Assignment 5 – Trees

Name: Kartik Banshi Katkar

Rollno : 29

Prn - 12111367

SY-IT-A

**Q.**

**Tree**

**A. Recursive**

1. **Inorder Tree Traversal**
2. **Preorder Tree Traversal**
3. **Postorder Tree Traversal**
4. **Implementation of various other functions**

**B. Non-Recursive traversal**

**Code:**

#include <bits/stdc++.h>

#define SPACE 10

using namespace std;

class TreeNode

{

public:

    int value;

    TreeNode \*left;

    TreeNode \*right;

    TreeNode()

    {

        value = 0;

        left = NULL;

        right = NULL;

    }

    TreeNode(int v)

    {

        value = v;

        left = NULL;

        right = NULL;

    }

};

class BST

{

public:

    TreeNode \*root;

    BST()

    {

        root = NULL;

    }

    bool isTreeEmpty()

    {

        if (root == NULL)

        {

            return true;

        }

        else

        {

            return false;

        }

    }

    void insertNode(TreeNode \*new\_node)

    {

        if (root == NULL)

        {

            root = new\_node;

            cout << "Value Inserted as root node!" << '\n';

        }

        else

        {

            TreeNode \*temp = root;

            while (temp != NULL)

            {

                // if (new\_node -> value == temp -> value) {

                // cout << "Value Already exist," <<

                // "Insert another value!" << '\n';

                // return;

                // } else

                if ((new\_node->value < temp->value) && (temp->left == NULL))

                {

                    temp->left = new\_node;

                    cout << "Value Inserted to the left!" << '\n';

                    break;

                }

                else if (new\_node->value < temp->value)

                {

                    temp = temp->left;

                }

                else if ((new\_node->value > temp->value) && (temp->right == NULL))

                {

                    temp->right = new\_node;

                    cout << "Value Inserted to the right!" << '\n';

                    break;

                }

                else

                {

                    temp = temp->right;

                }

            }

        }

    }

    TreeNode \*insertRecursive(TreeNode \*r, TreeNode \*new\_node)

    {

        if (r == NULL)

        {

            r = new\_node;

            cout << "Insertion successful" << '\n';

            return r;

        }

        if (new\_node->value < r->value)

        {

            r->left = insertRecursive(r->left, new\_node);

        }

        else if (new\_node->value > r->value)

        {

            r->right = insertRecursive(r->right, new\_node);

        }

        else

        {

            cout << "No duplicate values allowed!" << '\n';

            return r;

        }

        return r;

    }

    void print2D(TreeNode \*r, int space)

    {

        if (r == NULL) // Base case 1

            return;

        space += SPACE;           // Increase distance between levels 2

        print2D(r->right, space); // Process right child first 3

        cout << '\n';

        for (int i = SPACE; i < space; i++) // 5

            cout << " ";                    // 5.1

        cout << r->value << "\n";           // 6

        print2D(r->left, space);            // Process left child 6

    }

    void printPreorder(TreeNode \*r) //(current node, Left, Right)

    {

        if (r == NULL)

            return;

        /\* first print data of node \*/

        cout << r->value << " ";

        /\* then recur on left sutree \*/

        printPreorder(r->left);

        /\* now recur on right subtree \*/

        printPreorder(r->right);

    }

    /\* Iterative function for inorder tree

   traversal \*/

    void inOrder(TreeNode \*r)

    {

        stack<TreeNode \*> s;

        TreeNode \*curr = r;

        while (curr != NULL || s.empty() == false)

        {

            /\* Reach the left most Node of the

               curr Node \*/

            while (curr != NULL)

            {

                /\* place pointer to a tree node on

                   the stack before traversing

                  the node's left subtree \*/

                s.push(curr);

                curr = curr->left;

            }

            /\* Current must be NULL at this point \*/

            curr = s.top();

            s.pop();

            cout << curr->value << " ";

            /\* we have visited the node and its

               left subtree.  Now, it's right

               subtree's turn \*/

            curr = curr->right;

