MODULE - 1

⇒ ALGORITHM :- An algorithm is a sequence of an unambiguous instructions for solving a problem.

In addition, all algorithm must satisfy the following criteria. They are:

1. INPUT : Each & every algorithm must have 0, 1 or more inputs.
2. OUTPUT : The algorithm should produce correct result.
   Atleast 1 output has to be produced.
3. DEFINITENESS : Each programming instructions should be clear & unambiguous.
4. FINITENESS : The agorithm must terminate after a finite number of steps.
5. EFFECTIVENESS : Every instruction must be very basic statement so that it can be easily executed by a person using only pen & paper.

⇒ ALGORITHM SPECIFICATION :-

Algorithm can be described in 3 ways:

1. Natural language like English.
2. Graphic representation called flowchart

3. Pseudo-code method: It is the algorithm as program, which resembles programming language constructs.
4. Specify algorithms inputs & required outputs.
5. Compound statements are specified with { } or begin ... end
6. Record data types are specified through { }.

ex: record students
{
name datatype
.....
}

7. Operators are used accordingly in the algorithms when necessary

ex: A ← B
A + B = sum etc.

8. A conditional statement has the following forms

if < condition > then statement >
or
if < condition > then < statement -1 >
else < statement -2 >

9. The loop statements are employed. For, while & repeat-until.

→ while loop

while < condition > do
{ statement -1 > ... < statement -n > }

→ For loop:

For variable = value 1 to value-2
do
{ statement-1 > ... <statement-n> }

→ Repeat-until
repeat <statement-1> ... < statement-n>
until < condition >

10. There is only 1 type of procedure:

Algorithm, the heading takes the form, Algorithm Name (Parameter lists

⇒ ANALYSIS FRAMEWORK:

There are 2 kinds of efficiency. They are:—

1. Space efficiency: deals with the extra space the algorithm requires.
In that we have 3 types:
a) program space:- storing the machin language (programming language) generated by the compiler or assembler.
b) data space:- stores the data like constants, variables, keywords etc.
c) stack space:- stores the run address along with local variables (parameters that are passed into the functions)

2. Time efficiency :- how faster is the algorithm executed.

time efficiency can be measured by
a) speed of the computer.
b) choice of the programming language.
c) compiler used in generating the code.
d) choice of the algorithm.
e) number of input / output in the algorithm.

Algorithm efficiency depends on the input size n. & also on the specifics of particular or type of input.

Best case, Worst case, Average case :-

⇒ Worst case :

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, for which the algorithm runs the fastest among all the possible inputs of that size.

ALGORITHM

SequentialSearch (A[0..n-1], K)
//Searches for a given value in a given array by sequential search

//Input: An array A[0..n-1] and a search key K.
//Output: The index of the first element in A that matches K
// or -1 if there are no matching elements.

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1
```

$C_{worst}(n) = n$

⟹ Best case:

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, for which the algorithm runs the fastest among all possible inputs of that size.

ALGORITHM Sequential Search (A[0..n-1], K)

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1
```

$C_{best}(n) = 1$

⇒ Average Case :

The average-case complexity of an algorithm is the amount of time used by the algorithm, averaged over all possible inputs.

ALGORITHM
Sequential Search (A[0..n-1], K)

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1
```

$$C_{(avg)}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots n \cdot \frac{p}{n}\right] + n \cdot (1-p)$$

$$= \frac{p}{n}\left[1 + 2 + \cdots + i + \cdots + n\right] + n(1-p)$$

$$= \frac{p \cancel{n}(n+1)}{\cancel{n}(2)} + n(1-p)$$

$$= \frac{p(n+1)}{2} + n(1-p)$$

Mangal
Date: / /201

Performance Analysis

1. Space complexity : It is the total space taken by the algorithm with respect to the input size.

$$S(p) = C_p + S_p$$

Where,

S_p = space for the dynamic port

C_p = space required for the code segment.

Ex 1 : Consider following algorithm abc ()

