Міністерство освіти і науки України
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА
Факультет прикладної математики та інформатики

Кафедра програмування

ЛАБОРАТОРНА РОБОТА № 11

AVL дерева

з курсу "Алгоритми та структури даних"

Виконав:

Студент групи ПМІ-16

Бевз Маркіян Юрійович

Львів – 2024

Мета: Дослідити структуру AVL-дерев, у яких ситуаціях та для чого використовується, написати власну реалізацію.

Теоретичні відомості:

AVL дерева - це двійкові дерева пошуку, в яких різниця між висотою лівого та правого піддерева становить -1, 0 або +1.

Дерева AVL також називають самобалансуючим бінарним деревом пошуку. Ці дерева допомагають підтримувати логарифмічний час пошуку. Він названий на честь своїх винахідників (AVL) Адельсона-Вельського та Ландіса.

Принцип роботи AVL-дерев, навіщо потрібні?

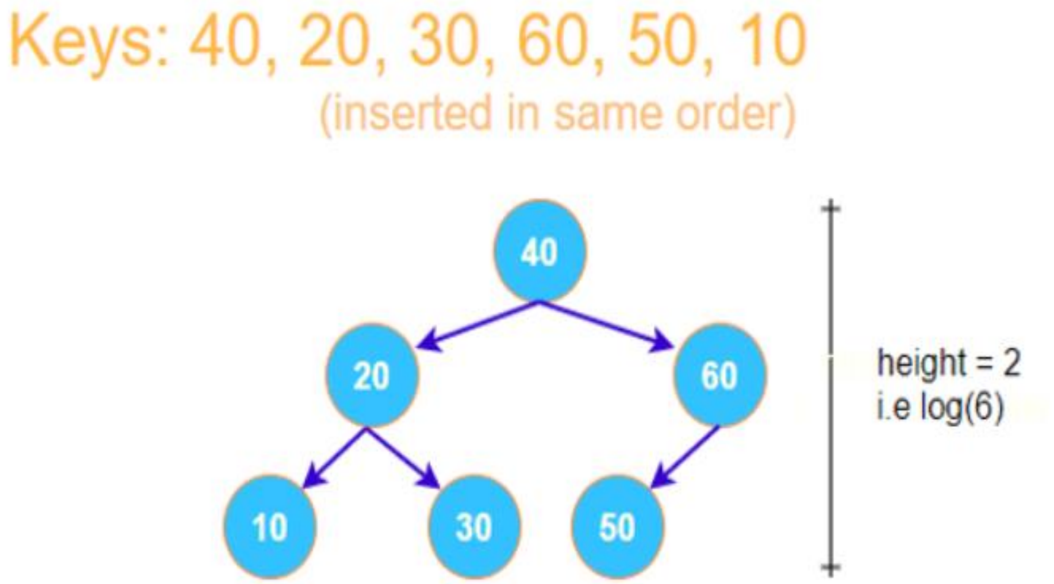
Щоб краще зрозуміти потребу в деревах AVL, розглянемо деякі недоліки простих бінарних дерев пошуку.

Розглянемо наступні ключі, вставлені в заданому порядку в бінарному дереві пошуку.

Висота дерева лінійно зростає, коли ми вставляємо ключі в порядку зростання їх значення. Отже, пошук, у гіршому випадку, приймає **O(n).**

Для пошуку елемента потрібен лінійний час, тому немає сенсу використовувати бінарне дерево пошуку(воно вироджиться у список). З іншого боку, якщо висота дерева збалансована, ми отримуємо кращий часу пошуку.

Давайте тепер розглянемо ті самі ключі, але вставлені в іншому порядку.



