**TASK 1:**

#ifndef BINARYSEARCHTREE\_H // include guard

#define BINARYSEARCHTREE\_H

#include<iostream>

template<class T>

class Node

{

public:

T data;

Node<T>\* left;

Node<T>\* right;

Node()

{

left = right = NULL;

}

};

template<class T>

class binarySearchTree

{

Node<T>\* root;

void insert(T, Node<T>\*&);

void deleteNode(T, Node<T>\*&);

void makeDeletion(Node<T>\*&);

bool search(T, Node<T>\*&);

T max(Node<T>\*&);

T min(Node<T>\*&);

void inorder(Node<T>\*&);

void preorder(Node<T>\*&);

void postorder(Node<T>\*&);

int Height(Node<T>\*&);

int NodeCount(Node<T>\*&);

int LeavesCount(Node<T>\*&);

int printnodeLevel(Node<T>\*&, T);

public:

binarySearchTree();

void Insert(T); //insert an element in the tree

void DeleteKey(T);// delete an element in the tree

bool SearchKey(T); //searches the desired element in the tree recursively

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

T 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(T); //prints level of a node in the tree

};

#endif BINARYSEARCHTREE\_H

#include"binarySearchTree.h"

using namespace std;

template<class T>

binarySearchTree<T>::binarySearchTree()

{

root = NULL;

}

template<class T>

void binarySearchTree<T>::insert(T data, Node<T>\*& ptr) //insert an element in the tree

{

if (ptr == NULL) {

ptr = new Node<T>;

ptr->data = data;

ptr->left = NULL;

ptr->right = NULL;

}

else if (ptr->data > data)

{

insert(data, ptr->left);

}

else if (ptr->data < data)

{

insert(data, ptr->right);

}

else {

cout << "Node not inserted!" << endl;

}

}

template<class T>

void binarySearchTree<T>::deleteNode(T data, Node<T>\*& ptr)

{

if (ptr == NULL) // node does not exist in the tree

{

cout << data << "not found.\n";

}

else if (data < ptr->data)

{

deleteNode(data, ptr->left); // find in left subtree

}

else if (data > ptr->data)

{

deleteNode(data, ptr->right); // find in right subtree

}

else

{

makeDeletion(ptr); // actually deletes node from BST

}

}

template<class T>

void binarySearchTree<T>::makeDeletion(Node<T>\*& node)

{

Node<T>\* temp; // Temperary pointer

if (node->right == NULL) { // case for leaf and one (left) child

temp = node;

node = node->left; // Reattach the left child

delete temp;

}

else if (node->left == NULL) { // case for one (right) child

temp = node;

node = node->right; // Reattach the right child

delete temp;

}

else

{ // case for two children.

temp = node->right; // Move one node to the right

while (temp->left)

{ // Go to the extreme left node

temp = temp->left;

}

temp->left = node->left; // Reattach the left subtree

temp = node;

node = node->right; // Reattach the right subtree

delete temp;

}

}

template<class T>

bool binarySearchTree<T>::search(T data, Node<T>\*& node)

{

if (node)

{

if (node->data == data)

{

return true;

}

else if (search(data, node->left))

{

return true;

}

else if (search(data, node->right))

{

return true;

}

else

{

return false;

}

}

else

{

return false;

}

}

template<class T>

T binarySearchTree<T>::max(Node<T>\*& ptr)

{

if (ptr->right)

{

return max(ptr->left);

}

else

{

return ptr->data;

}

}

template<class T>

T binarySearchTree<T>::min(Node<T>\*& ptr)

{

if (ptr->left)

{

return min(ptr->left);

}

else

{

return ptr->data;

}

}

template<class T>

void binarySearchTree<T>::inorder(Node<T>\*& ptr)//prints in - order traversal of the tree

{

if (ptr != NULL)

{

inorder(ptr->left);

cout << ptr->data << " ";

inorder(ptr->right);

}

}

template<class T>

void binarySearchTree<T>::preorder(Node<T>\*& ptr)//prints pre - order traversal of the tree

{

if (ptr != NULL)

{

cout << ptr->data << " ";

preorder(ptr->left);

preorder(ptr->right);

}

}

template<class T>

void binarySearchTree<T>::postorder(Node<T>\*& ptr) //prints post - order traversal of the tree

{

if (ptr != NULL)

{

postorder(ptr->left);

postorder(ptr->right);

cout << ptr->data << " ";

}

}

template<class T>

int binarySearchTree<T>::Height(Node<T>\*& ptr) {

int hleft = 0, hright = 0;

if (ptr)

{

hleft = Height(ptr->left);

hright = Height(ptr->right);

if (hleft > hright)

{

return hleft + 1;

