Data Structure:

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.

Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program. Then A data structure consists of a base storage method (e.g., an array) and one or more algorithms that are used to access or modify that data.

Algorithm:

Is a finite set of instructions which accomplish a particular task. It is not enough to specify the form of data structure, but we must give also the algorithm of operation to access these data structures.

Algorithm + Data structure = Program

Basic Concepts of Data Structures:

- Data: Is the fact that we can see and deal with in our daily life like; book, car, 1245, ….etc.
- Information: Is a collection of words, numbers, dates, or communicated material that have meaning.

Data is the raw material while the Information is the processed form of data

Problems and Programs:

The selection of a particular data structure will help the programmer to design more efficient programs that will be used to solve complexity problems.

The programmers have to strive hard to solve these problems. If the problem is analyzed and divided into sub problems, the task will be much easier.

The simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover, there may be more than one program to solve a problem.

Analysis of programs:

The choice of a particular program depends on following performance analysis and measurements.

- Space Complexity:

The amount of memory that program needs to run completion. The space needed by program consist of:

1. Instruction space. (fixed space)

2. Data space.

    • Constant and simple variables. (fixed space)

    • Fixed size structural variables, like array and structures. Fixed space.

    • Dynamically space. Such as strings.

3. Environment stack space.

Needed to store the information to resume the suspended functions. The information that be saved are:

    • Return address of the called functions.

• Values of all the parameters which are send and return between functions and called points.

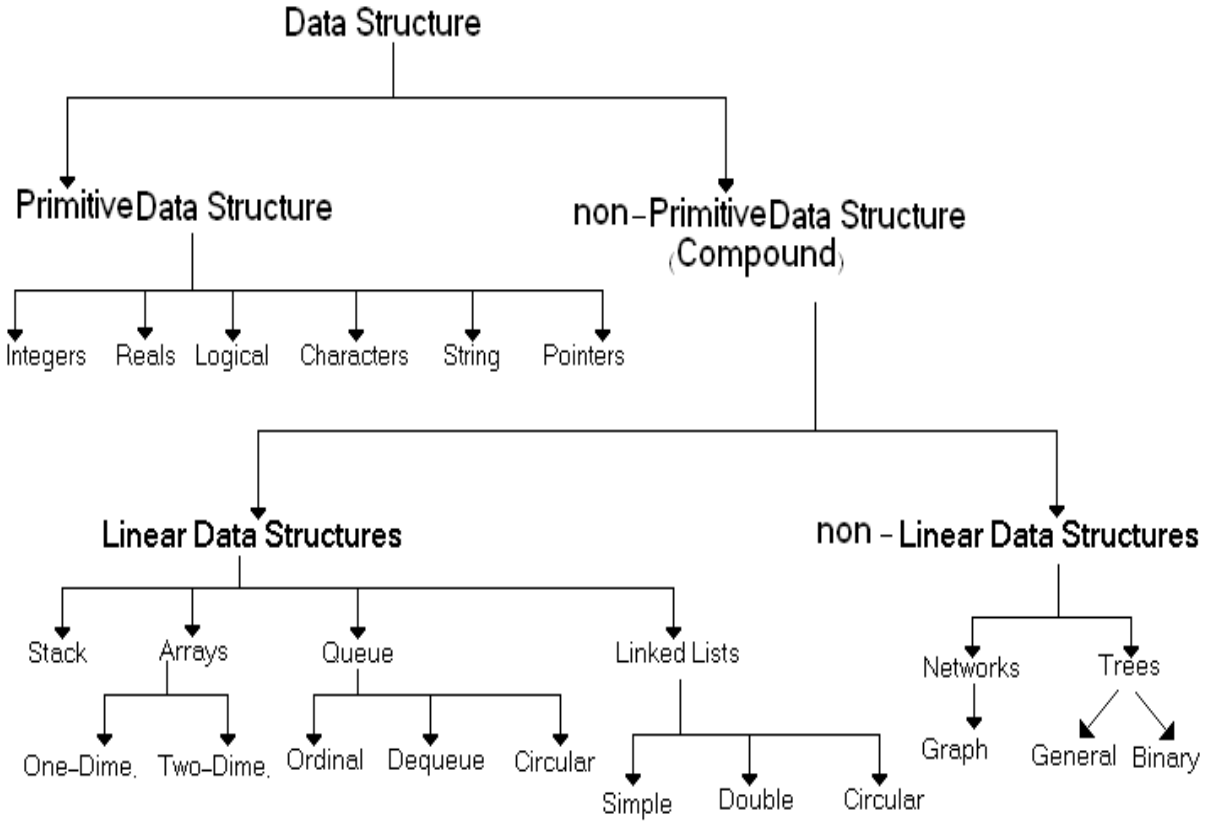
- Time Complexity:

The amount of time that program needs to run completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on.

Data Structure Types:

Programming languages support defined and use data items by providing formulas to single-value items such as (integer, real, character, and Boolean) and the multi-value items need to different data structures like (array, record, ..., etc.).

C++ supports the following data types:



Selection of Data Structure:

The choice of a particular data structure depends on two considerations:

1. it must be rich enough in structure to mirror the actual relationship of the data in the real world.
2. the structure should be simple enough that one can effectively process the data when necessary.

Before any data can be stored in memory, you must tell the computer how much space to reserve for data by using an abstract data type.

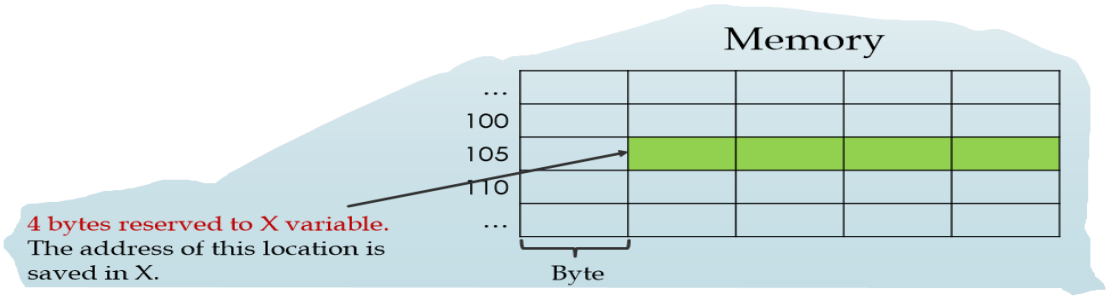
Memory is reserved by using a data type in a declaration statement. The form of a declaration statement varies depending on the programming language you use. Here is a declaration statement for C++:

```
Int my-variable;
```

We need to choose the suitable abstract data type for data that we want stored in memory, then use the abstract data type in a declaration statement to declare a variable. A variable is a reference to the memory location that we reserved using the declaration statement.

For Example:

```
int X;
```



Types of primitive data types

Data Type	Data Type Size in Bits (bytes)	Range of Values	Group
Byte	8 (1 byte)	–128 to 127	Integers
short 16	16 (2 bytes)	–32,768 to 32,767	Integers
int 32	32 (4 bytes)	–2,147,483,648 to 2,147,483,647	Integers
long 64	64 (8 bytes)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Integers
char 16 (Unicode)	16 (Unicode) (2 bytes)	65,536 (Unicode)	Characters
float 32	32 (4 bytes)	3.4e-038 to 3.4e+038	Floating-point
double 64	64 (8 bytes)	1.7e-308 to 1.7e+308	Floating-point
boolean	1	0 or 1	Boolean

Primitive data structures:

Standard primitive types are those types that are available on most computers as built-in features. They include the whole numbers, the logical truth values, and a set of printable characters. On many computers fractional numbers are also incorporated, together with the standard arithmetic operations. We denote these types by the identifiers INTEGER, REAL, BOOLEAN, CHAR, STRING, POINTER.

1. *Integer*:

The integer abstract data type group consists of four abstract data types used to reserve memory to store whole numbers: byte , short , int , and long depending on the nature of the data, sometimes an integer must be stored using a positive or negative sign, such as a +10 or –5. Other times an integer is assumed to be positive so there isn't any need to use a positive sign. An integer that is stored with a sign is called a signed number; an integer that isn't stored with a sign is called an unsigned number.

The declaration in C++:

```
Int x;
```

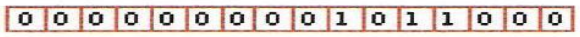
Generally, the largest positive integer encoded using n bits will be $(2^{n-1}-1)$.

- **Unsigned integer :-**

Type **unsigned** in C++ store its base-two representation in a fixed number w of bits (e. g., w=16 or w=32).

