Infix, prefix and postfix data structures

Chapter 4

Prepared by Assist . Prof .
Imad Matti
AL-mamoon University College / Cyber Security Engineering Department

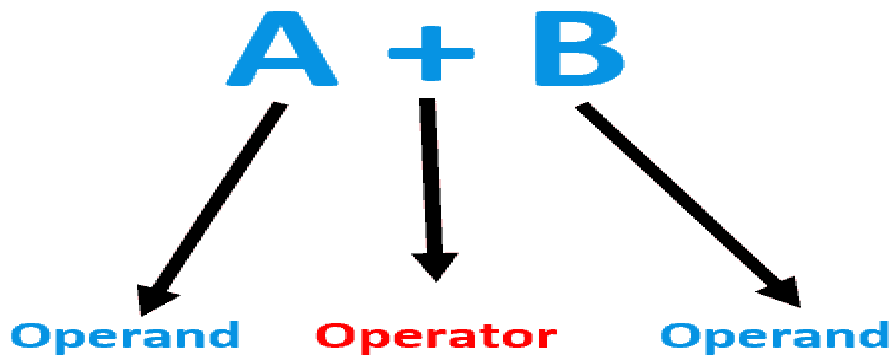
2024

Infix, Postfix and Prefix Expressions/Notations

There are three common expression notations: **infix**, **prefix**, **and** **postfix**.

1. Infix Expressions

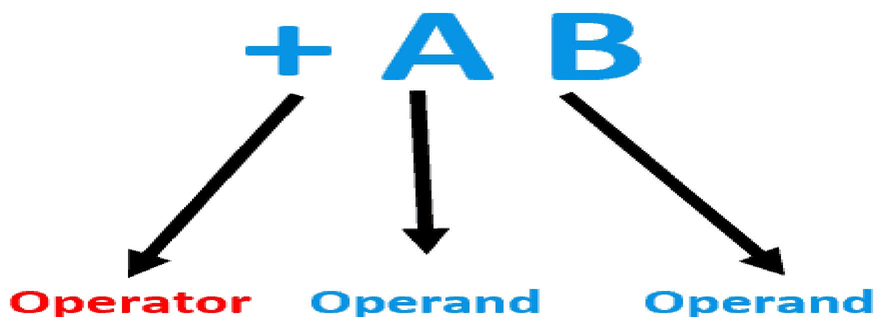
the operator "+" appears between the operands A and B in the expression "A + B". The following figure depicts the example:



2. Prefix Expressions

Prefix expressions, also known as Polish notation, place the operator before the operands.

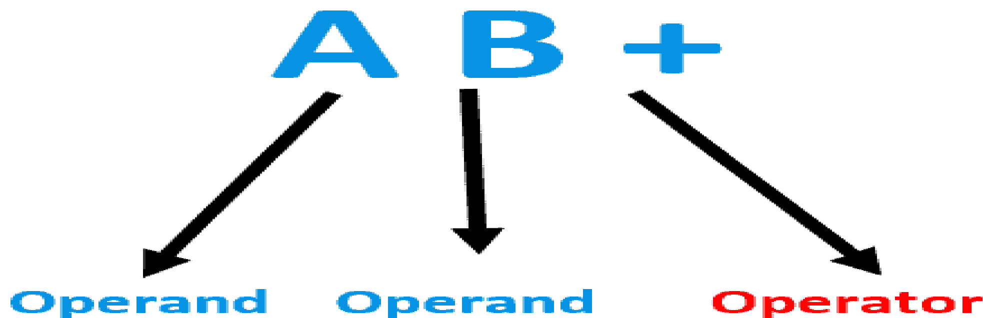
For example, the expression *"+ A B"*, we place the *"+"* operator before the operands *A* and *B*, as demonstrated in the image next:



3. Postfix Expressions

Postfix expressions, also known as reverse Polish notation, where we place the operator after the operands.

For instance, in the expression *"A B +"*, the *"+"* we place the operator after the operands *A* and *B*. The figure next depicts the example:



1. Infix Expressions

Infix expressions are mathematical expressions where the **operator is placed between its operands**. This is the most common mathematical notation used by humans. For example, the expression "2 + 3" is an infix expression, where the operator "+" is placed between the operands "2" and "3".
Infix notation is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently. This is because the order of operations must be taken into account, and parentheses can be used to override the default order of operations.

Common way of writing Infix expressions:

- Infix notation is the notation that we are most familiar with. For example, the expression "2 + 3" is written in infix notation.
- In infix notation, operators are placed between the operands they operate on. For example, in the expression "2 + 3", the addition operator "+" is placed between the operands "2" and "3".
- Parentheses are used in infix notation to specify the order in which operations should be performed. For example, in the expression "(2 + 3) * 4", the parentheses indicate that the addition operation should be performed before the multiplication operation.

Operator precedence rules:

Infix expressions follow operator precedence rules, which determine the order in which operators are evaluated. For example, multiplication and division have higher precedence than addition and subtraction. This means that in the expression "2 + 3 * 4", the multiplication operation will be performed before the addition operation.

Here's the table summarizing the operator precedence rules for common mathematical operators:

Operator	Precedence
Parentheses **()**	Highest
Exponents **^**	High
Multiplication *****	Medium
Division **/**	Medium
Addition **+**	Low
Subtraction **-**	Low

Evaluating Infix Expressions

Evaluating infix expressions requires additional processing to handle the order of operations and parentheses. First convert the **infix expression** to **postfix notation**. This can be done using a stack or a recursive algorithm. Then evaluate the postfix expression.

Advantages of Infix Expressions

- More natural and easier to read and understand for humans.
- Widely used and supported by most programming languages and calculators.

Disadvantages Infix Expressions

- Requires parentheses to specify the order of operations.
- Can be difficult to parse and evaluate efficiently.

