SLIIT ACADEMY

BSc (IT) – Year 2, Semester 2

# Design and Analysis of Algorithms
# Graphs Algorithm
# Lecture 07

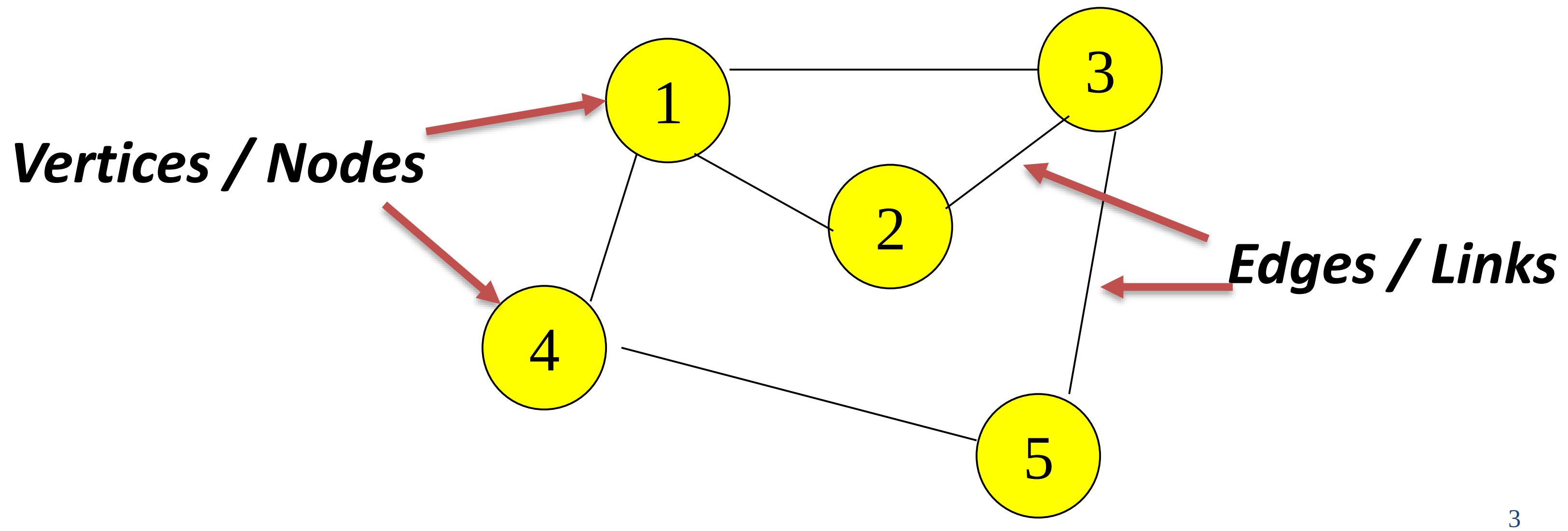
Anuruddha Abeysinghe
anuruddha.a@sliit.lk

## Today's Lecture

- Graph
- Applications.
- Terminology.
- Representation.
- Searching
    - DFS
    - BFS

# Graph Introduction

- Graph is a data structure which consists of **set of vertices** and **set of edges.**
- Graphs are a pervasive data structure in computer science, and algorithms for working with them are fundamental to the field. There are hundreds of interesting computational problems defined in terms of graphs.

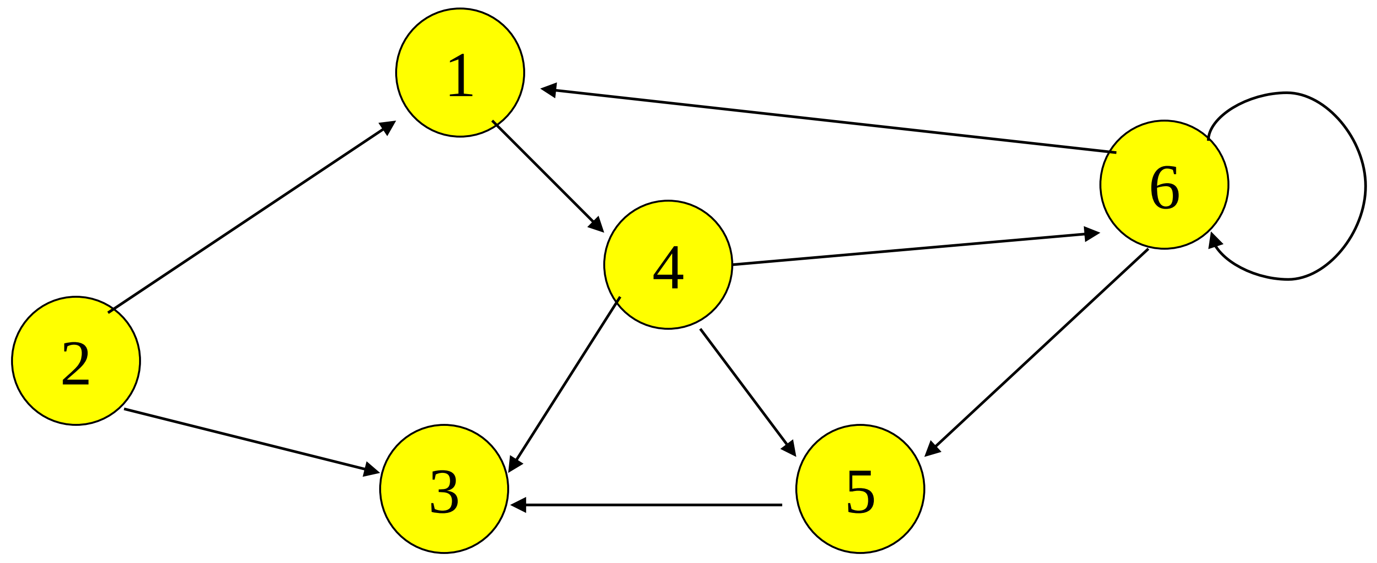


## Applications

- A network of roads : with cities as vertices and roads between cities as edges.
- An electronic circuit : with junctions as vertices and components as edges.
- Flights between cities : with cities as the vertices and flight from one city to another as edges.

## Graphs categorization

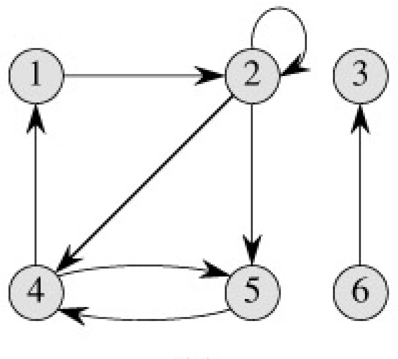
- ***<u>Directed Graphs (Digraph)</u>* :**
  - All the edges have direction.
  - Self loops are possible.



## Graph Terminology

- **Loop or self – edges:**

  An edge ( i,i ) is called a ***self edge*** or a ***loop***. Note that self-loops-edges from a vertex to itself-are possible.

