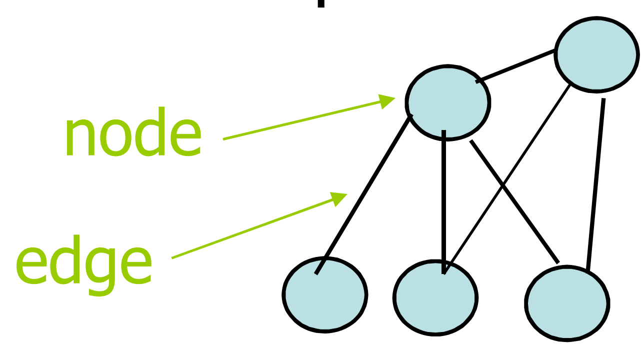
Data Structures

Graphs

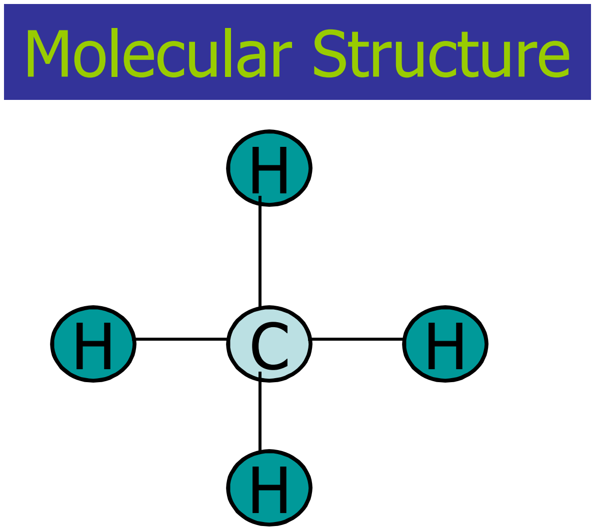
Graphs

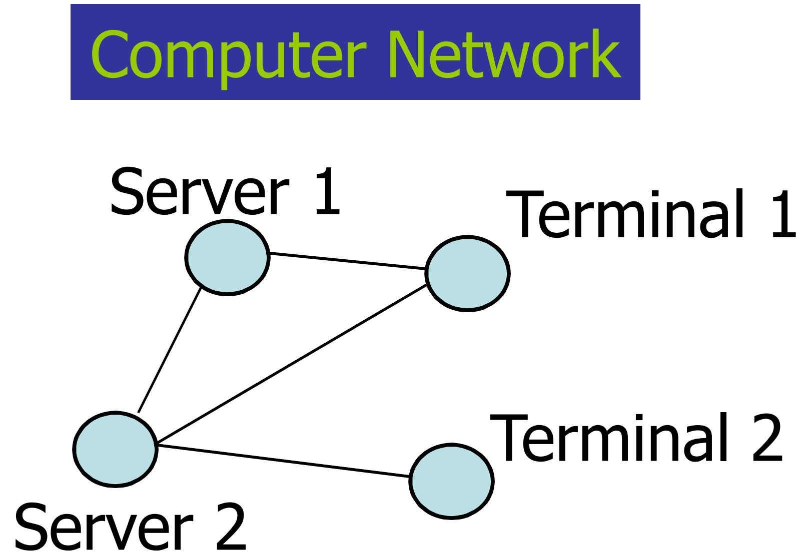
What is a graph?

- Graphs represent the relationships among data items
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between two items



Examples of graphs

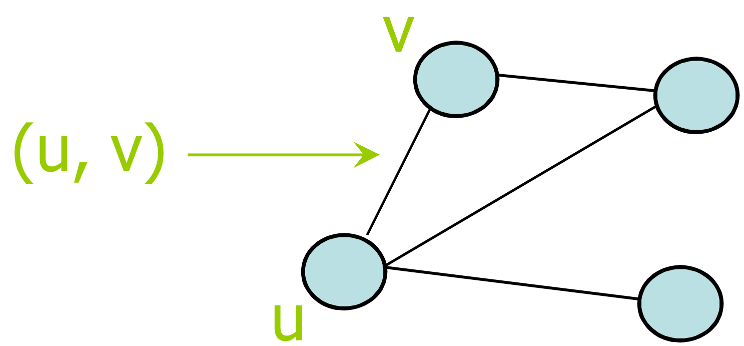




Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects

Formal Definition of graph

- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connected by an edge, such edge is denoted as (u, v)

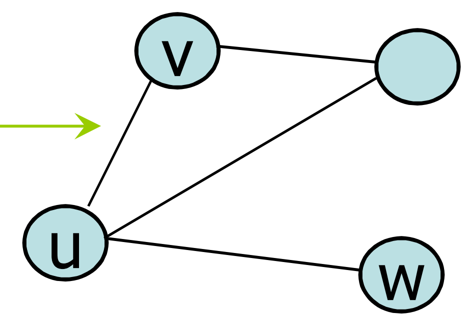


- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

Adjacent

- Two nodes u and v are said to be adjacent if $(u, v) \in E$

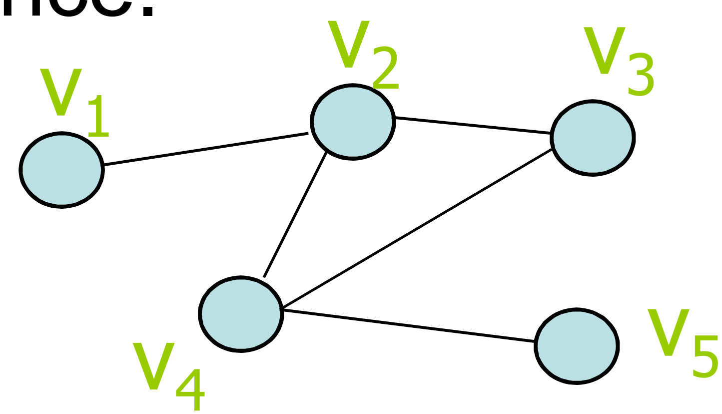
(u, v)



u and v are adjacent
v and w are not adjacent

Path and simple path

- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.

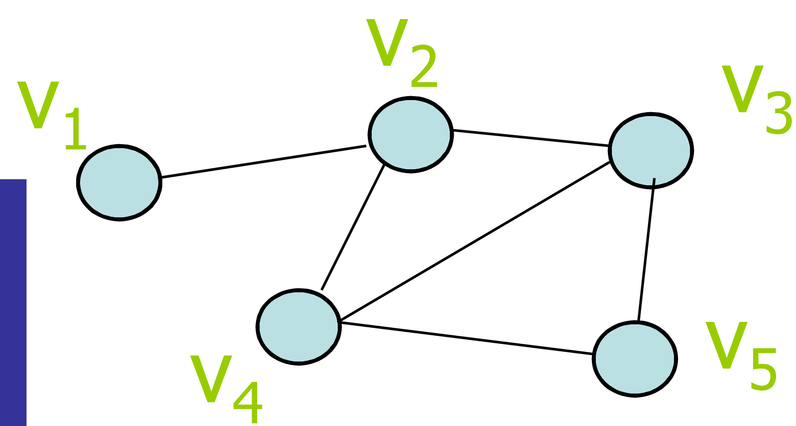


- v_2, v_3, v_4, v_2, v_1 is a path
- v_2, v_3, v_4, v_5 is a path, also it is a simple path

Cycle and simple cycle

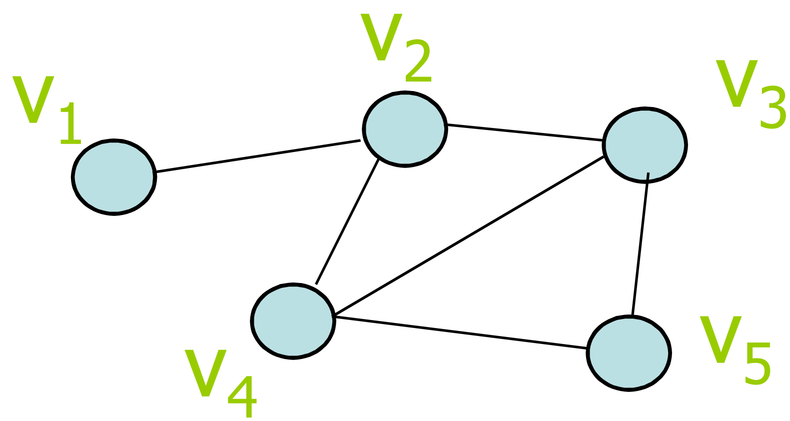
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes

- $v_2, v_3, v_4, v_5, v_3, v_2$ is a cycle
- v_2, v_3, v_4, v_2 is a cycle, it is also a simple cycle



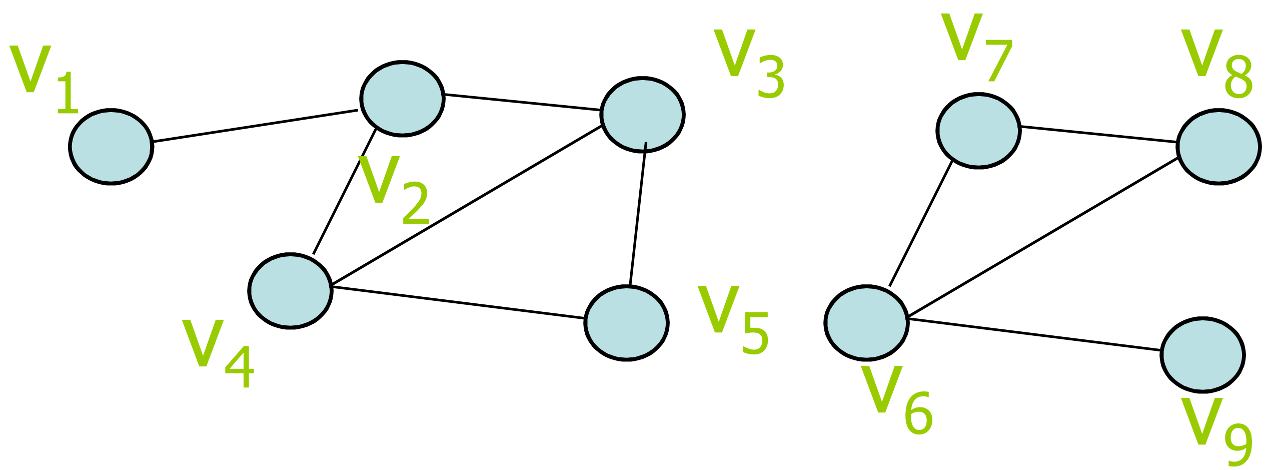
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

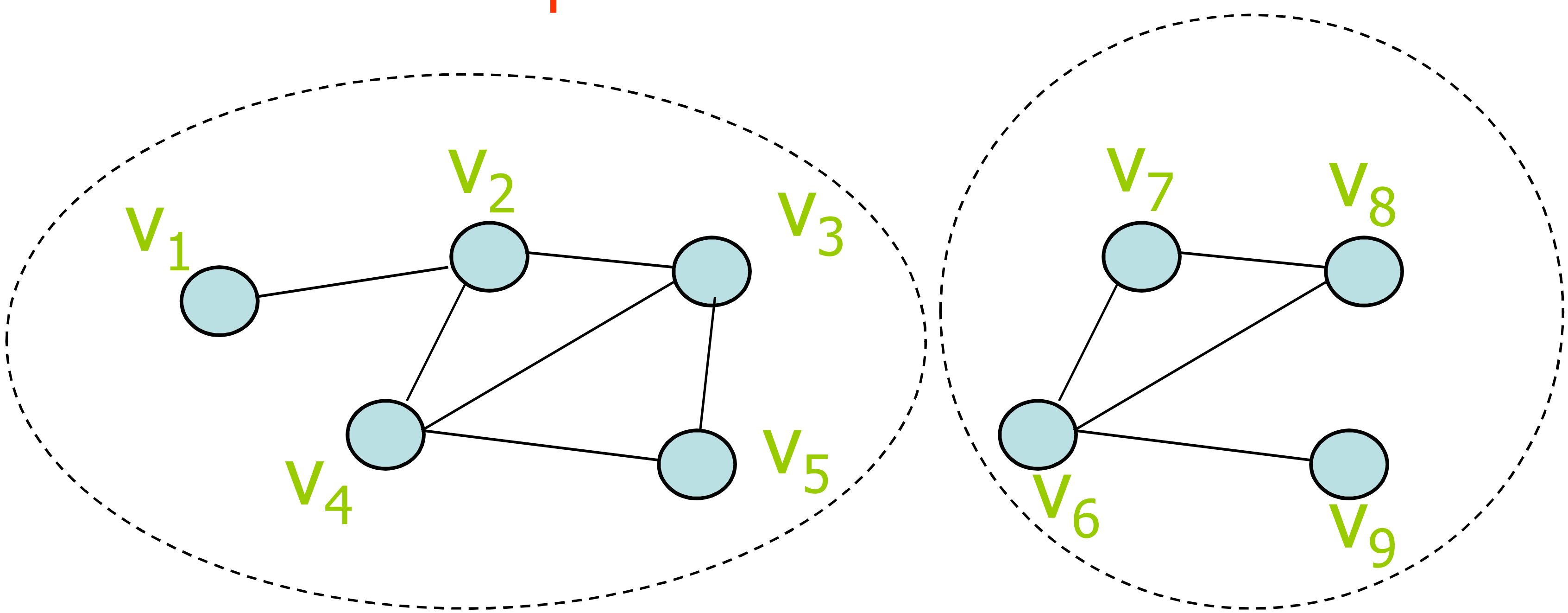
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

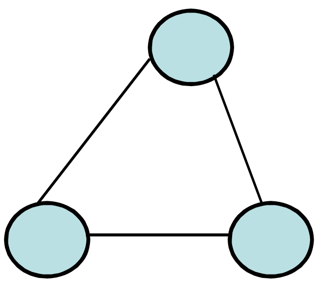
Connected component

- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.

