

Applied Programming

Assignment#3



November 6, 2022

Muhammad talha Arif

22F-3769

# **Question#1**

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();

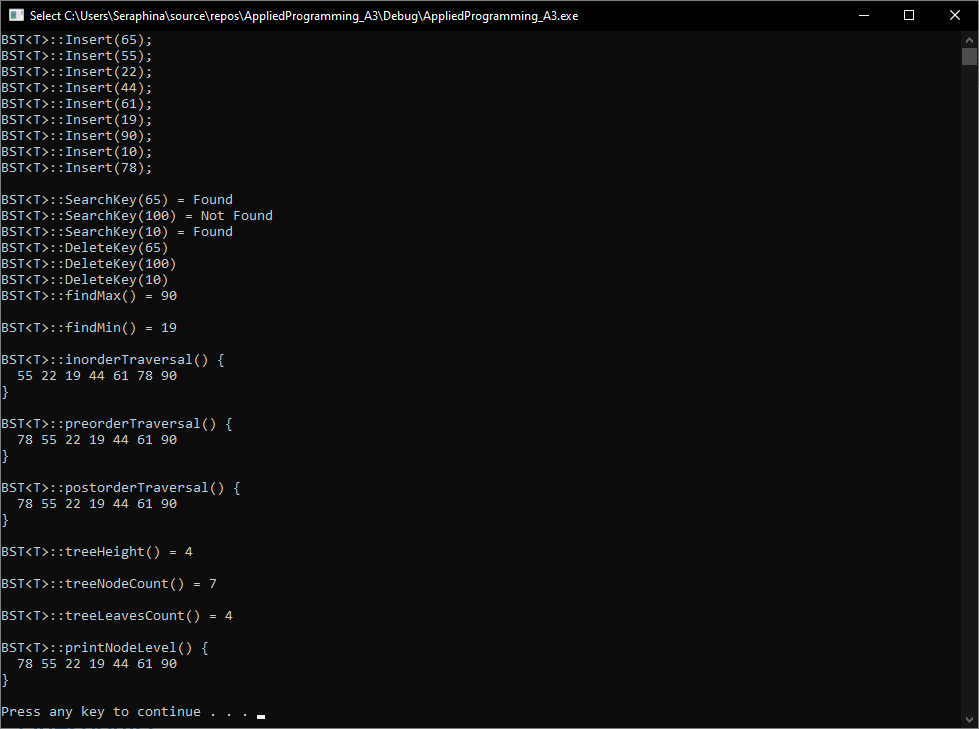
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



# **Question#2**

Mirror Binary Tree

## **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>::inorderTraversal() {\n ";

myIntBST.inorderTraversal();

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

cout << "BST<T>::ConvertIntoMirror();";

myIntBST.ConvertIntoMirror();

cout << endl;

cout << endl;

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

myIntBST.inorderTraversal();

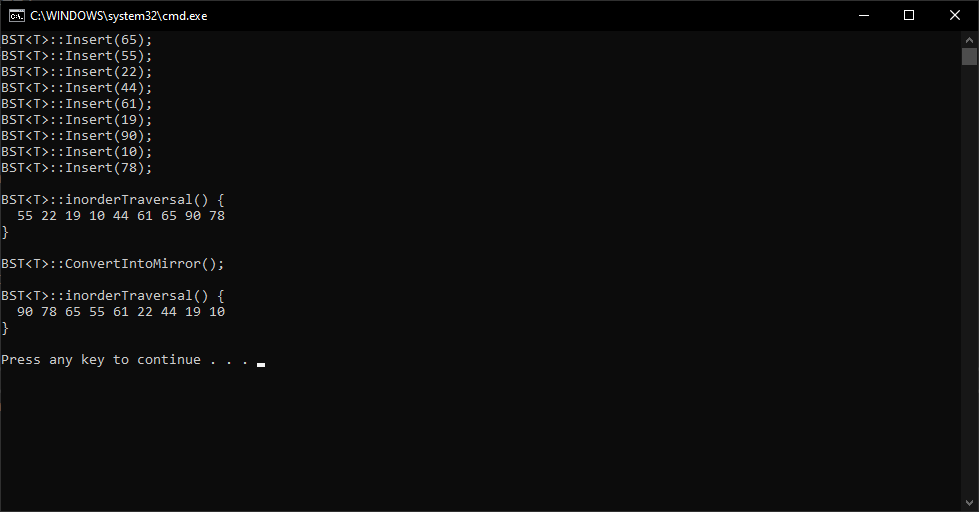
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

