AVL Tree

We've just explored the fascinating world of Binary Search Trees (BSTs). You know how they organize data efficiently for searching, insertion, and deletion. But there's a catch, isn't there? If you insert elements into a BST in a sorted (or nearly sorted) order, it can become **skewed** – essentially turning into a linked list. When that happens, all those wonderful O(logN) average-case time complexities for operations degrade to O(N) in the worst case. That's no good for large datasets!

This is where **AVL Trees** come to the rescue!

🌳 AVL Trees: The Self-Balancing Superheroes!

An **AVL Tree** is a special type of **Binary Search Tree** that is always **self-balancing**. What does "self-balancing" mean? It means that after every insertion or deletion operation, the AVL tree automatically performs a set of operations (called **rotations**) to ensure that it remains "balanced."

The Core Idea: Balance Factor

The "balance" in an AVL tree is maintained by a property called the **Balance Factor**.

- **Balance Factor of a Node:** It's the difference between the height of its left subtree and the height of its right subtree.
  Balance Factor (BF) = Height of Left Subtree - Height of Right Subtree
- **AVL Tree Property:** For an AVL tree to be balanced, the balance factor of *every single node* must be one of these three values: **-1, 0, or 1**.
  - BF = 0: Left and right subtrees have the same height.
  - BF = 1: Left subtree is one level taller than the right subtree.
  - BF = -1: Right subtree is one level taller than the left subtree.

If, after an insertion or deletion, any node's balance factor becomes 2 or -2, the tree is considered **unbalanced** at that node, and a **rotation** must be performed to restore the AVL property.

🔄 The Magic of Rotations: Restoring Balance

Rotations are the core mechanism by which an AVL tree self-balances. They are local transformations that rearrange nodes to reduce the height difference without violating the BST property. There are four fundamental types of rotations:

Let's denote the node where the imbalance occurs as Z. The child of Z that's on the "heavy" side is Y, and the grandchild that caused the imbalance is X.

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

- **Scenario:** A node Z becomes unbalanced (BF = 2) because a new node X was inserted into the left subtree of its left child (Y). The tree is "left-heavy-left."
- **Action:** Perform a **Right Rotation** around Z. Y becomes the new root of the subtree, Z becomes Y's right child, and Y's original right child (if any) becomes Z's left child.

  ```
     Z (BF = 2)           Y
    /                    / \
   Y                    X   Z
  / \                      / \
  X   T3 (T1, T2, T3 are subtrees)  T4   T3
  / \
  T1 T2

  (Rotate Right around Z)
  ```

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

- **Scenario:** A node Z becomes unbalanced (BF = -2) because a new node X was inserted into the right subtree of its right child (Y). The tree is "right-heavy-right."
- **Action:** Perform a **Left Rotation** around Z. Y becomes the new root of the subtree, Z becomes Y's left child, and Y's original left child (if any) becomes Z's right child.

  ```
    Z (BF = -2)            Y
     \                    / \
      Y                   Z   X
     / \                 / \ / \
    T1  X                T1 T2 T3 T4
       / \
      T2 T3

  (Rotate Left around Z)
  ```

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

- **Scenario:** A node Z becomes unbalanced (BF = 2) because a new node X was inserted into the right subtree of its left child (Y). The tree is "left-heavy-right."

- **Action:** This is a **two-step rotation**:
  1. Perform a **Left Rotation** around Y. This transforms the LR case into an LL case.
  2. Then, perform a **Right Rotation** around Z (as in the LL case).

```
   Z (BF = 2)          Z (BF = 2)          X
  /                   /                   / \
 Y                   X                     Y   Z
/ \                 / \                  / \ / \
T1  X                Y   T4                T1 T2 T3 T4
  / \              / \
 T2 T3              T1 T2
```

(1. Rotate Left around Y) (2. Rotate Right around Z)

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

- **Scenario:** A node Z becomes unbalanced (BF = -2) because a new node X was inserted into the left subtree of its right child (Y). The tree is "right-heavy-left."
- **Action:** This is a **two-step rotation**:
  1. Perform a **Right Rotation** around Y. This transforms the RL case into an RR case.
  2. Then, perform a **Left Rotation** around Z (as in the RR case).

```
 Z (BF = -2)          Z (BF = -2)          X
  \                    \                  / \
   Y                    X                  Z   Y
  / \                  / \               / \ / \
 X   T4                 T1  Y               T1 T2 T3 T4
/ \                       / \
T1 T2                       T2 T3
```

(1. Rotate Right around Y) (2. Rotate Left around Z)

🚀 Basic Operations in an AVL Tree (with Balancing)

The core operations (insert, delete, search) are similar to a regular BST, but with crucial additional steps to check and restore balance.

1. insert(data): Adding a New Node

1. **Standard BST Insertion:** Perform the insertion as you would in a normal BST (recursively traverse, go left for smaller, right for larger, insert at null spot).
2. **Update Height:** After inserting a node, and as the recursion unwinds, update the

height of each ancestor node.
3. **Calculate Balance Factor:** For each ancestor node, calculate its balance factor.
4. **Check for Imbalance:** If any node's balance factor becomes 2 or -2, an imbalance is detected.
5. **Perform Rotation(s):** Based on the type of imbalance (LL, RR, LR, RL), perform the necessary rotation(s) to restore balance. This rotation will return the new root of that subtree.

2. delete(data): Removing a Node

Deletion is more complex than insertion in AVL trees.

1. **Standard BST Deletion:** Perform the deletion as you would in a normal BST (handle leaf, one-child, two-child cases). For the two-child case, replace with inorder successor/predecessor and then delete the successor/predecessor.
2. **Update Height:** After deletion, and as the recursion unwinds, update the height of each ancestor node.
3. **Calculate Balance Factor:** For each ancestor node, calculate its balance factor.
4. **Check for Imbalance:** If any node's balance factor becomes 2 or -2, an imbalance is detected.
5. **Perform Rotation(s):** Based on the type of imbalance, perform the necessary rotation(s). Deletion can sometimes cause multiple ancestors to become unbalanced, so you might need to rebalance at each level as you return up the recursion stack.

3. search(data): Finding a Node

- **Concept:** This operation is identical to searching in a regular BST. No balancing is needed during a search.
- **Time Complexity:** O(logN) in both average and worst cases (because the tree is always balanced!). This is the main advantage of AVL trees over unbalanced BSTs.