CS221-L Data Structures and Algorithms Lab

Lab # 09

Submitted by:

- **Shayan Rizwan** [2024585]

Submitted to: Sir Adnan Haider

Semester: 3rd

Faculty of Computer Science and Engineering

GIK Institute of Engineering Sciences and Technology

Task # 1:

Implement an AVL Tree with an additional feature to track rebalancing activity for each node. For every node in the tree, maintain a counter that records the number of times the node's balance factor was recalculated and resulted in a change.

Example Output:

Node 10 was rebalanced 4 times
Node 5 was rebalanced 3 times
Node 20 was rebalanced 2 times

Code:

```
#include <iostream>

using namespace std;

// AVL Tree Node structure
struct AVLNode {
  int data;           // Value stored in node
  int height;         // Height of node (for balance calculation)
  int rebalanceCount; // Counter for rebalancing activities
  AVLNode* left;      // Pointer to left child
  AVLNode* right;     // Pointer to right child

  // Constructor to initialize node
  AVLNode(int value) {
    data = value;
    height = 0;     // New node has height 0 (leaf)
    rebalanceCount = 0;
    left = right = NULL;
  }
};

// AVL Tree Class
class AVLTree {
private:
  AVLNode* root;      // Root of the AVL tree

  // Helper: Get height of a node (handles null case)
  int getHeight(AVLNode* node) {
    if (node == NULL) {
```

```
    return -1;  // Empty node has height -1
  } else {
    return node->height;
  }
}

// Helper: Calculate balance factor (height difference between subtrees)
// Positive = left heavy, Negative = right heavy
int getBalanceFactor(AVLNode* node) {
  if (node == NULL) return 0;
  return getHeight(node->left) - getHeight(node->right);
}

// Helper: Update height based on children's heights
void updateHeight(AVLNode* node) {
  if (node == NULL) return;

  int leftHeight = getHeight(node->left);
  int rightHeight = getHeight(node->right);

  // Height is max of children's heights + 1
  if (leftHeight > rightHeight) {
    node->height = leftHeight + 1;
  } else {
    node->height = rightHeight + 1;
  }
}

// Right Rotation (for left-left imbalance)
AVLNode* rotateRight(AVLNode* y) {
  AVLNode* x = y->left;
  AVLNode* T2 = x->right;

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

  // Update heights after rotation
  updateHeight(y);
  updateHeight(x);

  return x; // New root after rotation
}
```

```
  // Left Rotation (for right-right imbalance)
  AVLNode* rotateLeft(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

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

    // Update heights after rotation
    updateHeight(x);
    updateHeight(y);

    return y; // New root after rotation
  }

  // Balance the tree and increment rebalance counter when needed
  // This function checks if a node is unbalanced and performs rotations
  AVLNode* balanceTree(AVLNode* node) {
    if (node == NULL) return node;

    // Update height of current node first
    updateHeight(node);

    // Calculate balance factor to check if rebalancing is needed
    int balance = getBalanceFactor(node);

    // KEY FIX: Only increment counter if this node is actually unbalanced
    // A node is unbalanced when |balance factor| > 1
    // This means it needs rebalancing through rotations
    if (balance > 1 || balance < -1) {
      // This node IS unbalanced, so increment its counter
      node->rebalanceCount++;
    }

    // Now perform the appropriate rotations based on imbalance type

    // Case 1: Left-Left (LL) - node is left heavy, left child is also left
heavy or balanced
    if (balance > 1 && getBalanceFactor(node->left) >= 0) {
      return rotateRight(node);
    }
```

```
    // Case 2: Left-Right (LR) - node is left heavy, but left child is right
