Graphs

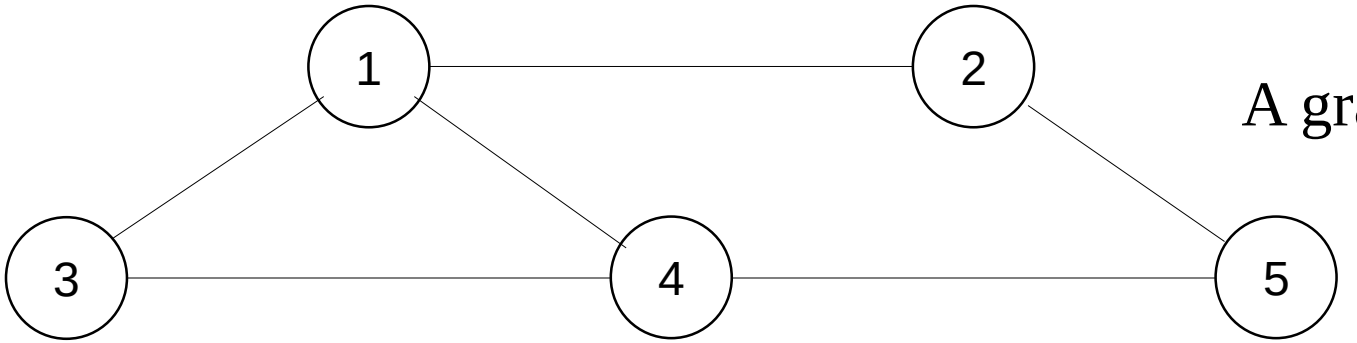
GRAPH – Definitions

- A **graph** G = (V, E) consists of
  - a set of ***vertices***, V, and
  - a set of ***edges***, E, where each edge is a pair (v,w) s.t. v,w ∈ V
- Vertices are sometimes called ***nodes.***
- ***The edges in a graph can be directed or undirected – the pair of vertices are ordered or unordered.***
- If the edge pair is ordered, then the graph is called a **directed graph (**also called *digraph)* .
- Normally a graph is an ***undirected graph***.

Graph – Definitions

- Two vertices of a graph are ***adjacent*** if they are connected by an edge.
- An edge (v,w) is
  - From v to w in case of digraph
  - Between v and w in case of graph
- A ***path*** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
  - i.e. $w_1, w_2, \ldots, w_N$ is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N\text{-}1$
- A ***simple path*** passes through a vertex only once.
- A ***cycle*** is a path that begins and ends at the same vertex.
- A ***simple cycle*** is a cycle that does not pass through other vertices more than once.

Graph – An Example



A graph G (undirected)

The graph G= (V,E) has 5 vertices and 6 edges:

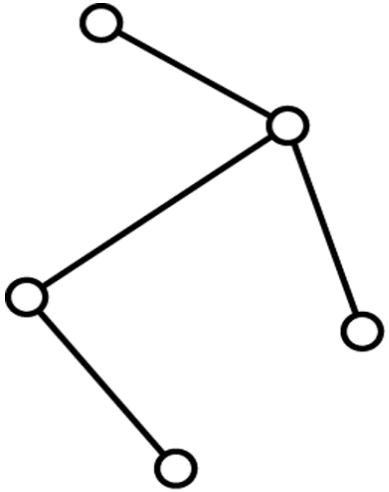
V = {1,2,3,4,5}

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

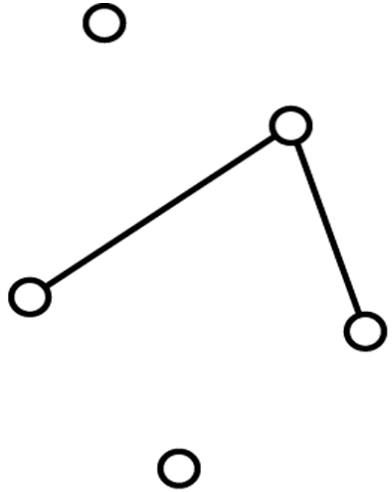
- *Adjacent:*
  1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1
- *Path:*
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)
- *Cycle:*
  1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)

Graph -- Definitions

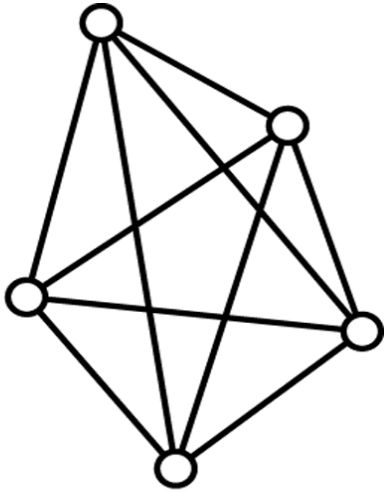
- A ***connected graph*** has a path between each pair of distinct vertices.
- A ***complete graph*** has an edge between each pair of distinct vertices.
  - A complete graph is also a connected graph. But a connected graph may not be a complete graph.



(a) **connected**



(b) **disconnected**



(c) **complete**

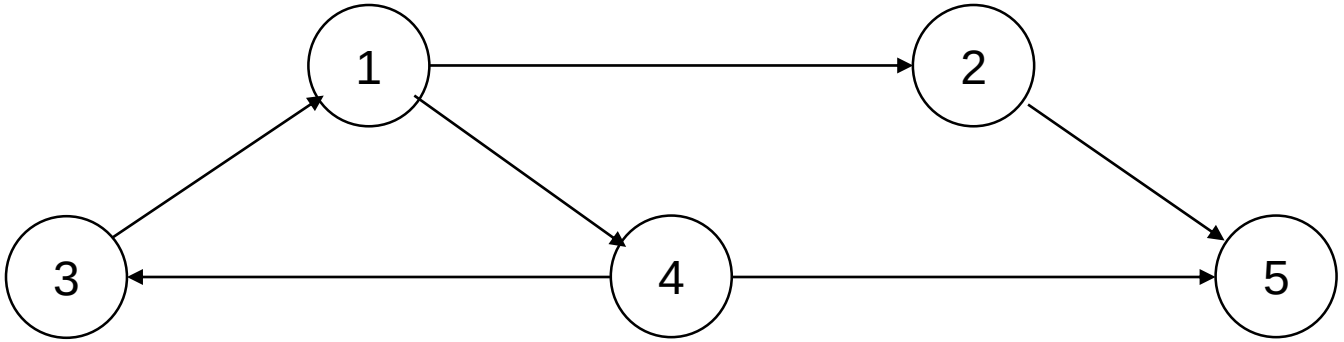
Directed Graphs

- If the edge pair is ordered then the graph is called a **directed graph (**also called *digraph)* .
- Each edge in a directed graph from a node to another or itself, and each edge is called a ***directed edge***.
- Vertex w is ***adjacent to*** v iff $(v,w) \in E$.
  - i.e. There is a direct edge from v to w
  - w is ***successor*** of v
  - v is ***predecessor*** of w
- A ***directed path*** between two vertices is a sequence of directed edges that begins at one vertex and ends at another vertex.
  - i.e. $w_1, w_2, \ldots, w_N$ is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N-1$

Directed Graphs

- A **cycle** in a directed graph is a path of length at least 1 such that $w_1 = w_N$.
  - This cycle is simple if the path is simple.
  - For undirected graphs, the edges must be distinct
- A **directed acyclic graph** (*DAG*) is a type of directed graph having no cycles.
- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
- A directed graph with this property is called **strongly connected**.
  - If a directed graph is not strongly connected, but the underlying graph (without considering direction of edges) is connected then the graph is **weakly connected**.

Directed Graph – An Example



The graph G= (V,E) has 5 vertices and 6 edges:

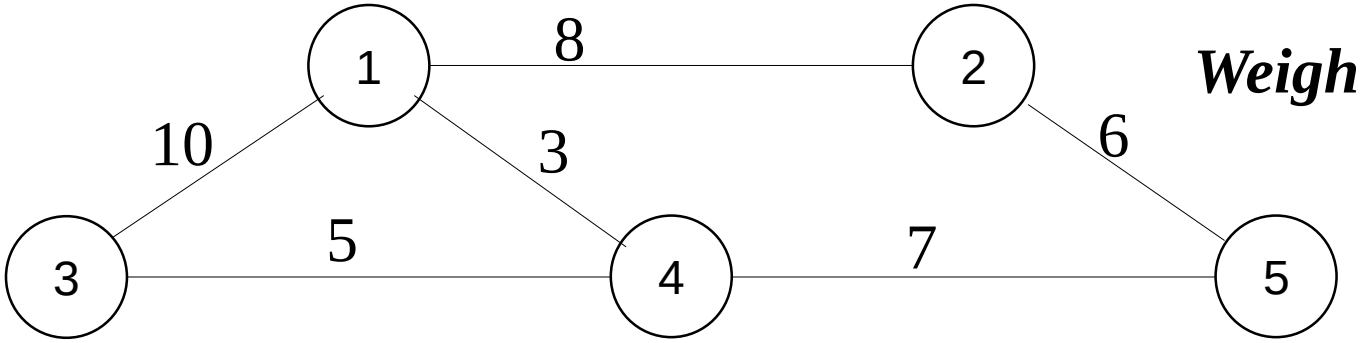
V = {1,2,3,4,5}

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

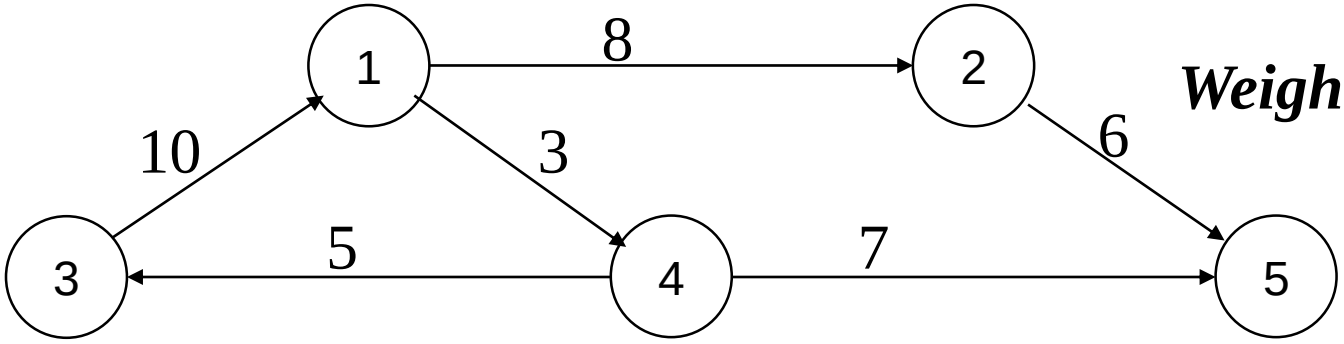
- *Adjacent:*
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*
  1,2,5 (a directed path),
- *Cycle:*
  1,4,3,1 (a directed cycle),

Weighted Graph

- We can label the edges of a graph with numeric values, the graph is called a ***weighted graph***.



Weighted (Undirected) Graph



Weighted Directed Graph

Graph Implementations

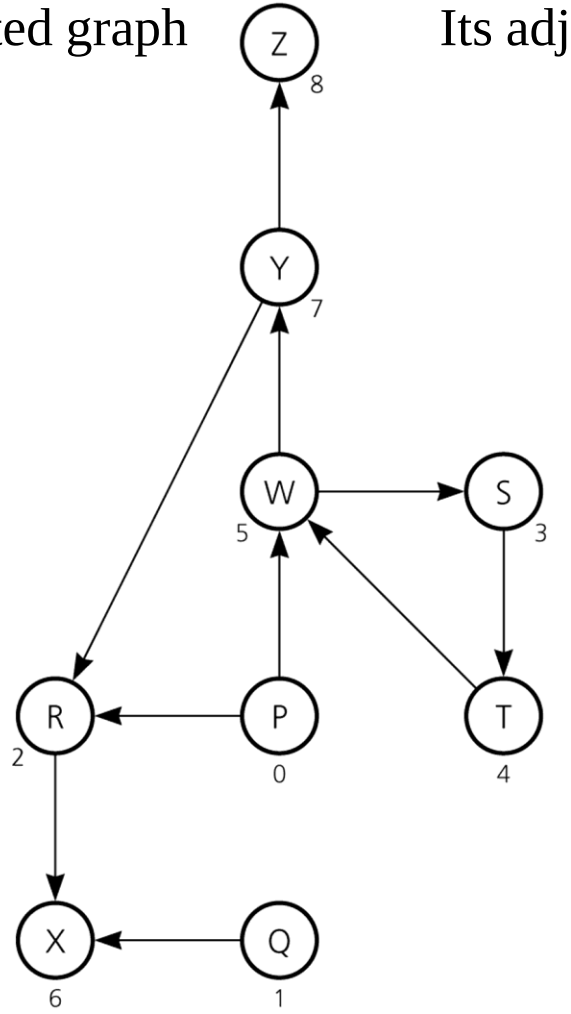
- The two most common implementations of a graph are:
  - ***Adjacency Matrix***
    - A two dimensional array
  - ***Adjacency List***
    - For each vertex we keep a list of adjacent vertices

Adjacency Matrix

- An ***adjacency matrix*** for a graph G with *n* vertices numbered 0,1,...,n-1 is defined as
  - G[i][j] = 1 if there is an edge between nodes i and j.
  - = 0 otherwise
- If a graph is *weighted*, then
  - G[i][j] is the weight when there is an edge between i and j.
  - is ∞ when there is no edge between i and j.
- Adjacency matrix of a graph (weighted or unweighted) is symmetric
- Adjacency matrix for a digraph can be defined in a similar manner.
- Space requirement $O(|V|^2)$

Adjacency Matrix – Example1

A directed graph

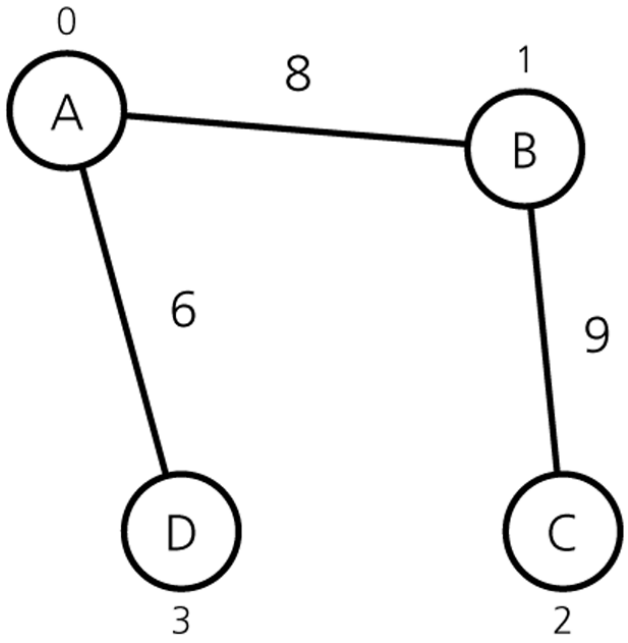


Its adjacency matrix

		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

Adjacency Matrix – Example2

An Undirected Weighted Graph



Its Adjacency Matrix

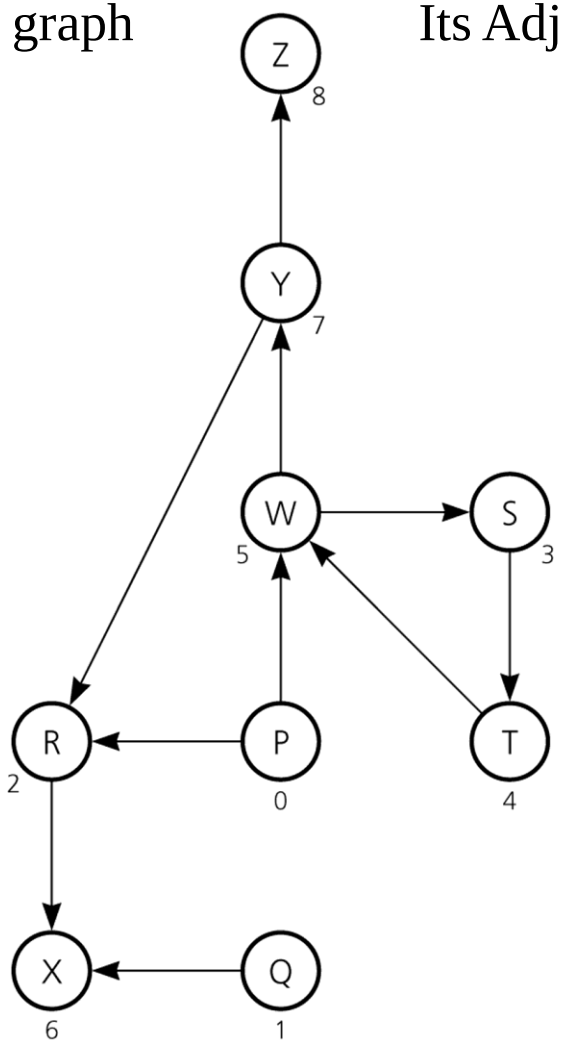
		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency List

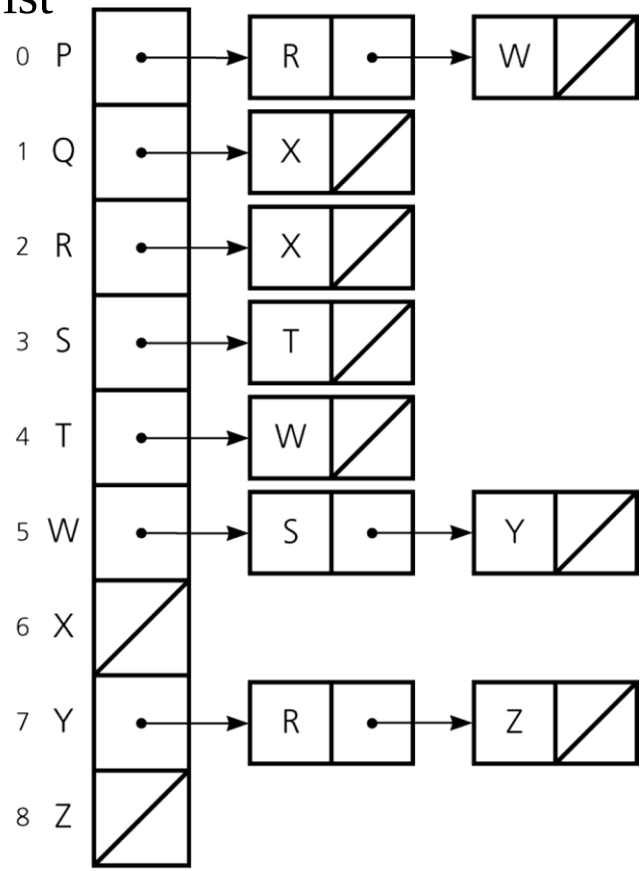
- An ***adjacency list*** for a graph with *n* vertices numbered 0,1,...,n-1 consists of *n* linked lists. The i^{th} linked list has a node for vertex *j* if and only if the graph contains an edge between vertices *i* and j (from vertex i to j in case of digraph).
- Adjacency list can be a better representation if the graph is sparse.
- Space requirement is O(|E| + |V|), which is linear in the size of the graph.
- In an undirected graph each edge (v, w) appears in two lists.
  - Space requirement is doubled.

Adjacency List – Example1

A directed graph

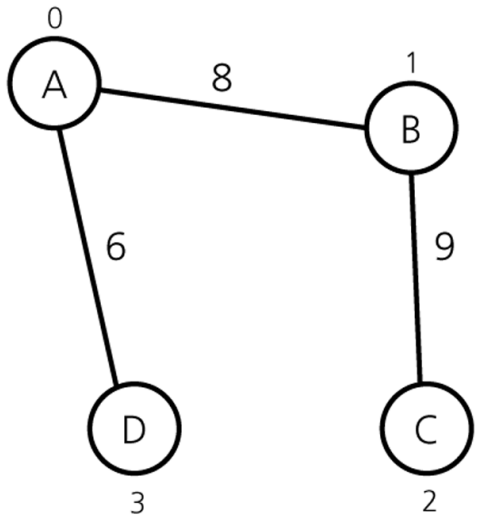


Its Adjacency List

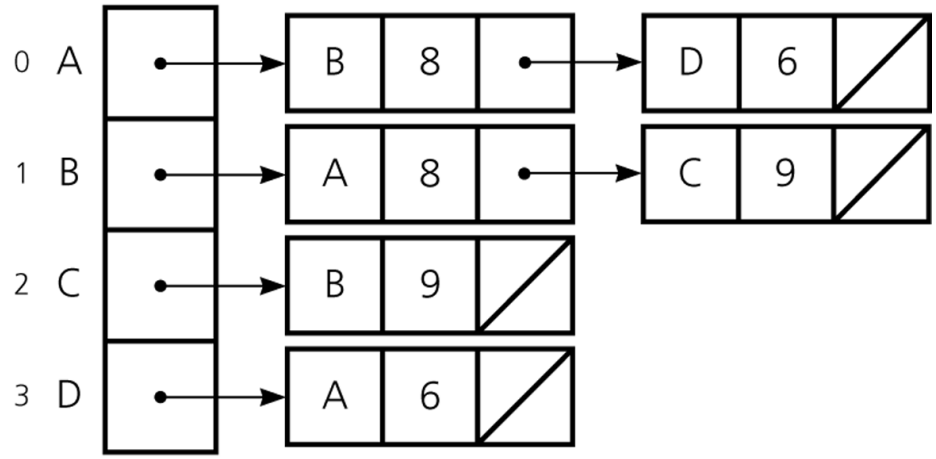


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List

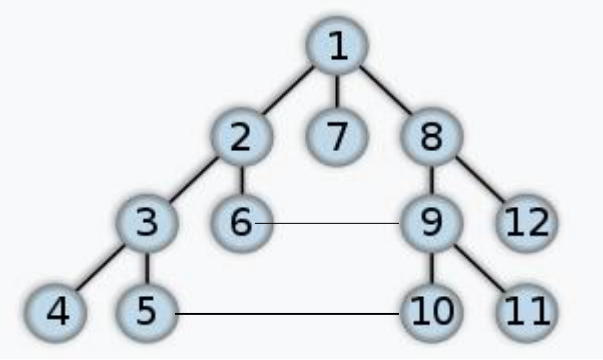


Graph Traversals

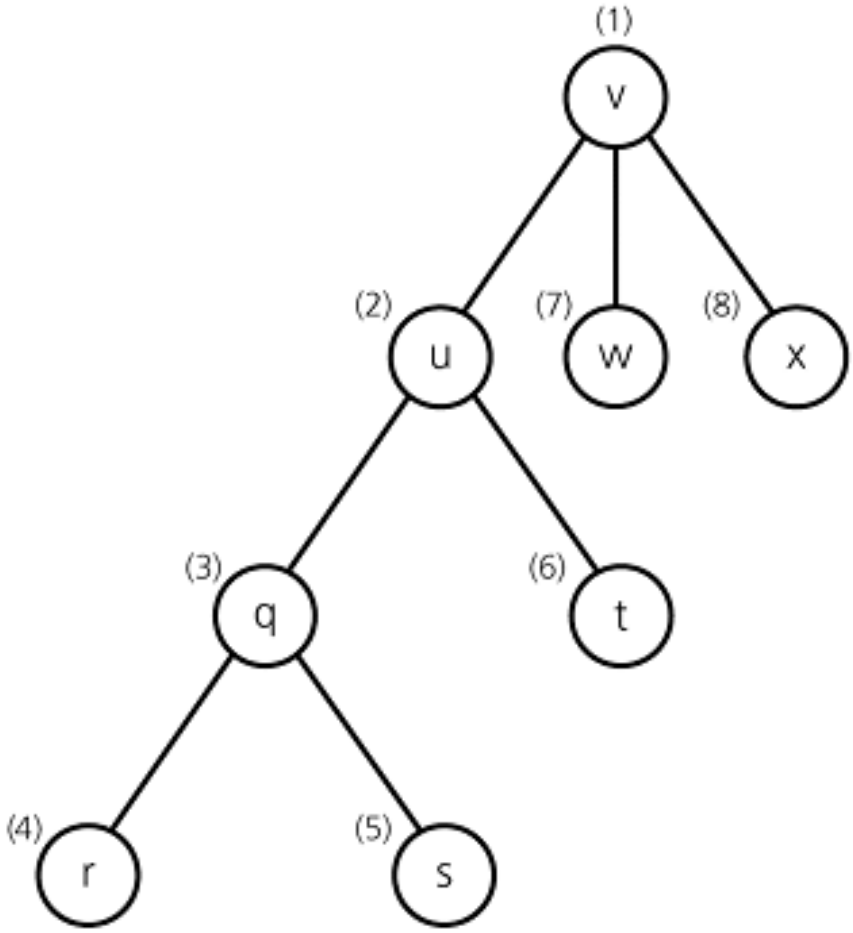
- A *graph-traversal* is to identify all vertices in the graph, that are reachable from a vertex v.
- All vertices of the graph can be identified by the graph traversal only when the graph is connected.
- A connected component is a subset of vertices that are reachable during traversal .
- A graph-traversal algorithm must mark each vertex during the visit and should not visit the same vertex again.
- Two standard graph-traversal algorithms are:
  - ***Depth-First Search (or Depth-First Traversal)***
  - ***Breadth-First Search (or Breadth-First Traversal)***

Depth-First Search

- For a given vertex v, the ***depth-first search*** algorithm proceeds along a path from v as deeply into the graph as possible before backing up.
- That is, after visiting a vertex v, the ***depth-first search*** algorithm visits an unvisited adjacent vertex(if exists) to the vertex v.
- In the graph, one possible order of visiting the nodes, starting from 1 can be 1, 2, 3, 4, 5, 10, 9, 11, 6, 8, 12,7
- There can be multiple such orders of visiting nodes in DFS.



Depth-First Search – Example

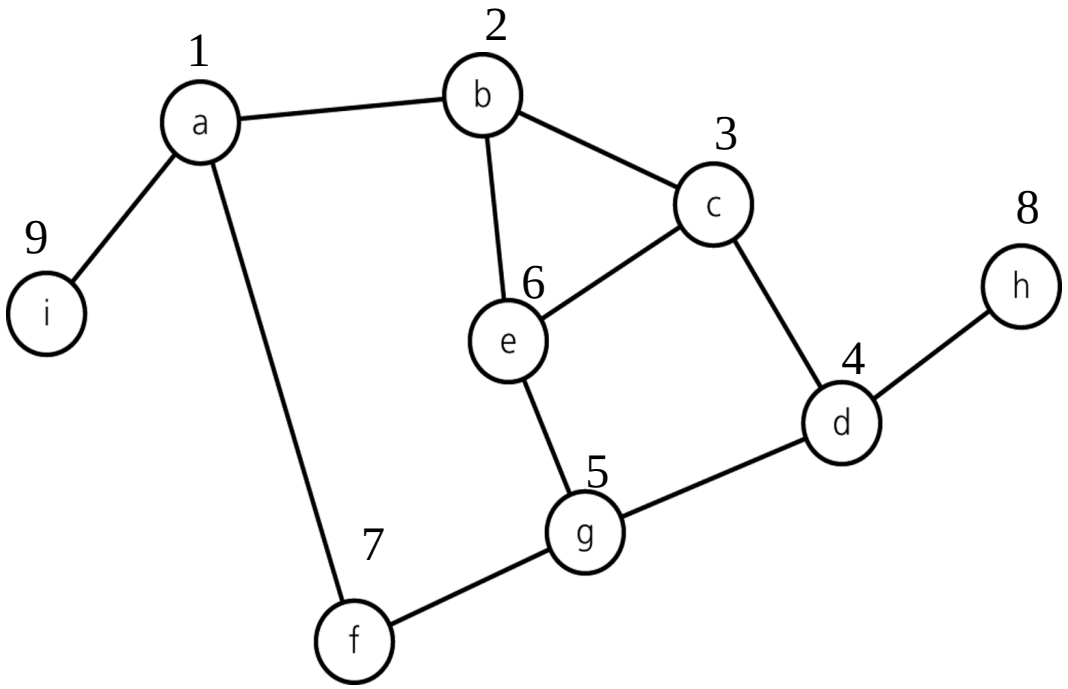


- A depth-first search of the graph starting from vertex v.
- Visit a vertex, then visit a vertex adjacent to that vertex.
- If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

Iterative Depth-First Search Algorithm

```
dfs( vertex v) {
  // Traverses a graph beginning at vertex v using DFS
  strategy
  s.createStack();
      s.push(v);  mark v;      // push v to s and mark it.
  while (!s.isEmpty()) {
        if (all vertices adjacent to the vertex
             on Top Of Stack are visited)
          s.pop();  // backtrack
        else {
           Let u be an unvisited vertex adjacent to the
           vertex on TOS;
          s.push(u); mark u;
       }
  }
}
```

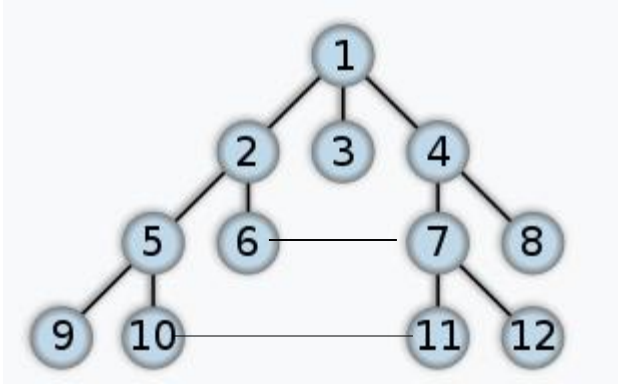
Trace of Iterative DFS – starting from vertex a



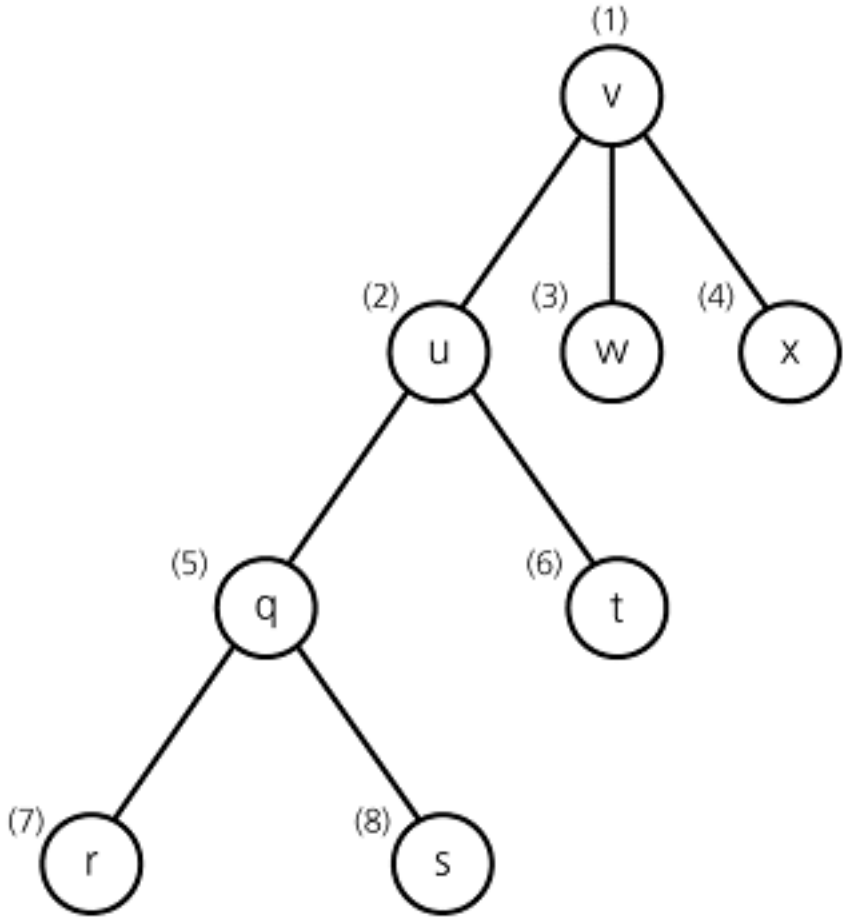
Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Breadth-First Search