}

else

{

return hright + 1;

}

}

else

{

return 0;

}

}

template<class T>

int binarySearchTree<T>::NodeCount(Node<T>\*& ptr)

{

int nodecount = 0;

if (ptr)

{

nodecount += NodeCount(ptr->left);

nodecount += NodeCount(ptr->right);

return nodecount + 1;

}

else

{

return 0;

}

}

template<class T>

int binarySearchTree<T>::LeavesCount(Node<T>\*& ptr)

{

int count = 0;

if (ptr)

{

if (ptr->left == NULL && ptr->right == NULL)

{

return count + 1;

}

else

{

count += LeavesCount(ptr->left);

count += LeavesCount(ptr->right);

}

}

else

{

return 0;

}

}

template<class T>

int binarySearchTree<T>::printnodeLevel(Node<T>\*& ptr, T data)

{

int height = 0;

if (ptr)

{

if (ptr->data == data)

{

return height + 1;

}

else

{

height = printnodeLevel(ptr->left, data) + 1;

height = printnodeLevel(ptr->right, data) + 1;

}

}

else

{

return 0;

}

return height;

}

//////////////////////////////////////////////////////////////////////////// Public Functions

template<class T>

void binarySearchTree<T>::Insert(T data) //insert an element in the tree

{

insert(data, root);

}

template<class T>

void binarySearchTree<T>::DeleteKey(T data)// delete an element in the tree

{

deleteNode(data, root);

}

template<class T>

bool binarySearchTree<T>::SearchKey(T data) //searches the desired element in the tree recursively

{

return search(data, root);

}

template<class T>

T binarySearchTree<T>::findMax()

{

return max(root);

}

template<class T>

T binarySearchTree<T>::findMin()

{

return min(root);

}

template<class T>

void binarySearchTree<T>::inorderTraversal()//prints in - order traversal of the tree

{

inorder(root);

}

template<class T>

void binarySearchTree<T>::preorderTraversal()//prints pre - order traversal of the tree

{

preorder(root);

}

template<class T>

void binarySearchTree<T>::postorderTraversal() //prints post - order traversal of the tree

{

postorder(root);

}

template<class T>

int binarySearchTree<T>::treeHeight()//returns the height of the tree recursively

{

return Height(root);

}

template<class T>

int binarySearchTree<T>::treeNodeCount() //returns the count of nodes in the tree

{

return NodeCount(root);

}

template<class T>

int binarySearchTree<T>::treeLeavesCount() //returns the count of leaves in the tree

{

return LeavesCount(root);

}

template<class T>

void binarySearchTree<T>::printNodeLevel(T data) //prints level of a node in the tree

{

cout << printnodeLevel(root, data) << endl;

}

int main()

{

int choice = -1, data = 0;

binarySearchTree<int> bst;

while (choice)

{

cout << "1.Insert" << endl;

cout << "2.DeleteKey" << endl;

cout << "3.SearchKey" << endl;

cout << "4.findMax" << endl;

cout << "5.findMin" << endl;

cout << "6.inorderTraversal" << endl;

cout << "7.preorderTraversal" << endl;

cout << "8.postorderTraversal" << endl;

cout << "9.treeHeight" << endl;

cout << "10.treeNodeCount" << endl;

cout << "11.treeLeavesCount" << endl;

cout << "12.printNodeLevel" << endl;

cout << "[0]Exit" << endl;

cout << "Enter Choice : ";

cin >> choice;

switch (choice)

{

case 0:

break;

case 1:

cout << "Enter value : ";

cin >> data;

bst.Insert(data);

break;

case 2:

cout << "Enter value : ";

cin >> data;

bst.DeleteKey(data);

break;

case 3:

cout << "Enter value : ";

cin >> data;

if (bst.SearchKey(data))

{

cout << "Data is Present" << endl;

}

else

{

cout << "Data is not Present" << endl;

}

break;

case 4:

cout << "The maximun value is : " << bst.findMax() << endl;

break;

case 5:

cout << "The minimum value is : " << bst.findMin() << endl;

break;

case 6:

bst.inorderTraversal();

cout << endl;

break;

case 7:

bst.preorderTraversal();

cout << endl;

break;

case 8:

bst.postorderTraversal();

cout << endl;

break;

case 9:

cout << "The height of the tree is : " << bst.treeHeight() << endl;

break;

case 10:

cout << "There are " << bst.treeNodeCount() << " nodes in the tree" << endl;

break;

case 11:

cout << "There are " << bst.treeLeavesCount() << " leaf nodes in the tree" << endl;

break;

case 12:

cout << "Enter value : ";

cin >> data;

cout << "The node hieght is :";

bst.printNodeLevel(data);

break;

default:

cout << "Invalid Entry" << endl;

break;

