

Definition

- **Data:** Collection of **raw facts**.
- **Data structure** is representation of the logical relationship existing between individual elements of data.
- **Data structure** is a specialized format for organizing and storing data in memory that considers not only the elements stored but also their relationship to each other.

Introduction

- Data structure affects the design of both structural & functional aspects of a program.

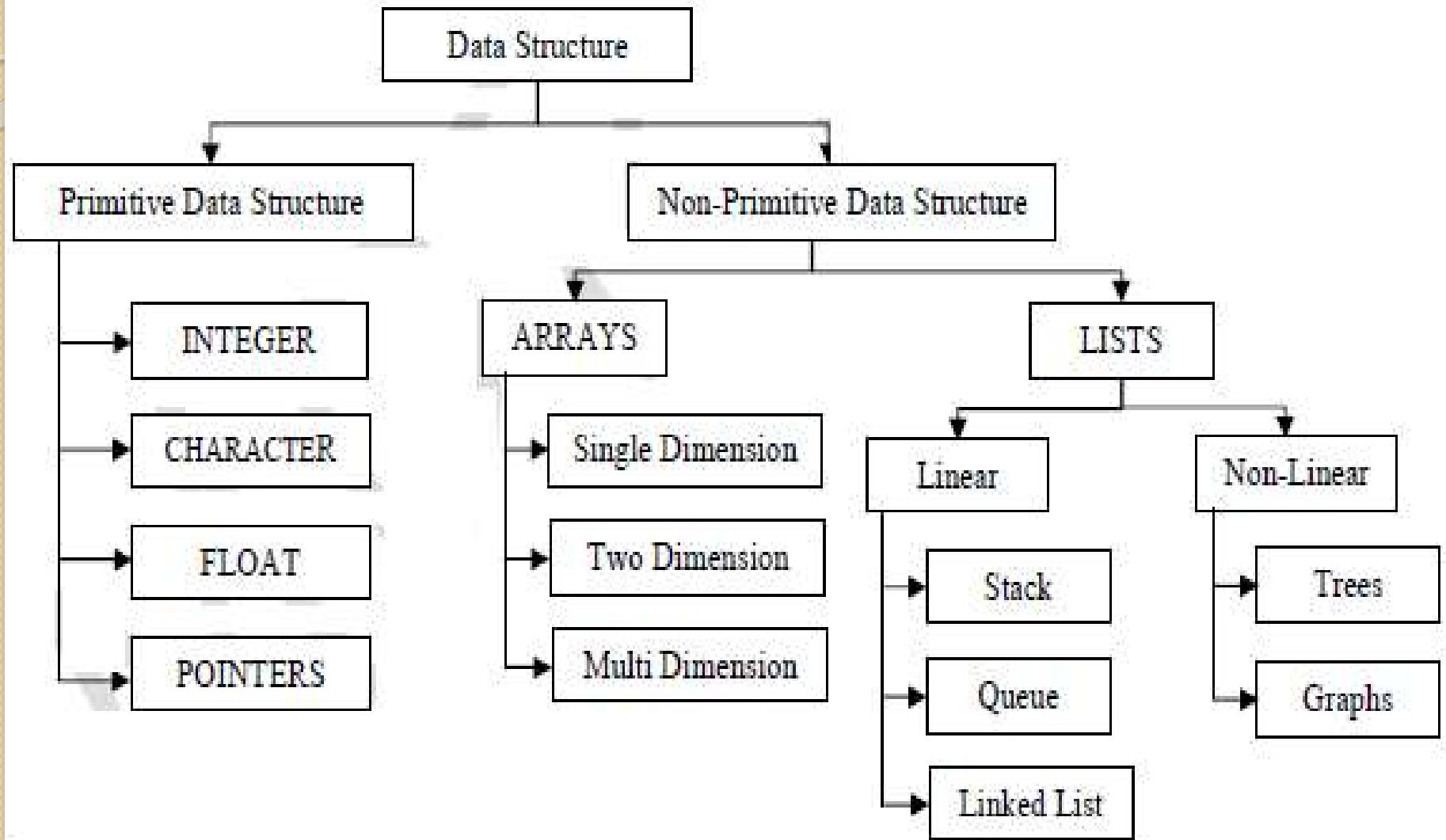
Program=algorithm + Data Structure

- You know that a algorithm is a step by step procedure to solve a particular function.

Classification of Data Structure

- Data structure are normally divided into two broad categories:
 - Primitive Data Structure
 - Non-Primitive Data Structure

Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- *Data structures that are directly operated upon the machine-level instructions are known as primitive data structures.*
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
 - Create
 - Selection
 - Updating
 - Destroy or Delete

Non-Primitive Data Structure

- There are more sophisticated data structures.
- *The Data structures that are derived from the primitive data structures are called Non-primitive data structure.*
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

Linear Data structures:

- *Linear Data structures are kind of data structure that has homogeneous elements.*
- The data structure in which elements are in a sequence and form a liner series.
- Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion.
- Some commonly used linear data structures are **Stack, Queue and Linked Lists.**

Non-Linear Data structures:

- *A Non-Linear Data structures is a data structure in which data item is connected to several other data items.*
- Non-Linear data structure may exhibit either a hierarchical relationship or parent child relationship.
- The data elements are not arranged in a sequential structure.
- The different non-linear data structures are **trees and graphs.**

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
 - Traversal
 - Insertion
 - Selection
 - Searching
 - Sorting
 - Merging
 - Destroy or Delete

Different between them

- A **primitive data structure** is generally a basic structure that is usually built into the language, such as an integer, a float.
- A **non-primitive data structure** is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

One dimensional array:

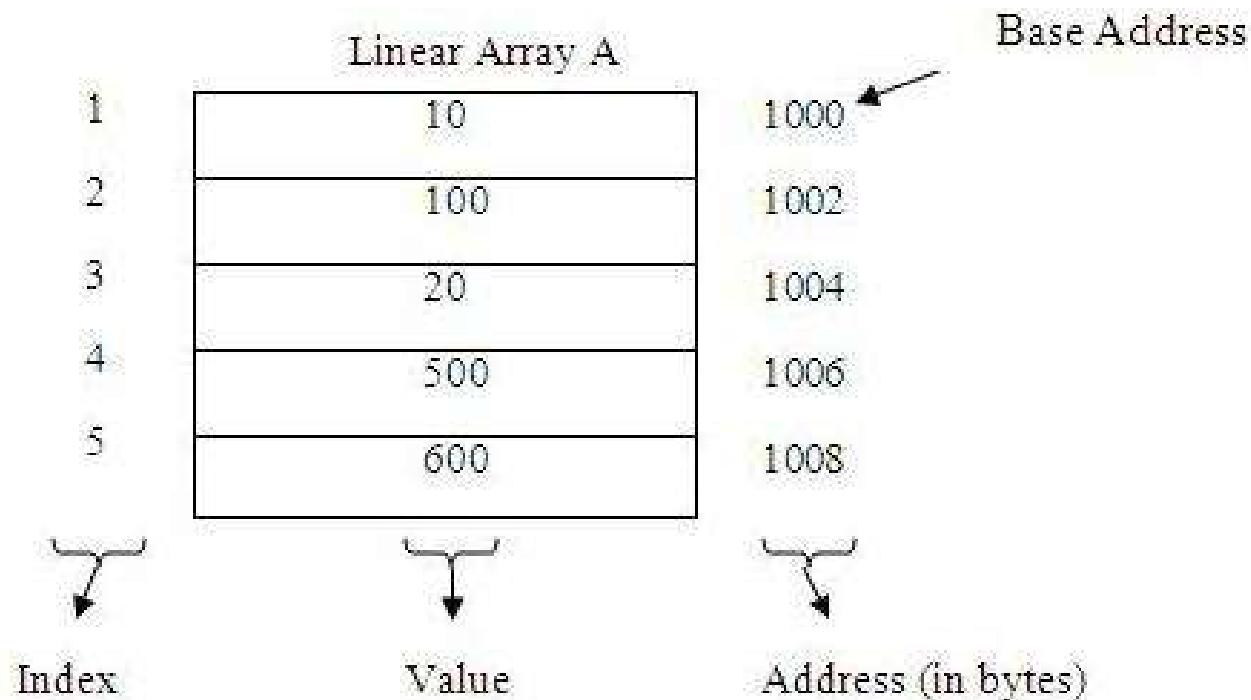
- *An array with only one row or column is called one-dimensional array.*
- It is finite collection of n number of elements of same type such that:
 - can be referred by indexing.
 - The syntax Elements are stored in continuous locations.
 - Elements x to define one-dimensional array is:
- **Syntax: Datatype Array_Name [Size];**
- Where,
 - Datatype : Type of value it can store (Example: int, char, float)
 - Array_Name: To identify the array.
- Size : The maximum number of elements that the array can hold.

Arrays

- Simply, declaration of array is as follows:
`int arr[10]`
- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Represent a Linear Array in memory

- The elements of linear array are stored in consecutive memory locations. It is shown below:



Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:
$$(\text{Upperbound}-\text{lowerbound})+1$$
- For the above array it would be $(9-0)+1=10$, where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop.

```
For(i=0;i<=9;i++)  
{   scanf("%d",&arr[i]);  
    printf("%d",arr[i]); }
```

Arrays types

- Single Dimension Array
 - Array with one subscript
- Two Dimension Array
 - Array with two subscripts (Rows and Column)
- Multi Dimension Array
 - Array with Multiple subscripts

Basic operations of Arrays

- Some common operation performed on array are:
 - Traversing
 - Searching
 - Insertion
 - Deletion
 - Sorting
 - Merging

Traversing Arrays

- **Traversing:** It is used to access each data item exactly once so that it can be processed.

E.g.

We have linear array A as below:

- 1 2 3 4 5
- 10 20 30 40 50

Here we will start from beginning and will go till last element and during this process we will access value of each element exactly once as below:

A [1] = 10
A [2] = 20
A [3] = 30
A [4] = 40
A [5] = 50

ALGORITHM: Traversal (A, LB, UB) A is an array with Lower Bound LB and Upper Bound UB.
Step 1: for LOC = LB to UB do
Step 2: PROCESS A [LOC]
 [End of for loop]
Step 3: Exit

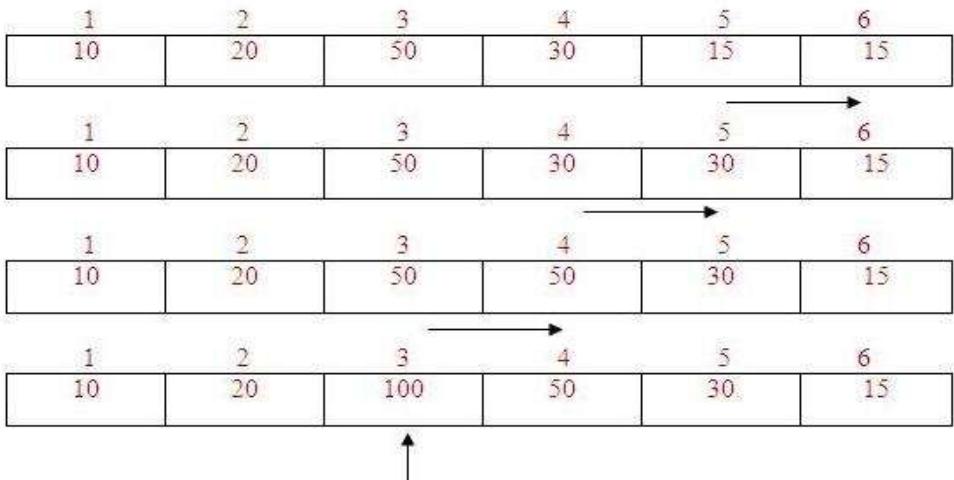
Insertion into Array

- **Insertion:** It is used to add a new data item in the given collection of data items.

E.g. We have linear array A as below:

1	2	3	4	5
10	20	50	30	15

New element to be inserted is 100 and location for insertion is 3. So shift the elements from 5th location to 3rd location downwards by 1 place. And then insert 100 at 3rd location. It is shown below:



ALGORITHM: Insert (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be inserted in the position Pos.

