CS-250

Data Structures and Algorithms

ASSIGNMENT 03

AVL TREE

Name: Mahum Samar

Class: BSCS-9B

CMS ID: 290647

# Code:

|  |
| --- |
| *//Mahum Samar //BSCS-9B //290647* #include <iostream> #include <string> #include <fstream> #include <sstream>  using namespace std;  *//both the classes implemented are of generic type* template<typename Type> class AvlNode {  *//class to create the nodes for binary search tree* public:  Type data;  AvlNode<Type> \*leftChild;  AvlNode<Type> \*rightChild; };  template<typename Type> class AvlTree {  *//class for implementing the binary search tree and its methods* public:  AvlNode<Type> \*root; *//root node for BTS  //loc and preloc used for searching in the BST* AvlNode<Type> \*loc;  AvlNode<Type> \*preloc;  int leafNodeCount = 0;  int onlyRightChildCount = 0;  int onlyLeftChildCount = 0;   bool IsEmpty() {  *//method to check if the tree is empty* return root == NULL;  }   void InsertWithDuplication(Type value) {  *//method to insert with duplication of value // Creating new node* AvlNode<Type> \*newNode = new AvlNode<Type>();  newNode->data = value;  if (!IsEmpty()) {  *// if the t is not empty* loc = root;  preloc = NULL;  while (loc != NULL) {  *// while the location of insertion is not found* if (value <= loc->data) {  *// if the value is less than loc value, pointer move to left child* preloc = loc;  loc = loc->leftChild;  } else {  *// if the value is greater than loc value, pointer move to Right child* preloc = loc;  loc = loc->rightChild;  }  }   *//* if (value <= preloc->data) {  *// if the value is less than loc value* preloc->leftChild = newNode;  } else {  *// if the value is greater than loc value* preloc->rightChild = newNode;  }  } else {  *// If tree is empty* root = newNode;  }  }   void Search(Type value) {  *// Method to search the value in the tree* loc = root;  preloc = NULL;  while (loc != NULL && loc->data != value) {  *// Until value is not found or reached the end of the tree* if (value < loc->data) {  preloc = loc;  loc = loc->leftChild;  } else {  preloc = loc;  loc = loc->rightChild;  }  }  }   void InsertWithoutDuplication(Type value) {  *//method to insert the new node if the value is not already existing in tree* Search(value);  *//if value is not found* if (loc == NULL) {  *//new node is created* AvlNode<Type> \*newNode = new AvlNode<Type>();  newNode->data = value;  if (preloc != NULL) {  *//if node is not root tree* if (value < preloc->data) {  *//if value is less* preloc->leftChild = newNode;  } else {  preloc->rightChild = newNode;  }  } else {  *//if tree is empty* root = newNode;  }  CheckBalancedAvlTree(root);  } else {  *//if value already existing* cout << "Duplicate insertion not allowed.\n";  }  }   void PreOrder(AvlNode<Type> \*treeNode) {  *// method used to print he values in prefix notation* if (treeNode != NULL) {  cout << treeNode->data << " ";  PreOrder(treeNode->leftChild);  PreOrder(treeNode->rightChild);  }  }   void InOrder(AvlNode<Type> \*treeNode) {  *//method used to print the values in incresing order* if (treeNode != NULL) {  InOrder(treeNode->leftChild);  cout << treeNode->data << " ";  InOrder(treeNode->rightChild);  }  }   void PostOrder(AvlNode<Type> \*treeNode) {  *//method used to print the values in postfix order* if (treeNode != NULL) {  PostOrder(treeNode->leftChild);  PostOrder(treeNode->rightChild);  cout << treeNode->data << " ";  }  }   int height(AvlNode<Type> \*treeNode) {  cout << "height function called.