Book

A Simplified Approach to Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala
Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar
Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors
Edition 2014

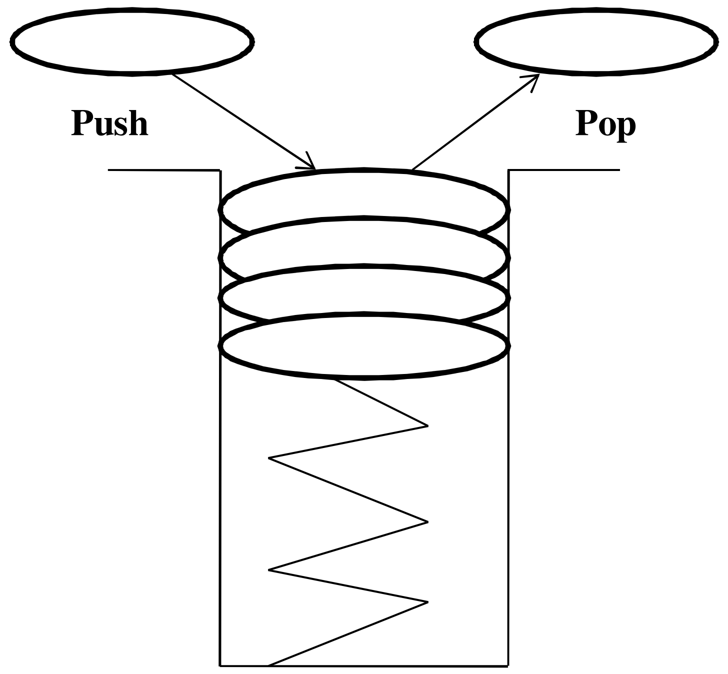
APPLICATIONS OF STACK

Contents

Definition

- Stack is one of the most commonly used linear data structures.
- In stack insertion and deletion of an element can occur at only end known as TOP.
- Insertion operation is known as PUSH and deletion is known as POP.
- Stack is also called LIFO(Last In First Out).
- It means that last item inserted to the stack will be the first item to be removed form the stack.

Operations On Stack



Stack Showing Push And Pop Operations From The Top Position

Applications of Stack

Stack is used in wide variety of applications. These are extensively used in system programming (compilers and operating system) and application programming .

Evaluation of Arithmetic Expression

An important application of stack is the compilation of arithmetic expression in the programming languages. The compiler must be able to translate the **infix notation** to **reverse polish notation**. Compilers accomplish this task of notation conversion with the help of stack.

Infix Notation

- In this notation, operator is placed between **its operands**.
  For example, **m * n**

- While solving the infix notation, the main consideration is the **precedence** of the operators and their **associativity**.

- For example, consider an expression,

  e = q * r + s

  In this expression, **q** and **r** are first multiplied and then **s** is added. That is, * operator has precedence over the **+** operator.

Example : Infix Notation

Consider an expression,

e = 4 – 2 ^ 4 + 8 * 3 + 18 / 3 + 6

2^4

^ having highest precedence over the other operators, will be solved first

e = 4 – 16 + 8 * 3 + 18 / 3 + 6

8*3

Now, * and / operations will be performed from left to right because both are having same level of precedence.

18/3

e = 4 – 16 + 24 + 18 / 3 + 6

4-16

e = 4 – 16 + 24 + 6 + 6

Now, + and – operations will be performed from left to right because both are having same level of precedence.

-12+24

e = - 12 + 24 + 6 + 6

12+6

e = 12 + 6 + 6

18+6

e = 18 + 6

e=24

Infix Notation(Contd.)

Precedence and associativity of the operators:

Priority	**Operator**	**Associativity**
1st	Brackets	Inner to Out and Left to Right
2nd	Exponent ^	Left to Right
3rd	* /	Left to Right
4th	+ -	Left to Right
5th	=	Right to Left

- The main problem with this notation is that the order of operator and operands in the expression does not uniquely decide the order in which operations are to be carried out.

Prefix Notation

- This notation is also popular with the name **polish notation.**

- In prefix notation, operator is placed before **its operands**. For example, ***mn**.

- The main characteristics of this notation is that the order in which the operations are to be carried out is completely determined by the position of operators and operands in the expression.

- While solving the arithmetic expression which is written in prefix/polish notation, there is no need to take care of any precedence rule.