Example :

$$88 = 0000000001011000_2$$



- **Signed integer :-**

Type **int** in C++ store in a fixed number of bits (W) using one of the following :-

a- Sign-magnitude representation :-

Save one bit for sign (0=+ , 1=-) and use base-two representation in the other bits.

Examples:

$88 \rightarrow 0000000001011000$ (↑ sign bit)

$-88 \rightarrow 1000000001011000$ (↑ sign bit)

•Advantages:

Signed magnitude easy to understand and encode.

•Disadvantages:

1.Need additional bit for sign.

2.Not convenient for arithmetic.

3.They are two representations of zero

•00000000 = + 0_{10}

•10000000 = - 0_{10}

b- Two's complement representation :-

For n≥0 :- use ordinary base-two representation with leading (sign) bit 0.

For n<0 :-

1) Find w-bit base-2 representation of n.
2) Complement each bit.
3) Add 1

Let's see this demonstrated in an example:
We want to encode the value -5 using 8 bits. To do so:

- write 5 in binary: 00000101
- switch it to its complement: 11111010
- add 1: 11111011
- the 8-bit binary representation of -5 is 11111011

Comments:

The highest-weighted bit is 1, so it is indeed a negative number.If you add 5 and -5 (00000101 and 11111011) the sum is 0 (with remainder 1).

Example :- (-88)

1. 88 as a 16-bit base-two number 0000000001011000

2. Complement this bit string 1111111110100111

3. Add 1 1111111110101000

•**Advantages:**

- Only has one value for zero.
- Convenient for arithmetic.

Overflow is:

•An error that occurs when the computer attempts to handle a number that is too large for it.

•When the result of some operation on **byte** representations **falls outside** this range, then the **value** that is represented by the result is **different** from the correct value.

2. Reals:

The declaration of the real variables in C++ as:

```
float var1,var2;

double speed, gravity;
```

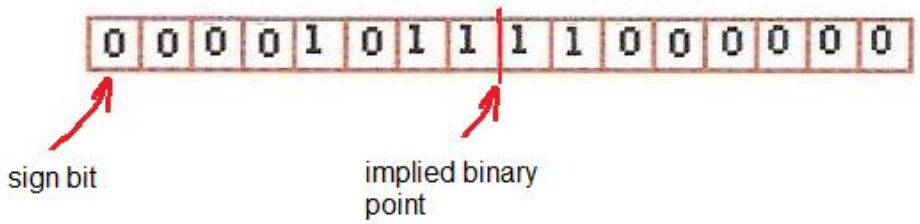
we can use two ways to represent the real numbers:-

a) Fixed point representation:-

If the word has (16 bit) it is possible to store mixed numbers(integer part & fractional part) in a single word as long as there is an agreed number of bits for the integer part. The position of the binary point is implied. If we have the value:-

$(11.75)_{10}$ ➔ $(01011.110000)_2$

It's representation will be as :-



If we use double length working to hold the real numbers, it is usual one for the integral part and the other for the fractional part.

b) Floating point representation:-

The fixed-point representation gives us a good precision but it is not enough for some scientific values like the electron weight or the distance between the sun and the earth ...etc. so to solve such a problem we will use the floating point representation to represent the real values.

Computers which work with real arithmetic use a system called floating point. In computing floating point (FP) describes a system for representing real numbers which supports a wide range of values. Declaration in C++:

Float x;

Ex:

$37.2510 = 100101.01_2 = 1.0010101 \times 2^5$

$7.62510 = 111.101_2 = 1.11101 \times 2^2$

$0.312510 = 0.0101_2 = 1.01 \times 2^{-2}$

3. Characters:

It's declaration in C++ :

```
char     a, b, c;
```

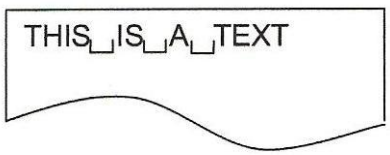
The character set defined by the International Standards Organization (ISO), and particularly its American version ASCII (American Standard Code for Information Interchange) is the most widely accepted set.

The tabulated of ASCII code consists of 95 printable (graphic) characters and 33 control characters, the latter mainly being used in data transmission and for the control of printing equipment.

In other words, we should like to be able to assume certain minimal properties of character sets, namely:

1. The type CHAR contains the 26 capital Latin letters, the 26 lower-case letters, the 10 decimal digits, and a number of other graphic characters.

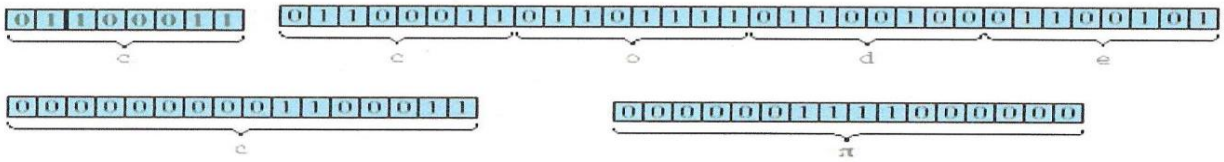
2. The type CHAR contains a non-printing, blank character and a line-end character that may be used as separators.



Representations of a text

Store numeric codes (ASCII, EBCDIC, Unicode) in (1 byte) for ASCII and EBCDIC, (2 bytes) for Unicode (see the following examples).

Basic operation : **comparison** to determine if =, <, > etc. --- use their numeric codes.



4. Strings:

It is a series of characters stored in a contiguous area and grouped together logically. It may be of any length including zero length.

String are stored in the memory into two ways:-

a. Fixed-length locations.
b. Variable –length locations.

In C arrays of characters are used to store and represent strings, which must be null-terminated ('\0'). The *string* type in C++ offers facilities that make string manipulation much easier and intuitive, for example it offers a wealth of string operations that are very easy to use.

5. Boolean (Logical) :-

The ***bool*** type represent Boolean (logical) values, for which the reserved keywords **true** and **false** may be used.

```
bool  flag = false;
```

6. Pointers:

It is a location in the memory contains the address of the storing of a specific value. Reference or access the data object through the pointer variable that points to it.

The computer memory is nothing but a sequence of cells the size of each one byte, and each one of these cells has a distinct address where the operating system addresses the memory with sequential numbers. The pointer is a variable like all other variables, but it differs from it in what it stores. It does not store regular data such as numbers and symbols, but rather the address of the variable that it refers to.

The address parameter (&)

When a variable is recommended, it will be stored in one of the cells available in the memory automatically by the compiler and the operating system. If we want to know where to store this variable, we precede the variable name with the symbol & .

For example, the instruction

```
a = &b
```

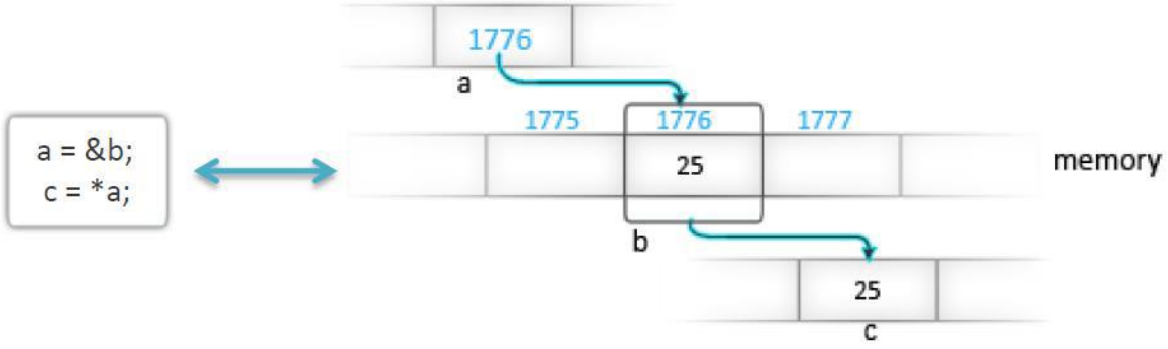
would put the address of variable b in variable a. The variable that contains the address of another variable is called a pointer .

The reference parameter (*)

If we precede the pointer a by the reference factor *, that is, if we write *a then the expression;

```
c = *a
```

Means place the value that referred to by the pointer a in the variable c.



Note the difference between the two statements
c = a; // c == 1776
c = * a; // c = = 25
From the above, the following statements are true:
b == 25
& b == 1776
a == 1776
* a == 25
* a == b

Pointer's declaration:

The general form for the declaration is:

type * pointer_name;

type: is the type of data the cursor will refer to.
pointer _ name: the name of the pointer.

