

Expressions

An expression is a collection of operators and operands that represents a specific value.

Here, 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 a direct value or variable or address of memory location.

Expression Types

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

- Infix Expression
- Postfix Expression
- Prefix Expression

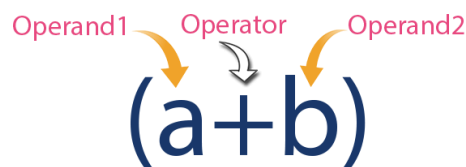
Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



Postfix Expression (Reverse Polish Notation)

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



Prefix Expression (Polish Notation)

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Example



Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, we can use the following steps:














1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
5. If the reading symbol is operator (+, -, *, / etc.), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression

$$(A + B) * (C - D)$$

The given infix expression can be converted into postfix expression using Stack data Structure as follows:

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY 	EMPTY
(Push '(' 	EMPTY
A	No operation Since 'A' is OPERAND 	A
+	'+' has low priority than '(' so, PUSH '+' 	A
B	No operation Since 'B' is OPERAND 	A B
)	POP all elements till we reach '(' POP '+' POP '(' 	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*' 	A B +
(PUSH '(' 	A B +
C	No operation Since 'C' is OPERAND 	A B + C
-	'-' has low priority than '(' so, PUSH '-' 	A B + C
D	No operation Since 'D' is OPERAND 	A B + C D
)	POP all elements till we reach '(' POP '-' POP '(' 	A B + C D -
\$	POP all elements till Stack becomes Empty 	$A B + C D = 20$

The final Postfix Expression is as follows:

$$A B + C D - *$$

Postfix Expression Evaluation using Stack Data Structure

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using stack data structure we can use the following steps:

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.), then perform two pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.










Example

Consider the following Expression:

Infix Expression $(5 + 3) * (8 - 2)$

Postfix Expression $5\ 3\ +\ 8\ 2\ -\ *$

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	<code>value1 = pop()</code> <code>value2 = pop()</code> <code>result = value2 + value1</code> <code>push(result)</code> 	<code>value1 = pop(); // 3</code> <code>value2 = pop(); // 5</code> <code>result = 5 + 3; // 8</code> <code>Push(8)</code> $(5 + 3)$
8	push(8) 	$(5 + 3)$
2	push(2) 	$(5 + 3)$
-	<code>value1 = pop()</code> <code>value2 = pop()</code> <code>result = value2 - value1</code> <code>push(result)</code> 	<code>value1 = pop(); // 2</code> <code>value2 = pop(); // 8</code> <code>result = 8 - 2; // 6</code> <code>Push(6)</code> $(8 - 2)$ $(5 + 3), (8 - 2)$
*	<code>value1 = pop()</code> <code>value2 = pop()</code> <code>result = value2 * value1</code> <code>push(result)</code> 	<code>value1 = pop(); // 6</code> <code>value2 = pop(); // 8</code> <code>result = 8 * 6; // 48</code> <code>Push(48)</code> $(6 * 8)$ $(5 + 3) * (8 - 2)$
\$ End of Expression	<code>result = pop()</code> 	Display (result) 48 As final result

Infix Expression $(5 + 3) * (8 - 2) = 48$

Postfix Expression $5\ 3\ +\ 8\ 2\ -\ *$ value is **48**