```
Algorithm abc(a, b, c)
{
  return a + b++ * c + (a + b - c) / (a + b)
                     + 4.0;
}
```

Here fixed component depends on the size of a, b & c.

So $S_p = 0$.

Ex 2 :

```
for (i = 0; i < n; i++)
  {
for (j = 0; j < 1; j++)
    {
      stmt;
    }
}
```

i	j	no. of time
0	0 ✗	0
1	0 ✓	1
	1 ✗	
2	0 ✓	2
	1 ✓	
	2 ✗	
⋮		
n		: n

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

$$f(n) = \frac{n^2+n}{2}$$

$$O(n^2)$$

Ex 3 : Algorithm Multiply (A, B, n)

```
for (i = 0; i < n; i++)
  {
   for (j = 0; j < n; j++)
                C[i, j] = 0;
    for (k = 0; k < n; k++)
      {
       C[i, j] = C[i, j] + A[i, k] * B[k, j];
      }
  }
}
```

Space

A - n^2 ($n \times n$ matrix)
B - n^2
c - n^2
n - 1
i - 1
j - 1
k - 1

$S(n) = 3n^2 + 4 \Rightarrow O(n^2)$

2. Time complexity: It is defined as the process of determining a formula for the total time required to execute the algorithm.

Ex1:

```
float Sum (float a[], int n)
  {
    float s = 0.0;          ⟶ 1
for (int i=1; i<=n; i++)    ⟶ n+1
  s += a[i];                ⟶ n
  return s;                 ⟶ 1
}                           2n+3 ⇒ O(n)
```

Ex 2:

```
    for (i=0; i<n; i++)     ⟶ n+1
  {
for (j=0; j<n; j++)         ⟶ n(n+1)
  {
    stmts;                  ⟶ n(n)
  }                         2n²+2n+1 ⇒ O(n²)
}
```

Mangal
Date: / /201

ASYMPTOTIC NOTATIONS

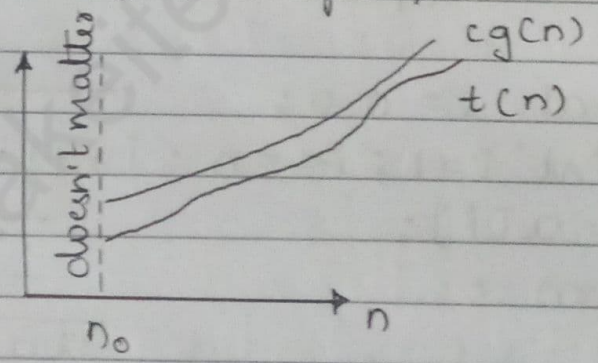
It is a symbolic representation to categorise the algorithmic efficiency at different instances of input of the same size, comparing in the standard efficiency classes.

1. Big Oh - O notation

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$

if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c & some non-negative integer n_0 such that

$$t(n) \leq cg(n) \quad \forall \quad n \geq n_0.$$



Q $5n^2 + 3n + 20 = O(n^2)$

Let $c = 5 + 3 + 20 = 28$

then if $n \geq n_0 = 1$

$5n^2 + 3n + 20 \leq 5n^2 + 3n^2 + 20n^2 = 28n^2$

thus

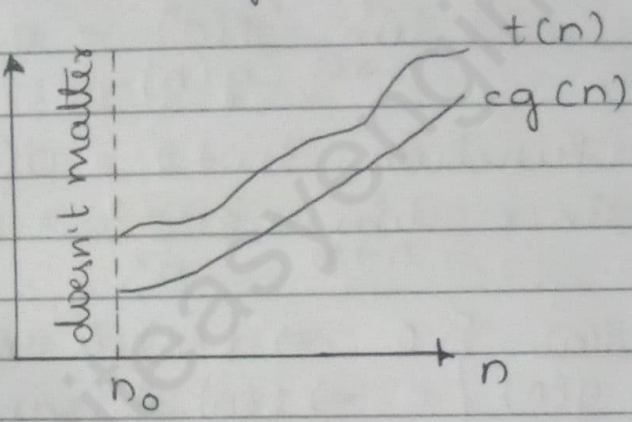
$$5n^2 + 3n + 20 = O(n^2)$$

2. Omega (Ω) notation

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$

if $t(n)$ is bounded below by some +ve constant multiple of $g(n)$ for all large n,
i.e., if there exist some positive constant c & some non-negative integer n_0 such that

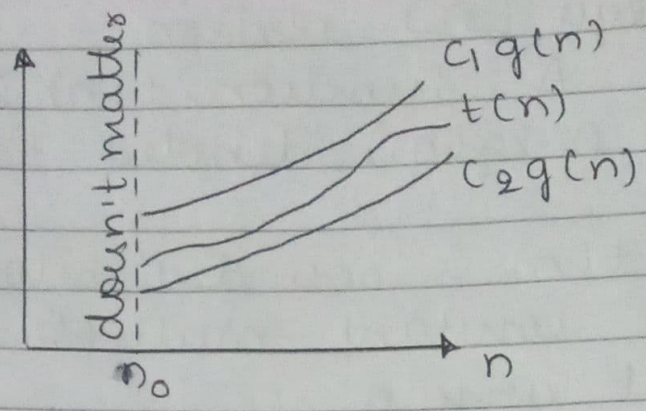
$$t(n) \geq c\, g(n) \quad \forall \quad n \geq n_0$$



3. Theta (Θ) notation :

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above & below by some +ve constant multiples of $g(n)$ for all large n, i.e., if there exist some +ve constants c_1 & c_2 & some nonnegative integer n_0 such that

$$c_2\, g(n) \leq t(n) \leq c_1 g(n)$$
$$\forall \quad n \geq n_0$$



4. Little-oh notation (o)

The function $f(n) = o(g(n))$
i.e., $f(n)$ is o g of n]
if & only if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

ex: The function $3n + 2 = o(n^2)$
since $\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$

$\lim_{n \to \infty} \frac{t(n)}{g(n)}$ { $0 \Rightarrow t(n)$ is smaller than $g(n)$
$c \Rightarrow t(n)$ is same order of $g(n)$
$\infty \Rightarrow t(n)$ has larger order of growth than $g(n)$

Properties of Asymptotic notations

1. General properties

If $f(n)$ is $O(g(n))$ then
$a * f(n) = O(g(n))$
where, a = some constant
ex: $f(n) = 2n^2 + 5$ then $O(n^2)$
if $7 \times f(n) = 7(2n^2 + 5) = 14n^2 + 35$

then $O(n^2)$

2. Reflexive properties

If $f(n)$ is given then $f(n) = O(f(n))$
ex: $f(n) = n^2$ then $O(n^2)$

3. Transitive properties

If $f(n)$ is $O(g(n))$ & $g(n)$ is $O(h(n))$
then $f(n) = O(h(n))$
ex: $f(n) = n$; $g(n) = n^2$; $h(n) = n^3$
$n \Rightarrow O(n^2)$ and $n^2 \Rightarrow O(n^3)$ then
$n \Rightarrow O(n^3)$

NOTE: The 3 properties are also applicable for Big-Ω & Big-Θ (Big-Theta).

4. Symmetric properties
If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$
eg: $f(n) = n^2$ & $g(n) = n^2$
$f(n) = \Theta(g(n)) \Rightarrow \Theta(n^2)$
$g(n) = \Theta(n^2)$

NOTE: It is applicable for Big-Θ only

5. Transpose symmetric
If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$
ex: $f(n) = n$ $g(n) = n^2$
then $n = O(n^2)$ & $n^2 = \Omega(n)$

NOTE: It is applicable for Big-O & Big-Ω

THEOREM: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

[The analogous assertions are true for the Ω and Θ notations as well]

PROOF: Since $t_1(n) \in O(g_1(n))$, there exist some +ve constant c_1 & some non-negative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \quad \forall \quad n \geq n_1$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \quad \forall \quad n \geq n_2$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &\leq c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 \, 2 \max\{g_1(n), g_2(n)\} \end{aligned}$$

Hence $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ with constants c & n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ & $\max\{n_1, n_2\}$ respectively.

Basic efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

Mangal
Date: / /201

class	Name	comments
1	constant	Short of best-case efficiencies, Algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	logarithmic	Logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have atleast linear running time.
n	linear	Algorithms that scan a list of size n belong to this class.
$n \log n$	linear ~~arithmetic~~ -ithmic	Many divide-and-conquer algorithms including mergesort & quicksort in the average case, fall into this category.
n^2	quadratic	characterizes efficiency of algorithms with 2 embedded loops. Several nontrivial algorithms from linear algebra fall into this class
2^n	exponential	Algorithms that generate all subsets of an n-element set. The term is also used in a broader sense to include this & larger orders of growth as well.
$n!$	factorial	Algorithms that generate all permutations of an n-element set.

fast ↓	1	constant	high time efficiency
	$\log n$	logarithmic	
	n	linear	
	$n \log n$	$n \log n$	
	n^2	quadratic	
	n^3	cubic	
	2^n	exponential	low time efficiency
slow ↓	$n!$	factorial	

MATHEMATICAL ANALYSIS OF NON-RECURSIVE & RECURSIVE ALGORITHMS

⇒ General plan for analyzing efficiency of non-recursive algorithms :-

1. Decide on parameter n indicating the input size of the algorithm.
2. Identify algorithm's basic operation.
3. Check whether the number of times the basic operation is executed depends only on the input size n.

   If it depends on the type of input, then investigate worst, average & best case efficiency seperately.
4. Set up summation for $C(n)$ reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation using standard formulas.

ex: Finding the largest element in a given array.

ALGORITHM MaxElement (A[0...n-1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n-1] of real no.'s
//Output: The value of the largest element in A

