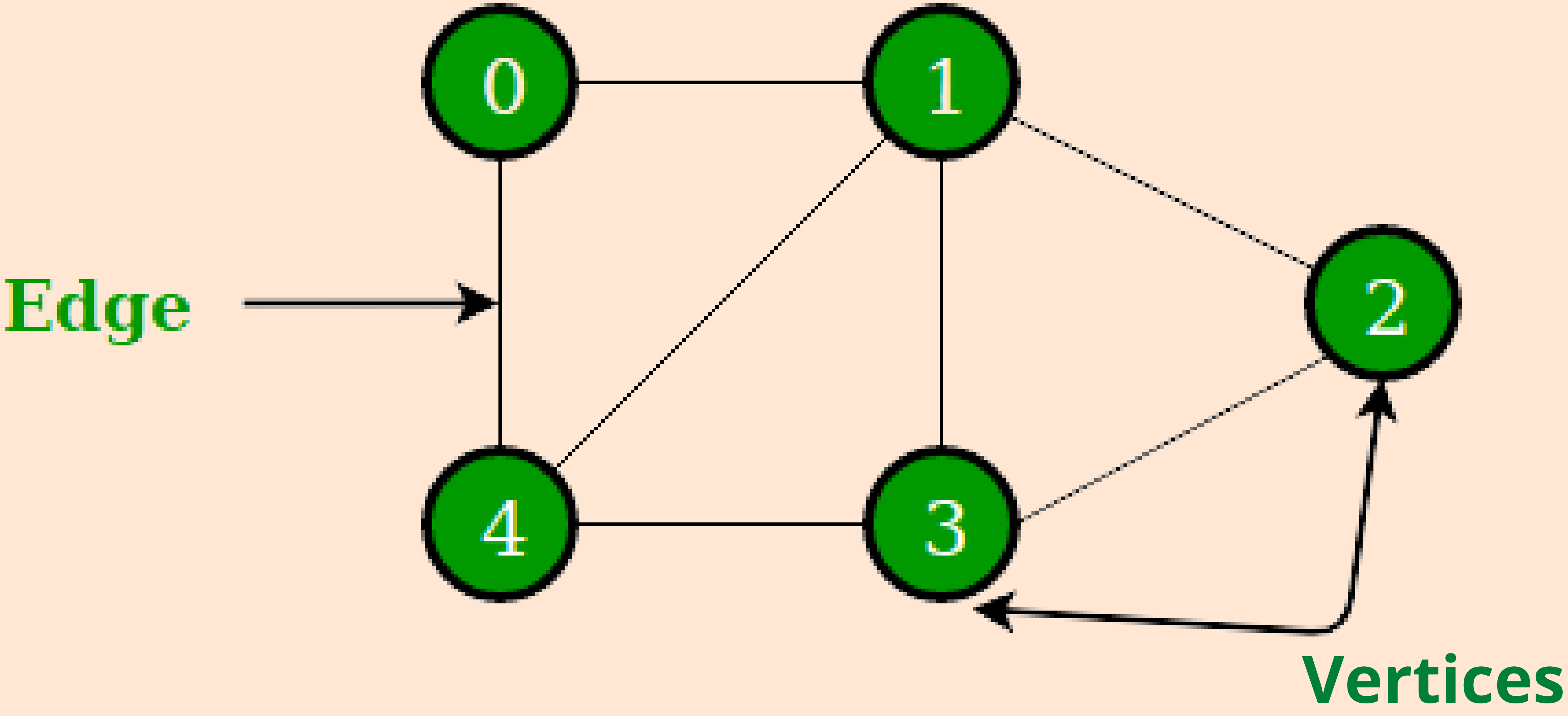
GRAPHS

Introduction

- Non-linear data structure

A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.



Applications

Google Maps

Facebook Network

World Wide Web

Important Terminologies

Graph - Data structure that consists of a set of **nodes** (vertices) and a set of **edges** that relate the nodes to each other

Edge :

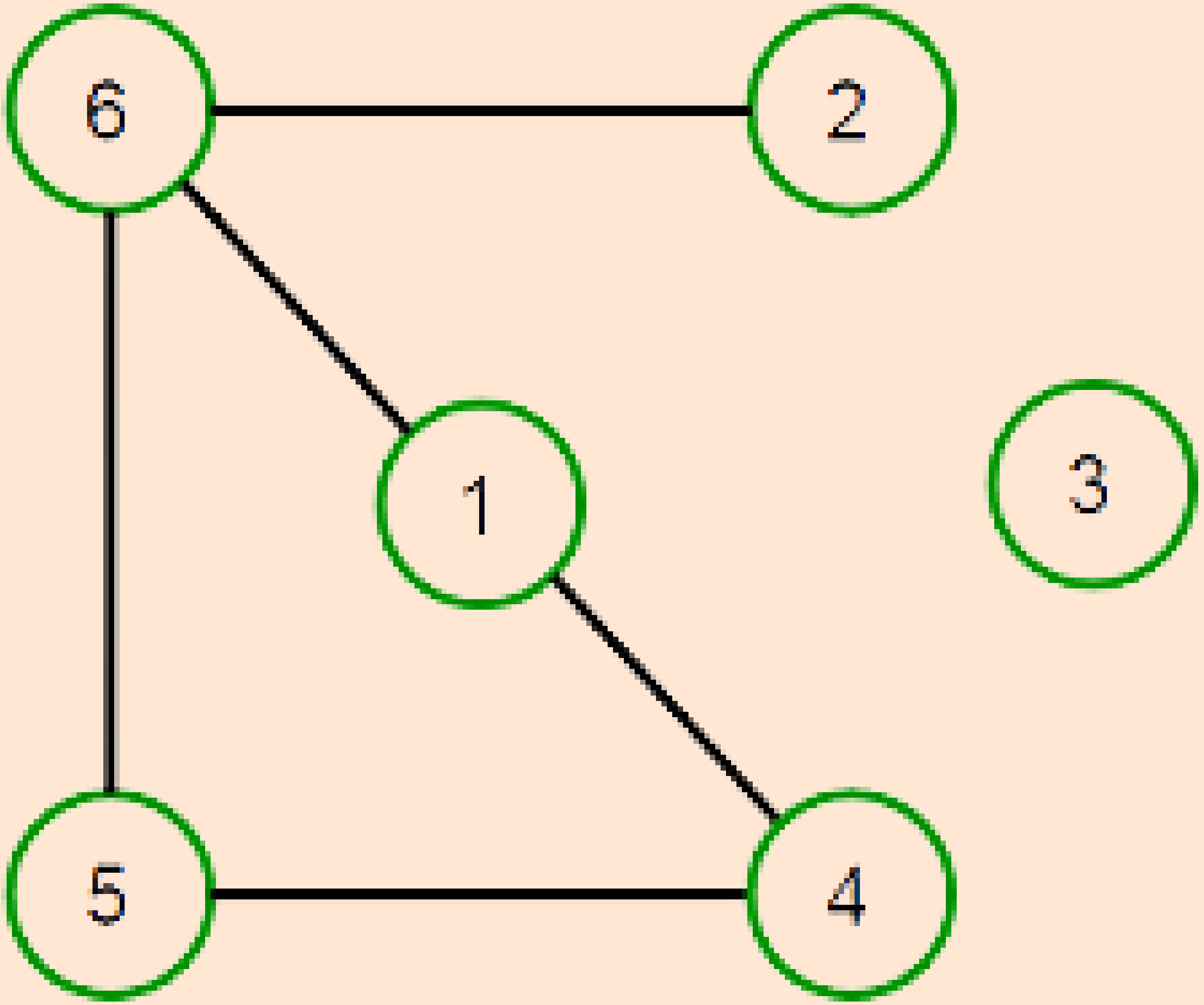
- An edge connects the vertices that define it
- In some cases, the vertices can be the same

