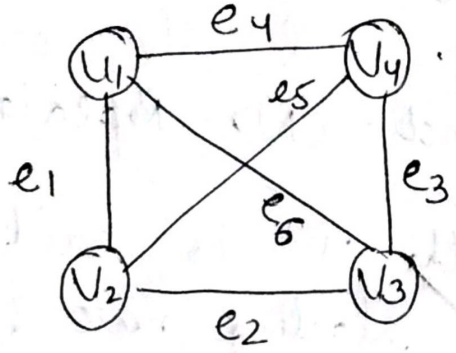
Graph :

* A Graph G=(V,E) consists of two sets.

1) A set V of elements called as nodes or vertices.

2) A set E of edges such that each edge $e \in E$ is identified by a unique pair (u, v) of nodes in V and is denoted as $e = [u, v]$.

* Graph is a non-linear data structure



$V(G) = \{V_1, V_2, V_3, V_4\}$

$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

$= \{(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1) - - \}$

* Edge e_1 connects the vertex V_1 & V_2. Hence V_1 and V_2 are the end points of edge e_1 and V_1 & V_2 are called as the adjacent nodes or neighbour.

Degree of a node (u) :

degree (u) is the no. of edges connected to the node u.

$\deg(v_4) = 3$, $\deg(v_3) = 3$

* if any node's degree is zero, then that node is called as isolated node.
* A path P of length n from a node to another node is the sequence of (n+1) nodes

e.g., $P = \{(v_4, v_2), (v_2, v_3)\}$

length = 2.

Note :- The path from one node to another is not unique.

Simple Path :- A path P is said to be simple if all the nodes in that path are distinct. (No Repeatation)

closed Path :- A path P is said to be closed if starting node & destination nodes are same.

$v_4 \to v_2 \to v_3 \to v_4 \to v_1$

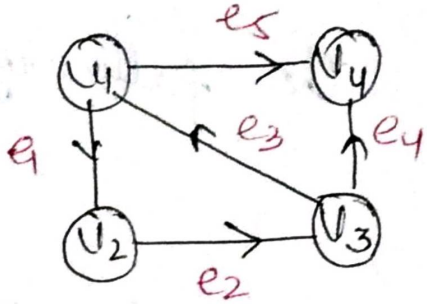
cycle :- cycle is a closed simple path of length 3 or more.

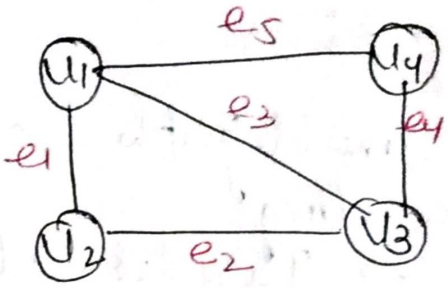
$v_1 \to v_2 \to v_3 \to v_4 \to v_1$

$v_1 \to v_2 \to v_3 \to v_1$

Types of Graphs :-

① Directed Vs Undirected Graph :-





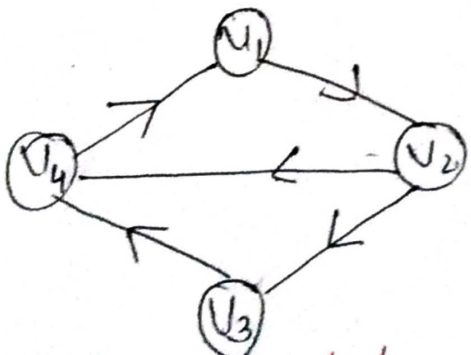
* e_5 is an edge from v_1 to v_4 but there is no edge from v_4 to v_7

* e_5 is an edge from v_1 to v_4 as well as from v_4 to v_1.

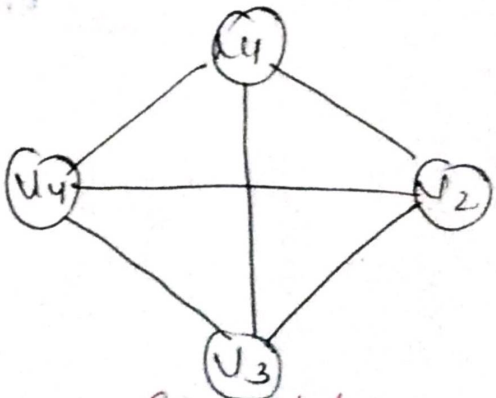
② Connected Vs Complete Graphs :-

⇒ A graph G is said to be connected if and only if there exists a ~~single~~ simple path between any two nodes of the graph.

⇒ A graph G is said to be a complete graph if each node u is adjacent to all other nodes v of G.



connected

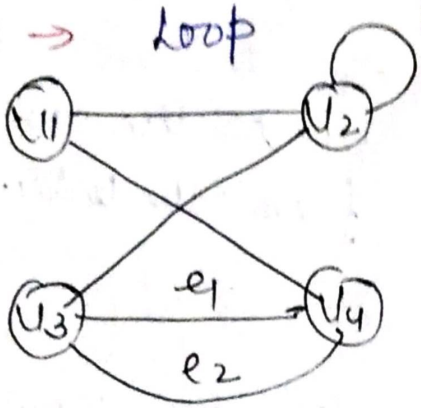


Complete

* If a complete graph contains n nodes, then there will be $\frac{n(n-1)}{2}$ no. of edges

Multigraph :-

* A multigraph consists of the following two things :
  - → Parallel edges
  - → Loop



$e_1 = (V_3, V_4)$, $e_2 = (V_3, V_4)$

Parallel edges :- Two different edges e_1 & e_2 are said to be parallel edges if they have same end points. e.g., e_1 & e_2

Loop :- An edge e is called a loop if it has identical end points e.g., e_3

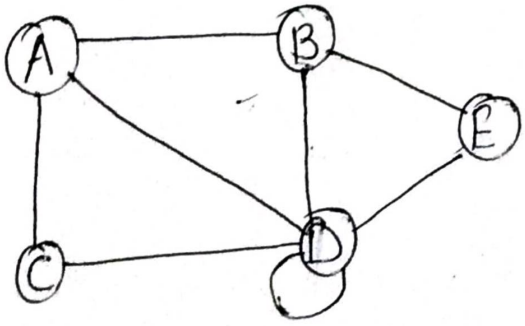
Pendant Vertex :- A vertex with degree one is known as pendant vertex.

Representation of Graph :-

1. Sequential representation using adjacency matrix
2. Linked representation using linked list and adjacency list

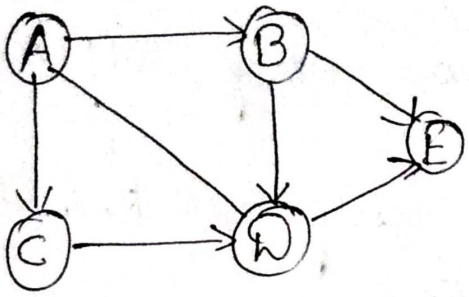
Adjacency Matrix :-

* In this representation the graph is represented using a matrix of size total no. of vertices by total no of vertices.
* Thus a graph with 4 vertices is represented using a matrix of size 4×4.
* The matrix is filled with either 1 or 0.
  - → 1 represents that there is an edge from row vertex to column vertex.
  - → 0 represents that there is no edge from row vertex to column vertex.