        } /\* end of while \*/

    }

    void preorderstack(TreeNode \*r)

    {

        if (r == NULL)

            return;

        stack<TreeNode \*> st;

        // start from root node (set current node to root node)

        TreeNode \*curr = r;

        // run till stack is not empty or current is

        // not NULL

        while (!st.empty() || curr != NULL)

        {

            // Print left children while exist

            // and keep pushing right into the

            // stack.

            while (curr != NULL)

            {

                cout << curr->value << " ";

                if (curr->right)

                    st.push(curr->right);

                curr = curr->left;

            }

            // We reach when curr is NULL, so We

            // take out a right child from stack

            if (st.empty() == false)

            {

                curr = st.top();

                st.pop();

            }

        }

    }

    void postOrderStack(TreeNode \*root)

    {

        if (root == NULL)

            return;

        // Create two stacks

        stack<TreeNode \*> s1, s2;

        // push root to first stack

        s1.push(root);

        TreeNode \*node;

        // Run while first stack is not empty

        while (!s1.empty())

        {

            // Pop an item from s1 and push it to s2

            node = s1.top();

            s1.pop();

            s2.push(node);

            // Push left and right children

            // of removed item to s1

            if (node->left)

                s1.push(node->left);

            if (node->right)

                s1.push(node->right);

        }

        // Print all elements of second stack

        while (!s2.empty())

        {

            node = s2.top();

            s2.pop();

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

        }

    }

    void printInorder(TreeNode \*r) // (Left, current node, Right)

    {

        if (r == NULL)

            return;

        /\* first recur on left child \*/

        printInorder(r->left);

        /\* then print the data of node \*/

        cout << r->value << " ";

        /\* now recur on right child \*/

        printInorder(r->right);

    }

    void printPostorder(TreeNode \*r) //(Left, Right, Root)

    {

        if (r == NULL)

            return;

        // first recur on left subtree

        printPostorder(r->left);

        // then recur on right subtree

        printPostorder(r->right);

        // now deal with the node

        cout << r->value << " ";

    }

    TreeNode \*iterativeSearch(int v)

    {

        if (root == NULL)

        {

            return root;

        }

        else

        {

            TreeNode \*temp = root;

            while (temp != NULL)

            {

                if (v == temp->value)

                {

                    return temp;

                }

                else if (v < temp->value)

                {

                    temp = temp->left;

                }

                else

                {

                    temp = temp->right;

                }

            }

            return NULL;

        }

    }

    TreeNode \*recursiveSearch(TreeNode \*r, int val)

    {

        if (r == NULL || r->value == val)

            return r;

        else if (val < r->value)

            return recursiveSearch(r->left, val);

        else

            return recursiveSearch(r->right, val);

    }

    int height(TreeNode \*r)

    {

        if (r == NULL)

            return -1;

        else

        {

            /\* compute the height of each subtree \*/

            int lheight = height(r->left);

            int rheight = height(r->right);

            /\* use the larger one \*/

            if (lheight > rheight)

                return (lheight + 1);

            else

                return (rheight + 1);

        }

    }

    /\* Print nodes at a given level \*/

    void printGivenLevel(TreeNode \*r, int level)

    {

        if (r == NULL)

            return;

        else if (level == 0)

            cout << r->value << " ";

        else // level > 0

        {

            printGivenLevel(r->left, level - 1);

            printGivenLevel(r->right, level - 1);

        }

    }

    void printLevelOrderBFS(TreeNode \*r)

    {

        int h = height(r);

        for (int i = 0; i <= h; i++)

            printGivenLevel(r, i);

    }

    TreeNode \*minValueNode(TreeNode \*node)

    {

        TreeNode \*current = node;

        /\* loop down to find the leftmost leaf \*/

        while (current->left != NULL)

        {

            current = current->left;

        }

        return current;

    }

    TreeNode \*deleteNode(TreeNode \*r, int v)

    {

        // base case

        if (r == NULL)

        {

            return NULL;