Prefix Expressions (Polish Notation)

Prefix expressions are also known as **Polish notation**, are a mathematical notation where the operator precedes its operands. This differs from the more common **infix notation**, where the operator is placed between its operands.

In prefix notation, the operator is written first, followed by its operands. For example, the infix expression "a + b" would be written as "+ a b" in prefix notation.

Evaluating Prefix Expressions

Evaluating prefix expressions can be useful in certain scenarios, such as when dealing with expressions that have a large number of nested parentheses or when using a stack-based programming language.

Advantages of Prefix Expressions

- No need for parentheses, as the operator always precedes its operands.
- Easier to parse and evaluate using a stack-based algorithm.
- Can be more efficient in certain situations, such as when dealing with expressions that have a large number of nested parentheses.

Disadvantages of Prefix Expressions

- Can be difficult to read and understand for humans.
- Not as commonly used as infix notation.

Postfix Expressions (Reverse Polish Notation)

Postfix expressions are also known as **Reverse Polish Notation (RPN)**, are a mathematical notation where the **operator follows its operands**. This differs from the more common infix notation, where the operator is placed between its operands.

In postfix notation, operands are written first, followed by the operator. For example, the infix expression "5 + 2" would be written as "5 2 +" in postfix notation.

Evaluating Postfix Expressions (Reverse Polish Notation)

Evaluating postfix expressions can be useful in certain scenarios, such as when dealing with expressions that have a large number of nested parentheses or when using a stack-based programming language.

Advantages of Postfix Notation

- Also eliminates the need for parentheses.
- Easier to read and understand for humans.
- More commonly used than prefix notation.

Disadvantages of Postfix Expressions

- Requires a stack-based algorithm for evaluation.
- Can be less efficient than prefix notation in certain situations.

Convert Infix expression to Postfix expression

- **Infix expression:** The expression of the form “a operator b” (a + b) i.e., when an operator is in-between every pair of operands.
  Postfix expression: The expression of the form “a b operator” (ab+) i.e., When every pair of operands is followed by an operator.

Examples:
Input (infix): *A + B * C + D*
Output (postfix): *ABC*+D+*
Input (infix): *((A + B) – C * (D / E)) + F*
Output (postfix): *AB+CDE/*-F+*

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.
Consider the expression: **a + b * c + d**

- The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix (or prefix) form before evaluation.

The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

How to convert an Infix expression to a Postfix expression?

To convert infix expression to postfix expression, use the ***stack data structure****. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.*

Conversion from INFIX to POSTFIX Algorithm

Step1:

Scan the infix expression from **left to right** for **tokens** (**operators**, **operands** & **parentheses**) and perform the step 2 to step 5 for each token in the expression.

Step 2:

If token is **operand**, append it in postfix expression.

Step 3:

If token is **left parenthesis (**, Push it in stack.

Step 4:

If token is an **operator**

1. **Pop all the operators** which are of higher or equal precedence then the incoming token and append them (in the same order) to the output expression.
2. After popping out all such operators, **push the new token** on stack.

Step 5:

If **)** which is right parenthesis is found

1. **Pop all the operators** from stack and append them to output string **till** you **encounter the opening parenthesis (** .
2. **Pop the left parenthesis** but don't append it to the output string (postfix notation does not have brackets).

Step 6:

1. When all tokens of infix expression have been scanned, **pop all the elements from the stack** and **append** them to the output string.
2. The output string is the corresponding **postfix notation**.

Below are the steps to implement the above idea:

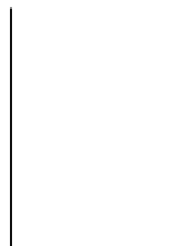
1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
   - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
   - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.

7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Illustration (example 1):

Consider the infix expression **exp = "a+b*c+d"**
and the infix expression is scanned using the iterator **i**, which is initialized as **i = 0**.

1st Step: *Here i = 0 and* **exp[i] = 'a'** *i.e., an operand. So add this in the postfix expression.*
Therefore, ***postfix = "a"****.*

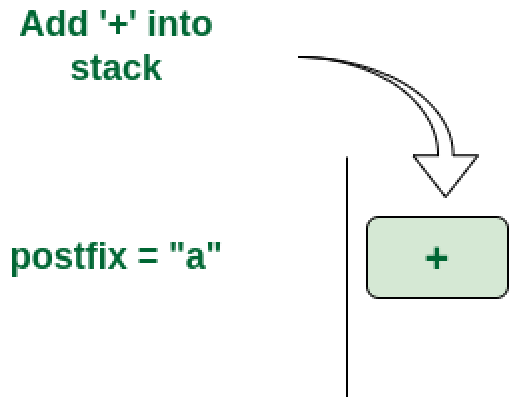


postfix = "a"

'a' is an operand. Add it in postfix expression

Add 'a' in the postfix

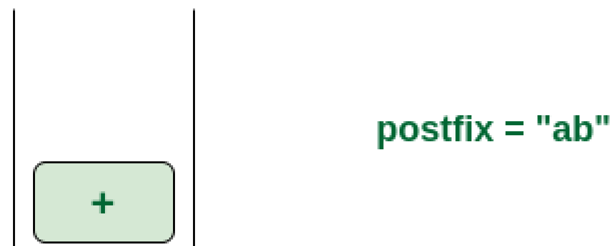
2nd Step: *Here i = 1 and exp[i] =* ***'+'*** *i.e., an operator.*
Push this into the stack.
postfix = "a" *and* ***stack = {+}****.*