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Directed graph representation :-



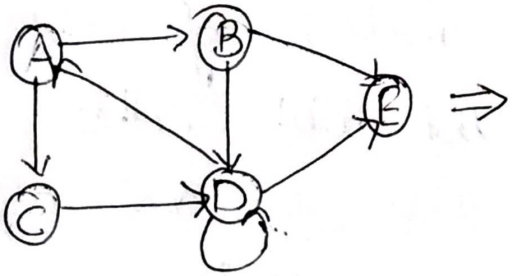
$\Rightarrow$

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

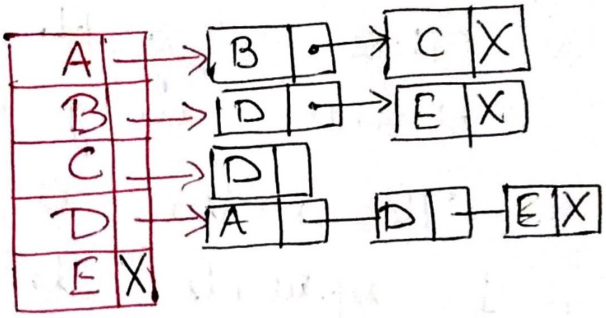
Adjacency list :-

* In this representation, every vertex of a graph contains list of its adjacent vertices.

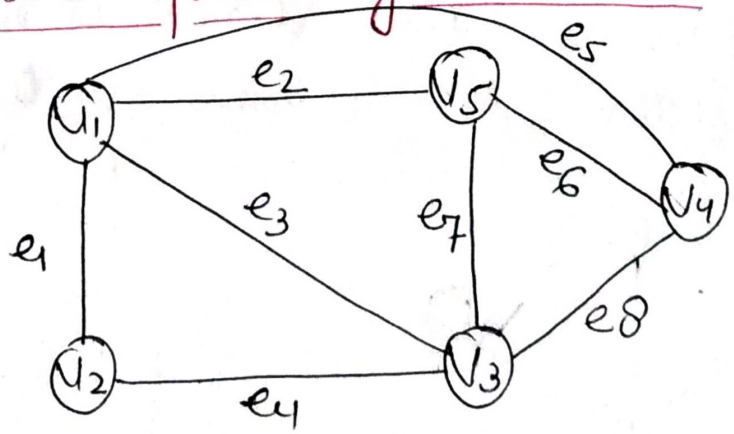
e.g.,



$\Rightarrow$



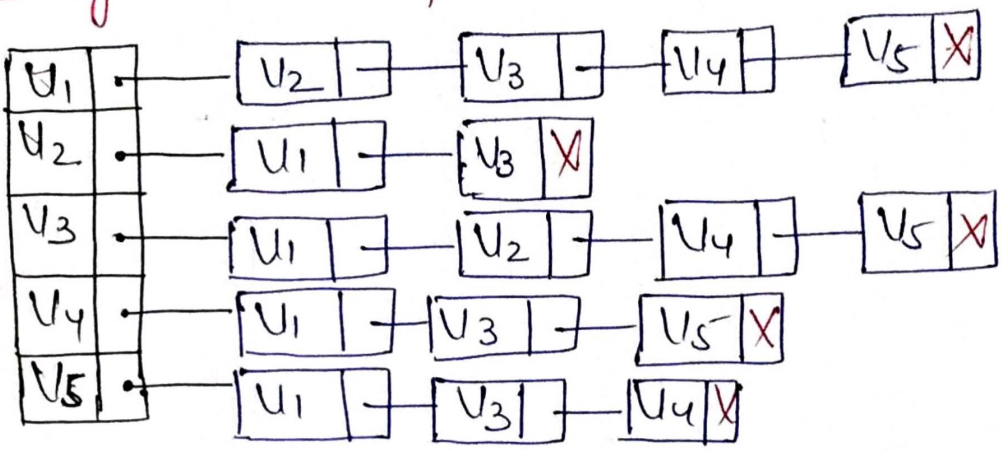
Q. Consider the following examples :-



Adjacency Matrix Representation :-

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	1	1
V_2	1	0	1	0	0
V_3	1	1	0	1	1
V_4	1	0	1	0	1
V_5	1	0	1	1	0

Adjacency List Representation :-



Incidence Matrix Representation :-

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
V_1	1	1	1	0	1	0	0	0
V_2	1	0	0	1	0	0	0	0
V_3	0	0	1	1	0	0	1	1
V_4	0	0	0	0	1	1	0	1
V_5	0	1	0	0	0	1	1	0

Traversing of a Graph :-

* There are two standard ways to traverse any graph :

(1) Breadth First Search (BFS)

(2) Depth First Search (DFS)

* Many graph algo require one to systematically examine the nodes and edges of a graph G.

* The Breadth first search (BFS) will use a queue as an auxiliary structure to hold nodes for future processing. and depth first search will use a stack.

* During the execution of our algo, each node N of G will be in one of three states, called the status of N, as follows :

Status = 1 : (Ready state) The initial state of N

Status = 2 : (Waiting State) The node N is on the queue or stack, waiting to be processed.

Status = 3 :- (Processed State) The node N has been processed.

Breadth - First Search

* The general idea behind a BFS beginning at a starting node A is as follows:
  - → First examine the starting node A.
  - → Then examine all the neighbours of A
  - → Then examine all the neighbours of neighbours of A and so on.
  - → We need to guarantee that no node is processed more than once.
  - → This is accomplished by using a queue to hold nodes that are waiting to be processed & by using a field STATUS which tells us the current status of any node.

Algorithm (BFS) :-

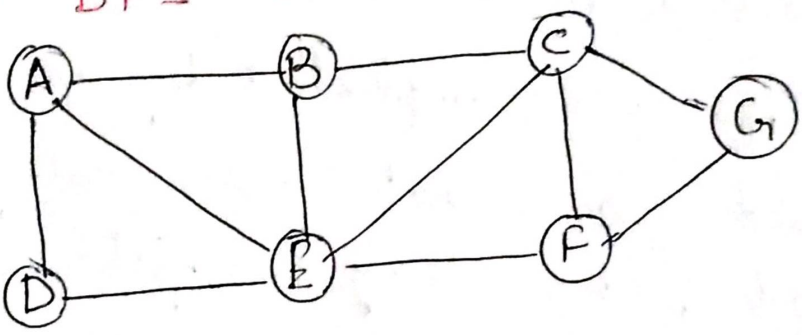
1. Initialize all nodes to the ready state (STATUS = 1)
2. Put the starting node A in the QUEUE & change its status to waiting (STATUS = 2).
3. Repeat step 4 & 5 until QUEUE is empty.
4. Remove the ~~first~~ front node N of QUEUE.

Process N & chang the status of N to processed STATUS = 3.

5. Add to the rear of QUEUE all the neighbours of N that are in steady state (STATUS = 1) & then change their status to the waiting state (STATUS = 2)
[End of step 3 loop]

6. Exit

Q Consider the following example to perform BFS traversal

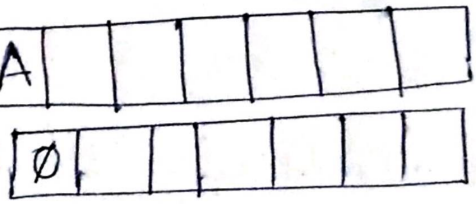


List

A:- B, D, E
B: A, C, E
C: B, E, F, G
D: A, E
E: A, B, C, D, F
F: C, E, G
G: C, F

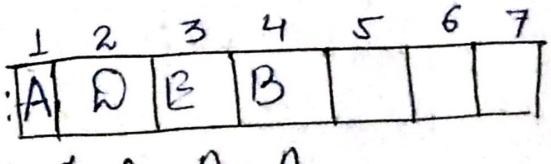
(i) Initially add A to QUEUE and add NULL to ORIG as follows:-

FRONT = 1
REAR = 1



(ii) Remove front element A from QUEUE Set front = front + 1 & add to QUEUE the neighbours of A

Front = 2
Rear = 4



* Origin A of all three edges is added to ORIG!

