EXPRESSION
(infix to postfix\prefix conversion)

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. **Infix notation**
2. **Prefix notation**
3. **Postfix notation**

Infix to postfix conversion algorithm

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C++ function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction:

add(A, B)

Note that the operator add (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression, and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. Human beings are quite used to work with mathematical expressions in **infix notation**, which is rather complex.

Using **infix notation**, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first.

But in a **postfix expression** operands appear before the operator, so there is no need for operator precedence and other rules.

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Algorithm

1. Push “(” onto stack, and add“)” to the end .
2. Scan from left to right and repeat Steps 3 to 6 for each element until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator $\otimes$ is encountered, then:
   (a) Repeatedly pop from stack, if not its precedence higher precedence than $\otimes$.
   (b) Add $\otimes$ to stack.
6. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from stack until a left parenthesis is encountered.
   (b) Remove the left parenthesis.
7. Exit.

For, example consider the following arithmetic **Infix** to **Postfix** expression

(A+(B*C-(D/E^F)*G)*H)

ch	STACK	Output (postfix)
	#	
(	#(	
A	#(	A
+	#(+	A
(	#(+(	A
B	#(+(	AB
*	#(+(*	AB
C	#(+(*	ABC
-	#(+(-	ABC*
(	#(+(-(	ABC*D
D	#(+(-(	ABC*D

ch	STACK	Output(postfix)
/	#(+(-(/	ABC*D
E	#(+(-(/	ABC*DE
^	#(+(-(/^	ABC*DE
F	#(+(-(/^	ABC*DEF
)	#(+(-	ABC*DEF^/
*	#(+(-*	ABC*DEF^/
G	#(+(-*	ABC*DEF^/G
)	#(+	ABC*DEF^/G*
*	#(+*	ABC*DEF^/G*-
H	#(+(*	ABC*DEF^/G*-H
)	#	ABC*DEF^/G*-H*+

CONVERTING INFIXTO PRETFIX EXPRESSION

The rules to be remembered during infix to postfix conversion are:

1. **Reading Expression from “right to left” character by character.**
2. **We have Input, Prefix_Stack & Stack.**
3. **Now converting this expression to Prefix.**

For, example consider the following arithmetic **Infix** to **Pretfix** expression

((A+B) * (C+D) / (E-F)) + G

Input	Prefix_Stack	Stack
G	G	Empty
+	G	+
)	G	+)
)	G	+))

F	G F	+))
-	G F	+)) -
E	G F E	+)) -
(	G F E -	+)
/	G F E -	+) /
)	G F E -	+) /)
D	G F E – D	+) /)
+	G F E – D	+) /) +
C	G F E – D C	+) /) +
(	G F E – D C +	+) /
*	G F E – D C +	+) / *
)	G F E – D C +	+) / *)
B	G F E – D C + B	+) / *)
+	G F E – D C + B	+) / *) +
A	G F E – D C + B A	+) / *) +
(	G F E – D C + B A +	+) / *

(	G F E – D C + B A + * /	+
Empty	G F E – D C + B A + * / +	Empty

((A+B) * (C+D) / (E-F)) + G

- Now pop-out Prefix_Stack to output (or simply reverse).

Prefix expression is

+ / * + A B + C D – E F G

Note:

The priority of operations is :

1. **( ) is higher priority**
2. **^**
3. **\ ***
4. **+ - is lower priority**