Node: Fundamental unit out of which graphs are formed.

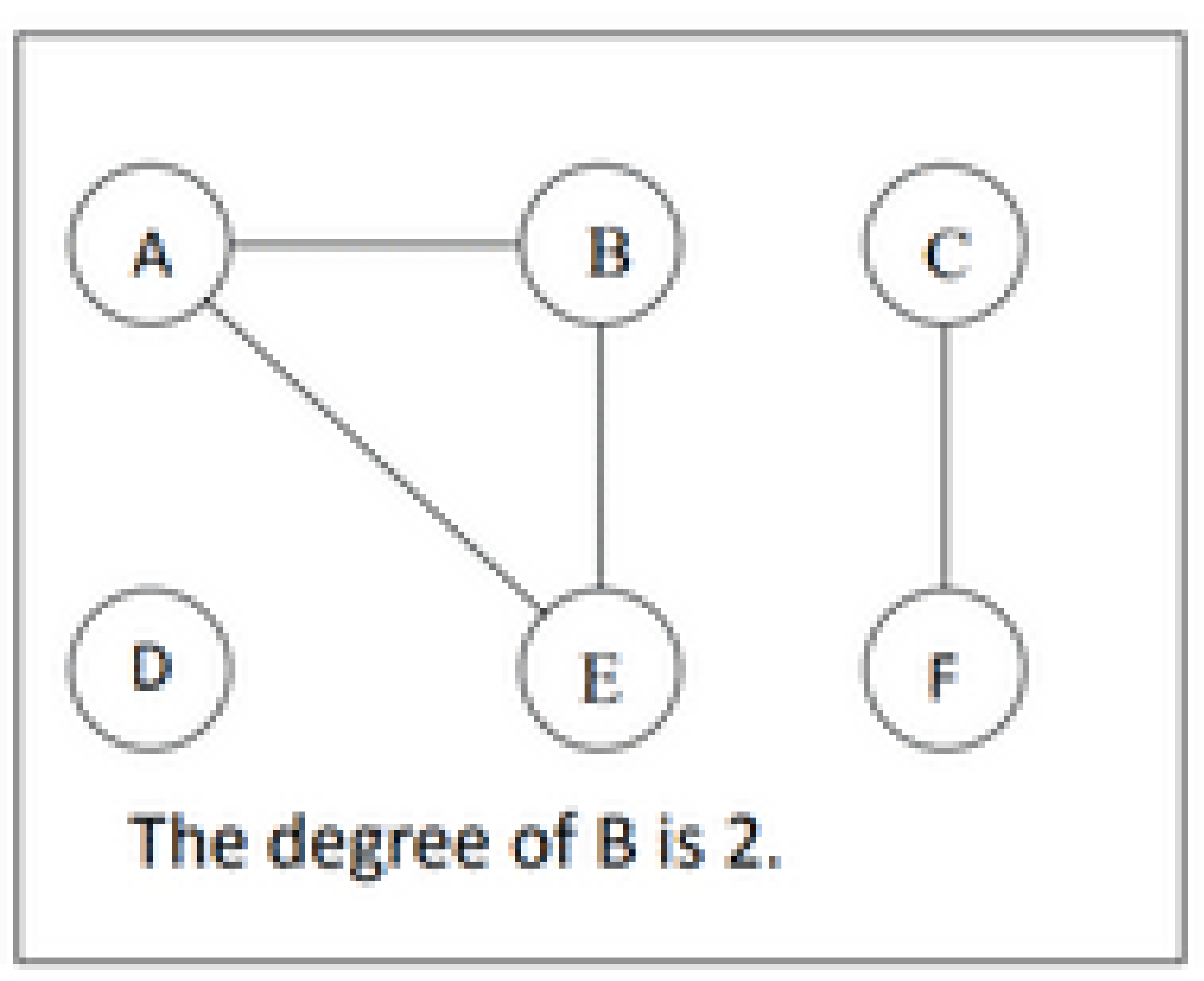
$G = (V, E)$

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

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



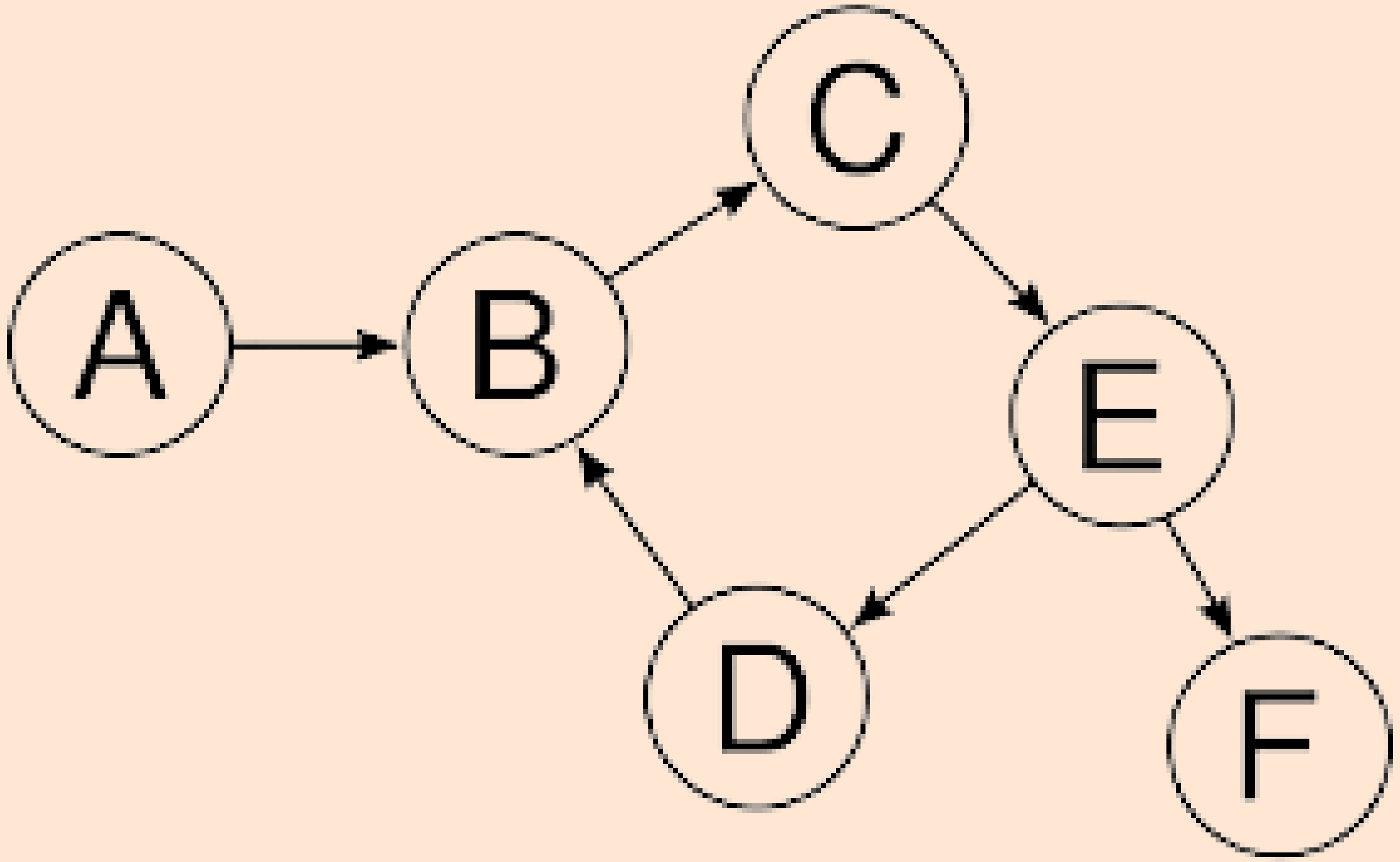
Degree - Number of edges incident on a node