Prefix Notation (contd.)

In order to translate an arithmetic expression from infix to polish notation, we will do it step by step by using [ ] (**square brackets**) to indicate the **partial conversion.**

Example 1:

Iin = (a – b)/ c

= **[- ab] / c**

Ipre = / - abc

Example 2:

Iin = (**x – y**)*((**z + v**) / f)

= [- xy]*(**[+ zv] / f**)

= **[- xy]*[/ + zvf]**

Ipre = * - xy / + zvf

Postfix Notation

- The postfix notation is also known as **reverse polish notation**.
- In this notation, operator is placed **after its operands**.
- For example, **mn +, mn -, mn *, mn / and mn ^.**
- The fundamental characteristics of this notation is that there is no need of parenthesis to designate the hierarchy of operators.
- In this notation, order of operations is completely determined by the order of operands and its operators.

CONVERSION FROM INFIX TO POSTFIX

Example 1: Iin = (a – b) / c

= [a b -] / c

Ipost = a b – c /

Example 2: Iin = (x – y) * ((z + v)/ f)

= [x y -] * ([z v +]/ f)

= [x y -] * [z v + f /]

Ipost = x y – z v + f / *

Algorithm: Convert an arithmetic expression ‘I’ written in infix notation into its equivalent postfix expression ‘P’.

STEP 1: Push a left parenthesis (onto the stack.

STEP 2: Append a right parenthesis) at the end of Given expression **I.**

STEP 3: Repeat steps from 4 to 8 by scanning **I** character by character from left to right until the stack is empty.

STEP 4: If the current character in **I** is a white space, simply ignore it.

STEP 5: If the current character in **I** is an operand, write it as the next element of the postfix expression **P.**

STEP 6: If the current character in **I** is a left parenthesis (, push it onto the stack.

Algorithm: (Continued)

STEP 7: If the current character in **I** is an operator , Then

- Pop operators (if there is any) at the top of stack while they have **equal or higher precedence** than the current operator and put the popped operators in the postfix expression **P.**
- Push the currently scanned operator on the stack.

STEP 8: If the current character in **I** is a right parenthesis, Then

- Pop operators from the top of the stack and insert them in the postfix expression **P** until a left parenthesis is encountered at the top of the stack.
- Pop and discard left parenthesis (from the stack.

STEP 9: Exit

Conversion of infix expression I into its postfix expression P

Character Scanned	Status Of Stack	Postfix Expression P
	(	
(	(	
6	((	
+	((	6
2	((+	6
)	(	6 2 +
*	(	6 2 +
5	(*	6 2 +
-	(*	6 2 + 5
8	(-	6 2 + 5 *
/	(-	6 2 + 5 * 8
4	(- /	6 2 + 5 * 8
)	Null	6 2 +5 *8 4 /-

Evaluation Of Postfix Notation

ALGORITHM: Evaluate an arithmetic expression 'P' written in postfix notation and calculates the result of the expression in variable 'Value'.

```
STEP 1: Scan P from left to right and Repeat steps 2 and 3 for
        each scanned character until end of the expression.
STEP 2: If scanned character is an operand, push it onto the
        stack.
STEP 3: If the scanned character is an operator Then
        Pop the two top elements a and b from the stack where a
        is the top element and b is the next to top element.
        Apply the operator on the operands b and a and push the
        result onto the stack.
        [End loop]
STEP 4: Set Value = Stack [Top]
STEP 5: Print: "The value of the expression is ": Value
STEP 6: Exit
```

Evaluation Of Postfix Notation

Character Scanned	Stack
6	
2	6
+	8
5	8
*	40
8	40
4	40 8
/	40 2
-	38

Matching parenthesis

- A stack can be used for syntax verification of the arithmetic expression i.e. matching parenthesis.
- To accomplish this task, the expression is scanned from left to right character by character.
- Whenever a left parenthesis is encountered, we push it onto the stack.
- When we encounter a right parenthesis], or), or}, the status of the stack is checked.
- If the stack is empty then we have a right parenthesis in the expression that does not have the corresponding left parenthesis in the expression showing the mistake in the expression.

Matching Parenthesis

- If the stack is not empty, we will pop the topmost element from the stack and compare it with the scanned right parenthesis. If both the parenthesis are not of the same type then it shows a mistake in the expression. But, if both the parenthesis are of the same type then the same procedure is repeated until the whole expression is scanned and stack is empty.

