# Applied Programming



# Name: M. Hassan Sattar

# Roll No: 22F-3773

**Section: MS-1A**

# Submitted to: Sir. Muhammad Yousaf

**TASK-1**

**Consider the following functions to implement the Tree ADT.**

**Insert(): insert an element in the tree**

**DeleteKey(): delete an element in the tree  ,**

**SearchKey(): searches the desired element in the tree recursively**

**findMax(): finds the maximum element in the tree recursively**

**findMin(): finds the minimum element in the tree recursively**

**inorderTraversal() : prints in-order traversal of the tree**

**preorderTraversal(): prints pre-order traversal of the tree**

**postorderTraversal(): prints post-order traversal of the tree**

**treeHeight(): returns the height of the tree recursively**

**treeNodeCount(): returns the count of nodes in the tree**

**treeLeavesCount(): returns the count of leaves in the tree**

**printNodeLevel(): prints level of a node in the tree  The class will be created in a file “binarySearchTree.h” and the functions will be implemented in “binarySearchTree.cpp”.**

**Make sure to create the class as a template so it can run for multiple data types. Your main function should like: int main(){ // Sample Input: 65 55 22 44 61 19 90 10 78 52 BST myIntBST; // test all functions here**

#include <iostream>

using namespace std;

template <class T>

struct Node

{

T value;

Node\* left, \* right;

};

template <class T>

class BST

{

private:

Node<T>\* tree;

void preorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

cout << node->value << ' ';

preorderTraversal(node->left);

preorderTraversal(node->right);

}

void inorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

preorderTraversal(node->left);

cout << node->value << ' ';

preorderTraversal(node->right);

}

void postorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

preorderTraversal(node->left);

preorderTraversal(node->right);

cout << node->value << ' ';

}

int findMax(Node<T>\* node)

{

if (node->right == NULL)

return node->value;

findMax(node->right);

}

int findMin(Node<T>\* node)

{

if (node->left == NULL)

return node->value;

findMin(node->left);

}

Node<T>\* DeleteKey(Node<T>\* node, T value)

{

if (node == NULL)

return NULL;

if (value < node->value)

node->left = DeleteKey(node->left, value);

else if (value > node->value)

node->right = DeleteKey(node->right, value);

else

{

if (node->left == NULL)

{

Node<T>\* temp = node->right;

delete node;

return temp;

}

else if (node->right == NULL)

{

Node<T>\* temp = node->left;

delete node;

return temp;

}

Node<T>\* temp = node->right;

while (temp->left != NULL)

temp = temp->left;

node->value = temp->value;

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

}

return node;

}

bool SearchKey(Node<T>\* node, T value)

{

if (node == NULL)

return false;

if (value > node->value)

return SearchKey(node->right, value);

else if (value < node->value)

return SearchKey(node->left, value);

else

return true;

}

int treeHeight(Node<T>\* node)

{

if (node == NULL)

return 0;

else

{

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

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

if (left\_height > right\_height)

return left\_height + 1;

else

return right\_height + 1;

}

}

int treeNodeCount(Node<T>\* node)

{

if (node == NULL)

return 0;

return 1 + treeNodeCount(node->left) + treeNodeCount(node->right);

}

int treeLeavesCount(Node<T>\* node)

{

if (node == NULL)

return 0;

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

return 1;

else

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

}

void printNodeOnEachLevel(Node<T>\* node, int level)

{

if (node == NULL)

return;

if (level == 1)

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

else if (level > 1)

{

printNodeOnEachLevel(node->left, level - 1);

printNodeOnEachLevel(node->right, level - 1);

}

}

void printNodeLevel(Node<T>\* node)

{

int height = treeHeight(node);

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

{

printNodeOnEachLevel(node, i);

cout << endl;

}

}

public:

BST() : tree(NULL) {}

void Insert(T value)

{

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

node->value = value;

node->left = node->right = NULL;

if (tree == NULL)

{

tree = node;

}

else

{

Node<T>\* temp = tree;

Node<T>\* parent = NULL;

