Poornaprajna Institute of Management, Udupi

Subject: Advanced Data Structure And Algorithms

Subject Code: MCAH104

Name of the Faculty:
Priya K
Asst. Professor
Dept. of Computer Applications
PIM, Udupi

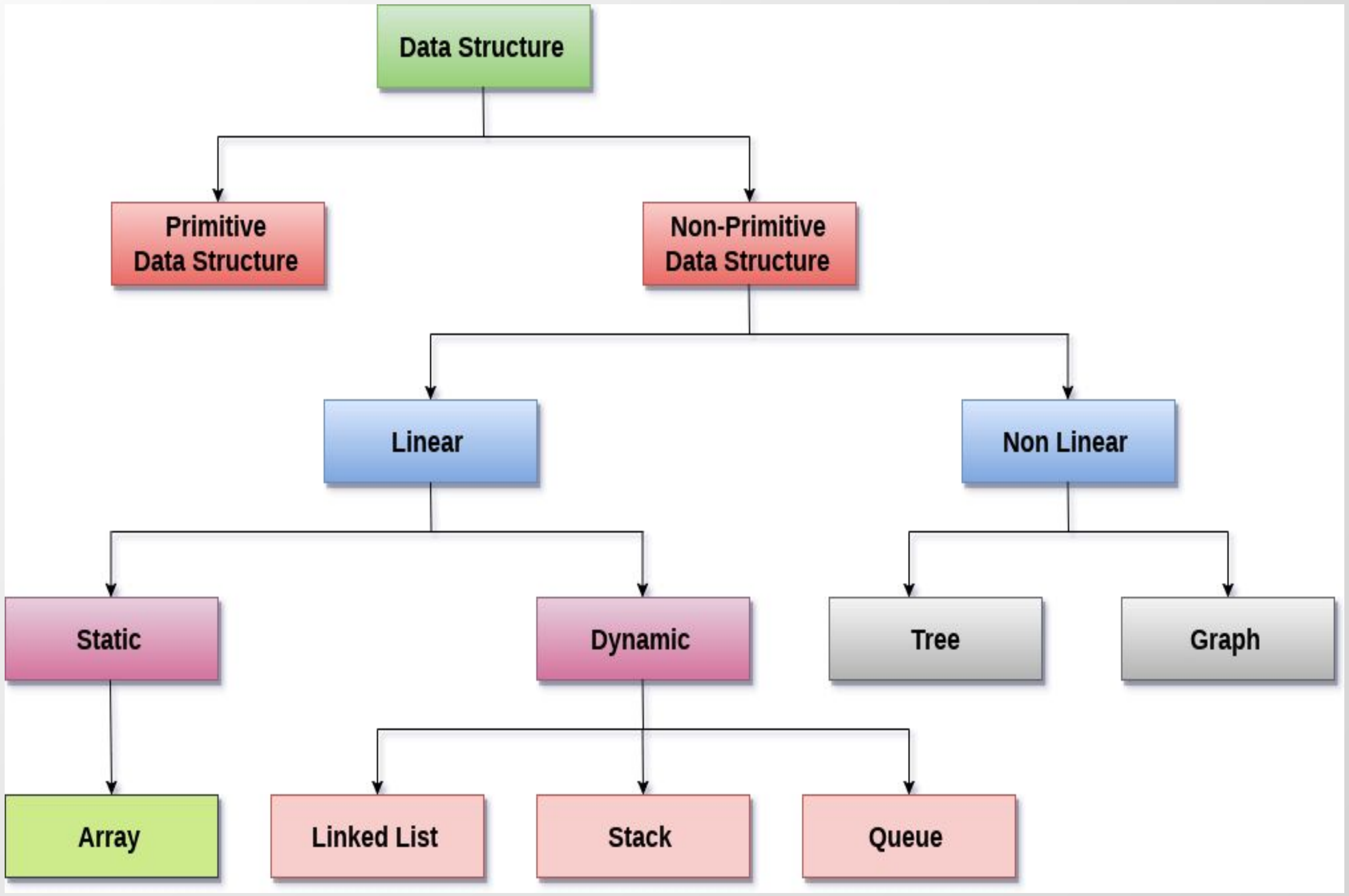
Review of Basic Data Structures:

Introduction:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer.

Examples of Data Structures are arrays, Linked List, Stack, Queue, etc

Types of Data Structure:



Primitive Data structure:

Integer: This is used to represent a number without decimal point Example: 12,90.

Float and Double: This is used to represent a number with decimal point Example: 12.5 , 90.3.

Character: This is used to represent a single character. Example:'C' 'a

Non-Primitive Data Structure:

Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

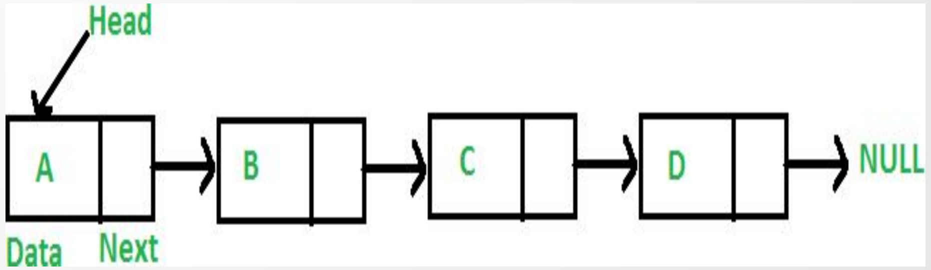
Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

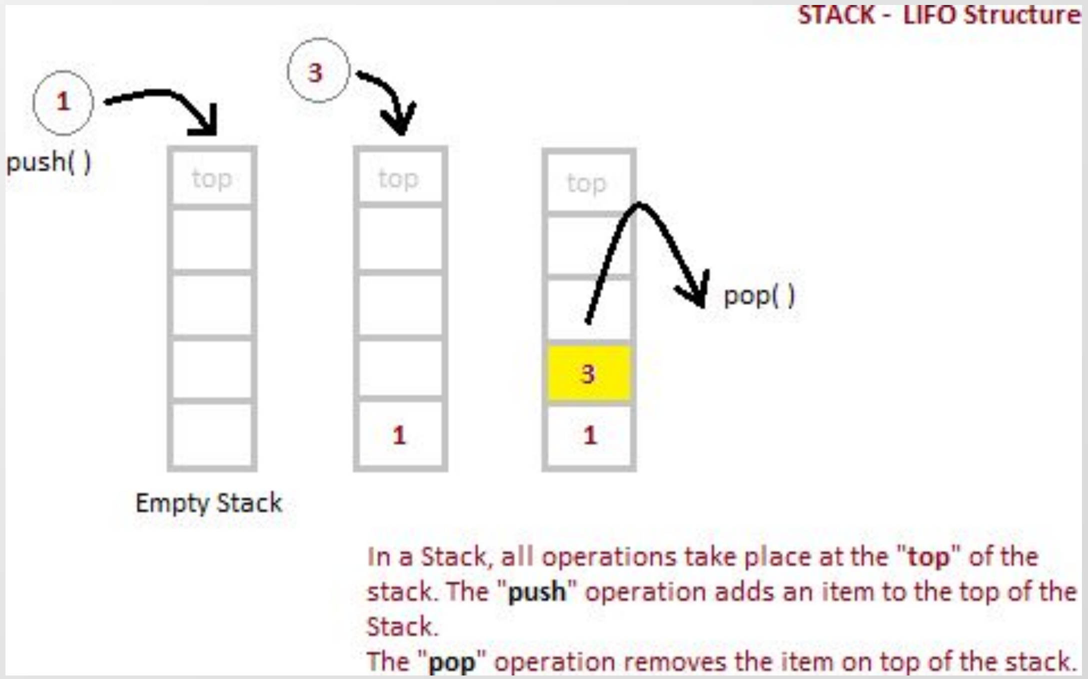
age[0], age[1], age[2], age[3]…..

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.



Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

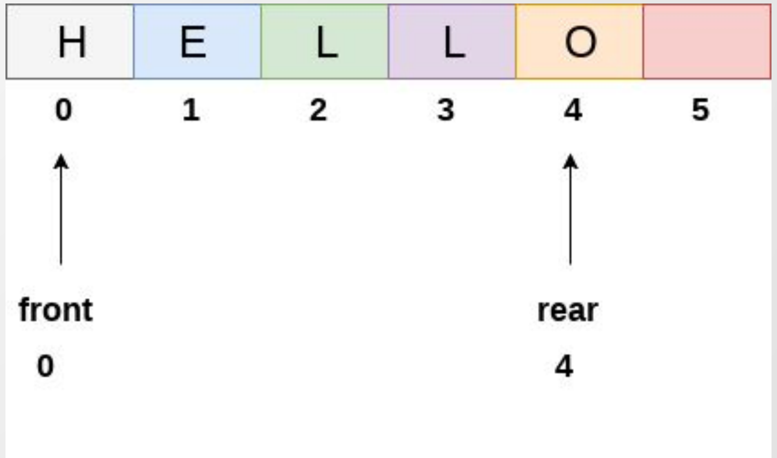
.



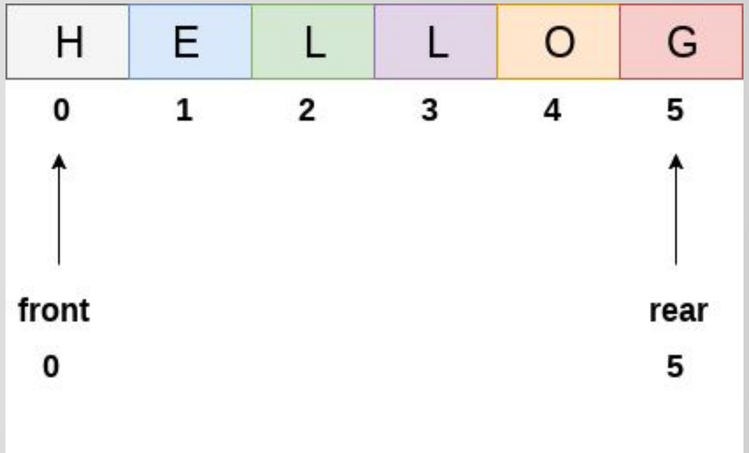
In a Stack, all operations take place at the "**top**" of the stack. The "**push**" operation adds an item to the top of the Stack.
The "**pop**" operation removes the item on top of the stack.