## }

## **Screenshot**



# **Question#3**

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();

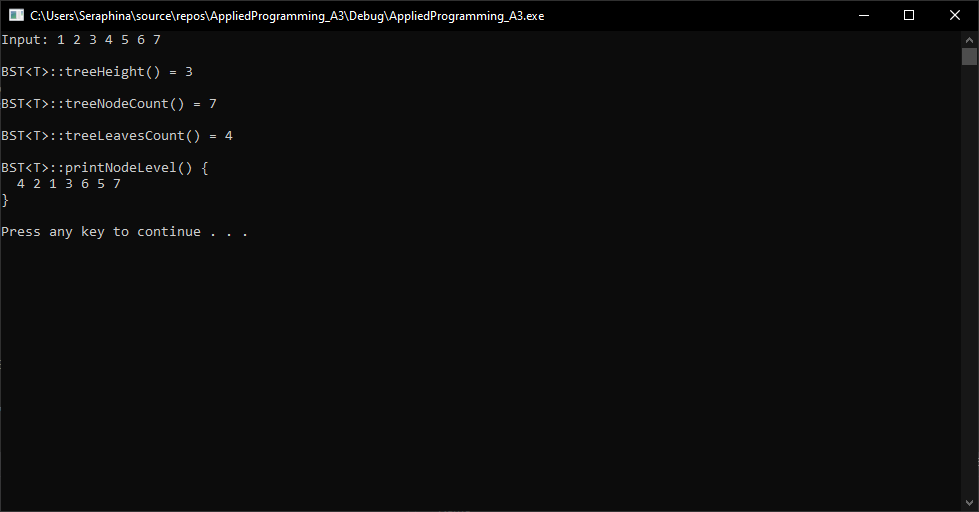
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

## }

## **Screenshot**



# **Question#4**

Binary Tree

## **A)**

## **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;

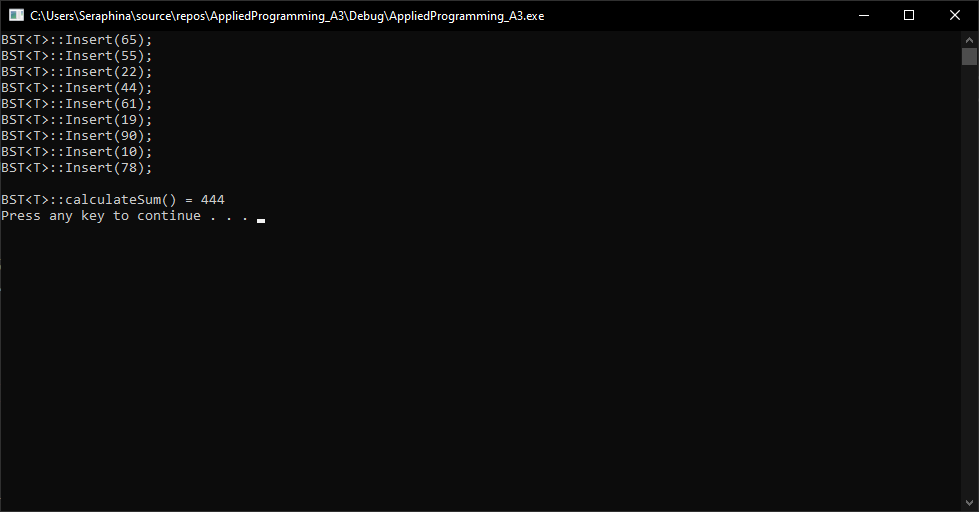
cout << "BST<T>::calculateSum() = " << myIntBST.calculateSum() << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



## **B)**

## **Code**

#include <iostream>

#include <stack>

using namespace std;

struct Node

{

int value;

Node\* lhs, \* rhs;

