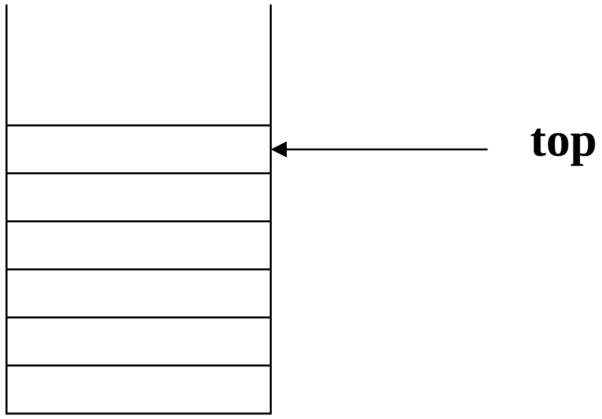
STACKS

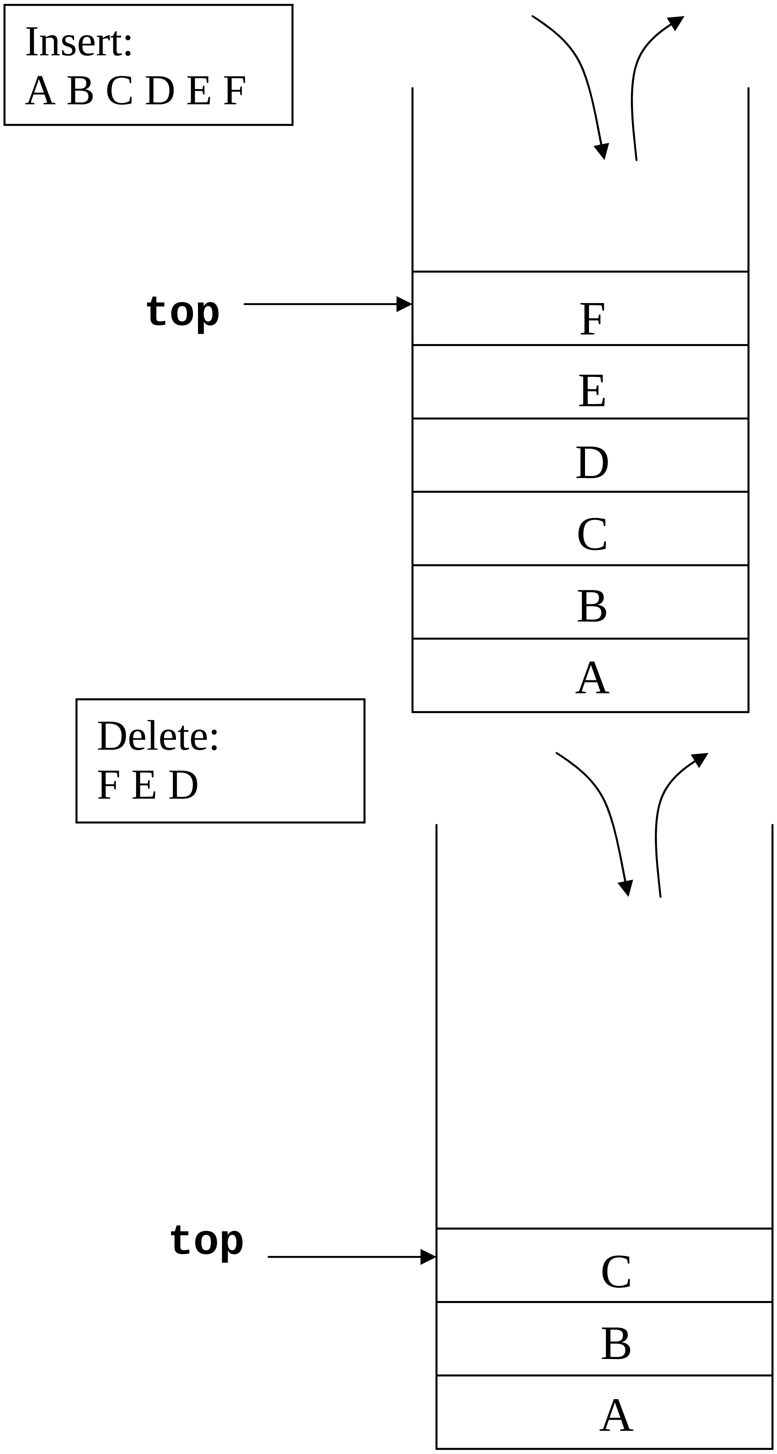
A *stack* is a linear data structure for collection of items , with the restriction that items can be added one at a time and can only be removed in the reverse order in which they were added. The last item represents the *top* of the stack. Such a stack resembles a stack of trays in a cafeteria, or stack of boxes. Only the top tray can be removed from the stack and it is the last one that was added to the stack. A tray can be removed only if there are some trays on the stack, and a tray can be added only if there is enough room to hold more trays.