        }

        // If the key to be deleted is smaller than the root's key,

        // then it lies in left subtree

        else if (v < r->value)

        {

            r->left = deleteNode(r->left, v);

        }

        // If the key to be deleted is greater than the root's key,

        // then it lies in right subtree

        else if (v > r->value)

        {

            r->right = deleteNode(r->right, v);

        }

        // if key is same as root's key, then This is the node to be deleted

        else

        {

            // node with only one child or no child

            if (r->left == NULL)

            {

                TreeNode \*temp = r->right;

                delete r;

                return temp;

            }

            else if (r->right == NULL)

            {

                TreeNode \*temp = r->left;

                delete r;

                return temp;

            }

            else

            {

                // node with two children: Get the inorder successor (smallest

                // in the right subtree)

                TreeNode \*temp = minValueNode(r->right);

                // Copy the inorder successor's content to this node

                r->value = temp->value;

                // Delete the inorder successor

                r->right = deleteNode(r->right, temp->value);

                // deleteNode(r->right, temp->value);

            }

        }

        return r;

    }

};

int main()

{

    BST obj;

    int option, val;

    do

    {

        cout << "What operation do you want to perform? " << '\n'

             << "Select Option number accordingly." << '\n';

        cout << "1. Insert Node" << '\n';

        cout << "2. Search Node" << '\n';

        cout << "3. Delete Node" << '\n';

        cout << "4. Traversal BST values through Recursion" << '\n';

        cout << "5. Print Height of the Tree" << '\n';

        cout << "6. Traversal BST values through Non Reacursive approach" << '\n';

        cout << "7. Clear Screen" << '\n';

        cout << "0. Exit Program" << '\n';

        cin >> option;

        // Node n1;

        TreeNode \*new\_node = new TreeNode();

        switch (option)

        {

        case 0:

            break;

        case 1:

            cout << "INSERT" << '\n';

            cout << "Enter VALUE of TREE NODE to INSERT in BST: ";

            cin >> val;

            new\_node->value = val;

            // obj.root= obj.insertRecursive(obj.root,new\_node);

            obj.insertNode(new\_node);

            cout << '\n';

            break;

        case 2:

            cout << "SEARCH" << '\n';

            cout << "Enter VALUE of TREE NODE to SEARCH in BST: ";

            cin >> val;

            // new\_node = obj.iterativeSearch(val);

            new\_node = obj.recursiveSearch(obj.root, val);

            if (new\_node != NULL)

            {

                cout << "Value found" << '\n';

            }

            else

            {

                cout << "Value NOT found" << '\n';

            }

            break;

        case 3:

            cout << "DELETE" << '\n';

            cout << "Enter VALUE of TREE NODE to DELETE in BST: ";

            cin >> val;

            new\_node = obj.iterativeSearch(val);

            if (new\_node != NULL)

            {

                obj.deleteNode(obj.root, val);

                cout << "Value Deleted" << '\n';

            }

            else

            {

                cout << "Value NOT found" << '\n';

            }

            break;

        case 4:

            cout << "2D PRINT: " << '\n';

            obj.print2D(obj.root, 5);

            cout << '\n';

            cout << "Print Level Order BFS: \n";

            obj.printLevelOrderBFS(obj.root);

            cout << '\n';

            cout << "PRE-ORDER: ";

            obj.printPreorder(obj.root);

            cout << '\n';

            cout << "IN-ORDER: ";

            obj.printInorder(obj.root);

            cout << '\n';

            cout << "POST-ORDER: ";

            obj.printPostorder(obj.root);

            cout << '\n';

            break;

        case 5:

            cout << "TREE HEIGHT" << '\n';

            cout << "Height : " << obj.height(obj.root) << '\n';

            break;

        case 6:

            cout << "PRE-ORDER: ";

            obj.preorderstack(obj.root);

            cout << '\n';

            cout << "IN-ORDER: ";

            obj.inOrder(obj.root);

            cout << '\n';

            cout << "POST-ORDER: ";

            obj.printPostorder(obj.root);

            cout << '\n';

            cout << '\n';

            break;

        case 7:

            system("cls");

            break;

        default:

            cout << "Enter Proper Option number " << '\n';