}

system("pause");

system("cls");

}

}

**TASK 2:**

#include<iostream>

using namespace std;

template<class T>

class Node{

public:

T data;

Node<T>\* left;

Node<T>\* right;

Node(){

left = right = NULL;

}

};

template<class T>

class binarySearchTree{

Node<T>\* root;

void insert(T data, Node<T>\*& ptr){ //function to insert an element in the tree

if (ptr == NULL) { //if empty and nothing else exist

ptr = new Node<T>;

ptr->data = data;

ptr->left = NULL;

ptr->right = NULL;

}

else if (ptr->data > data){

insert(data, ptr->left);

}

else if (ptr->data < data){

insert(data, ptr->right);

}

}

void deleteNode(T data, Node<T>\*& ptr){

if (ptr == NULL){ // node does not exist in the tree

cout << data << "not found.\n";

}

else if (data < ptr->data){

deleteNode(data, ptr->left); // find in left subtree

}

else if (data > ptr->data){

deleteNode(data, ptr->right); // find in right subtree

}

else{

makeDeletion(ptr); // actually deletes node from BST

}

}

void makeDeletion(Node<T>\*& node){

Node<T>\* temp; // Temperary pointer

if (node->right == NULL) { // case for leaf and one (left) child

temp = node;

node = node->left; // Reattach the left child

delete temp;

}

else if (node->left == NULL) { // case for one (right) child

temp = node;

node = node->right; // Reattach the right child

delete temp;

}

else{ // case for two children.

temp = node->right; // Move one node to the right

while (temp->left){ // Go to the extreme left node

temp = temp->left;

}

temp->left = node->left; // Reattach the left subtree

temp = node;

node = node->right; // Reattach the right subtree

delete temp;

}

}

void inorder(Node<T>\*& ptr){//prints in - order traversal of the tree

if (ptr){

inorder(ptr->left);

cout << ptr->data << " ";

inorder(ptr->right);

}

}

int Height(Node<T>\*& ptr) {

int hleft = 0, hright = 0;

if (ptr){

hleft = Height(ptr->left);

hright = Height(ptr->right);

if (hleft > hright){

return hleft + 1;

}

else{

return hright + 1;

}

}

else{

return 0;

}

}

void mirror(Node<T>\*& ptr1, Node<T>\*& ptr2){

if (ptr1){

ptr2 = new Node<T>;

ptr2->data = ptr1->data;

mirror(ptr1->left, ptr2->right);

mirror(ptr1->right, ptr2->left);

}

}

public:

binarySearchTree(){

root = NULL;

}

void Insert(T data){ //insert an element in the tree

insert(data, root);

}

void DeleteKey(T data){// delete an element in the tree

deleteNode(data, root);

}

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

inorder(root);

}

int treeHeight(){//returns the height of the tree recursively

return Height(root) - 1;

}

void Mirror(binarySearchTree<T>& bst1, binarySearchTree<T>& bst2){

mirror(bst1.root, bst2.root);

}

};

int main(){

binarySearchTree<int> bst1, bst2;

int arr[8] = { 5,3,7,1,2,6,8 };

for (int i = 0; i < 8; i++){

bst1.Insert(arr[i]);

}

cout << "Tree : ";

bst1.inorderTraversal();

cout << endl;

cout << "Mirror Tree : ";

bst1.Mirror(bst1, bst2);

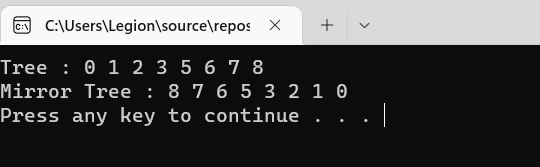
bst2.inorderTraversal();

cout << endl;

system("pause");

return 0;

}

****

**TASK 3:**

**CPP:**

#include<iostream>

#include "binarySearchTree.h"

using namespace std;

class TreeNode {

public:

int value;

TreeNode\* left, \* right;

};

TreeNode\* newNode(int value);

TreeNode\* sortedArrayToBST(int arr[], int start, int end) {

if (start > end) {

return NULL;

}

int mid = (start + end) / 2;

TreeNode\* nodePtr = newNode(arr[mid]);

nodePtr->left = sortedArrayToBST(arr, start, mid - 1);

nodePtr->right = sortedArrayToBST(arr, mid + 1, end);

return nodePtr;

}

TreeNode\* newNode(int value) {

TreeNode\* nodePtr = new TreeNode();

nodePtr->value = value;

nodePtr->left = NULL;

nodePtr->right = NULL;

return nodePtr;

}

void display(TreeNode\* nodePtr) {

if (nodePtr == NULL) {

return;