heavy
    if (balance > 1 && getBalanceFactor(node->left) < 0) {
      node->left = rotateLeft(node->left);
      return rotateRight(node);
    }

    // Case 3: Right-Right (RR) - node is right heavy, right child is also
right heavy or balanced
    if (balance < -1 && getBalanceFactor(node->right) <= 0) {
      return rotateLeft(node);
    }

    // Case 4: Right-Left (RL) - node is right heavy, but right child is left
heavy
    if (balance < -1 && getBalanceFactor(node->right) > 0) {
      node->right = rotateRight(node->right);
      return rotateLeft(node);
    }

    return node; // No rotation needed, tree is balanced
  }

  // Recursive insertion with balancing
  // Inserts a value and rebalances the tree on the way back up
  AVLNode* insertRecursive(AVLNode* node, int value) {
    // Step 1: Standard BST insertion
    if (node == NULL) {
      return new AVLNode(value);
    }

    // Insert in left subtree if value is smaller
    if (value < node->data) {
      node->left = insertRecursive(node->left, value);
    }
    // Insert in right subtree if value is larger
    else if (value > node->data) {
      node->right = insertRecursive(node->right, value);
    }
    // Duplicate values not allowed
    else {
      return node;
```

```
    }

    // Step 2: After insertion, balance this node on the way back up
    // This checks if current node became unbalanced due to insertion
    // and performs rotations if needed
    return balanceTree(node);
  }

  // Inorder traversal to display rebalance counts
  // Visits nodes in sorted order (left, root, right)
  void displayRebalanceCounts(AVLNode* node) {
    if (node == NULL) return;

    // Traverse left subtree
    displayRebalanceCounts(node->left);

    // Display current node's rebalance count
    cout << "Node " << node->data << " was rebalanced "
       << node->rebalanceCount << " time";
    if (node->rebalanceCount != 1) {
      cout << "s";
    }
    cout << endl;

    // Traverse right subtree
    displayRebalanceCounts(node->right);
  }

  // Helper to delete tree (clean up memory)
  void deleteTree(AVLNode* node) {
    if (node == NULL) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
  }

public:
  // Constructor
  AVLTree() {
    root = NULL;
  }

  // Destructor
```

```
  ~AVLTree() {
    deleteTree(root);
  }

  // Public method to insert value
  void insert(int value) {
    root = insertRecursive(root, value);
  }

  // Public method to display rebalance counts
  void displayRebalanceStats() {
    if (root == NULL) {
      cout << "Tree is empty!" << endl;
      return;
    }

    cout << "\n--- Rebalancing Statistics ---" << endl;
    displayRebalanceCounts(root);
  }

  // Get height of tree
  int getTreeHeight() {
    return getHeight(root);
  }

  // Check if tree is balanced at root level
  bool isBalanced() {
    int balance = getBalanceFactor(root);
    if (balance < 0) balance = -balance; // Get absolute value
    return balance <= 1;
  }
};

// Main function to demonstrate AVL Tree with rebalancing tracking
int main() {
  AVLTree avl;
  int n, value;

  // Get number of elements
  cout << "Enter number of elements to insert: ";
  cin >> n;

  // Insert elements one by one
```

```
  cout << "Enter " << n << " integer elements:" << endl;
  for (int i = 0; i < n; i++) {
    cin >> value;
    avl.insert(value);
  }

  // Display rebalancing statistics for all nodes
  avl.displayRebalanceStats();

  // Additional validation information
  cout << "\n--- Validation ---" << endl;
  cout << "Tree height: " << avl.getTreeHeight() << endl;
  cout << "Tree is ";
  if (avl.isBalanced()) {
    cout << "balanced";
  } else {
    cout << "NOT balanced";
  }
  cout << endl;

  return 0;
}
```