\n";  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";   *//method used to find the height of the tree* if (treeNode == NULL) {  *//base case* return -1;  } else {  *//recursive case  //until the leaf node is reached // its left and right child are transversed to find the height* cout << "Node for which function called have value:\t" << treeNode->data << endl;  return (1 + max(height(treeNode->leftChild), height(treeNode->rightChild)));  }  }   int depth(AvlNode<Type> \*treeNode, int depthOfParentTree) {  *//method to find the depth of the tree* if (treeNode != NULL) {  *//if tree is not empty* int d = 1 + depthOfParentTree; *//while tranversing the depth is incremented by 1* cout << "Node : " << treeNode->data << " at depth : " << d << endl; *//depth of each node printed* int depthLeftTree = depth(treeNode->leftChild, 1 + depthOfParentTree);  int depthRightTree = depth(treeNode->rightChild, 1 + depthOfParentTree);  return max(depthLeftTree, depthRightTree); *//max depth retured* } else {  return depthOfParentTree;  }  }   void Delete(Type value) { *// method to delete the value from the subtree* Search(value); *//search the value in the BST* if (loc == NULL) {  cout << "Value is not found.\n";  } else {  *//if value is found   //case 1:  //Deleting the leaf node* if (loc->leftChild == NULL && loc->rightChild == NULL) {  if (preloc == NULL) {  *//if only one node in tree* root = NULL;  } else if (preloc->leftChild == loc) {  *//if searched node is left child* preloc->leftChild = NULL;  } else {  *//if searched node is right child* preloc->rightChild = NULL;  }  *//deletes the searched node* delete loc;  CheckBalancedAvlTree(root);  } else  *//case 2  //node with only one subtree  //case 2a node with only left child* if (loc->leftChild != NULL && loc->rightChild == NULL) {  if (preloc == NULL) {  *//if the root node* root = loc->leftChild;  } else if (loc == preloc->leftChild) {  *//if searched node is the left child of parent* preloc->leftChild = loc->leftChild;  } else {  *//if searched node is right child of parent* preloc->rightChild = loc->leftChild;  }  delete loc;  CheckBalancedAvlTree(root);  } else  *//case 2b  //node with only right subtree* if (loc->leftChild == NULL && loc->rightChild != NULL) {  if (preloc == NULL) {  *//if root node* root = loc->rightChild;  } else if (loc == preloc->leftChild) {  *//if left child of parent* preloc->leftChild = loc->rightChild;  } else {  *//if right child of parent* preloc->rightChild = loc->rightChild;  }  delete loc;  CheckBalancedAvlTree(root);  } else  *//case 3  //deleting the node having two children* if (loc->leftChild != NULL && loc->rightChild != NULL) {  *//finding the logical successor of loc i.e. largest value* AvlNode<Type> \*preloc1 = loc;  AvlNode<Type> \*loc1 = loc->leftChild;    while (loc1->rightChild != NULL) {  *//to find right most node* preloc1 = loc1;  loc1 = loc1->rightChild;  }   *//code to unlink it from the tree* if (loc->leftChild == loc1) {  loc->leftChild = loc1->leftChild;  } else if (loc1->leftChild != NULL) {  preloc1->rightChild = loc1->leftChild;  } else {  preloc1->rightChild = NULL;  }   *//updating the connections with respect to the loc1 i.e. largest node in the subtree of the node to be deleted.* loc1->leftChild = loc->leftChild;  loc1->rightChild = loc->rightChild;   if (preloc == NULL) {  root = loc1;  } else if (preloc->rightChild == loc) {  preloc->rightChild = loc1;  } else {  preloc->leftChild = loc1;  }  delete loc;  CheckBalancedAvlTree(root);  }   }  }   void DestroyTreeAndRoot() {  *//method to destory the root* DestroyTree(this->root);  root = NULL;  }   void DestroyTree(AvlNode<Type> \*treeNode) {  *//method to destroy the tree recursively* if (treeNode != NULL) {  DestroyTree(treeNode->leftChild);  DestroyTree(treeNode->rightChild);  delete treeNode;  }  }   int BalanceFactor(AvlNode<Type> \*treeNode) {  *//method to find the balance factor of a given node.* cout << "Balance Factor Calculate function called.\n";  if (treeNode == NULL) {  *//if node is null* return 0;  }  *//recursive call  //balance factor function calls the height method recusively.  //returns the height of left subtree subtracted by the height of right subtree.