AVL Tree Guide

An AVL Tree is a self-balancing binary search tree (BST) where the height difference (balance factor) between the left and right subtrees of any node is at most 1.

Each node in an AVL tree has:

- key: The value stored in the node.
- left: A reference to the left child.
- right: A reference to the right child.
- height: The height of the node.

The balance factor of a node is:
Balance Factor = height(left subtree) - height(right subtree)

A tree is balanced if the balance factor of every node is between -1 and 1.

Insertion in an AVL Tree

Insertion follows standard BST insertion but includes rebalancing if needed.

Steps:

1. Insert the node like in a BST
2. Update the height of each affected node.
3. Compute the balance factor.
4. If the balance factor is outside the range −1,1-1,1−1,1, perform rotation(s) to rebalance the tree.

Types of Rotations:

- Right Rotation (Single Rotation - LL Case): Performed when inserting into the left subtree of the left child.

- Left Rotation (Single Rotation - RR Case): Performed when inserting into the right subtree of the right child.
- Left-Right Rotation (Double Rotation - LR Case): Performed when inserting into the right subtree of the left child.
    - First, left rotation on the left child.
    - Then, right rotation on the node.
- Right-Left Rotation (Double Rotation - RL Case): Performed when inserting into the left subtree of the right child.
    - First, right rotation on the right child.
    - Then, left rotation on the node.

Example AVL Tree Insertion Code:

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

def get_height(node):
    return node.height if node else 0

def get_balance(node):
    return get_height(node.left) - get_height(node.right) if node else 0

def rotate_right(y):
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = max(get_height(y.left), get_height(y.right)) + 1
    x.height = max(get_height(x.left), get_height(x.right)) + 1
    return x

def rotate_left(x):
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    x.height = max(get_height(x.left), get_height(x.right)) + 1
    y.height = max(get_height(y.left), get_height(y.right)) + 1
    return y

def insert(root, key):
    if root is None:
        return TreeNode(key)

    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

    root.height = 1 + max(get_height(root.left), get_height(root.right))

    balance = get_balance(root)

    # Rotation cases
    if balance > 1 and key < root.left.key:
        return rotate_right(root)
    if balance < -1 and key > root.right.key:
        return rotate_left(root)
    if balance > 1 and key > root.left.key:
        root.left = rotate_left(root.left)
        return rotate_right(root)
    if balance < -1 and key < root.right.key:
        root.right = rotate_right(root.right)
        return rotate_left(root)

    return root
```

Deletion in AVL Tree

Deletion follows BST deletion rules, but the tree is rebalanced after deletion.

Steps:

1. Perform standard BST deletion.
2. Update the height of affected nodes.
3. Compute the balance factor.
4. If the balance factor is outside the range −1,1-1,1−1,1, perform rotations to restore balance.

Example Deletion Code:

```
def delete(root, key):
    if root is None:
        return root

    if key < root.key:
        root.left = delete(root.left, key)
    elif key > root.key:
        root.right = delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        temp = find_min(root.right)
        root.key = temp.key
        root.right = delete(root.right, temp.key)

    if root is None:
        return root

    root.height = 1 + max(get_height(root.left), get_height(root.right))

    balance = get_balance(root)

    if balance > 1 and get_balance(root.left) >= 0:
        return rotate_right(root)
    if balance > 1 and get_balance(root.left) < 0:
        root.left = rotate_left(root.left)
        return rotate_right(root)
    if balance < -1 and get_balance(root.right) <= 0:
        return rotate_left(root)
    if balance < -1 and get_balance(root.right) > 0:
        root.right = rotate_right(root.right)
        return rotate_left(root)

    return root
```

AVL Tree Traversals

Since an AVL tree is a BST with extra balancing, traversal methods are the same as in a normal BST.

- Inorder Traversal (Left, Root, Right) → Returns elements in sorted order.
- Preorder Traversal (Root, Left, Right) → Used for tree reconstruction.
- Postorder Traversal (Left, Right, Root) → Used for deleting a tree.

Example traversal function:

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)
```

Building an AVL Tree from a List

You can create an AVL tree dynamically by inserting values from a list.

```
def build_avl_from_list(values):
    root = None
    for value in values:
        root = insert(root, value)
    return root
```

Example usage:

```
values = [10, 5, 15, 2, 7, 12, 18]
root = build_avl_from_list(values)
inorder(root)  # Output: 2 5 7 10 12 15 18
```

AVL Tree Rules for LLM Completion

To ensure the LLM can complete an AVL tree when given an incomplete structure, it should:

- Learn insertion balancing patterns from examples.
- Understand traversal outputs and expected orders.
- Recognize AVL properties, including balance factors and rotations.

Prompt Examples for the LLM

- "Given the AVL tree {10, 5, 15}, insert 7, 12, and 18."
  Expected Response: {10, 5, 15, 7, 12, 18} (with necessary rotations)
- "What is the inorder traversal of {7, 4, 12, 2, 6, 9, 15}?"
  Expected Response: 2, 4, 6, 7, 9, 12, 15
- "Delete node 6 from {7, 4, 12, 2, 6, 9, 15}."
  Expected Response: {7, 4, 12, 2, 9, 15} (rebalanced if necessary)