Lab Assignment 2

Implement stack as an abstract data type using singly linked list and use this ADT for conversion of infix expression to postfix,prefix and evaluation of postfix and prefix expression.

Code

```
#include <iostream>
#include <string>
#include <cmath>
#define MAXSIZE 100
using namespace std;

char stack[MAXSIZE];
int top = -1;

// Function to push an element onto the stack
void push(char element) {
    if (top == MAXSIZE - 1) {
        cout << "Stack is full" << endl;
    } else {
```

```
        stack[++top] = element;
    }
}

// Function to pop an element from the stack
char pop() {
    if (top == -1) {
        cout << "Stack is empty" << endl;
        return '\0'; // Return null character if stack is
empty
    } else {
        return stack[top--];
    }
}

// Function to get the top element of the stack
without popping
char peek() {
    if (top == -1) {
        return '\0'; // Return null character if stack is
empty
    } else {
        return stack[top];
    }
```

```
}

// Function to return precedence of operators
int precedence(char op) {
    if (op == '^' || op == '$') {
        return 3;
    }
    if (op == '*' || op == '/') {
        return 2;
    }
    if (op == '+' || op == '-') {
        return 1;
    }
    return 0;
}

// Function to convert infix expression to postfix
expression
string infix_to_postfix(const string& infix) {
    string postfix;
    for (size_t i = 0; i < infix.length(); i++) {
        char ch = infix[i];
        if (isalnum(ch)) {
            postfix += ch;
```

```
        } else if (ch == '(') {
            push(ch);
        } else if (ch == ')') {
            while (top != -1 && peek() != '(') {
                postfix += pop();
            }
            pop(); // Remove '(' from the stack
        } else { // The character is an operator
            while (top != -1 && precedence(peek()) >=
precedence(ch)) {
                postfix += pop();
            }
            push(ch);
        }
    }
    while (top != -1) {
        postfix += pop();
    }
    return postfix;
}

// Function to reverse a string
string reverse_string(const string& str) {
    string reversed;
```

```
    for (size_t i = str.length(); i > 0; i--) {
        reversed += str[i - 1];
    }
    return reversed;
}

// Function to convert infix expression to prefix
expression
string infix_to_prefix(const string& infix) {
    // Reverse the infix expression
    string reversed_infix = reverse_string(infix);

    // Replace '(' with ')' and vice versa
    for (size_t i = 0; i < reversed_infix.length(); i++) {
        if (reversed_infix[i] == '(') {
            reversed_infix[i] = ')';
        } else if (reversed_infix[i] == ')') {
            reversed_infix[i] = '(';
        }
    }

    // Convert modified infix to postfix
    string postfix = infix_to_postfix(reversed_infix);
```

```
    // Reverse the postfix expression to get prefix
    return reverse_string(postfix);
}

// Function to evaluate postfix expression
int evaluate_postfix(const string& postfix) {
    int eval_stack[MAXSIZE];
    int eval_top = -1;

    for (size_t i = 0; i < postfix.length(); i++) {
        char ch = postfix[i];
        if (isalnum(ch)) {
            // Convert the character to integer and push
to eval_stack
            eval_stack[++eval_top] = ch - '0'; // Assuming
single-digit operands
        } else { // The character is an operator
            int val2 = eval_stack[eval_top--];
            int val1 = eval_stack[eval_top--];
            int result;
            switch (ch) {
                case '+': result = val1 + val2; break;
                case '-': result = val1 - val2; break;
                case '*': result = val1 * val2; break;
```

```
                case '/': result = val1 / val2; break;
                case '^': result = (int)pow(val1, val2);
break;
                default: result = 0; // Default case to
handle unexpected operators
            }
            eval_stack[++eval_top] = result;
        }
    }
    return eval_stack[eval_top];
}

// Function to evaluate prefix expression
int evaluate_prefix(const string& prefix) {
    int eval_stack[MAXSIZE];
    int eval_top = -1;

    // Process the prefix expression from right to left
    for (int i = prefix.length() - 1; i >= 0; i--) {
        char ch = prefix[i];
        if (isalnum(ch)) {
            // Convert the character to integer and push
to eval_stack
```

```
            eval_stack[++eval_top] = ch - '0'; // Assuming
single-digit operands
        } else {
            int val1 = eval_stack[eval_top--];
            int val2 = eval_stack[eval_top--];
            int result;
            switch (ch) {
                case '+': result = val1 + val2; break;
                case '-': result = val1 - val2; break;
                case '*': result = val1 * val2; break;
                case '/': result = val1 / val2; break;
                case '^': result = (int)pow(val1, val2);
break;
                default: result = 0; // Default case to
handle unexpected operators
            }
            eval_stack[++eval_top] = result;
        }
    }
    return eval_stack[eval_top];
}

int main() {
    string infix;
```

```
    cout << "Enter an infix expression: ";
    cin >> infix;

    string postfix = infix_to_postfix(infix);
    cout << "Postfix expression: " << postfix << endl;

    int postfix_result = evaluate_postfix(postfix);
    cout << "Result of postfix evaluation: " <<
postfix_result << endl;

    string prefix = infix_to_prefix(infix);
    cout << "Prefix expression: " << prefix << endl;

    int prefix_result = evaluate_prefix(prefix);
    cout << "Result of prefix evaluation: " <<
prefix_result << endl;

    return 0;
}
```

Output

```
/tmp/xD9Re0mpgM.o
Enter an infix expression: 5+7*8
Postfix expression: 578*+
Result of postfix evaluation: 61
Prefix expression: +5*78
Result of prefix evaluation: 61


=== Code Execution Successful ===
```