

Graphs

- Graphs (sometimes referred to as networks) offer a way of expressing relationships between pairs of items

What makes graphs so special?

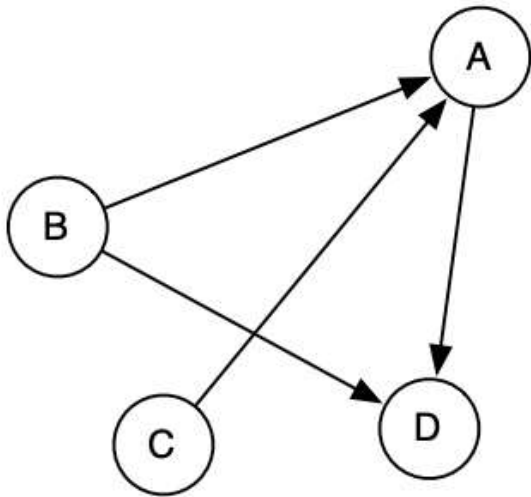
- What makes graphs special is that they represent relationships.
- relationships between things from the most abstract to the most concrete, e.g., mathematical objects, things, events, people are what makes everything interesting.

Trees captures relationships too, so why are graphs more interesting?

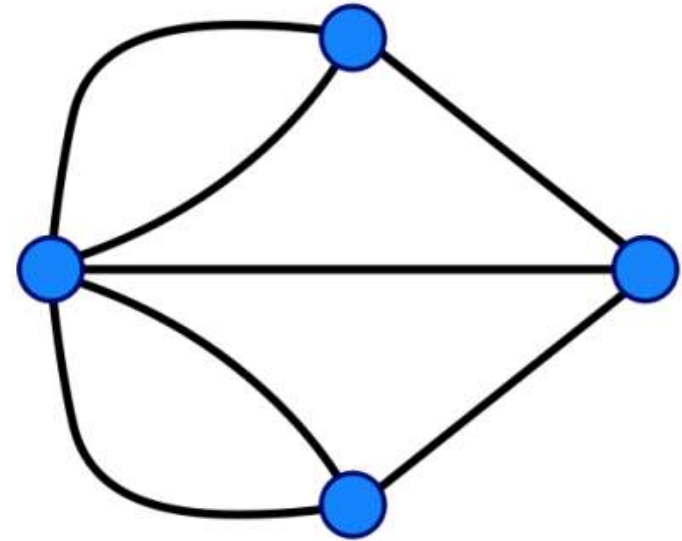
- Graphs are more interesting than other abstractions such as tree, which can also represent certain relationships,
- because graphs are more expressive.
- For example, in a tree, tree cannot be cycles, and multiple paths between two nodes.

GRAPHS – Definitions

- Graph is a data structure that consists of following two components:
 - A finite set of vertices also called as nodes.
 - A finite set of ordered pair of the form (u, v) called as edge.
- 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. $v, w \in V$
- Vertices are sometimes called *nodes*, edges are sometimes called *arcs*.
- If the edge pair is ordered then the graph is called a **directed graph** (also called *digraphs*) .
- We also call a normal graph (which is not a directed graph) an *undirected graph*.
 - When we say graph we mean that it is an undirected graph.



Directed Graph

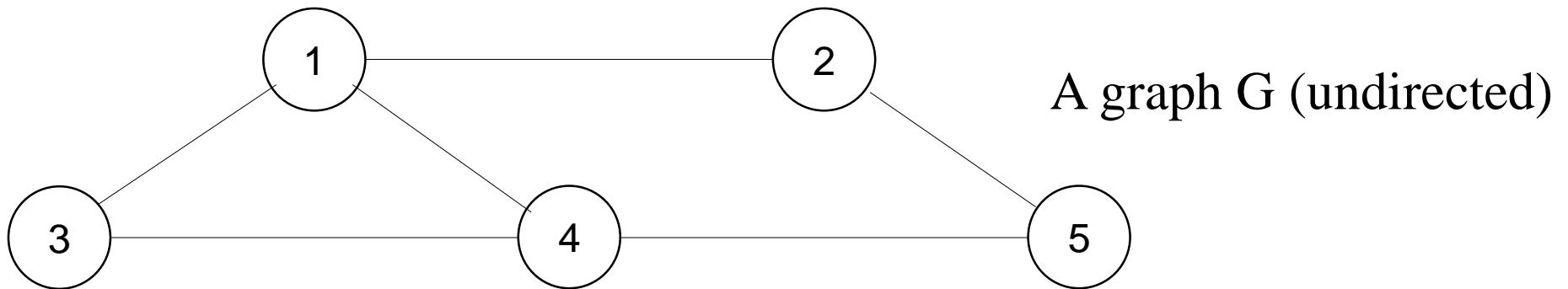


Undirected Graph

Graph – Terminology

- Two vertices of a graph are *adjacent* if they are joined by an edge.
- Vertex w is *adjacent to* v iff $(v,w) \in E$.
 - In an undirected graph with edge (v, w) and hence (w,v) w is adjacent to v and v is adjacent to w .
- 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, \dots, w_N is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N-1$
- 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.
- A *degree* of a vertex: the number of vertices adjacent to the vertex V .

Graph – An Example



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

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

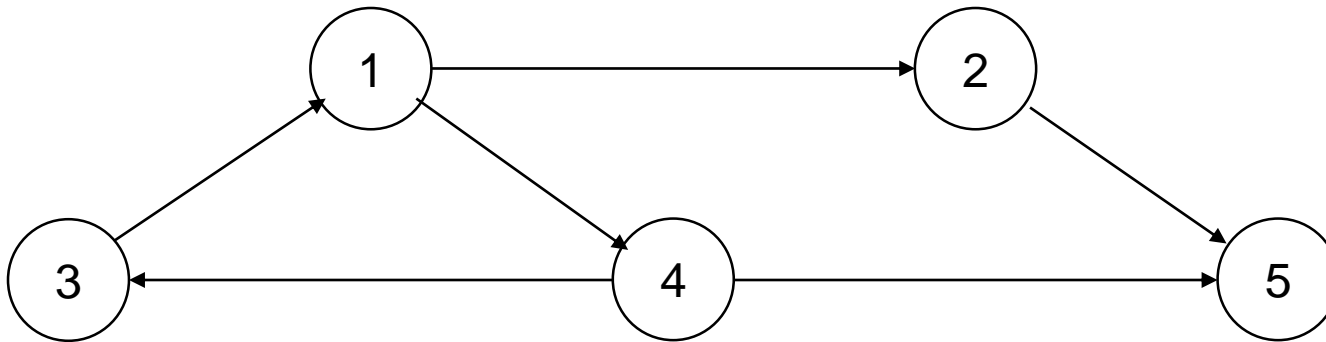
$$E = \{ (1,2), (1,3), (1,4), (2,5), (3,4), (4,5), (2,1), (3,1), (4,1), (5,2), (4,3), (5,4) \}$$

- *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)

Directed Graphs

- If the edge pair is ordered then the graph is called a **directed graph** (also called *digraphs*) .
- Each edge in a directed graph has a direction, and each edge is called a *directed edge*.
- Definitions given for undirected graphs apply also to directed graphs, with changes that account for direction.
- 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, \dots, w_N is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N-1$

Directed Graph – An Example



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

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

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

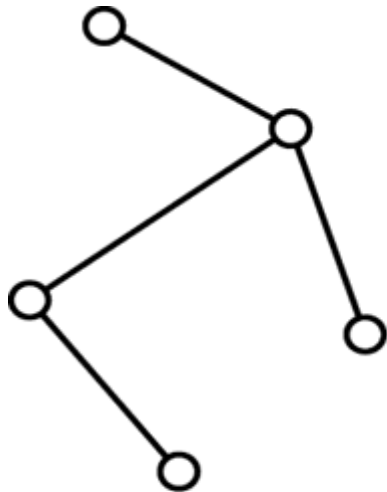
- *Adjacent:*
2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*
1, 2, 5 (a directed path),
- *Cycle:*
1, 4, 3, 1 (a directed cycle),

GRAPHS – Definitions

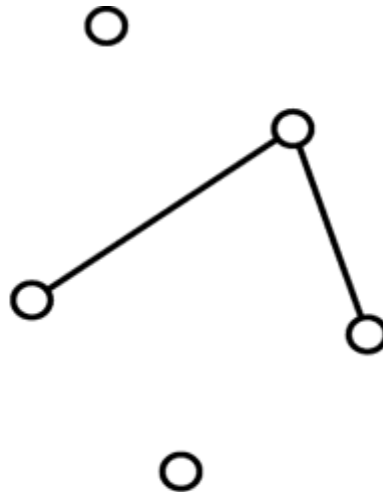
- Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#) for more applications of graph.