// cout << "DEBUG;" << endl;

while (temp != NULL)

{

if (value < temp->value)

{

// cout << "LEFT;" << endl;

parent = temp;

temp = temp->left;

}

else if (value > temp->value)

{

// cout << "RIGHT;" << endl;

parent = temp;

temp = temp->right;

}

else

{

cout << "Node already Exists...\n";

return;

}

}

if (value < parent->value)

parent->left = node;

else if (value > parent->value)

parent->right = node;

}

}

int findMax()

{

int result = findMax(tree);

return result;

}

int findMin()

{

int result = findMin(tree);

return result;

}

void preorderTraversal()

{

preorderTraversal(tree);

}

void inorderTraversal()

{

inorderTraversal(tree);

}

void postorderTraversal()

{

postorderTraversal(tree);

}

void DeleteKey(T value)

{

DeleteKey(tree, value);

}

void SearchKey(T value)

{

if (SearchKey(tree, value) == true)

{

cout << "Node with value[" << value << "], has been found" << endl;

}

else

{

cout << "Node with value[" << value << "], not found" << endl;

}

}

int treeHeight()

{

int height = treeHeight(tree);

return height;

}

int treeNodeCount()

{

int nodes = treeNodeCount(tree);

return nodes;

}

int treeLeavesCount()

{

int leaves = treeLeavesCount(tree);

return leaves;

}

void printNodeLevel()

{

printNodeLevel(tree);

}

};

int main()

{

BST<int> bst; // 65 55 22 44 61 19 90 10 78 52

bst.Insert(65);

bst.Insert(55);

bst.Insert(22);

bst.Insert(44);

bst.Insert(61);

bst.Insert(19);

bst.Insert(90);

bst.Insert(10);

bst.Insert(78);

bst.Insert(52);

bst.DeleteKey(10);

bst.DeleteKey(65);

bst.DeleteKey(100); // This value doesn't exist

bst.SearchKey(55);

bst.SearchKey(90);

bst.SearchKey(100); // This value doesn't exist

cout << "findMax(): " << bst.findMax() << endl;

cout << "findMin(): " << bst.findMin() << endl;

cout << "Pre Order: ";

bst.preorderTraversal();

cout << endl;

cout << "In Order: ";

bst.inorderTraversal();

cout << endl;

cout << "Post Order: ";

bst.postorderTraversal();

cout << endl;

cout << "treeHeight(): " << bst.treeHeight() << endl;

cout << "treeNodeCount(): " << bst.treeNodeCount() << endl;

cout << "treeLeavesCount(): " << bst.treeLeavesCount() << endl;

cout << "printNodeLevel(): " << endl;