Queue: Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

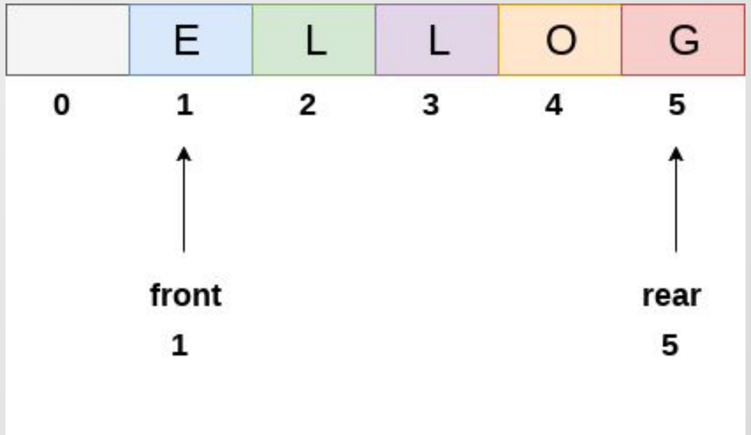
it follows First-In- First-Out (FIFO) methodology for storing the data items.



Queue



Queue after inserting an element



Queue after deleting an element

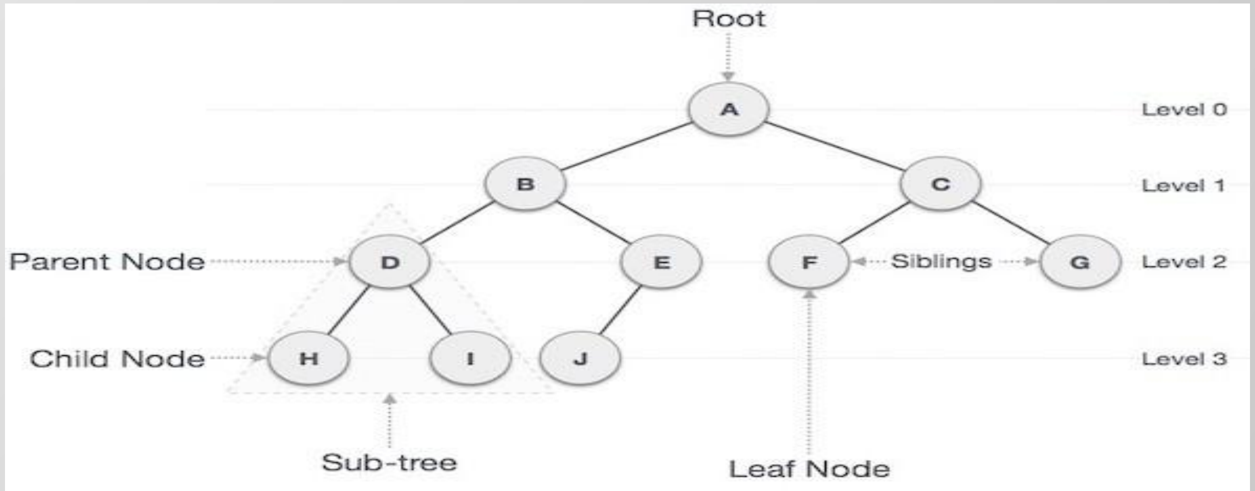
Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are:

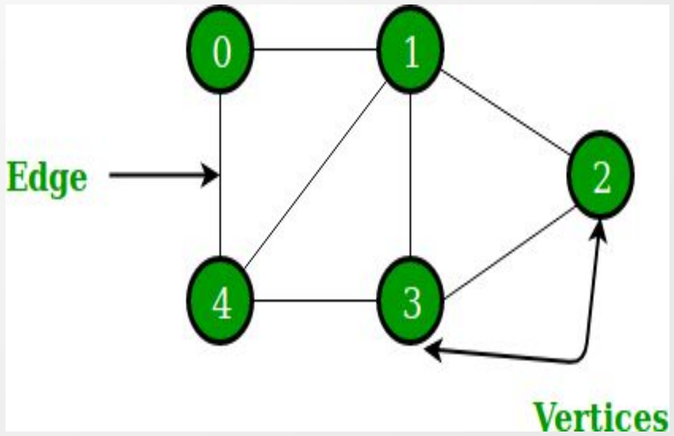
Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes.

The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node.



Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.



Need of Data Structures:

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Array

An array is a sequential collection of elements of same data type and stores data elements in a continuous memory location. The elements of an array are accessed by using an index.

How to declare an array:

datatype array_name [array_size] ;

For example, take an array of integers 'n'. int n[6];

n[ ] is used to denote an array 'n'. It means that 'n' is an array. So, int n[6] means that 'n' is an array of 6 integers.

Here, 6 is the size of the array i.e. there are 6 elements in the array ‘n’.

int n[ ] = {2, 3, 15, 8, 48, 13};

In this case, we are declaring and assigning values to the array at the same time. Here, there is no need to specify the array size because compiler gets it from

{ 2,3,15,8,48,13 }.

Index of an Array:

int n[ ] = {2, 3, 15, 8, 48, 13};

Element	2	3	15	8	48	13
index	1	2	3	4	5	6

Every element of an array has its index. We access any element of an array using its index.

Pictorial view of the above mentioned array is:

0, 1, 2, 3, 4 and 5 are indices. It is like they are identity of 6 different elements of an array.

Index always starts from 0. So, the first element of an array has a index of 0.

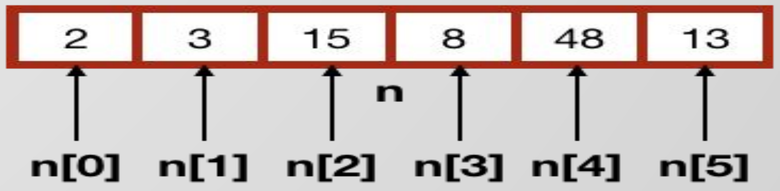
We access any element of an array using its index and the syntax to do so is:

array_name[index]

For example, if the name of an array is 'n', then to access the first element (which is at 0 index), we write n[0].

Here n[0] is 2

n[1] is 3 etc



Assigning Values to Array:

To initialize it, assign a value to each of the elements of the array. n[0] = 2;

n[1] = 4;

n[2] = 8;

It is just like we are declaring some variables and then assigning values to them. int x,y,z;

x=2; y=4; z=8;

Thus, the first way of assigning values to the elements of an array is by doing so at the time of its declaration.

int n[ ]={ 2,4,8 };

And the second method is declaring the array first and then assigning values to its elements. int n[3];

n[0] = 2;

n[1] = 4;

n[2] = 8;

Two Dimensional Arrays:

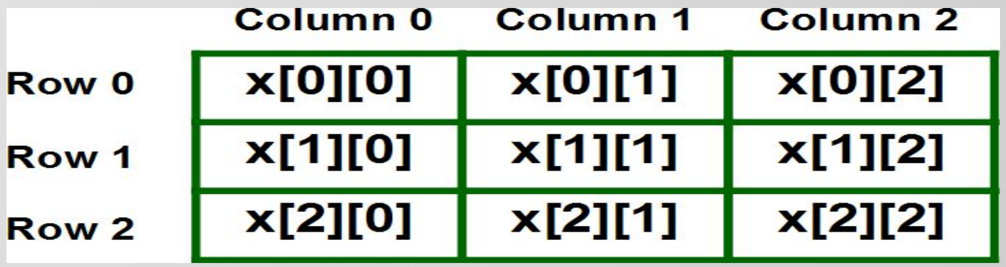
2-dimensional arrays also exist and are generally known as matrix. These consist of rows and columns. A two – dimensional array can be seen as a table with ‘x’ rows and ‘y’ columns.
A two – dimensional array with 3 rows and 3 columns is shown below:

Ex: int x[3][3];

Assigning Values to a 2 D Array:

Same as in one-dimensional array, we can assign values to the elements of a 2-dimensional array in 2 ways as well.
In the first method, just assign a value to the elements of the array. If no value is assigned to any element, then its value is assumed to be zero.
Suppose we declared a 2-dimensional array a[2][2]. Now, we need to assign values to its elements. int a[2][2];

```
a[0][0]=1;
a[0][1]=2;
a[1][0]=3;
a[1][1]=4;
```



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

The second way is to declare and assign values at the same time as we did in one-dimensional array.

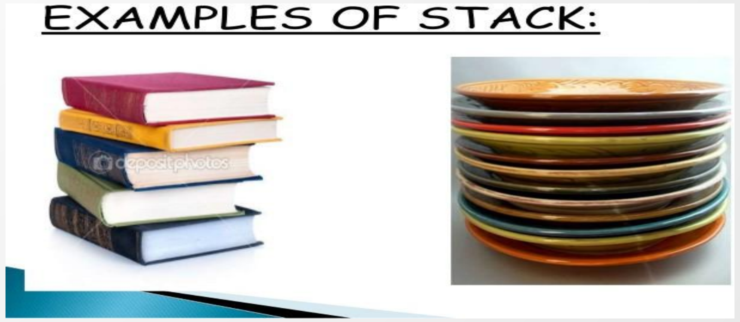
```
 int a[2][3] = { 1, 2, 3, 4, 5, 6 };
```

Here, value of a[0][0] is 1, a[0][1] is 2, a[0][2] is 3, a[1][0] is 4, a[1][1] is 5 and a[1][2] is 6. We can also write the above code as:

```
int a[2][3] = {
                    {1, 2, 3},
                    {4, 5, 6 }
                  };
```

STACK

A stack is a useful data structure in programming. It is just like a plates kept on top of each other.



Put a new plate on top Remove the top plate

If you want the plate at the bottom, you have to first remove all the plates on top. Such an arrangement is called Last In First Out - the last item that was placed is the first item to go out.

stack is a linear structure in which items may be added or removed only at one end.

Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack.

Stack is a linear data structure.

Elements are arranged in sequential order.

Basic features of Stack:

Stack is a LIFO(Last in First out) structure.

push() function is used to insert new elements into the Stack.

pop() function is used to remove the element from the stack.