Graph -- Definitions

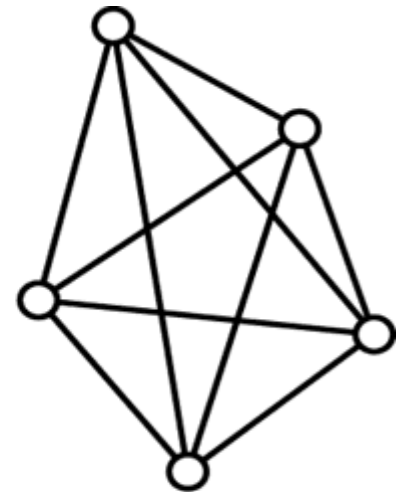
- A **connected graph** has a path between each pair of distinct vertices.
- A **complete graph** has an edge between each pair of distinct vertices.
 - A complete graph is also a connected graph. But a connected graph may not be a complete graph.



(a) **connected**



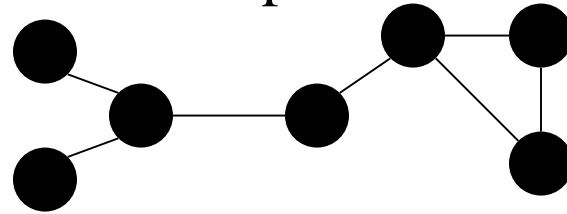
(b) **disconnected**



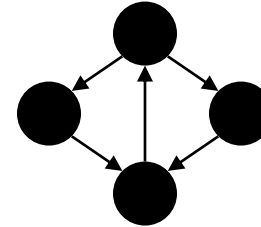
(c) **complete**

Connectivity

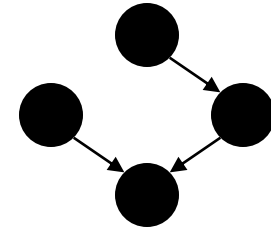
Undirected graphs are *connected* if there is a path between any two vertices



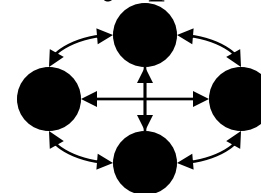
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



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

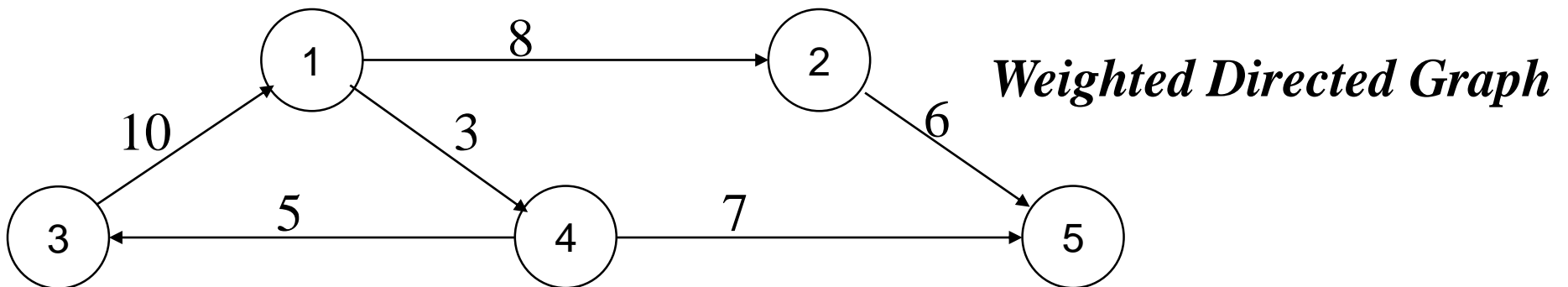
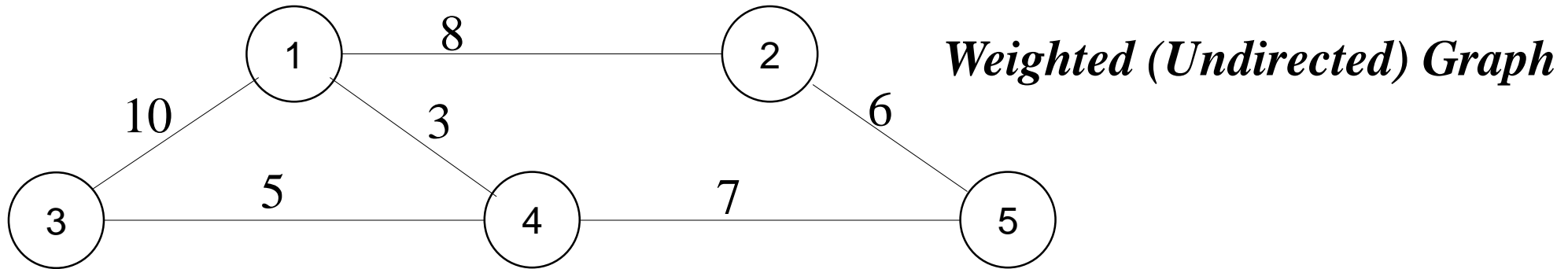


A *complete* graph has an edge between every pair of vertices



Weighted Graph

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



Applications of Graphs

- **Social network graphs:** to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom
- **Transportation networks.** In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them.
- **Utility graphs.** The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them
- **Document link graphs.** The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites
- ...

Graph Implementations

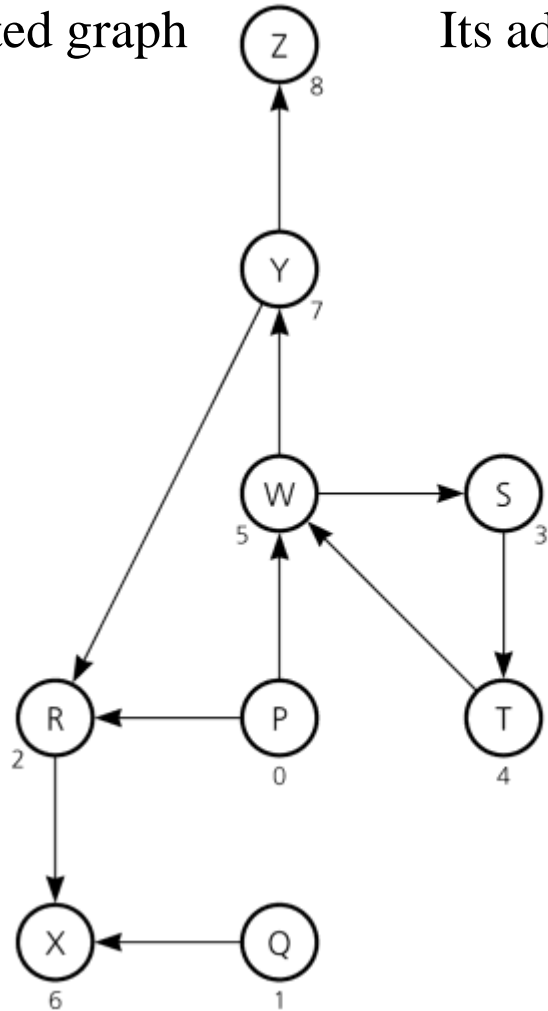
- The two most common implementations of a graph are:
 - *Adjacency Matrix*
 - A two dimensional array
 - *Adjacency List*
 - For each vertex we keep a list of adjacent vertices

Adjacency Matrix

- An *adjacency matrix* for a graph with n vertices numbered $0, 1, \dots, n-1$ is an n by n array *matrix* such that $matrix[i][j]$ is 1 (true) if there is an edge from vertex i to vertex j , and 0 (false) otherwise.
- When the graph is *weighted*, we can let $matrix[i][j]$ be the weight that labels the edge from vertex i to vertex j , instead of simply 1, and let $matrix[i][j]$ equal to ∞ instead of 0 when there is no edge from vertex i to vertex j .
- Adjacency matrix for an undirected graph is symmetrical.
 - i.e. $matrix[i][j]$ is equal to $matrix[j][i]$
- Space requirement $O(|V|^2)$
- Acceptable if the graph is dense.