Step 1: for I = N-1 down to Pos

$A[I+1] = A[I]$

[End of for loop]

Step 2: $A[Pos] = ITEM$

Step 3: $N = N + 1$

Step 4: Exit

Deletion from Array

- **Deletion:** It is used to delete an existing data item from the given collection of data items.

For example: Let $A[4]$ be an array with items 10, 20, 30, 40, 50 stored at consecutive locations.

Suppose item 30 has to be deleted at position 2. The following procedure is applied.

- Copy 30 to ITEM, i.e. Item = 30.
- Move Number 40 to the position 2.
- Move Number 50 to the position 3.

A[0]	10
A[1]	20
A[2]	30
A[3]	40
A[4]	50

A[0]	10
A[1]	20
A[2]	30
A[3]	40
A[4]	50

A[0]	10
A[1]	20
A[2]	40
A[3]	50
A[4]	

A[0]	10
A[1]	20
A[2]	40
A[3]	50
A[4]	

ALGORITHM: Delete (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be deleted in the position Pos and it is stored into variable Item.

- ```
Step 1: ITEM = A [Pos]
Step 2: for I = Pos down to N-1
 A[I] = A[I+1]
 [End of for loop]
Step 3: N = N-1
Step 4: Exit
```

# Searching in Arrays

- **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.

E.g. We have linear array A as below:

|           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|
| <b>1</b>  | <b>2</b>  | <b>3</b>  | <b>4</b>  | <b>5</b>  |
| <b>15</b> | <b>50</b> | <b>35</b> | <b>20</b> | <b>25</b> |

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

- 1) Compare 20 with 15  
20 ≠ 15, go to next element.
- 2) Compare 20 with 50  
20 ≠ 50, go to next element.
- 3) Compare 20 with 35  
20 ≠ 35, go to next element.
- 4) Compare 20 with 20  
20 = 20, so 20 is found and its location is 4.

# Linear Search

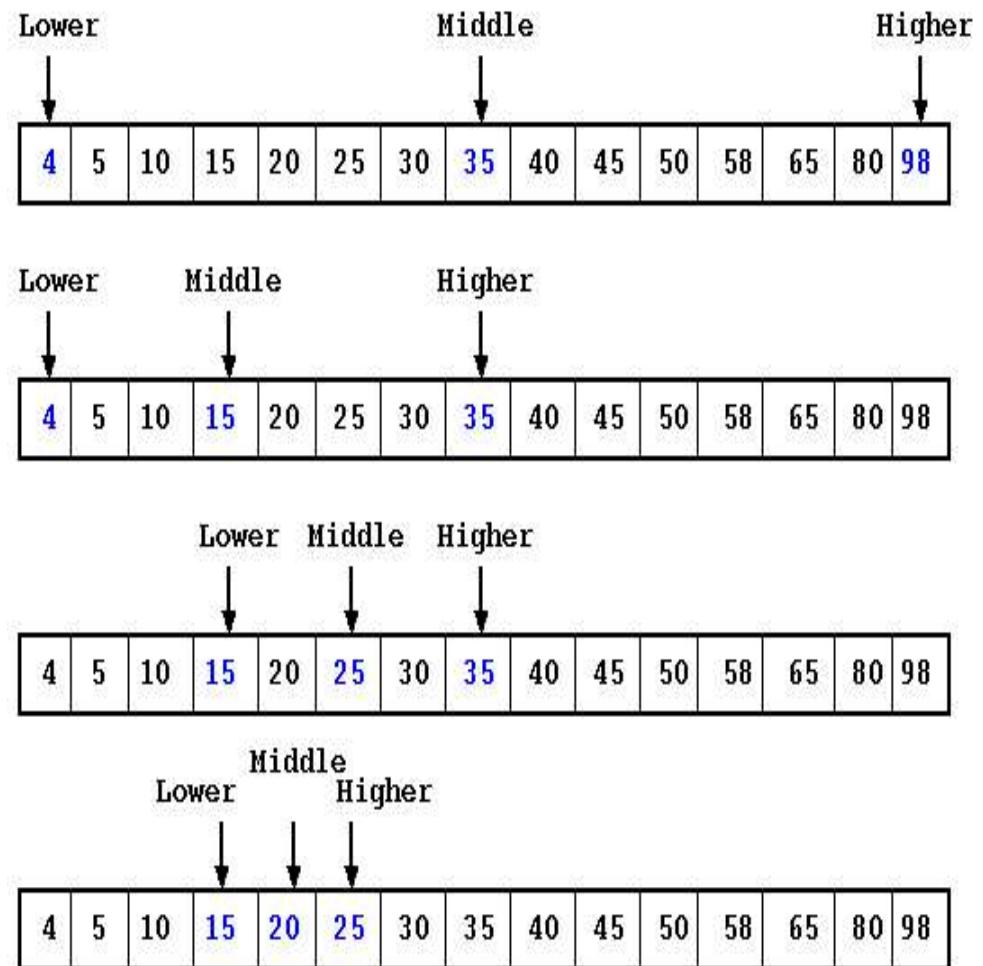
## Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of **ITEM** using sequential search.

1. Start
2. Set  $J = 0$
3. Repeat steps 4 and 5 while  $J < N$
4. IF  $LA[J]$  is equal **ITEM** THEN GOTO STEP 6
5. Set  $J = J + 1$
6. PRINT  $J, ITEM$
7. Stop

# Binary Search

- The binary search algorithm can be used with only sorted list of elements.
- Binary Search first divides a large array into two smaller sub-arrays and then recursively operate the sub-arrays.
- Binary Search basically reduces the search space to half at each step



# Binary Search

- Example: Consider the following elements stored in an array and we are searching for the element 67. The trace of the algorithm is given below.

|      |      |      |      |      | BEG & END          | MID = (BEG+END)/2        | Compare                     | Location |
|------|------|------|------|------|--------------------|--------------------------|-----------------------------|----------|
| A[0] | A[1] | A[2] | A[3] | A[4] | BEG = 0<br>END = 4 | MID = (0+4)/2<br>MID = 2 | 67 > 39<br>(Does not match) | LOC = -1 |

The search element i.e. 67 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.

|      |      |      |      |      |                    |                          |                             |          |
|------|------|------|------|------|--------------------|--------------------------|-----------------------------|----------|
| A[0] | A[1] | A[2] | A[3] | A[4] | BEG = 3<br>END = 4 | MID = (3+4)/2<br>MID = 3 | 67 > 47<br>(Does not match) | LOC = -1 |
|------|------|------|------|------|--------------------|--------------------------|-----------------------------|----------|

The search element i.e. 67 is greater than the element in the middle position i.e. 47 then continues the search to the right portion of the middle element.

|      |      |      |      |      |                    |                          |                             |          |
|------|------|------|------|------|--------------------|--------------------------|-----------------------------|----------|
| A[0] | A[1] | A[2] | A[3] | A[4] | BEG = 4<br>END = 4 | MID = (4+4)/2<br>MID = 4 | 67 > 57<br>(Does not match) | LOC = -1 |
|------|------|------|------|------|--------------------|--------------------------|-----------------------------|----------|

The search element i.e. 67 is greater than the element in the middle position i.e. 57 then continues the search to the right portion of the middle element.

|                                   |      |      |      |      |                    |                                                               |
|-----------------------------------|------|------|------|------|--------------------|---------------------------------------------------------------|
| A[0]                              | A[1] | A[2] | A[3] | A[4] | BEG = 5<br>END = 4 | Since the condition (BEG <= END) is false the comparison ends |
| We get the output as 67 Not Found |      |      |      |      |                    |                                                               |

# Binary Search

```
Procedure binary_search
 A ← sorted array
 n ← size of array
 x ← value to be searched

 Set lowerBound = 1
 Set upperBound = n

 while x not found
 if upperBound < lowerBound
 EXIT: x does not exists.

 set midPoint = lowerBound + (upperBound - lowerBound) / 2

 if A[midPoint] < x
 set lowerBound = midPoint + 1

 if A[midPoint] > x
 set upperBound = midPoint - 1

 if A[midPoint] = x
 EXIT: x found at location midPoint
 end while

end procedure
```

# Searching

## ➤ Difference between Linear Search and Binary Search

|   | Linear Search                                          | Binary Search                                                      |
|---|--------------------------------------------------------|--------------------------------------------------------------------|
| 1 | This can be used in sorted and unsorted array          | This can be used only in sorted array                              |
| 2 | Array elements are accessed sequentially               | One must have direct access to the middle element in the sub list. |
| 3 | Access is very slow                                    | Access is faster.                                                  |
| 4 | This can be used in single and multi dimensional array | Used only in single dimensional array.                             |
| 5 | This technique is easy and simple in implementing      | Complex in operation                                               |

# Sorting

## ➤ Sorting the elements in an array:

- *Sorting is the arrangement of elements of the array in some order.*
- There are different sorting methods like Bubble Sort, Selection Sort, Shell Sort, Quick sort, Heap Sort, Insertion Sort etc.

## ➤ Insertion Sort:

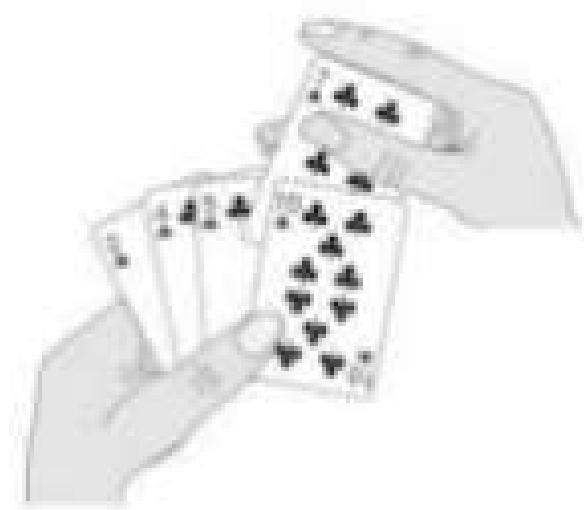
- In Insertion sort, the first element of the array is assumed to be in the correct position next element is considered as the key element and compared with the elements before the key element and is inserted in its correct position.
- Example: Consider the following array contains 8 elements as follows:

| A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] |
|------|------|------|------|------|------|------|------|
| 45   | 26   | 23   | 56   | 29   | 36   | 12   | 4    |

| Pass        | Location | A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] |
|-------------|----------|------|------|------|------|------|------|------|------|
| I=1         | J=0      | 45   | 26   | 23   | 56   | 29   | 36   | 12   | 4    |
| I=2         | J=0      | 26   | 45   | 23   | 56   | 29   | 36   | 12   | 4    |
| I=3         | J=3      | 23   | 26   | 45   | 56   | 29   | 36   | 12   | 4    |
| I=4         | J=2      | 23   | 26   | 45   | 56   | 29   | 36   | 12   | 4    |
| I=5         | J=3      | 23   | 26   | 29   | 45   | 56   | 36   | 12   | 4    |
| I=6         | J=0      | 23   | 26   | 29   | 36   | 45   | 56   | 12   | 4    |
| I=7         | J=0      | 12   | 23   | 26   | 29   | 36   | 45   | 56   | 4    |
| Sorted List |          | 4    | 12   | 23   | 26   | 29   | 36   | 45   | 56   |

# Insertion Sort

- **ALGORITHM: Insertion Sort (A, N)** A is an array with N unsorted elements.
  - Step 1: for  $I=1$  to  $N-1$
  - Step 2:  $J = I$   
    While( $J \geq I$ )  
        if ( $A[J] < A[J-1]$ ) then  
            Temp =  $A[J]$ ;  
             $A[J] = A[J-1]$ ;  
             $A[J-1] = Temp$ ;  
        [End if]  
         $J = J-1$   
    [End of While loop]  
[End of For loop]
  - Step 3: Exit