Examples:
int * number;
char * character;
float * greatnumber;

We have three pointers, each of which refers to a different type of data, however, each one of these pointers reserves the same size in memory according to the

operating system. As for the data that these pointers refer to, they have different sizes in memory and are of different types.

Benefits of pointers: –

1. pointers are more efficient in handling arrays & data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. pointers permit references to functions & there by facilitating passing of functions as arguments to other functions.
4. The use of pointers arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C++ to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures. Such as structures, linked lists, queues, stacks & trees.
7. Pointers reduce length & complexity of programs.
8. They increase the execution speed & thus reduce the program execution time.

Now have a look at this code:

// my first pointer
#include <iostream>
using namespace std;
int main ()
{
int firstvalue, secondvalue;
int * mypointer;
mypointer = &firstvalue;
*mypointer = 10;
mypointer = &secondvalue;
*mypointer = 20;
cout << "firstvalue is " << firstvalue << endl;
cout << "secondvalue is " << secondvalue << endl;
return 0;
}
	Program result:
	
	firstvalue is 10
	secondvalue is 20

Notice that even though we have never directly set a value to either firstvalue or secondvalue, both end up with a value set indirectly through the use of mypointer. This is the procedure:

First, we have assigned as value of mypointer a reference to firstvalue using the reference operator (&). And then we have assigned the value 10 to the memory location pointed by mypointer, that because at this moment is pointing to the memory location of firstvalue, this in fact modifies the value of firstvalue.

Non Primitive Data Structure:

A) Linear Data Structure:

A data structure is said to be linear if it is elements form a sequence or a linear list. There are two basic ways of representing such linear structures in memory:

1) One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called Arrays.
2) The other way is to have the linear relationship between the elements represented by means of Pointers or Links. These structures are called Linked Lists.

The operations are normally performs on any linear structure, whether it is an array or a linked list, include the following:-

1) Traversing: It is the operation of passing on the elements of the structure.
2) Searching: it is the operation of trying to find an element in the structure, and locating its position.

3) Insertion: It is the operation of inserting an element to the structure elements.
4) Deletion: It is the operation of erasing an element from the structure elements.
5) Sorting: It is the operation of arranging the elements of the structure in an ascending or descending manner.
6) Merging: It is the operation of joining two structures in one structure.

1) Linear Arrays or (One Dimensional Arrays):

It is a list of finite number (N) of homogeneous data elements such that:-

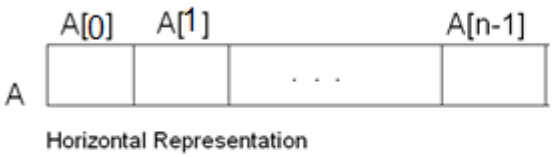
a) The elements of the array are referenced respectively by an index set consisting of (N) consecutive numbers.
b) The elements of the array are stored respectively in successive memory locations.
c) (N) denotes the length or the size of the array :

Length = UpperBoundary – LowerBoundary

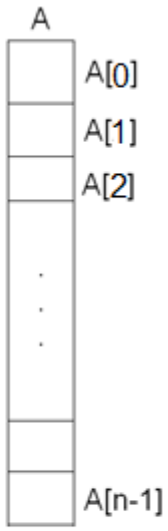
d) The elements of an array A may be denoted by: A0, A1,…,An-1 =====➔ or A[0], A[1], …, A[n-1]
e) The array subscript operator Is an integer such that

$$0 \leq i \leq n-1$$

f) The array is pictured as the following:-



Horizontal Representation



Vertical Representation

g) The declaration of the array must give three items of information:-

1) The name of the array,
2) The data type,
3) The index set of the array.

Declaring One-Dimensional Arrays in C++ :

element_type *array_name* [CAPACITY];

Where
element_type is any type,
array_name is the name of the array --- any valid identifier.
CAPACITY (a positive integer constant) is the number of elements in the array.

Example:-

```
int  x[7];
```

Accessing Linear Array in Memory:-

The memory of the computer is simply a sequence of addressed locations as pictured previously , and that the computer has direct access to the elements of a memory locations when the address of the memory cell is known. The computer uses the following formula to calculate the address of any element in the linear array named as example (A):-

$$Loc(A[k]) = Base(A) + W\,(k)$$

Where :-

Loc(A[k]) = the address of the element A[k] of the array A.

Base(A) = the address of the first element of the array A.

W = number of words per memory cell for the array A.

K = the index of the element.

Example (1):-

Consider the array (A) which Records the degrees of a (50) students in data structure and the base address is (200) and each element is saved in a memory cell having (4) words per memory cell, then find the address of the element (30).

Loc(A[k]) = Base(A) + W (k)

Loc(A[30]) = 200 + 4 (30)

Loc(A[30]) = 320

Example (2):-

If you have the one dimensional array X(15), the X(8) elements address is (186) , and the base address to the array is (170), then find the number of the bytes used to store every element in the memory?

2) Two Dimensional Arrays: or (Matrices)

A two dimensional (M x N) array is a collection of (m . n) data elements such that each element is specified by a pair of integers (such as i, j) called subscripts with the property that:-

$$0 \leq i \leq m-1 \quad and \quad 0 \leq j \leq n-1$$

$$A\ i,j \quad or \quad A[i][j]$$

Declaring Tow-Dimensional Arrays in C++ :

a- Usual form of declaration :

element_type array_name[NUM_ROWS] [NUM_COLUMNS] ;

Example :

```
Double    scoresTable [NUM_ROWS][NUM_COLUMNS] ;
```

b- Initializing :

List the initial values in braces, row by row; may use internal braces for each row to improve readability.

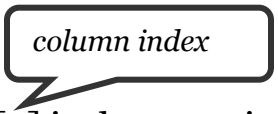
Example :

```
Double  rates[2][3] = {{0.50, 0.55, 0.53},          // first row
                       {0.63, 0.58, 0.55} } ;   // second row
```

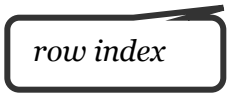
Processing tow-dimensional arrays :

Use doubly-indexed variables :

Example :



scoresTable [2] [3] is the entry in row 2 (numbered from 0)



and column 3 (numbered from 0)

Typically use nested loops to vary the two indices, most often in a row wise manner.

Representation of Two Dimensional Arrays:

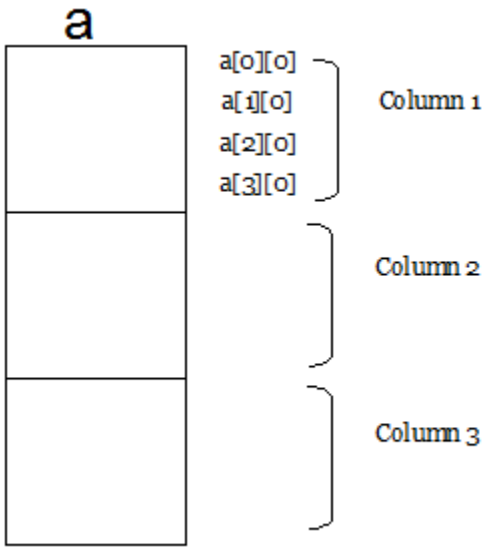
The array will be represented in the memory by a block of (m x n) sequential memory locations the programming language will store the array (A) either in :-

- Column by Column, is what is called ***Column-Major-order*.**
- Or Row by Row , or ***Row-major-order***.

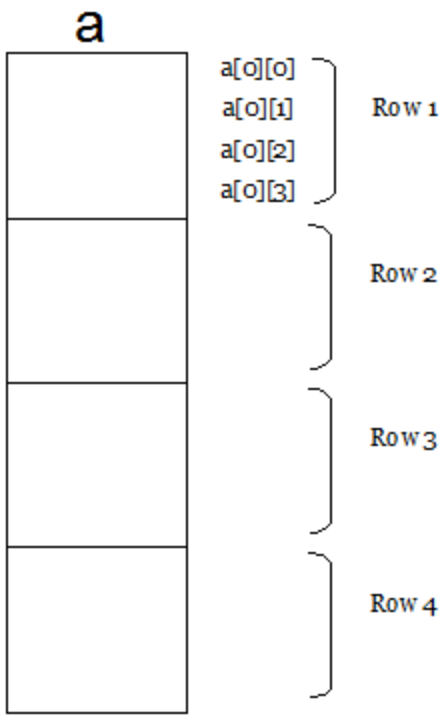
If we take the matrix A(4,3) we can represent it as the following:-

$$a[4][3] = \begin{pmatrix} a00 & a01 & a02 \\ a10 & a11 & a12 \\ a20 & a21 & a22 \\ a30 & a31 & a32 \end{pmatrix}$$

The array representation in ***Column-major order*** :-



The matrix representation in ***Row-major order*** :-



We can compute the address of any element in tow-dimensional array like (a[i][j]), by using one of the following formulas :-

1- Column-Major-order :-

$$\text{Loc}(a[i][j]) = \text{base address}(a) + w[m(j)+(i)]$$

2- Row-Major-order :-

$$\text{Loc}(a[i][j]) = \text{base address}(a) + w[n(i)+(j)]$$

Example :-

Consider the array f[25][4], suppose that the base address of (f) is (200) and there are (4) words per memory cell to save each element of the array, use the *row-major-order* to find the address of the element f[11][2].

$$\text{Loc}(f[i][j]) = \text{base address}(f) + w[n(i)+(j)]$$

$$\text{Loc}(f[11][2]) = 200 + 4 * [4 * (11) + (2)]$$

$$= 200 + 4* [46]$$

$$= 384$$

Example (2):-

Compute the address of the element F[5][2] if we have a column-major order method?

