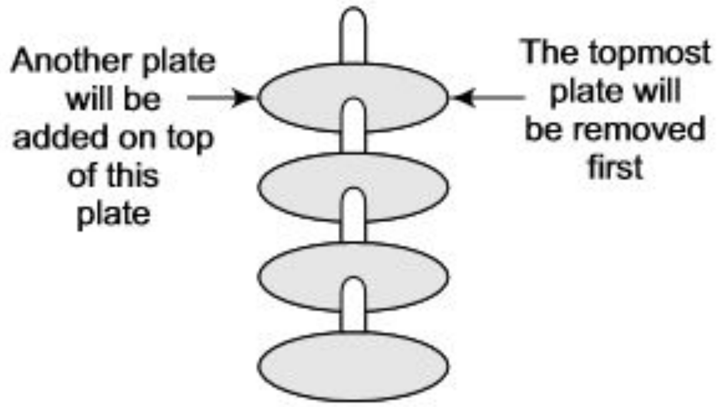
Unit-2

Stack & Queue

Stack

- Stack is an important data structure which stores its elements in an ordered manner.
- Take an analogy of a pile of plates where one plate is placed on top of the other as shown in the following figure.


Another plate will be added on top of this plate
The topmost plate will be removed first

- A plate can be removed from the topmost position. Hence, you can add and remove the plate only at/from one position that is, the topmost position

Stack

- ***“A stack is a linear data structure which can be implemented either using an array or a linked list”.***

- The elements in a stack are added and removed only from one end, which is called **top**.

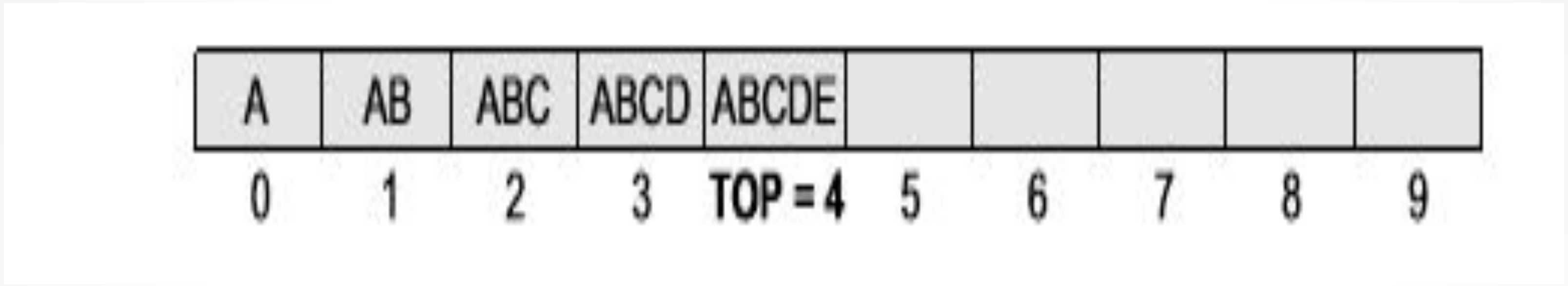
- Hence, a stack is called a **LIFO (Last In First Out)** data structure as the element that was inserted last is the first one to be taken out.

Array representation of Stack

- In computer memory, stacks can be represented as a linear array.
- Every stack has a variable **TOP** associated with it.
- TOP is used to store <u>the address of topmost element of the stack</u>.
- There is another variable named **MAX** which will be used <u>to store the maximum number of elements that the stack can hold</u>.
- If **TOP=NULL** means that **<u>the stack is Empty</u>**.
- If **TOP=MAX-1** means that **<u>the stack is Full.</u>**

Array representation of Stack

- An example stack :



- The stack in above figure shows that **TOP=4**, so insertions and deletions will be done at this position.
- In this stack, five more elements can still be stored.

Operations on a Stack

- A stack has four basic operations:

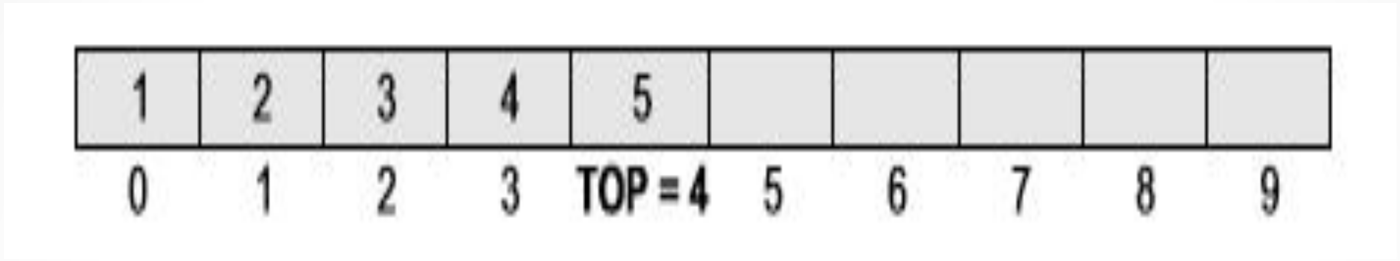
1. **Push:** adds an element to the top of the stack.
2. **Pop:** removes the element from the top of the stack.
3. **Peek:** returns the value of the topmost element of the stack of specified index.
4. **Update:** changes the value of element given by user of the stack.

1. Push Operation

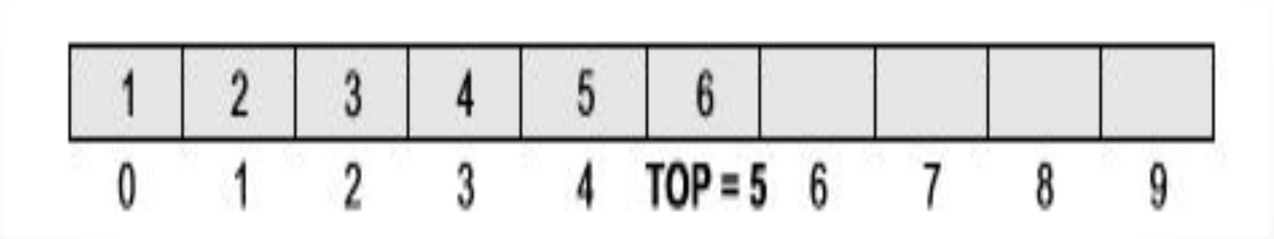
- The **PUSH** operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- However before inserting the value, we must first check if **TOP=MAX-1**, as it would mean that the stack is full and no further insertions can be done on it.
- If an attempt is made to insert an element in a stack that is already full, an **OVERFLOW message** is printed.

1. Push Operation

- An example of stack:

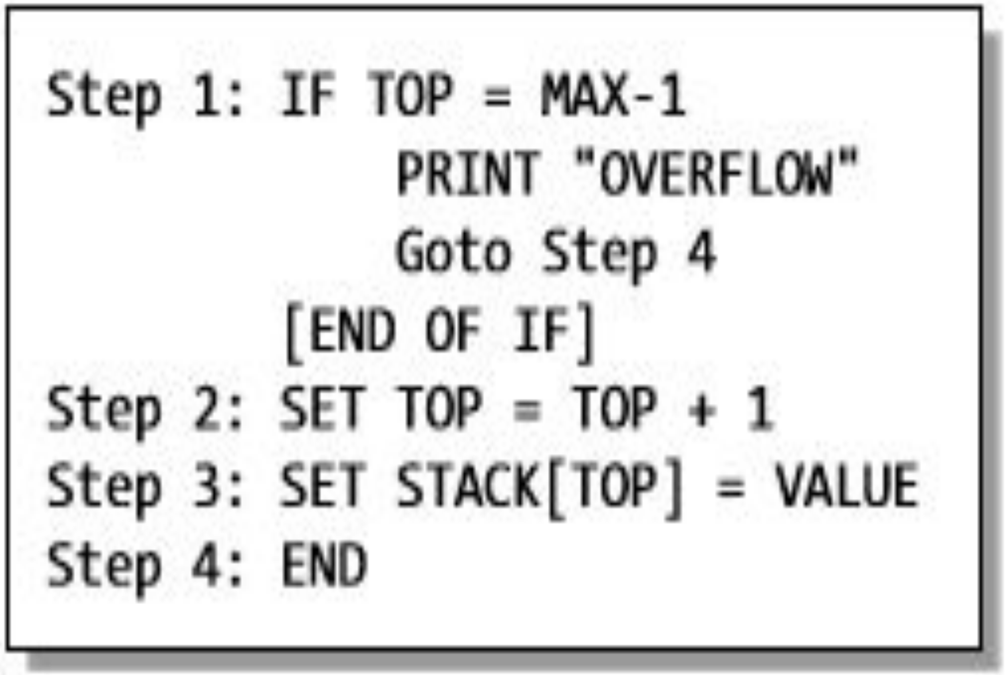


- Stack after insertion:



