AVL Tree Basics

Can you teach me AVL tree in easy manner step by step.

Yes! Let’s go step by step to understand **AVL Trees** in an easy way.

---

1. What is an AVL Tree?

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

---

2. Why AVL Tree?

A normal **BST can become skewed**, meaning that some operations like searching can take **O(n) time** instead of **O(log n)**.
To keep the tree balanced and operations efficient, we use **AVL Trees**.

---

3. Balance Factor

For each node in an AVL Tree:
Balance Factor = Height of Left Subtree - Height of Right Subtree

- If **Balance Factor is -1, 0, or 1**, the tree is balanced.
- If **Balance Factor is less than -1 or greater than 1**, the tree is unbalanced and requires rotation.

---

4. Rotations in AVL Tree

If an insertion or deletion makes the tree unbalanced, we fix it using **rotations.** There are **four types of rotations**:

Case 1: Left-Left (LL) Rotation

- Occurs when a node is inserted **in the left subtree of the left child**.
- Fixed by performing a **Right Rotation (Single Rotation)**.

Case 2: Right-Right (RR) Rotation

- Occurs when a node is inserted **in the right subtree of the right child**.
- Fixed by performing a **Left Rotation (Single Rotation)**.

Case 3: Left-Right (LR) Rotation

- Occurs when a node is inserted **in the right subtree of the left child**.
- Fixed by **Left Rotation on the left child**, then **Right Rotation on the node**.

Case 4: Right-Left (RL) Rotation

- Occurs when a node is inserted **in the left subtree of the right child**.
- Fixed by **Right Rotation on the right child**, then **Left Rotation on the node**.

---

5. Insertion in AVL Tree

1. Insert the node as in a normal BST.
2. Update the height of affected nodes.
3. Calculate the balance factor.
4. If balance factor is **not in range [-1, 0, 1]**, perform the required rotation.

---

6. Example of AVL Tree Insertion

Let's insert **10, 20, 30, 40, 50, 25** step by step:

1. Insert `10`

```
10
```

2. Insert `20`

```markdown
10
  \
   20
```

3. Insert `30` (Right-Right Case → Left Rotation at `10`)

```markdown
  20
 /  \
10   30
```

4. Insert `40`

```markdown
  20
 /  \
10   30
       \
        40
```

5. Insert `50` (Right-Right Case → Left Rotation at `30`)

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

6. Insert `25` (Right-Left Case → Right Rotation at `40`, then Left Rotation at `20`)

```markdown
```

```
    30
   /  \
  20   40
 /  \     \
10   25    50
```

Now the tree is balanced.

---

7. Deletion in AVL Tree

1. Delete the node as in BST.
2. Update the height of affected nodes.
3. Calculate balance factor.
4. If unbalanced, perform necessary rotations.

---

8. Time Complexity

- **Search: O(log n)**
- **Insertion: O(log n)**
- **Deletion: O(log n)**

Summary

✔ AVL Tree is a self-balancing BST.
✔ Balance Factor helps keep the tree balanced.
✔ Four types of rotations are used to maintain balance.
✔ Operations take **O(log n)** time.

Would you like to see Java code for insertion and rotations?

Can you reach me the 4 rotation in AVL tree in simple manner

Yes! Let’s go step by step to understand the **four rotations in an AVL Tree** in a simple way.

Why Do We Need Rotations?

After inserting or deleting a node, the AVL tree might become **unbalanced**. If the **Balance Factor** (Height of Left Subtree - Height of Right Subtree) is **less than -1 or greater than 1**, we need to **perform rotations** to balance the tree.

---

1. Left-Left (LL) Rotation → Right Rotation

- **When?**
  A node is inserted **in the left subtree of the left child**.
- **Fix?**
  Perform a **Right Rotation**.

Example:

Before LL Rotation (Unbalanced)

markdown

```

        30
       /
      20
     /
    10
```

- Here, node `10` is inserted in the left subtree of `20`, making `30` unbalanced.
- **Balance Factor of `30` = 2** (since left subtree is too tall).

After Right Rotation at `30`

```markdown
  20
 /  \
10   30
```

- Now the tree is balanced.

---

2. Right-Right (RR) Rotation → Left Rotation

- **When?**
  A node is inserted **in the right subtree of the right child**.
- **Fix?**
  Perform a **Left Rotation**.

Example:

Before RR Rotation (Unbalanced)

```markdown
  10
    \
     20
       \
        30
```

- Here, node `30` is inserted in the right subtree of `20`, making `10` unbalanced.
- **Balance Factor of `10`** = **-2** (since right subtree is too tall).

After Left Rotation at `10`

```markdown
  20
 /  \
10   30
```

- Now the tree is balanced.

---

3. Left-Right (LR) Rotation → Left Rotation + Right Rotation

- **When?**

  A node is inserted **in the right subtree of the left child**.
- **Fix?**

  Perform **Left Rotation on left child**, then **Right Rotation**.

Example:

Before LR Rotation (Unbalanced)

```markdown
        30
       /
      10
        \
         20
```

- Here, node `20` is inserted in the right subtree of `10`, making `30` unbalanced.
- **Balance Factor of `30` = 2** (since left subtree is too tall).

Step 1: Left Rotation at `10`

```markdown
        30
       /
      20
     /
    10
```

Step 2: Right Rotation at `30`

```markdown
```

```
        20
       /  \
      10   30
```

- Now the tree is balanced.