(iii) F = 3, R = 4

Q | A | D | E | B | | | |

Orig ∅ A A A

(iv) F = 4, R = 6

Q | A | D | E | B | C | F | |
1 2 3 4 5 6 7

Orig ∅ A A A E E

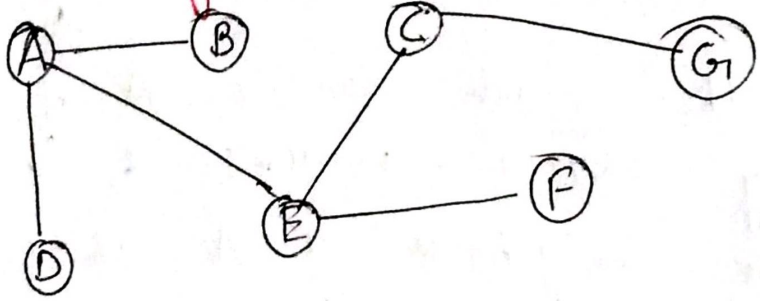
(v) F = 5, R = 7

Q | A | D | E | B | C | F | G |

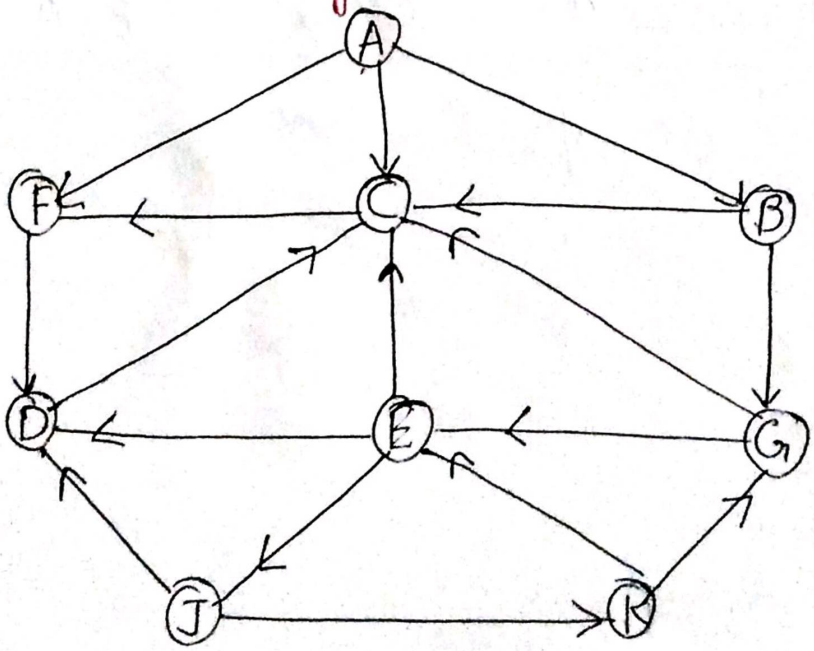
Orig ∅ A A A E E C

(vi) F = 6, R = 7

Final Result of BFS is :-



Q : Consider the graph below



Adjacency list
A : F, C, B
B : G, C
C : F
D : C
E : D, C, J
F : D
G : C, E
J : D, K
K : E, G

(a) Initially, add A to QUEUE & add NULL to ORIG as follows:-

FRONT = 1	QUEUE = A
REAR = 1	ORIG = ϕ

(b) Remove the front element A from QUEUE by setting FRONT = FRONT + 1 & add to QUEUE the neighbours of A as follows:-

FRONT = 2	QUEUE = A, F, C, B
REAR = 4	ORIG = ϕ, A, A, A

(c) Remove the front element F from the Q by setting F = F + 1 & add neighbours of F.

F = 3
R = 5

QUEUE: A, F, C, B, D
ORIG: ϕ A, A, A, F

(d) $f = 4$, $R = 5$

Q : A, ~~F~~, C, B, D

O : ϕ A, A, A, F

neighbour F of C can't be added as f is not in the ready state.

(e) $F = 5$, $R = 6$

Q : ~~A~~, ~~F~~, ~~C~~, ~~B~~, D, G

O : ϕ, A, A, A, F, B

(f) $F = 6$, $R = 6$

Q : ~~A~~, ~~F~~, ~~C~~, ~~B~~, ~~D~~, G

O : ϕ, A, A, A, F, B

(g) $F = 7$, $R = 7$

Q : ~~A~~, ~~F~~, ~~C~~, ~~B~~, ~~D~~, ~~G~~, E

O : ϕ, A, A, A, F, B, G

(h) $F = 8$, $R = 8$

Q : A, ~~F~~, ~~C~~, B, ~~D~~, ~~G~~, ~~E~~, J

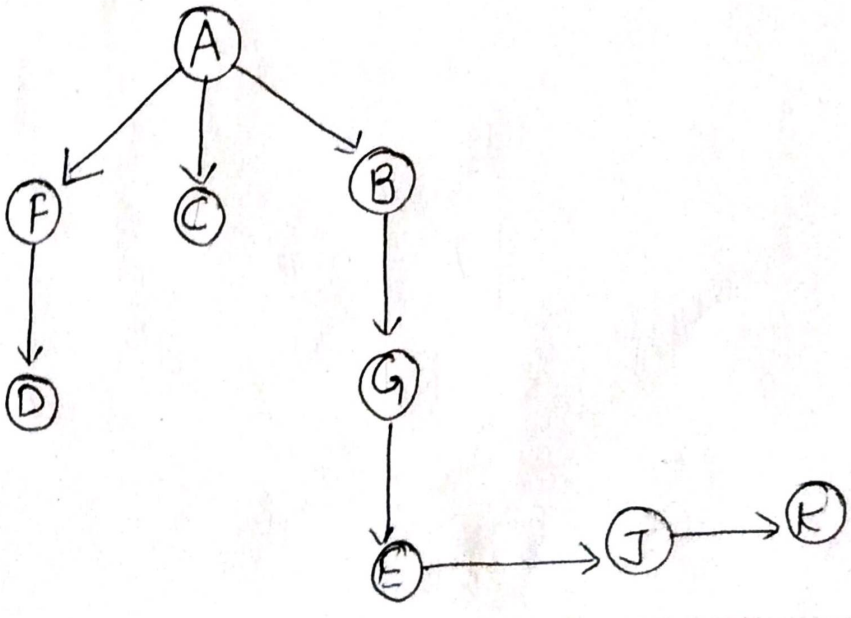
O : ϕ A, A, A, F, B, G, E

(i) $F = 9$, $R = 9$

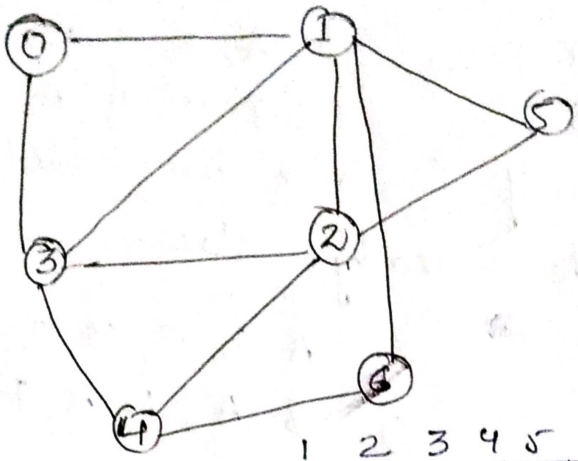
Q : ~~A~~, ~~F~~, ~~C~~, ~~B~~, ~~D~~, ~~G~~, ~~E~~, ~~J~~, K

