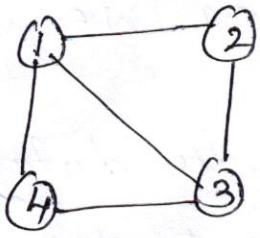
Graph

Definition:

Graph is a non-linear data structure and can be defined by the pair $G = (V, E)$ where,

V = finite and non-empty set of vertices (nodes).

E = set of edges which are the pair of vertices



$V = \{1, 2, 3, 4\}$.

$E = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 3,4\rangle\}$.

Fig: Undirected graph

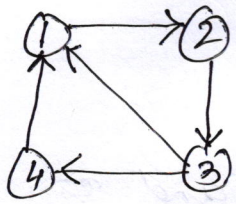
Basic Terminologies:

1. Undirected graph: A graph, which has unordered pair of vertices, is called undirected graph (edges have no direction).

   $\langle 2, 3\rangle$ and $\langle 3, 2\rangle$ are both same edge.

2. Directed graph: A graph in which each node is assigned a direction. Each edge is represented by an ordered pair of vertices $\langle v_1, v_2\rangle$ where v_1 is the tail and v_2 is the head of the edge.

   $\langle v_1, v_2\rangle$ and $\langle v_2, v_1\rangle$ will represent different edges.

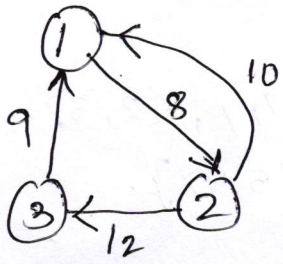


$V = \{1, 2, 3, 4\}$

$E = \{\langle 1,2\rangle, \langle 2,3\rangle, \{3,1\}, \langle 3,4\rangle, \langle 4,1\rangle\}$

Fig: directed graph

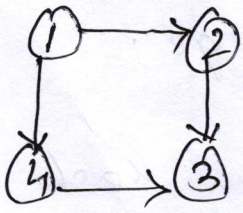
3. Weighted graph: A graph is said to be weighted if every edge in the graph is assigned some non-negative value as weight



The weight may be the distance of the edge or the cost to travel along the edge.

Fig: Weighted directed graph

4. Adjacent nodes: A node V_1 is adjacent to or neighbor of another node V_2 if there is an edge from V_1 to node V_2.



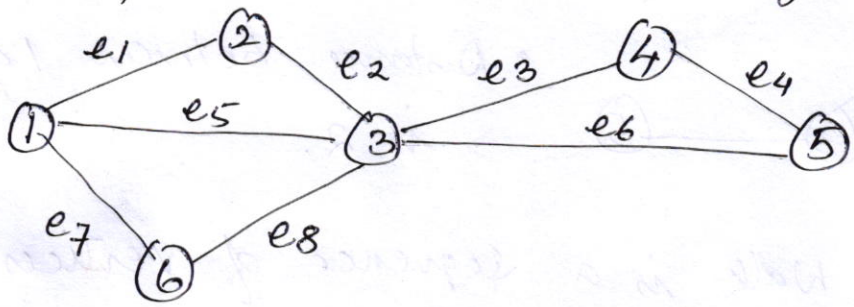
Eg: node 1 is adjacent to node 2. and node 2 is adjacent from node 1.

$e = [u, v]$, then the nodes u and v are called the endpoints of e and u and v are said to be adjacent nodes or neighbours.

5. Incidence: In an undirected graph, the edge $\langle v_0, v_1\rangle$ is incident on nodes v_0 and v_1.

In a diagraph the edge $\langle v_0, v_1 \rangle$ is incident from node v_0 and is incident to node v_1.

6. Path and length of path: Path is a sequence of vertices and edges in which any two consecutive edges are incident and no vertex is repeated. The total number of edges on a path is called its length.

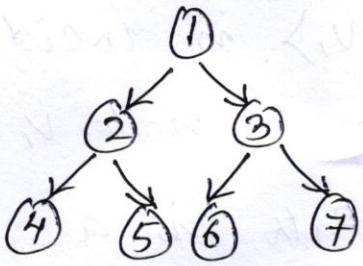


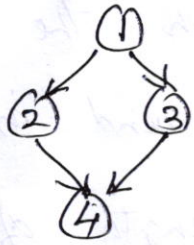
Path	length
$1 e_1 2 e_2 3 e_3 4 e_4 5$	4
$1 e_5 3 e_6 6$	2

7. Cycle and Hamiltonian cycle: A cycle is a path that starts and ends at the same vertex. Hamiltanian cycle is a path in an undirected or directed graph that visits each vertex exactly once.

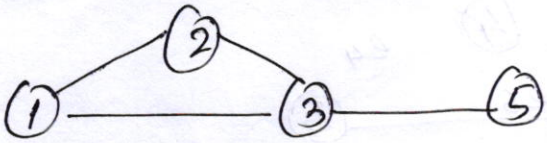
8. Directed acycling graph (DAG): A directed acycling graph is a directed graph that has no cycle:

Eg: All trees are DAG (but all DAGs are not trees)





9. Distance: Distance between two vertices v_1 and v_2 is the length of the shortest path between v_1 and v_2.



• Distance between 1 and 5 is 2.

10. Walk: Walk is a sequence of vertices and edges in which any two consecutive edges are incident and vertex and edge both can be repeated.

11. Isolated node: If $\deg(n) = 0$, that is, if n does not belong to any edge, then n is called an isolated node.

~~12. Connected graph~~.

12. Connected graph: A graph G is said to be connected if there is a path between any two of its nodes.

13. Complete graph: A graph G is said to be

complete if every node u in G is adjacent to every other node v in G. A complete graph with n node will have $n(n-1)/2$ edges.

14. Indegree and outdegree: In a directed graph, vertices have both indegrees and outdegrees: The indegree of a vertex is the number of edges leading to that vertex and the outdegree of a vertex is the number of edges leading away from that vertex.

⊛ Representation of Graph

There are two ways:

1. Sequential representation / Adjacency Matrix
2. Adjacency list / Linked Representation.

1. Sequential Representation / Adjacency matrix

In sequential representation we use a 2-Dimensional array of order $n \times n$ where n is the total number of vertices in the graph.

A graph with n nodes takes n^2 space to represent it sequentially in memory and also takes $O(n^2)$ time to perform the graph operations and to solve the graph

problems.

1. Adjacency matrix
2. Path matrix
3. Weighted matrix

1. Adjacency matrix.

Suppose G is a graph with n nodes and suppose the vertices of graph G are ordered and are called 1, 2, 3, ..., n. Then the adjacency matrix A of the graph G is an n x n matrix defined as follows:

A[i][j] = 1 if there is an edge from node i to node j.

= 0 otherwise.

Such a matrix which contains only 1 or 0 is called a bit matrix or Boolean matrix. The adjacency matrix of an undirected simple graph is a symmetric matrix.

In the adjacency matrix of an undirected simple graph, the number of '1' is twice the number of edge in the graph.

2. Path matrix : Suppose G is a directed graph with n nodes and suppose the vertices of graph G are ordered and are called 1, 2, 3, ..., n. Then the path matrix P of the graph G is an n x n matrix defined as follows:

P[i][j] = 1 if there is a path from vertex i to vertex j.

= 0 otherwise

3. Weighted matrix : The weighted matrix W of a weighted graph G with n nodes is an n x n matrix which can be defined as.

W[i][j] = w if there is a direct edge of weight w from node i to node j.

= ∞ otherwise.

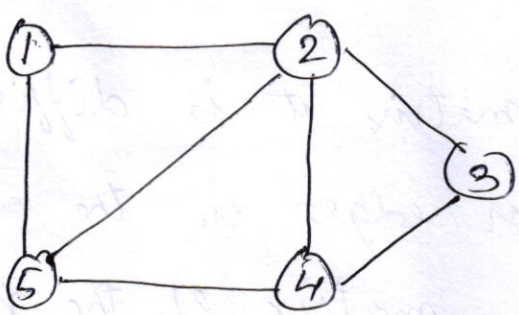


fig 1

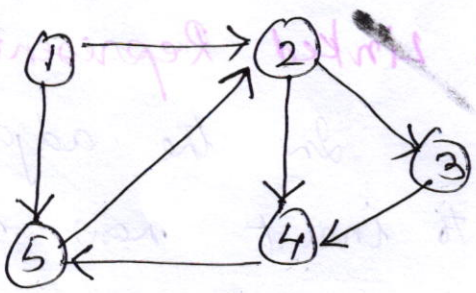


fig 2

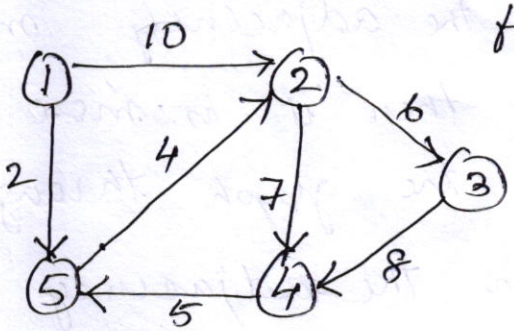


fig 3

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

Adjacency Matrix of fig 1 (Symmetric Matrix)

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

Adjacency Matrix of fig 2.

	1	2	3	4	5
1	∞	10	∞	∞	2
2	∞	∞	6	7	∞
3	∞	∞	∞	8	∞
4	∞	∞	∞	∞	5
5	∞	4	∞	∞	∞

Weighted Matrix of fig 3.

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

Path Matrix of fig 2.

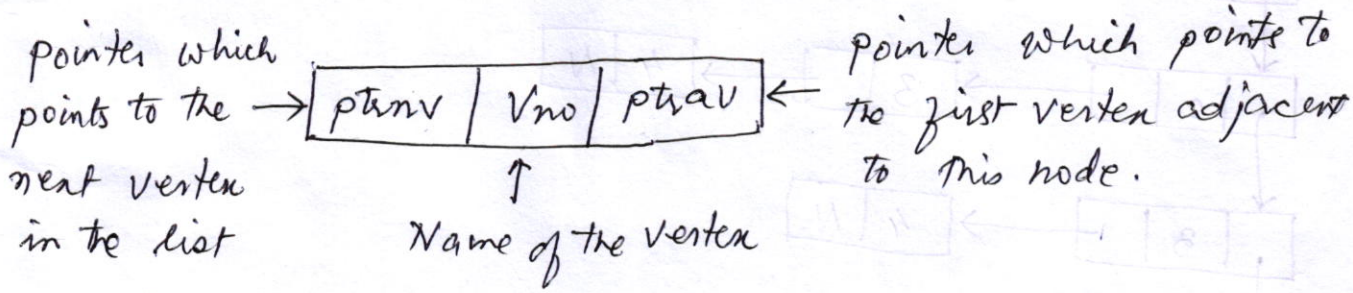
2. Linked Representation

In the adjacency matrix it is difficult to insert new nodes or edges in the graph. If the adjacency matrix of the graph is sparse then it is more efficient to represent the graph through adjacency list. In the adjacency list representation

of graph, we will maintain two lists. First list will keep track of all the nodes in the graph and the second list will maintain a list of adjacency nodes for each node to keep the track of the edges.

→ Structure of a node in the vertex list:

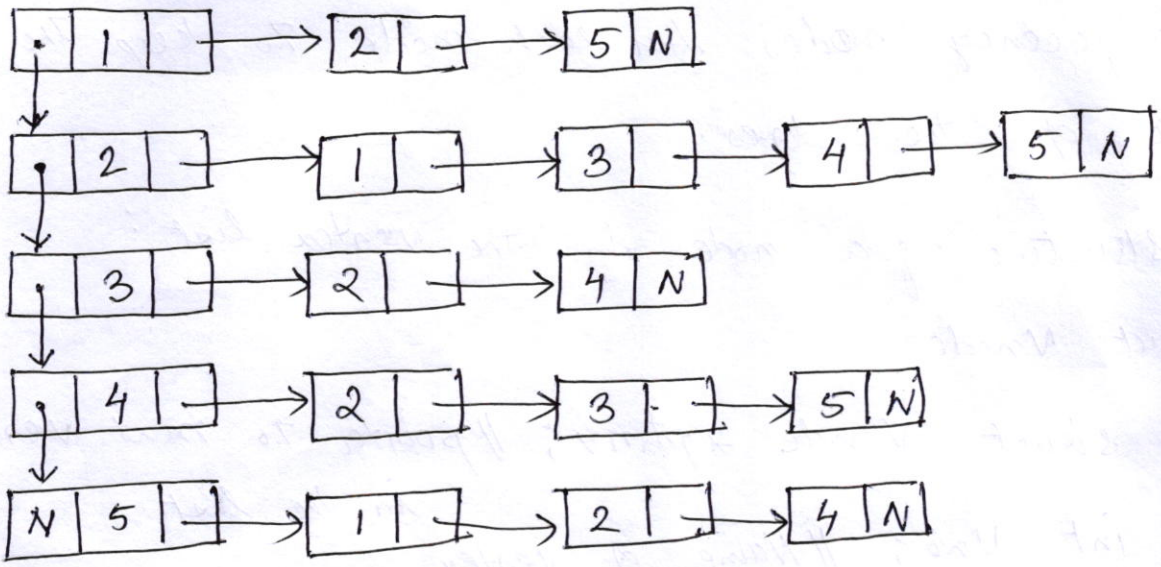
```
struct Vnode
{ struct Vnode *ptrnv; // pointer to next vertex
                          in the list
  int Vno; // Name of vertex
  Struct Enode *ptrav; // pointer to adjacent
                          vertex to this node.
};
struct Vnode vertex;
```



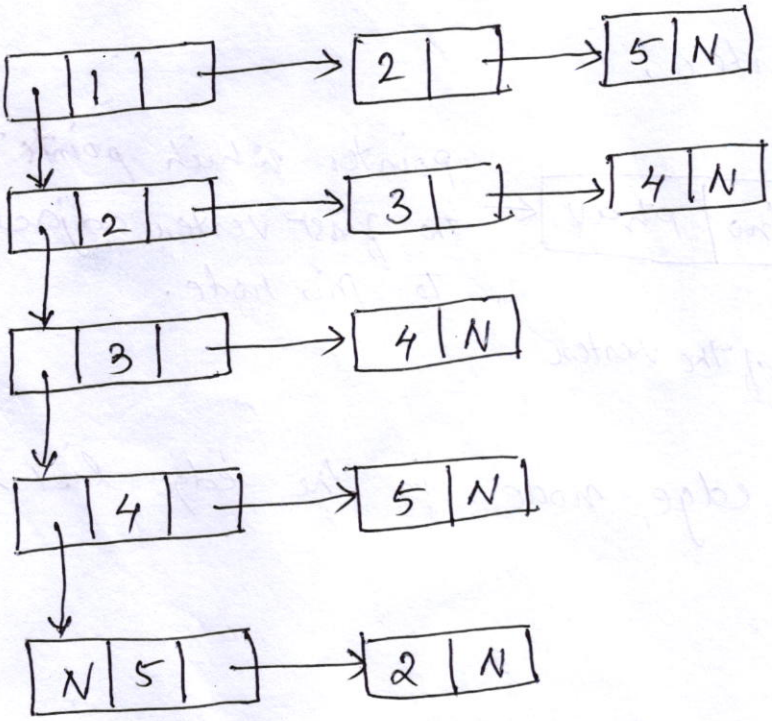
→ Structure of an edge node in the edge list:

```
Struct Enode
{ int Vno;
  struct Enode .*ptrav;
};
struct Enode edge;
```

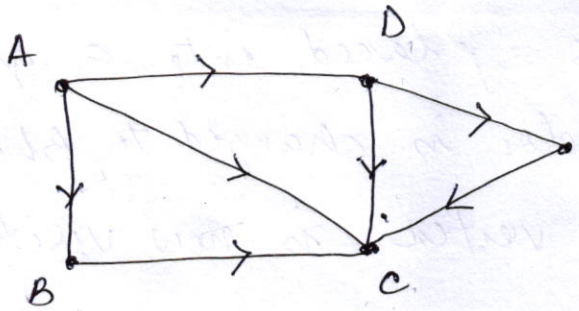
Destination node of the edge → | Vno | ptrAV | ← Pointer which points to the next adjacent node.



Adjacency list of the graph in fig 1.

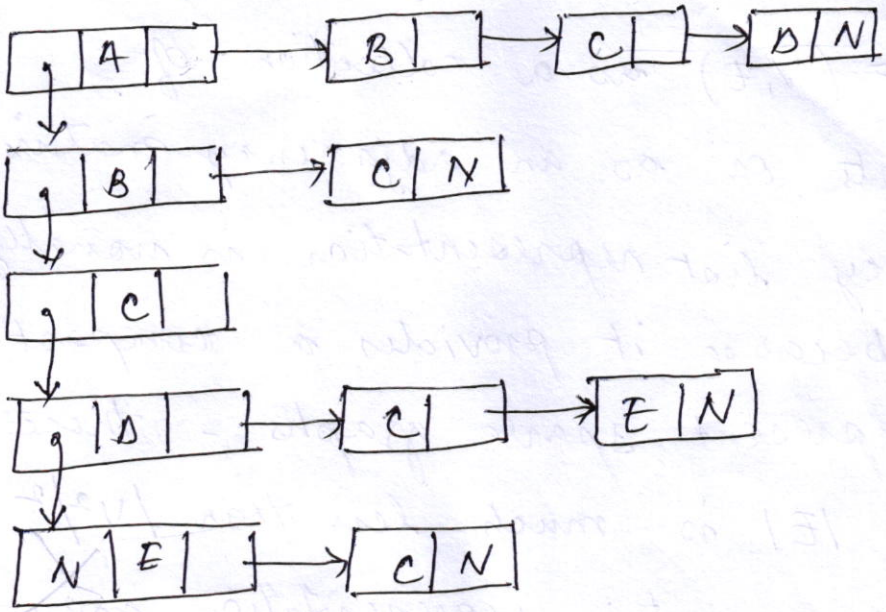


Adjacency list of the graph in fig 2.



Node	Adjacency list
A	B, C, D
B	C
C	
D	C, E
E	C

Adjacency list of the above graph



⊛ Traversal of a Graph.

There are two standard ways to traverse a graph.

1. Depth First Search (DFS - Stack)
2. Breadth First Search (BFS - Queue).

To distinguish between the visited vertices and the unvisited vertices, we give the color to each and every node of the graph. Initially all the vertices have color WHITE to indicate that they are not yet visited.

When a node is placed into a queue or stack, then the color is changed to BLACK to indicate that the vertex is now visited.

(*) There are two standard ways to represent a graph. $G = (V, E)$ as a collection of adjacency lists or as an adjacency matrix.

The adjacency list representation is usually preferred, because it provides a compact way to represent sparse graphs - those for which $|E|$ is much less than $|V|^2$

An adjacency matrix representation may be preferred, however when the graph is dense - $|E|$ is close to $|V|^2$.

(*) Breadth First Search (BFS)

Given a graph $G = (V, E)$ and a source vertex s, breadth first search systematically explores the edges of G to discover every vertex that is reachable from s. It produces a breadth-first tree with s that contains all reachable vertices. For any vertex v

⊛ Breadth-first search

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For every vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k+1$.

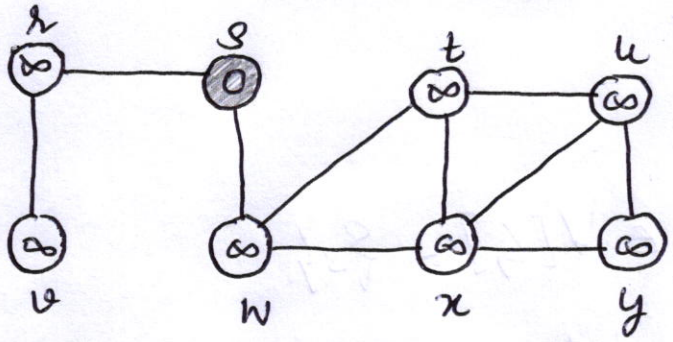
The breadth-first search procedure BFS below assumes that the input graph $G = (V, E)$ is represented using adjacency lists. The color of each vertex $u \in V$ is stored in the variable color[u], and the predecessor of u is stored in the variable $\pi[u]$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $\pi[u] = NIL$. The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$. The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

```
BFS(G, s)
1.  for each vertex u ∈ V[G] - {s}
2.       do color[u] ← WHITE
3.          d[u] ← ∞
4.          π[u] ← NIL
5.  color[s] ← GRAY
6   d[s] ← 0
7.  π[s] ← NIL
8.  Q ← ∅
9.  ENQUEUE (Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12.         for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                     then color[v] ← GRAY
15                          d[v] ← d[u] + 1
16                          π[v] ← u
17                          ENQUEUE(Q, v)
18       color[u] ← BLACK
```

Adjacency List

r - v, s
s - r, w
t → w, x, u
u → t, x, y
v - r
w - s, t, x
x - w, t, u, y.
y - x, u.

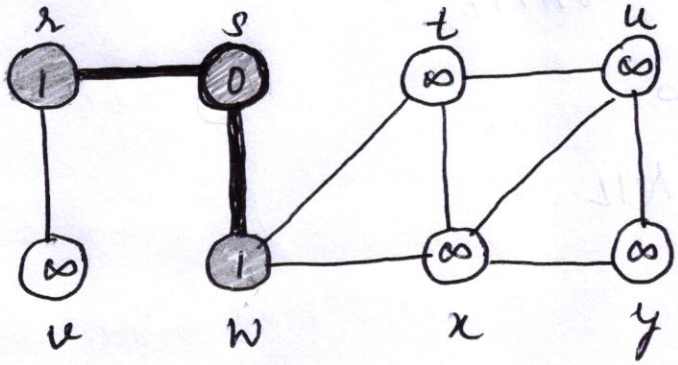
(a)



Q | s |
d[u] 0

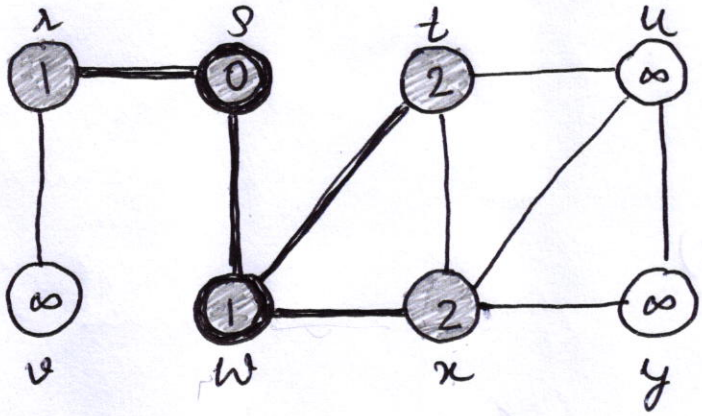
Adj[s] → w, r.

(b)



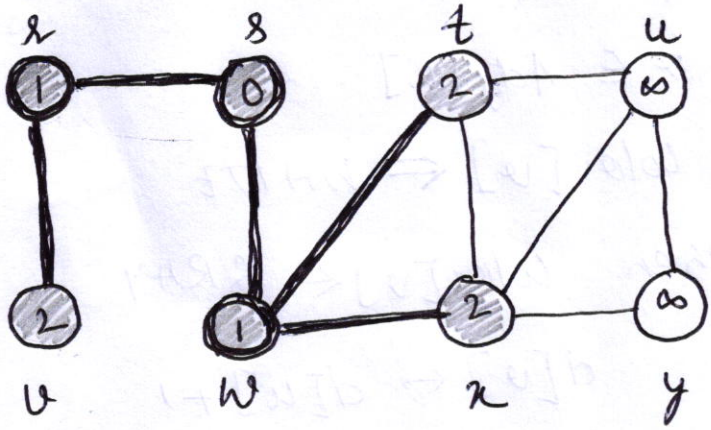
Q | w | r |
1 1

(c)



Q | r | t | x |
1 2 2

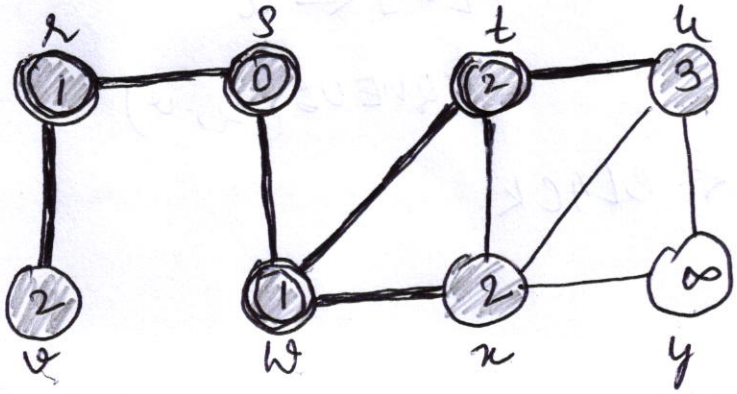
(d)



Q | t | x | v |
2 2 2

t - w, x, u.

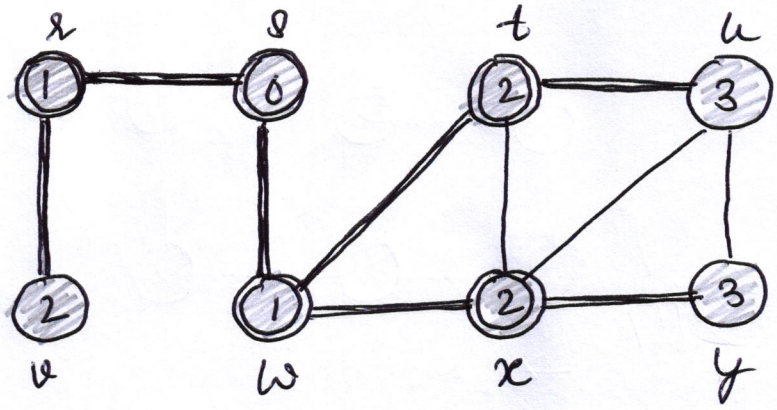
(e)



Q | x | v | u |
2 2 3

$x \to w, t, y$.

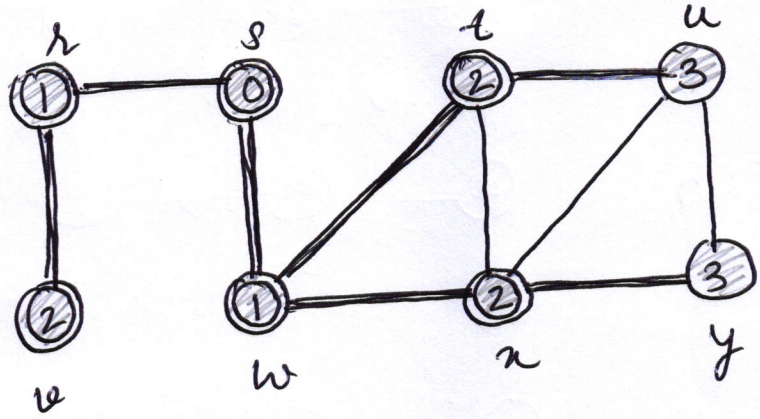
(f)



Q

v	u	y
2	3	3

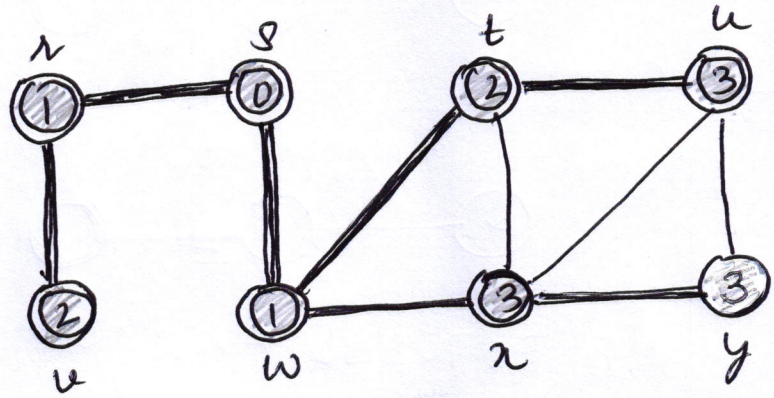
(g)



Q

u	y
3	3

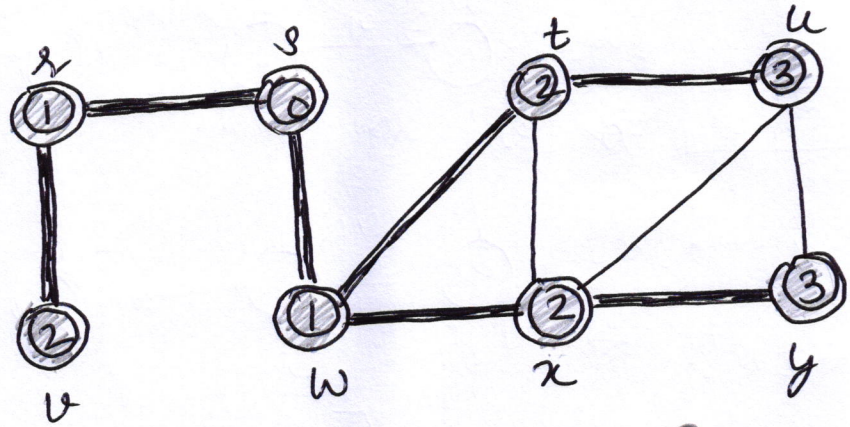
(h)



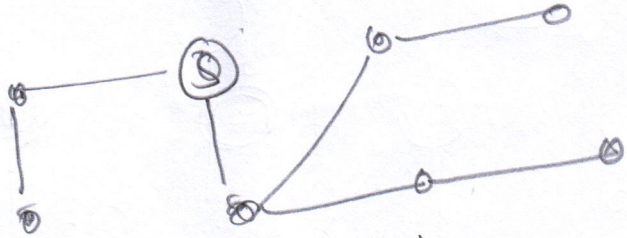
Q

y
3

(i)



Q $\emptyset$



BFS tree.

⊛ Depth-first Search

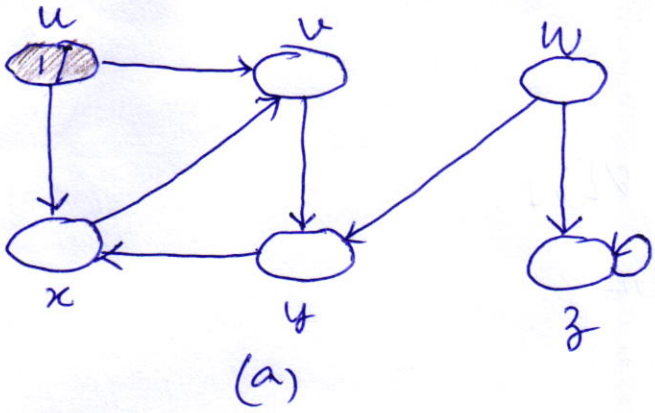
The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph wherever possible. In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in breadth-first search, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, depth-first search records this event by setting v's predecessor field $\pi[v]$ to u. Unlike BFS, whose predecessor

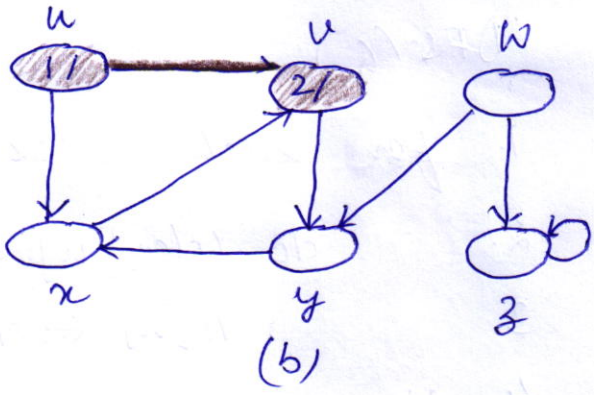
subgraph forms a tree, the predecessor subgraph produced by DFS may be composed of several trees, because the search may be repeated from multiple sources.

Besides creating a depth-first forest, DFS also timestamps every vertex. Each vertex v has two timestamps: the first timestamp d[v] records when v is first discovered (and grayed) and the second timestamp f[v] records when the search finishes examining v's adjacency list (and blackens v).

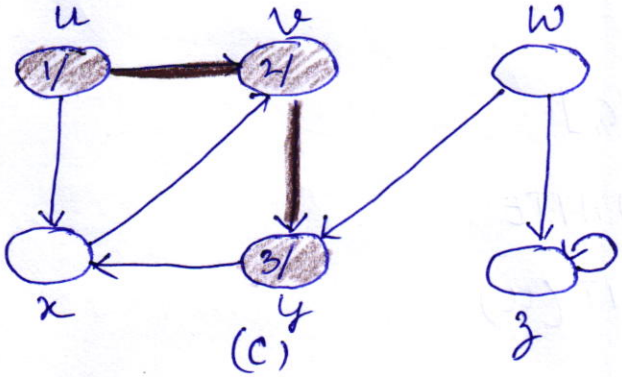
```
DFS(G)
1. for each vertex u ∈ V[G]
2.     do color[u] ← WHITE
3.        π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]
6.     do if color[u] = WHITE
7.        then DFS-VISIT(u)

DFS-VISIT(u)
1. color[u] ← GRAY  / White vertex u has just been discovered
2. time ← time + 1
3. d[u] ← time
4. for each v ∈ Adj[u]  / Explore edge (u, v)
5.     do if color[v] = WHITE
6.        then π[v] ← u
7.             DFS-VISIT(v)
8. color[u] ← BLACK  / Blacken u; it is finished
9. f[u] ← time ← time + 1
```


u
v
w
1
x
y
z

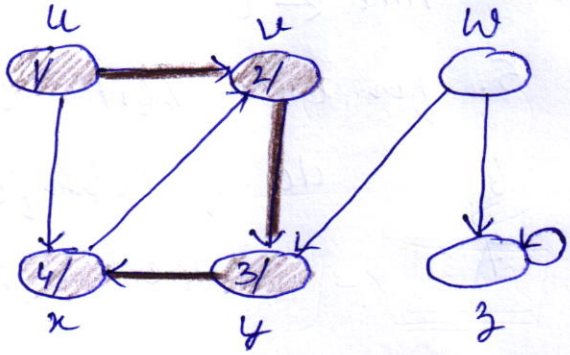
(a)


u
v
w
1
2
x
y
z

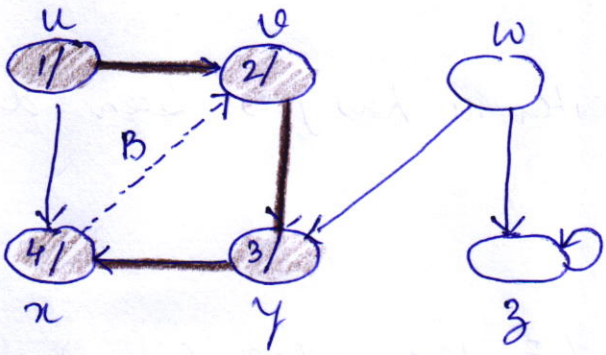
(b)


u
v
w
1
2
3
x
y
z

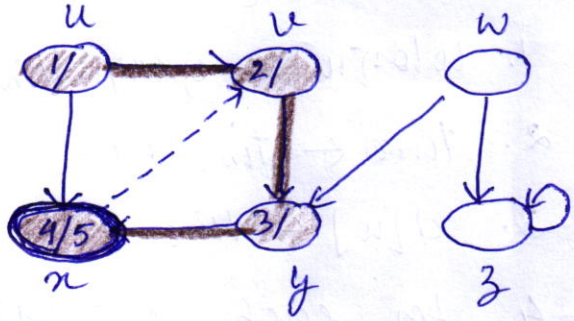
(c)


u
v
w
1
2
4
3
x
y
z

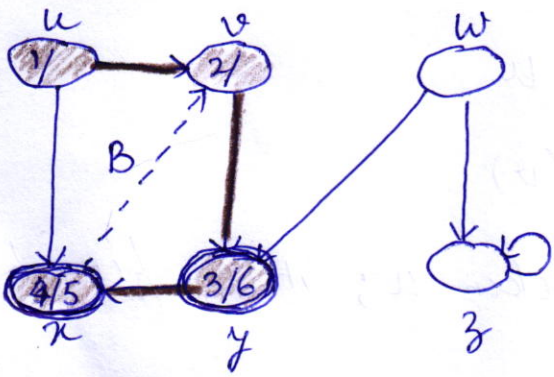
(d)


u
v
w
1
2
B
4
3
x
y
z

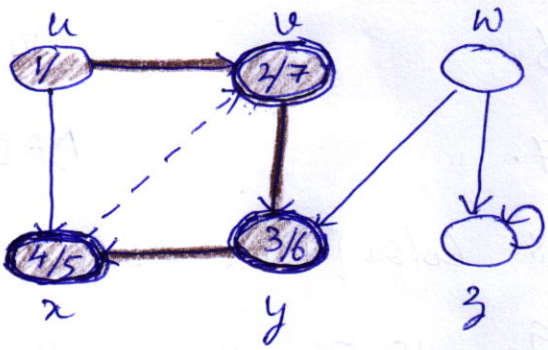
(e)


u
v
w
1
2
4/5
3
x
y
z

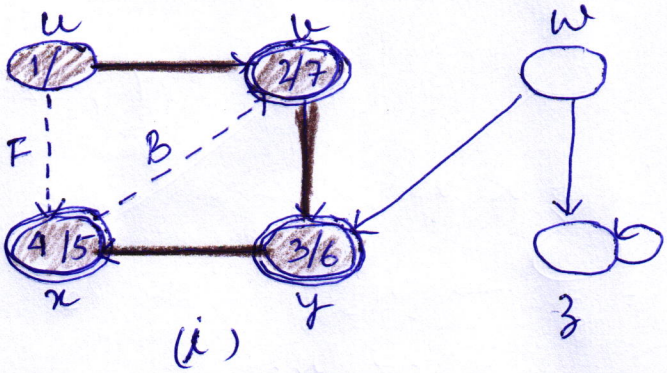
(f)


u
v
w
1
2
B
4/5
3/6
x
y
z

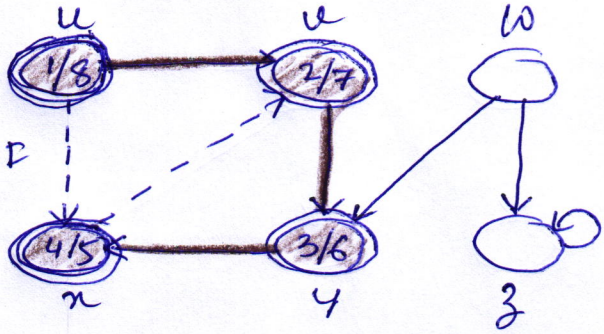
(g)


u
v
w
1
2/7
4/5
3/6
x
y
z

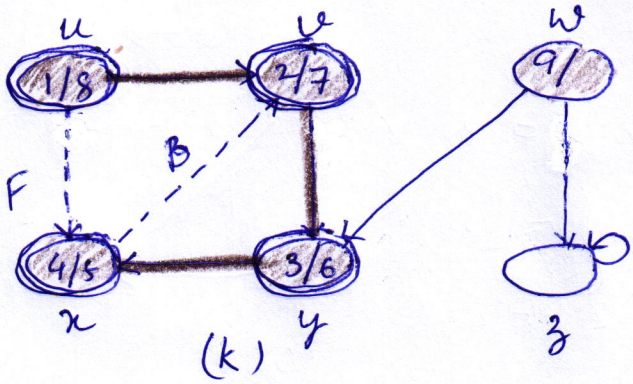
(h)



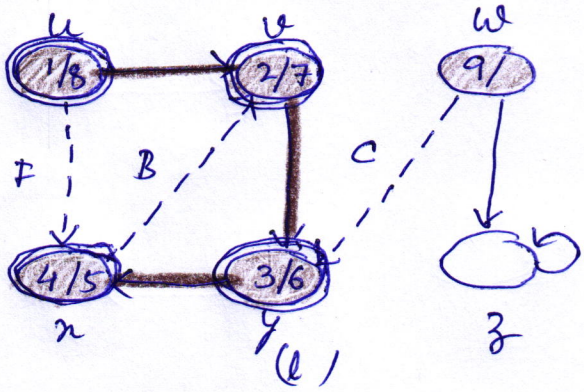
(i)



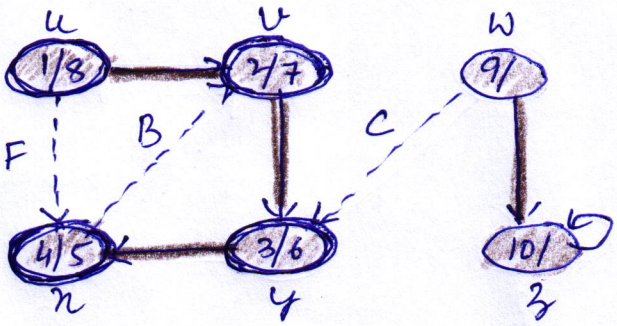
(j)



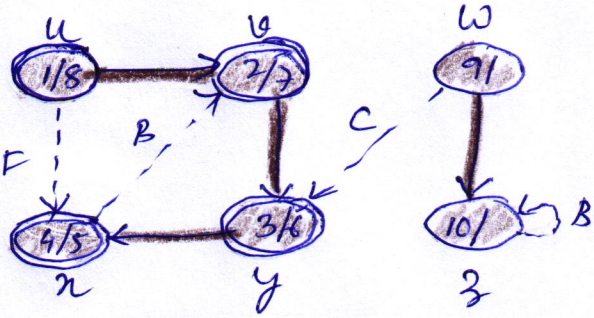
(k)



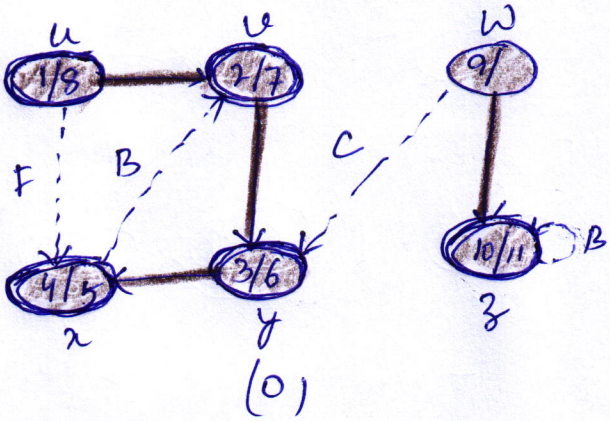
(l)



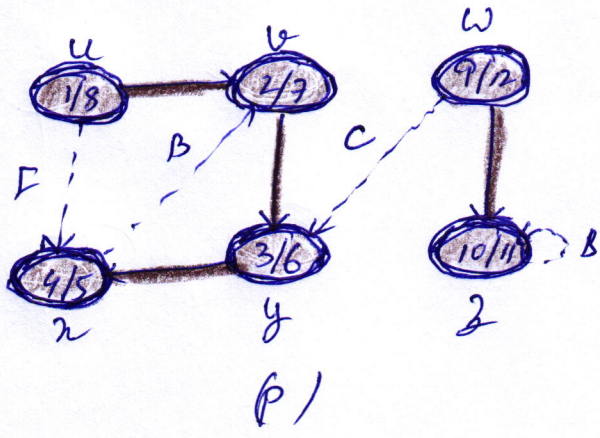
(m)



(n)



(o)



(p)

B, C or F: Non tree edges labelled according to whether they are back, cross or forward.