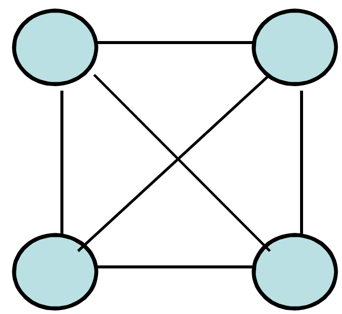


Complete graph

- A graph is complete if each pair of distinct nodes has an edge



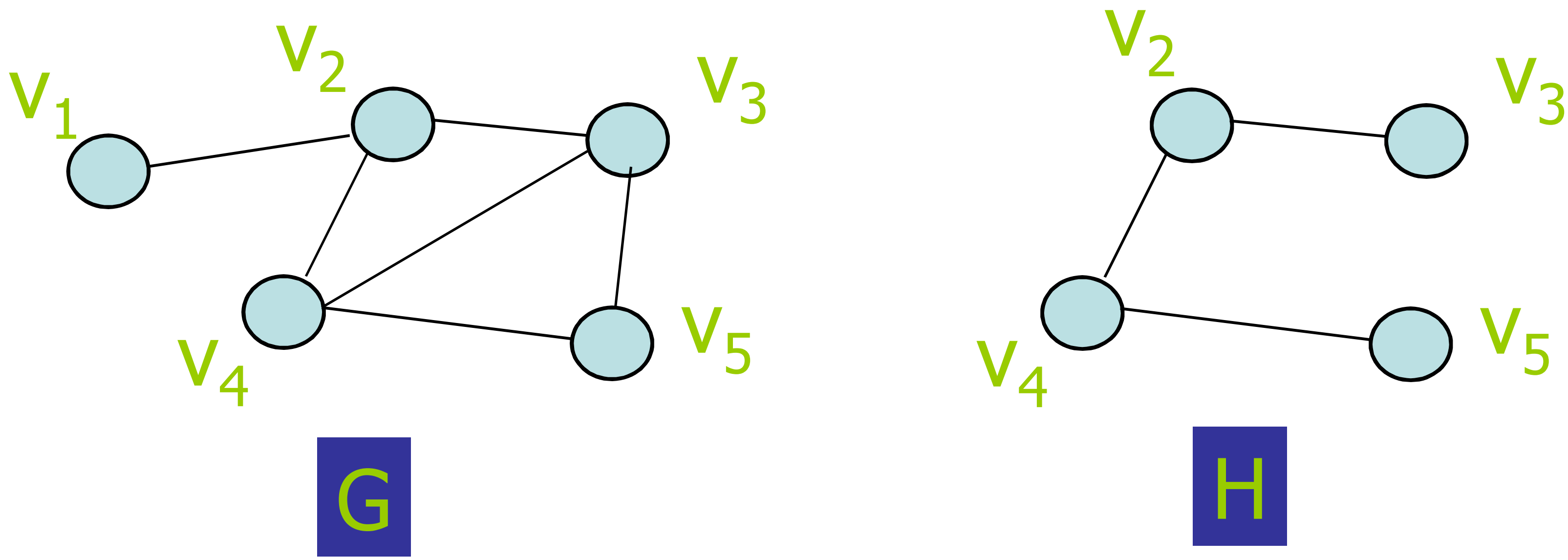
Complete graph with 3 nodes



Complete graph with 4 nodes

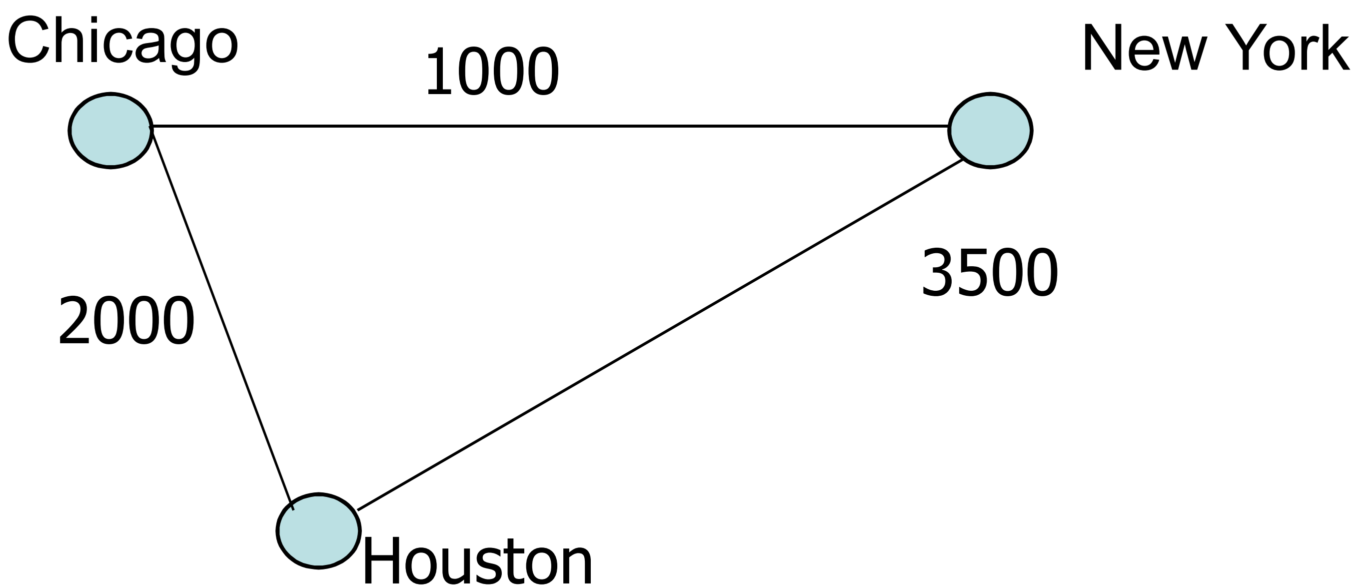
Subgraph

- A subgraph of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$.



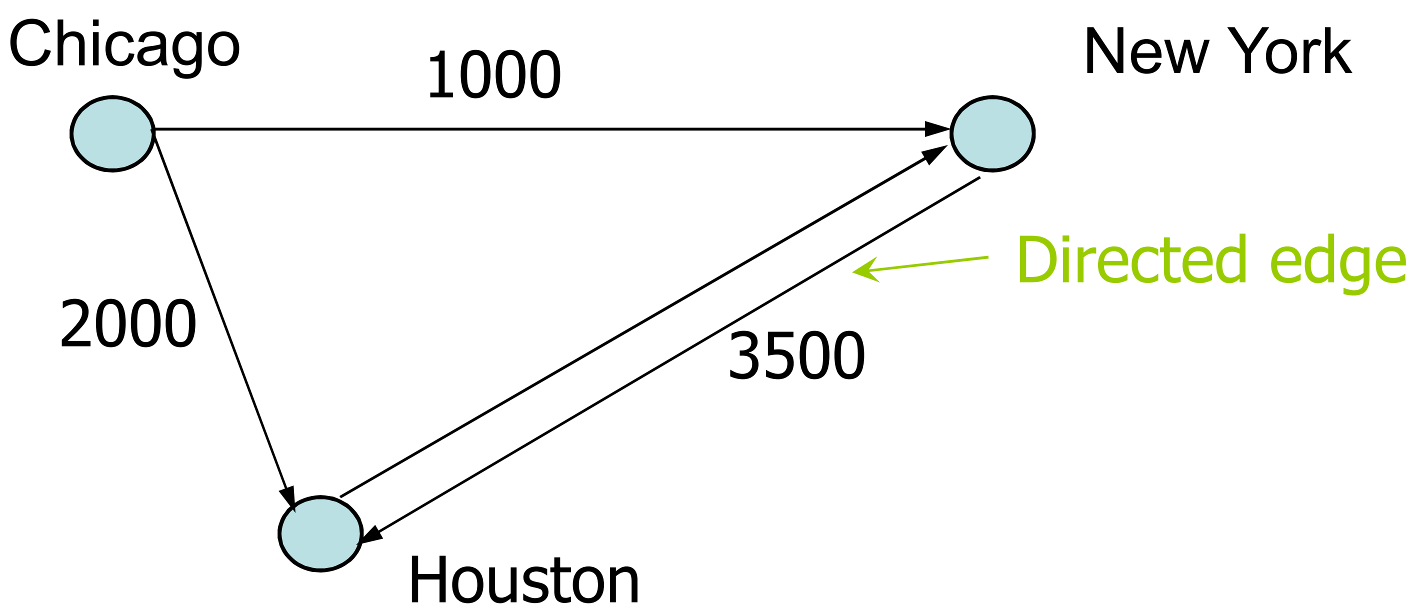
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph

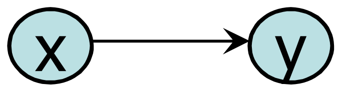


Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed edge
- A directed graph is a graph where every edges is a directed edge



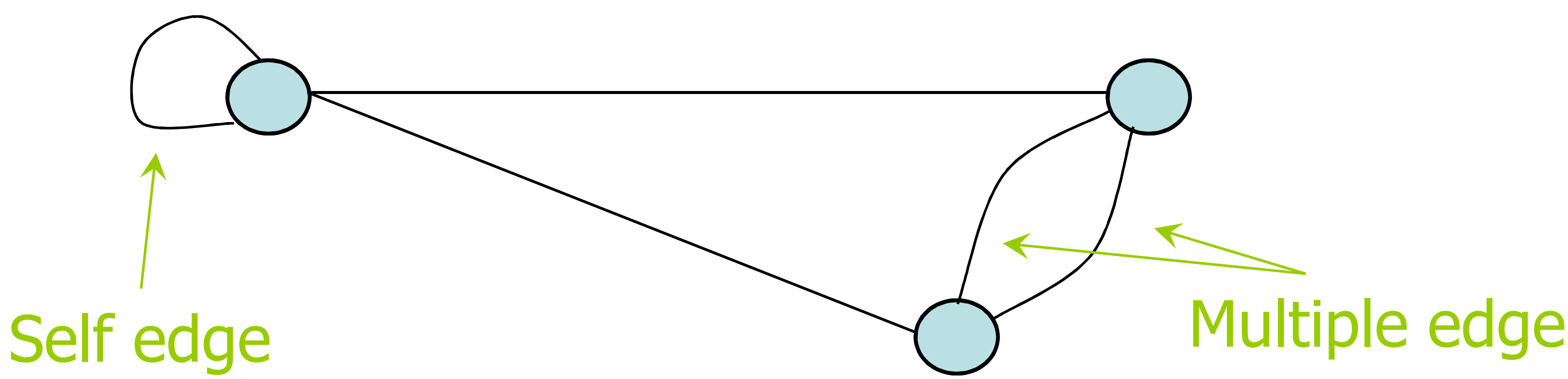
More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

Multigraph

- A graph cannot have duplicate edges.
- Multigraph allows multiple edges and self edge (or loop).