Stack is empty. Push '+' into stack

Push '+' in the stack

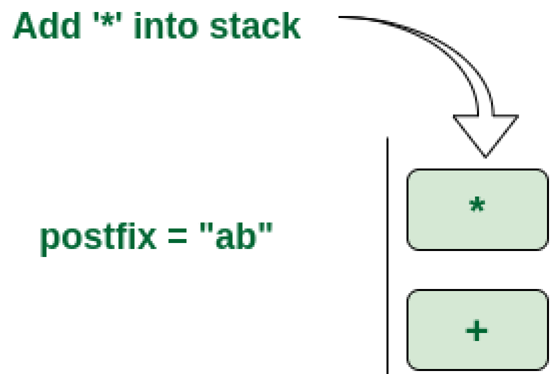
3rd Step: *Now i = 2 and exp[i] = '**b**' i.e., an operand.*
So add this in the postfix expression.
***postfix = "ab"** and **stack = {+}**.*



'b' is an operand. Add it in postfix expression

Add 'b' in the postfix

4th Step: *Now i = 3 and exp[i] = '*' i.e., an operator. Push this into the stack.*
***postfix* = "ab" *and* stack = {+, *}.**

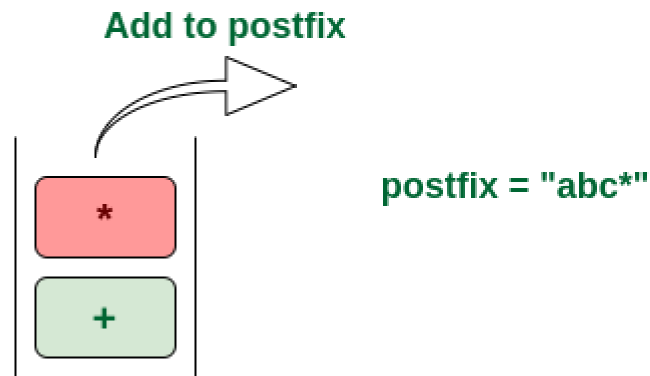


Push '' in the stack*

5th Step: *Now i = 4 and exp[i] = 'c' i.e., an operand. Add this in the postfix expression.* ***postfix* = "abc"** *and* **stack = {+, *}.**

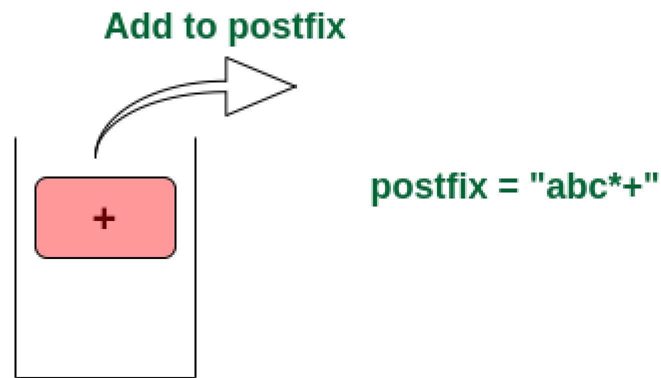
Add 'c' in the postfix

6th Step: *Now i = 5 and exp[i] = '+' i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. '*' is popped and added in postfix. So **postfix = "abc*"** and **stack = {+}**.*



Pop '' and add in postfix*

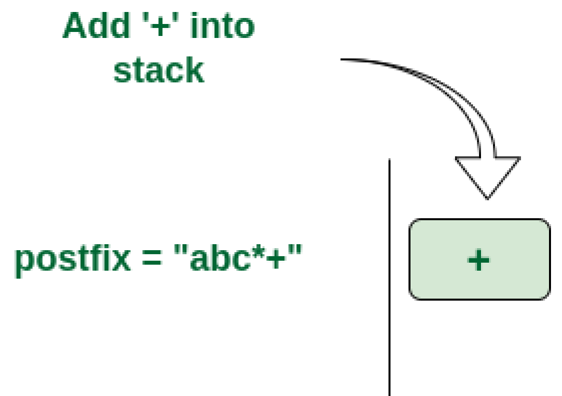
*Now top element is '**+**' that also doesn't have less precedence. Pop it. **postfix = "abc*+"**.*



'+' and stack top has same precedence

Pop ‘+’ and add it in postfix

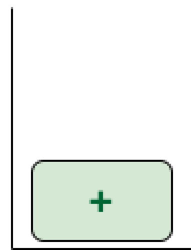
*Now stack is empty. So push **‘+’** in the stack. **stack = {+}**.*



Stack is empty. Push '+' into stack

Push ‘+’ in the stack

7th Step: *Now i = 6 and exp[i] = 'd' i.e., an operand. Add this in the postfix expression.* ***postfix = "abc*+d"****.*

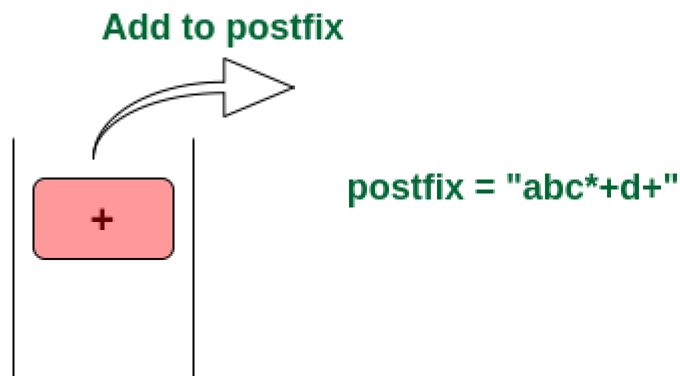


postfix = "abc*+d"

'd' is an operand. Add it in postfix expression

Add 'd' in the postfix

Final Step: *Now no element is left. So empty the stack and add it in the postfix expression.* ***postfix = "abc*+d+"****.*



Nothing left. So pop all operators

Pop '+' and add it in postfix

Example 2:

Let the incoming infix expression be:

A * (B + C) – D / E

To convert it to postfix expression we follow the following stages:

Stage 1:

Stack is empty and we only have the infix expression.