}

cout << nodePtr->value << " ";

display(nodePtr->left);

display(nodePtr->right);

}

int main() {

int n, arr[] = { 0,1,2,3,4,5,6 };

n = sizeof(arr) / sizeof(arr[0]);

TreeNode\* nodePtr = sortedArrayToBST(arr, 0, n - 1);

cout << "BST of a sorted array: ";

display(nodePtr);

cout << "\n\n";

system("pause");

return 0;

}

**HEADER FILE:**

#pragma once

#include<iostream>

using namespace std;

template<class T>

class TreeNode {

public:

T value;

TreeNode\* left, \* right;

TreeNode(T value, TreeNode\* left, TreeNode\* right);

TreeNode();

};

template<class BST>

class binareySearchTree {

private:

TreeNode<BST>\* nodePtr;

public:

void Insert(BST element);

BST DeleteKey(int element, TreeNode<BST>\*& nodePtr);

BST SearchKey(int element, TreeNode<BST>\* nodePtr);

BST findMax(TreeNode<BST>\* nodePtr);

BST findMin(TreeNode<BST>\* nodePtr);

BST inorderTraversal(TreeNode<BST>\* nodePtr);

BST preorderTraversal(TreeNode<BST>\* nodePtr);

BST postorderTraversal(TreeNode<BST>\* nodePtr);

BST treeHeight(TreeNode<BST>\* nodePtr);

BST treeNodeCount(TreeNode<BST>\* nodePtr, BST count = 0);

BST treeLeavesCount(TreeNode<BST>\* nodePtr);

BST printNodeLevel(TreeNode<BST>\* nodePtr, int value, int level);

TreeNode<BST>\* ret();

}

**CPP:**

#include"binarySearchTree.h"

#include<iostream>

using namespace std;

template<class T>

TreeNode<T>::TreeNode() {

value = 0;

left = right = NULL;

}

template<class T>

TreeNode<T>::TreeNode(T value, TreeNode\* left, TreeNode\* right) {

this->value = value;

this->left = NULL;

this->right = NULL;

}

template<class BST>

void binareySearchTree<BST> ::Insert(BST element) {

TreeNode<BST>\* newNode = new TreeNode<BST>;

newNode->value = element;

newNode->left = newNode->right = NULL;

if (!nodePtr) {

nodePtr = newNode;

}

else {

TreeNode<BST>\* nodePtr = nodePtr;

while (true) {

if (element < nodePtr->value) {

if (nodePtr->left != NULL) {

nodePtr = nodePtr->left;

}

else {

nodePtr->left = newNode;

return;

}

}

else if (element > nodePtr->value) {

if (nodePtr->right != NULL) {

nodePtr = nodePtr->right;

}

else {

cout << "Duplicate value found in teee.\n";

break;

}

}

}

}

}

template<class BST>

BST DeleteKey(int element, TreeNode<BST>\*& nodePtr) {

TreeNode\* tempNodePtr;

if (nodePtr == NULL) {

cout << element << " not found.\n";

}

else if (element < nodePtr->value) {

DeleteKey(element, nodePtr->left);

}

else if (element > nodePtr->right) {

DeleteKey(element, nodePtr->right);

}

else if (nodePtr->right == NULL) {

tempNodePtr = nodePtr;

nodePtr = nodePtr->left;

delete tempNodePtr;

}

else if (nodePtr->left == NULL) {

tempNodePtr = nodePtr;

nodePtr = nodePtr->right;

delete tempNodePtr;

}

else {

tempNodePtr = nodePtr->right;

21F - 9631DS Assignment 3BCS - 3C

while (tempNodePtr->left) {

tempNodePtr = tempNodePtr->left;

}

tempNodePtr->left = nodePtr->left;

tempNodePtr = nodePtr;

nodePtr = nodePtr->right;

delete tempNodePtr;

DeleteKey(nodePtr);

}

}

template<class BST>

BST SearchKey(int element, TreeNode<BST>\* nodePtr) {

if (nodePtr == NULL || nodePtr->value == element) {

return 1;

}

else if (nodePtr->value < element) {

return SearchKey(nodePtr->right, element);

return SearchKey(nodePtr->left, element);

return 0;

}

else {

cout << "Key not found.\n";

}

}

template<class BST>

BST findMax(TreeNode<BST>\* nodePtr) {

if (nodePtr == NULL) {

return max;

}

int data = nodePtr->value;

int leftT = findMax(nodePtr->left);

int rightT = findMax(nodePtr->right);

if (leftT > data) {

data = leftT;

}

if (rightT > data) {

data = rightT;

}

return data;

}

template<class BST>

BST findMin(TreeNode<BST>\* nodePtr) {

if (nodePtr == NULL) {

return min;