Algorithm:

Syntax verification by scanning an arithmetic expression ‘I’

```
STEP 1: Scan the expression I from left to right and Repeat
        steps 2 to 4 for each scanned character until the end
        of the expression is reached.
STEP 2: If the scanned character is left parenthesis then
        push it onto the stack.
STEP 3: If the scanned character is an operator or operand
        then ignore it.
STEP 4: If the scanned character is a right parenthesis Then
a)   If Stack[Top] = Null Then
     Print “There is no left parenthesis corresponding to right
     parenthesis”.
     Exit
[End If]
```

Algorithm(Contd.)

```
b. Pop the top element from the stack and compare it with
   currently scanned right parenthesis.
c. If both are **not corresponding** Then
      Print "The braces are not in proper order".
      Exit
      [End If]
[End Loop]
STEP 5 :If Stack [Top] != Null Then
            Print: "There is no right parenthesis corresponding to the
            left parenthesis".
            Exit
[End If]
STEP 6: Exit.
```

Example

Let us check the order of braces in an arithmetic expression **I** using above algorithm.

I = [(5 + 6) * 7 – {7 / 4} + (3 * 2) – 8]

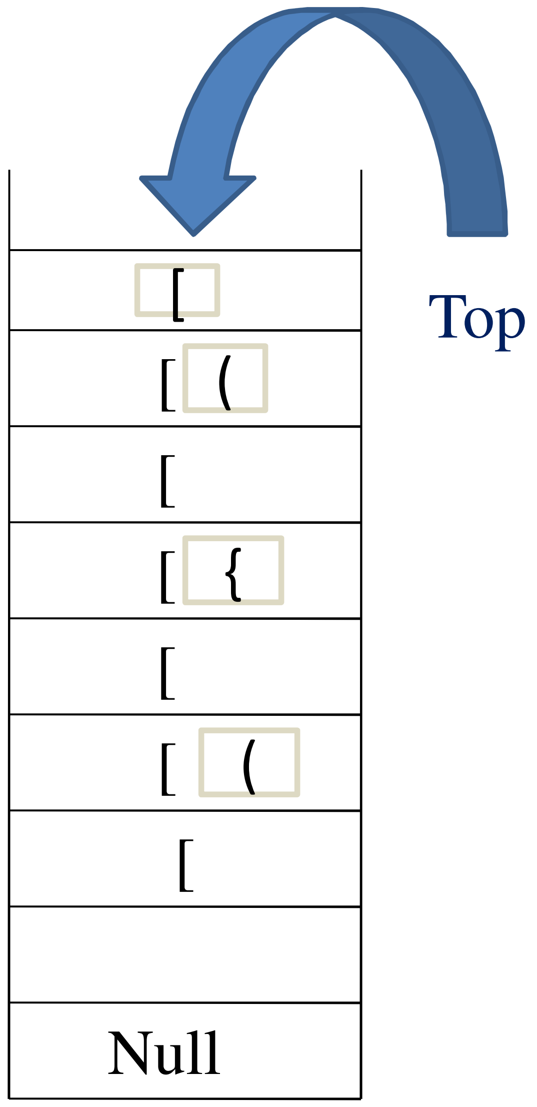
Character Scanned	Status of Stack
[	[
(	[(
)	[
{	[{
}	[
(	[(
)	[
]	Null

Example

I=[(5+6)*7-{7/4}+(3*2)-8]

[
(
)
{
}
(
)
]

Character Scanned



Stack

Quick Sort

- Quick sort is an important application of stack. Sorting means, the arrangement of the elements of the list in some logical order.
- Quick sort algorithm which is an important application of stack uses **divide and conquer** policy for sorting the list of elements.
- In divide and conquer policy, the problem to be solved is divided into sub- problems repeatedly until we reach the smallest size sub-problems whose solution is easy to find.
- Then solution of these small sub-problems is combined to obtain the solution of the whole problem.

Quick Sort(Contd.)

Consider an unsorted list of 10 elements:

5 8 2 11 1 33 4 3 100 7

←

Here, while scanning the list from **right to left** the first element 5 will be interchange with the element smaller than 5.

