What is an AVL Tree?

An **AVL Tree** is a **self-balancing binary search tree (BST)** where:

1. The difference in height between the **left** and **right** subtrees of any node is at most **1**.
2. If the height balance is ever violated after inserting or deleting a node, the tree performs **rotations** to restore balance.

This ensures that operations like **insertion, deletion, and search** all run in **O(log n)** time complexity.

AVL Tree Rules

1. Binary Search Tree (BST) Property

- Each node has a value.
- The left subtree of a node contains **only nodes with values smaller** than the node's value.
- The right subtree contains **only nodes with values greater** than the node's value.
- This rule must always hold after inserting or deleting a node.

2. Balance Factor

- The **balance factor (BF)** of a node is calculated as: BF=height of left subtree−height of right subtreeBF = {height of left subtree} - {height of right subtree}
- The balance factor can be **-1, 0, or 1** for a tree to remain balanced.
- If the balance factor is **less than -1 or greater than 1**, the tree is unbalanced and must be **rotated**.

3. Rotations (Rebalancing)

When inserting or deleting nodes causes an imbalance, we perform **rotations** to restore balance. There are **four types**:

Single Rotations

1. **Right Rotation (LL Rotation)**: When a node is inserted into the **left subtree of the left child** (LL Case).
    - Fix: Rotate **right** on the unbalanced node.
2. **Left Rotation (RR Rotation)**: When a node is inserted into the **right subtree of the right child** (RR Case).
    - Fix: Rotate **left** on the unbalanced node.

Double Rotations

3. **Left-Right Rotation (LR Rotation)**: When a node is inserted into the **right subtree of the left child** (LR Case).
   - Fix: First, perform a **left rotation** on the left child, then a **right rotation** on the unbalanced node.
4. **Right-Left Rotation (RL Rotation)**: When a node is inserted into the **left subtree of the right child** (RL Case).
   - Fix: First, perform a **right rotation** on the right child, then a **left rotation** on the unbalanced node.

Insertion Process in an AVL Tree

1. **Insert** the node following **BST rules**.
2. **Update heights** of affected nodes.
3. **Check balance factor** for each node from the inserted node up to the root.
4. If the balance factor is **outside the range [-1,1]**, perform the necessary **rotation(s)**.

Deletion Process in an AVL Tree

1. **Delete** the node following **BST rules**.
2. **Update heights** of affected nodes.
3. **Check balance factor** from the deleted node up to the root.
4. If the balance factor is **outside the range [-1,1]**, perform the necessary **rotation(s)**.

Example Walkthrough

Example 1: Inserting 10, 20, 30 (RR Rotation)

1. Insert **10** → Tree is balanced. ✅
2. Insert **20** → Still balanced. ✅

Insert **30** → **Unbalanced at 10** (BF = -2) → **Perform Left Rotation (RR Case)**.
  10 \ 20 \ 30   →   10 / 20 \ 30

3.

Example 2: Inserting 30, 20, 10 (LL Rotation)

1. Insert **30** → Tree is balanced. ✅
2. Insert **20** → Still balanced. ✅

Insert **10** → **Unbalanced at 30** (BF = +2) → **Perform Right Rotation (LL Case)**.
10 / 20 / 30 → 10 / 20 \ 30

Final Notes
- **AVL trees ensure O(log n) operations** by maintaining balance.
- **Rotations help restore balance** when needed.
- **Insertion and deletion both require checking the balance factor** and fixing violations.