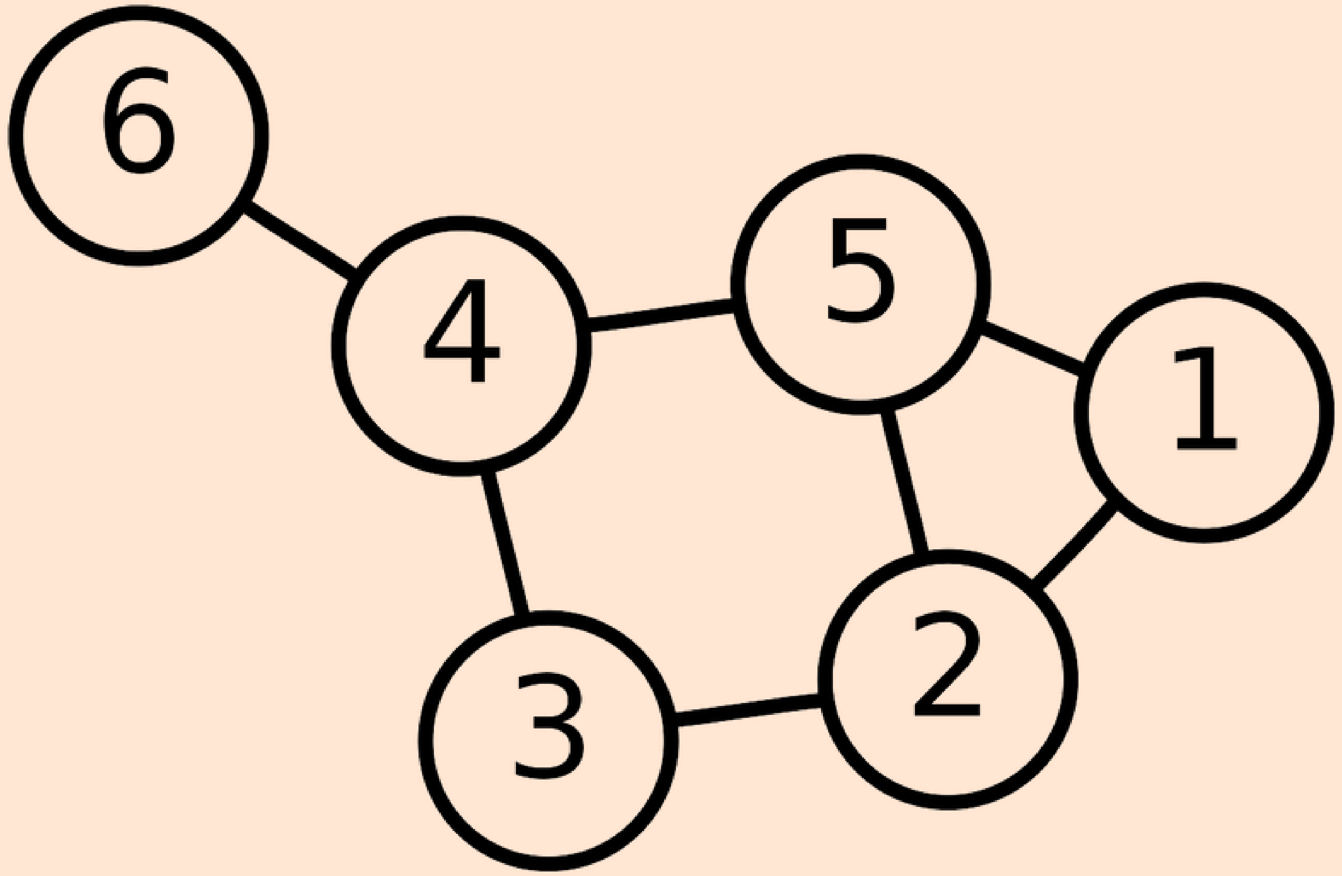
Types of Graphs

An edge (u, v) is said to be directed from u to v if the pair (u, v) is ordered with u preceding v, otherwise it is undirected.