5 8 2 11 1 33 4 3 100 7

→

Now, starting from first position i.e. from element **3**, scan the list from **left to right** by comparing each 26element with **5**.This time meeting an element larger than 5, we will stop scanning the list and interchange it with 5.

3 8 2 11 1 33 4 5 100 7

←

Quick Sort(Contd.)

Now, starting from element **8**, scan the list from **right to left** by comparing each element with **5** and stop scanning the list on meeting an element smaller than **5**, and interchange the currently scanned element with **5**.

```
3     5    2     11     1     33     4    8     100   7
     →
```

Now, starting from element **4**, scan the list from left to right by comparing each element with **5**, meeting an element greater than **5** and interchange the element with **5**.

```
3     4     2     11   1     33     5    8    100   7
                                   ←
```

Similarly, starting from element **11**, scan the element from **right to left** by comparing each element with **5** and interchange it with a smaller element.

Quick Sort(Contd.)

$$\begin{array}{cccccccccc} 3 & 4 & 2 & 5 & 1 & 33 & 11 & 8 & 100 & 7 \\ & & & \rightarrow & & & & & & \end{array}$$

Now, starting from element **1**, scan the list from **right to left.** Here, this time there is no element which is larger than **5**. It means the element **5** is at the correct position in the list and all the elements smaller than **5** are on the left side and elements larger than **5** are on the right side of the element **5**.

$$\underbrace{3 \quad 4 \quad 2 \quad 1}_{\textbf{Left Sublist}} \qquad \mathbf{5} \qquad \underbrace{33 \quad 11 \quad 8 \quad 100 \quad 7}_{\textbf{Right Sublist}}$$

Now, the task of sorting is reduced to sorting the two sublists.

Quick Sort (Contd.)

Sorting Left Sublist:

In this pass the fist element of the list will occupy its correct position in the list.

3 4 2 1

In 2nd pass of sorting sublist scan the list from left to right and meeting an element larger than element 3 interchange this element with 3.

1 4 2 3

Now scan the list starting from right to left by comparing each element with 3 and an element meeting an element smaller than 3 interchange it with 3.

1 3 2 4

Quick Sort (Contd.)

Sorting Right Sublist :

In first pass of sorting right sublist first element of the sublist i.e. 33 will occupy its correct position in the list .

33 11 8 100 7

In second pass scan the list left to right and an meeting an element larger than 33 stop scannig and interchange with 33.

7 11 8 100 33

Now scan the list from right to left and finding an element smaller than 33 interchange it with 33.

7 11 8 33 100

Quick Sort(Contd.)

- Sorting Right Sublist:

  7 11 8   33   100

  Left Sublist   **Right Sublist**

  Sorting Left Sublist:

  7 11 8

  In first pass first element 7 occupy its correct position.

  7 11 8

  The sorted sublist is:

  7 8 11

Quick Sort (Contd.)

Sorted Array is:

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
1	2	3	4	5	7	8	11	33	100

In this array all elements are sorted(ascending order)

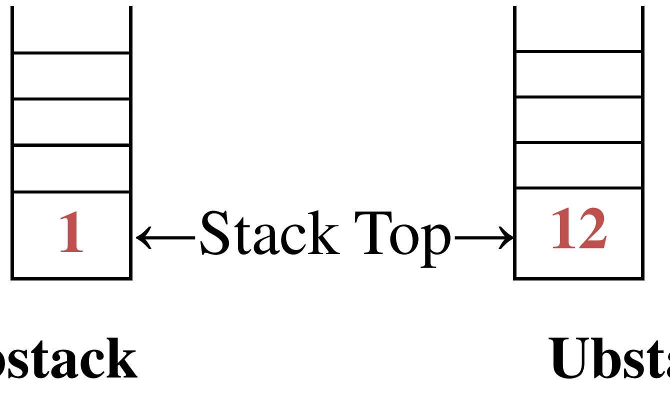
Quick Sort(Contd.)