O : ϕ, A, A, A, F, B, G, E, J

(j)

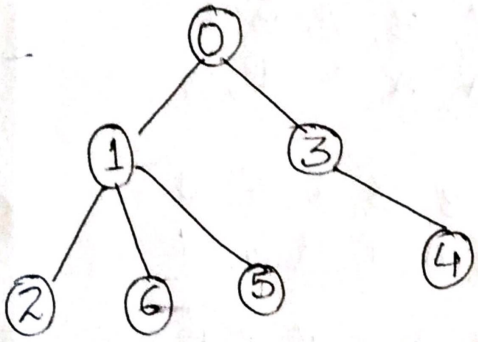


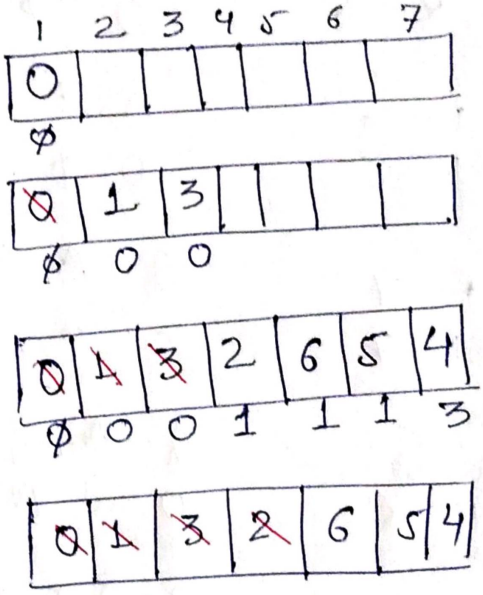
Q:



0: 1, 3
1: 0, 3, 2, 6, 5
2: 3, 1, 4, 5
3: 0, 1, 2, 4
4: 2, 3, 6
5: 1, 2
6: 1, 4

Start with '0'



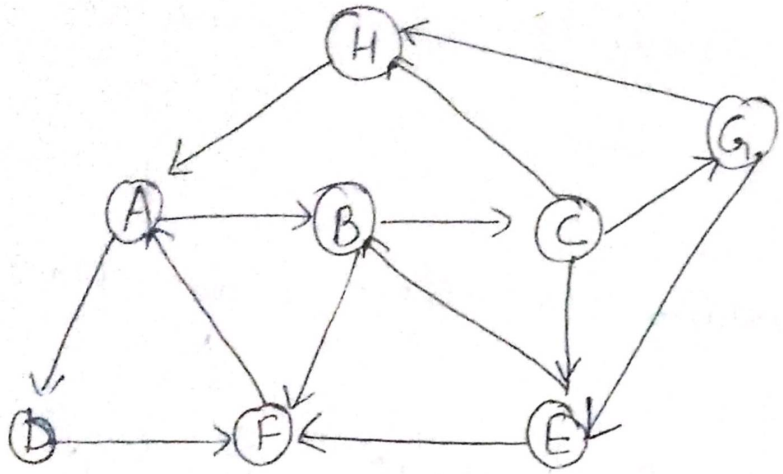


After coming to a dead end, that is, to the end of path P, we backtrack on P until we can continue along another path

Algorithm (DFS) :-

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push starting node A onto stack.
3. Repeat step 4 & 5 until STACK is empty.
4. Pop the top node N of STACK. Process N & change its status to the processed (STATUS = 3)
5. Push onto the STACK all the neighbours of N that are in ready state & change their status to waiting (STATUS=2) [End of step 3 loop]
6. Exit

Example : consider the following graph :-

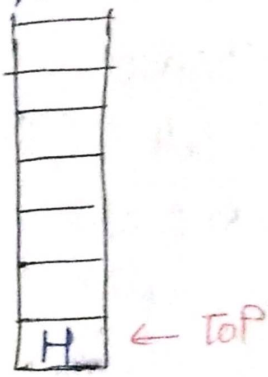


Adjacency List

A: B, D
B: C, F
C: E, G, H
D: F
E: B, F
F: A
G: E, H
H: A

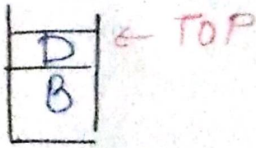
Step 1 :- Lets Take initial vertex as H

Now pop the top element of the stack i.e; H, print it & push all the neighbours of H onto the stack that are in ready state.



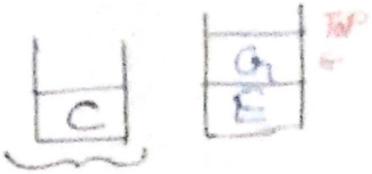
Step 2 :- Pop the top element i.e., A, print it & push all the neighbours of A that are in ready state

Step 3 :- Pop the top element i.e, D, print it & push all the neighbours of D i.e, F.



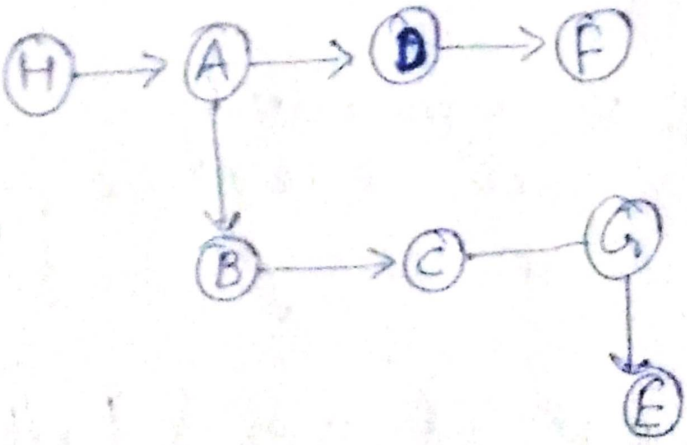
Step 4 :- Pop F & push its neighbours but there is no neighbour in ready state. Now pop B & push all its neighbours in ready state.

Step 5 :- Pop C & push E, G

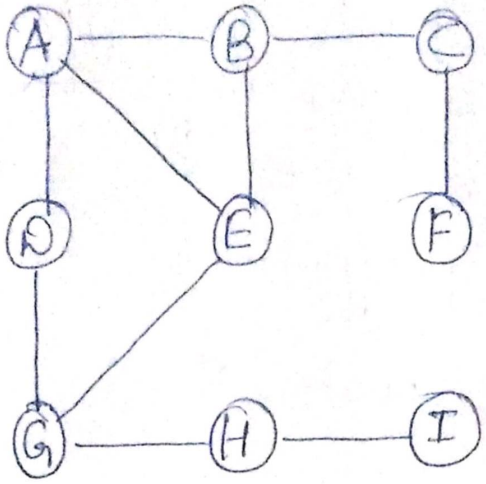


Step 6 :- Pop G & push all its neighbours. but neither E nor H is in ready state.

Step 7 :- Thus the final spanning tree will be :-



Example :-

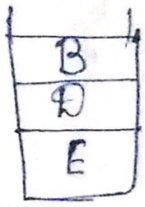


DFS

A: B, E, D
B: A, C, E
C: F
D: A, G
E: A, B, G
F: C
G: D, E, H
H: G, I
I: H

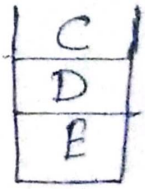
Step 1 : Start with A :

Step 2 :- Pop A & push E, D, B



Step 3 :- Pop B & push ~~E~~ as A & ~~E~~. status is 3 now not 1.