Output:

```
Enter number of elements to insert: 5
Enter 5 integer elements:
9 4 5 10 20

--- Rebalancing Statistics ---
Node 4 was rebalanced 0 times
Node 5 was rebalanced 0 times
Node 9 was rebalanced 2 times
Node 10 was rebalanced 0 times
Node 20 was rebalanced 0 times

--- Validation ---
Tree height: 2
Tree is balanced
```

Task # 2:

Write a function to traverse a Binary Search Tree (BST) and print all paths from the root node to every leaf node.

Example BST Traversal Output:

Path: 50→30→20
Path: 50→30→40
Path: 50→70→60
Path: 50→70→80

Code:

```
#include <iostream>
using namespace std;

// Definition of a single node in the Binary Search Tree
struct Node {
  int data;           // Value stored in the node
  Node* left;         // Pointer to the left child
  Node* right;        // Pointer to the right child

  // Constructor to initialize a new node with given value
  Node(int value) {
    data = value;
    left = NULL;
    right = NULL;
  }
};

// Function to insert a new value into the BST
// Returns the root of the modified tree
Node* insert(Node* root, int value) {
  // Base case: if tree is empty or we reached a NULL position
  // Create and return a new node
  if (root == NULL) {
    return new Node(value);
  }

  // Recursive case: traverse left or right based on BST property
```

```
  // If value is smaller, insert in left subtree
  if (value < root->data) {
    root->left = insert(root->left, value);
  }
  // If value is greater, insert in right subtree
  else if (value > root->data) {
    root->right = insert(root->right, value);
  }
  // If value already exists, do nothing (no duplicates)

  return root;
}

// Helper function to print a single path stored in the array
// Prints from index 0 to pathLength-1 with arrow separators
void printPath(int path[], int pathLength) {
  cout << "Path: ";
  for (int i = 0; i < pathLength; i++) {
    cout << path[i];
    // Add arrow separator between nodes, but not after the last node
    if (i < pathLength - 1) {
      cout << " -> ";
    }
  }
  cout << endl;
}

// Recursive function to find and print all root-to-leaf paths
// Parameters:
//   node: current node being processed
//   path: array storing the current path from root to this node
//   pathLength: number of nodes in the current path
void printAllPaths(Node* node, int path[], int pathLength) {
  // Base case: if current node is NULL, return immediately
  if (node == NULL) {
    return;
  }

  // Add current node's data to the path array
  path[pathLength] = node->data;
  pathLength++;  // Increment path length after adding current node

  // Check if current node is a leaf node
```

```
  // A leaf node has no left child and no right child
  if (node->left == NULL && node->right == NULL) {
    // We've reached a leaf, so print the complete path
    printPath(path, pathLength);
  }
  else {
    // Not a leaf node, so continue traversing both subtrees
    // Recursively traverse left subtree
    printAllPaths(node->left, path, pathLength);

    // Recursively traverse right subtree
    printAllPaths(node->right, path, pathLength);
  }

  // Backtracking happens automatically when function returns
  // The pathLength parameter is passed by value, so it resets
  // when we return to the previous recursive call
}

// Wrapper function to initiate the path printing process
void findAllPaths(Node* root) {
  // Handle edge case: empty tree
  if (root == NULL) {
    cout << "Tree is empty!" << endl;
    return;
  }

  // Array to store the current path during traversal
  // Maximum possible height of BST is the number of nodes
  // Allocating 1000 as a safe upper limit for path storage
  int path[1000];

  // Start the recursive traversal from root with empty path
  printAllPaths(root, path, 0);
}

int main() {
  // Initialize an empty BST
  Node* root = NULL;

  // Build the example BST from the problem statement
  //           50
  //          /  \
```

```
  //         30    70
  //        / \    / \
  //       20 40  60 80

  root = insert(root, 50);
  root = insert(root, 30);
  root = insert(root, 70);
  root = insert(root, 20);
  root = insert(root, 40);
  root = insert(root, 60);
  root = insert(root, 80);

  cout << "All paths from root to leaf nodes:" << endl;
  cout << "==================================" << endl;

  // Find and print all root-to-leaf paths
  findAllPaths(root);

  return 0;
}
```

Output:

```
All paths from root to leaf nodes:
==================================
Path: 50 -> 30 -> 20
Path: 50 -> 30 -> 40
Path: 50 -> 70 -> 60
Path: 50 -> 70 -> 80
```

Task # 3:

Write a function rangeSum(root, L, R) that calculates the sum of all node values in a Binary Search Tree (BST) that lie within a given inclusive range [L, R].