Infix expression	Stack	Postfix expression
A * (B + C) – D / E		

Stage 2:

The first token is **operand A** (operands are appended to the output as it is.

Infix expression	Stack	Postfix expression
* (B + C) – D / E		A

Stage 3:

The token is * since stack is empty it is pushed into the stack.

Infix expression	Stack	Postfix expression
(B + C) – D / E	*	A

Stage 4:

. Next token is **(** the precedence of open parenthesis , when it is to go inside, is maximum.

. But when another operator is to come on the top of **(** then its precedence is least.

Infix expression	Stack	Postfix expression
B + C) – D / E	* (	A

Stage 5:

Next token is B an operand which will go to the output expression as it is.

Infix expression	Stack	Postfix expression
+ C) – D / E	* (	A B

Stage 6:

Next token is **+** operator, we consider the precedence of **top element in the stack** which is **(** . The outgoing precedence of open parenthesis is the least. So **+** gets **pushed into the stack**.

Infix expression	Stack	Postfix expression
C) – D / E	* (+	A B

Stage 7:

Next token is **C** operand, it appended to the output expression.

Infix expression	Stack	Postfix expression
) – D / E	* (+	A B C

Stage 8:

Next token is **)** it means that **pop all the elements from stack** and **append them to the output expression till** we read an opening parenthesis.

Infix expression	Stack	Postfix expression
– D / E	*	A B C +

Stage 9:

Next token is – operator. The precedence of operator on top of the stack is * is more than that of Minus, so **we pop Multiply** and **append it to output expression**. Then **push minus in the stack**.

Infix expression	Stack	Postfix expression
D / E	-	A B C + *

Stage 10:

Next operand is **D** get **appended to the output expression**.

Infix expression	Stack	Postfix expression
/ E	-	A B C + * D

Stage 11:

Next, we will insert the **division** operator into the stack because its precedence is more than that of minus.

Infix expression	Stack	Postfix expression
E	- /	A B C + * D

Stage 12:

The last token is **E** operator, so we insert it to the output expression as it is.

Infix expression	Stack	Postfix expression
	- /	A B C + * D E

Stage 13:

The input expression is complete now. So we pop the stack and append to the output expression.

Infix expression	Stack	Postfix expression
		A B C + * D E / -

Example 3: (converting infix to postfix expression)

infix expression: **A * B- (C + D) + E**

Input Character	Operations on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and Append to Postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End	Pop till Empty		**AB*CD+-E+**

Evaluation of Postfix Expression using Stack:

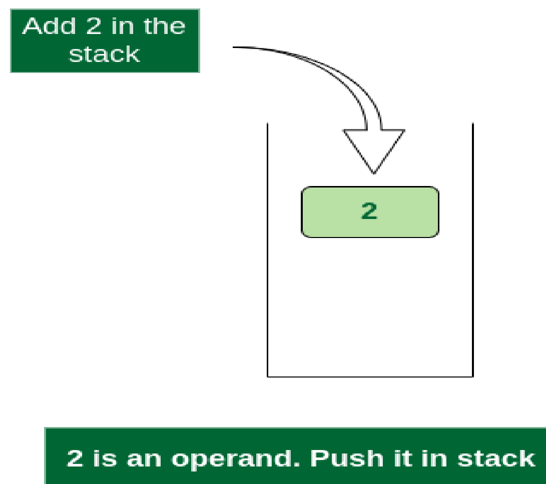
To evaluate a postfix expression we can use a stack.
Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

Illustration:

Follow the below illustration for a better understanding:

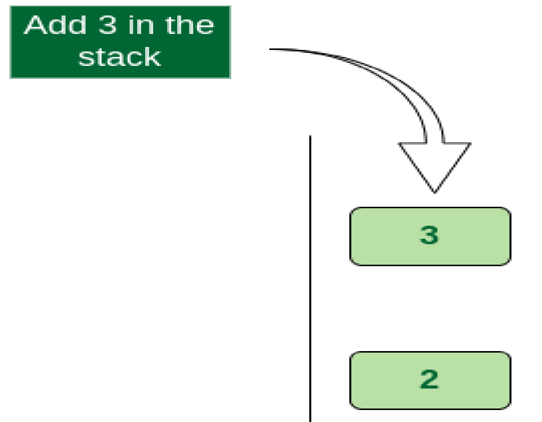
Consider the expression: exp = **"2 3 1 * + 9 -"**

- *Scan* ***2****, it's a number, So push it into stack. Stack contains '2'.*



Push 2 into stack

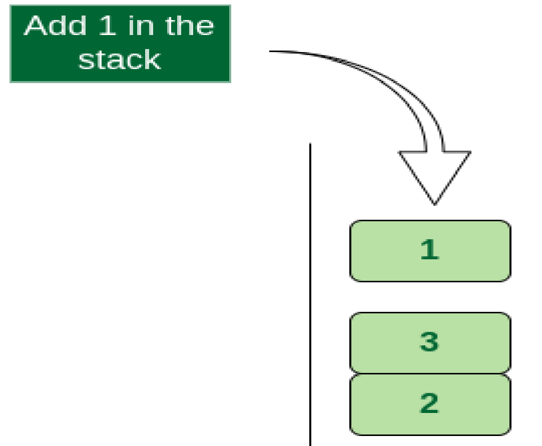
- **Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top)**