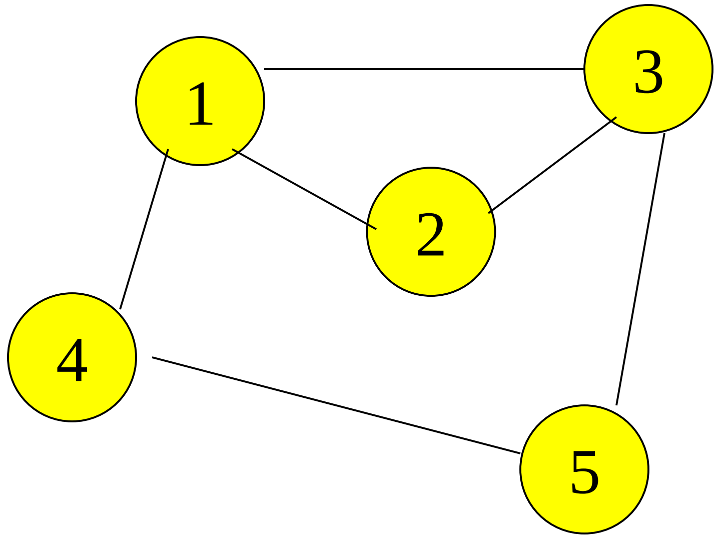


(a)

# Graphs categorization

- ***Undirected Graphs:***

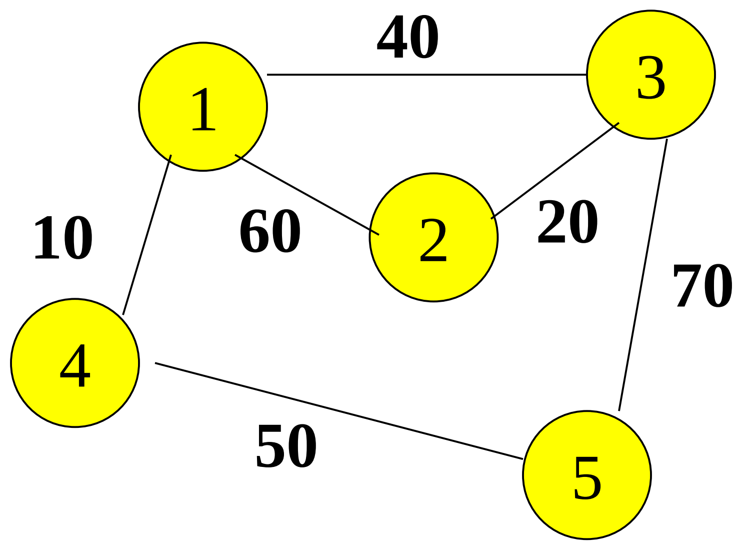
  Edges are not directed.



## Graphs categorization

- ***Weighted Graphs:***

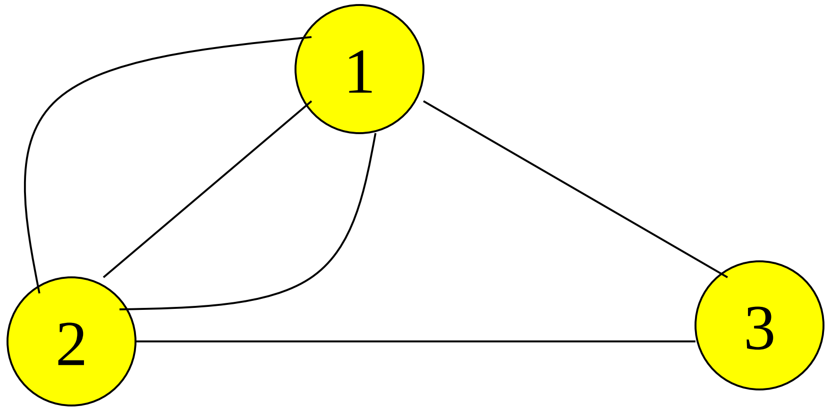
All the edges have been assigned a weight.



## Graphs categorization

- ***Multigraphs:***

  If the same pair of vertices have more than one edge.



## Graph Terminology

- **G = ( V, E )**

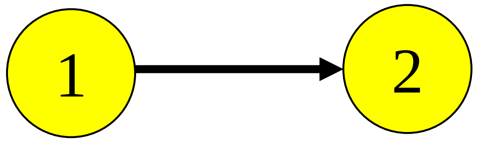
  where V is set of vertices and E is set of  edges.

➢ Edges in a directed graph are ordered pairs.

➢ Edges in a undirected graph are unordered pairs.

## Graph Terminology

- When the graph is a directed one the edge (i,j) is different from (j,i).



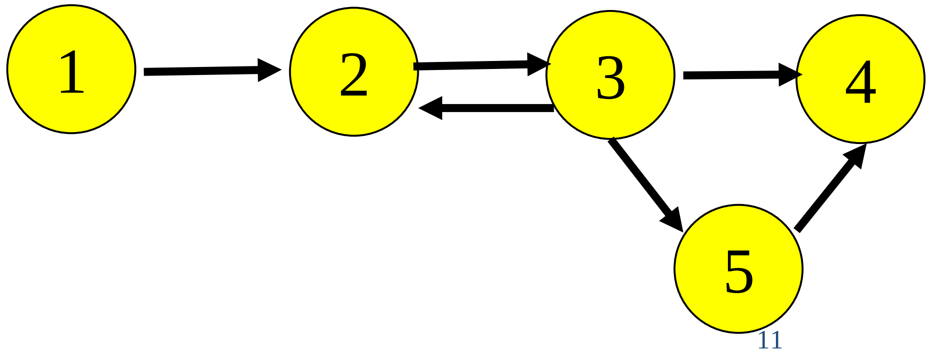
(i,j) Orientation is i to j.

Therefore here edge is (1,2).

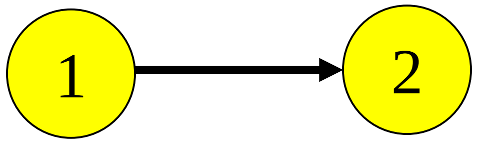
Exercise : Draw the **Directed graph**

if V={1,2,3,4,5} and

E={(1,2),(2,3),(3,4),(3,5),(5,4),(3,2)}



## Graph Terminology



Vertex 1 is **adjacent to** vertex 2.

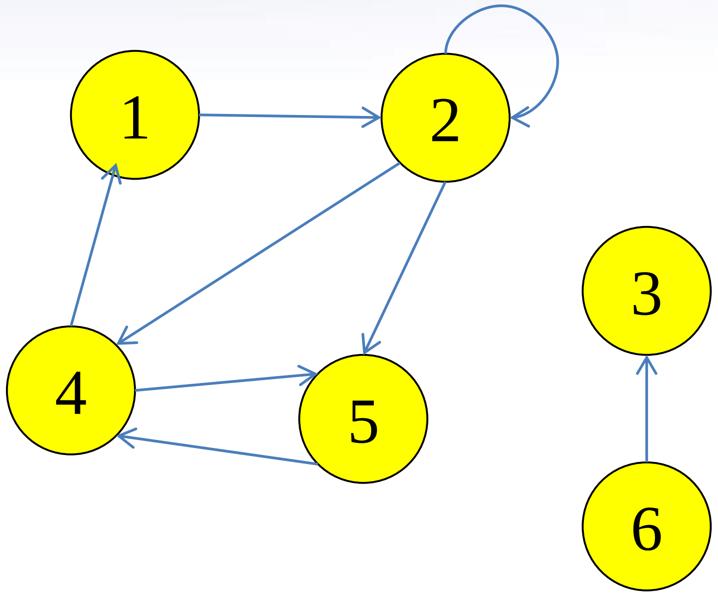
Vertex 2 is **adjacent from** vertex 1.

Vertex 1 is  **incident to** vertex 2.

Vertex 2 is **incident from** vertex 1.