---

4. Right-Left (RL) Rotation → Right Rotation + Left Rotation

- **When?**

  A node is inserted **in the left subtree of the right child**.
- **Fix?**

  Perform **Right Rotation on right child**, then **Left Rotation**.

Example:

Before RL Rotation (Unbalanced)

```markdown
        10
          \
           30
          /
         20
```

- Here, node `20` is inserted in the left subtree of `30`, making `10` unbalanced.
- **Balance Factor of `10`** = **-2** (since right subtree is too tall).

Step 1: Right Rotation at `30`

```markdown
        10
          \
           20
```

```
             \
              30
```

Step 2: Left Rotation at `10`

```markdown
        20
       /  \
      10   30
```

- Now the tree is balanced.

Summary of Rotations

Case	When does it happen?	Fix
LL	Insert in **left subtree of left child**	**Right Rotation**
RR	Insert in **right subtree of right child**	**Left Rotation**
LR	Insert in **right subtree of left child**	**Left Rotation + Right Rotation**
RL	Insert in **left subtree of right child**	**Right Rotation + Left Rotation**

Would you like me to show Java code for these rotations?

Yes please

Here's the Java implementation for all four **rotations in an AVL Tree** with a simple structure.

---

1. AVL Tree Node Structure

Each node will have:

- A **value** (data).
- Left and right **child pointers**.
- A **height** for balance calculation.

```java
class Node {
    int value, height;
    Node left, right;

    Node(int value) {
        this.value = value;
        this.height = 1; // Initial height is 1 (leaf node).
    }
}
```

---

2. Utility Functions

These functions will help with height calculation and balance checking.

```java
class AVLTree {
    Node root;

    // Get height of a node
    int height(Node node) {
        return (node == null) ? 0 : node.height;
    }

    // Get balance factor
    int getBalanceFactor(Node node) {
        return (node == null) ? 0 : height(node.left) - height(node.right);
    }

    // Update height after insertion/deletion
    void updateHeight(Node node) {
        node.height = 1 + Math.max(height(node.left), height(node.right));
    }
}
```

---

3. Rotations

Now, let's implement the **four types of rotations**.

(1) Right Rotation (for LL Case)

java

```java
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    updateHeight(y);
    updateHeight(x);

    // Return new root
    return x;
}
```

(2) Left Rotation (for RR Case)

java

```java
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    updateHeight(x);
    updateHeight(y);

    // Return new root
    return y;
}
```

(3) Left-Right Rotation (for LR Case)

```java
Node leftRightRotate(Node node) {
    node.left = leftRotate(node.left);
    return rightRotate(node);
}
```

(4) Right-Left Rotation (for RL Case)

```java
Node rightLeftRotate(Node node) {
    node.right = rightRotate(node.right);
    return leftRotate(node);
}
```

---

4. Insertion with Auto Balancing

After inserting a node like in a normal BST, we check the **Balance Factor** and apply the appropriate rotation.

```java
Node insert(Node node, int value) {
    // Step 1: Perform normal BST insertion
    if (node == null) return new Node(value);

    if (value < node.value)
        node.left = insert(node.left, value);
    else if (value > node.value)
        node.right = insert(node.right, value);
    else
        return node; // No duplicates
    // Step 2: Update height
    updateHeight(node);

    // Step 3: Get balance factor
    int balance = getBalanceFactor(node);

    // Step 4: Perform appropriate rotation

    // Left-Left Case (LL)
    if (balance > 1 && value < node.left.value)
        return rightRotate(node);

    // Right-Right Case (RR)
    if (balance < -1 && value > node.right.value)
        return leftRotate(node);

    // Left-Right Case (LR)
    if (balance > 1 && value > node.left.value)
        return leftRightRotate(node);

    // Right-Left Case (RL)
    if (balance < -1 && value < node.right.value)
        return rightLeftRotate(node);

    return node;
}
```

---

5. Print In-Order Traversal (To Check Output)

java

```
void inOrder(Node node) {
    if (node != null) {
        inOrder(node.left);
        System.out.print(node.value + " ");
        inOrder(node.right);
    }
}
```

---

6. Running the AVL Tree

java

```
public class Main {
    public static void main(String[] args) {
        AVLTree tree = new AVLTree();

        tree.root = tree.insert(tree.root, 30);
        tree.root = tree.insert(tree.root, 20);
        tree.root = tree.insert(tree.root, 10); // LL Rotation will happen
        tree.root = tree.insert(tree.root, 40);
        tree.root = tree.insert(tree.root, 50); // RR Rotation will happen
        tree.root = tree.insert(tree.root, 25); // LR Rotation will happen
        tree.root = tree.insert(tree.root, 35); // RL Rotation will happen

        System.out.print("In-Order Traversal: ");
        tree.inOrder(tree.root);
    }
}
```

Expected Output (Balanced AVL Tree)

mathematica

```
In-Order Traversal: 10 20 25 30 35 40 50
```

Key Takeaways

✔ **LL → Right Rotation** (When inserted in left-left).

✔ **RR → Left Rotation** (When inserted in right-right).

✔ **LR → Left Rotation + Right Rotation** (When inserted in left-right).

✔ **RL → Right Rotation + Left Rotation** (When inserted in right-left).

Would you like a deletion function as well?

Is this conversation helpful so far?