**AVL Trees: Introduction and Detection of Imbalance**

**Introduction to AVL Trees**

An **AVL Tree** (Adelson-Velsky and Landis Tree) is a **self-balancing Binary Search Tree (BST)** where the difference in height between the left and right subtrees of any node (called the **balance factor**) is at most **1**.

The **AVL Tree** enforces a key principle: the height difference between the left and right subtrees of any node must remain minimal. If the height imbalance exceeds a certain threshold, the AVL Tree automatically performs **rotations** to restore balance. This ensures that the tree maintains its **binary search efficiency**

AVL trees ensure that search, insertion, and deletion operations maintain **O(log n) time complexity**, preventing the worst-case scenario of a skewed BST (**O(n) complexity**).

**Time complexity**

|  |  |
| --- | --- |
| **Operation** | **Best Case** |
| Search | O(log n) |
| Insertion | O(log n) |
| Deletion | O(log n) |

**Detection of Imbalance**

The **balance factor** of a node is calculated as:

***balance factor = right subtree height - left subtree height***

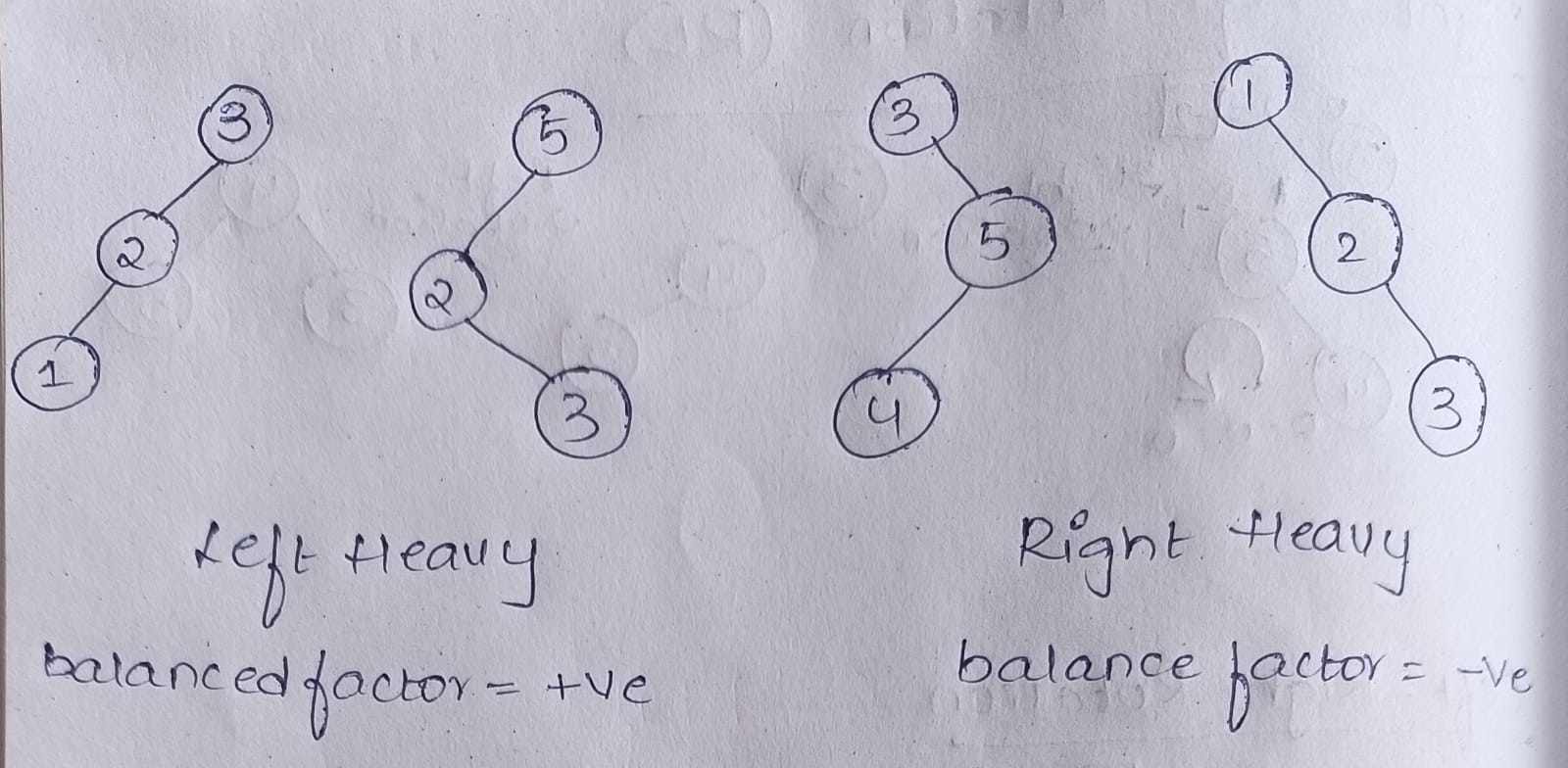
A node’s balance factor will fall into one of two categories: ***balanced or imbalanced.***