## Graph Terminology

- **Directed Graph**



G=( V,E )

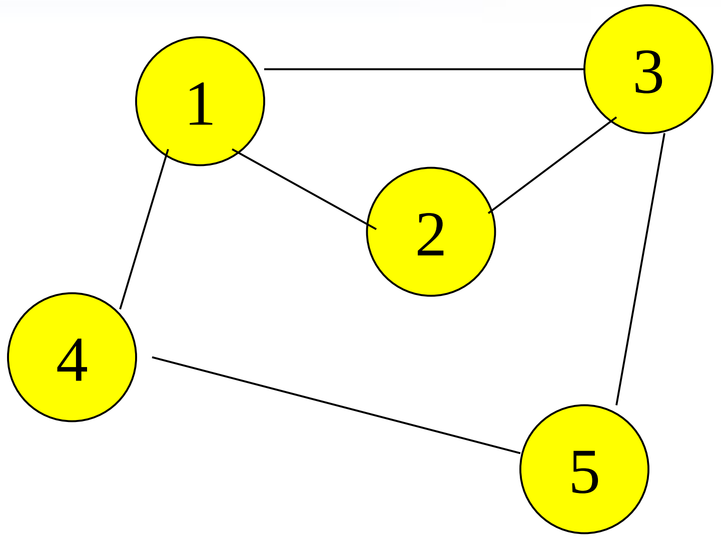
V={1,2,3,4,5,6}

E={(1,2), (2,2),(2,4),(2,5),(4,1) ,(4,5) ,(5,4),(6,3)}

- Edges ***leaving vertex 2*** or ***incident from vertex 2*** are (2,2),(2,4) and (2,5).
- Edges ***enters vertex 2*** or ***incident to vertex 2*** are (1,2) and (2,2).

# Graph Terminology

- **Undirected Graph**



G=( V,E )

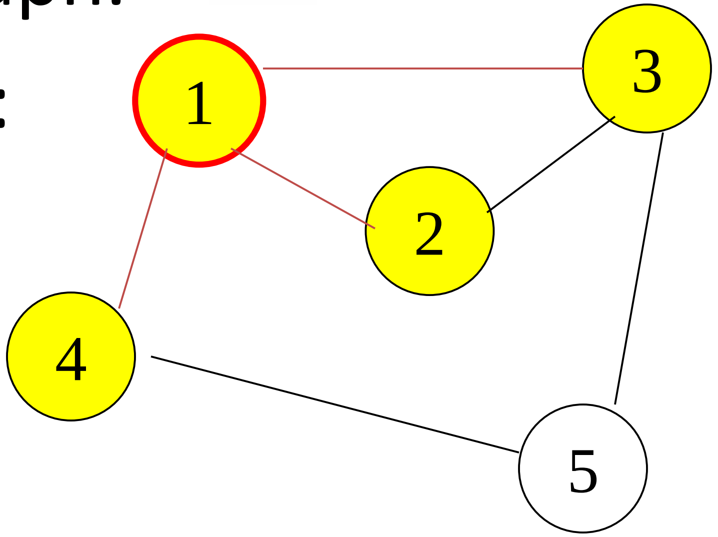
V={1,2,3,4,5}

E={(1,3),(1,2),(1.4),(4,5),(2,3),(3,5)}

- Edges ***incident on vertex 2*** are (1,2) and (2,3).

## Graph Terminology

- **Adjacent vertices**: If (i,j) is an edge of the graph.
- Eg:



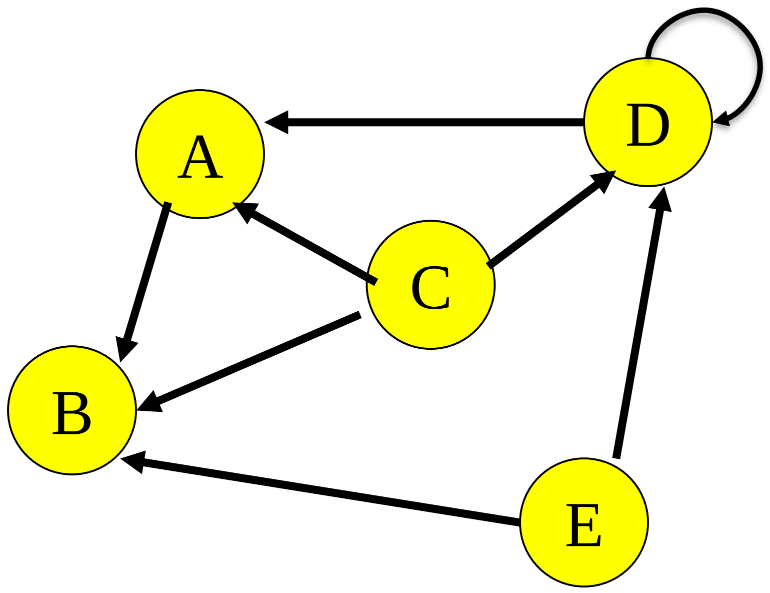
In Considering vertice 1, vertice **5** is not a adjacent since there is no edge between 1 and 5.

Others are adjacent vertices.

# Graph Terminology

The degree *d*(*v*) of a vertex *v* is the number of edges incident to *v*.

- In directed graphs, **in-degree** is the number of incoming edges at the vertex and **out-degree** is the number of outgoing edges from the vertex.
- In undirected graphs, degree is the number of edges incident on it.



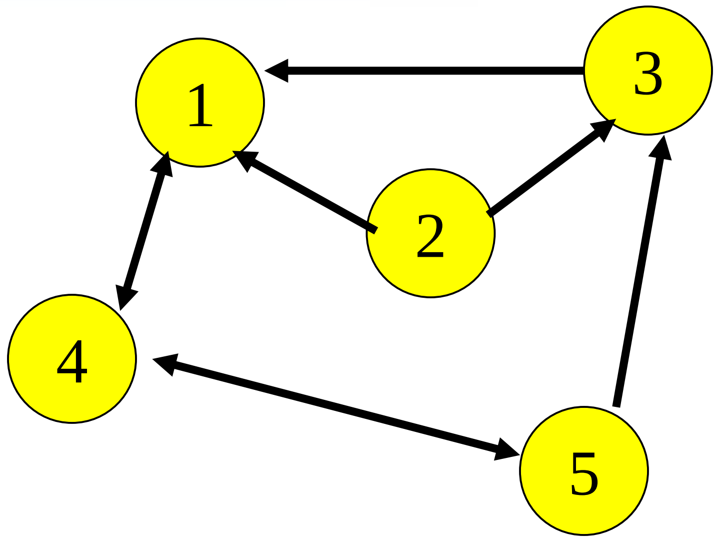
***Activity:***
***Fill the following table.***

| | In Degree | Out Degree |
|---|---|---|
| A | | |
| B | | |
| C | | |
| D | | |
| E | | |

**Example: The indegree of A is 2, its outdegree is 1.**

# Graph Terminology

- Simple Path: <u>If all vertices in the path are distinct</u>.

