# 

# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

# Lab 11: Balanced BST Operations

**Name: M.Haadhee Sheeraz Mian**

**Reg no: 478359**

**Date: 31st July, 2025**

**Time: 3:00pm – 4:50pm**

# Course Instructor: Dr. Farzana Jabeen

# Lab 11: Balanced BST Operations

**Objectives**

Objective of this lab is to give understanding of balanced BSTs through hands-on tasks involving insertion, and deletion operations in AVL trees.

**Tools/Software Requirement**

Visual Studio C++

**Lab Tasks:**

**Task 1: Implement AVL Tree Insertion**

**1. Instructions:**

* Implement insertion in an AVL tree, as discussed in class.
* Your implementation must handle all rotation cases to maintain balance:

■ Right Rotation (LL)

■ Left Rotation (RR)

■ Left-Right Rotation (LR)

■ Right-Left Rotation (RL)

**2. Steps:**

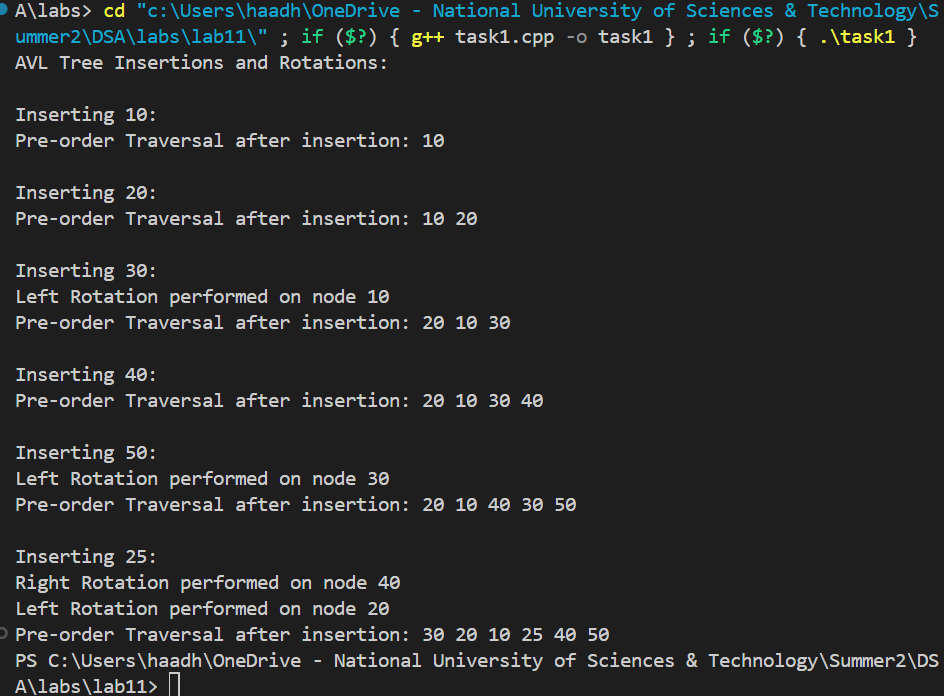
* Start by writing a basic BST insertion.
* Add balancing logic to detect and correct imbalance after insertion.
* Test by inserting a sequence of integers and validate that the tree remains balanced.

**3. Requirements:**

* Output the tree structure after each insertion (use pre-order traversal for testing).
* Ensure your code handles edge cases (e.g., inserting duplicates).