A stack is defined in terms of its behavior. The common operations associated with a stack are as follows:

1. **push**: adds a new item on top of a stack.
2. **pop**: removes the item on the top of a stack
3. **isEmpty**: Check to see if the stack is empty
4. **isFull**: Check to see if stack is already full
5. **returnTop**: Indicate which item is at the top

A stack is a dynamic structure. It changes as elements are added to and removed from it. It is also known as a LIFO (Last In First Out) structure.



Insert:
A B C D E F

Delete:
F E D

A stack is very useful in situations when data have to be stored and then retrieved in the reverse order. Some applications of stack are listed below:

Function Calls:

We have already seen the role stacks plays in nested function calls. When the main program calls a function named F, a stack frame for F gets pushed on top of the stack frame for main. If F calls another function G, a new stack frame for G is pushed on top of the frame for F. When G finishes its processing and returns, its frame gets popped off the stack, restoring F to the top of the stack.

Large number Arithmetic:

As another example, consider adding very large numbers. Suppose we wanted to add 353,120,457,764,910,452,008,700 and 234,765,000,129,654,080,277. First of all note that it would be difficult to represent the numbers as integer variables, as they cannot hold such large values. The problem can be solved by treating the numbers as strings of numerals, store them on two stacks, and then perform addition by popping numbers from the stacks.

Evaluation of arithmetic expressions:

Stacks are useful in evaluation of arithmetic expressions. Consider the expression

5 * 3 +2 + 6 * 4

The expression can be evaluated by first multiplying 5 and 3, storing the result in A, adding 2 and A, saving the result in A. We then multiply 6 and 4 and save the answer in B. We finish off by adding A and B and leaving the final answer in A.

A = 15 2 +
= 17
B = 6 4 *
= 24
A = 17 24 +
= 41

We can write this sequence of operations as follows:

5 3 * 2 + 6 4 * +

This notation is known as **postfix** notation and is evaluated as described above. We shall shortly show how this form can be generated using a stack.

Basically there are 3 types of notations for expressions. The standard form is known as the **infix** form. The other two are **postfix** and **prefix** forms.

Infix: operator is between operands A + B

Postfix : operator follows operands A B +

Prefix: operator precedes operands   + A B

Note that all infix expressions can not be evaluated by using the left to right order of the operators inside the expression. However, the operators in a postfix expression are ALWAYS in the correct evaluation order. Thus evaluation of an infix expression is done in two steps. The first step is to convert it into its equivalent postfix expression. The second step involves evaluation of the postfix expression. We shall see in this section, how stacks are useful in carrying out both the steps. Let us first examine the basic process of infix to postfix conversion.

Infix to postfix conversion:

a + b * c   **Infix form**

(precedence of * is higher than of +)

a + (b * c) convert the multiplication

a + (b c *) convert the addition

a (b c *) + Remove parentheses

a b c * + **Postfix form**

Note that there is no need of parentheses in postfix forms.

Example 2:

(A + B) * C   **Infix form**

(A B +) * C   Convert the addition

(A B +) C *   Convert multiplication

A B + C *   **Postfix form**

No need of parenthesis anywhere

Example 3:

a + ((b * c) / d)

a + ((b c *) /d)

(precedence of * and / are same and they are left associative)

a + (b c * d /)

a b c * d / +

- More examples

Infix	**Postfix**
(a + b) * (c – d)	a b + c d - *
a – b / (c + d * e)	a b c d e * + / -
((a + b) * c – (d – e))/(f + g)	a b + c * d e - - f g + /

Order of precedence for operators:

multiplication (*) and division (/)

addition (+) and subtraction (-)

The association is assumed to be left to right.

i.e. a + b + c = (a+b)+c = ab+c+

Evaluating a Postfix Expression

We can evaluate a postfix expression using a stack. Each operator in a postfix string corresponds to the previous two operands . Each time we read an ***operand*** we push it onto a stack.