};

bool XorGate(bool A, bool B);

bool NorGate(bool A, bool B);

bool OrGate(bool A, bool B);

bool NandGate(bool A, bool B);

bool AndGate(bool A, bool B);

bool NotGate(bool A);

class BST

{

private:

Node\* Tree;

void applyAndGate(Node\* n, int& result)

{

if (n == NULL)

{

return;

}

if (result == -1)

{

result = n->value;

}

else

{

result = AndGate(result, n->value);

}

applyAndGate(n->lhs, result);

applyAndGate(n->rhs, result);

}

void applyXorGate(Node\* n, int& result)

{

if (n == NULL)

{

return;

}

if (result == -1)

{

result = n->value;

}

else

{

result = XorGate(result, n->value);

}

applyXorGate(n->lhs, result);

applyXorGate(n->rhs, result);

}

void preOrderTraversal(Node\* n)

{

if (n == NULL)

{

return;

}

cout << n->value << " ";

preOrderTraversal(n->lhs);

preOrderTraversal(n->rhs);

}

public:

BST()

{

Tree = NULL;

}

void Insert(int value)

{

Node\* node = new Node();

node->value = value;

node->lhs = NULL;

node->rhs = NULL;

if (Tree == NULL)

{

Tree = node;

}

else

{

Node\* temp = Tree;

Node\* prev = NULL;

while (temp != NULL)

{

if (value < temp->value)

{

prev = temp;

temp = temp->lhs;

}

else if (value >= temp->value)

{

prev = temp;

temp = temp->rhs;

}

else

{

printf("Duplicate not allowed...\n");

return;

}

}

if (value < prev->value)

prev->lhs = node;

else if (value >= prev->value)

prev->rhs = node;

}

}

void preOrderTraversal()

{

preOrderTraversal(Tree);

}

int applyAndGate()

{

int result = -1;

applyAndGate(Tree, result);

return result;

}

int applyXorGate()

{

int result = -1;

applyXorGate(Tree, result);

return result;

}

};

int main()

{

BST\* bst = new BST();

int value = 101;

stack<int> s;

while (value > 0)

{

s.push(value % 10);

value /= 10;

}

while (!s.empty())

{

cout << "BST::Insert(" << s.top() << ")" << endl;

bst->Insert(s.top());

s.pop();

}

cout << "BST::applyAndGate() = ";

cout << bst->applyAndGate() << endl;

cout << "BST::applyXorGate() = ";

cout << bst->applyXorGate() << endl;

system("PAUSE");

return 0;

}

bool NotGate(bool A)

{

return !A;

}

bool AndGate(bool A, bool B)

{

return (A & B);

}

bool NandGate(bool A, bool B)

{

return NotGate(AndGate(A, B));

}

bool OrGate(bool A, bool B)

{

return (A | B);

}

bool NorGate(bool A, bool B)

{

return NotGate(OrGate(A, B));

}

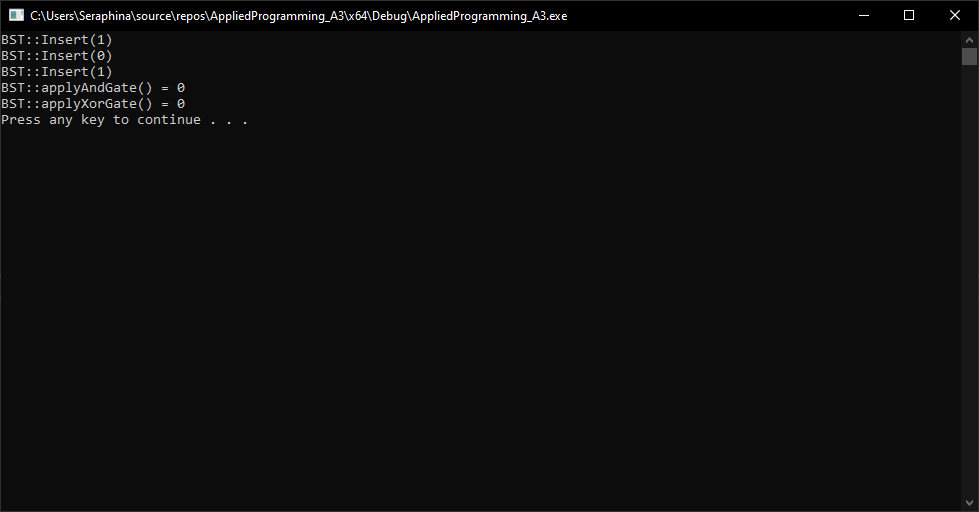
bool XorGate(bool A, bool B)