1. Push Operation

- Algorithm to insert an element in a stack:



```
Step 1: IF TOP = MAX-1
              PRINT "OVERFLOW"
              Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

2. Pop Operation

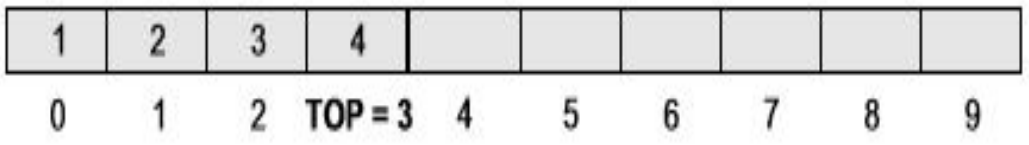
- The **POP** operation is used <u>to delete an element from the stack</u>.

- However before deleting the value, we must check if **TOP=NULL**, as it would mean that the stack is empty and therefore no further deletions can be done on it.

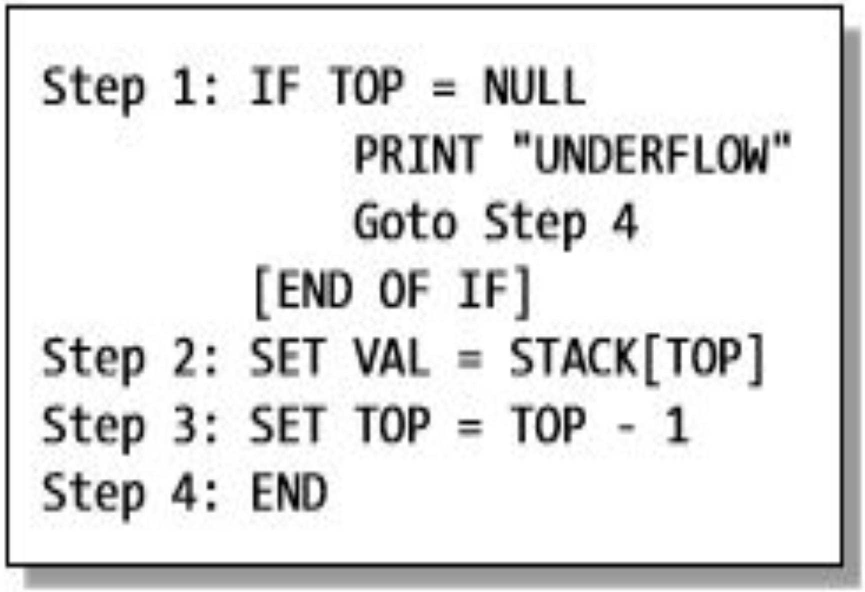
- If an attempt to delete a value from a stack that is already empty is made, an **UNDERFLOW message** is printed.

2. Pop Operation

- Stack after deletion:



- Algorithm to delete an element from the stack:



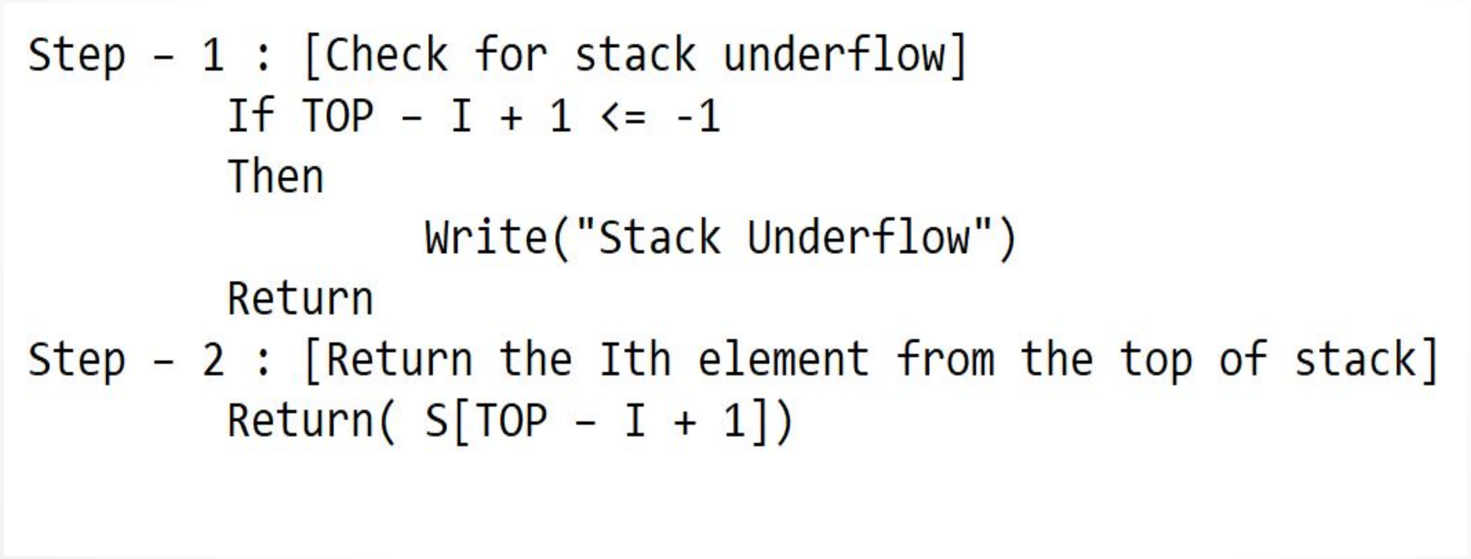
```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

3. Peek Operation

- **PEEK** is an operation that returns the value of the element of the stack of specified index without deleting it from the stack.

3. Peek Operation

- Given a vector S (consisting of N elements) representing a sequentially allocated stack, and a pointer TOP denoting the top element of the stack, this function returns the value of the Ith element from the top of the stack. The element is not deleted by this function.



```
Step - 1 : [Check for stack underflow]
        If TOP - I + 1 <= -1
        Then
                Write("Stack Underflow")
        Return
Step - 2 : [Return the Ith element from the top of stack]
        Return( S[TOP - I + 1])
```

4. Update Operation

- UPDATE is an operation that updates or changes the value of specified element with specific index.
- However, the update operation first checks if the stack is empty or contains some element.

```
Step 1: [ Check for Stack Underflow]
        IF TOP – i + 1 = NULL
                PRINT "UNDERFLOW"
        [END OF IF]
Step 2: [ Change ith element from the top of Stack]
            Set STACK[TOP-i+1] = X
Step 3: END
```

Applications of Stack

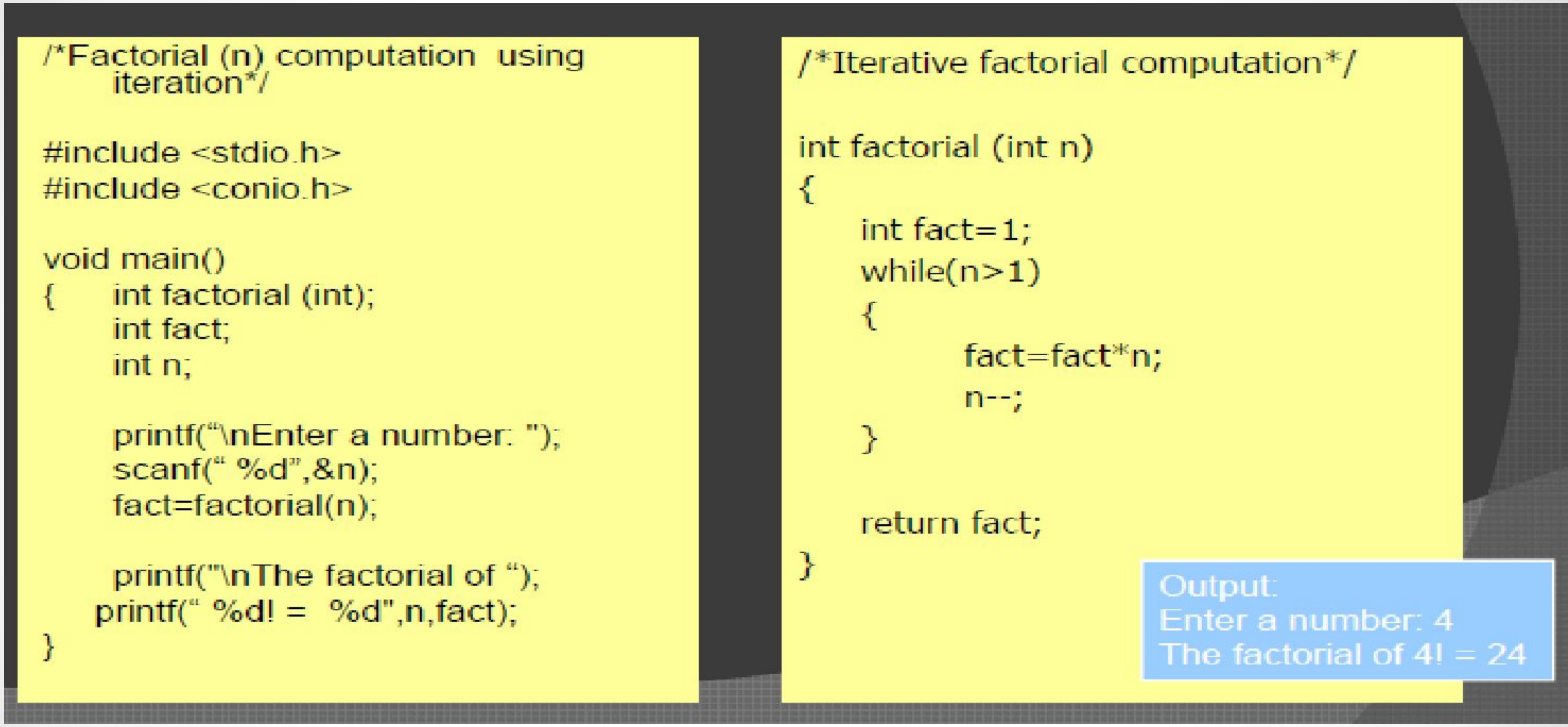
- ❑ Recursion
- ❑ Balancing Symbol
- ❑ Polish Notations

Recursion

- A function calls itself is called **Recursion**.
- Recursion is a repetitive process in which function calls itself repeatedly until some predefined condition is met.
- The recursive function must have a stopping condition (Termination/ Anchor Condition) or else the function would never terminate. So ultimately program **goes into infinite loop.**
- Examples where recursion has been used are:

  Factorial, Fibonacci sequence (number) and Tower of Hanoi, etc..

Programming (factorial with iteration)

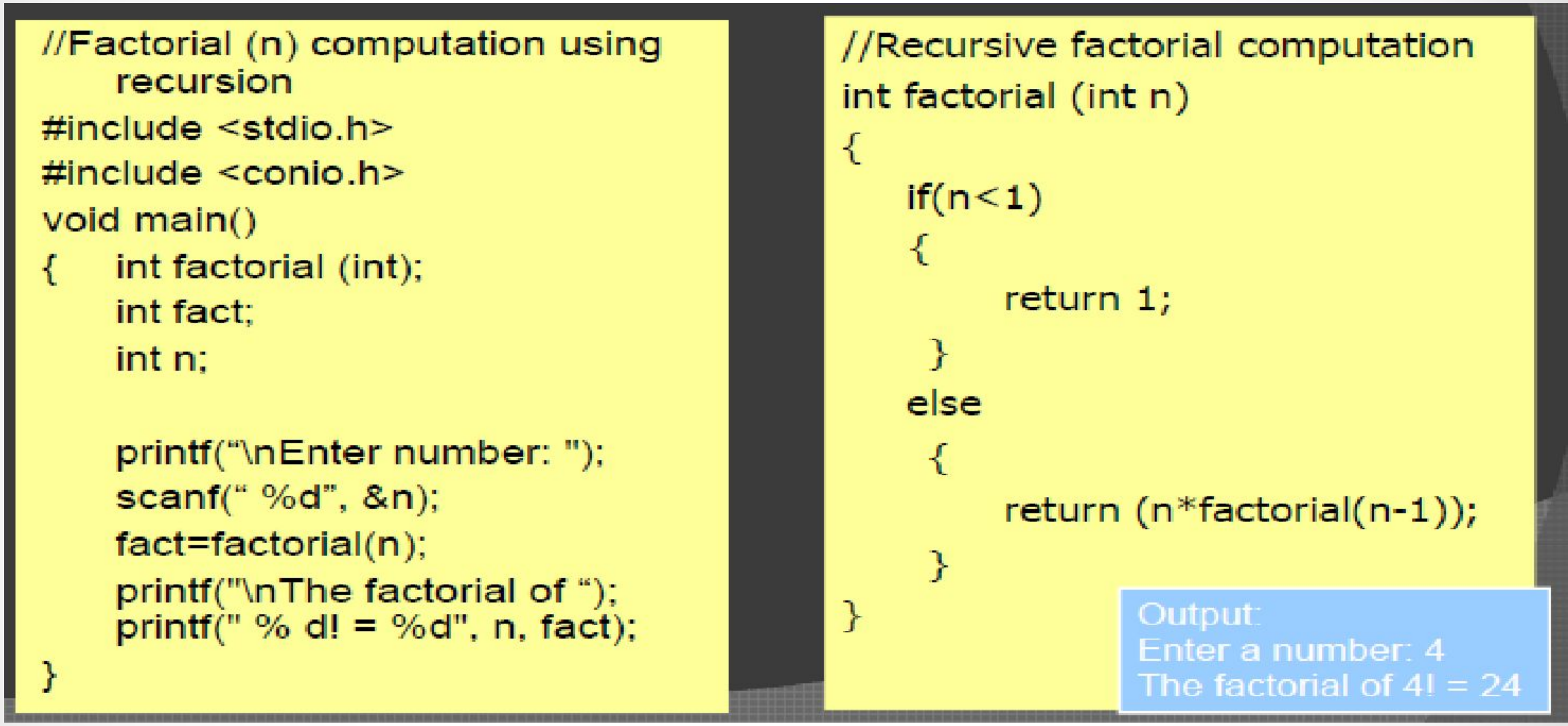


```
/*Factorial (n) computation  using
      iteration*/

#include <stdio.h>
#include <conio.h>

void main()
{      int factorial (int);
       int fact;
       int n;

       printf("\nEnter a number: ");
       scanf(" %d",&n);
       fact=factorial(n);

       printf("\nThe factorial of ");
     printf(" %d! =  %d",n,fact);
}
```

```
/*Iterative factorial computation*/

int factorial (int n)
{
      int fact=1;
      while(n>1)
      {
            fact=fact*n;
            n--;
      }

      return fact;
}
```

```
Output:
Enter a number: 4
The factorial of 4! = 24
```

