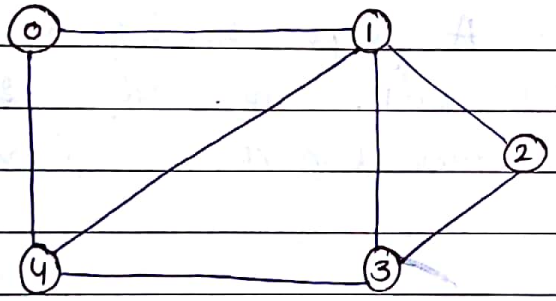
Unit 4

★ Graph :- A graph is simply a set of points together with a set of lines connecting various points. It is a non-linear data structure.

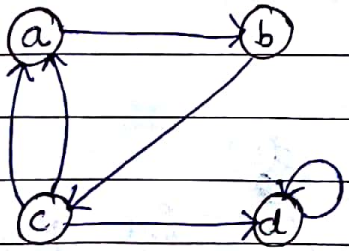
It is a pictorial representation of a set of objects where some pairs of objects are connected by links.

Vertices :- The interconnected objects are represented by points.

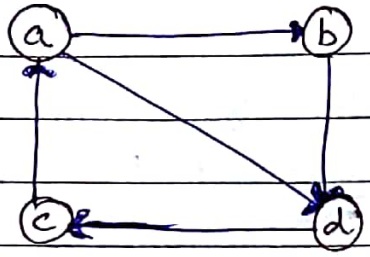
Edges :- The links that connect the vertices.



Directed graph :- In graph theory, a directed graph is a graph that is set of vertices connected by edges, where the edges have a direction associated with them. It is also called digraph.



Undirected graph :- Undirected graph have edges that do not have a directed. the edges indicate a two way relationship, in both directions. all the edges are bidirectional.



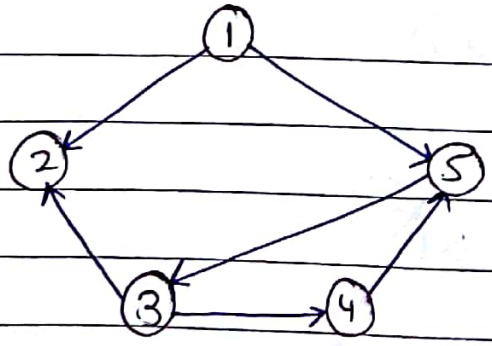
Directed acyclic graph :- A directed acyclic graph is a graph that have no cycles whereas in a cyclic graph it have direction.

Weighted graph :- A graph having a weight or number, associated with some edge. It is a graph in which each branch is given a numerical weight.

Representation of Graph :-

1. Adjacency Matrix (array)
2. Adjacency list (link list)

1. Adjacency Matrix :-

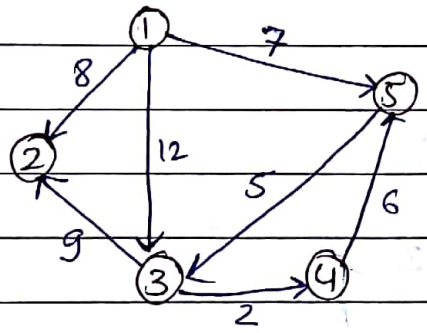


int graph [5][5]

a[i][j]

vertices	1	2	3	4	5
1	0	1	1	0	1
2	0	0	0	0	0
3	0	1	0	1	0
4	0	0	0	0	1
5	0	0	1	0	0

Adjacency matrix is a 2D array of size V x V where V is the number of vertices in a graph. We can also use weighted graph.