Property of graph

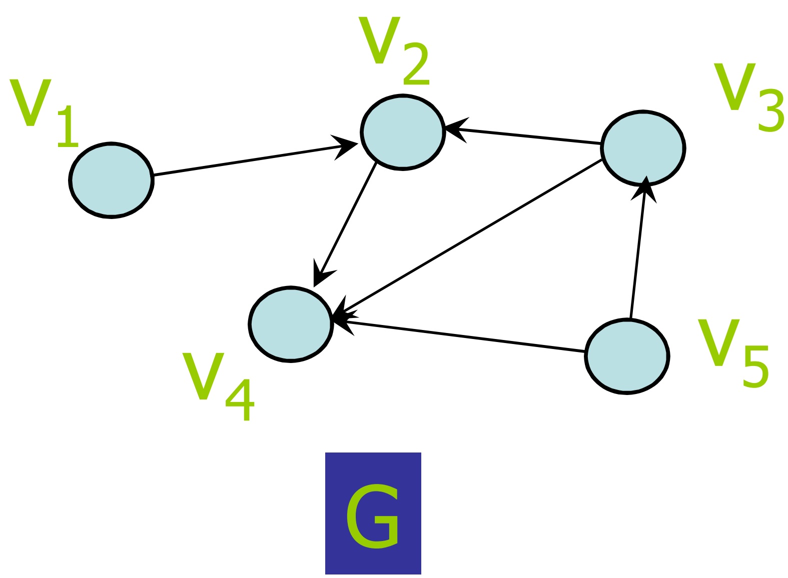
- A undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes have exactly n-1 edges.
- A connected undirected graph with n nodes must have at least n-1 edges.

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional array
- Adjacency list
  - Represent a graph using n linked lists where n is the number of vertices

Adjacency matrix for directed graph

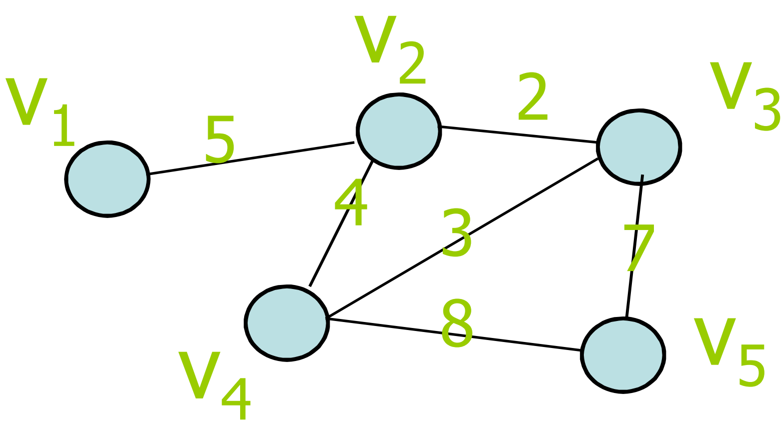
$$\text{Matrix}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$



		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Adjacency matrix for weighted undirected graph

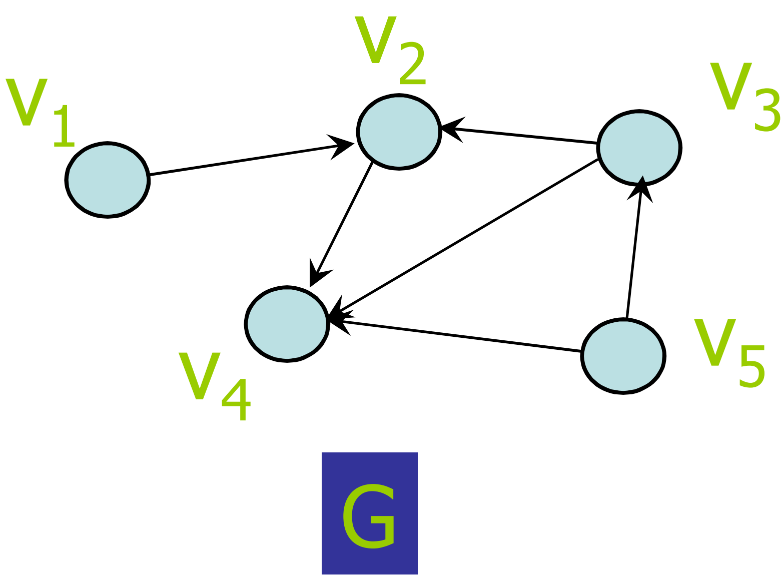
Matrix[i][j] = $w(v_i, v_j)$ if $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$
∞ otherwise



G

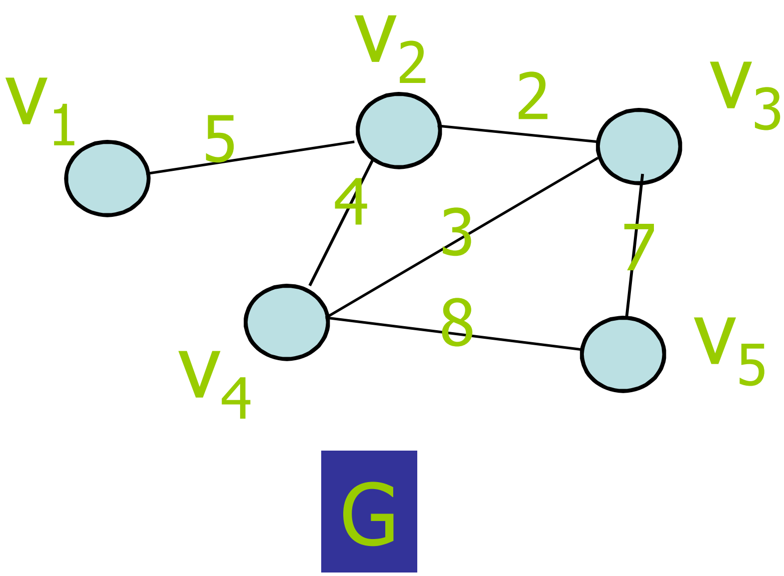
		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	∞	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Adjacency list for directed graph



1	v_1	$\rightarrow v_2$
2	v_2	$\rightarrow v_4$
3	v_3	$\rightarrow v_2 \rightarrow v_4$
4	v_4	
5	v_5	$\rightarrow v_3 \rightarrow v_4$

Adjacency list for weighted undirected graph



1	v_1	$\rightarrow v_2(5)$
2	v_2	$\rightarrow v_1(5) \rightarrow v_3(2) \rightarrow v_4(4)$
3	v_3	$\rightarrow v_2(2) \rightarrow v_4(3) \rightarrow v_5(7)$
4	v_4	$\rightarrow v_2(4) \rightarrow v_3(3) \rightarrow v_5(8)$
5	v_5	$\rightarrow v_3(7) \rightarrow v_4(8)$

Pros and Cons

- Adjacency matrix
  - Allows us to determine whether there is an edge from node i to node j in O(1) time
- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is sparse, adjacency list requires less space

Graph Traversal Algorithm

- To traverse a tree, we use tree traversal algorithms like pre-order, in-order, and post-order to visit all the nodes in a tree
- Similarly, graph traversal algorithm tries to visit all the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node v will visit only a subset of nodes, that is, the connected component containing v.

Two basic traversal algorithms

- Two basic graph traversal algorithms:
  - Depth-first-search (DFS)
    - After visit node v, DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up
  - Breadth-first-search (BFS)
    - After visit node v, BFS strategy visits every node adjacent to v before visiting any other nodes

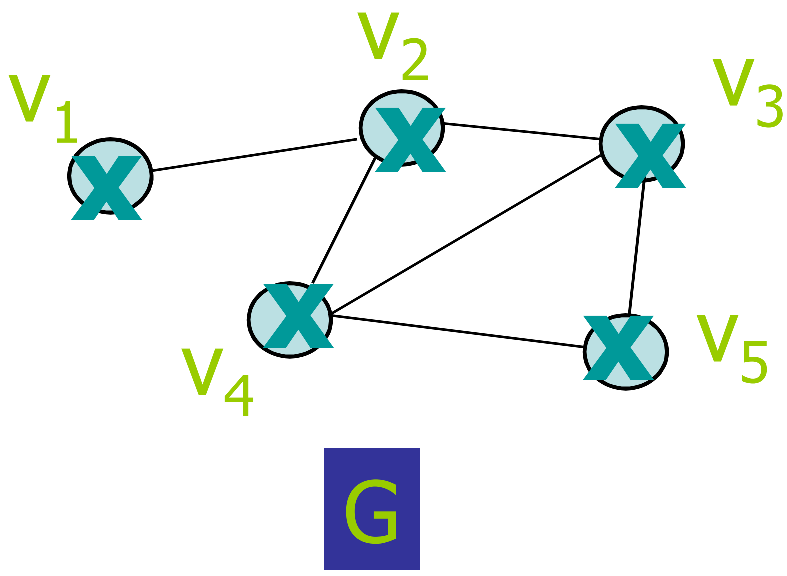
Depth-first search (DFS)

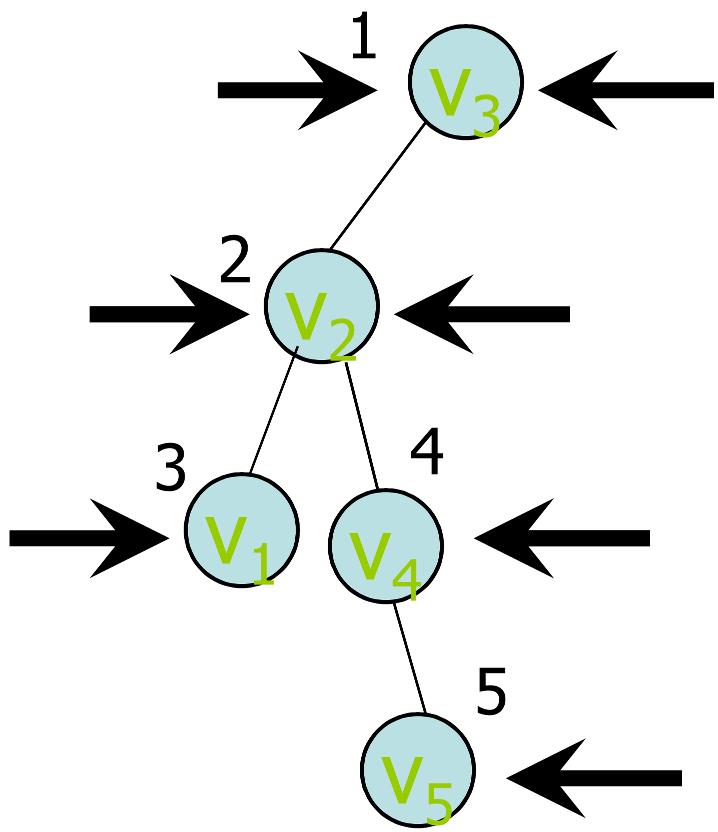
- DFS strategy looks similar to pre-order. From a given node v, it first visits itself. Then, recursively visit its unvisited neighbours one by one.
- DFS can be defined recursively as follows.

```
Algorithm dfs(v)
print v; // you can do other things!
mark v as visited;
for (each unvisited node u adjacent to v)
    dfs(u);
```