3 is an operand. Push it in stack

Push 3 into stack

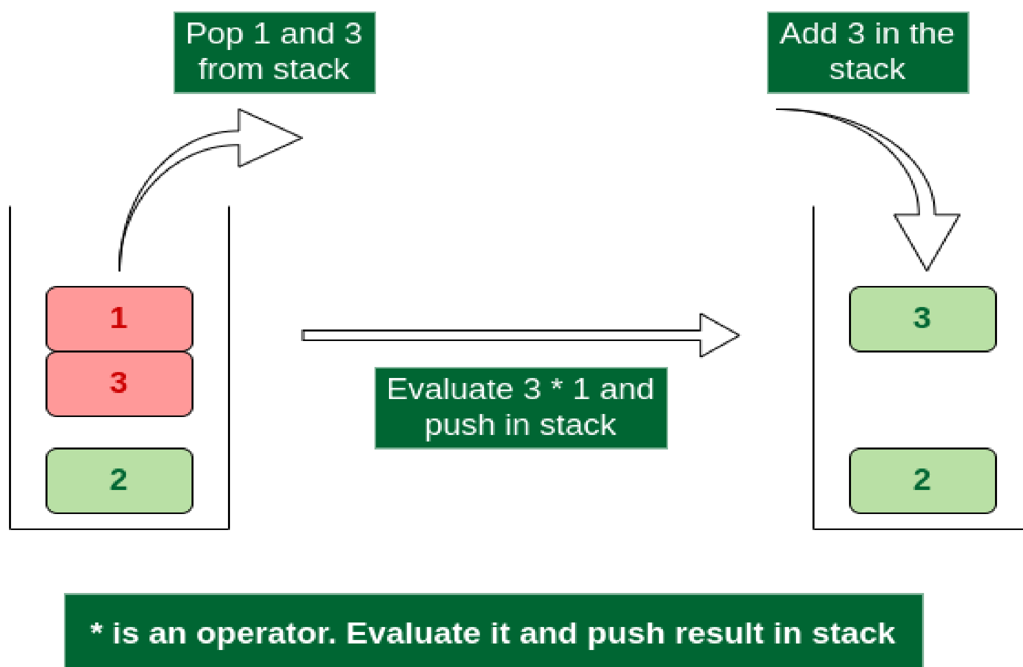
- **Scan 1, again a number, push it to stack, stack now contains '2 3 1'**



1 is an operand. Push it in stack

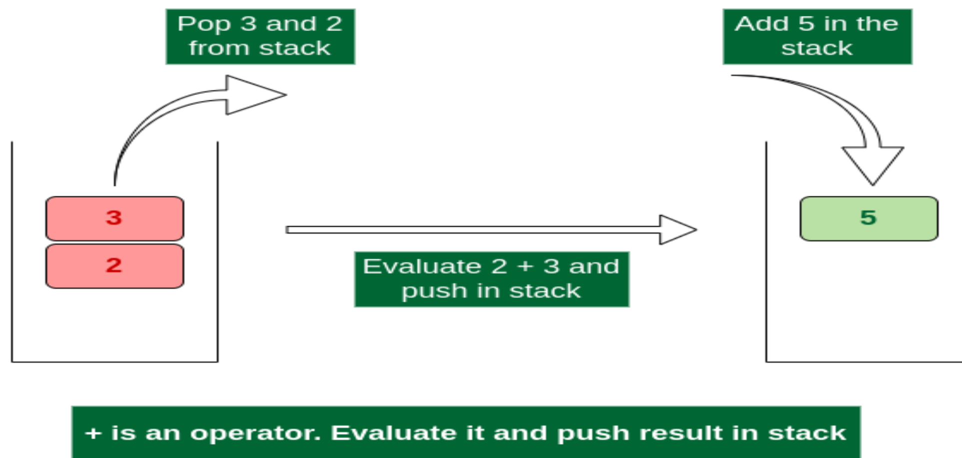
Push 1 into stack

- **Scan *, it's an operator. Pop two operands from stack, apply the * operator on operands. We get 3*1 which results in 3. We push the result 3 to stack. The stack now becomes '2 3'.**



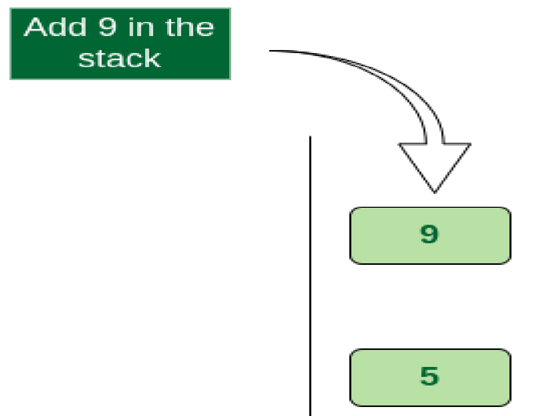
Evaluate * operator and push result in stack

- **Scan +, it's an operator. Pop two operands from stack, apply the + operator on operands. We get 3 + 2 which results in 5. We push the result 5 to stack. The stack now becomes '5'.**



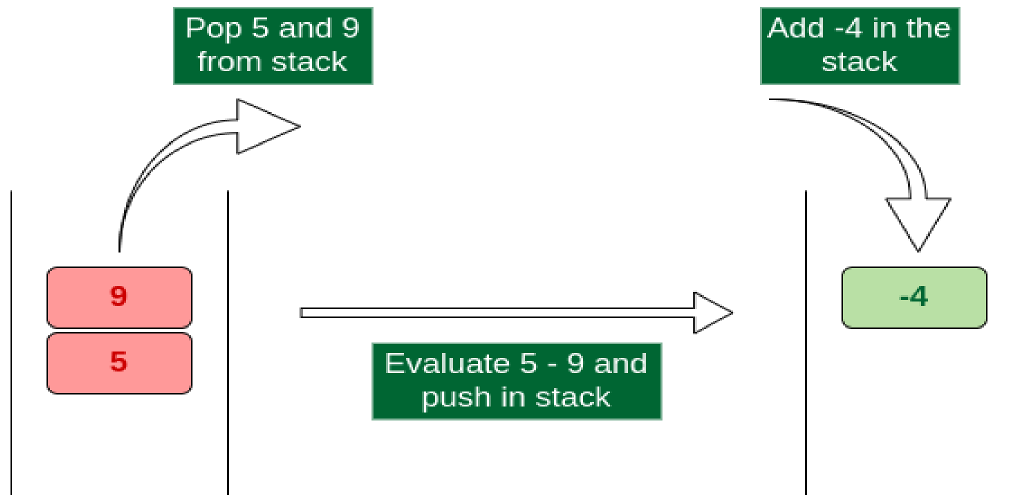
Evaluate + operator and push result in stack