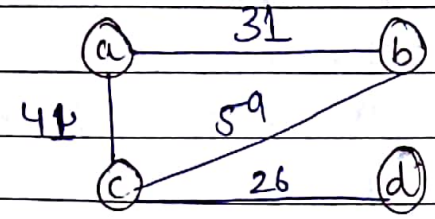


a[i][j]

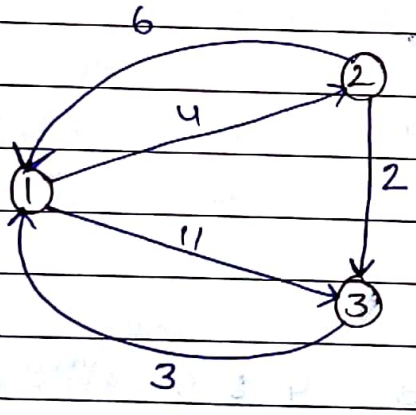
vertices	1	2	3	4	5
1	0	8	12	0	7
2	0	0	0	0	0
3	0	9	0	2	0
4	0	0	0	0	6
5	0	0	5	0	0

fig. for weighted graph it means cost is given.

Adjacency matrix of a given graph :-



vertices	1	2	3	4
1	∞	31	41	∞
2	31	∞	59	∞
3	41	59	∞	26
4	∞	∞	29	∞

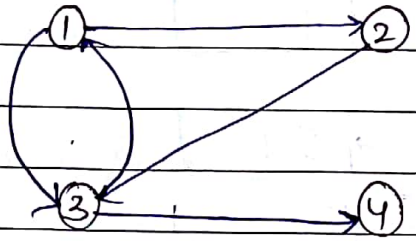


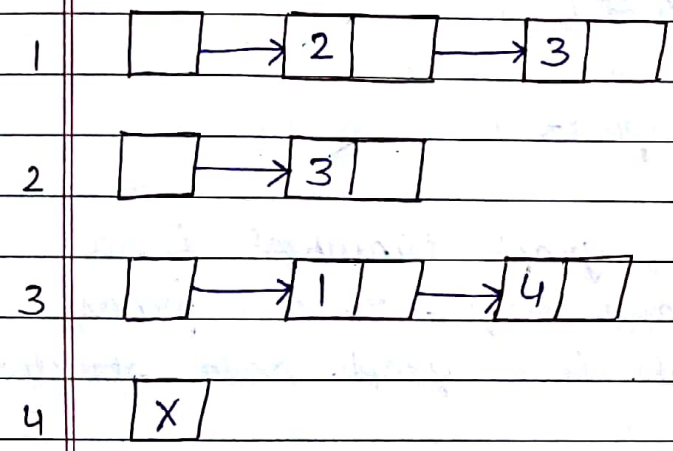
vertices	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

cost adjacent matrix

2. Adjacency list:- An array of linked list is used size of the array is equal to number of vertices In this representation, every vertex of graph contains list of its adjacent vertices. It is an array a of separate lists.

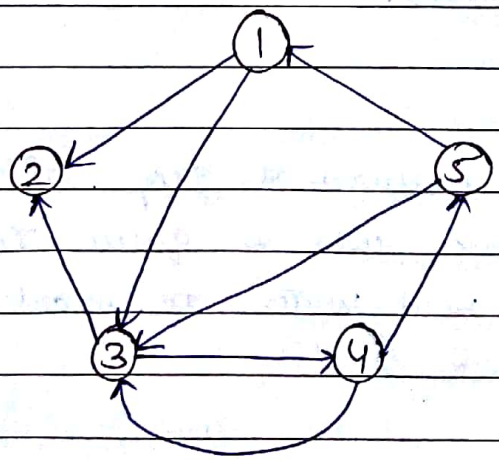
eg:- (1)

