Recursion & Stack: Conceptual Study for Implementation of Stack in Recursion

Pankaj Kumar Gupta[1], Prof. P. K. Tyagi[2]
[1]*Assistant Professor & Head, BCA Department, Durga Prasad Baljeet Singh (PG) College, Anoopshahr District Bulandshahr*
[2]*Professor & Head, Department of Statistics, Durga Prasad Baljeet Singh (PG) College, Anoopshahr District Bulandshahr*

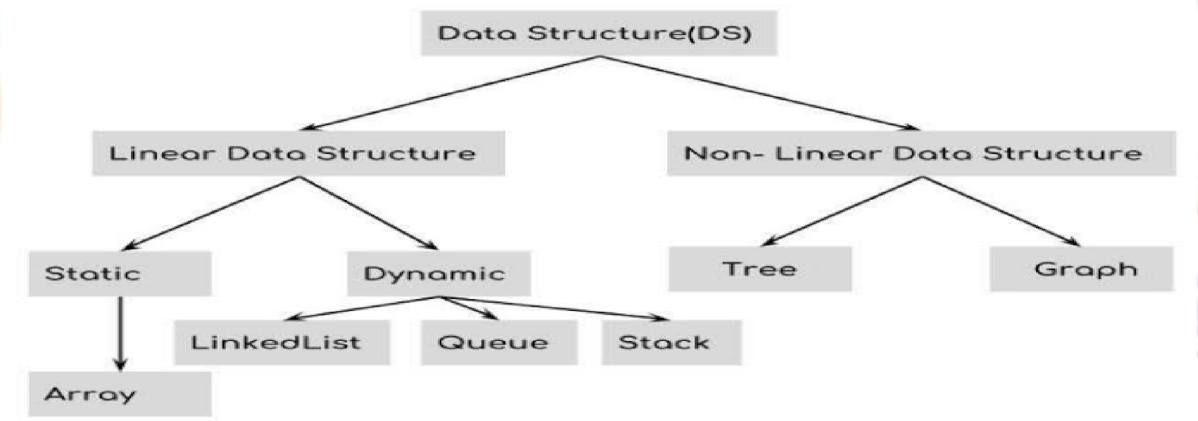
Abstract: **This paper describes the detailed conceptual study of implementation of Stack Data structure in Recursion. In computer science, recursion is a programming technique which uses function or algorithm that invokes itself. Stack is a LIFO data structure in that Item is pushed & popped at same place named Top of Stack. A Recursive Program always uses stack to store the result of calculation made inside recursive procedure.**

Keywords: Recursion, Stack, LIFO.

Data Structure: Computer is an electronic machine which is used for processing and manipulation of Data. Programmer wants to process the data he/she would require main memory to store that data.
In order to make computer work we need to know:
1. How to represent the data in computer?
2. How to access the data?
3. How to solve the problem step by step?

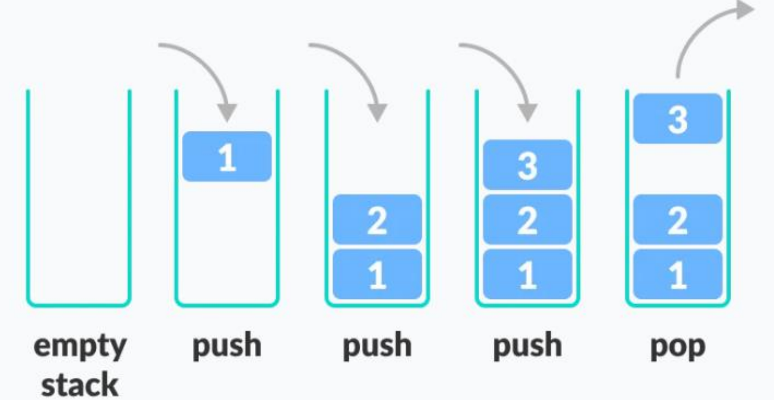
For performing the above task, we use data structures. Now discuss about what is Data Structure? Data structure is a way of representing the logical and mathematical relationship existing among the data.
Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
We can also define data structures as a logical & mathematical model of an organization of data items.
Classification of Data Structure: Data Structures can be classified into two basic categories:
1. Linear data Structures.
2. Non-Linear Data Structures.

1. Linear or Sequential Data Structures: The data structures in which data elements are managed in a particular sequence are called as linear data structures. For example: Array, Stack, Linked List etc.
2. Non-Linear Data Structures: The data structures in which data elements are organized in any arbitrary order or they are not arranged in any particular sequence are called as Non-Linear Data Structures. For example: Tree & Graph.



Out of all Data Structures Stack is described in brief in this paper.

Stack is a **L**ast **I**n **F**irst **O**ut (LIFO) Series in which Items are pushed & popped at same place named Top of the Stack. **Stack** is a Linear List in which the last item added to the list will be the first removing item from the list.



The insertion operation is called as **PUSH** operation and deletion operation as **POP** operation. Since insertion and deletion operations are performed at one end of a stack, the elements can only be removed in the opposite orders from that in which they were added to the stack.

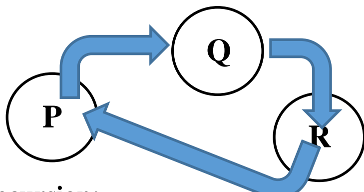
Applications of Stack:

1. Recursion.
2. Tracking the Function Calls.
3. Evaluating the Expressions.
4. Reversing Strings.
5. Hardware Interrupts Servicing.
6. Finding the Combinatorial Problems using Backtracking.

Out of all Applications **Recursion** is described in brief in this paper.

Recursion: A Function that invokes itself is known as recursive function and this process of invoking/calling itself is known as **Recursion**. In other words, the process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

Recursion: Let us consider A function **P** is invoking function **Q** and function **Q** is invoking function **R** & again function **R** is invoking function **P** then this procedure of invoking becomes **Recursion**.