}

int data = nodePtr->value;

int leftT = findMin(nodePtr->left);

int rightT = findMin(nodePtr->right);

if (leftT < data) {

data = leftT;

}

if (rightT < data) {

data = rightT;

}

return data;

21F - 9631DS Assignment 3BCS - 3C

}

template<class BST>

BST inorderTraversal(TreeNode<BST>\* nodePtr) {

if (nodePtr == NULL) {

return;

}

inorderTraversal(nodePtr->left);

cout << nodePtr->value << " ";

inorderTraversal(nodePtr->right);

}

template<class BST>

BST preorderTraversal(TreeNode<BST>\* nodePtr) {

if (nodePtr == NULL) {

return;

}

cout << nodePtr->value << " ";

preorderTraversal(nodePtr->left);

preorderTraversal(nodePtr->right);

}

template<class BST>

BST postorderTraversal(TreeNode<BST>\* nodePtr) {

if (nodePtr == NULL) {

return;

}

postorderTraversal(nodePtr->left);

postorderTraversal(nodePtr->right);

cout << nodePtr->value << " ";

}

template<class BST>

BST treeHeight(TreeNode<BST>\* nodePtr)

{

if (nodePtr != NULL) {

int leftSide = treeHeight(nodePtr->left);

int rightSide = treeHeight(nodePtr->right);

if (leftSide > rightSide) {

return(leftSide + 1);

}

else {

return (rightSide + 1);

}

}

}

template<class BST>

BST treeNodeCount(TreeNode<BST>\* nodePtr, BST count = 0) {

if (nodePtr != NULL) {

treeNodeCount(nodePtr->left);

count++;

treeNodeCount(nodePtr->right);

}

return count;

}

template<class BST>

BST treeLeavesCount(TreeNode<BST>\*nodePtr) {

if (nodePtr == NULL) {

return 0;

}

if (nodePtr->left == NULL && nodePtr->right == NULL) {

return 1;

}

else {

return treeLeavesCount(nodePtr->left) + treeLeavesCount(nodePtr->right);

}

}

template<class BST>

BST printNodeLevel(TreeNode<BST>\* nodePtr, int value, int level) {

if (nodePtr == NULL) {

return 0;

}

if (nodePtr->value == value) {

return level;

}

int downLevel = printNodeLevel(nodePtr->left, value, level + 1);

if (downLevel != 0) {

return downLevel;

}

downLevel = printNodeLevel(nodePtr->left, value, level + 1);

return downLevel;

return printNodeLevel(nodePtr, value, 1);

}

template<class BST>

TreeNode<BST>\* ret()

{

return nodePtr;

}

**TASK 4(A):**

#include<iostream>

using namespace std;

class BBT {

public:

int value;

BBT\* left, \* right;

};

BBT\* newNode(int value) {

BBT\* newNode = new BBT;

newNode->value = value;

newNode->left = newNode->right = NULL;

return newNode;

}

BBT\* insert(BBT\* node, int key) {

if (node == NULL)

return newNode(key);

if (key < node->value)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

int addBBT(BBT\* nodePtr) {

if (nodePtr == NULL) {

return 0;

}

return(nodePtr->value + addBBT(nodePtr->left) + addBBT(nodePtr->right));

}

int main() {

BBT\* root = NULL;

int choice;

int n;

do {

cout << "1- Insert" << endl;

cout << "2- Sum of all nodes" << endl;

cout << "3- Exit" << endl;

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the data to insert: ";

cin >> n;

root = insert(root, n);

break;

case 2:

int sum = addBBT(root);

cout << "Sum of all the tree elements is: " << sum << endl;

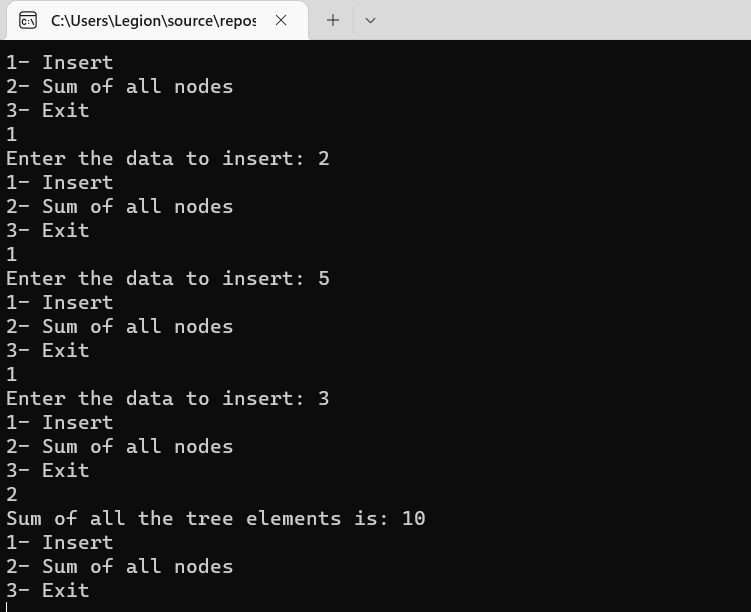
break;

