

mood-book



UNIT-5

UNIT-4 GRAPHS

Defn:- A graph is a non-linear datastructure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are referred as arcs.

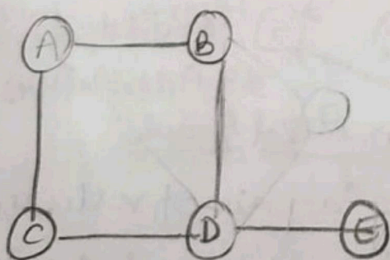
⇒ More formally a graph can be defined as $G = (V, E)$

→ A Graph is a pair of sets (V, E) , where V is the set of vertices & E is the set of Edges,

Connecting the pair of Vertices (nodes).

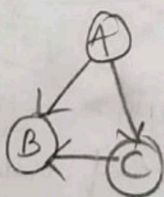
⇒ Graphs are used to solve many real-life problems. Graphs are used in social networks like Facebook, LinkedIn etc., for example, in Facebook, each person is represented with a Vertex (node). Each node is a structure and contains information like user id, username, genders etc.,

⇒ Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network



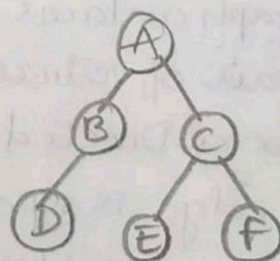
Graph 1

Set representation of Gr-1
 $V = \{A, B, C, D, E\}$
 $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$



Graph 2

$V = \{A, B, C\}$
 $E = \{(A, B), (A, C), (C, B)\}$



Graph 3

$V = \{A, B, C, D, E, F\}$
 $E = \{(A, B), (A, C), (B, D), (C, E), (C, F)\}$

Basic concepts on Graphs:-

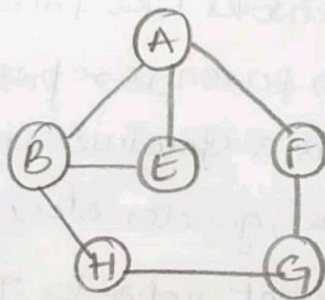
Vertex:- Each node of a graph is represented as a vertex. In the following example, the labeled circles represent vertices. ^{i.e., A to G}

Edge:- Edge represents a path between two vertices (or) a line between two vertices.

⇒ In the following ex A to B, B to C etc..

Adjacency:- Two nodes or vertices are adjacent if they are connected to each other through an edge.

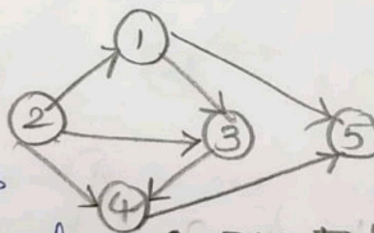
EX:- E is adjacent to B, A is adjacent to F etc..



Types of Graphs:-

① Directed Graph:

⇒ If a graph contains ordered pair of vertices, is said to be a Directed graph.



Set of Vertices $V = \{1, 2, 3, 4, 5\}$

Set of Edges $E = \{(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)\}$

Fig: Directed Graph

⇒ If an edge is represented using a pair of vertices (V_1, V_2) , the edge is said to be directed from V_1 to V_2 but not V_2 to V_1 .

⇒ In the pair (V_1, V_2)

where $V_1 \Rightarrow$ is called the start vertex. $\$$

$V_2 \Rightarrow$ the end vertex.

(2)

② Undirected Graph:-

⇒ If a graph contains unordered pair of vertices, is said to be undirected graph.

⇒ In this graph, pair of vertices represents the same edge

Ex:- $(1,2) \Leftrightarrow (2,1)$. $\textcircled{1} - \textcircled{2}$

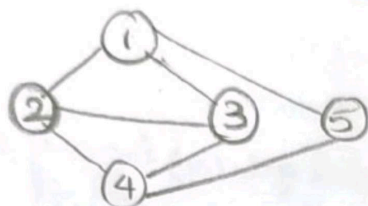


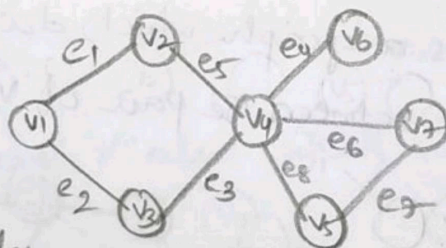
fig undirected graph

Set of vertices $V = \{1, 2, 3, 4, 5\}$
Set of edges $E = \{(1, 2), (1, 3), (1, 5),$
 $(2, 1), (2, 3), (2, 4), (3, 1), (3, 2),$
 $(3, 4), (4, 5), (4, 2), (4, 3),$
 $(5, 1), (5, 4)\}$

⇒ In an undirected graph, the nodes are connected by undirected arcs.

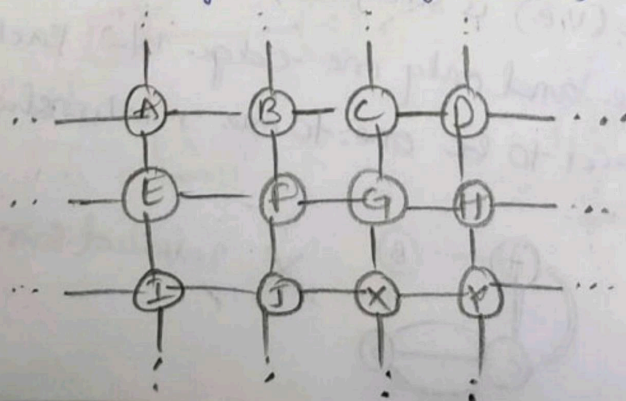
③ Finite Graph:-

A graph $G = (V, E)$ is said to be finite in case the number of vertices & edges in the graph is finite (or) countable in number.



④ Infinite graph:-

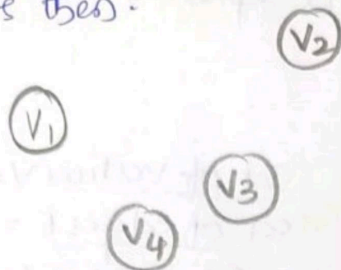
A graph $G = (V, E)$ is said to be infinite in case the number of edges & vertices in graph is infinite in number (not countable)



⑤ Null Graph:-

A graph $G=(V,E)$ is said to be a null graph in case there is a number of vertices exist but no edge exists that connects them.

Ex:-



(A graph having no edges (or) having only the isolated vertices)

⑥ Trivial graph:-

and no edge.

A graph with only one vertex, is called a Trivial Graph.

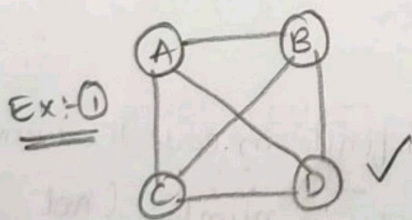
Ex:-



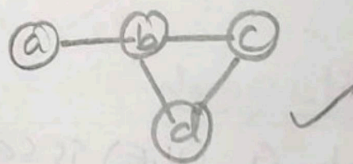
In the example, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.

⑦ Simple Graph:-

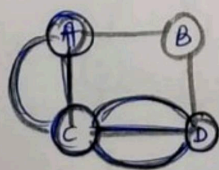
A simple Graph is a graph which does not contain more than one edge between the pair of vertices.



Ex: ②

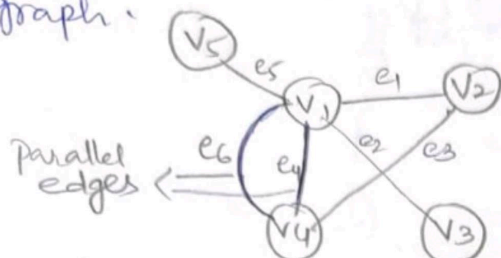


(or) A graph $G=(V,E)$ is said to be a simple graph, where there is one and only one edge b/w each pair of vertices, which also said to be one to one relationships b/w 2 elements.



✗ invalid simple graph

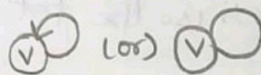
⑧ Multi Graph:- A graph which contains some parallel edges but doesn't contain any self-loop is called multi graph.



(Ex: Road map) $\rightarrow$ many routes one destination

Parallel edges: If 2 vertices are connected with more than one edge then such edges are called parallel edges. i.e., many routes but one destination.

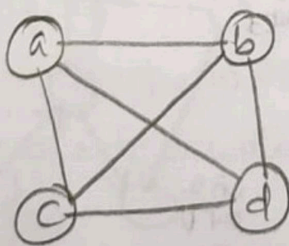
Loop:- An edge which joins to a same vertex or to itself is called a loop or a self-loop.



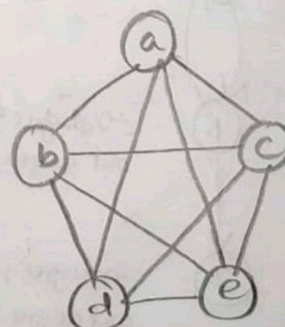
⑨ Complete Graph:-

A simple graph with n vertices is called a complete graph if the degree of each vertex is $n-1$. i.e., one vertex is attached with $(n-1)$ edges.

A complete graph is also called full graph.

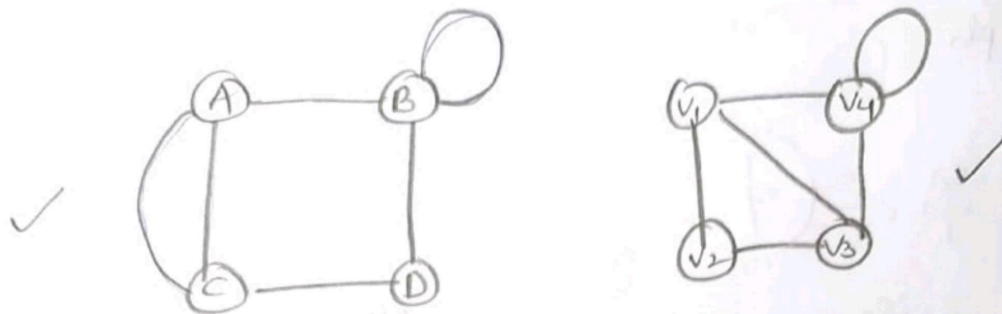


Ex: ①



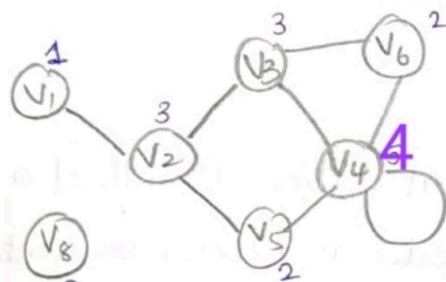
Ex: ②

⑩ Pseudo graph:- A graph G with self loop^{or self edges} and may also have some multiple edges is called pseudo graph.

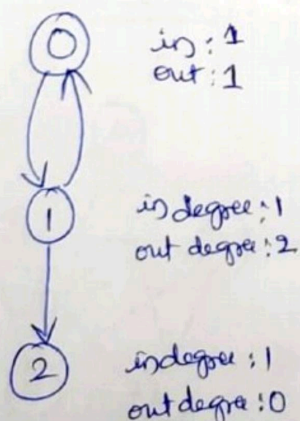


Degree of a vertex:- Tot no. of edges linked to a vertex is called its degree.

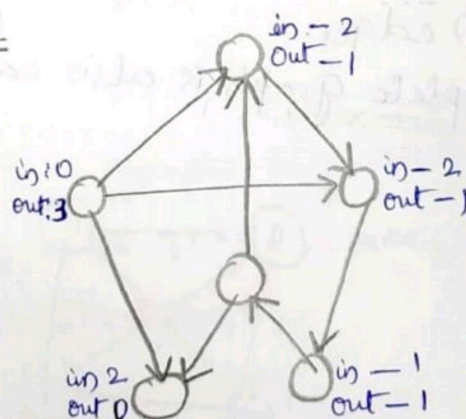
Indegree:- The indegree of a vertex is the tot. no. of edges coming to that node.
outdegree:- is the tot no. of edges going out from that node.



Ex: ①



Ex: - ②

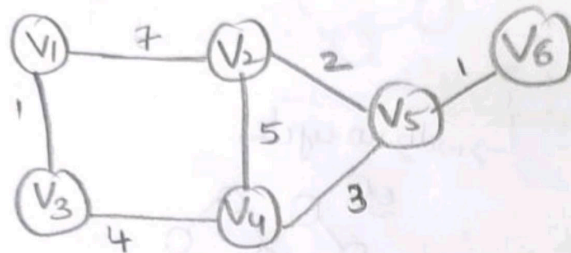


Ex: ③

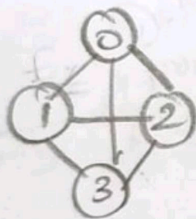
(4)

⑪ Weighted graph (or) Labelled graph:-

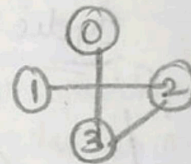
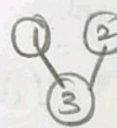
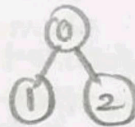
A graph $G=(V,E)$ is said to be a labeled or weighted graph, where each of the edges in the graph holds some weight or value that denotes the cost of traversal through that edge.



⑫ Subgraph:- A graph $G'=(V',E')$ whose vertices and edges are subsets of another graph $G=(V,E)$



Graph G

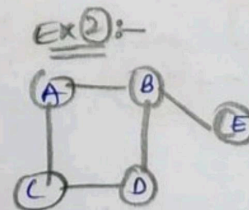
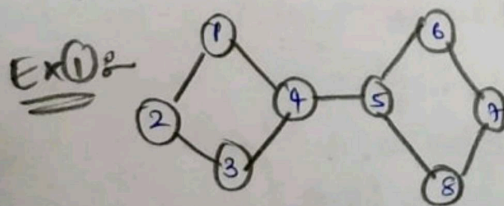


Sub graphs of G

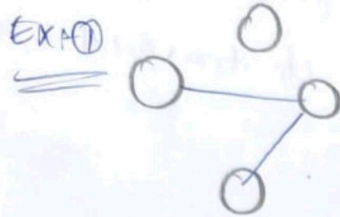
⑬ Connected graph

A connected graph is a graph in which it is possible to get from every vertex in the graph to every other vertex through a series of edges, called a path.

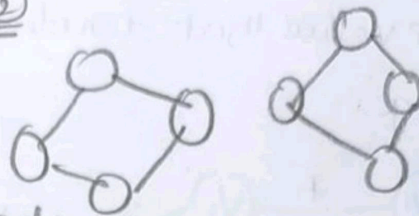
(or)
A graph is said to be connected if there exists a path b/w every pair of vertices v_i and v_j .



- (14) Disconnected graph:- is said to be disconnected if it is not connected. i.e., if there exist 2 nodes in such that no path in G has those nodes as endpoints.

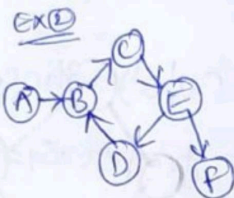
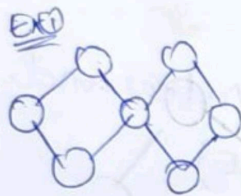


EX: (2)



- (15) cyclic and Acyclic graphs

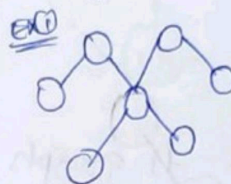
→ with atleast one cycle



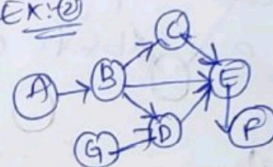
cyclic graph

Cyclic Graph is a directed graph which contains a path from at least one node back to itself.

→ with no cycles



EX: (2)

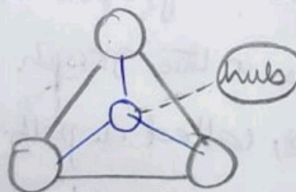


Acyclic graph

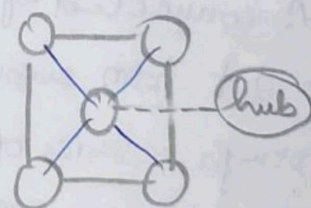
- (16) wheel graphs:-

A wheel graph is formed by connecting a single universal vertex to all vertices of a cycle.
(Hub)

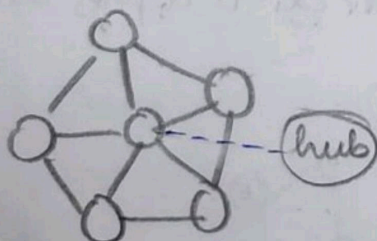
if $\rightarrow n$ - vertices
then $\rightarrow 2(n-1)$ - edges.



4 vertices
 $2(4-1) = 2(3) = 6$ edges



5 vertices
 $2(5-1) = 2(4) = 8$ edges



6 vertices
 $2(6-1) = 2(5) = 10$ edges

Topic 3: Graph Representation:-

The two most common ways of representing a graph is as follows.

- ① Adjacency Matrix.
- ② Adjacency List.

① Adjacency Matrix:-

⇒ An Adjacency Matrix is a square matrix used to represent a finite graph. The elements of the matrix indicates whether pair of vertices $V \times V$ adjacent or not in the graph.

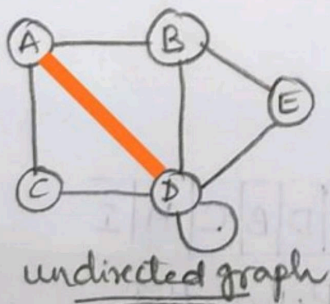
⇒ The graph is represented using a matrix of size total number of vertices by a total number of vertices.

for ex:- a graph with 4 vertices is represented using a matrix of size 4×4 .

⇒ In this matrix both rows and columns represent vertices this matrix is filled with either 0 or 1.

Here '1' represents that there is an edge from row vertex to col. vertex
'0' " " " " " no edge

EX:-①



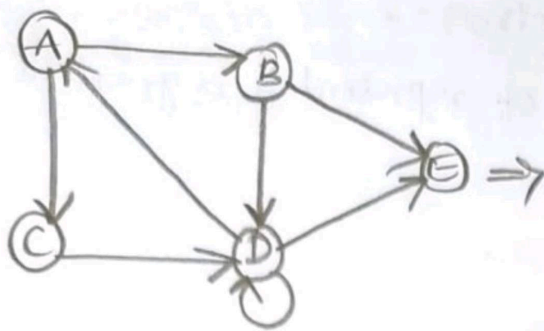
⇒

	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

Adjacency Matrix

Adjacency matrix of undirected graph is symmetric matrix which means an $\text{edge}(i, j) = \text{edge}(j, i)$.

EX:- An $\text{edge}(A, B) = \text{edge}(B, A)$
 $\text{edge}(B, E) = \text{edge}(E, B)$

Directed graph

	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 matrix

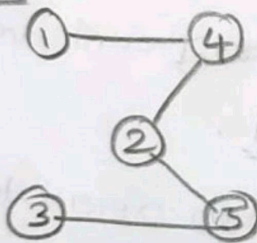
NOTE:- An Adjacency matrix of a directed graph is not symmetric $adj[i][j]=1$, indicated a directed edge from vertex i to vertex j .

$$edge(B, E) \neq edge(E, B)$$

Ex:- Adja. matrix of a directed graph is not symmetric.

If there is an edge (B, E) , there is not an edge (E, B) .

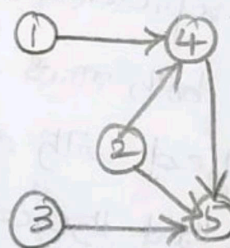
Ex:-2

undirected graph

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

Adjacency Matrix representation of undirected graph

Ex:-2

Directed graph

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

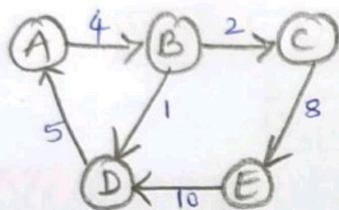
Adjacency Matrix representation of directed graph

(6)

Representation of Directed & undirected weighted graph:

⇒ Representation of weighted graph is different. Instead of filling the entry by 1, the non-zero entries of the matrix are represented by the weight of respective edges.

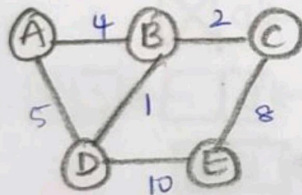
⇒ The weighted directed graph along with the adj. matrix representation is shown in the following figure.



weighted Directed graph

$$\begin{matrix}
 & \begin{matrix} A & B & C & D & E \end{matrix} \\
 \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \begin{bmatrix}
 0 & 4 & 0 & 0 & 0 \\
 0 & 0 & 2 & 1 & 0 \\
 0 & 0 & 0 & 0 & 8 \\
 5 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 10 & 0
 \end{bmatrix}
 \end{matrix}$$

Adjacency Matrix



weighted undirected graph

$$\begin{matrix}
 & \begin{matrix} A & B & C & D & E \end{matrix} \\
 \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \begin{bmatrix}
 0 & 4 & 0 & 5 & 0 \\
 4 & 0 & 2 & 1 & 0 \\
 0 & 2 & 0 & 0 & 8 \\
 5 & 1 & 0 & 0 & 10 \\
 0 & 0 & 8 & 10 & 0
 \end{bmatrix}
 \end{matrix}$$

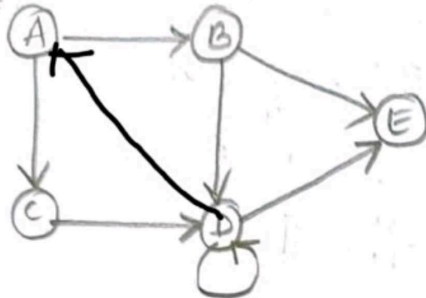
Adjacency Matrix

② Adjacency List:-

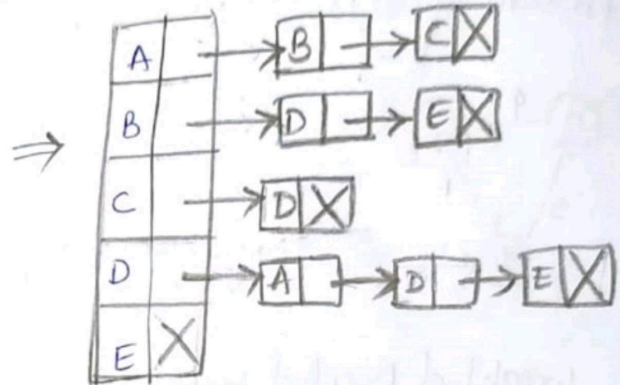
In this representation, every vertex of a graph contains list of its adjacent vertices.

for example, consider the following directed graph representation implemented using linked list.

Ex: ①

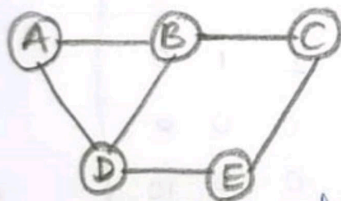


directed graph

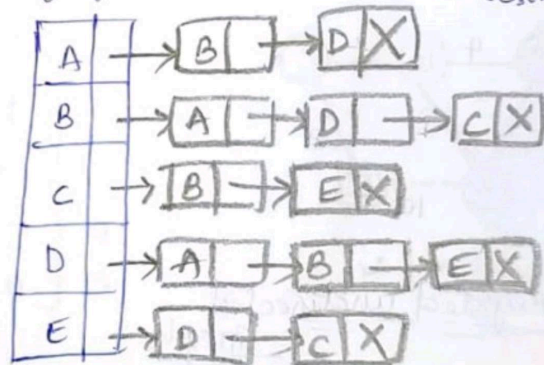


Adjacency list

Ex: ② Consider the following fig of undirected graph implemented using linked list



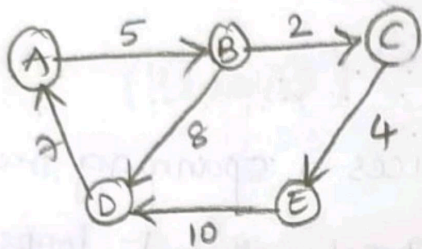
undirected graph



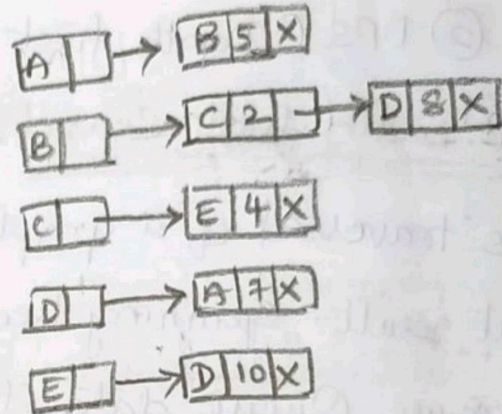
An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node.

If all the adjacent nodes are traversed then store to NULL in the pointer field of last node of the list.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The Adj List representation of a directed graph is shown in the following figure.



Weighted Directed graph



Adjacency List

Topic: Graph Traversals:-

There are two graph traversal techniques and they are as follows.

① BFS (Breadth first search)

② DFS (Depth first search)

① Breadth first search:- (BFS) (Queues)

BFS traversal of a graph produces a spanning tree as final result. Spanning tree is a graph without loops. we use Queue data structure with max size of tot number of vertices in the graph to implement BFS traversal.

Algm to implement BFS traversal:-

step ①:- Define a Queue of size tot. no. of vertices in graph.

step ②:- select any vertex as starting point for traversal. visit that vertex and insert it into the Queue.

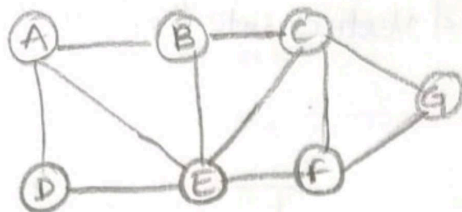
step ③:- Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

step ④:- When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

step ⑤:- Repeat step ③ and ④ until Queue becomes empty.

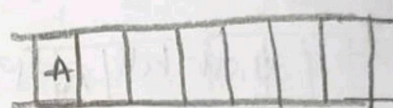
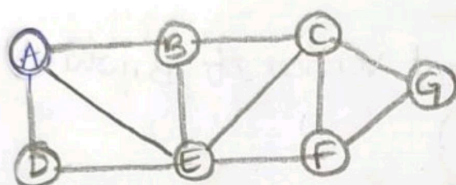
step ⑥:- when Queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

Consider the following ex: graph to perform BFS traversal



Step 1: Select A vertex as a starting point (visit A)

- Insert A into the Queue.



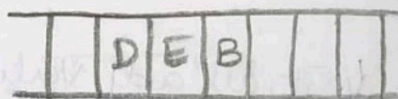
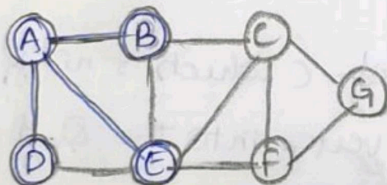
Queue

Step 2:-

- visit all adjacent vertices of A which are not visited (D, E, B).

- Insert newly visited vertices into the Queue and delete

A from the Queue.

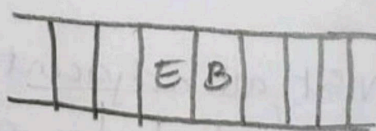
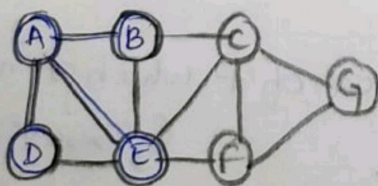


Queue

Step 3:-

visit all adjacent vertices of D, which are not visited (there is no vertex).

- Delete D from the Queue.

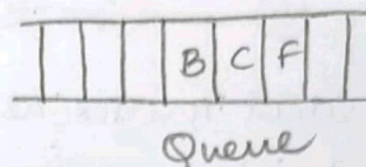
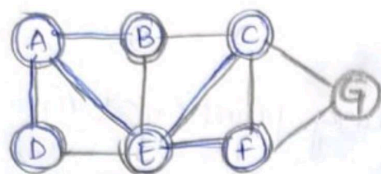


Queue

Result : A, D, E, B... (P.T.O)

Step 4:- Visit all adjacent vertices of E which are not Visited (C, F).

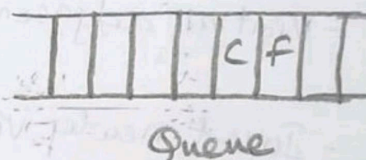
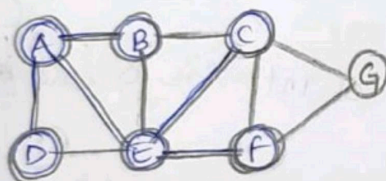
- Insert newly visited vertices into the Queue & delete E from the Queue.



Step 5:-

Visit all adjacent vertices of B, which are not Visited (there is no vertex).

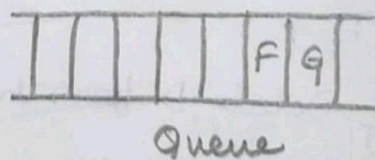
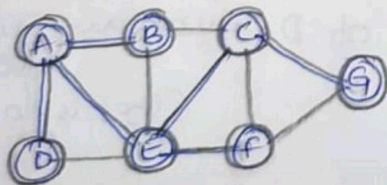
- Delete B from the Queue.



Step 6:-

visit all adj vertices of C which are not visited (G).

- Insert newly visited vertex into the Queue & delete C from the Queue.



Step 7:-

visit all adjacent vertices of F which are not visited

(there is no vertex)

- Delete F from the Queue.

(Same diagram) ↑

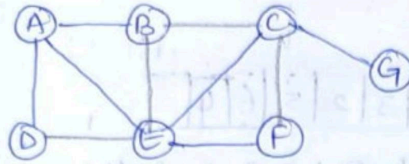


Result: A, D, E, B, C, F, G...

Step (8):-

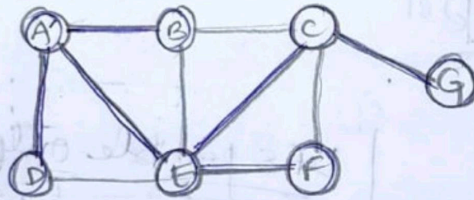
- visit all adjacent vertices of G which are not visited (there is no vertex).

Delete G from the Queue.



- Queue became empty. So, stop the BFS process.

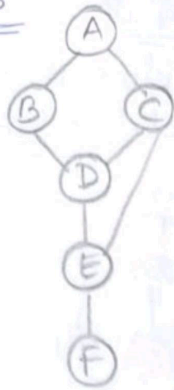
- final result of BFS is a spanning tree as shown below



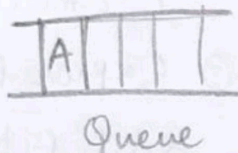
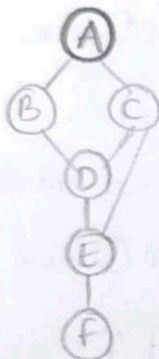
Ex 12

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

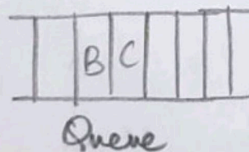
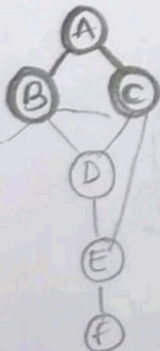
Ex:- ② BFS



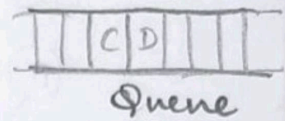
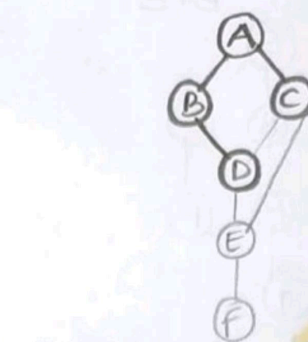
Step ①:- select A as start vertex
Insert A into Queue



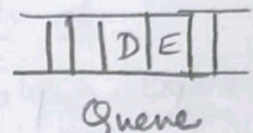
Step ②:- visit all non-visited adjacent vertices of A
→ Insert newly visited vertices into Queue & delete A from the Queue



Step ③:- Visit all non-visited adj. vertices of B & insert newly visited vertices into Queue & delete B from the Queue.

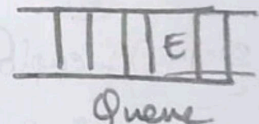


Step ④:- Visit all non-visited adj. vertices of C & insert newly visited vertices into Queue & delete C from Queue

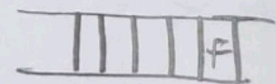


Step ⑤:- Visit all non-visited adj. vertices of D & insert newly visited vertices into Queue (there is no vertex) delete D from Queue.

(Same diagram)

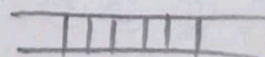


Step ⑥:- Visit all non-visited adj. of E (i.e. F) & delete E from Queue.



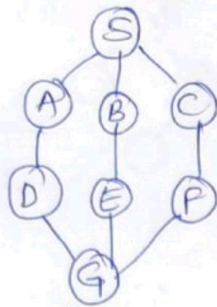
Step ⑦:- no new non-visited adj. vertex from F so, delete F from Queue

BFS → A, B, C, D, E, F (done)

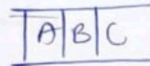
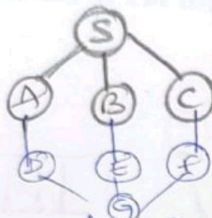


EX:- (3):-

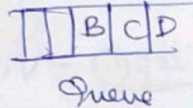
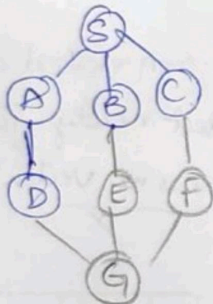
BFS



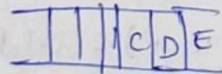
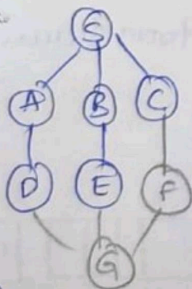
step 1:- start vertex is S is set
 step 2:- explore S i.e. (A, B, C) into Queue



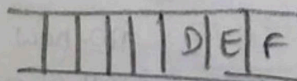
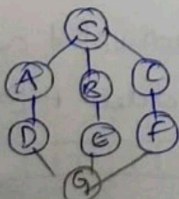
step 3:- explore A i.e. (D)
 delete A from Queue



step 4:- explore B i.e. (E)
 delete B from Queue

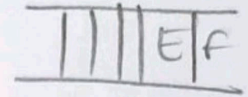
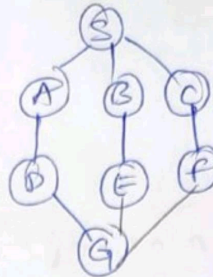


step 5:- explore C i.e. (F)
 delete C from Queue



step 6:-

Explore D i.e. (G)
 delete D from Queue



step 7:- Explore E (no unvisited vertex)
 delete E from Queue

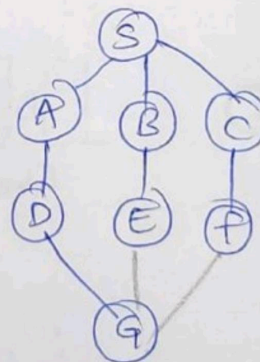
(Same Dgm)

step 8:- Explore F (no unvisited vertex)
 delete F from Queue

(Same Dgm) ↑

②

BFS:- S, A, B, C, D, E, F



Depth First Search (DFS):-

DFS traversal of a graph produces a spanning tree as final result. Spanning tree is a graph without loops. We use stack data structure with max. size of tot. no. of vertices in the graph to implement DFS traversal.

Algm:-

We use the following steps to implement DFS traversal.

Step ①:- Define a stack of size tot no. of vertices in graph.

Step ②:- Select any vertex as starting point for traversal.

Visit that vertex and push it on to the stack.

Step ③:- Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

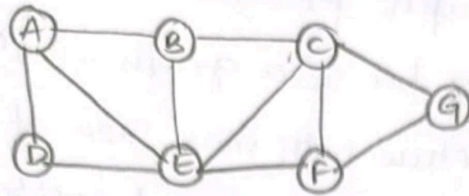
Step ④:- Repeat step ③ until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step ⑤:- When there is no new vertex to visit then use backtracking and pop one vertex from the stack.

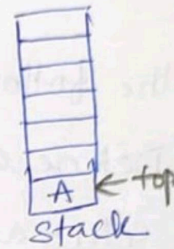
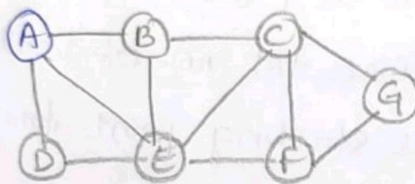
Step ⑥:- Repeat steps ③ ④ and ⑤ until stack becomes Empty.

Step ⑦:- When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph. Backtracking is coming back to the vertex from which we reach the current vertex.

Consider the following example graph to perform DFS traversal.



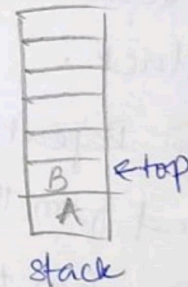
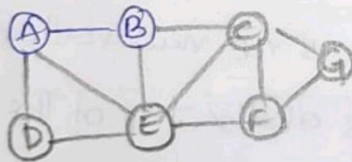
Step ①:- Select the vertex A as starting point (visit A)
Push A into the stack.



Step ②:-

Visit any adjacent vertex of A which is not visited (B).

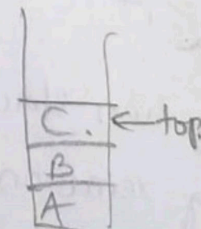
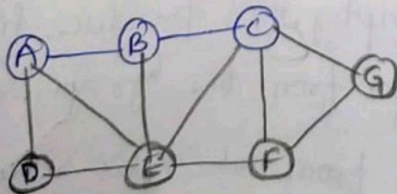
- Push newly visited vertex B on to the stack



Step ③:-

- Visit any adjacent vertex of B which is not visited (C).

- push C on to the stack

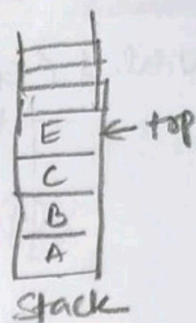
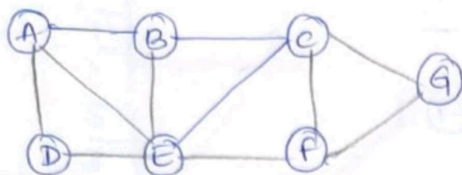


DFS Result: A, B, C, E, D, F, G

(11)

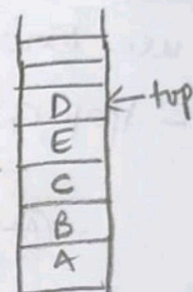
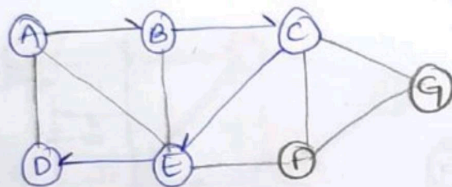
Step 4:- Visit any adjacent vertex of C which is not Visited (E)

- Push E on to the stack.



Step 5:- Visit any adjacent vertex of E which is not Visited (D).

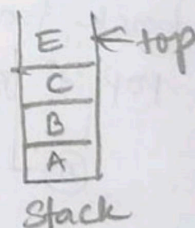
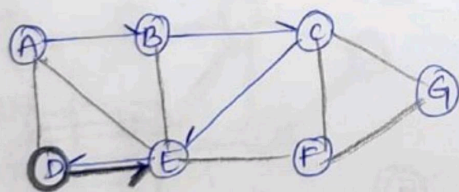
- Push D on to the stack.



Step 6:-

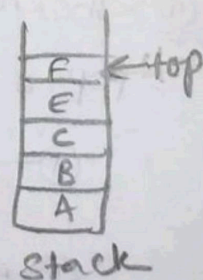
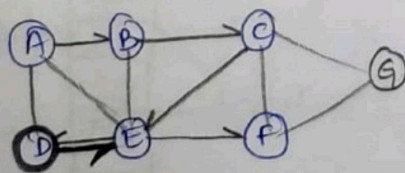
there is no new vertex to be visited from D.

So, we back track and pop D from the stack.



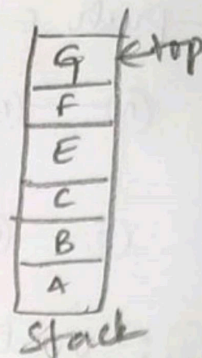
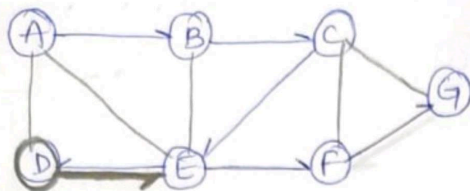
Step 7:- Visit any adjacent vertex of E which is not Visited (F).

- Push F into the stack



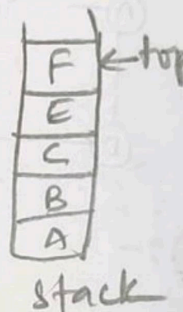
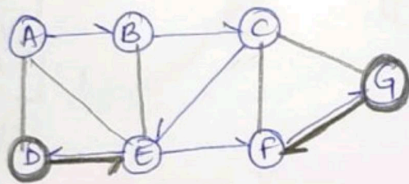
Step 8:- Visit any adjacent vertex of f which is not visited (G).

- push G on to the stack.



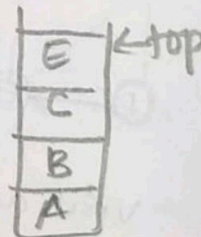
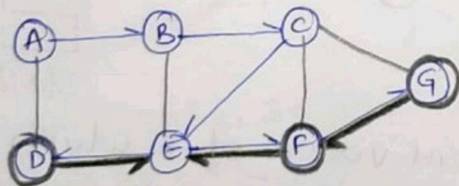
Step 9:- There is no new vertex to be visited from G . So use backtrack.

- Pop G from the stack.



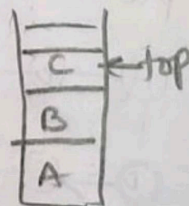
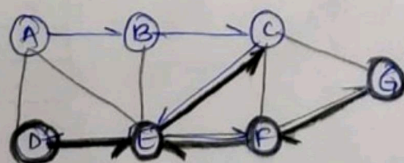
Step 10:- there is no new vertex to be visited from F . So use backtrack.

pop F from the stack.



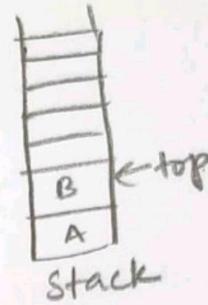
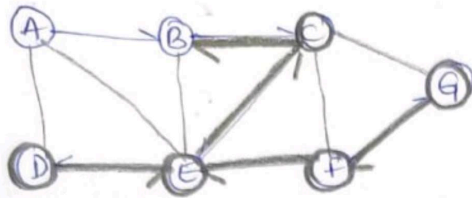
Step 11:- There is no new vertex to be visited from E . So use BS.

- Pop E from stack.



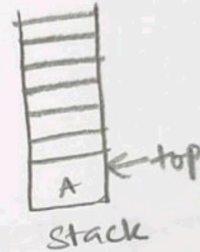
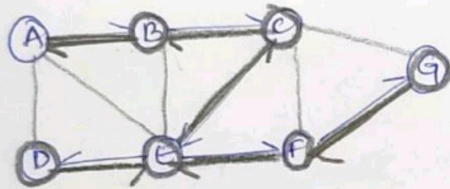
Step (12):- There is no new vertex to be visited from C, so, use Backtrack.

- Pop C from the stack



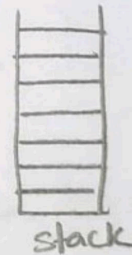
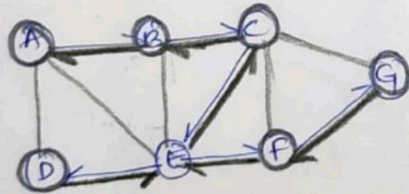
Step (13):- There is no new vertex to be visited from B, so, use Back track.

- Pop B from the stack.



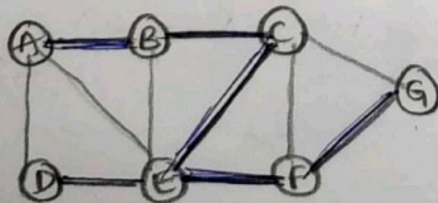
Step (14):- There is no new vertex to be visited from A, so, use Back track.

- Pop A from the stack.

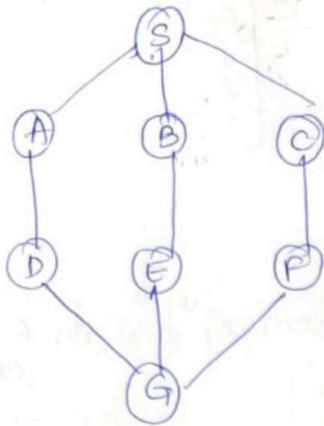


Stack became Empty, so stop DFS traversal.

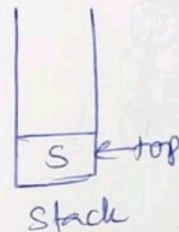
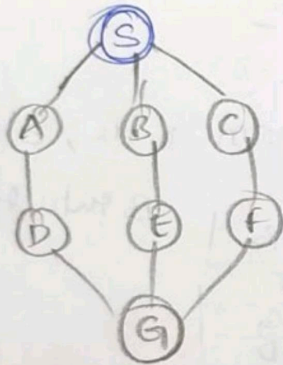
- final result of DFS traversal is following spanning tree.



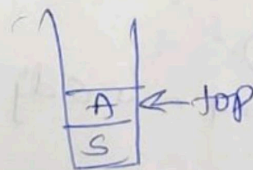
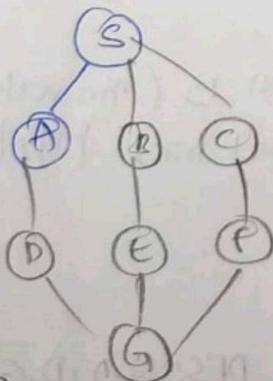
DFS Examples :-



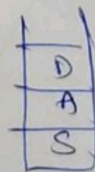
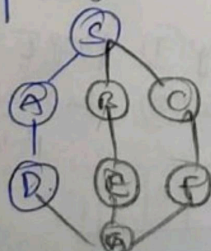
Step ① - select S as start vertex
push 'S' into stack



Step ② - Visit any adjacent vertex from S (ie, A)
push 'A' into stack
which is not visited.

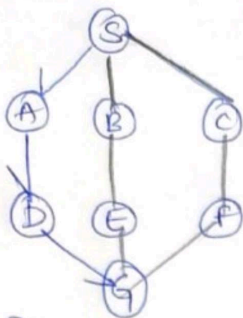


Step ③ - Visit any ^{new} adj vertex from A (ie, D) & push D into stack



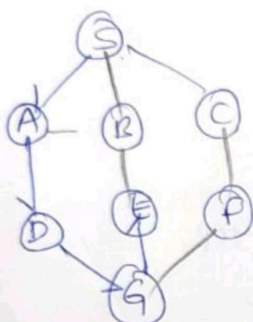
Ans: S, A, D... (P.T.O)

Step 4 - Visit new adj vertex of D i.e. G push G into stack



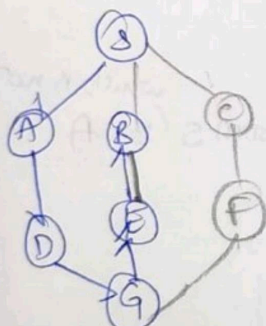
G
D
A
S

Step 5 - Visit any new adj vertex from G i.e. E push E into stack



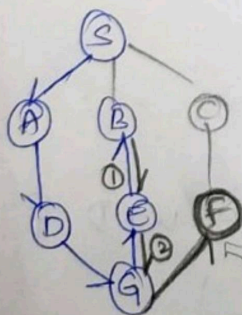
E
G
D
A
S

Step 6 - Visit any new adj vertex from E i.e. B push B into stack



B
E
G
D
A
S

Step 7 - Visit any new adj vertex from B (no vertex)
we Back track pop B from stack



E
G
D
A
S

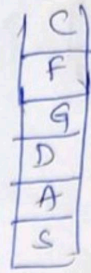
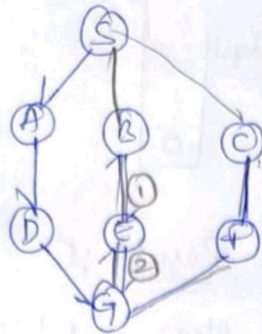
DFS: S, A, D, G, E, B, F...

Step 7 pop E (no new adj vertex)

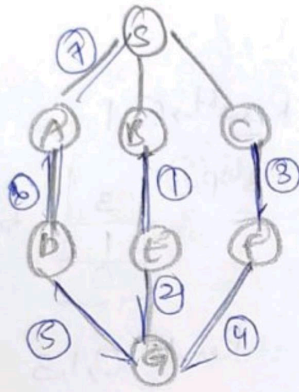
Step 8 new adj vertex from G i.e. F push F into stack

S
A
D
G
F

Step 9:- new adj vertex from F is C push C into stack.



Step 10:- No new adj vertex from C so pop C from stack (dgm 1)



~~S, A, D, G, E, B, F, C~~

Step 11:- pop F

(12) pop G

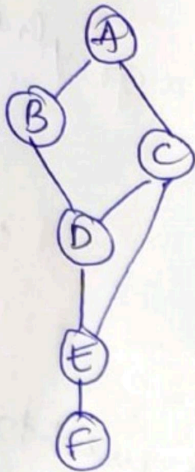
(13) pop D

(14) pop A

(15) pop S

DFS:- ~~S, A, D, G, E, B, F, C~~

→ S, A, D, G, E, B, F, C



BFS (Queue): Start vertex (A) $\rightarrow$ A, C, B, D, E, F

(or)

A, B, C, D, E, F

Start vertex (D)

D, E, C, B, F, A

(or)

E, D, C, F, B, A

...

DFS (Stack): SV(A) $\rightarrow$ A, B, D, E, C, F

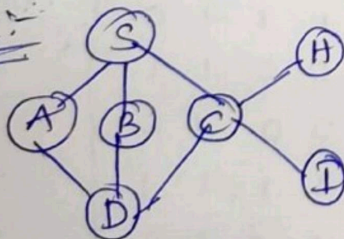
SV(D) $\rightarrow$ D, B, A, C, E, F

SV(C) $\rightarrow$ C, D, B, A, E, F

(BT)

...

Ex:-



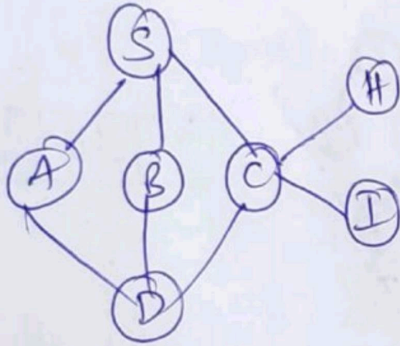
BFS (Queue): Start vertex (A) $\rightarrow$ A, S, D, B, C, H, I

" (H) $\rightarrow$ H, C, I, S, D, A, B

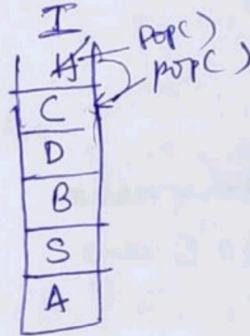
(P.T.O)

A, S, D, B, C, H, I

H, C, I, S, D, A, B

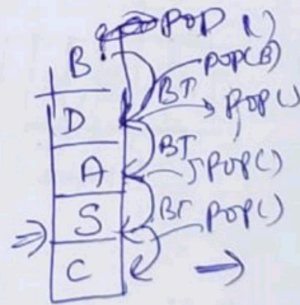
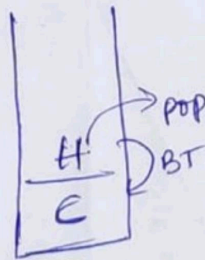


DFS (Stack): Start vertex (A):



Q1 Result:- A, S, B, D, C, H, I

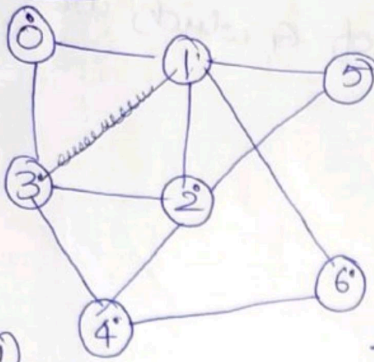
Start vertex (C) ⇒



Q2 Result:- C, H, S, A, D, B, I

Start vertex (B):

Example:-



output 1

BFS (Queue) : SV(0) → Result 0, 1, 3, 2, 5, 6, 4

DFS (stack) :

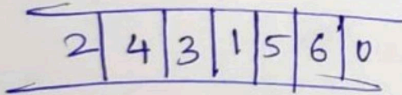


0, 1, 6, 4, 3, 2, 5

output 2

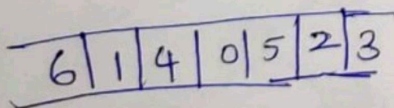
BFS possible ops:-

SV as (2)



BFS :- 2, 4, 3, 1, 5, 6, 0.

SV as (6)

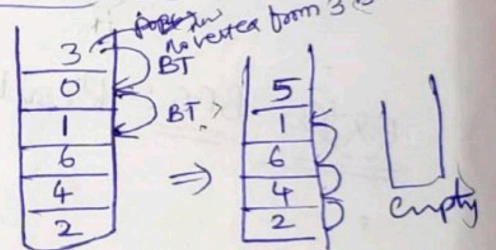


output 6, 1, 4, 0, 5, 2, 3

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

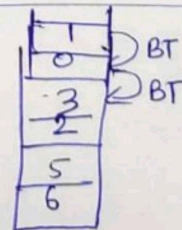
DFS possible outputs

SV as (2)



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

SV as (6)



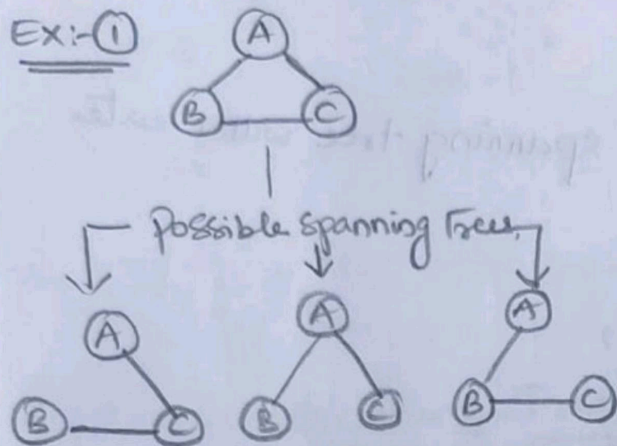
Result : 6, 5, 2, 3, 0, 1, 4

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

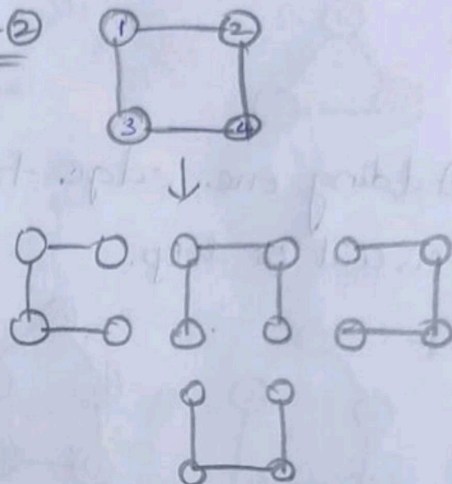
Spanning Tree:-

Def:- A graph, which contains all vertices with minimum number of edges.

Ex:-1



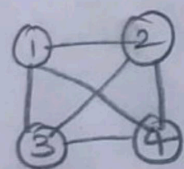
Ex:-2



Properties of Spanning Tree:-

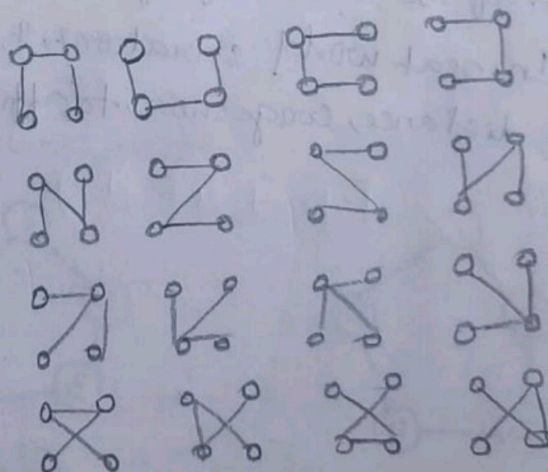
- ① The spanning tree does not have any cycles (loops)
- ② Spanning tree has $n-1$ edges, where n is no. of vertices
- ③ A complete graph can have maximum n^{n-2} no. of spanning trees

Ex:-



$$n^{n-2} \Rightarrow 4^{4-2} \Rightarrow 4^2 \Rightarrow 16$$

16 possible no. of spanning trees can be drawn from above graph



- ④ From a complete graph, by removing $\max^m e - n + 1$ edges, we can construct a spanning tree.

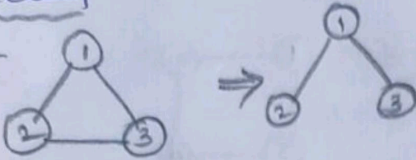
EX:- (1) $6 - 4 + 1 \Rightarrow 3$ edges.

- ⑤ A connected graph G can have more than one spanning tree.

(2)

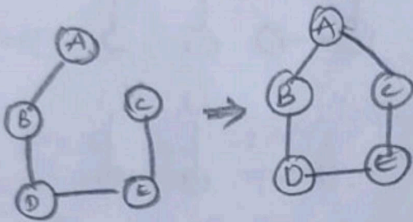
⑥ Removing one edge to the spanning tree will make the graph disconnected, i.e., the spanning tree is minimally connected.

EX:-



⑦ Adding one edge to the spanning tree will create a circuit or loop.

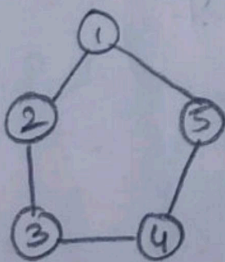
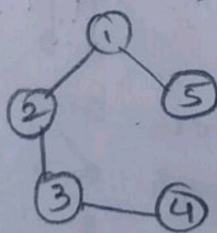
EX:-



Minimum Spanning Tree (MST):-

In a weighted, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

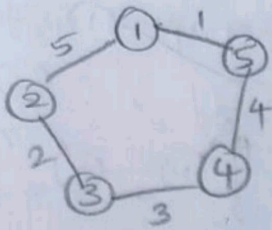
In real world situations, this weight can be measured as distance, congestion, traffic load etc.,
(overcrowding)


 $G = (V, E)$

 $G' = (V', E')$
spanning tree of
Graph G

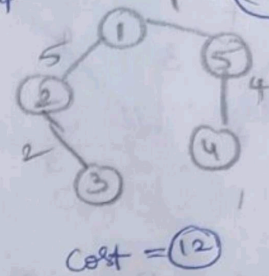
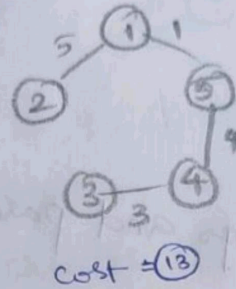
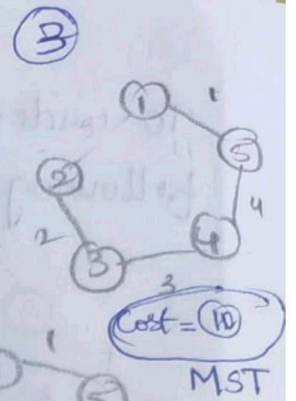
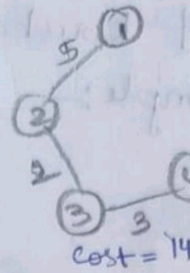
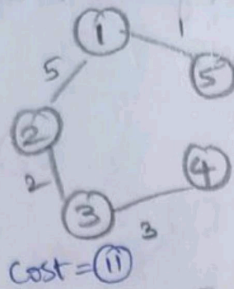
Relation between $V' \subseteq V$ is $V' = V$
i.e., no. of vertices in spanning tree (G')
equals to no. of vertices in original graph (G)

 $E' \subseteq E$
 $E' = |V| - 1$ Eg: $5 - 1 = 4$ edges

Ex:- for MST:-



Graph



Minimum spanning-Tree Algorithm:-

Two important spanning tree algms are ..

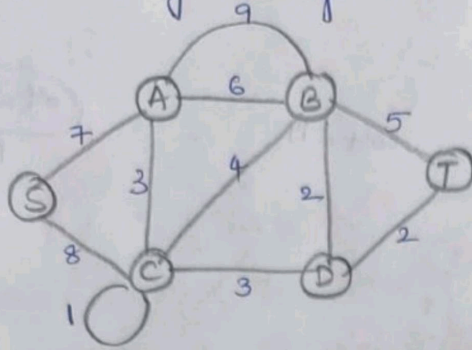
- ① Kruskal's Algorithm
- ② Prim's Algorithm.

Kruskal's Algorithm:-

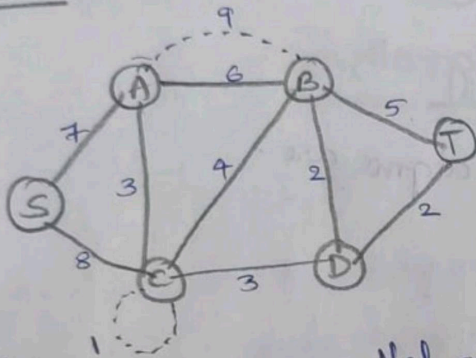
(Refer Text Book)

(4)

To understand Kruskal's algorithm let us consider the following example:-



Step 1:- Remove all loops and parallel edges.

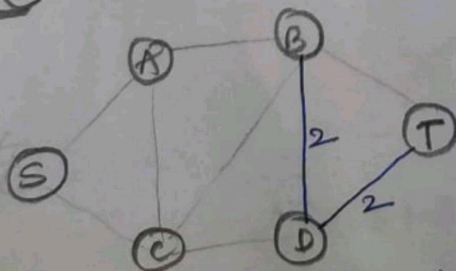


In case of parallel edges, keep the one which has the least cost and remove all others.

Step 2:- Arrange all edges in their increasing order (ascending) of weight.

B,D	D,T	A,C	C,D	C,B	B,T	A,B	S,A	S,C
2	2	3	3	4	5	6	7	8

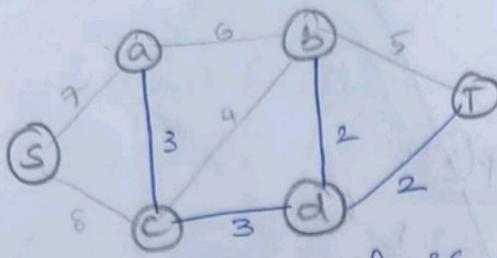
Step 3:- Add the edge with least weightage.



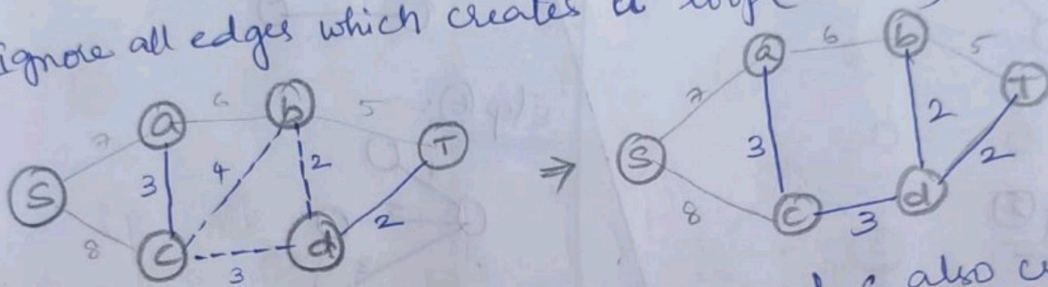
⇒ (Adding the edge does not violate the spanning tree properties so, continue with next edge selection.)

(5)

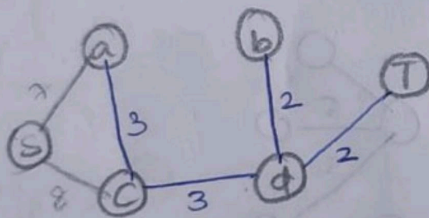
Next cost is 3, & associated edges are AC & CD
add them again.



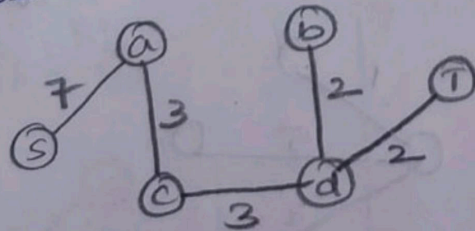
Next cost in the table is 4, & we observed that adding it will create a circuit/loop on the graph. So, ignore it.
∴ Ignore all edges which creates a loop (circuit)



We observe the edges with cost 5 and 6 also create circuits, we ignore them and move on.



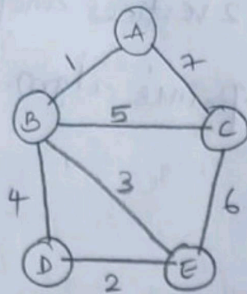
Now we are left with one ~~add~~ node to be added.
Between the two least cost edges available 7 and 8,
we shall add the edge with cost 7.



$$\text{Cost} = 7 + 3 + 3 + 2 + 2 = 17$$

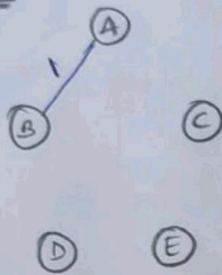
④ ⑥

Ex:-2 Construct min^m spanning tree using Kruskal's Algm:

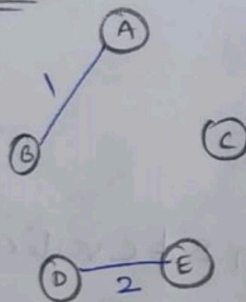


AB	DE	BE	BD	BC	CE	AC
1	2	3	4	5	6	7

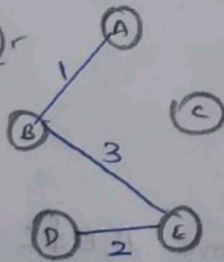
step 1:-



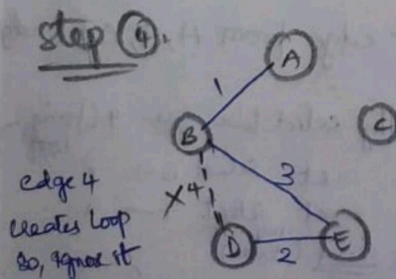
step 2:-



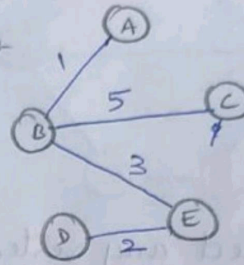
step 3:-



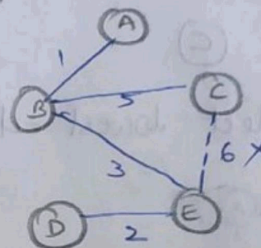
step 4:-



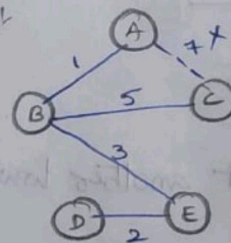
step 5:-



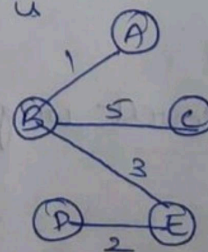
step 6:-



step 7:-



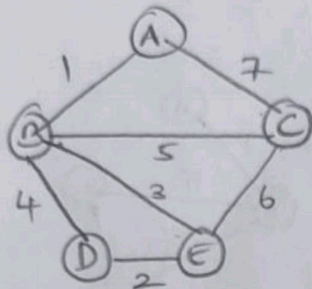
Min^m spanning tree using Kruskal's
algm is



$$\text{Min cost} = 1 + 3 + 5 + 2 = 11$$

② Prim's algm:- select one vertex as root then find min^m weight from that vertex & then find next (Refer TextBook) min^m weight for 2 vertices and soon.....

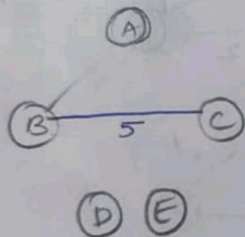
Construct min^m spanning tree using Prim's algm.



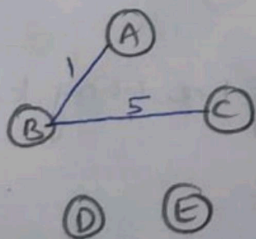
step ①:- Select any vertex from graph

③

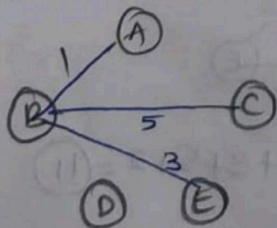
step ②:- select lowest edge from the C vertex.



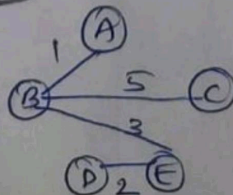
step ③:- select another lowest weight edge from B & C vertices



step ④:- select lowest weight edge from above A, B & C vertices



step ⑤:- select lowest edge from A, B, C & E edges



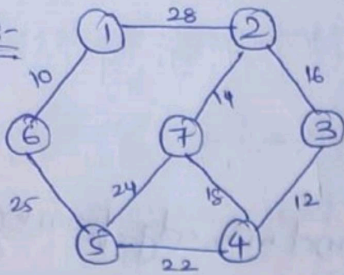
Why select low - i.e. 4 (forming loop)
not select 6 →
not select 7 →

Final (Tree)

$$5 + 3 + 2 + 1 = 11$$

Ex: ③ Construct min spanning trees using prim's & kruskal's

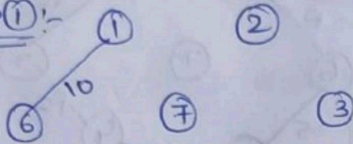
Ex:-



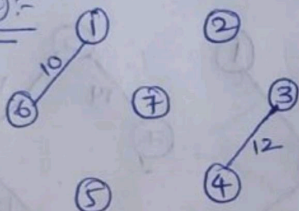
Kruskal's algm :-

10	12	14	16	18	22	24	25	28
----	----	----	----	----	----	----	----	----

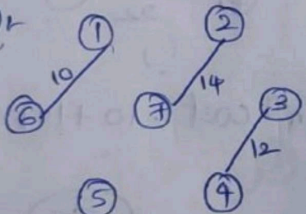
Step ①:-



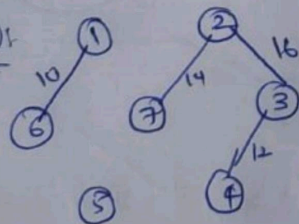
Step ②:-



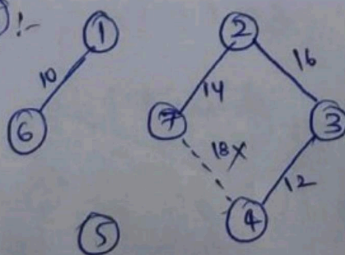
Step ③:-



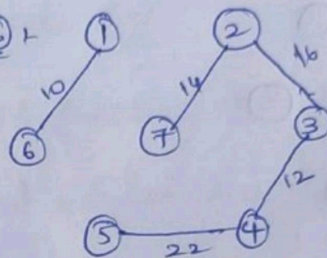
Step ④:-



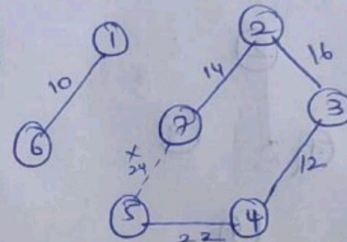
Step ⑤:-



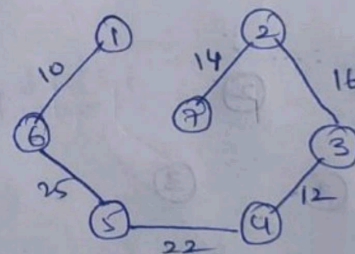
Step ⑥:-



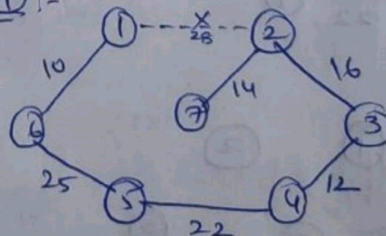
Step ⑦:-



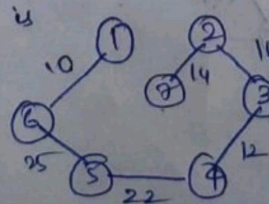
Step ⑧:-



Step ⑨:-

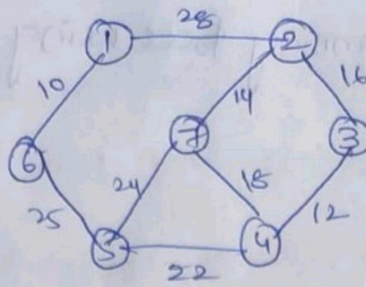


MST of above graph using kruskal's algm



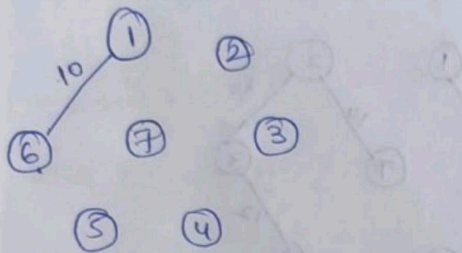
$$10 + 25 + 22 + 14 + 16 + 12 = 99$$

Using
Prims algo

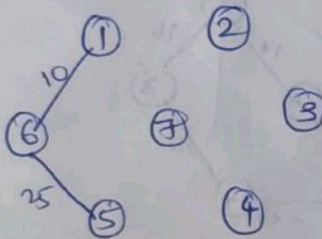


Step ①:-

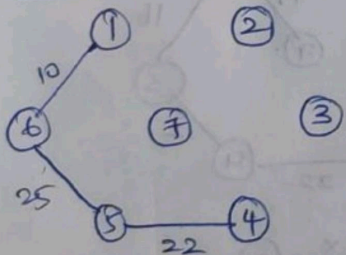
Select any vertex from graph as root & find min edge from that vertex



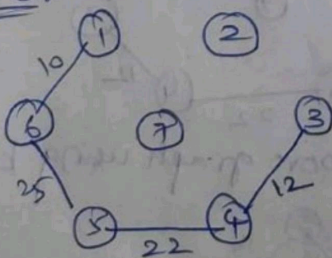
Step ②:-



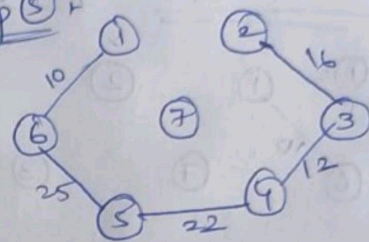
Step ③:-



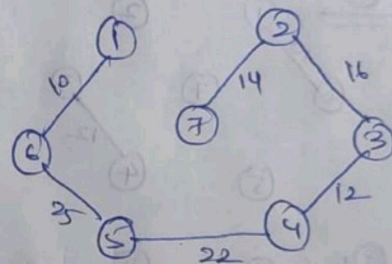
Step ④:-



Step ⑤:-



Step ⑥:-



$$\text{Min}^m \text{ cost} = 10 + 14 + 25 + 22 + 12 + 16$$

$$= 99$$

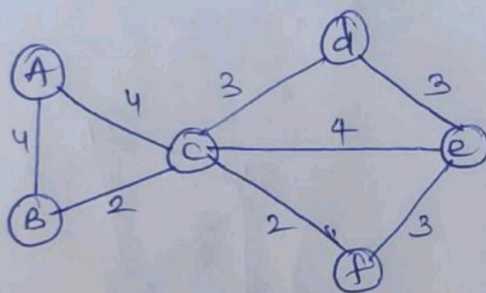
(9)

Q) will both algms always produce the same min spanning tree (MST) for any given graph?

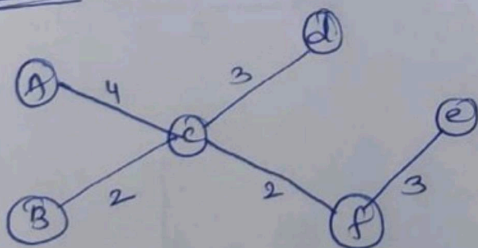
Case (1):- If all edges weights are distinct (not same/duplicate) then both algms will produce same MSTree with leads same cost. Ex:- (←) Refer Ex(1) in page (4)

Case (2):- when all edge weights are not distinct (if it has duplicate values) then the algms may not always produce the same MSTree but the cost of the MST produced would always be same in both cases

Ex:-



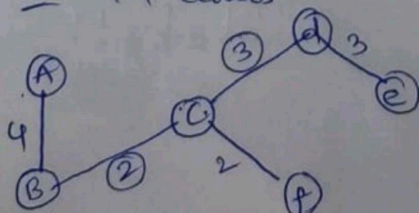
Prims:-



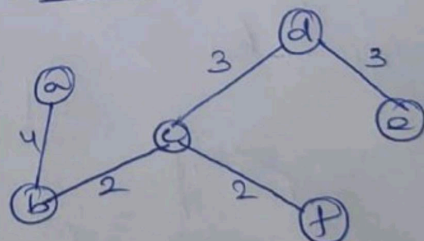
(or)

$$4 + 2 + 3 + 2 + 3$$

$$= 14 \text{ units}$$

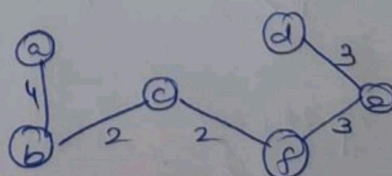


Kruskal's



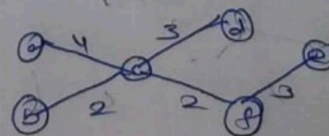
14 units

(or)

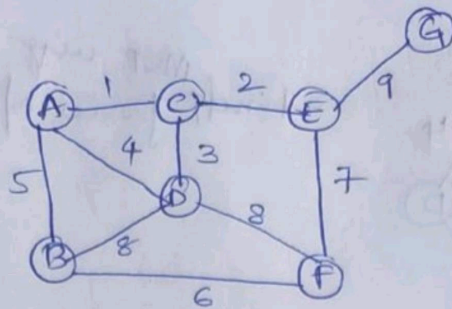
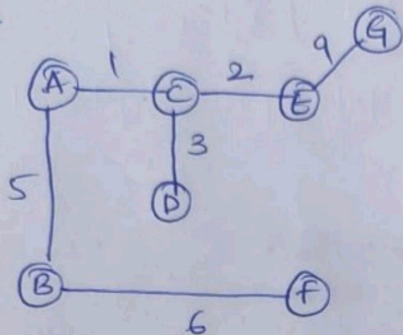


14 units

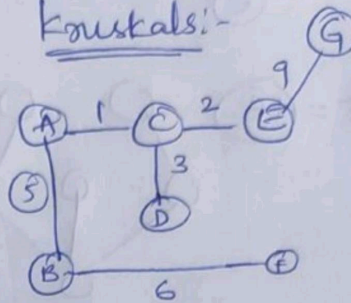
(or)



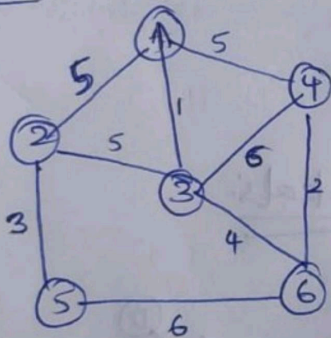
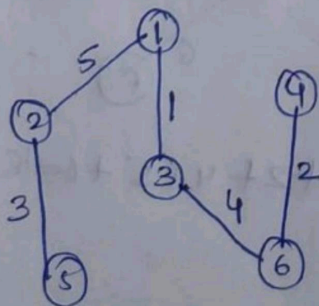
14 units

Practice Ex ①Prims

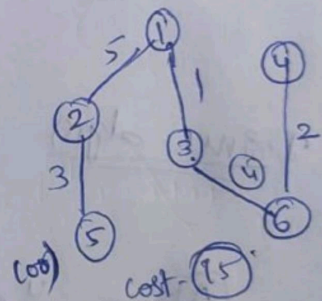
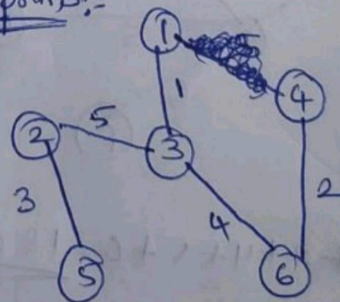
$$\text{Cost} = 1 + 5 + 3 + 2 + 9 + 6 = 26$$

Kruskals:-

$$\text{Cost} = 26$$

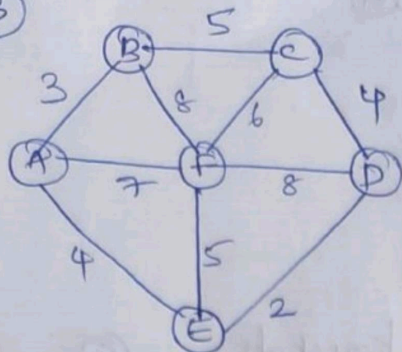
PE :- ②Kruskals

$$\text{Cost} = 1 + 5 + 3 + 4 + 2 = 15$$

Prims:-

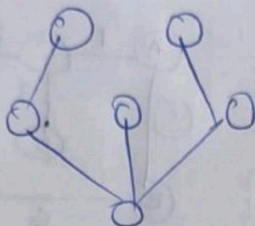
$$\text{Cost} = 3 + 5 + 1 + 4 + 2 = 15$$

PE:- (3)

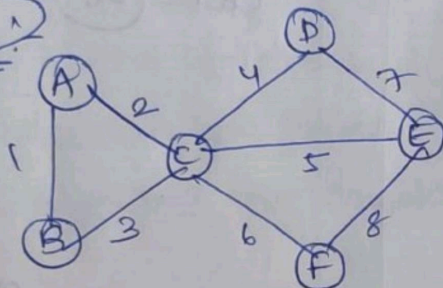


find ^{MST using} points of Kruskal's

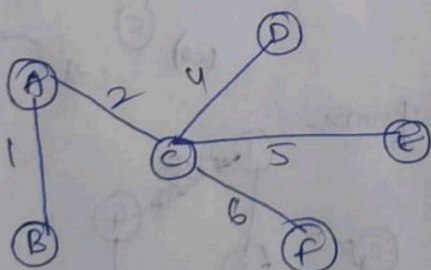
Ans: Cost = 18



PE:- (4)

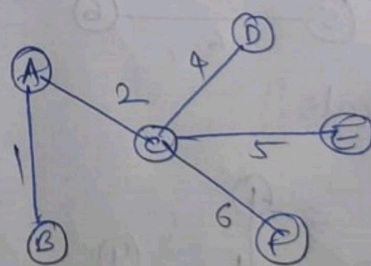


Prims algm



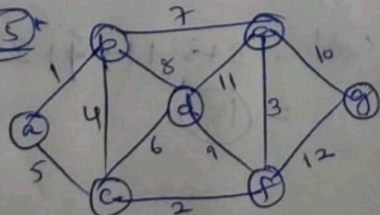
$1+2+4+5+6=18$ cost

Kruskal's:



$1+2+4+5+6=18$ cost

PE:- (5)



Ans: - Hent cost 26

UNIT-5

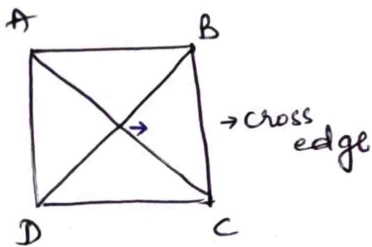
Graph Theory

Part-II

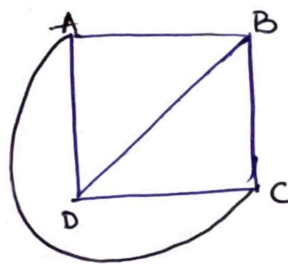
①

Planar graph:-

A graph G is said to be "planar Graph", If it can be drawn in a plane without any edges crossing, otherwise it is called "non-planar Graph".

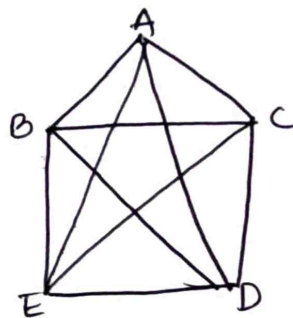
Ex:-Graph G

The given Graph G is a Non-planar Graph



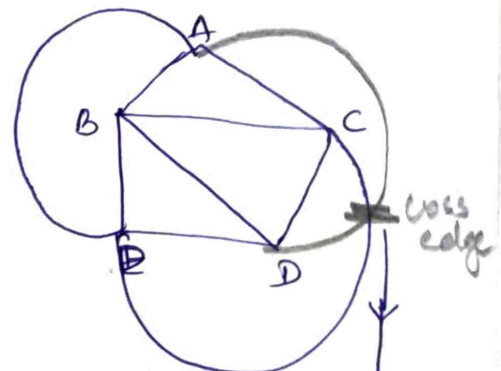
This Graph G is a planar graph because It Doesnot contain any edges.

Ex:- show that a complete Graph of 5 vertices is Non-planar.



complete Graph with 5 vertices
(K_5)

It is a Non-planar Graph



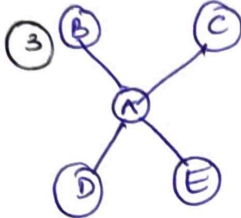
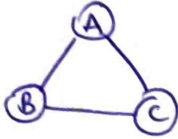
At this Point 2 edges cross each other
Hence we can say K_5 is non-planar graph

Planar Graph Examples:

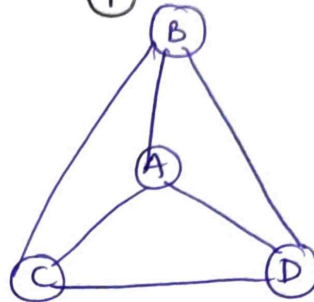
Ex (1)



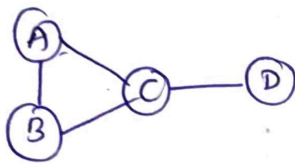
(2)



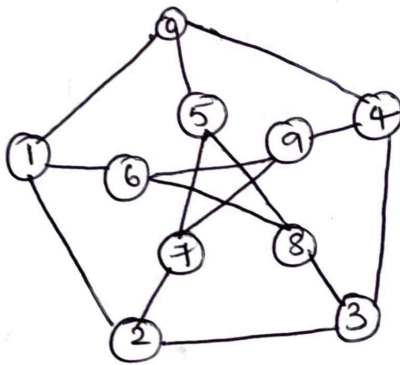
(7)



Ex: (5)



Petersen Graph:- undirected graph with 10 vertices and 15 edges.



Complete Graph:-

A complete graph is a graph that has an edge b/w every single vertex is a graph.

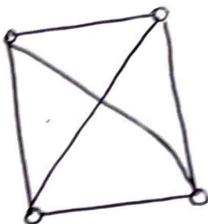
⇒ we represent a complete graph with n vertices using the symbol " K_n ".

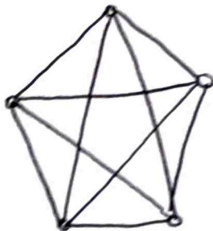
⇒ A complete graph for 2 vertices are represented as K_2

Ex:- $K_2 \Rightarrow$ 

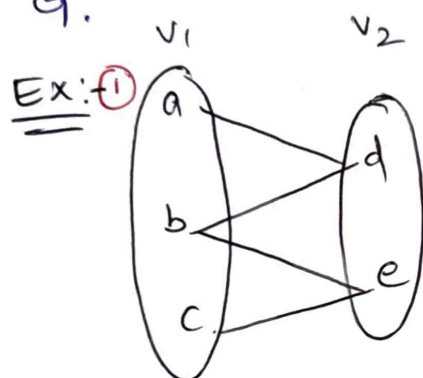
Similarly:- A complete graph for 3, 4, 5 vertices are represented as K_3, K_4, K_5 .

$K_3 \Rightarrow$ 

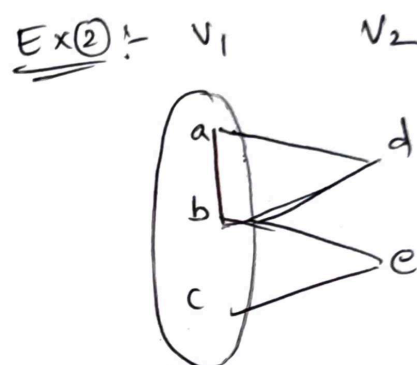
$K_4 \Rightarrow$ 

$K_5 \Rightarrow$ 

Bipartite graph: - A graph $G(V, E)$ is called Bi-partite if its vertex set $V(G)$ can be partitioned into two non-empty disjoint subsets $V_1(G)$ and $V_2(G)$ in such a way that each edge $e \in E(G)$ has its one end point in $V_1(G)$ and other end point in $V_2(G)$, the partition $V = V_1 \cup V_2$ called bi-partition of G .



✓ Valid Bipartite Graph



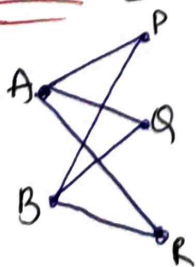
✗ Not Valid Bipartite Graph
($\because$ there is an edge b/w $a \& b$)

The above Example is Bipartite graph but not Complete Bipartite graph since there is no edge b/w $d-c$ & $e-a$.

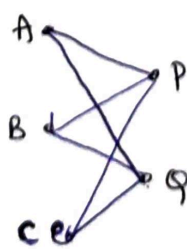
Complete Bipartite graph

A bipartite graph G is called a Complete bipartite graph if there is an edge b/w every vertex in V_1 and every vertex in V_2 .

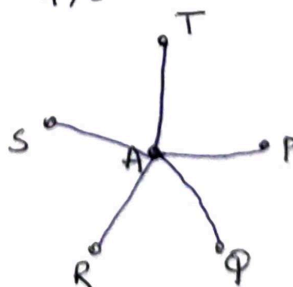
Ex: ① $K_{2,3}$

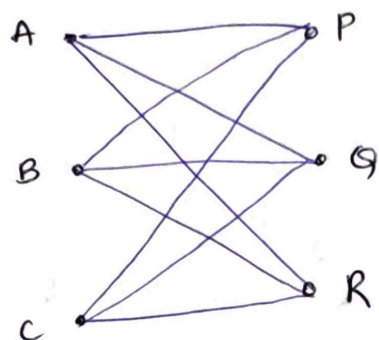


Ex: ② $K_{3,2}$



$V_1 = \{A, B, C\}$
 $V_2 = \{P, Q, R, S, T\}$
 $K_{1,5}$

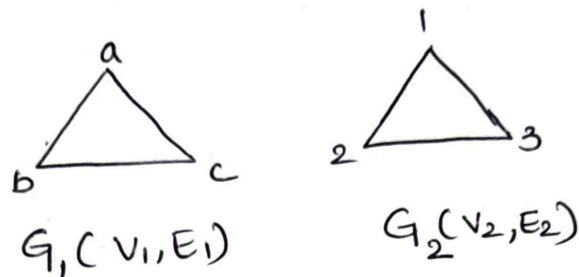


$K_{3,3}$ 

Complete bipartite graph $K_{3,3}$ is known as
Kuratowski's graph..

Isomorphism:-

Two graphs are said to be isomorphic if there are perhaps the same graphs, just drawn differently with different names i.e...



formal def:- Two graphs G_1 and G_2 are said to be isomorphic, if there is one-to-one correspondence b/w the vertices and edges, and incidence relationship is preserved.

More formally:- $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are isomorphic to each other if there is a bijection function

$$f: V_1(G_1) \longrightarrow V_2(G_2)$$

= o =

Isomorphism in Graph Theory:-

Definition- Consider 2 Graphs $G=(V,E)$ and $G'(V',E')$.

Suppose there exists a function $f: V \rightarrow V'$ such that

(i) f is one-to-one and onto

(ii) for all vertices A, B of G , the edge $\{A, B\} \in E$

if and only if the edge $\{f(A), f(B)\} \in E'$

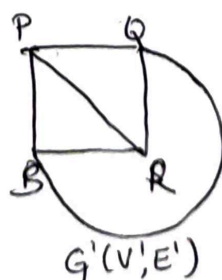
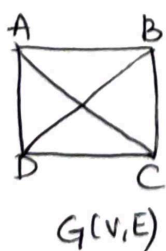
Then f is called an "Isomorphism" b/w G and G' and we say that G and G' are "Isomorphic Graphs".

Another Defn:-

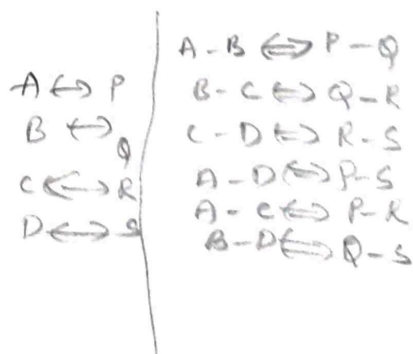
Two Graphs G and G' are said to be Isomorphic if there is a one-to-one correspondence b/w their vertices and between their edges such that the adjacency of vertices is preserved. Such Graphs will have the same structure, differing only in the way their vertices and edges are labelled or only in the way they are represented geometrically.

when G and G' are Isomorphic, we write it as $G \cong G'$.

The symbol ' $\cong$ ' represents Isomorphism.



$G \cong G'$



criteria or steps to be followed for checking given 2 ^⑤
Graphs are Isomorphic or not:-

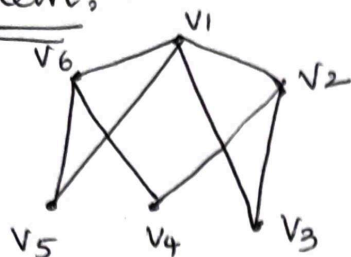
- ① Both Graphs have the same number of vertices.
- ② Both Graphs have the same number of edges.
- ③ Both graphs have an equal number of vertices with a given degree.
- ④ one-to-one correspondence between the vertices of two Graphs
- ⑤ Edge preserving is also satisfied b/w the two Graphs.
- ⑥ Smallest possible cycle length is same for two Graphs.
- ⑦ Adjacency matrices of 2 Graphs should be the same.

∴ By satisfying the above conditions we can say that 2 Graphs are Isomorphic, It can be represented by

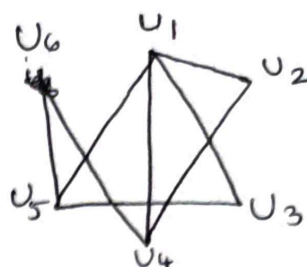
$$\boxed{G \cong G'}$$

== o ==

Problem:-



Graph G $G(V, E)$



Graph G' $G'(V', E')$

Verify the above 2 Graphs. G and G' are Isomorphic or not.

- ① Number of Vertices $|V| = 6$
- ② Number of Edges $|E| = 8$
- ③ Degree of all vertices in Graph G is

$$= \{4, 3, 2, 2, 2, 3\}$$

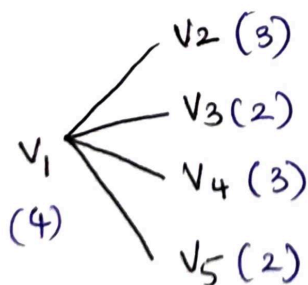
- ④ cycle length : Smallest possible cycle length

$$V_1 - V_2 - V_3 - V_1 : 3 \quad \checkmark$$

$$V_1 - V_6 - V_5 - V_1 : 3 \quad \checkmark$$

$$V_1 - V_2 - V_4 - V_6 - V_1 : 4 \quad \times$$

- ⑤ Vertex preserving



$$G \quad V_1 \leftrightarrow U_1$$

- ① Number of vertices $|V'| = 6$
- ② Numbers of edges $|E'| = 8$
- ③ Degree of all vertices in Graph G' is

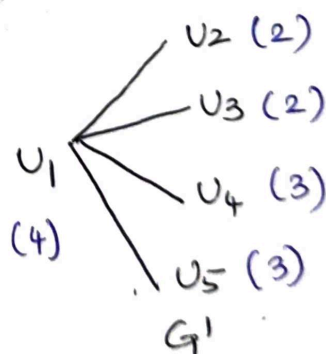
$$= \{4, 2, 2, 3, 3, 2\}$$

- ④ Smallest possible cycle length

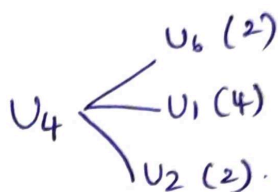
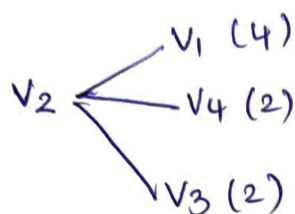
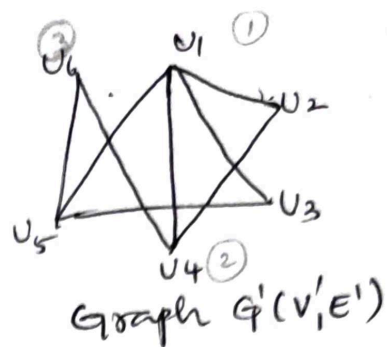
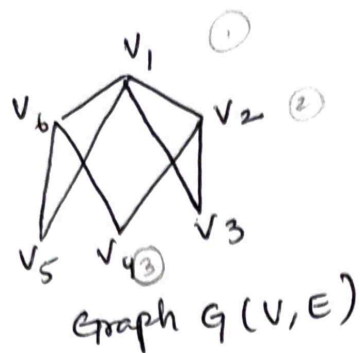
$$U_1 - U_2 - U_4 - U_1 : 3$$

$$U_1 - U_5 - U_3 - U_1 : 3$$

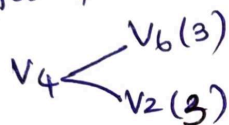
- ⑤ Vertex preserving



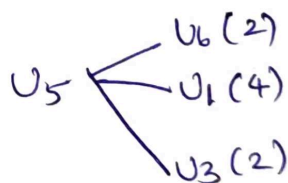
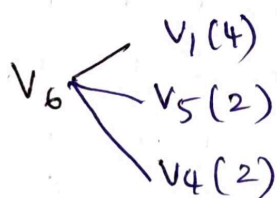
$V_2 \rightarrow$ just see degree of V_2 i.e. 3 check and select 3 degree vertex in G' . Select either U_4 or U_5



for $V_4 \rightarrow$ just see degree $V_4(2)$ check and select 2 degree vertex in G' (U_4)
select either U_2 or U_6

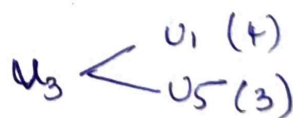
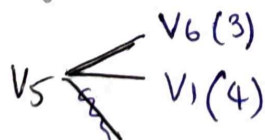


$V_4 \leftrightarrow U_6$
 $V_3 \leftrightarrow U_2$ ($\because$ from ②)
 $V_6 \leftrightarrow U_5$ ($\because$ from ③)



$V_5 \leftrightarrow U_3$

proof



$V_5 \leftrightarrow U_3$

$\therefore$ Vertex preserving

$\{V_1, V_2, V_3, V_4, V_5, V_6\}$

vertex preserving

$\{U_1, U_4, U_2, U_6, U_3, U_5\}$

⑥ Edge preserving:-

$$\{v_1, v_2\} \Leftrightarrow \{u_1, u_4\}$$

$$\{v_1, v_3\} \Leftrightarrow \{u_1, u_2\}$$

$$\{v_1, v_5\} \Leftrightarrow \{u_1, u_3\}$$

$$\{v_1, v_6\} \Leftrightarrow \{u_1, u_5\}$$

$$\{v_4, v_6\} \Leftrightarrow \{u_6, u_5\}$$

$$\{v_4, v_2\} \Leftrightarrow \{u_6, u_4\}$$

~~$$\{v_4, v_6\} \Leftrightarrow \{u_6, u_5\} \text{ (Repeated)}$$~~

$$\{v_5, v_6\} \Leftrightarrow \{u_3, u_5\}$$

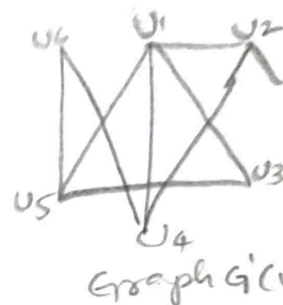
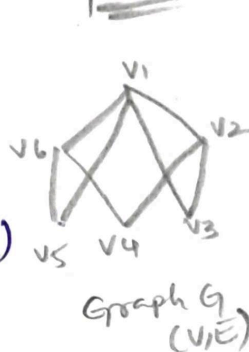
$$\{v_2, v_3\} \Leftrightarrow \{u_4, u_2\}$$

vertex preserving

$$G = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

$$G' = \{u_1, u_4, u_2, u_6, u_3, u_5\}$$

Proof connect edges



⑦ Adjacency matrix

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	1	0	1	1
v_2	1	0	1	1	0	0
v_3	1	1	0	0	0	0
v_4	0	1	0	0	0	1
v_5	1	0	0	0	0	1
v_6	1	0	0	1	1	0

⑦ Adjacency matrix

	u_1	u_2	u_3	u_4	u_5	u_6
u_1	0	1	1	0	1	1
u_2	1	0	1	1	0	0
u_3	1	1	0	0	0	0
u_4	0	1	0	0	0	1
u_5	1	0	0	0	0	1
u_6	1	0	0	1	1	0

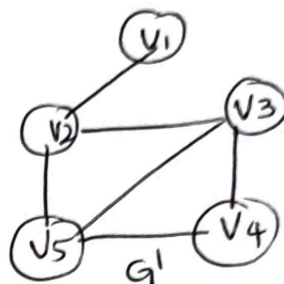
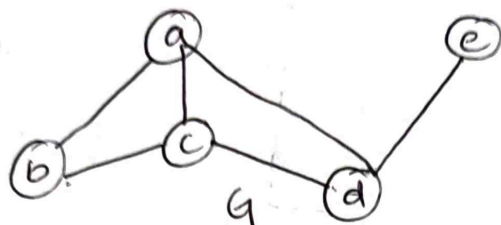
$\therefore G$ and G' are Isomorphic to each other

$$\therefore G \cong G'$$

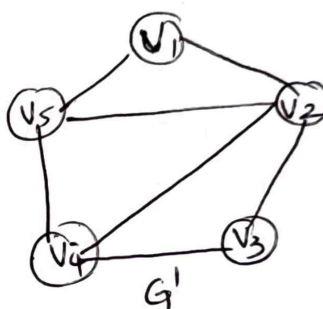
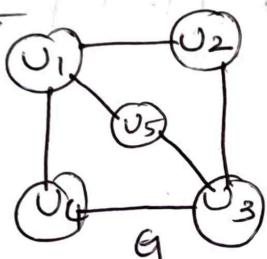
H/W:-

(7)

Ex(2) :- Verify the below 2 Graphs G and G' are Isomorphic or not.

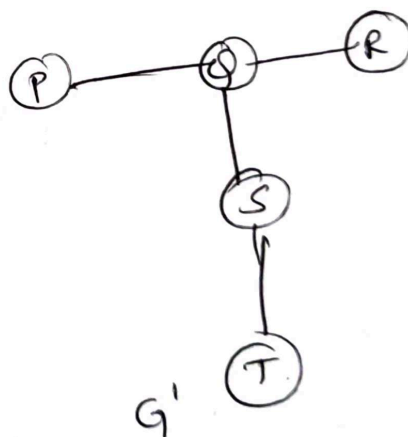
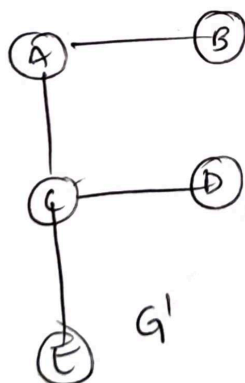


Hint: $\cong$

Ex(3):-

Hint Not Isomorphic
No. Edges in G & $G' = 6$

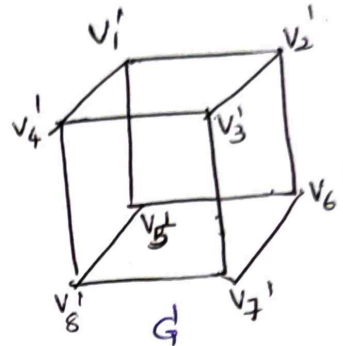
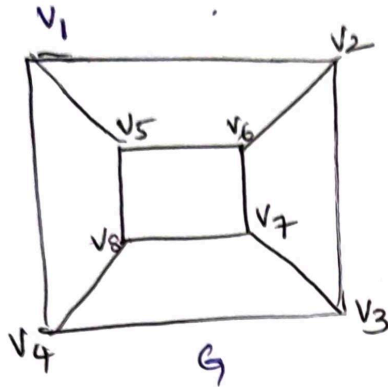
No. Edges in $G' = 7 \therefore G$ is not Isomorphic to G' .

Ex(4):-

Hint $\cong$ (Isomorphic)

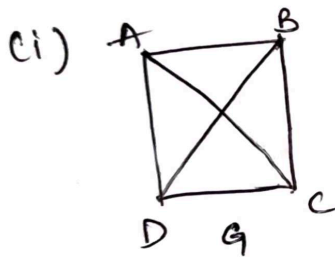
Problems circuits:-

verify if they are isomorphic or not

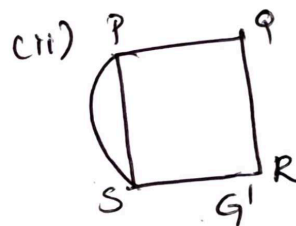


Ans:- $G \cong G'$

Q) s/t the following graphs are not isomorphic



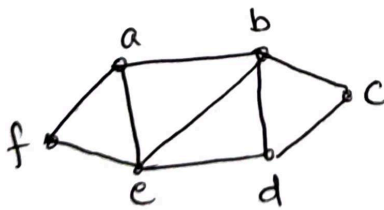
$$|E(G)| = 6$$



$$|E(G')| = 5$$

$\therefore$ not isomorphic.

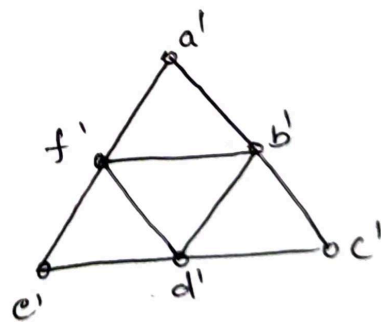
(ii)



$$|V| = 6$$

$$|E| = 9$$

$$d(G) = \{2, 2, 3, 3, 4, 4\}$$



$$|V| = 6$$

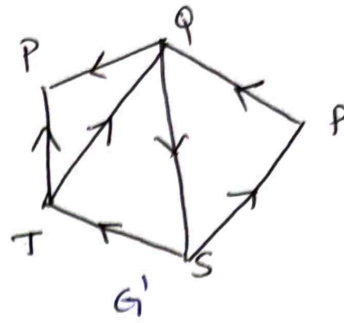
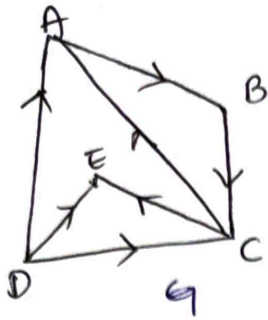
$$|E| = 9$$

$$d(G') = (2, 2, 2, 4, 4, 4)$$

$\therefore$ not isomorphic.

Ex:-

(9)



for G graph $A \rightarrow \text{Indegree} = 2$
 In G' , no element having in degree 2
 $\therefore$ It is not Isomorphic

Euler circuits and Hamiltonian Graphs! -

Basic terminologies:-

Walk:- A walk is defined as a finite alternating (changing) sequence of vertices & edges, beginning & ending with vertices, such that each edge is incident with the vertices preceding & following it.

~~repeated~~ edges
~~repeated~~ vertices
* No edge appear more than once in a walk, a vertex however may appear more than once

Closed walk:-

A walk is said to be closed if the first and last vertices are the same vertex. i.e. that means you start walking at a vertex & end up at the same.

Open walk:-

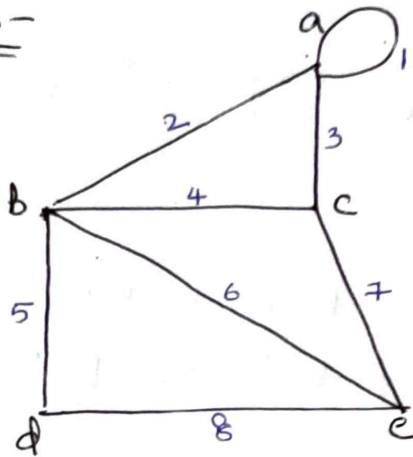
A walk is said to be open if the first and the last vertices are different. i.e., the terminal vertices are different.

~~Vertex~~ Trail:- is an open walk where vertices can repeat, but ~~Edges~~ not edges

~~Vertex~~ Path:- is an open walk with no repetition of vertices & edges
~~Edges~~

Circuit:- is a closed walk where vertices can repeat, but not edges

Cycle:- is a closed walk neither vertex nor edges can repeat. But since it is closed, the first and last vertices are the same (one repetition).

Ex:-

① a2b5d8e

walk
 ✓ Vertices
 ✓ Edges
 (open walk)

path
 ✓
 X Edges
 X Vertices

② a2b6e8d5b4c

✓
 (open walk)

X
 (b repeated)

③ a3c7e6b4c7e6b

X 6,7
 (repeated edge traversal)

X 7,6, edges &
 c-vertex repeated

④ a1a3c7e

✓
 (open walk) No repeated
 edge traversal

X (repeated a vertex)

⑤ a3c4b2a

✓
 (closed walk)

X
 (repeated a vertex)

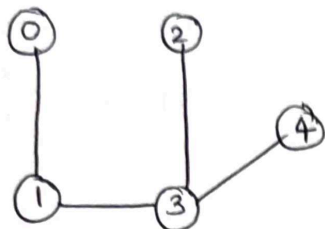
⑥ a3c7e6b4c

✓
 (open walk)

X
 (repeated c vertex)

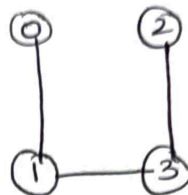
Connected graph:- A graph is said to be connected when there is a path between every pair of vertices. (10)

Ex:-



connected graph

($\because$ There is a path between pair of vertices)

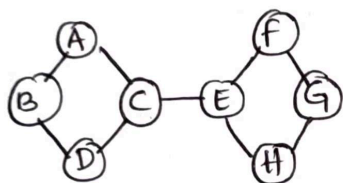


(4)

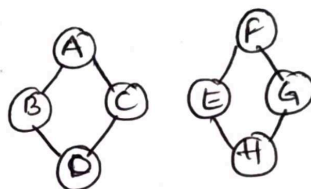
disconnected Graph

($\because$ there is no path b/w 0 to 4 (or) 2 to 4 (or) 1 to 4 (or) 3 to 4)

Ex:- (2)



connected graph



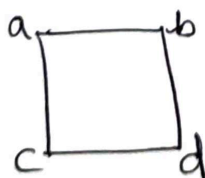
Disconnected Graph

Euler circuit/ cycle:-

A closed walk which visits every edge of the graph exactly once (Vertex can be repeated but not edges)

Euler Graph:- A Graph G which contains a euler cycle is called Euler Graph. (or) circuit

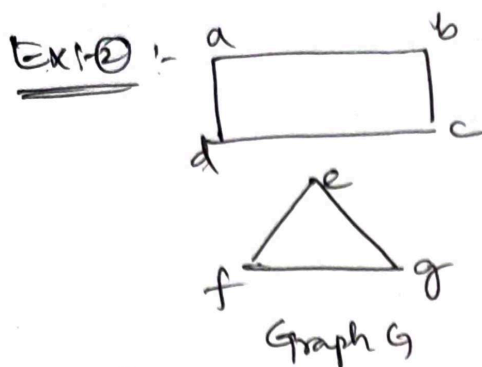
Ex:-



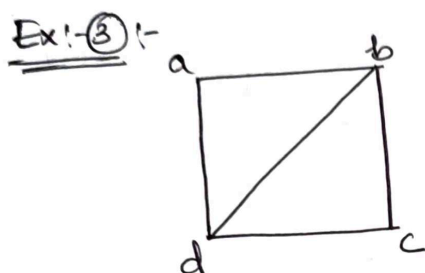
closed walk

a-b-d-c-a (covers all edges)

Euler Graph (or) Eulerian Graph



It is not a Euler Graph because it is not connected graph



⇒ It is not a Euler Graph. Let's take a-b-d-a (closed walk) edges not repeated even it is not Euler graph

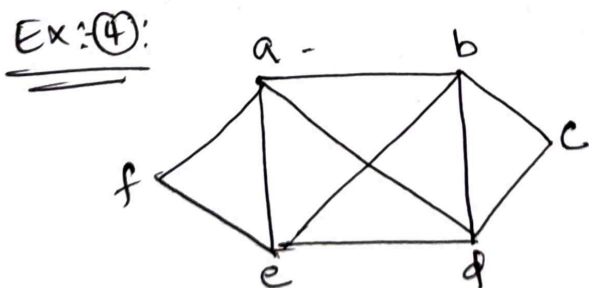
Let take other path like

a-b-c-d-a

⇒ It is also not Euler Graph why because d-b edge not covered (traversed)

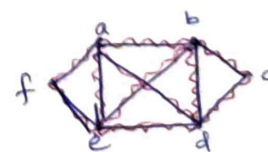
It has to cover all edges in graph at once...
Here b-c & c-d not covered (or) traversed.

∴ Ex-3 not Euler Graph.



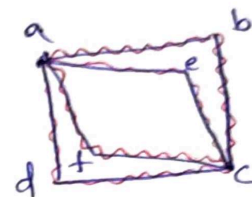
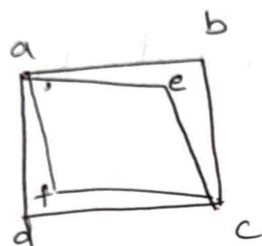
a-f-e-b-a-b-d-e-b-c-d-a

∴ It is valid Euler Graph



Ex:-(4):-

The above G is not the Euler Graph.

Ex:-(5):-

The above Graph is Euler Graph ($\because$ a-b-c-e-a-f-c-d-a
(∞))

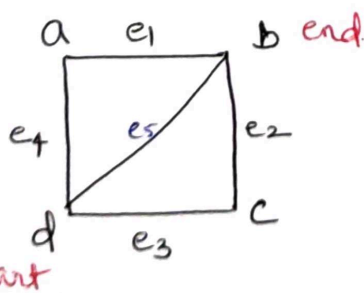
a-b-c-d-a-f-c-e-a

$\therefore$ Summary on Euler Graph

- no edge repetition
- vertex may repeated
- closed walk

Open Euler walk:-

An open walk which visits every edge of the Graph exactly once. (starts from one pt and ends at another point
(vertex) (vertex))

Ex:-

walk-seq:- d-e₃-c-b-d-c-d

d-e₄-a-e₁-b-e₅-d-e₃-c-e₂-b

↑ start vertex ↑ end vertex

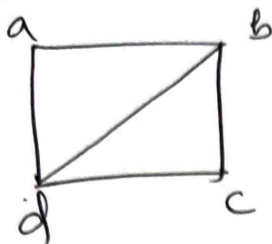
starts & ends not same (vertex)

So, it is not Euler Graph but we can say Open Euler walk (Semi-Euler Graph)

Semi-Euler Graph (or) Euler path:-

It a graph contains a open Euler walk.

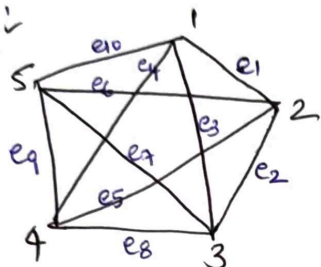
Ex:-



d-a-b-c-d-b

It is a semi-Euler Graph.

Ex: ⑥ :-

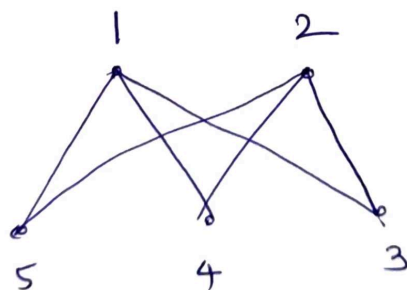


walk seq:-

①-e₁-2-e₂-3-e₃-1-e₄-4-e₅-2-e₆-5-e₇-3-e₈-4-e₉-5-e₁₀-①

✓ Valid Euler Graph closed walk which visits all edges exactly once (vertices may be repeated)

Ex: ⑦ :-



walk-seq:- 1-e₂-3-e₄-2-e₃-4-e₆-1-e₁-5-e₅-2

It is not a closed walk, so, it is not a Euler ckt).

∴ It is not a Euler Graph...but it is a open walk why because starting & endg vertex not same & it visits all edges of a Graph. ∴ It is a Open Euler Graph | semi EG

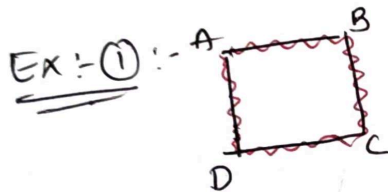
Hamiltonian cycle, Hamiltonian Graph:-

Hamiltonian cycle/circuit:-

In a connected Graph, a closed walk that visits every vertex of the Graph G exactly once except the starting and ending vertices. It is called Hamiltonian cycle. (no need to cover all edges in Graph)

Hamiltonian Graph:-

A Graph G is said to be Hamiltonian if it has Hamiltonian circuit.

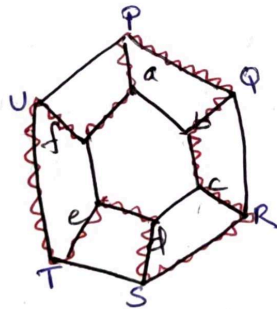


(A) $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ (A)

closed walk + covered all vertices

$\therefore$ Hamiltonian Graph.

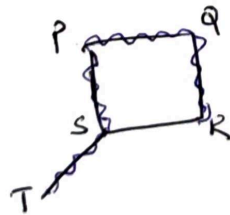
Ex:- (2) :-



P-Q-b-c-R-s-d-e-t-u-f-a-P

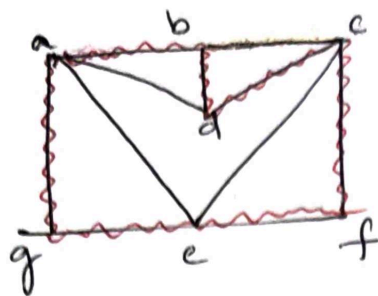
$\therefore$ It is a Hamiltonian Graph.

Ex:- (3) :-



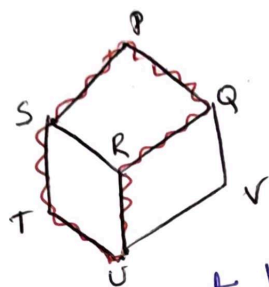
T-S-P-Q-R-~~X~~
(vertex xp)

$\therefore$ It is not Hamiltonian Graph.

Ex:-④

$$a-b-d-c-f-e-g-a$$

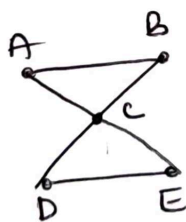
$\therefore$ It is Hamiltonian Graph.

Ex:-⑤

$\therefore$ It is not Hamiltonian Graph ($\because$ vertex missed)

Hamiltonian Path:-

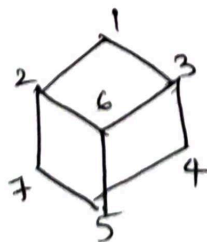
start from one vertex and ends at another vertex and covers all vertices

Ex:-①

$$A-B-C-D-E$$

It is not a Hamiltonian cycle X
Hence it is not a Hamiltonian Graph.

$\therefore$ It is a Hamiltonian path ✓

Ex:-②

$$1-2-6-3-4-5-7$$

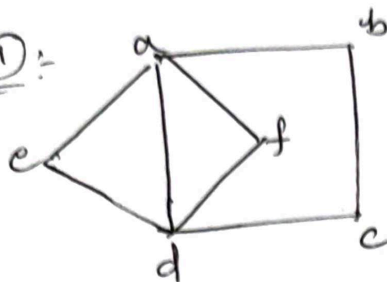
It is an open walk.

Hence it is a Hamiltonian path.

It is not a Hamiltonian Graph X

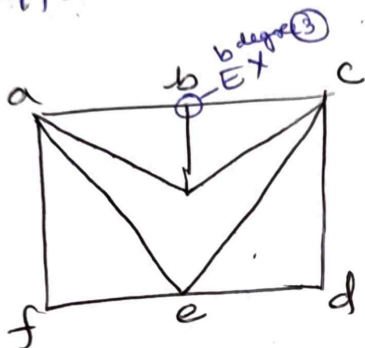
Check the given Graphs are Euler or Hamiltonian or both.

Ex(1):



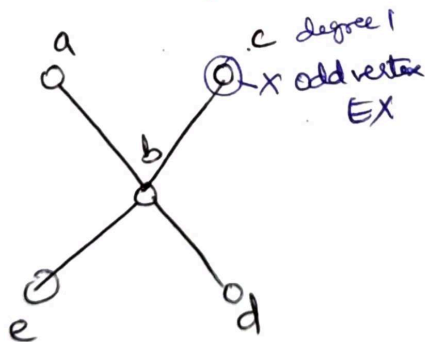
Euler ✓ $e-a-b-c-d-f-a-d-e$
Hamiltonian X

Ex(2):



Euler X
Hamiltonian ✓
 $a-b-c-d-e-f-a$

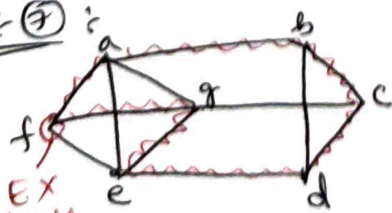
Ex(3):



H X

E X

Ex(4):

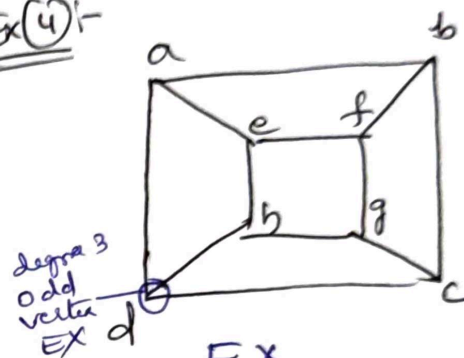


EX
3 degree
odd

EX H ✓

$a-b-c-d-e-g-f-a$

Ex(4):



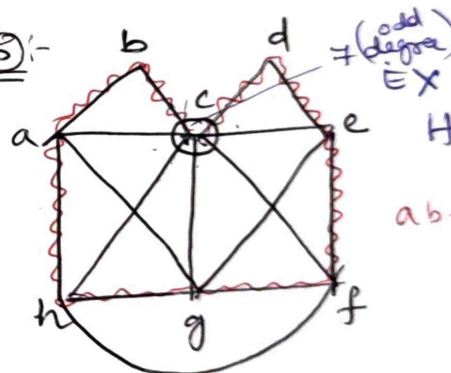
degree 3
odd
vertex
EX d

EX

H ✓

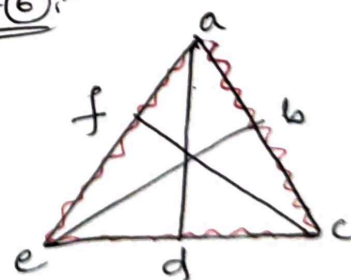
$a-e-f-b-c-g-d-a$

Ex(5):



$ab-cdefg$
?

Ex(6):

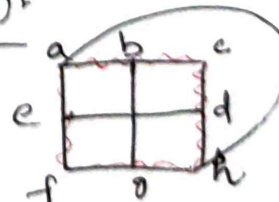


EX

H ✓

$(a-b-c-d-e-f-a)$

Ex(8):



H ✓ EX

$a-b-c-d-h-g-f-e-H$