A node is ***balanced*** if its balance factor is equal to -1, -0, or 1. When a node is balanced, there is no action needed.

A node is ***imbalanced*** if its balance factor is less than or equal to -2, or greater than or equal to 2. If a node’s balance factor indicates it to be unbalanced we will need to perform one or four rotations.

**Types of Imbalances in AVL Trees**

1. **Left-Left (LL) Imbalance** → Requires **Right Rotation**
2. **Right-Right (RR) Imbalance** → Requires **Left Rotation**
3. **Left-Right (LR) Imbalance** → Requires **Left Rotation** followed by **Right Rotation**
4. **Right-Left (RL) Imbalance** → Requires **Right Rotation** followed by **Left Rotation**



**Right Rotation in AVL Trees**

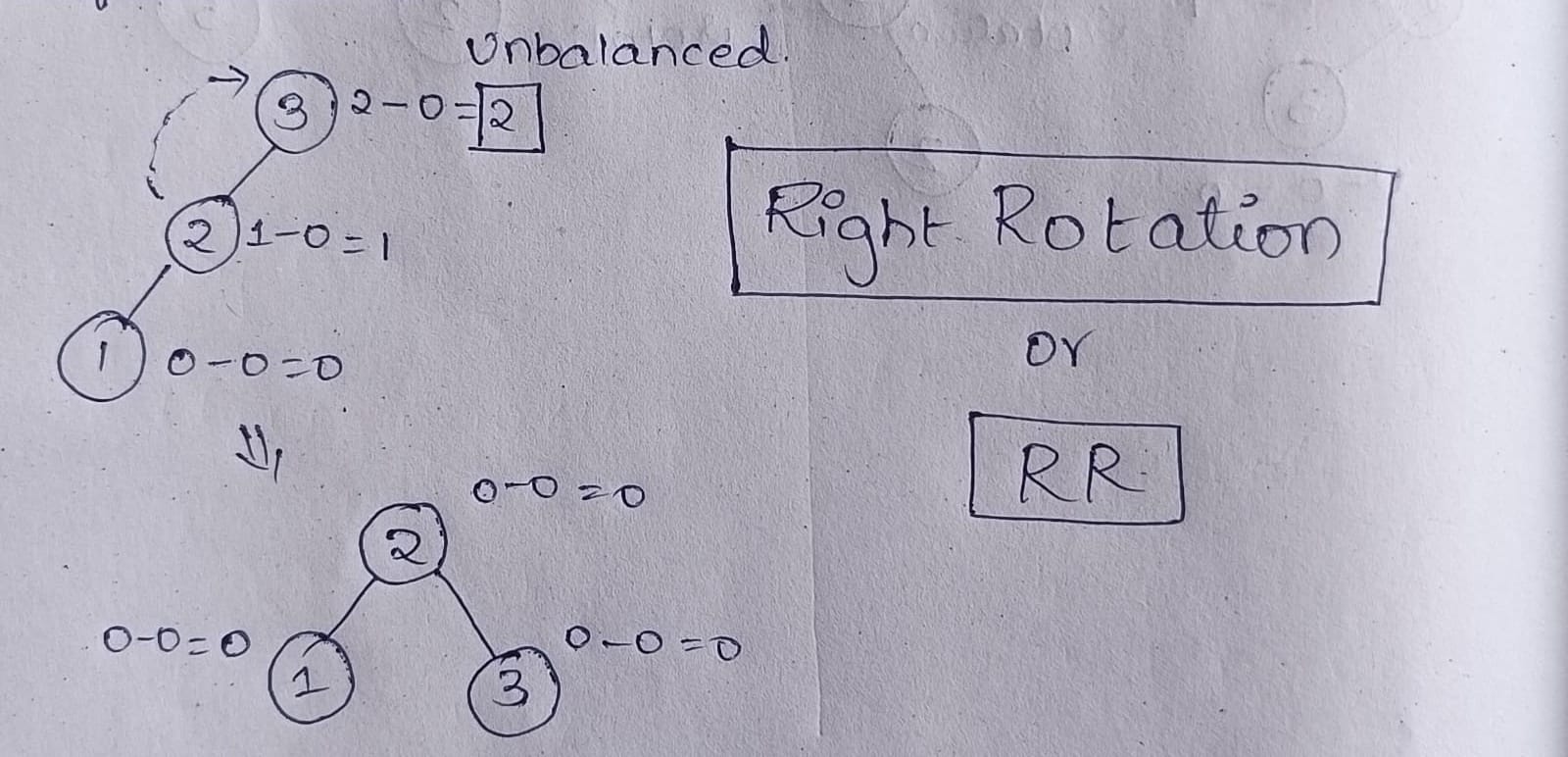
A **Right Rotation** (also called an RR rotation) is performed when a node is inserted into the **left subtree of its left child**, causing an imbalance. This helps in restoring the AVL tree's balance while maintaining the binary search tree properties.