{

return (A ^ B);

}

## **Screenshot**



# **Question#5**

Binary Search Tree Applications

## **A)**

## **Code**

/\*

Code Taken from

https://www.acwing.com/solution/content/19658/

\*/

#include <iostream>

#include <vector>

using namespace std;

#define LL long long

const int mod = 1000000007;

class Solution

{

private:

vector<LL> fact;

LL power(LL x, LL y)

{

LL tot = 1, p = x;

for (; y; y >>= 1)

{

if (y & 1)

tot = tot \* p % mod;

p = p \* p % mod;

}

return tot;

}

LL select(int n, int m)

{

return fact[n] \* power(fact[m], mod - 2) % mod \* power(fact[(LL)n - m], mod - 2) % mod;

}

int solve(const vector<int>& nums)

{

if (nums.size() == 0)

return 1;

vector<int> lo, hi;

const int n = nums.size();

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

{

if (nums[i] < nums[0]) lo.push\_back(nums[i]);

else hi.push\_back(nums[i]);

}

return select(n - 1, lo.size()) \* solve(lo) % mod \* solve(hi) % mod;

}

public:

int numOfWays(vector<int>& nums)

{

const int n = nums.size();

fact.resize(n);

fact[0] = 1;

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

{

fact[i] = fact[(LL)i - 1] \* i % mod;

}

return (solve(nums) - 1 + mod) % mod;

}

};

int main()

{

vector<int> vec = { 3,4,5,1,2 };

int size = vec.size();

Solution solution;

int numOfWays = solution.numOfWays(vec);

cout << "Input: nums = [";

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

{

cout << vec[i];

if (i < size - 1)

cout << ",";

}

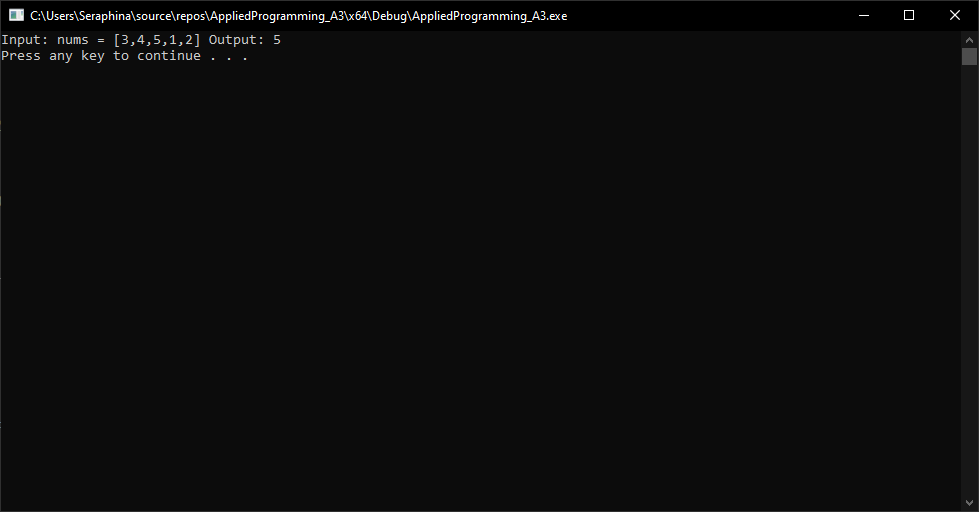
cout << "] Output: " << numOfWays << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



## **B)**

## **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>::convertToGreaterTree() = " << endl;

myIntBST.convertToGreaterTree();

cout << endl;

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

myIntBST.inorderTraversal();

