

Applied Programming

Assignment#4



November 20, 2022

Muhammad talha Arif

22F-3769

# **Question#1**

AVL Tree ADT

## **Code**

#include "binarySearchTree.h"

using namespace std;

int main()

{

/\*

// Sample Input: 65 55 22 44 61 19 90 10 78

52

BST<int> myIntBST;

// test all functions here

\*/

BST<int> myIntBST;

int arr[] = { 65, 55, 22, 44, 61, 19, 90, 10, 78 };

int size = sizeof(arr) / sizeof(arr[0]);

for (int i = 0; i < size; i++)

{

cout << "BST<T>::Insert(" << arr[i] << ");" << endl;

myIntBST.Insert(arr[i]);

}

cout << endl;

cout << "BST<T>::SearchKey(65) = " << (myIntBST.SearchKey(65) ? "Found" : "Not Found") << endl;

cout << "BST<T>::SearchKey(100) = " << (myIntBST.SearchKey(100) ? "Found" : "Not Found") << endl;

cout << "BST<T>::SearchKey(10) = " << (myIntBST.SearchKey(10) ? "Found" : "Not Found") << endl;

cout << "BST<T>::DeleteKey(65)";

myIntBST.DeleteKey(65);

cout << endl;

cout << "BST<T>::DeleteKey(100)";

myIntBST.DeleteKey(100);

cout << endl;

cout << "BST<T>::DeleteKey(10)";

myIntBST.DeleteKey(10);

cout << endl;

cout << "BST<T>::findMax() = ";

cout << myIntBST.findMax() << endl;

cout << endl;

cout << "BST<T>::findMin() = ";

cout << myIntBST.findMin() << endl;

cout << endl;

cout << "BST<T>::inorderTraversal() {\n ";

myIntBST.inorderTraversal();

cout << "\n}\n" << endl;

cout << "BST<T>::preorderTraversal() {\n ";

myIntBST.preorderTraversal();

cout << "\n}\n" << endl;

cout << "BST<T>::postorderTraversal() {\n ";

myIntBST.preorderTraversal();

cout << "\n}\n" << endl;

cout << "BST<T>::treeHeight() = ";

cout << myIntBST.treeHeight() << endl;

cout << endl;

cout << "BST<T>::treeNodeCount() = ";

cout << myIntBST.treeNodeCount() << endl;

cout << endl;

cout << "BST<T>::treeLeavesCount() = ";

cout << myIntBST.treeLeavesCount() << endl;

cout << endl;

cout << "BST<T>::printNodeLevel() {\n ";

myIntBST.preorderTraversal();

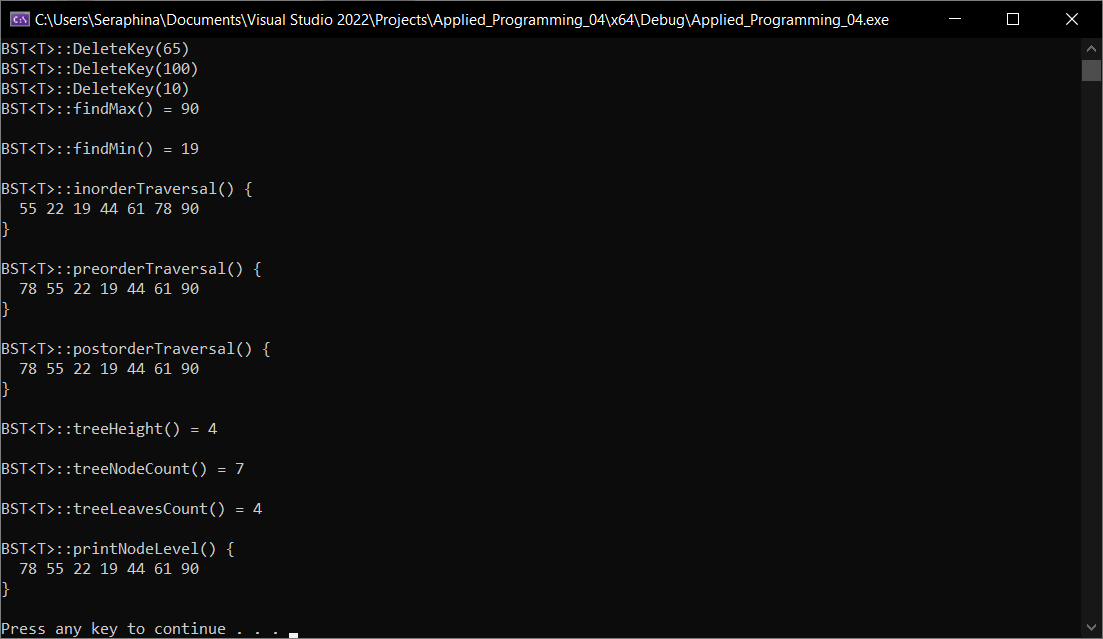
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



