



# Data Structures and its Applications

---

**Dinesh Singh**

Department of Computer Science & Engineering

# DATA STRUCTURES AND ITS APPLICATIONS

---

## Infix to Postfix and Prefix Expressions – Implementation

**Dinesh Singh**

Department of Computer Science & Engineering

- Consider the sum of A and B expressed as  $A + B$
- This representation is called infix.
- There are two alternate notations for expressing sum of A and B using the symbols A , B and + , these are
  - Prefix :  $+ A B$
  - Postfix :  $A B +$
- The prefixes “Pre” , “post” and “in” refers to the relative position of the operator with respect to the two operands.
- In prefix, operators precedes the two operands
- In postfix, the operator follows the two operands
- In infix, the operator is in between the two operands

### Conversion of Infix to Postfix

- For Example : consider the expression  $A + B * C$
- The evaluation of the above expression requires the knowledge of precedence of operators
- $A + B * C$  can be expressed as  $A + (B * C)$  as multiplication takes precedence over addition
- Applying the rules of precedence the above infix expression can be converted to postfix as follows

$$\begin{aligned} A + B * C &= A + ( B * C ) \\ &= A + ( BC * ) \quad \text{convert the multiplication} \\ &= A ( B C * ) + \quad \text{convert the addition} \\ &= A B C * + \end{aligned}$$

### Conversion of Infix to Prefix

- For Example : consider the expression  $A + B * C$
- Applying the rules of precedence the above infix expression can be converted to prefix as follows

$$\begin{aligned} A + B * C &= A + ( B * C ) \\ &= A + ( * B C ) \quad \text{convert the multiplication} \\ &= + A ( * B C ) + \quad \text{convert the addition} \\ &= + A * B C \end{aligned}$$

Applying the rules of precedence the table shows the conversion of Infix to Postfix and Prefix Expression

Infix	Postfix	Prefix
$A + B * C + D$	$A B C * + D +$	$++ A * B C D$
$(A + B) * (C + D)$	$A B + C D + *$	$* + A B + C D$
$A * B + C * D$	$A B * C D * +$	$+ * A B * C D$
$A + B + C + D$	$A B + C + D +$	$+++ A B C D$
$A \$ B * C - D + E / F / (G + H)$	$A B \$ C * D - E F / G H + / +$	$+ - * \$ A B C D / / E F + G H$
$((A + B) * C - (D - E)) \$ (F + G)$	$A B + C * D E - - F G + \$$	$\$ - * + A B C - D E + F G$
$A - B / (C * D \$ E)$	$A B C D E \$ * / -$	$- A / B * C \$ D E$

$\$$  is the exponentiation operator and its precedence is from right to left  
For example  $A \$ B \$ C = A \$ (B \$ C)$

opstk is the empty stack

while(not end of input)

{

    symb = next input character

    // if the input symbol is an operand , add it to the postfix string

    if(symb is an operand)

        add symb to postfix string

else

{

    // pop the contents of the stack while the precedence of

    // top of the stack is greater than the precedence of the symbol scanned

    //prcd function returns true in this case

    while(!empty(opstk ) and ( prcd(stacktop(opstk),symb))

    {

        topsymb=pop(opstk)

        add topsymb to postfix string

    }

```
/* if prcd function returns false, then
if the stack is empty and symbol is not ')', push symb on to the stack*/
    if( empty(opstk) || symb!=')')
        push(opstk,symb)
    else
        topsymb=pop(opstk); // if symb is ')', pop '(' from the stack
}
while(!empty(opstk)
{
    topsymb=pop(opstk)
    add topsymb to postfix string
}
}
```