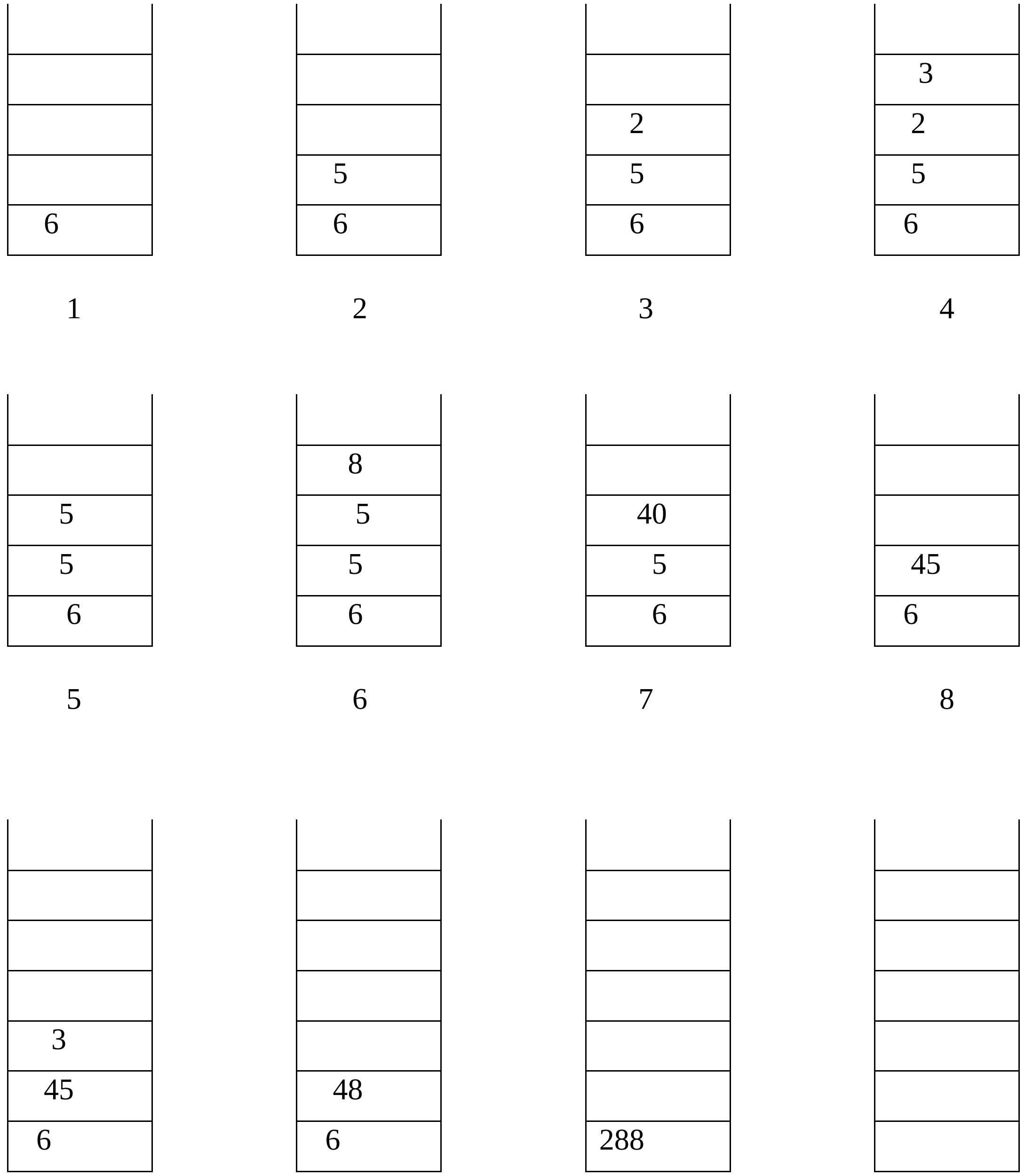
When we reach an ***operator*** its associated operands (the top two elements on the stack) are popped out from the stack.

We then perform the indicated operation on them and push the result on top of the stack so that it will be available for use as one of the ***operands*** for the next ***operator*** .

The following example shows how a postfix expression can be evaluated using a stack.

Example

6 5 2 3 + 8 * + 3 + *



The process stops when there are no more operator left in the string. The result of evaluating the expression is obtained just by popping off the single element. More examples will be done in the lecture and recitation labs.

Converting an Infix Expression to Postfix

A stack can also be used to convert an infix expression in standard form into postfix form. We shall assume that the expression is a legal one (*i.e.* it is possible to evaluate it). When an operand is read, it will be placed on output list (printed out straight away). The operators are pushed on a stack. However, if the priority of the top operator in the stack is higher than the operator being read, then it will be put on output list, and the new operator pushed on to the stack. The priority is assigned as follows.

Priority

1. (Left parenthesis in the expression
2. * /
3. + –
4. (Left parenthesis inside the stack

The left parenthesis has the highest priority when it is read from the expression, but once it is on the stack, it assumes the lowest priority.

