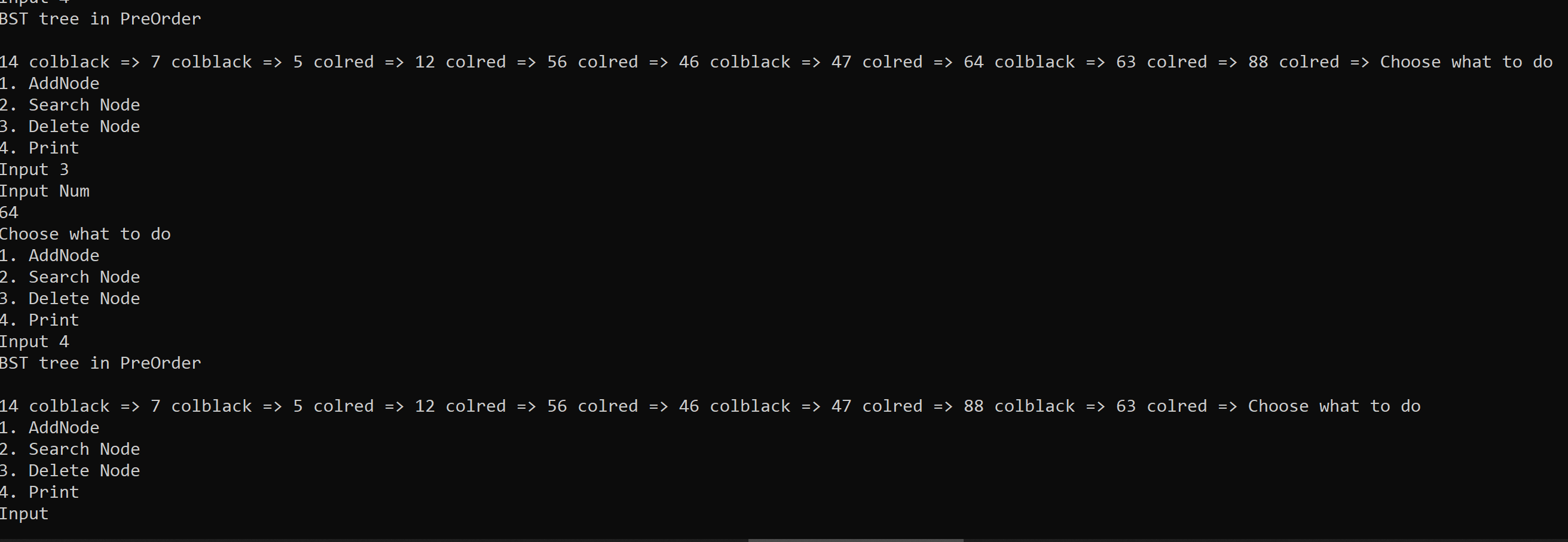


У результаті отримаємо таке дерево

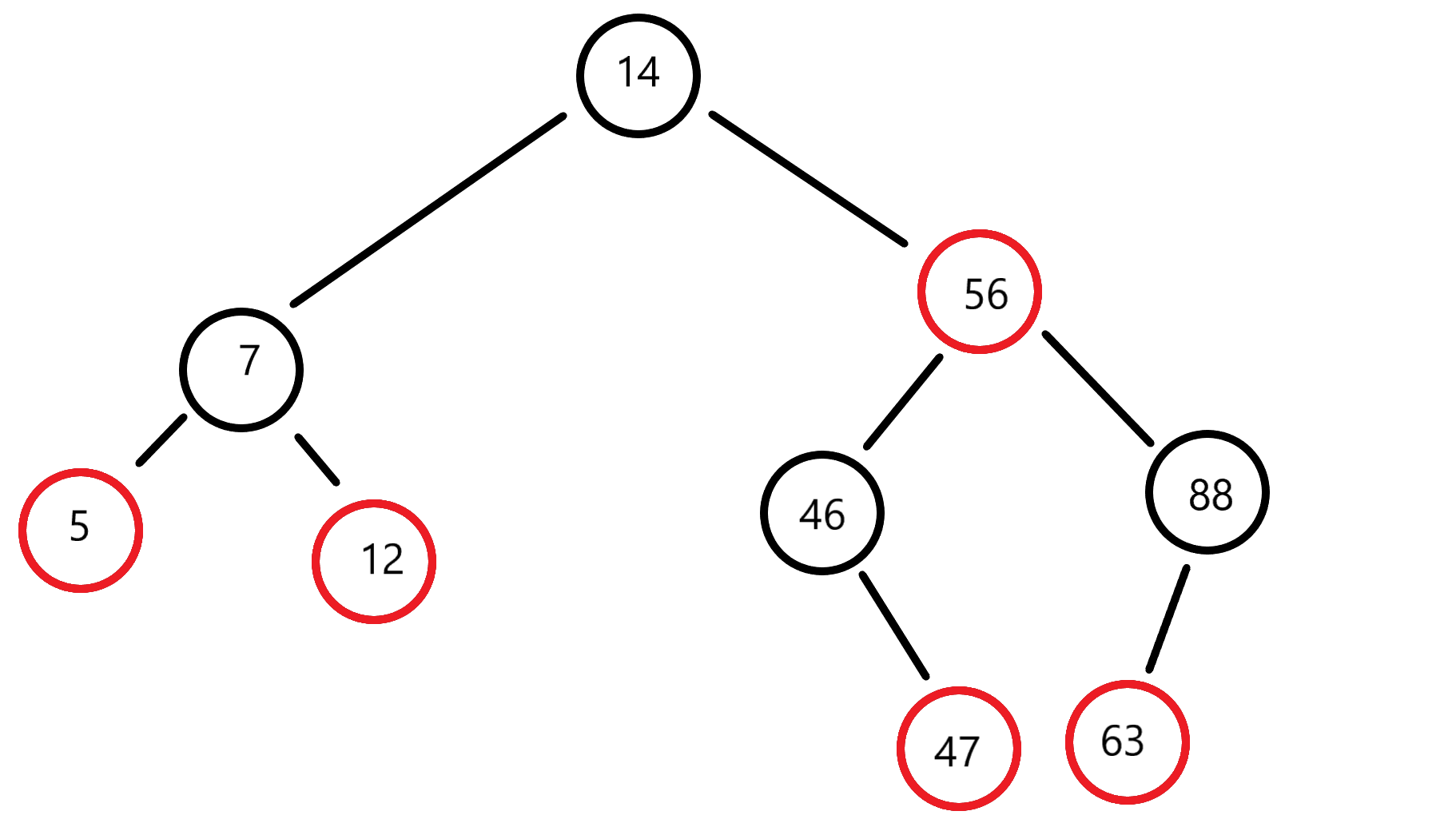


Як видно усі властивості червоно-чорного дерева пошуку збереглись;

Тому спробуємо видалити вузол із значенням 64;

