APPLICATION OF STACK

Arithmetic expressions:

Any valid combination of arithmetic operators, variables and constants is called an arithmetic expression.

→Arithmetic expressions can be represented using three different notations:

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

Infix notation:

→*In infix notation* the operators appear between the operands.

→This notation is the usual way of representing arithmetic expressions in the real world.

For example:

A+B, 5/3*4, x – y + z / w, (a + b) * c

All he above expressions are called infix arithmetic expressions.

→Infix notations needs extra information to make the order of evaluation of the operators clear: i.e. rules must be built into a programming language about operator precedence and associativity and brackets to allow users to override these rules.

Prefix notation:

→In prefix notation the operators appear before the operands.

→This is a special notation to represent arithmetic expressions, which is also called *Polish-notation.*

→Infix arithmetic expressions are absolutely bracket or parenthesis free.

For example:

infix expression	Prefix expression
A+B	+AB
A-B/C	-A/BC
(A+B)*C	* + AB C
A/B-C*D^E	- / AB*C^DE

Steps to convert infix to prefix expression:

Step-1→completely parenthesize (apply bracket) the whole infix expression according to the precedence of operators.

Step-2→replace each left-bracket (parenthesis) by the corresponding enclosed operator.

Step-3→finally remove all the right brackets (parenthesis).

Example→1: Input infix expression: A + B

Step-1: (A + B)

Step-2: + A B)

Step-3: + A B ←The final prefix expression.

Example→2: Input infix expression: A + B / C

Step-1: (A + (B / C))

Step-2: + A / B C))

Step-3: + A / B C ←The final prefix expression.

Example→3: Input infix expression: A * (B / C) + D ^ E

Step-1: ((A * (B / C)) + (D ^ E))

Step-2: + * A / B C)) ^ D E))

Step-3: + * A / B C ^ D E ←The final prefix expression.

NOTE:

→You can observe infix arithmetic expressions are completely bracket free.

→The relative position of the operators determines the precedence i.e. order of evaluation.

→You need not to recognize or remember the precedence of operators to evaluate an expression.

→Prefix expressions are evaluated from left to right, where the operators act on the two nearest operands on the right.

→Although prefix operators are evaluate from left to right, they use values to their right, and if these values themselves involve in other computations then this changes the order that the operators have to be evaluated.

For example: / * 4 + 5 3 2 = 16

In the above example, although division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division, similarly the addition has to happen before the multiplication.

→Therefore like infix expressions, still prefix expressions require repetitive scanning of the expression from left to right that consumes more execution time for evaluation.

Postfix notation:

→This is also called reverse polish-notation.

→In this notation the operators appear after the operands.

For example:

Infix expression	Postfix expression
A+B	A B +
A-B/C	A B C / -
(A+B)*C	A B + C *
A/B-C*D^E	A B / C D E ^ * -

→The order of evaluation of operators is always left to right and brackets can't be used to change the order.

→Operators act on the values immediately to the left of them.

→Because postfix operators use values to their left, any values involving computations will already have been calculated as we go left to right, and so the order of evaluation of the operators is not disrupted like in prefix expression.

For example: 4 5 3 + * 2 / = 16

If you will move from left side of the above postfix expression, after reading 4, 5 and 3, the + operator comes, so + is evaluated on 5 and 3 i.e. 5+3 = 8, then next * operator comes which is evaluate on 4 and 8 i.e. 4 * 8 = 32, then / operator comes which is evaluate on 32 and 2 i.e. 32/2 = 16. So the final value of the above expression is 16 which we got by only one time scanning of expression.

→You can observe infix arithmetic expressions are completely bracket free.

→The relative position of the operators determines the precedence i.e. order of evaluation.

→You need not to recognize or remember the precedence of operators to evaluate an expression.

→The postfix notation has more advantage then prefix notation because it does not require repetitive scanning of the expression for evaluation.

→A postfix expression can be evaluated by only one go from left to right.

Steps to convert infix to postfix expression:

Step-1→Completely parenthesize (apply bracket) the whole expression according to the precedence or order evaluation of operators.

Step-2→Replace each right-bracket by the corresponding enclosed operator.

Step-3→Finally remove all the left brackets (parenthesis).

Example→1: Input infix expression: A + B

Step-1: (A + B)

Step-2: (A B +

Step-3: A B + ←The final postfix expression.

Example→2: Input infix expression: A + B / C

Step-1: (A + (B / C))

Step-2: (A (B C / +

Step-3: A B C / + ←The final postfix expression.

Example→3: Input infix expression: A * (B / C) + D ^ E

Step-1: ((A * (B / C)) + (D ^ E))

Step-2: ((A (B C / * (D E ^ +

Step-3: A B C / * D E ^ + ←The final postfix expression.

NOTE:

→You can observe postfix arithmetic expressions are also completely bracket (parenthesis) free.

→The relative position of the operators determines the precedence i.e. order of evaluation.

Advantage of representing infix expressions using special notations (infix or postfix):

→Arithmetic expressions represented in infix or postfix notation are completely parenthesis (bracket) free.

→Always evaluated from left to right.