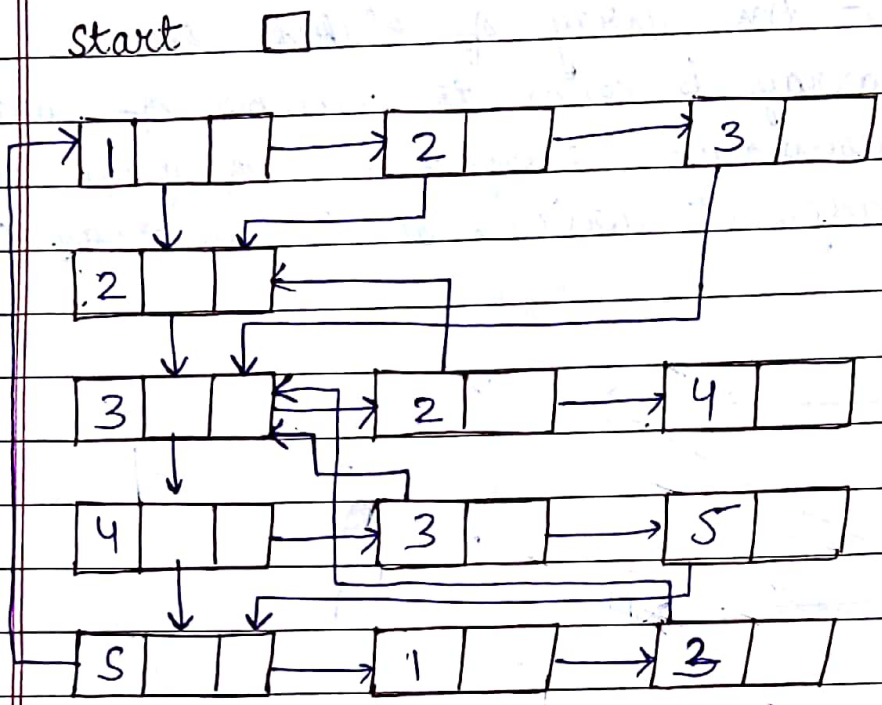




Node list →

eg:- (2)





Traversing in graph:-

In computer science, graph traversal is a form of traversal and refers to the process of visiting each node is a graph data structure exactly once.

It is classified in two ways:-

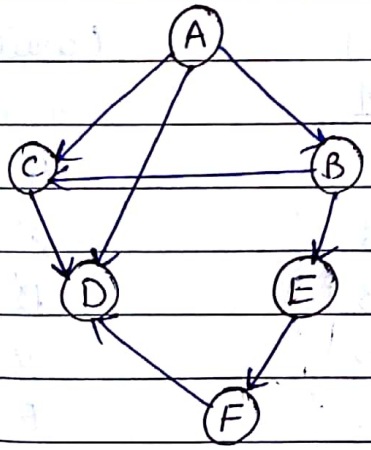
(i) BFS (Breadth first search)
(ii) DFS (Depth first search).

i) BFS:-

BFS algorithm traverse a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search.

Algorithm :-

1. Initialise all nodes to the ready states (States = 1).
2. Put the starting node A in queue and change its states (Status = 2).
3. Repeat step 4 and 5 until queue is empty.
4. Remove the front node N. of queue process N and change the status (States = 3).
5. Add to the rear of queue all the neighbour (States = 1) and change their status to the waiting states (status = 2)
6. Exit.



	(ready) Status 1	(waiting) Status 2
A	~~X~~ ~~Z~~ 3	~~A~~ ~~B~~
B	~~X~~ ~~Z~~ 3	~~C~~
C	~~X~~ ~~Z~~ 3	~~D~~
D	~~X~~ ~~Z~~ 3	~~E~~
F	~~X~~ ~~Z~~ 3	~~F~~
F	~~X~~ ~~Z~~ 3	

(process) States 3

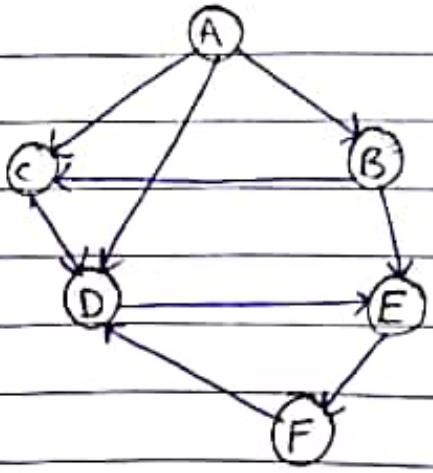
A	B	C	D	E	F	

ii) DFS :-

DFS algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search.