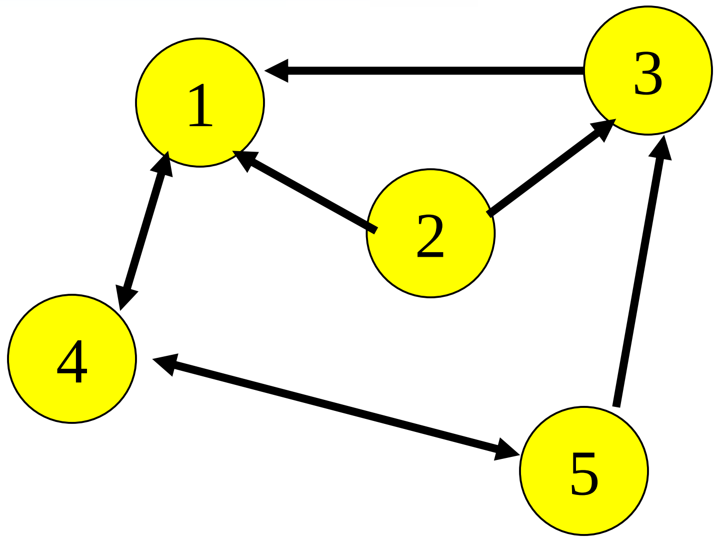


1,4,5,3 is a simple path.

But 1,4,5,4 is not a simple path.

## Graph Terminology

- Length : sum of the lengths of the edges on the path.



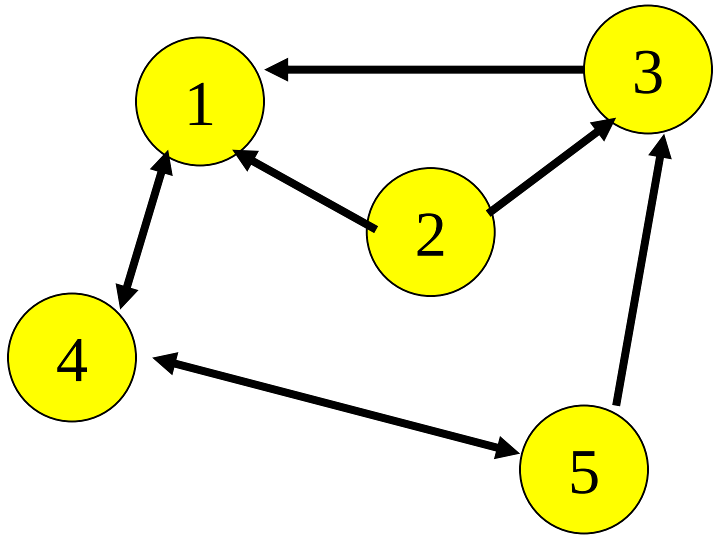
For the path 1,4,5,3

The length is 3.

# Graph Terminology

**If there is a path from A to B, Vertex B is said to be reachable from A**

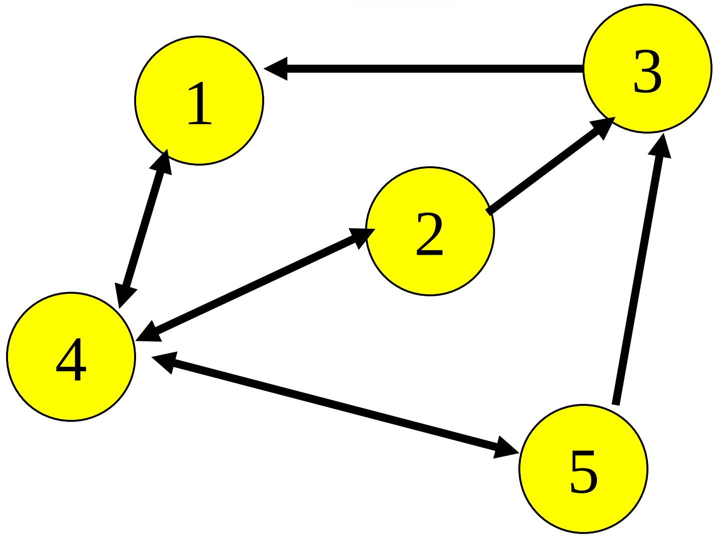
**A *circuit* is a path whose first and last vertices are the same.**



The path 3,1,4,5,3 is a circuit.

## Graph Terminology

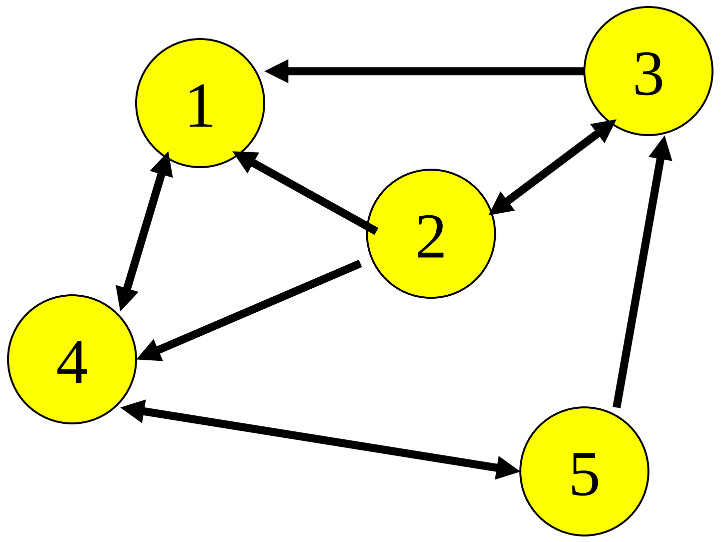
**A simple circuit is a <u>cycle</u> if except for the first (and last) vertex, no other vertex appears more than once.**



The path **3**,1,4,5,**3** is a cycle but **3**,1,4,2,4,5,**3** is not a cycle.

## Graph Terminology

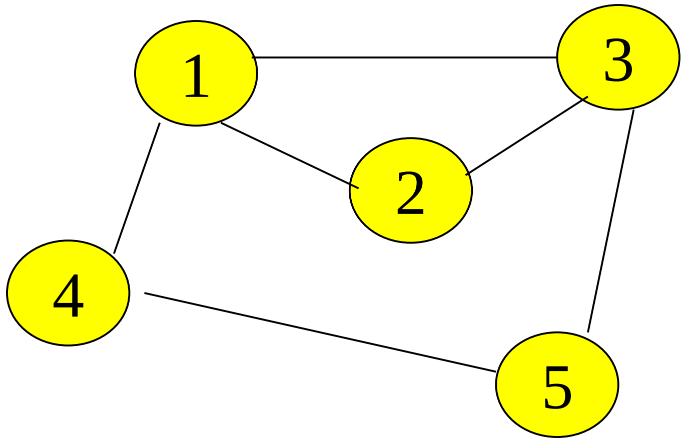
A Hamiltonian cycle of a graph G is a cycle that contains all the vertices of G.

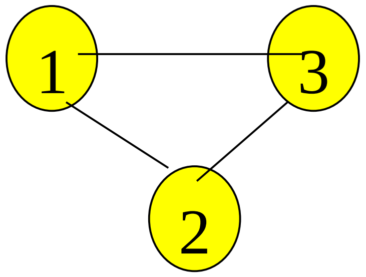


3,2,1,4,5,3 is a Hamiltonian cycle.

