Design and Analysis of AVL Tree

Introduction

An AVL (Adelson-Velsky and Landis) tree is a self-balancing Binary Search Tree (BST) where the heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is performed to restore this property. This ensures that the tree remains approximately balanced, guaranteeing logarithmic time complexity for search, insert, and delete operations. In this report, we extend the standard BST design to an AVL tree by incorporating additional mechanisms for maintaining balance. Specifically, we:

1. Add a "height" attribute to each node to keep track of the height of the subtree rooted at that node.
2. Implement insertion and deletion algorithms that perform necessary rotations to maintain the AVL balance property.

Implementation of the AVL modified tree can be found here, which includes a sample output text and the code that generated the text.

AVL Tree Modifications

- **Node Structure**:
  - `key`: The value stored in the node.
  - `left`: Reference to the left child.
  - `right`: Reference to the right child.
  - `height`: The height of the subtree rooted at this node.
- **Balance Factor**: For any node, the balance factor is defined as:

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

  The AVL property requires that the balance factor of every node be -1, 0, or +1.

Insertion and Deletion Algorithms

Insertion and deletion in an AVL tree involve similar steps to maintain the balance of the tree. Both operations may require rotations to restore the AVL balance property.

Rotations

There are two fundamental rotation operations used to rebalance an AVL tree:

1. **Left Rotation**:

Algorithm 1 Left Rotation

```
procedure LeftRotate(z)
    y ← z.right
    T2 ← y.left
    y.left ← z
    z.right ← T2
    z.height ← 1 + max(GetHeight(z.left), GetHeight(z.right))
    y.height ← 1 + max(GetHeight(y.left), GetHeight(y.right))
    return y
end procedure
```

2. **Right Rotation**:

Algorithm 2 Right Rotation

```
procedure RightRotate(z)
    y ← z.left
    T3 ← y.right
    y.right ← z
    z.left ← T3
    z.height ← 1 + max(GetHeight(z.left), GetHeight(z.right))
    y.height ← 1 + max(GetHeight(y.left), GetHeight(y.right))
    return y
end procedure
```

Insertion Procedure

Algorithm 3 AVL Insertion

```
procedure INSERT(node, key)
    if node = None then
        return new AVLNode(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                                        ▷ Ignore duplicates
    end if
    node.height ← 1 + max(GETHEIGHT(node.left), GETHEIGHT(node.right))
    balance ← GETBALANCE(node)
    if balance > 1 and key < node.left.key then
        return RIGHTROTATE(node)
    end if
    if balance < −1 and key > node.right.key then
        return LEFTROTATE(node)
    end if
    if balance > 1 and key > node.left.key then
        node.left ← LEFTROTATE(node.left)
        return RIGHTROTATE(node)
    end if
    if balance < −1 and key < node.right.key then
        node.right ← RIGHTROTATE(node.right)
        return LEFTROTATE(node)
    end if
    return node
end procedure
```

Deletion Procedure

Deletion is not the only operation that can unbalance an AVL tree. Deletion may also disrupt the balance, requiring appropriate rotations to restore the AVL property. Below is the detailed deletion algorithm aligned with the provided Python implementation.

Algorithm 4 AVL Deletion

```
procedure DELETE(node, key)
    if node = None then
        return node
    end if
    if key < node.key then
        node.left ← DELETE(node.left, key)
    else if key > node.key then
        node.right ← DELETE(node.right, key)
    else                                          ▷ Node with one child or no child
        if node.left = None then
            return node.right
        else if node.right = None then
            return node.left
        end if        ▷ Node with two children: Get the inorder successor (smallest in the right subtree)
        temp ← GETMINVALUENODE(node.right)
        node.key ← temp.key
        node.right ← DELETE(node.right, temp.key)
    end if                                        ▷ If the tree had only one node then return
    if node = None then
        return node
    end if
    node.height ← 1 + max(GETHEIGHT(node.left), GETHEIGHT(node.right))
    balance ← GETBALANCE(node)                    ▷ Balance the node if necessary
    if balance > 1 and GETBALANCE(node.left) ≥ 0 then
        return RIGHTROTATE(node)
    end if
    if balance > 1 and GETBALANCE(node.left) ¡ 0 then     node.left ← LEFTROTATE(node.left)

    return RIGHTROTATE(node)
end if
if balance < −1 and GETBALANCE(node.right) ≤ 0 then
    return LEFTROTATE(node)
end if
if balance < −1 and GETBALANCE(node.right) ¿ 0 then     node.right ← RIGHTROTATE(node.right)

return LEFTROTATE(node)
end if
return node
end procedure
```

Helper Procedure: GetMinValueNode

When deleting a node with two children, we need to find the inorder successor (the smallest node in the right subtree). This helper procedure accomplishes that.