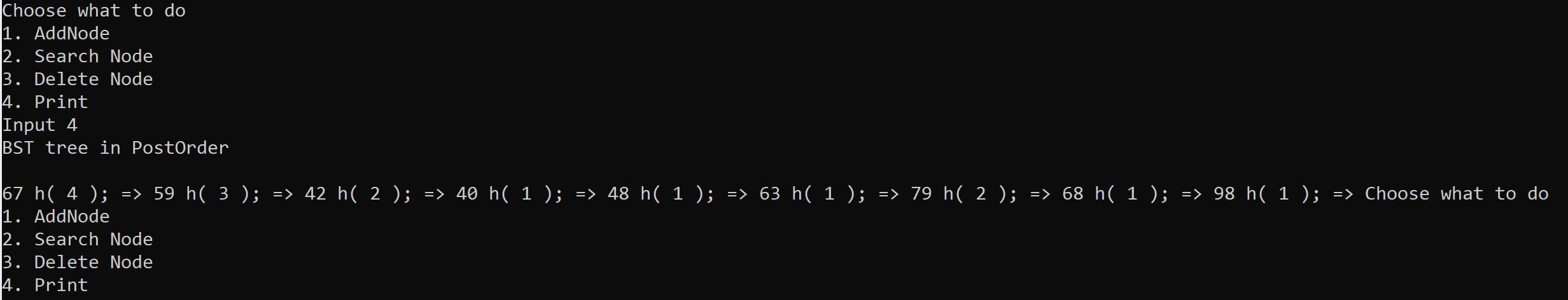


Після видалення вузла 64 отримуємо дерево такого вигляду

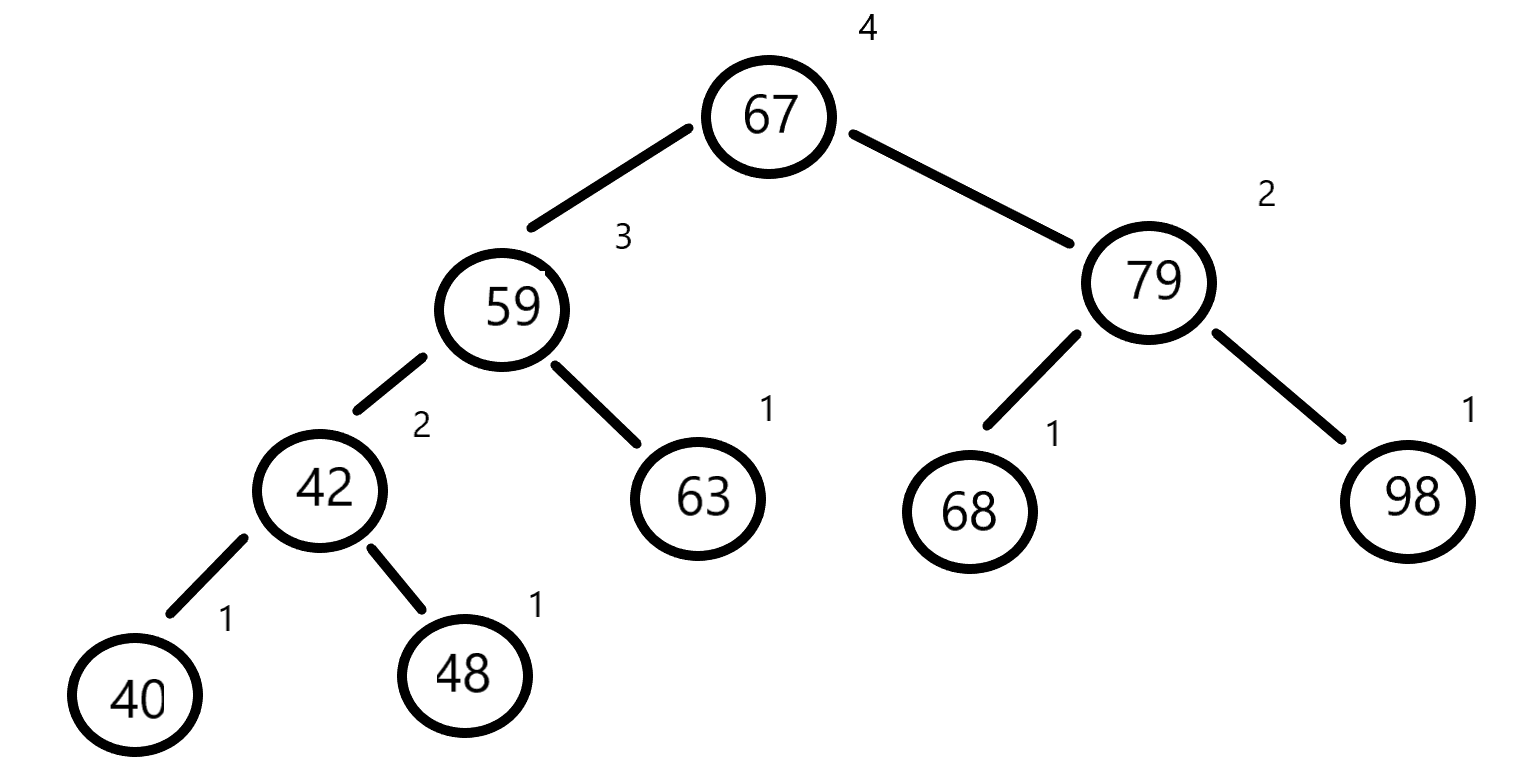


