AVL Tree

Introduction

An **AVL Tree** is a type of **self-balancing Binary Search Tree (BST)**.

- Named after **Adelson-Velsky and Landis**, who invented it.
- It maintains **balance** by ensuring the **height difference** between the left and right subtree of any node is at most **1.**

Balance Factor (BF):

BF=height(leftSubtree)−height(rightSubtree)BF = height(leftSubtree) - height(rightSubtree)BF=height(leftSubtree)−height(rightSubtree)

- BF = -1, 0, +1 → Valid (balanced).
- If BF < -1 or BF > +1 → Tree is **unbalanced**, requires **rotation**.

Why AVL Tree

Problem with Normal BST

- A Binary Search Tree works well **only if it is balanced**.
- But in the worst case, a BST can become **skewed** (like a linked list).

Example (insert in order): `10, 20, 30, 40, 50`

```
10
  \
   20
     \
      30
        \
         40
           \
            50
```

- This is basically a **linked list**, not a tree.
- Height = `n`
- Search, Insert, Delete = **O(n)** (worst case).

Solution → Self-Balancing BST

- AVL Tree was invented to solve this problem.
- AVL Tree **balances itself** after every insertion or deletion using **rotations**.
- Ensures height ≈ `log(n)` always.

Balanced AVL Tree for same input `10, 20, 30, 40, 50`:

```
      30
     /  \
   20    40
  /        \
10          50
```

- Height = `log n`
- Search, Insert, Delete = **O(log n)** always.

Advantages of AVL Tree

Faster searching compared to unbalanced BST.
Guarantees worst-case performance = **O(log n)**.
Good choice when applications involve **lots of searches**.

When not to use AVL?

- AVL requires **more rotations** during insertion/deletion (overhead).
- If your application does **lots of insertions/deletions** but fewer searches → **Red-Black Tree** may be better.
- If your application does **more searching than updates** → **AVL is ideal**.

Properties of AVL Tree

1. **Height-balanced BST.**
2. **Balance Factor** of every node is `-1, 0, or +1`.

3. **Time Complexity**:
   - Search → `O(log n)`
   - Insertion → `O(log n)` (may need rotation)
   - Deletion → `O(log n)` (may need rotation)
4. Worst-case height of AVL tree = **O(log n)**.

Rotations in AVL Tree

Rotations are used to **rebalance the tree** when BF goes outside `[-1, +1]`.

🔹 Types of Rotations

1. **Right Rotation (LL Rotation)** – Occurs when a node is inserted into the **left subtree of left child**.
2. **Left Rotation (RR Rotation)** – Occurs when a node is inserted into the **right subtree of right child**.
3. **Left-Right Rotation (LR Rotation)** – Occurs when a node is inserted into the **right subtree of left child**.
4. **Right-Left Rotation (RL Rotation)** – Occurs when a node is inserted into the **left subtree of right child**.

Diagram (conceptual):

- **LL (Right Rotation)**

```
    z                  y
    /                 / \
  y       →          x   z
 /
```

```
x
```

- **RR (Left Rotation)**

```
z                 y
 \               / \
  y     →       z   x
   \
    x
```

- **LR (Left-Right Rotation)**

```
  z                z                  x
  /               /                  / \
y        →       x        →         y   z
  \             /
   x           y
```

- **RL (Right-Left Rotation)**

```
z                z                  x
 \                \                / \
  y      →         x      →       z   y
 /                  \
x                    y
```

Insertion in AVL Tree

1. Perform normal **BST insertion**.
2. Update height of ancestor nodes.
3. Compute balance factor.
4. If unbalanced → apply **appropriate rotation** (LL, RR, LR, RL).

Deletion in AVL Tree

1. Perform normal **BST deletion**.
2. Update height of ancestor nodes.
3. Compute balance factor.
4. If unbalanced → apply rotations.

Applications of AVL Trees

- Databases and indexing (where fast search is critical).
- Memory management (OS).
- Used when data is **read-heavy** and **balanced search time** is important.