Programming (factorial with recursion)

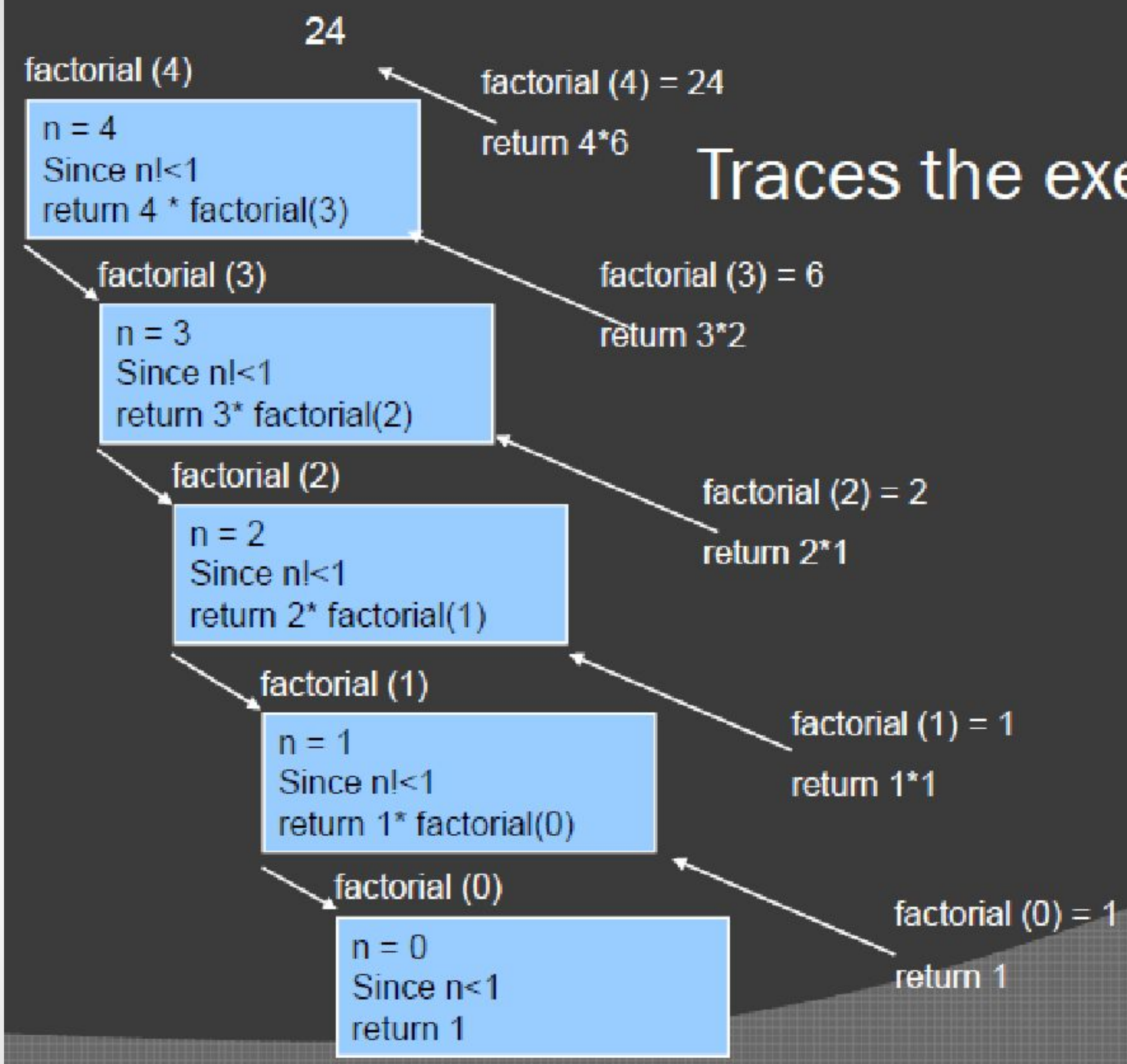


```
//Factorial (n) computation using
     recursion
#include <stdio.h>
#include <conio.h>
void main()
{    int factorial (int);
     int fact;
     int n;

     printf("\nEnter number: ");
     scanf(" %d", &n);
     fact=factorial(n);
     printf("\nThe factorial of ");
     printf(" % d! = %d", n, fact);
}
```

```
//Recursive factorial computation
int factorial (int n)
{
    if(n<1)
    {
          return 1;
     }
    else
     {
          return (n*factorial(n-1));
     }
}
```

```
Output:
Enter a number: 4
The factorial of 4! = 24
```

Traces the execution



Recursion

```
#include<stdio.h>
#include<conio.h>
int factorial(int n);
void main()
{
    int n,z;
    clrscr();
    printf("\n Enter Value : ");
    scanf("%d",&n);

    z=factorial(n);
    printf("\n Factorial is : %d",z);

}
```

Recursion

```
int factorial(int n)
{
   int f;
   if(n<1)
     return 1;
   else
     f=n*factorial(n-1);
   return f;
}
```

Iteration Vs. Recursion

Sr . No.	Basis for Comparison	Recursion	Iteration
1	Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
2	Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
3	Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.

Iteration Vs. Recursion

Sr . No.	Basis for Comparison	Recursion	Iteration
4	Condition	If the function does not coverage to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
5	Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
6	Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
7	Speed	Slow in execution.	Fast in execution.

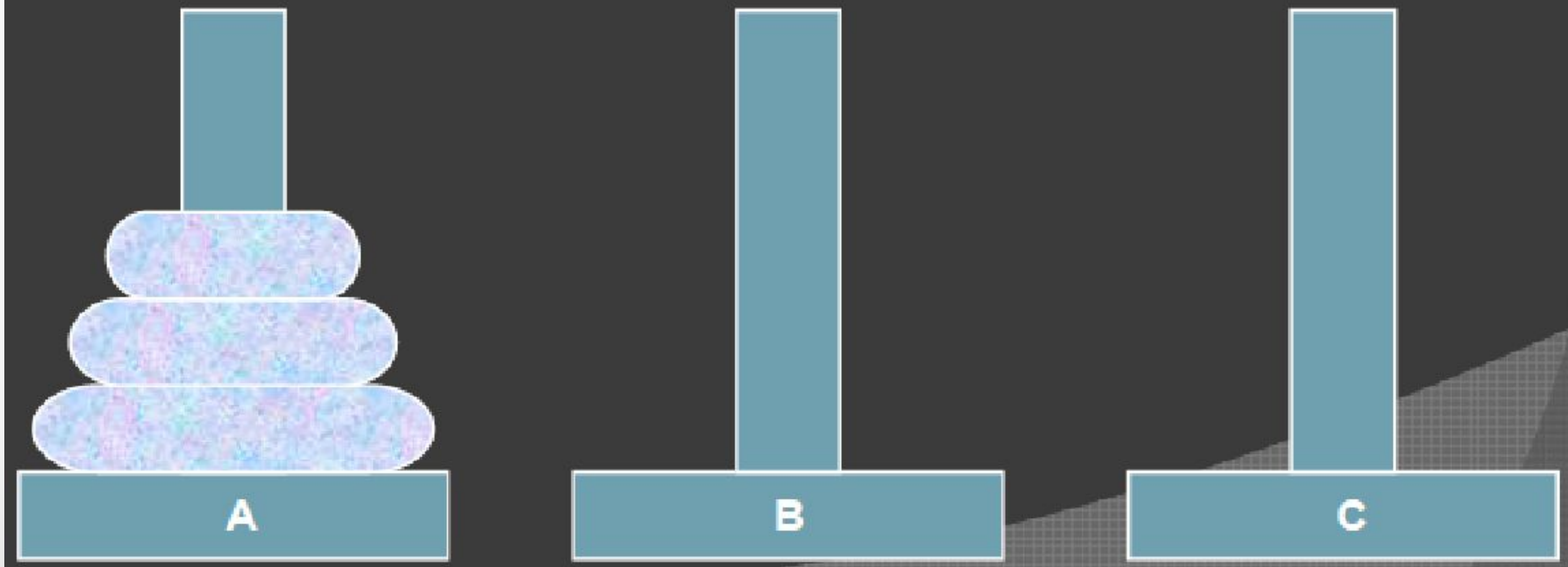
Iteration Vs. Recursion

Sr . No.	Basis for Comparison	Recursion	Iteration
8	Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
9	Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

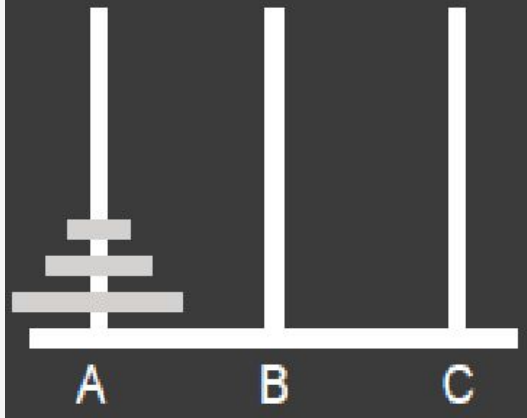
Towers of Hanoi (Recursion)

- Suppose 3 pegs labeled A, B, C and a set of disks (varying sizes)
- Arranged the biggest disk at the bottom to the smallest disk at the top
- Each peg must Transfer the disk, one by one from peg A to C, using peg B as an auxiliary
- always be arranged form the biggest at the bottom to the smallest at the top .

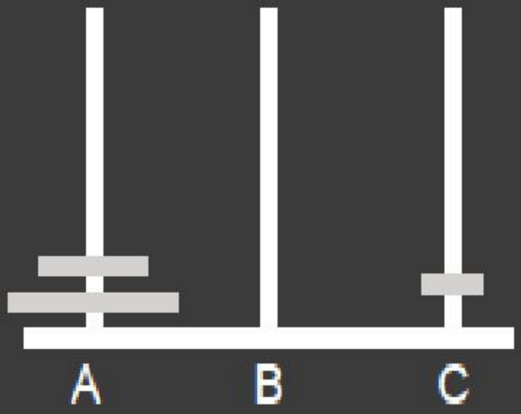
Illustration: Initial state for Towers of Hanoi



Solution to the Towers of Hanoi for disks = 3

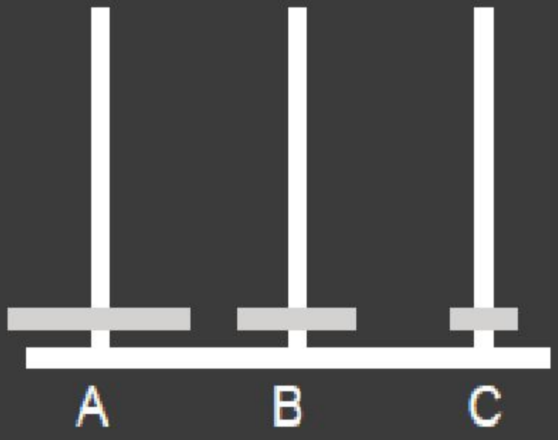


Initial

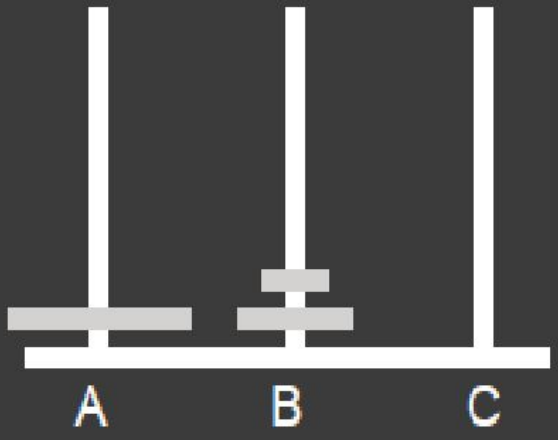


Step 1: A -> C

Continue...

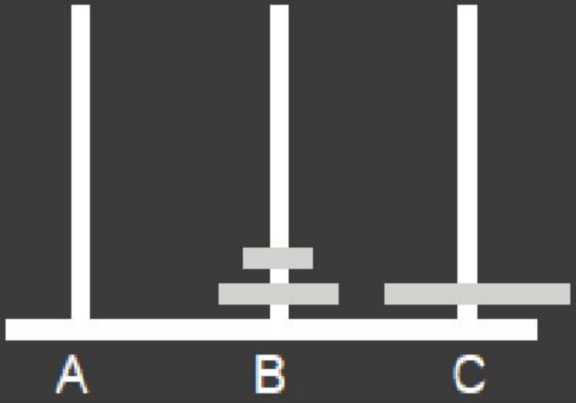


Step 2: A -> B

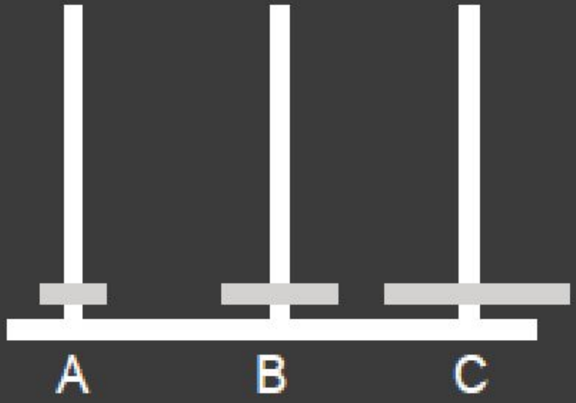


Step 3: C -> B

Continue...

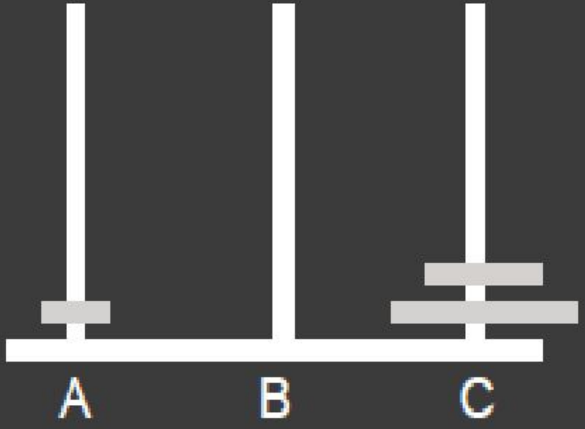


Step 4: A -> C

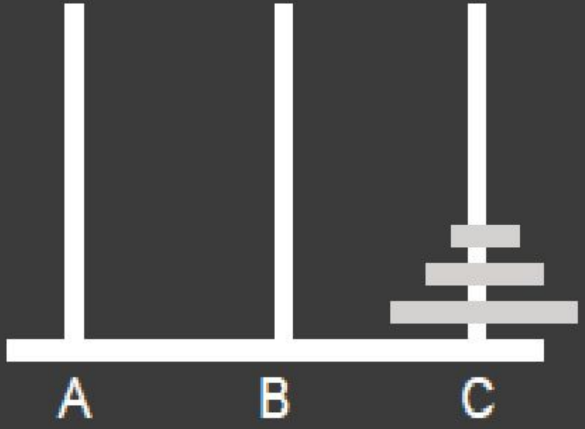


Step 5: B -> A

Continue...



Step 6: B -> C



Step 7: A -> C

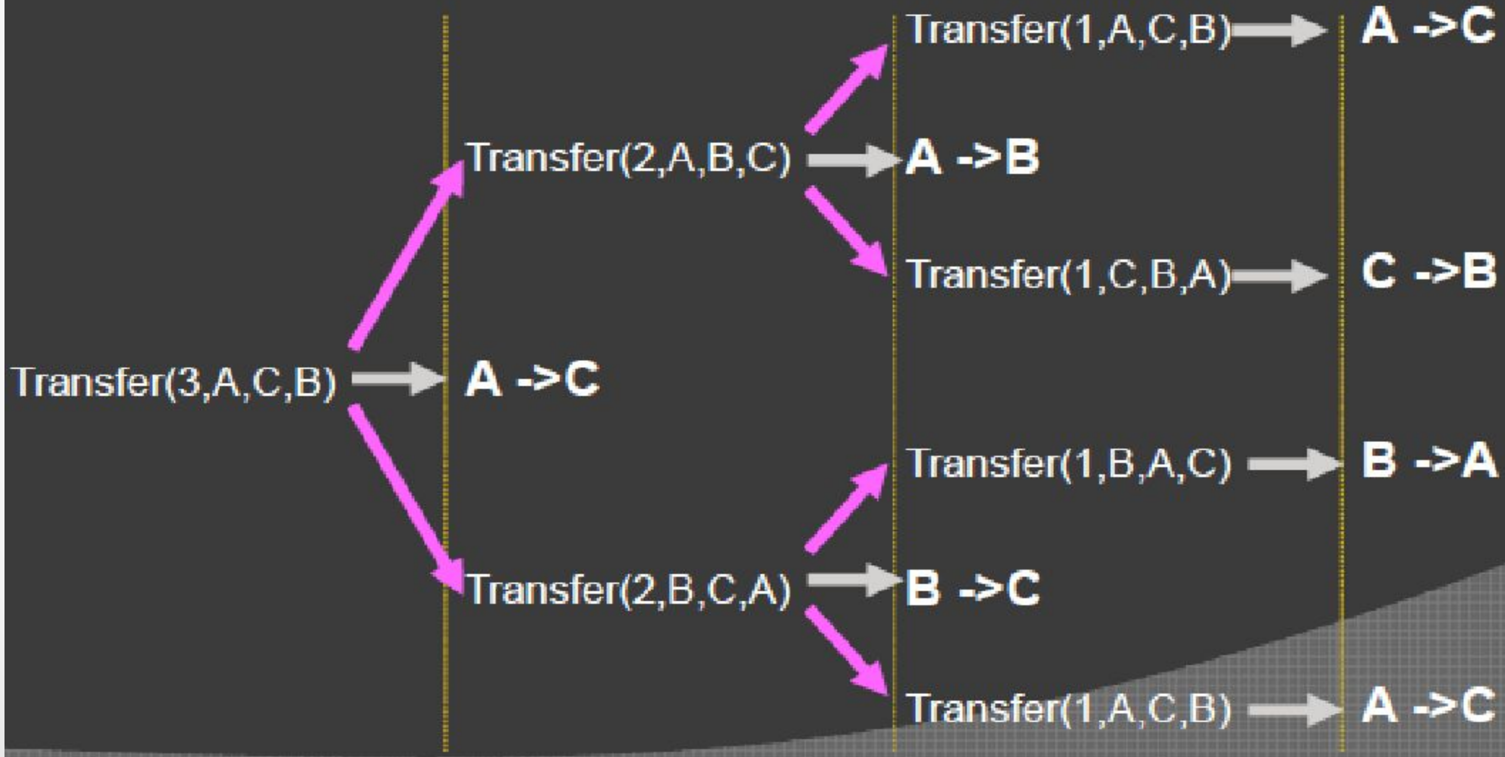
Programming

```
//Tower of Hanoi
#include <stdio.h>
#include <conio.h>

void main()
{
      void transfer(int, char,char,char);
      int n;
      printf("\nEnter number of disks: ");
      scanf(" %d", &n);
      if(n>0)
              transfer(n , 'A','C','B');

  getch();
```

```
Enter number of disks: 3
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
```

```
void transfer(int n, char from, char to, char work)
{
      if (n==1)
      printf("\nMove disk 1 from peg %c to peg %c", from,to);

      else
      {

      transfer(n-1, from, work, to);

      printf("\nMove disk %d from peg %c to peg %c",n,from,to);

      transfer(n-1,work, to , from);
      }
}
```

Illustration of solution to Towers of Hanoi problem for n = 3



Balancing Symbol

- Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out hundred lines of diagnostics without identifying the real error.
- A useful tool in this situation is program that checks whether everything is balanced.
- Thus, every right brace, bracket, and parenthesis must correspond to its left counterpart.
- The sequence [()] is legal, but [(] is wrong.

- Obviously, it is not worthwhile writing a huge program for this, but it turns out that it is easy to check these things.
- For simplicity, we will just check for balancing parentheses, brackets, and braces and ignore any other character that appears.
- The simple algorithm uses stack and is as follows:
    - Make an empty stack.
    - Read characters until end of file.
    - If it is a closing symbol, then if the stack is empty report an error.
    - Otherwise, pop the stack.
    - If the symbol popped is not the corresponding opening symbol, then report an error.
    - At end of file, if the stack is not empty report an error.

Polish Notations

- Following are three different but equivalent notations od writing algebraic expression:

1. Infix Notation
2. Postfix Notation
3. Prefix Notation

Infix Notation

- In the infix notation, the operator is placed between the operands.
- For example, A + B; here, the plus operator is place between the two operands A and B.
- Although we find it easy to write expressions using the infix notation, computers find them difficult to parse as they need a lot information such as operator precedence, associativity rules and brackets which dictate the rules, to evaluate the expression.
- So computers work more efficiently with expressions written using prefix and postfix notations.