Algorithm 5 Get Minimum Value Node

```
procedure GetMinValueNode(node)
    current ← node
    while current.left ≠ None do
        current ← current.left
    end while
    return current
end procedure
```

Search Procedure

The search operation remains unchanged from the standard BST, leveraging the AVL tree's balanced nature to ensure logarithmic time complexity.

Algorithm 6 AVL Search

```
procedure Search(node, key)
    if node = None or node.key = key then
        return node
    end if
    if key < node.key then
        return Search(node.left, key)
    else
        return Search(node.right, key)
    end if
end procedure
```

Additional Procedures

While the primary focus is on insertion, deletion, and search operations, the following helper methods are essential for testing and maintaining the AVL tree's properties.

In-order Traversal

Algorithm 7 In-order Traversal

```
procedure InOrder(node)
    if node = None then
        return []
    end if
    return InOrder(node.left) + [node.key] + InOrder(node.right)
end procedure
```

Print Tree

This procedure helps visualize the tree structure by printing it sideways.

Algorithm 8 Print Tree

```
procedure PrintTree(node, indent = "", last = 'updown')
    if node = None then
        return
    end if
    indent ← indent + ""
    PrintTree(node.right, indent, 'right')
    print (indent)
    if last = 'updown' then
        print "Root—-",
    else if last = 'right' then
        print "R—-",
    else if last = 'left' then
        print "L—-",
    end if
    print (node.key)
    PrintTree(node.left, indent, 'left')
end procedure
```

Check BST Property

Ensures that the tree satisfies the Binary Search Tree properties.

Algorithm 9 Check BST Property

```
procedure IsBSTUtil(node, left, right)
    if node = None then
        return True
    end if
    if left ≠ None and node.key ≤ left then
        return False
    end if
    if right ≠ None and node.key ≥ right then
        return False
    end if
    return (IsBSTUtil(node.left, left, node.key) and IsBSTUtil(node.right, node.key, right))
end procedure
procedure IsBST(node)
    return IsBSTUtil(node, None, None)
end procedure
```

Check AVL Balance

Verifies that the tree maintains AVL balance properties.

Algorithm 10 Check AVL Balance

```
procedure IsBalancedUtil(node)
    if node = None then
        return True
    end if
    balance ← GetBalance(node)
    if |balance| > 1 then
        return False
    end if
    return (IsBalancedUtil(node.left) and IsBalancedUtil(node.right))
end procedure
procedure IsBalanced(node)
    return IsBalancedUtil(node)
end procedure
```

Print Balance Factors

Prints the balance factor of each node in the tree.

Algorithm 11 Print Balance Factors

```
procedure PrintBalanceFactors(node)
    if node = None then
        return
    end if
    PrintBalanceFactors(node.left)
    print "Node " node.key " has balance factor " GetBalance(node)
    PrintBalanceFactors(node.right)
end procedure
```

Time Complexity Analysis

The AVL tree maintains its balance by ensuring that the height of the tree remains logarithmic relative to the number of nodes. Let's analyze the time complexity $T(n)$ for the major operations: search, insert, and delete.

Search Operation

- **Recurrence Relation**:

$$T_{\text{search}}(n) = T\left(\frac{n}{2}\right) + c$$

- **Solution**: By solving the recurrence relation, we have:

$$T_{\text{search}}(n) = c \cdot \log_2 n + d$$

  where c and d are constants.

Insert Operation

- **Recurrence Relation**:

$$T_{\text{insert}}(n) = T\left(\frac{n}{2}\right) + c$$

- **Solution**: Solving the recurrence relation similarly gives:

$$T_{\text{insert}}(n) = c \cdot \log_2 n + d$$

Delete Operation

- **Recurrence Relation**:

$$T_{\text{delete}}(n) = T\left(\frac{n}{2}\right) + c$$

- **Solution**: Solving the recurrence relation:

$$T_{\text{delete}}(n) = c \cdot \log_2 n + d$$

Total Time for m Operations

Assuming that all operations (search, insert, delete) are performed m times, the total time $T_{\text{total}}(m)$ can be expressed as:

$$T_{\text{total}}(m) = m \cdot (c \cdot \log_2 n + d)$$

Where:

- c is the constant time per operation (search, insert, delete).
- n is the number of nodes in the tree at any point.
- d is a constant representing lower-order terms.

Conclusion

The AVL tree enhances the standard BST by maintaining balance through rotations, ensuring that the height of the tree remains logarithmic relative to the number of nodes. This balance guarantees that search, insert, and delete operations perform efficiently, with time complexities proportional to $\log_2 n$. The addition of the "height" attribute and the implementation of rotation mechanisms are critical for maintaining the AVL property, thereby ensuring optimal performance across all major operations.