Directed Graph



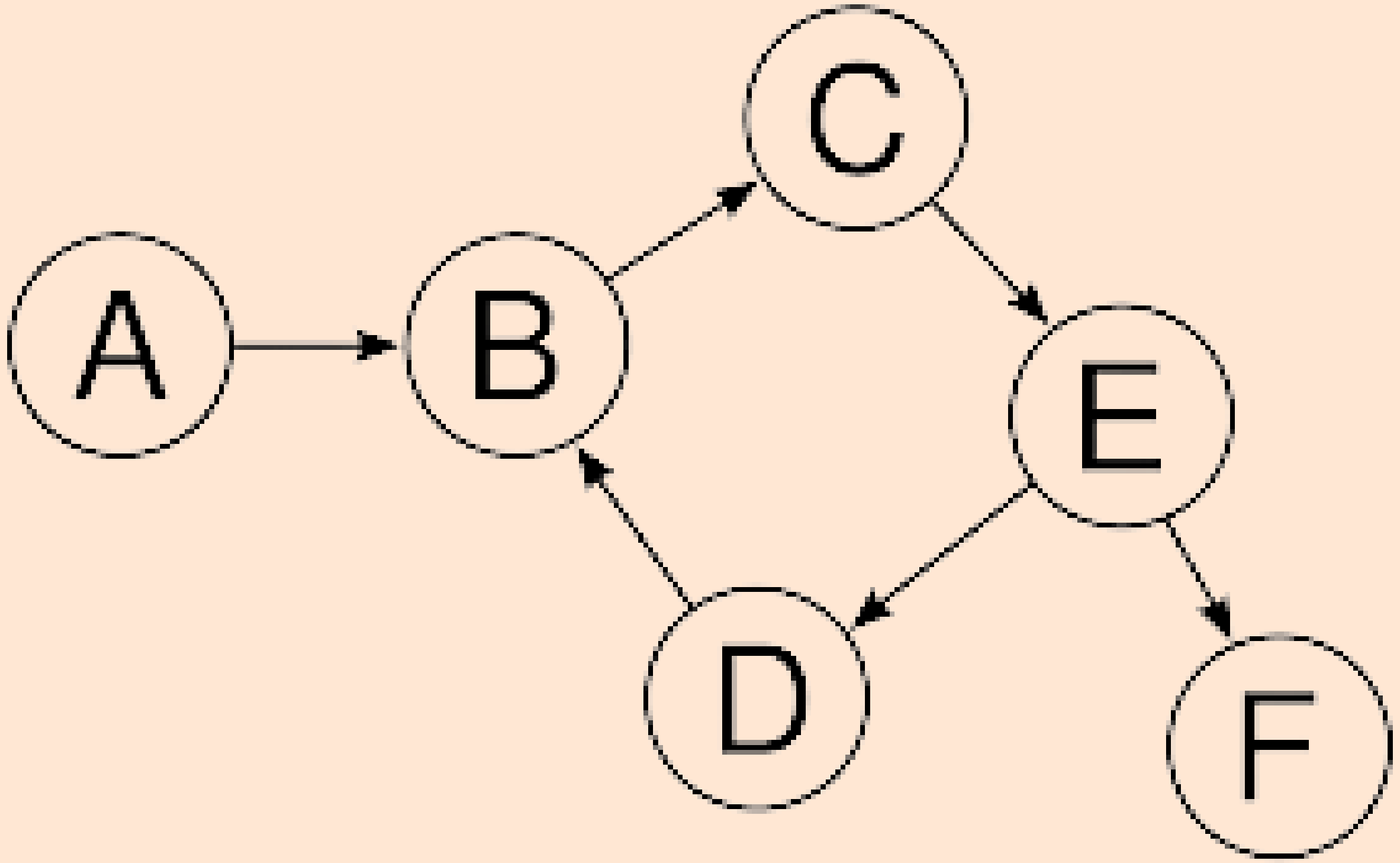
Undirected Graph



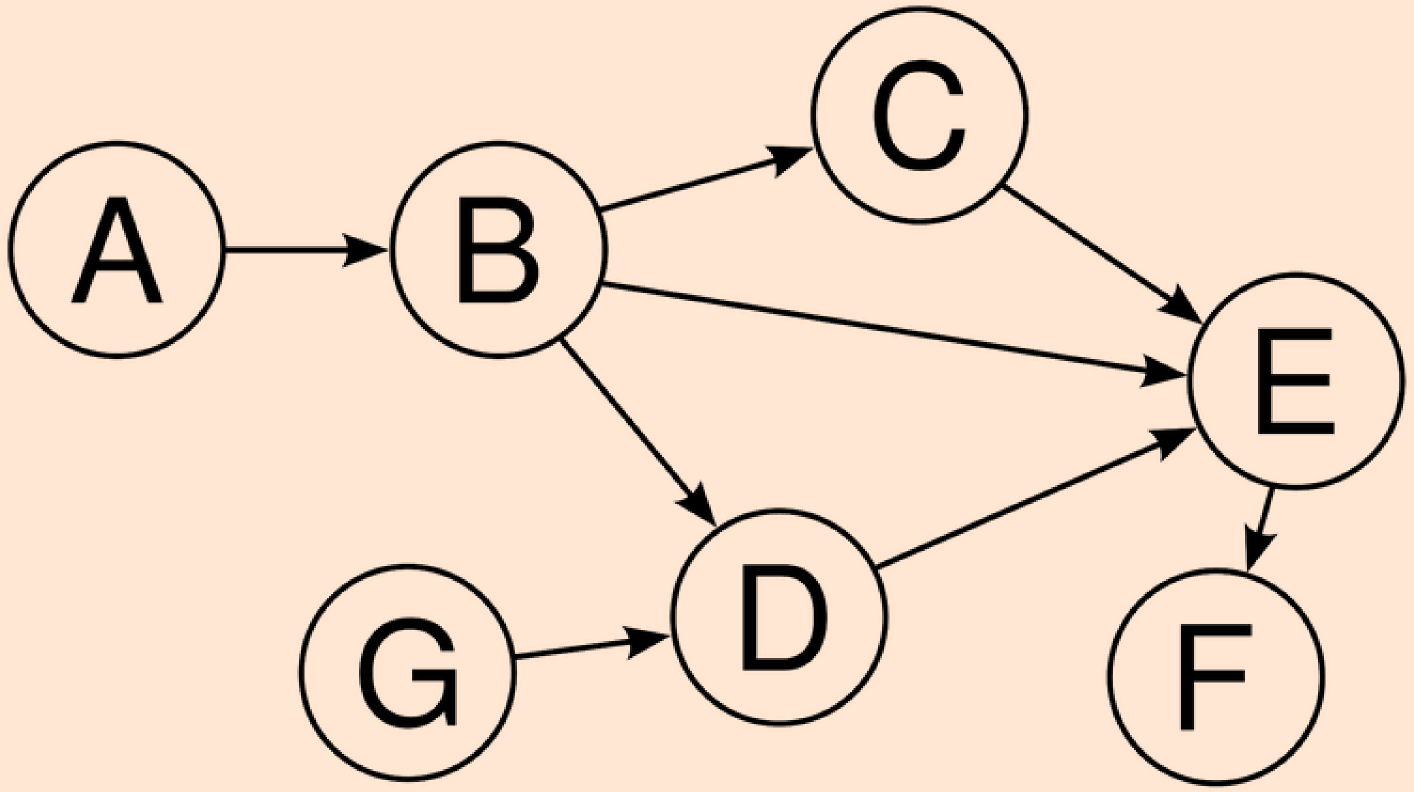
A path is a sequence of vertices such that there is an edge from each vertex to its successor.

A path from a vertex to itself is called a cycle.

