

AVL Tree Insertion, Imbalance, and Rebalancing Notes

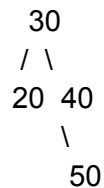
What is an AVL Tree?

An AVL tree is a self-balancing binary search tree. After every insertion (or deletion), it ensures that the tree remains approximately balanced, so that search, insert, and delete operations remain $O(\log n)$ in time.

What is the height of a node?

- The height of a node is the number of edges on the longest path from that node to a leaf.
- A leaf node has height 0.
- An empty subtree has height -1 (used for easy balance factor math).

Example:



- $\text{height}(20) = 0$
- $\text{height}(50) = 0$
- $\text{height}(40) = 1$
- $\text{height}(30) = \max(1, 0) + 1 = 2$

What is a balance factor?

- Balance Factor = $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- A node is **balanced** if its balance factor is -1, 0, or 1
- If the balance factor becomes < -1 or > 1 , the node is **unbalanced**

Why do imbalances happen?

When inserting a node into an AVL tree:

- The tree grows taller at some point

- This increase in height may cause one side of a node's subtree to be taller than the other by more than 1
- This triggers an imbalance and requires a rotation to fix it

Types of Imbalance & Fixes:

1. **LL (Left-Left) Case**

- Insertion occurs in the left subtree of the left child

Insert in order: 30 → 25 → 20

```

    30
   /
  25
 /
20

```

- Node 30 has balance factor = 2 (left-heavy)
- Fix: **Right rotation** at 30

Result:

```

    25
   / \
  20  30

```

2. **RR (Right-Right) Case**

- Insertion occurs in the right subtree of the right child

Insert in order: 10 → 15 → 20

```

    10
     \
     15
      \
      20

```

- Node 10 has balance factor = -2 (right-heavy)
- Fix: **Left rotation** at 10

Result:

```
  15
 /  \
10   20
```

3. **LR (Left-Right) Case**

- Insertion occurs in the right subtree of the left child

Insert in order: 30 → 20 → 25

```
  30
 /
20
 \
 25
```

- Node 30 has balance factor = 2 (left-heavy)
- Fix: **Left rotation** at 20 → **Right rotation** at 30 (double rotation)

Result:

```
  25
 /  \
20   30
```

4. **RL (Right-Left) Case**

- Insertion occurs in the left subtree of the right child

Insert in order: 10 → 20 → 15

```
  10
 \
  20
 /
15
```

- Node 10 has balance factor = -2 (right-heavy)
- Fix: **Right rotation** at 20 → **Left rotation** at 10 (double rotation)

Result:

```
    15
   /  \
  10   20
```

Quick Recap of Rotations:

- LL → Single Right Rotation
- RR → Single Left Rotation
- LR → Left at child, Right at parent
- RL → Right at child, Left at parent

How to detect imbalance?

After each insertion:

1. Recurse back up to the root
2. For each ancestor node, calculate:
 - Balance Factor = $\text{height}(\text{left}) - \text{height}(\text{right})$
3. If balance factor is out of range (greater than 1 or less than -1), perform the required rotation.

Why does AVL balancing matter?

Without rebalancing:

- The tree can become skewed (like a linked list)
- Performance drops from $O(\log n)$ to $O(n)$

AVL trees guarantee efficient lookups, inserts, and deletes, especially in real-time or performance-sensitive systems.