- **Scan 9, it's a number. So we push it to the stack. The stack now becomes '5 9'.**



Push 9 into stack

- Scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result -4 to the stack. The stack now becomes '-4'.



Evaluate '-' operator and push result in stack

- **There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).**

So the result becomes -4.

Follow the steps mentioned below to evaluate postfix expression using stack:

- **Create a stack to store operands (or values).**
- **Scan the given expression from left to right and do the following for every scanned element.**
    - **If the element is a number, push it into the stack.**
    - **If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.**
- **When the expression is ended, the number in the stack is the final answer.**

Example 2:evaluate the postfix expression 234*+

Input	Stack	Postfix evaluation
2 3 4 * +	*empty*	Push 2 into stack
3 4 * +	2	Push 3 into stack
4 * +	3 2	Push 4 into stack
* +	4 3 2	Pop 4 & pop 3 from stack and do 3 *4 , push 12 into stack
+	12 2	Pop 12 & Pop 2 from stack do 2 + 12, push 14 into stack
	14	

Input	Stack	Postfix evaluation
3 4 * 2 5 * +	*empty*	Push 3 into stack
4 * 2 5 * +	3	Push 4 into stack
* 2 5 * +	4 3	Pop 4 & pop 3 from stack do 3*4, Push 12
2 5 * +	12	Push 2 into stack
5 * +	2 12	Push 5 into stack
* +	5 2 12	Pop 5, Pop 2 from stack do 2*5, Push 10 into stack
+	10 12	Pop 10 & Pop 12 from stack do 12 + 10, push 22 into stack
	22	

the algorithm for postfix evaluation is :

- Read a character
- Convert to int and push
- If the character is an operator pop the stack twice to obtain two operands
- Push the outcome of operation

Examples: convert the following infix expressions to postfix expressions

Input	Stack	Postfix evaluation
2*3 + 4*5	Nothing in stack	
*3+4*5	Nothing in stack	2
3+4*5	*	2
+4*5	*	23
4*5	+	23*
*5	+	23*4
5	*+	23*4
	*+	23*45
	+	23*45*
	Nothing in stack	**23*45*+**

Input	Stack	Postfix evaluation
2-3+4-5*6	Nothing in stack	
-3+4-5*6	Nothing in stack	2
3+4-5*6	-	2
+4-5*6	-	23
4-5*6	+	23-
-5*6	+	23-4
5*6	-	23-4+
*6	-	23-4+5
6	*-	23-4+5

	*-	23-4+56
	-	23-4+56*
	Nothing in stack	23-4+56*-

Input	**Stack**	**Postfix evaluation**
(2-3+4)*(5+6*7)	Nothing in stack	
2-3+4)*(5+6*7)	(	
-3+4)*(5+6*7)	(	2
3+4)*(5+6*7)	(-	2
+4)*(5+6*7)	(-	23
4)*(5+6*7)	(+	23-
)*(5+6*7)	(+	23-4
*(5+6*7)	Nothing in stack	23-4+
(5+6*7)	*	23-4+
5+6*7)	(*	23-4+
+6*7)	(*	23-4+5
6*7)	+(*	23-4+5
*7)	+(*	23-4+56
7)	*+(*	23-4+56
)	*+(*	23-4+567
	*	23-4+567*+
	Nothing in stack	**23-4+567*+***

Convert Infix To Prefix Notation

Given an infix expression, the task is to convert it to a prefix expression.

Infix Expression: *The expression of type* ***a 'operator' b*** *(a+b, where + is an operator) i.e., when the operator is between two operands.*
Prefix Expression: *The expression of type '****operator' a b*** *(+ab where + is an operator) i.e., when the operator is placed before the operands.*

