/\*

ASSIGNMENT NO. 4

NAME- ABRAR SHAIKH ROLL NO. - 23570

TOPIC- Expression Tree

\*/

#include <iostream>

using namespace std;

// Node structure for expression tree

struct Node {

char data;

Node\* left;

Node\* right;

};

// Stack class for tree nodes

class Stack {

private:

Node\* arr[100];

int top;

public:

Stack() {

top = -1;

}

void push(Node\* node) {

arr[++top] = node;

}

Node\* pop() {

return arr[top--];

}

bool isEmpty() {

return top == -1;

}

};

// Utility function to create a new node

Node\* createNode(char data) {

Node\* newNode = new Node();

newNode->data = data;

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

return newNode;

}

// Function to check if the character is an operator

bool isOperator(char ch) {

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/');

}

// Function to build expression tree from postfix

Node\* constructTreeFromPostfix(char postfix[]) {

Stack stack;

int i = 0;

while (postfix[i] != '\0') {

char ch = postfix[i];

// If operand, create a node and push it

if (ch >= '0' && ch <= '9') {

stack.push(createNode(ch));

}

// If operator, pop two nodes, make them children, and push new node

else if (isOperator(ch)) {

Node\* node = createNode(ch);

node->right = stack.pop(); // Right child

node->left = stack.pop(); // Left child

stack.push(node);

}

i++;

}

return stack.pop(); // Final node is the root of the expression tree

}

// Function to build expression tree from prefix

Node\* constructTreeFromPrefix(char prefix[], int length) {

Stack stack;

// Traverse the prefix expression from right to left

for (int i = length - 1; i >= 0; i--) {

char ch = prefix[i];

// If operand, create a node and push it

if (ch >= '0' && ch <= '9') {

stack.push(createNode(ch));

}

// If operator, pop two nodes, make them children, and push new node

else if (isOperator(ch)) {

Node\* node = createNode(ch);

node->left = stack.pop(); // Left child

node->right = stack.pop(); // Right child

stack.push(node);

}

}

return stack.pop(); // Final node is the root of the expression tree

}

// Inorder traversal (Left, Root, Right)

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

// Preorder traversal (Root, Left, Right)

void preorder(Node\* root) {

if (root != NULL) {

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

preorder(root->left);

preorder(root->right);

}

}

// Postorder traversal (Left, Right, Root)

void postorder(Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

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

}

}

//inorder Non-Recursive Traversal

void inorderNonRecursive(Node \*root)

{

Stack s;

Node \*tmp=root;

//current should not be null (if null there is no node), stack should not be empty (initially can be)

while(tmp!=NULL || !s.isEmpty())

{

while(tmp!=NULL)

{

s.push(tmp);

tmp=tmp->left;

}

//popping stored left subtree elements

tmp=s.pop();

//printing data

cout<<tmp->data<<" ";

//traversing right subtree

tmp=tmp->right;

}

}

//preorder Non-Recursive Traversal

void preorderNonRecursive(Node \*root)

{

Stack s;

if(root==NULL)

return;

s.push(root);

while(!s.isEmpty())

{

Node \*temp=s.pop();

cout<<temp->data<<" ";

if(temp->right!=NULL)

s.push(temp->right);

if(temp->left!=NULL)

s.push(temp->left);

}

}

//postorder Non-Recursive

void postorderNonRecursive(Node \*root)

{

if(root==NULL)

return;

Stack s,s1;

s.push(root);

while(!s.isEmpty())

{

Node \*temp=s.pop();

s1.push(temp);

if(temp->left!=NULL)

s.push(temp->left);

if(temp->right!=NULL)

s.push(temp->right);

}

while(!s1.isEmpty())

cout<<s1.pop()->data<<" ";

}

int main() {

char expression[100];

int choice;

cout << "Enter 1 for Postfix or 2 for Prefix expression: ";

cin >> choice;

if (choice == 1) {

cout << "Enter a postfix expression (without spaces): ";

cin >> expression;

// Construct the expression tree from postfix

Node\* root = constructTreeFromPostfix(expression);