Example:

BST={10,5,15,3,7,18}

Range = [7, 15]

Output = 7+10+15=32

Code:

```
#include <iostream>
using namespace std;

// Definition of a single node in the Binary Search Tree
struct Node {
  int data;           // Value stored in the node
  Node* left;         // Pointer to the left child
  Node* right;        // Pointer to the right child

  // Constructor to initialize a new node with given value
  Node(int value) {
    data = value;
    left = NULL;
    right = NULL;
  }
};

// Function to insert a new value into the BST
// Returns the root of the modified tree
Node* insert(Node* root, int value) {
  // Base case: if tree is empty or we reached a NULL position
  // Create and return a new node
  if (root == NULL) {
    return new Node(value);
```

```
  }

  // Recursive case: traverse left or right based on BST property
  // If value is smaller, insert in left subtree
  if (value < root->data) {
    root->left = insert(root->left, value);
  }
  // If value is greater, insert in right subtree
  else if (value > root->data) {
    root->right = insert(root->right, value);
  }
  // If value already exists, do nothing (no duplicates)

  return root;
}

// Function to calculate sum of all nodes in range [L, R]
// This uses the BST property to optimize the traversal
// Parameters:
//   root: current node being processed
//   L: lower bound of range (inclusive)
//   R: upper bound of range (inclusive)
// Returns: sum of all node values within [L, R]
int rangeSum(Node* root, int L, int R) {
  // Base case: if current node is NULL, return 0
  // No contribution to sum from empty subtree
  if (root == NULL) {
    return 0;
  }

  // Initialize sum to 0
  int sum = 0;

  // OPTIMIZATION 1: Check if current node is less than L
  // If current node < L, then ALL nodes in left subtree are also < L
  // (due to BST property: left < root)
  // So we can skip the entire left subtree
  if (root->data < L) {
    // Only explore right subtree (which might have values >= L)
    return rangeSum(root->right, L, R);
  }

  // OPTIMIZATION 2: Check if current node is greater than R
```

```
  // If current node > R, then ALL nodes in right subtree are also > R
  // (due to BST property: root < right)
  // So we can skip the entire right subtree
  if (root->data > R) {
    // Only explore left subtree (which might have values <= R)
    return rangeSum(root->left, L, R);
  }

  // If we reach here, it means L <= root->data <= R
  // Current node is within range, so include it in sum
  sum = root->data;

  // Recursively explore both subtrees
  // Left subtree might have values >= L (and <= current node)
  // Right subtree might have values <= R (and >= current node)
  sum += rangeSum(root->left, L, R);
  sum += rangeSum(root->right, L, R);

  return sum;
}

// Helper function to perform inorder traversal and display tree
// This helps visualize the BST structure
void inorderTraversal(Node* root) {
  if (root == NULL) return;

  // Traverse left subtree
  inorderTraversal(root->left);

  // Print current node
  cout << root->data << " ";

  // Traverse right subtree
  inorderTraversal(root->right);
}

int main() {
  // Initialize an empty BST
  Node* root = NULL;
  int n, value;

  // Get number of nodes from user
  cout << "Enter number of nodes in BST: ";
```

```
  cin >> n;

  // Build the BST by inserting values
  cout << "Enter " << n << " integer values:" << endl;
  for (int i = 0; i < n; i++) {
    cin >> value;
    root = insert(root, value);
  }

  // Display the BST in sorted order (inorder traversal)
  cout << "\nBST (in sorted order): ";
  inorderTraversal(root);
  cout << endl;

  // Get the range [L, R] from user
  int L, R;
  cout << "\nEnter the range [L, R]:" << endl;
  cout << "Enter L (lower bound): ";
  cin >> L;
  cout << "Enter R (upper bound): ";
  cin >> R;

  // Calculate and display the range sum
  int result = rangeSum(root, L, R);

  cout << "\n========================================" << endl;
  cout << "Range: [" << L << ", " << R << "]" << endl;
  cout << "Sum of nodes in range: " << result << endl;
  cout << "========================================" << endl;

  return 0;
}
```

Output:

```
Enter number of nodes in BST: 5
Enter 5 integer values:
9 4 5 10 20

BST (in sorted order): 4 5 9 10 20

Enter the range [L, R]:
Enter L (lower bound): 1 15
Enter R (upper bound):
========================================
Range: [1, 15]
Sum of nodes in range: 28
========================================
```

Task # 4:

Implement an AVL Tree insertion function with a special rule that restricts insertions which cause multiple imbalances during the same operation.