Step 4 :- Pop C & Push F

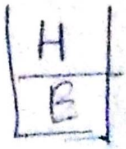


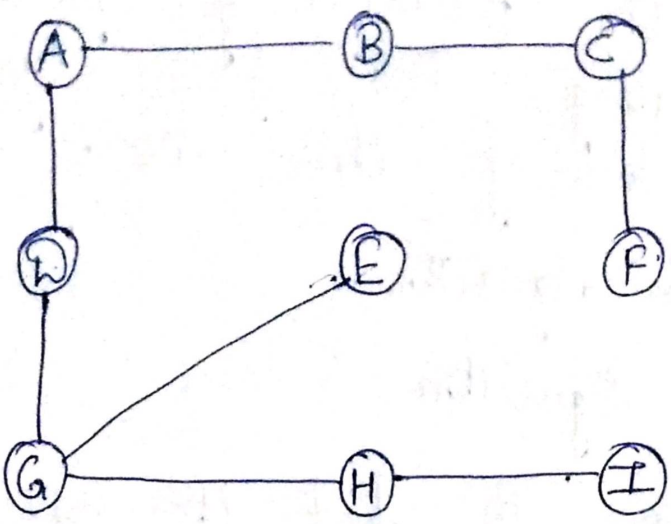
Step 5 :- Pop F, Now nothing is left to push as neighbours of F as C is already visited.

Step 6 :- pop D and Push G

Step 7 :- pop G & push H

Step 8 :- pop H & push I



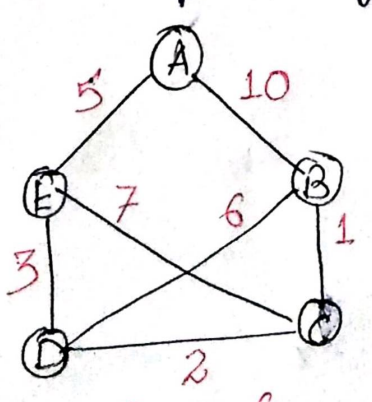


Minimum Spanning Tree :-

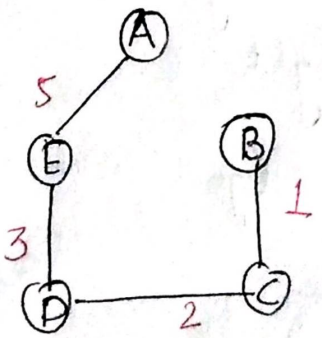
* A spanning tree of a graph G is a subgraph which is basically a tree & containing all the vertices of G but no circuit.

* A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight.

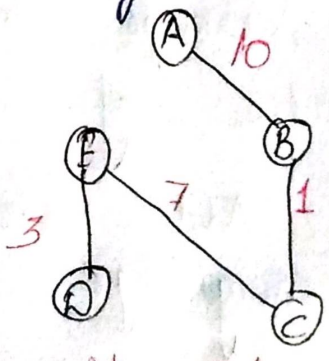
* Weight of the tree is defined as the sum of weights of all its edges.



Graph G.



wt = 11



wt = 21

* To solve the problem of finding minimum spanning tree, few algo. have been designed. Two of them are :-

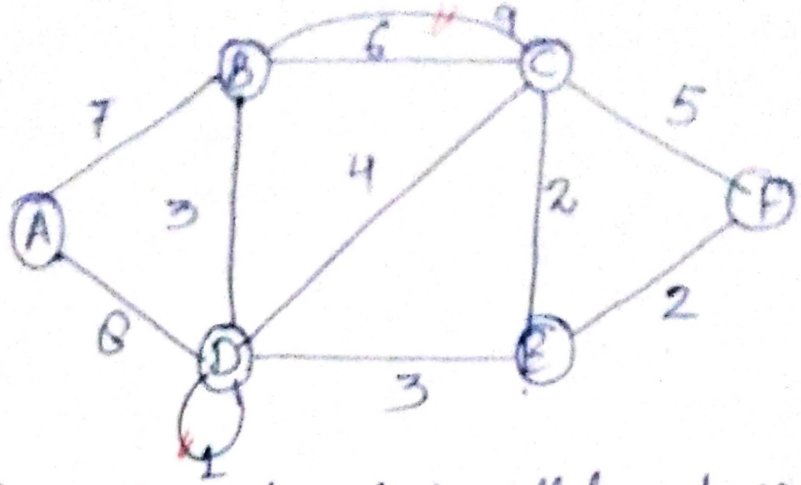
1) Kruskal's algorithm
2) Prim's algorithm

* Kruskal's algo. to find the minimum cost spanning tree uses the greedy approach.
* In this algo. an edge is selected in such a manner that it contains a min. weight & upon adding that it doesn't include any cycle.

Steps for Kruskal's algo. :-

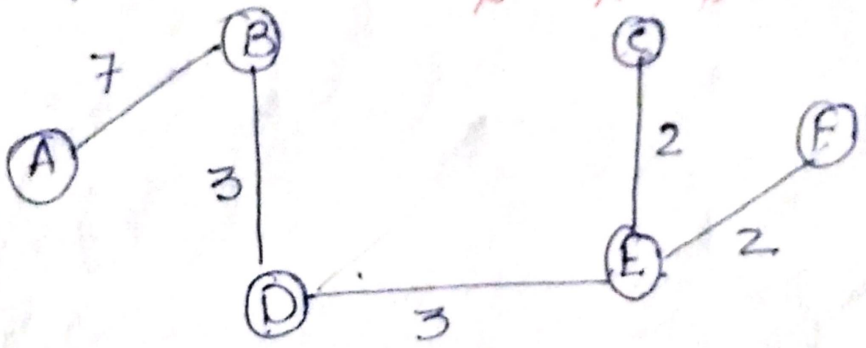
* Remove all the self loops.
* Remove all the parallel edges. Keep the one which has the least cost associated and remove all others.
* Arrange all edges in their increasing order of weight.
* Add the edge which has the least weightage

Example :



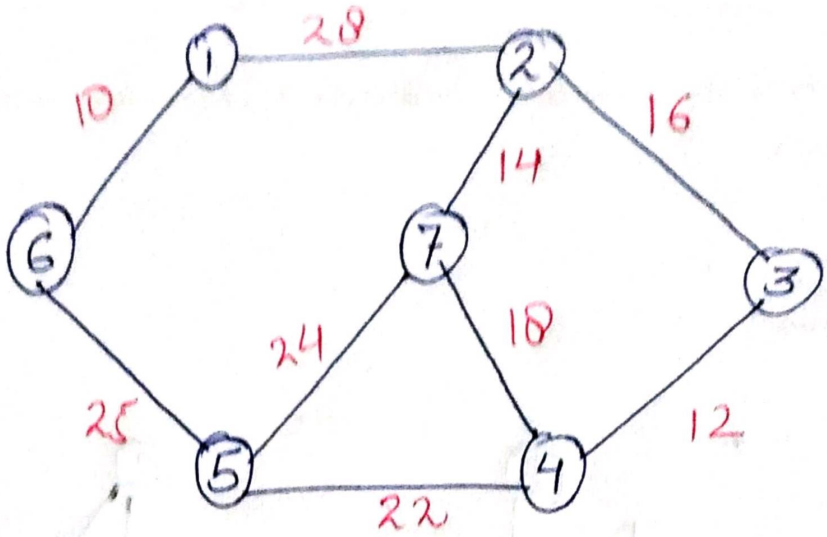
Remove all self loops & parallel edges.