* cout << "Node for which function called have value:\t" << treeNode->data << endl;  return height(treeNode->leftChild) - height(treeNode->rightChild);  }   AvlNode<Type> \*LeftRotate(AvlNode<Type> \*x) {  *//method to left rotate the subtree which is left left heavy* cout << "Left Rotate function called.\n";  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "Node for which function called have value:\t" << x->data << endl; *//for the tree like this // C -> x // / // B -> y // / // A* AvlNode<Type> \*y = x->rightChild;  x->rightChild = y->leftChild;  y->leftChild = x;  PostOrder(root);*//printing in post order* return y;  }   AvlNode<Type> \*RightRotate(AvlNode<Type> \*x) {  *//method to right rotate the subtree  //which is right right heavy // A -> x // \ // B -> y // \ // C* cout << "Right Rotate function called.\n";  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "Node for which function called have value:\t" << x->data << endl;  AvlNode<Type> \*y = x->leftChild;  x->leftChild = y->rightChild;  y->rightChild = x;  PostOrder(root);*//printing in post order* return y;  }   AvlNode<Type> \*RightLeftRotate(AvlNode<Type> \*x) {  *//method to right left rotate the subtree  //which is right left heavy // A -> x // \ // C -> y // / // B* cout << "Right Left Rotate function called.\n";  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "Node for which function called have value:\t" << x->data << endl;  AvlNode<Type> \*y = x->rightChild;  if (BalanceFactor(y) == 1) {  *//when right child of x is having balance factor of 1* y = RightRotate(y);  }  x->rightChild = y; *//left rotate* return LeftRotate(x);  }   AvlNode<Type> \*LeftRightRotate(AvlNode<Type> \*x) {  *//method to left right rotate subtree  //which is left right heavy // for the case like below  // C -> x // / // A -> y // \ // B* cout << "Left Right Rotate function called.\n";  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "Node for which function called have value:\t" << x->data << endl;  AvlNode<Type> \*y = x->leftChild;  if (BalanceFactor(y) == -1) { *// if left subtree have balance factor of -1* y = LeftRotate(y);  }  x->leftChild = y;   return RightRotate(x);  }   AvlNode<Type> \*CheckBalancedAvlTree(AvlNode<Type> \*treeNode) {  cout << "method for checking the balancin gof tree after each insertion or deletion.\n";  *//this is the function that checks the tree is balanced or not after each insertion and deletion  //this method uses bottom up approach* if (treeNode != NULL) {  *//goes to the leaf node recursively* CheckBalancedAvlTree(treeNode->leftChild);  CheckBalancedAvlTree(treeNode->rightChild);  }  *//after transversing tree in post order  //the balance factor is checked for each node* if (BalanceFactor(treeNode) == -2) {  *//if balance factor of node is -2  // left rotate or right-left rotate function needs to be called  //search function is called so that the preloc and loc can be get  //to get the reference of the parent of the node which is balanced  //so that its address can be updated in its parent after rotation function returns.* Search(treeNode->data);  if (BalanceFactor(treeNode->rightChild) == 1 && BalanceFactor(treeNode->rightChild->rightChild) == 0) {  *//if left rotate case* if (preloc != NULL) {  *//if the unbalanced node is not the root node* if (preloc->rightChild == treeNode) { *//if right child of parent* preloc->rightChild = LeftRotate(treeNode);  } else {  *//if left child of parent* preloc->leftChild = LeftRotate(treeNode);  }  } else {  *//if the unbalanced node is root node* root = LeftRotate(treeNode);  }  } else {  *//if the double rotation case occurs  //unbalanced node have right-left heavy subtree* if (preloc != NULL) {  if (preloc->rightChild == treeNode) {   preloc->rightChild = RightLeftRotate(treeNode);  } else {  preloc->leftChild = RightLeftRotate(treeNode);  }   } else {   root = RightLeftRotate(treeNode);  }  }  }   if (BalanceFactor(treeNode) == 2) {  *//if balance factor of node is -2* Search(treeNode->data);  if (BalanceFactor(treeNode->leftChild) == 1 && BalanceFactor(treeNode->leftChild->leftChild) == 0) {  *//if right rotate  //left left heavy subtree* if (preloc != NULL) {  *//if unbalanced node is not root* if (preloc->rightChild == treeNode) {   preloc->rightChild = RightRotate(treeNode);  } else {  preloc->leftChild = RightRotate(treeNode);  }  } else {  *//if unbalanced node is root* root = RightRotate(treeNode);  }  } else {  *//if unbalanced node needs double rotation* if (preloc != NULL) {  *//if not root* if (preloc->rightChild == treeNode) {   preloc->rightChild = LeftRightRotate(treeNode);  } else {  preloc->leftChild = LeftRightRotate(treeNode);  }   } else { *//if root* root = LeftRightRotate(treeNode);  }  }  }  }  };     int main() { AvlTree<int> \*avlTree = new AvlTree<int>(); *// int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 1,13, 45, 99, 104, 8};//for all types of rotation // int inputArray[] = {3,2,1}; //for right rotate* int inputArray[] = {1, 2, 3}; *//for left rotate* int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  avlTree->InsertWithoutDuplication(inputArray[i]);  }   cout << "Values of the tree are:\n";  avlTree->PreOrder(avlTree->root);  cout << "\n\nValues after deleting 3 from the tree are:\n";  avlTree->Delete(3);  cout << "\nValues of the tree are:\n";  avlTree->PreOrder(avlTree->root);  return 0; } |

# Code:

|  |
| --- |
| *//Mahum Samar //BSCS-9B //290647* #include <iostream> #include <string> #include <fstream> #include <sstream>  using namespace std;  class StudentData;  template<typename Type> class AvlTree;   class StudentData { public:  unsigned short int studentID;  string name;   void InsertStudent(unsigned short int studentId, string name) {  this->name = name;  this->studentID = studentId;  } };  *//both the classes implemented are of generic type* template<typename Type> class AvlNode {  *//class to create the nodes for binary search tree* public:  StudentData student;  AvlNode<Type> \*leftChild;  AvlNode<Type> \*rightChild; };  template<typename Type> class AvlTree {  *//class for implementing the binary search tree and its methods* public:  AvlNode<Type> \*root; *//root node for BTS  //loc and preloc used for searching in the BST* AvlNode<Type> \*loc;  AvlNode<Type> \*preloc;  int leafNodeCount = 0;  int onlyRightChildCount = 0;  int onlyLeftChildCount = 0;   bool IsEmpty() {  *//method to check if the tree is empty* return root == NULL;  }   void Search(int value) {  *// Method to search the value in the tree* loc = root;  preloc = NULL;  while (loc != NULL && loc->student.studentID != value) {  *// Until value is not found or reached the end of the tree* if (value < loc->student.studentID) {  preloc = loc;  loc = loc->leftChild;  } else {  preloc = loc;  loc = loc->rightChild;  }  }  }   void InsertWithoutDuplication(StudentData \*value) {  *//method to insert the new node if the value is not already existing in tree* Search(value->studentID);  *//if value is not found* if (loc == NULL) {  *//new node is created* AvlNode<Type> \*newNode = new AvlNode<Type>();  newNode->student = \*value;  if (preloc != NULL) {  *//if node is not root tree* if (value->studentID < preloc->student.studentID) {  *//if value is less* preloc->leftChild = newNode;  } else {  preloc->rightChild = newNode;  }  } else {  *//if tree is empty* root = newNode;  }  CheckBalancedAvlTree(root);  } else {  *//if value already existing* cout << "Duplicate insertion not allowed.\n";  }  }   void PreOrder(AvlNode<Type> \*treeNode) {  *// method used to print he values in prefix notation* if (treeNode != NULL) {  cout << treeNode->student.studentID << "\t";  cout << treeNode->student.name << "\n";  PreOrder(treeNode->leftChild);  PreOrder(treeNode->rightChild);  }  }   void InOrder(AvlNode<Type> \*treeNode) {  *//method used to print the values in incresing order* if (treeNode != NULL) {  InOrder(treeNode->leftChild);  cout << treeNode->student.studentID << "\t";  cout << treeNode->student.name << "\n";  InOrder(treeNode->rightChild);  }  }   void PostOrder(AvlNode<Type> \*treeNode) {  *//method used to print the values in postfix order* if (treeNode != NULL) {  PostOrder(treeNode->leftChild);  PostOrder(treeNode->rightChild);  cout << treeNode->student.studentID << "\t";  cout << treeNode->student.name << "\n";  }  }   int height(AvlNode<Type> \*treeNode) { *// cout << "height function called.\n"; // cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";   //method used to find the height of the tree* if (treeNode == NULL) {  *//base case* return -1;  } else {  *//recursive case  //until the leaf node is reached // its left and right child are transversed to find the height // cout << "Node for which function called have value:\t" << treeNode->student.studentID << endl;* return (1 + max(height(treeNode->leftChild), height(treeNode->rightChild)));  }  }   int depth(AvlNode<Type> \*treeNode, int depthOfParentTree) {  *//method to find the depth of the tree* if (treeNode != NULL) {  *//if tree is not empty* int d = 1 + depthOfParentTree; *//while tranversing the depth is incremented by 1 // cout << "Node : " << treeNode->student << " at depth : " << d << endl; //depth of each node printed* int depthLeftTree = depth(treeNode->leftChild, 1 + depthOfParentTree);  int depthRightTree = depth(treeNode->rightChild, 1 + depthOfParentTree);  return max(depthLeftTree, depthRightTree); *//max depth retured* } else {  return depthOfParentTree;  }  }   void Delete(Type value) { *// method to delete the value from the subtree* Search(value->studentID); *//search the value in the BST* if (loc == NULL) {  cout << "Value is not found.\n";  } else {  *//if value is found   //case 1:  //Deleting the leaf node* if (loc->leftChild == NULL && loc->rightChild == NULL) {  if (preloc == NULL) {  *//if only one node in tree* root = NULL;  } else if (preloc->leftChild == loc) {  *//if searched node is left child* preloc->leftChild = NULL;  } else {  *//if searched node is right child* preloc->rightChild = NULL;  }  *//deletes the searched node* delete loc;  CheckBalancedAvlTree(root);  } else  *//case 2  //node with only one subtree  //case 2a node with only left child* if (loc->leftChild != NULL && loc->rightChild == NULL) {  if (preloc == NULL) {  *//if the root node* root = loc->leftChild;  } else if (loc == preloc->leftChild) {  *//if searched node is the left child of parent* preloc->leftChild = loc->leftChild;  } else {  *//if searched node is right child of parent* preloc->rightChild = loc->leftChild;  }  delete loc;  CheckBalancedAvlTree(root);  } else  *//case 2b  //node with only right subtree* if (loc->leftChild == NULL && loc->rightChild != NULL) {  if (preloc == NULL) {  *//if root node* root = loc->rightChild;  } else if (loc == preloc->leftChild) {  *//if left child of parent* preloc->leftChild = loc->rightChild;  } else {  *//if right child of parent* preloc->rightChild = loc->rightChild;  }  delete loc;  CheckBalancedAvlTree(root);  } else  *//case 3  //deleting the node having two children* if (loc->leftChild != NULL && loc->rightChild != NULL) {  *//finding the logical successor of loc i.e. largest value* AvlNode<Type> \*preloc1 = loc;  AvlNode<Type> \*loc1 = loc->leftChild;    while (loc1->rightChild != NULL) {  *//to find right most node* preloc1 = loc1;  loc1 = loc1->rightChild;  }   *//code to unlink it from the tree* if (loc->leftChild == loc1) {  loc->leftChild = loc1->leftChild;  } else if (loc1->leftChild != NULL) {  preloc1->rightChild = loc1->leftChild;  } else {  preloc1->rightChild = NULL;  }   *//updating the connections with respect to the loc1 i.e. largest node in the subtree of the node to be deleted.* loc1->leftChild = loc->leftChild;  loc1->rightChild = loc->rightChild;   if (preloc == NULL) {  root = loc1;  } else if (preloc->rightChild == loc) {  preloc->rightChild = loc1;  } else {  preloc->leftChild = loc1;  }  delete loc;  CheckBalancedAvlTree(root);  }   }  }   void DestroyTreeAndRoot() {  *//method to destory the root* DestroyTree(this->root);  root = NULL;  }   void DestroyTree(AvlNode<Type> \*treeNode) {  *//method to destroy the tree recursively* if (treeNode != NULL) {  DestroyTree(treeNode->leftChild);  DestroyTree(treeNode->rightChild);  delete treeNode;  }  }   int BalanceFactor(AvlNode<Type> \*treeNode) {  *//method to find the balance factor of a given node. // cout << "Balance Factor Calculate function called.\n";* if (treeNode == NULL) {  *//if node is null* return 0;  }  *//recursive call  //balance factor function calls the height method recusively.  //returns the height of left subtree subtracted by the height of right subtree. // cout << "Node for which function called have value:\t" << treeNode->student.studentID << endl;* return height(treeNode->leftChild) - height(treeNode->rightChild);  }   AvlNode<Type> \*LeftRotate(AvlNode<Type> \*x) {  *//method to left rotate the subtree which is left left heavy // cout << "Left Rotate function called.\n"; // cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"; // cout << "Node for which function called have value:\t" << x->student.studentID << endl; //for the tree like this // C -> x // / // B -> y // / // A* AvlNode<Type> \*y = x->rightChild;  x->rightChild = y->leftChild;  y->leftChild = x; *// PostOrder(root);//printing in post order* return y;  }   AvlNode<Type> \*RightRotate(AvlNode<Type> \*x) {  *//method to right rotate the subtree  //which is right right heavy // A -> x // \ // B -> y // \ // C  // cout << "Right Rotate function called.\n"; // cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"; // cout << "Node for which function called have value:\t" << x->student.studentID << endl;* AvlNode<Type> \*y = x->leftChild;  x->leftChild = y->rightChild;  y->rightChild = x; *// PostOrder(root);//printing in post order* return y;  }   AvlNode<Type> \*RightLeftRotate(AvlNode<Type> \*x) {  *//method to right left rotate the subtree  //which is right left heavy // A -> x // \ // C -> y // / // B  // cout << "Right Left Rotate function called.\n"; // cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"; // cout << "Node for which function called have value:\t" << x->student.studentID << endl;* AvlNode<Type> \*y = x->rightChild;  if (BalanceFactor(y) == 1) {  *//when right child of x is having balance factor of 1* y = RightRotate(y);  }  x->rightChild = y; *//left rotate* return LeftRotate(x);  }   AvlNode<Type> \*LeftRightRotate(AvlNode<Type> \*x) {  *//method to left right rotate subtree  //which is left right heavy // for the case like below  // C -> x // / // A -> y // \ // B   // cout << "Left Right Rotate function called.\n"; // cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"; // cout << "Node for which function called have value:\t" << x->student.studentID << endl;* AvlNode<Type> \*y = x->leftChild;  if (BalanceFactor(y) == -1) { *// if left subtree have balance factor of -1* y = LeftRotate(y);  }  x->leftChild = y;   return RightRotate(x);  }   AvlNode<Type> \*CheckBalancedAvlTree(AvlNode<Type> \*treeNode) { *// cout << "method for checking the balancin gof tree after each insertion or deletion.