## Graph Terminology

**A subgraph of a graph G = (V,E) is a graph H(U,F) such that U ⊆ V and F⊆ E.**





G=( V,E )

V={1,2,3,4,5}

E={(1,3),(1,2),(1.4),(4,5),(2,3),(3,5)}

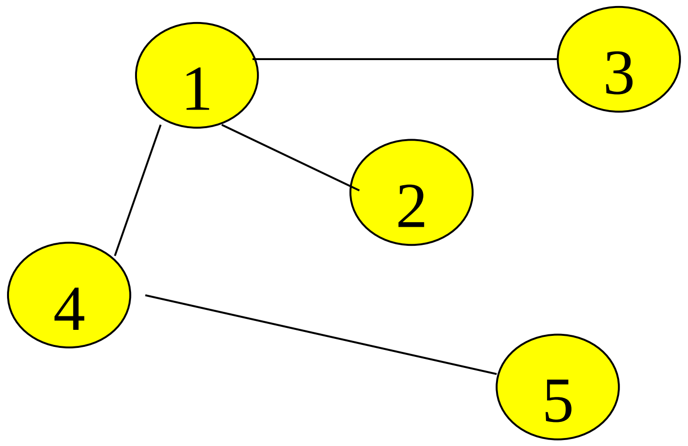
H=( U,F )

U={1,2,3}

F={1,3),(1,2),(2,3)}

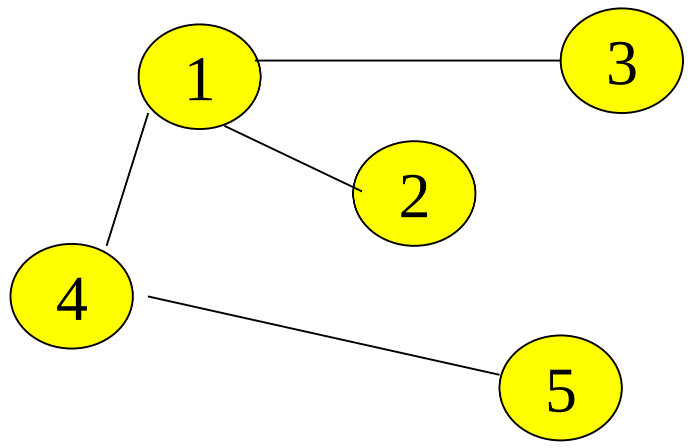
## Graph Terminology

**A graph is said to be <u>connected</u> if there is at least one path from every vertex to every other vertex in the graph.**



# Graph Terminology

- Tree : A connected undirected graph that contains no cycles is called a tree.
- Forest : A graph it does not contain  a cycle is called a forest.
- *G = (V, E)* is undirected graph, then
  - is a tree
  - has no cycle, and connected
  - has *|V|-1* edges, and connected



# Representation of Graphs.

- There are two standard ways to represent a graph $G = (V, E)$: as a collection of **adjacency lists** or as an **adjacency matrix**. Either way is applicable to both directed and undirected graphs.

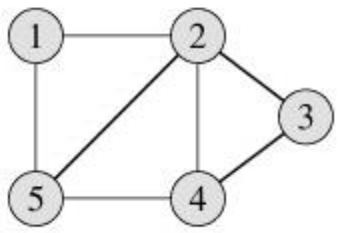
- The ***adjacency-list*** representation is usually preferred, because it provides a compact way to represent ***sparse*** graphs-those for which $|E|$ is much less than $|V|^2$.

Sparse Graph → **$|E| < |V|^2$**

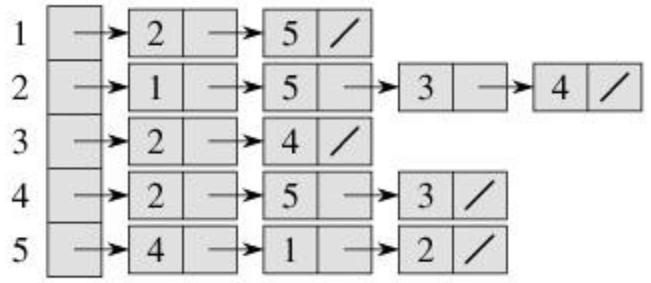
- An ***adjacency-matrix*** representation may be preferred, however, when the graph is ***dense***-$|E|$ is close to $|V|^2$.

Dense Graph → **$|E| \approx |V|^2$**

# Representation of Graphs.



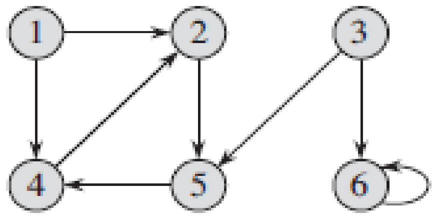
(a)



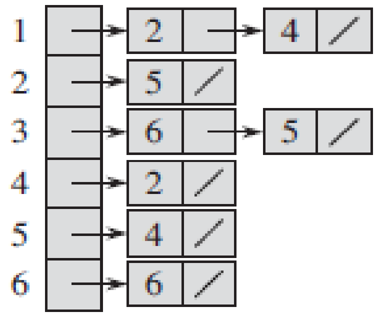
(b)

$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

(c)



(a)



(b)

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

(c)

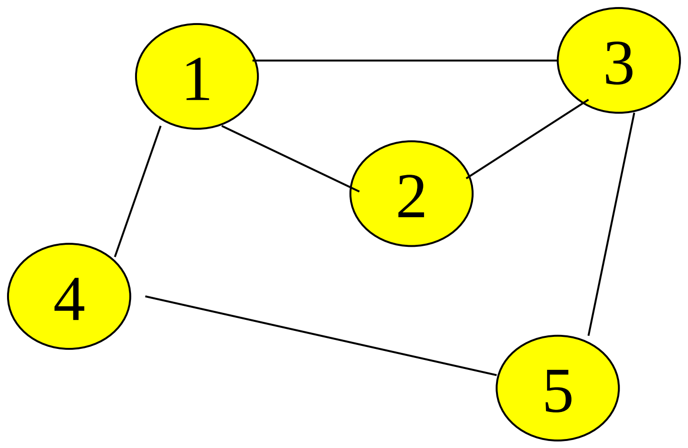
## Representation of Graphs.

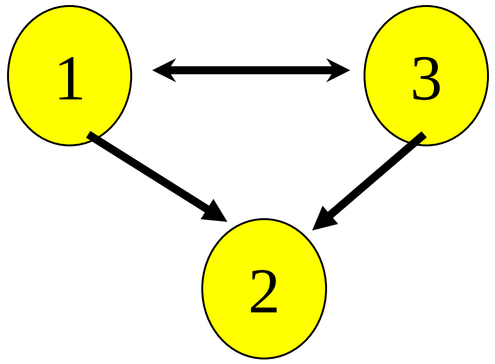
- Adjacency matrices.

The adjacency matrix of an n-vertex graph G = (V, E) is an n×n matrix A.

Each element of A is either 0 or 1.

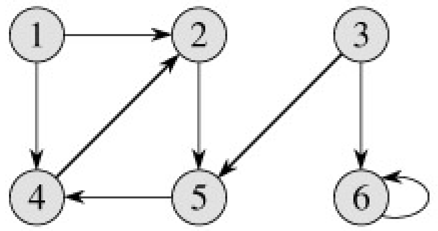
Ex: Find the adjacency matrix for the graphs.



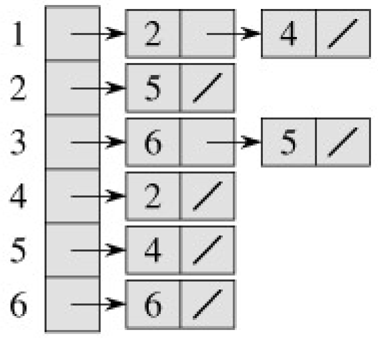


# Representation of Graphs.

- Adjacency List : Adjacency list is an array of lists. Each individual list shows what vertices a given vertex is adjacent to.



