# Practical 2 Source Code:-

#include <iostream>

#include <string> #include<math.h>

using namespace std;

// Node structure for a singly linked list struct Node { char data; Node\* next;

};

// Stack class using a singly linked list class Stack { public: Stack() { top = nullptr;

}

bool isEmpty() {

return top == nullptr;

}

void push(char data) { Node\* newNode = new Node; newNode->data = data; newNode->next = top;

top = newNode;

}

char pop() {

if (isEmpty()) { cout << "Stack is empty!" << endl;

return '\0';

}

char poppedData = top->data; Node\* temp = top; top = top->next; delete temp; return poppedData;

}

char peek() { if (isEmpty()) { cout << "Stack is empty!" << endl;

return '\0';

}

return top->data;

}

private: Node\* top; };

// 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 check precedence of operators int precedence(char ch) {

if (ch == '^') { return 3;

} else if (ch == '\*' || ch == '/') { return 2;

} else if (ch == '+' || ch == '-') { return 1; } else {

return -1;

}

}

// Function to convert infix expression to postfix

string infixToPostfix(string infix) { Stack stack; string postfix;

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

if (isOperand(ch)) { postfix += ch; } else if (ch == '(') { stack.push(ch); } else if (ch == ')') {

while (!stack.isEmpty() && stack.peek() != '(') {

postfix += stack.pop();

}

stack.pop(); // Pop the '('

} else {

while (!stack.isEmpty() && precedence(ch) <= precedence(stack.peek())) { postfix += stack.pop();

}

stack.push(ch);

}

}

while (!stack.isEmpty()) {

postfix += stack.pop();

}

return postfix;

}

// Function to convert infix expression to prefix

string infixToPrefix(string infix) { string reversedInfix; for (int i = infix.length() - 1; i >= 0; i--) { reversedInfix += infix[i];

}

string reversedPrefix = infixToPostfix(reversedInfix); string prefix;

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

}

return prefix;

}

// Function to evaluate postfix expression int evaluatePostfix(string postfix) {

Stack stack;

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

if (isOperand(ch)) {

stack.push(ch - '0'); // Convert character to integer

} else { int op2 = stack.pop(); int op1 = stack.pop();

int result;

switch (ch) { case '+': result = op1 + op2;

break; case '-':

result = op1 - op2;

break; case '\*': result = op1 \* op2;

break; case '/':

result = op1 / op2;

break; case '^':

result = pow(op1, op2); break; default:

cout << "Invalid operator!" << endl;

return 0;

}

stack.push(result);

}

}

return stack.pop();

}

// Function to evaluate prefix expression

int evaluatePrefix(string prefix) { string reversedPrefix;

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

reversedPrefix += prefix[i];

}

return evaluatePostfix(reversedPrefix);

}

int main() {

string infixExpression;

cout << "Enter an infix expression: "; getline(cin, infixExpression);

string postfix = infixToPostfix(infixExpression); cout << "Postfix expression: " << postfix << endl;

string prefix = infixToPrefix(infixExpression); cout << "Prefix expression: " << prefix << endl;

int postfixResult = evaluatePostfix(postfix);

cout << "Evaluation of postfix expression: " << postfixResult << endl;

int prefixResult = evaluatePrefix(prefix);

cout << "Evaluation of prefix expression: " << prefixResult << endl;

return 0;

}

# Output:-