Example (3):-

Compute the address of the element A[12][15] of the two dimensional array A[20][22] , and the size of every element is (2) words (use the column-major order then the row –major-order), the base (a) is (80)?

Application of Arrays :-

We can use the arrays in most of the operations of the computer, such as searching, sorting and representing of the stack, queue and tree.

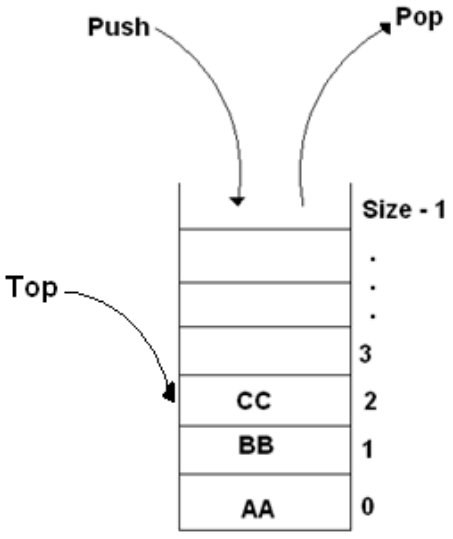
Stacks:

A stack is a linear structure (list of elements) in which items may be inserted or deleted only at one end, called the Top of the stack, that is the last item to be added to the stack is the first item to be removed, thus the stack are also called (Last – In – First – Out) lists or (LIFO).

A stack is represented in the memory by means of an array and a pointer to the top element of the stack.

There are two operations associated with stacks:-

1- Push ➔ to insert an item to the stack.
2- Pop ➔ to delete an item from the stack.



Stack Behavior :-

- The variable Top is the number of the first element in the stack.
- A stack is **empty** when **(top=-1)** and it's **full** when **(top=size-1)**, so the initial value to the **top** is **(-1)** and top value is always less than the real number of elements in the stack by **1.**

- A **stack underflow** happens when one tries to **pop** on an **empty stack.**
- A **stack overflow** happens when one tries to **push** onto a **full stack.**

Algorithms of the stack:-

There are two main algorithms to manipulating stacks but also there are two other algorithms to update and reach any element in the stack.

1- Push Algorithm:

An attempt to add an item to a full stack causes overflow.

```
Procedure Push(item)
        Global Stack, Top, Size
if    Top = size – 1     then
        output('Stack Overflow')
else
        Top ← top + 1
        Stack[Top] ← item
End if
End Push
```

2- Pop Algorithm:

An attempt to remove an item from an empty stack causes underflow.

```
Procedure Pop(item)
        Global Stack, Top, Size
if    Top =  – 1     then
        output('Stack underflow')
else
```

```
        item ← Stack[Top]
        Stack[Top] ← 0    or    blank
        Top ← top - 1

End if
End Pop
```

3- Procedure Update:

This procedure changes the value of the (ith) element from the top of the stack to the value contained in X.

```
Procedure Update(Stack, Top, I, X)
        If   (Top – I + 1 < 0)   then
                Output('Stack underflow')
        Else
                Stack[Top– I + 1] ← X
        End If
End Update
```

4- Function Peep:

This function returns the value of the(ith) element from the top of the stack. The element is not deleted by this function.

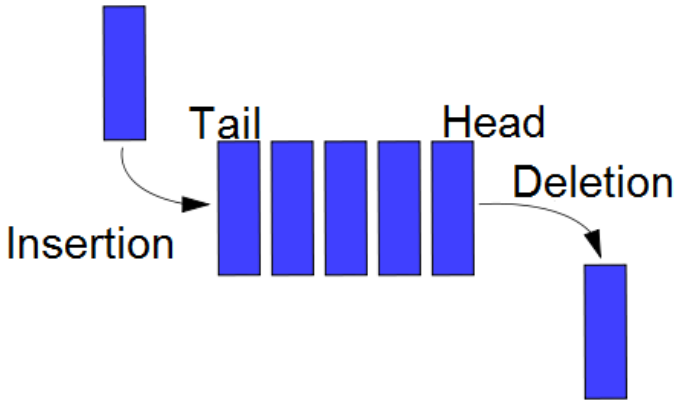
```
Function Peep(Stack, Top, I)

        If   (Top – I + 1 < 0)   then
                Output('Stack underflow')
        Else
                Peep ← Stack[Top– I + 1]
        End If
End Peep
```

Application of Stacks:

1- Reversal of inputs.
2- Converting a decimal number into a binary number.
3- Using stacks to execute any main program with its own subprograms(execution – time – stack) to store return addresses during subprograms executions.
4- Using stack to execute recursive algorithms.
5- Using stacks to evaluate arithmetic expressions.
6- to control the transfer of data into or out of a computer.
7- The undo-mechanism in an editor.
8- Back & forward buttons in a Web browser.

Queues :-

The queue is a linear list of elements where items are deleted at one end (called the "***front***" or ***"head"*** of queue) and inserted at the other end (called the "***rear"*** or "***tail"*** of the queue). The first element to be inserted into the queue will be the first to be removed. Thus queues are sometimes referred to as ***F***irst ***I***n ***F***irst ***O***ut (FIFO) lists.



Queues are found in everyday life, the automobiles waiting to pass through an intersection form a queue, an important example of a queue in computer science occurs in a time sharing system or a queue of jobs waiting to be printed on a printer.

There are three kinds of queues :-

1- Simple (ordinary) queue.

2- Circular queue. (Ring)

3- Double-ended-queue (Deque).

Queue Representation:-

We can represent the queue by means of Linear array and two pointers head and tail, or by Linked – lists and Pointers.

Applications of Queues :-

1- In a multitasking operating system, the CPU time is shared between multiple processes. At a given time, only one process is running, all the others are "sleeping". The CPU time is administered by the scheduler. The scheduler keeps all current processes in a queue with the active process at the front of the queue.
2- Printer queue: When files are submitted to a printer, they are placed in the printer queue.
3- Using a queue in I/O buffers like read the data from a file in C++.

1- Simple (ordinary) queue :-

The queue is a linear list that permits the insertion to be performed at one end and the deletion on the other end. If the head element is deleted and the queue is full the elements must be moved to the first cell of the queue.

Algorithm for Insertion :-

```
Procedure  Ordinary-Queue-Insertion(queue, head, tail, item, size)
    If  tail-head+1 = size then
        Output(" Queue is Full – Overflow")
    Else
        If   head = -1   then
            Head ← 0
            Tail ← -1
        Else
            If   tail = size-1   then
                For    i ← 0 to (tail – head) do
                    Queue[i] ← queue[head + i]
                Endfor
                Tail ← size – head -1
                Head ← 0
```

```
            Endif
        Endif
        Tail ← tail +1
        Queue[tail] ← item

    Endif
End procedure
```

Algorithm for Deletion :-

```
Procedure  Ordinary-Queue-Deletion(queue, head, tail, size)
    If  head = -1 then
        Output(" Queue is Empty – Underflow")
    Else
        Output(" Queue  head is Deleted ")
        Queue[head] ← " "  or 0
        If   head +1 > tail   then
            Head ←  -1
            Tail ←  -1
        Else
```

```
            head ← head +1
        Endif
    Endif
End procedure
```

2- ***Circular queue :- (Ring)***

Removing an element from the queue is an expensive because all remaining elements have to be moved by one position. A more efficient implementation is obtained if we consider the array as being "circular".

It is like the ordinary queue having two ends one for insertion and the other for deletion. It is represented by a linear array and two pointers head and tail, or by linked lists and pointers.