(a)

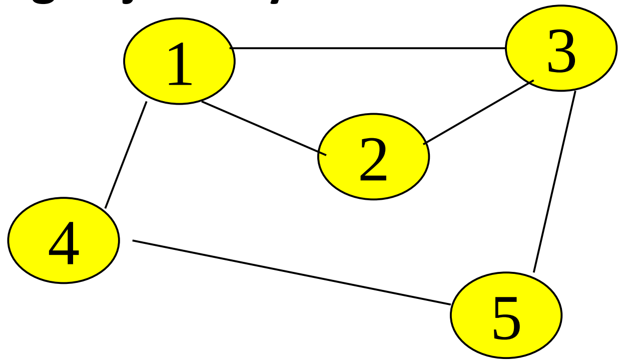


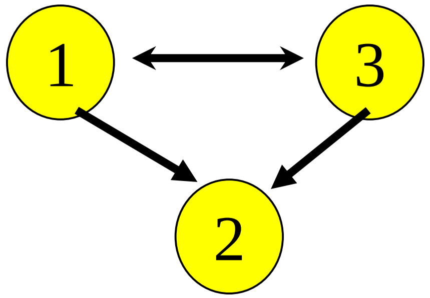
(b)

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

(c)

**Ex: Represent the graph using adjacency list.**





## Searching Graphs.

- Why do we do search?

1.To find the paths

2.To find the connectivity.

Breadth First Search (BFS)
Implemented with a queue.

Depth First Search (DFS).

Implemented with a stack.

# Breadth First Search (BFS)

- 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 any 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.

- To keep track of progress, breadth-first search colors each vertex white(for not visited nodes), gray(for processing node), or black(for visited node). All vertices start out white and may later become gray and then black.

- Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*. Whenever a white vertex *v* is discovered in the course of scanning the adjacency list of an already discovered vertex *u*, the vertex *v* and the edge (*u*, *v*) are added to the tree. We say that *u* is the *predecessor* or *parent* of *v* in the breadth-first tree .

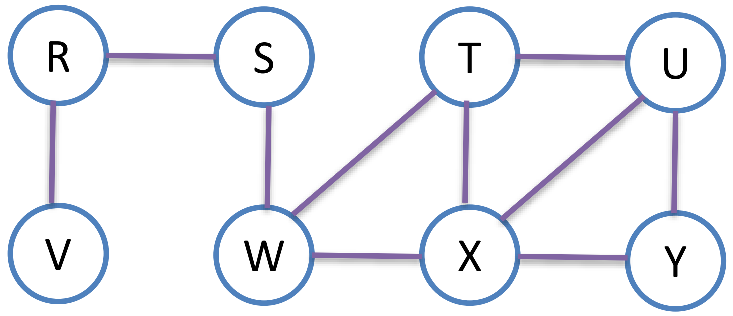
- The distance from the source *s* to vertex *u* computed by the algorithm is stored in *d*[*u*]. The algorithm also uses a first-in, first-out queue *Q* to manage the set of gray vertices.

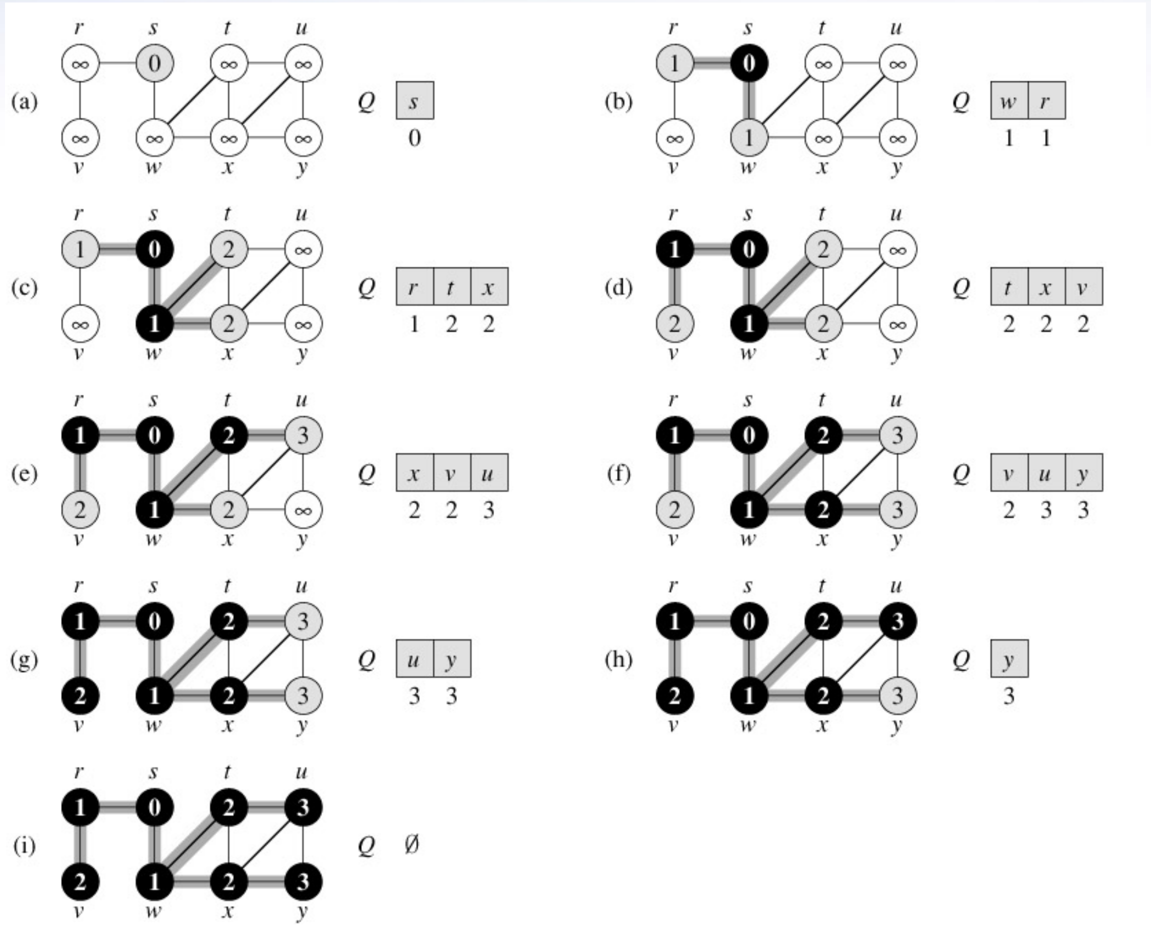
# Breadth First Search (BFS)

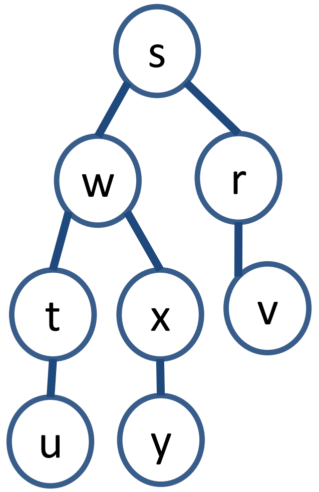
```
BFS($G$, $s$)
1  for each vertex $u \in V[G] - \{s\}$
2       do $color[u] \leftarrow$ WHITE
3       $d[u] \leftarrow \infty$
4       $\pi[u] \leftarrow$ NIL
5  $color[s] \leftarrow$ GRAY
6  $d[s] \leftarrow 0$
7  $\pi[s] \leftarrow$ NIL
8  $Q \leftarrow \emptyset$
9  ENQUEUE($Q$, $s$)
10  while $Q \neq \emptyset$
11      do $u \leftarrow$ DEQUEUE($Q$)
12            for each $v \in Adj[u]$
13                do if $color[v]$ = WHITE
14                      then $color[v] \leftarrow$ GRAY
15                           $d[v] \leftarrow d[u] + 1$
16                           $\pi[v] \leftarrow u$
17                           ENQUEUE($Q$, $v$)

18   $color[u] \leftarrow$ BLACK
```

Example : Do the BFS and draw the queue for the following graph. What is the order of nodes visited.