Postfix Notation

- In the postfix notation, as the name suggests the operator is placed after the operands.
- For example, if an expression is written as A+B in infix notation, the same expression is written as AB+ in the postfix notation.
- The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.
- Expression (A + B) * C is written is postfix notation as:

  AB+C*

Postfix Notation

- Postfix operation does not even follow the rules of operator precedence.
- The operator that occurs first in the expression is operated first on the operands.
- For example, given a postfix notation AB+C*; while evaluation addition will be performed prior to multiplication.
- **Exercise: Convert following Infix expressions into Postfix expressions:**

(A – B) * (C + D)

Prefix Notation

- ❑ Prefix notation is same as postfix notation but having only a difference: the operator is placed before the operands.
- ❑ For example: if A + B is an expression in the infix notation, then the corresponding expression in a prefix notation is given by: +AB
- ❑ **Exercise: Convert following Infix expressions into Prefix expressions:**

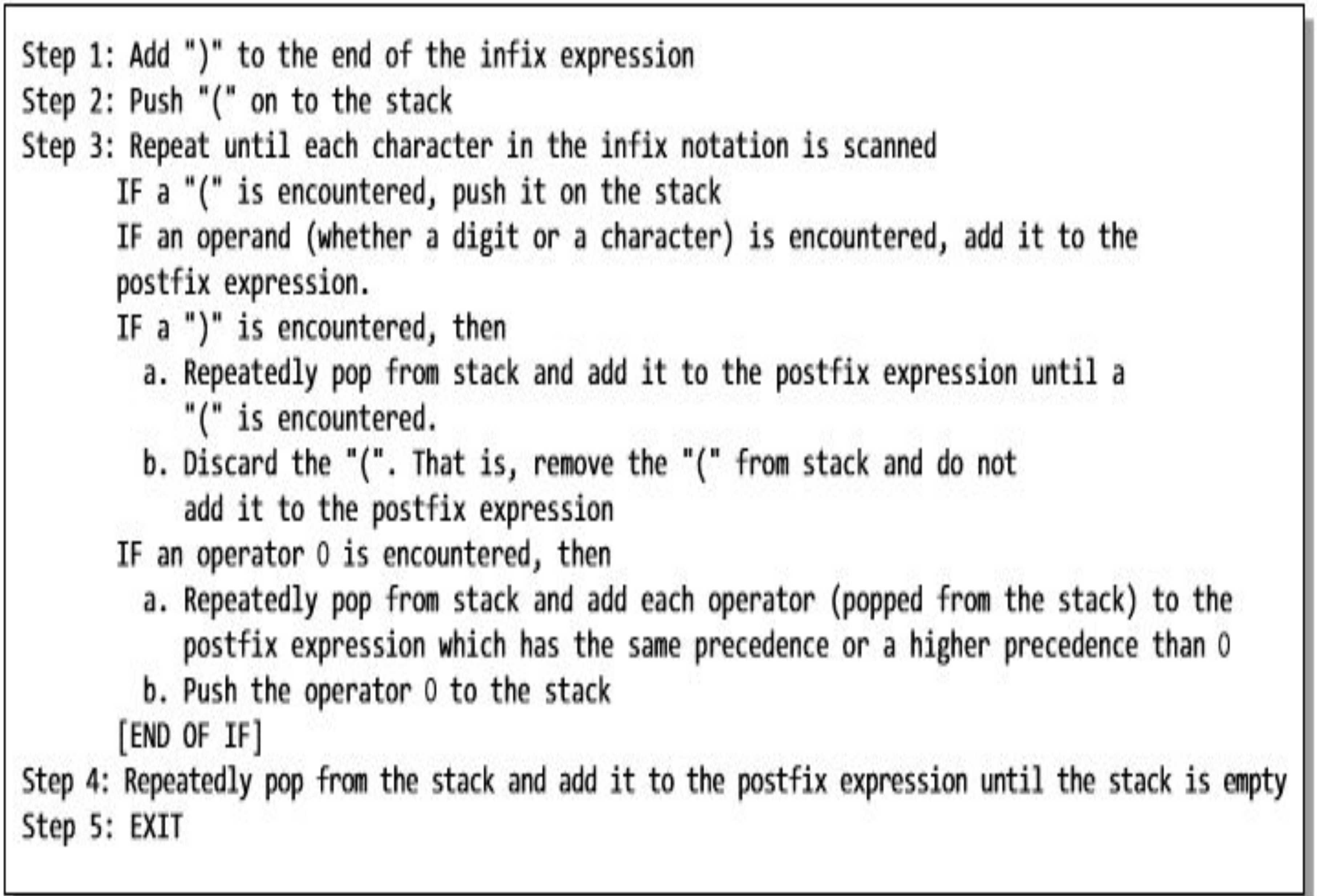
1. (A + B) * C
2. (A – B) * (C + D)

Convert Infix to Postfix Notation

- Let I be an algebraic expression written in infix notation.
- I may contain parentheses, operands, and operators.
- For simplicity of the algorithm, only +,-,*,/ and % operators have been used.
- Given algorithm will transform an infix expression into postfix expression.

Convert Infix to Postfix Notation

- The algorithm accepts an infix expression that may contain operators, operands and parentheses.
- The operators with the same precedence are performed from left-to-right.
- The algorithm uses a stack to temporarily holds operators.
- The postfix expression is obtained from left to right using the operands from the infix expression and the operators are removed from the stack.
- The very first step is to push a left parenthesis in the stack and add a corresponding right parenthesis at the end of the infix expression.
- The algorithm is repeated until the stack is empty.



```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        IF a ")" is encountered, then
          a. Repeatedly pop from stack and add it to the postfix expression until a
             "(" is encountered.
          b. Discard the "(". That is, remove the "(" from stack and do not
             add it to the postfix expression
        IF an operator O is encountered, then
          a. Repeatedly pop from stack and add each operator (popped from the stack) to the
             postfix expression which has the same precedence or a higher precedence than O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Convert Infix to Postfix Notation

Operator	PRIORITY
^	1
* , /	2
+ , -	3

<u>Rules</u>

No Two Operator of same priority can stay together in the stack column.
If Previous Operator having the higher priority than current operator then previous operator will be popped out.

Convert Infix to Postfix Notation

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

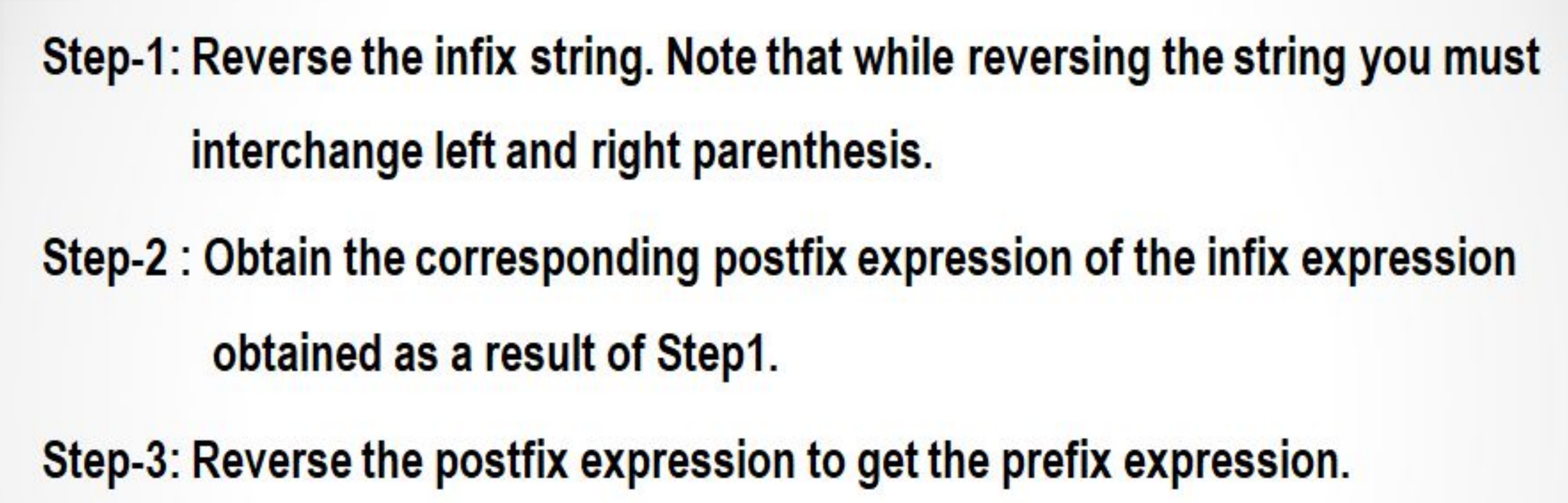
Infix Character Scanned	STACK	Postfix Expression
(	(	
A	(	A
+	(+	A
B	(+	AB
/	(+/	AB
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*(	ABC/D
+	(+*(+	ABC/D
E	(+*(+	ABC/DE
)	(+*	ABC/DE+
-	(-	ABC/DE+*+
F	(-	ABC/DE+*+F
)		ABC/DE+*+F-

Convert Infix to Postfix Notation

(A*B+C/(D-E))

Infix Character Scanned	STACK	Postfix Expression
(	(	
A	(	A
*	(*	A
B	(*	AB
+	(+	AB*
C	(+	AB*C
/	(+/	AB*C
(	(+/(	AB*C
D	(+/(	AB*CD
-	(+/(-	AB*CD
E	(+/(-	AB*CDE
)	(+/	AB*CDE-
)		AB*CDE-/+

Convert Infix to Prefix Notation



Step-1: Reverse the infix string. Note that while reversing the string you must interchange left and right parenthesis.

Step-2 : Obtain the corresponding postfix expression of the infix expression obtained as a result of Step1.

Step-3: Reverse the postfix expression to get the prefix expression.

(A*B+C/(D-E))   ((E-D)/C+B*A)

Infix Character Scanned	STACK	Postfix Expression
(	(	
(	((	
E	((	E
-	((-	E
D	((-	ED
)	(	ED-
/	(/	ED-
C	(/	ED-C
+	(+	ED-C/
B	(+	ED-C/B
*	(+*	ED-C/B
A	(+*	ED-C/BA
)		ED-C/BA*+

+*AB/C-DE

Queue

- ❑ Like stacks, **queues are also one of the important data structures which store their elements in an ordered manner**.
- ❑ Examples :
  - People moving in an escalator.
  - People waiting for bus.
  - People standing outside the ticketing window of a cinema hall
  - Luggage kept on conveyor belt.
  - Cars lined for filling petrol.
  - Cars lined at a toll bridge.
- ❑ Same is the case with data structure.

Queue

- A queue is a **FIFO (First In First Out)** data structure in which **each element that was inserted first is the first one to be taken out**.

- The elements in a queue are **added at one end called the rear** and **removed from the other end called the front**.

- Queues can be implemented using either **arrays or linked list**.

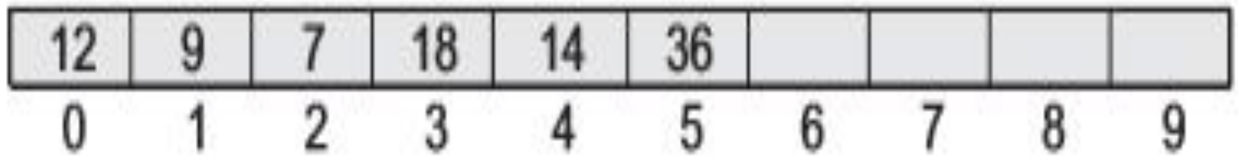
Types of Queues

- ❑ There are four types of Queue:

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

Simple Queue

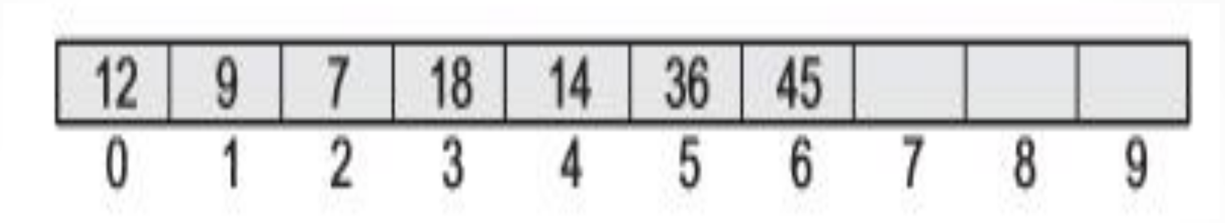
- Queues can easily be represented using linear arrays.
- Every queue will have front and rear variables that will point to the position from where deletions and insertions can be performed respectively.



- Front : 0, rear: 5

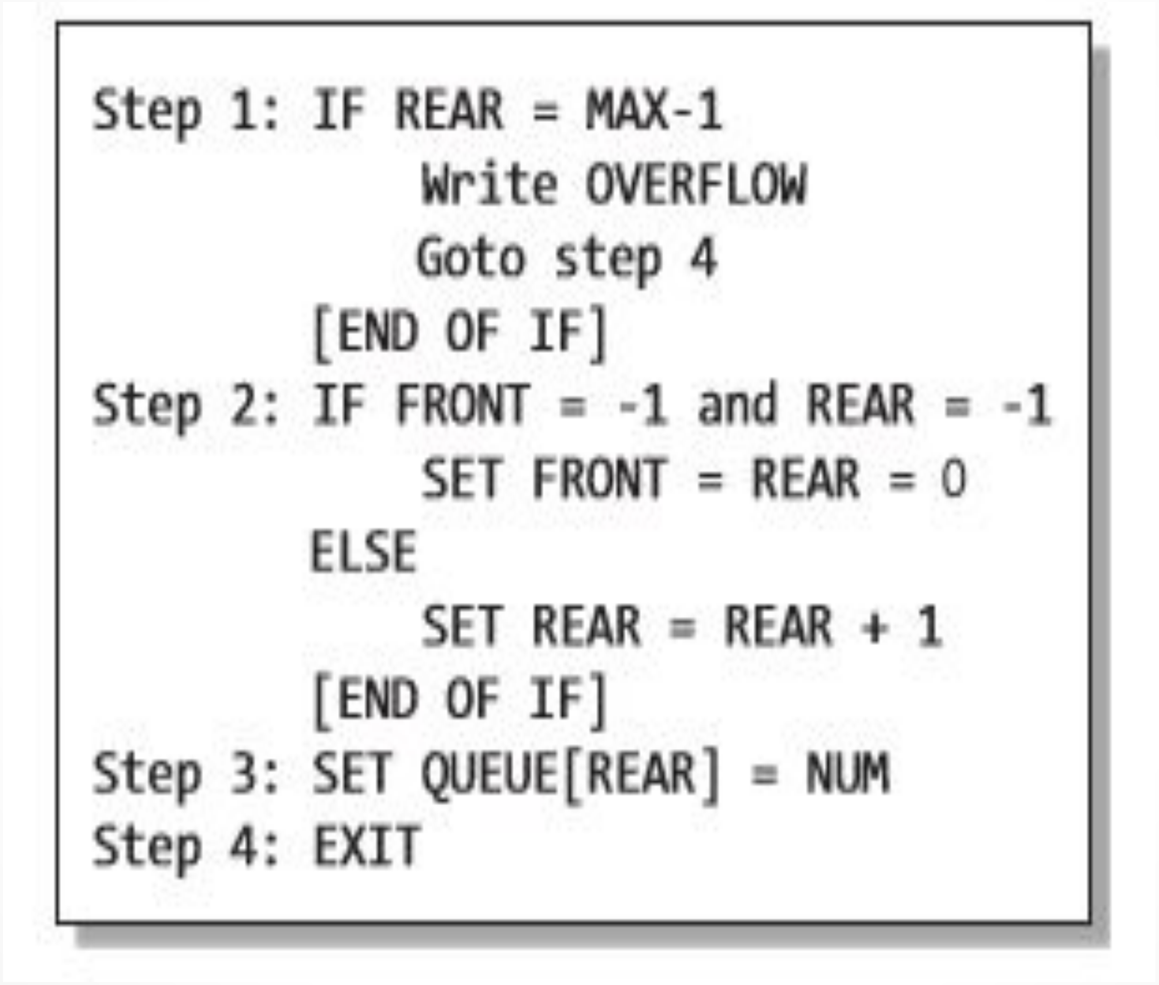
Operations on a simple Queue: Insertion

- Now if new value is needed to be added, say 45: then rear would be incremented by 1.



- Front : 0, rear: 6
- Every time a new element has to be added, the same procedure is repeated.
- An **overflow** will occur when we try to insert an element into **a queue that is already full.**
- When **rear=MAX-1,** where MAX is the size of a queue overflow occurs.
- Note that MAX-1 has been written as array index starts from 0.

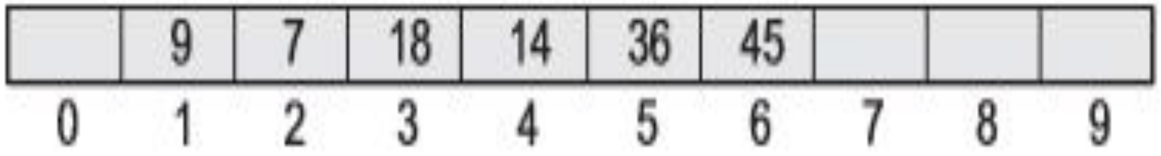
An algorithm to insert an element in a simple queue