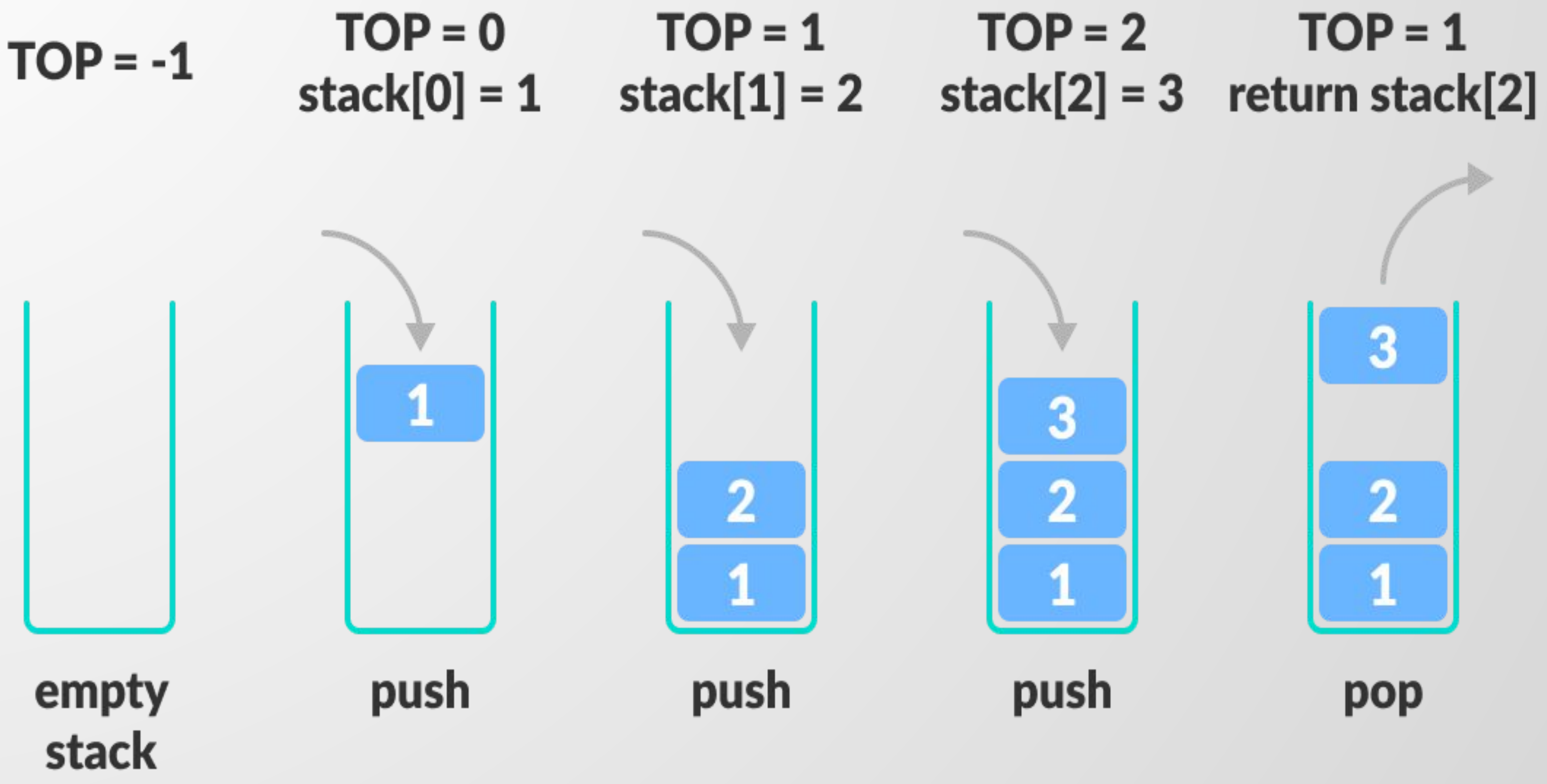
isEmpty(): This operation indicates whether the stack is empty or not.

isFull(): This operation indicates whether the stack is full or not.

peek() :*the peek* operation retrieves the element at the top of the stack without removing it from the stack.

Both insertion and removal are allowed at only one end of Stack called Top.

Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.


TOP = -1
TOP = 0
stack[0] = 1
TOP = 1
stack[1] = 2
TOP = 2
stack[2] = 3
TOP = 1
return stack[2]
1
2
1
3
2
1
3
2
1
empty stack
push
push
push
pop

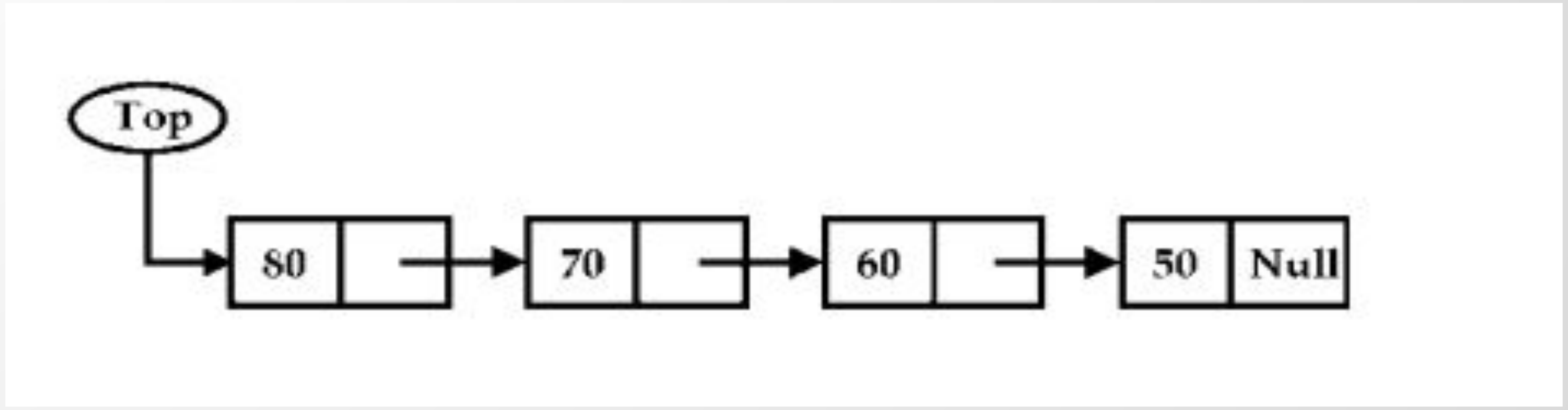
Algorithm:

```
PUSH:
PUSH(STACK,TOP,MAXSTK,ITEM)
This procedure pushes an ITEM onto a stack 1.[stack already filled?]
If TOP=MAXSTK-1 then print overflow and return
Set TOP=TOP+1[Increase TOP by 1]
Set STACK[TOP]=ITEM [Insert ITEM in new TOP position]
Return

POP:
POP(STACK,TOP,ITEM)
This procedure deletes the top element of stack & assign it to the variable ITEM
[Stack has an item to be removed?]
If TOP=-1 then print UNDERFLOW and return
set ITEM=STACK[TOP] [assign TOP element to ITEM]
set TOP=TOP-1 [Decrement TOP by 1]
4.Return
```

Linked List Representation of Stacks:

PUSH:



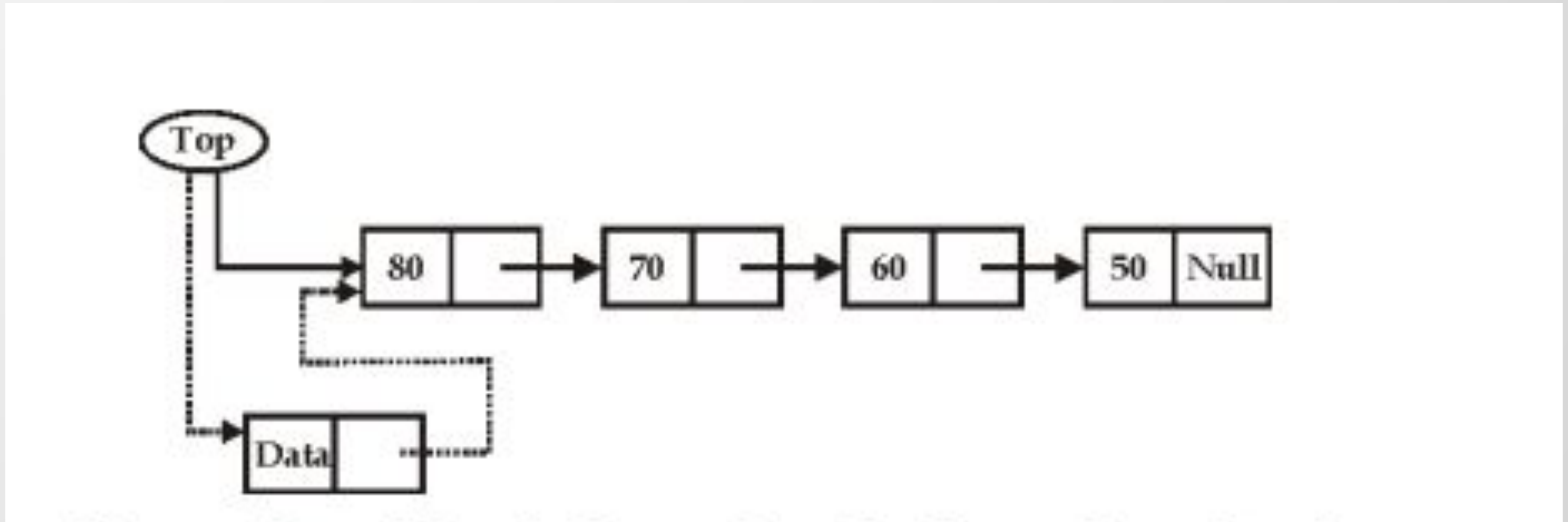
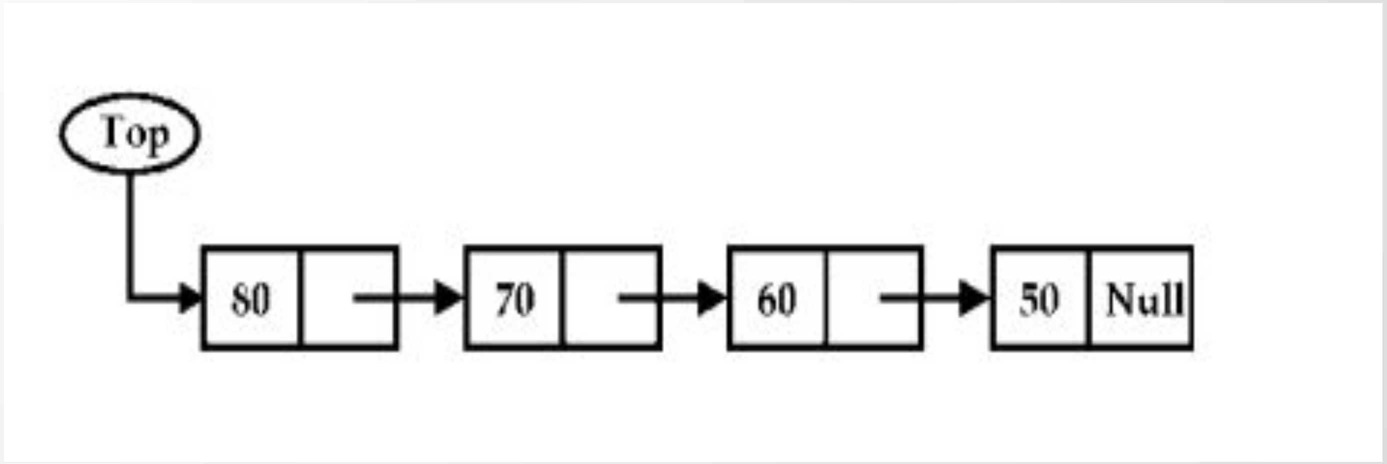