Adjacency Matrix – Example1

A directed graph

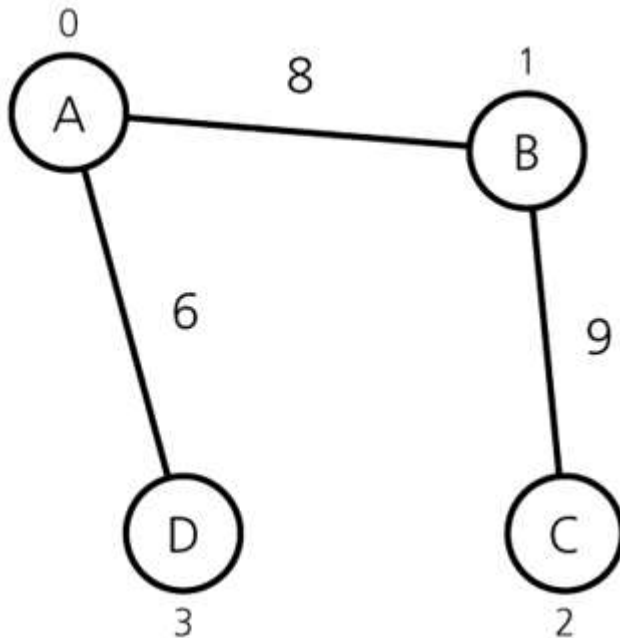


Its adjacency matrix

		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	∞	∞	∞

```

#include <iostream>

using namespace std;

class GraphMatrix{
    bool** adjMatrix;
    int vertices;
public:
    GraphMatrix(int vertices){
        this->vertices = vertices;
        adjMatrix = new bool*[vertices]; //array of arrays

        for(int i=0; i<vertices; i++){
            //each array element is an array
            //2D matrix
            adjMatrix[i] = new bool[vertices];

            //initialize each cell to false, not connected
            for(int j=0; j<vertices; j++){
                adjMatrix[i][j] = false; //i is row, j is column
            }
        }
    }
    void addEdge(int v, int w){
        adjMatrix[v][w] = true;
        adjMatrix[w][v] = true;
    }

    bool isAdjacent(int v, int w){
        return adjMatrix[v][w];
    }

    void printAdjacencyList(int v){
        cout<<v<<": ";
        for(int i=0; i<vertices; i++)
            if(adjMatrix[v][i])
                cout<<i<<" ";
    }
};

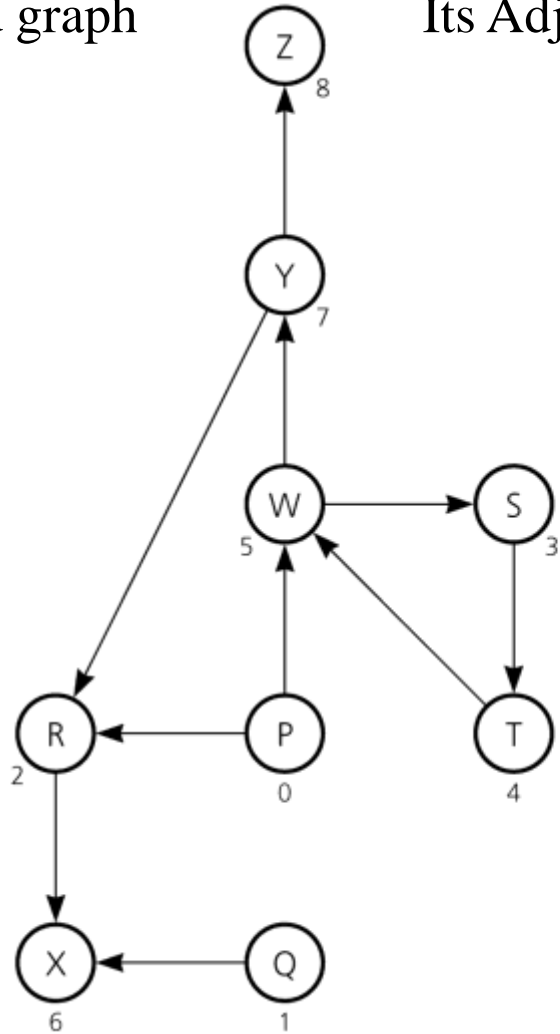
```

Adjacency List

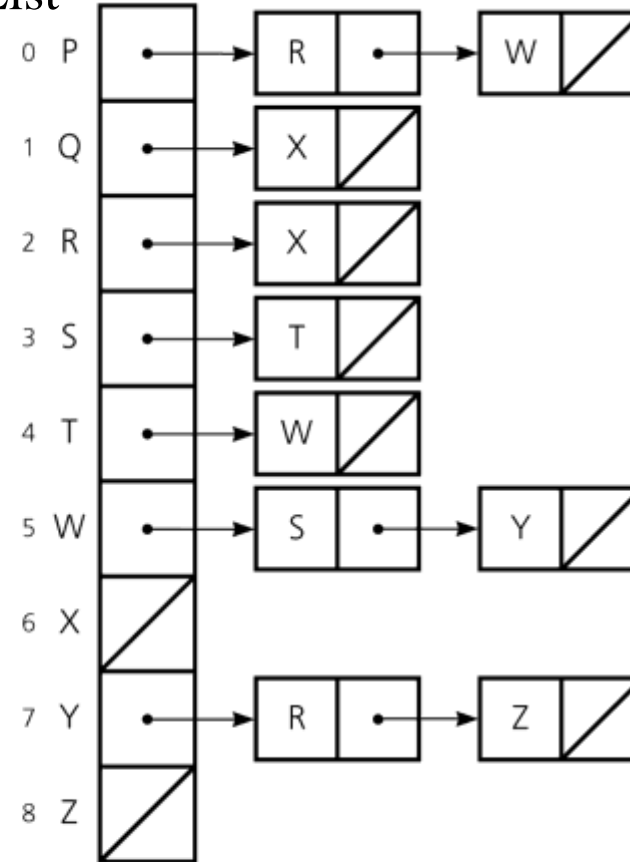
- An *adjacency list* for a graph with n vertices numbered $0, 1, \dots, n-1$ consists of n linked lists. The i^{th} linked list has a node for vertex j if and only if the graph contains an edge from vertex i to vertex j .
- Adjacency list is a better solution if the graph is sparse.
- Space requirement 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