C,E	E,F	D,E	D,B	C,D	C,F	B C	A,B	A,D
2	2	3	3	4	5	6	7	8
				X	X	X		X

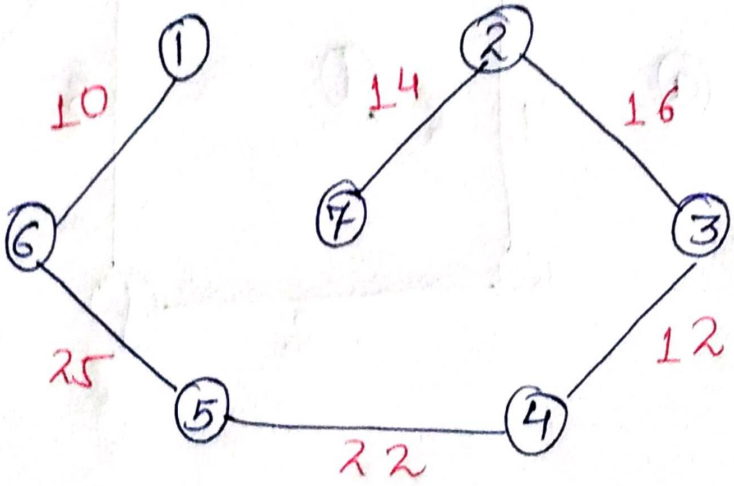


Practice Questions on Kruskal's / Prim's

Q(1)

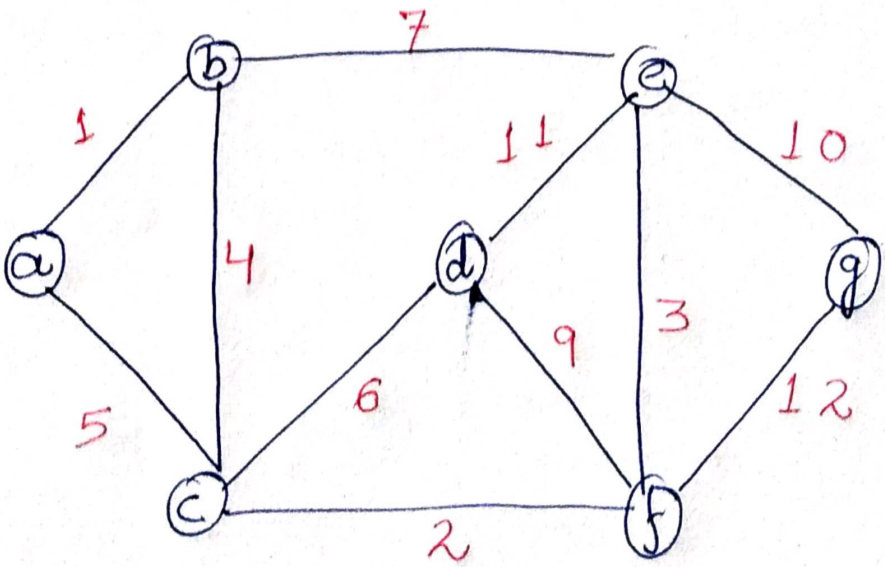


Ans

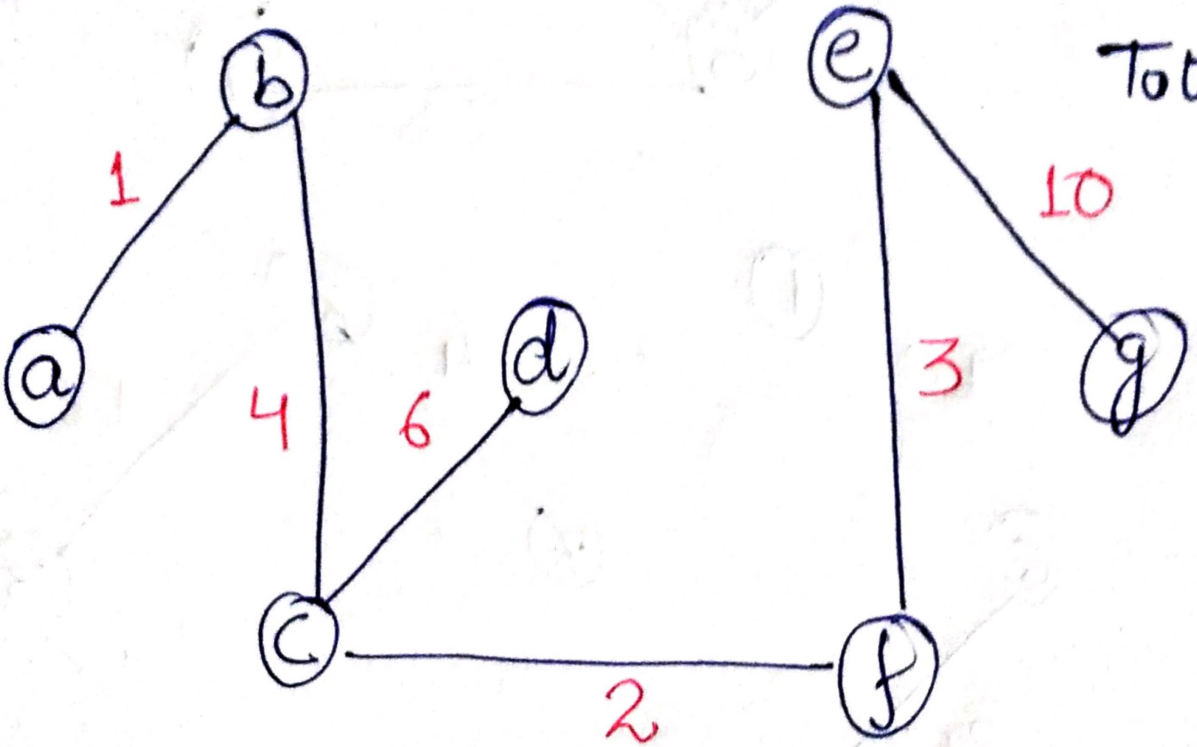


Total = 99

Q(2)