## Infix to postfix conversion - algorithm

---

- $\text{prcd}(\text{OP1}, \text{OP2})$  is a function that compares the precedence of the top of the stack (OP1) and the input symbol (OP2) and returns TRUE if the precedence is greater else returns FALSE.
- Some of the return values of prcd function
  - $\text{prcd}(' * ', ' + ')$  returns TRUE :  $\text{prcd}(' * ', ' / ')$  returns TRUE
  - $\text{prcd}(' + ', ' * ')$  returns FALSE :  $\text{prcd}(' / ', ' * ')$  returns TRUE
  - $\text{prcd}(' + ', ' + ')$  returns TRUE :
  - $\text{prcd}(' + ', ' - ')$  returns TRUE
  - $\text{prcd}(' - ', ' - ')$  returns TRUE
  - $\text{prcd}(' \$ ', ' \$ ')$  returns FALSE : exponentiation operator, associatively from right to left
  - $\text{prcd}(' ( ', \text{op})$  ,  $\text{prcd}(\text{op} ' ( ')$  returns FALSE for any operator
  - $\text{prcd}(\text{op} , ' ) ')$  returns TRUE for any operator
  - $\text{Prcd}(' ( ', ' ) ')$  returns FALSE

- The motivation behind the conversion algorithm is the desire to output the operators in the order in which they are to be executed.
- If an incoming operator is of greater precedence than the one on top of the stack, this new operator is pushed on to the stack.
- If on the other hand , the precedence of the new operator is less than the one on the top of the stack, the operator on the top of the stack should be executed first.
- Therefore the top of the stack is popped out and added to the postfix and the incoming symbol is compared with the new top and so on.
- Parenthesis in the input string override the order of operations
- When a left parenthesis is scanned, it is pushed on to the stack
- When its associated right parenthesis is found, all the operators between the two parenthesis are placed on the output string. Because they are to be executed before any operators appearing after the parenthesis

## Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for  $A + B * C$

symb	postfix string	opstk	Remarks
A	A	Empty	symb is operand , add 'A'on to the postfix string
+	A	+	symb is operator, and stack empty, push + on to the stack
B	AB	+	symb is operand , Add 'B' to the postfix string
*	AB	+*	symb is operator, precedence of + (stack top) is < than precedence of *, therefore push * on to the stack

## Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for  $A + B * C$

symb	postfix string	opstk	Remarks
C	ABC	+ *	symb is operand , add C on to the postfix string
End of input	ABC*	+	Pop * from the stack and add to the postfix string
-	ABC*+	Empty	Pop + from the stack and add to the postfix string

## Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for  $(A + B) * C$

symb	postfix string	opstk	Remarks
(	-	Empty	Stack empty, push '(' on to the stack
A	A	(	Symb is an operand, add 'A' to the postfix string
+	A	(+	Precedence of top of the stack '(' is less than '+', push '+' on to the stack
B	AB	(+	Symb is an operand, add B to the postfix string
)	AB+	(	Precedence of stack top '+' is > than ')' Pop '+' from the stack and add to the postfix string
	AB+	empty	Precedence of stack top stack top '(' is not greater than ') and symb is ')', therefore pop '(' from the stack

## Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for  $(A + B) * C$

symb	postfix string	opstk	Remarks
*	AB+	*	Stack empty, push '*' on to the stack
C	AB+C	*	Symb is an operand, add 'C' to the postfix string
End of string	AB+C*	Empty	End of input, pop * from the stack and add to the postfix.

## Infix to postfix conversion – another algorithm

---

```
void convert_postfix(char *infix,char*postfix)
{
    i=0;
    char ch;
    j=0;
    push(s,&top,'#');
    while(infix[i]!='\0')
    {
        ch=infix[i];
        //while the precedence of top of stack is greater than the
        //precedence of the input symbol, pop and add to the postfix
        while(stack_prec(peep(s,top))>input_prec(ch))
            postfix[j++]=pop(s,&top);
```

```
if(input_prec(ch)!=stack_prec(peep(s,top)))  
    push(s,&top,ch);  
else  
    pop(s,&top);  
i++;  
}  
while(peep(s,top)!='#')  
    postfix[j++]=pop(s,&top);  
postfix[j]='\0';  
}
```