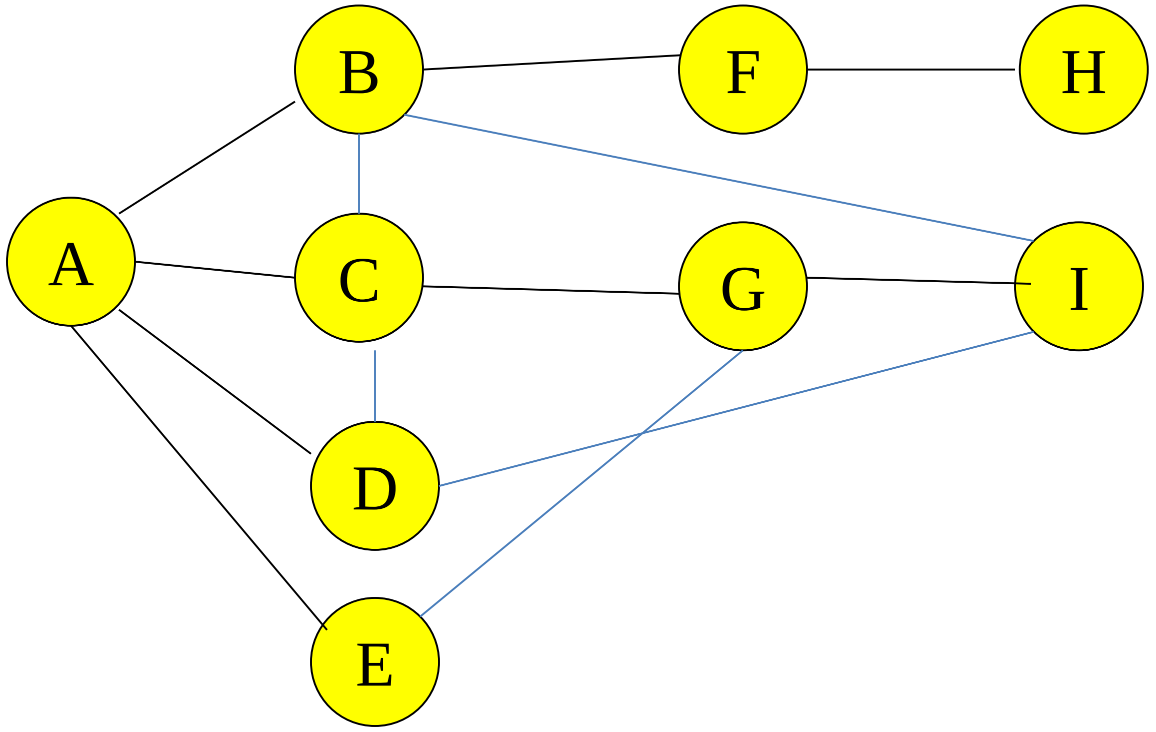


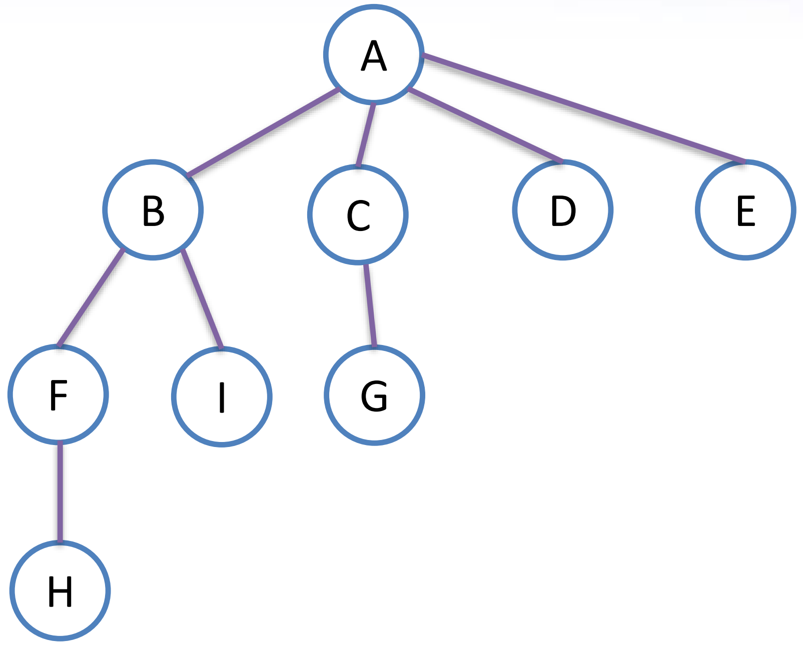




# Breadth First Search (BFS)

EX: Do the BFS and draw the queue for the following graph. What is the order of nodes visited.





## Breadth First Search (BFS)

- The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\theta$(E), the total time spent in scanning adjacency lists is O(E). The overhead for initialization is O(V), and thus the total running time of the BFS procedure is O(V+E).**BFS procedure is O(V+E).**

- Total time to queue operation is O(v).
- Total time to scanning the adjacency list is O(E).
- Running time of BFS is O(V+E).

# Depth First Search (DFS)

- It search the graph deeper whenever possible. Search is implemented by using a stack.
- 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.

## Depth First Search (DFS)

- Unlike breadth-first search, whose predecessor sub graph forms a tree, the predecessor sub graph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources.

- As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely.

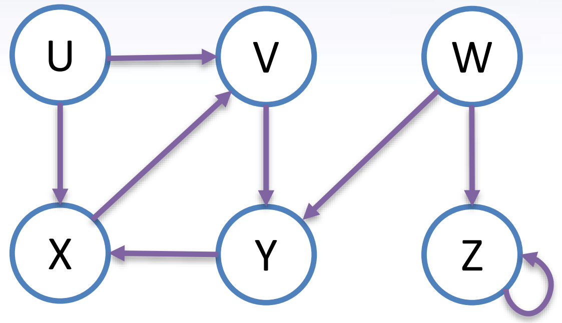
- Besides creating a depth-first forest, depth-first search also ***timestamps*** each vertex.