```
Step 1: IF REAR = MAX-1
              Write OVERFLOW
              Goto step 4
          [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
              SET FRONT = REAR = 0
          ELSE
              SET REAR = REAR + 1
          [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

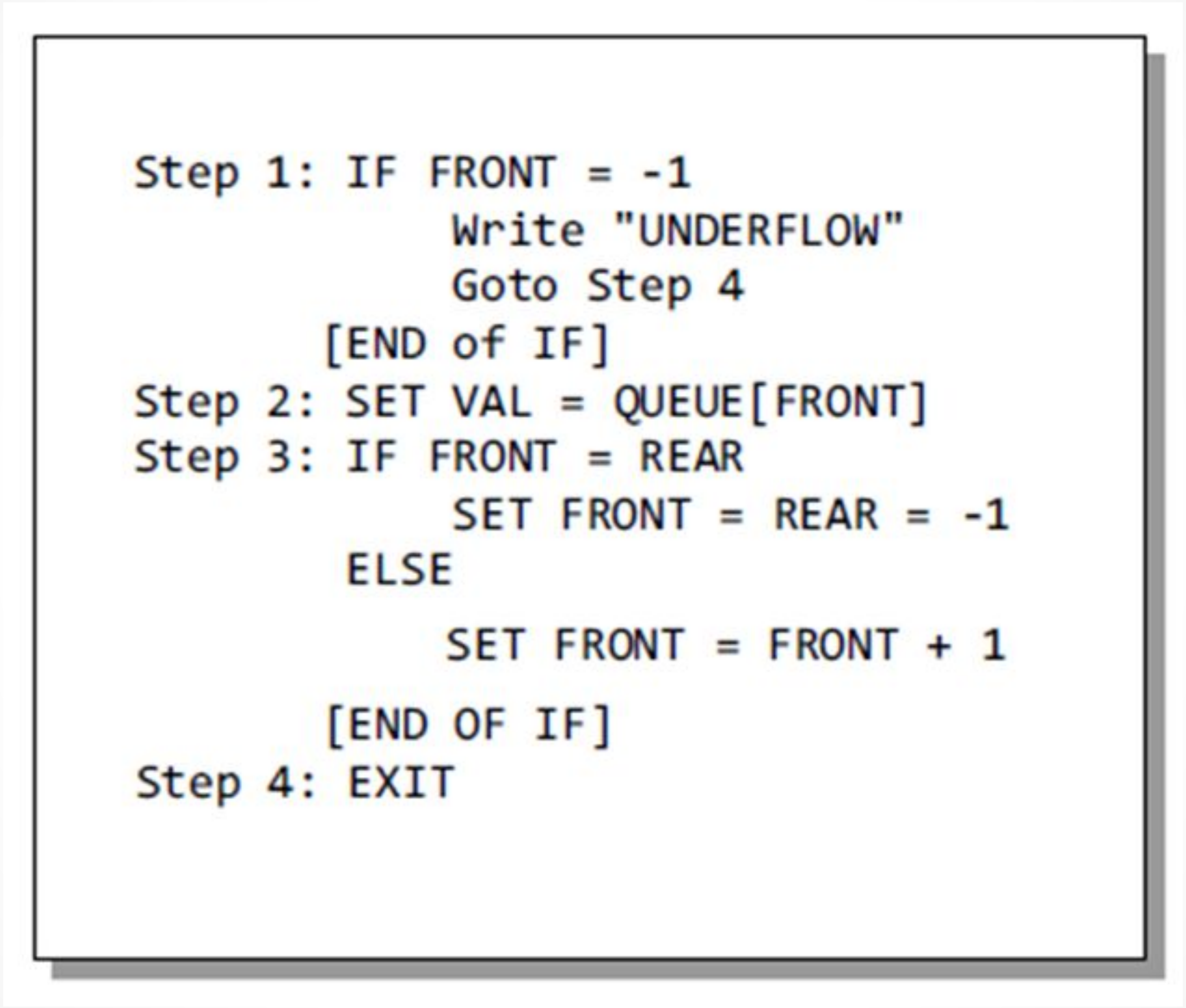
Operations on a simple Queue: Deletion

- Now if we want to delete an element from the queue, then the value of front will be incremented.
- Deletions are done from this one end of the queue only.



- Front : 1, rear: 6
- An **underflow** condition occurs when we try to delete an element from **a queue that is already empty**.
- If **front=-1 and rear=-1**, this means there is no element in the queue.