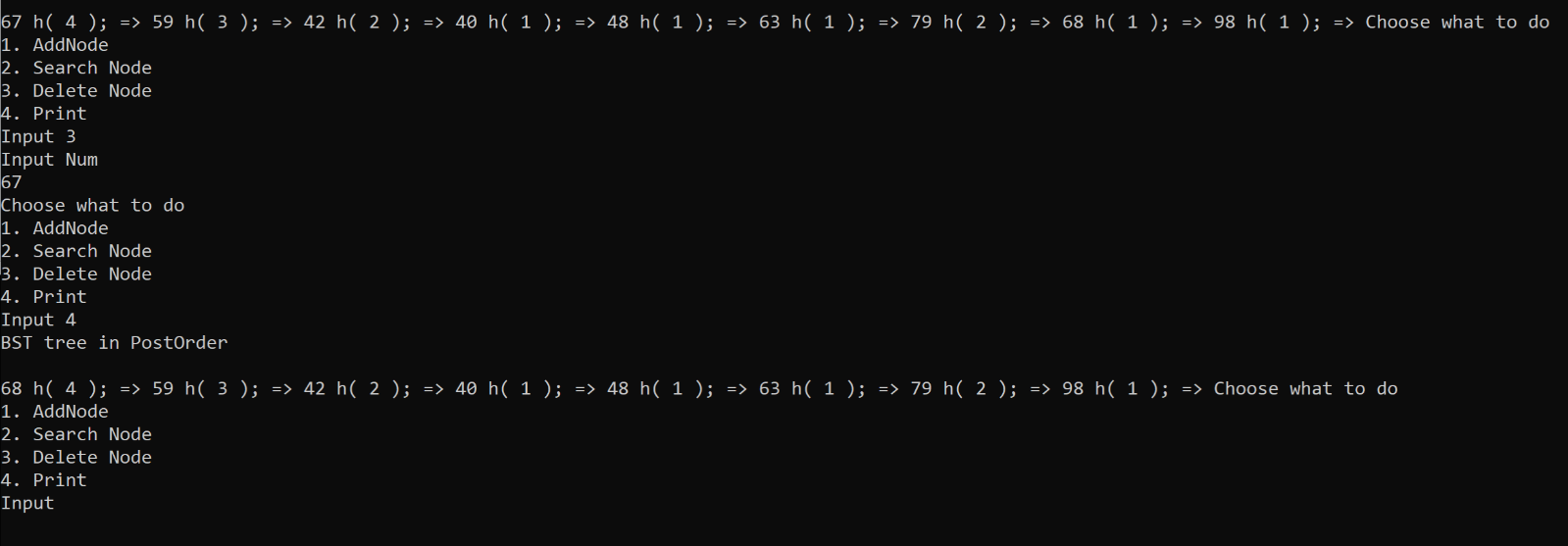
Бачимо, що усі властивості червоно-чорного дерева збереглись.