Algorithm:-

1. Initialize all the nodes to the ready status (Status=1).
2. Put the starting node in STACK and change its status (Status = 2).
3. Repeat step 4 to 5 until stack is empty.
4. Remove the top node of STACK process and change the status (status = 3).
5. Add to the top of STACK all the neighbours of that are in the ready state (Status = 1) and change their states to the waiting states (Status=3).
6. Exit.



	(ready) Status 1	(waiting) Status 2
A	✗	A
B	✗	B
C	✗	C
D	✗	D
E	✗	E
F	✗	F

Status 3

A	C	D	B	E	F

Stack is empty.

Warshall's algorithm:-

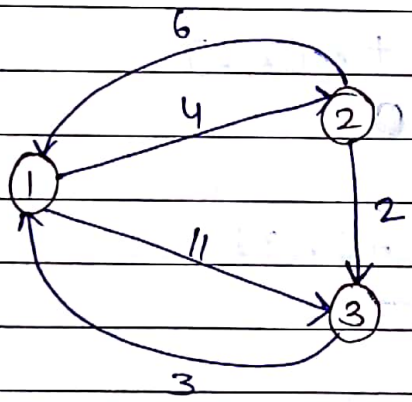
The Warshall's algorithm is an algorithm for finding shortest path in a weighted graph with +ve or -ve edge weight. A find the length of the

shortest between all pairs of vertices.

Algorithm :-

```
for i := 1 to n do
    for j := 1 to n do
    A[i,j] := cost[i,j];

for   k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := min (A[i,j], A[i,k] + A[k,j]);
```



A

i/j	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

A'

i/j	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A^2

i/j	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3

i/j	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

A[1,2] , A[1,1] + A[2,2]
4 , 0 + 0

A[2,1] , A[2,2] + A[2,1]
6 , 0 + 6

A[1,3] , A[1,1] + A[1,3]
11 , 0 + 11

A[2,1] , A[2,2] + A[2,1]
6 , 0 + 6

A[2,2] , A[2,2] + A[2,2]
0 , 0 + 0

A[2,3] , A[2,2] + A[2,3]
2 , 0 + 2

Ans:-

i|j	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Sorting:- In data structure sorting is a process by which we arrange the data in a logical order. this logical order can be either ascending (smaller to larger) or decending order (larger to smaller).

Sorting is related to searching, sorting arrange data in sequence which makes searching better or easier.

Types of sorting :-

1. Internal sorting
2. External sorting

1. Internal sorting :- In this technique all the sorted data present in main memory.
following types of internal sorting :-

i) Bubble sort
ii) Insertion sort
iii) Quick sort
iv) Heap sort
v) Selection sort

2. External sorting :- In this tecnique sorted data is present in secondary memory. Because the size of data is large it cannot fit in the main memory. There is only one type of external sorting is merge sort.

A. Insertion sort :-

Algo

1. Repeat step 2 to 4 for K = 2, 3 . . . N.
2. Set temp := A[K] and Ptr := K-1.
3. Repeat while temp < A[Ptr] and Ptr > 1.
   a) Set A[Ptr+1] := A[Ptr]
   b) Set Ptr := Ptr - 1
4. Set A[Ptr+1] := temp
5. Exit.