DFS example

- Start from v_3



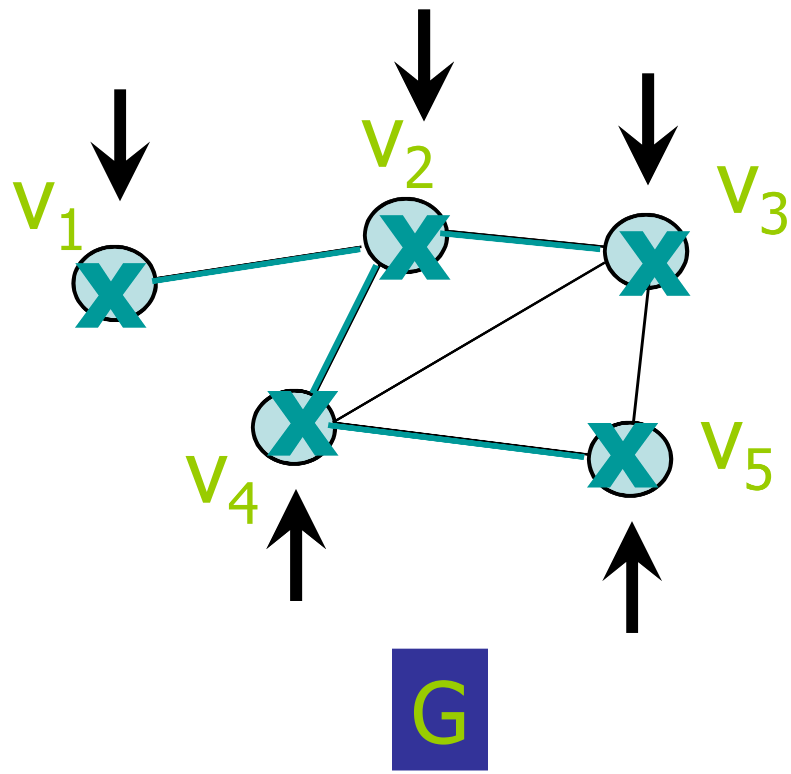


DFS algorithm

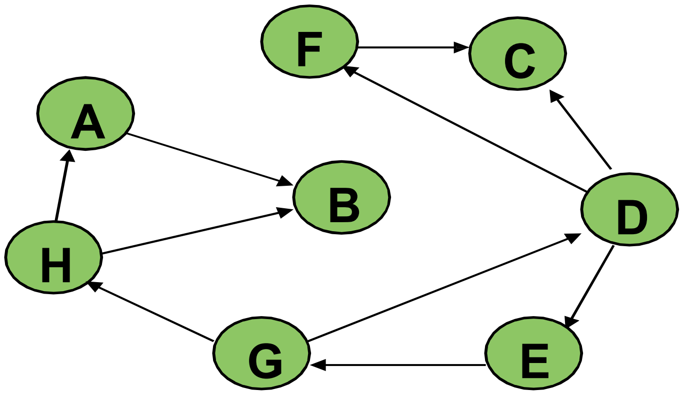
```
Algorithm dfs(v)
s.createStack();
s.push(v);
mark v as visited;
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x)
            s.pop(); // blacktrack
    else {
            select an unvisited node u adjacent to x;
            s.push(u);
            mark u as visited;
    }
}
```

Non-recursive DFS example

visit	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4 , v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty



Walk-Through

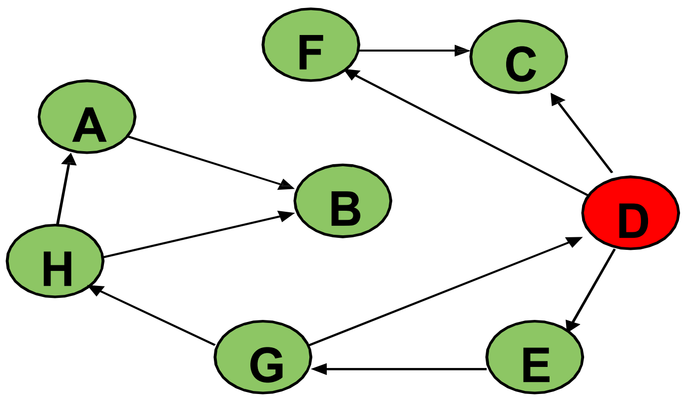


Visited Array

A	
B	
C	
D	
E	
F	
G	
H	

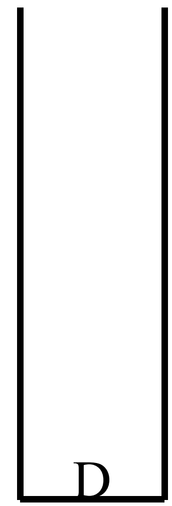
Task: Conduct a depth-first search of the graph starting with node D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

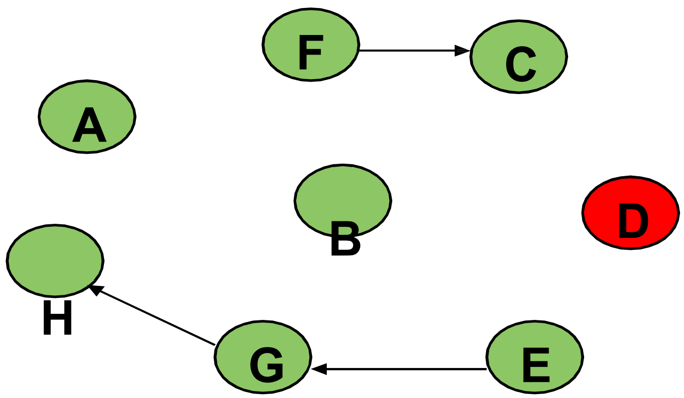


Visit D

The order nodes are visited:

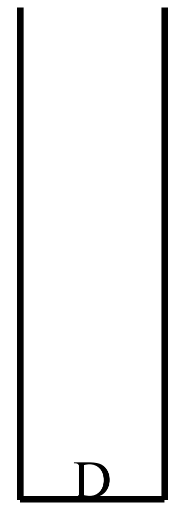
D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

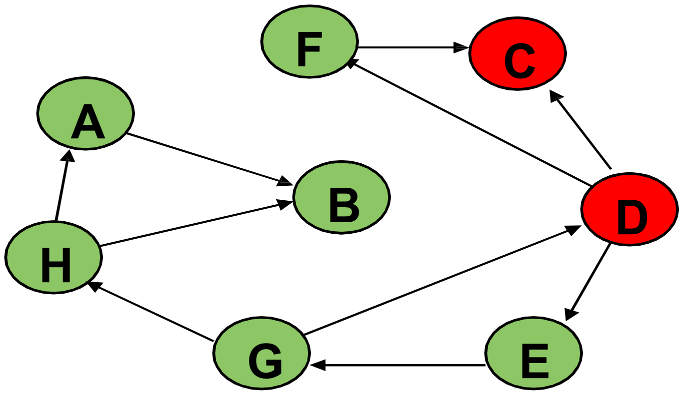


The order nodes are visited:

D

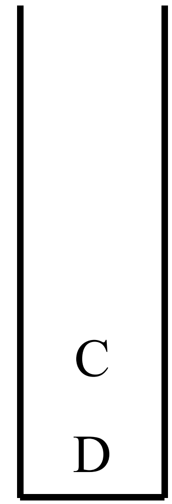
Consider nodes adjacent to D, decide to visit C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

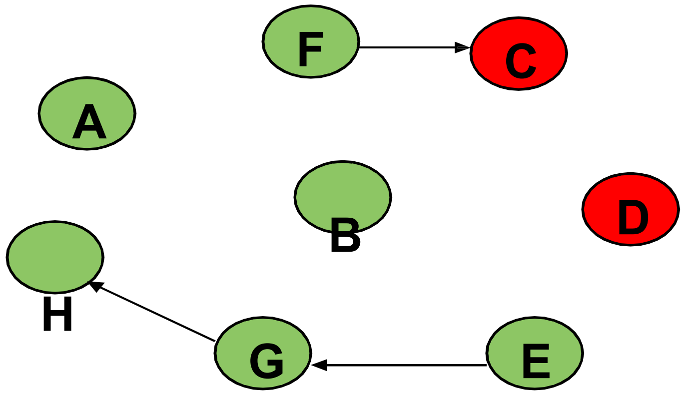


Visit C

The order nodes are visited:

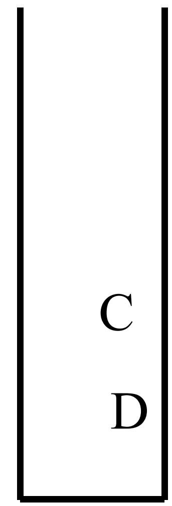
D, C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

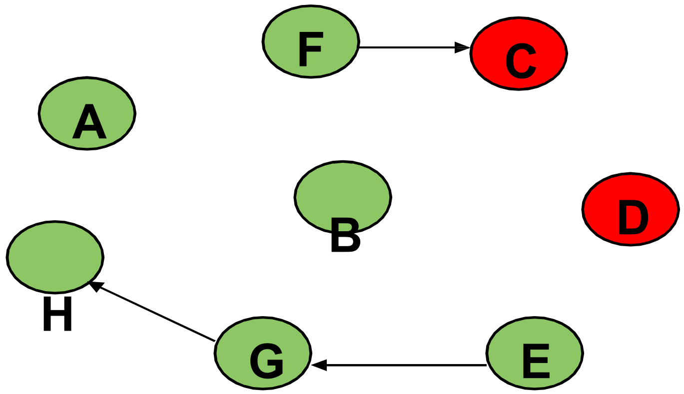


The order nodes are visited:

D, C

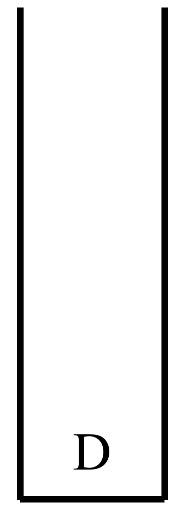
No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

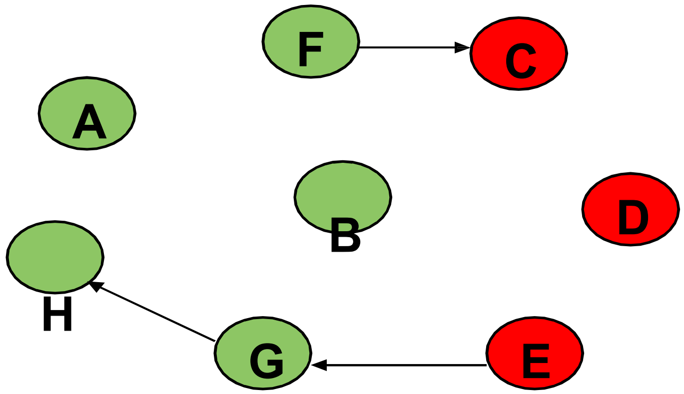


The order nodes are visited:

D, C

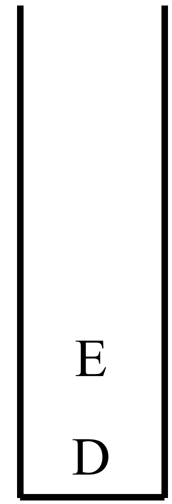
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

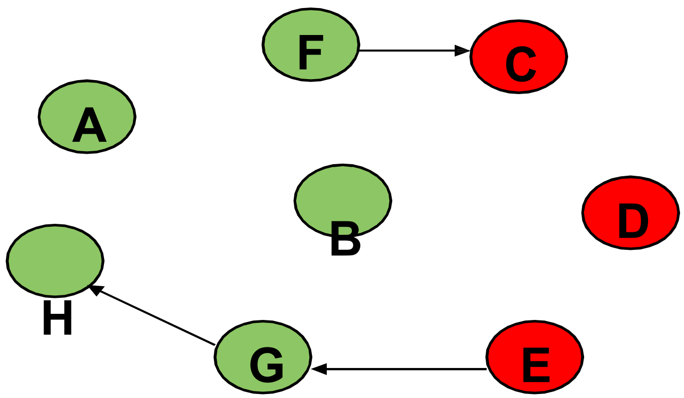


The order nodes are visited:

D, C, E

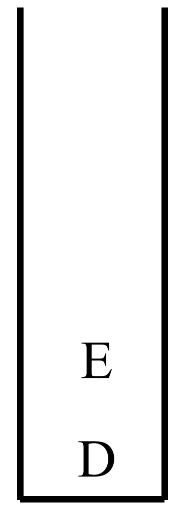
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

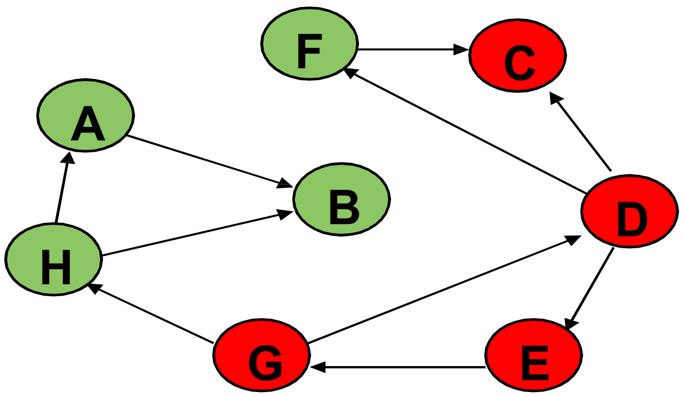


The order nodes are visited:

D, C, E

Only G is adjacent to E

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

G

E

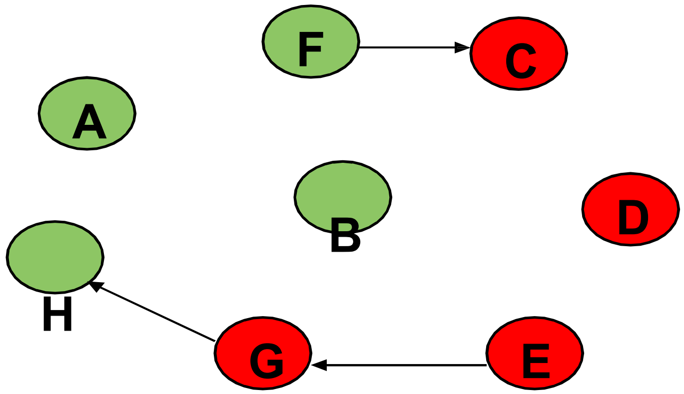
D

Visit G

The order nodes are visited:

D, C, E, G

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

G

E

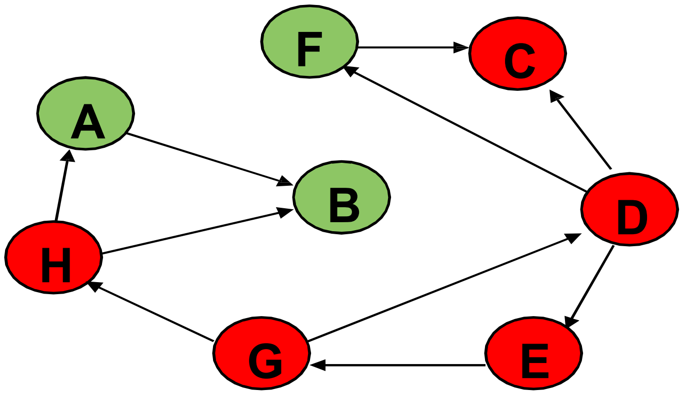
D

The order nodes are visited:

D, C, E, G

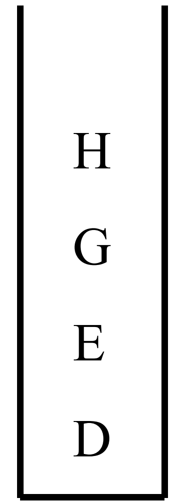
Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

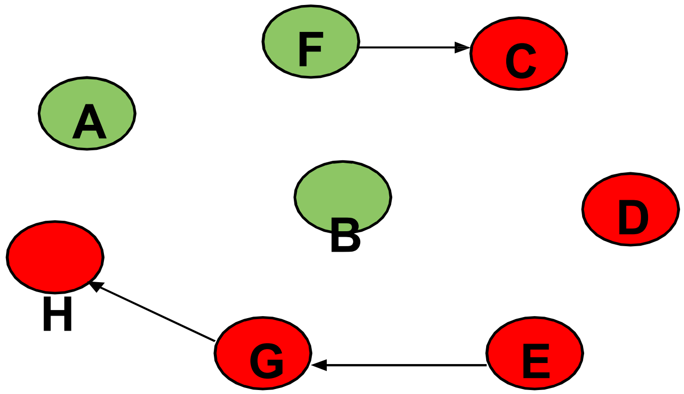


Visit H

The order nodes are visited:

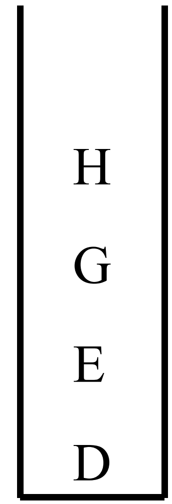
D, C, E, G, H

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

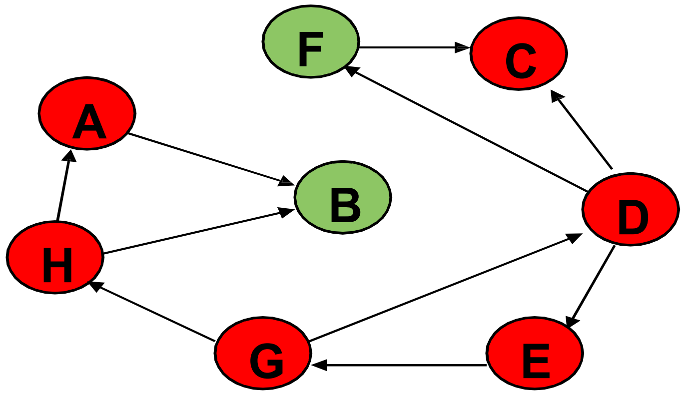


The order nodes are visited:

D, C, E, G, H

Nodes A and B are adjacent to F. Decide to visit A next.

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

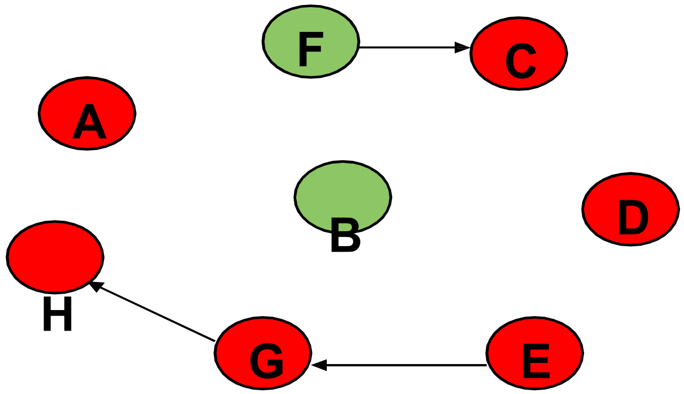
A
H
G
E
D

Visit A

The order nodes are visited:

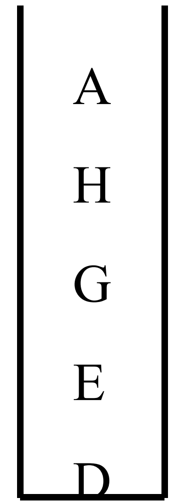
D, C, E, G, H, A

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

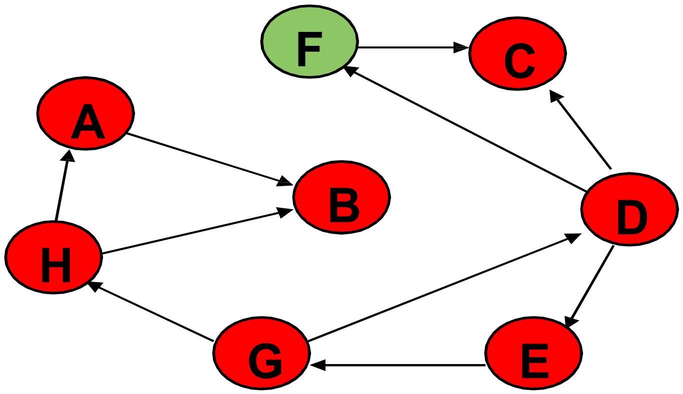


The order nodes are visited:

D, C, E, G, H, A

Only Node B is adjacent to A. Decide to visit B next.

Walk-Through



The order nodes are visited:

D, C, E, G, H, A, B

Visited Array

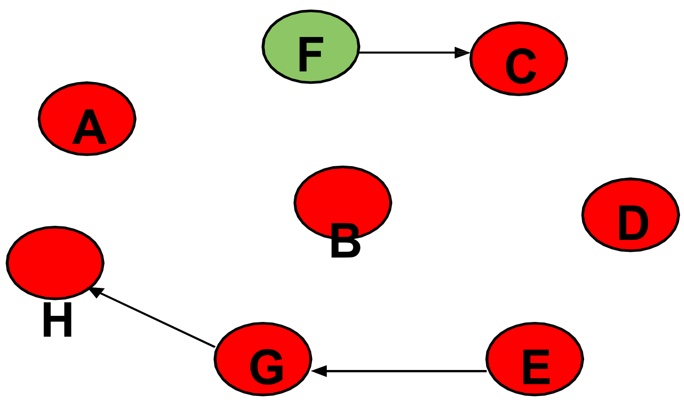
A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

B

A
H
G
E
D

Visit B

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

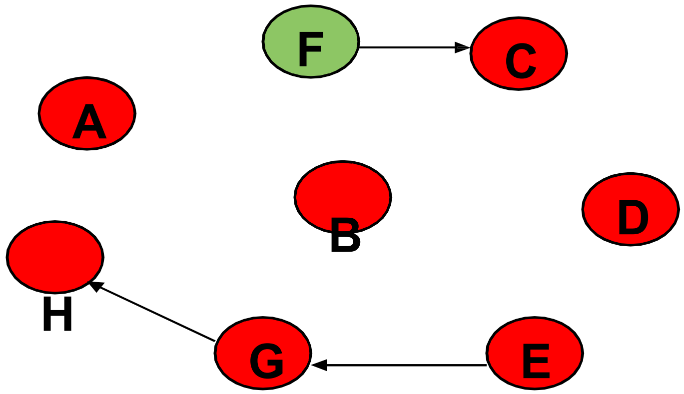
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A, B

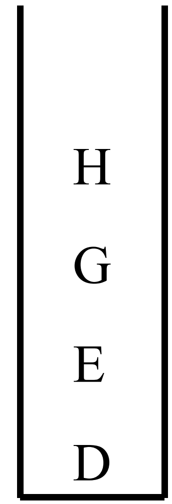
No unvisited nodes adjacent to B. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

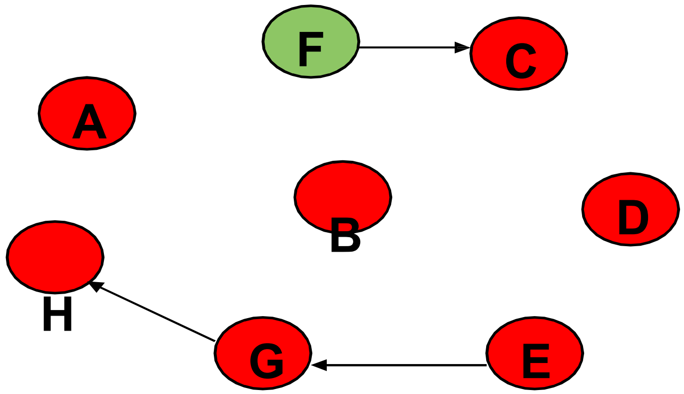


The order nodes are visited:

D, C, E, G, H, A, B

No unvisited nodes adjacent to A. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

G

E

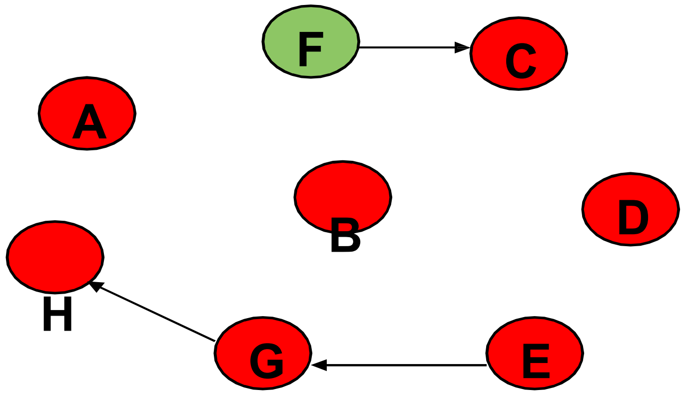
D

The order nodes are visited:

D, C, E, G, H, A, B

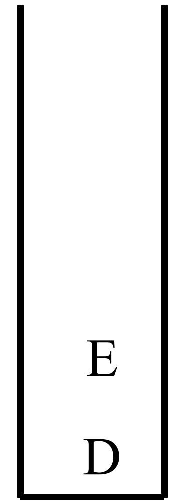
No unvisited nodes adjacent to H. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√



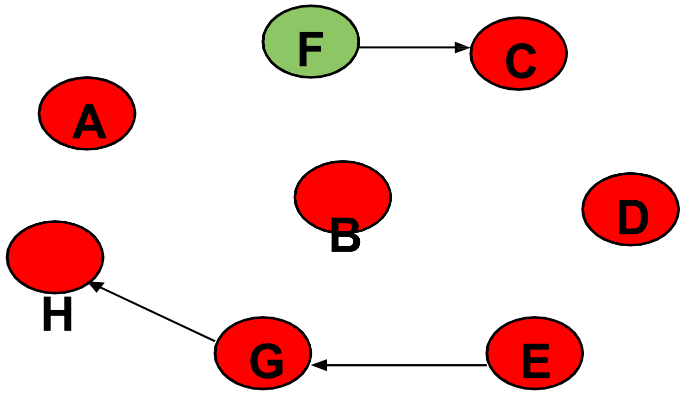
The order nodes are visited:

D, C, E, G, H, A, B

No unvisited nodes adjacent to G.

