|  |  |
| --- | --- |
| **AVL Tree in C++** | |
| #include <iostream>  #include <algorithm> // For max() function  using namespace std;  struct node {  int val;  struct node\* left;  struct node\* right;  int ht;  } ;  int height(node\* n) {  if (n == NULL)  return -1;  return n->ht;  }  int balanceFactor(node\* n) {  if (n == NULL)  return 0;  return height(n->left) - height(n->right);  }  node\* rotateRight(node\* y) {  node\* x = y->left;  node\* T2 = x->right;  // Perform rotation  x->right = y;  y->left = T2;  // Update heights  y->ht = max(height(y->left), height(y->right)) + 1;  x->ht = max(height(x->left), height(x->right)) + 1;  // Return new root  return x;  }  node\* rotateLeft(node\* x) {  node\* y = x->right;  node\* T2 = y->left;  // Perform rotation  y->left = x;  x->right = T2;  // Update heights  x->ht = max(height(x->left), height(x->right)) + 1;  y->ht = max(height(y->left), height(y->right)) + 1;  // Return new root  return y;  }  node\* newNode(int value) {  node\* n = new node();  n->val = value;  n->left = NULL;  n->right = NULL;  n->ht = 0; // Height of the node is set to 0  return n;  }  node\* insert(node\* root, int new\_val) {  // Perform the normal BST insert  if (root == NULL)  return newNode(new\_val);  if (new\_val < root->val)  root->left = insert(root->left, new\_val);  else if (new\_val > root->val)  root->right = insert(root->right, new\_val);  else  return root; // Duplicate values are not allowed  // Update the height of the ancestor node  root->ht = 1 + max(height(root->left), height(root->right));  // Get the balance factor  int bf = balanceFactor(root);  // If the node becomes unbalanced, there are 4 cases:  // Case 1 - Left Left  if (bf > 1 && new\_val < root->left->val)  return rotateRight(root);  // Case 2 - Right Right  if (bf < -1 && new\_val > root->right->val)  return rotateLeft(root);  // Case 3 - Left Right  if (bf > 1 && new\_val > root->left->val) {  root->left = rotateLeft(root->left);  return rotateRight(root);  }  // Case 4 - Right Left  if (bf < -1 && new\_val < root->right->val) {  root->right = rotateRight(root->right);  return rotateLeft(root);  }  // Return the (unchanged) root pointer  return root;  }  // In-order traversal to print the tree in sorted order  void inOrderTraversal(node\* root) {  if (root != NULL) {  inOrderTraversal(root->left);  cout << root->val << " ";  inOrderTraversal(root->right);  }  }  // Pre-order traversal to show the structure of the AVL tree  void preOrderTraversal(node\* root) {  if (root != NULL) {  cout << root->val << " ";  preOrderTraversal(root->left);  preOrderTraversal(root->right);  }  }  // Main function to test the AVL tree implementation  int main() {  node\* root = NULL;  // Insert values into the AVL tree  root = insert(root, 10);  root = insert(root, 20);  root = insert(root, 30);  root = insert(root, 40);  root = insert(root, 50);  root = insert(root, 25);  // In-order traversal of the AVL tree (should be sorted)  cout << "In-order traversal: ";  inOrderTraversal(root);  cout << endl;  // Pre-order traversal of the AVL tree (shows the structure)  cout << "Pre-order traversal: ";  preOrderTraversal(root);  cout << endl;  return 0;  } | Input Sequence: 10, 20, 30, 40, 50, 25 We'll look at:   * The value being inserted * Balancing case (if any) * Rotation applied * Root after insertion  🌱 Step-by-Step Insertion and Rotations  | **Step** | **Inserted Value** | **Balance Factor (BF) at Unbalanced Node** | **Case** | **Rotation** | **New Root** | | --- | --- | --- | --- | --- | --- | | 1 | 10 | — | — | — | 10 | | 2 | 20 | — | — | — | 10 | | 3 | 30 | -2 at 10 (BF = -2) | Right Right | Left Rotate(10) | 20 | | 4 | 40 | -1 at 20 | — | — | 20 | | 5 | 50 | -2 at 20 (BF = -2) | Right Right | Left Rotate(20) | 30 | | 6 | 25 | +2 at 40.left (20), then -1 at 40 | Right Left | Right(30) then Left(40) | 30 |  ✅ Final AVL Tree Structure (via Pre-Order Traversal) The pre-order traversal (root, left, right) shows the structure:  30 20 10 25 40 50 📜 In-Order Traversal This gives sorted order of elements (left-root-right):  10 20 25 30 40 50 🔁 Explanation of RotationsCase 1: 10 → 20 → 30  * Inserting 30 caused right-right imbalance at 10. * Single left rotation at 10.  Case 2: 20 → 30 → 40 → 50  * Inserting 50 caused right-right imbalance at 20. * Single left rotation at 20.  Case 3: 50 → 25  * Inserting 25 caused right-left imbalance at 40. * First, right rotation at 30 (child), then left rotation at 40. |
| In-order traversal: 10 20 25 30 40 50  Pre-order traversal: 30 20 10 25 40 50 | |