# **Question#2**

BST with Minimal Height

## **Code**

#include "binarySearchTree.h"

using namespace std;

int main()

{

/\*

// Sample Input: 65 55 22 44 61 19 90 10 78

52

BST<int> myIntBST;

// test all functions here

\*/

BST<int> myIntBST;

int array[] = { 1, 2, 3, 4, 5, 6, 7 };

int size = sizeof(array) / sizeof(array[0]);

cout << "Input: ";

for (auto i : array)

{

cout << i << ' ';

}

cout << endl;

cout << endl;

myIntBST.ConvertSortedArrayToTree(array, size);

cout << "BST<T>::treeHeight() = ";

cout << myIntBST.treeHeight() << endl;

cout << endl;

cout << "BST<T>::treeNodeCount() = ";

cout << myIntBST.treeNodeCount() << endl;

cout << endl;

cout << "BST<T>::treeLeavesCount() = ";

cout << myIntBST.treeLeavesCount() << endl;

cout << endl;

cout << "BST<T>::printNodeLevel() {\n ";

myIntBST.preorderTraversal();

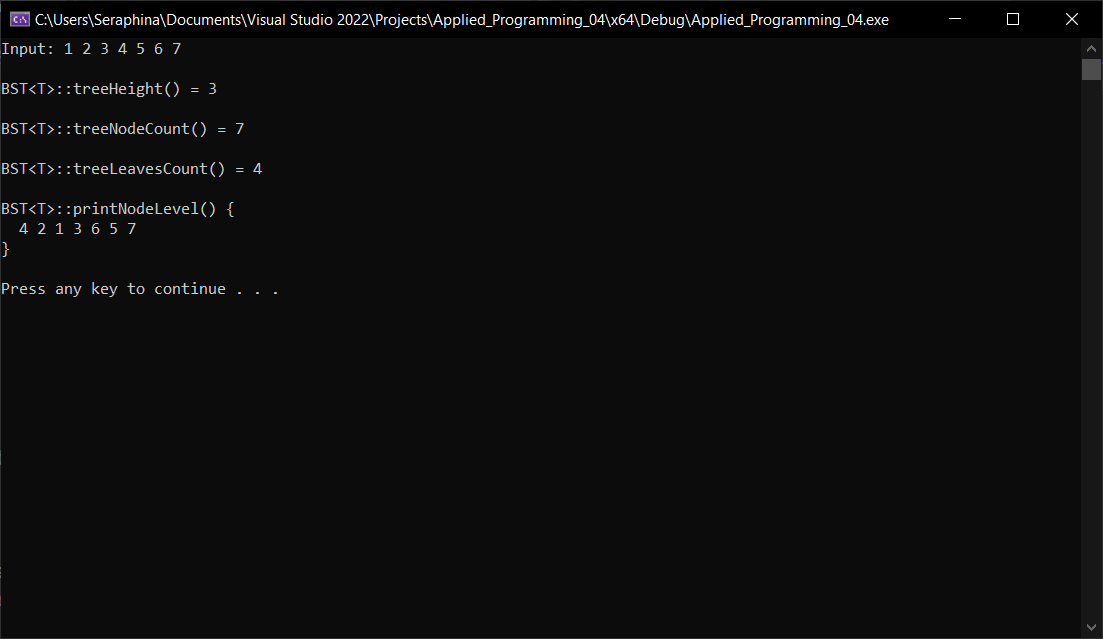
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