# Merging from Array

- **Merging:** It is used to combine the data items of two sorted files into single file in the sorted form  
We have sorted linear array A as below:

|    |    |    |    |    |     |
|----|----|----|----|----|-----|
| I  | 2  | 3  | 4  | 5  | 6   |
| 10 | 40 | 50 | 80 | 95 | 100 |

And sorted linear array B as below:

|    |    |    |    |
|----|----|----|----|
| I  | 2  | 3  | 4  |
| 20 | 35 | 45 | 90 |

After merging merged array C is as below:

|    |    |    |    |    |    |    |    |    |     |
|----|----|----|----|----|----|----|----|----|-----|
| I  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 10 | 20 | 35 | 40 | 45 | 50 | 80 | 90 | 95 | 100 |

# Two dimensional array

- A **two dimensional array** is a collection of elements and each element is identified by a pair of subscripts. (  $A[3][3]$  )
- The elements are stored in continuous memory locations.
- The elements of two-dimensional array as rows and columns.
- The number of rows and columns in a matrix is called as the order of the matrix and denoted as  $m \times n$ .
- The number of elements can be obtained by multiplying number of rows and number of columns.

|      | A[0] | A[1] | A[2] |
|------|------|------|------|
| A[0] | 10   | 20   | 30   |
| A[1] | 40   | 50   | 60   |
| A[2] | 70   | 80   | 90   |

# Representation of Two Dimensional Array:

- A is the array of order  $m \times n$ . To store  $m*n$  number of elements, we need  $m*n$  memory locations.
- The elements should be in contiguous memory locations.
- There are two methods:
  - Row-major method
  - Column-major method

# Two Dimensional Array:

- Row-Major Method: All the first-row elements are stored in sequential memory locations and then all the second-row elements are stored and so on. Ex: A[Row][Col]
- Column-Major Method: All the first column elements are stored in sequential memory locations and then all the second-column elements are stored and so on. Ex: A [Col][Row]

|      |    |         |
|------|----|---------|
| 1000 | 10 | A[0][0] |
| 1002 | 20 | A[0][1] |
| 1004 | 30 | A[0][2] |
| 1006 | 40 | A[1][0] |
| 1008 | 50 | A[1][1] |
| 1010 | 60 | A[1][2] |
| 1012 | 70 | A[2][0] |
| 1014 | 80 | A[2][1] |
| 1016 | 90 | A[2][2] |

**Row-Major Method**

|      |    |         |
|------|----|---------|
| 1000 | 10 | A[0][0] |
| 1002 | 40 | A[1][0] |
| 1004 | 70 | A[2][0] |
| 1006 | 20 | A[0][1] |
| 1008 | 50 | A[1][1] |
| 1010 | 80 | A[2][1] |
| 1012 | 30 | A[0][2] |
| 1014 | 60 | A[1][2] |
| 1016 | 90 | A[2][2] |

**Col-Major Method**

# Advantages of Array:

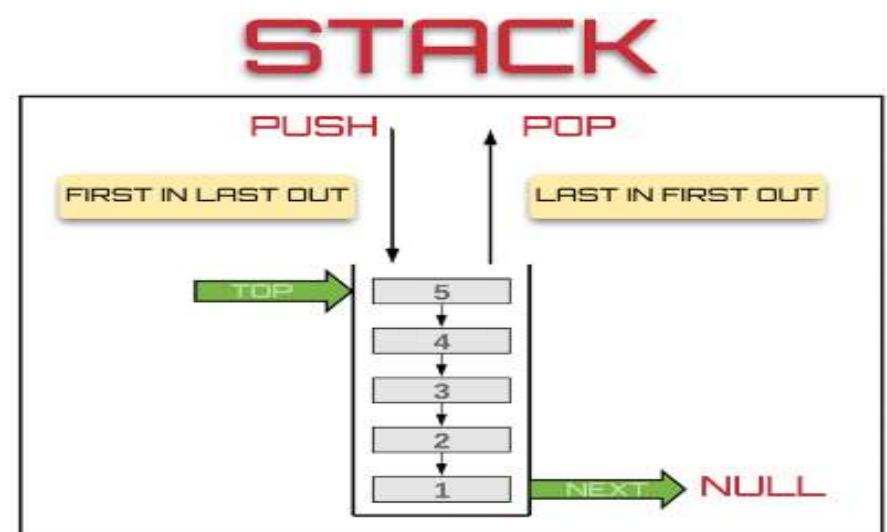
- It is used to represent multiple data items of same type by using single name.
- It can be used to implement other data structures like linked lists, stacks, queues, tree, graphs etc.
- Two-dimensional arrays are used to represent matrices.
- Many databases include one-dimensional arrays whose elements are records.

# Disadvantages of Array

- We must know in advance the how many elements are to be stored in array.
- Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or decreased.
- Array is fixed size; if we allocate more memory than requirement then the memory space will be wasted.
- The elements of array are stored in consecutive memory locations. So insertion and deletion are very difficult and time consuming.

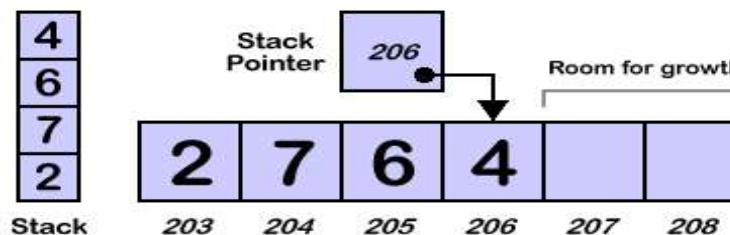
# Stack

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

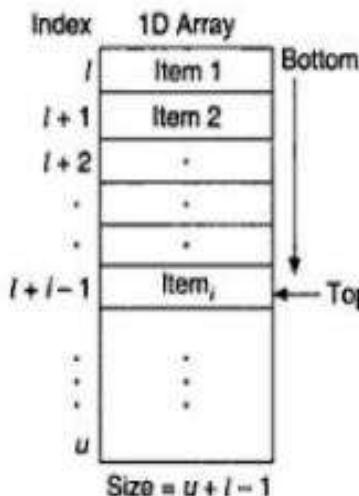


# Representation of Stack in Memory

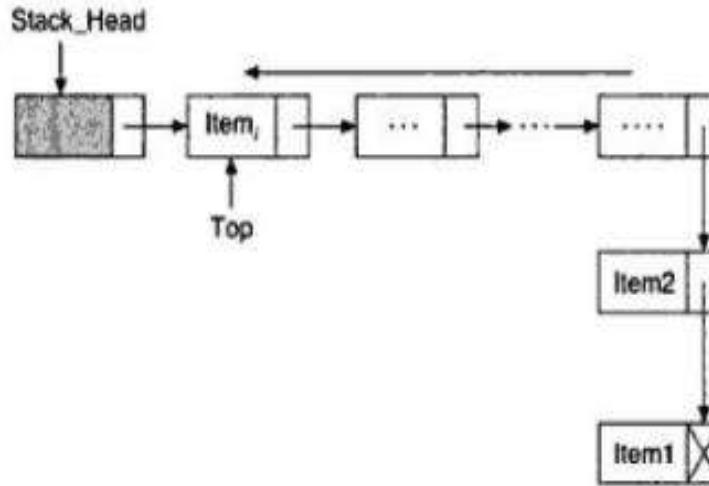
- The stack can be implemented into two ways:



- Using arrays (Static implementation)
- Using pointer (Dynamic implementation)



(a) Array representation of a stack



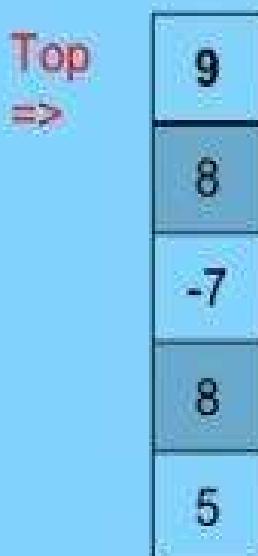
(b) Linked list representation of a stack

# Operation on Stacks:

- `Stack( )`: It creates a new stack that is empty. It needs no parameter and returns an empty stack.
- `push(item)`: It adds a new item to the top of the stack.
- `pop( )`: It removes the top item from the stack.
- `peek( )`: It returns the top item from the stack but does not remove it.
- `isEmpty( )`: It tests whether the stack is empty.
- `size( )`: It returns the number of items on the stack.

# Stack Conditions

- Depending on implementation, may be necessary to check if stack is full -- attempt to add item to a full stack is an overflow error

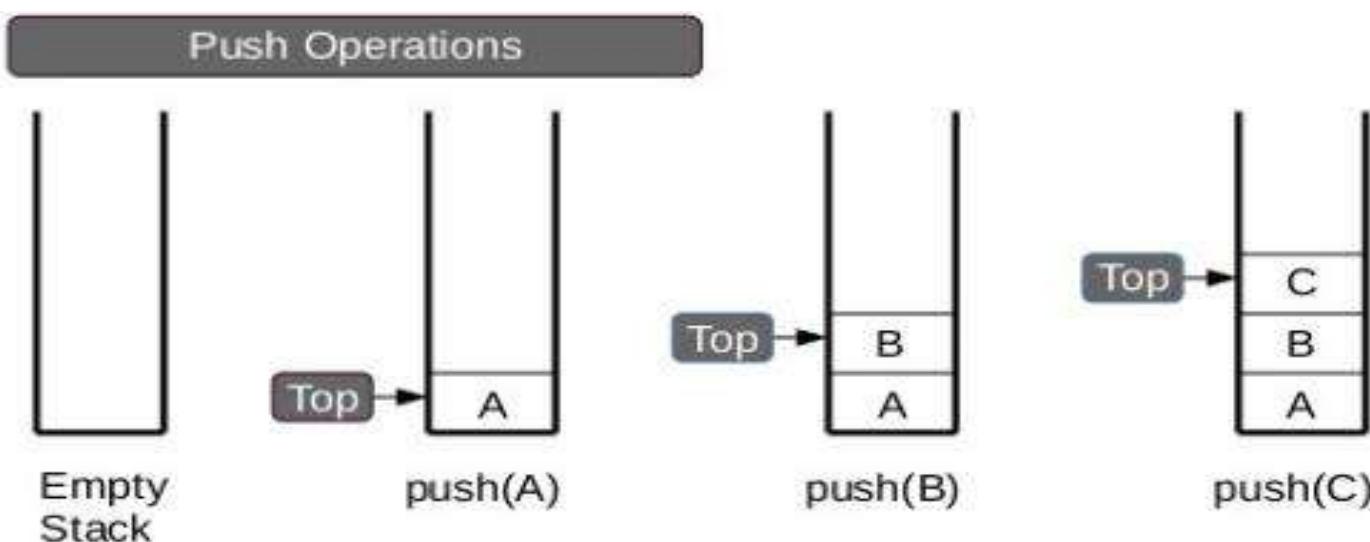


- Important to know if stack is empty -- attempt to remove an item from an empty stack is an underflow error.



# PUSH Operation

- The process of adding one element or item to the stack is represented by an operation called as the PUSH operation.*



## OVERFLOW STATE

If the stack is full and does not contain enough space to accept the given item

# PUSH Operation:

- *The process of adding one element or item to the stack is represented by an operation called as the PUSH operation.*
- The new element is added at the topmost position of the stack.

## ALGORITHM:

PUSH (STACK, TOP, SIZE, ITEM)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array. ITEM to be inserted.