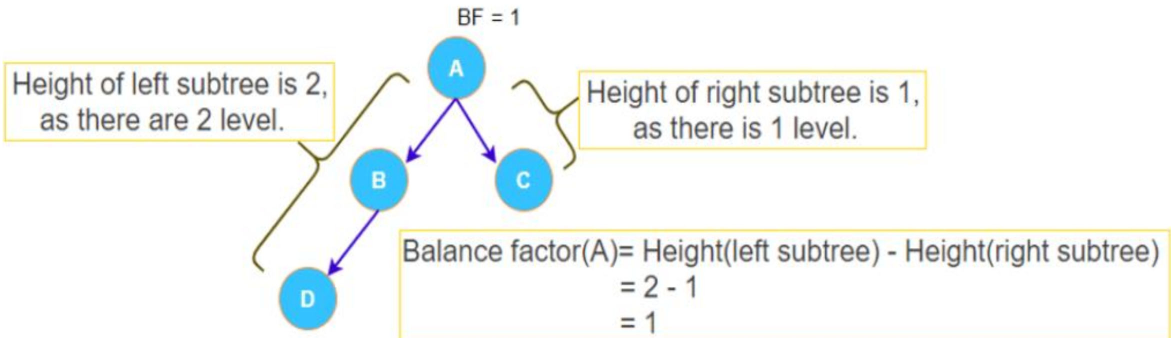
Тут ключі однакові, але оскільки вони вставлені в іншому порядку, вони займають різні позиції, а висота дерева залишається збалансованою. Отже, пошук не займатиме більше O(log n) для будь-якого елемента дерева. Тепер очевидно, що якщо вставлення виконано правильно, висота дерева може бути збалансованою.

У деревах AVL ми перевіряємо висоту дерева під час вставки operації. Внесено зміни, щоб підтримувати збалансовану висоту без порушення основних властивостей бінарного дерева пошуку.

Фактор балансу в деревах AVL

Коефіцієнт балансу (BF) є фундаментальним атрибутом кожного вузла в деревах AVL, який допомагає контролювати висоту дерева.

Властивості фактора балансу:



- **Коефіцієнт балансу відомий як різниця між висотою лівого та правого піддерева.**

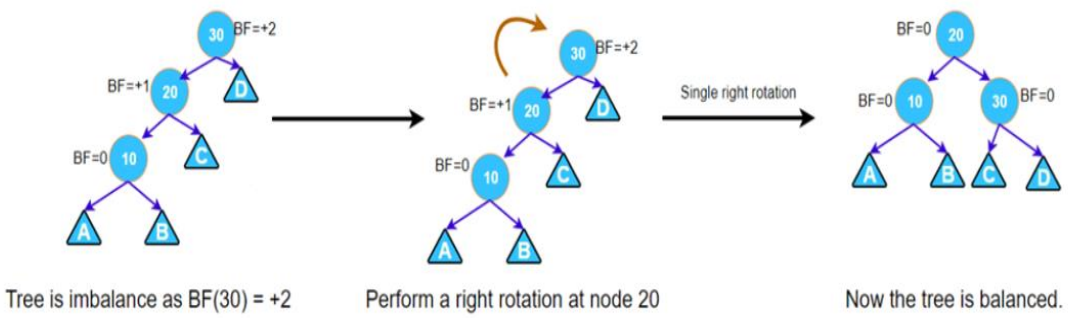
- **Фактор балансу (вузол) = висота (вузол->ліворуч) – висота (вузол->праворуч)**
- **Дозволені значення BF: –1, 0 і +1.**
- **Значення –1 вказує на те, що праве піддерево містить одне додаткове, тобто дерево «важке» справа.**
- **Значення +1 вказує, що ліве піддерево містить одне зайве, тобто дерево залишається «важким».**
- **Значення 0 показує, що дерево містить рівні вузли з кожного боку, тобто дерево ідеально збалансоване.**

Обертання AVL-дерева

Щоб збалансувати дерево AVL, при вставці або видаленні вузла з дерева виконуються обертання.