# **Question#3**

Sum Pairs

## **Code**

#include "binarySearchTree.h"

using namespace std;

int main()

{

/\*

// Sample Input: 65 55 22 44 61 19 90 10 78

52

BST<int> myIntBST;

// test all functions here

\*/

BST<int> myIntBST;

int arr[] = { 65, 55, 22, 44, 61, 19, 90, 10, 78, 66 };

int size = sizeof(arr) / sizeof(arr[0]);

for (int i = 0; i < size; i++)

{

cout << "BST<T>::Insert(" << arr[i] << ");" << endl;

myIntBST.Insert(arr[i]);

}

cout << endl;

cout << "BST<T>::printNodeLevel() {\n ";

myIntBST.preorderTraversal();

cout << "\n}\n" << endl;

cout << "BST<T>::isPairPresent(66) {\n ";

myIntBST.isPairPresent(66);

cout << "\n}\n" << endl;

cout << "BST<T>::isPairPresent(90) {\n ";

myIntBST.isPairPresent(90);

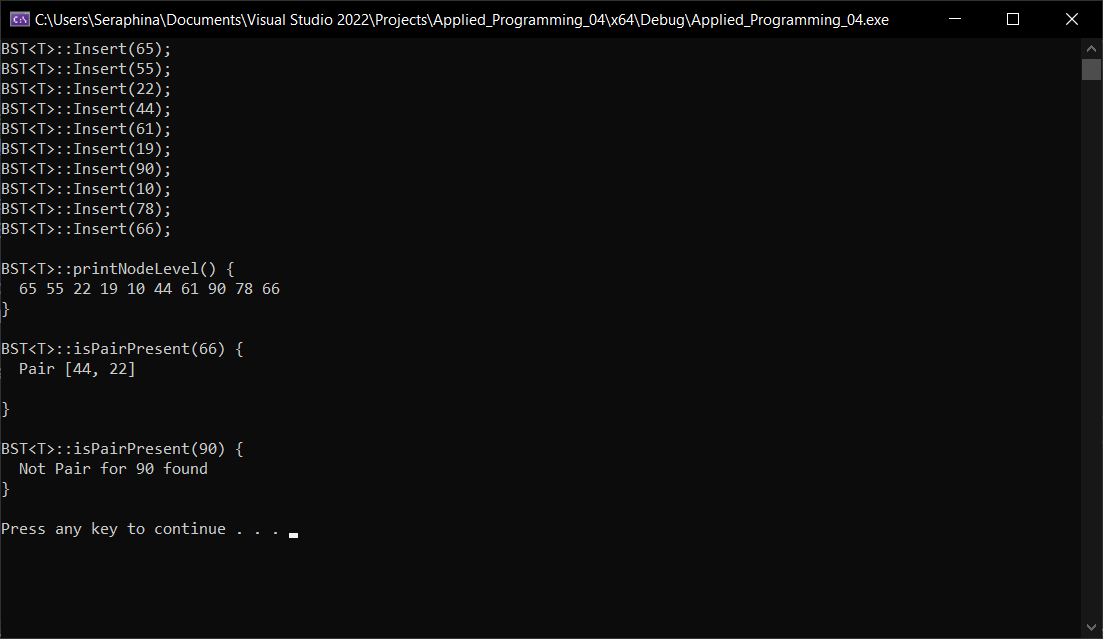
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



# **binarySearchTree.h**

#pragma once

#include <iostream>

#define max(x, y) \

x > y ? x : y

template <class type>

struct Node

{

type value;

Node<type>\* lhs, \* rhs;

int height;

};

template <class type>

class BST

{

private:

Node<type>\* Tree;