Consider a linear list **'L'** having **12** numbers :

29	**35**	**42**	**17**	**39**	**12**	**25**	**54**	**10**	**72**	**19**	**85**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

An Unsorted Array 'L' of 12 Elements

Initially, the lower bound and upper bound of given linear list will be pushed onto the two different stacks named **Lbstack** and **Ubstack** respectively as shown below:



Stacks containing the lower bound and Upper Bound of the list

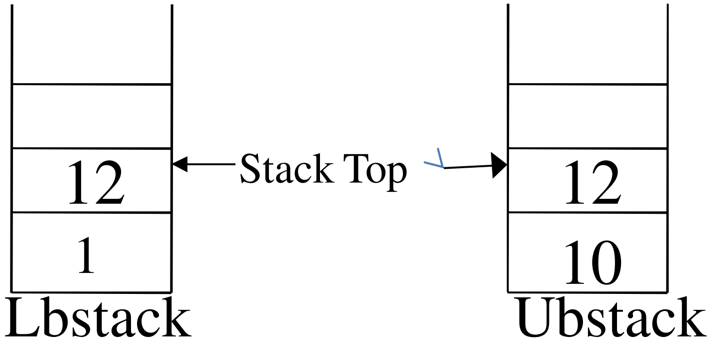
Quick Sort(Contd.)

Now, the first reduction step will be performed on the list after popping the indices of the list from both the stacks making them empty.

29 35 42 17 39 12 25 54 10 72 19 **85**

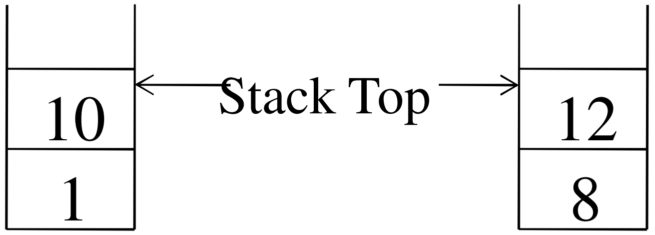
←

29 35 42 17 39 12 25 54 10 72 19 85



19 35 42 17 39 12 25 54 10 72 29 85

19 29 42 17 39 12 25 54 10 72 35 85

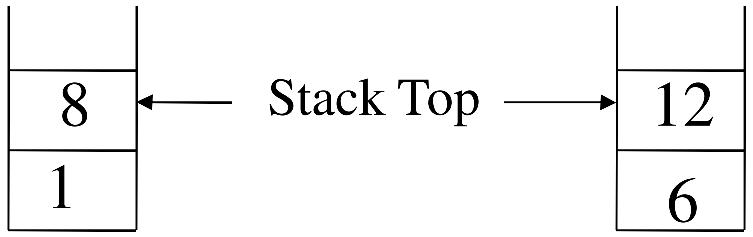


Quick sort(contd.)

19 10 42 17 39 12 25 54 29 72 35 85

19 10 29 17 39 12 25 54 42 72 35 85

- ***Now the status of stack is :***



19 10 25 17 39 12 29 54 42 72 35 85

19 10 25 17 29 12 39 54 42 72 35 85

→

-

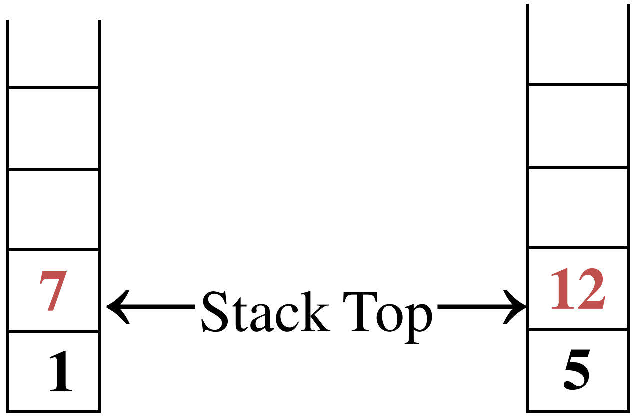
Quick Sort(Contd.)

19	**10**	**25**	**17**	**12**	**<u>29</u>**	**39**	**54**	**42**	**72**	**35**	**85**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**1O**	**11**	**12**

Left Sublist **Right Sublist**

Two Sublists Created After The First Reduction Step

After the completion of first step, the first element has occupied the correct position in the list and two sublists have been created. The lower and upper indices of newly created sublists will be pushed into the stacks **LbStack** and **UbStack** as shown in figure.



Quick Sort(Contd.)

Now, the same procedure will be applied on the sublist whose lower and upper indices will be popped from top of the stacks **LbStack** and **UbStack** respectively.