Algorithm for Insertion :-

```
Procedure Circular-Queue-Insertion(queue, head, tail, item, size)
    If (tail-head+1 = 0 or tail-head+1 = size) then
        Output(" Queue is Full – Overflow")
    Else
        If head = -1 then
            Head ← 0
            Tail ← -1
```

```
            Else
                If    tail = size-1 then
                    Tail ← -1
                Endif
            Endif
            Tail ← tail +1
            Queue[tail] ← item

        Endif
End procedure
```

Algorithm for Deletion :-

```
Procedure  Cicular-Queue-Deletion(queue, head, tail, size)
        If  head = -1 then
            Output(" Queue is Empty – Underflow")
        Else
            Output(" Queue  head is Deleted ")
            Queue[head] ← " "  or 0
            If    head =  tail    then
                Head ←  -1
```

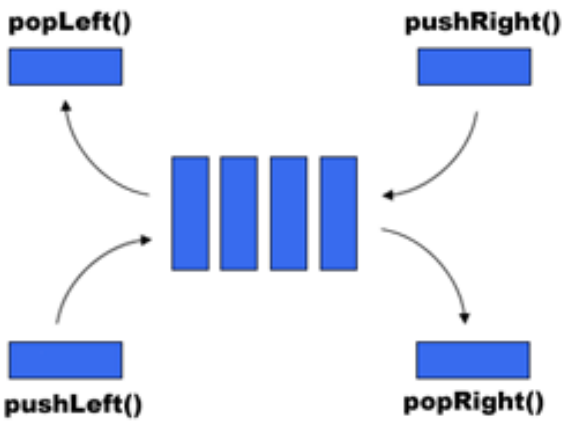
```
            Tail ← -1
        Else
            head ← head +1
        Endif
        If  head >= size  then
            Head ← 0
        Endif
    Endif
End procedure
```

3- <u>Double – Ended - queue :- (Deque)</u>

Is a linear list in which insertion and deletion are made from either the front or the back of the queue. It is more general than the stack and queue. The double ended queue is called ***Deque*** (***D****ouble – **E**nded – **que**ue*)

There are two variations of deque ; the input-restricted deque (allow insertion at only one end) and the output –restricted deque (allow deletion at one end).

An alternative way to picture the deque is as two stacks joined together at the base. The two stacks will use the same memory area.



There are at least two common ways to efficiently implement a deque: with a modified ***dynamic array*** or with a ***doubly linked list***.

Applications

1- deque can be used the algorithm of implementing task scheduling for several processors.
2- Simulation (as in military operations)
3- Time sharing

Disadvantages of Arrays :-

1- The size of the array is fixed.

1- Because of the fixed size, the most convenient thing for programmers to do is to allocate arrays which seem "large enough".
2- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. As we will see, linked lists allocate memory for each element separately and only when necessary.

Dynamic and Static Memory

1. ***Static memory allocation*** refers to the process of allocating memory at compile-time before the associated program is executed.

   An application of this technique involves a program module (e.g. function or subroutine) declaring static data locally, such that these data are inaccessible in other modules unless references to it are passed as parameters or returned. A single copy of static data is retained and accessible through many calls to the function in which it is declared.

2. ***Dynamic memory allocation*** refers to the process of allocating memory as required at run-time, and under the control of the program.

   Dynamically allocated memory exists until it is released either explicitly by the programmer or by the garbage collector, it is said that an object so allocated has a dynamic lifetime.

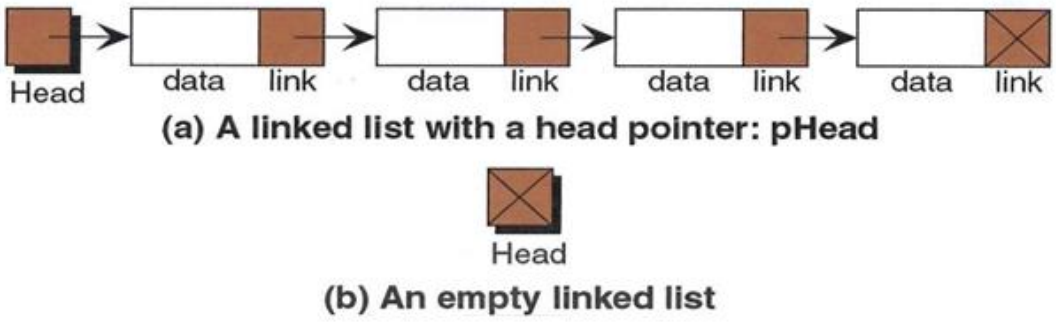
   Usually, memory is allocated from a large pool of unused memory area called The ***heap*** which is an area of memory that is dynamically allocated, (also called the free store). Since the precise location of the allocation is not known in advance, the memory is accessed indirectly, usually via a reference.

garbage collection (GC)

In computer science, garbage collection (GC) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve the problems of manual memory management in Lisp programming language.

Linear Linked Lists (L. L. L.) :-

A linked list is one of fundamental data structures, and can be used to implement other data structures. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next (and/or) previous nodes. The last node has a null at the end. A 'stopping' is needed at the end, when running through the list.



(a) A linked list with a head pointer: pHead

(b) An empty linked list

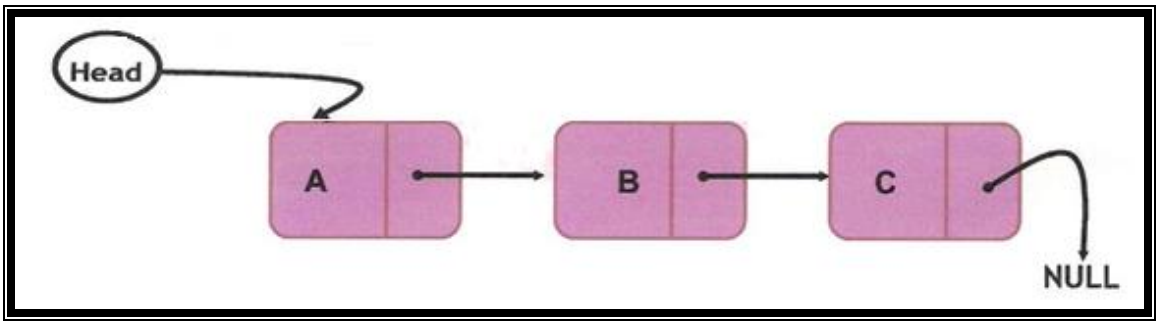
Types of linked lists :-

There are three kinds of linked lists :-

- Singly Linked List
- Circularly Linked List.
- Doubly Linked List

1- Simple Linked List :- (Single)

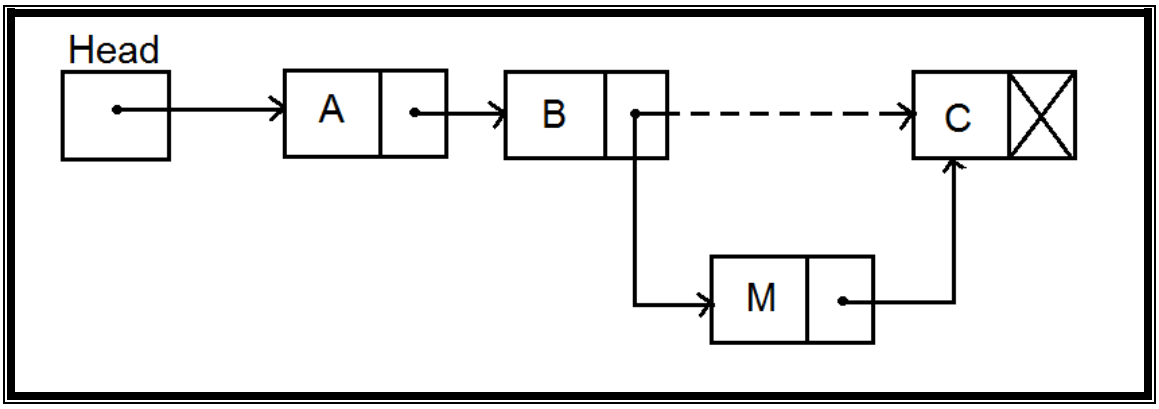
It is a sequence of nodes in which each node linked to the following node by a pointer.