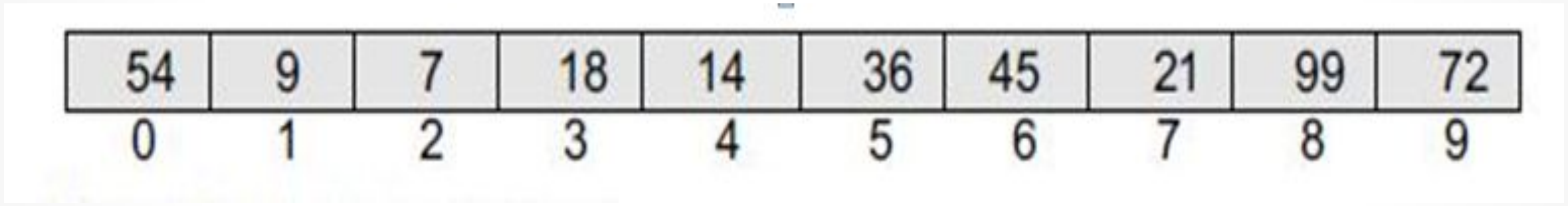
An algorithm to delete an element from a simple queue



```
Step 1: IF FRONT = -1
              Write "UNDERFLOW"
              Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
              SET FRONT = REAR = -1
        ELSE
              SET FRONT = FRONT + 1
        [END OF IF]
Step 4: EXIT
```

Circular Queue

- ❑ In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.
- ❑ Look at the queue shown in Figure :

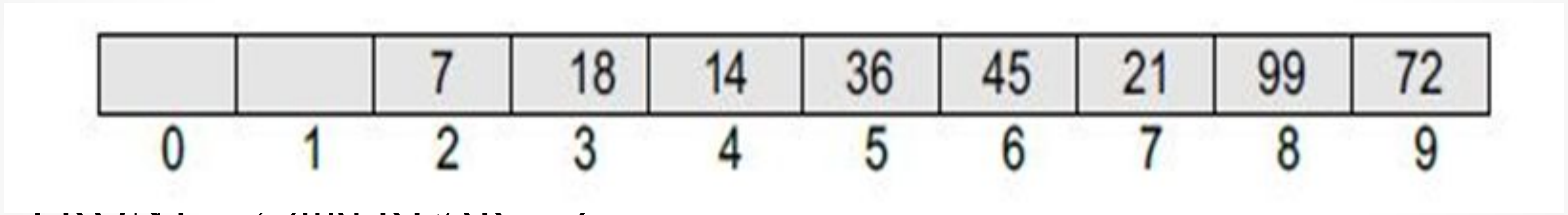


54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

- ❑ FRONT = 0 and REAR = 9.
- ❑ Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted

Circular Queue

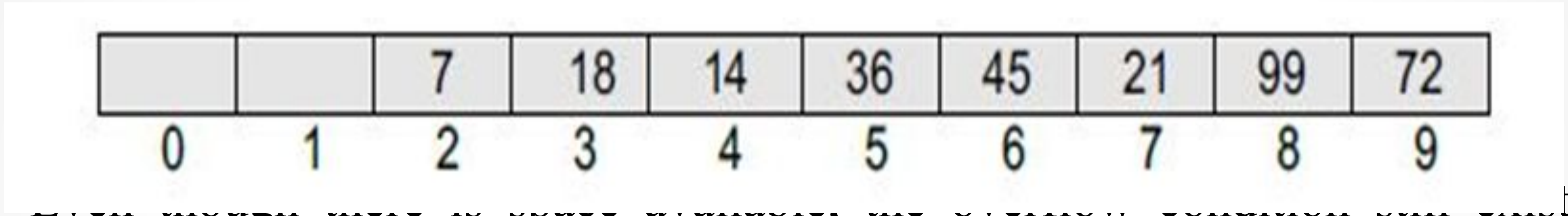
- Consider a scenario in which two successive deletions are made.
- The queue will then be given as shown in Figure.



- FRONT = 2 and REAR = 9.

Circular Queue

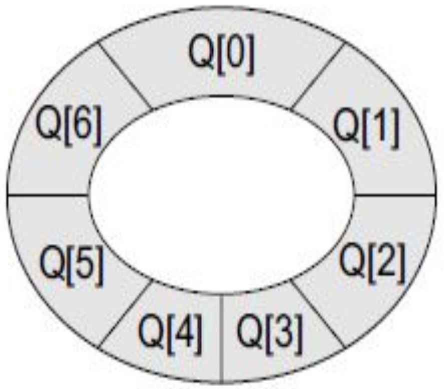
- Suppose we want to insert a new element in the queue shown in Fig.



- Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true.
- This is a major drawback of a linear queue.

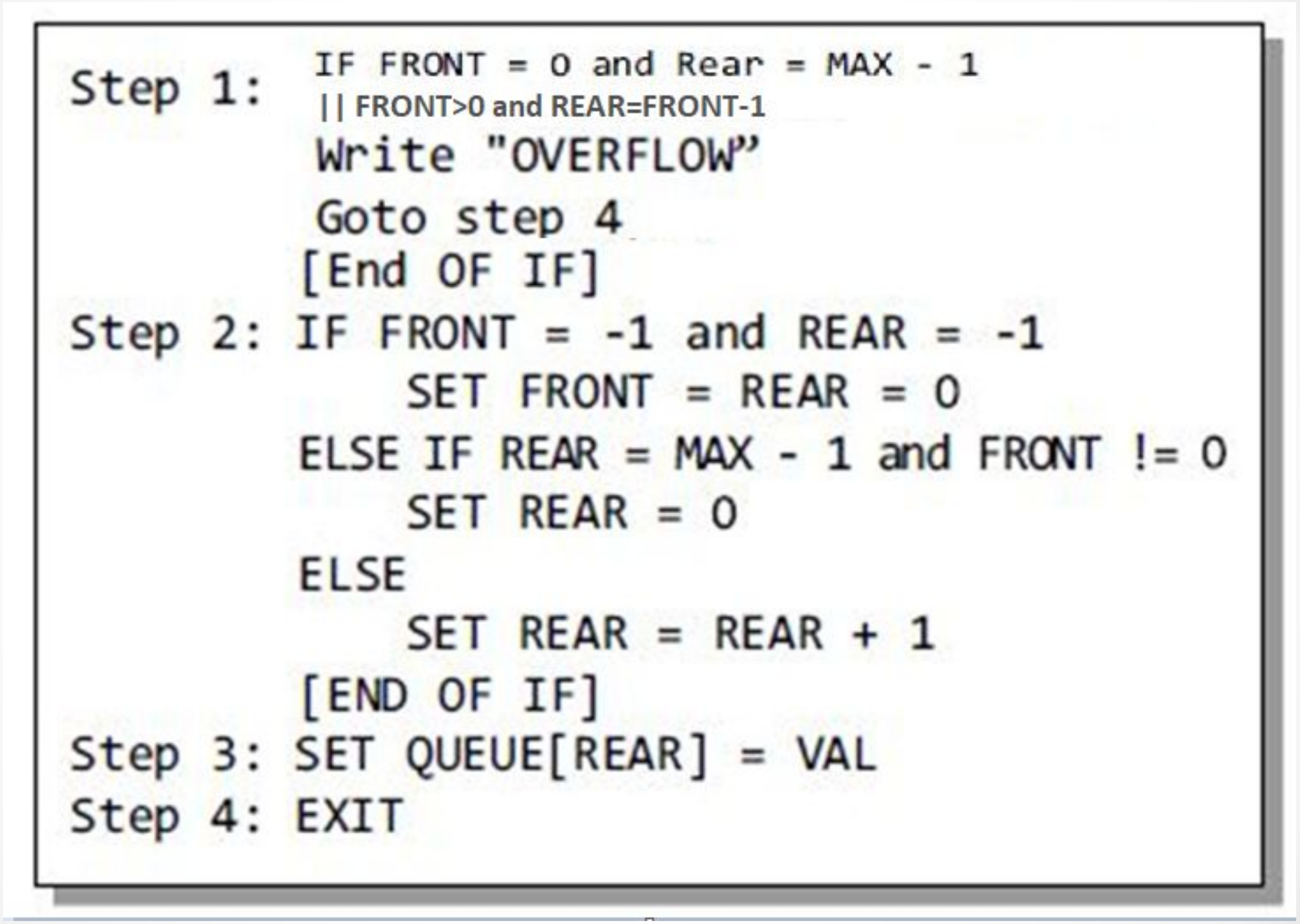
Circular Queue

- In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in Fig.



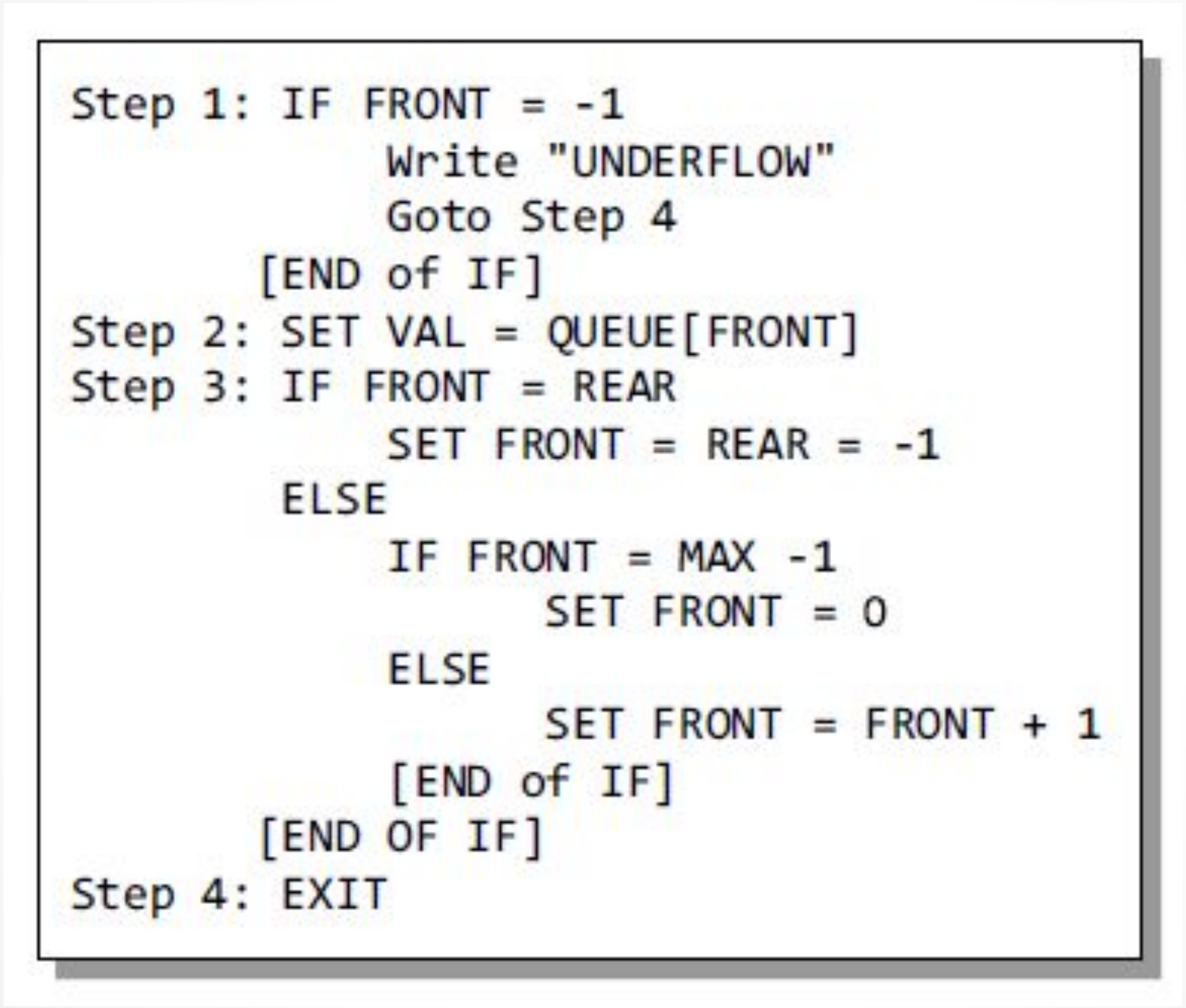
- The circular queue w nt = 0 and rear = Max – 1.
- A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations.