```
max val ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
        maxval ← A[i]
    return maxval
```

ANALYSIS: 1. Input size: the number of elements = n (size of the array)
2. Two operations can be considered to be as basic operation i.e.,
   a) Comparison:: A[i] > maxval
   b) Assignment:: maxval ← A[i]
Here the comparison statement is considered to be the basic operation of the algorithm.
3. NO best, worst, average cases - because the no. of comparisons will be same for all arrays of size n & it is not dependent on type of input.
4. Let C(n) denotes no. of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable

i within the bound between 1 & n-1.

$$C(n) = \sum_{i=1}^{n-1} 1$$

This is an essay sum to compute because it is nothing other than 1 repeated n-1 times. Thus

$$C(n) = \sum_{i=1}^{n-1} 1 \qquad 1+1+1+\cdots+1$$

[(n-1) number of times]

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Algorithm Unique Elements (A[0..n-1])

//Check whether all the elements in a given array are distinct
//Input : An array A[0..n-1]
//Output : Returns true if all the elements in A are distinct and false otherwise

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] == A[j]
            return false
return true
```

⇒ General plan for analyzing efficiency of recursive algorithms :—

1. Decide on parameter n indicating input size.
2. Identify algorithm's basic operation.
3. Check whether the number of times

the basic operation is executed depends only on the input size n.

If it also depends on the type of input, investing worst, average & best case efficiency seperately.

4. Set up recurrence relation, with an appropriate initial condition, for the no. of times the algorithm's basic operation is executed.
5. Solve the recurrence.

