#include <iostream>

#include <stack>

using namespace std;

// Node class to represent nodes in the expression tree

class Node{

public:

char data;

Node\* left;

Node\* right;

Node(char val){

data=val;

left=NULL;

right=NULL;

}

};

// Tree class for expression tree operations

class Tree {

public:

stack<Node\*> s;

Node\* root;

Tree();

bool isOperand(char a); //Function to check if character is operand

bool isOperator(char a); //Function to check if character is operator

void inOrdertraversalwithRec(Node\* root); //Function to perform in-order traversal using recursion

void preOrdertraversalwithRec(Node\* root); //Function to perform Pre-order traversal using recursion

void postOrdertraversalwithRec(Node\* root); //Function to perform Post-order traversal using recursion

void inOrdertraversalwithoutRec(Node\* root); //Function to perform iterative in-order traversal

void preOrdertraversalwithoutRec(Node\* root); //Function to perform iterative pre-order traversal

void postOrdertraversalwithoutRec(Node\* root); //Function to perform iterative post-order traversal

void expressionTree(const char postfix[]); //Function to perform expression tree operations

bool validatePostfixExpression(const char postfix[]); //Function to perform postfix expression validations

};

Tree::Tree(){

root=NULL;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Function to check if character is operand\*\*\*\*\*\*\*\*\*\*\*\*\*//

bool Tree::isOperand(char a){

return isalnum(a);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Function to check if character is operator\*\*\*\*\*\*\*\*\*\*\*\*\*//

bool Tree::isOperator(char a){

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

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Recursive Inorder Traversal\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::inOrdertraversalwithRec(Node\* root){

if(root==NULL) // Base case: if the current node is NULL, return

return;

inOrdertraversalwithRec(root->left); // Recursively traverse the left subtree

cout<<root->data<<" "; // Print the data of the current node

inOrdertraversalwithRec(root->right); // Recursively traverse the right subtree

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Recursive Preorder Traversal\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::preOrdertraversalwithRec(Node\* root){

if(root==NULL) // Base case: if the current node is NULL, return

return;

cout<<root->data<<" "; // Print the data of the current node

preOrdertraversalwithRec(root->left); // Recursively traverse the left subtree

preOrdertraversalwithRec(root->right); // Recursively traverse the right subtree

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Recursive Postorder Traversal\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::postOrdertraversalwithRec(Node\* root){

if(root==NULL)

return;

postOrdertraversalwithRec(root->left); // Recursively traverse the left subtree

postOrdertraversalwithRec(root->right); // Recursively traverse the right subtree

cout<<root->data<<" "; // Print the data of the current node

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Iterative Inorder Traversal\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::inOrdertraversalwithoutRec(Node\* root){

Node\* t=root;

while(t!=NULL || !s.empty()){ // Traverse left subtree and push nodes onto the stack

while(t!=NULL){

s.push(t);

t=t->left;

}

t=s.top();

s.pop(); // Pop a node from the stack, print its data, and traverse its right subtree

cout<<t->data<<" ";

t=t->right;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Iterative Preorder Traversal\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::preOrdertraversalwithoutRec(Node\* root){

if(root==NULL)

return;

stack<Node\*>st;

st.push(root);

while(!st.empty()){

Node\* t=st.top(); // Pop a node from the stack, print its data, and push its right and left children onto the stack

st.pop();

cout<<t->data<<" ";

if(t->right){

st.push(t->right);

}

if(t->left){

st.push(t->left);

}

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Iterative Postorder Traversal\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::postOrdertraversalwithoutRec(Node\* root){

if (root == NULL)

return;

stack<Node \*> s1, s2;

Node \*temp = root;

s1.push(temp);

while(!s1.empty()){

temp = s1.top();

s1.pop(); // Pop a node from the first stack, push it onto the second stack, and push its left and right children onto the first stack

s2.push(temp);

if(temp->left != NULL){

s1.push(temp->left);

}

if(temp->right != NULL){

s1.push(temp->right);

}

}

while (!s2.empty()){ // Pop nodes from the second stack and print their data

cout<<s2.top()->data << " ";

s2.pop();

}

cout << endl;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* Postfix Expression Validation \*\*\*\*\*\*\*\*\*\*\*\*\*//

bool Tree::validatePostfixExpression(const char postfix[]) {

int i = 0;

int operandCount = 0;

int operatorCount = 0;

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

char curr = postfix[i];

if(isOperand(curr)){ // Check if the current character is an operand

operandCount++;

}

else if(isOperator(curr)){ // Check if the current character is an operator

if(operandCount<2){

cout<<"Invalid postfix expression: Not enough operands for operator "<<curr<<endl;

return false; // Ensure there are enough operands to perform the operation

}

operandCount--;

operatorCount++;

}

else{ // Invalid character in the postfix expression

cout<<"Invalid postfix expression: Invalid character "<<curr<<endl;

return false;

}

i++;

}

if(operandCount - operatorCount >= 1){ // Check if the number of operands and operators match

cout<<"Invalid postfix expression: Number of operands and operators do not match" <<endl;

return false;

}

return true;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*Expression Tree\*\*\*\*\*\*\*\*\*\*\*\*\*//

void Tree::expressionTree(const char postfix[]){

if(!validatePostfixExpression(postfix)){ // Postfix expression validation

cout<<"Expression validation failed."<<endl;

return;

}

int i=0;

stack<Node\*>tree;

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

char curr=postfix[i];

if(isOperand(curr)){ // If current character is operand

Node\* t=new Node(curr);

tree.push(t);

}

else if(isOperator(curr)){ // If current character is operand

Node\* op=new Node(curr);

op->right=tree.top();

tree.pop();

op->left=tree.top();

tree.pop();

tree.push(op);

}

i++;

}

root=tree.top(); // Set the root of the expression tree to the top of the stack

}

int main() {

Tree t;

char postfix[100];

while (true) {

cout<<endl;

cout << "1. Enter postfix expression" << endl;

cout << "2. Inorder traversal without recursion" << endl;

cout << "3. Preorder traversal without recursion" << endl;

cout << "4. Postorder traversal without recursion" << endl;

cout << "5. Inorder traversal with recursion" << endl;

cout << "6. Preorder traversal with recursion" << endl;

cout << "7. Postorder traversal with recursion" << endl;

cout << "8. Exit" << endl;

cout << "Enter your choice: ";

cout<<endl;

cout<<endl;

int choice;

cin >> choice;

switch (choice) {

case 1:

cin.ignore();

cout << "Enter postfix expression: ";

cin.getline(postfix, sizeof(postfix));

t.expressionTree(postfix);

break;

case 2:

cout<<"Inorder Traversal: ";

t.inOrdertraversalwithoutRec(t.root);

cout<<endl;

break;

case 3:

cout<<"Preorder Traversal: ";

t.preOrdertraversalwithoutRec(t.root);

cout<<endl;

break;

case 4:

cout<<"postorder Traversal: ";

t.postOrdertraversalwithoutRec(t.root);

cout<<endl;

break;

case 5:

cout<<"Inorder Traversal: ";

t.inOrdertraversalwithRec(t.root);

cout<<endl;

break;

case 6:

cout<<"preorder Traversal: ";

t.preOrdertraversalwithRec(t.root);

cout<<endl;

break;

case 7:

cout<<"postorder Traversal: ";

t.postOrdertraversalwithRec(t.root);

cout<<endl;

break;

case 8:

return 0;

default:

cout << "Invalid choice. Please try again." << endl;

}

}

return 0;

}