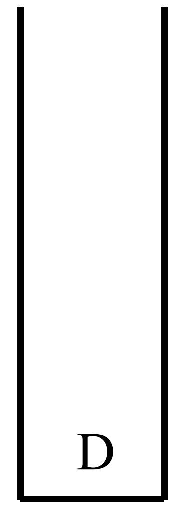
Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

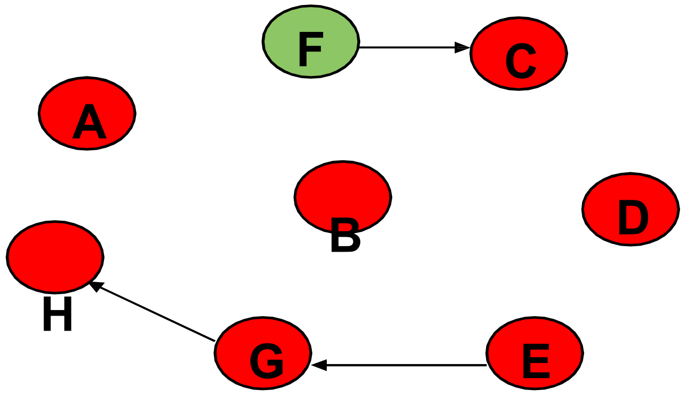


The order nodes are visited:

D, C, E, G, H, A, B

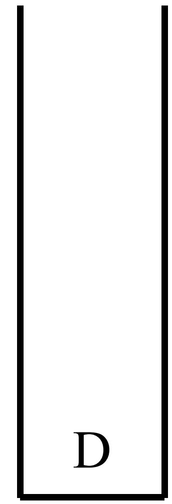
No unvisited nodes adjacent to E. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

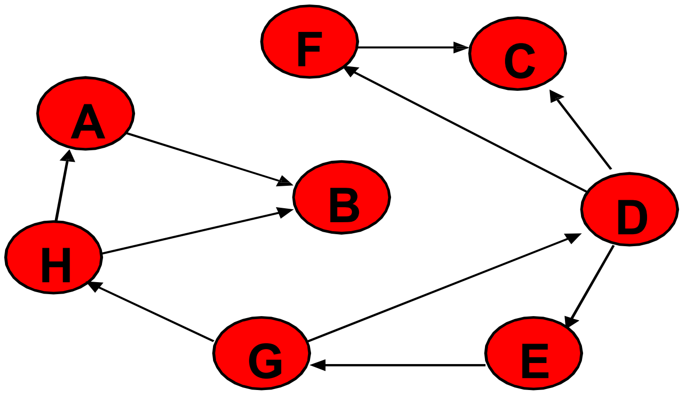


The order nodes are visited:

D, C, E, G, H, A, B

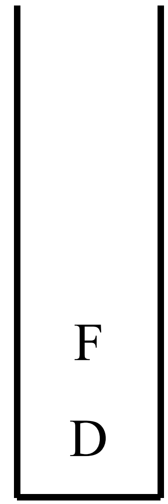
F is unvisited and is adjacent to D. Decide to visit F next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

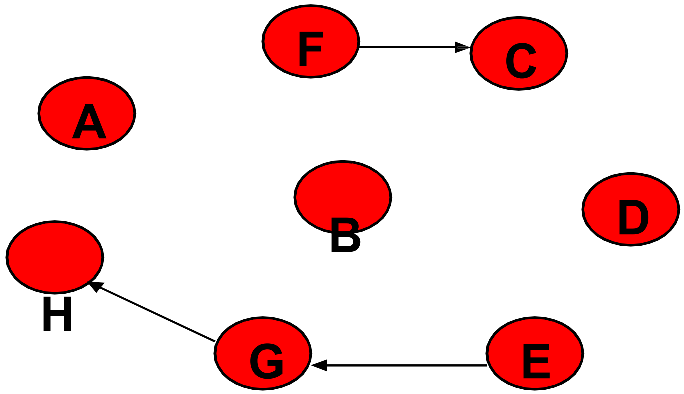


Visit F

The order nodes are visited:

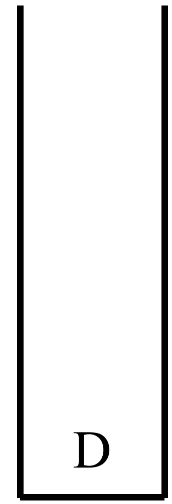
D, C, E, G, H, A, B, F

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

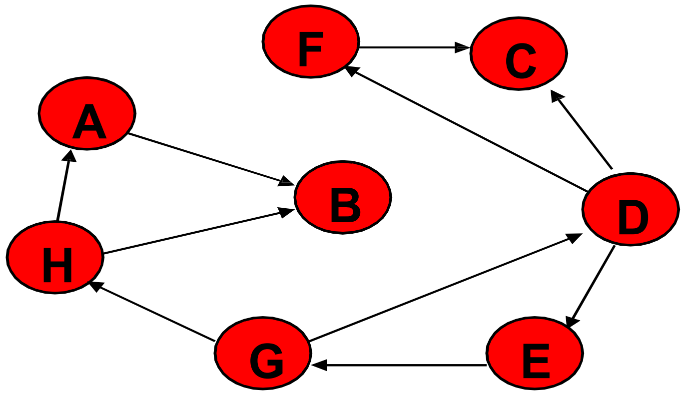


The order nodes are visited:

D, C, E, G, H, A, B, F

No unvisited nodes adjacent to F. Backtrack.

Walk-Through



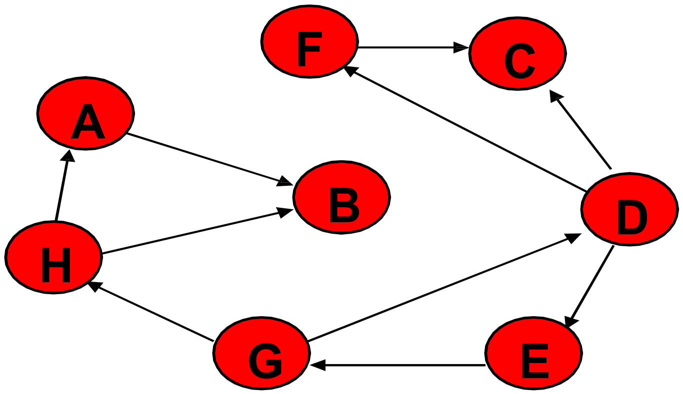
Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:
D, C, E, G, H, A, B, F

No unvisited nodes adjacent to D. Backtrack.

Walk-Through



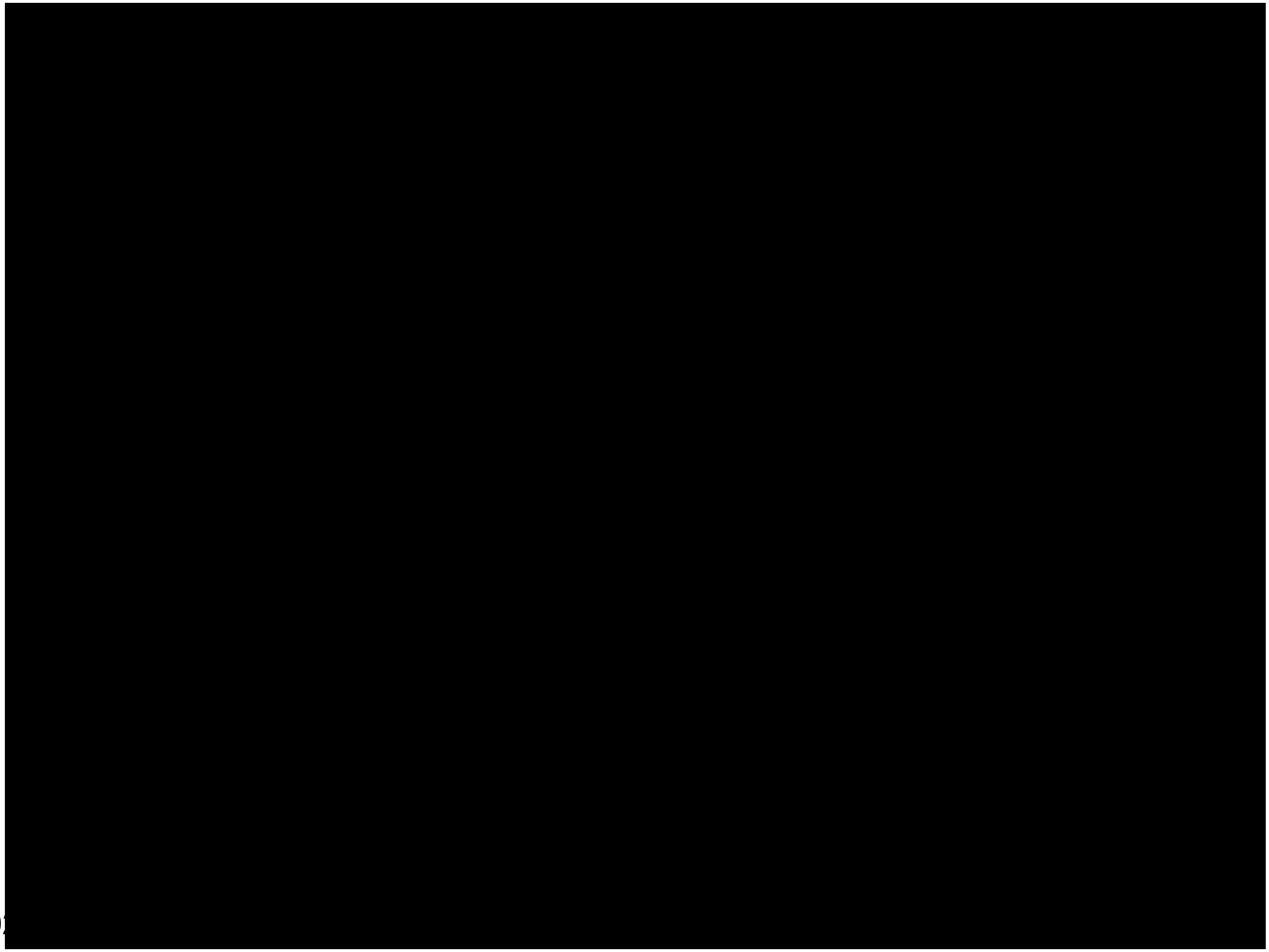
Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:
D, C, E, G, H, A, B, F

Stack is empty. Depth-first traversal is done.

DFS

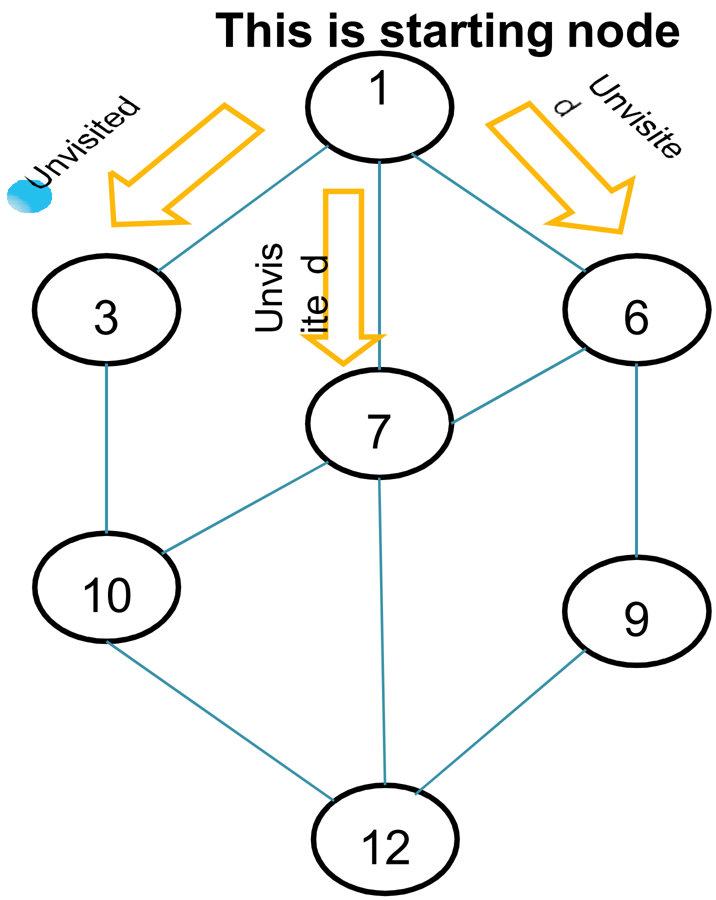


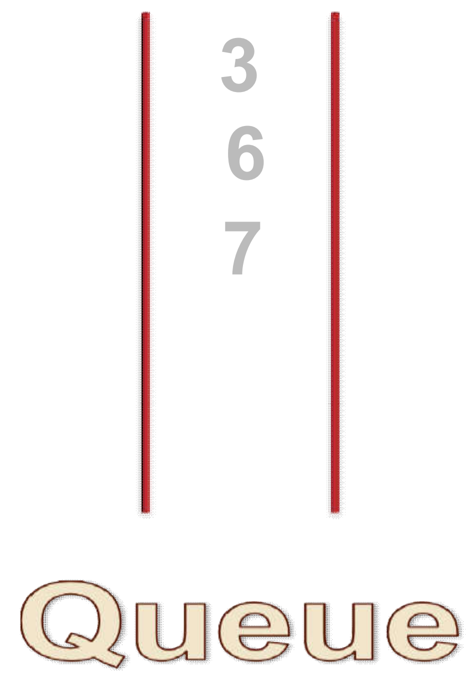
Breadth-first search (BFS)

- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits every node adjacent to v before visiting any other nodes.
  - 1. Visit v
  - 2. Visit all v's neigbours
  - 3. Visit all v's neighbours' neighbours
  - …
- Similar to level-order, BFS is based on a queue.