1. АВЛ дерево

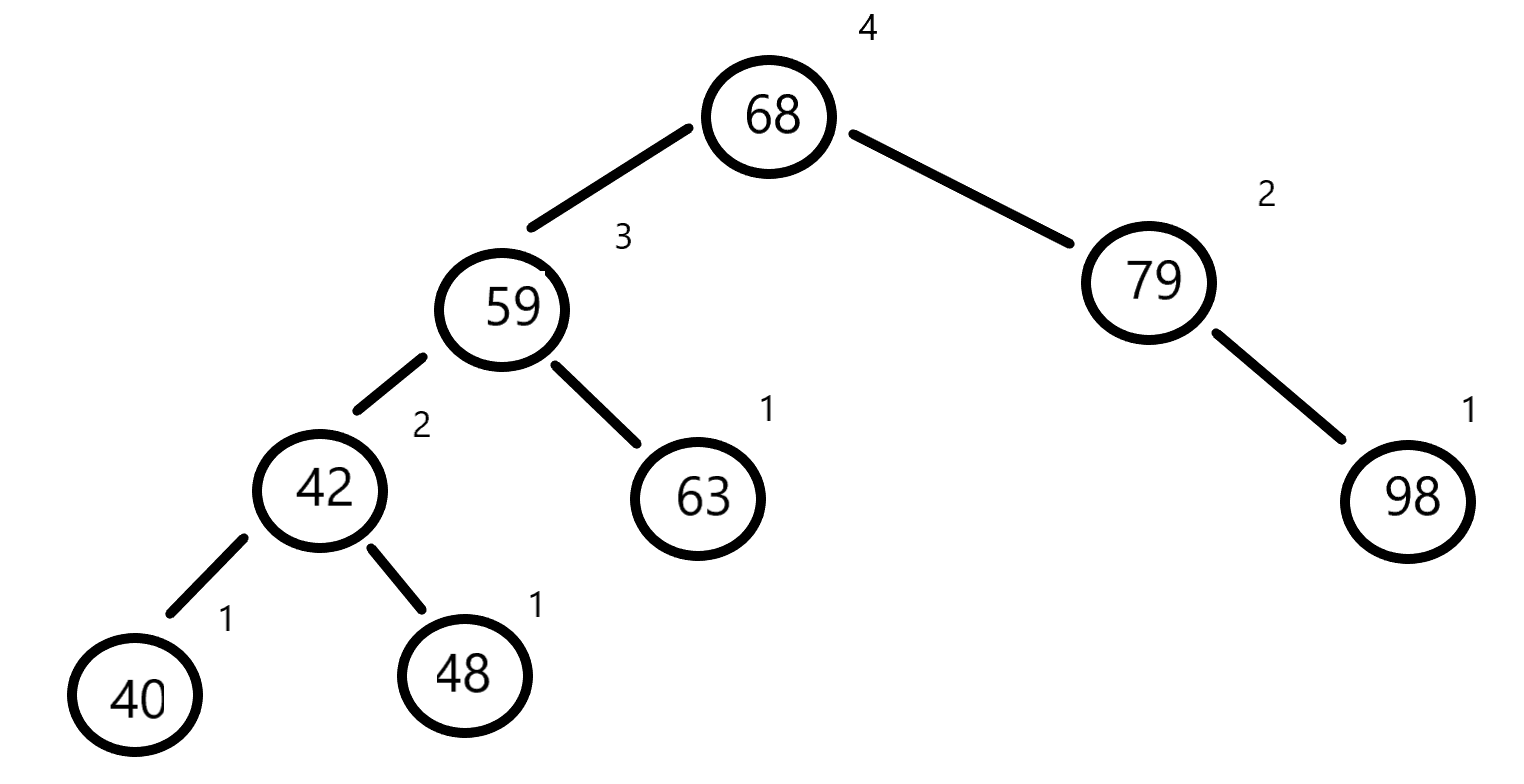


У результаті отримаємо таке дерево:



У результаті отримали АВЛ дерево яке є збалансованим, тому спробуємо видалити корінь дерева 67:

Після видалення вузла 67 отримуємо таке дерево:



Після видалення кореня видно що дерево залишилось збалансованим;

**Аналіз отриманих помилок:**

Протягом усього життєвого циклу розробки даної програми, виявлялись проблеми. В основному це були проблеми у відновленні властивостей дерев. Наприклад у червоно-чорному дереві для видалення та додавання існує багато випадків де треба враховувати не тільки властивість вузла який ми хочемо додати або видалити, а і його «батьків», нащадків та братів (колір та позицію). Також виникали проблеми і в програмуванні АВЛ дерева де треба було правильно визначити висоту поточного вузла та після виконання поворотів треба оновлювати висоти вузлів які повертаються. І в кінці кінців найсумнішим чи може найщасливішим було коли у сотнях рядків коду ти загубив додаткову зміну кольору для дядька, коли треба було викликати метод для знаходження найменшого вузла не в усьому під-дереві, а тільки у правій частині. Хоча дана лабораторна робота виявилась чи не найскладнішою з усього першого курсу, бо було важко в першу чергу зрозуміти що від тебе хочуть використати правильний алгоритм який працюватиме в усіх випадках а не тільки в окремих.

**Висновок:**

Під час виконання даної лабораторної роботи, було опановано знання щодо різних структур даних дерев. Також було опановано навички програмування даних структур даних та методів їх обробки.