CS-250

Data Structures and Algorithms

Lab 12

Binary Search Tree

Name: Mahum Samar

Class: BSCS-9B

CMS ID: 290647

# Code:

The main code for the classes is given below.

And the main functions for testing of each method is given separately.

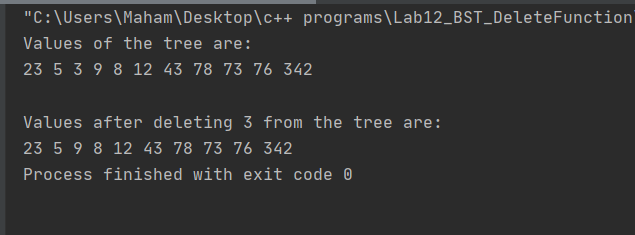
|  |
| --- |
| #include <iostream>  using namespace std;  *//both the classes implemented are of generic type* template<typename Type> class BTNode {  *//class to create the nodes for binary search tree* public:  Type data;  BTNode<Type> \*leftChild;  BTNode<Type> \*rightChild; };  template<typename Type> class BinarySearchTree {  *//class for implementing the binary search tree and its methods* public:  BTNode<Type> \*root; *//root node for BTS  //loc and preloc used for searching in the BST* BTNode<Type> \*loc;  BTNode<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* BTNode<Type> \*newNode = new BTNode<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* BTNode<Type> \*newNode = new BTNode<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;  }  } else {  *//if value already existing* cout << "Duplicate insertion not allowed.\n";  }  }   void PreOrder(BTNode<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(BTNode<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(BTNode<Type> \*treeNode) {  *//method used to print the values in postfix order* if (treeNode != NULL) {  PostOrder(treeNode->leftChild);  PostOrder(treeNode->rightChild);  cout << treeNode->data << " ";  }  }   void MinimumValue(BTNode<Type> \*root) {  *//method used to find the smallest value in the tree* if (!IsEmpty()) {  *//if tree is not empty* BTNode<Type> \*current = root;  while (current->leftChild != NULL) {  *//as its binary search tree so left childs of the nodes are tested* current = current->leftChild;  }  cout << "Minimum Value : " << current->data << endl;  } else {  *//if tree is empty* cout << "The tree is empty.\n";  }  }   void MaximumValue(BTNode<Type> \*root) {  *//method used to find the max value in the tree* if (!IsEmpty()) {  *//if tree is not empty* BTNode<Type> \*current = root;  while (current->rightChild != NULL) {  *//as it is BST so right child of the nodes are tested for max value* current = current->rightChild;  }  cout << "Maximum Value : " << current->data << endl;  } else {  *//if tree is empty* cout << "The tree is empty.\n";  }   }   int height(BTNode<Type> \*treeNode) {  *//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* return (1 + max(height(treeNode->leftChild), height(treeNode->rightChild)));  }  }   int depth(BTNode<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;  } 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;  } 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;  } 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* BTNode<Type> \*preloc1 = loc;  BTNode<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;  }   }  }   void DestroyTreeAndRoot() {  *//method to destory the root* DestroyTree(this->root);  root = NULL;  }   void DestroyTree(BTNode<Type> \*treeNode) {  *//method to destroy the tree recursively* if (treeNode != NULL) {  DestroyTree(treeNode->leftChild);  DestroyTree(treeNode->rightChild);  delete treeNode;  }  }   void PrintLeafNode(BTNode<Type> \*treeNode) {  *//task 03  //method to print the leaf nodes of the tree* if (treeNode == NULL) {  *//base case* return;  }  if (treeNode->leftChild == NULL && treeNode->rightChild == NULL) {  cout << treeNode->data << " ";  *//base case* return;  }   if (treeNode->leftChild != NULL) {  *//recursive call for the left child* PrintLeafNode(treeNode->leftChild);  }   if (treeNode->rightChild != NULL) { *// recursive call for right subtree* PrintLeafNode(treeNode->rightChild);  }  }   BTNode<Type> \*DeleteLeafNode(BTNode<Type> \*treeNode) {  *//Task 05  //method to delete the leaf nodes from tree* if (treeNode == NULL) { *// base case* return NULL;  }  if (treeNode->leftChild == NULL && treeNode->rightChild == NULL) {  *//base case  //deleting the node* delete treeNode;  return NULL;  }   *//recursive calls* treeNode->leftChild = DeleteLeafNode(treeNode->leftChild);  treeNode->rightChild = DeleteLeafNode(treeNode->rightChild);   return treeNode;  }   void CountNodes(BTNode<Type> \*treeNode) {  *//task 4  //method to count the leaf nodes, nodes with only left child, nodes with only right child* if (treeNode != NULL) {  if (treeNode->leftChild == NULL && treeNode->rightChild == NULL) {  *//leaf node* leafNodeCount++;  } else if (treeNode->leftChild != NULL && treeNode->rightChild == NULL) {  *//node with only left child* onlyLeftChildCount++;  } else if (treeNode->leftChild == NULL && treeNode->rightChild != NULL) {  *//node with only right subtree* onlyRightChildCount++;  }  *//recursive calls* CountNodes(treeNode->leftChild);  CountNodes(treeNode->rightChild);  }   }   BTNode<Type> \*DeleteLeftSubtree(BTNode<Type> \*treeNode) {  *//task 06  //method to delete the node having only the left sub tree* if (treeNode != NULL) {  *//preorder transversal* DeleteLeftSubtree(treeNode->leftChild);  DeleteLeftSubtree(treeNode->rightChild);  *//deleting the node if node is the only left subtree* if (treeNode->leftChild != NULL && treeNode->rightChild == NULL) {  Delete(treeNode->data);  }  } }  }; |