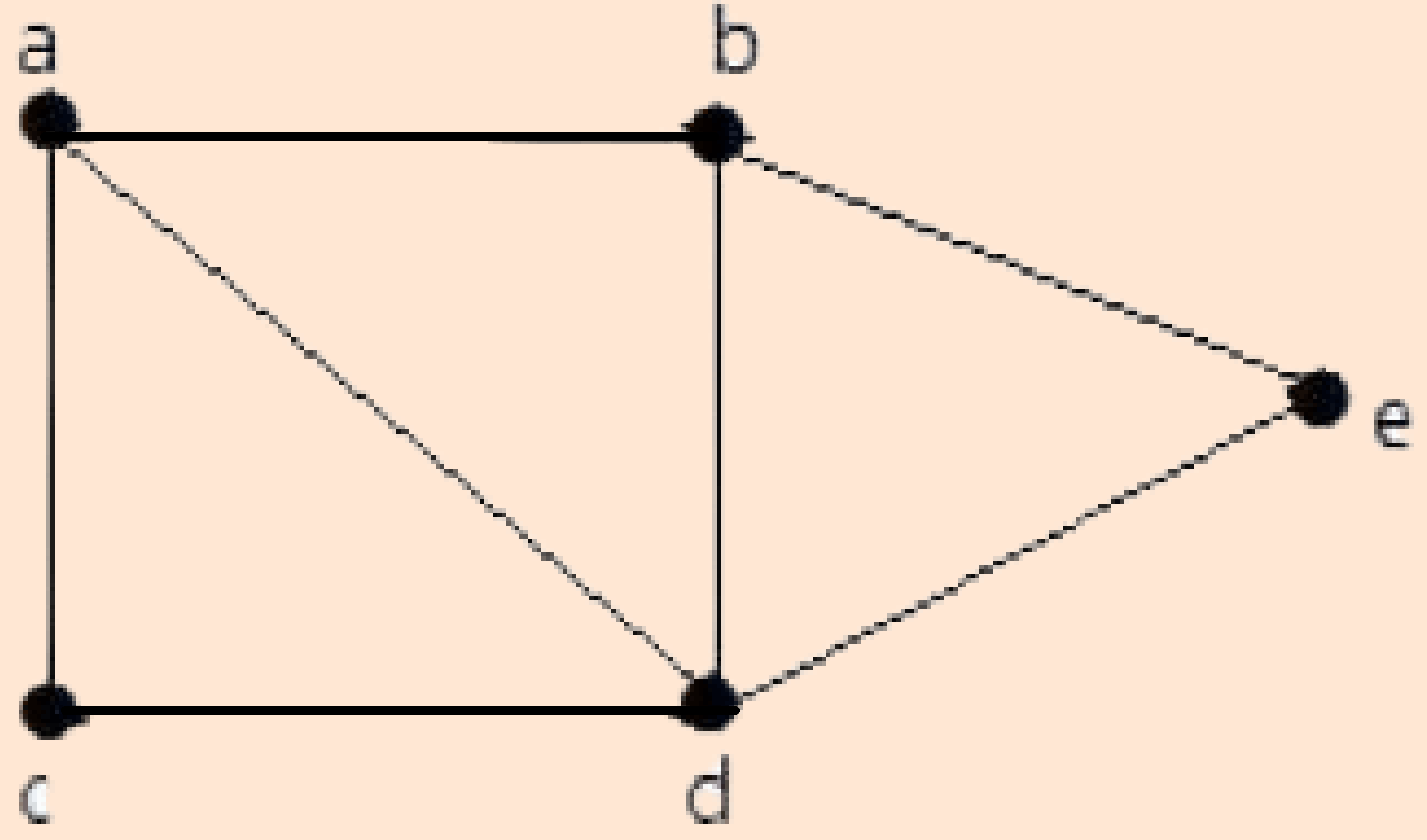
Cyclic Graph



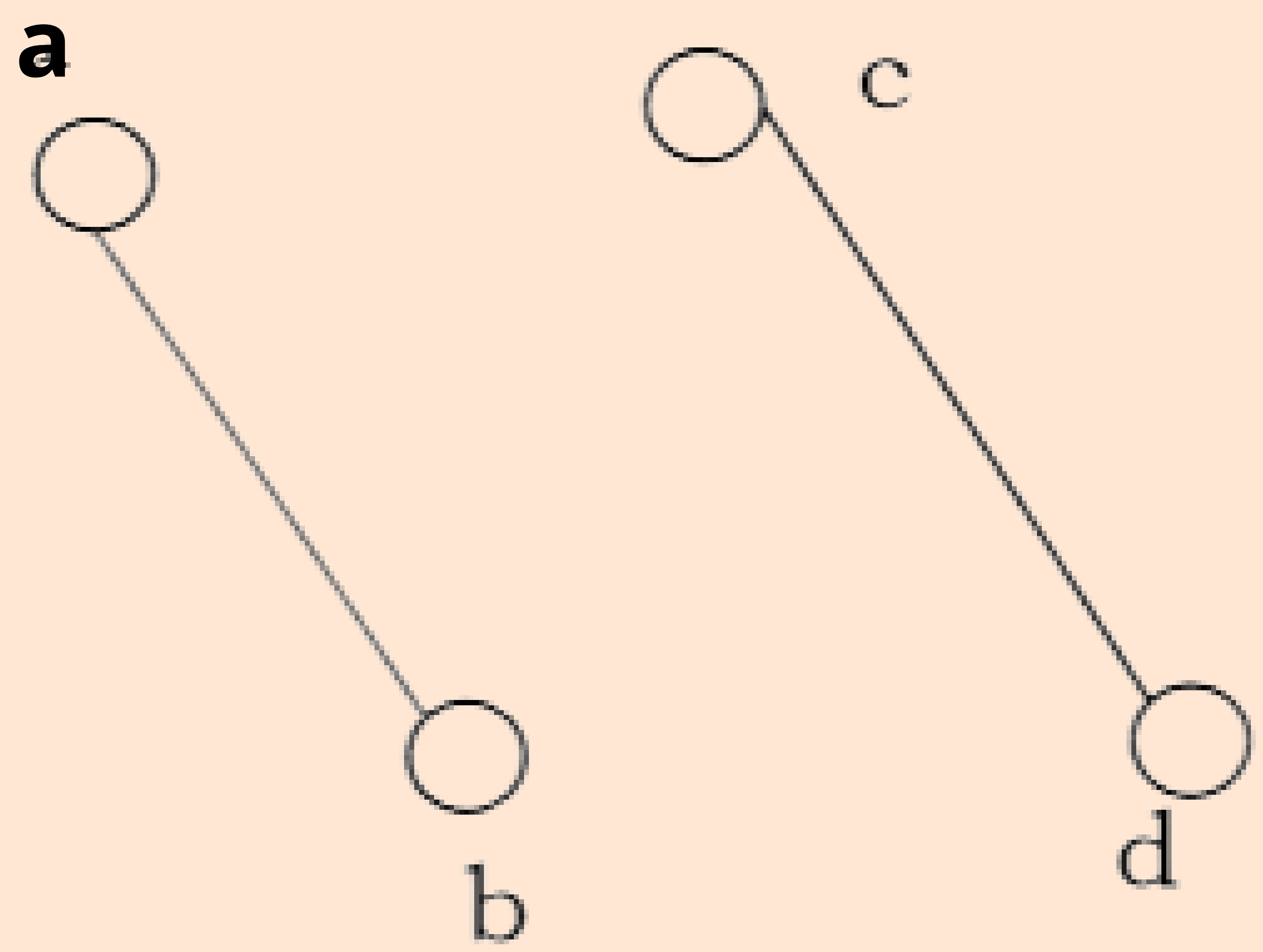