- In BFS, after visiting a vertex v, all the nodes that are adjacent to it, before visiting any other node.
- The breadth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - One possible order in which the nodes can be visited in BFS, starting from node 1, can be 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12



Breadth-First Search– Example

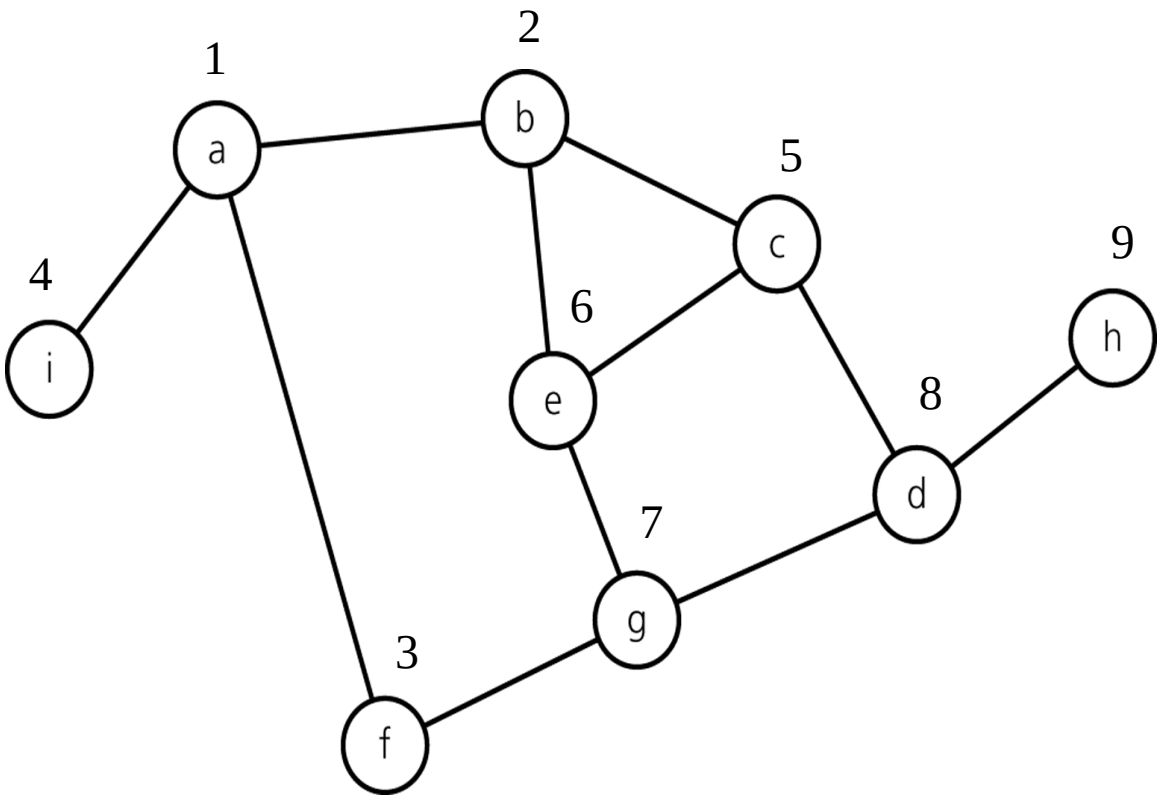


- A breadth-first search of the graph starting from vertex v.
- Visit a vertex, then visit all vertices adjacent to that vertex.

Iterative Breadth-First Search Algorithm

```
bfs(vertex v) {
// Traverses a graph beginning at vertex v
// by using breath-first strategy: Iterative Version
   q.createQueue();
   q.enqueue(v); // add v to the queue and mark it
   Mark v as visited;
   while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         q.enqueue(u);
      }
   }
}
```

Trace of Iterative BFT – starting from vertex a



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)