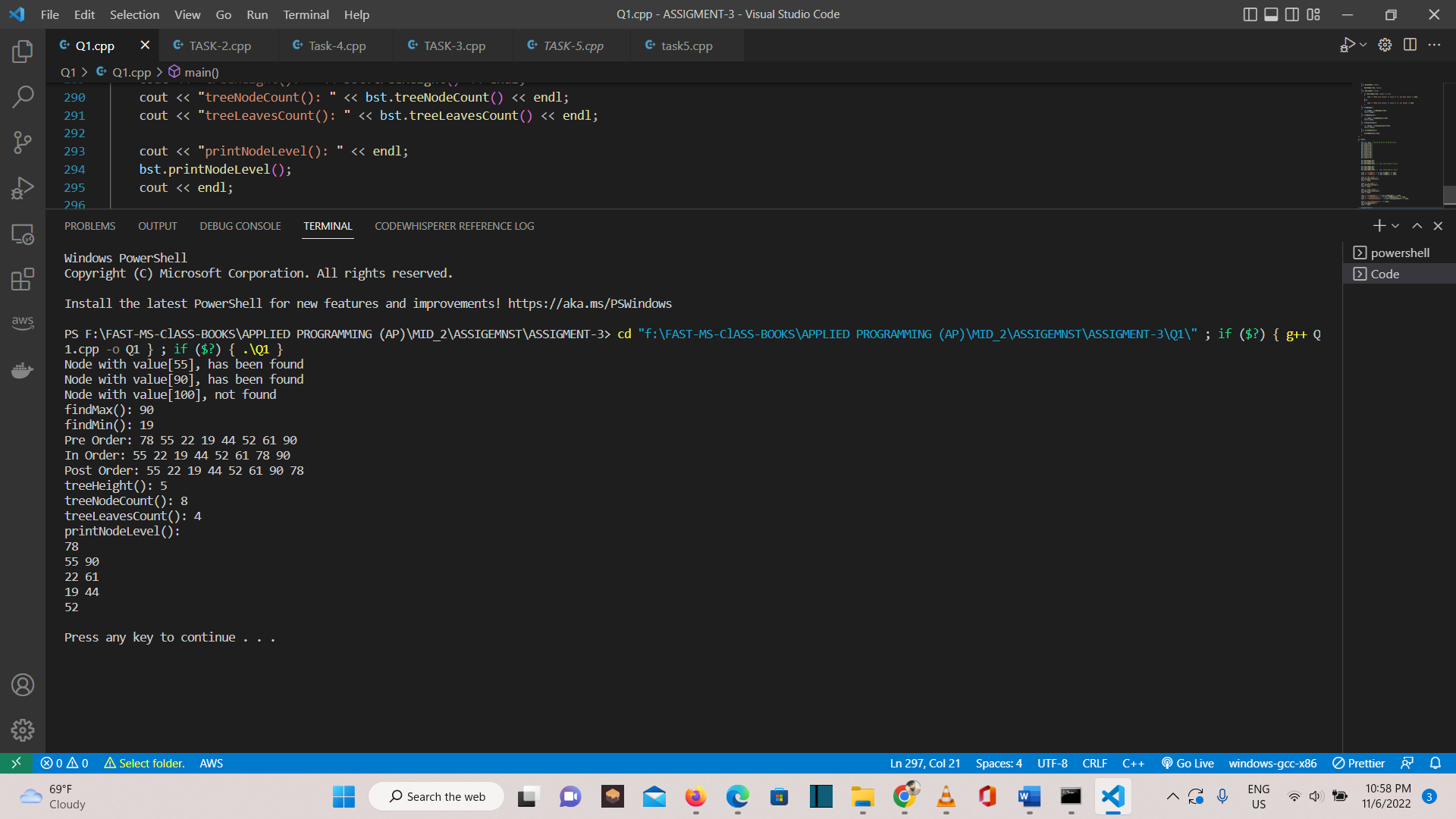
bst.printNodeLevel();

cout << endl;

system("PAUSE");

return 0;

}



**TASK-2**

**Write a recursive function that converts the given binary tree into its mirror tree. Hint: The mirror of a binary tree T is another binary Tree M(T) with left and right children of all non-leaf nodes interchanged. Following attached diagram depicts the mirror trees however, you are required to consider only non-leaf nodes for the implementation of this task**

#include <iostream>

using namespace std;

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data)

{

struct Node\* node = (struct Node\*)

malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

void mirror(struct Node\* node)

{

if (node == NULL)

return;

else

{

struct Node\* temp;

mirror(node->left);

mirror(node->right);

/\* swap the pointers in this node \*/

temp = node->left;

node->left = node->right;

node->right = temp;

}

}

void inOrder(struct Node\* node)

{

if (node == NULL)

return;

inOrder(node->left);

cout << node->data << " ";

inOrder(node->right);

}

// Driver Code

int main()

{

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

/\* Print inorder traversal of the input tree \*/

cout << "Inorder traversal of the constructed tree \n";

inOrder(root);

/\* Convert tree to its mirror \*/

mirror(root);