Acyclic Graph



Connected Graph

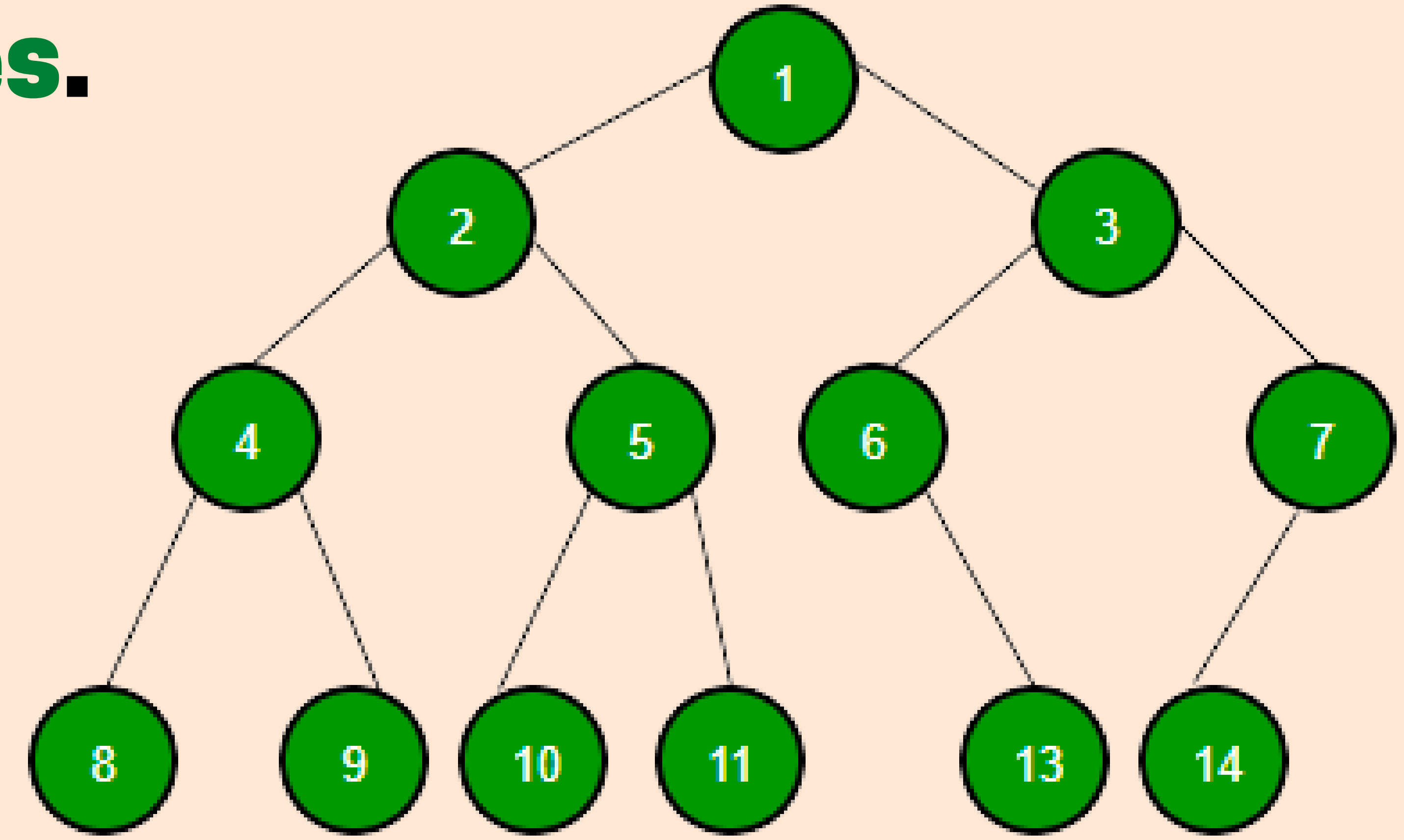