Special Rule for this Task:

If more than one node becomes imbalanced (BF = +2 or -2) during a single insertion:
Stop the insertion immediately.
Do not modify the tree.

Code:

```
#include <iostream>
using namespace std;

struct AVLNode {
  int data;
  int height;
  AVLNode* left;
  AVLNode* right;

  AVLNode(int value) {
    data = value;
    height = 0;
    left = right = NULL;
  }
};

class AVLTree {
private:
  AVLNode* root;
  int imbalanceCount;

  int getHeight(AVLNode* node) {
    if (node == NULL) return -1;
    return node->height;
  }

  int getBalanceFactor(AVLNode* node) {
```

```
    if (node == NULL) return 0;
    return getHeight(node->left) - getHeight(node->right);
  }

  void updateHeight(AVLNode* node) {
    if (node == NULL) return;
    int leftHeight = getHeight(node->left);
    int rightHeight = getHeight(node->right);
    node->height = (leftHeight > rightHeight ? leftHeight : rightHeight) +
1;
  }

  AVLNode* copyTree(AVLNode* node) {
    if (node == NULL) return NULL;
    AVLNode* newNode = new AVLNode(node->data);
    newNode->height = node->height;
    newNode->left = copyTree(node->left);
    newNode->right = copyTree(node->right);
    return newNode;
  }

  void deleteTree(AVLNode* node) {
    if (node == NULL) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
  }

  AVLNode* insertBST(AVLNode* node, int value) {
    if (node == NULL) return new AVLNode(value);

    if (value < node->data) {
      node->left = insertBST(node->left, value);
    } else if (value > node->data) {
      node->right = insertBST(node->right, value);
    } else {
      return node;
    }

    updateHeight(node);
    int balance = getBalanceFactor(node);

    // CORRECTED: Check if |BF| > 1 (not just == 2 or == -2)
```

```
    if (balance > 1 || balance < -1) {
      imbalanceCount++;
    }

    return node;
  }

  void inorderDisplay(AVLNode* node) {
    if (node == NULL) return;
    inorderDisplay(node->left);
    cout << node->data << " ";
    inorderDisplay(node->right);
  }

public:
  AVLTree() {
    root = NULL;
    imbalanceCount = 0;
  }

  ~AVLTree() {
    deleteTree(root);
  }

  bool insert(int value) {
    AVLNode* backup = copyTree(root);
    imbalanceCount = 0;
    root = insertBST(root, value);

    if (imbalanceCount > 1) {
      deleteTree(root);
      root = backup;
      return false;
    } else {
      deleteTree(backup);
      return true;
    }
  }

  void displayInorder() {
    if (root == NULL) {
      cout << "Tree is empty!" << endl;
    } else {
```

```
      cout << "Tree (inorder): ";
      inorderDisplay(root);
      cout << endl;
    }
  }
};

int main() {
  AVLTree avl;
  int value;

  cout << "AVL Tree with Special Insertion Rule" << endl;
  cout << "Keep entering values. Program stops if insertion causes multiple
imbalances.\n" << endl;


  cout << "Insert values in a balanced pattern (middle-first, alternating,
etc.):" << endl << endl;

  while (true) {
    cout << "Enter value: ";
    cin >> value;

    if (avl.insert(value)) {
      cout << "Inserted " << value << " successfully.\n" << endl;
    } else {
      cout << "\nINSERTION REJECTED! Value " << value << " causes multiple
imbalances." << endl;
      cout << "Program stopped.\n" << endl;
      break;
    }
  }

  cout << "Final tree (inorder): ";
  avl.displayInorder();

  return 0;
}
```

Output:

```
AVL Tree with Special Insertion Rule
Keep entering values. Program stops if insertion causes multiple imbalances.

Insert values in a balanced pattern (middle-first, alternating, etc.):

Enter value: 20
Inserted 20 successfully.

Enter value: 10
Inserted 10 successfully.

Enter value: 30
Inserted 30 successfully.

Enter value: 40
Inserted 40 successfully.

Enter value: 50

INSERTION REJECTED! Value 50 causes multiple imbalances.
Program stopped.

Final tree (inorder): Tree (inorder): 10 20 30 40
```