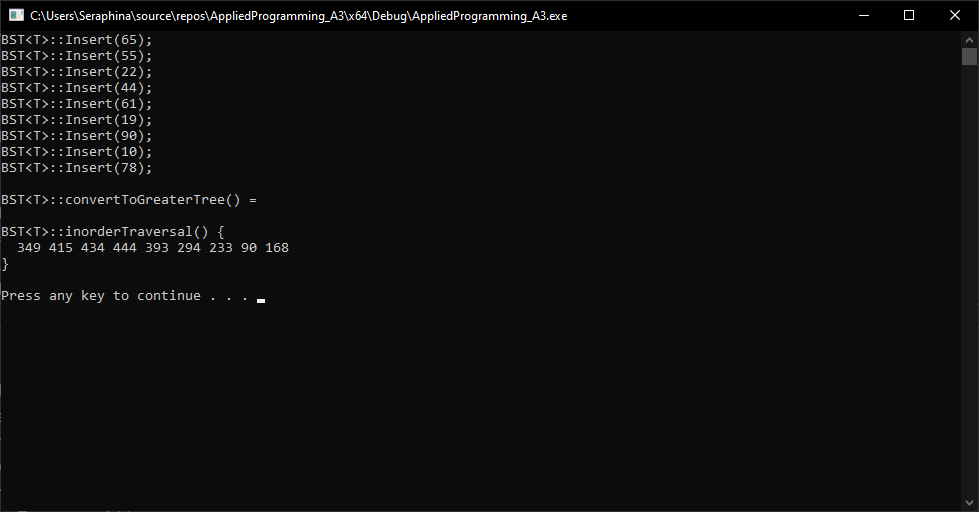
cout << "\n}\n" << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



# **binarySearchTree.h**

#pragma once

#include <iostream>

template <class T>

struct Node

{

T value;

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

};

template <class T>

class BST

{

private:

Node<T>\* Tree;

Node<T>\* DeleteKey(Node<T>\* n, T key)

{

if (n == NULL)

{

return NULL;

}

if (key < n->value)

{

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

}

else if (key > n->value)

{

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

}

else

{

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

{

return NULL;

}

else if (n->lhs == NULL)

{

Node<T>\* temp = n->rhs;

delete n;

return temp;

}

else if (n->rhs == NULL)

{

Node<T>\* temp = n->lhs;

delete n;

return temp;

}

Node<T>\* temp = n->rhs;

while (temp->lhs != NULL)

{

temp = temp->lhs;

}

n->value = temp->value;

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

}

return n;

}

bool SearchKey(Node<T>\* n, T key)

{

if (n == NULL)

{

return false;

}

if (key > n->value)

{

return SearchKey(n->rhs, key);

}

else if (key < n->value)

{

return SearchKey(n->lhs, key);

}

return true;

}

int findMax(Node<T>\* n)

{

if (n->rhs == NULL)

return n->value;

return findMax(n->rhs);

}

int findMin(Node<T>\* n)

{

if (n->lhs == NULL)

return n->value;

return findMin(n->lhs);

}

void preorderTraversal(Node<T>\* n)

{

if (n == NULL)

return;

std::cout << n->value << ' ';

preorderTraversal(n->lhs);

preorderTraversal(n->rhs);

}

void inorderTraversal(Node<T>\* n)

{

if (n == NULL)

return;

preorderTraversal(n->lhs);

std::cout << n->value << ' ';

preorderTraversal(n->rhs);

}

void postorderTraversal(Node<T>\* n)

{

if (n == NULL)

return;

preorderTraversal(n->lhs);

preorderTraversal(n->rhs);

std::cout << n->n << ' ';

}

int treeHeight(Node<T>\* n)

{

if (n == NULL)

{

return 0;

}

else

{

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

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

if (lhs\_height > rhs\_height)

return lhs\_height + 1;

else

return rhs\_height + 1;

}

}

int treeNodeCount(Node<T>\* n)

{

if (n == NULL)

{

return 0;

}

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

}

/\*

Code Taken from

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

\*/

int treeLeavesCount(Node<T>\* n)

{

if (n == NULL)

{

return 0;

}

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

{

return 1;

}

else

{

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

}

}

void printGivenLevel(Node<T>\* n, int lvl)

{

if (n == NULL)

{

return;

}

if (lvl == 1)