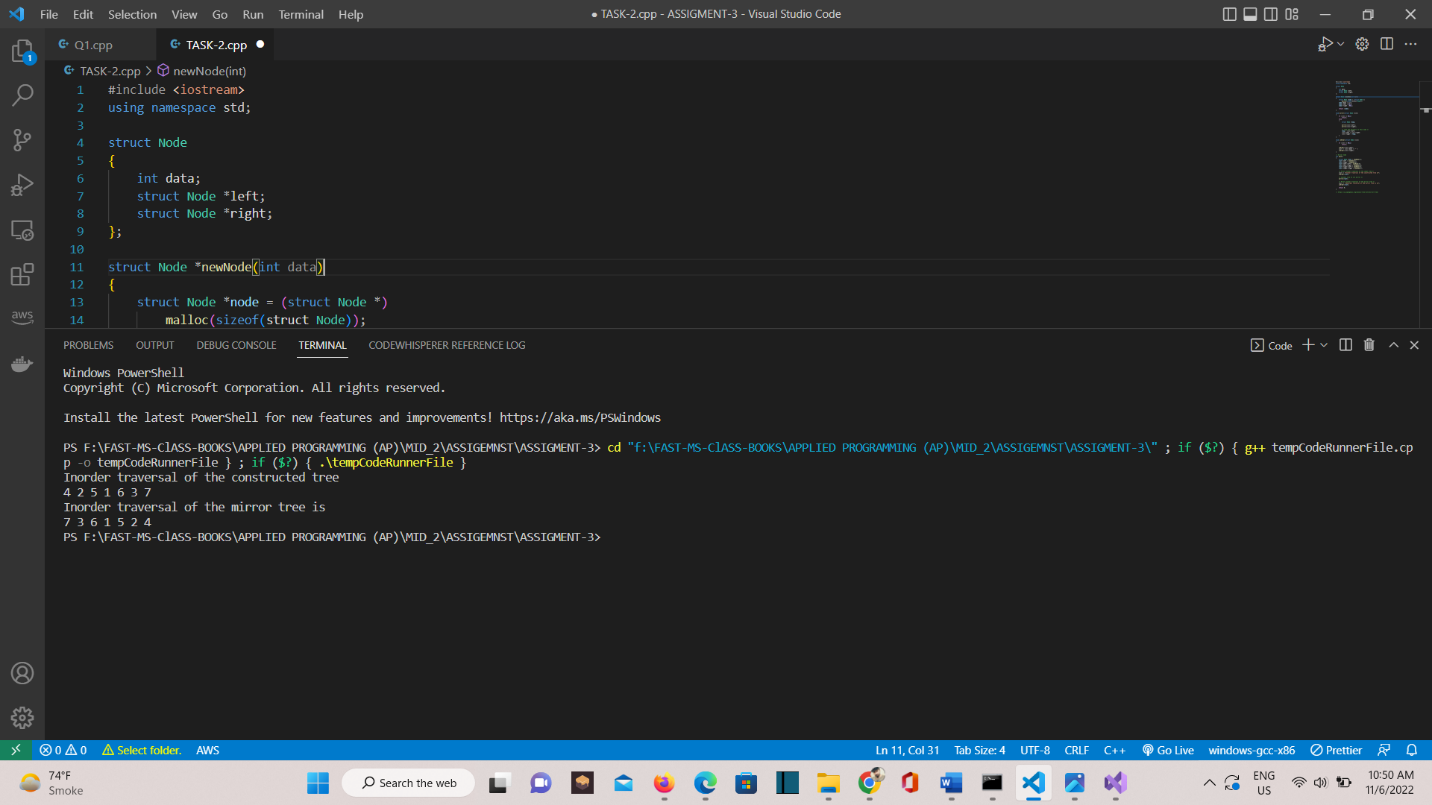
/\* Print inorder traversal of the mirror tree \*/

cout << "\nInorder traversal of the mirror tree is \n";

inOrder(root);

return 0;

}



**TASK-3**

**Convert a sorted array to a BST of minimal height (Use the library you implemented in task1)**

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* left;

Node\* right;

};

Node\* ADD\_NODE(int data);

class MirrorTree

{

public:

void MINIMNAL\_HEIGHT(int arr[], int start, int end);

void PRE\_ORDER(Node\* node);

};

Node\* MINIMNAL\_HEIGHT(int arr[], int start, int end)

{

if (start > end)

return NULL;

int mid = (start + end) / 2;