Unconnected Graph



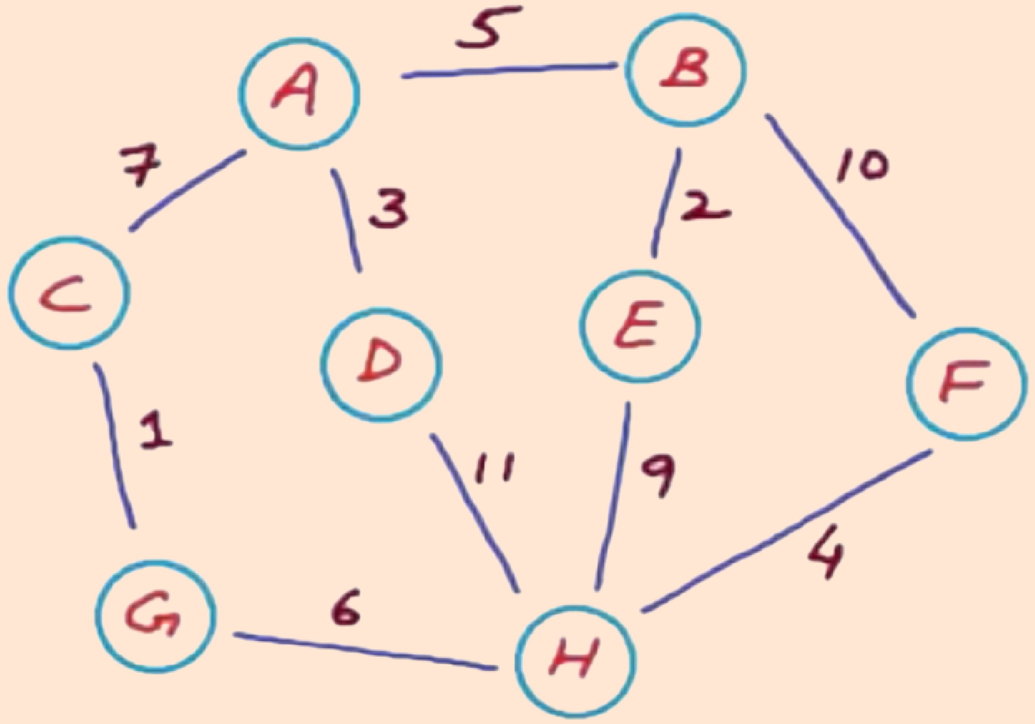
Is tree a graph?