Illustration of Push Operation Performed on Stack

```
PUSH_LINKSTACK(INFO,LINK,TOP,AVAIL,ITEM)
This procedure pushes an ITEM into a linked stack
1.[available space?] if AVAIL=NULL, then write UNDERFLOW and exit
2. [remove first node from AVAIL list
    Set NEW=AVAIL and AVAIL=LINK[AVAIL]
3. Set INFO[NEW]=ITEM  [copies ITEM into new node]
4. Set LINK[NEW]=TOP [new node points to the original top node in the stack]
5. Set TOP=NEW [reset TOP to point to the new node at the top of the stack]
6. exit.
```

Pop:



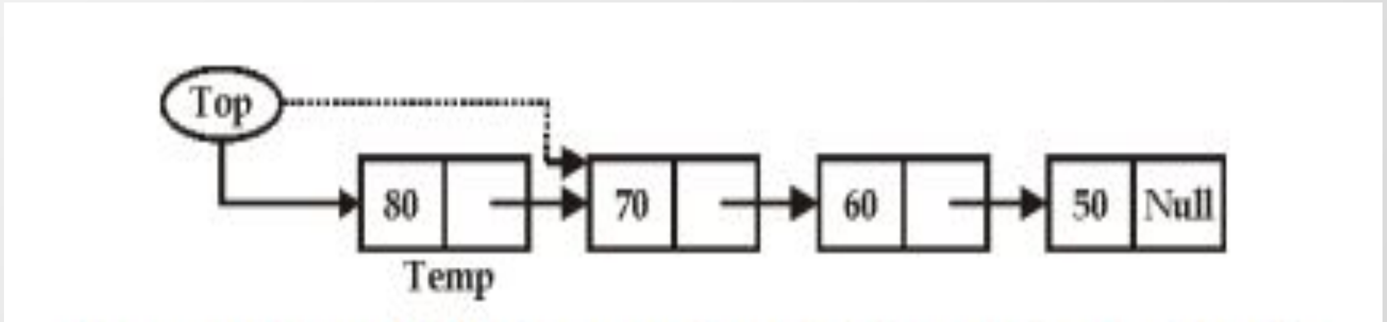


Illustration of Pop Operation Performed on a Stack

```
POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)
This procedure deletes the top element of a linked stack and assigns it to the variable ITEM
1.[stack has an item to be removed?]
   If TOP=NULL then write: UNDERFLOW and exit
2. Set ITEM=INFO[TOP] [copies the top element of stack into ITEM]
3. Set TEMP=TOP and TOP=LINK[TOP]
  [ old value of the TOP pointer in TEMP and reset TOP to point to the next element in the
stack]
4. [Return deleted node to the AVAIL node]
     set LINK[TEMP]=AVAIL and AVAIL=TEMP
5.exit.
```

Applications of Stack:

Expression Evaluation

Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

i. Infix to Postfix
ii. Infix to Prefix
iii. Postfix to Infix
iv. Prefix to Infix

Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis
Checking Stack is used to check the proper opening and closing of parenthesis.
String Reversal
Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.
Function Call
Stack is used to keep information about the active functions or subroutines.
Undo/Redo operations: The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
Browser history: Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.

Advantages of Stack:

Easy implementation: Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.

Efficient memory utilization: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.

Fast access time: Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.

Helps in function calls: Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.

Supports backtracking: Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.

Used in Compiler Design: Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.

Enables undo/redo operations: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

Disadvantages of Stack:

Limited capacity: Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.

No random access: Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.

Memory management: Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.

Not suitable for certain applications: Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.

Stack overflow and underflow: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.

Recursive function calls limitations: While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program.

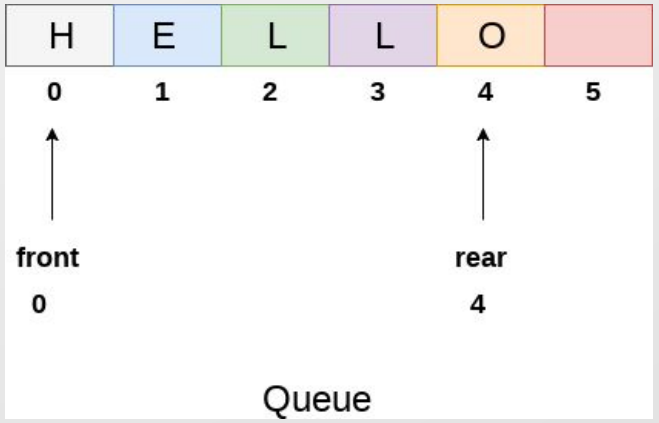
QUEUE:

Queue is a linear list of elements in which insertion can take place at one end called REAR end, and deletion can take place at other end called FRONT end.

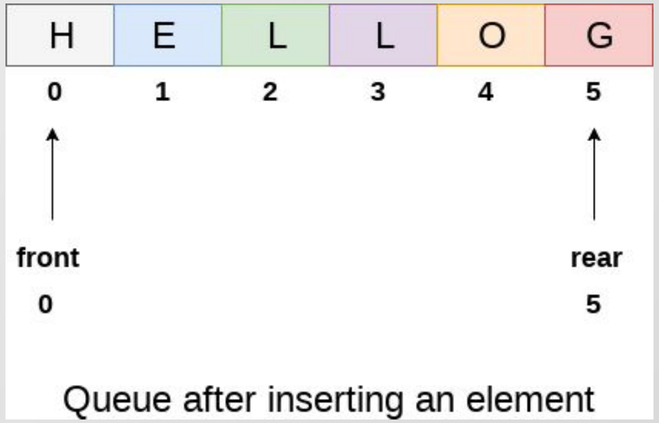
The term FRONT and REAR are used in describing a linear list only when it is implemented as queue.

Queue are also called First-In-First out (FIFO) lists, since the first element in a queue will be the first element out of the queue.

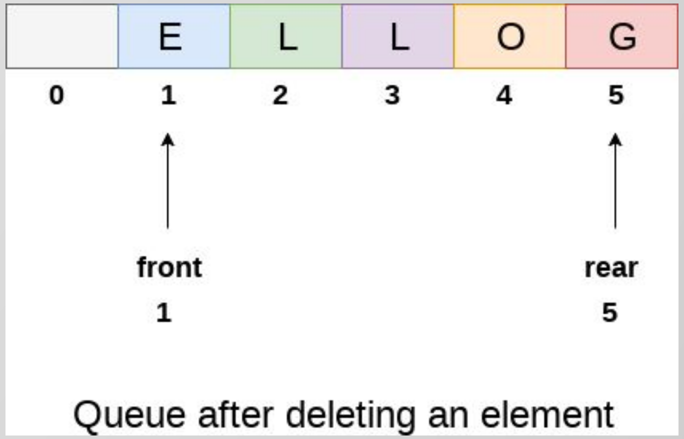
In other words, the order in which elements enter a queue will be the first element out of the queue.



Queue



Queue after inserting an element



Queue after deleting an element

Algorithm:

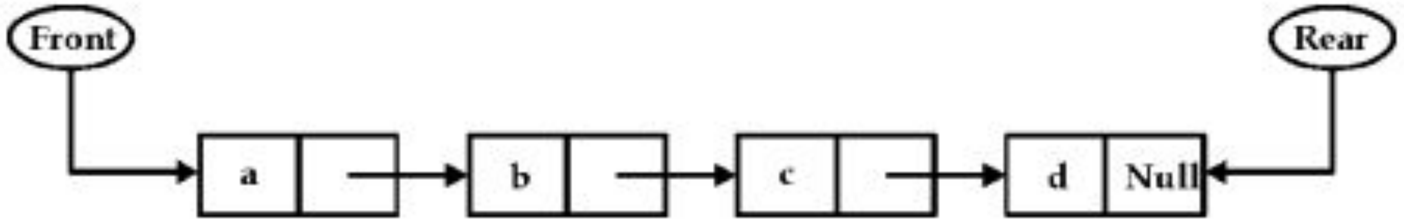
```
QINSERT(QUEUE,N,FRONT,REAR,ITEM)
This procedure inserts an element ITEM into a queue.
1.[Queue already filled?]
If FRONT=1 and REAR=N , then
Write: OVERFLOW, and return
2. [Find new value of REAR]
If FRONT=NULL, then[Queue initially empty]
Set FRONT=1 and REAR=1
Else if REAR=N ,then
Set REAR=1
Else:
Set REAR=REAR+1
[End of If structure]
3.set QUEUE[REAR]=ITEM[This inserts new element]
4.Return
```

```
QDELETE[QUEUE,N,FRONT,REAR,ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.
1.[Queue already empty?]
If FRONT=NULL, then write: UNDERFLOW , and return
2. set ITEM=QUEUE[FRONT]
3.[find new value of FRONT]
If FRONT=REAR then [Queue has only one element to start]
Set FRONT=NULL and REAR=NULL
Else if FRONT=N then
Set FRONT=1 then
Else
Set FRONT=FRONT+1
[end of If structure]
4.return.
```