// Old Functions

Node<type>\* DeleteKey(Node<type>\* node, type key);

bool SearchKey(Node<type>\* node, type key);

int findMax(Node<type>\* node);

int findMin(Node<type>\* node);

void preorderTraversal(Node<type>\* node);

void inorderTraversal(Node<type>\* node);

void postorderTraversal(Node<type>\* node);

int treeHeight(Node<type>\* node);

int treeNodeCount(Node<type>\* node);

int treeLeavesCount(Node<type>\* node);

void printGivenLevel(Node<type>\* node, int lvl);

void printNodeLevel(Node<type>\* node);

// AVL Functions

Node<type>\* InitiateNode(type value);

int GetHeight(Node<type>\* node);

int GetBalance(Node<type>\* node);

Node<type>\* RotateLeft(Node<type>\* node);

Node<type>\* RotateRight(Node<type>\* node);

Node<type>\* Insert(Node<type>\* node, type value);

// Task2, More Function

Node<type>\* ConvertSortedArrayToTree(int\* arr, int s\_offset, int e\_offset);

// Task3, More Function

bool pairSearch(Node<type>\* node, int value);

bool isPairPresent(Node<type>\* node, int value);

public:

BST();

void Insert(type value); // : insert an element in the tree

void DeleteKey(type key); // : delete an element in the tree

bool SearchKey(type key); // : searches the desired element in the tree recursively

int findMax(); // : finds the maximum element in the tree recursively

int findMin(); // : finds the minimum element in the tree recursively

void inorderTraversal(); // : prints in - order traversal of the tree

void preorderTraversal(); // : prints pre - order traversal of the tree

void postorderTraversal(); // : prints post - order traversal of the tree

int treeHeight(); // : returns the height of the tree recursively

int treeNodeCount(); // : returns the count of nodes in the tree

int treeLeavesCount(); // : returns the count of leaves in the tree

void printNodeLevel(); // : prints level of a node in the tree

// Task2, Extra Function

void ConvertSortedArrayToTree(int\* arr, int size);

// Task3, Extra Function

bool isPairPresent(type value);

};

// Private Functions

//

template <class type>

bool BST<type>::pairSearch(Node<type>\* node, int value)

{

if (Tree == NULL)

{

return false;

}

Node<type>\* root = Tree;

bool status = false;

while (root != NULL && status != true)

{

if (root->value == value && node != root)

{

std::cout << "Pair [" << root->value << ", " << node->value << "]";

std::cout << std::endl;

status = true;

break;

}

else if (value < root->value)

{

root = root->lhs;

}

else

{

root = root->rhs;

}

}

return status;

}

template <class type>

bool BST<type>::isPairPresent(Node<type>\* node, int value)

{

if (node == NULL)

{

return false;

}

return pairSearch(node, value - node->value) || isPairPresent(node->lhs, value) || isPairPresent(node->rhs, value);

}

template <class type>

Node<type>\* BST<type>::ConvertSortedArrayToTree(int\* arr, int s\_offset, int e\_offset)

{

if (s\_offset > e\_offset)

{

return NULL;

}

int m\_offset = (s\_offset + e\_offset) / 2;

Node<type>\* root = new Node<type>;

root->value = arr[m\_offset];

root->lhs = ConvertSortedArrayToTree(arr, s\_offset, m\_offset - 1);

root->rhs = ConvertSortedArrayToTree(arr, m\_offset + 1, e\_offset);

return root;

}

// Older Functions

template <class type>

Node<type>\* BST<type>::DeleteKey(Node<type>\* node, type key)

{

if (node == NULL)

{

return node;

}

if (key < node->value)

{

node->lhs = DeleteKey(node->lhs, key);

}

else if (key > node->value)

{

node->rhs = DeleteKey(node->rhs, key);

}

else

{

if (!node->lhs || !node->rhs)

{

Node<type>\* temp = node->lhs ? node->lhs : node->rhs;

if (temp == NULL)

{

temp = node;

node = NULL;