Step 1: if TOP = N then [Check Overflow]  
PRINT “ STACK is Full or Overflow”  
Exit

[End if]

Step 2: TOP = TOP + 1 [Increment the TOP]

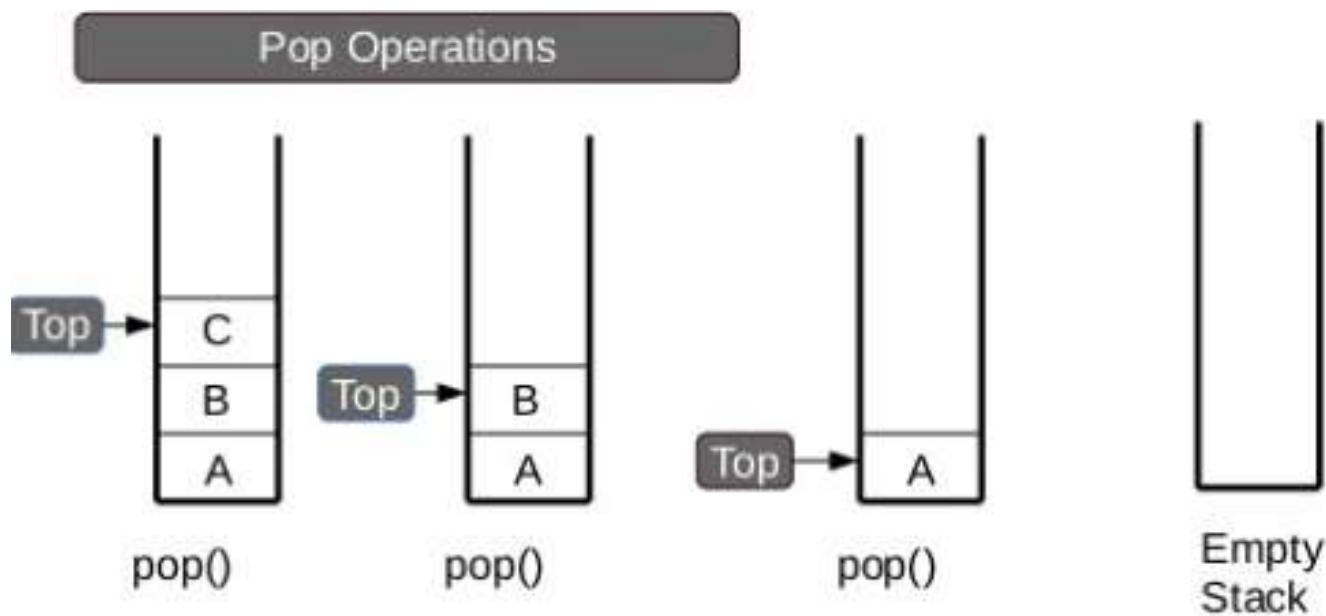
Step 3: STACK[TOP] = ITEM [Insert the ITEM]

Step 4: Return

# POP Operation

*The process of deleting one element or item from the stack is represented by an operation called as the POP operation.*

When elements are removed continuously from a stack, it shrinks at same end i.e., *top*



# POP Operation

*The process of deleting one element or item from the stack is represented by an operation called as the POP operation.*

**ALGORITHM:** POP (STACK, TOP, ITEM)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array. ITEM to be inserted.

Step 1: if TOP = 0 then [Check Underflow]

PRINT “ STACK is Empty or Underflow”

Exit

[End if]

Step 2: ITEM = STACK[TOP] [copy the TOP Element]

Step 3: TOP = TOP - 1 [Decrement the TOP]

Step 4: Return

# PEEK Operation

*The process of returning the top item from the stack but does not remove it called as the POP operation.*

**ALGORITHM:** PEEK (STACK, TOP)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array.

Step 1: if TOP = NULL then [Check Underflow]

        PRINT “ STACK is Empty or Underflow”

        Exit

        [End if]

Step 2: Return (STACK[TOP] [Return the top  
            element of the stack]

Step 3: Exit

# Application of Stacks

- It is used to reverse a word. You push a given word to stack – letter by letter and then pop letter from the stack.
- “Undo” mechanism in text editor.
- Backtracking: This is a process when you need to access the most recent data element in a series of elements. Once you reach a dead end, you must backtrack.
- Language Processing: Compiler’ syntax check for matching braces in implemented by using stack.
- Conversion of decimal number to binary.
- To solve tower of Hanoi.
- Conversion of infix expression into prefix and postfix.
- Quick sort
- Runtime memory management.

# Arithmetic Expression

- An expression is a combination of operands and operators that after evaluation results in a single value.
  - Operand consists of constants and variables.
  - Operators consists of {, +, -, \*, /, ), ] etc.
  - Expression can be

**Infix Expression:** If an operator is in between two operands, it is called infix expression.

- Example:  $a + b$ , where a and b are operands and + is an operator.

**Postfix Expression:** If an operator follows the two operands, it is called postfix expression.

- Example:  $ab +$

**Prefix Expression:** an operator precedes the two operands, it is called prefix expression.

- Example:  $+ab$

# Arithmetic Expression

Three notations for the given arithmetic expression are listed below:

Infix:  $((A + ((B \wedge C) - D)) * (E - (A/C)))$

Prefix:  $* + A - ^ BCD - E/AC$

Postfix:  $ABC \wedge D - + EAC / - *$

EXAMPLE: Let us illustrate the procedure *InfixToPostfix* with the following arithmetic expression:

*Input:*  $(A + B)^C - (D * E) / F$  (infix form)

| <i>Read symbol</i> | <i>Stack</i> | <i>Output</i>       |
|--------------------|--------------|---------------------|
| Initial            | (            |                     |
| 1                  | ((           |                     |
| 2                  | ((           | A                   |
| 3                  | ((+          | A                   |
| 4                  | ((+          | AB                  |
| 5                  | (            | AB+                 |
| 6                  | (^           | AB+                 |
| 7                  | (^           | AB + C              |
| 8                  | (-           | AB + C ^            |
| 9                  | ( - (        | AB + C ^            |
| 10                 | ( - (        | AB + C ^ D          |
| 11                 | ( - ( *      | AB + C ^ D          |
| 12                 | ( - ( *      | AB + C ^ DE         |
| 13                 | ( -          | AB + C ^ DE *       |
| 14                 | ( - /        | AB + C ^ DE *       |
| 15                 | ( - /        | AB + C ^ DE * F     |
| 16                 | ( - /        | AB + C ^ DE * F / - |

*Output:* A B + C ^ DE \* F / - (postfix form)

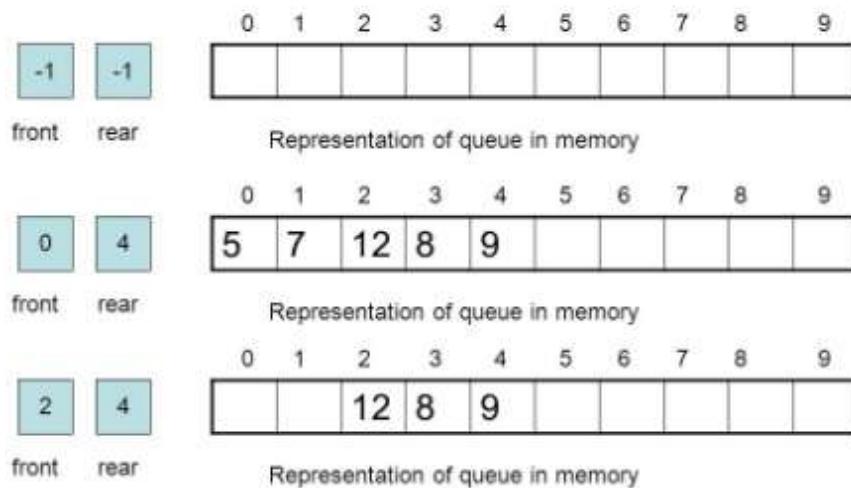
# Queue

- *A queue is an ordered collection of items where an item is inserted at one end called the “rear” and an existing item is removed at the other end, called the “front”.*
- Queue is also called as FIFO list i.e. First-In First-Out.
- In the queue only two operations are allowed enqueue and dequeue.
- Enqueue means to insert an item into back of the queue.
- Dequeue means removing the front item. The people standing in a railway reservation row are an example of queue.

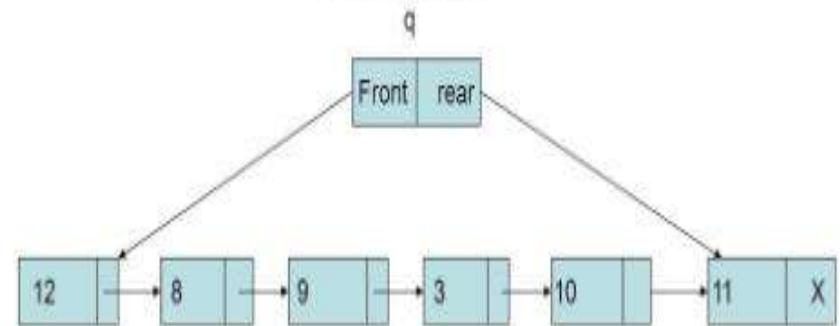
# Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

**Array representation of linear queue**



**Representation of a queue in memory**

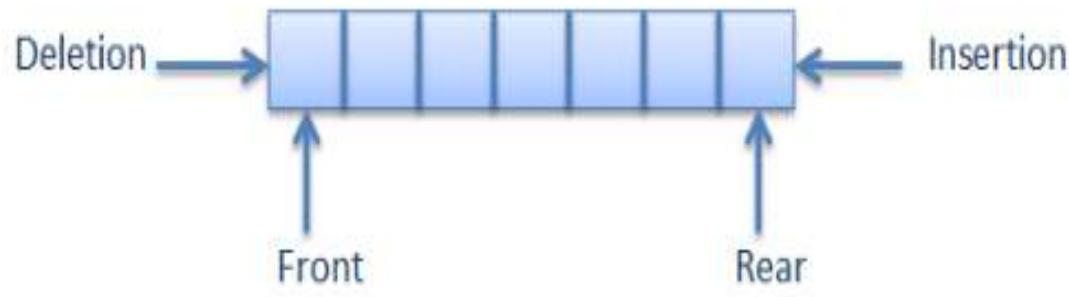


# Types of Queues

- Queue can be of four types:
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - De-queue ( Double Ended Queue)

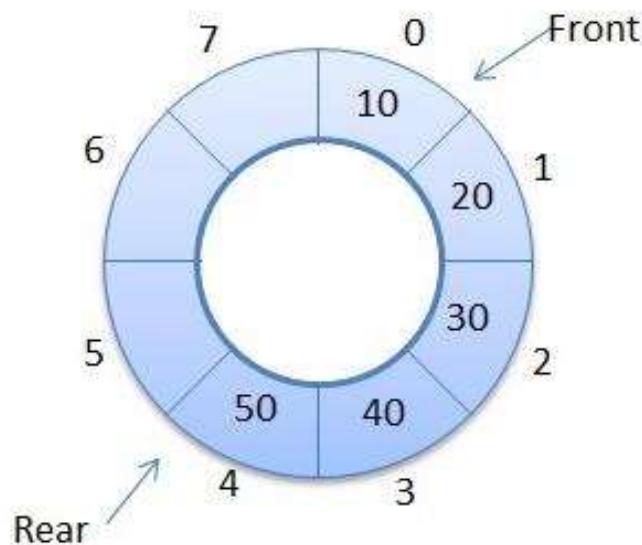
# Simple Queue

- Simple Queue: In simple queue insertion occurs at the rear end of the list and deletion occurs at the front end of the list.



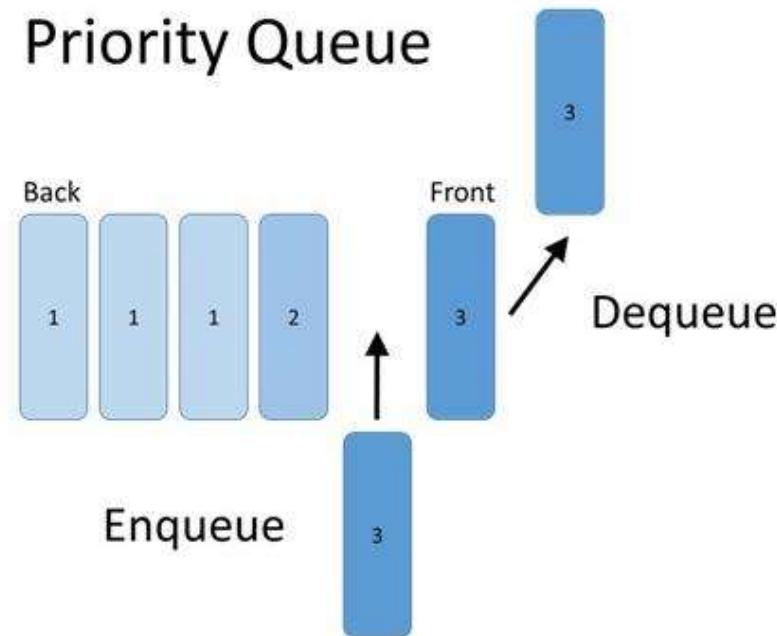
# Circular Queue

- Circular Queue: A circular queue is a queue in which all nodes are treated as circular such that the last node follows the first node.



