Assignment 6

Manan Patel

1 Design

1. Add a "height" value in each node to record the height of the local tree with the node as root and allow the parent node to access it when calculating the balance factor.
Ans:
To extend the BST into an AVL tree, I added a height attribute to each node. This height attribute records the height of the subtree rooted at each node. The purpose of tracking the height is to allow easy calculation of the balance factor for each node. The balance factor is essential in determining whether the tree is balanced and helps to trigger rebalancing operations when needed.

The height of a node is calculated as:

$$\text{height} = 1 + \max(\text{height of left child, height of right child})$$

Using this height value, the balance factor for a node is determined by:

$$\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

Pseudocode for Height Calculation and Balance Factor:

Algorithm 1 Height Calculation and Balance Factor

Height Calculation for each node:
node.height = $1 + \max(\text{height(node.left)}, \text{height(node.right)})$

Balance Factor Calculation:
balance = height(node.left) - height(node.right)
return balance

2.Write the "insertion" algorithm, which can call additional algorithms to do rebalancing when needed. When applicable, the algorithms in the lecture notes ("left-rotate", "right-rotate", etc.) can be directly called, as far as the input/output remains the same.

Ans:

The insertion algorithm in an AVL tree is similar to a regular BST insertion but includes additional steps to maintain balance. After each insertion, the height of the affected nodes is updated. The balance factor of each node on the path back to the root is calculated to check if it falls within the range [-1, 1]. If any node is found to be imbalanced, rotations are applied to restore balance.

There are four types of rebalancing rotations in an AVL tree:

- **Left-Left (LL) Case:** When a node's left child's left subtree causes imbalance, a *right rotation* is applied.
- **Right-Right (RR) Case:** When a node's right child's right subtree causes imbalance, a *left rotation* is applied.
- **Left-Right (LR) Case:** When a node's left child's right subtree causes imbalance, a *left rotation* is first applied to the left child, followed by a *right rotation* on the node.
- **Right-Left (RL) Case:** When a node's right child's left subtree causes imbalance, a *right rotation* is first applied to the right child, followed by a *left rotation* on the node.

These rotations rearrange the nodes to restore balance while preserving the in-order sequence of the AVL tree.

Pseudocode for Insertion and Rebalancing:

Algorithm 2 AVL Tree Insertion with Rebalancing

```
Insert(node, key):
if node is None then
  return new TreeNode(key)
end if
if key < node.key then
  node.left = Insert(node.left, key)
else if key > node.key then
  node.right = Insert(node.right, key)
else
  return node
end if
Update the height of the node:
node.height = 1 + max(height(node.left), height(node.right))
Calculate balance factor:
balance = height(node.left) - height(node.right)
Rebalancing Conditions:
if balance > 1 and key < node.left.key then
  {Left-Left Case}return RightRotate(node)
end if
if balance < -1 and key > node.right.key then
  {Right-Right Case}return LeftRotate(node)
end if
if balance > 1 and key > node.left.key then
  {Left-Right Case}node.left = LeftRotate(node.left) return RightRo-
  tate(node)
end if
if balance < -1 and key < node.right.key then
  {Right-Left Case}node.right = RightRotate(node.right) return LeftRo-
  tate(node)
end if
return node
```