}

else

{

node = temp;

}

delete temp;

}

else

{

Node<type>\* temp = node->rhs;

while (temp->lhs != NULL)

{

temp = temp->lhs;

}

node->value = temp->value;

node->rhs = DeleteKey(node->rhs, temp->value);

}

}

if (node == NULL)

{

return node;

}

node->height = 1 + max(GetHeight(node->lhs), GetHeight(node->rhs));

int balance = GetBalance(node);

if (balance > 1 && GetBalance(node->lhs) >= 0)

{

return RotateRight(node);

}

if (balance > 1 && GetBalance(node->lhs) < 0)

{

node->lhs = RotateLeft(node->lhs);

return RotateRight(node);

}

if (balance < -1 && GetBalance(node->rhs) <= 0)

{

return RotateLeft(node);

}

if (balance < -1 && GetBalance(node->rhs) > 0)

{

node->rhs = RotateRight(node->rhs);

return RotateLeft(node);

}

return node;

}

template <class type>

bool BST<type>::SearchKey(Node<type>\* node, type key)

{

if (node == NULL)

{

return false;

}

if (key > node->value)

{

return SearchKey(node->rhs, key);

}

else if (key < node->value)

{

return SearchKey(node->lhs, key);

}

return true;

}

template <class type>

int BST<type>::findMax(Node<type>\* node)

{

if (node->rhs == NULL)

{

return node->value;

}

return findMax(node->rhs);

}

template <class type>

int BST<type>::findMin(Node<type>\* node)

{

if (node->lhs == NULL)

{

return node->value;

}

return findMin(node->lhs);

}

template <class type>

void BST<type>::preorderTraversal(Node<type>\* node)

{

if (node == NULL)

{

return;

}

std::cout << node->value << " ";

preorderTraversal(node->lhs);

preorderTraversal(node->rhs);

}

template <class type>

void BST<type>::inorderTraversal(Node<type>\* node)

{

if (node == NULL)

{

return;

}

preorderTraversal(node->lhs);

std::cout << node->value << " ";

preorderTraversal(node->rhs);

}

template <class type>

void BST<type>::postorderTraversal(Node<type>\* node)

{

if (node == NULL)

{

return;

}

preorderTraversal(node->lhs);

preorderTraversal(node->rhs);

std::cout << node->value << " ";

}

template <class type>

int BST<type>::treeHeight(Node<type>\* node)

{

if (node == NULL)

{

return 0;

}

else

{

int lhs\_height = treeHeight(node->lhs);

int rhs\_height = treeHeight(node->rhs);

if (lhs\_height > rhs\_height)

{

return lhs\_height + 1;

}

else

{

return rhs\_height + 1;

}

}

}

template <class type>

int BST<type>::treeNodeCount(Node<type>\* node)

{

if (node == NULL)

{

return 0;

}

return 1 + treeNodeCount(node->lhs) + treeNodeCount(node->rhs);

}

/\*

Code Taken from

https://www.geeksforgeeks.org/print-level-order-traversal-line-line/

\*/

template <class type>

int BST<type>::treeLeavesCount(Node<type>\* node)

{

if (node == NULL)

{

return 0;

}

if (!node->lhs && !node->rhs)

{

return 1;

}

else

{

return treeLeavesCount(node->lhs) + treeLeavesCount(node->rhs);

}

}

template <class type>

void BST<type>::printGivenLevel(Node<type>\* node, int lvl)

{

if (node == NULL)

{

return;

}

if (lvl == 1)

{

std::cout << " " << node->value;

}

else if (lvl > 1) {

printGivenLevel(node->lhs, lvl - 1);

printGivenLevel(node->rhs, lvl - 1);

}

}

template <class type>

void BST<type>::printNodeLevel(Node<type>\* node)

{

int height = treeHeight(node);

for (int i = 1; i <= height; i++)

{

printGivenLevel(node, i);

std::cout << std::endl;

}

}

// Newer Functions

template <class type>

Node<type>\* BST<type>::InitiateNode(type value)