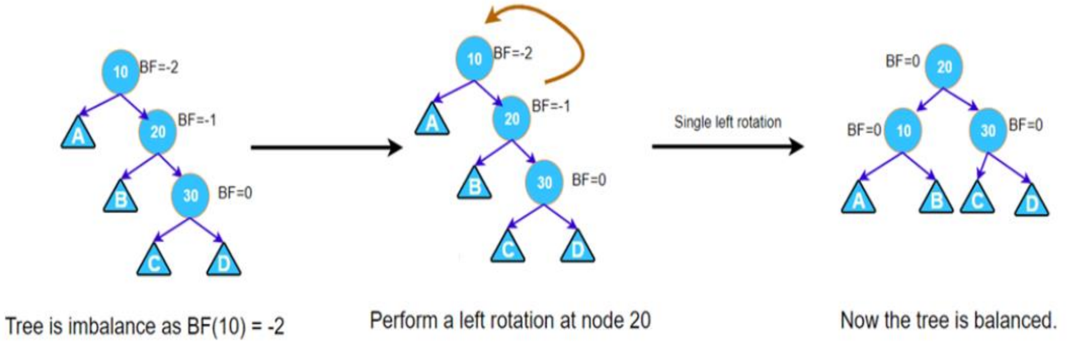
Виконуємо Обертання **LL**, обертання **RR**, обертання **LR** та обертання **RL**.

1. **LL** - Ліве обертання (**Left-Left Rotation**): Це обертання виконується, коли незбалансованість виникає через зростання висоти лівого піддерева лівого нащадка кореня. Ось як це працює:
    - Корінь лівого піддерева стає новим коренем.
    - Піддерево правого сина кореня стає лівим піддеревом попереднього кореня.
    - Корінь дерева стає правим сином нового кореня.

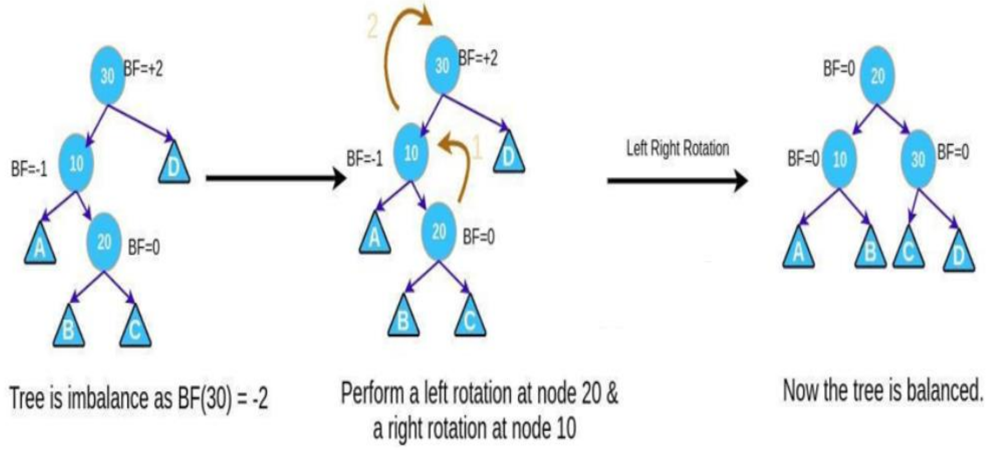


2. **RR** - Праве обертання (**Right-Right Rotation**): Це обертання виконується, коли незбалансованість виникає через зростання висоти правого піддерева правого нащадка кореня. Ось як це працює:
    - Корінь правого піддерева стає новим коренем.
    - Піддерево лівого сина кореня стає правим піддеревом попереднього кореня.