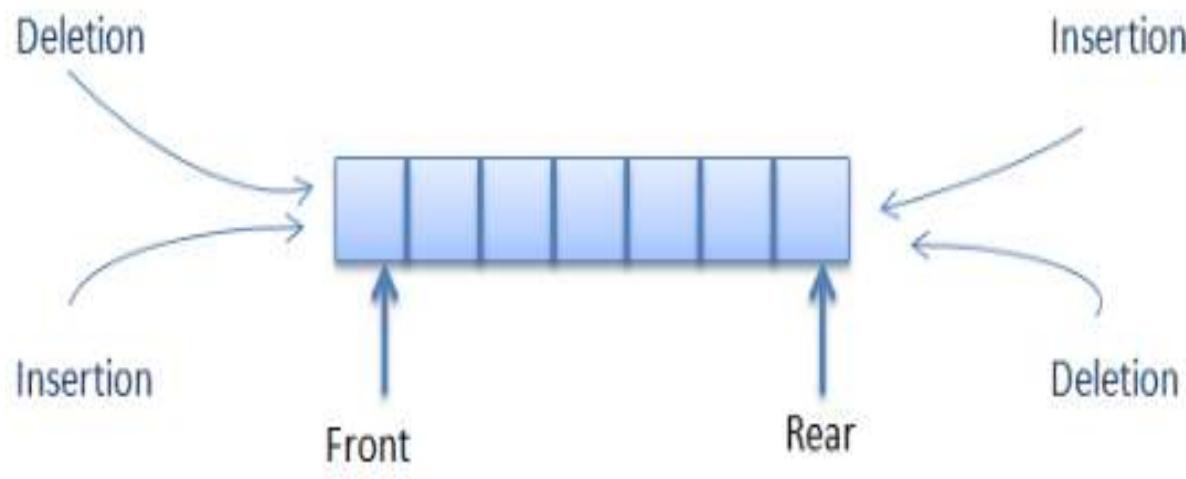
# Priority Queue

- A priority queue is a queue that contains items that have some present priority. An element can be inserted or removed from any position depending upon some priority.



# Dequeue Queue

- Dequeue: It is a queue in which insertion and deletion takes place at the both ends.



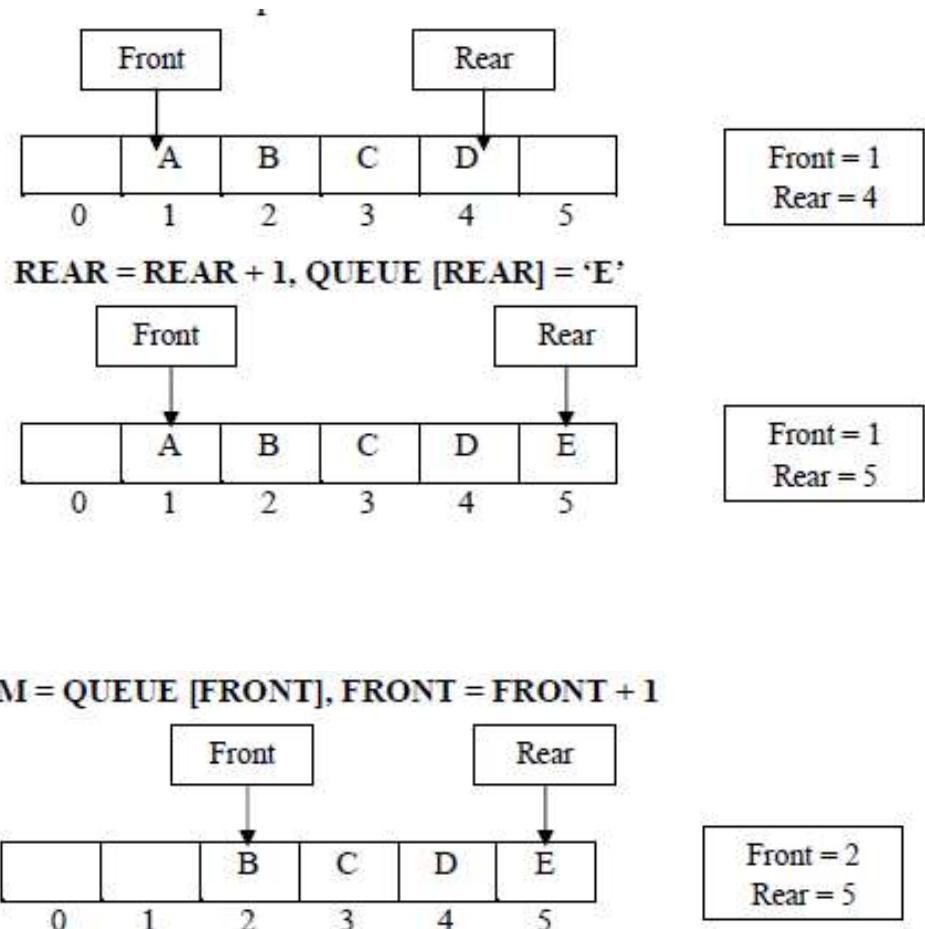
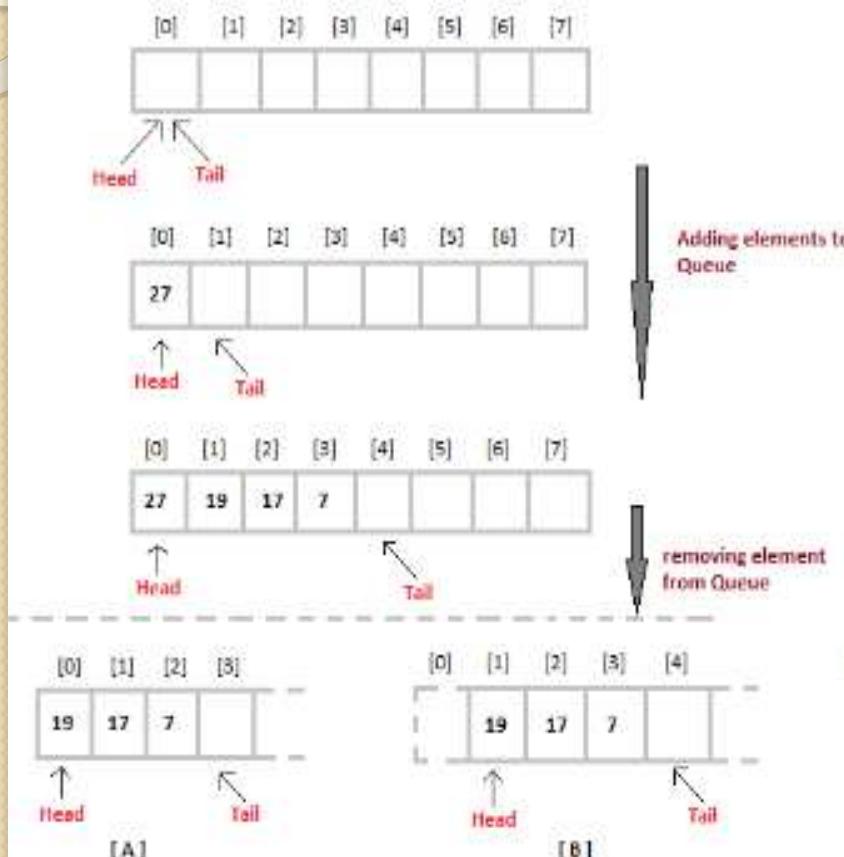
# Operation on Queues

- **Queue( ):** It creates a new queue that is empty.
- **enqueue(item):** It adds a new item to the rear of the queue.
- **dequeue( ):** It removes the front item from the queue.
- **isEmpty( ):** It tests to see whether the queue is empty.
- **size( ):** It returns the number of items in the queue.

# Memory Representation of a queue using array

- Queue is represented in memory using linear array.
- Let QUEUE is a array, two pointer variables called FRONT and REAR are maintained.
- The pointer variable FRONT contains the location of the element to be removed or deleted.
- The pointer variable REAR contains location of the last element inserted.
- The condition  $\text{FRONT} = \text{NULL}$  indicates that queue is empty.
- The condition  $\text{REAR} = N-1$  indicates that queue is full.

# Memory Representation of a queue using array



# Queue Insertion Operation (ENQUEUE):

- **ALGORITHM: ENQUEUE (QUEUE, REAR, FRONT, ITEM)**

QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if REAR = N-1 then [Check Overflow]

    PRINT “QUEUE is Full or Overflow”

    Exit

    [End if]

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] |
|------|------|------|------|------|
| 8    | 12   | 4    | 18   | 34   |

Step 2: if FRONT = NULL then [Check Whether Queue is empty]

    FRONT = -1

    REAR = -1

    else

        REAR = REAR + 1 [Increment REAR Pointer]

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] |
|------|------|------|------|------|
|      |      |      |      |      |

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] |
|------|------|------|------|------|
| 8    | 12   |      |      |      |

Step 3: QUEUE[REAR] = ITEM [Copy ITEM to REAR position]

Step 4: Return

# Queue Deletion Operation (DEQUEUE)

**ALGORITHM: DEQUEUE (QUEUE, REAR, FRONT, ITEM)**

QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if FRONT = NULL then [Check Whether Queue is empty]

    PRINT “QUEUE is Empty or Underflow”

    Exit

    [End if]

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] |
|------|------|------|------|------|
|      |      |      |      |      |

Step 2: ITEM = QUEUE[FRONT]

Step 3: if FRONT = REAR then [if QUEUE has only one element]

    FRONT = NULL

    REAR = NULL

    else

        FRONT = FRONT + 1 [Increment FRONT pointer]

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] |
|------|------|------|------|------|
| 8    |      |      |      |      |

Step 4: Return

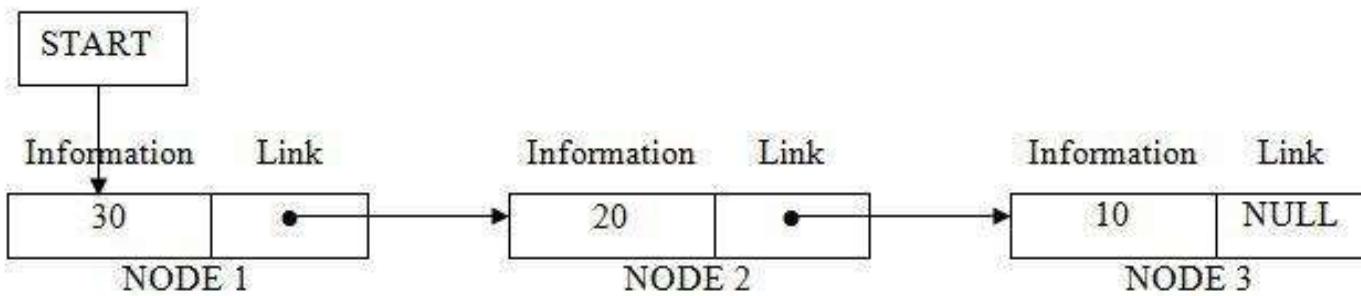
| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] |
|------|------|------|------|------|
| 8    | 12   |      |      |      |

# Application of Queue

- Simulation
- Various features of Operating system
- Multi-programming platform systems.
- Different types of scheduling algorithms
- Round robin technique algorithms
- Printer server routines
- Various application software's is also based on queue data structure.