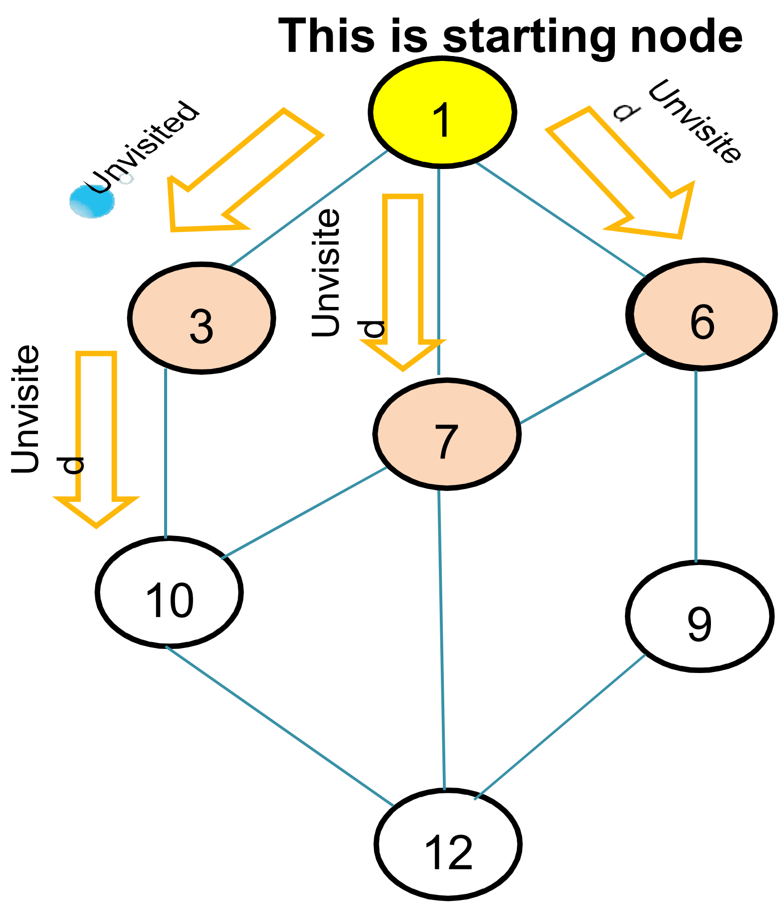
Algorithm for BFS

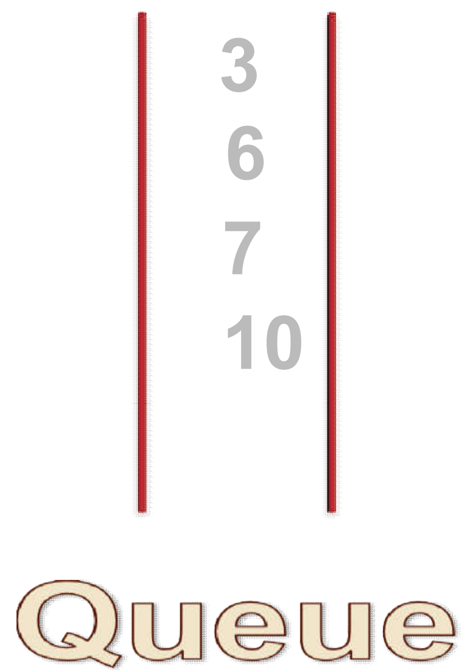
```
Algorithm bfs(v)
q.createQueue();
q.enqueue(v);
mark v as visited;
while(!q.isEmpty()) {
    w = q.dequeue();
    for (each unvisited node u adjacent to w) {
        q.enqueue(u);
        mark u as visited;
    }
}
```