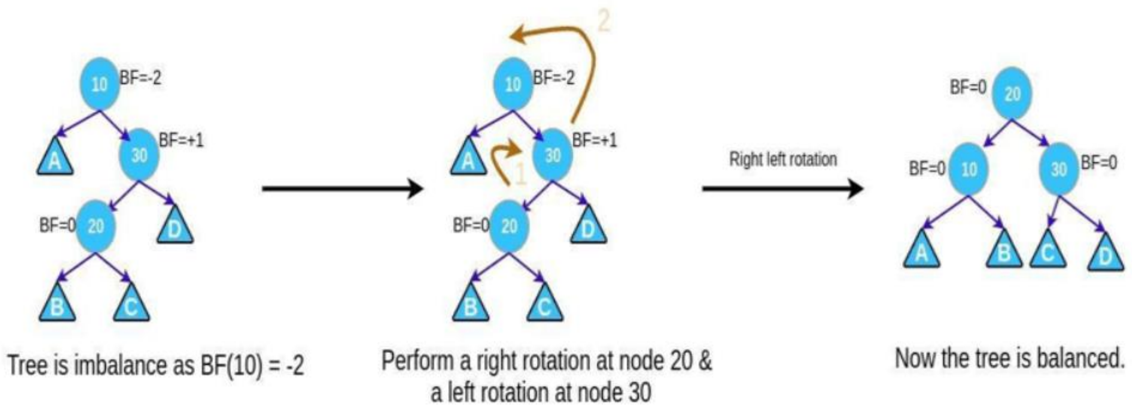
- Корінь дерева стає лівим сином нового кореня.



3. **LR** - Обертання вправо-вліво (**Right-Left Rotation**): Це обертання виконується, коли незбалансованість виникає через зростання висоти правого піддерева лівого нащадка кореня. Ось як це працює:
   - Виконується праве обертання для правого піддерева.
   - Потім виконується ліве обертання для дерева в цілому.

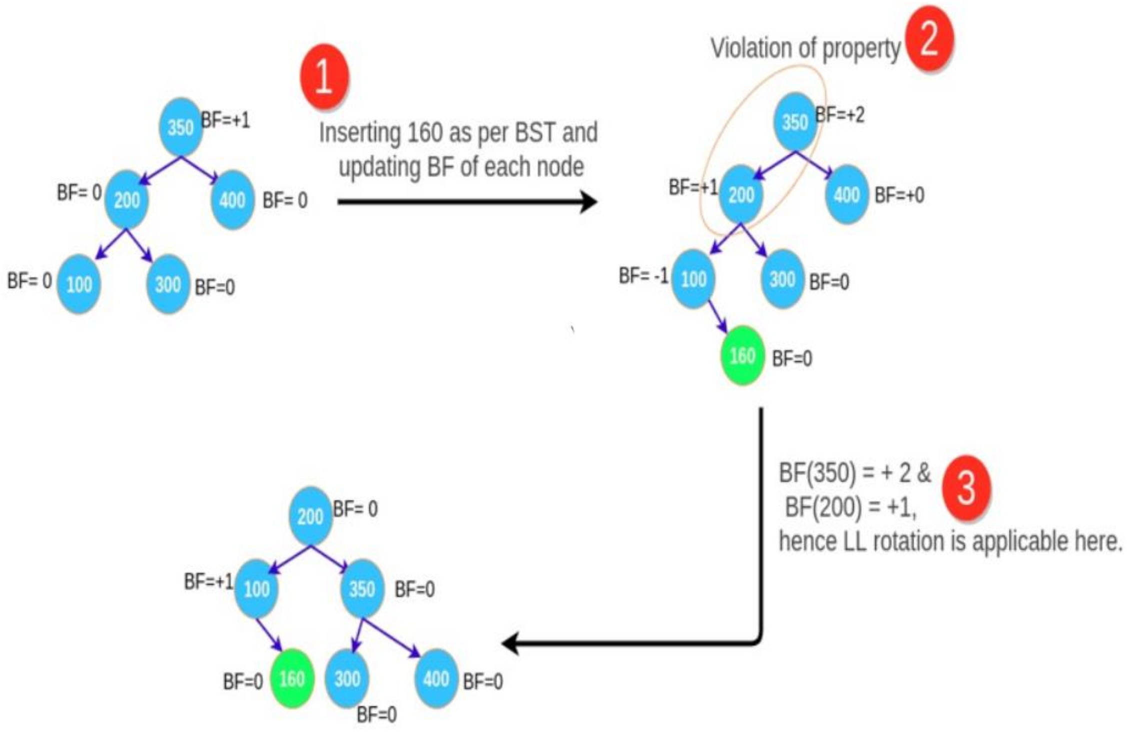


4. **RL** - Обертання вліво-вправо (**Left-Right Rotation**): Це обертання виконується, коли незбалансованість виникає через зростання висоти лівого піддерева правого нащадка кореня. Ось як це працює:
   - Виконується ліве обертання для лівого піддерева.
   - Потім виконується праве обертання для дерева в цілому**.**