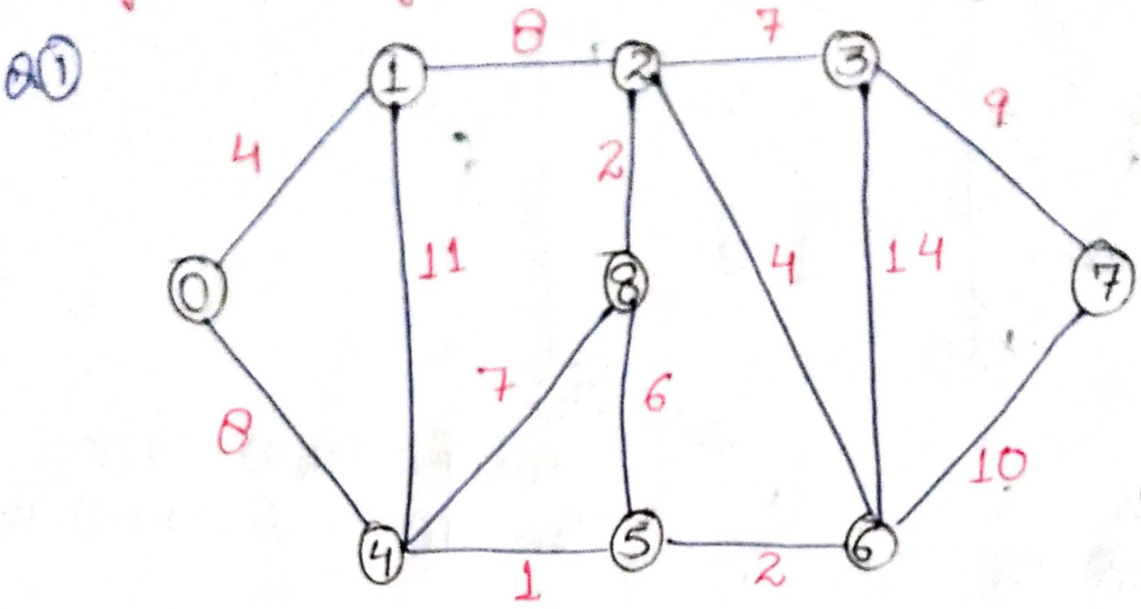


Ans



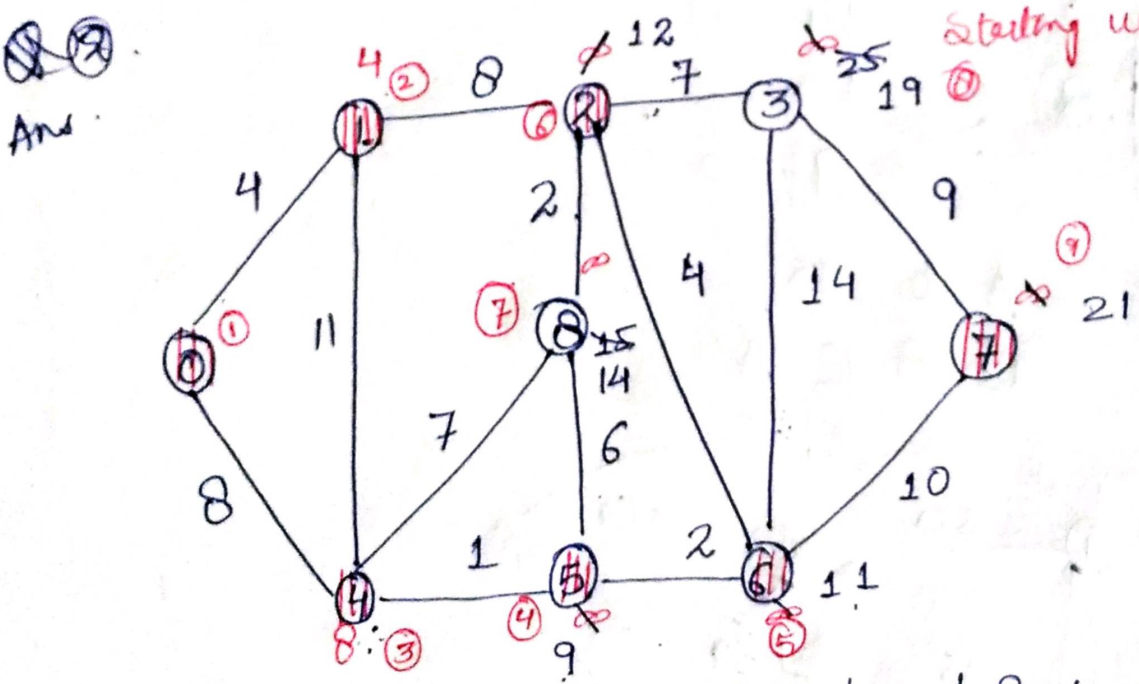
Total = 26

Dijkstra Algorithm



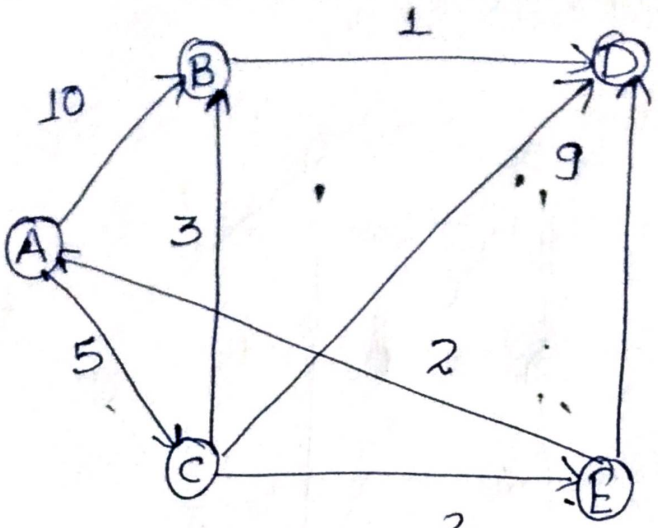
And

Starting vertex ⓪



	1	2	3	4	5	6	7	8
0	4	12	19	8	9	11	21	14

Q 2 :-



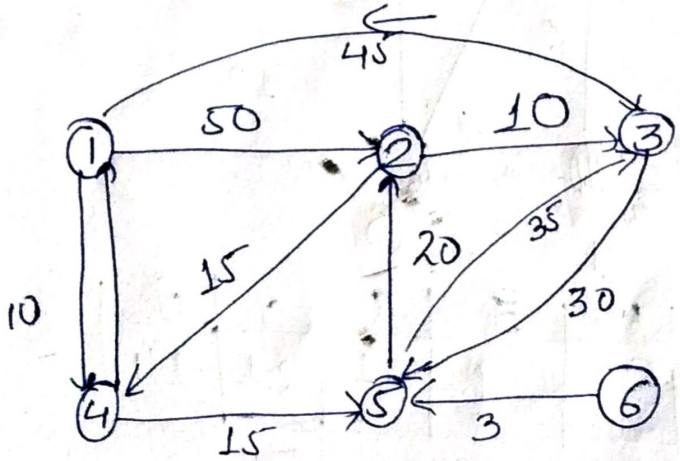
A	B	C	D	E
C	10	5	∞	∞
E	8		14	7
B	8		13	
D			9	

$d(u) + c(u,v) < d(v)$
then $d(v) = d(u) + c(u+v)$

Path :- A to D

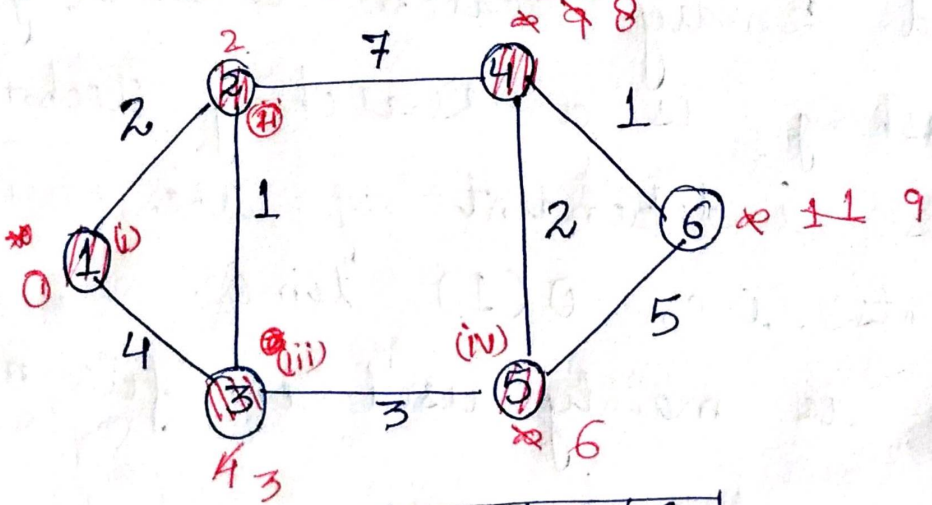
D B C A

Q 3 :-



1	2	3	4	5	6
4	50	45	10	∞	∞
5	50	45	10	25	∞
2	45	45	10	25	∞
3	45	45	10	25	∞
6	45	45	10	25	∞

Q 4:-

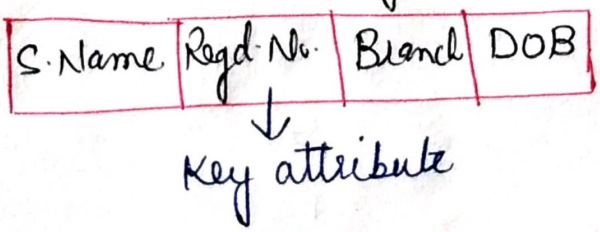


	2	3	4	5	6
1	2	3	8	6	9

Hashing :-

* This searching technique is independent of the no. of elements.
* As in searching, linear search takes O(n) time and Binary Search takes O(lg n) time but hashing is a searching technique which is independent of the number of elements. i.e; O(1) time.
* Hashing is mainly used in file management.