Node\* root = ADD\_NODE(arr[mid]);

root->left = MINIMNAL\_HEIGHT(arr, start, mid - 1);

root->right = MINIMNAL\_HEIGHT(arr, mid + 1, end);

return root;

}

Node\* ADD\_NODE(int data)

{

Node\* node = new Node();

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

void PRE\_ORDER(Node\* node)

{

if (node == NULL)

return;

cout << node->data << " ";

PRE\_ORDER(node->left);

PRE\_ORDER(node->right);

}

int main()

{

int arr[] = { 5, 6, 7, 8, 9, 10, 11 };

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

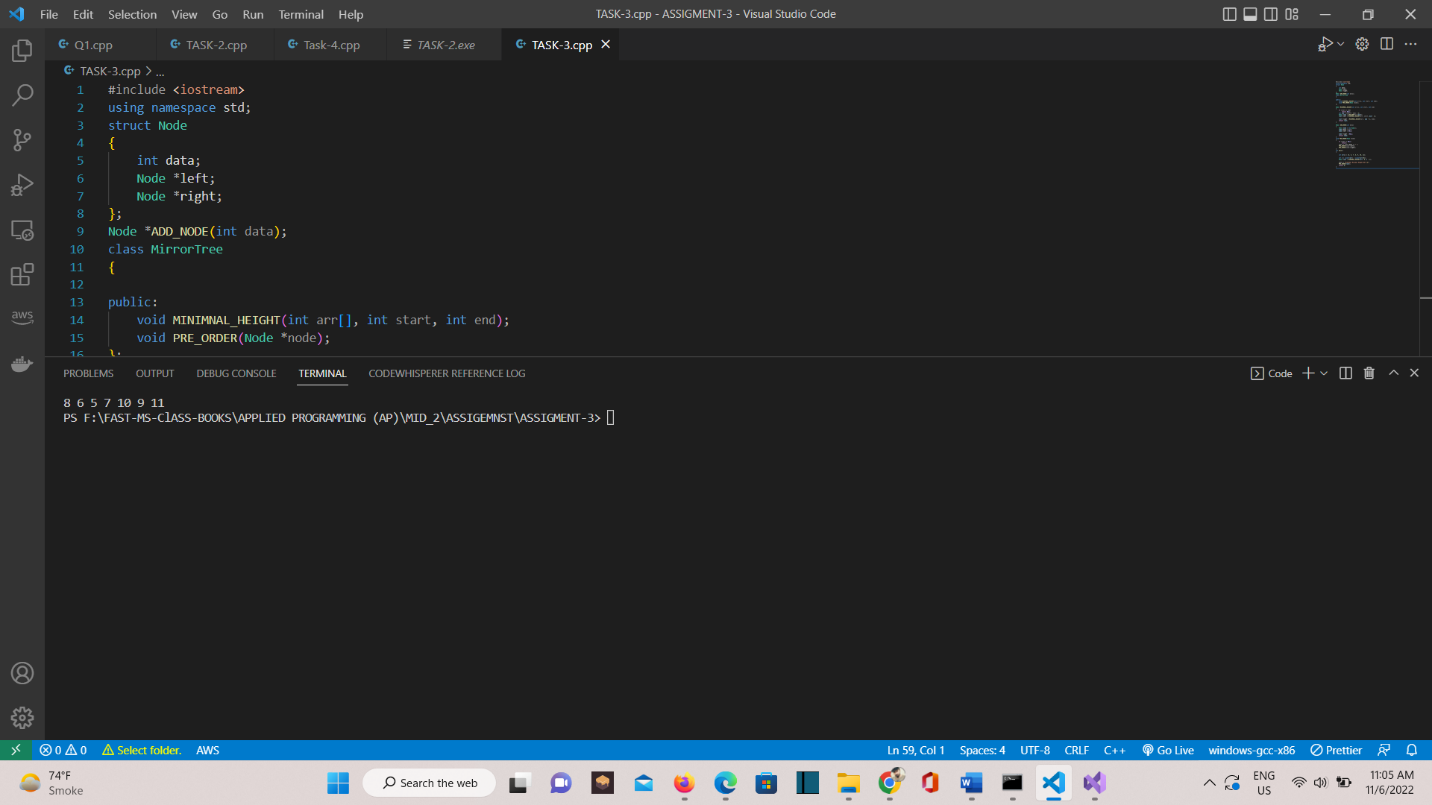
Node\* root = MINIMNAL\_HEIGHT(arr, 0, n - 1);

cout << "PreOrder of Minimal Height \n";