**4. Deliverable:**

* Submit the code with comments explaining each rotation and the final output of the balanced tree.
* #include <iostream>
* using namespace std;
* // Node structure
* struct Node {
* int key;
* Node\* left;
* Node\* right;
* int height;
* Node(int val) : key(val), left(NULL), right(NULL), height(1) {}
* };
* // Utility to get height
* int getHeight(Node\* node) {
* return node ? node->height : 0;
* }
* // Utility to get balance factor
* int getBalance(Node\* node) {
* return node ? getHeight(node->left) - getHeight(node->right) : 0;
* }
* // Update height of a node
* void updateHeight(Node\* node) {
* if (node)
* node->height = 1 + max(getHeight(node->left), getHeight(node->right));
* }
* // Right Rotation (LL Case)
* Node\* rightRotate(Node\* y) {
* Node\* x = y->left;
* Node\* T2 = x->right;
* // Rotation
* x->right = y;
* y->left = T2;
* // Update heights
* updateHeight(y);
* updateHeight(x);
* cout << "Right Rotation performed on node " << y->key << "\n";
* return x;
* }
* // Left Rotation (RR Case)
* Node\* leftRotate(Node\* x) {
* Node\* y = x->right;
* Node\* T2 = y->left;
* // Rotation
* y->left = x;
* x->right = T2;
* // Update heights
* updateHeight(x);
* updateHeight(y);
* cout << "Left Rotation performed on node " << x->key << "\n";
* return y;
* }
* // Insert into AVL Tree and return new root
* Node\* insert(Node\* node, int key) {
* if (!node) return new Node(key);
* // Basic BST Insertion
* if (key < node->key)
* node->left = insert(node->left, key);
* else if (key > node->key)
* node->right = insert(node->right, key);
* else {
* cout << "Duplicate key " << key << " ignored.\n";
* return node; // duplicate keys not allowed
* }
* // Update height and balance
* updateHeight(node);
* int balance = getBalance(node);
* // Case 1: LL
* if (balance > 1 && key < node->left->key)
* return rightRotate(node);
* // Case 2: RR
* if (balance < -1 && key > node->right->key)
* return leftRotate(node);
* // Case 3: LR
* if (balance > 1 && key > node->left->key) {
* node->left = leftRotate(node->left);
* return rightRotate(node);
* }
* // Case 4: RL
* if (balance < -1 && key < node->right->key) {
* node->right = rightRotate(node->right);
* return leftRotate(node);
* }
* return node;
* }
* // Pre-order traversal for tree structure output
* void preOrder(Node\* root) {
* if (root) {
* cout << root->key << " ";
* preOrder(root->left);
* preOrder(root->right);
* }
* }
* // Run and test insertions
* int main() {
* Node\* root = NULL;
* int values[] = { 10, 20, 30, 40, 50, 25 };
* cout << "AVL Tree Insertions and Rotations:\n";
* for (int i = 0; i < 6; i++) {
* cout << "\nInserting " << values[i] << ":\n";
* root = insert(root, values[i]);
* cout << "Pre-order Traversal after insertion: ";
* preOrder(root);
* cout << "\n";
* }
* return 0;
* }

****

**Task 2: AVL Tree Deletion Logic and Coding**

**1. Part 1: Algorithm Development**

* Work in pairs to brainstorm the logic for deletion in an AVL tree.
* Key points to consider:

■ How to handle the deletion of nodes with 0, 1, or 2 children.

■ How to rebalance the tree after deletion.

* Document your logic as a step-by-step algorithm.

**2. Part 2: Coding**

* Implement the deletion operation in your AVL tree.
* Integrate rebalancing logic (rotations) to maintain the AVL property.
* Test with different scenarios, such as deleting leaf nodes, internal nodes, and the root.

**3. Deliverable:**

* Submit the algorithm as a separate document (e.g., PDF).
* Provide code and screenshots of the AVL tree before and after deletions.

**Deliverables**



* C++ code file implementing the above tasks.
* Brief comments in the code explaining the logic.
* Word Documents includes the screenshots of code and output.

**Note:** Students are required to upload the lab on LMS before deadline.

Use proper indentation and comments.