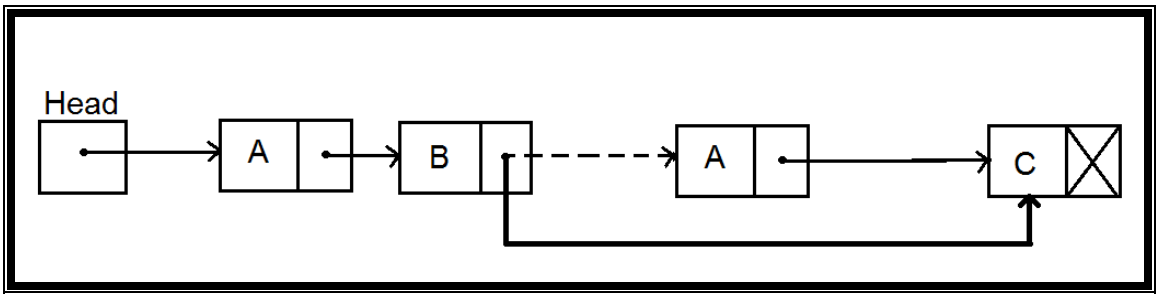
Each node in the list has two fields:-

1- The data field.
2- A pointer represented by an arrow pointed to the next list element (node).

Linked lists are important data structure because they can be modified easily , for example a new node can be inserted easily.



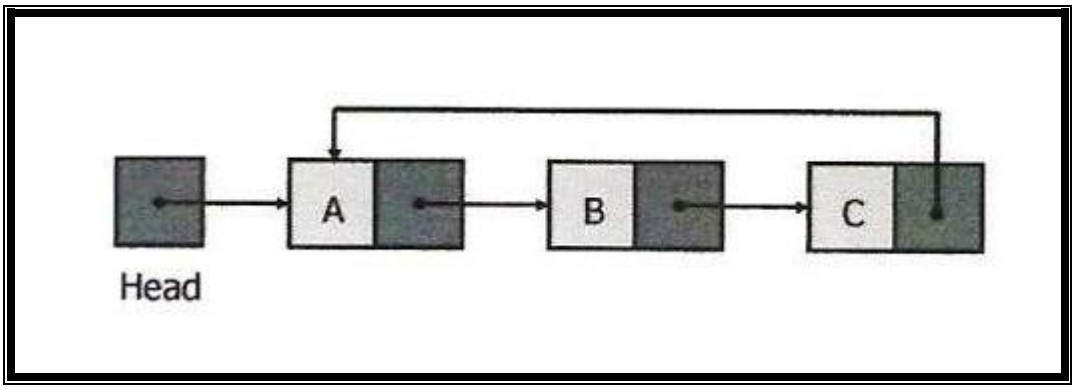
Similarly it is quite easy to delete a list node. One pointer value has to be changed . the following figure illustrates the idea:-



2- Circularly Linked List :- (Ring)

An important limitation of the simple linked list is that one can only go forward in it, in the direction of the links. There is no provision for moving in the other direction.

A simple solution is to provide a link from the last node to the header as shown in the figure:-



The result list is called a ring. We can still only move in one direction in a ring but at least when we get to the end we can go back to the beginning and examine those nodes preceding the one we started with. specifically, we can start the search of a ring at any node , instead of just at the header.

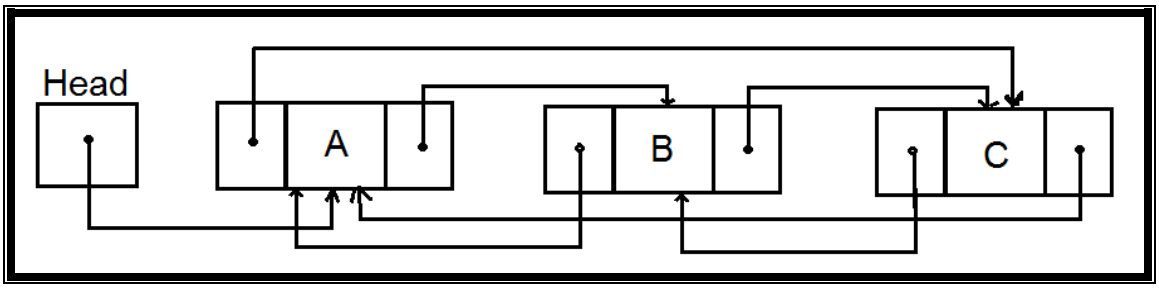
3- Doubly Linked Lists :-

A more generally useful solution to the problem of the one- way nature of the simple linked list is to include two links in each node.

A node in a doubly linked list has at least three fields :-

- A left link field
- A data filed
- A right link field

In doubly linked list , we can move either direction with ease, on the other hand, a doubly linked list requires more memory than either the simple linked list or a ring, since each node must contain two link components instead of one. However, a doubly linked lists are a little slower than singly linked lists. The following figure will show the doubly linked list:-



Note : that a doubly linked list is also a ring, since the header and the last element are linked.

Applications of Linear Linked Lists :-

We can use the linear linked list to implement the operations of the stack and the queue if we think of the stack or the queue as a linked list.

We may also use the doubly linked lists to represent the general trees and the binary trees.

Advantages of Linear Linked List :-

1. Memory is allocated when the program is run, therefore the data structure is only as big as it needs to be.
2. Memory is conserved.

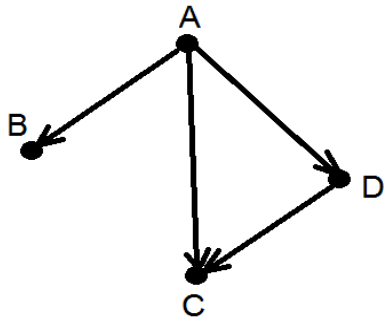
Disadvantages of Linear Linked Lists :-

1. Each node of the list takes more memory.
2. Processing is slower.
3. The data structure is not random access.
4. Processing must be done in sequential order.

Non Linear Data Structures :-

1- Graphs :-

It is a set of points connected by lines, as shown in the following figure:



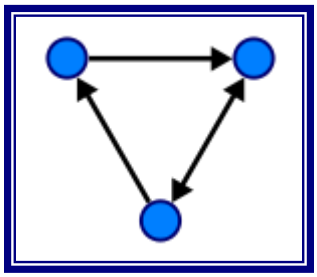
It is also called Network, map, and flowchart. A graph is a data structure consisting of:

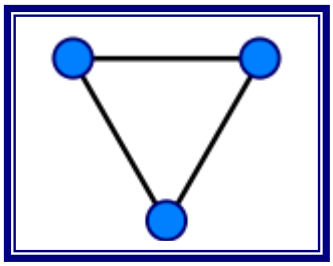
- a set of *vertices* (or nodes).
- a set of *edges* (or links) connecting the vertices (or nodes).

The two vertices are called the edge *endpoints*. Graphs are ubiquitous in computer science. The most important use of a graph is to show the relations among entities. They are used to model real-world systems such as the ***Internet*** (each node represents a router and each edge represents a connection between routers); ***airline connections*** (each node is an airport and each edge is a flight); or a ***city road network*** (each node represents an intersection and each edge represents a block). The wireframe drawings in computer graphics are another example of graphs.

A graph may be either *undirected* or *directed(digraph)*. Intuitively, an undirected edge models a "two-way" or "duplex" connection between its endpoints, while a directed edge is a one-way connection, and is typically drawn as an arrow. A directed edge is often called an *arc*. Mathematically, an undirected edge is an unordered pair of vertices, and an arc is an ordered pair. For example, a road network might be modeled as a directed graph, with one-way streets indicated by an arrow between endpoints in the appropriate direction, and two-way streets shown by a pair of parallel directed edges going both directions between the endpoints.

Directed Graph (digraph)





Undirected Graph

A **path** in a directed graph **must follow the direction of the arrows**, ***or***, it's a sequence of vertices in which each successive pair is an edge. A **cycle** is a path in which the first and last vertex are the same and there are no repeated edges.

Applications of Graphs :-

- Electronic circuits :-
  - Printed circuit board
  - Integrated circuit
- Transportation networks :-

- Highway network
- Flight network

- Computer networks :-
- Local area network
- Internet
- Web

- Databases :-
  - Entity-relationship diagram

Graph Algorithms :-

Graph algorithms are include :-

- Searching for a path between two nodes :- can be used in game playing, AI, route finding..
- Finding shortest path between two nodes.
- Finding a possible *ordering* of nodes given some constraints. e.g., finding order of modules to take; order of actions to complete a task.

Graph Representation :-

One of the methods to represent the graphs is the Adjacency matrix method.