→We need not to remember or recognize the precedence of operators in order to evaluate an expression.

→The relative position of the operators determine the order of evaluation.

→Evaluation of expressions in computer becomes faster in comparison to infix expressions.

Application of stacks:

→Stacks are used in those applications where data items need to be processed or retrieved exactly in the reverse order in which they are inserted or stored.

→Used in implementation of recursion.

→Conversion of infix to prefix or postfix expressions.

→Evaluation of prefix or postfix expressions.

Infix to postfix conversion algorithm using stack

/*Input: E is an infix arithmetic expression */

/*Output: An equivalent postfix expression G

/*This algorithm uses a stack to temporarily store the left brackets '('and operators during conversion process /*

Algorithm Infix_to_postfix(E) /* E is an infix expression */

```
1. Append a right bracket  ')' at end of the expression E
2. PUSH  left bracket '(' on the stack
3. Scan E from left to right, read one symbol at time and store the symbol in
   variable S
4. Repeat steps 5 to 9  WHILE the stack is not empty
5. If  S is an operand  then write the symbol S onto the output expression
6. If S is a left bracket '(' , then PUSH it onto the stack
7. If S is an operator then
        a) Repeatedly POP the stack until the popped operator's precedence
           is greater than equal to the scanned operator S and the popped
           element is not a left bracket.
           →Write the popped operators onto output expression
        b) Again push the final popped item, then PUSH the current scanned
           symbol S
8. If S is a right bracket ')' , then
```

a) **Repeatedly POP the stack and write the popped operators to output until a left bracket '(' is not popped.**

b) **Discard final popped left bracket '(' , don't write that on output.**

9. **Scan the next symbol S**
10. **After the step 4 loop is over, the final output is the required postfix expression.**
11. **STOP**

Infix to postfix conversion procedure with an example

Let's consider the following input infix expression E:

E→ A+ B * C - (D + E * K ^ F / M) * N

Append a ')' at right most side of E as show below:

E→ A+ B * C - (D + E * K ^ F / M) * N)

Push a left bracket '(' onto the stack as show in step-1 of the table below:

Then:

E→ A+ B * C - (D + E * K ^ F / M) * N

$\uparrow$

Scan or read the above expression from A(left to right)

Step No.	Current symbol scanned	STACK	OUTPUT EXPRESSION
1		(	
2	A	(	A
3	+	(	A
4	B	(+	A B
5	*	(+ *	A B
6	C	(+ *	A B C
7	-	(-	A B C * +
8	(	(- (	A B C * +
9	D	(- (	A B C * + D
10	+	(- (+	A B C * + D
11	E	(- (+	A B C * + D E
12	*	(- (+ *	A B C * + D E
13	K	(- (+ *	A B C * + D E K
14	^	(- (+ * ^	A B C * + D E K
15	F	(- (+ * ^	A B C * + D E K F
16	/	(- (+ /	A B C * + D E K F ^ *
17	M	(- (+ /	A B C * + D E K F ^ * M
18	)	(-	A B C * + D E K F ^ * M / +

19	*	(- *	A B C * + D E K F ^ * M / +
20	N	(- *	A B C * + D E K F ^ * M / + N
21	)		A B C * + D E K F ^ * M / + N * -
		STOP	Postfix expr→ last output expression

E→ A+ B * C- (D + E * K ^ F / M) * N)

↑ STOP here

Evaluation of postfix expression using stack

Algorithm Evaluate_postfix(E) /* E is an input postfix expression

```
1. Create an empty stack
2. Scan E from left to right, read one symbol at time and store the symbol in
   variable S
3. Repeat steps 4 to 6  until all the symbol in E does not vanish
4. If S is an operand, then PUSH  S onto the stack
5. If S is an operator, then POP two items from stack
        a)  Operand_2←1st popped item
            Operand_1←2nd popped item
        b)  Evaluate the operator S on the operand_1 & operand_2
            result ← operand_1  S(operator)  operand_2
            PUSH the result onto the stack
6. Scan the next symbol S
7. After termination of step 3 loop, finally POP the stack to print the final
   result of the expression.
8. STOP
```

Example: To evaluate a postfix expression using stack

Input infix expression: A + (B - C) * D / E ^ F = 11

The value of A→5 , B→9 , C → 6 , D→4 , E→ 2 , F→1

After putting the above values: 5 + (9 – 6) * 4 / 2 ^ 1 → 11

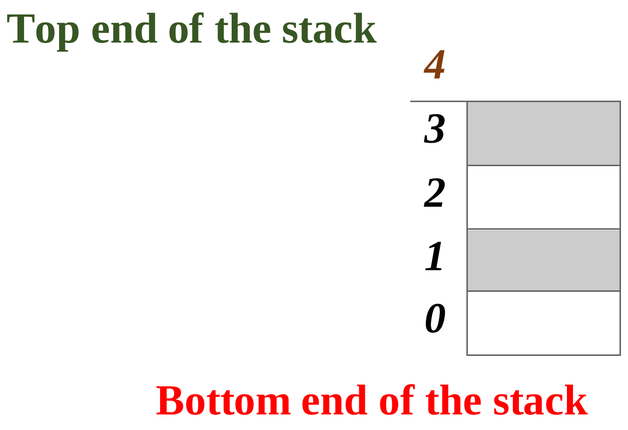
Equivalent postfix expression: A B C - D * E F ^ / +

After putting values: 5 9 6 - 4 * 2 1 ^ / +

We will evaluate the following postfix expression using stack:

E → 5 9 6 - 4 * 2 1 ^ / +

Create an empty stack:

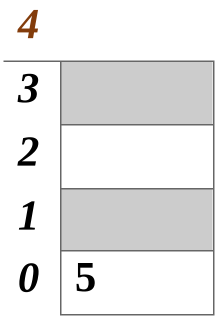


Start the scanning the expression E, by reading only one symbol at a time from left most side of E.