PRE\_ORDER(root);

return 0;

}



**TASK-4**

1. **Write a recursive function that will calculate the sum of all nodes given in binary tree**

#include <stdio.h>

#include <stdlib.h>

// node of binary tree

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* root = NULL;

struct Node\* ADD\_NODE(int data)

{

// Create a new node

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// CALCULATING\_SUM()

int CALCULATING\_SUM(struct Node\* temp)

{

int sum, sumLeft, sumRight;

sum = sumRight = sumLeft = 0;

// Check whether tree is empty

if (root == NULL)

{

printf("Tree is empty\n");

return 0;

}

else

{

if (temp->left != NULL)

sumLeft = CALCULATING\_SUM(temp->left);

if (temp->right != NULL)

sumRight = CALCULATING\_SUM(temp->right);

sum = temp->data + sumLeft + sumRight;

return sum;

}

}

int main()

{

// Add\_NODE nodes to the binary tree

root = ADD\_NODE(5);

root->left = ADD\_NODE(2);

root->right = ADD\_NODE(9);

root->left->left = ADD\_NODE(1);

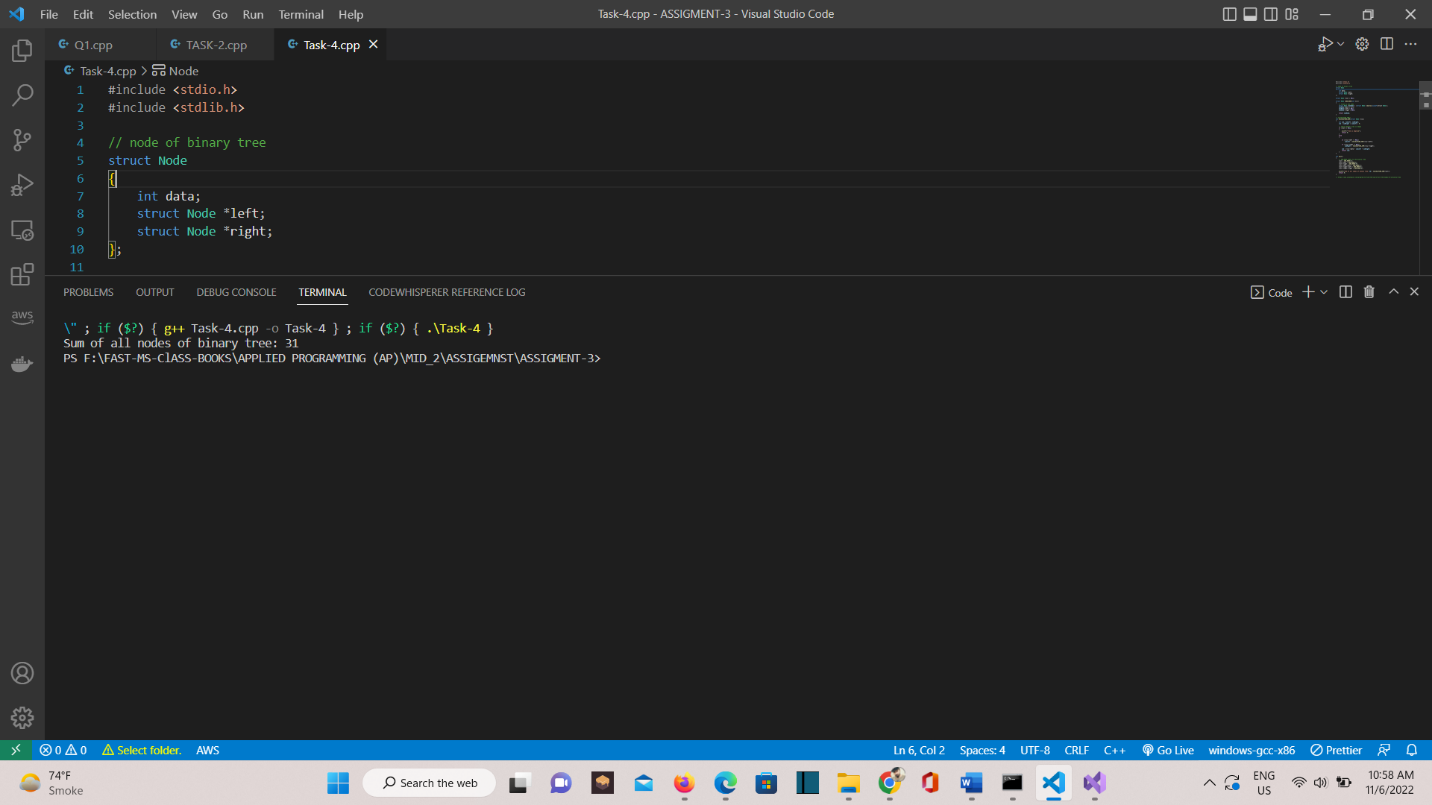
root->right->left = ADD\_NODE(8);

root->right->right = ADD\_NODE(6);

printf("Sum of all nodes of binary tree: %d", CALCULATING\_SUM(root));

return 0;

}



**TASK-5**

Given an array nums that represents a permutation of integers from 1 to n. We are going to construct a binary search tree (BST) by inserting the elements of nums in order into an initially empty BST. Find the number of different ways to reorder nums so that the constructed BST is identical to that formed from the original array nums. For example, given nums = [2,1,3], we will have 2 as the root, 1 as a left child, and 3 as a right child. The array [2,3,1] also yields the same BST but [3,2,1] yields a different BST. Return the number of ways to reorder nums such that the BST formed is identical to the original BST formed from nums.

Input: nums = [3,4,5,1,2]

Output: 5 [3,1,2,4,5] [3,1,4,2,5] [3,1,4,5,2] [3,4,1,2,5] [3,4,1,5,2]

#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[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[i - 1] \* i % mod;

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

}

};

int main()

{

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

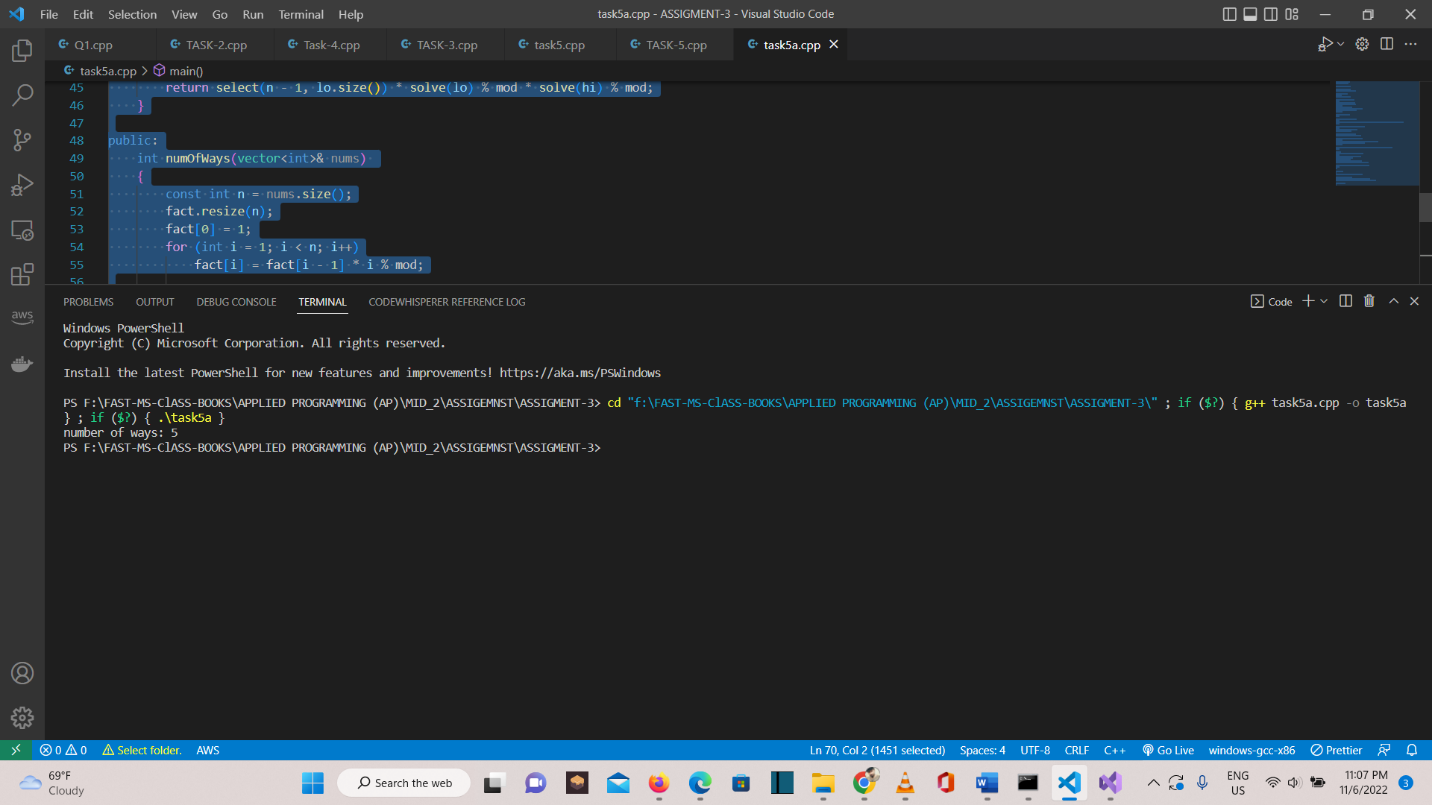
Solution solution;

int num\_of\_ways = solution.numOfWays(vec);

cout << "number of ways: " << num\_of\_ways << endl;

return 0;

}



**TASK-5b**

Given the root of a Binary Search Tree (BST), convert it to a Greater Tree such that every value of the original BST is changed to the original value plus the sum of all nodes greater than the current in BST. As a reminder, a binary search tree is a tree that satisfies these constraints:

● The left subtree of a node contains only nodes with values less than the node's value.

● The right subtree of a node contains only nodes with values greater than the node's value.

● Both the left and right subtrees must also be binary search trees

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* left, \* right;

};

struct Node\* ADD\_NODE(int data)

{

struct Node\* temp = new Node;

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

void BST\_GREATER(Node\* root, int\* sum)

{

if (root == NULL)

{

return;

}

BST\_GREATER(root->right, sum);

\*sum += root->data;

root->data = \*sum;

BST\_GREATER(root->left, sum);

}

void convertBST\_GREATER(Node\* root)

{

int sum = 0;

BST\_GREATER(root, &sum);

}

void InORDER(Node\* root)

{

if (root == NULL)

{

return;

}

InORDER(root->left);

cout << root->data << "-> ";

InORDER(root->right);

}

int main()

{

Node\* root = ADD\_NODE(5);

root->left = ADD\_NODE(2);

root->right = ADD\_NODE(7);

root->left->left = ADD\_NODE(1);

root->left->right = ADD\_NODE(3);

root->right->left = ADD\_NODE(6);

root->right->right = ADD\_NODE(8);

root->left->left->left = NULL;

root->left->right->right = ADD\_NODE(4);

root->right->right->left = NULL;

root->right->right->right = ADD\_NODE(9);

cout << "Binary Search Tree is:\n";

InORDER(root);

convertBST\_GREATER(root);

cout << "\n\nBST\_GREATER :\n";

InORDER(root);

return 0;

}