}

} while (choice != 3);

return 0;

}

****

**TASK 4(B):**

#include<iostream>

using namespace std;

class BBT {

public:

int value;

BBT\* left, \* right;

};

BBT\* newNode(int value) {

BBT\* newNode = new BBT;

newNode->value = value;

newNode->left = newNode->right = NULL;

return newNode;

}

BBT\* insert(BBT\* node, int key) {

if (node == NULL)

return newNode(key);

if (key < node->value)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

int OR(BBT\* nodePtr, int& value) {

if (nodePtr != NULL) {

OR(nodePtr->left, value);

OR(nodePtr->right, value);

if (nodePtr->value + value == 1 || nodePtr->value + value == 2) {

value = 1;

}

else if (nodePtr->value + value == 0) {

value = 0;

}

return value;

}

}

int AND(BBT\* nodePtr, int& value) {

if (nodePtr != NULL) {

AND(nodePtr->left, value);

AND(nodePtr->right, value);

if (nodePtr->value \* value == 1) {

value = 1;

}

else if (nodePtr->value \* value == 0) {

value = 0;

}

return value;

}

}

int XOR(BBT\* nodePtr, int& value) {

if (nodePtr != NULL) {

XOR(nodePtr->left, value);

XOR(nodePtr->right, value);

if (nodePtr->value != value) {

value = 0;

}

else {

value = 1;

}

return value;

}

}

int main() {

BBT\* root = NULL;

int orgate = 0, andgate = 0, xorgate = 0;

int choice;

int n;

do {

cout << "1- Insert" << endl;

cout << "2- OR Gate" << endl;

cout << "3- AND Gate" << endl;

cout << "4- XOR Gate" << endl;

cout << "5- Exit" << endl;

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the data to insert: ";

cin >> n;

root = insert(root, n);

break;

case 2:

orgate = OR(root, orgate);

cout << "OR gate: " << orgate << endl;

break;

case 3:

andgate = AND(root, andgate);

cout << "AND gate: " << andgate << endl;

break;

case 4:

xorgate = XOR(root, xorgate);

cout << "XOR gate: " << xorgate << endl;

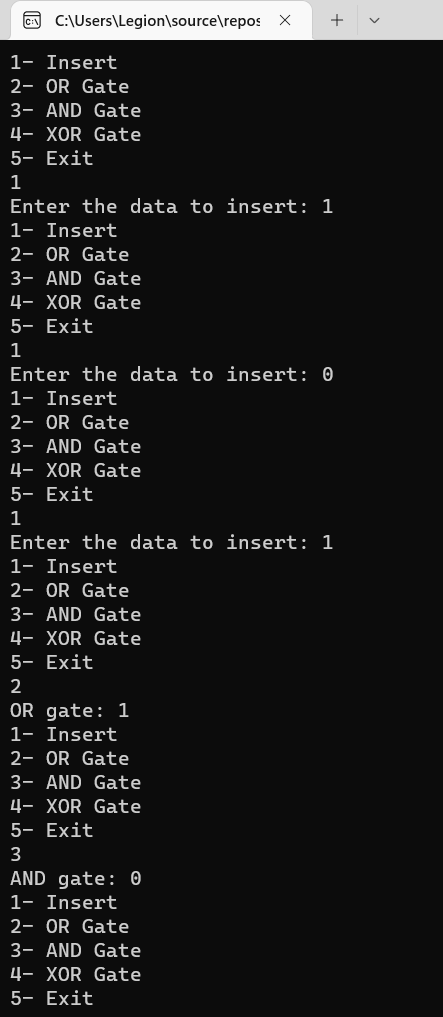
break;

}

} while (choice != 5);

return 0;

}

****

**Text

Description automatically generated**

**QUESTION 5:**

**TASK 1:**

#include<iostream>

using namespace std;

template<class T>

class Node{

public:

T data;

Node<T>\* left= NULL;

Node<T>\* right=NULL;

};

template<class T>

class binarySearchTree{

Node<T>\* root;

void insertintree(T data, Node<T>\*& ptr){ //insert an element in the tree

if (ptr == NULL) {

ptr = new Node<T>;

ptr->data = data;

ptr->left = NULL;

ptr->right = NULL;

}

else if (ptr->data > data){

insertintree(data, ptr->left);