{

Node<type>\* node = new Node<type>;

node->value = value;

node->lhs = node->rhs = NULL;

node->height = 1;

return node;

}

template <class type>

int BST<type>::GetHeight(Node<type>\* node)

{

if (node == NULL)

{

return 0;

}

return node->height;

}

template <class type>

int BST<type>::GetBalance(Node<type>\* node)

{

if (node == NULL)

{

return 0;

}

return GetHeight(node->lhs) - GetHeight(node->rhs);

}

template <class type>

Node<type>\* BST<type>::RotateLeft(Node<type>\* node)

{

Node<type>\* x = node->rhs;

Node<type>\* y = x->lhs;

x->lhs = node;

node->rhs = y;

node->height = max(GetHeight(node->lhs), GetHeight(node->rhs)) + 1;

x->height = max(GetHeight(x->lhs), GetHeight(x->rhs)) + 1;

return x;

}

template <class type>

Node<type>\* BST<type>::RotateRight(Node<type>\* node)

{

Node<type>\* x = node->lhs;

Node<type>\* y = x->rhs;

x->rhs = node;

node->lhs = y;

node->height = max(GetHeight(node->lhs), GetHeight(node->rhs)) + 1;

x->height = max(GetHeight(x->lhs), GetHeight(x->rhs)) + 1;

return x;

}

template <class type>

Node<type>\* BST<type>::Insert(Node<type>\* node, type value)

{

if (node == NULL)

{

return InitiateNode(value);

}

if (value < node->value)

{

node->lhs = Insert(node->lhs, value);

}

else if (value > node->value)

{

node->rhs = Insert(node->rhs, value);

}

else

{

return node;

}

node->height = 1 + max(GetHeight(node->lhs), GetHeight(node->rhs));

int balance = GetBalance(node);

if (balance > 1 && value < node->lhs->value)

{

return RotateRight(node);

}

if (balance < -1 && value > node->rhs->value)

{

return RotateLeft(node);

}

if (balance > 1 && value > node->lhs->value)

{

node->lhs = RotateLeft(node->lhs);

return RotateRight(node);

}

if (balance < -1 && value < node->rhs->value)

{

node->rhs = RotateRight(node->rhs);

return RotateLeft(node);

}

return node;

}

// Public Functions

template <class type>

BST<type>::BST()

{

Tree = NULL;

}

template <class type>

void BST<type>::Insert(type value)

{

Tree = Insert(Tree, value);

}

template <class type>

void BST<type>::DeleteKey(type key)

{

DeleteKey(Tree, key);

}

template <class type>

bool BST<type>::SearchKey(type key)

{

return SearchKey(Tree, key);

}

template <class type>

int BST<type>::findMax()

{

return findMax(Tree);

}

template <class type>

int BST<type>::findMin()

{

return findMin(Tree);

}

template <class type>

void BST<type>::inorderTraversal()

{

inorderTraversal(Tree);

}

template <class type>

void BST<type>::preorderTraversal()

{

preorderTraversal(Tree);

}

template <class type>

void BST<type>::postorderTraversal()

{

postorderTraversal(Tree);

}

template <class type>

int BST<type>::treeHeight()

{

return treeHeight(Tree);

}

template <class type>

int BST<type>::treeNodeCount()

{

return treeNodeCount(Tree);

}

template <class type>

int BST<type>::treeLeavesCount()

{

return treeLeavesCount(Tree);

}

template <class type>

void BST<type>::printNodeLevel()

{

printNodeLevel();

}

template <class type>

void BST<type>::ConvertSortedArrayToTree(int\* arr, int size)

{

Tree = ConvertSortedArrayToTree(arr, 0, size - 1);

}

/\*

Taken From

https://www.geeksforgeeks.org/find-a-pair-with-given-sum-in-bst/

\*/

template <class type>

bool BST<type>::isPairPresent(type value)

{

bool status = isPairPresent(Tree, value);

if (!status)

std::cout << "Not Pair for " << value << " found";

return status;

}