Вставка в дерева AVL

Вставка майже така ж, як і в простих бінарних деревах пошуку. Після кожної вставки ми балансуємо висоту дерева. Складність вставки O(log n) .



крок 1: Вставте вузол у дерево AVL, використовуючи той самий алгоритм вставки, що й BST. У наведеному вище прикладі вставте 160.

крок 2: після додавання вузла коефіцієнт балансу кожного вузла оновлюється. Після вставки 160 коефіцієнт балансу кожного вузла оновлюється.

крок 3: Тепер перевірте, чи будь-який вузол порушує діапазон коефіцієнта балансу, якщо коефіцієнт балансу порушено, а потім виконайте обертання, використовуючи наведений нижче випадок. У наведеному вище прикладі коефіцієнт балансу 350 порушується, і тут стає застосовним випадок 1, ми виконуємо обертання LL і дерево знову балансується.

1. Якщо BF(вузол) = +2 і BF(вузол -> лівий дочірній) = +1, виконайте обертання LL.
2. Якщо BF(node) = -2 і BF(node -> right-child) = 1, виконайте обертання RR.
3. Якщо BF(вузол) = -2 і BF(вузол -> правий дочірній) = +1, виконайте поворот RL.
4. Якщо BF(вузол) = +2 і BF(вузол -> лівий дочірній) = -1, виконайте обертання LR.

Видалення в деревах AVL

Видалення також є дуже простим. Ми видаляємо за тією ж логікою, що й у простих бінарних деревах пошуку. Після видалення ми за потреби реструктуруємо дерево, щоб зберегти його збалансовану висоту.

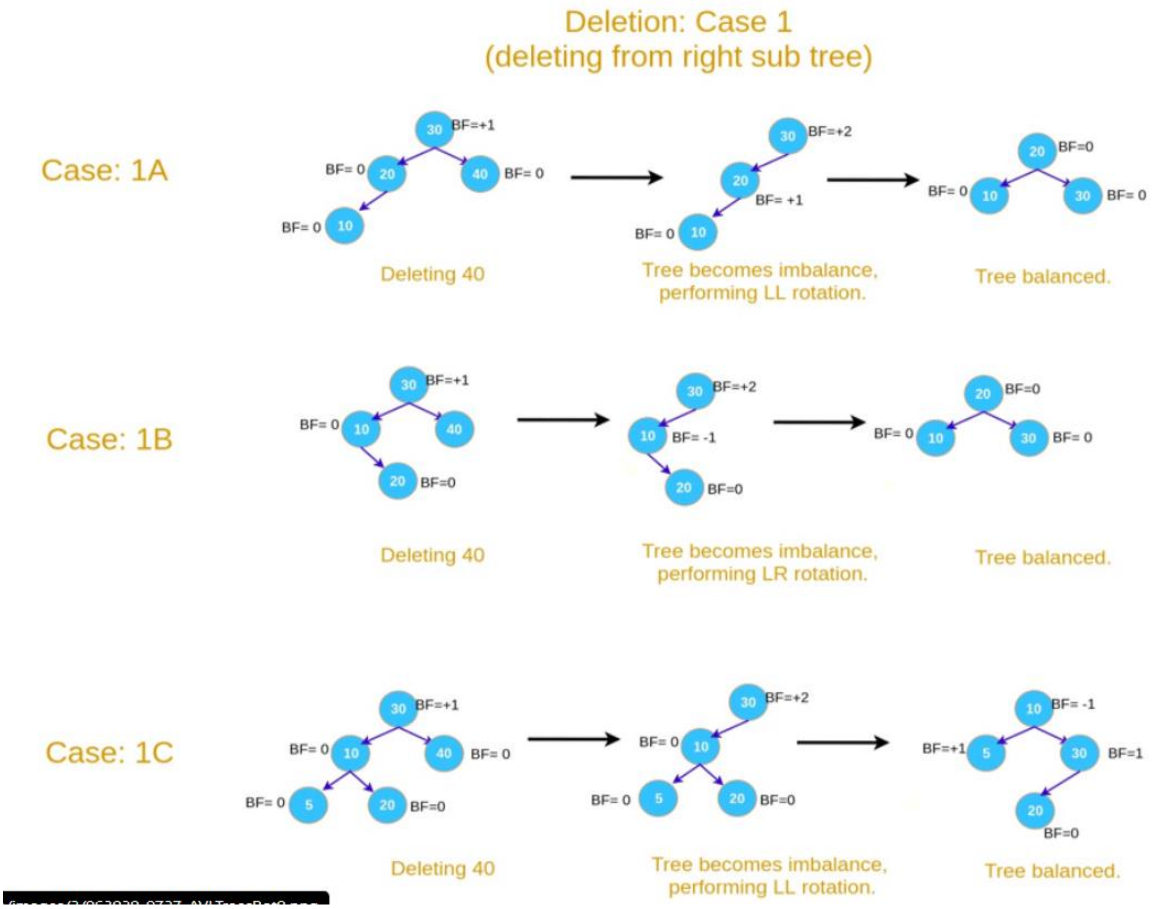
Крок 1: Знайдіть елемент у дереві.

Крок 2: Видаліть вузол відповідно до видалення BST.

Крок 3: Можливі два випадки:

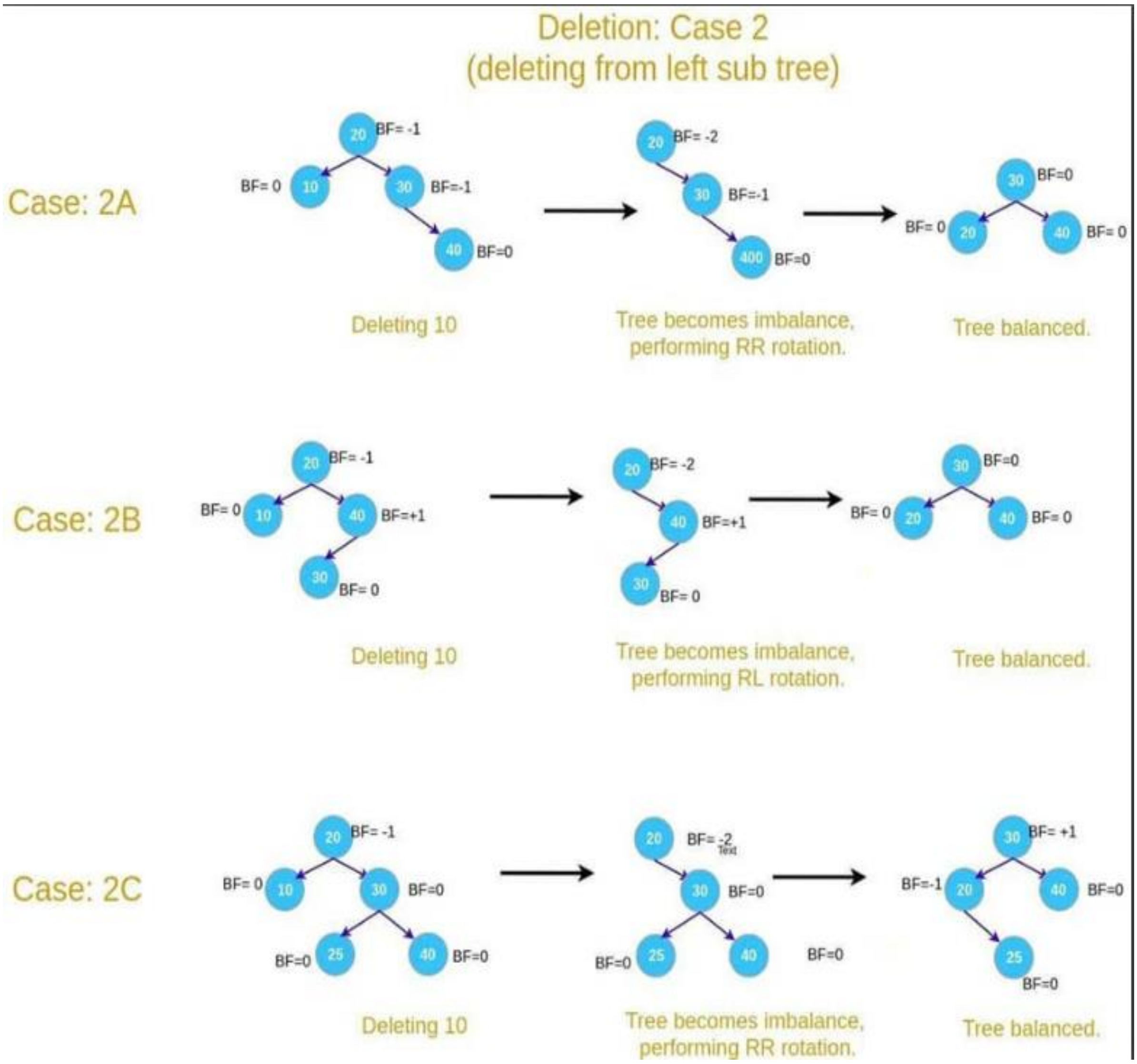
Випадок 1: Видалення з правого піддерева.

- 1A. Якщо BF(вузол) = +2 і BF(вузол -> лівий дочірній) = +1, виконайте обертання LL.
- 1B. Якщо BF(вузол) = +2 і BF(вузол -> лівий дочірній) = -1, виконайте обертання LR.
- 1C. Якщо BF(вузол) = +2 і BF(вузол -> лівий дочірній) = 0, виконайте обертання LL.



Випадок 2: Видалення з лівого піддерева.

- 2A. Якщо BF(node) = -2 і BF(node -> right-child) = -1, виконайте обертання RR.
- 2B. Якщо BF(вузол) = -2 і BF(вузол -> правий дочірній) = +1, виконайте поворот RL.
- 2C. Якщо BF(node) = -2 і BF(node -> right-child) = 0, виконайте обертання RR.



Хід роботи:

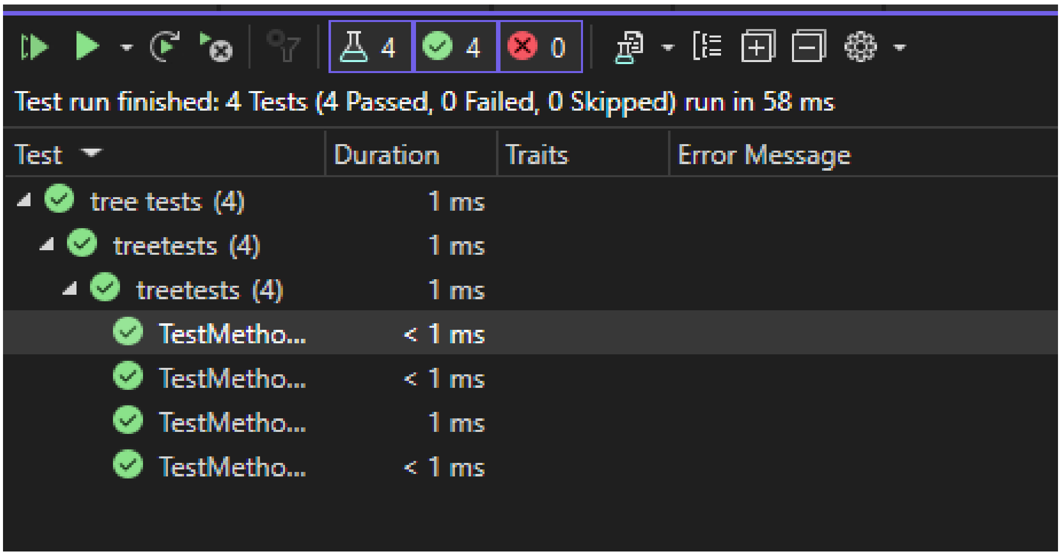
Після вивчення теоретичної частини та принципу роботи алгоритму **AVL-дерева** я реалізував його. Також було створено 4 тести для перевірки правильності роботи функцій у різних випадках. Нижче буде прикріплено результат виконання програми, тестів, та їх код.

```
Enter values that will be stored in AVL-tree (type "stop" to stop):
11 13 9 34 5 22 1 stop
AVL Tree (Graph representation) before deletion:

                    *34

          *22

                    *13

*11

                    *9

          *5

                    *1
Enter the element you want to delete: 22
AVL Tree (Graph representation) after deletion:

          *34

                    *13

*11

                    *9

          *5

                    *1
No such value in tree (12)

D:\c++\self-balancing AVL tree\x64\Debug\self-balancing AVL tree.exe (process 37148) exited with code 0.
Press any key to close this window . . .
```

```cpp
TEST_METHOD(TestMethod1)
{
    // Створення дерева
    Node* root = nullptr;

    // Вставка елементів
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 15);
    root = insert(root, 5);

    // Перевірка коректності вставки
    Assert::IsTrue(search(root, 10) != nullptr);
    Assert::IsTrue(search(root, 20) != nullptr);
    Assert::IsTrue(search(root, 30) != nullptr);
    Assert::IsTrue(search(root, 15) != nullptr);