# Task 01(a)

## Code:

|  |
| --- |
| int main() { *//deleting a leaf node* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }   cout << "Values of the tree are:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  cout << "\n\nValues after deleting 3 from the tree are:\n";  binarySearchTree->Delete(3);  binarySearchTree->PreOrder(binarySearchTree->root);  return 0; } |

## Output:

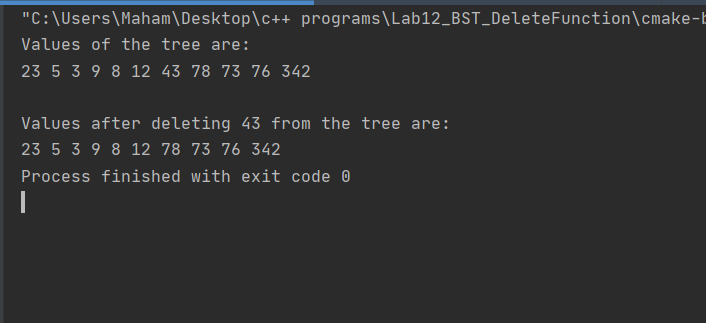


# Task 01(b)

## Code:

|  |
| --- |
| int main() { *//deleting node with only one subtree* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Values of the tree are:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  cout << "\n\nValues after deleting 43 from the tree are:\n";  binarySearchTree->Delete(43);  binarySearchTree->PreOrder(binarySearchTree->root);  return 0; } |

## Output:

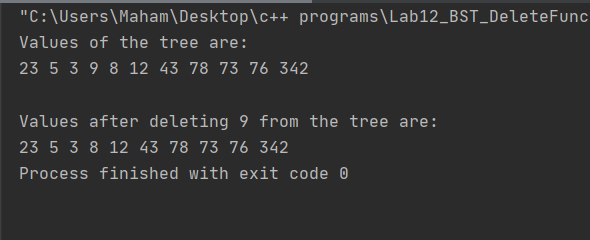


# Task 01(c)

## Code:

|  |
| --- |
| int main() { *//deleting node with two children* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Values of the tree are:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  cout << "\n\nValues after deleting 9 from the tree are:\n";  binarySearchTree->Delete(9);  binarySearchTree->PreOrder(binarySearchTree->root);  return 0; } |

## Output:

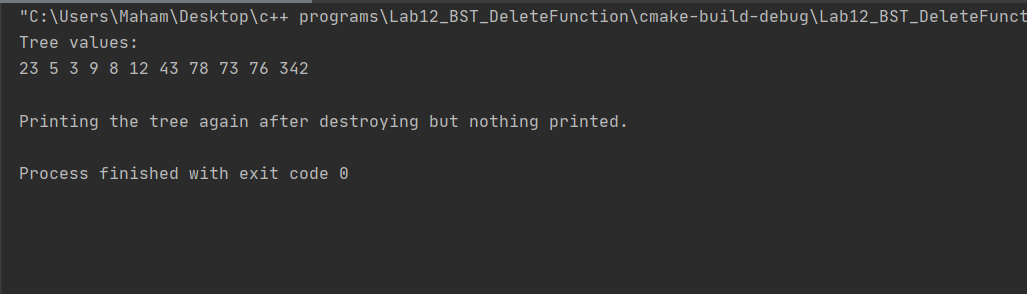


# Task 02

## Code:

|  |
| --- |
| int main() { *//destroy the tree* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Tree values:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  binarySearchTree -> DestroyTreeAndRoot();  cout << "\n\nPrinting the tree again after destroying but nothing printed.\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  return 0;  } |