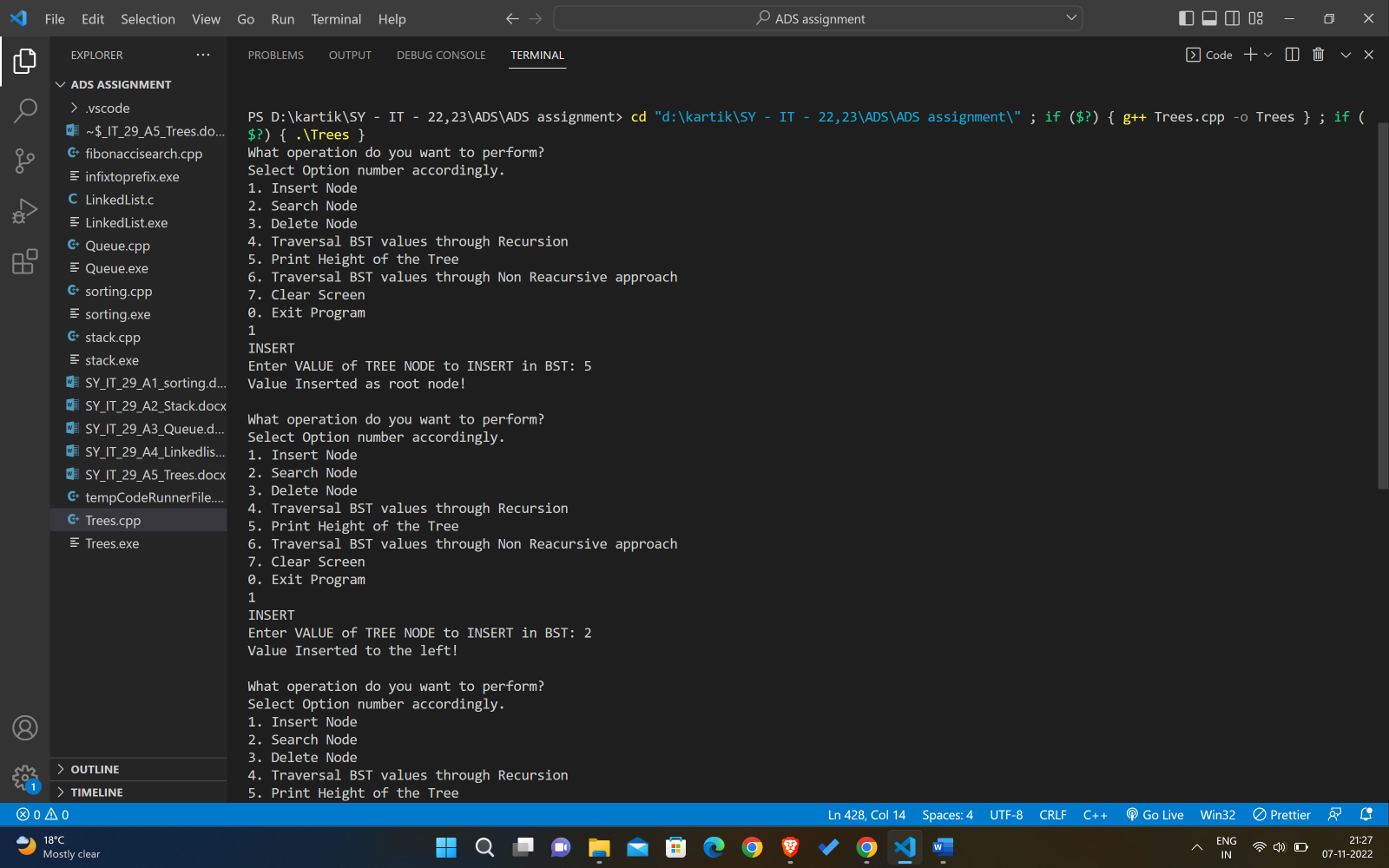
        }

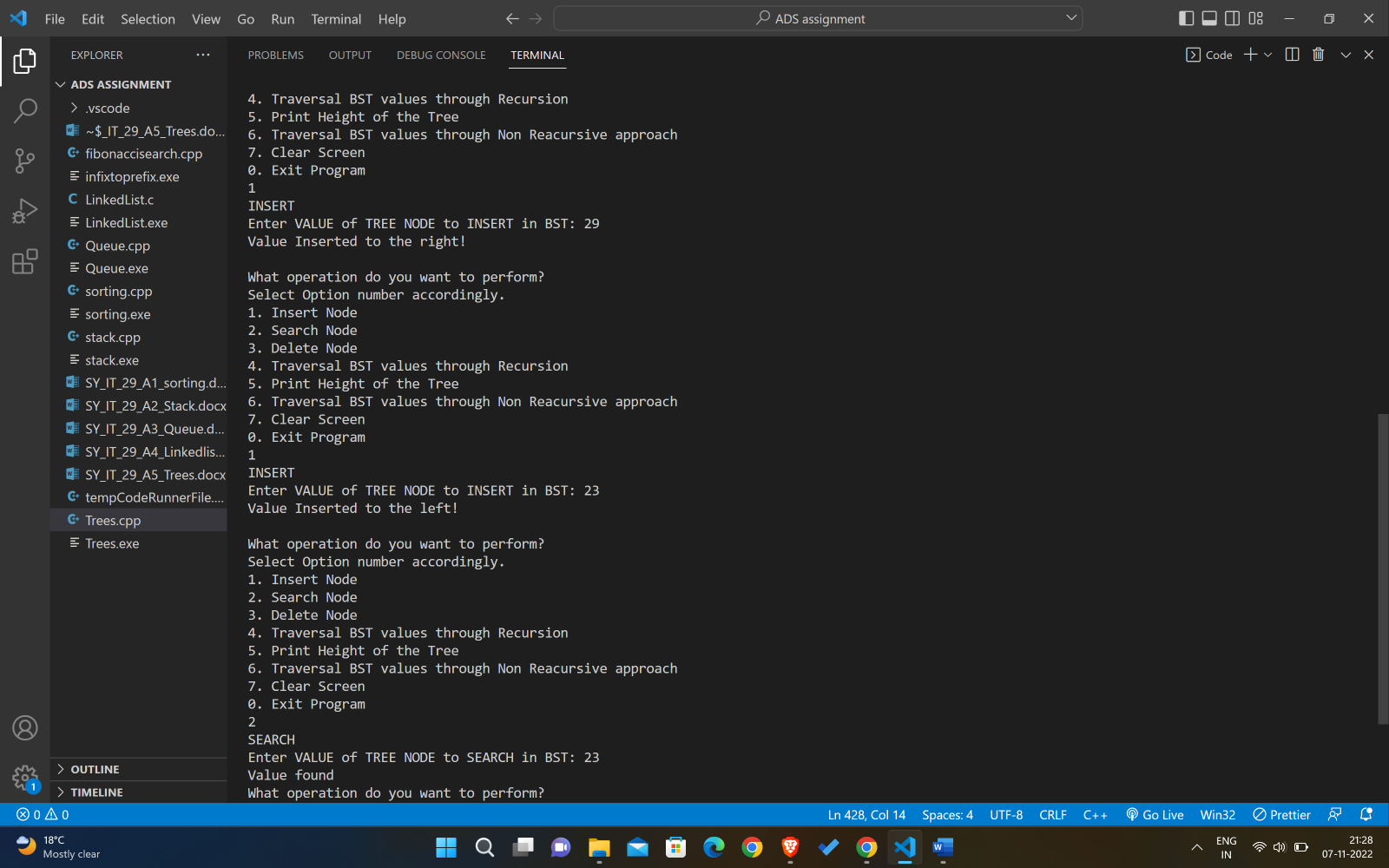
    } while (option != 0);

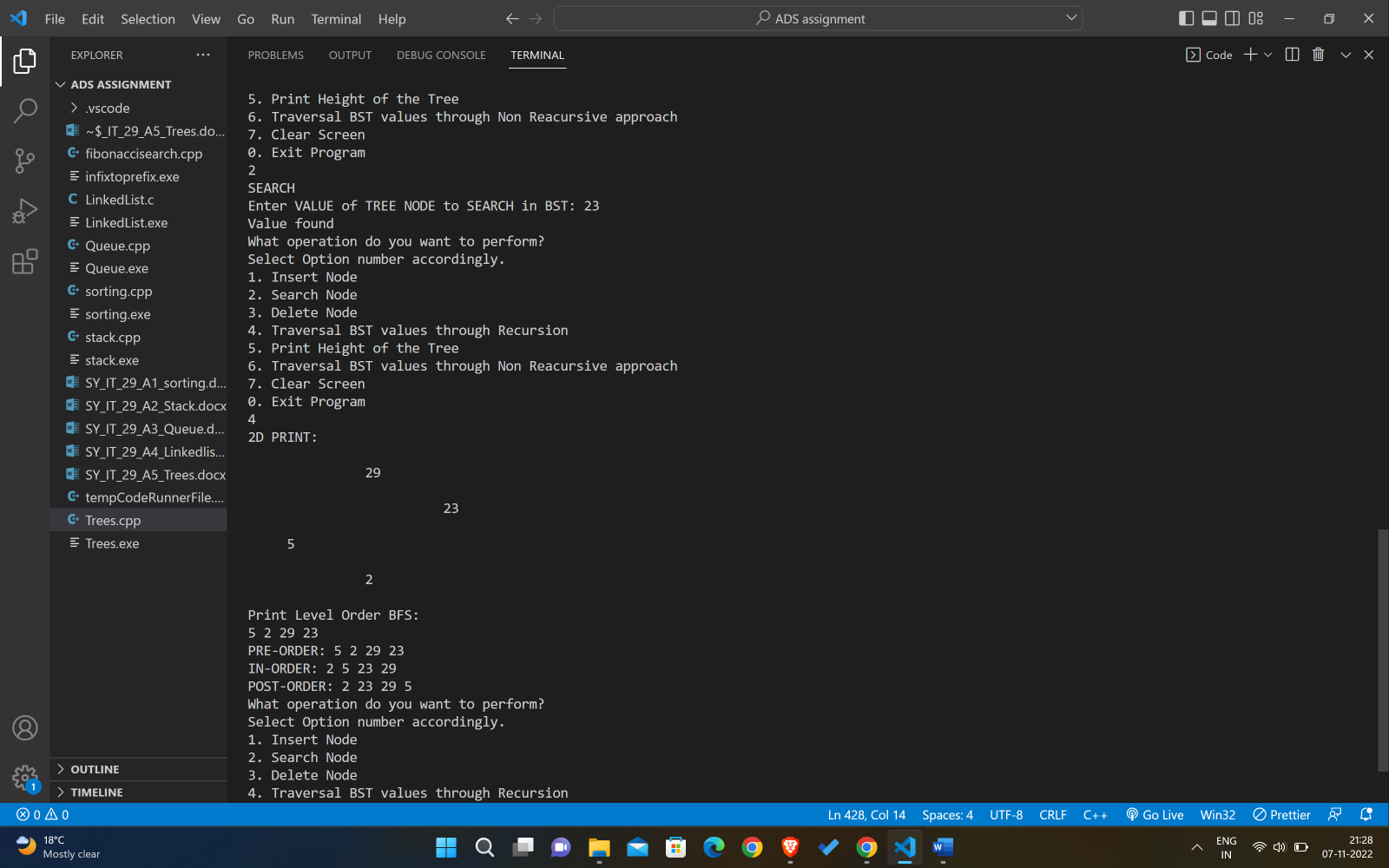
    return 0;

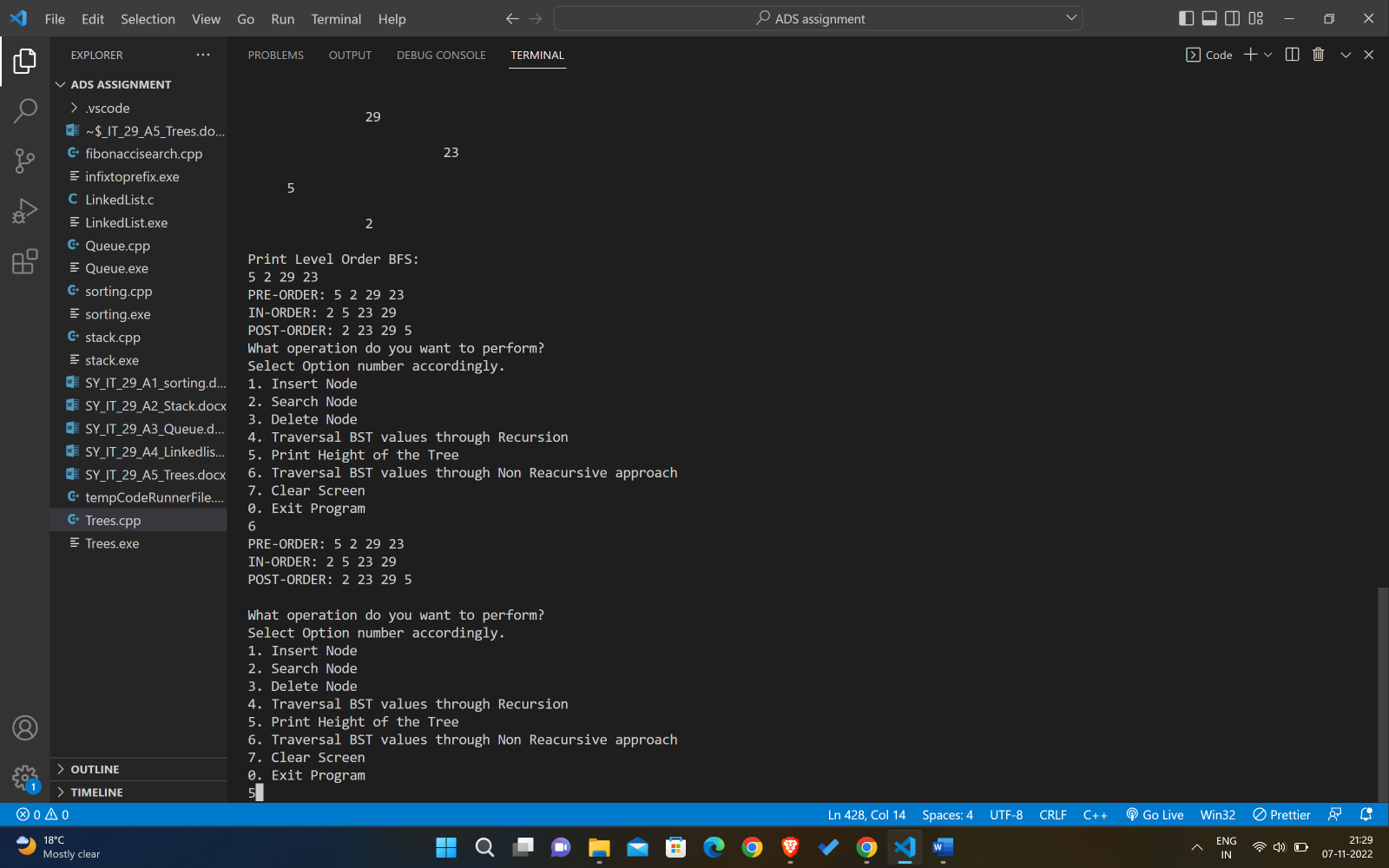
}

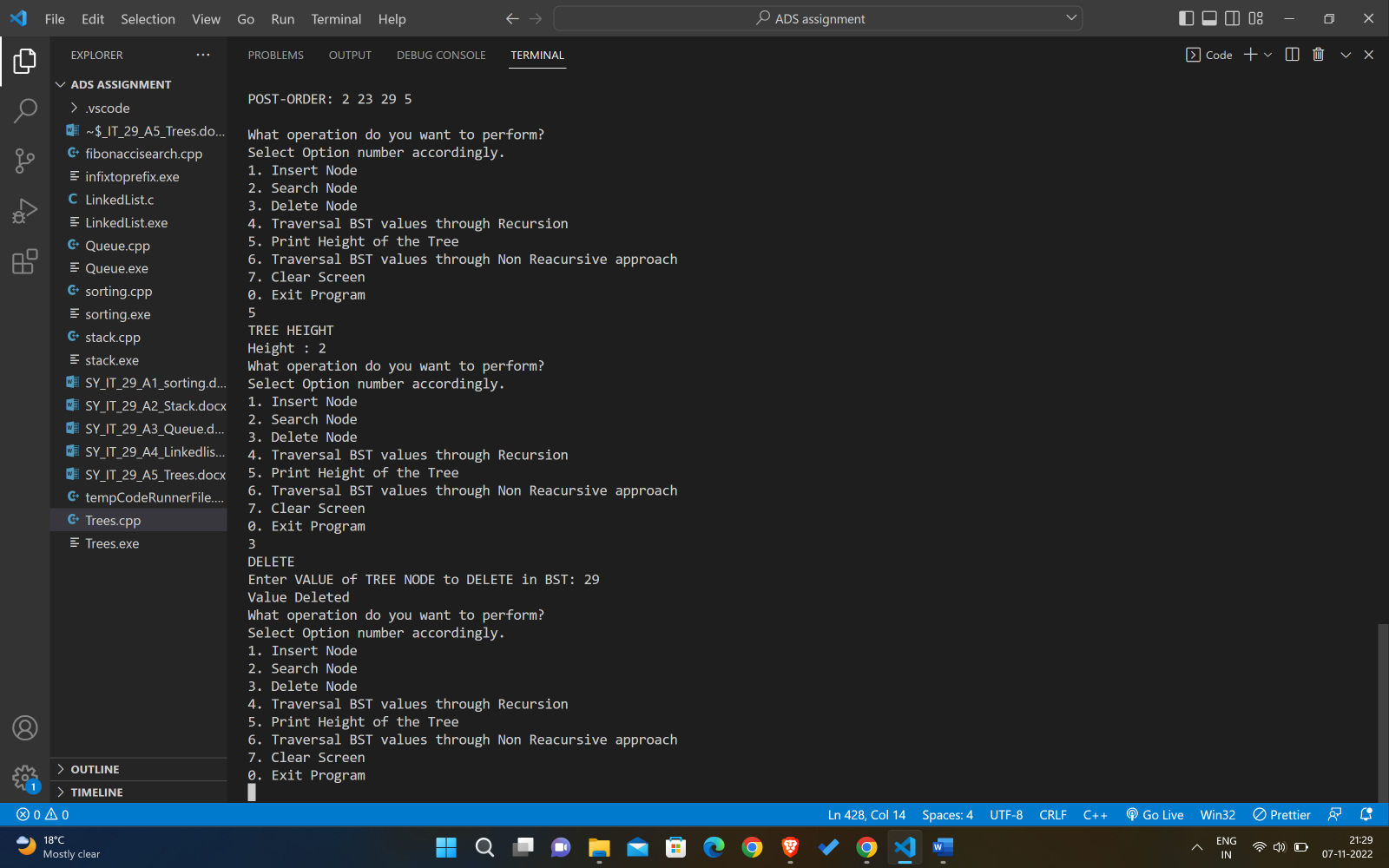
**Input / output:**

****

****

****

****

****