CS-250

Data Structures and Algorithms

Lab 11

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.

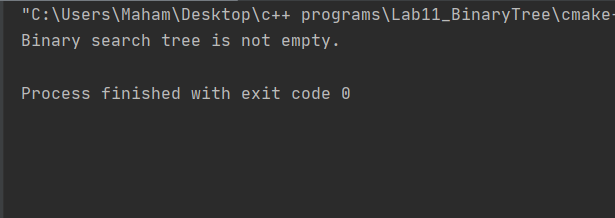
|  |
| --- |
| #include <iostream> using namespace std; *//both the classes implemented are of generic type* template <typename Type> class BTNode {  *//class to create the nodes for binary search tree* public:  Type data;  BTNode<Type> \*leftChild;  BTNode<Type> \*rightChild; };  template <typename 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;   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;  }  }   }; |

1. bool IsEmpty()

## Code:

|  |
| --- |
| int main() { *//search method* 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]);  }    if(binarySearchTree -> IsEmpty())  {  cout << "Binary search tree is empty.\n";  }  else  {  cout << "Binary search tree is not empty.\n";  }   return 0; } |

## Output:

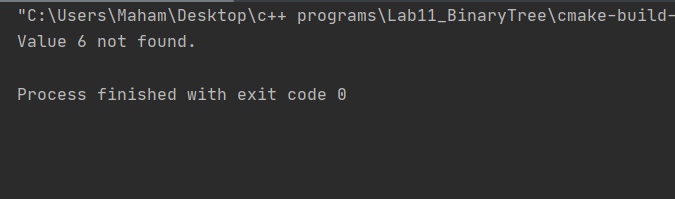


1. void Search(template value)

## Code:

|  |
| --- |
| int main() { *//search method* 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]);  }   binarySearchTree -> Search(6);  if(binarySearchTree -> loc != **NULL**)  {  cout << "Value 6 found.\n";  }  else  {  cout << "Value 6 not found.\n";  }   return 0; } |

## Output:

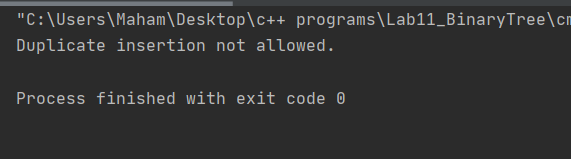


1. Void InsertWithoutDuplication(template value)

## Code:

|  |
| --- |
| int main() { *//insertion without duplication* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23,43,5,23,78,9,342,3,73,76,12,8};  int length = sizeof(inputArray)/sizeof(inputArray[0]);  for(int i = 0 ; i < length ; i++)  {  binarySearchTree -> InsertWithoutDuplication(inputArray[i]);  }   return 0; } |

## Output:

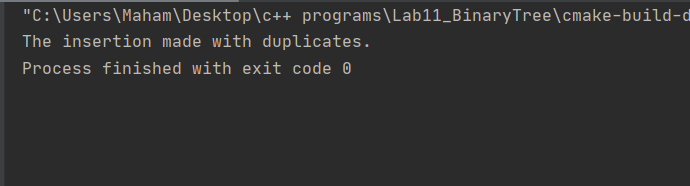


1. Void InsertWithDuplication(template value)

## Code:

|  |
| --- |
| int main() { *//insertion with duplication* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23,43,5,23,78,9,342,3,73,76,12,8};  int length = sizeof(inputArray)/sizeof(inputArray[0]);  for(int i = 0 ; i < length ; i++)  {  binarySearchTree -> InsertWithDuplication(inputArray[i]);  }  cout << “The insertion made with duplicates.”;  return 0; } |

## Output:



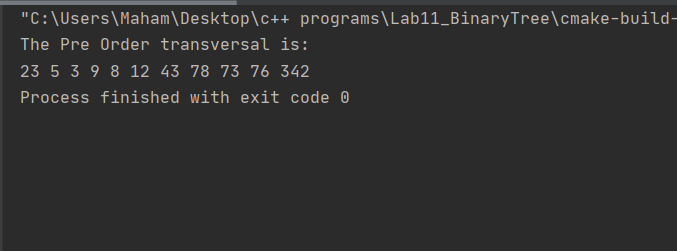
1. Implement the following tree traversal methods

PreOrder traversal

## Code:

|  |
| --- |
| int main() { *//pre order transversal* 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 << "The Pre Order transversal is:\n";   binarySearchTree -> PreOrder(binarySearchTree -> root);   return 0; } |