## Output:

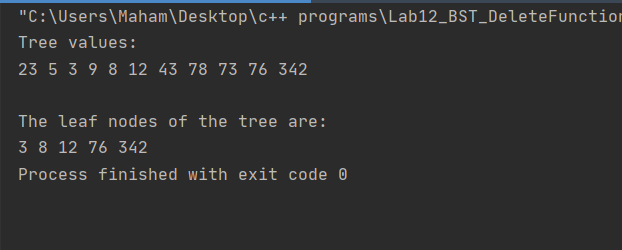


# Task 03

## Code:

|  |
| --- |
| int main() { *//printing the leaf nodes* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Tree values:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  cout << "\n\nThe leaf nodes of the tree are:\n";  binarySearchTree -> PrintLeafNode(binarySearchTree -> root);  return 0; } |

## Output:

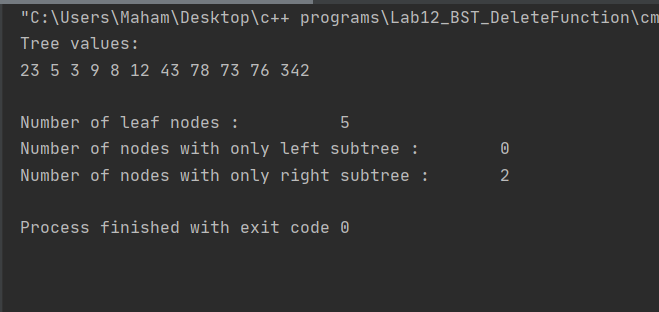


# Task 04

## Code:

|  |
| --- |
| int main() { *//counting the nodes* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Tree values:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  binarySearchTree -> CountNodes(binarySearchTree -> root);  cout << "\n\nNumber of leaf nodes :\t\t" << binarySearchTree -> leafNodeCount << endl;  cout << "Number of nodes with only left subtree :\t" << binarySearchTree -> onlyLeftChildCount << endl;  cout << "Number of nodes with only right subtree :\t" << binarySearchTree -> onlyRightChildCount << endl;   return 0; } |

## Output:

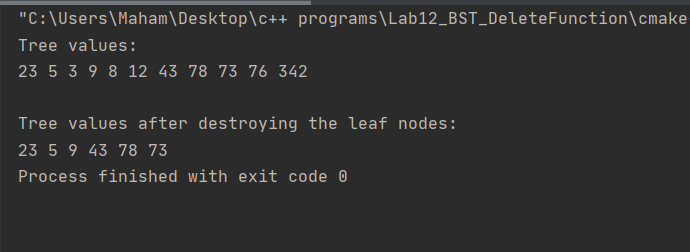


# Task 05

## Code:

|  |
| --- |
| int main() { *//deleting leaf nodes* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23, 43, 5, 78, 9, 342, 3, 73, 76, 12, 8};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Tree values:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  binarySearchTree -> DeleteLeafNode(binarySearchTree -> root);  cout << "\n\nTree values after destroying the leaf nodes:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  return 0; } |

## Output:



# Task 06

## Code:

|  |
| --- |
| int main() { *//delete nodes with only left subtree* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {50, 25, 75, 15, 18, 30, 80, 77};  int length = sizeof(inputArray) / sizeof(inputArray[0]);  for (int i = 0; i < length; i++) {  binarySearchTree->InsertWithoutDuplication(inputArray[i]);  }  cout << "Values in the tree are:\n";  binarySearchTree -> PreOrder(binarySearchTree -> root);  binarySearchTree->DeleteLeftSubtree(binarySearchTree->root);  cout << "\n\nValues in the tree after deleting the node having only left subtree are:\n";  binarySearchTree->PreOrder(binarySearchTree->root);  return 0; } |

## Output:

