DATA STRUCTURES AND ALGORITHMS

CHALLENGE 1 - GROUP 11508289
ARITHMETIC EXPRESSION CALCULATION
01/11/2020

INTRODUCTION

Option: Input expressions contain both integers and floating-point numbers.
The group has done the calculation and transformed the expression.

Name	ID
La Ngọc Hồng Phúc	19127511
Hồ Lâm Bảo Khuyên	19127189
Nguyễn Tất Trường	19127082
Trần Quốc Tuấn	19127650

Contents

Research

OVERALL

The way to write expression is known as notation. There are three ways to demonstrate an arithmetic expression:

1. Infix notation
2. Prefix – Polish notation
3. Postfix – Reverse Polish notation

Three of these have differences in syntax. Because of the operator's precedence, the infix notation is not suitable for computer programming, so we have to find a way to convert mathematical problems to the others.

Infix notation: This expression is easy to understand. The operator is between two operands.
e.g. A+B, C^D.

Prefix notation: In this expression, the operator is laid on the left side of two operands.
e.g. +AB, ^CD

Postfix notation: This is a reverse model of the prefix notation; the operator is written after the operands.
e.g. AB+, CD^

Precedence level of operators: This is a descending list.

Operator description	operator
Curly brackets	{}
Square brackets	[]
Round brackets	()
Power	^
Multiplication, division, power	*, /
Addition, subtraction	+ , -

CONVERTING AN INFIX EXPRESSION TO A PREFIX EXPRESSION

(A + B) * (C + D) => * + A B + C D

Illustrate the algorithm:

1. Scan the reverse input string from right to left until the stack is empty.
2. We will use a stack variable to store operators and a string to store the output.
3. Put the close bracket at the top of the stack and open bracket at the end of the input.
4. If we get an operand, it will be added to the output.
5. If we encounter an operator, then:
   We will repeatedly pop from the stack variable the operator which has higher or the same precedence level and add it to the output
6. Add the scanned operator to the stack.
7. If we encounter the open operator, then:
   We pop elements on the top of the stack until we encounter the close bracket and remove the scanned element.
   e.g.: convert (E-F) / (B*A) to prefix expression:
   Reverse input:)A*B(/)F-E((

Scanned	Stack	Output
	)	
)	))	
A	))	A
*****	))*	A
B	))*	A B
(	)	A B *
/	) /	A B *
)	) /)	A B *
F	) /)	A B * F
-	) /) -	A B * F
E	) /) -	A B * F E
(	) /	A B * F E -
(	)	A B * F E - /
empty	empty	end

Reverse again the output, we get: / - E F * A B

Illustration the code fragment

```
string infixToPrefix(string str)
{
    reverse str

    replace open bracket with close bracket and vice versa

    str = '(' + str + ')';
    stack<char> char_stack;
    string output;

    for i = 0 to str.size()-1
        if str[i] is operand, add str[i] to output

        else if str[i] is open bracket, push on char_stack

        else if str[i] is close bracket

            while top of char_stack != '('
                add top of char_stack to output
                remove character has recently added

            Pop ‘(‘ from char_stack

        else

            if top of char_stack is operator
                while str[i] has priority <= top of char_stack
                    add top of char_stack to output
                    pop character has recently added from char_stack

                Push current operator on stack


    reverse output

    return output;
}
```

CONVERTING AN INFIX EXPRESSION TO A POSTFIX EXPRESSION

(A + B) * (C + D) => A B + C D + *

Illustrate the algorithm:

1. Scan the string from left to right to catch operators and operands until the stack is empty.
2. We will use a stack variable to store operators and a string to store operands
3. Put the open bracket at the top of the stack and the close bracket at the end of the input string to check if we have scanned correctly.
4. If we encounter a left parenthesis, add it to the stack variable.
5. If we catch an operand, it will be added to the output string.
6. If we catch an operator, then the precedence level will be checked:If on the top of the stack, there is a higher or the same level operator then we pop that operator from the stack and add it to output string. Else add the caught operator to the stack.
7. If we got a right parenthesis:
   we will pop the stack until the open bracket is caught and remove it. The operator got from stack is added to the output. After checking the stack, we delete the scanned bracket.
8. Finish the loop, we got the postfix expression.

e.g. This is an example: convert (A + B) * (C – D) to postfix expression

Scanned	Stack	Output
	(	
(	((	
A	((	A
+	((+	A
B	((+	A B
)	(	A B +
*****	(*	A B +
(	(*(	A B +
C	(*(	A B + C
-	(*(-	A B + C
D	(*(-	A B + C D
)	(*	A B + C D - *
empty	empty	end

The result: A B + C D - *

The illustration for the code fragment

```
string infixToPostfix(string str)
{
    str = '(' + str + ')';
    stack<char> char_stack;
    string output;

    for str[0] to str[str.size()-1]
        if str[i] is operand, add str[i] to output

        else if str[i] is open bracket, push on char_stack

        else if str[i] is close bracket

            while top of char_stack != '('
                add top of char_stack to output
                remove character has recently added

            Pop ‘(‘ from char_stack

        else {

            if top of char_stack is operator
                while str[i] has priority <= top of char_stack
                    add top of char_stack to output
                    pop character has recently added from char_stack

                Push current operator on stack
           }


    return output;
}
```

CONVERTING A PREFIX EXPRESSION TO A POSTFIX EXPRESSION

* + A B + C D => A B + C D + *

Illustrate the algorithm

1. The input from right to left until the input is empty.
2. We will use a stack variable and a temporary string for this convert.
3. If we get an operand from scanning, it will be pushed to the stack.
4. If we encounter an operator, then:
   Pop from stack two operands called op1 and op2 correspondingly.
   Using a temporary string to connect the scanned operator and two operands as structure: op1 + op2 + operator.
   Then push that string to the stack, it will become a new operand.
5. Finally, when the input is empty, we initialize a new string and pop everything from the stack so that we get a postfix expression.

e.g.: convert * - A / E F + * C D G

Scanned	stack	description
G	G	
D	G D	
C	G D C	
*	G C D*	String: C D * and push back to the stack
+	C D* G +	String: C D * G +
F	C D * G + F	
E	C D * G + F E	
/	C D * G + E F /	String: E F /
A	C D * G + E F / A	
-	C D * G + A E F / -	String: A E F / -
*	A E F / - C D * G + *	String: A E F / - C D * G + *
Empty	A E F / - C D * G + *	

Push everything from to stack to a new string we got the result: A E F / - C D * G + *

The illustration code

```
string preToPost(string input)
{
    stack<string> str;

    use for loop reading input from right to left
        if input[i] is operator
           pop two operands from stack str to string op1, op2
            string temp = op1 + op2 + input[i];
            push temp on stack str

        if symbol is an operand
            push it to the stack

    // stack contains only the Postfix expression
    return str.top();
}
```

Reference

https://scanftree.com/Data_Structure/infix-to-prefix

https://www.includehelp.com/c/infix-to-postfix-conversion-using-stack-with-c-program.aspx

https://www.geeksforgeeks.org/prefix-postfix-conversion/

https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/

https://runestone.academy/runestone/books/published/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html