Ex: Tower of Hanoi

Algorithm : Tower (n, S, T, D)
//Input : no. of discs 'n'
3 - poles - S, T, D
//Output : n-discs on distination pole
if (n=1) then
move disc 1 from S to D
else
Tower (n-1), S, D, T)
move n^{th} disk from S to D
Tower (n-1), T, S, D)
end if

Analysis : 1. Input parameter - 'n' no. of disks
2) B.O ⇒ moving discs
3) Depends only on 'n' = no variation
4) Algorithm

Mangal
Date: / /201

Tower (n, S, T, D) = { move disc 1 from S to D (n = 1); Tower (n-1, S, D, T); move n^{th} disc from S to D (n > 1); Tower (n-1, T, S, D) }

B.O recurrence relation

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

5) $m(n) = 1 + 2M(n-1)$ until $M(1) = 1$

$= 1 + 2[1 + 2M(n-2)]$

$= 1 + 2 + 2^2 M(n-2)$

$= 1 + 2 + 2^2 [1 + 2M(n-3)]$

$= 2^0 + 2^1 + 2^3 M(n-3)$

$= 2^0 + 2^1 + 2^3 + \cdots 2^{n-1} M(n-(n-1))$

$= 2^0 + 2^1 + 2^3 + \cdots 2^{n-1}$

$= GP$

Sum of Geometric Progression

$$S_n = \frac{a(r^n - 1)}{r - 1}$$

$a = 1, \quad r = 2$

$$M(n) = \frac{1(2^n - 1)}{2 - 1} \Rightarrow 2^n - 1$$

For large value of n

$M(n) \approx 2^n$

$M(n) \in \Theta(2^n)$

Exponential order of growth

IMPORTANT PROBLEM TYPES

1. Sorting

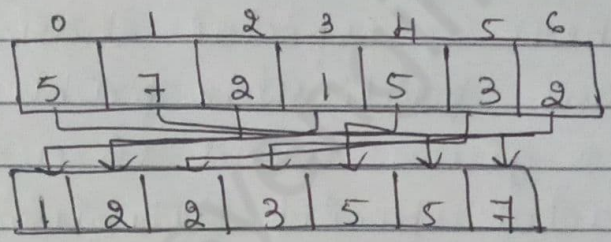
Ordering of elements based required manner.

ex: Bubble sort, merge sort, radix sort, insertion sort etc

Properties of sorting algorithm

a) Stable property: Algorithm is said to be stable if it maintains the relative order of the duplicate elements even after sorting.

ex:



b) In-place property: Algorithm is said to be in-place if the algorithm dosnot consume extra memory.

ex: Bubble sort, selection, insertion sort

2. Searching

The searching problem deals with finding a given value, called a search key, in a given set.

a) Numeric searching

searching simple / set of numbers (patterns)

b) Non-numeric searching

searching characters / sub-string

For searching, there is no single algorithm that fits all situation best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays and so on.

3. String Processing

A string is a sequence of characters from an alphabet.

Strings of particular interest are text strings, which comprise letters, numbers & special characters.

One particular problem searching for a given word in a text - has attracted special attention from researchers referred to as string matching.

4. Graph Problems

The oldest & most interesting areas in algorithmics is graph algorithms.

Graph is a collection of points called vertices, some of which are connected by line segments called edges.

Graphs can be used for modeling a wide variety of applications, including transportation, communication, social & economic networks, project scheduling & games.

Mangal
Date: / /201

Basic graph algorithms include graph-traversal algorithms, shortest-path algorithms & topological sorting for graphs with directed edges.

5. Combinatorial Problems

First, the number of combinatorial objects typically grows extremely fast with problem's size, reaching unimaginable magnitudes even for moderate-sized

Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

FUNDAMENTAL DATA STRUCTURES

A data structure can be defined as a particular scheme of organizing related data items.

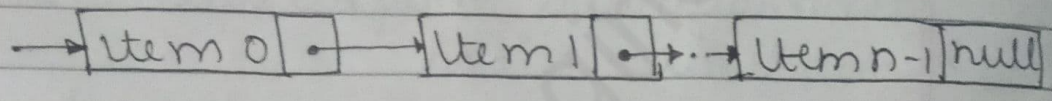
1. Linear Data Structures:

a) array: An array is a sequence of n items of the same data type that stored contiguously in computer memory and made accessible by specifying a value of the array's index.