Step-1→ scan 1st symbol from left

S←5 (operand), so PUSH(5)

E → 5 9 6 - 4 * 2 1 ^ / +

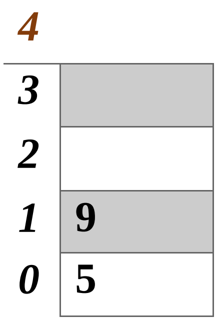


(Status of stack)

Step-2→ scan 2nd symbol from left

S←9 (operand), so PUSH(9)

E → 5 9 6 - 4 * 2 1 ^ / +

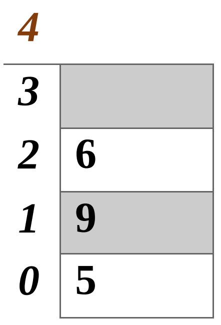


(Status of stack)

Step-3→ scan 3rd symbol from left

S←'6' (operand), PUSH(6)

E →5 9 6 - 4 * 2 1 ^ / +



(Status of stack)

Step-4→ scan 4th symbol from left

S←'-' (operator), don't push

E →5 9 6 - 4 * 2 1 ^ / +

Operand_2←6 (1st popped item)

Operand _1←9 (2nd popped item)

Evaluate S: Result← 9 - 6 = 3

PUSH(result)→ PUSH(3)

4	
3	
2	
1	3
0	5

(Status of stack)

Step-5→ scan 5th symbol from left

S←'4' (operator), PUSH(4)

E →5 9 6 - 4 * 2 1 ^ / +

(Status of stack)

4	
3	
2	4
1	3
0	5

Step-6→ scan 6th symbol from left

S←'*' (operator), don't PUSH

E →5 9 6 - 4 * 2 1 ^ / +

Operand_2←4 (1st popped item)

Operand _1←3 (2nd popped item)

Evaluate S: Result← 3 * 4 = 12

PUSH(result)→ PUSH(12)

4	
3	
2	
1	12
0	5

(Status of stack)

Step-7→ scan 7th symbol from left

S←'2' (operand), so PUSH(2)

E →5 9 6 - 4 * 2 1 ^ / +

(Status of stack)

4	
3	2
2	12
1	5
0	5

Step-8→ scan 8th symbol from left

S←'1' (operand), PUSH(1)

E →5 9 6 - 4 * 2 1 ^ / +

(Status of stack)

4	
3	1
2	2
1	12
0	5

Step-9→ scan 9th symbol from left

S←'^' (operator), don't PUSH

E →5 9 6 - 4 * 2 1 ^ / +

Operand_2←1 (1st popped item)

Operand _1←2 (2nd popped item)

Evaluate S: Result← 2 ^ 1 = 2

PUSH(result)→ PUSH(2)

4	
3	
2	2
1	12
0	5

(Status of stack)

Step-10→ scan 10th symbol from left

S←'/' (operator), don't PUSH

E →5 9 6 - 4 * 2 1 ^ / +

Operand_2←2 (1st popped item)

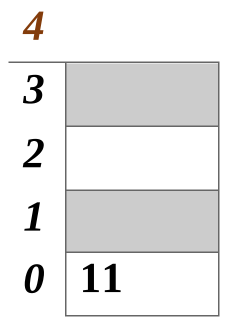
Operand _1←12 (2nd popped item)

Evaluate S: Result← 12 / 2 = 6

PUSH(result)→ PUSH(6)

4	
3	
2	
1	6
0	5

(Status of stack)



Step-11→ scan 11th symbol from left

S←'+' (operator), don't PUSH

E →5 9 6 - 4 * 2 1 ^ / +

Operand_2←6 (1st popped item)

Operand _1←5 (2nd popped item)

Evaluate S: Result← 5 + 6 = 11

PUSH(result)→ PUSH(11)

(Status of stack)

E →5 9 6 - 4 * 2 1 ^ / + (no more symbols to be scanned)

Step-12→ Now there are no more symbols in E to be scanned,

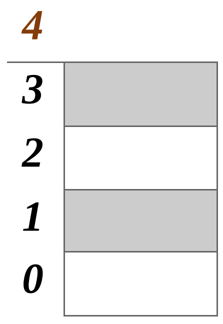
So stop scanning the expression E

Finally pop the stack and display the result

Result←POP the stack

Print(result) ← 11

Now stack is empty



(now stack is empty)