Note :- File : collection of several similar records
Records :- Collection of attributes or fields.



* Suppose F is a file of n records with a set K of keys which can uniquely identify the records in F.
* Let the file F is stored in the memory by means of a table T of m memory locations & L is the set of memory addresses for the locations in T.

* K is the set of n Keys.

⇒ The hash function H is a function from set K of Keys to the set L of memory addresses.

$$H : K \rightarrow L$$

Hash Function :-

Two major criteria to select a hash function are;

(i) It should be very easy and quick to compute.

(ii) It should uniformly distribute the hash addresses throughout the set L so that there are a minimum no. of collisions.

* There are three popular hash function :-

1) Division Method
2) Midsquare "
3) Folding "

① **Division Method** :- Choose a number m larger than the number n of keys of K. (The no. m is usually chosen to be a prime no. or a no. without small divisors, since this frequently minimizes the no. of collisions)

* The hash function is defined by,

$$H(k) = k \pmod m \quad \text{or}$$

$$H(K) = K \pmod m + 1$$

* Here $k \pmod m$ denotes remainder when k is divided by m. The second formula is used when we want the hash addresses to range from 1 to m rather than 0 to m-1.

② **Midsquare Method** :- The key k is squared. Then the hash function H is defined by;

$$H(k) = l$$

* where l is obtained by deleting digits from both ends of k^2.
* We emphasize that the same positions

of k^2 must be used for all of the keys.

(3) Folding Method :- The key k is partitioned into a no. of parts $k_1, k_2 \text{ —— } k_r$, where each part, except possibly the last has the same no. of digits as the required address. Then the parts are added together ignoring the last carry. i.e.,

$$H(k) = k_1 + k_2 + \text{ — } k_r$$

where the leading digit carries, if any are ignored. Sometimes for extra "milling", the even no. parts, k_2, k_4 — are each reversed before addition.

Example :-

Consider the company in following example, each of whose 68 employees is assigned a unique 4-digit employee number. Suppose L consists of 100 two-digit addresses 00, 01, 02 — 99. We apply the above hash functions to each of the following employee numbers.

3205, 7148, 2345

(a) **division method :-** lets choose $m = 97$ then

$H(3205) = 4$ [3205 mod 97 = 4]

$H(7148) = 67$ [7148 mod 97 = 67]

$H(2345) = 17$ [2345 mod 97 = 17]

Note :- In case the memory addresses begin with 01 rather than 00, we choose the f^n $H(k) = k \pmod m + 1$ to obtain

$H(3205) = 5$, $H(7148) = 68$, $H(2345) = 18$

(b) **Mid Square Method :-**

K :	3205	7148	2345
K^2 :	10272025	51093904	5499025
H(K) :	72	93	99

Note :- counting from R to L, 4th & 5th digits are chosen.

(c) **Folding Method :-** Chopping the key k into two parts & adding yields the following hash address.

$H(3205) = 32 + 05 = 37$

$H(7148) = 71 + 48 = 19$ (1 is ignored)

$H(2345) = 23 + 45 = 68$

alternatively one may want to reverse the second part before adding:

$H(3205) = 32 + 50 = 82$
$H(7148) = 71 + 84 = 52$
$H(2345) = 23 + 54 = 77$

Collision Resolution :-

* Suppose we want to add a new record R with key k to our file F. but suppose the memory location address H(k) is already occupied. This situation is called collision.

* Collisions are almost impossible to avoid. e.g., Suppose a student class has 24 students & suppose the table has space for 365 records.

* One random hash function is to chose the students birthday as the hash address. although load factor $\lambda = 24/365 \approx 7\%$. is very small.

Note :- The ratio of no. n of keys in K (which is the no. of records in F) to the no. m of hash addresses in L. $\lambda = n/m$ is called as load factor.

* The efficiency of a hash function with a collision resolution procedure is measured by the average no. of probes (key comparisons) needed to find the location of the record with a given key k.
* Specifically we are intrested in two quantities :-

$S(\lambda)$ = Avg. no. of probes for successful search.
$U(\lambda)$ = Avg. no of probes for unsuccessful search.

(1) Open Addressing : (a) Linear Probing

* Suppose that a new record R with key k is to be added to the memory table T, but the memory location with hash address $H(k) = h$ is already filled, One way to resolve the collision is to assign R to the first available location following $T[h]$.

Note :- We assume the table T with m locations is circular, so $T[1]$ comes after $T[m]$.

Accordingly, with such a collision procedure we'll search for the record R in the in the table T by linearly searching the locations T[h], T[h+1] - — until finding R or an unsuccessful search. This procedure of collision resolution is called linear probing.

e.g.; A table has 11 memory locations T[1], T[2] - — T[11]

Suppose F consists of 8 records & hash addresses

A,	B,	C,	D,	E,	X,	Y,	Z
4,	8,	2,	11,	4,	11,	5,	1

Suppose 8 records are entered into the table in the above order. Then file F will appear in memory as follows:

Table T :-

X	C	Z	A	E	Y		B		D	
1	2	3	4	5	6	7	8	9	10	11

Address :-

Q:- 3, 2, 9, 6, 11, 13, 7, 12 (use division method)

$h(k) = 2k + 3$

$m = 10$

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3

Key	location		probes
3	(2×3+3) % 10	9	1
2	(2×2+3) % 10	7	1
9	(2×9+3) % 10	1	1
6	(2×6+3) % 10	5	1
11	(2×11+3) % 10	5	2
13	(2×13+3) % 10	9	2
7	(2×7+3) % 10	7	2
12	(2×12+3) % 10	7	6

(b) **Quadratic Probing** :- The main disadvantage of linear probing is that the records tend to cluster i.e., appear next to one another. To minimize clustering quadratic probing is used.

$(u + i^2)$ % n where i = 0 to m-1

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

for 11 :-
(5 + 1) % 10 = 6

for 13 :-
$(9+1^2)$ % 10
= 0

for 7 :-
$(7+1^2)$ % 10 = 8

Key	location		probes
3	(2×3+3) % 10	9	1
2	(2×2+3) % 10	7	1
9	(2×9+3) % 10	1	1
6	(2×6+3) % 10	5	1
11	(2×11+3) % 10	5	2
13	(2×13+3) % 10	9	2
7	(2×7+3) % 10	7	2
12	(2×12+3) % 10	7	5

(c) **Double Hashing** :- Here a second Hash function H' is used for resolving a collision. As;

Suppose a record R with key k has the Hash addresses $H(k) = h$ and $H'(k) = h' \neq m$. Then we linearly search the location with addresses.

$$h,\ h+h',\ h+2h',\ h+3h' \ldots$$

Note : if m is a prime no., then the above sequence will access all the locations in the table T.

Chaining

* Chaining involves maintaining two tables in memory. First of all, as before, there is a table T in memory which contains the records in F except that T now has an additional field LINK, which is used so that all records in T with the same Hash address h may be linked together to form a linked list

Example :- Suppose we have 8 records with the following hash addresses.

Record :	A	B	C	D	E	X	Y	Z
H(k) :	4	8	2	11	4	11	5	1