DSA (CSE102) Practice Questions for Lab

April 8, 2025

Practice Questions

Answer the following questions by implementing each function for an AVL tree. The AVL tree in this lab uses the right–left balance factor convention, where:

$$\text{BF(node)} = \text{height(right subtree)} - \text{height(left subtree)}$$

A positive BF indicates that the node is right heavy (which may require a left or right–left rotation), while a negative BF indicates left heaviness (which may need a right or left–right rotation).

Question 1: Implement the Balance Factor Function
Write a function that calculates the balance factor for a given node in an AVL tree. Your function should compute and return the difference in height between the right and left subtrees of the node using the formula:

$$\text{BF(node)} = \text{height(right)} - \text{height(left)}$$

Question 2: Implement Rotation Functions
Implement the following AVL tree rotation functions:

(a) **Single Left Rotation (RR Rotation)**
(b) **Single Right Rotation (LL Rotation)**
(c) **Left-Right Rotation (LR Rotation)**
(d) **Right-Left Rotation (RL Rotation)**

Each function should take as input an unbalanced node (and its subtree pointers) and return the new subtree root after performing the rotation(s).

Question 3: Implement Insertion
Write an insertion function for an AVL tree that:

(i) Inserts a new key using the standard Binary Search Tree insertion method.
(ii) Updates node heights and computes balance factors.
(iii) Detects any AVL imbalances and performs the appropriate rotation(s) (including both LR and RL rotations when needed) to restore balance.

Question 4: Implement Deletion
Write a function to delete a given key from an AVL tree. Your function should:

(i) Perform a standard BST deletion (handling leaf nodes, nodes with one child, and nodes with two children).
(ii) Retrace the path from the deletion point to the root to update heights and balance factors.
(iii) Identify any AVL imbalances and restore the AVL property by applying the necessary rotations (including both LR and RL rotations).

Solutions

Below are the function implementations required to answer the practice questions.

Helper Functions and Data Structure

```cpp
// Node structure definition
struct Node {
    int key;
    int height;
    Node* left;
    Node* right;
};

// Utility function to return maximum of two integers.
int maxVal(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Returns the height of a node.
int height(Node* node) {
    if (node == nullptr) {
        return 0;
    } else {
        return node->height;
    }
}

// Creates a new node with the given key.
Node* newNode(int key) {
    Node* node = new Node;
    node->key = key;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}
```

Question 1: Balance Factor Function

```cpp
// Computes the Balance Factor (BF) of a node
// BF = height(right subtree) - height(left subtree)
int getBalanceFactor(Node* node) {
    if (node == nullptr) {
        return 0;
    } else {
        return height(node->right) - height(node->left);
    }
}
```

Question 2: Rotation Functions

Single Left Rotation (RR Rotation)

```
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = maxVal(height(x->left), height(x->right)) + 1;
    y->height = maxVal(height(y->left), height(y->right)) + 1;

    return y;
}
```

Single Right Rotation (LL Rotation)

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = maxVal(height(y->left), height(y->right)) + 1;
    x->height = maxVal(height(x->left), height(x->right)) + 1;

    return x;
}
```

Left-Right Rotation (LR Rotation)

```
Node* leftRightRotate(Node* node) {
    // First, left rotate the left child.
    node->left = leftRotate(node->left);
    // Then, right rotate the node.
    return rightRotate(node);
}
```

Right-Left Rotation (RL Rotation)

```
Node* rightLeftRotate(Node* node) {
    // First, right rotate the right child.
    node->right = rightRotate(node->right);
    // Then, left rotate the node.
    return leftRotate(node);
}
```

Question 3: Insertion Function

```
Node* insertNode(Node* node, int key) {
    // Standard BST insertion.
    if (node == nullptr) {
        return newNode(key);
    }

    if (key < node->key) {
        node->left = insertNode(node->left, key);
    } else if (key > node->key) {
        node->right = insertNode(node->right, key);
    } else {
        return node;  // Duplicate keys not allowed.
    }

    // Update the height of the node.
    node->height = 1 + maxVal(height(node->left), height(node->right));

    // Compute the balance factor.
    int balance = getBalanceFactor(node);

    // If node is right heavy (BF > +1)
    if (balance > 1) {
        if (key > node->right->key) {
            // RR case: single left rotation.
            return leftRotate(node);
        } else if (key < node->right->key) {
            // RL case: right rotate the right child, then left rotate.
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
    }

    // If node is left heavy (BF < -1)
    if (balance < -1) {
        if (key < node->left->key) {
            // LL case: single right rotation.
            return rightRotate(node);
        } else if (key > node->left->key) {
            // LR case: left rotate the left child, then right rotate.
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
    }

    return node;
}
```

Question 4: Deletion Function

```
// Finds the node with the smallest key in a subtree.
Node* nodeWithMinimumValue(Node* node) {
    Node* current = node;
    while (current != nullptr && current->left != nullptr) {
        current = current->left;
    }
    return current;
}

// Deletes a key from the AVL tree and rebalances the tree.
Node* deleteNode(Node* root, int key) {
    // Standard BST deletion.
    if (root == nullptr) {
        return root;
    }

    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        // Node found.
        if (root->left == nullptr || root->right == nullptr) {
            Node* temp;
            if (root->left != nullptr) {
                temp = root->left;
            } else {
                temp = root->right;
            }

            if (temp == nullptr) {
                // No child case.
                temp = root;
                root = nullptr;
            } else {
                // One child case.
                *root = *temp;
            }
            delete temp;
        } else {
            // Node with two children: get the in-order successor (smallest in the
                right subtree).
            Node* temp = nodeWithMinimumValue(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

    if (root == nullptr) {
        return root;
    }

    // Update the height.
    root->height = 1 + maxVal(height(root->left), height(root->right));

    // Compute the balance factor.
    int balance = getBalanceFactor(root);
```

```

    // If root is right heavy.
    if (balance > 1) {
        if (getBalanceFactor(root->right) >= 0) {
            // RR case.
            return leftRotate(root);
        } else {
            // RL case.
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }
    }

    // If root is left heavy.
    if (balance < -1) {
        if (getBalanceFactor(root->left) <= 0) {
            // LL case.
            return rightRotate(root);
        } else {
            // LR case.
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }
    }

    return root;
}
```