Real world examples of recursion:

1	Cauliflower	
2	Cabbage	
3	Sunflower	

Using a recursive algorithm, We can solve the problems quite easily. Like Towers of Hanoi (TOH), Inorder/ Preorder/ Postorder Tree Traversals, Depth First Search (DFS) technique of Graph, etc. A recursive function solves

a particular problem by calling a copy of itself and solving smaller sub problems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Ways of Thinking: If any problem is given, then we can think to solve the same in two ways:
1. Iteration
2. Recursion

Like:
To find the factorial of any input number we may think in two ways:

1. **Iterative Thinking:**
n! = n * (n-1) * (n-2) * ....* 2 * 1
Using for loop we can solve above problem like:
```
int i, n, f;
n = 5;
f = 1;
for ( i = n; i >= 1; i++)
{
    f = f * i;
}
print f;
```

2. **Recursive Thinking:**
n! = n * (n-1)!
Using recursive function we can solve above problem like:
```
int fact ( int n )
{
    int f;
    if ( n = = 0)
    {
            return 1;
    }
    else
    {
            f  = n * fact ( n-1);
            return n;
    }
}
```

In the above example, we have used a condition (i.e. n = = 1) to break the chain of recursion, this condition is known as **base case** or **base criteria.** In this concept larger value of a number is being converted as smaller by decreasing every time 1 till the base case is reached.
So we can say in other words a **base case** is a condition for which recursive function never invokes itself.

Need of Recursion: Recursion is a wonderful technique by which we can reduce the length of our code and make it easier to read and write. It has numerous benefits over the iteration technique. A task that can be defined with its similar subtask, recursion is one of the best solutions for it.

Properties of Recursion:
- Performing the same operations multiple times with different inputs.

- In every step, we try smaller inputs to make the problem smaller.
- Base criteria/condition is required to stop the recursion otherwise infinite loop will occur.

A Mathematical Interpretation: Let us consider a problem that a programmer has to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

$$f(n) = 1 + 2 + 3 + \ldots\ldots + n$$

Using Iteration

or

$$f(n) = \begin{cases} 1 & \text{if } n=1 \\ n + f(n-1) & \text{if } n>1 \end{cases}$$

Using Recursion

The difference between above two concepts is that in first concept a loop is used for finding the sum and in second concept function f(n) itself is being called inside the function f(n), this concept is called as recursion, and the function containing recursion is called recursive function, at the end, this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.

A particular problem is being solved by using recursion

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we want to find the sum of Numbers from 1 to n. The base case for sum function would be n = 1. We return 1 when n = 1.

```
int sum(int n)
{
   if (n == 1)
   {
      return 1;
   }
   else
   {
      res= n+sum(n-1);
      return res;
    }
}
```

Stack Overflow error occurs in recursion

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int sum(int n)
{
   // wrong base case (it may cause
   // stack overflow).
   if (n == 100)
   {
      return 1;
   }
   else
   {
```

```
        res= n+sum(n-1);
        return res;
      }
}
```

If sum(10) is called, it will call sum(9), sum(8), sum(7)…… and so on but the number will never reach 100. So, the base case is not reachable. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

Difference between direct and indirect recursion

A function fun is called direct recursive if it calls the same function fun. A function **P** is called indirect recursive if it calls another function say **Q** and **Q** calls **P** directly or indirectly. The difference between direct and indirect recursion is shown as:

```
// An example of direct recursion
void directRecFun()
{
   // Some code....
   directRecFun();
}

// An example of indirect recursion
void indirectRecFun1()
{
   // Some code...
   indirectRecFun2();
}
void indirectRecFun2()
{
   // Some code...
   indirectRecFun1();
}
```

Now we will describe:

Implementation of Stack in Recursion:

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure which is the stack data structure.

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Direct Recursion can be further classified into four following categories:

1. **Rear Recursion**
2. **Front Recursion.**
3. **Hierarchical Recursion.**
4. **Argumented Recursion.**

1. Rear Recursion:

If a function invokes itself and this invoke is the last statement in the recursive function, then it is known as **Rear Recursion.** That means no line after that recursive invoke. The function does not process anything so no return statement is required. This type of recursion is also known as Tail Recursion. In this case No Stack is required to store the result.

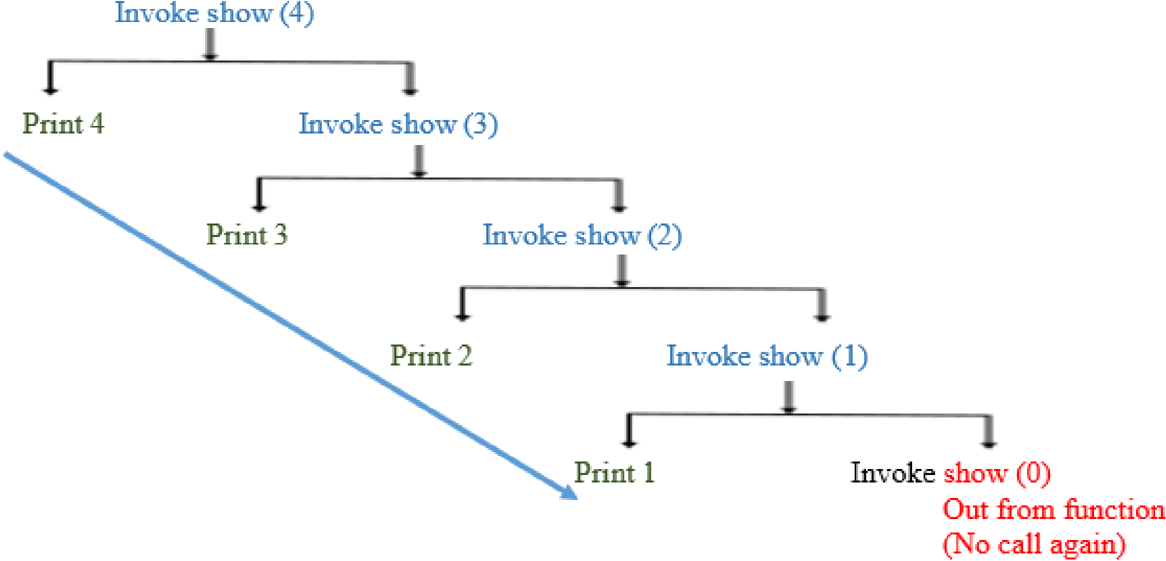
Example:

```
void show(int x)
{
   if (x > 0)
   {
      cout<< x << " ";
       show(x - 1);
   }
}
main()
{
   int x = 4;
   show(x);
   return 0;
}
```

Output

4 3 2 1

Explanation of above code is as follows:



2. **Front Recursion:**

If a function invokes itself and this invoke is the first statement in the recursive function, then it is known as **Front Recursion.** That means no line before that recursive invoke. The function does not process anything at the time of calling and all operations are performed at returning time, so there must be a return statement in such a recursive function. This type of recursion is also known as Head Recursion. In this case a Stack is required to store the results of every invoke of recursive function.

Example:

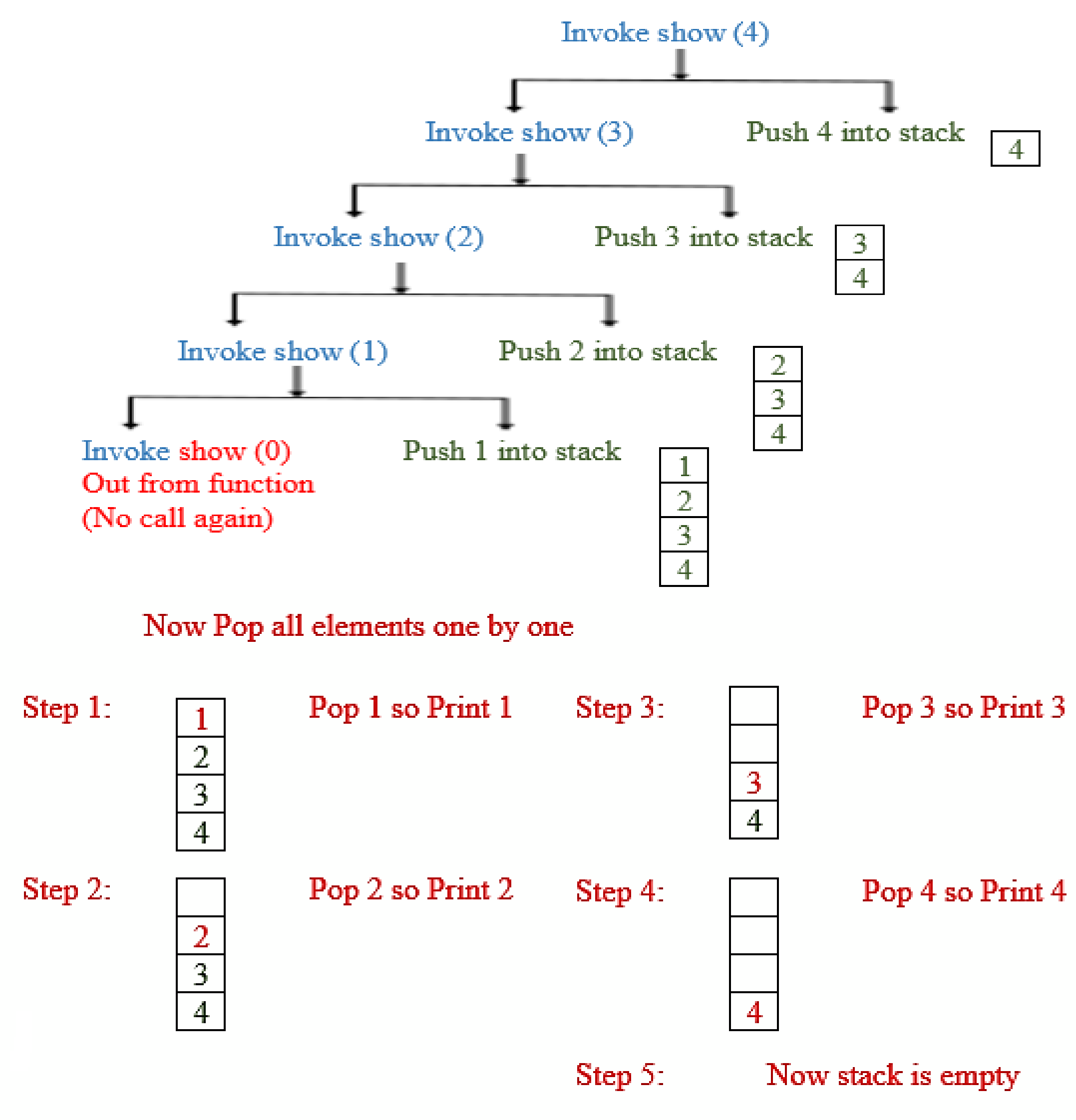
```
void show(int x)
```