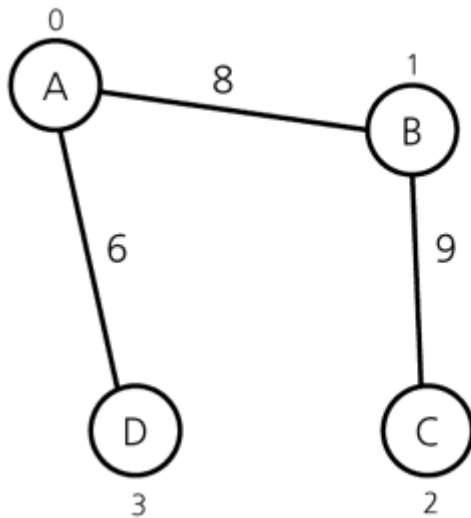


Its Adjacency List

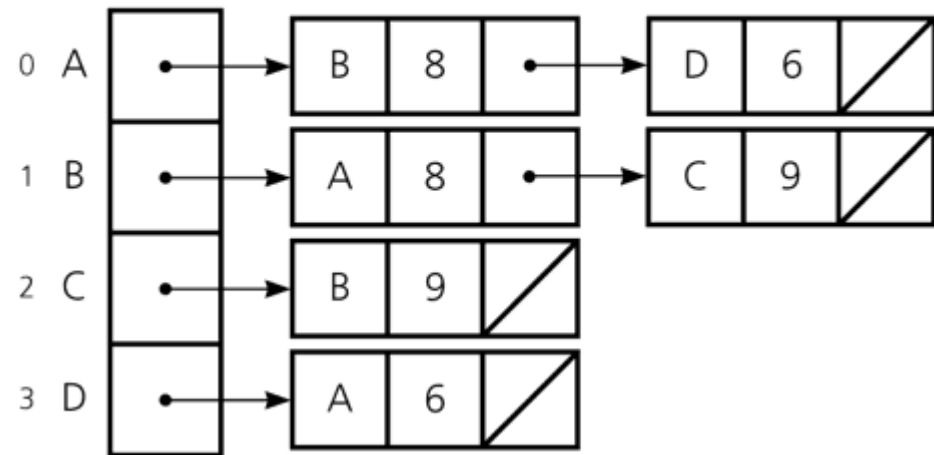


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List



```

#include "LinkedList.h"

class GraphList{
    LinkedList<int>* adjList;
    int vertices;

    public:
        GraphList(int vertices){
            this->vertices = vertices;

            adjList = new LinkedList<int>[vertices]; //array of linked lists
        }

        void addEdge(int v, int w){
            adjList[v].insert(w, adjList[v].zeroth());
            adjList[w].insert(v, adjList[w].zeroth());
        }
        bool isAdjacent(int v, int w){
            return adjList[v].find(w) != 0;
        }

        void printAdjacencyList(int v){
            cout<<v<<":";
            adjList[v].print();
        }

};

```

Adjacency Matrix vs Adjacency List

- Two common graph operations:
 1. Determine whether there is an edge from vertex i to vertex j .
 2. Find all vertices adjacent to a given vertex i .
- An adjacency matrix supports operation 1 more efficiently.
- An adjacency list supports operation 2 more efficiently.
- An adjacency list often requires less space than an adjacency matrix.
 - Adjacency Matrix: Space requirement is $O(|V|^2)$
 - Adjacency List : Space requirement is $O(|E| + |V|)$, which is linear in the size of the graph.
 - Adjacency matrix is better if the graph is dense (too many edges)
 - Adjacency list is better if the graph is sparse (few edges)

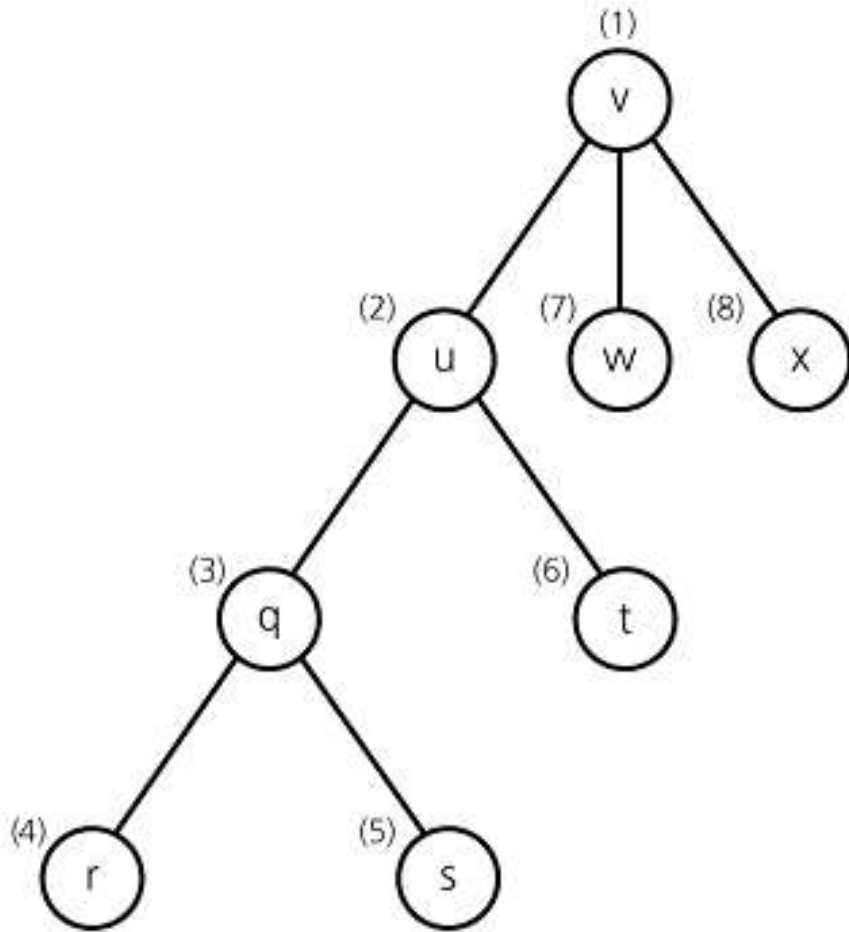
Graph Traversals

- A *graph-traversal* algorithm starts from a vertex v , visits all of the vertices that can be reachable from the vertex v .
- A graph-traversal algorithm visits all vertices if and only if the graph is connected.
- A connected component is the subset of vertices visited during a traversal algorithm that begins at a given vertex.
- A graph-traversal algorithm must mark each vertex during a visit and must never visit a vertex more than once.
 - Thus, if a graph contains a cycle, the graph-traversal algorithm can avoid infinite loop.
- We look at two graph-traversal algorithms:
 - *Depth-First Traversal*
 - *Breadth-First Traversal*

Depth-First Traversal

- For a given vertex v , the *depth-first traversal* 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 traversal* algorithm visits (if possible) an unvisited adjacent vertex to vertex v .
- The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v .
 - We may visit the vertices adjacent to v in sorted order.

Depth-First Traversal – Example



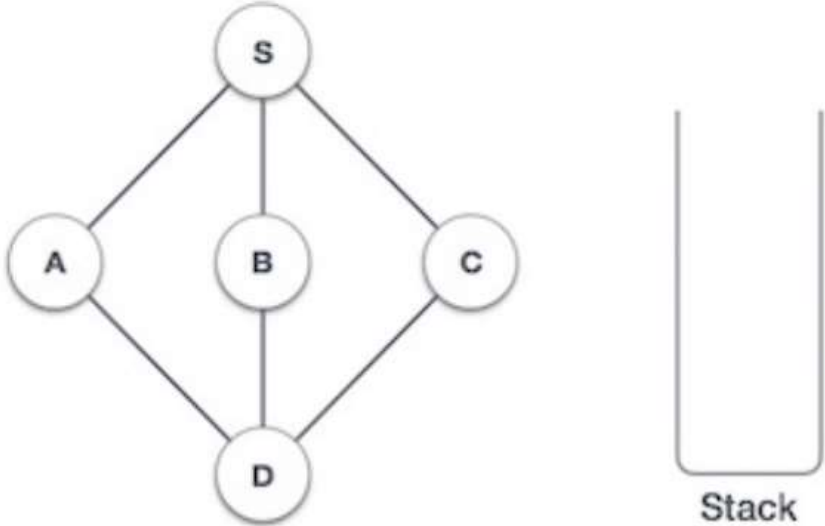
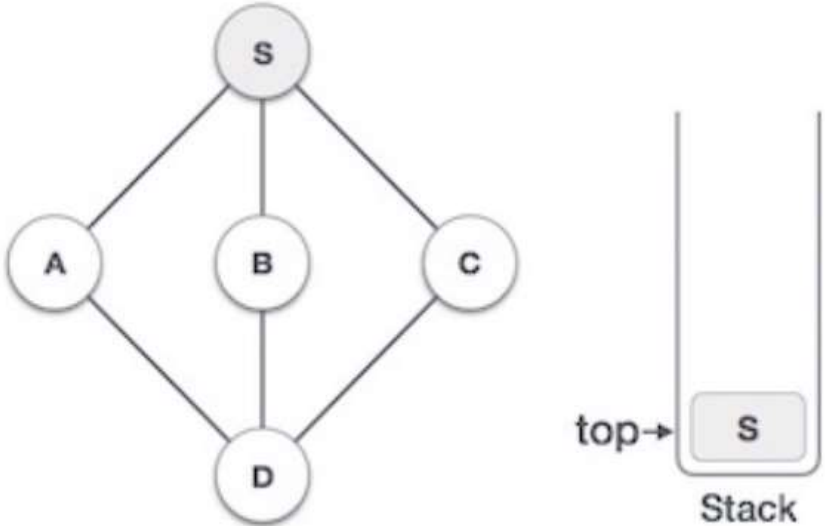
- A depth-first traversal of the graph starting from vertex v.
- Visit a vertex, then visit a vertex adjacent to that vertex.
- If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