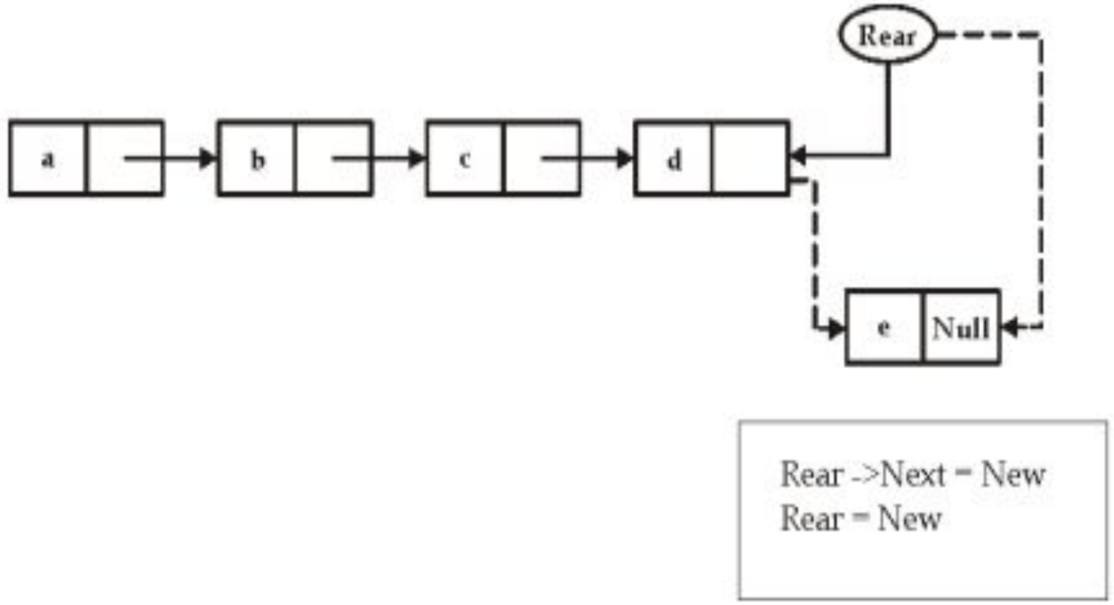
- **Operations on Queue:**
- Mainly the following four basic operations are performed on queue:
- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **isEmpty():** This operation indicates whether the queue is empty or not.
- **isFull():** This operation indicates whether the queue is full or not.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Linked List Representation of Queue:

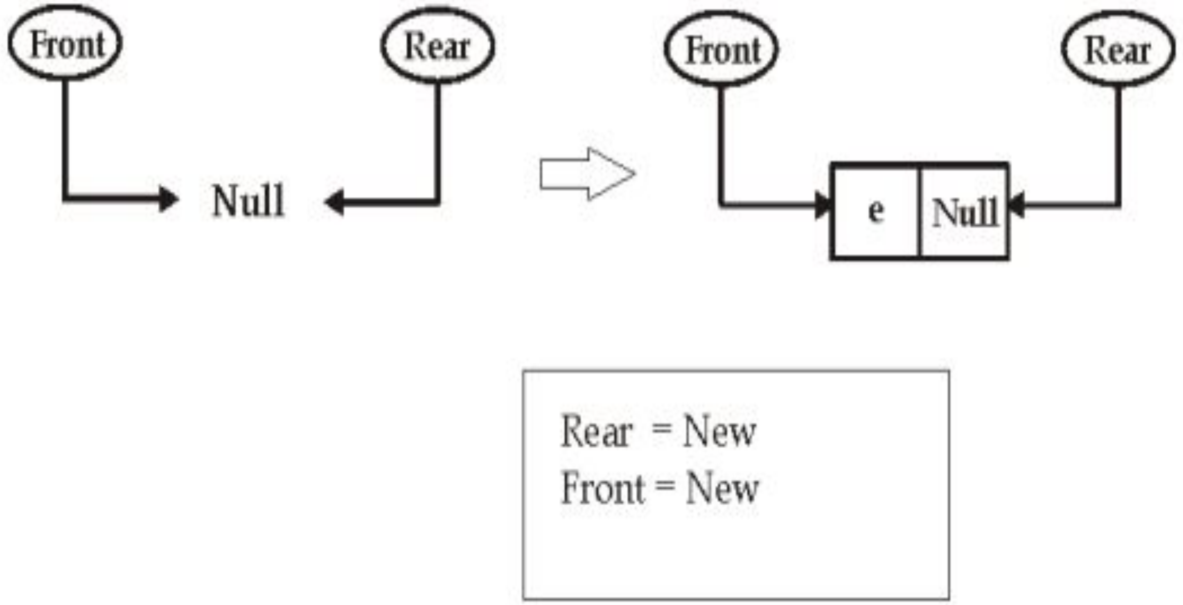
Insert:



A Queue Maintained using a Linked List



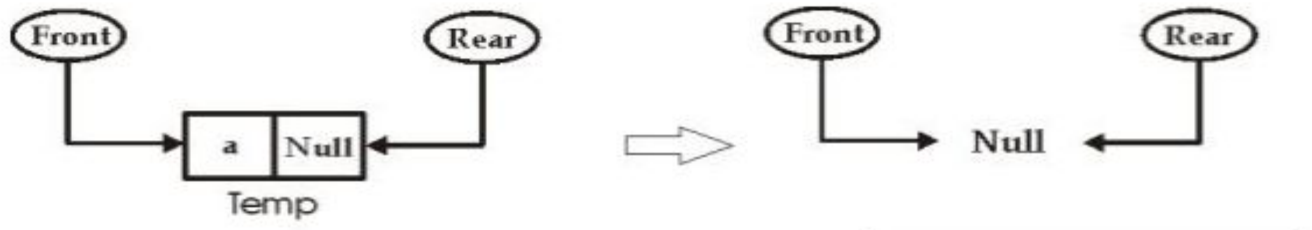
This insertion of a New element '*e*' in the queue



This insertion of a New element '*e*' in the empty queue

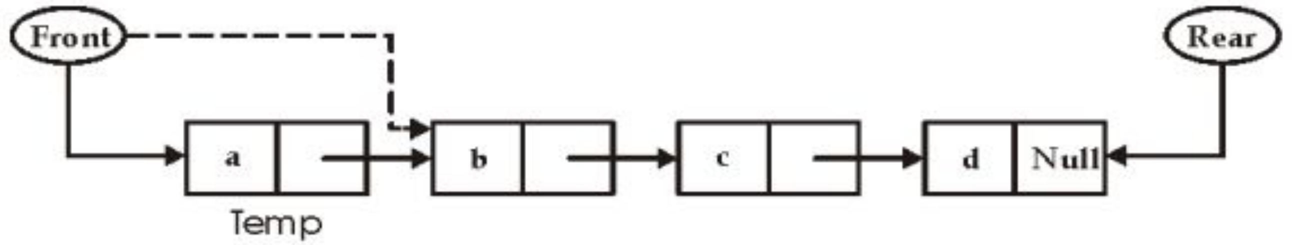
```
LINKQ_INSERT(INFO,LINK,FRONT,REAR,AVAIL,ITEM)
This procedure inserts an ITEM into a linked queue
1. [available space?] if AVAIL=NULL , then write UNDERFLOW and exit
2. [Remove first node from AVAIL list]
Set NEW=AVAIL and AVAIL=LINK[AVAIL]
3. Set INFO[NEW]=ITEM and LINK[NEW]=NULL
   [copies ITEM into new node]
4. if(FRONT=NULL ) then FRONT=REAR=NEW
   [if Queue is empty then ITEM is the first element in the queue]
else set LINK[REAR]=NEW and REAR=NEW
  [REAR points to the new node appended to the end of the list]
5.Exit.
```

Delete:



Temp = Front
Front = Null
Rear = Null
Deallocate Temp

Deletion of an element from the Queue having only one node

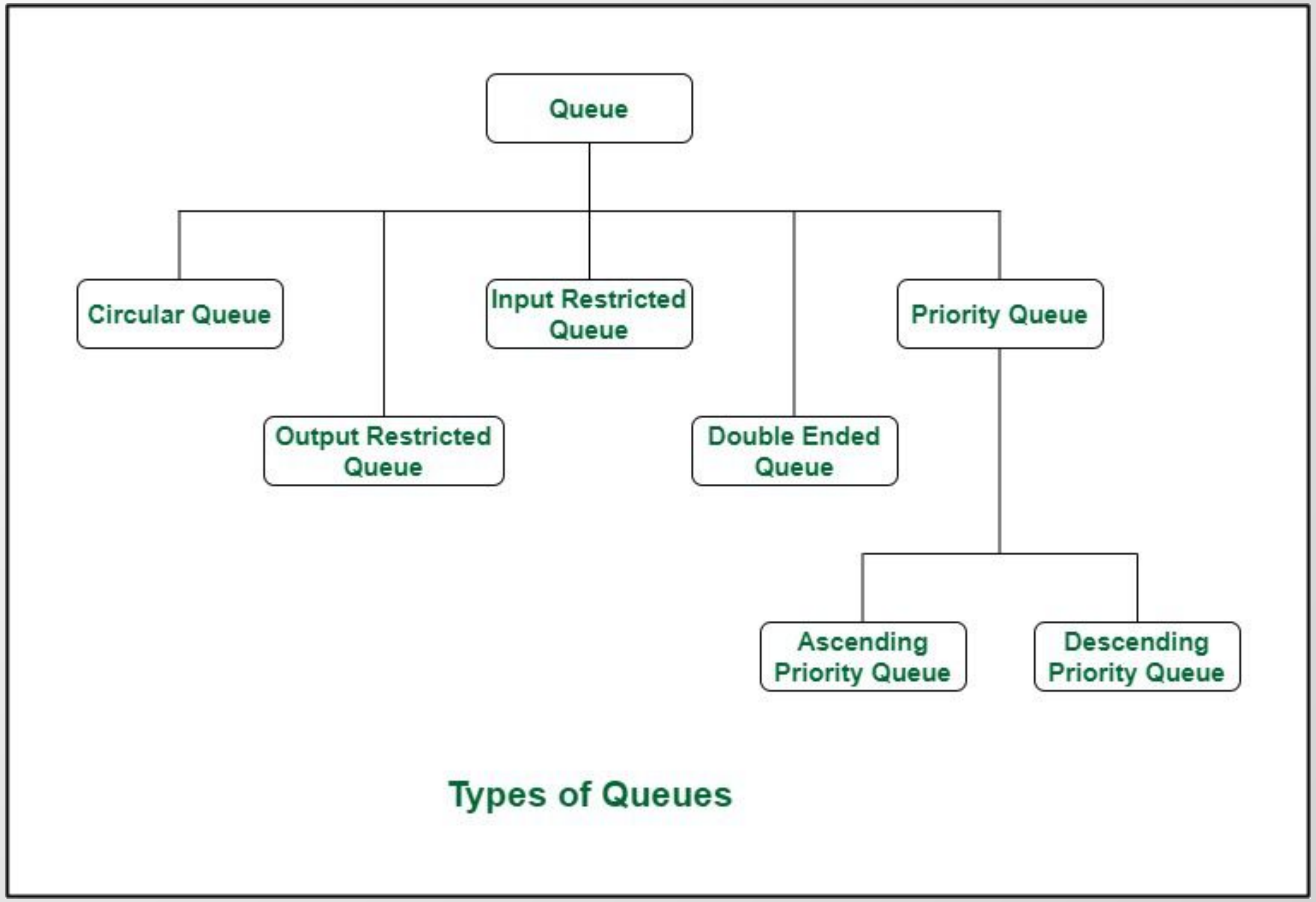


Temp = Front
Front = Front -> Next
Deallocate Temp

Deletion of an element from the Queue

```
LINKQ_DELETE(INFO,LINK,FRONT,REAR,AVAIL,ITEM)
This procedure deletes the front element of the linked queue and stores it in ITEM
1.[linked queue empty?]
If (FRONT=NULL) then write : UNDERFLOW and exit
2. If FRONT==REAR then
 FRONT=REAR=NULL;
3. Else set TEMP=FRONT
    [ if linked queue is nonempty, remember FRONT in a temporary variable TEMP]
4. ITEM=INFO[TEMP]
5. FRONT=LINK[TEMP]
    [ reset FRONT to point to the next element in the queue]
6. LINK[TEMP]=AVAIL and AVAIL=TEMP
    [return the deleted node TEMP to the AVAIL list]
7.Exit
```

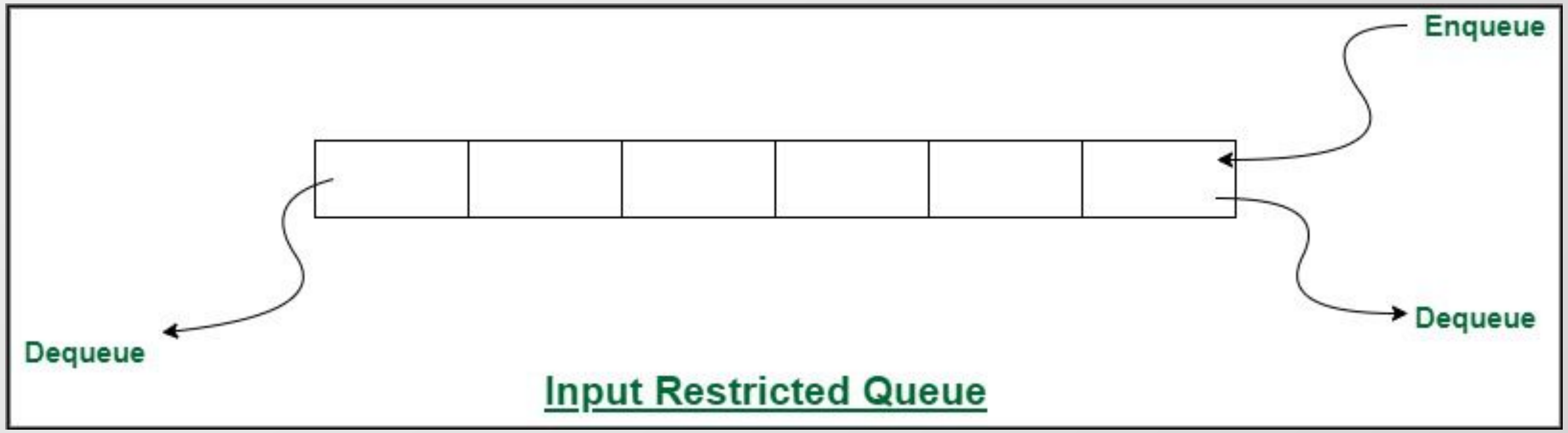
Types of Queues:



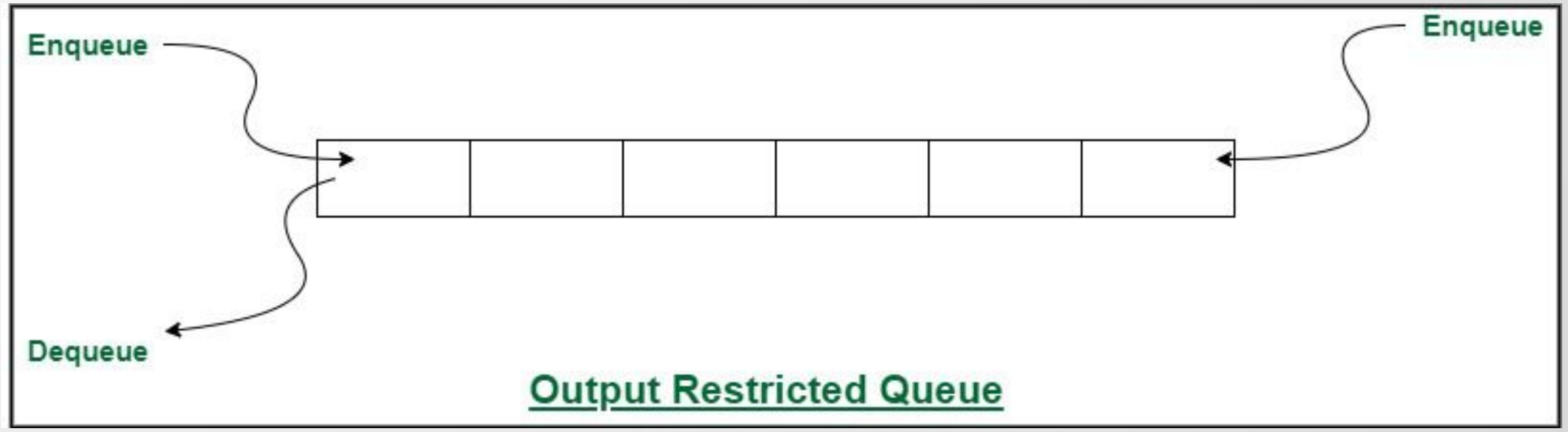
Circular Queue: Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.

Input restricted Queue: In this type of Queue, the input can be taken from one side only(rear) and deletion of elements can be done from both sides(front and rear).

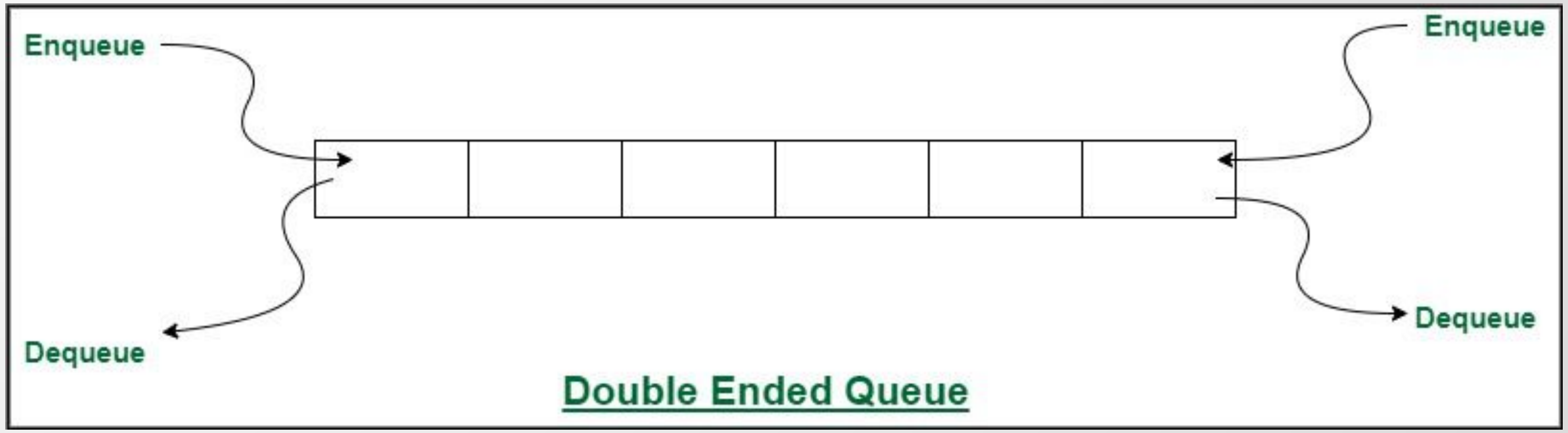
This kind of Queue does not follow FIFO(first in first out). This queue is used in cases where the consumption of the data needs to be in FIFO order but if there is a need to remove the recently inserted data for some reason and one such case can be irrelevant data, performance issue, etc.



Output restricted Queue: In this type of Queue, the input can be taken from both sides(rear and front) and the deletion of the element can be done from only one side(front). This queue is used in the case where the inputs have some priority order to be executed and the input can be placed even in the first place so that it is executed first.