To start with , the stack is empty. The infix expression is read from left to right. If the character is an OPERAND, it is not put on the stack. It is simply printed out as part of the post fix expression.
The stack stores only the OPERATORS. The first operator is pushed on the stack. For all subsequent operators, priority of the incoming operator will be compared with the priority of the operator at the top of the stack.

If the priority of the incoming-operator is higher than the priority of topmost operator-on-the-stack , it will be pushed on the stack.
If the priority of the incoming-operator is same or lower than the priority of the operator at the top of the stack, then the operator at top of the stack will be popped and printed on the output expression.

The process is repeated if the priority of the incoming-operator is still same or lower than the next operator-in-the stack.
When a left parenthesis is encountered in the expression it is immediately pushed on the stack, as it has the highest priority. However, once it is inside the stack, all other operators are pushed on top of it, as its inside-stack priority is lowest.

When a right parenthesis is encountered, all operators up to the left parenthesis are popped from the stack and printed out. The left and right parentheses will be discarded. When all characters from the input infix expression have been read, the operators remaining inside the stack, are printed out in the order in which they are popped.

Here is an algorithm for converting an infix expression into its postfix form.

Algorithm:

```
 while there are more characters in the input
 {
     Read next symbol ch in the given infix expression.
     If ch is an operand put it on the output.
     If ch is an operator i.e.* , /, +, -,  or  (
      {
          If stack is empty push ch onto stack;
          Else check the item op at the top of the stack;
          while (more items in the stack   &&
                       priority(ch) <= priority (op) )
          {
               pop op and append it to the output, provided it
            is not an open parenthesis
                     op = top element of stack
          }
          push ch onto stack
     }
          If  ch is right parenthesis ')'
               Pop items from stack until left parenthesis
                   reached
               Pop  left parenthesis and discard both left
               and right parenthesis
}/* now no more characters in the infix expression*/

Pop remaining items in the stack to the output.
```

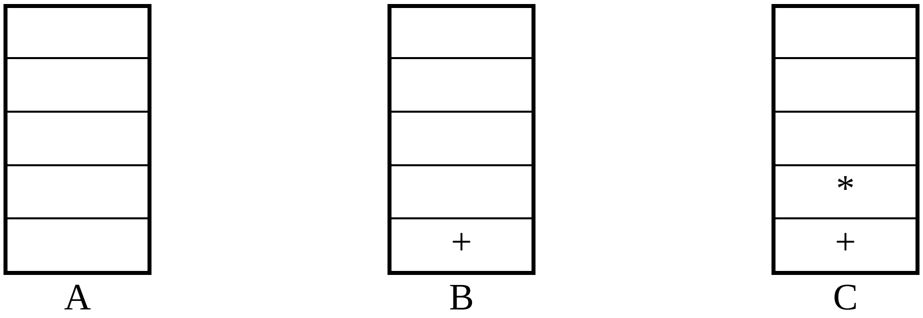
Using a stack to convert an Infix expression into its corresponding postfix form (some examples)

Consider the following expressions with the infix notation. We want to transform these into postfix expressions using a stack. We also trace the state of the operator stack as each character of the infix expression is processed.

The contents of the operator stack at the indicated points in the infix expressions (points A, B and C) are shown below for each case

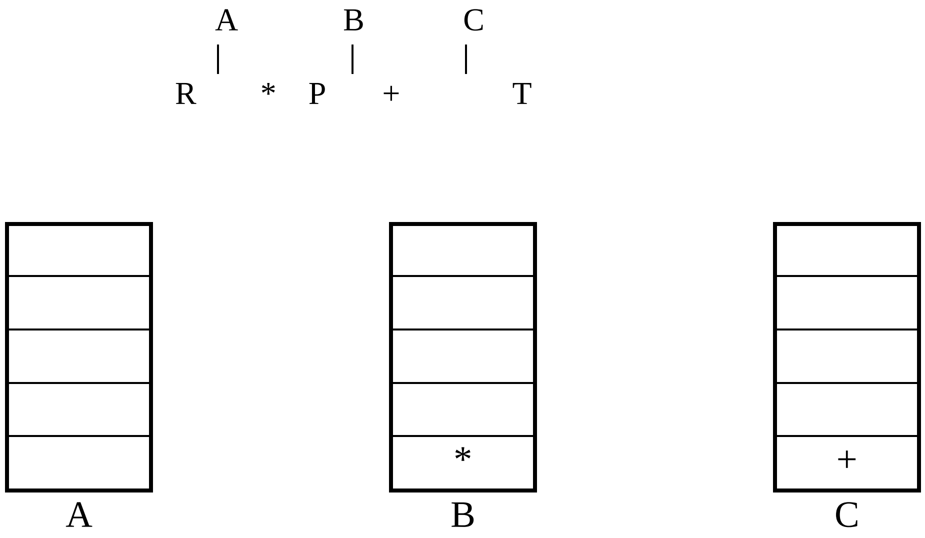
1.

```
   A      B      C
   |      |      |
 M    +  P   *       Q
```



