# Practical 4 Source Code:-

#include <iostream>

#include <stack> #include <string>

using namespace std;

// Node structure for an expression tree struct Node { char data; Node\* left;

Node\* right;

Node(char ch) { data = ch;

left = right = nullptr;

}

};

// Function to check if a character is an operand bool isOperand(char ch) {

return (ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// Function to construct an expression tree from a postfix expression

Node\* constructExpressionTreePostfix(string postfix) { stack<Node\*> stack;

for (int i = 0; i < postfix.length(); i++) { char ch = postfix[i];

if (isOperand(ch)) {

Node\* newNode = new Node(ch); stack.push(newNode);

} else {

if (stack.size() < 2) {

cerr << "Invalid postfix expression." << endl; return nullptr;

}

Node\* right = stack.top(); stack.pop();

Node\* left = stack.top();

stack.pop();

Node\* newNode = new Node(ch);

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

stack.push(newNode);

}

}

if (stack.size() != 1) {

cerr << "Invalid postfix expression." << endl; return nullptr;

}

return stack.top();

}

// Function to construct an expression tree from a prefix expression

Node\* constructExpressionTreePrefix(string prefix) { stack<Node\*> stack;

for (int i = prefix.length() - 1; i >= 0; i--) { char ch = prefix[i];

if (isOperand(ch)) {

Node\* newNode = new Node(ch);

stack.push(newNode);

} else {

if (stack.size() < 2) {

cerr << "Invalid prefix expression." << endl;

return nullptr;

}

Node\* left = stack.top(); stack.pop();

Node\* right = stack.top();

stack.pop();

Node\* newNode = new Node(ch);

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

stack.push(newNode);

}

}

if (stack.size() != 1) {

cerr << "Invalid prefix expression." << endl;

return nullptr;

}

return stack.top();

}

// Recursive in-order traversal void inOrderTraversalRecursive(Node\* root) {

if (root == nullptr) { return;

}

inOrderTraversalRecursive(root->left); cout << root->data << " ";

inOrderTraversalRecursive(root->right);

}

// Recursive pre-order traversal void preOrderTraversalRecursive(Node\* root) {

if (root == nullptr) { return;

}

cout << root->data << " "; preOrderTraversalRecursive(root->left);

preOrderTraversalRecursive(root->right);

}

// Recursive post-order traversal void postOrderTraversalRecursive(Node\* root) {

if (root == nullptr) {

return;

}

postOrderTraversalRecursive(root->left); postOrderTraversalRecursive(root->right); cout << root->data << " ";

}

// Non-recursive in-order traversal void inOrderTraversalNonRecursive(Node\* root) {

stack<Node\*> stack; Node\* current = root;

while (current != nullptr || !stack.empty()) { while (current != nullptr) { stack.push(current);

current = current->left;

}

current = stack.top(); stack.pop(); cout << current->data << " ";

current = current->right;

}

}

// Non-recursive pre-order traversal void preOrderTraversalNonRecursive(Node\* root) {

stack<Node\*> stack; Node\* current = root;

while (current != nullptr || !stack.empty()) { while (current != nullptr) { cout << current->data << " "; stack.push(current);

current = current->left;

}

current = stack.top(); stack.pop(); current = current->right;

}

}

// Non-recursive post-order traversal

void postOrderTraversalNonRecursive(Node\* root) { stack<Node\*> stack1; stack<Node\*> stack2;

Node\* current = root;

while (current != nullptr) { stack1.push(current); current = current->left;

}

while (!stack1.empty()) { current = stack1.top(); stack1.pop();

stack2.push(current);

current = current->right; while (current != nullptr) { stack1.push(current);

current = current->left;

}

}

while (!stack2.empty()) { cout << stack2.top()->data << " ";

stack2.pop();

}

}

int main() {

string postfixExpression, prefixExpression;

cout << "Enter a postfix expression: ";

getline(cin, postfixExpression);

cout << "Enter a prefix expression: "; getline(cin, prefixExpression);

Node\* postfixTree = constructExpressionTreePostfix(postfixExpression);

Node\* prefixTree = constructExpressionTreePrefix(prefixExpression);

if (postfixTree) { cout << "\nIn-order traversal (postfix): "; inOrderTraversalRecursive(postfixTree); cout << endl;

inOrderTraversalNonRecursive(postfixTree);

cout << endl;

cout << "Pre-order traversal (postfix): "; preOrderTraversalRecursive(postfixTree); cout << endl;

preOrderTraversalNonRecursive(postfixTree); cout << endl;

cout << "Post-order traversal (postfix): "; postOrderTraversalRecursive(postfixTree); cout << endl;

postOrderTraversalNonRecursive(postfixTree); cout << endl;

}

if (prefixTree) { cout << "\nIn-order traversal (prefix): "; inOrderTraversalRecursive(prefixTree); cout << endl;

inOrderTraversalNonRecursive(prefixTree); cout << endl;

cout << "Pre-order traversal (prefix): "; preOrderTraversalRecursive(prefixTree); cout << endl;

preOrderTraversalNonRecursive(prefixTree); cout << endl;

cout << "Post-order traversal (prefix): "; postOrderTraversalRecursive(prefixTree); cout << endl;

postOrderTraversalNonRecursive(prefixTree); cout << endl;

}

return 0;

}

# Output:-