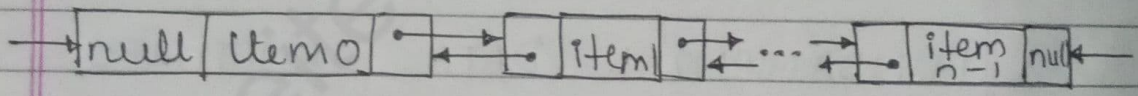
ex: | Item[0] | Item[1] | ... | Item[n-1] |

b) Linked list : It is a sequence of zero or more elements called nodes, each containing 2 kinds of information : some data & one or more links called pointers to other nodes of the linked list.

⇒ Singly linked list : each node except the last one contains a single pointer to the next element



⇒ Doubly linked list : every node, except the first & the last, contains pointers to both its successor & its predecessor



c) List : A list is finite sequence of data items i.e., a collection of data items arranged in a certain linear order.

The basic operations performed on this data structure for are searching, inserting and deleting an element.

⇒ Stack : It is a list in which insertions & deletions can be done only at the end. This end is called the top because a stack is usually visualized not

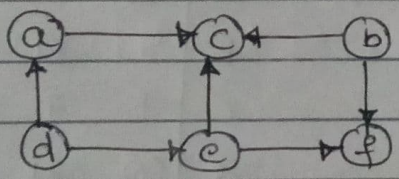
horizontally but vertically.

⇒ Queue : It is a list from which elements are deleted from one end of the structure called the front & new elements are added to the other end called the rear.

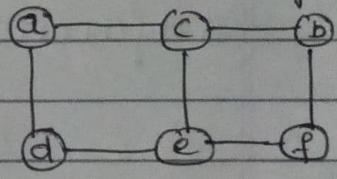
2. Graphs :- Graph is a collection of points in the plane called vertices or nodes.

Graph

Directed (digraphs)

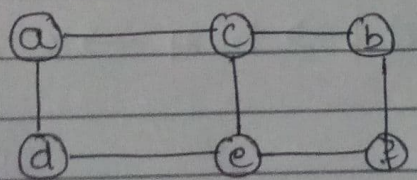


Undirected graph



Graph representation :-

1. Adjacency matrix & adjacency list



adjacency matrix :

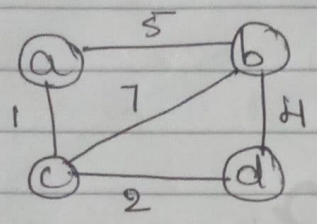
	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

adjacency list:

a	→ c → d
b	→ c → f
c	→ a → b → e
d	→ a → e
e	→ c → d → f
f	→ b → e

Weighted Graphs : A graph with numbers assigned to its edges.

ex :



adjacency matrix:

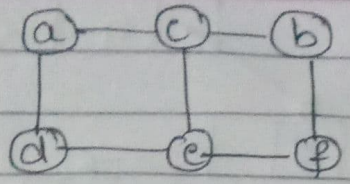
	a	b	c	d
a	∞	5	1	∞
b	5	0	7	4
c	1	7	0	2
d	∞	4	2	0

adjacency list:

a	→ b, 5 → c, 1
b	→ a, 5 → c, 7 → d, 4
c	→ a, 1 → b, 7 → d, 2
d	→ b, 4 → c, 2

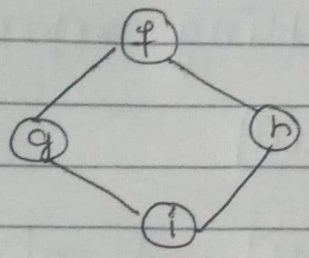
Paths and cycles

Path : A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent vertices that starts with u & ends with v.

ex: 

here, acbf is path

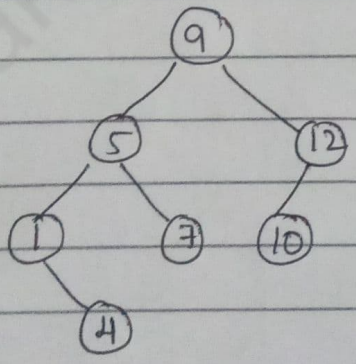
Cycle: Cycle is a path that starts & ends at the same vertex & doesnot traverse the same edge more than once.