## Output:

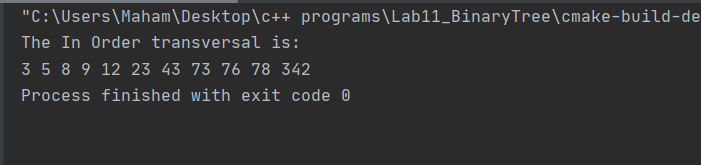


1. InOrder traversal

## Code:

|  |
| --- |
| int main() {  *//in order transversal* 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 << "The In Order transversal is:\n";   binarySearchTree -> InOrder(binarySearchTree -> root);   return 0; } |

## Output:

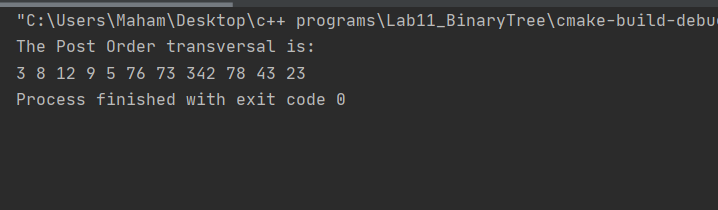


1. PostOrder traversal

## Code:

|  |
| --- |
| int main() {  *//post order transversal* 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 << "The Post Order transversal is:\n";   binarySearchTree -> PostOrder(binarySearchTree -> root);   return 0; } |

## Output:

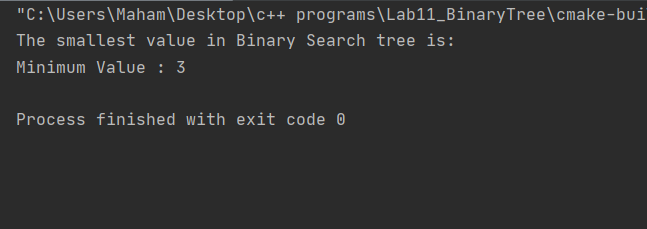


1. Implement a function that prints the smallest value of a BST.

## Code:

|  |
| --- |
| int main() {  *//minimum value* 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 << "The smallest value in Binary Search tree is:\n";   binarySearchTree -> MinimumValue(binarySearchTree -> root);   return 0; } |

## Output:

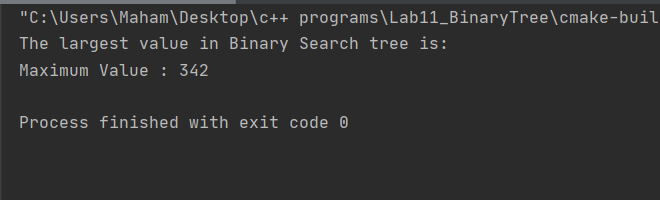


1. Implement a function that prints the largest value of a BST.

## Code:

|  |
| --- |
| int main() {  *//maximum value* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23,43,5,23,78,9,342,3,73,76,12,8};  int length = sizeof(inputArray)/sizeof(inputArray[0]);  for(int i = 0 ; i < length ; i++)  {  binarySearchTree -> InsertWithoutDuplication(inputArray[i]);  }  cout << "The largest value in Binary Search tree is:\n";   binarySearchTree -> MaximumValue(binarySearchTree -> root);   return 0; } |

## Output:

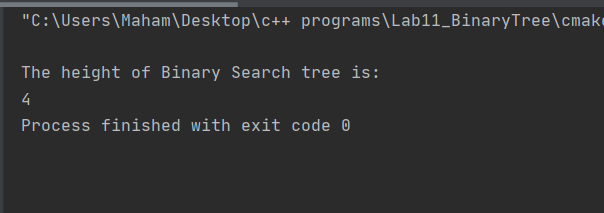


1. Implement a function to calculate the height of a BST.

## Code:

|  |
| --- |
| int main() {  *//height of binary search 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 << "\nThe height of Binary Search tree is:\n";   cout << binarySearchTree -> height(binarySearchTree -> root);   return 0; } |

## Output:



1. Implement a function that calculates the depth of a BST.

## Code:

|  |
| --- |
| int main() {  *//height of binary search tree* BinarySearchTree<int> \*binarySearchTree = new BinarySearchTree<int>();  int inputArray[] = {23,43,5,23,78,9,342,3,73,76,12,8};  int length = sizeof(inputArray)/sizeof(inputArray[0]);  for(int i = 0 ; i < length ; i++)  {  binarySearchTree -> InsertWithDuplication(inputArray[i]);  }   int depth = binarySearchTree -> depth(binarySearchTree -> root,-1);   cout << "\nThe depth of Binary Search tree is:\n" << depth << endl;   return 0; } |

## Output:

