# Some more facts about graph

# Isolated, Pendant vertices, Pseudo graph

- A vertex of degree 0 is called **isolated**, since it is not adjacent to any vertex.
- **Note:** A vertex with a **loop** at it has at least degree 2 and, by definition, is **not isolated**, even if it is not adjacent to any **other** vertex.
- A vertex of degree 1 is called **pendant**. It is adjacent to exactly one other vertex.

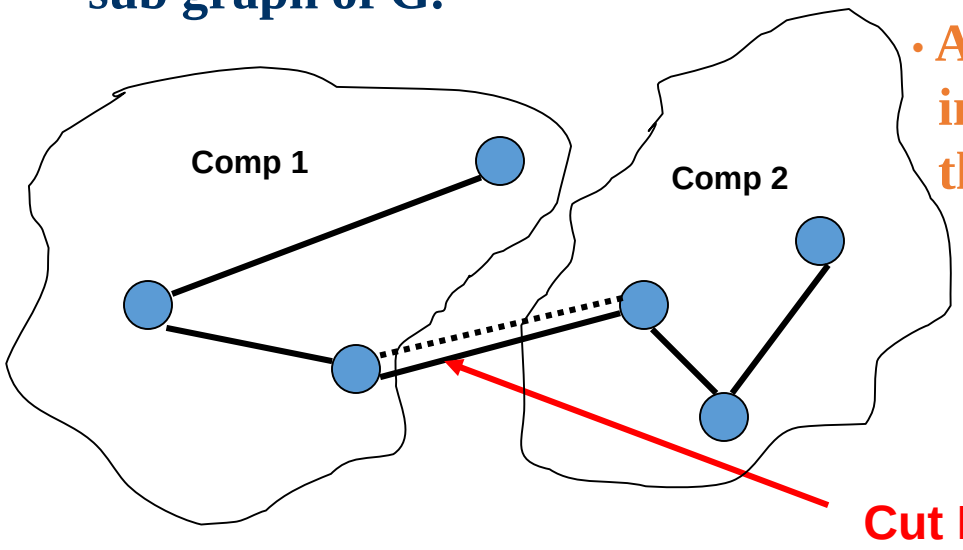
A pseudo graph is a graph G with a self-loop and numerous edges. A pseudograph is a graph in which loops (edges that connect a vertex to itself) and multiple edges (more than one edge connecting two vertices) can exist.

# Connected Graph

- Two vertices u and v are *connected* in an undirected graph iff there is a path from u to v (and v to u).

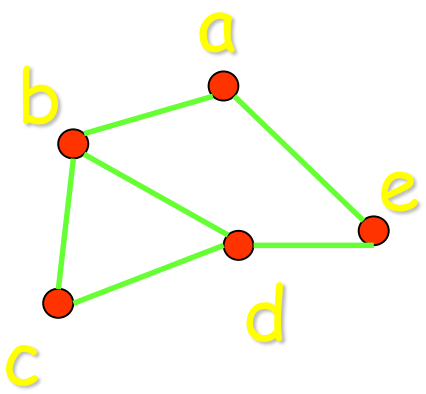
- An undirected graph is *connected* iff for every pair of distinct vertices u and v in V(G) there is a path from u to v in G.

- A *connected component* of an undirected graph G is a maximal connected sub graph of G.

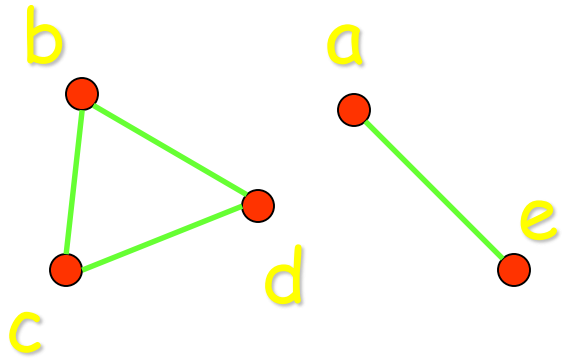



- An edge e ε E is called a *cut edge* in G – if its removal from G increases the number of connected components by at least *One*.

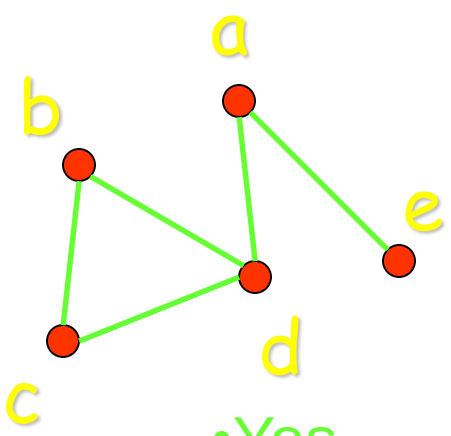
•**Example:** Are the following graphs connected?



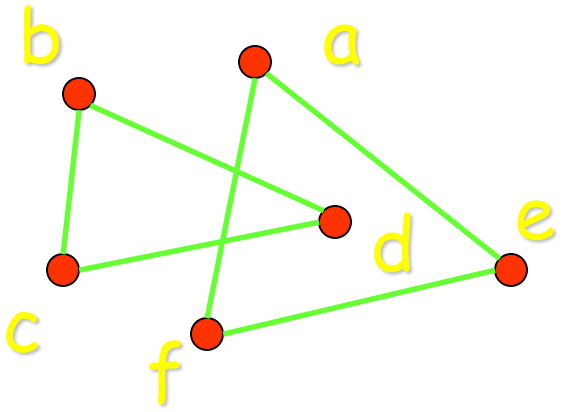

•Yes.




•No.




•Yes.

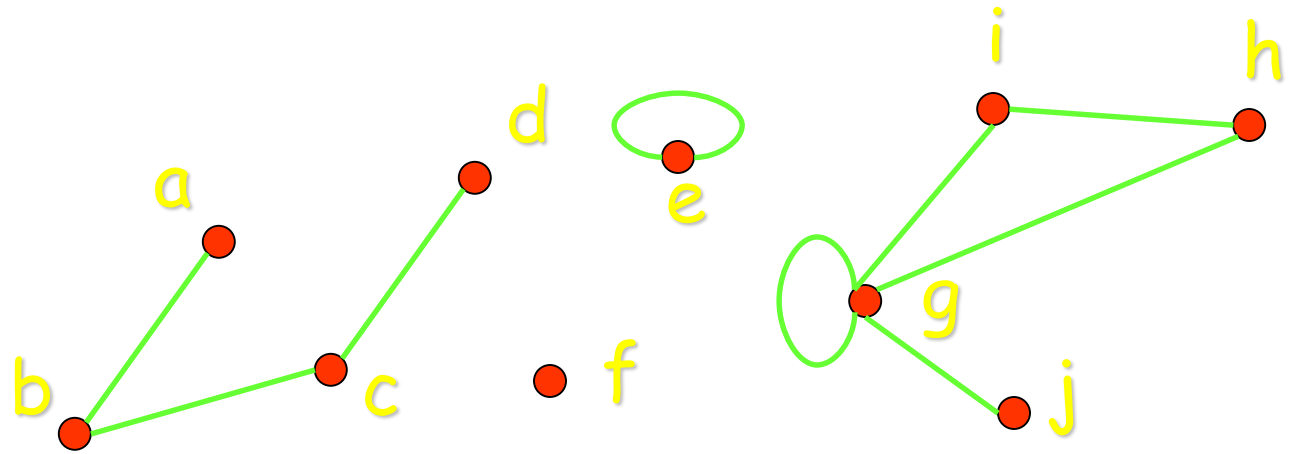



•No.

# Connectivity (Contd.)

- **Definition:** A graph that is not connected is the union of two or more connected sub graphs, each pair of which has no vertex in common. These disjoint connected sub graphs are called the **connected components** of the graph.
- **Definition:** A **connected component** of a graph G is a maximal connected sub graph of G.
- E.g., if vertex v in G belongs to a connected component, then all other vertices in G that is connected to v must also belong to that component.

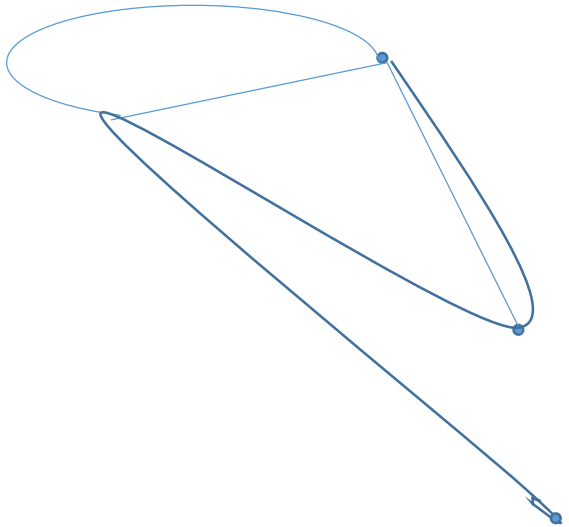
•**Example:** What are the connected components in the following graph?




•**Solution:** The connected components are the graphs with vertices {a, b, c, d}, {e}, {f}, {i, g, h, j}.

# Multigraph

- A multigraph is a graph which is permitted to have multiple edges, that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge. But no loops are allowed.

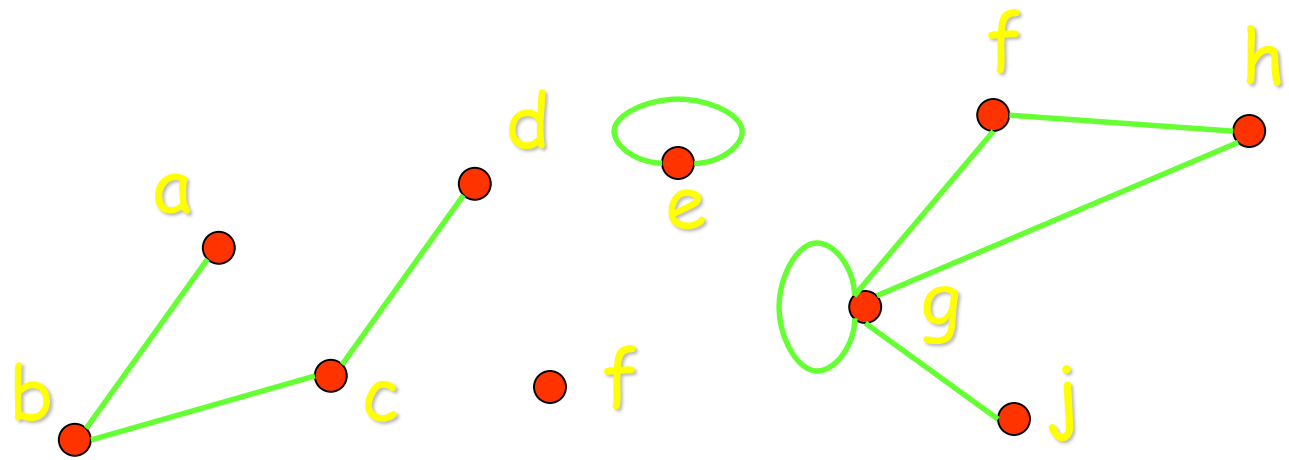


# Pseudograph

- A pseudograph is an undirected graph that allows loops (edges that connect a vertex to itself) and multiple edges (more than one edge connecting two vertices). A simple graph is a graph that does not allow for loops or multiple edges.

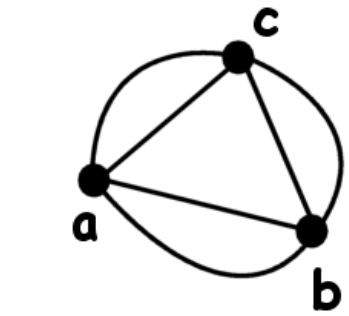
# Graph Terminology

•**Example:** Which vertices in the following graph are isolated, which are pendant, and what is the maximum degree? What type of graph is it?

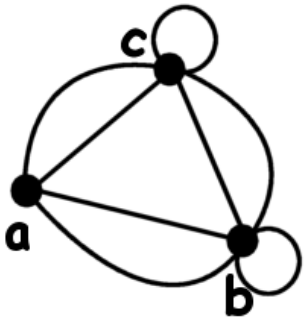



•**Solution:** Vertex f is isolated, and vertices a, d and j are pendant. The maximum degree is deg(g) = 5.
This graph is a pseudograph (undirected, loops).

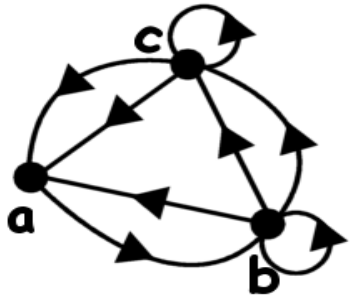
# Directed Multigraphs,Pseudographs




a)




b)




c)

| Graph | Directed | Muliple Edges Between Vertices | Loops Allowed |
|---|---|---|---|
| Simple Graph | No | No | No |
| Multigraph | No | Yes | No |
| Pseudograph | No | Yes | Yes |
| Directed Graph | Yes | Yes | No |
| Directed Multigraph | Yes | Yes | Yes |

**a) Multigraph (undirected) b) Pseudo graph c) Directed multigraph**

**Multigraph :** A multigraph is an undirected graph that can have more than one edge connecting the same pair vertices, but loops are not allowed.

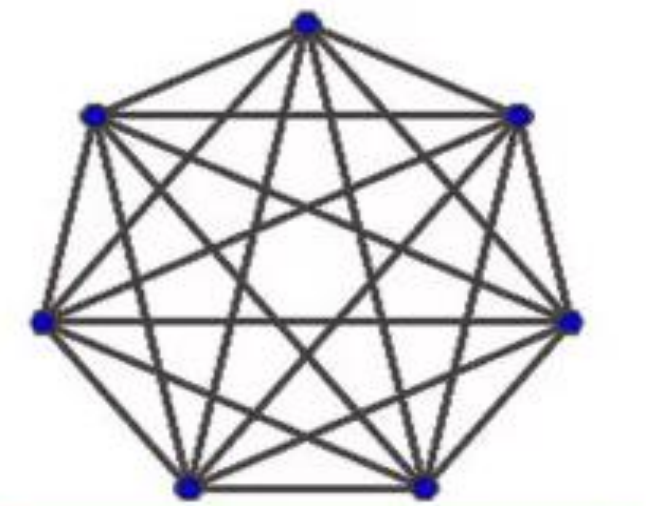
**Pseudograph :** A pseudograph is an undirected graph that can have more than one edge connecting the same pair vertices, and loops are allowed.

# Complete Graph

- A complete graph is a graph in which each vertex is connected to every other vertex OR
- A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. OR
- A complete graph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction).
- The complete graph on n vertices is denoted by $K_n$.
  $K_n$ has $n(n-1)/2$ edges and is a regular graph of degree $n-1$.
- A regular graph is a graph where each vertex has the same number of neighbors; i.e. every vertex has the same degree. A regular directed graph must also satisfy the stronger condition that the indegree and outdegree of each internal vertex are equal to each other

- A graph H is called the complement of a graph G = (V,E) if H = (V,F) Where $F = E(K_{|V|} - E)$.

# Complete Graph



**Complete Graph**

| | |
|---|---|
| vertices : | n |
| Edges : | n(n-1)/2 |

# Isomorphism of graphs

- **Definition:** The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there is a bijection (an one-to-one and onto function) f from $V_1$ to $V_2$ with the property that a and b are adjacent in $G_1$ if and only if f(a) and f(b) are adjacent in $G_2$, for all a and b in $V_1$.
- Such a function f is called an **isomorphism**.
- In other words, $G_1$ and $G_2$ are isomorphic if their vertices can be ordered in such a way that the adjacency matrices $M_{G_1}$ and $M_{G_2}$ are identical.

# Cont..

- From a visual standpoint, $G_1$ and $G_2$ are isomorphic if they can be arranged in such a way that their **displays are identical** (of course without changing adjacency).
- However, for two simple graphs, each with n vertices, there are **n! possible isomorphisms** that we have to check in order to show that these graphs are isomorphic.
- However, showing that two graphs are **not** isomorphic is easy.

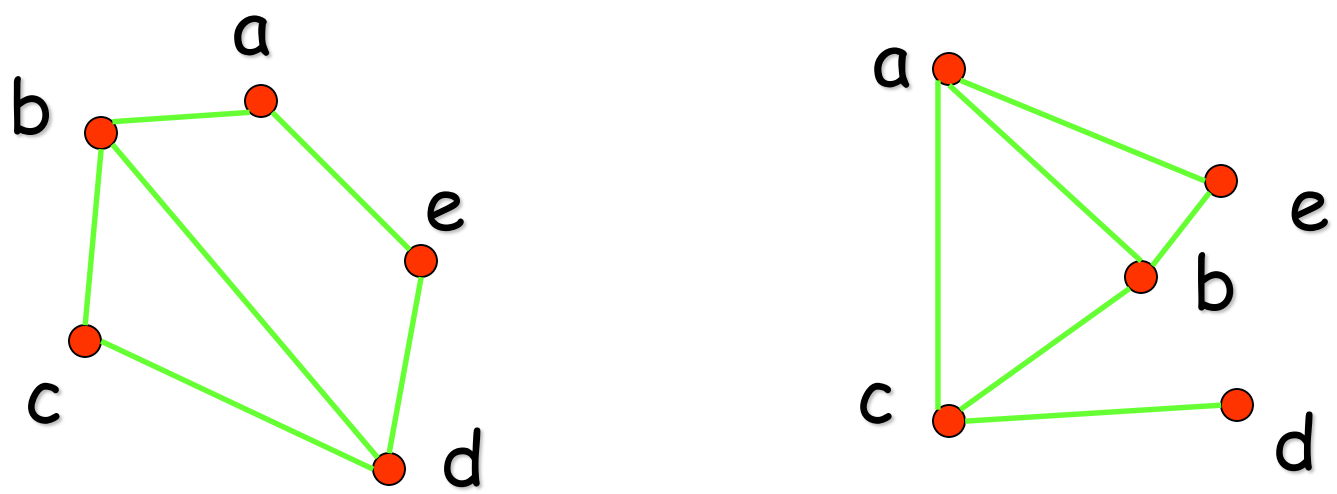
# Checking Non-isomorphism

- We can check **invariants**, that is, properties that two isomorphic simple graphs must both have.
- For example, they must have
  - **the same number of vertices,**
  - **the same number of edges, and**
  - **the same degrees of vertices.**
- Note that two graphs that **differ** in any of these invariants are not isomorphic, but two graphs that **match** in all of them are not necessarily isomorphic.

•**Example I:** Are the following two graphs isomorphic?


•**Solution:** Yes, they are isomorphic, because they can be arranged to look identical. You can see this if in the right graph you move vertex b to the left of the edge {a, c}. Then the isomorphism f from the left to the right graph is: f(a) = e, f(b) = a,
f(c) = b, f(d) = c, f(e) = d.

•**Example II:** How about these two graphs?




•**Solution:** No, they are not isomorphic, because they differ in the degrees of their vertices.

•Vertex d in right graph is of degree one, but there is no such vertex in the left graph.

# Handshaking Theorem

- The Handshaking Theorem states that in any undirected graph, the sum of the degrees of all vertices is twice the number of edges.

Let G = (V, E) be an undirected graph with e edges. Then

$\sum_{v \in V} \deg(v) = 2e$

- How many edges are there in a graph with 10 vertices, each of degree 6?

The sum of the degrees of the vertices is $6 \cdot 10 = 60$. According to the Handshaking Theorem, it follows that $2e = 60$, so there are 30 edges.

# Let us understand handshaking…

- There are n people in a party
- Some pair of people shake hands
- No multiple hand shakes between the same persons
- If x shakes y's hand then y shakes x' hand
- x does not shake x' hand

1. (nodes in a graph)
2. (edges)
3. Not a multigraph
4. undirected
5. No loop

# Another lemma..

- V1 = Set of vertices with odd degree
- V2= Set of vertices with even degree
- $2|E| = \sum_{v \in V} \deg(v) = \sum_{v \in V1} \deg(v) + \sum_{v \in V2} \deg(v)$
- Since both $2|E|$ and $\sum_{v \in V1} \deg(v)$ are even,
- $\sum_{v \in V2} \deg(v)$ must be even.
- Since $\deg(v)$ is odd for all $v \in V2$, $|V2|$ must be even.

Th: An undirected graph has an ***even number of vertices of odd degree***.

# Graph Traversal

- Traversing means visiting each node in some systematic way
- The elements of the graph to be visited are generally the nodes of graph.
- It is always possible to traverse the graph nodes efficiently in an implementation dependent manner (Adjacency matrix, Adjacency list or linked rep)
- But, we need a traversal that corresponds to the graph structure of object. This means the visiting sequence should relate to the adjacency structure of the graph.
- But defining a traversal that relates to the structure of graph is more complex than tree or any other data structure due to three reasons:

# Problems with traversal

- There is no ‘root node’ or first node of graph
- After deciding the first node, only those nodes which are reachable from the starting node can be traversed
- There may remain other nodes in the graph that have not been visited because those may not be connected
- This means we may have to select another starting node
- There is no natural order among successors of a particular node.

# Cont..

- Thus there is no apriori order in which the successors of a particular node should be visited.
- A node of a graph may have more than one predecessor and it is possible for a node to be visited before one of its predecessor
- In a tree, we never encounter a node more than once but while traversing a graph, there may be a possibility that we reach a node more than once. So, we have to keep status of each node whether visited or not
- Unlike tree, in graph, for the same traversal technique, we may have more than one traversal

# Breadth First traversal or search

- BFT or BFS is a technique that begins at the root (user defined start node) and explores all the neighboring or adjacent nodes or successor nodes. Then for each of those nodes, the algorithm explores all the unexplored nodes of these nodes until all the node are visited or goal node is found.
- This method visits all successors of a visited node before visiting any successors of any of those successors.

# BFS algorithm

A Boolean array is maintained for visited nodes

1. Choose the starting node
2. Visit/take all the nodes adjacent to that starting node
3. Continue the same, for all other adjacent nodes which are adjacent to starting node
4. Maintain the status of vising nodes
5. Suppose V0 is the starting node and V11, V12, V13 are the adjacent nodes of V0….

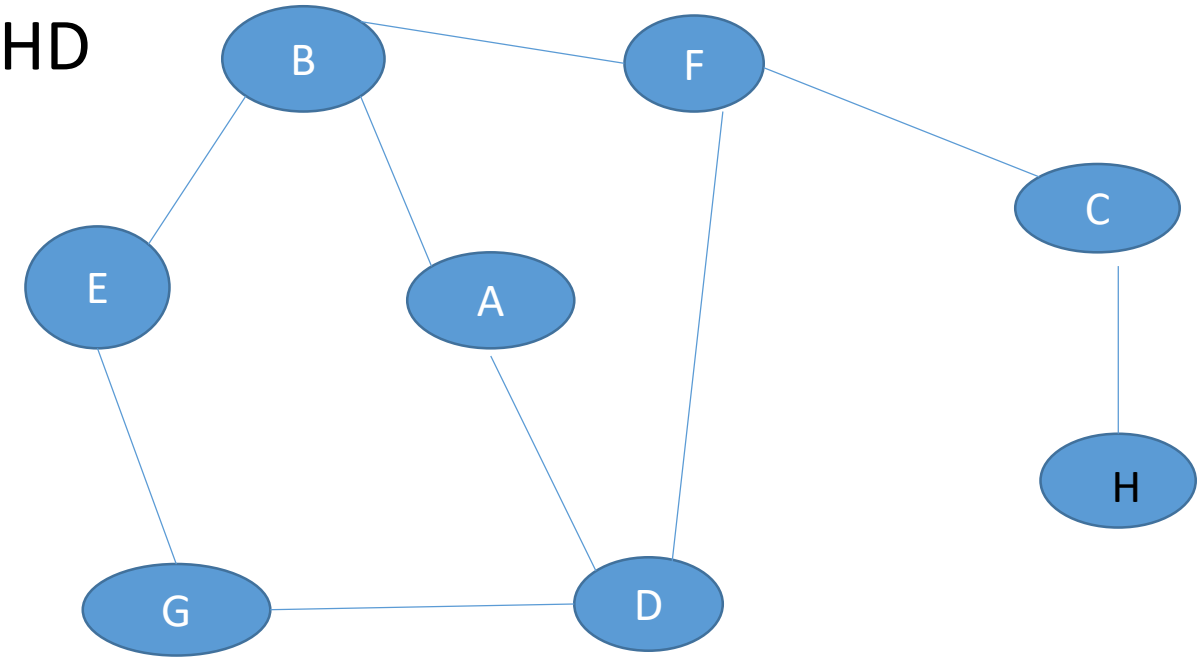
# BFS through Queue

1. Insert starting node in the queue
2. Delete an element from queue and insert all the successors of starting node in the queue

- Repeat step 2 until queue is empty

# Example

- BFS: ABDGEFCH
- DFS: ABEGFCHD



# BFS

- Start with A
- Mark it visited and put the node A in resultant string R (R: A)
- Put all the adjacent nodes of A into a queue
- Now delete a node from queue, mark it visited and add to R(say B)
- Put all the adjacent node of B into queue
- Continue with this process till all the nodes are visited or

# Depth First traversal or search

- The depth first traversal or search progresses by expanding the starting node of G and then going deeper and deeper until goal node is found or until a node is encountered which has no successor
- DFS begins with a starting node A, then it examines nodes along a path P which begins at A. This means first neighbor of A is processed then a neighbor of neighbor of A is processed and so on.
- The algorithm proceeds like this until we reach dead-end.
- On reaching the dead end, we backtrack to find another path P’
- A stack is used to implement depth first traversal

# DFS algorithm

A Boolean array is maintained for visited nodes

1. Choose the starting node
2. Visit one of the adjacent node to that starting node
3. Continue the same, for all other adjacent node which are adjacent to this node
4. Keep moving till a dead-end is reached
5. Suppose V0 is the starting node and V11, V12, V13 are the adjacent nodes of V0….

# DFS

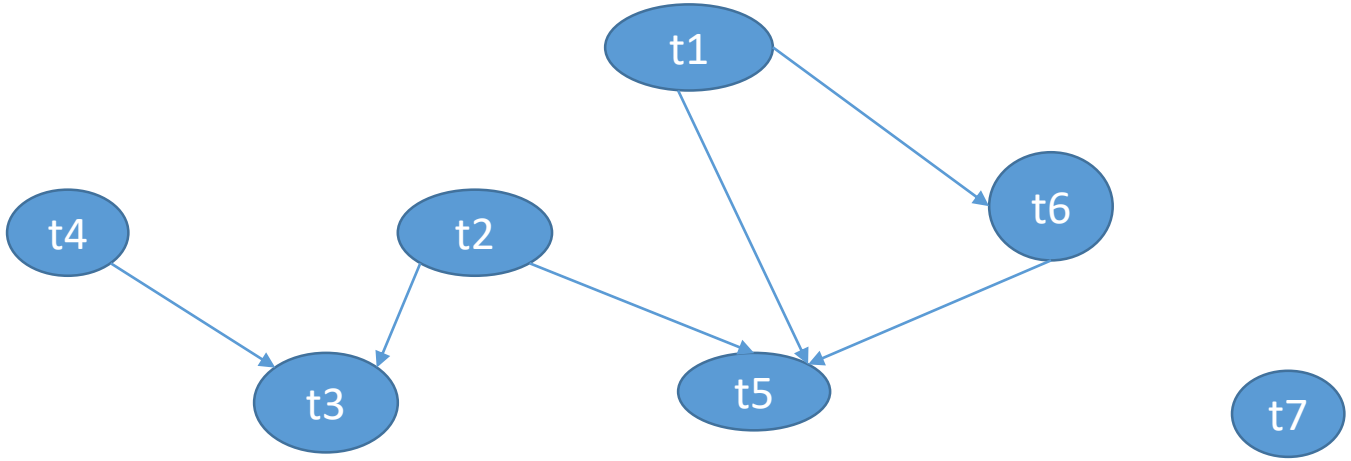
- Start with starting node A
- Put it in the stack, Mark it visited and put in the resultant string R
- Put a node adjacent to this node in stack
- Mark it visited and put in the resultant string R
- Then put adjacent node of this new node, if all the nodes already visited then pop it
- Continue with the same process till the stack is empty

# Topological sorting

- How to solve the problem of course registration if pre-requisites of course have to be followed strictly
- how to schedule jobs if 7 jobs have to be done to complete a task and there is dependency among these jobs

# Topological Sorting

- Let this graph shows the dependencies…



# Topological Sorting

- If G is a directed graph with no directed cycles i.e. graph is directed and acyclic then topological ordering T of G is a sequential listing of all vertices such that for all u,v belonging to G, u precedes v in topological ordering.
- Thus topological ordering is a linear ordering of nodes of G such that if(u,v) is a edge then u always appears before v.

# Algorithm to find Topological Sorting

- Find the in-degree of each vertex v in G
- Insert all vertices with zero in-degree in a queue
- Repeat 4 and 5 till queue is empty
- Remove front vertex from queue
- For each neighbor w of vertex v do:
- (a) decrease the in-degree of w by 1 and remove edge from v to w
- (b) If in-degree of w is zero, then add w to the rear of queue