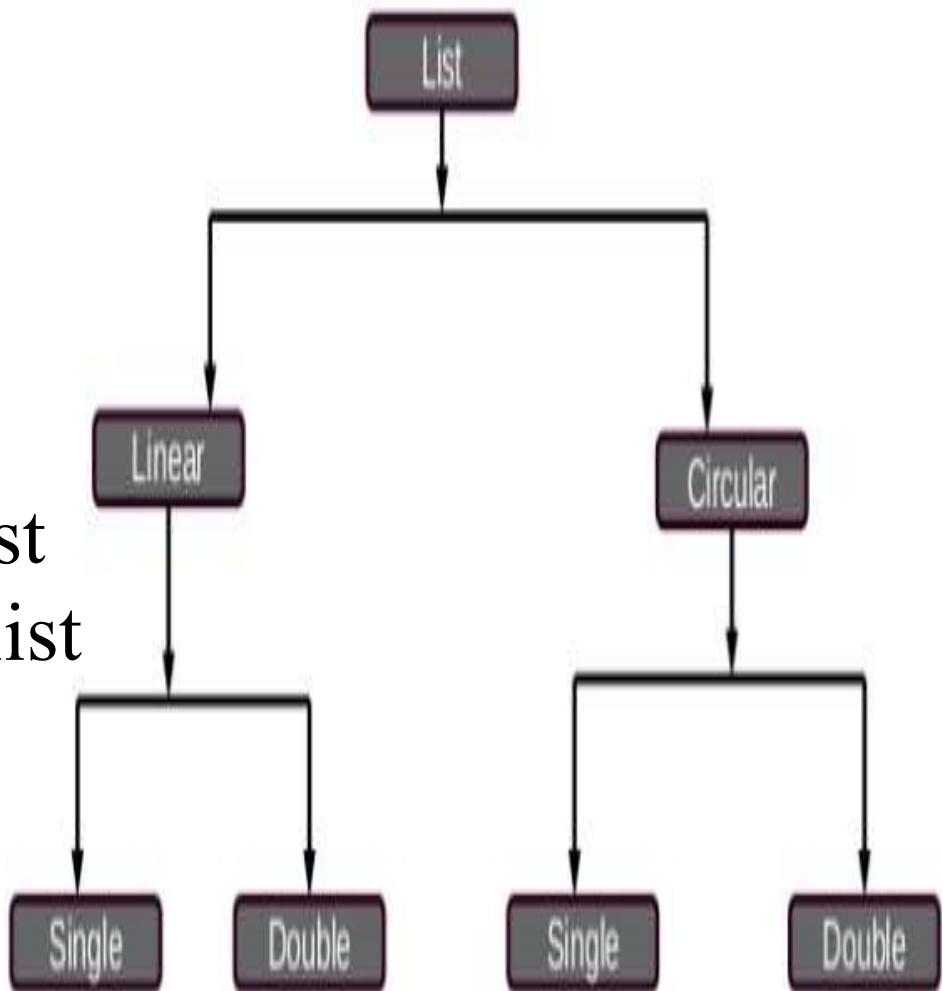
# Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items called ***nodes***.
- Lists are the most commonly used non-primitive data structures.
- Each nodes is divided into two parts:
  - The first part contains the information of the element.
  - The second part contains the memory address of the next node in the list. Also called Link part.



# Lists

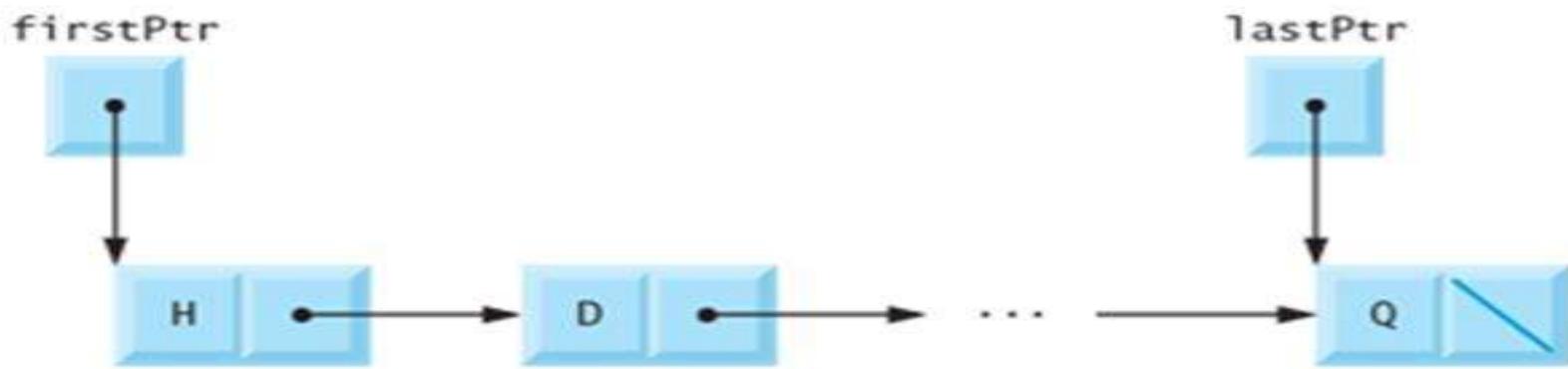
- Types of linked lists:
  - Single linked list
  - Doubly linked list
  - Single circular linked list
  - Doubly circular linked list



# Single linked list

*A singly linked list contains two fields in each node - an information field and the linked field.*

- The **information** field contains the data of that node.
  - The **link** field contains the memory address of the next node.
- There is only one link field in each node, the linked list is called singly linked list.

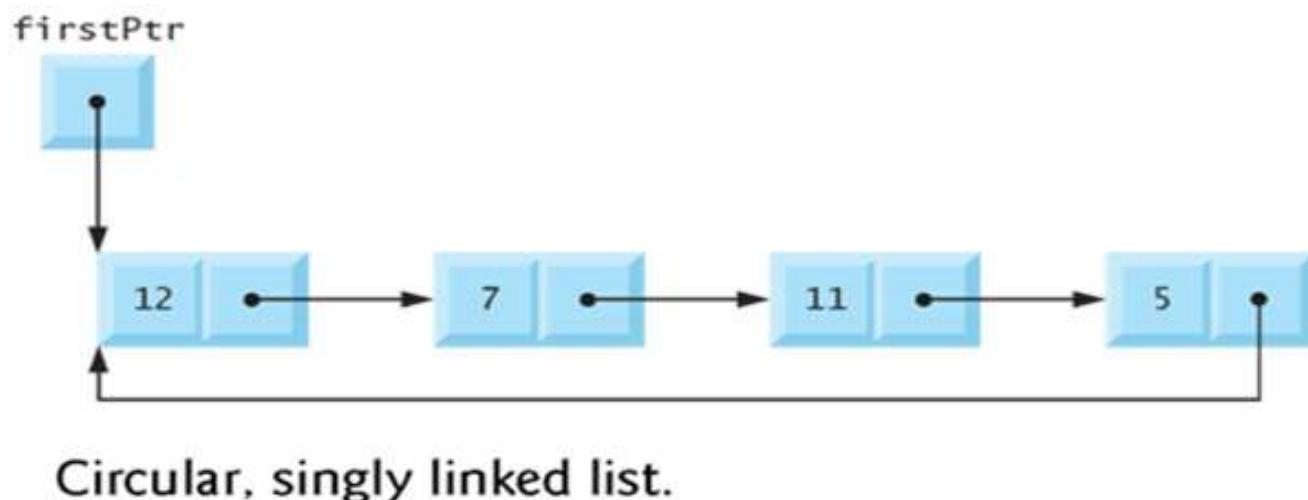


| A graphical representation of a list.

# Single circular linked list

*The link field of the last node contains the memory address of the first node, such a linked list is called circular linked list.*

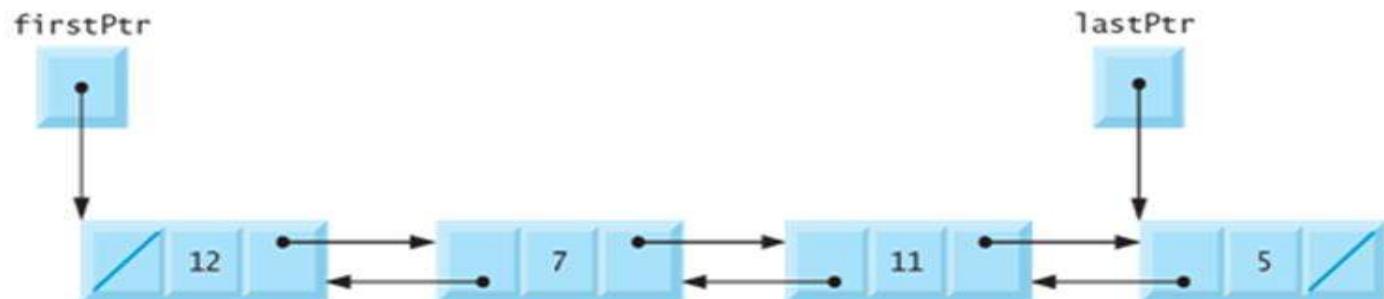
- In a circular linked list every node is accessible from a given node.



# Doubly linked list

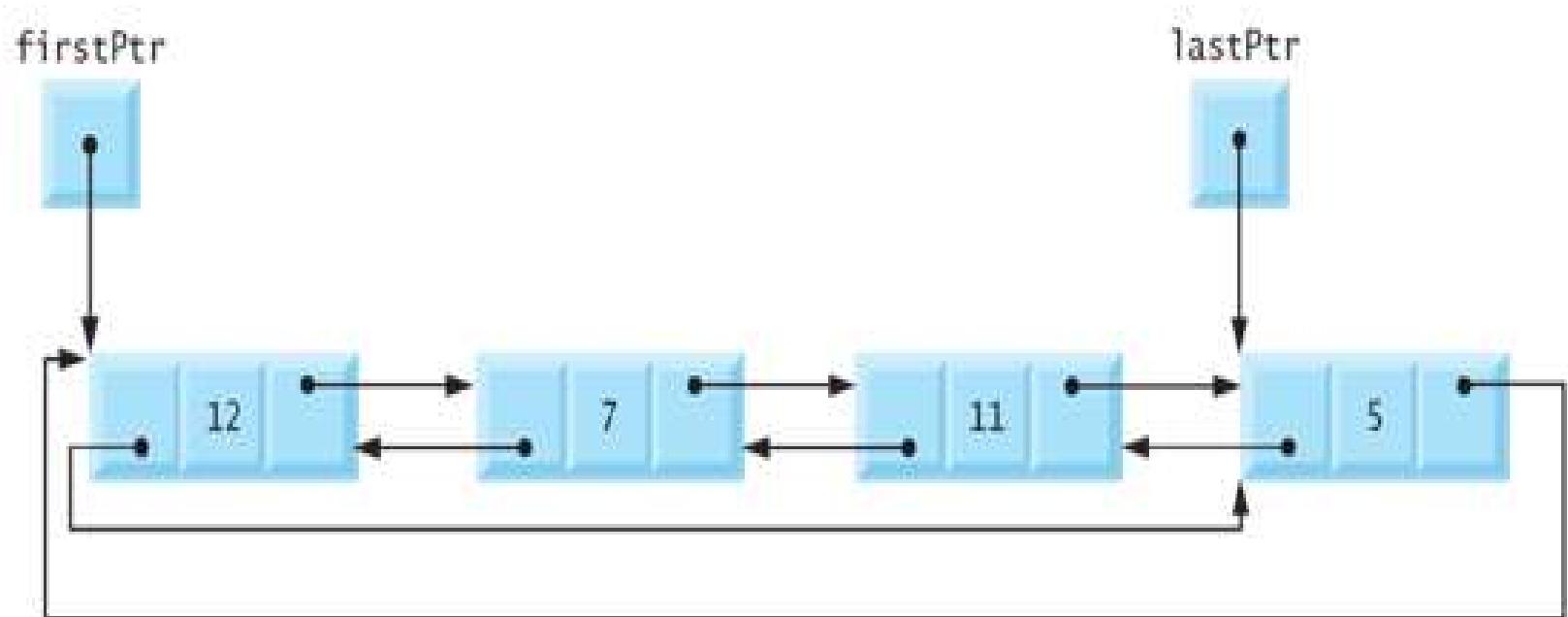
*It is a linked list in which each node is points both to the next node and also to the previous node.*

- In doubly linked list each node contains three parts:
  - FORW : It is a pointer field that contains the address of the next node
  - BACK: It is a pointer field that contains the address of the previous node.
  - INFO: It contains the actual data.
- In the first node, if BACK contains NULL, it indicated that it is the first node in the list.
- The node in which FORW contains, NULL indicates that the node is the last node.