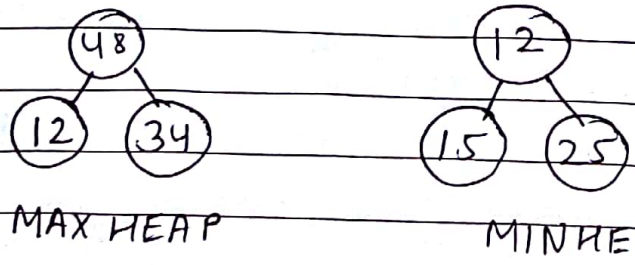
B. Quick sort :- (Fastest)

Algo → Quick sort (K [ ], LB, UB)

1. Initialize Flag := True.
2. If LB < UB then
   a) Set I := LB , J := UB+1 , KEY := K[LB]
   b) Repeat Step (i) to (v) while (Flag = True)
      (i) set I := I+1
      (ii) Repeat while (K[I] < Key) do
           SET I := I+1
      (iii) Set J := J-1
      (iv) Repeat while (K[J] > Key) do
           Set J := J-1
      (v) If (I <= J) then
           Swap K[I] and K[J]
           else
           Flag := False
   c) Swap K[LB] and K[J]
   d) call Quicksort (K[], LB, J-1)
   e) call Quicksort (K[], J+1, UB).
3. Exit.

C. Heap Sort :-

① It is a binary tree. complete binary tree.
② Value of node should be greater than its children called MAXHEAP.
③ Value of node should be smaller than its children called MINHEAP.



MAX HEAP MINHEAP

Algo

Heapsort (A, N)

1. Repeat for (J = 1 to N-1) do
   call INSHEAP (A, J, A[J+1])
2. Repeat while (N > 1) do
   (a) call DELHEAP (A, N, ITEM)
   (b) Set A[N+1] := ITEM
3. Exit.

INSHEAP (TREE, N, ITEM)

1. Set N := N+1 and PTR := N
2. Repeat steps 3 to 6 while PTR > 1
3. Set PAR := ⌊PTR/2⌋
4. If ITEM <= TREE[PAR], then
   Set TREE[PTR] := ITEM & Return
5. Set TREE[PTR] := TREE[PAR]
6. Set PTR := PAR.
7. Set TREE[1] := PAR
8. Return

DELHEAP (TREE, N, ITEM)

1. Set ITEM := TREE[1]
2. Set LAST := TREE[N] & N := N-1
3. Set PTR := 1, LEFT := 2 & RIGHT := 3.
4. Repeat steps 5 to 7 while RIGHT ≤ N
5. If (LAST ≥ TREE[LEFT] and LAST ≥ TREE [Right] then
   Set TREE[PTR] := LAST & Return
6. If (TREE[RIGHT] ≤ TREE[LEFT] then
   (a) Set TREE[PTR] := TREE[LEFT]
   (b) PTR := LEFT.

else
(a) set TREE [PTR] := TREE [RIGHT]
(b) Ptr := RIGHT.

7. Set LEFT := L * PTR & RIGHT := LEFT + 1
8. If (LEFT = N) and LAST < TREE [LEFT]) then
TREE [Ptr] := TREE [LEFT]
Ptr := LEFT.
9. Set TREE [PTR] := LAST
10. Return.

D. Selection sort :-
Algo.
1. Repeat step 2, 3, 4 for K = 1, 2, ... n-1. @ Set min := K
2. Repeat for i := K+1, K+2 ... n
If A [i] < A [min] then
Set min := i.
3. If (min ≠ K) then
temp := A [K]
A [K] := A [min];
A [min] := temp;
4. Exit.

E. Merge sort :-
Algo
Mergesort (low, high)
{
if (low < high) then
{
mid := [(low + high) / 2];
Mergesort (low, mid);
Mergesort (mid+1, high);

```
    Merge ( low, mid, high);
  }
}

Merge (low, mid, high)
{
    h := low
    i := low
    j := mid + 1
    while ((h ≤ mid) and (j ≤ high)) do
    {
      if (a[h] ≤ a[j]) then
      {
        b[i] := a[h];
        h := h+1;
      }
      else
      {
        b[i] i= a[j];
        j := j+1;
      }
      i := i+1
    }

    if (h > mid) then
      for k := j to high do
      {
        b[i] := a[K]
        i := i+1;
      }
```

```
else
    for k:=h to mid do
    {
        b[i]:=a[k]
        i:=i+1
    }
    for k:= low to high do
        a[k]:=b[k];
}
}
```