Here, the reduction step will be applied on the right sublist first whose lower and upper indices are 7 and 12 respectively.

After the completion of the reduction step, the element at index number 7 i.e. 39 will occupy the correct position in the sublist and the sublist will be divided into two new sublists whose indices will be pushed onto the stacks.

This procedure will be repeated until the whole list is stored.

Algorithm:

Sort an array 'L' with 'n' elements Quick sort (L, n)

```
STEP 1: Set stacktop = ∅
STEP 2: If n > 1 Then
            Set stacktop = 1
            Set LbStack [stacktop] =1          ∅
            Set UbStack [stacktop] = n
            [End If]
STEP 3: Repeat Steps 4 to 7 while stacktop =
STEP 4: Set Begin = LbStack [stacktop]
           Set End = UbStack [stacktop]
           Set stacktop = stacktop – 1
STEP 5: Loc = Splitpass (L, Begin, End)
```

Algorithm(Contd.)

```
STEP 6: If Begin < Loc - 1 Then
              Set stacktop = stacktop +1
              Set Lbstack [stacktop] = Begin
              Set UbStack [stacktop] = Loc -1
        [End If]
STEP 7: If End > Loc + 1 Then
              Set stacktop = stacktop + 1
              Set LbStack [stacktop] = Loc + 1
              Set UbStack [stacktop]   =   End
        [End If]
        [End Loop]
STEP 8: Exit
```

ALRORITHM

SplitPass (L, Begin, End)

```
STEP 1: Set Left = Begin, Right = End, Loc = Begin And Flag =False
STEP 2: Repeat steps 3 to 6 While Flag = False
STEP 3: Repeat While L [Loc] < = L [Right] And Loc !=Right
                Set Right= Right – 1
                [End Loop]
STEP 4: If Loc = Right Then
                Set Flag = True
            Else If L[Loc] > L[Right] Then
                  Interchange L[Loc] and L[Right]
                  Set Loc =  Right
                 [End If]
```

Algorithm(Contd.)

```
STEP 5: Repeat While L[Loc] > = L[Left] AND  Loc !=Left
                Set Left =Left + 1
              [End Loop]
STEP 6: If Loc = Left Then
                Set Flag= True
            Else If L[Loc] < L[Left] Then
                Interchange L[Loc] and L[Left]
                Set Loc = Left
             [End If]
             [End Loop]
STEP 7: Return Loc
```

Complexity Analysis of Quick Sort Algorithm

- Complexity of a sorting algorithm is represented by function f(n) i.e. number of comparisons required to sort the list of elements.
- While analyzing quick sort algorithm, the worst case occurs when after each reduction step, the list is partitioned into two sublists with one of them being empty. In this situation, the 1st element will be compared with n - 1 elements to remain at its original position.
- During the second reduction step, 2nd element will be compared with n- 2 elements to remain at its original position and so on. So, in the worst case, the complexity will be:

$$\mathbf{f(n) = (n\text{-}1) + (n\text{-}2) + (n\text{-}3) + \ldots\ldots. + 3 + 2 + 1 = 0(n^2)}$$

Complexity Analysis of Quick Sort Algorithm

- The average case occurs, when after each reduction step of the algorithm, it produces two sublists of approximately same size. To calculate the average case complexity, two assumptions are to be made.
  - The size of the list **n** should be the power of 2 i.e. $\mathbf{n = 2^m}$, for some positive integer value of **m.**
  - After each reduction step sublists formed are approximately of equal size.
  - This procedure will continue until there is **n** sub lists of size 1 each.

So, total number of comparisons **f(n)** will be:

$$\mathbf{f(n) = O(n \log_2 n)}$$

Recursion

- Stack is also used to implement recursive procedure in programming languages like PASCAL, ALGOL, C, C++.
- Recursion is very important and powerful tool for developing algorithms for various problems.
- Recursion is the ability of a procedure either to call itself or calling to some other procedure may result in call to the original procedure.
- Two very important conditions/ requirements that must be satisfied by any procedure to be defined recursively are:
  - There must be a decision criterion that stops the further call to the procedure called **base criteria**.
  - Each time a procedure calls itself either directly or indirectly, it must be **nearer to the solution** i.e. nearer to the base criteria.