An algorithm to insert an element in a circular queue



```
Step 1: IF FRONT = 0 and Rear = MAX - 1
        || FRONT>0 and REAR=FRONT-1
         Write "OVERFLOW"
         Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
             SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
             SET REAR = 0
        ELSE
             SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

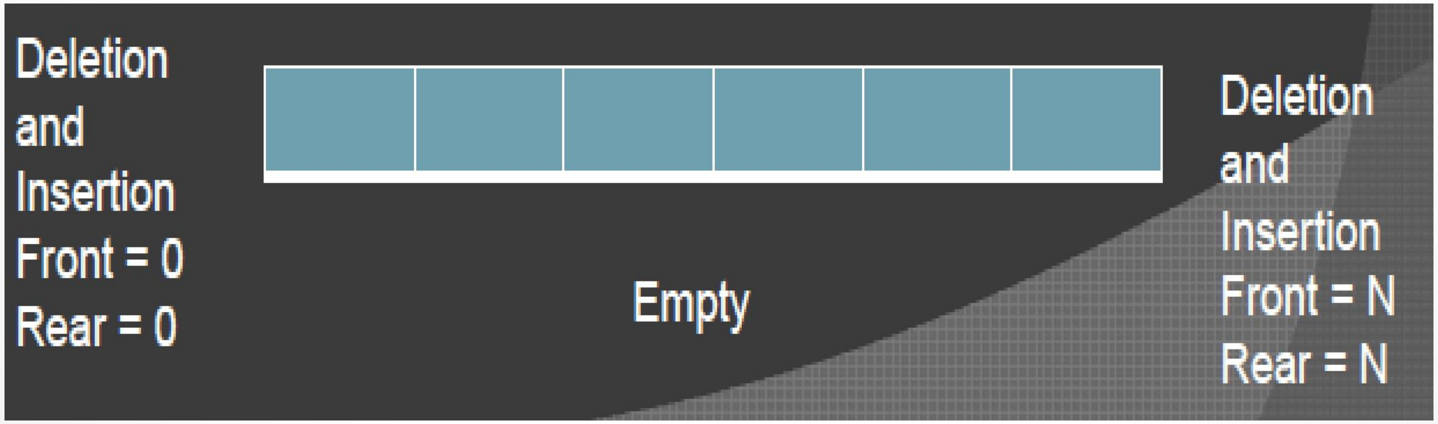
An algorithm to delete an element from a circular queue



```
Step 1: IF FRONT = -1
              Write "UNDERFLOW"
              Goto Step 4
          [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
              SET FRONT = REAR = -1
         ELSE
              IF FRONT = MAX -1
                    SET FRONT = 0
              ELSE
                    SET FRONT = FRONT + 1
              [END of IF]
          [END OF IF]
Step 4: EXIT
```

Double Ended Queue

- Double – ended queue is an abstract data type similar to an ordinary queue, except that it permits to insert and delete from both sides.
- Double – ended queue is also known as deque which is pronounced as ‘deck’.



- It supports both stack – like and queue – like capabilities.
- Deque is useful where the data to be stored has to be ordered, compact storage is needed, and the retrieval of data elements has to be faster.
- There are two other forms of deque :
  - Input restricted deque [Insertion at one end]
  - Output restricted deque [Deletion at one end]

Priority Queue

- A queue in which it is possible to insert or remove an element at any position depending on some priority is called priority queue.
- A priority queue is a collection of elements where the elements are stored according their priority levels.
- The order in which the elements should be removed is decided by the priority of the element.
- The following rules are applied to maintain a priority queue.
    - The element with a higher priority is processed before any element of lower priority.
    - If there were elements with the same priority, then the element added first in the queue would get processed first.

- Priority queues are used for implementing job scheduling by the operating system where jobs with higher priority are to be processed first.

- Two ways to implement a priority queue
  - Sorted list (priority wise)
    - Advantage : deletion is easy; elements are stored by priority, so just delete from the beginning of the list.
    - Disadvantage : insertion is hard; it is necessary to find the proper location for insertion.
  - Unsorted list
    - Advantage : insertion is easy; just add elements at the end of the list.
    - Disadvantage : deletion is hard; it is necessary to find the highest priority element first.

Applications of Queue

- When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a line printer are placed on a queue.
- Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters ae queues, because service is first – come first – served.
- Another e.g. concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first – come first – served basis, so the data structure is a queue.

- Calls to large companies are generally placed on a queue when all operators are busy.
- In large universities, where resources are limited, students must sign a waiting list if all terminals are occupied. The student who has been at a terminal the longest is forced off first, and the student who has been waiting the longest is the next user to be allowed on.
- A whole branch of mathematics, known as queueing theory, deals with computing, probabilistically, how long users expect to wait on a line, how long the line gets, and other such questions. The answer depends on how frequently users arrive to the line and how long it takes to process a user once the user is served.
- The answer depends on how frequently users arrive to the line and how long it takes to process a user once the user is served.

Simulation

- Any process or situation that we wish to simulate is considered as a system.
- A System may be defined as a group of objects interacting to produce some result.
- For e.g., an industry is a group of people and machines working together to produce some product.
- A powerful tool that can be used to study behavior of systems is simulation.
- **Simulation is the process of forming an abstract model of a real world scenario to understand the effect of modifications and the introduction of various strategies on the situation.**
- It allows the user to experiment with real and proposed situations without actually observing its occurrence.
- The major advantage of simulation is that it permits experimentation without modifying the real solution.
- A model of the system must be produced to simulate a situation.

- Moreover, to determine the structure of a model, the entities, attributes, and activities of the system should be determined.
- **Entities represent the object of interest in the simulation.**
- **Attributes denote the characteristics of these entities.**
- **An activity is the process that causes a change of the system state.**
- **An event is an occurrence of an activity at a particular instance of time.**
- **The state of the system at any given time is specified by the attributes of the entities and the relation between entities at that time.**
- E.g. consider the situation in which you are waiting in line for a service at a bank. In general, the more clerks there are, the faster the line moves. The bank manager wants to keep his customers happy by reducing their waiting time but at the same time he does not want to employ any more service clerks than he has to. Being able to simulate the effect of adding more clerks during peak business hours allows the manager to plan more effectively.

Figure 1: **An unbuffered bank model without a waiting line.**

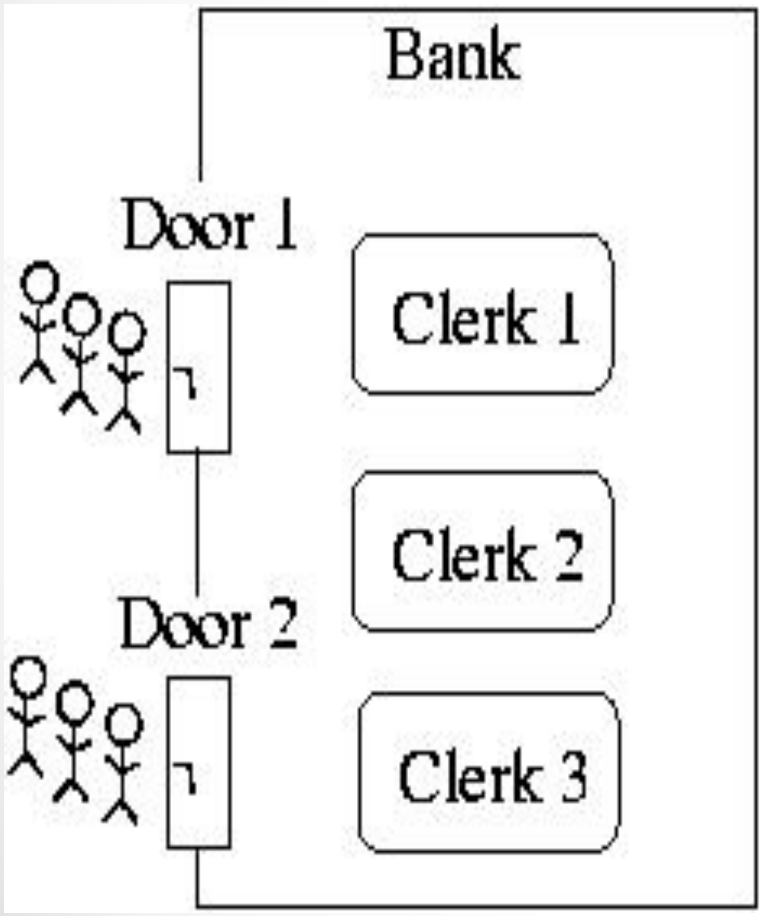
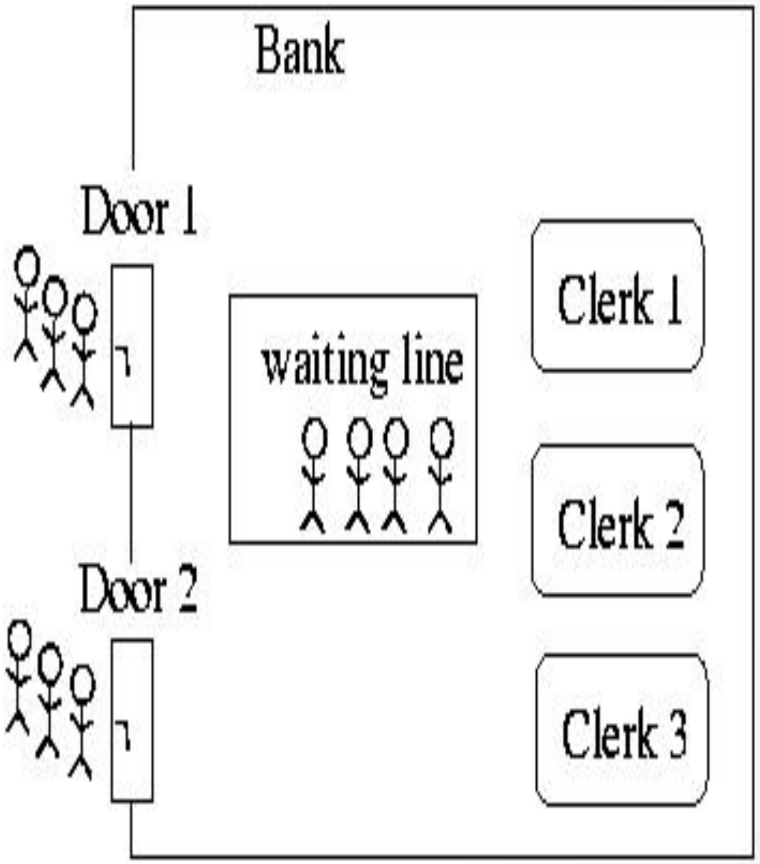


Figure 2: **A buffered bank model with a waiting line.**



Stack Vs. Queue

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one variable is used. It points to the top of the stack.	In queues, two different variables are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.

Thank You...!!!!