Double_ended_Queue: Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



PRIORITY QUEUE:

A priority queue is a collection of elements such that each element has assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

- an element of higher priority is processed before any element of lower priority.
- Two element with the same priority are processed according to the order in which they were added to the queue.

applications of Queue:

Task Scheduling: Queues can be used to schedule tasks based on priority or the order in which they were received.

Resource Allocation: Queues can be used to manage and allocate resources, such as printers or CPU processing time.

Batch Processing: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

Message Buffering: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

Event Handling: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

Traffic Management: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

- **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
- **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
- **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
- **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
- **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

- When a resource is shared among multiple consumers. Examples include_CPU scheduling, Disk_Scheduling.
- When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO_Buffers, pipes, etc.
- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- CPU Scheduling
- Memory management
- Queues in routers/ switches
- Mail Queues
- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don’t have an internet connection then these messages are queued on the server of WhatsApp.

Circular Queue Operations:

PROCEDURE CQINSERT(X) Given F and R as the pointers to the front and rear element of a circular queue CQ[ ]. The array CQ contains MAX number of elements. X is the element to be inserted at the rear.

```
1.[Overflow?]
    If ((R+1)%MAX=F) Then:
        Write ('Overflow')
        Return
2.[ Increment rear]
    R:=(R+1)%MAX
3.[Insert element]
    CQ[R]:=X
4.[Is the front pointer properly set?]
    If F=-1
    Then F:=0
    Return
```

FUNCTION CQDELETE() Given F and R as the pointers to the front and rear end of a circular queue. The array CQ contains MAX no of elements. This function deletes and returns the last element of the queue. Y is the temporary variable.

```
1.[underflow?]
    If F=-1 Then:
        write ('Underflow')
    Return 0
2.[Delete element]
        Y:=CQ[F]
3.[Queue Empty?]
    If F=R  Then:
        F=R=-1
        Else
        F=(F+1)%MAX
4.  Return Y
```

LINKED LIST:

Introduction:

A linked list is a sequence of data structures, which are connected together via links.

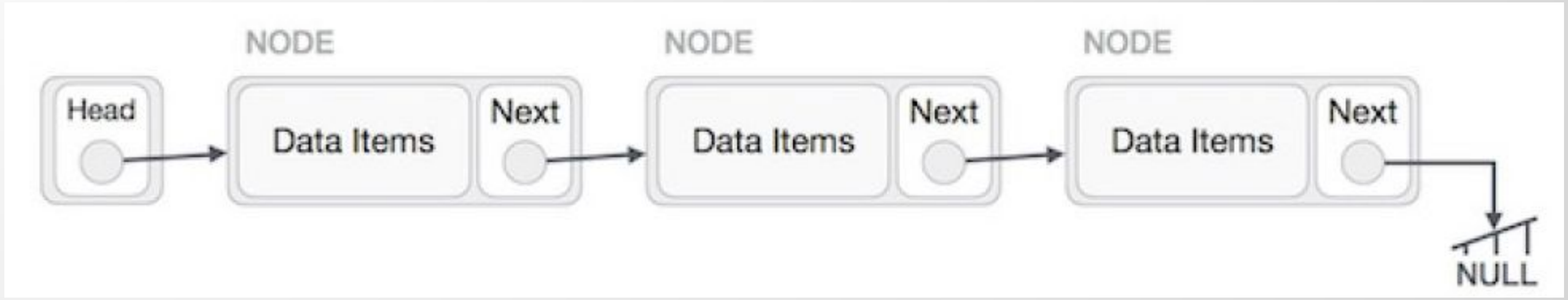
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

Link − Each link of a linked list can store a data called an element.

Next − Each link of a linked list contains a link to the next link called Next.

Linked List Representation:

Linked list can be visualized as a chain of nodes, where every node points to the next node.



- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List:

Following are the various types of linked list.

Simple Linked List − Item navigation is forward only.

Doubly Linked List − Items can be navigated forward and backward.

Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations:

Following are the basic operations supported by a list.

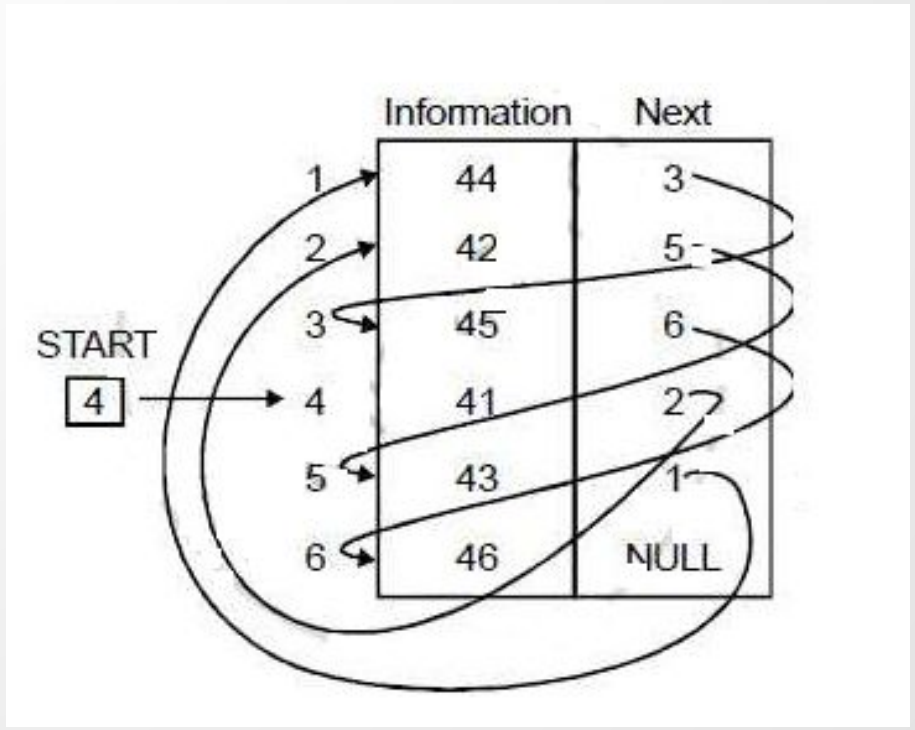
Insertion − Adds an element at the beginning of the list.

Deletion − Deletes an element at the beginning of the list.

Display − Displays the complete list.

Search − Searches an element using the given key.

Representation of LinkedList in memory:



Linked list can be represented in memory by using two arrays respectively known as INFO and LINK, such as INFO[K] and LINK[K] contains information of element and next node address respectively.
The list also requires a variable 'START', which contains address of first node.
Pointer field of last node denoted by NULL which indicates the end of list.

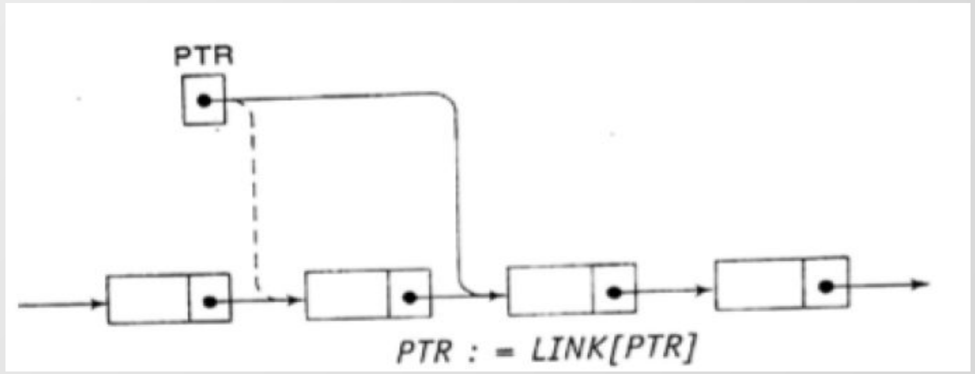
TRAVERSING A LINKED LIST:

Let LIST be a linked list In memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST.

Suppose we want to traverse LIST in order to process each node exactly once.

Traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed.

Accordingly , LINK[PTR] points to the next node to be processed. Thus the assignment PTR=LINK[PTR] moves the pointer to the next node in the list.



Algorithm are as follows:

Initialize PTR or START then process INFO[PTR],the information at the first node. update PTR by assignment PTR=LINK[PTR], so that PTR points to the second node. Then process INFO[PTR], the information at the second node. Again update PTR by the assignment PTR=LINK[PTR],and then process INFO[PTR],the information at the third node. And so on. Continue until PTR=NULL, which signals the end of the list.

Algorithm:

(Traversing a Linked List)Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

```
1. Set PTR=START.[Initializes pointer PTR]
2. Repeat steps 3 and 4 while PTR ≠NULL
3. Apply PROCESS to INFO[PTR]
4. Set PTR=LINK[PTR] [PTR now points to the next node]
   [End of step2 loop]
5. Exit.
```

Searching a Linked List:

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Searching is basically performed on two types of Linked lists:

a). Unsorted LL

b). Sorted LL

LIST is Unsorted:

Suppose the data in LIST are not necessarily sorted. Then one searches for ITEM in LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST.

Before we update the pointer PTR by PTR=LINK[PTR].

We require two tests. first we have to check to see whether we have reached the end of the list. i.e first we check to see whether PTR=NULL If not, then we check to see whether INFO[PTR]=ITEM.

The two tests cannot be performed at the same time, since INFO[PTR] is not defined when PTR=NULL.

Accordingly, we use the first test to control the execution of a loop, and we let the second test take place inside the loop.

Algorithm: SEARCH(INFO,LINK,START,ITEM,LOC)

LIST is a linked list in memory. This algorithm finds the location of LOC of the node where ITEM first appears in LIST , or sets LOC=NULL.

```
•Step 1: Set PTR:=START
•Step 2: Repeat while PTR≠ NULL
            If ITEM = INFO[PTR], then:
            Set LOC := PTR
            Return
            Else:
            Set PTR:= LINK[PTR]
            [End of If structure]
            [End of Step 2 Loop]
•Step 3: [Search is unsuccessful] Set LOC:=NULL
•Step 4: Return
```

Complexity of this algorithm is same as that of linear search algorithm for linear array. Worst case running time is proportional to number n of elements in LIST and the average case running time is proportional to n/2 with condition that ITEM appears once in LIST but with equal probability in any node of LIST.