{

std::cout << " " << n->data;

}

else if (lvl > 1) {

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

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

}

}

void printNodeLevel(Node<T>\* n)

{

int height = treeHeight(n);

int i;

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

{

printGivenLevel(n, i);

std::cout << std::endl;

}

}

void ConvertIntoMirror(Node<T>\* node)

{

if (node == NULL)

{

return;

}

else {

Node<T>\* temp;

ConvertIntoMirror(node->lhs);

ConvertIntoMirror(node->rhs);

temp = node->lhs;

node->lhs = node->rhs;

node->rhs = temp;

}

}

Node<T>\* 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<T>\* root = new Node<T>();

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;

}

int calculateSum(Node<T>\* n)

{

int sum\_left = 0, sum\_right = 0;

if (n == NULL)

{

return 0;

}

else

{

if (n->lhs)

{

sum\_left = calculateSum(n->lhs);

}

if (n->rhs)

{

sum\_right = calculateSum(n->rhs);

}

return n->value + sum\_left + sum\_right;

}

}

void convertToGreaterTree(Node<T>\* n, int\* sum)

{

if (n == NULL)

{

return;

}

convertToGreaterTree(n->rhs, sum);

\*sum += n->value;

n->value = \*sum;

convertToGreaterTree(n->lhs, sum);

}

public:

BST();

void Insert(T value);

void DeleteKey(T key);

bool SearchKey(T key);

int findMax();

int findMin();

void inorderTraversal();

void preorderTraversal();

void postorderTraversal();

int treeHeight();

int treeNodeCount();

int treeLeavesCount();

void printNodeLevel();

// Task2, Extra Function

void ConvertIntoMirror();

// Task3, Extra Function

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

// Task4, Extra Function

int calculateSum();

// Task5, Extra Function

void convertToGreaterTree();

};

template <class T>

BST<T>::BST()

{

Tree = NULL;

}

template <class T>

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

{

Node<T>\* node = new Node<T>();

node->value = value;

node->lhs = NULL;

node->rhs = NULL;

if (Tree == NULL)

{

Tree = node;

}

else

{

Node<T>\* temp = Tree;

Node<T>\* prev = NULL;

while (temp != NULL)

{

if (value < temp->value)

{

prev = temp;

temp = temp->lhs;

}

else if (value > temp->value)

{

prev = temp;

temp = temp->rhs;

}

else

{

printf("Duplicate not allowed...\n");

return;

}

}

if (value < prev->value)

prev->lhs = node;

else if (value > prev->value)

prev->rhs = node;

}

}

template <class T>

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

{

DeleteKey(Tree, key);

}

template <class T>

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

{

bool status = SearchKey(Tree, key);

/\* if (status)

{

std::cout << "BST<T>::SearchKey(" << key << "), exists..." << std::endl;

}

else

{

std::cout << "BST<T>::SearchKey(" << key << "), not exists..." << std::endl;

}\*/

return status;

}

template <class T>

int BST<T>::findMax()

{

return findMax(Tree);

}

template <class T>

int BST<T>::findMin()

{

return findMin(Tree);

}

template <class T>

void BST<T>::inorderTraversal()

{

inorderTraversal(Tree);

}

template <class T>

void BST<T>::preorderTraversal()

{

preorderTraversal(Tree);

}

template <class T>

void BST<T>::postorderTraversal()

{

postorderTraversal(Tree);

}

template <class T>

int BST<T>::treeHeight()

{

return treeHeight(Tree);

}

template <class T>

int BST<T>::treeNodeCount()

{

return treeNodeCount(Tree);

}

template <class T>

int BST<T>::treeLeavesCount()

{

return treeLeavesCount(Tree);

}

template <class T>

void BST<T>::printNodeLevel()

{

printNodeLevel();

}

template <class T>

void BST<T>::ConvertIntoMirror()

{

ConvertIntoMirror(Tree);

}

template <class T>

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

{

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

}

template <class T>

int BST<T>::calculateSum()

{

return calculateSum(Tree);

}

template <class T>

void BST<T>::convertToGreaterTree()

{

int dummy = 0;

convertToGreaterTree(Tree, &dummy);

}