```
{
   if (x > 0)
   {
     show(x - 1);
     cout<< x << " ";
     }
}
main()
{
   int x = 4;
   show(x);
   return 0;
}
```

Output

1 2 3 4

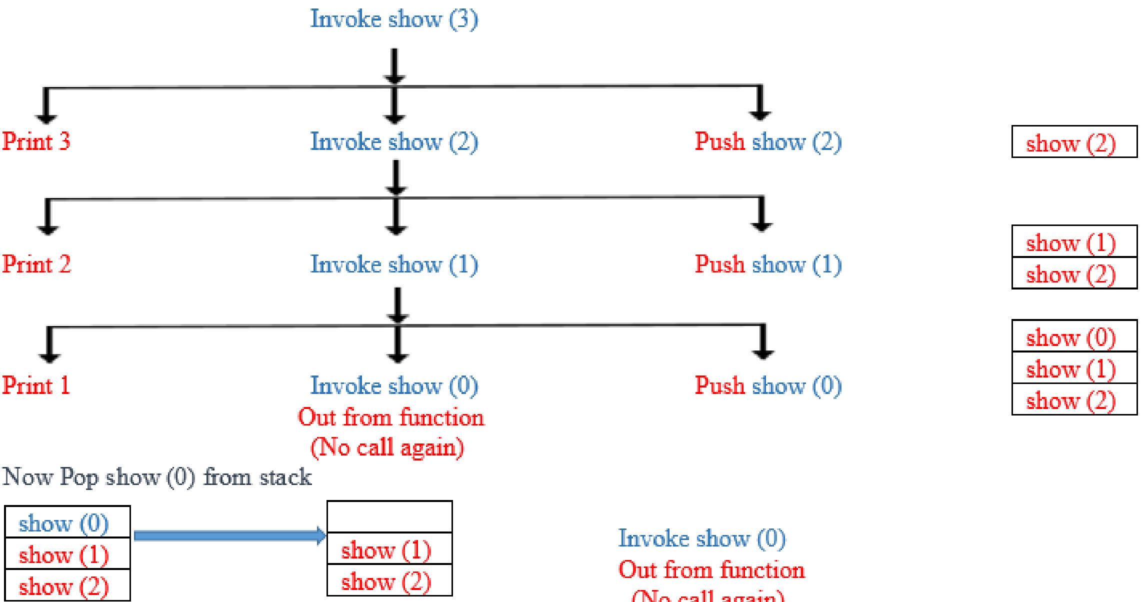
Explanation of above code is as follows:

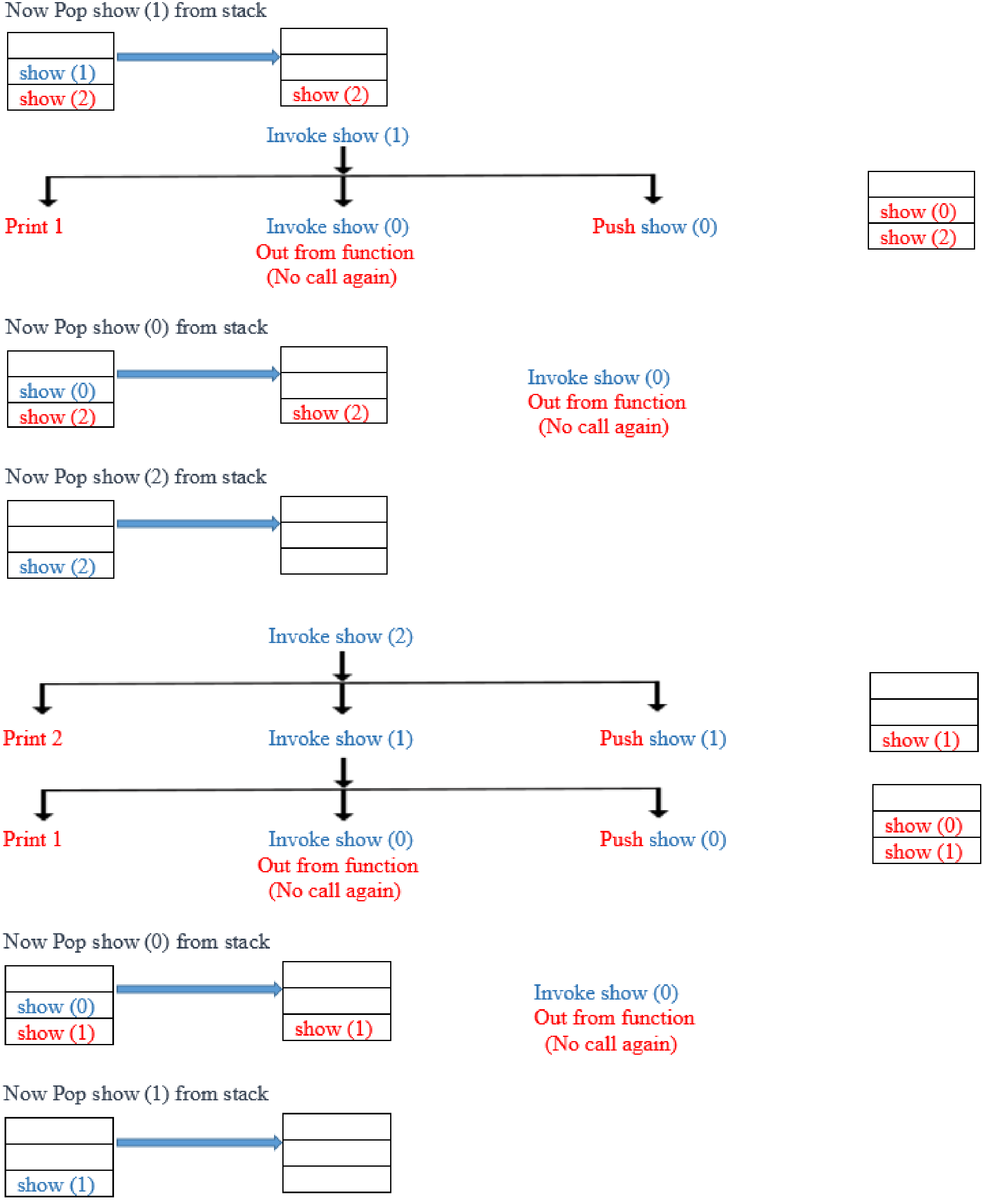


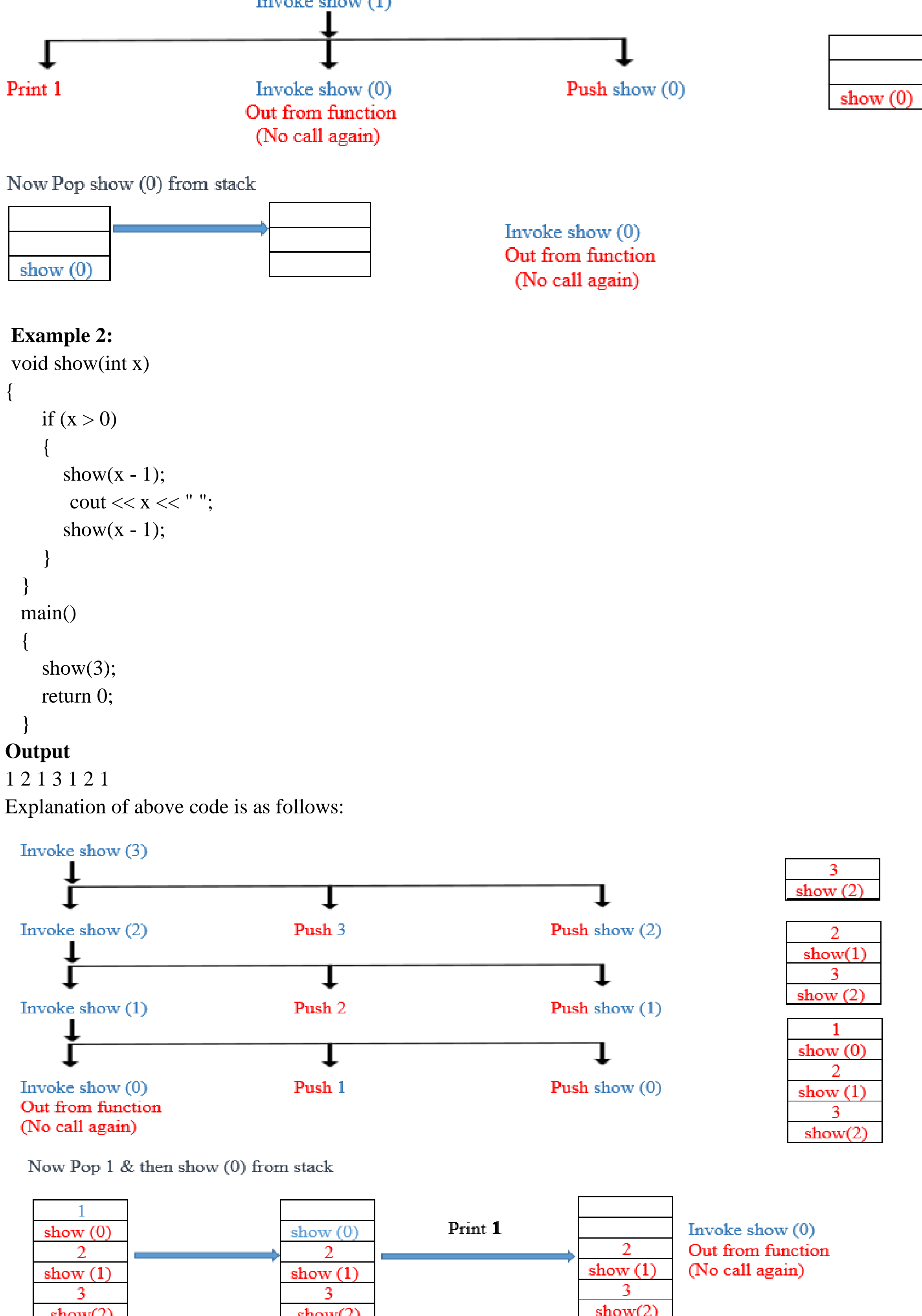
3. **Hierarchical Recursion:**
If a function invokes itself only for one time, then it is known as **Linear Recursion.** If a function invokes itself for more than once, then it is known as **Non-Linear Recursion.** This non-linear recursion is also termed as **Hierarchical or Tree Recursion**. If it has no return statement then no stack is required, but if it has a return statement then it requires stack to store the same.
Example 1:

```
void show(int x)
{
   if (x > 0)
   {
      cout << x << " ";
      show(x - 1);
      show(x - 1);
   }
}
main()
{
   show(3);
   return 0;
}
```

Output
3 2 1 1 2 1 1
Explanation of above code is as follows:




Now Pop show (1) from stack
show (1)
show (2)
show (2)
Invoke show (1)
Print 1
Invoke show (0)
Out from function
(No call again)
Push show (0)
show (0)
show (2)
Now Pop show (0) from stack
show (0)
show (2)
show (2)
Invoke show (0)
Out from function
(No call again)
Now Pop show (2) from stack
show (2)
Invoke show (2)
Print 2
Invoke show (1)
Push show (1)
show (1)
Print 1
Invoke show (0)
Out from function
(No call again)
Push show (0)
show (0)
show (1)
Now Pop show (0) from stack
show (0)
show (1)
show (1)
Invoke show (0)
Out from function
(No call again)
Now Pop show (1) from stack
show (1)



Example 2:

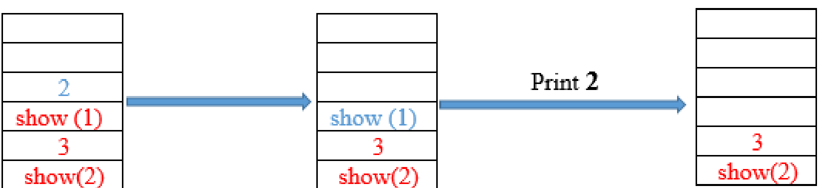
```
void show(int x)
{
    if (x > 0)
    {
        show(x - 1);
         cout << x << " ";
        show(x - 1);
    }
 }
 main()
 {
    show(3);
    return 0;
 }
```

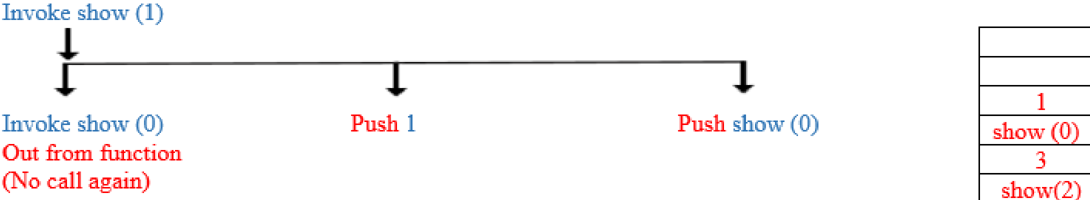
Output

1 2 1 3 1 2 1

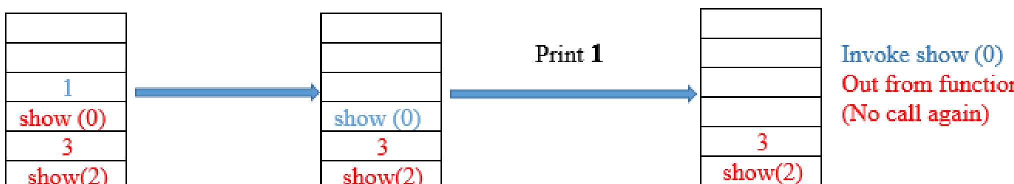
Explanation of above code is as follows:

Now Pop 2 & then show (1) from stack

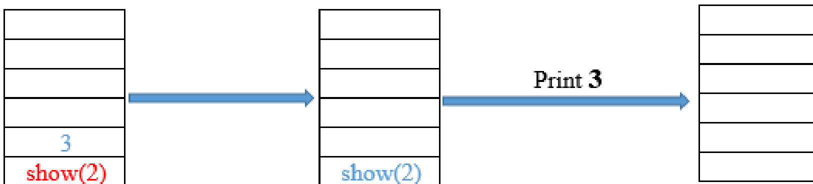


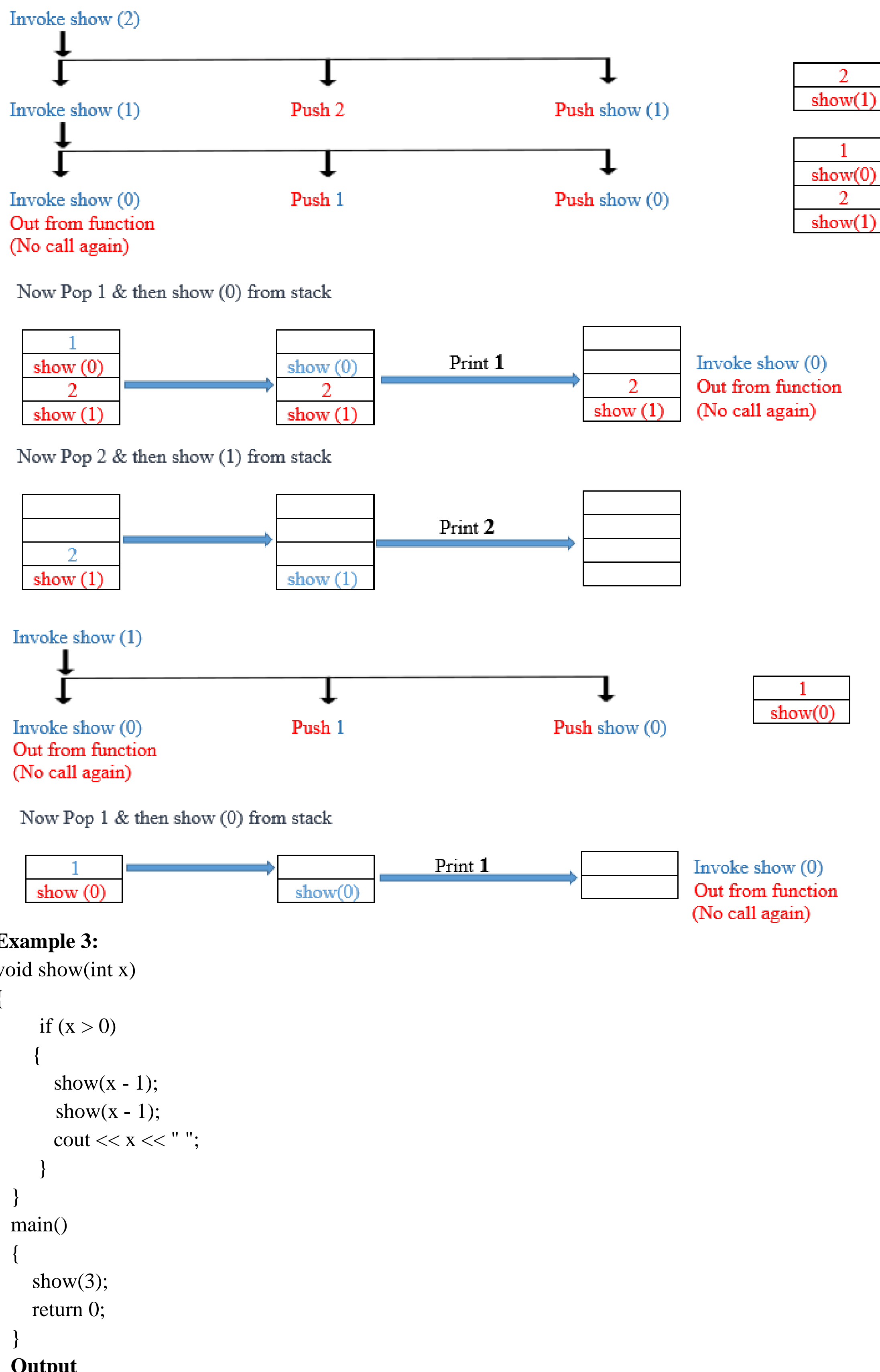


Now Pop 1 & then show (0) from stack



Now Pop 3 & then show (2) from stack





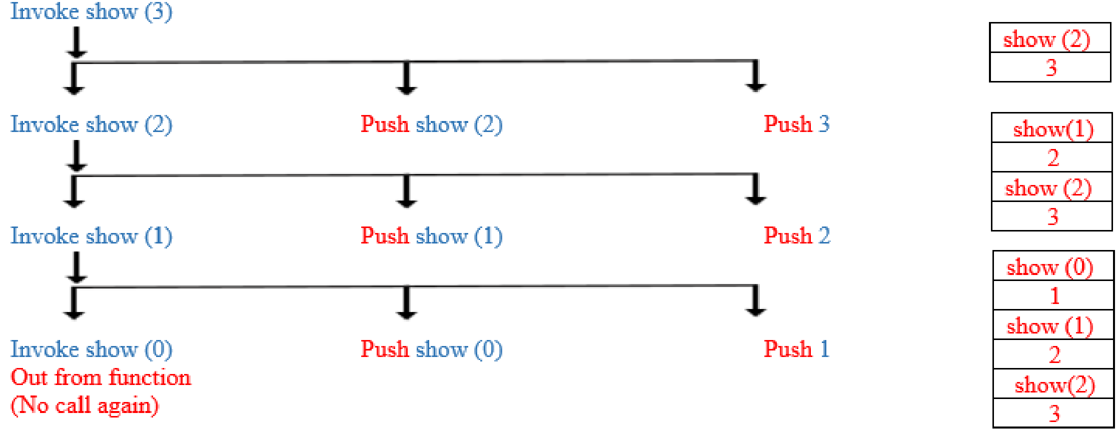
Example 3:

```
void show(int x)
{
      if (x > 0)
     {
        show(x - 1);
        show(x - 1);
        cout << x << " ";
     }
 }
 main()
 {
    show(3);
    return 0;
 }
```

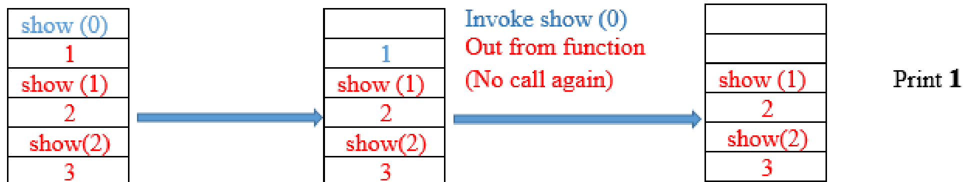
Output

1 1 2 1 1 2 3

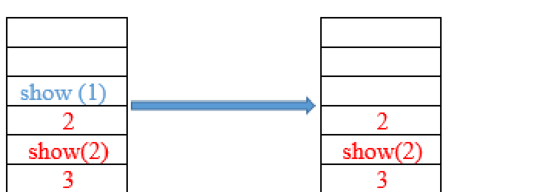
Explanation of above code is as follows:

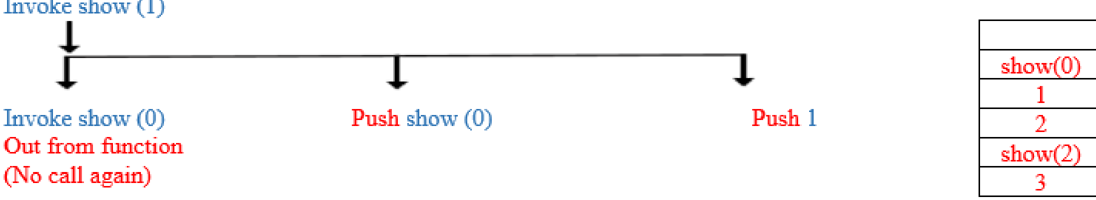


Now Pop show (0) & then 1 from stack

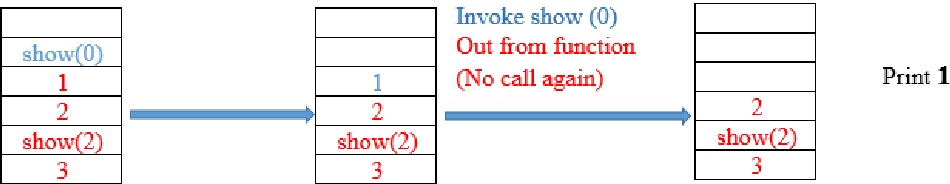


Now Pop show (1) from stack

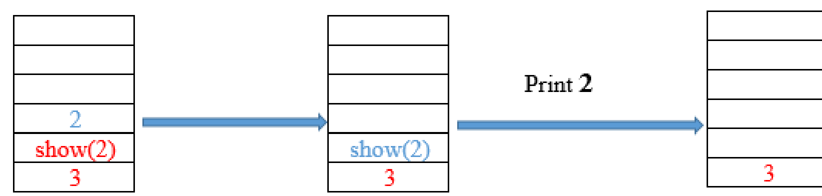


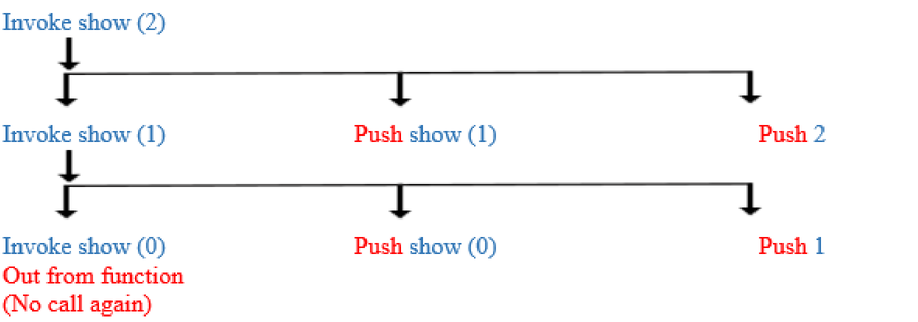


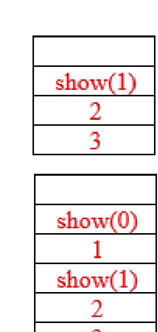
Now Pop show (0) & then 1 from stack



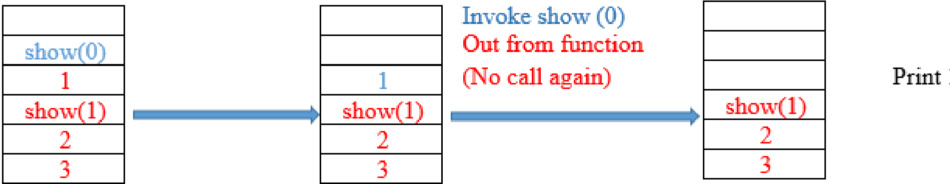
Now Pop 2 & then show (2) from stack

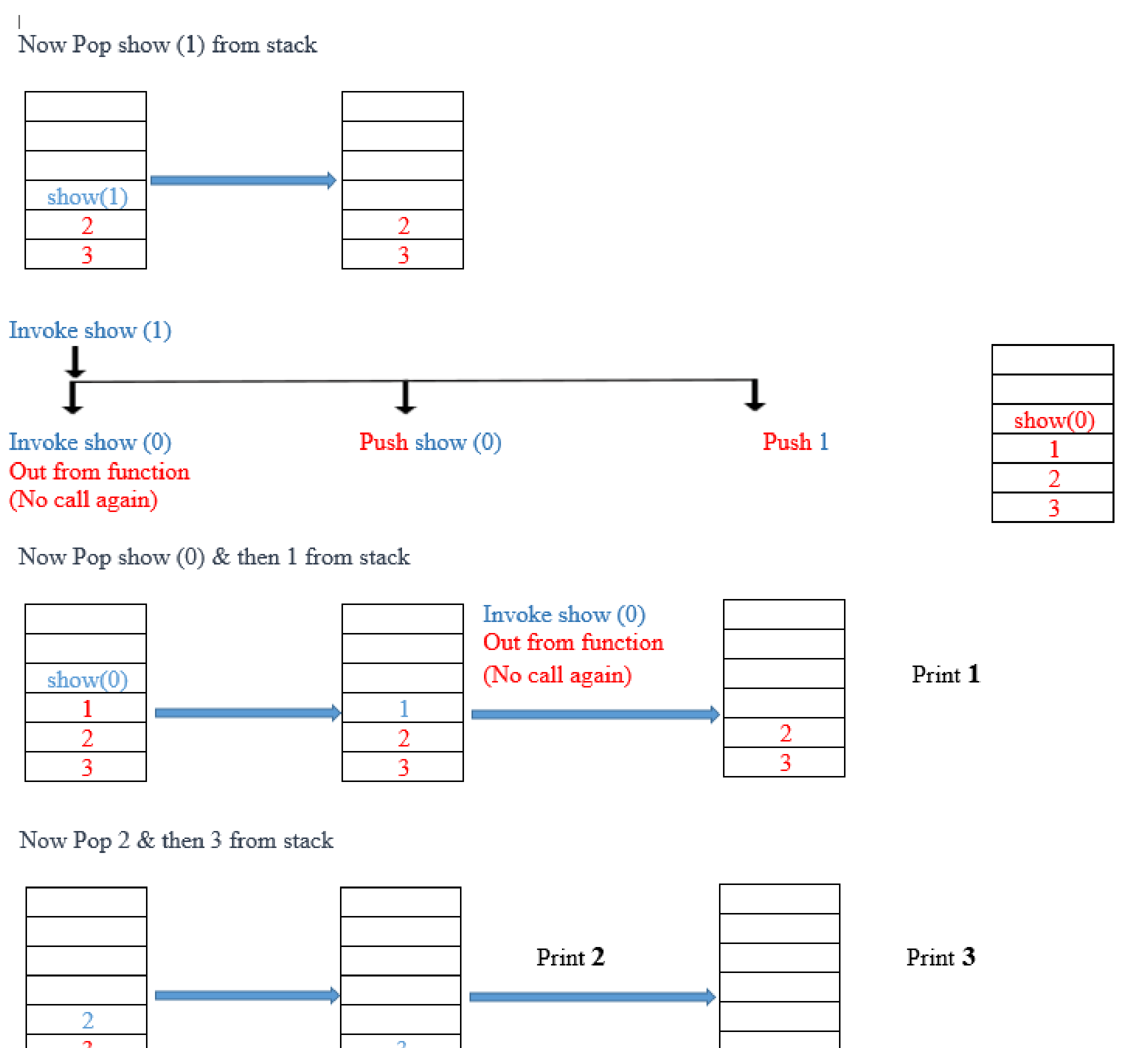






Now Pop show (0) & then 1 from stack





4. **Argumented Recursion:**

If a function invokes itself and in this invoke a parameter passed to this function is again that same recursive function, then it is known as **Argumented Recursion.** This is also known as Nested Recursion. That means **"recursion inside recursion".** To work with this recursion a stack is required to store the results.

Example:

```
int show(int n)
{
   if (n >= 100)
      return n - 10;
   return show(show(n + 11));
}
main()
{
   int r;
   r = show(95);
   cout << " " << r;
```

```
    return 0;
  }
```

Output

90

Explanation of above code is as follows:

Invoke show (95)

Invoke show (show (106))
(95+11)

Invoke show (96)
(106-10)

Invoke show (show (107))
(96+11)

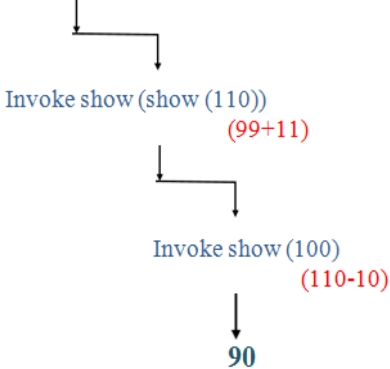
Invoke show (97)
(107-10)

Invoke show (show (108))
(97+11)

Invoke show (98)
(108-10)

Invoke show (show (109))
(98+11)

Invoke show (99)
(109-10)



REFERENCE

[1] Introduction to Algorithms, 3rd Edition (The MIT Press).
[2] Data Structures and Algorithms in Java.
[3] Data Structures (Revised First Edition) | Schaum's Outline Series by Seymour Lipschutz.
[4] T&P Of Data Structures With C+ by HUBBARD, McGraw Hill.
[5] Algorithms and Data Structures Foundations and Probabilistic Methods for Design and Analysis By Helmut Knebl.
[6] Algorithms in a Nutshell By George T. Heineman, Gary Pollice, Stanley Selkow .
[7] Data Structures and Algorithm Analysis in C++, Third Edition By Clifford A. Shaffer.
[8] Arrays: A Theoretical Approach of Memory Allocation, Pankaj Kumar Gupta & Dr. P. K. Tyagi, International Journal of Essential Sciences, Vol-14 No. 1 & 2 2020.
[9] Conceptual Discussion on Operations of Array: Traveral, Insertion & Deletion by Pankaj Kumar Gupta & Dr. P. K. Tyagi, International Journal of Research in all subjects in Multi Languages, Vol-10 Issue 3, March: 2022.
[10] https://www.programiz.com/dsa/stack
[11] https://www.biitonline.co.in
[12] https://javascript.info/
[13] https://www.geeksforgeeks.org/stack-data-structure/