**Steps of Right Rotation**

1. **Identify the Imbalance:**
   * The imbalance occurs at node **5** (balance factor **2**), meaning the left subtree is heavier.
   * Node **3** is the left child of **5**, and node **2** is the left child of **3**.
2. **Perform Right Rotation:**
   * Node **3** becomes the new root.
   * Node **5** moves to the right of **3**.
   * Node **2** remains as the left child of **3**.

**Final Balanced Tree:**

After the right rotation, the tree becomes balanced, ensuring that search, insertion, and deletion operations maintain an efficient time complexity of **O(log n)**.



**Left Rotation in AVL Trees**

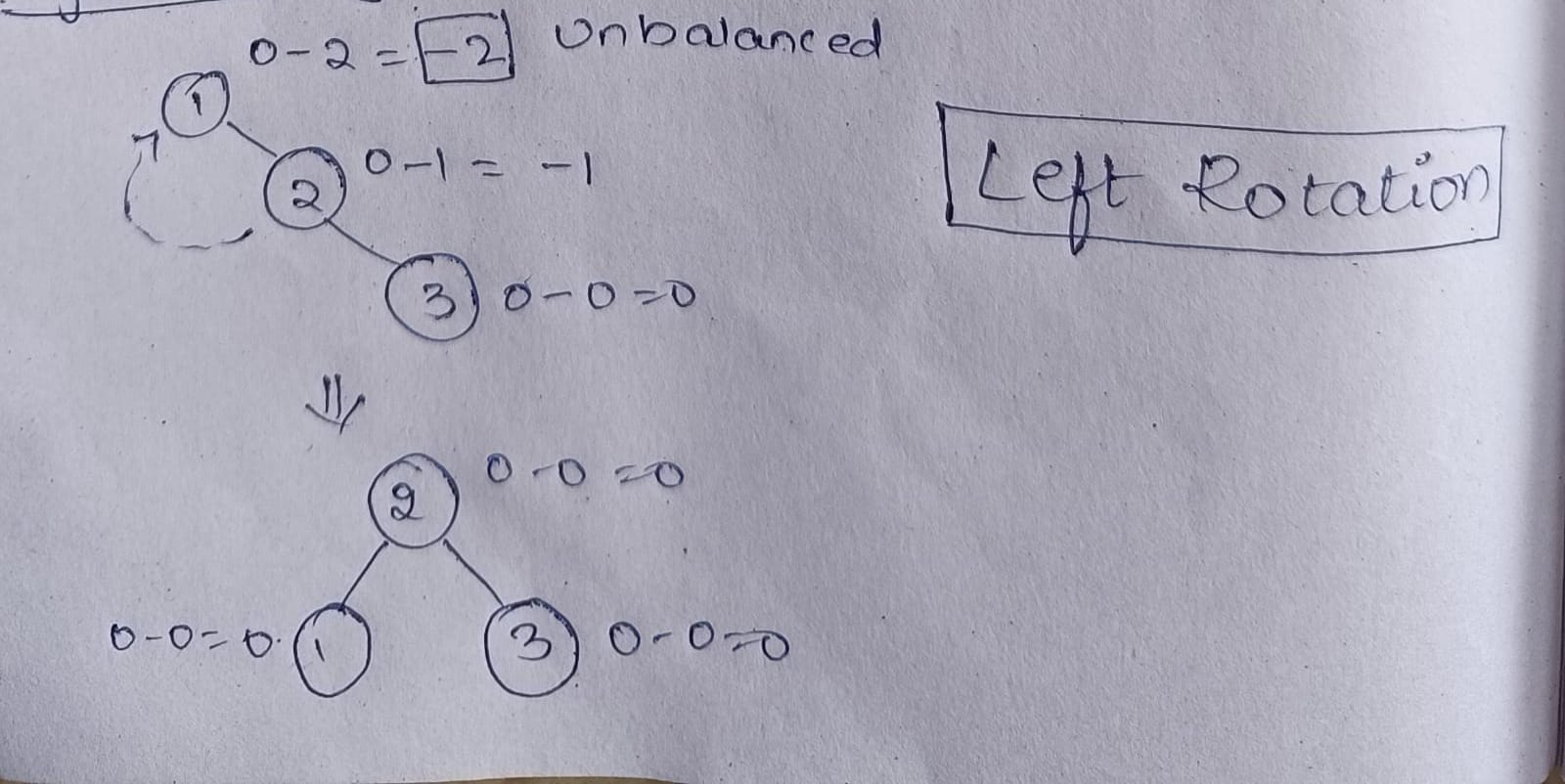
A **Left Rotation** (also called an LL rotation) is performed when a node is inserted into the **right subtree of its right child**, causing an imbalance. This rotation helps restore the AVL tree's balance while maintaining the binary search tree (BST) properties.

**Steps of Left Rotation**

1. **Identify the Imbalance:**
   * The imbalance occurs at node **1** (balance factor **-2**), meaning the right subtree is heavier.
   * Node **2** is the right child of **1**, and node **3** is the right child of **2**.
2. **Perform Left Rotation:**
   * Node **2** becomes the new root.
   * Node **1** moves to the left of **2**.
   * Node **3** remains as the right child of **2**.

**Final Balanced Tree:**

After the left rotation, the tree becomes balanced, ensuring efficient search, insertion, and deletion operations with a time complexity of **O(log n)**



**Right-Left (RL) Rotation in AVL Trees**

A **Right-Left Rotation** is a **combination** of two rotations:

1. **Right Rotation** on the right subtree.
2. **Left Rotation** on the root.