This is starting node
1
3
6
7
10
9
12
Unvisited
Unvisite d
Unvis ite d

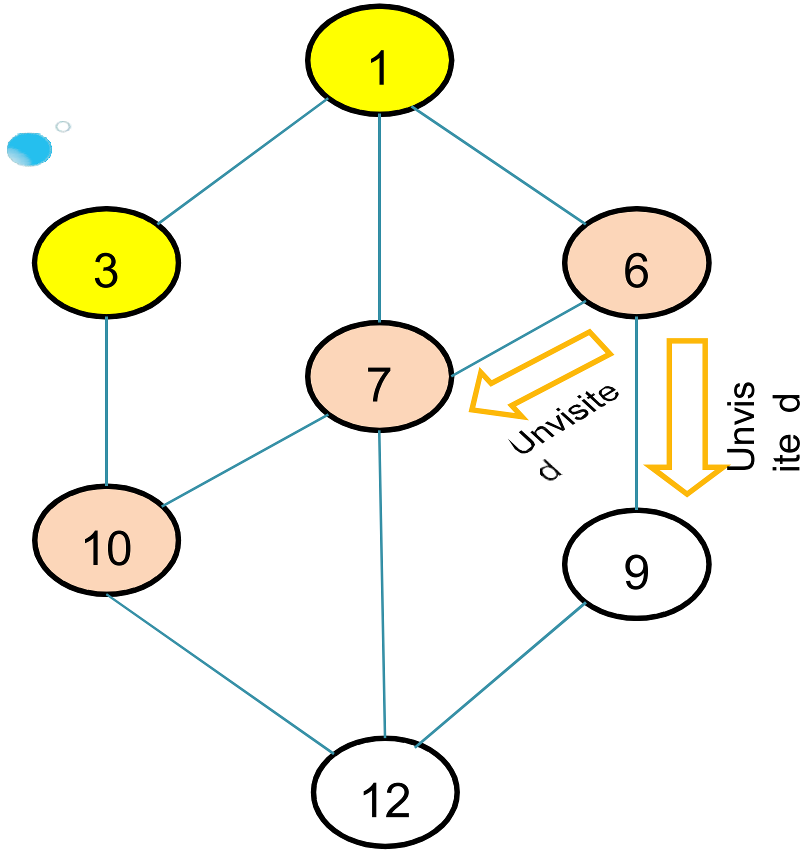

3
6
7
Queue

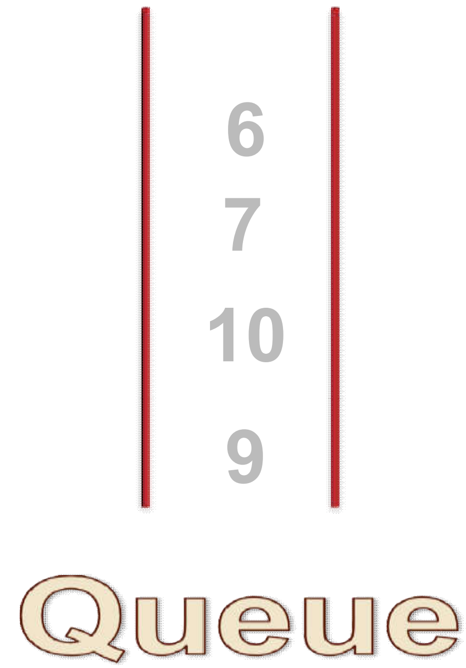
1


This is starting node
1
3
6
7
10
9
12
Unvisited
Unvisite
d
Unvisite
d
Unvisite
d

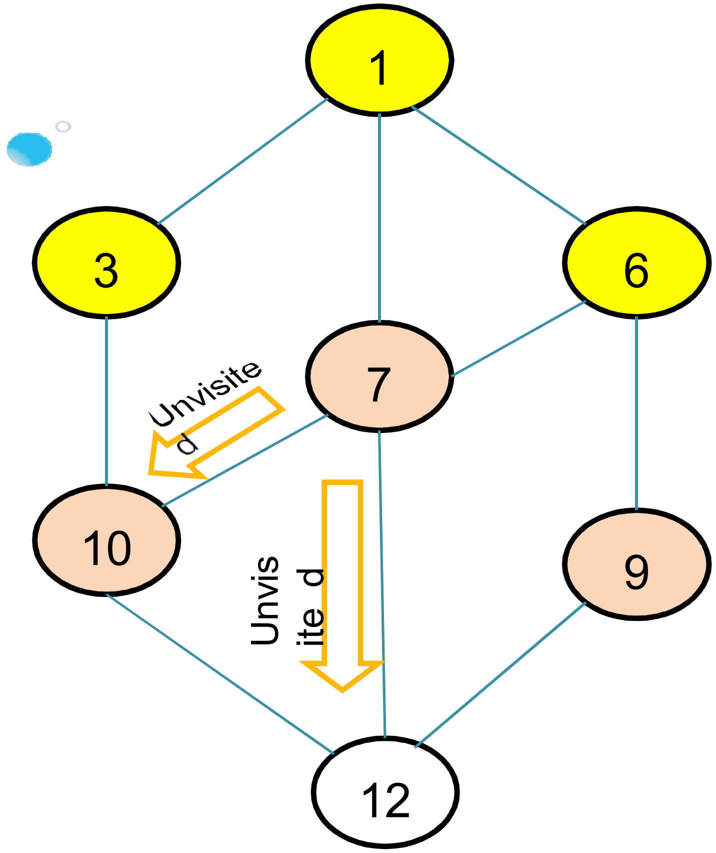

3
6
7
10
Queue

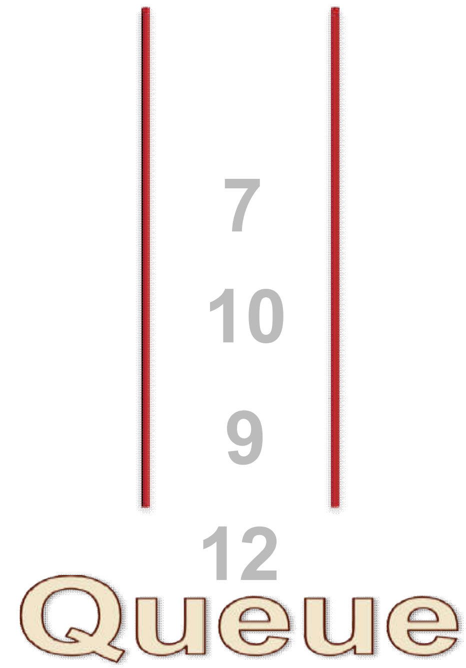
1 3



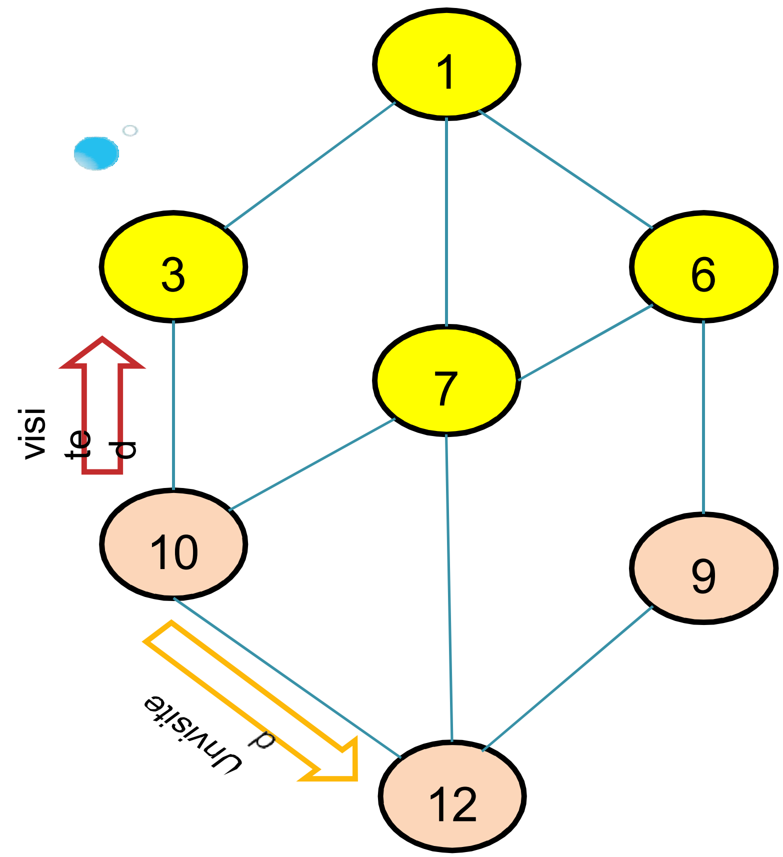


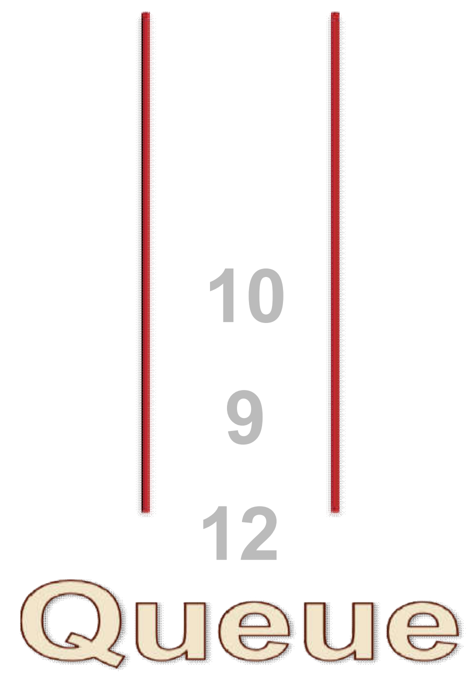
1 3 6



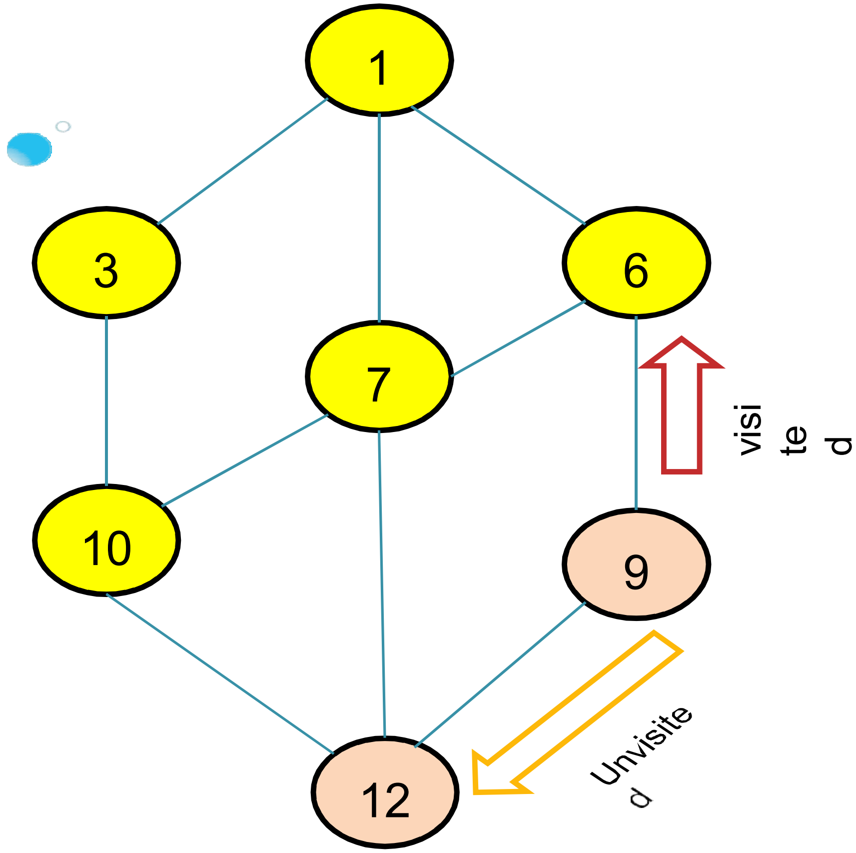


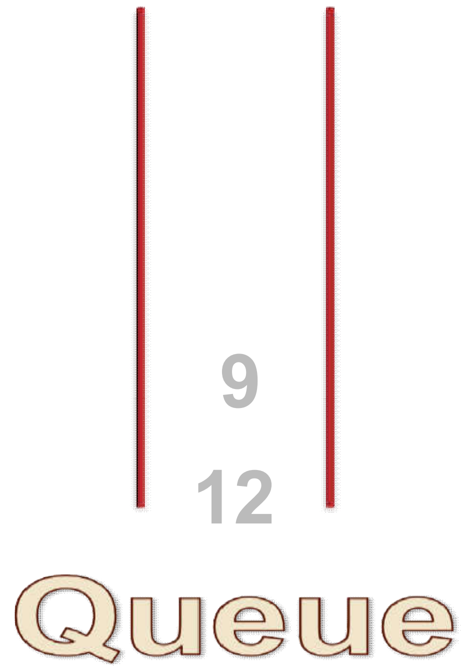
1 3 6 7



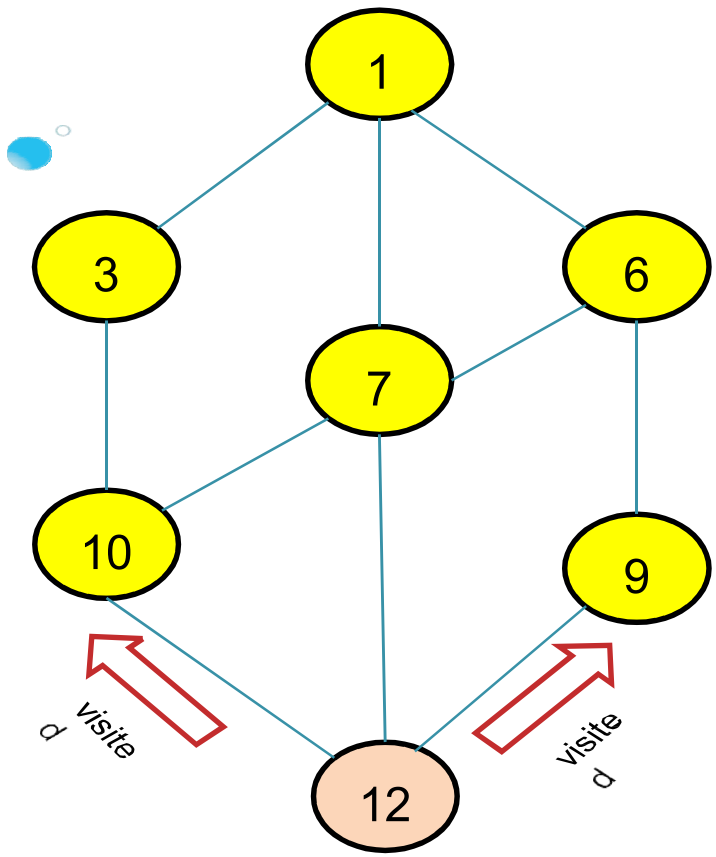


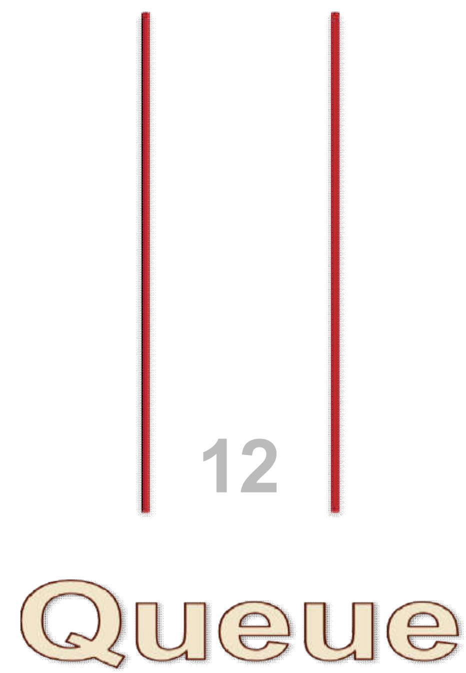
1 3 6 7 10





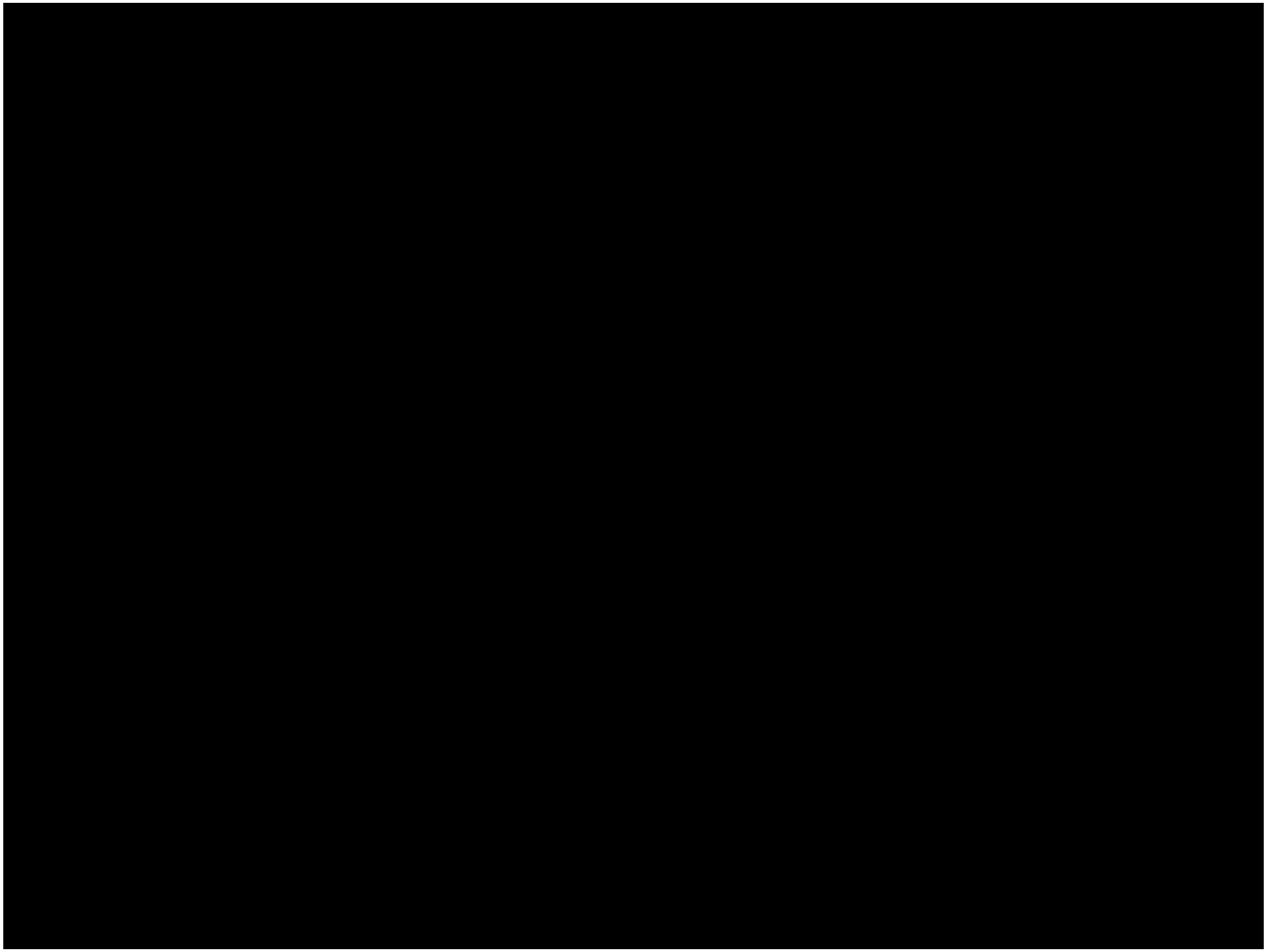
1 3 6 7 10 9





1 3 6 7 10 9 12

BFS



THANKS