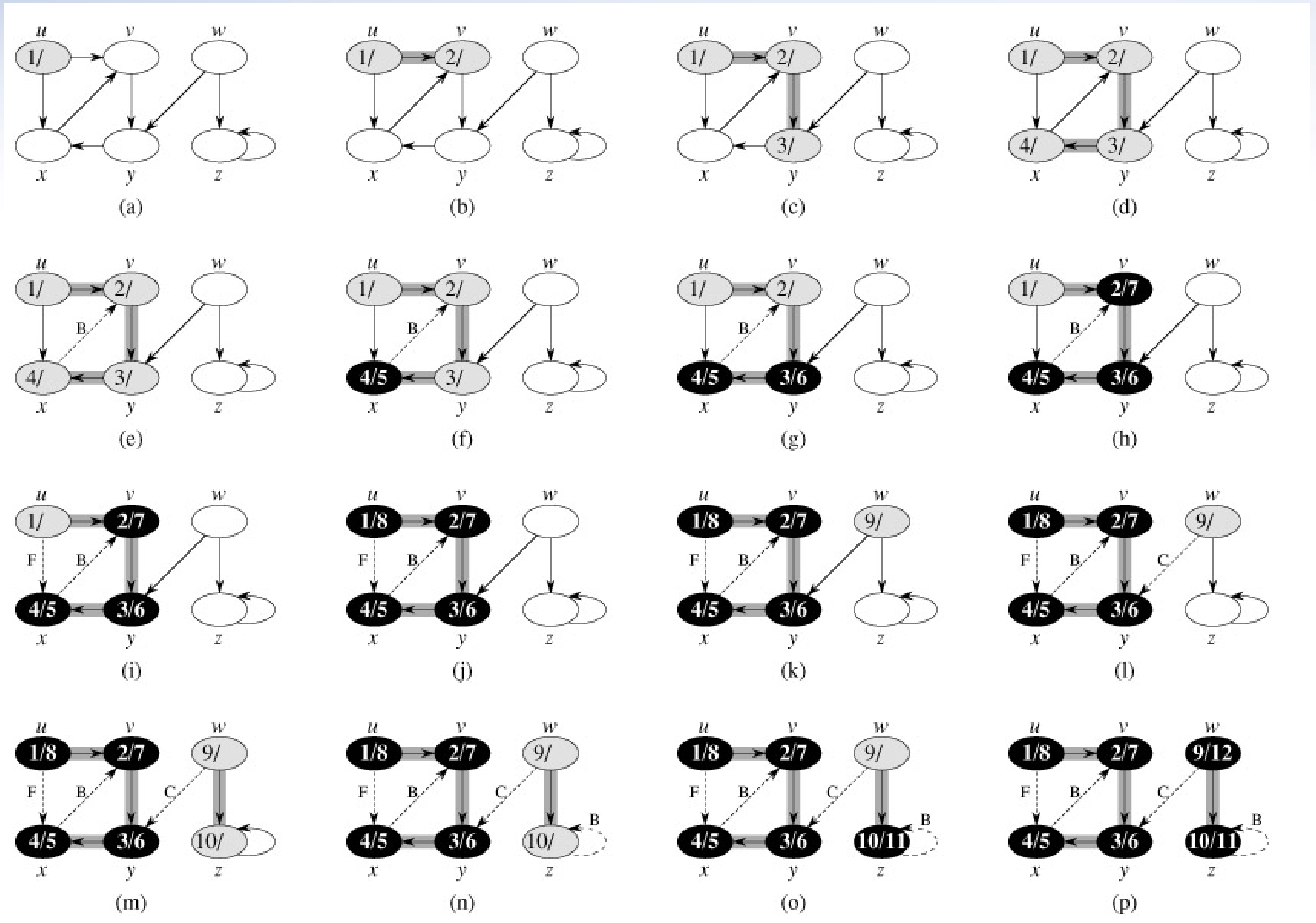
- Each vertex *v* has two timestamps: the first timestamp *d*[*v*] records when *v* is first discovered (and grayed), and the second timestamp *f* [*v*] records when the search finishes examining *v*'s adjacency list (and blackens *v*). These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search.

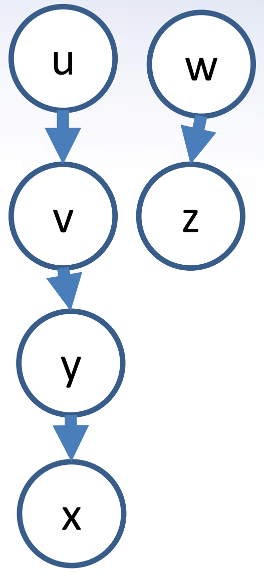
```
DFS($G$)
1  for each vertex $u \in V[G]$
2      do $color[u] \leftarrow$ WHITE
3         $\pi[u] \leftarrow$ NIL
4  $time \leftarrow 0$
5  for each vertex $u \in V[G]$
6      do if $color[u]$ = WHITE
7            then DFS-VISIT($u$)

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

# Depth First Search (DFS)

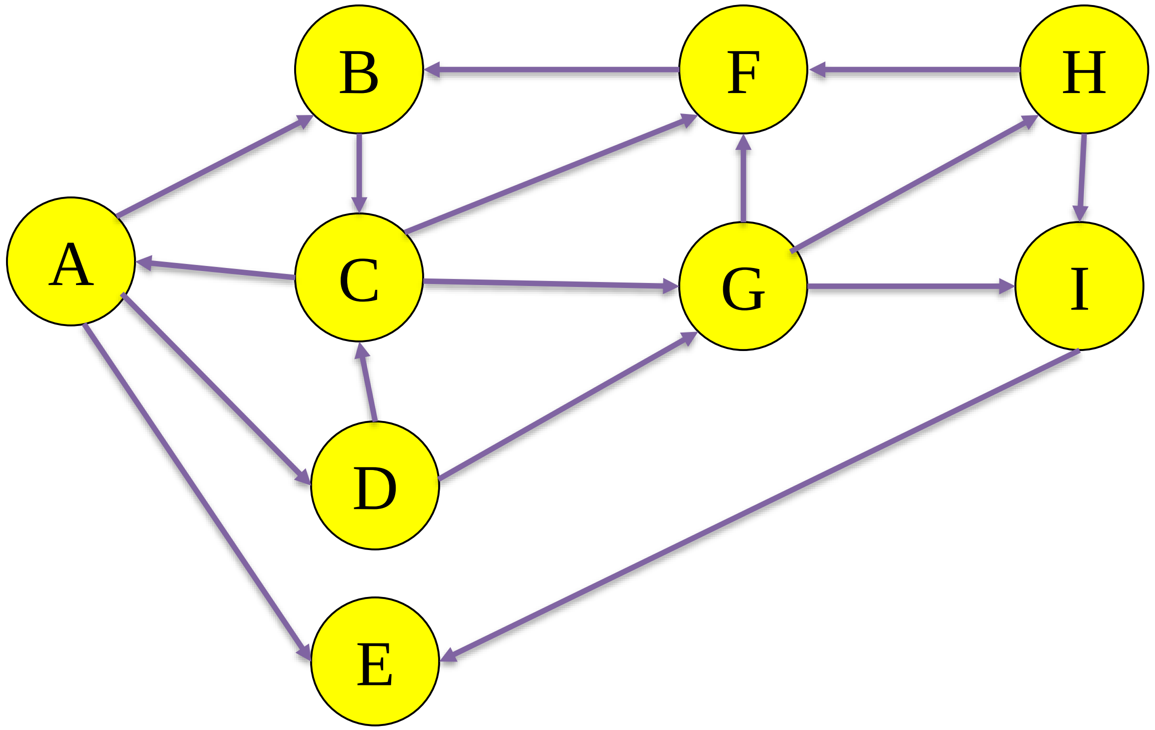






# Depth First Search (DFS)

EX: Do the DFS and draw the queue for the following graph. What is the order of nodes visited.



## Analysis DFS.

- Lines 1-2 and 5-7 take Θ(v).
- The loop on lines 3-6 is executed |adj[v]| times.
- Total length of adjacency list is Θ(E).
- Running time of DFS is Θ(v+E).

## Summary

- What is graph.
- Graph terminology.
- Graph representation.
- Searching
- BFS
- DFS

## Questions ???

Thank You

anuruddha.a@sliit.lk