Example Of Recursion

Factorial Function

It is a recursively defined problem. The factorial of a positive number **n** is the product of positive integers from **1 to n**. The Factorial of a number is represented symbolically by placing a symbol '!' next to it. The factorial of a positive integer **n** will be defined as:

$$n! = 1 \times 2 \times 3 \times 4 \times \dots\dots..\times (n-1) \times n$$

The value of the factorial function for zero is taken as 1. For e.g.

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

Thus, the formal definition of the factorial function can be given as:

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1)! & if\ n \geq 1 \end{cases}$$

Example Of Recursion

ALGORITHM : Calculate the value of n! recursively

Factorial (n)LE

```
    If n = 0 Then
        Set Fact = 1
        Return
    Else
        Set Fact = n * Factorial (n-1)
    Return
   [End If]
```

Example Of Recursion(Contd.)

Fibonacci Series

A Fibonacci series is a sequence of numbers which is usually denoted by $F_{0,} F_1, F_2\ F_3, \ldots\ldots., F_{n.}$ The series is as shown below:
0, 1,1, 2, 3, 5, 8, 13, 21,……

Generally, in a Fibonacci series, each succeeding term is a sum of two preceding terms. The recursive procedure for finding the nth term of the Fibonacci series can be defined as:

$$Fibo(n) = \begin{cases} n & if\ n = 0\ or\ 1 \\ Fibo(n-1) + Fibo(n-2) & if\ n > 1 \end{cases}$$

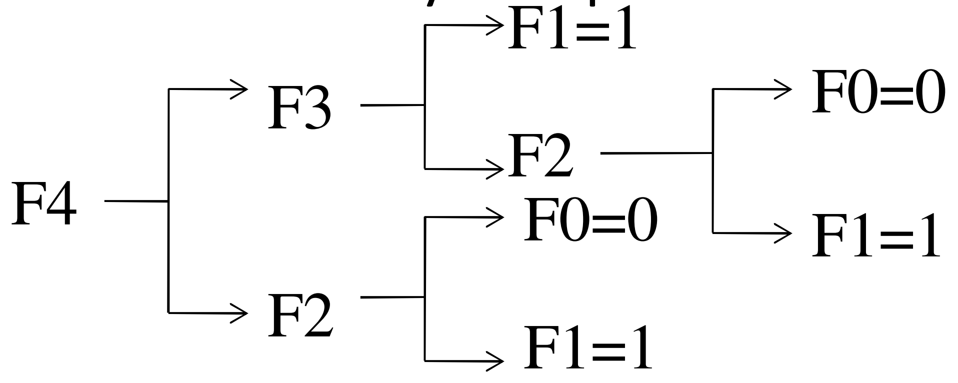
Example Of Recursion(Contd.)

ALGORITHM: Find the nth term of a Fibonacci series recursively.

```
Fibonacci (n)
If n =0 Then
      Set Fibo = 0
Return
Else if n = 1 Then
      Set Fibo = 1
Return
Else
      Set Fibo = Fibonacci (n – 1) + Fibonacci (n -2)
           Return
[End If]
```

Example Of Recursion(Contd.)

- Consider an example of finding the 4th term of the Fibonacci series recursively. The procedure followed for it is shown below:



To find the 4th term recursively ,it is necessary to find the 2nd and 3rd term first.And in turn to find the 3rd term,it is necessary to compute the 2nd and 1st term first.To find the 2nd term of the series,it will be required to find the 1st term and 0th term.By using the values of 0th and 1st term explicitly and backtracking the steps followed,we can find the 4th term of the series.

When To Use Recursion

There are many factors that affect the choice of procedure for solving a given problem:

- Computer Memory Required
- Processing Time Required
- Time Required for developing the Algorithm
- Time Required for Debugging

It is always advisable to consider a tree structure for a given problem. If the tree structure is simple then use of non-recursive procedure is suitable. If the tree structure appears quite bushy with duplication of tasks, then recursive procedure is suitable.

Demerits Of Recursion

1. Many programming languages do not support recursion. Hence recursive mathematical function is implemented using interactive methods.
2. Mathematical functions are implemented using recursion at the cost of execution time and memory space.
3. A recursive procedure can be called from within or outside itself and to show its proper functioning it has to save the return addresses in same order so that, a return to the proper location will result when the return to a calling statement is made.
4. A special care is required to put a stopping condition at which the recursive function will stop.