List is Sorted:

Suppose the data in LIST are sorted. Again we search for ITEM in LIST by traversing the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one of LIST. We can stop once ITEM exceeds INFO[PTR].

•**Algorithm: SRCHSL(INFO,LINK,START,ITEM,LOC)**

LIST is sorted list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC=NULL.

```
•Step 1: Set PTR:= START
•Step 2:Repeat while PTR ≠ NULL
            If ITEM > INFO[PTR], then:
            Set PTR := LINK[PTR]
            Else If ITEM = INFO[PTR], then:
            Set LOC := PTR
            Return
            Else:
            Set LOC:=  NULL
            Return
           [End of If structure]
    [End of step 2 Loop]
•Step 3: Set LOC:= NULL
•Step 4: Return
```

<u>Insertion Algorithm:</u>

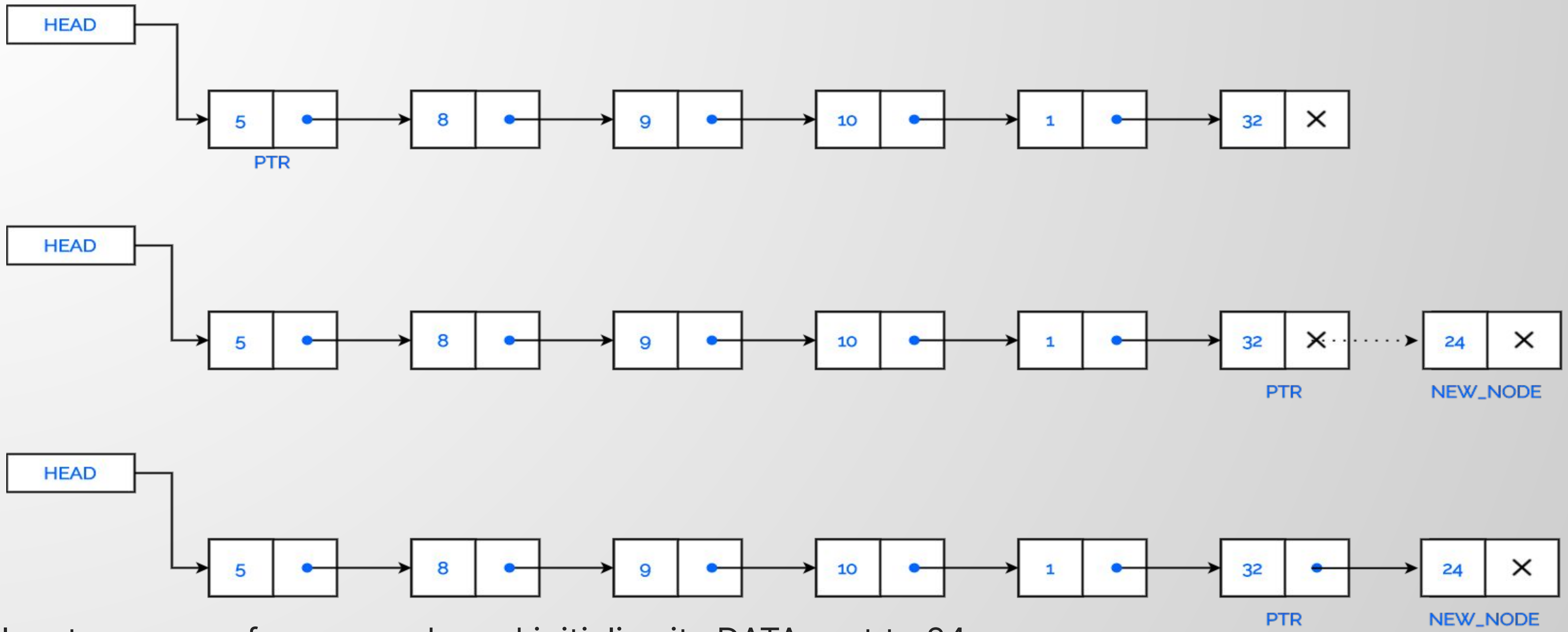
Insertion at a beginning of a List:

INFIRST(INFO,LINK,START,AVAIL,ITEM)

This algorithm inserts ITEM as the first node in the list.

```
Step 1.[OVERFLOW?] If AVAIL=NULL, then Write: UNDERFLOW and exit.
Step 2.[Remove first node from avail list]
Set NEW=AVAIL and AVAIL=LINK[AVAIL]
Step 3. Set INFO[NEW]=ITEM [Copies new data into new node]
Step 4. Set LINK[NEW]=START [New node now points to original first node]
Step 5. Set START=NEW [changes START so it points to the new node]
Step 6.Exit.
```

Insert a Node at the end of a Linked list:



- Allocate memory for new node and initialize its DATA part to 24.
- Traverse to last node.
- Point the NEXT part of the last node to the newly created node.
- Make the value of next part of last node to NULL.

Inserting a node to the end of a linked list:

```
Step 1: IF AVAIL = NULL
Write UNDERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> LINK
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> LINK = NULL
Step 6: SET TEMP=START
Step 7: Repeat Step 8 while TEMP-> LINK != NULL
Step 8: SET TEMP = TEMP -> LINK
[END OF LOOP]
Step 9: SET TEMP -> LINK= NEW_NODE
Step 10: EXIT
```

Inserting a node at a given position of a linked list:

```
Step 1: IF AVAIL = NULL
Write UNDERFLOW
Go to Step 11
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> LINK
Step 4: SET NEW_NODE -> INFO = VAL
Step 5: SET NEW_NODE -> LINK = NULL
Step 6: SET TEMP=START
Step 7: Repeat Step 8 while  TEMP->INFO!=X && TEMP-> LINK != NULL
Step 8: SET TEMP = TEMP -> LINK
[END OF LOOP]
Step 9: SET NEW_NODE -> LINK= TEMP->LINK
Step 10: SET TEMP -> LINK= NEW_NODE
Step 11: EXIT
```

Deletion in singly linked list at the end:

- **Step 1:** IF START = NULL
- Write UNDERFLOW
  Go to Step 8
  [END OF IF]
- **Step 2:** SET PTR = START
- **Step 3:** Repeat Steps 4 and 5 while PTR ->LINK!= NULL
- **Step 4:** SET PREV = PTR
- **Step 5:** SET PTR = PTR -> NEXT
- [END OF LOOP]
- **Step 6:** SET PREV -> LINK = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT

Delete a node from the end of the singly linked list:

```
Step 1:IF HEAD = NULL Write UNDERFLOW
 Go to Step 8 [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR ->LINK!= NULL
Step 4: SET PREV = PTR
Step 5:SET PTR = PTR -> LINK
 [END OF LOOP]
Step 6: SET PREV -> LINK = NULL
Step 7: FREE  PTR
Step 8: EXIT
```

Searching and Sorting:

Insertion Sort:

INSERTION(A,N)

This algorithm sorts the array A with N elements.

```
1. Set A[0]=-∞ [Initializes sentinel element.]
2. 2. Repeat steps 3 to 5 for K=2,3…..N
3. 3. Set TEMP=A[K] AND PTR=K-1;
4.  Repeat while TEMP<A[PTR]
        (a) set A[PTR+1]=A[PTR]  [Moves element forward]
        (b) set PTR=PTR-1
        [End of loop]
5. Set A[PTR+1]=TEMP [Inserts element in proper place]
    [End of step2 loop]
6. Return.
```

Selection Sort:

```
MIN(A, K, N, LOC)
An array A is in memory. This procedure finds the location LOC of the smallest
element among A[K], A[K + 1],….., A[NJ]
1. Set MIN := A[K] and LOC:=K. [Initializes pointers.]
2. Repeat for J =K+1, K+ 2, .., N:
       If MIN > A[J], then: Set MIN := A[J]  and LOC=J
       [End of loop.]
3. Return.

(Selection Sort) SELECTION(A, N)
This algorithm sorts the array A with N elements.
1.Repeat Steps 2 and 3 for K=1, 2, ..., N- 1
2. Call MIN(A, K, N, LOC).
3. [Interchange A[K] and A[LOC],]
      Set TEMP :=A[K], A[K] := A[LOC] and A[LOC] := TEMP.
      [End of Step1 loop.]
4. Exit.
```

Binary Search:

BINARY(DATA, LB, UB, ITEM, LOC) Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC=NULL.

```
1.[Initialize segment variables.]
 Set BEG := LB, END;=UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG<=END and DATA[MID]!= ITEM.
3.  If ITEM < DATA[MID], then:
   Set END=MID-1.
Else
Set BEG=MID+1.
[End of if structure]
4. Set MID := INT((BEG + END)/2).
[End of Step 2 loop.]
5. If DATA[MID]= ITEM, then:
Set LOC := MID
Else
Set LOC := NULL.
|End of If structure).
6. Exit.
```

(Bubble Sort):

```
BUBBLE(DATA, N)
Here DATA is an array with N elements. This algorithm sorts the elements in DATA.
1. Repeat Steps 2 and 3 for K =1 to N - 1.
    2.Set PTR := 1. [Initializes pass pointer PTR.]
    3.Repeat while PTR <=N - K: [Executes pass.]
        (a) If DATA[PTR] > DATA[PTR + ], then:
                Interchange DATA[PTR] and DATA[PTR + 1].
            [End of If structure.]
        (b) Set PTR := PTR + 1.
        [End of inner loop.]
    [End of Step 1 outer loop.]
4. Exit.
```

MergeSort:

Procedure:

```
Step1: [compare low is less than high]
If LOW<=HIGH
Step 2:[find middle position]
MID=(LOW+HIGH)/2
Step3: [call the recursive function to split left subarray]
Call mergesort(A,LOW,MID)
Step4: [call the recursive function to split right subarray]
Call mergesort(A,MID+1,HIGH)
Step5:[combine Sorted subarray]
Merge(A,LOW MID HIGH)
```

Algorithm:

```
Step1: [initialize]
I=LOW; J=MID+1; K=LOW
Step2: [compare corresponding elements &output the smallest]
Repeat while I<=MID && J<=HIGH
If A[I]<=A[J]
Then TEMP[K]=A[I]
I=I+1; K=K+1
Else
TEMP[K]=A[J]
J=J+1; K=K+1
```