Adjacency matrix method :-

Use (N × N) array of Boolean values:

	0	1	2	3
0	F	T	T	F
1	T	F	T	F
2	T	T	F	T
3	F	F	T	F

(or can just use integer, and 1/0)

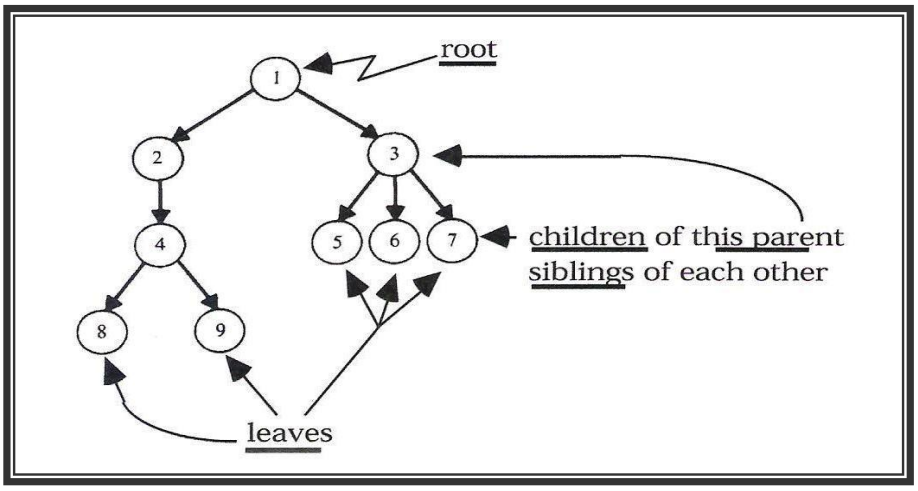
If array name is G, then G[n][m] = T if edge exists between node n and node m.

2- Trees :-

A **tree** consists of a finite set of elements called ***nodes*** (or ***vertices***) and a finite set of ***directed arcs*** that connect pairs of nodes. If the tree is not empty, then one of the nodes, called the ***root***, has no incoming arcs, but every other node in the tree can be reached from the root by a unique path (a sequence of consecutive arcs).

A ***leaf*** is a node with no outgoing arcs. Nodes directly accessible (using one arc) from a node are called the ***children*** of that node, which is called the ***parent*** of these children; these nodes are ***siblings*** of each other.

If we remove (delete) the root of the tree we get a ***forest*** which is a set of disjoint trees.

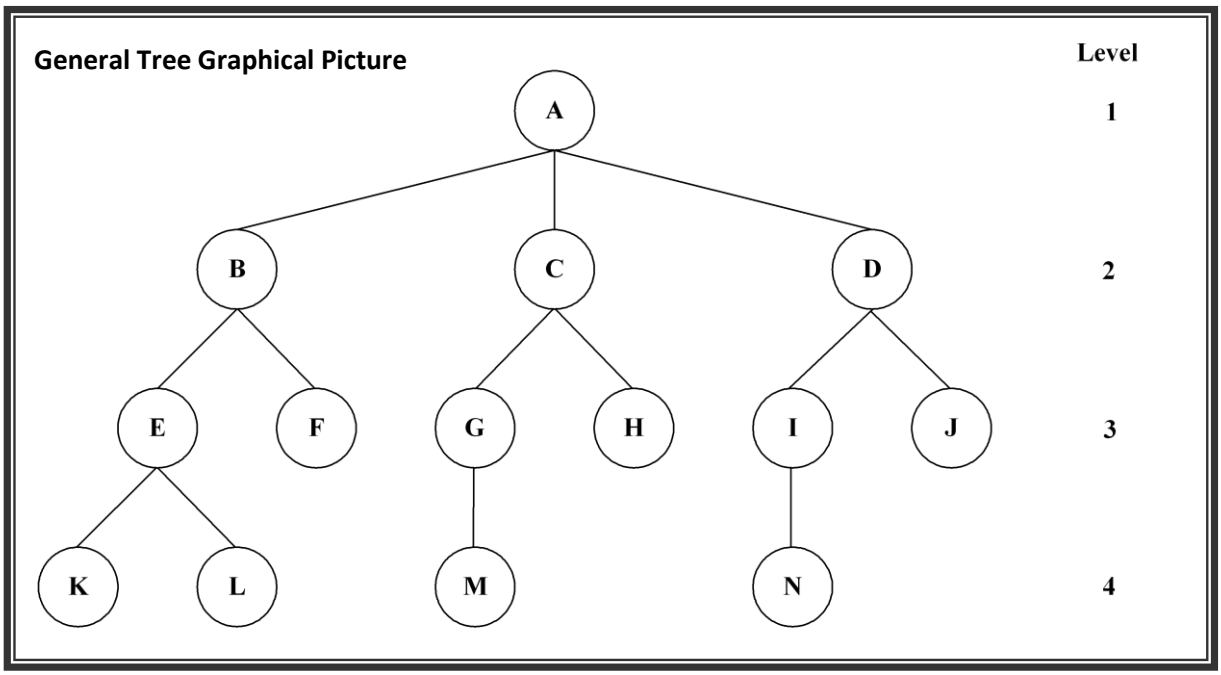


Important terms :-

1- The **depth of a node** is the length of the path (or the number of edges) from the root to that node.
2- The **height of a node** is the longest path from that node to its leaves. The height of a tree is the height of the root.
3- The **degree of a node** is the number of subtrees of the node.
4- The **degree of a tree** is the maximum degree of the nodes in the tree
5- The **ancestors of the node** are all the nodes along the path from the root to that node.
6- The **depth of a tree** it also called **height** of a tree. It is the maximum level of the tree.

There are two kinds of trees :-

1. General trees.
2. Binary trees.



1- General tree in which the number of subtrees for any node is not required to be 0, 1, or 2. The tree may be highly structured and therefore have 3 subtrees per node in which case it is called a ***ternary*** tree. However, it is often the case that the number of subtrees for any node may be variable. Some nodes may have 1 or no subtrees, others may have 3, some 4, or any other combination. The ternary tree is just a special case of a general tree (as is true of the ***binary*** tree).

2- Binary tree is a tree in which each node has at most 2 children. Each Node has zero, one, or two children. This assertion makes many tree operations simple and efficient.

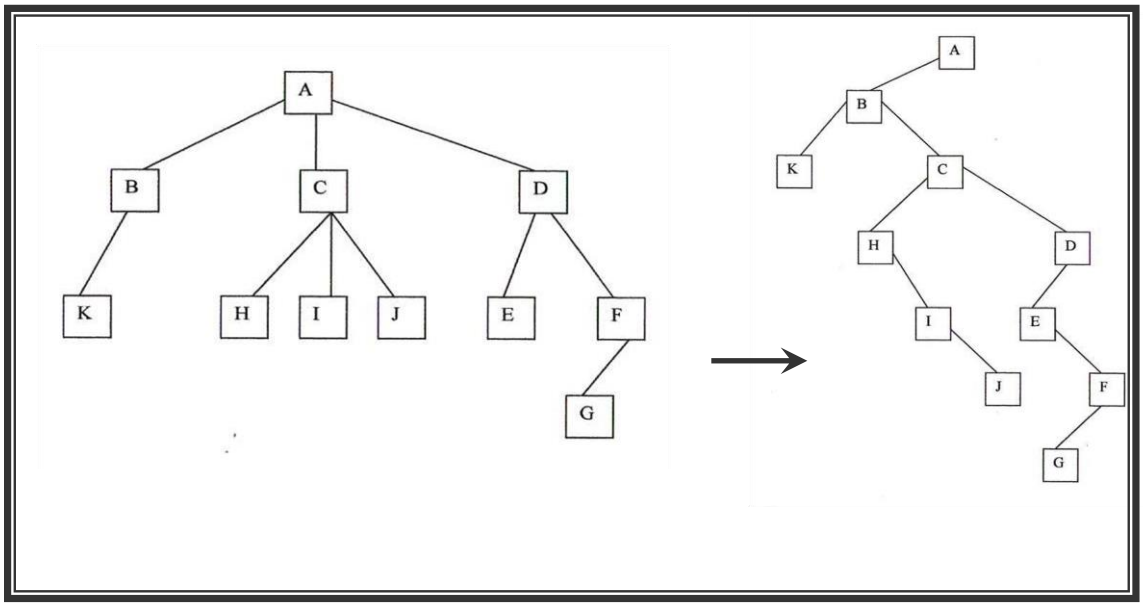
Creating a Binary Tree from a General Tree :-

The process of converting the general tree to a binary tree is as follows:

1. Use the root of the general tree as the root of the binary tree

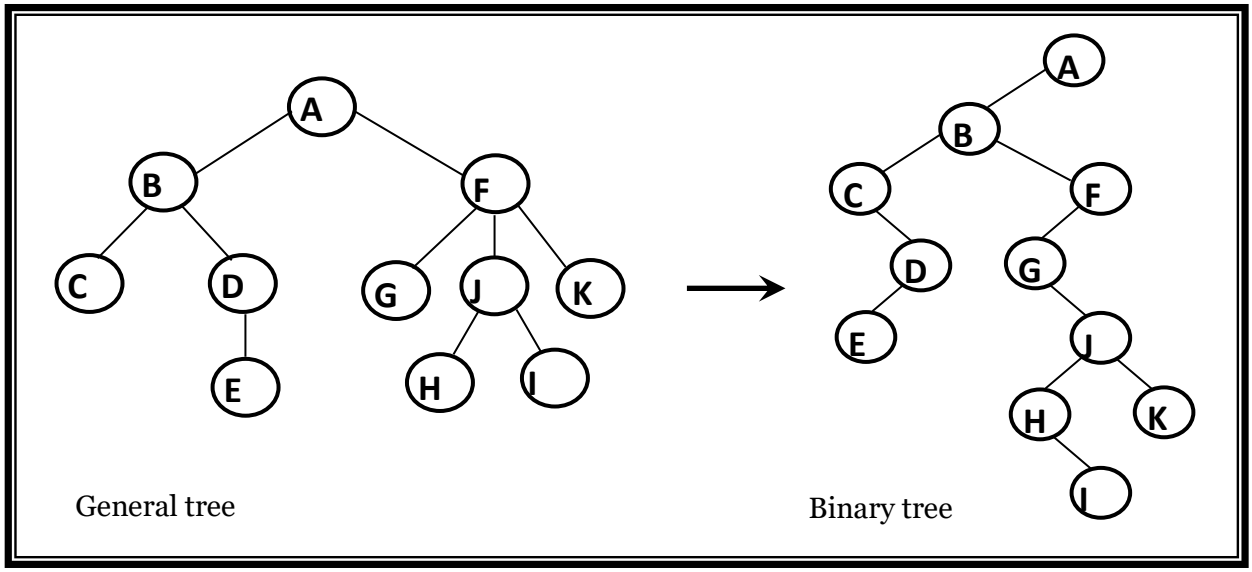
2. Find the root's children (left & right) by the following steps :-
    a. Determine the first child of the root, (this is the leftmost node in the general tree at the next level).
    b. Insert this node in binary tree (consider it the left child to the root in binary tree).
    c. Determine the first sibling (to the first child of the root in general tree) then consider it the right child to the left child in binary tree.
    d. Continue finding the other siblings (to the first child of the root in general tree) in successive manner, and consider each one as the right child to his previous sibling.
3. Consider the left child in (b) as a root then repeat the above process to find the left and right child to it.
4. Stopping is done when we complete visit all nodes in general tree,(in other word) the number of nodes in binary tree which we created it equal to number of nodes in general tree.

EX1 :- ***Convert the following general tree to a binary tree***



General tree Binary tree

EX2 :-

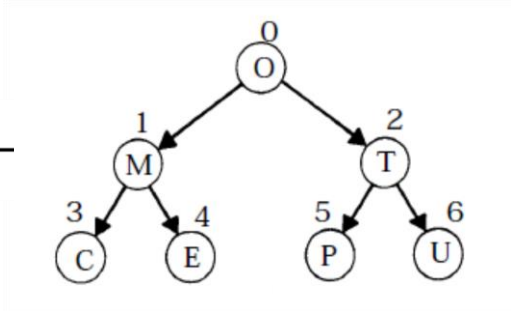


Representing of Trees :-

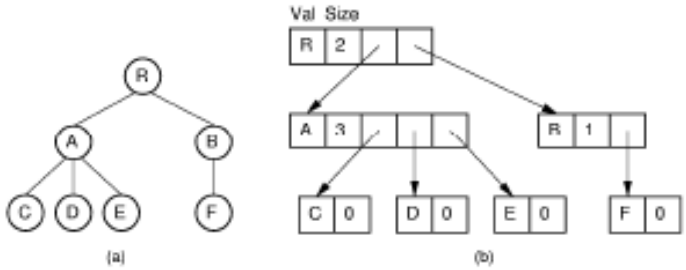
1. *Linear representation, using array-based implementation :-*

   An array can be used to store trees. We just number the nodes level by level, from left to right and store nodes sequentially. (Very efficient for Binary trees).

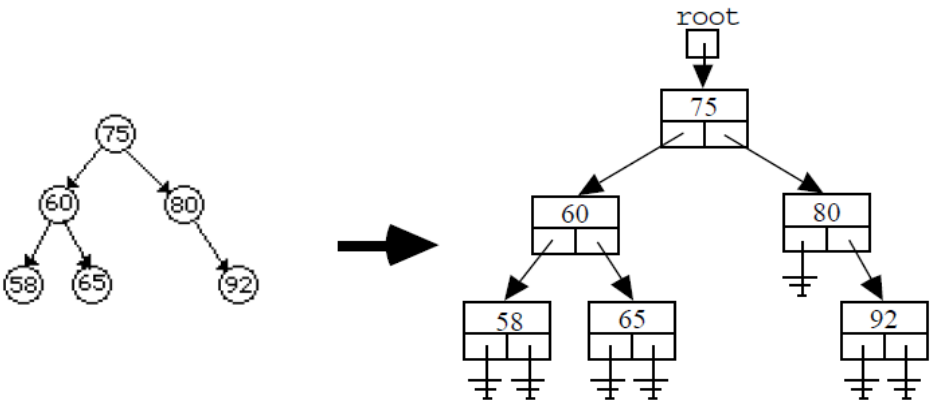
i	0	1	2	3	4	5	6
T[*i*]	O	M	T	C	E	P	U



2. *Linked representation, using pointers as :-*
    a. A node with multiple pointers (to represent the general tree)



    b. to overcome the problem of the wasted space by using two pointers only in each node, one to the left child , and the other to the right child.(representing Binary Trees)



Tree Traversals Methods :-

Many problems require we visit the nodes of a tree in a systematic way: tasks such as counting how many nodes exist or finding the maximum element. There are three types of binary tree traversal :-

1. Preorder traversal : Current node, left subtree, right subtree (NLR).

2. Inorder traversal : Left subtree, current node, right subtree (LNR).

3. Postorder traversal : Left subtree, right subtree, current node (LRN).

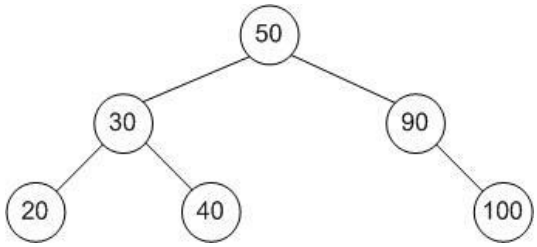
1. A preorder traversal visits the root of a subtree, then the left and right subtrees recursively.
2. An inorder traversal visits the left subtree, the root of a subtree, and then the right subtree recursively.

3. A postorder traversal visits the left and right subtrees recursively, then the root node of the subtree.

Recursion is inherent in tree traversal, and all of the traversal algorithms are written as recursive procedures. We should always check for a NULL left or right child pointer, since some nodes may have only one child node, and leaf nodes have no children nodes.

<u>Examples of Tree Traversals</u> :-

Preorder (NLR): 50, 30, 20, 40, 90, 100
Inorder (LNR): 20, 30, 40, 50, 90, 100
Postorder (LRN): 20, 40, 30, 100, 90, 50



Applications of Trees :-

We see the tree structure in many contexts outside of computer science. The most familiar examples are :-

- Family tree.
- Company organization chart.
- Table of contents in the books.
- *Biological classifications*.

Tree data structures are evident in many software applications. For example :-

1. Search algorithms rely on tree structures to order data so that sub-linear search times are achievable. For example (Implementation of *binary search tree* in search algorithm).
2. Data compression algorithms require that encoded data be uniquely decodable. In Huffman coding, a code is constructed as a *binary coding tree* that defines an unique *prefix code*, which guarantees unique decodable.
3. Many databases use a *B- tree* data structure that provides very fast keyed access to records stored on disk.
4. Operating systems provide a hierarchical tree structured file system to allow users to organize their files into directories (folders). The hierarchical tree structure guarantees a unique name for each directory or file.
5. In computer games where the current position is the root of the tree; the branches lead to positions that might occur later in the game.
6. Language compilers (like C++) use tree data structures as part of the parsing phase of translation to machine code. For example, arithmetic expressions are represented using a *binary expression tree* that captures the precedence of arithmetic operators.