ex: 

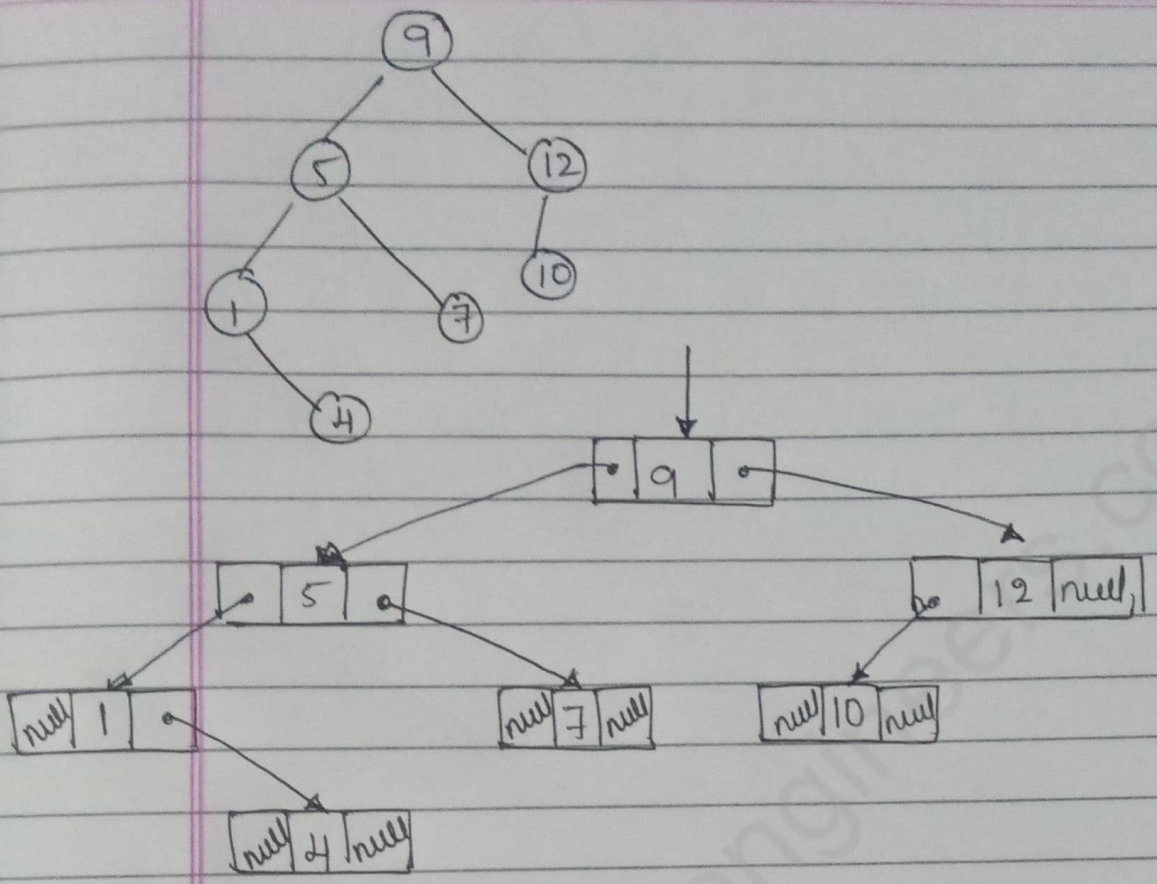
here, fhigf is cycle.

3. Trees:

Binary tree: A tree in which every vertex has no more than 2 children & each child is designated as either a left child or right child of its parent.

ex: 

Binary search tree: Each parental vertex is larger than all the numbers in its left subtree & smaller than all the numbers in its right subtree.



First child - next sibling representation