// Display the tree in different orders

cout << "Inorder (Infix) Expression: ";

inorder(root);

cout << endl;

cout << "Preorder (Prefix) Expression: ";

preorder(root);

cout << endl;

cout << "Postorder (Postfix) Expression: ";

postorder(root);

cout << endl;

// Display the tree in different orders nonrecursive

cout << "NonRecursive Inorder (Infix) Expression: ";

inorderNonRecursive(root);

cout << endl;

cout << "NonRecursive Preorder (Prefix) Expression: ";

preorderNonRecursive(root);

cout << endl;

cout << "NonRecursive Postorder (Postfix) Expression: ";

postorderNonRecursive(root);

cout << endl;

}

else if (choice == 2) {

cout << "Enter a prefix expression (without spaces): ";

cin >> expression;

// Get the length of the prefix expression

int length = 0;

while (expression[length] != '\0') {

length++;

}

// Construct the expression tree from prefix

Node\* root = constructTreeFromPrefix(expression, length);

// Display the tree in different orders

cout << "Inorder (Infix) Expression: ";

inorder(root);

cout << endl;

cout << "Preorder (Prefix) Expression: ";

preorder(root);

cout << endl;

cout << "Postorder (Postfix) Expression: ";

postorder(root);

cout << endl;

// Display the tree in different orders nonrecursive

cout << "NonRecursive Inorder (Infix) Expression: ";

inorderNonRecursive(root);

cout << endl;

cout << "NonRecursive Preorder (Prefix) Expression: ";

preorderNonRecursive(root);

cout << endl;

cout << "NonRecursive Postorder (Postfix) Expression: ";

postorderNonRecursive(root);

cout << endl;

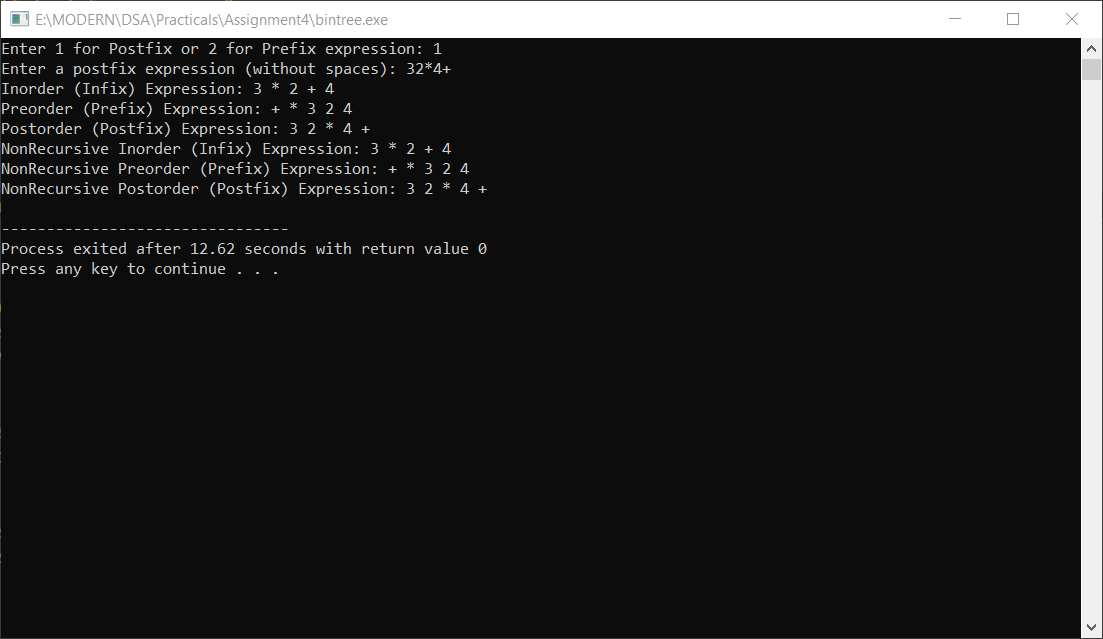
}

else {

cout << "Invalid choice!" << endl;

}

return 0;

}