Recursive Depth-First Traversal Algorithm

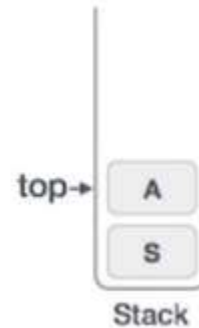
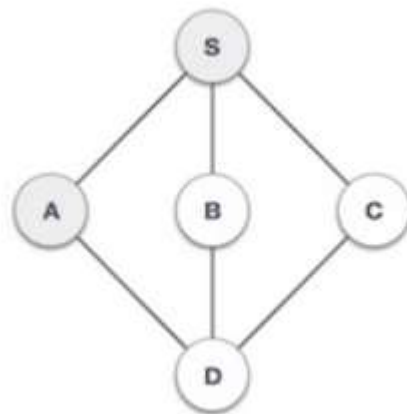
```
dft(in v:Vertex) {  
    // Traverses a graph beginning at vertex v  
    // by using depth-first strategy  
    // Recursive Version  
    Mark v as visited;  
    for (each unvisited vertex u adjacent to v)  
        dft(u)  
}
```

Iterative Depth-First Traversal Algorithm

```
dft(in v:Vertex) {  
    // Traverses a graph beginning at vertex v  
    // by using depth-first strategy: Iterative Version  
    s.createStack();  
    // push v into the stack and mark it  
    s.push(v);  
    Mark v as visited;  
    while (!s.isEmpty()) {  
        if (no unvisited vertices are adjacent to the vertex on  
            the top of stack)  
            s.pop(); // backtrack  
        else {  
            Select an unvisited vertex u adjacent to the vertex  
                on the top of the stack;  
            s.push(u);  
            Mark u as visited;  
        }  
    }  
}
```

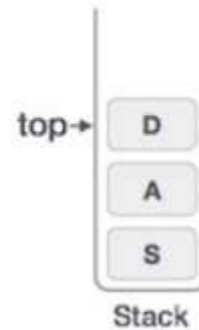
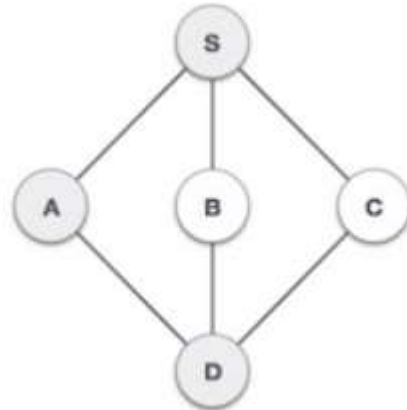
Step	Traversal	Description
1.		Initialize the stack.
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3.



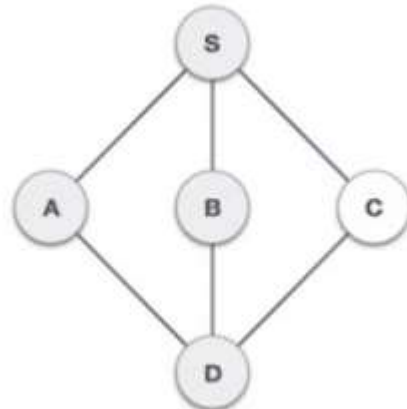
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4.



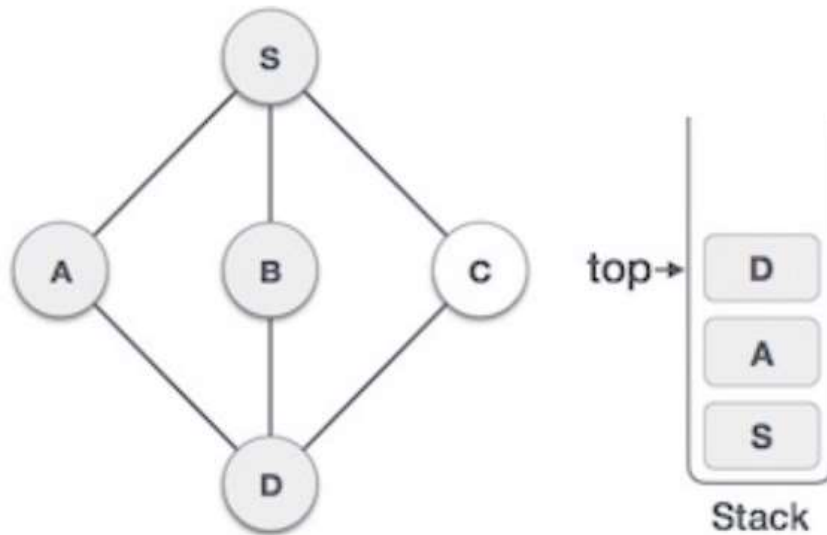
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5.



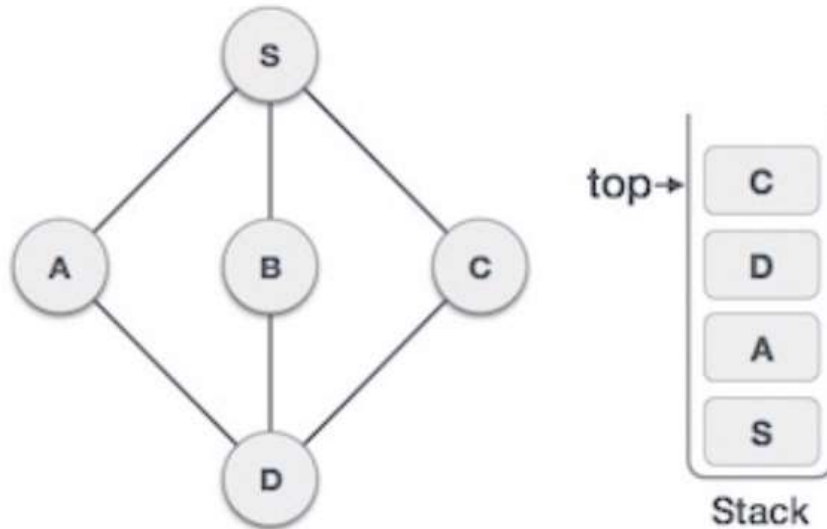
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6.



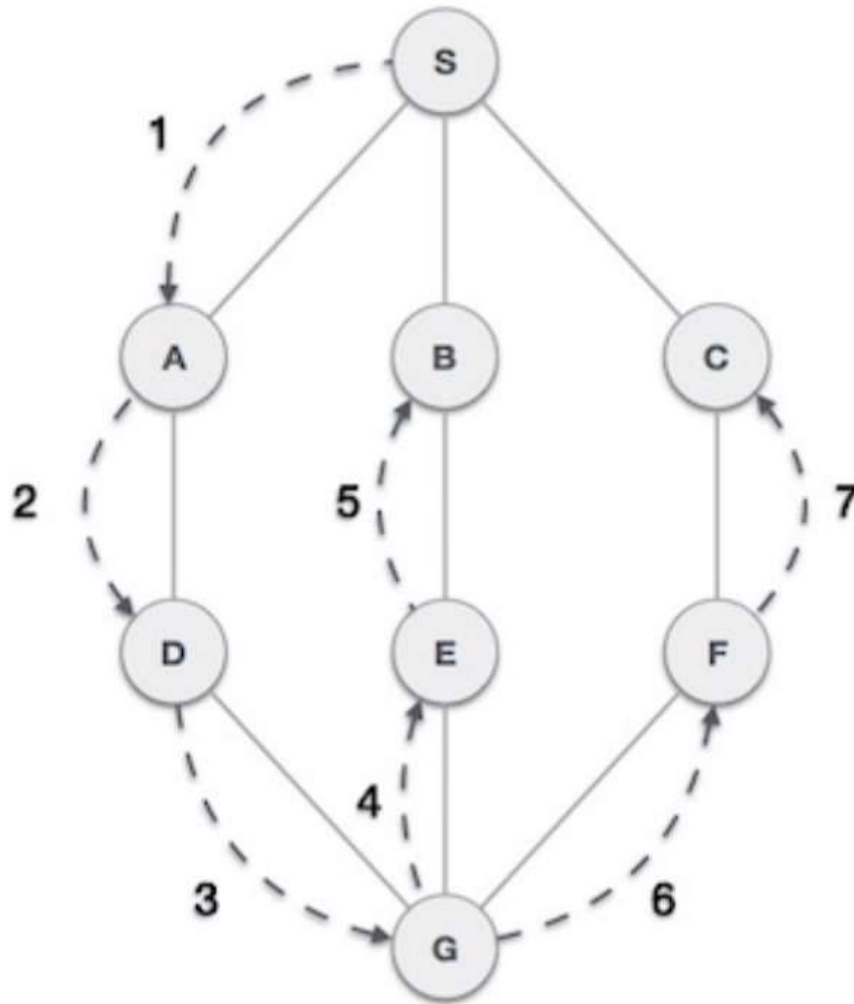
We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7.

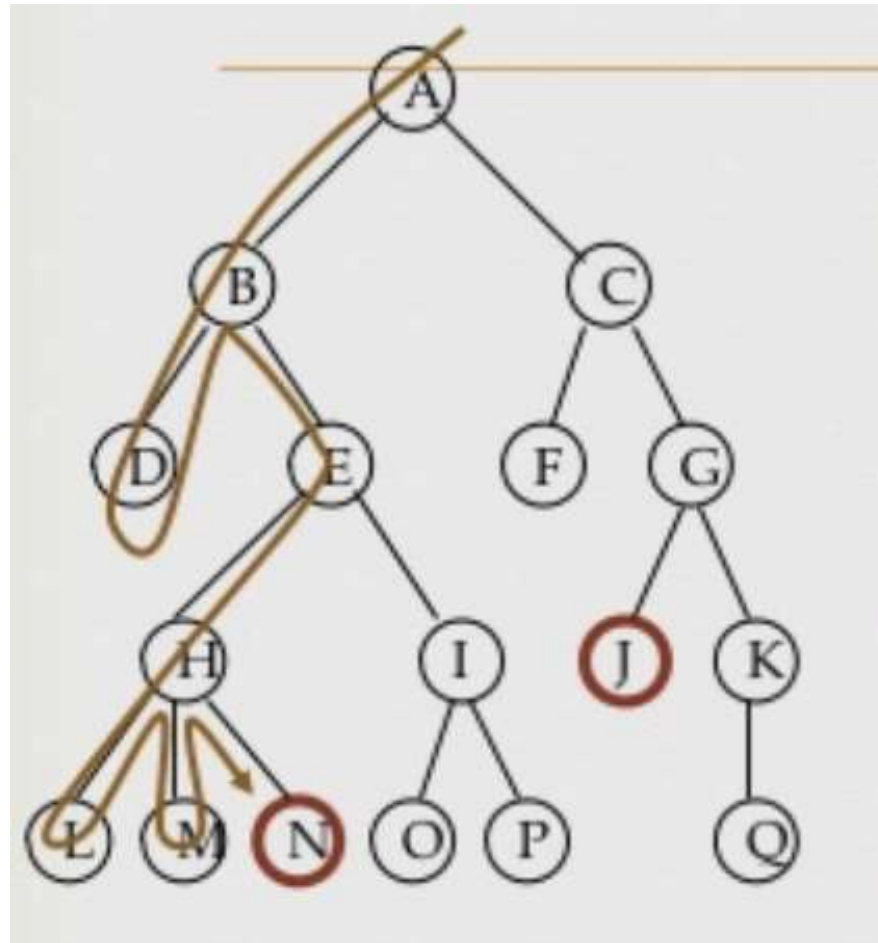


Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Example:

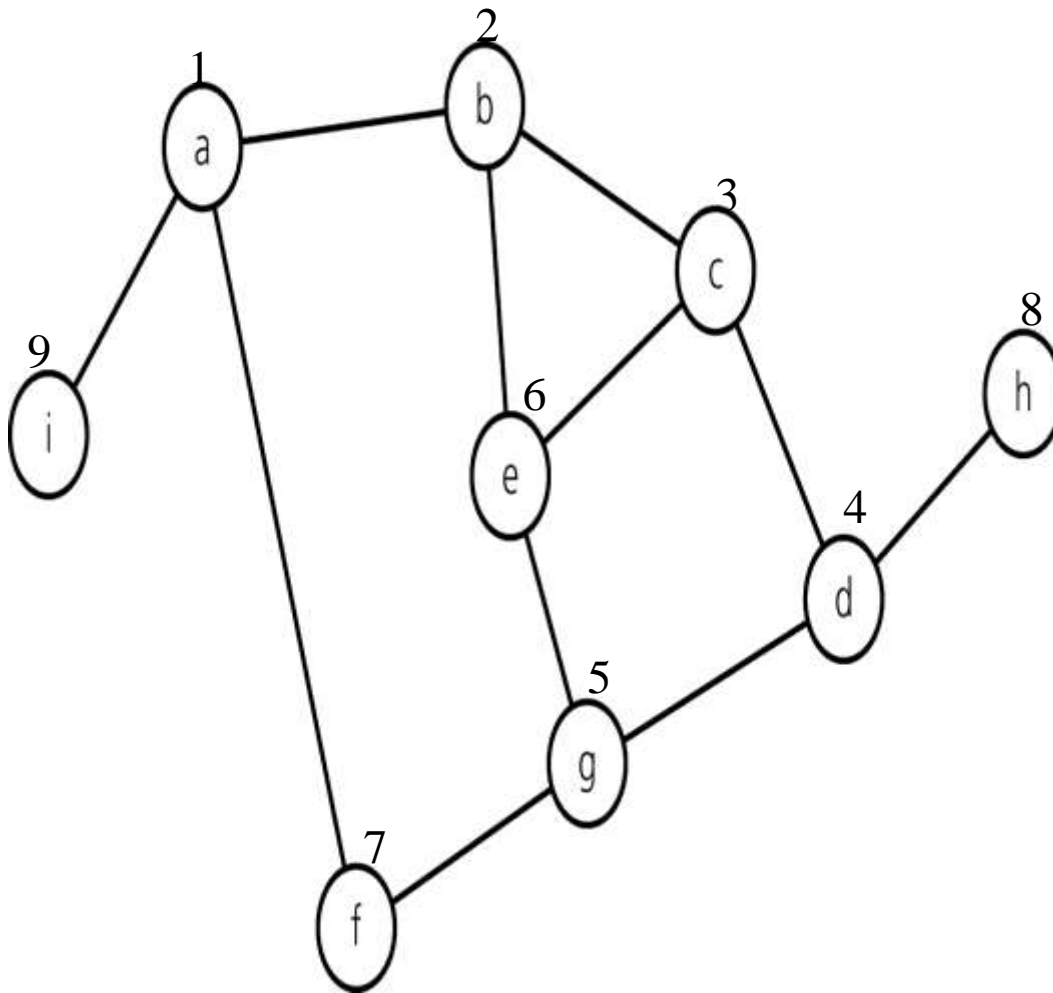


Example-1



A, B, D, E, H, L, M, N, I, O, P, C, F, G, J, K, Q

Trace of Iterative DFT – starting from vertex a

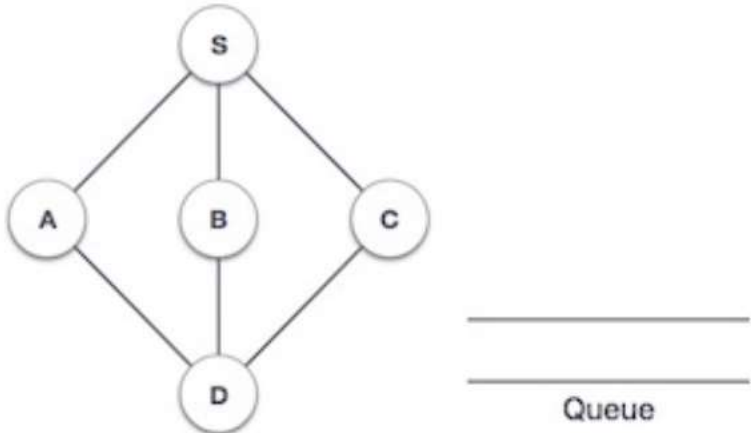
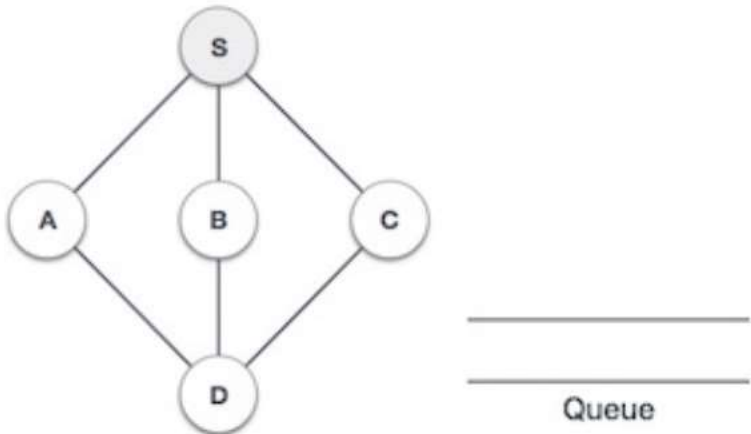


<u>Node visited</u>	<u>Stack (bottom to top)</u>
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	(empty)

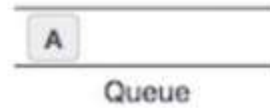
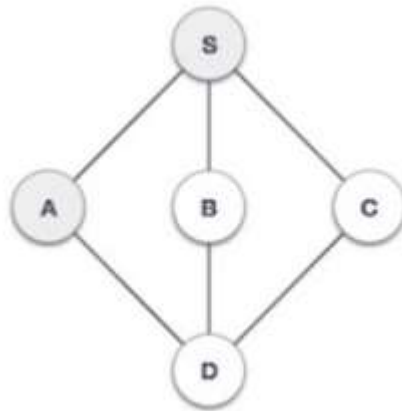
Breath-First Traversal

- After visiting a given vertex v , the breadth-first traversal algorithm visits every vertex adjacent to v that it can before visiting any other vertex.
- The breath-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v .
 - We may visit the vertices adjacent to v in sorted order.

Breath-First Traversal – Example

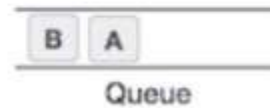
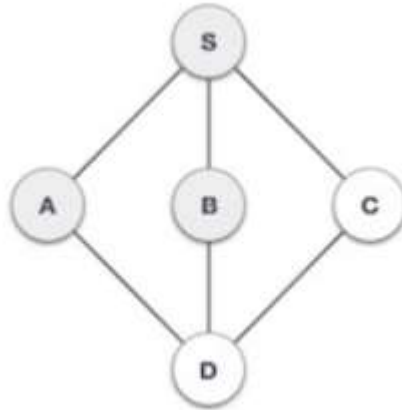
Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting S (starting node), and mark it as visited.

3.



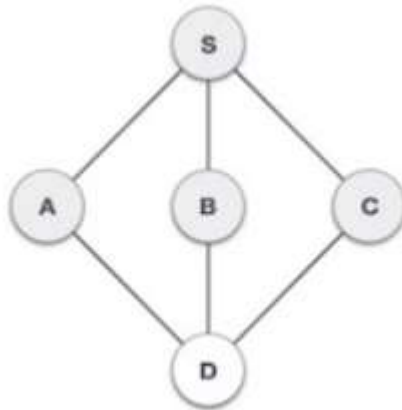
We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4.



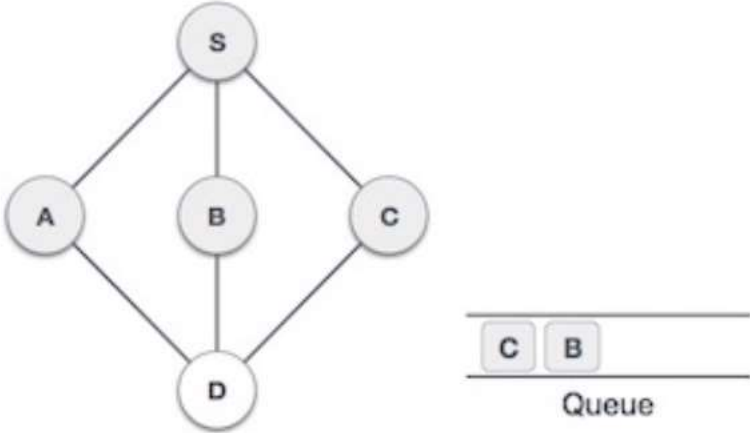
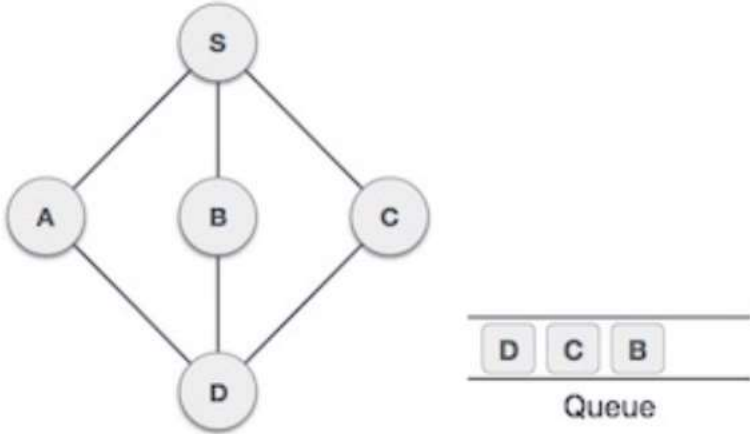
Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5.

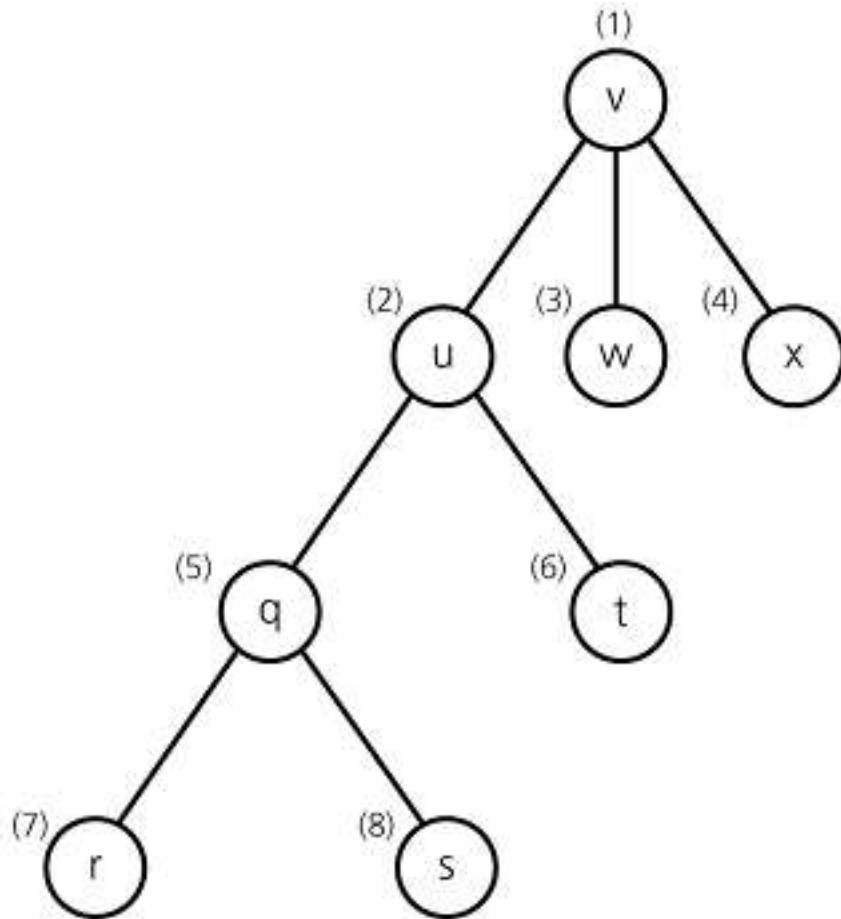


Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

Breadth-First Traversal – Example

6.	 <p>The graph has five nodes: S (top), A (middle-left), B (middle-center), C (middle-right), and D (bottom). Edges connect S to A, B, and C; A to D; B to D; and C to D. A queue is shown below the graph containing nodes C and B, with C at the front.</p>	Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A .
7.	 <p>The graph is the same as in step 6. The queue now contains nodes D, C, and B, with D at the front.</p>	From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

Breadth-First Traversal – Example

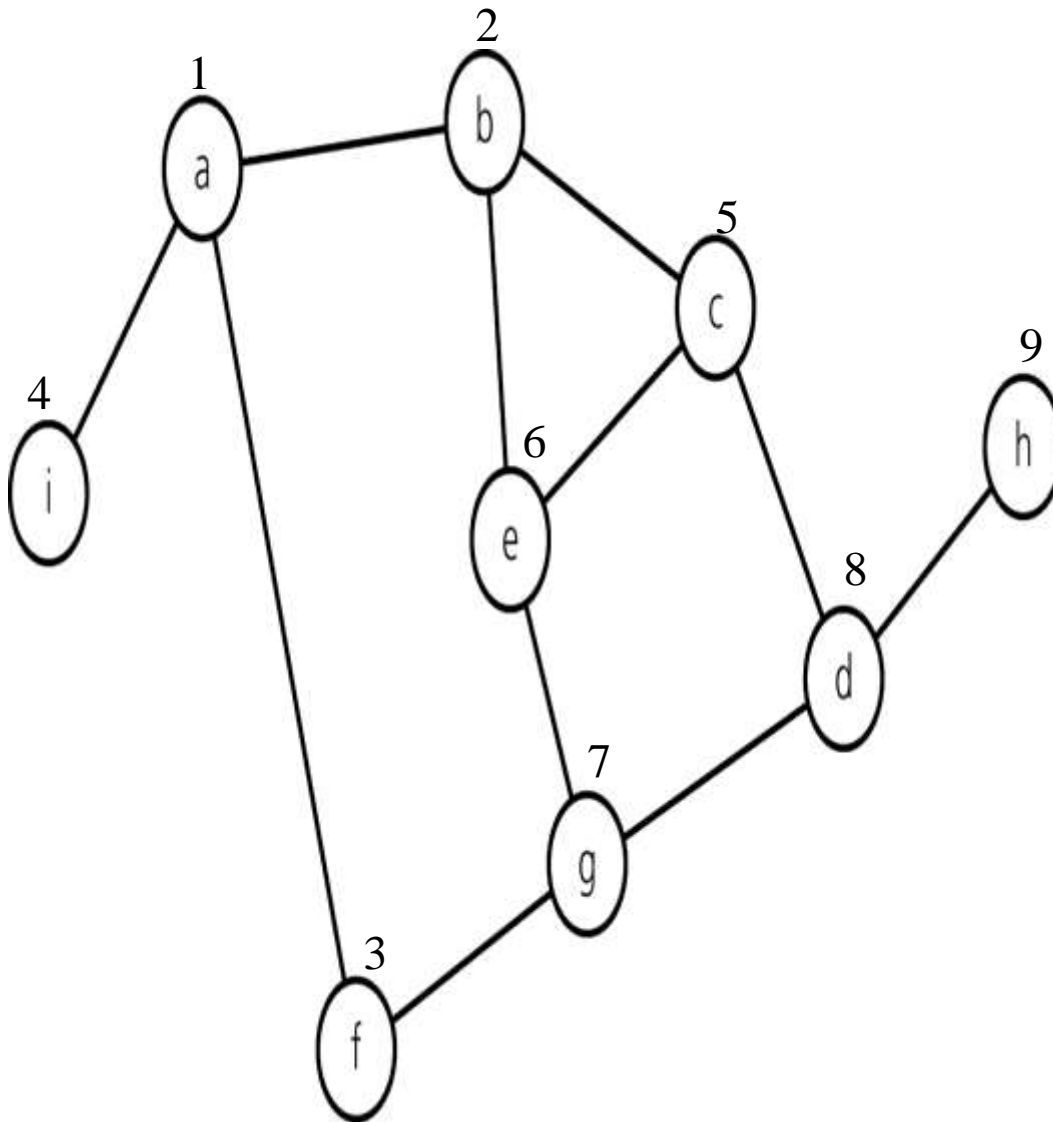


- A breadth-first traversal of the graph starting from vertex v.
- Visit a vertex, then visit all vertices adjacent to that vertex.

Iterative Breath-First Traversal Algorithm

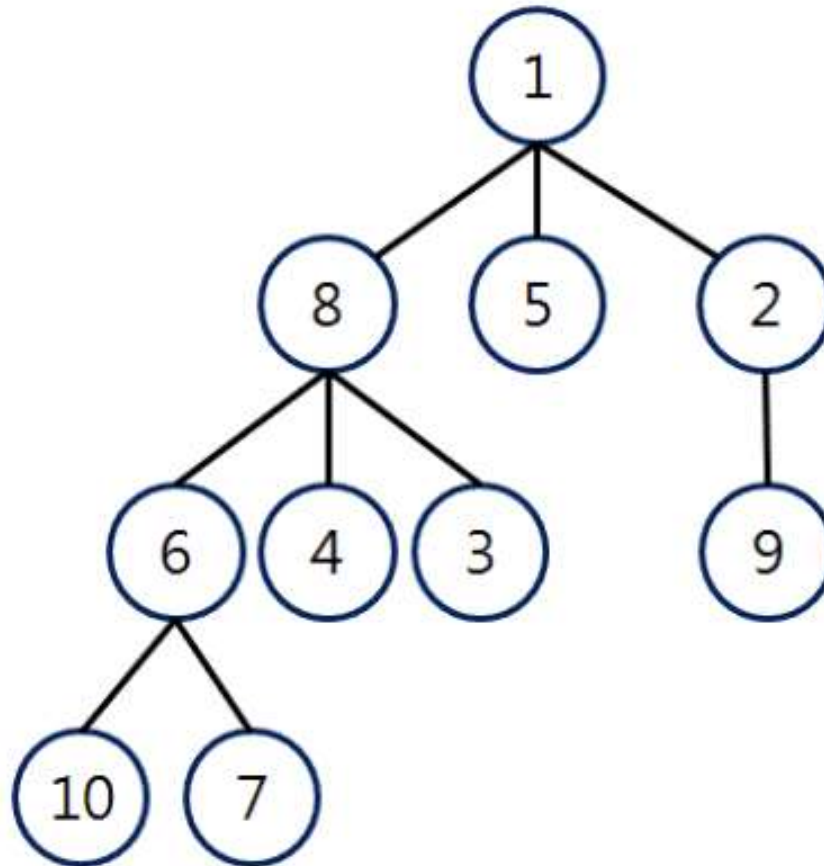
```
bft(in v:Vertex) {  
    // Traverses a graph beginning at vertex v  
    // by using breath-first strategy: Iterative Version  
    q.createQueue();  
    // add v to the queue and mark it  
    q.enqueue(v);  
    Mark v as visited;  
    while (!q.isEmpty()) {  
        q.dequeue(w);  
      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

Trace of Iterative BFT – starting from vertex a



<u>Node visited</u>	<u>Queue (front to back)</u>
a	a (empty)
b	b
f	bf
i	bfi
	fi
c	fic
e	fice
	ice
g	iceg
	ceg
	eg
d	egd
	gd
	d
	(empty)
h	h
	(empty)

Example



Order : 1 → 8 → 5 → 2 → 6 → 4 → 3 → 9 → 10 → 7