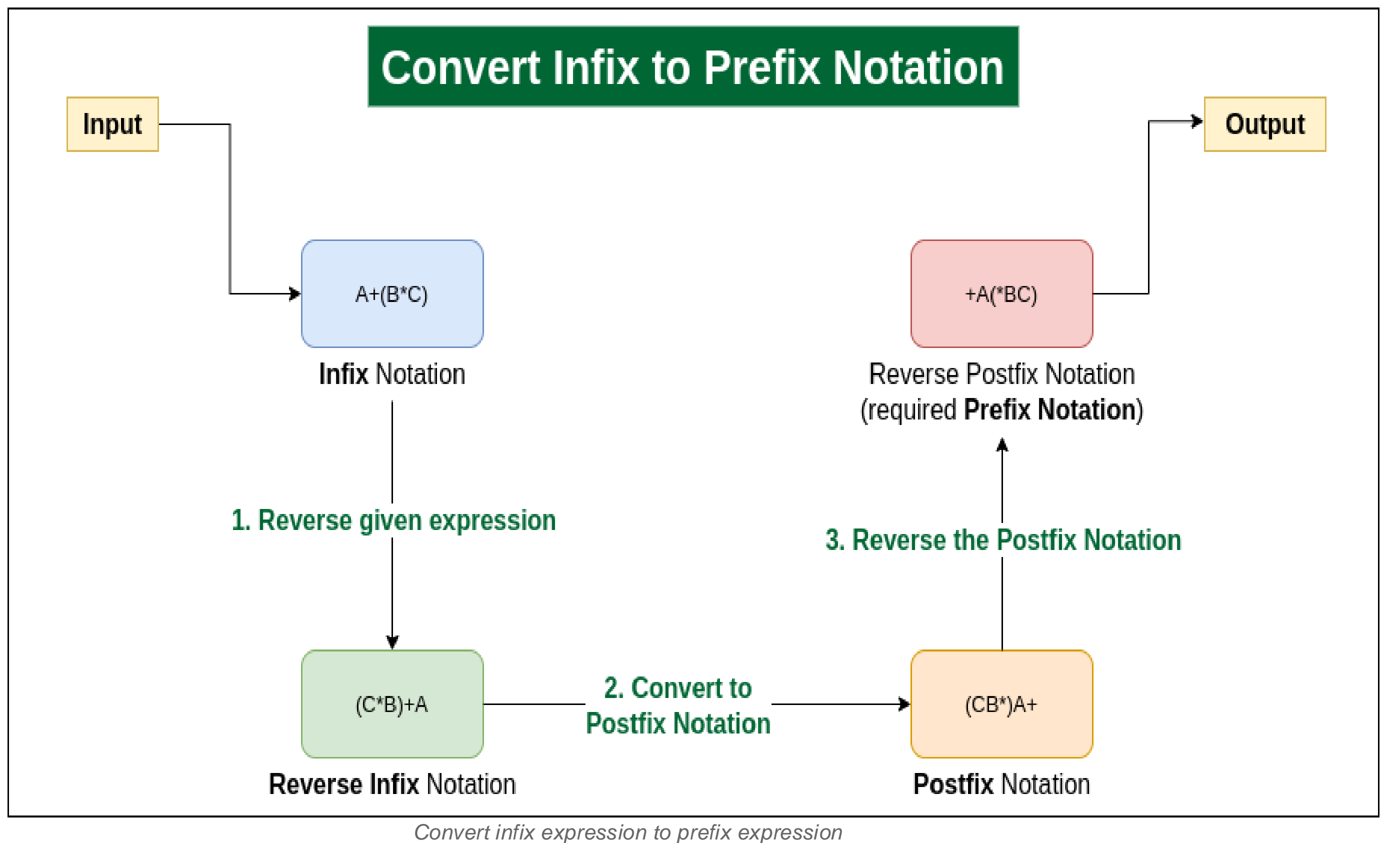
How to convert infix expression to prefix expression?

To convert an infix expression to a prefix expression, we can use the ***stack data structure****. The idea is as follows:*

- ***Step 1:*** *Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.*
- ***Step 2:*** *Convert the reversed* ***infix expression to "nearly" postfix expression****.*
    - *While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.*
- ***Step 3:*** *Reverse the postfix expression.*

The stack is used to convert infix expression to postfix form.

Illustration:
See the below image for a clear idea:



Convert infix expression to prefix expression

Rules for the conversion of infix to prefix expression:

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.
- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the

stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.

- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is ')', then push it into the stack.
- If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
- If the top of the stack is ')', push the operator on the stack.
- At the end, reverse the output.

Conversion of Infix to Prefix using Stack

Consider the following infix expression:

K + L - M * N + (O^P) * W/U/V * T + Q

If we are converting the expression from infix to prefix, we need first to reverse the infix expression.

The Reverse expression would be:

Q + T * V/U/W *) P^O(+ N*M - L + K

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Example 1: (converting infix to prefix expression)

Input expression	stack	prefix expression
Q		Q
+	+	Q
T	+	QT

*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*

L	**++-**	**QTVUWPO^*//*NM*L**
+	**++-+**	**QTVUWPO^*//*NM*L**
K	**++-+**	**QTVUWPO^*//*NM*LK**
		QTVUWPO^*//*NM*LK+-++

The above expression, i.e., QTVUWPO^*//*NM*LK+-++, is not a final expression. We need to reverse this expression to obtain the prefix expression.

Infix to prefix conversion

Expression = **(A+B^C)*D+E^5**
Step 1. Reverse the infix expression.
5^E+D*)C^B+A(
Step 2. Make Every '(' as ')' and every ')' as '('
5^E+D*(C^B+A)
Step 3. Convert expression to postfix form.

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

Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
*(C^B+A)	+	5E^D	Print

(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Step 4. Reverse the expression.

+*+A^BCD^E5

Result

+*+A^BCD^E5

Infix Expression : A+B*(C^D-E)				
Reverse Infix expression:)E-D^C(*B+A				
Reverse brackets: (E-D^C)*B+A				
Token	**Action**	**Result**	**Stack**	**Notes**
(	Push **(** to stack		(	
E	Add **E** to the result	E	(	
-	Push **-** to stack	E	(-	
D	Add **D** to the result	ED	(-	
^	Push **^** to stack	ED	(- ^	
C	Add **C** to the result	EDC	(- ^	
)	Pop **^** from stack and add to result	EDC^	(-	Do process until (is popped from stack
	Pop **-** from stack and add to result	EDC^-	(	
	Pop **(** from stack	EDC^-		
*	Push ***** to stack	EDC^-	*	
B	Add **B** to the result	EDC^-B	*	
+	Pop ***** from stack and add to result	EDC^-B		- has lower precedence than ^
	Push **+** to stack	EDC^-B*	+	
A	Add **A** to the result	EDC^-B*A	+	
	Pop **+** from stack and add to result	EDC^-B*A+		Given expression is iterated, do Process till stack is not Empty, It will give the final result
Prefix Expression (Reverse Result): +A*B-^CDE				

Steps to Convert Infix to Prefix

Step 1: Reverse the Infix Expression
The first step in converting an infix expression to a prefix expression is to reverse the given infix expression. This means you will read the expression from right to left instead of left to right.
Example:
Infix expression: (A + B) * C
Reversed infix expression: C * (B + A)
Step 2: Replace Parentheses
After reversing the infix expression, replace the opening parenthesis “(” with a closing parenthesis “)” and vice versa.
Example:
Reversed infix expression: C *)B + A(
Modified expression: C * (B + A)
Step 3: Obtain the Postfix Expression
The next step is to convert the modified infix expression to a postfix expression. To do this, you’ll use the stack data structure and follow the order of operations (PEMDAS/BODMAS rules).
Example:
Modified expression: C * (B + A)
Postfix expression: C B A + *
Step 4: Reverse the Postfix Expression
Finally, reverse the postfix expression to obtain the prefix expression.