**Solution**

|  |
| --- |
| Solution |
| Tasks Code screenshot:  #include <iostream>  using namespace std;  // Node structure  struct Node {      int key;      Node\* left;      Node\* right;      int height;      Node(int val) : key(val), left(NULL), right(NULL), height(1) {}  };  // Utility to get height  int getHeight(Node\* node) {      return node ? node->height : 0;  }  // Utility to get balance factor  int getBalance(Node\* node) {      return node ? getHeight(node->left) - getHeight(node->right) : 0;  }  // Update height of a node  void updateHeight(Node\* node) {      if (node)          node->height = 1 + max(getHeight(node->left), getHeight(node->right));  }  // Right Rotation (LL Case)  Node\* rightRotate(Node\* y) {      Node\* x = y->left;      Node\* T2 = x->right;      // Rotation      x->right = y;      y->left = T2;      // Update heights      updateHeight(y);      updateHeight(x);      cout << "Right Rotation performed on node " << y->key << "\n";      return x;  }  // Left Rotation (RR Case)  Node\* leftRotate(Node\* x) {      Node\* y = x->right;      Node\* T2 = y->left;      // Rotation      y->left = x;      x->right = T2;      // Update heights      updateHeight(x);      updateHeight(y);      cout << "Left Rotation performed on node " << x->key << "\n";      return y;  }  // Insert into AVL Tree and return new root  Node\* insert(Node\* node, int key) {      if (!node) return new Node(key);      // Basic BST Insertion      if (key < node->key)          node->left = insert(node->left, key);      else if (key > node->key)          node->right = insert(node->right, key);      else {          cout << "Duplicate key " << key << " ignored.\n";          return node; // duplicate keys not allowed      }      // Update height and balance      updateHeight(node);      int balance = getBalance(node);      // Case 1: LL      if (balance > 1 && key < node->left->key)          return rightRotate(node);      // Case 2: RR      if (balance < -1 && key > node->right->key)          return leftRotate(node);      // Case 3: LR      if (balance > 1 && key > node->left->key) {          node->left = leftRotate(node->left);          return rightRotate(node);      }      // Case 4: RL      if (balance < -1 && key < node->right->key) {          node->right = rightRotate(node->right);          return leftRotate(node);      }      return node;  }  // Pre-order traversal for tree structure output  void preOrder(Node\* root) {      if (root) {          cout << root->key << " ";          preOrder(root->left);          preOrder(root->right);      }  }  // Utility to find the node with the smallest key  Node\* minValueNode(Node\* node) {      Node\* current = node;      while (current->left != NULL)          current = current->left;      return current;  }  // Delete a node from AVL Tree  Node\* deleteNode(Node\* root, int key) {      if (!root) return root;      // Step 1: Normal BST deletion      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 || !root->right) {              Node\* temp = root->left ? root->left : root->right;              if (!temp) { // No child                  temp = root;                  root = NULL;              } else { // One child                  \*root = \*temp;              }              delete temp;          } else {              // Two children: Get inorder successor              Node\* temp = minValueNode(root->right);              root->key = temp->key;              root->right = deleteNode(root->right, temp->key);          }      }      // If tree had only one node      if (!root) return root;      // Step 2: Update height      updateHeight(root);      // Step 3: Check balance      int balance = getBalance(root);      // Step 4: Balance the tree      // LL      if (balance > 1 && getBalance(root->left) >= 0)          return rightRotate(root);      // LR      if (balance > 1 && getBalance(root->left) < 0) {          root->left = leftRotate(root->left);          return rightRotate(root);      }      // RR      if (balance < -1 && getBalance(root->right) <= 0)          return leftRotate(root);      // RL      if (balance < -1 && getBalance(root->right) > 0) {          root->right = rightRotate(root->right);          return leftRotate(root);      }      return root;  }  // Run and test insertions  int main() {      Node\* root = NULL;      int values[] = { 10, 20, 30, 40, 50, 25 };      cout << "AVL Tree Insertions and Rotations:\n";      for (int i = 0; i < 6; i++) {          cout << "\nInserting " << values[i] << ":\n";          root = insert(root, values[i]);          cout << "Pre-order Traversal after insertion: ";          preOrder(root);          cout << "\n";              cout << "\nDeleting 40:\n";          root = deleteNode(root, 40);          cout << "Pre-order after deletion: ";          preOrder(root);          cout << "\n";          cout << "\nDeleting 30:\n";          root = deleteNode(root, 30);          cout << "Pre-order after deletion: ";          preOrder(root);          cout << "\n";          }      return 0;  }  Tasks Output Screenshot: |