Experiment No. 6

Aim:

To implement a program for expression conversion from **infix to postfix** and evaluate the postfix expression using stack, considering operands and operators as single characters and only +, –, and * operators.

Objective:

- To understand the concept of stack data structure.
- To learn infix to postfix expression conversion.
- To evaluate postfix expressions using a stack.
- To handle expressions with single-character operands and operators (+, –, *).

Problem Statement:

Write a program to convert a given infix expression into its equivalent postfix form and then evaluate the postfix expression. Both operands and operators in the expression are single characters, and only +, -, and * operators are allowed. Use stack data structure for both conversion and evaluation.

. Theory:

Stack is mainly used for expression conversion and expression evaluation purpose.
In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows:
An **expression** is a collection of operators and operands that represents a specific value. In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
Operands are the values on which the operators can perform the task. Here operand can be adirect value or variable or address of memory location.

Based on the operator position, expressions are divided into three types. They are as follows:

- Infix Expression
- Postfix Expression
- Prefix Expression

Infix Expression

In Infix expression, operator is used in between operands.
The general structure of an Infix expression is as follows...
Operand1 Operator Operand2
Example : a + b

Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".
The general structure of Postfix expression is as follows...
Operand1 Operand2 Operator
Example : a b +

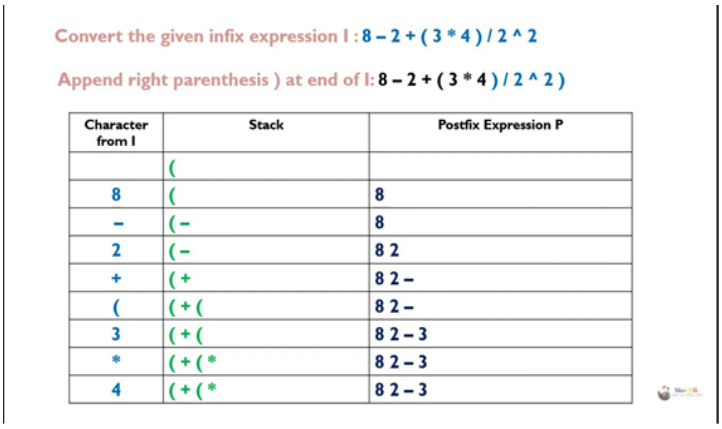
Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".
The general structure of Prefix expression is as Operator Operand1 Operand2
Example : + a b

Infix to postfix conversion:

1.Scan through an expression, getting one token at a time.
2.Fix a priority level for each operator. For example, from high to low:
- (unary negation)
* /
+ - (subtraction)
3.Thus, high priority corresponds to high number in the table. If the token is anoperand, do not stack it. Pass it to the output.
4. If token is an operator or parenthesis, do the following:
i.Pop the stack until you find a symbol of lower priority number than the current one.

An incoming left parenthesis will be considered to have higher priority than any other symbol.A left parenthesis on the stack will not be removed unless an incoming right parenthesis isfound.
The popped stack elements will be written to output.


Convert the given infix expression I : 8 – 2 + (3 * 4) / 2 ^ 2

Append right parenthesis) at end of I: 8 – 2 + (3 * 4) / 2 ^ 2)

Character from I	Stack	Postfix Expression P
	(	
8	(	8
–	(–	8
2	(–	8 2
+	(+	8 2 –
(	(+ (	8 2 –
3	(+ (	8 2 – 3
*	(+ (*	8 2 – 3
4	(+ (*	8 2 – 3

Fig: Infix to postfix conversion

i. Stack the current symbol.
ii. If a right parenthesis is the current symbol, pop the stack down to (and including) the firstleft parenthesis. Write all the symbols except the left parenthesis to the output (i.e. write the operators to the output).
iii. After the last token is read, pop the remainder of the stack and write any symbol (exceptleft parenthesis) to output.

Postfix Evaluation:

Postfix Expression: 82-34*22^/+

Algorithm:

Infix to Postfix Conversion:

1. Initialize an empty stack for operators.
2. Scan the infix expression from left to right.
3. If the scanned character is an operand, append it to the postfix output.
4. If the character is (, push it to the stack.
5. If the character is), pop and append to postfix until (is found, then discard (.
6. If the character is an operator (+, –, *):
   - Pop from the stack while the top operator has higher or equal precedence, append it to postfix.
   - Push the current operator on the stack.
7. After processing all characters, pop all remaining operators from the stack and append them to postfix.

Postfix Evaluation:

1. Initialize an empty stack.
2. Scan the postfix expression from left to right.
3. If the scanned character is an operand (assumed to be digit), push it onto the stack.
4. If the scanned character is an operator, pop two operands from the stack.
5. Perform the operation with the popped operands and push the result back onto the stack.
6. After the expression is completely scanned, the stack will have one element which is the result.

Input:

Note: Write input you are giving to your program

Output:

Note: Write output of your program

Conclusion:

Thus, we implemented expression conversion as infix to postfix and its evaluation using stack.