Doubly linked list.

# Doubly circular linked list



**Fig. 19.12 | Circular, doubly linked list.**

# Operation on Linked List

- The operation that are performed on linked lists are:
  - Creating a linked list
  - Traversing a linked list
  - Inserting an item into a linked list.
  - Deleting an item from the linked list.
  - Searching an item in the linked list
  - Merging two or more linked lists.

# Creating a linked list

- The nodes of a linked list can be created by the following structure declaration.

```
struct Node
{
 int info;
 struct Node *link;
}*node1, node2;
```

- Here info is the information field and link is the link field.
- The link field contains a pointer variable that refers the same node structure. Such a reference is called as ***Self addressing pointer***.

# Operator new and delete

- Operators new allocate memory space.
  - Operators new [ ] allocates memory space for array.
- Operators delete deallocate memory space.
  - Operators delete [ ] deallocate memory space for array.

# Traversing a linked list:

- Traversing is the process of accessing each node of the linked list exactly once to perform some operation.
- **ALGORITHM: TRAVERS (START, P)** START contains the address of the first node. Another pointer p is temporarily used to visit all the nodes from the beginning to the end of the linked list.

Step 1:  $P = \text{START}$

Step 2: while  $P \neq \text{NULL}$

Step 3:      PROCESS data ( $P$ )      [Fetch the data]

Step 4:       $P = \text{link}(P)$                           [Advance P to next node]

Step 5: End of while

Step 6: Return

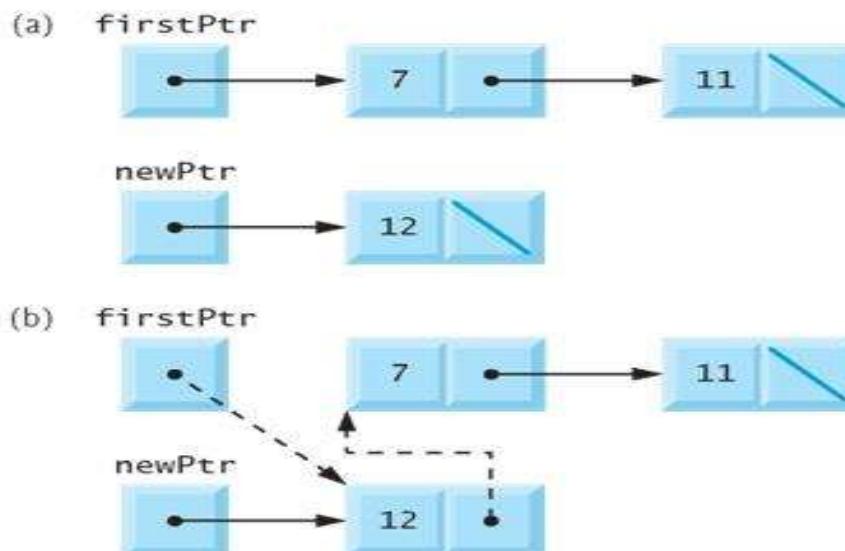
# Inserting a node into the linked list

- Inserting a node at the beginning of the linked list
- Inserting a node at the given position.
- Inserting a node at the end of the linked list.

# Inserting node at Front

**Inserting a node at the beginning of the linked list**

1. Create a node.
2. Fill data into the data field of the new node.
3. Mark its pointer field as NULL
4. Attach this newly created node to START
5. Make the new node as the START node.



Operation `insertAtFront` represented graphically.

# Inserting node at Front

- ALGORITHM: INS\_BEG (START, P)  
START contains the address of the first node.

Step 1:  $P \leftarrow \text{new Node};$

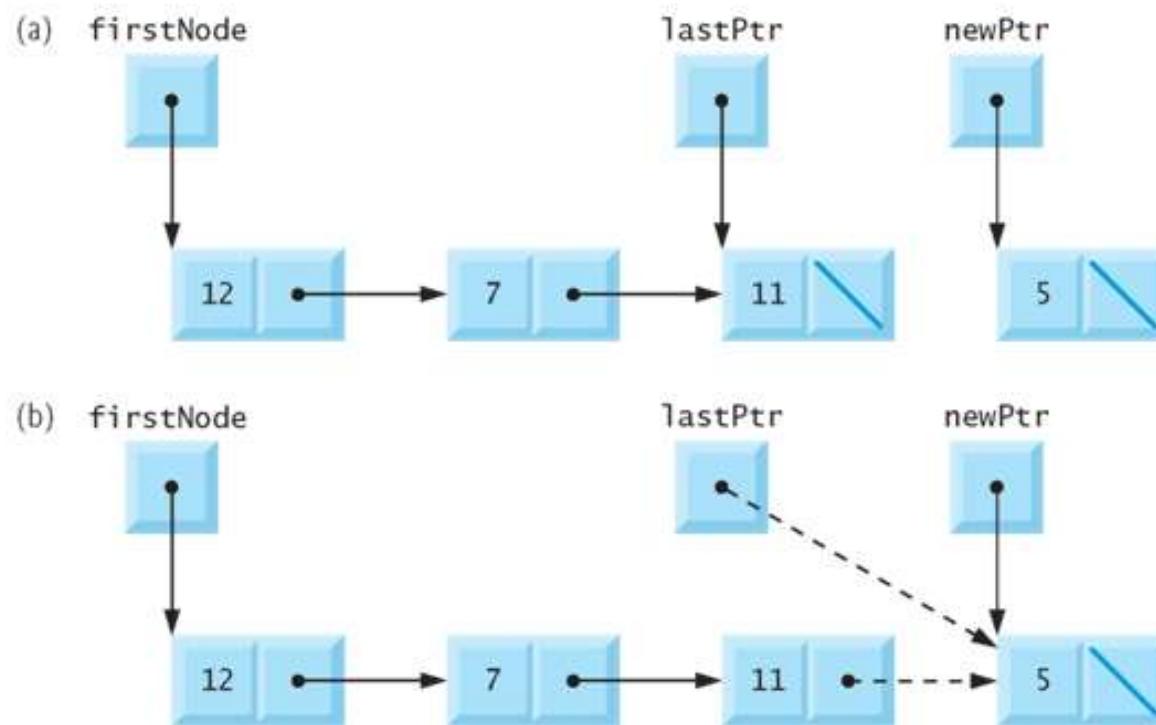
Step 2:  $\text{data}(P) \leftarrow \text{num};$

Step 3:  $\text{link}(P) \leftarrow \text{START}$

Step 4:  $\text{START} \leftarrow P$

Step 5: Return

# Inserting node at Last



| Operation `insertAtBack` represented graphically.

# Inserting node at Last

- ALGORITHM: INS\_END (START, P) START contains the address of the first node.

Step 1: START

Step 2:  $P \leftarrow \text{START}$  [identify the last node]

while  $P \neq \text{null}$

$P \leftarrow \text{next}(P)$

End while

Step 3:  $N \leftarrow \text{new Node};$

Step 4:  $\text{data}(N) \leftarrow \text{item};$

Step 5:  $\text{link}(N) \leftarrow \text{null}$

Step 6:  $\text{link}(P) \leftarrow N$

Step 7: Return

# Inserting node at a given Position

ALGORITHM: INS\_POS (START, P) START contains the address of the first node.

Step 1: START

Step 2:  $P \leftarrow \text{START}$  [Initialize node]

Count  $\leftarrow 0$

Step 3: while  $P \neq \text{null}$

count  $\leftarrow \text{count} + 1$

$P \leftarrow \text{next}(P)$

End while

Step 4: if ( $\text{POS}=1$ )

Call function INS\_BEG()

else if ( $\text{POS}=\text{Count}+1$ )

Call function INS\_END()

else if ( $\text{POS} \leq \text{Count}$ )

$P \leftarrow \text{Start}$

For( $i=1; i \leq \text{pos}; i++$ )

$P \leftarrow \text{next}(P);$

end for

[create]  $N \leftarrow \text{new node}$

$\text{data}(N) \leftarrow \text{item};$

$\text{link}(N) \leftarrow \text{link}(P)$

$\text{link}(P) \leftarrow N$

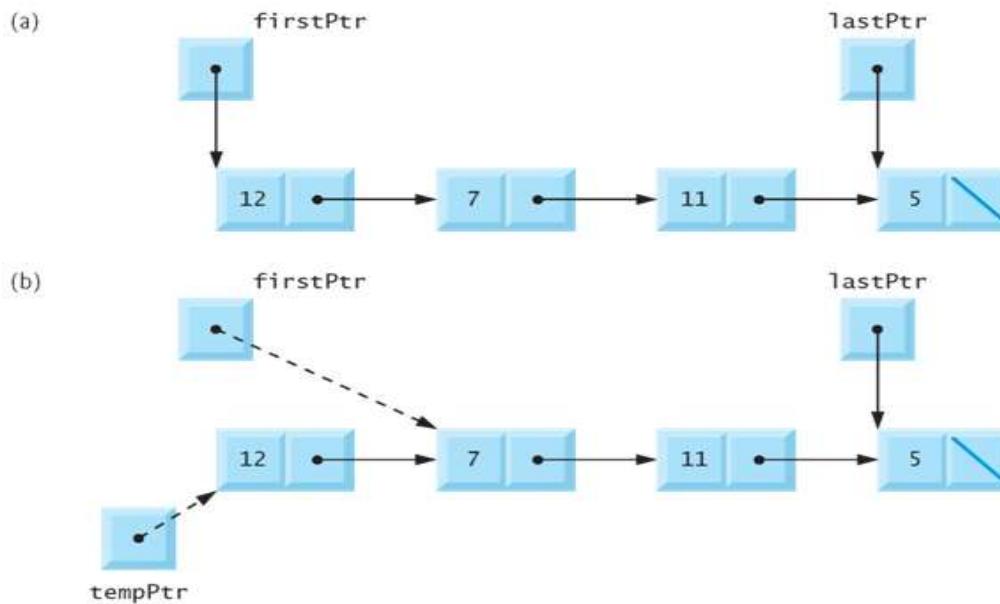
else

PRINT “Invalid position”

Step 5: Return

# Deleting an node

- Deleting an item from the linked list:
  - Deletion of the first node
  - Deletion of the last node
  - Deletion of the node at the give position



19.8 | Operation `removeFromFront` represented graphically.

# Deleting node from end

**ALGORITHM:** DEL-END (P1, P2, START) This used two pointers P1 and P2. Pointer P2 is used to traverse the linked list and pointer P1 keeps the location of the previous node of P2.