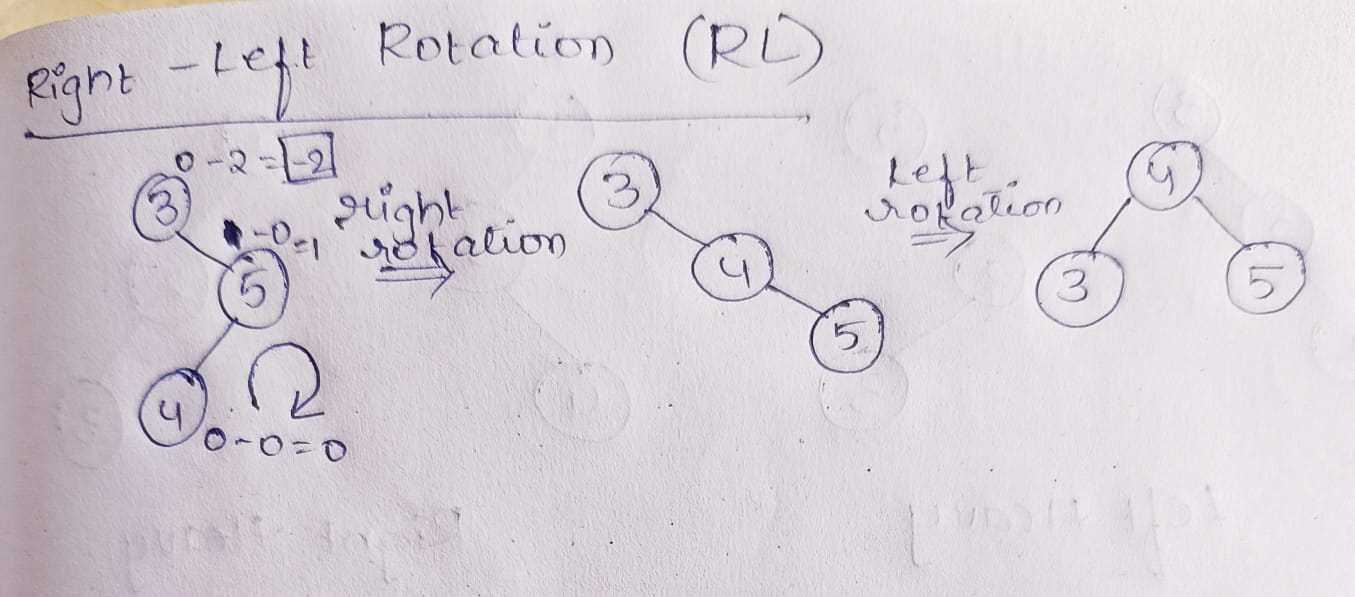
This rotation is required when a node is inserted into the **left subtree of the right child**, making the tree unbalanced.

**Steps of Right-Left (RL) Rotation:**

1. **Identify the Imbalance:**
   * The imbalance occurs at node **3** (balance factor **-2**).
   * Node **5** is its right child, but **4** is inserted into the left of **5**, causing the RL imbalance.
2. **Perform Right Rotation on the Right Subtree:**
   * Rotate **5** to the right, making **4** its new parent.
   * Now, **4** becomes the new right child of **3**.
3. **Perform Left Rotation on the Root:**
   * Rotate **3** to the left.
   * Node **4** becomes the new root.
   * Node **3** becomes the left child of **4**.
   * Node **5** remains the right child of **4**.

### ****Final Balanced Tree:****

After applying the RL rotation, the AVL tree is balanced, ensuring an optimal search, insertion, and deletion time complexity of **O(log n)**.



**Left-Right (LR) Rotation in AVL Trees**

A **Left-Right Rotation** is a **combination** of two rotations:

1. **Left Rotation** on the left subtree.
2. **Right Rotation** on the root.

This rotation is required when a node is inserted into the **right subtree of the left child**, making the tree unbalanced.

**Steps of Left-Right (LR) Rotation:**

1. **Identify the Imbalance:**
   * The imbalance occurs at node **5** (balance factor **2**).
   * Node **2** is its left child, but **3** is inserted into the right of **2**, causing the LR imbalance.
2. **Perform Left Rotation on the Left Subtree:**
   * Rotate **2** to the left, making **3** its new parent.
   * Now, **3** becomes the new left child of **5**.
3. **Perform Right Rotation on the Root:**
   * Rotate **5** to the right.
   * Node **3** becomes the new root.
   * Node **2** remains the left child of **3**.
   * Node **5** becomes the right child of **3**.

### ****Final Balanced Tree:****

After applying the **LR rotation**, the AVL tree becomes balanced, ensuring optimal operations with **O(log n)** time complexity.