}

else if (ptr->data < data){

insertintree(data, ptr->right);

}

else {

cout << "Node not inserted!" << endl;

}

}

void deletion(T data, Node<T>\*& ptr){

if (ptr == NULL){ // node does not exist in the tree

cout << data << "not found.\n";

}

else if (data < ptr->data){

deletion(data, ptr->left); // find in left subtree

}

else if (data > ptr->data){

deletion(data, ptr->right); // find in right subtree

}

else{

makeDeletion(ptr); // actually deletes node from BST

}

}

void makeDeletion(Node<T>\*& node){

Node<T>\* temp; // Temperary pointer

if (node->right == NULL) { // case for leaf and one (left) child

temp = node;

node = node->left; // Reattach the left child

delete temp;

}

else if (node->left == NULL) { // case for one (right) child

temp = node;

node = node->right; // Reattach the right child

delete temp;

}

else{ // case for two children.

temp = node->right; // Move one node to the right

while (temp->left){ // Go to the extreme left node

temp = temp->left;

}

temp->left = node->left; // Reattach the left subtree

temp = node;

node = node->right; // Reattach the right subtree

delete temp;

}

}

bool equal(Node<T>\*& ptr1, Node<T>\* const ptr2){

if (ptr1 && ptr2){

if (ptr1->data != ptr2->data){

return false;

}

else if (equal(ptr1->left, ptr2->left) && equal(ptr1->right, ptr2->right)){

return true;

}

else {

return false;

}

}

return true;

}

public:

binarySearchTree(){

root = NULL;

}

void Insert(T data){ //insert an element in the tree

insertintree(data, root);

}

void DeleteKey(T data){// delete an element in the tree

deletion(data, root);

}

bool operator == (const binarySearchTree<T>& bst) {

if (equal(root, bst.root)){

return true;

}

else{

return false;

}

}

};

template<class T>

void permutations(binarySearchTree<T>& bst1, T arr[], int size, int l, int& count){

// Base case

if (l == size){

binarySearchTree<T> bst2;

for (int i = 0; i < size + 1; i++){

bst2.Insert(arr[i]);

}

if (bst1 == bst2){

cout << "{";

for (int i = 0; i < size + 1; i++){

cout << arr[i] << ",";

}

cout << "}" << endl;

count++;

}

for (int i = 0; i < size + 1; i++){

bst2.DeleteKey(arr[i]);

}

}

else{

for (int i = l; i <= size; i++){

swap(arr[l], arr[i]); // Function to swap

permutations(bst1, arr, size, l + 1, count); // Recursion called

swap(arr[l], arr[i]);

}

}

}

int main(){

binarySearchTree<int> bst;

int\* arr = NULL, size, count = 0;

cout << " Enter the array size " << endl;

cin >> size;

arr = new int[size];

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

cout << "ARRAY[" << i << "] : ";

cin >> arr[i];

bst.Insert(arr[i]);

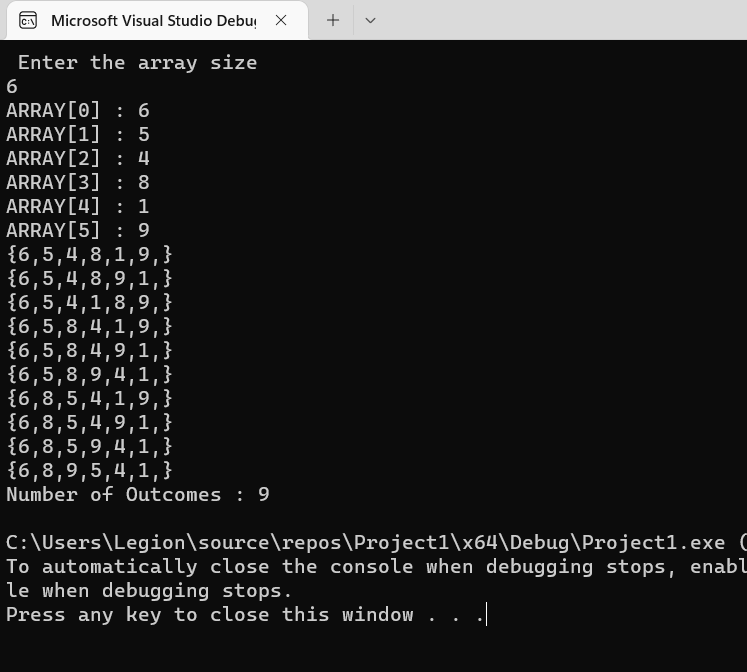
}

permutations(bst, arr, size - 1, 0, count);

cout << "Number of Outcomes : " << count - 1 << endl;//as count counts all outcoms

delete[]arr;