    Assert::IsTrue(search(root, 5) != nullptr);

    // Видалення елемента
    root = deleteNode(root, 20);

    // Перевірка видалення елемента
    Assert::IsNull(search(root, 20));
}
```

```cpp
TEST_METHOD(TestMethod2)
{
    // Створення дерева
    Node* root = nullptr;

    // Вставка елементів
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 25);

    // Перевірка балансування
    Assert::AreEqual(height(root), 3);
    Assert::AreEqual(getBalance(root), -1);
}

TEST_METHOD(TestMethod3)
{
    // Створення дерева
    Node* root = nullptr;

    // Вставка елементів
    root = insert(root, 5);
    root = insert(root, 3);
    root = insert(root, 8);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 7);
    root = insert(root, 9);

    // Перевірка коректності вставки
    Assert::IsTrue(search(root, 5) != nullptr);
    Assert::IsTrue(search(root, 3) != nullptr);
    Assert::IsTrue(search(root, 8) != nullptr);
    Assert::IsTrue(search(root, 2) != nullptr);
    Assert::IsTrue(search(root, 4) != nullptr);
    Assert::IsTrue(search(root, 7) != nullptr);
    Assert::IsTrue(search(root, 9) != nullptr);

    // Видалення елементів
    root = deleteNode(root, 3);
    root = deleteNode(root, 8);

    // Перевірка видалення елементів
    Assert::IsNull(search(root, 3));
    Assert::IsNull(search(root, 8));
}
```

```
TEST_METHOD(TestMethod4)
{
    // Створення дерева
    Node* root = nullptr;

    // Вставка елементів
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 10);
    root = insert(root, 25);

    // Перевірка балансування
    Assert::AreEqual(height(root), 3);
    Assert::AreEqual(getBalance(root), 1);
}
```



Висновок: У цій лабораторній роботі ми дослідили структуру та використання AVL-дерев, які є ефективними для швидкого пошуку, вставки і видалення даних у великих обсягах. Написання власної реалізації дозволило глибше розібратися у принципах самобалансування дерева та оптимізації роботи з даними.