## Infix to postfix conversion – another algorithm

### Precedence table

Operator	Input Precedence	Stack Precedence
<b>+, -</b>	<b>1</b>	<b>2</b>
<b>*, /</b>	<b>3</b>	<b>4</b>
<b>\$</b>	<b>6</b>	<b>5</b>
<b>Operands</b>	<b>7</b>	<b>8</b>
<b>)</b>	<b>0</b>	<b>- : Never pushed on to stack</b>
<b>(</b>	<b>9</b>	<b>0</b>
<b>#</b>	<b>-</b>	<b>-1</b>

Example 1:

Input :  $A * B + C / D$

Output :  $+ * A B / C D$

Example 2 :

Input :  $(A - B / C) * (D / K - L)$

Output :  $* - A / B C - / D K L$

1: Reverse the infix expression i.e  $A+B*C$  will become  $C*B+A$ .

Note while reversing each '(' will become ')' and each ')' becomes '('.

2: Obtain the postfix expression of the modified expression i.e  $CB*A+$ .

3: Reverse the postfix expression. Hence in our example prefix is  $+A*BC$ .

1. **When converting an infix expression to a postfix expression using a stack, which of the following operations is performed when a right parenthesis ) is encountered?**
  - a) Push ) onto the stack.
  - b) Pop operators from the stack until a ( is encountered.
  - c) Discard all operators and continue scanning.
  - d) Push all remaining operators into output.

1. **When converting an infix expression to a postfix expression using a stack, which of the following operations is performed when a right parenthesis ) is encountered?**
  - a) Push ) onto the stack.
  - b) Pop operators from the stack until a ( is encountered.
  - c) Discard all operators and continue scanning.
  - d) Push all remaining operators into output.

**2. What is the postfix form of the expression  $(A + B) * (C - D)$ ?**

- a)  $ABCD+-*$
- b)  $AB+CD-*$
- c)  $ABCD+*-$
- d)  $AB+C-D^*$

2. What is the postfix form of the expression  $(A + B) * (C - D)$ ?

a) ABCD+-\*

b) AB+CD-\*

c) ABCD+\*-

d) AB+C-D\*

**3. Which of the following statements about operator precedence during conversion is true?**

- a) Operators with higher precedence are pushed to the output queue before the stack.
- b) When a new operator is read, operators of higher or equal precedence are popped from the stack to the output.
- c) Precedence of operators is ignored when parentheses are used.
- d) None of the above.

**3. Which of the following statements about operator precedence during conversion is true?**

- a) Operators with higher precedence are pushed to the output queue before the stack.
- b) When a new operator is read, operators of higher or equal precedence are popped from the stack to the output.
- c) Precedence of operators is ignored when parentheses are used.
- d) None of the above.



**4. While converting infix to prefix expression using a stack, what is the most efficient approach?**

- a) Convert infix to postfix, then reverse it.
- b) Reverse the infix, swap parentheses, convert to postfix, and then reverse the output.
- c) Use two stacks for operators and operands.
- d) Directly scan the infix expression from left to right.

**4. While converting infix to prefix expression using a stack, what is the most efficient approach?**

- a) Convert infix to postfix, then reverse it.
- b) Reverse the infix, swap parentheses, convert to postfix, and then reverse the output.
- c) Use two stacks for operators and operands.
- d) Directly scan the infix expression from left to right.

**5. Consider the infix expression  $A + B * C - D ^ E ^ F$ . What will be the correct postfix form (considering standard precedence and associativity)?**

a)  $ABC^*+DEF^{^^}-$

b)  $AB+C*DEF^{^^}-$

c)  $AB+C*D^E^F-$

d)  $ABC*DE^F^{^^}+$

5. Consider the infix expression  $A + B * C - D ^ E ^ F$ . What will be the correct postfix form (considering standard precedence and associativity)?

a)  $ABC^*+DEF^^-$

b)  $AB+C*DEF^^-$

c)  $AB+C*D^E^F-$

d)  $ABC*DE^F^+_-$



**THANK YOU**

---

**Dinesh Singh**

Department of Computer Science & Engineering