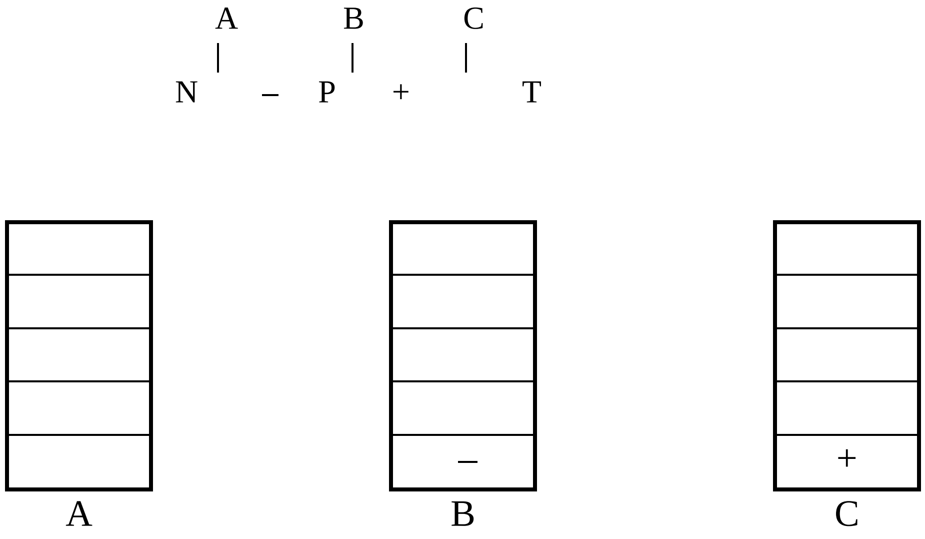
Resulting Postfix Expression : **M P Q * +**

2.



Resulting Postfix Expression : **R P * T +**

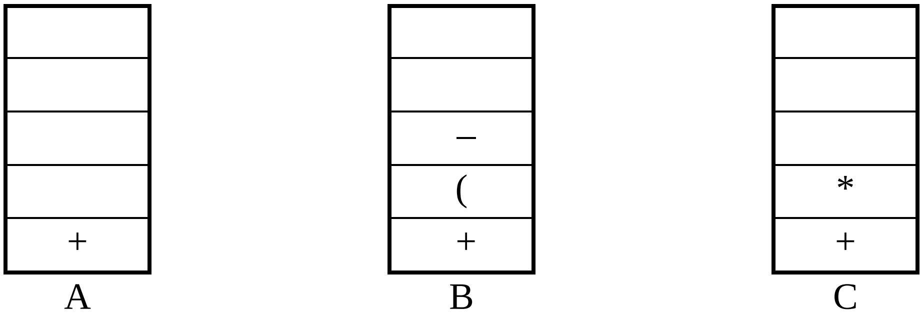
3.



Resulting Postfix Expression : **N P – T +**

4.

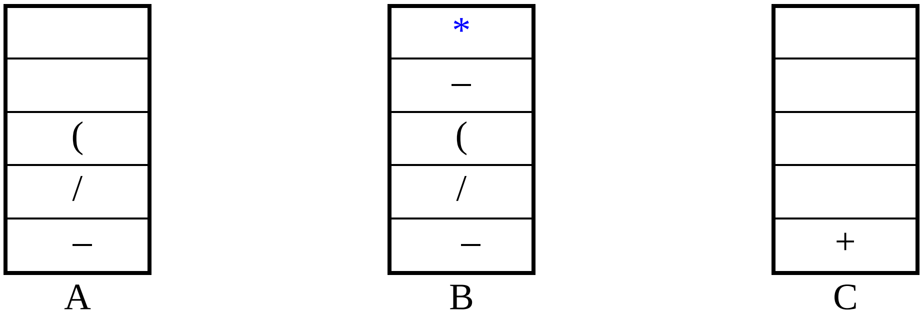
```
          A         B         C
          |         |         |
 M  +     (   P  –   K  )  *      T
```



Resulting Postfix Expression : **M P K – T * +**

5.

```
          A           B        C
          |           |        |
M –   P / (  T – K  * N   ) +      R
```



Resulting Postfix Expression : **M P T K N * – / – R +**