\n";  //this is the function that checks the tree is balanced or not after each insertion and deletion  //this method uses bottom up approach* if (treeNode != NULL) {  *//goes to the leaf node recursively* CheckBalancedAvlTree(treeNode->leftChild);  CheckBalancedAvlTree(treeNode->rightChild);  }  *//after transversing tree in post order  //the balance factor is checked for each node* if (BalanceFactor(treeNode) == -2) {  *//if balance factor of node is -2 // left rotate or right-left rotate function needs to be called //search function is called so that the preloc and loc can be get //to get the reference of the parent of the node which is balanced //so that its address can be updated in its parent after rotation function returns.* Search(treeNode->student.studentID);  if (BalanceFactor(treeNode->rightChild) == 1 && BalanceFactor(treeNode->rightChild->rightChild) == 0) {  *//if left rotate case* if (preloc != NULL) {  *//if the unbalanced node is not the root node* if (preloc->rightChild == treeNode) { *//if right child of parent* preloc->rightChild = LeftRotate(treeNode);  } else {  *//if left child of parent* preloc->leftChild = LeftRotate(treeNode);  }  } else {  *//if the unbalanced node is root node* root = LeftRotate(treeNode);  }  } else {  *//if the double rotation case occurs  //unbalanced node have right-left heavy subtree* if (preloc != NULL) {  if (preloc->rightChild == treeNode) {   preloc->rightChild = RightLeftRotate(treeNode);  } else {  preloc->leftChild = RightLeftRotate(treeNode);  }   } else {   root = RightLeftRotate(treeNode);  }  }  }   if (BalanceFactor(treeNode) == 2) {  *//if balance factor of node is -2* Search(treeNode->student.studentID);  if (BalanceFactor(treeNode->leftChild) == 1 && BalanceFactor(treeNode->leftChild->leftChild) == 0) {  *//if right rotate  //left left heavy subtree* if (preloc != NULL) {  *//if unbalanced node is not root* if (preloc->rightChild == treeNode) {   preloc->rightChild = RightRotate(treeNode);  } else {  preloc->leftChild = RightRotate(treeNode);  }  } else {  *//if unbalanced node is root* root = RightRotate(treeNode);  }  } else {  *//if unbalanced node needs double rotation* if (preloc != NULL) {  *//if not root* if (preloc->rightChild == treeNode) {   preloc->rightChild = LeftRightRotate(treeNode);  } else {  preloc->leftChild = LeftRightRotate(treeNode);  }   } else { *//if root* root = LeftRightRotate(treeNode);  }  }  }  }  };  class Controller { public:  string line;  string array[2];   void FileRead(AvlTree<StudentData> \*avlTree) { *// cout<<"File read method\n\n";   //method to read file* ifstream file;  file.open("studentData.csv");  *// cout<<"File opened\n";* getline(file, line);  stringstream ss(line);  for (int i = 0; i <= 1; i++) {  *//loop to read first row containing column name* getline(ss, array[i], ',');  } *// cout << endl;* while (getline(file, line)) {  *//loop to read each row and split it into the fields in which data is to be stored* StudentData \*studentData = new StudentData();   stringstream ss(line);   *//to split read row into 2 columns* for (int i = 0; i <= 1; i++) {  getline(ss, array[i], ',');  }   try {   studentData->studentID=(unsigned short int)stoi(array[0]);  studentData->name=array[1]; *// cout<<"array[0]"<<array[0]; // cout<<"array[1]"<<array[1];* avlTree->InsertWithoutDuplication(studentData);  } catch (exception) {   }  }   } };      int main() { *//to store the student name and roll number in avl tree //* Controller \*controller = new Controller();   AvlTree<StudentData> \*avlTree = new AvlTree<StudentData>();  *// cout<<"file read function called\n";* controller->FileRead(avlTree);  cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";   cout<<"Transversing AVL Tree in post order:\n";  avlTree->PostOrder(avlTree->root);   cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout<<"Transversing AVL Tree in pre order:\n";  avlTree->PreOrder(avlTree->root);   cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout<<"Transversing AVL Tree in In order:\n";  avlTree->InOrder(avlTree->root);   cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";   return 0; } |