Example:
Postfix expression: C B A + *
Reversed postfix expression (prefix): * + A B C

Example 2:
Infix expression: A + B * C
Step 1:
Reverse the infix expression:
Reversed infix expression: C * B + A
Step 2:
Replace parentheses (none in this example):
Step 3:
Convert to postfix:
Postfix expression: C B * A +
Step 4:
Reverse the postfix expression to get the prefix:
Prefix expression: + A * B C

Home work Quizzes

Select the right answer for the following MCQ questions.

1. When an operand is read, which of the following is done?
a) It is placed on to the output
b) It is placed in operator stack
c) It is ignored
d) Operator stack is emptied

2. What should be done when a left parenthesis '(' is encountered?
a) It is ignored
b) It is placed in the output
c) It is placed in the operator stack
d) The contents of the operator stack is emptied

3. Which of the following is an infix expression?
a) (a+b)*(c+d)
b) ab+c*
c) +ab

d) abc+*

4. What is the postfix expression for the infix expression a-b-c?

a) -ab-c
b) ab – c –
c) – -abc
d) -ab-c

5. What is the postfix expression for the following infix expression?

```
a/b^c-d
```

a) abc^/d-
b) ab/cd^-
c) ab/^cd-
d) abcd^/-

6. Which of the following statement is incorrect with respect to infix to postfix conversion algorithm?

a) operand is always placed in the output
b) operator is placed in the stack when the stack operator has lower precedence
c) parenthesis are included in the output
d) higher and equal priority operators follow the same condition

7. In infix to postfix conversion algorithm, the operators are associated from?

a) right to left
b) left to right
c) centre to left
d) centre to right

8. What is the corresponding postfix expression for the given infix expression?

```
a*(b+c)/d
```

a) ab*+cd/
b) ab+*cd/
c) abc*+/d
d) abc+*d/

9. What is the corresponding postfix expression for the given infix expression?

```
a+(b*c(d/e^f)*g)*h)
```

a) ab*cdef/^*g-h+
b) abcdef^/*g*h*+
c) abcd*^ed/g*-h*+
d) abc*de^fg/*-*h+

10. Which of the following is true about infix, postfix, and prefix expressions?
a) Infix expressions require parentheses for precedence, while postfix and prefix expressions do not.
b) Postfix expressions require parentheses for operator precedence.
c) Prefix expressions cannot be evaluated without parentheses.
d) Infix and postfix expressions are evaluated from right to left.

11. Which data structure is most suitable for converting an infix expression to a postfix expression?
a) Queue
b) Stack
c) Array
d) Tree

12. Convert the following infix expression to the postfix expression
(A + B) * C ?

13. What is the postfix form of the expression A + B * C?
a) + A B * C
b) A B C * +
c) A + B * C
d) A * B + C

Answer: b) A B C * +5.

14. Convert 2 * 3 / (2 – 1) + 5 * 3 into postfix form.

15. Convert (A + B) * (C – D) into postfix form.

infix, Postfix, and Prefix Quizes

1. Infix Expression: (AX + (B * C)) ;
Postfix Expression:
Prefix Expression:

2. Infix Expression: ((AX + (B * CY)) / (D E)) ;
Postfix Expression:
Prefix Expression:

3. Infix Expression: ((A + B) * (C + E)) ;
Postfix Expression:
Prefix Expression:

4. Infix Expression: (AX * (BX * (((CY + AY) + BY) * CX))) ;
Postfix Expression:
Prefix Expression:

5. Infix Expression: ((H * ((((A + ((B + C) * D)) * F) * G) * E)) + J) ;
Postfix Expression:
Prefix Expression:

6. **Infix expression:** *A + B * C + D*
Postfix Expression:
Prefix Expression:

7. Infix expression: *((A + B) – C * (D / E)) + F*

Postfix Expression:
Prefix Expression:

9. Evaluate(find the result) the following infix expression
Input: str = “**100 200 + 2 / 5 * 7 +**”
Output: **757**

Solutions:

Infix Expression: (AX + (B * C)) ;
Postfix Expression: AX B C * +
Prefix Expression: + AX * B C

Infix Expression: ((AX + (B * CY)) / (D E)) ;
Postfix Expression: AX B CY * + D E /
Prefix Expression: / + AX * B CY D E

Infix Expression: ((A + B) * (C + E)) ;
Postfix Expression: A B + C E + *
Prefix Expression: * + A B + C E

Infix Expression: (AX * (BX * (((CY + AY) + BY) * CX))) ;
Postfix Expression: AX BX CY AY + BY + CX * * *
Prefix Expression: * AX * BX * + + CY AY BY CX

Infix Expression: ((H * ((((A + ((B + C) * D)) * F) * G) * E)) + J) ;
Postfix Expression: H A B C + D * + F * G * E * * J +
Prefix Expression: + * H * * * + A * + B C D F G E J

Postfix: *ABC*+D+*

Postfix: *AB+CDE/*-F+*