Step 1: START

Step 2:  $P2 \leftarrow \text{START};$

Step 3: while ( link(P2) != NULL)

$P1 \leftarrow P2$

$P2 \leftarrow \text{link}(P2)$

While end

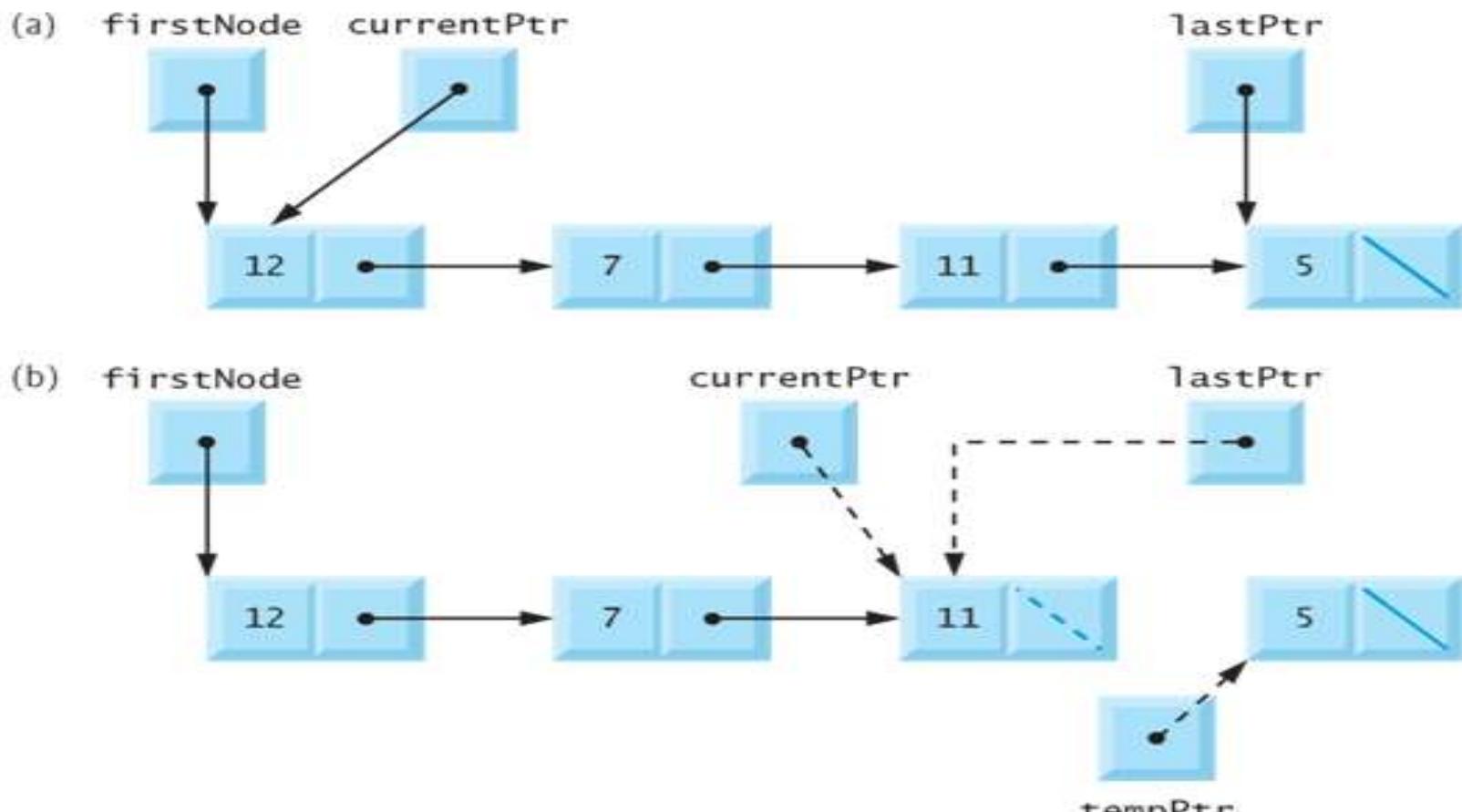
Step 4: PRINT data(p2)

Step 5:  $\text{link}(P1) \leftarrow \text{NULL}$

Free(P2)

Step 6: STOP

# Deleting node from end



| Operation `removeFromBack` represented graphically.

# Non-Linear Data structures

- *A Non-Linear Data structures is a data structure in which data item is connected to several other data items.*
- The data items in non-linear data structure represent hierarchical relationship.
- Each data item is called node.
- The different non-linear data structures are
  - Trees
  - Graphs.

# Trees

- A tree is a data structure consisting of nodes organized as a hierarchy.
- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

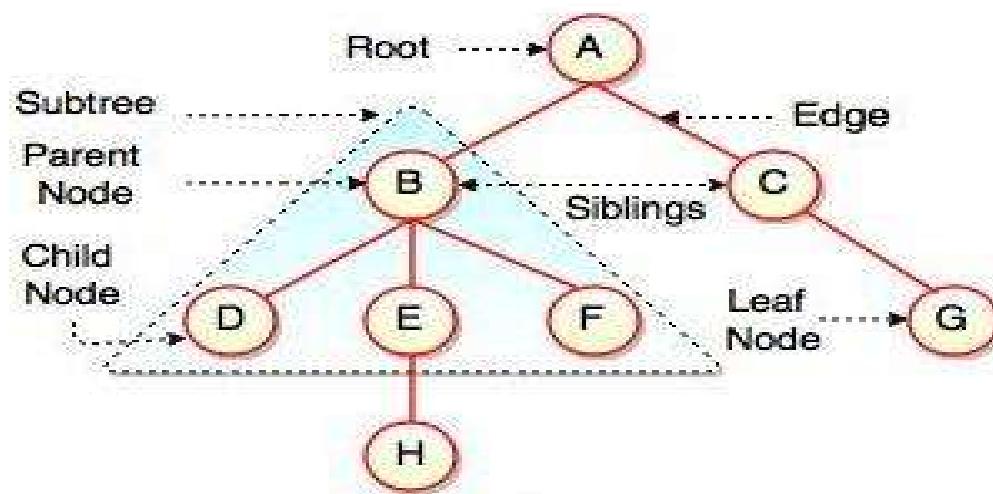
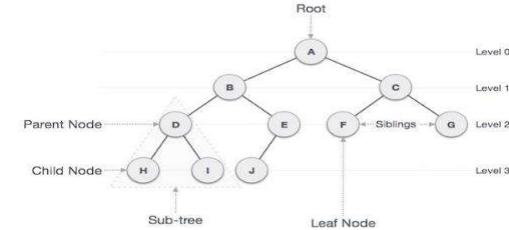


Fig. Structure of Tree

# Terminology of a Tree



| Field          | Description                                                                                            |
|----------------|--------------------------------------------------------------------------------------------------------|
| Root           | Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent. |
| Parent Node    | Parent node is an immediate predecessor of a node.                                                     |
| Child Node     | All immediate successors of a node are its children.                                                   |
| Siblings       | Nodes with the same parent are called Siblings.                                                        |
| Path           | Path is a number of successive edges from source node to destination node.                             |
| Height of Node | Height of a node represents the number of edges on the longest path between that node and a leaf.      |
| Height of Tree | Height of tree represents the height of its root node.                                                 |
| Depth of Node  | Depth of a node represents the number of edges from the tree's root node to the node.                  |
| Degree of Node | Degree of a node represents a number of children of a node.                                            |
| Edge           | Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf. |

# Binary Tree

- A binary tree is an ordered tree in which each internal node can have maximum of two child nodes connected to it.
- A binary tree consists of:
  - A node ( called the root node)
  - Left and right sub trees.
- A Complete binary tree is a binary tree in which each leaf is at the same distance from the root i.e. all the nodes have maximum two subtrees.

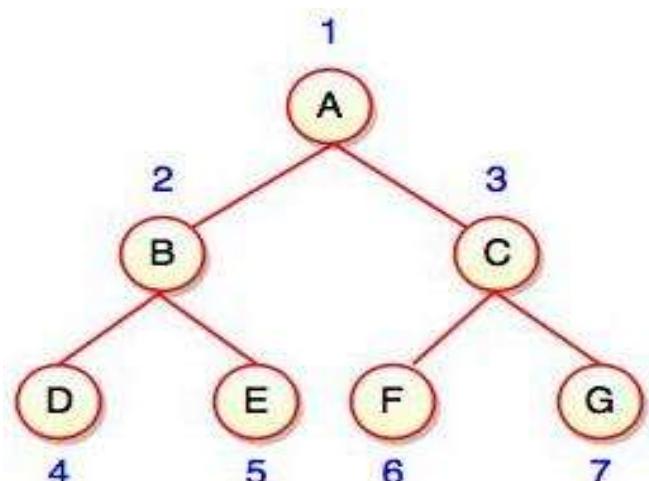


Fig. Binary Tree using Array

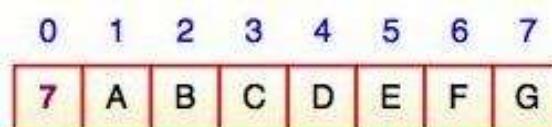


Fig. Location Number of an Array in a Tree

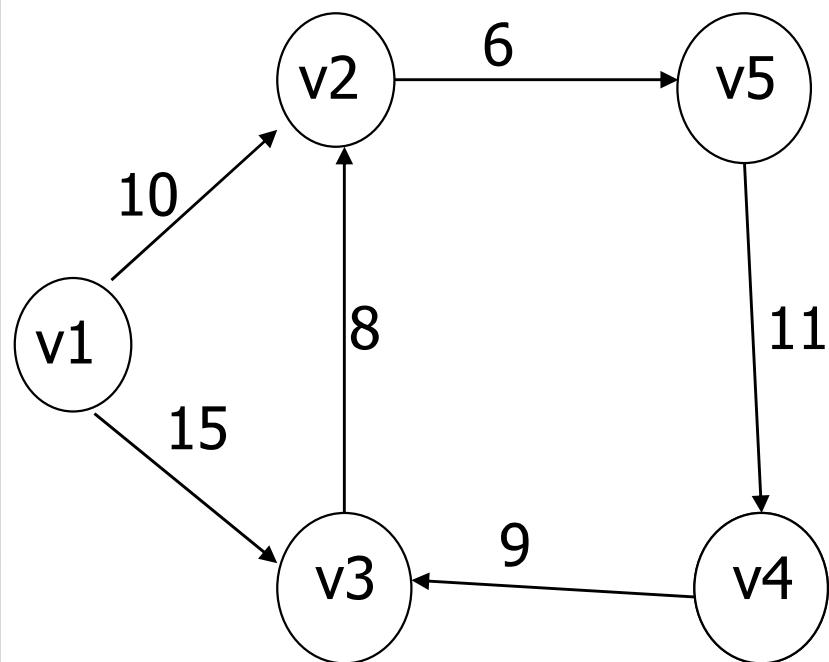
Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

# Graph

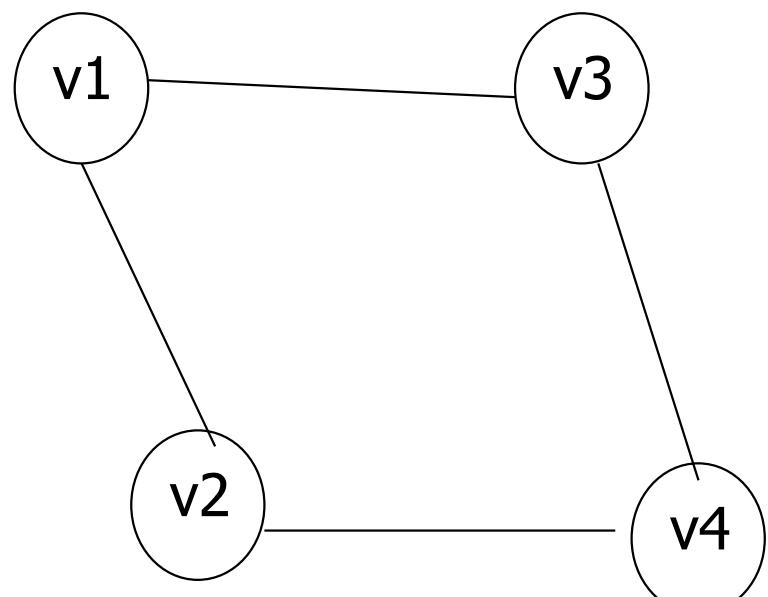
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- *A graph is a set of vertices and edges which connect them.*
- A graph is a collection of nodes called vertices and the connection between them called edges.
- Definition: A graph  $G(V,E)$  is a set of vertices  $V$  and a set of edges  $E$ .

# Graph

- Example of graph:



[a] Directed &  
Weighted Graph



[b] Undirected Graph

# Graph

- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

# Graph

- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph