# Data Structure and Algorithms

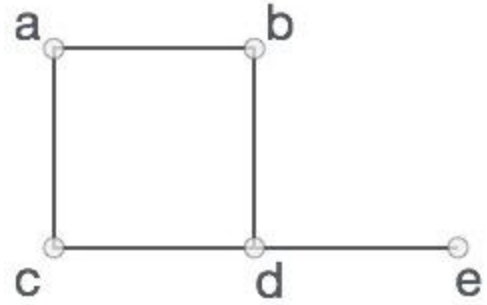
## Topic :- Graphs

# Graph :-

1. Definition
2. Types of graphs
3. Terminology
4. Representation
5. Uses of graph

# What is Graph ?

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.
- Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −




- In the graph,

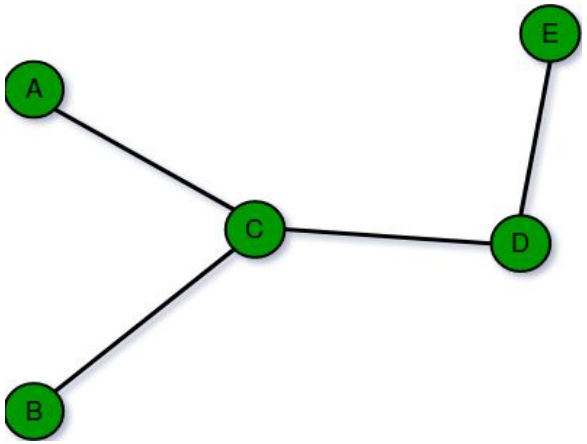
  V = {a, b, c, d, e}

  E = {ab, ac, bd, cd, de}

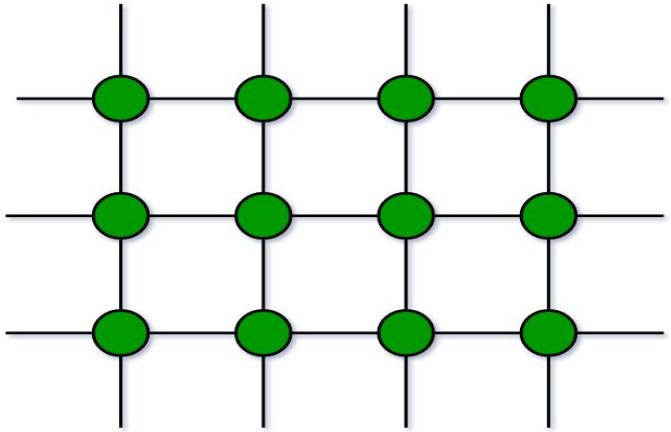
- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices .

# Types of Graph

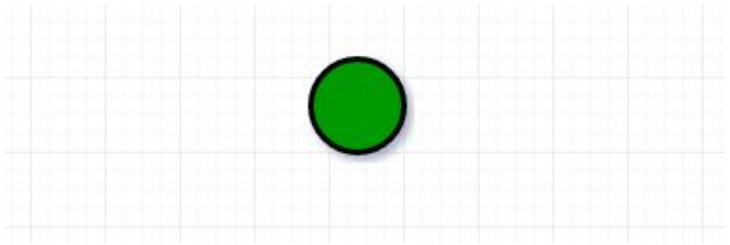
1. **Finite Graphs:** A graph is said to be finite if it has finite number of vertices and finite number of edges.




1. Finite graph



2. Infinite graph

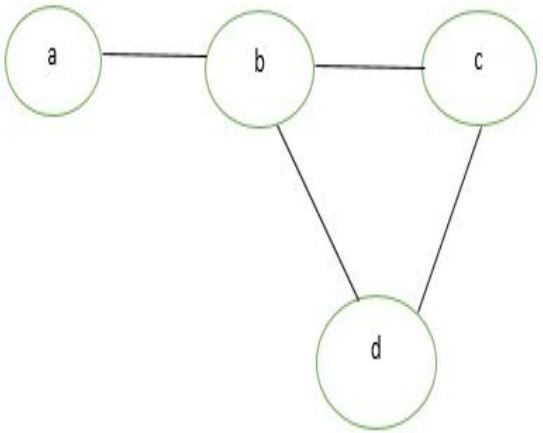


3. Trivial graph

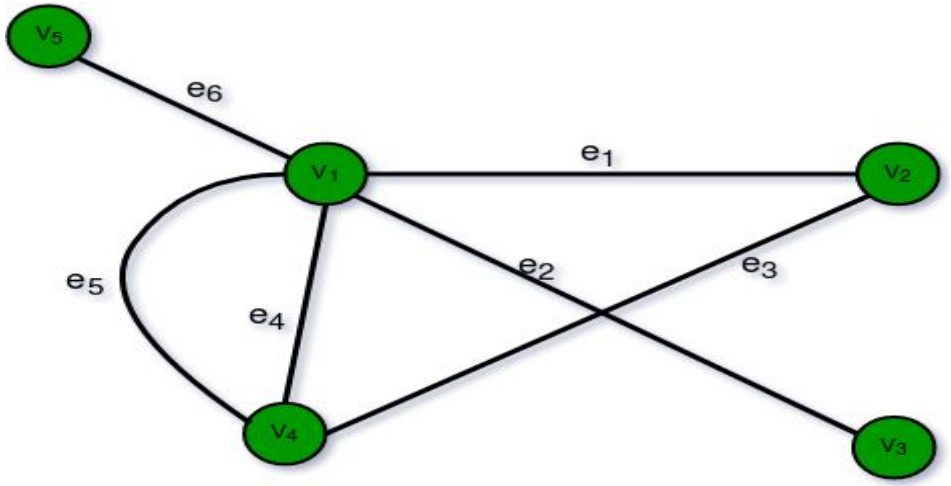
2. **Infinite Graph:** A graph is said to be infinite if it has infinite number of vertices as well as infinite number of edges.

3. **Trivial Graph:** A graph is said to be trivial if a finite graph contains only one vertex and no edge.

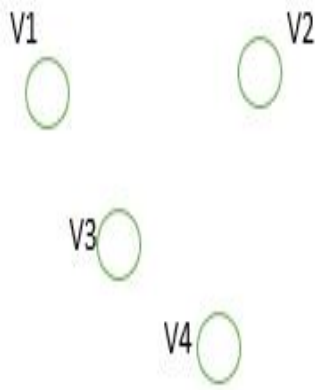
4. **Simple Graph:** A simple graph is a graph which does not contains more than one edge between the pair of vertices. A simple railway tracks connecting different cities is an example of simple graph.




4. Simple graph




5. Multi graph




6. Null graph

**5. Multi Graph:** Any graph which contain some parallel edges but doesn't contain any self-loop is called multi graph. For example A Road Map

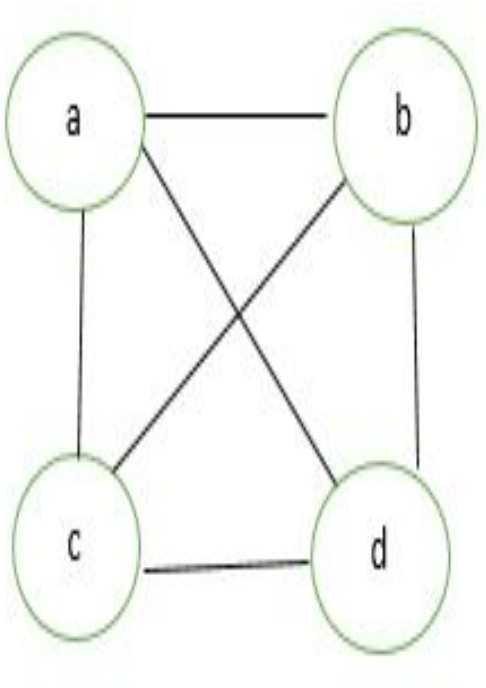
**Parallel Edges:** If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.

**Loop:** An edge of a graph which join a vertex to itself is called loop or a self-loop.

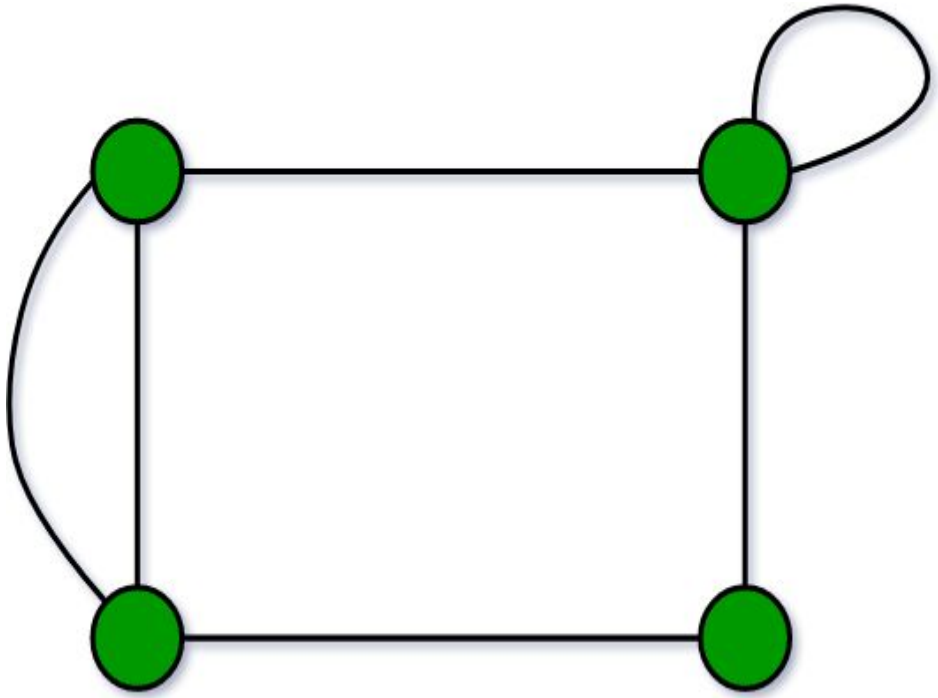
**6. Null Graph:** A graph of order n and size zero that is a graph which contain n number of vertices but do not contain any edge.

**7. Complete Graph:** A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.

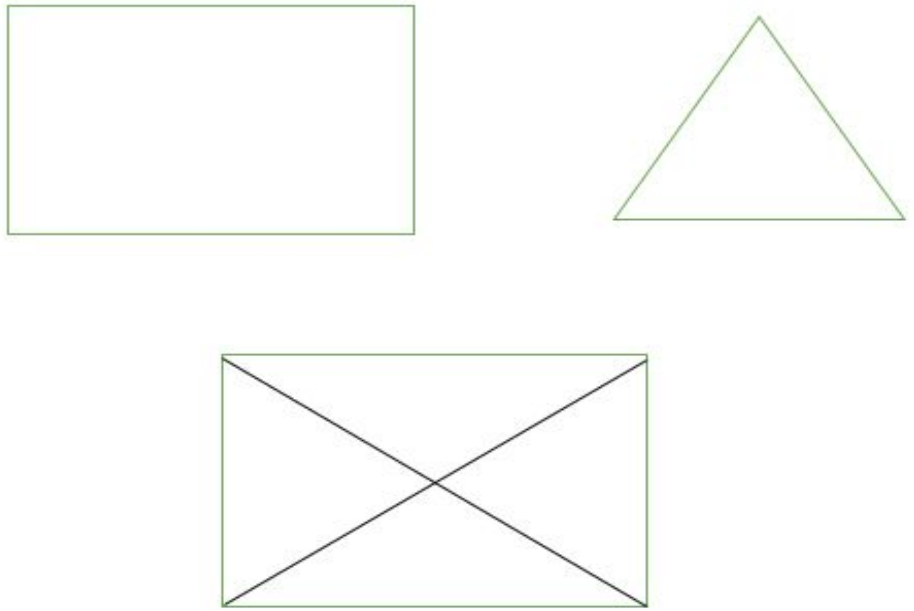
**8. Pseudo Graph:** A graph G with a self loop and some multiple edges is called pseudo graph.




**7. Complete Graph**



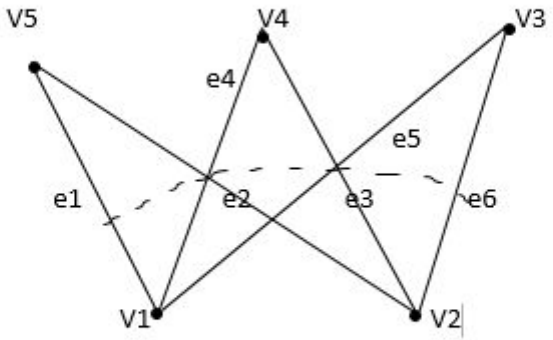
**8. Pseudo Graph**



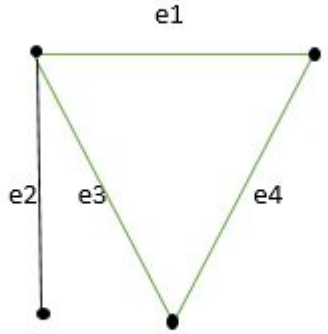
9. **Regular Graph**

9. **Regular Graph:** A simple graph is said to be regular if all vertices of a graph G are of equal degree. All complete graphs are regular but vice versa is not possible.

**10. Bipartite Graph:** A graph G = (V, E) is said to be bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has its one end in V1(G) and other end in V2(G).
The partition V1 U V2 = V is called Bipartite of G.
Here in the figure:
V1(G)={V5, V4, V3}
V2(G)={V1, V2}




**10. Bipartite Graph**




**11. Labelled Graph**

**11. Labelled Graph:** If the vertices and edges of a graph are labelled with name, data or weight then it is called labelled graph. It is also called *Weighted Graph*.
12. **Connected or Disconnected Graph:** A graph G is said to be connected if for any pair of vertices (Vi, Vj) of a graph G are reachable from one another. Or a graph is said to be connected if there exist atleast one path between each and every pair of vertices in graph G, otherwise it is disconnected. A null graph with n vertices is disconnected graph consisting of n components. Each component consist of one vertex and no edge.

## Two Important kinds of graphs

- Directed
- Undirected

1. A **directed** graph, or **digraph**, is a graph in which the edges are ordered pairs
    - (v, w) ≠ (w, v)
2. An **undirected** graph is a graph in which the edges are unordered pairs
    - (v, w) == (w, v)

# Directed vs. Undirected Graphs

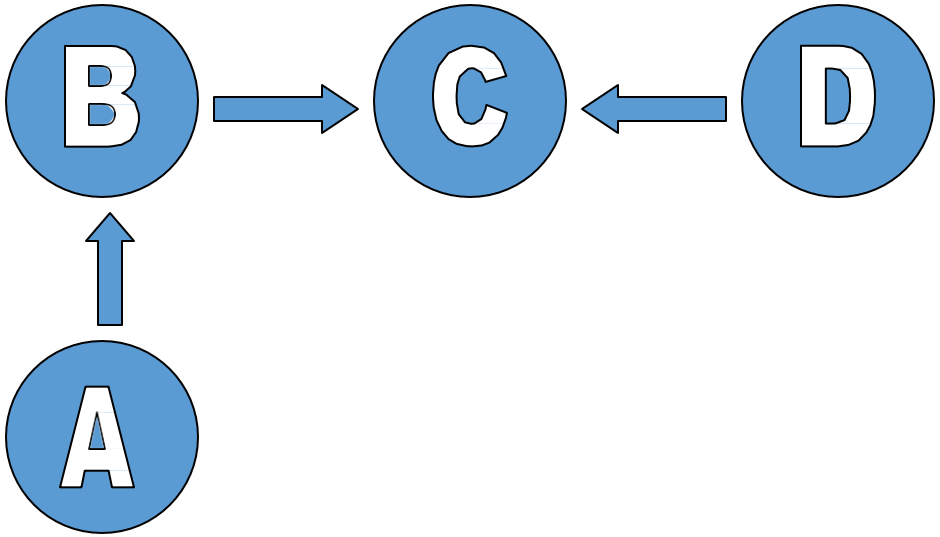
- Undirected edge has no orientation (no arrow head)
- Directed edge has an orientation (has an arrow head)
- Undirected graph – all edges are undirected
- Directed graph – all edges are directed

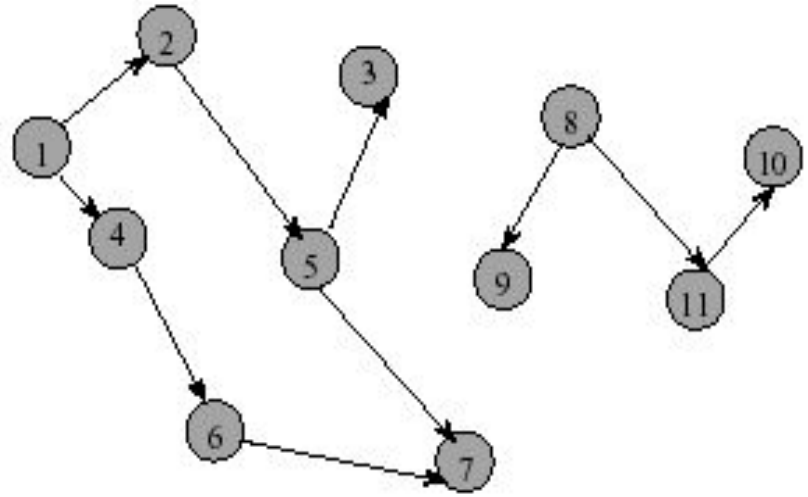

**undirected edge**


**directed edge**

# Introduction: Directed Graphs

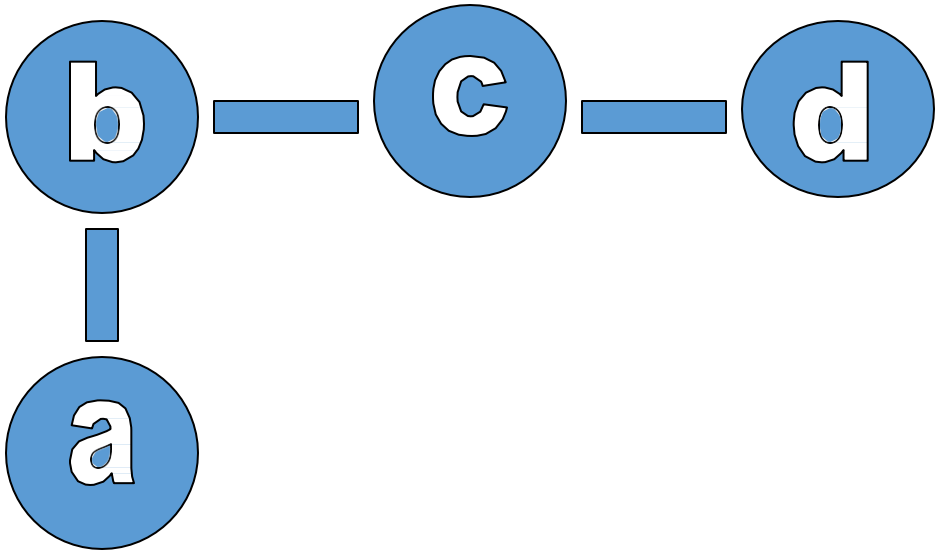
- In a directed graph, the edges are arrows.
- Directed graphs show the flow from one node to another and not vise  versa.

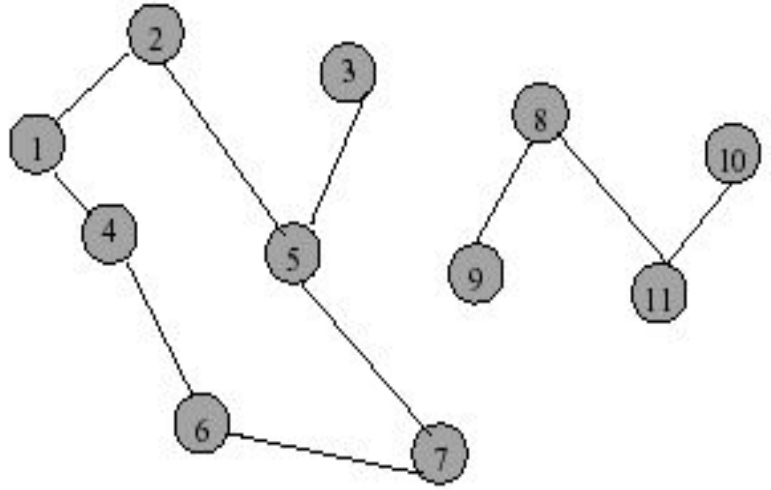


# Introduction: Undirected Graphs

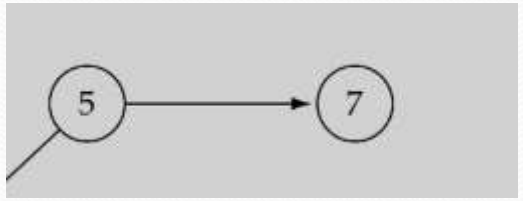
- In a Undirected graph, the edges are lines.
- UnDirected graphs show a relationship between two nodes.



# Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge

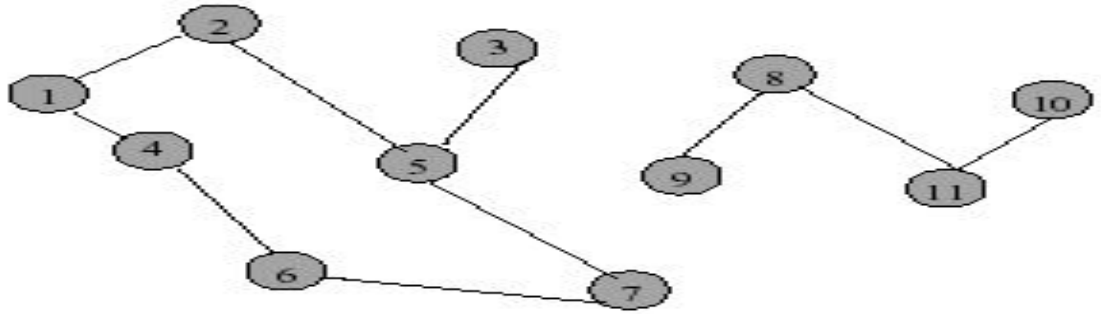



5 is adjacent to 7
7 is adjacent from

- Path: a sequence of vertices th at connect two nodes in a graph
- A simple path is a path in which all vertices, except possibly in the first and last, are different.
- Complete graph: a graph in which every vertex is directly connected to every other vertex

# Terminology

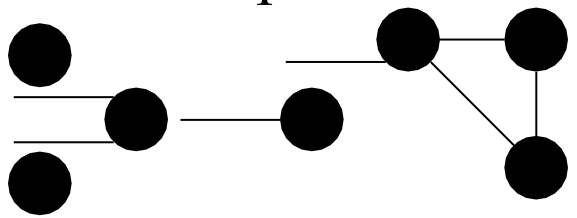
- A cycle is a simple path with the same start and end vertex.
- The **degree** of vertex *i* is the no. of edges incident on vertex *i*.



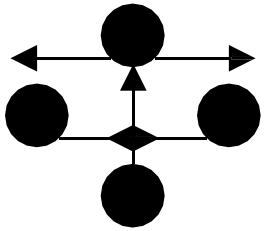

e.g., degree(2) = 2, degree(5) = 3, degree(3) = 1

# Terminology

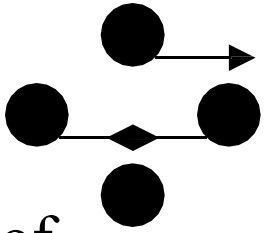
Undirected graphs are *connected* if there is a path between any two vertices



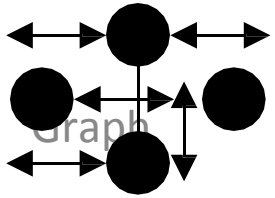
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



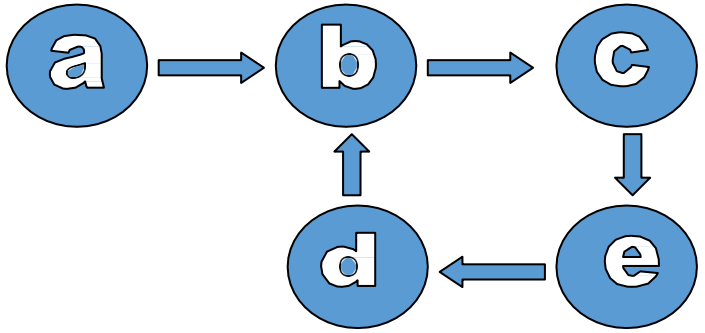
Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices



# Terminology

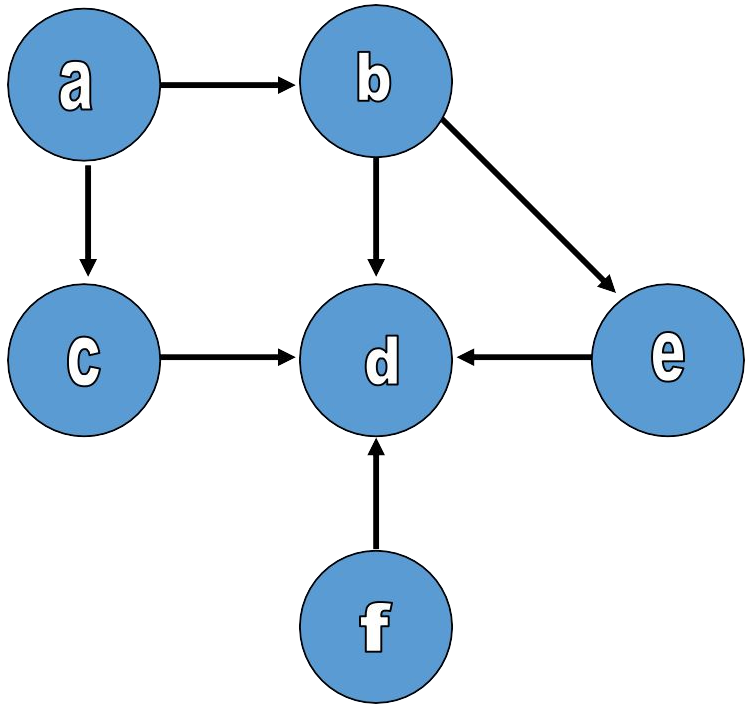



- An **acyclic path** is a path which does not follow a sequence.
- A **cyclic path** is a path such that
  - There are at least two vertices on the path
  - $w_1 = w_n$ (path starts and ends at same vertex)
  - And also maintains the sequence

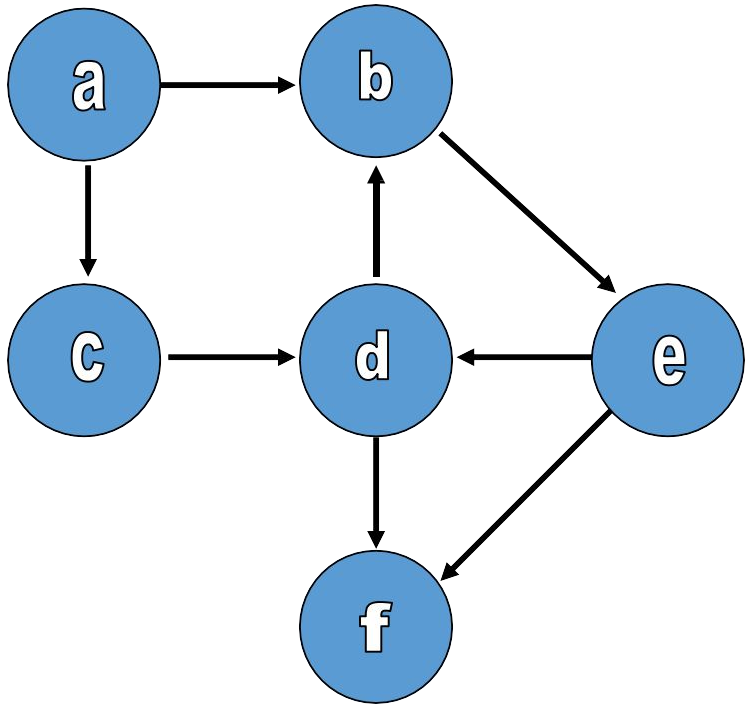
# Test Your Knowledge

Cyclic or Acyclic?

1.



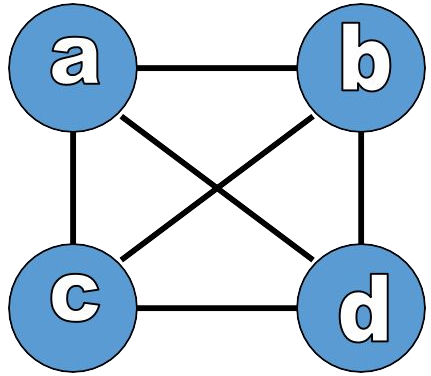

2.



# Terminology

- A directed graph that has *no* cyclic paths is called a **DAG** (a Directed Acyclic Graph).
- An undirected graph that has an edge between every pair of vertices is called a **complete** graph.

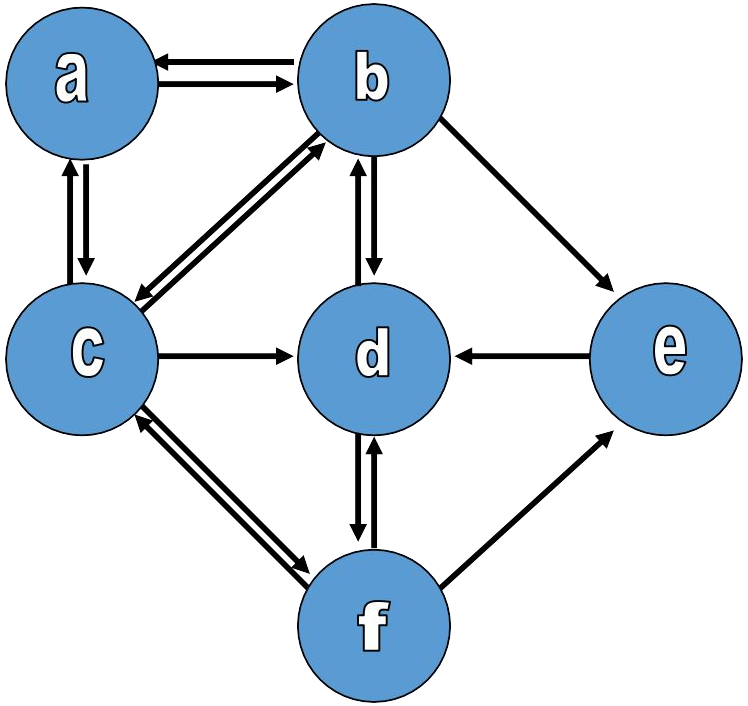
**Note:** A directed graph can also be a complete graph; in that case, there must be an edge from every vertex to every other vertex.



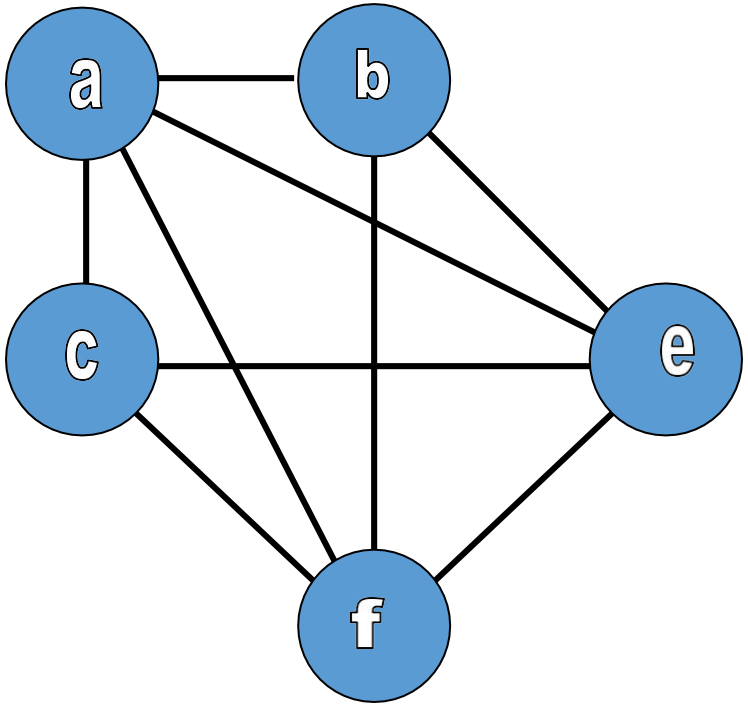
# Test Your Knowledge

## Complete, or “Acomplete” (Not Complete)

**1.**



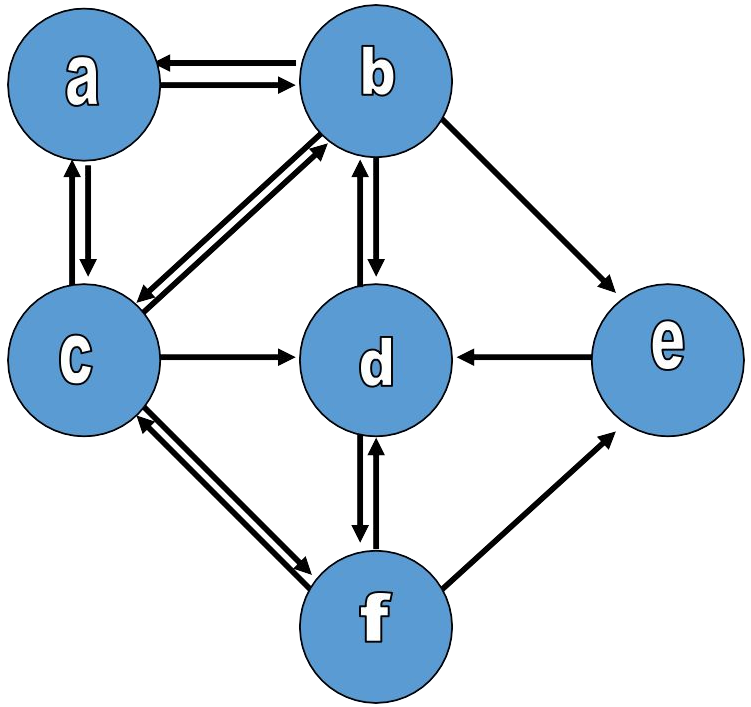

**2.**



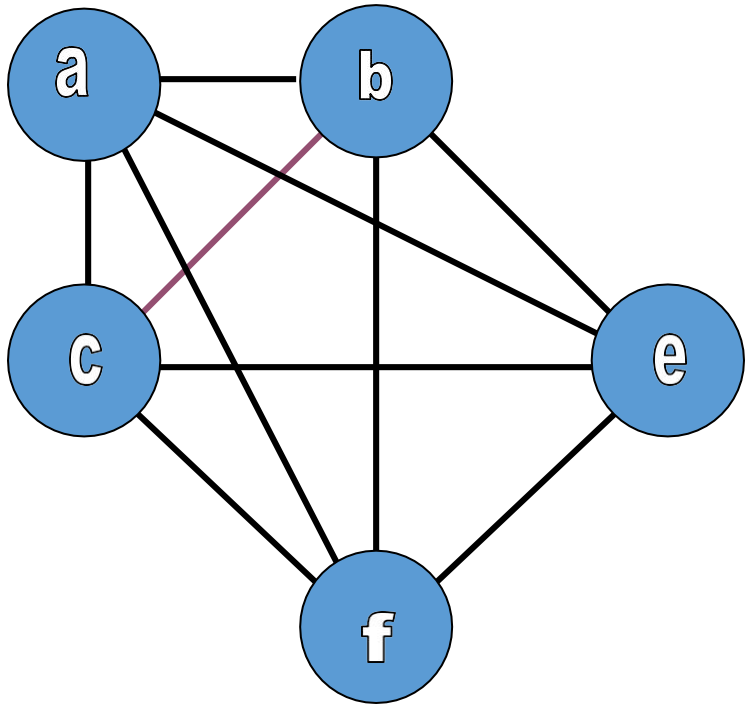
# Test Your Knowledge

## Complete, or “Acomplete” (Not Complete)

1.




2.

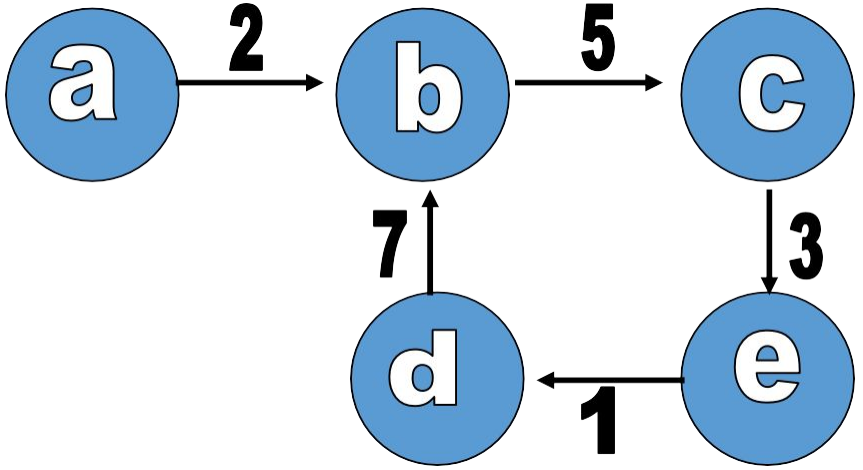


# Terminology

- An undirected graph is **connected** if a path exists from every vertex to every other vertex
- A directed graph is **strongly connected** if a path exists from every vertex to every other vertex
- A directed graph is **weakly connected** if a path exists from every vertex to every other vertex, disregarding the direction of the edge

# Terminolo
# gy

- A graph is known as a **weighted graph** if a weight or  metric is associated with each edge.



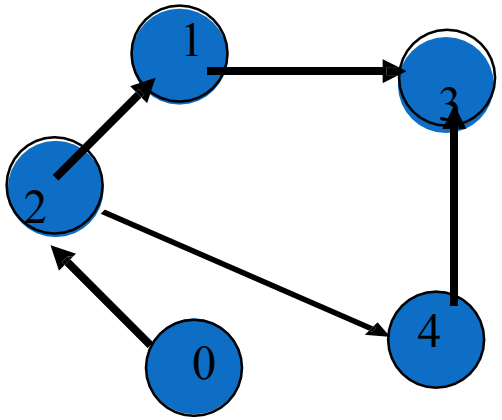
# Representation of Graph

# Graph Representation

- For graphs to be computationally useful, they have to be conveniently represented in programs
- There are two computer representations of graphs:
  - Adjacency matrix representation
  - Adjacency lists representation

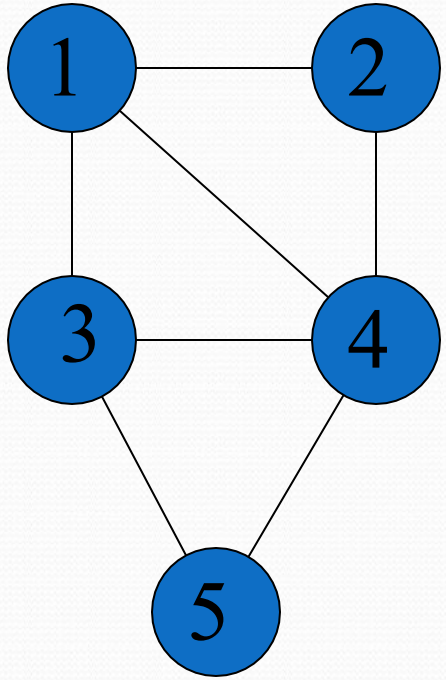
# Adjacency Matrix

- A square grid of boolean values
- If the graph contains N vertices, then the grid contains N rows and N columns
- For two vertices numbered I and J, the element at row I and column J is true if there is an edge from I to J, otherwise false




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | false | false | true | false | false |
| 1 | false | false | false | true | false |
| 2 | false | true | false | false | true |
| 3 | false | false | false | false | false |
| 4 | false | false | false | true | false |

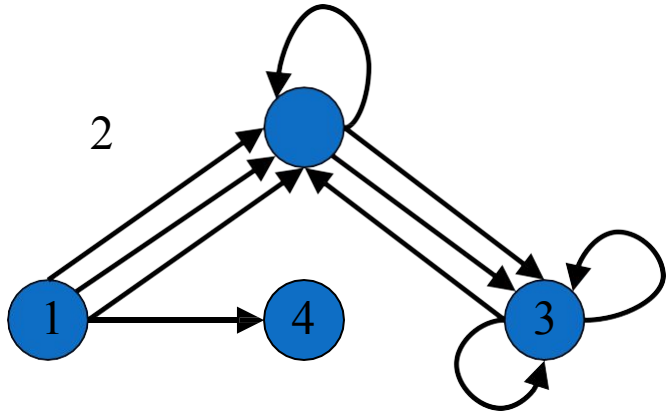
# Adjacency Matrix




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 | 1 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |

# Adjacency Matrix

## Directed Multigraphs




A :

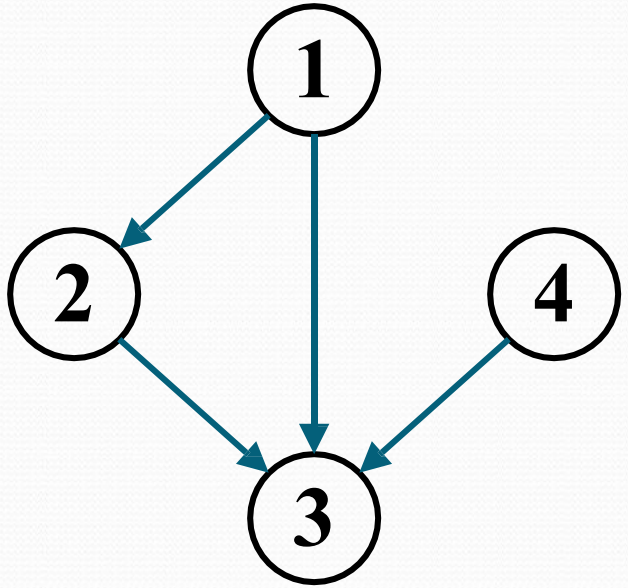
$$\begin{pmatrix} 0 & 3 & 0 & 1 \\ 0 & 1 & 2 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

# Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

# Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to $v$
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex

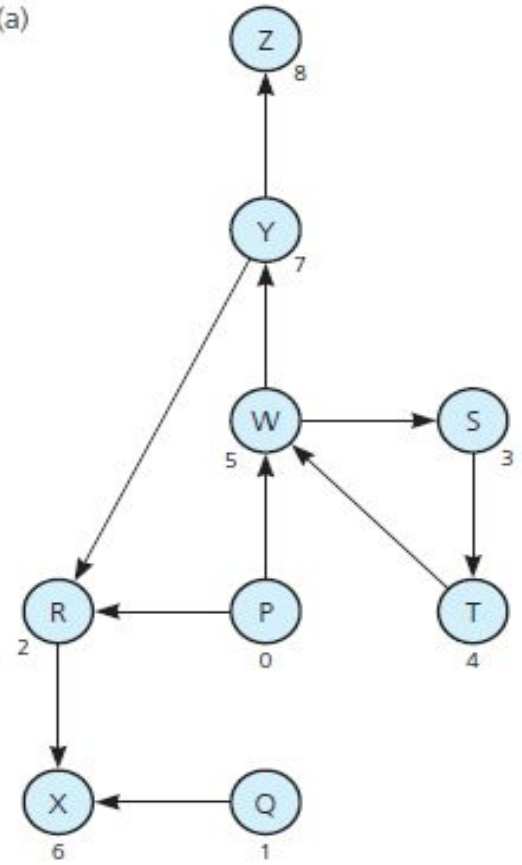


# Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex $v$ = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    $$\Sigma \text{ out-degree}(v) = |E|$$
    For undirected graphs, # items in adjacency lists is $\Sigma$
    $$\text{degree}(v) = 2\,|E|$$
- So: Adjacency lists take O(V+E) storage

# Implementing Graphs

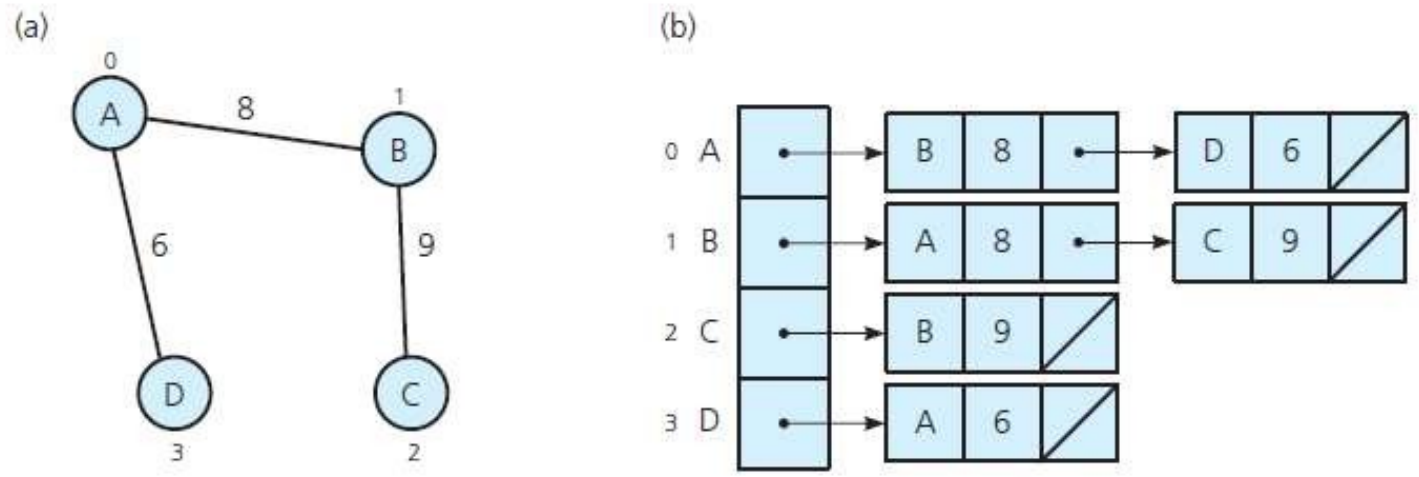
(a)




(b)

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | P | Q | R | S | T | W | X | Y | Z |
| 0 | P | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | Q | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | R | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | S | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | T | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | W | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 6 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | Y | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 8 | Z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Implementing Graphs




- (a) A weighted undirected graph and
  (b) its adjacency list

# Uses For Graphs

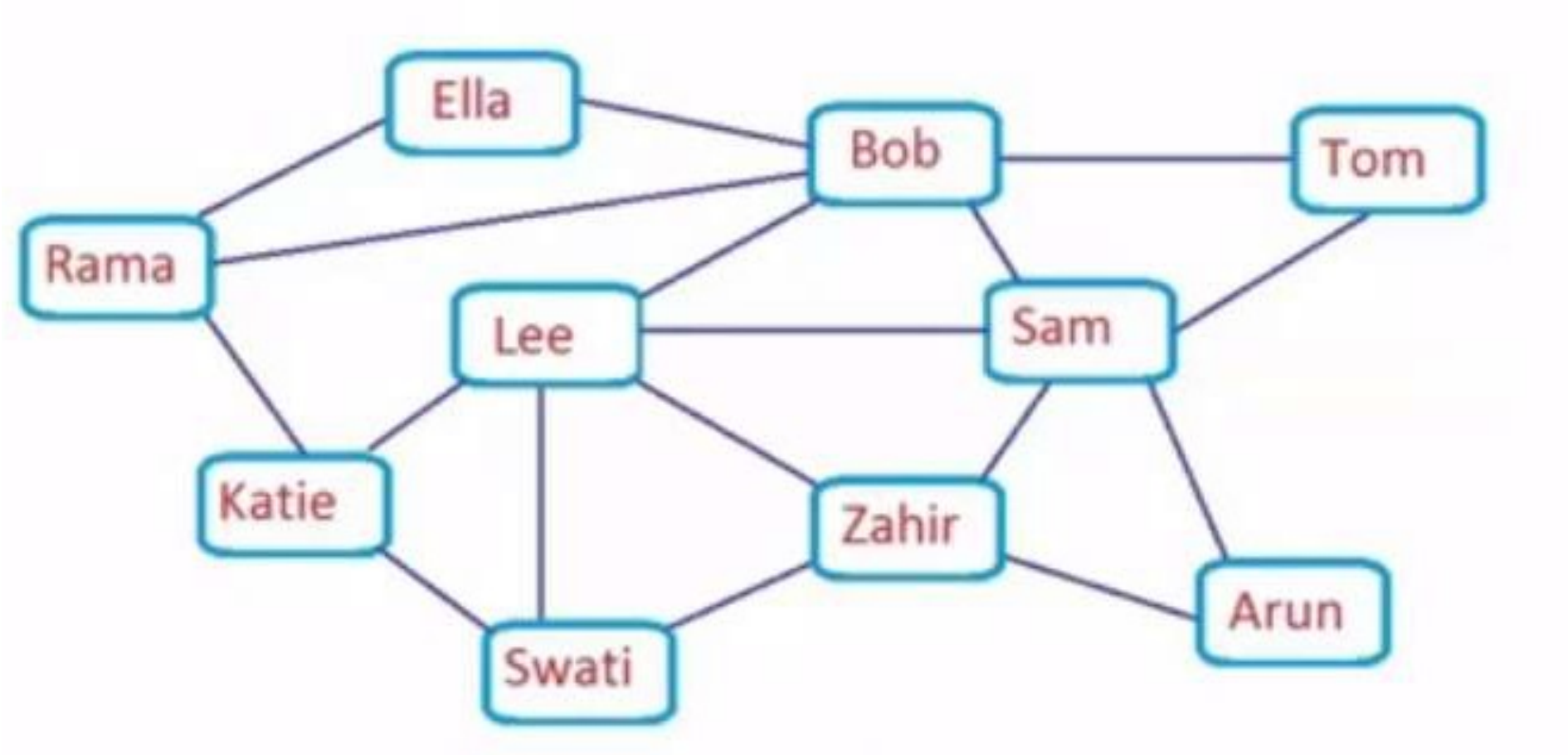
# Uses for Graphs

- **Computer network:** The set of vertices V represents the set of computers in the network. There is an edge (u, v) if and only if there is a direct communication link between the computers corresponding to u and v.

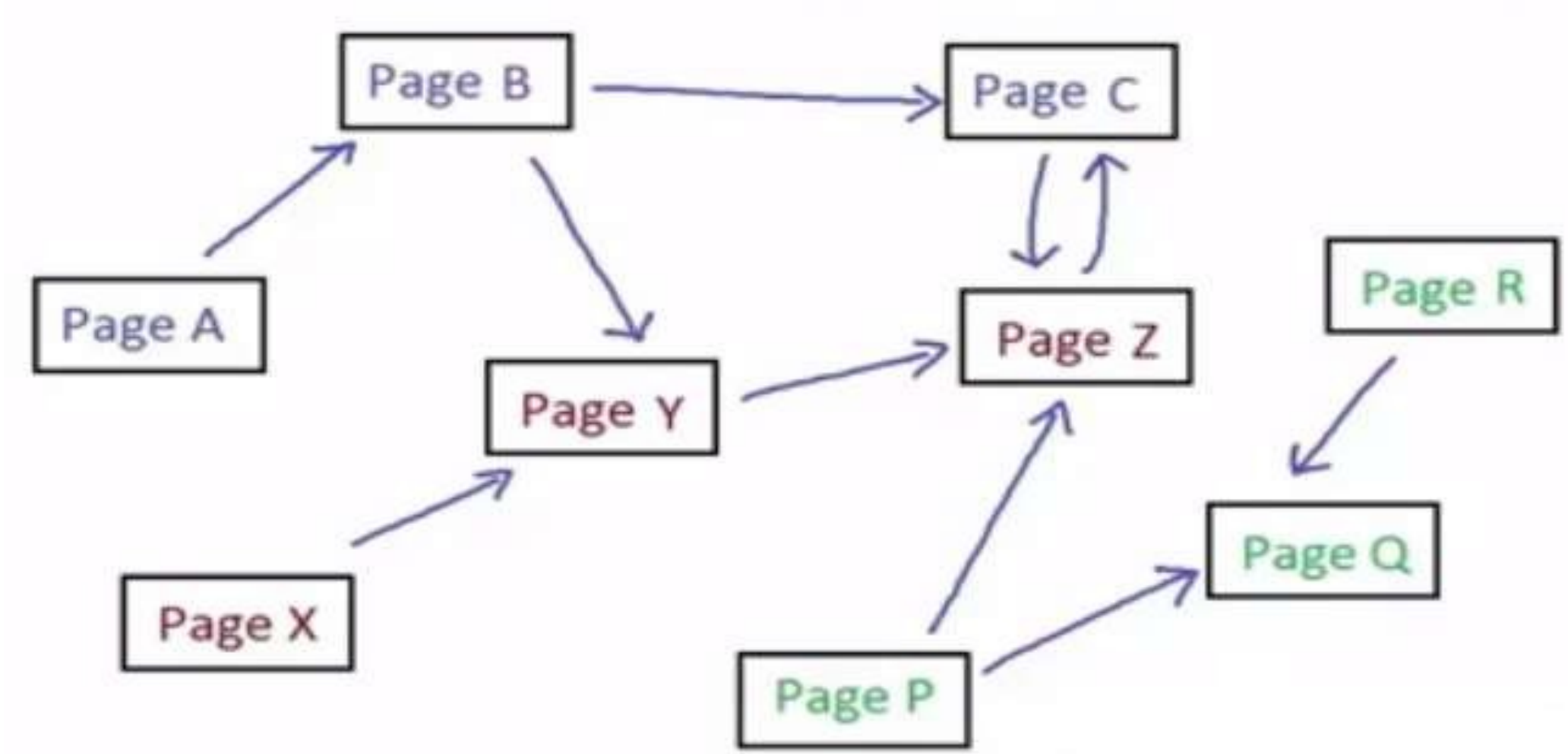
# Uses for Graphs

- **Two-Player Game Tree:** All of the possibilities in a board game like chess can be represented in a graph. Each vertex stands for one possible board position. (For chess, this is a very big graph!)

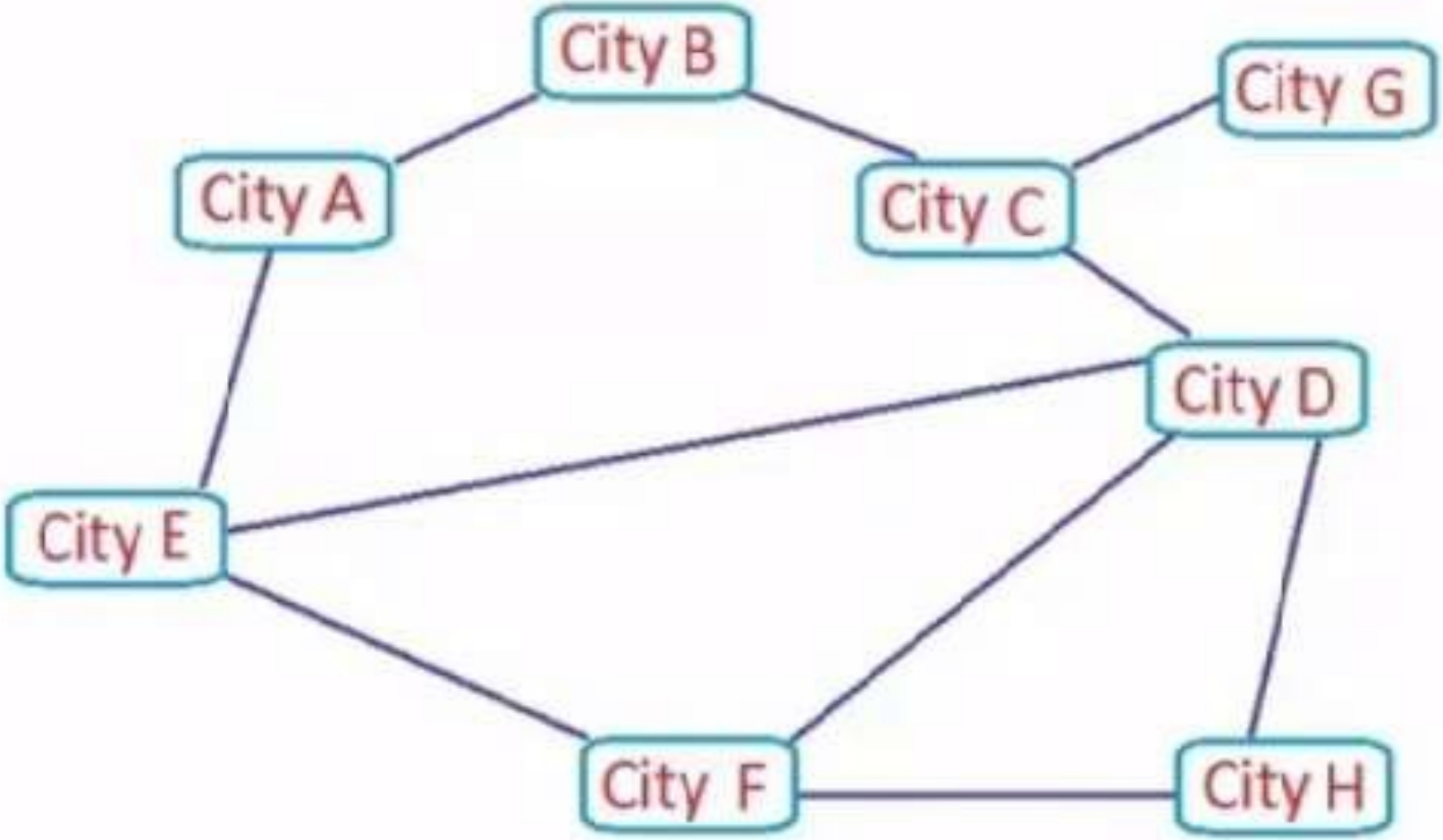
# Social Media (Facebook)



# Website
s

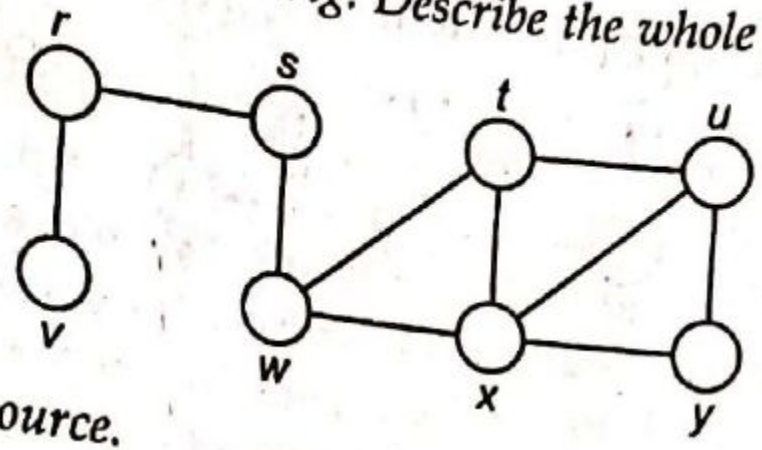


# Intercity Road Network



# BFS

...own in the Fig. Describe the whole process of BFS.




Using vertex u as the source.

Solution.

# DFS

# Topological Sorting

Directed acyclic graphs or DAG's are used for topological sorts. A topological sort of a directed acyclic graph $G=(V, E)$ is a linear ordering of $u, v \in V$ such that if $(u, v) \in E$ then $u$ appears before $v$ in this ordering. If $G$ is cyclic, no such ordering exists .

**TOPOLOGICAL-SORT($G$)**

1. call DFS ($G$) to compute finishing times $f[v]$ for each vertex $v$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Topological sort can also be viewed as placing all the vertices along a horizontal line so that all directed edges go from left to right. DAG's are used in many applications to indicate precedence.

We can perform a topological sort in time $\theta(V+E)$, since DFS takes $\theta(V+E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

```
TOPOLOGICAL-SORT (G)
1.  for each vertex u ∈ V
2.      do in-degree[u] ← 0
3.  for each vertex u ∈ V
4.      do for each v ∈ Adj[u]
5.          do in-degree[v] ← in-degree[v] +1
6.  Q ← ϕ
7.  for each vertex u ∈ V
8.      do if in-degree[u] = 0
9.          then ENQUEUE(Q,u)
10. while Q ≠ ϕ
11.     do u ← DEQUEUE(Q)
12.         output u
13.         for each v ∈ Adj[u]
14.             do in-degree[v] ← in-degree[v] – 1
15.                 if  in-degree[v] = 0
16.                 then ENQUEUE(Q,v)
17.     do if in-degree[u] ≠ 0
18.         then report that there is a cycle
```

# KOSARAJU ALGO

**Kosaraju's algorithm** efficiently computes the strongly connected components of a directed graph.

Kosaraju's algorithm

Strongly-connected Components (G)

1. call DFS(G) to compute finishing times $f[u]$ for each vertex $u$ .
2. compute $G^T$
3. call DFS($G^T$), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$
4. produce as output the vertices of each tree in the DFS forest formed in line 3 as a separate SCC.

# Brute force

1) Find all pairs shortest path using Floyd warshall algo.
2) Check if between any two pairs the distance is infinity
   ( i.e unreachable expect self loops)
   if True
      than
         component is not SCC
   ELSE
      Strongly Connected Componets

# data structure

## Topics:

**BFS – Breadth First Search**

**DFS – Depth First Search**

# What is BFS ?

- BFS stands for Breadth **F**irst **S**earch.
- BFS is an algorithm for traversing or searching a tree or graph data structures.
- It uses a queue data structure for implementation.
- In BFS traversal we visit all the nodes level by level and the traversal completed when all the nodes are visited.

# Algorithm for BFS :

**Step 1:** Initialize all nodes with status=1.(ready state)

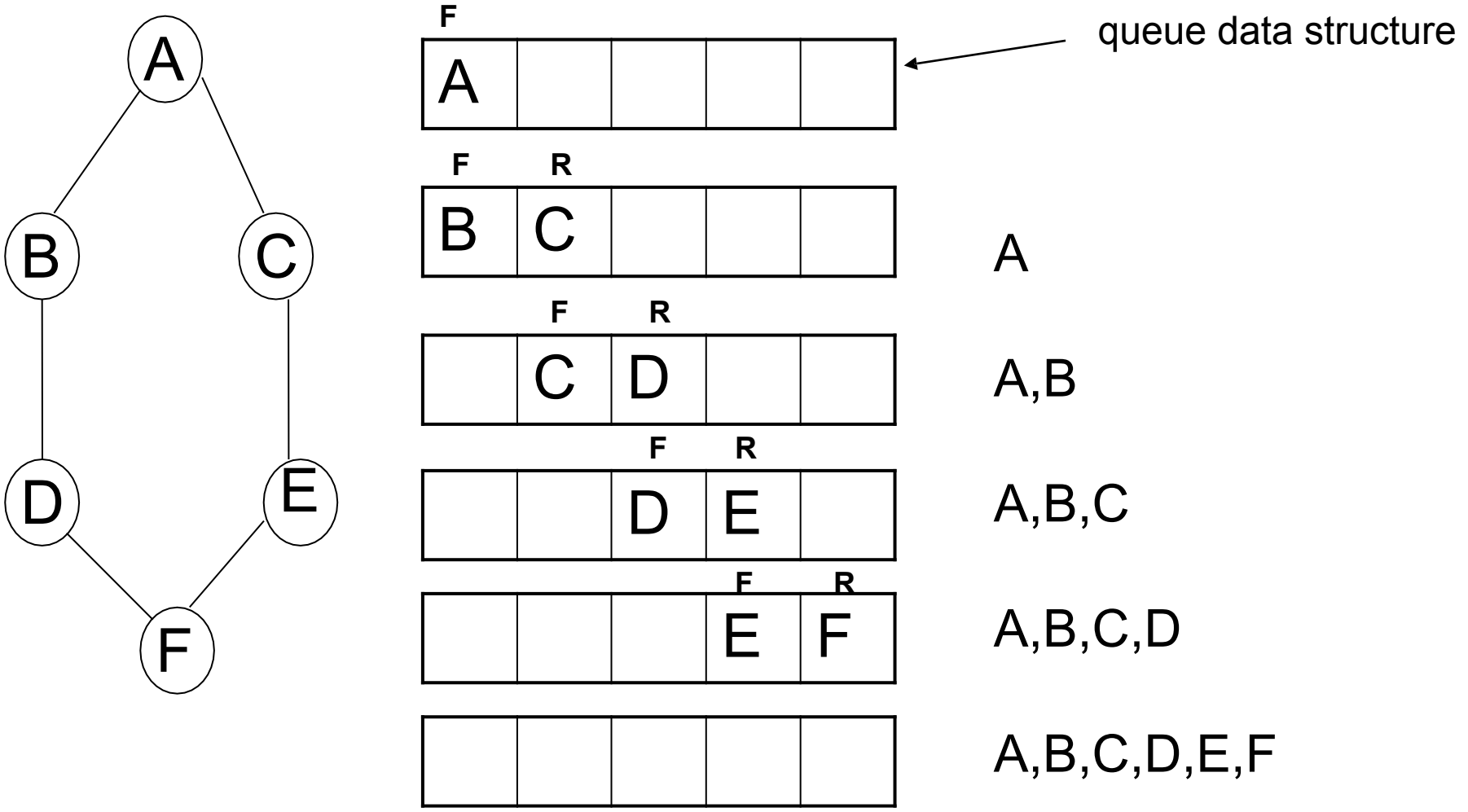
**Step 2:** Put starting node in a queue and change status to status=2.(waiting state)

**Step 3:** loop:
repeat step 4 and step 5 until queue gets empty.

**Step 4:** Remove front node N from queue, process them and change the status of N to status=3.(processed state)

**Step 5:** Add all the neighbours ofN to the rear of queue and change status to status=2.(waiting status)

# Working of BFS :



# Applications of Breadth First Traversal

1) **Shortest Path and Minimum Spanning Tree for unweighted graph** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) **Peer to Peer Networks.** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

4) **Social Networking Websites:** In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

5) **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.

# What is DFS ?

- **DFS stands for Depth First Search.**
- **DFS is an algorithm for traversing or searching a tree or graph data structures.**
- **It uses a stack data structure for implementation.**
- **In DFS one starts at the root and explores as far as possible along each branch before backtracking.**

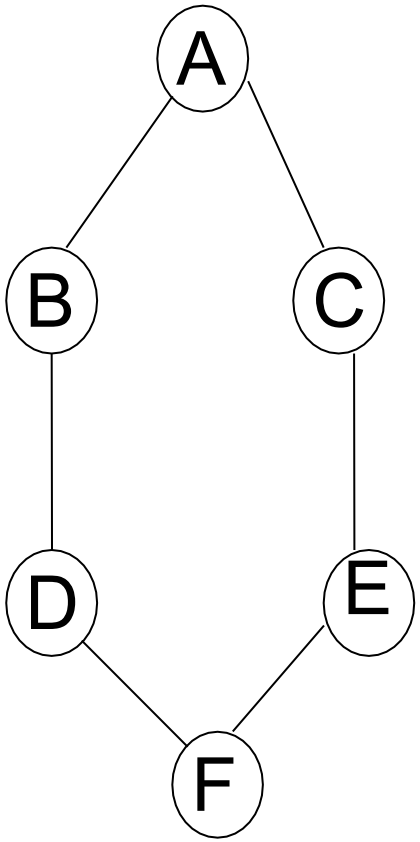
# Algorithm of DFS

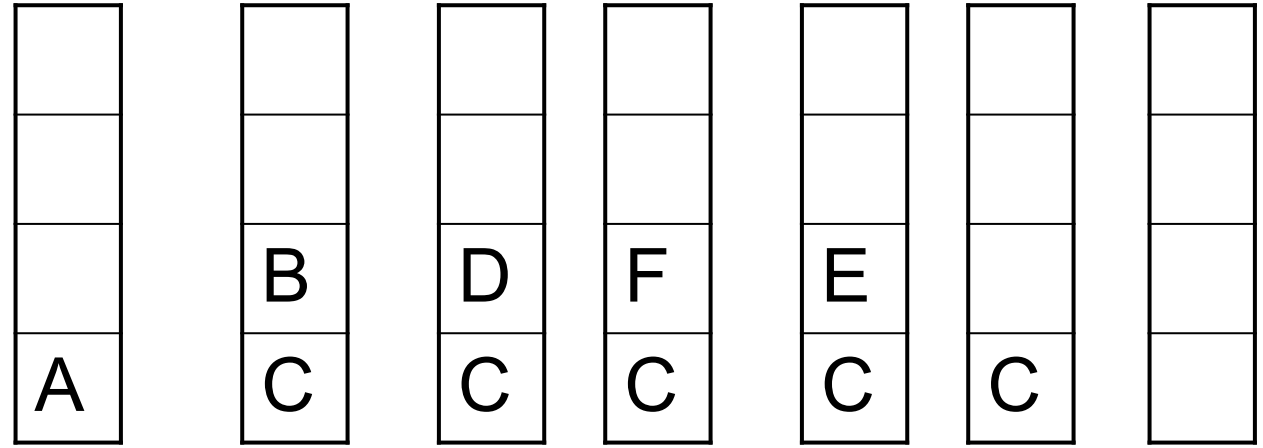
**[1]-- Initialize all nodes with status=1.(ready state) [2]– Put starting node in the stack and change status to status=2(waiting state).**

**[3]– Loop:-**
**Repeat step- 4 and step- 5 until stackGet empty. [4]– Pop or Remove top node N from stack process them and change the status of N processed state (status=3).**

**[5]– Push or Add all the neighbours of N to the top of stack and change status to waiting status-2.**

# Working of DFS :




output

A

A
B

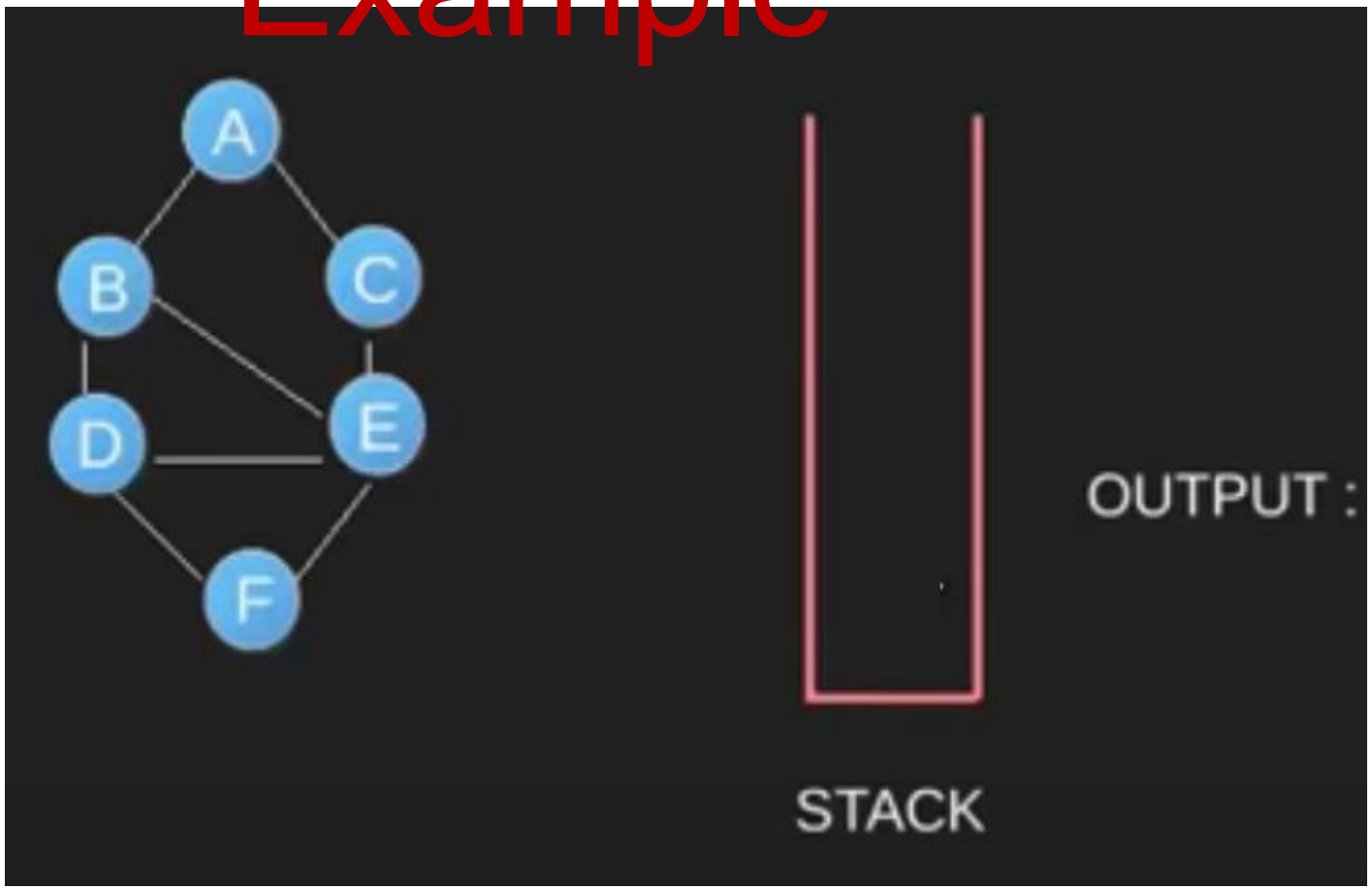
A
B
D

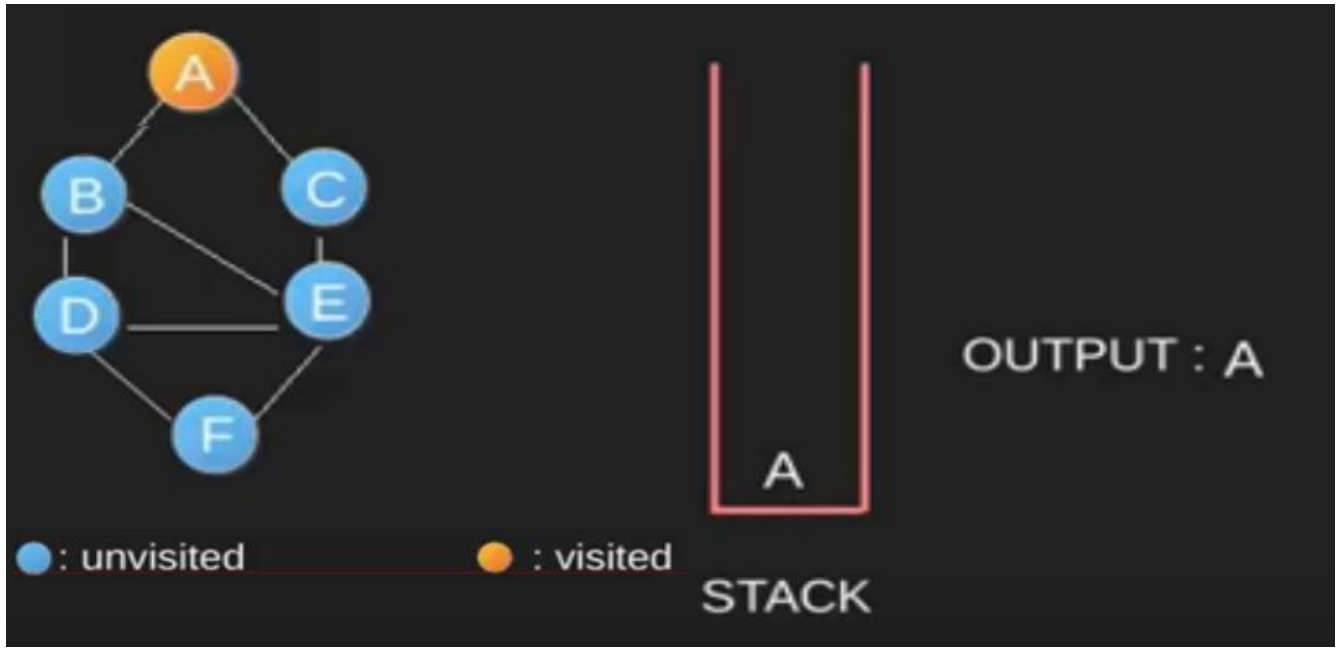
A
B
D
F

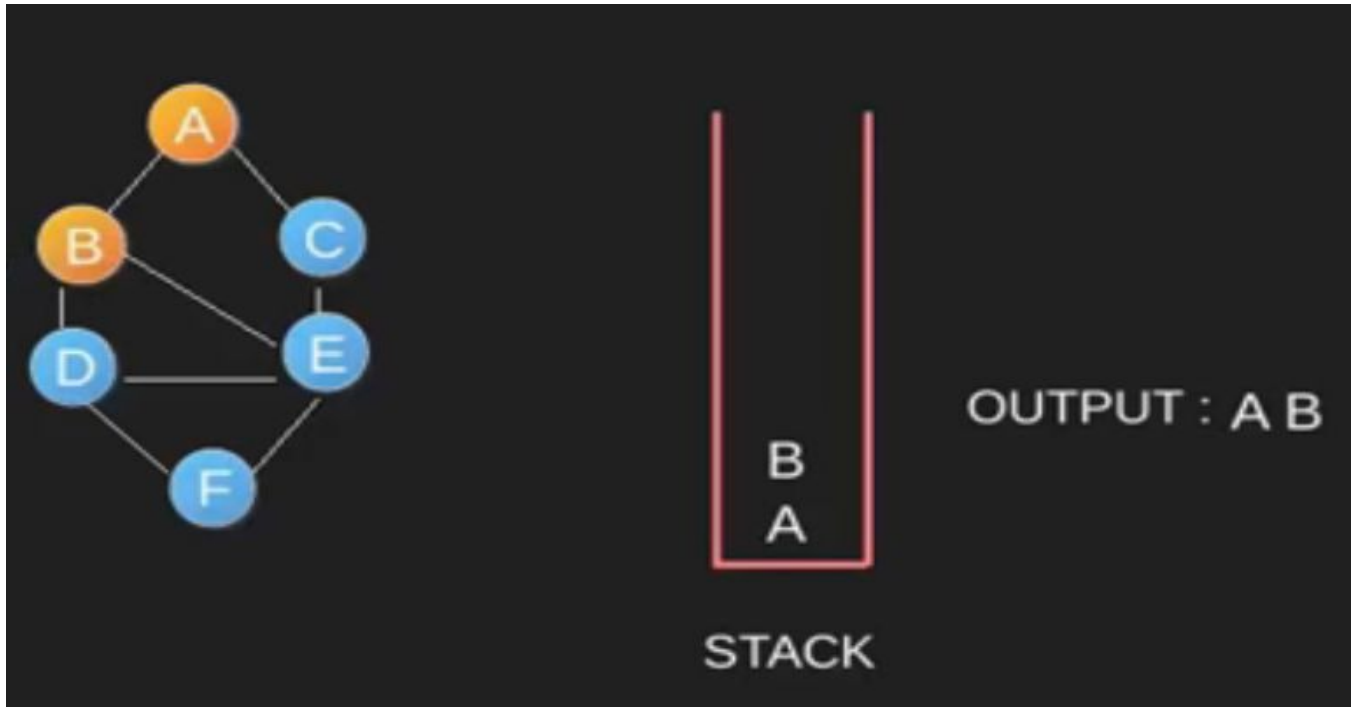
A
B
D
F
E

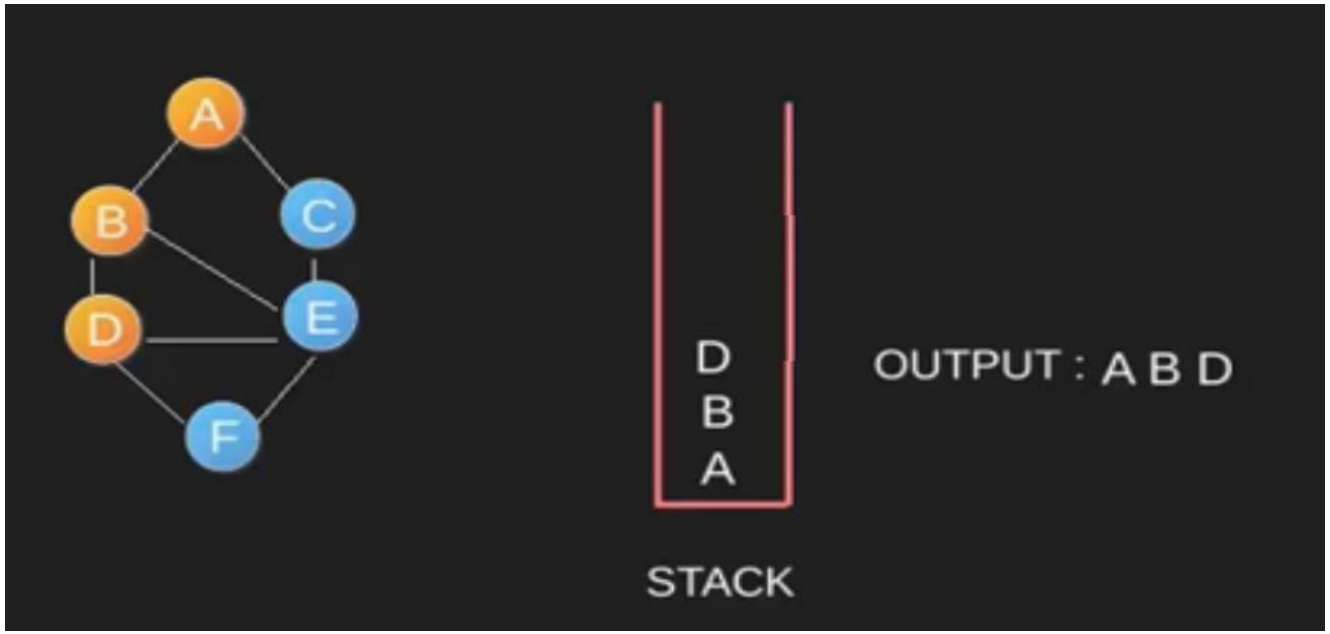
A
B
D
F
E
C

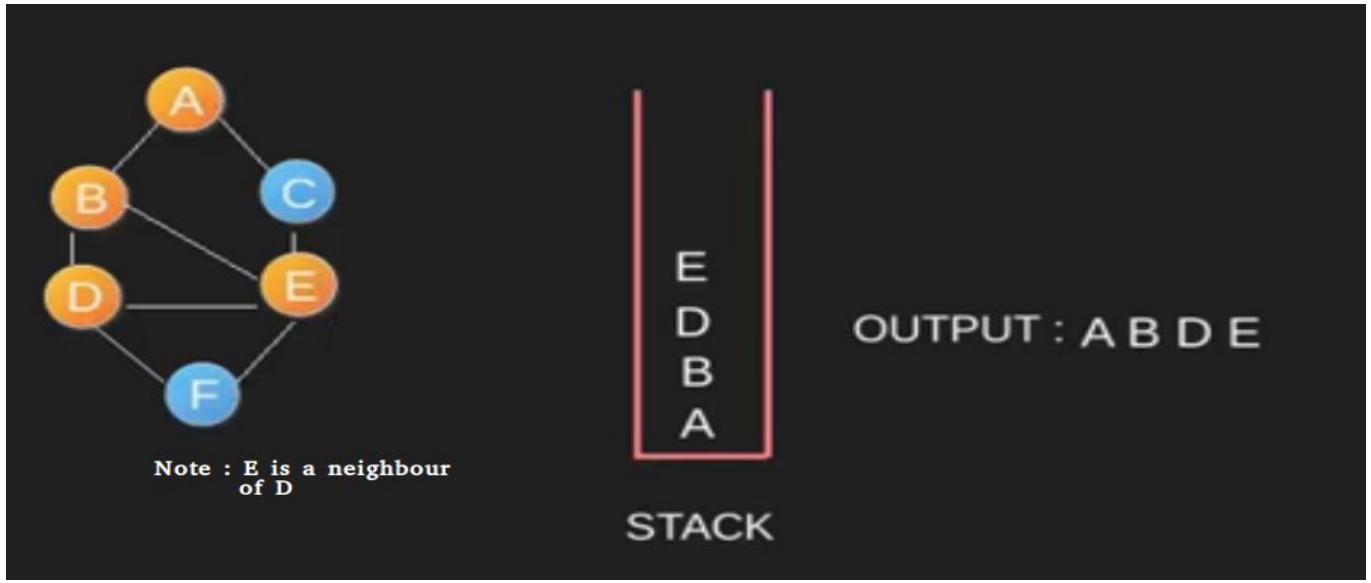
# Example

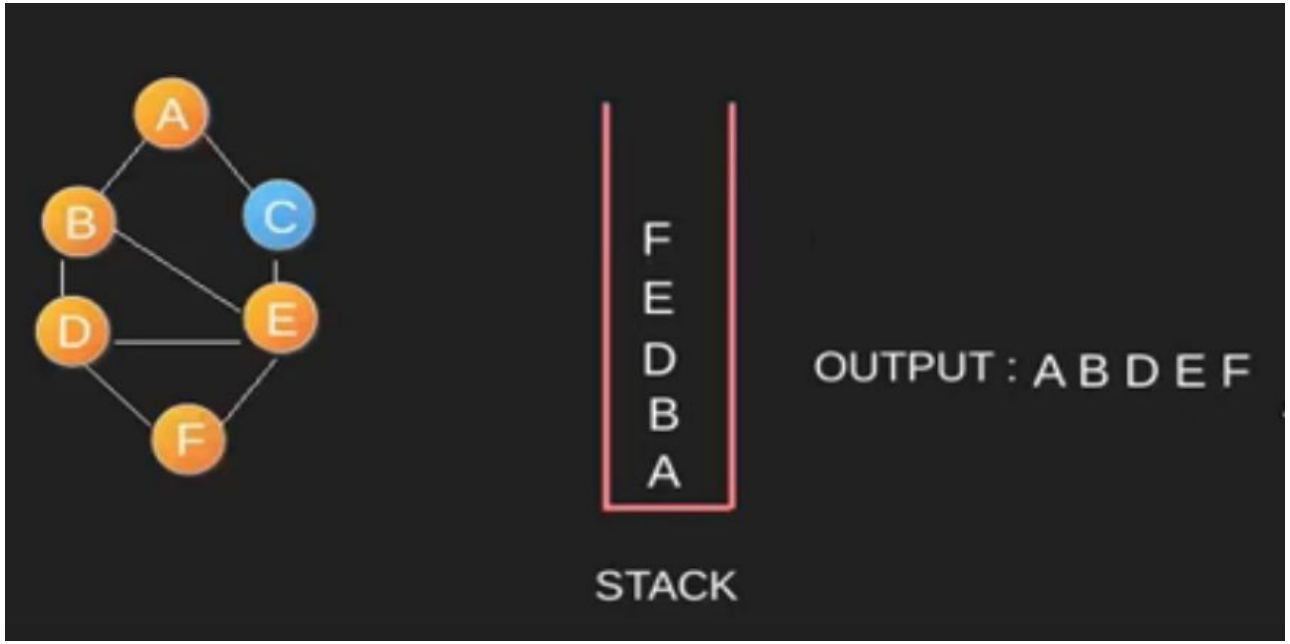


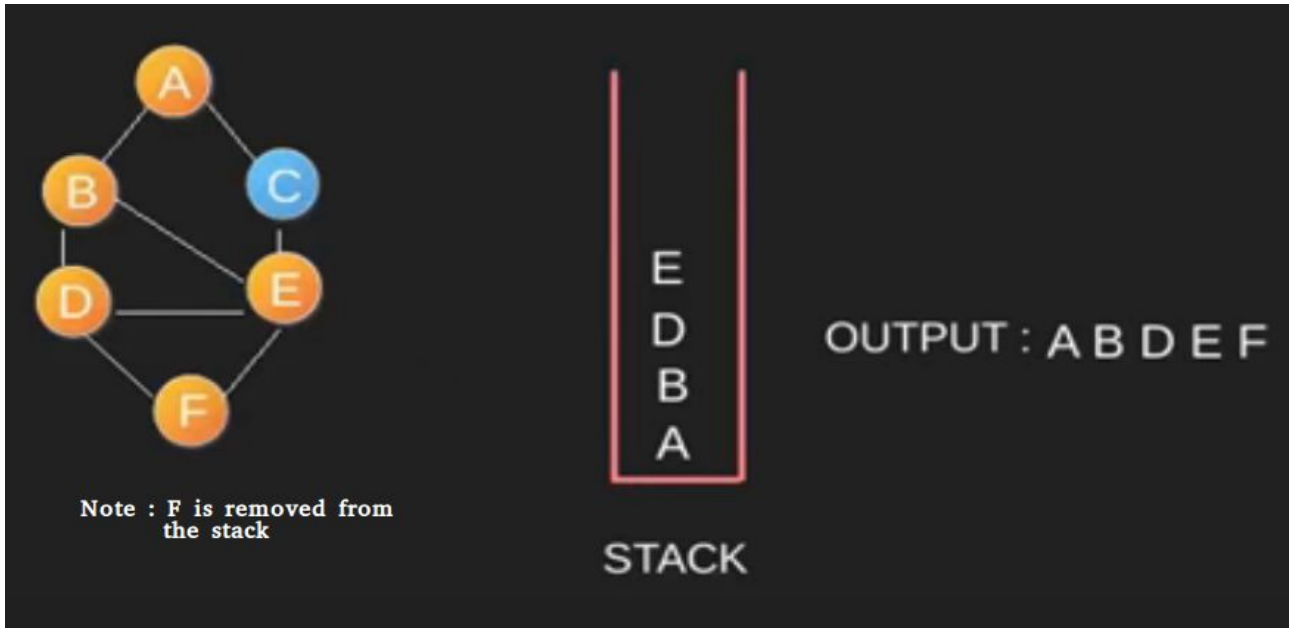

A
B
C
D
E
F
: unvisited
: visited
A
STACK
OUTPUT : A

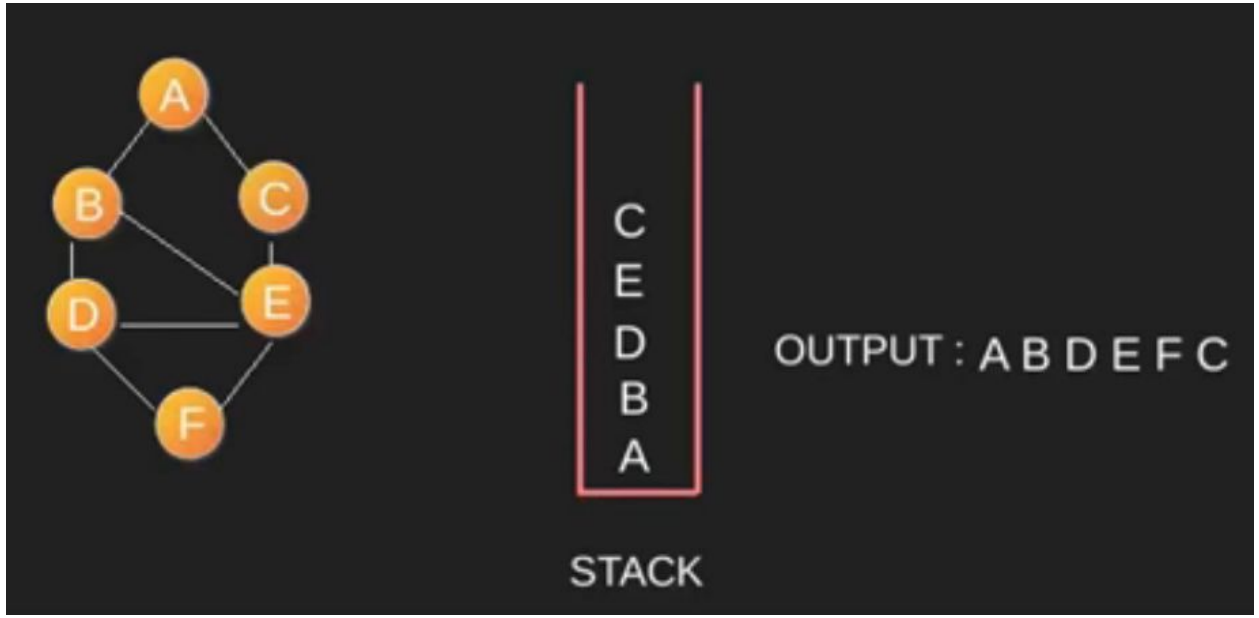

A
B
C
D
E
F
B
A
STACK
OUTPUT : A B

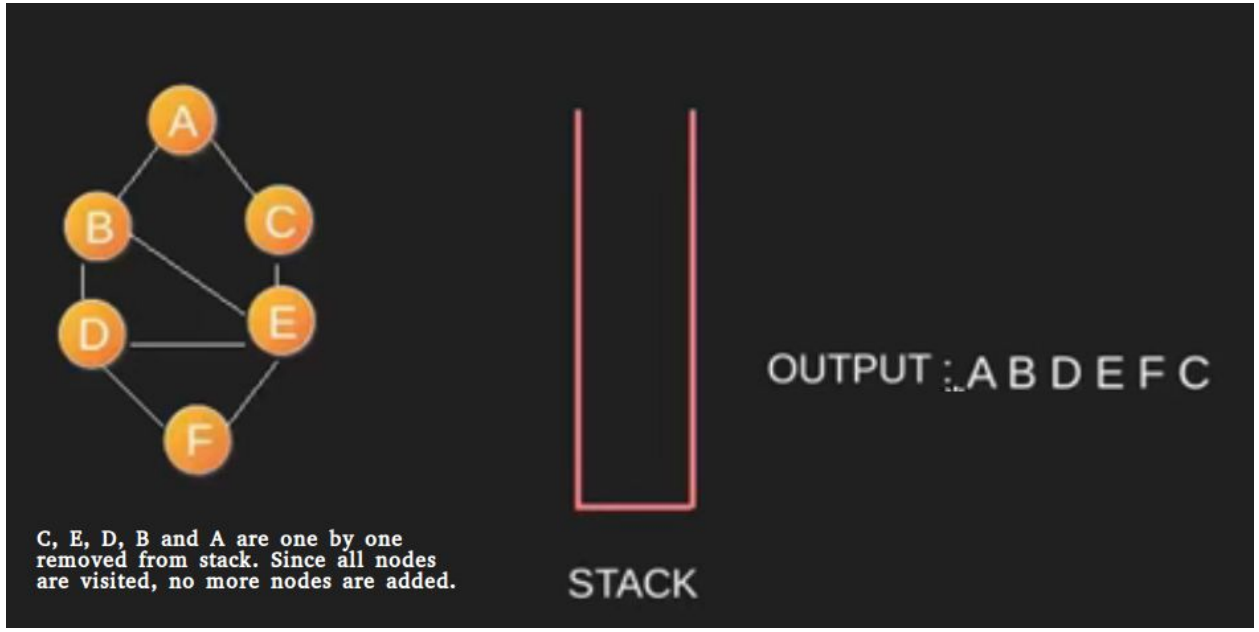

A
B
C
D
E
F
D
B
A
STACK
OUTPUT : A B D


A
B
C
D
E
F
Note : E is a neighbour of D
E
D
B
A
STACK
OUTPUT : A B D E


A
B
C
D
E
F
F
E
D
B
A
STACK
OUTPUT : A B D E F


A
B
C
D
E
F
E
D
B
A
STACK
OUTPUT : A B D E F
Note : F is removed from the stack




Final output

# Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph. Following are the problems that use DFS as a building block.

**1)** For a weighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

**2) Detecting cycle in a graph**
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

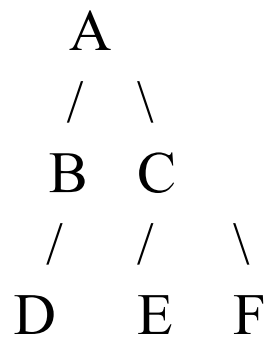
**3) Path Finding**
We can specialize the DFS algorithm to find a path between two given vertices u and z.
i) Call DFS(G, u) with u as the start vertex.
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

# difference between DFS a

| DFS | BFS |
| --- | --- |
| 1. DFS stands for Depth First Search. | 1. BFS stands for Breadth First Search. |
| 2. DFS can be done with the help of STACK i.e., LIOF. | 2. BFS can be done with the help of QUEUE i.e., FIOF. |
| 3. In DFS has higher time and space complexity, because at a time it needs to back tracing in graph for traversal. | 3. In BFS the space & time complexity is lesser as there is no need to do back tracing |

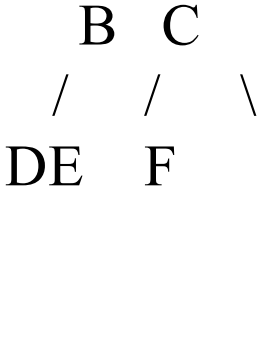
# DFS

4. DFS is more faster then BFS.
5. DFS requires less memory compare to BFS.
6. DFS is not so useful in finding shortest path.
7. Example :




# BFS

4. BFS is slower than DFS.
5. BFS requires more memory compare to DFS.
6. BFS is useful in finding shortest path.
7. Example :



# Thank you