}

****

**TASK 2:**

#include<iostream>

using namespace std;

template<class T>

class Node{

public:

T data;

Node<T>\* left= NULL;

Node<T>\* right=NULL;

};

template<class T>

class binarySearchTree{

Node<T>\* root;

void insertintree(T data, Node<T>\*& ptr){ //insert an element in the tree

if (ptr == NULL) {

ptr = new Node<T>;

ptr->data = data;

ptr->left = NULL;

ptr->right = NULL;

}

else if (ptr->data > data){

insertintree(data, ptr->left);

}

else if (ptr->data < data){

insertintree(data, ptr->right);

}

else {

cout << "Node not inserted!" << endl;

}

}

void deletion(T data, Node<T>\*& ptr){

if (ptr == NULL){ // node does not exist in the tree

cout << data << "not found.\n";

}

else if (data < ptr->data){

deletion(data, ptr->left); // find in left subtree

}

else if (data > ptr->data){

deletion(data, ptr->right); // find in right subtree

}

else{

makeDeletion(ptr); // actually deletes node from BST

}

}

void makeDeletion(Node<T>\*& node){

Node<T>\* temp; // Temperary pointer

if (node->right == NULL) { // case for leaf and one (left) child

temp = node;

node = node->left; // Reattach the left child

delete temp;

}

else if (node->left == NULL) { // case for one (right) child

temp = node;

node = node->right; // Reattach the right child

delete temp;

}

else{ // case for two children.

temp = node->right; // Move one node to the right

while (temp->left){ // Go to the extreme left node

temp = temp->left;

}

temp->left = node->left; // Reattach the left subtree

temp = node;

node = node->right; // Reattach the right subtree

delete temp;

}

}

bool equal(Node<T>\*& ptr1, Node<T>\* const ptr2){

if (ptr1 && ptr2){

if (ptr1->data != ptr2->data){

return false;

}

else if (equal(ptr1->left, ptr2->left) && equal(ptr1->right, ptr2->right)){

return true;

}

else {

return false;

}

}

return true;

}

public:

binarySearchTree(){

root = NULL;

}

void Insert(T data){ //insert an element in the tree

insertintree(data, root);

}

void DeleteKey(T data){// delete an element in the tree

deletion(data, root);

}

bool operator == (const binarySearchTree<T>& bst) {

if (equal(root, bst.root)){

return true;

}

else{

return false;

}

}

};

template<class T>

void permutations(binarySearchTree<T>& bst1, T arr[], int size, int l, int& count){

// Base case

if (l == size){

binarySearchTree<T> bst2;

for (int i = 0; i < size + 1; i++){

bst2.Insert(arr[i]);

}

if (bst1 == bst2){

cout << "{";

for (int i = 0; i < size + 1; i++){

cout << arr[i] << ",";

}

cout << "}" << endl;

count++;

}

for (int i = 0; i < size + 1; i++){

bst2.DeleteKey(arr[i]);

}

}

else{

for (int i = l; i <= size; i++){

swap(arr[l], arr[i]); // Function to swap

permutations(bst1, arr, size, l + 1, count); // Recursion called

swap(arr[l], arr[i]);

}

}

}

int main(){

binarySearchTree<int> bst;

int\* arr = NULL, size, count = 0;

cout << " Enter the array size " << endl;

cin >> size;

arr = new int[size];

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

cout << "ARRAY[" << i << "] : ";

cin >> arr[i];

bst.Insert(arr[i]);

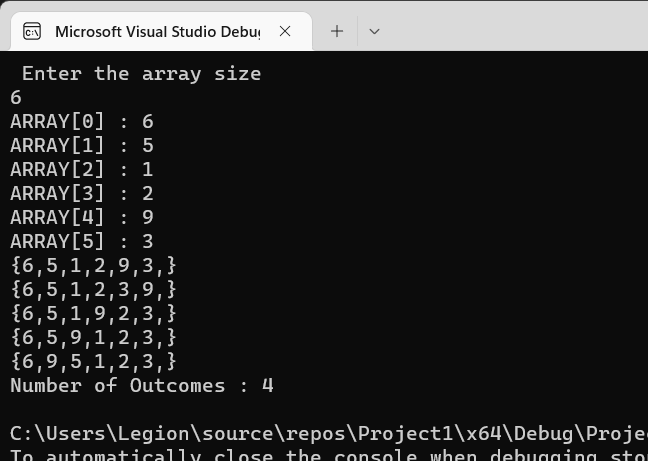
}

permutations(bst, arr, size - 1, 0, count);

cout << "Number of Outcomes : " << count - 1 << endl;//as count counts all outcoms

delete[]arr;

}

****