- Tree is an **undirected, connected and acyclic** graph
- Represents **hierarchical structure** in a graphical form.
- A tree with **n vertices has (n-1) edges**.



Graph Representation

1 Edge List




```
class Edge
{
    string startVertex;
    string endVertex;
    int weight;
};
```

Time and Space Complexity

We will address two questions

1) How much time will it take to find all nodes adjacent to a given node?

2) How much time will it take to find if two nodes are connected or not?

Space Complexity - O(V) + O(E)

Finding adjacent nodes - O(E)
FInding if 2 nodes are connected - O(E)

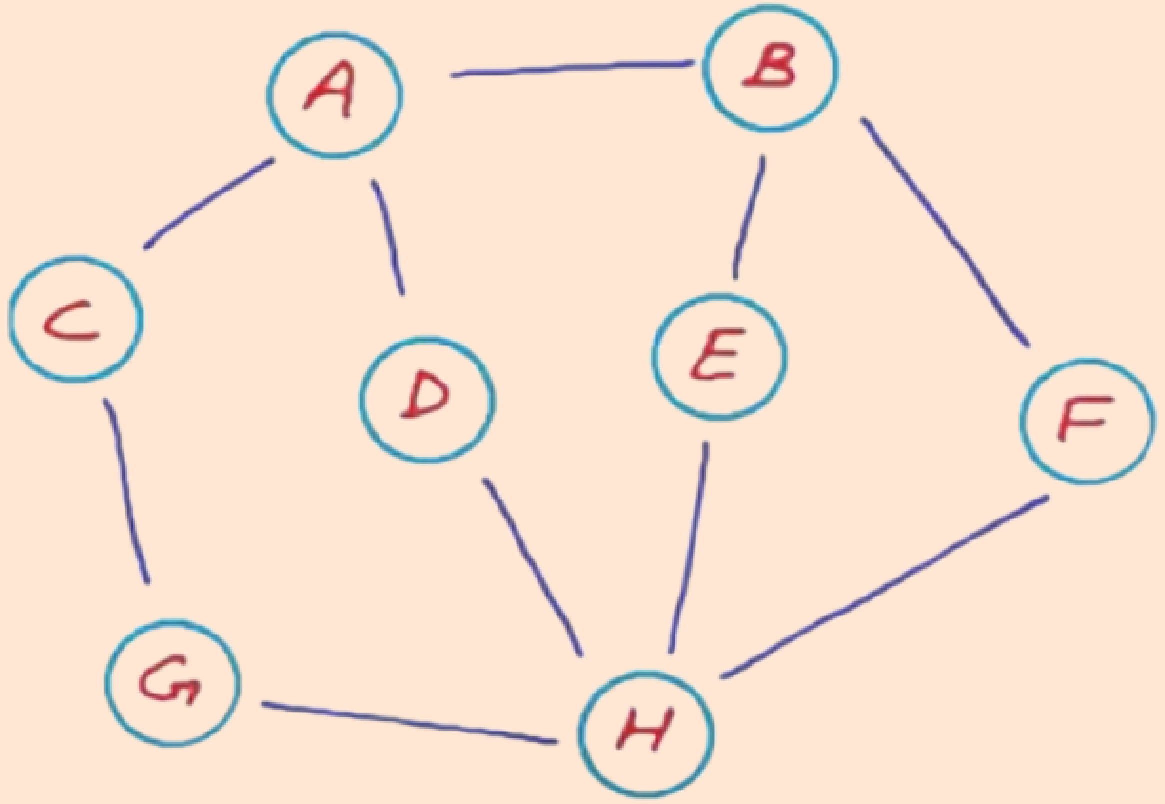
Time Complexity

Since E ~ V*V and V~n then O(n^2) which is ugly!!!!

We need to improve

2 Adjacency Matrix

$$A_{ij} = \begin{cases} 1, & \text{if } \exists \text{ edge from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$$



Vertex List

A
B
C
D
E
F
G
H
⇓

	0	1	2	3	4	5	6	7
0	0	1	1	1	0	0	0	0
1	1	0	0	0	1	1	0	0
2	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	0	1
6	0	0	1	0	0	0	0	1
7	0	0	0	1	1	1	1	0

Symmetric in case of undirected graph!

Time and Space Complexity

We will address two questions

1) How much time will it take to find all nodes adjacent to a given node?

2) How much time will it take to find if two nodes are connected or not?

Space Complexity - $O(V) + O(V*V)$

Finding adjacent nodes - $O(V)$
FInding if 2 nodes are connected - $O(1)$ / $O(V)$

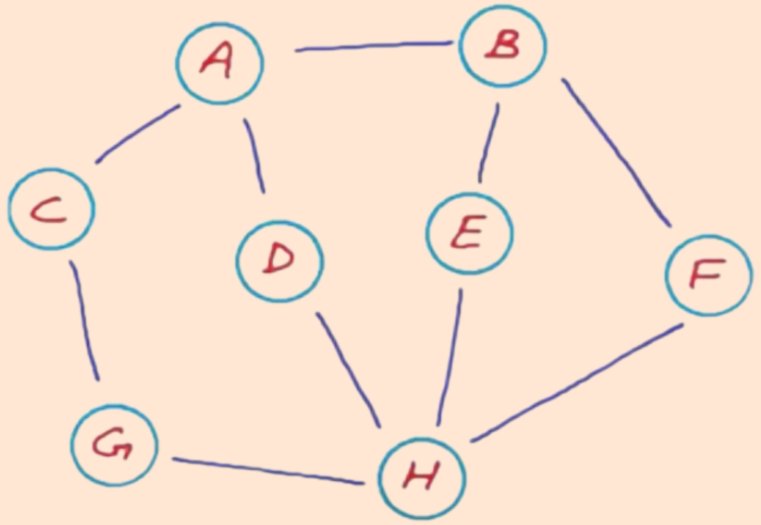
Time Complexity

We are using a lot of space since most graphs are sparse

We need to improve further!!!!!!

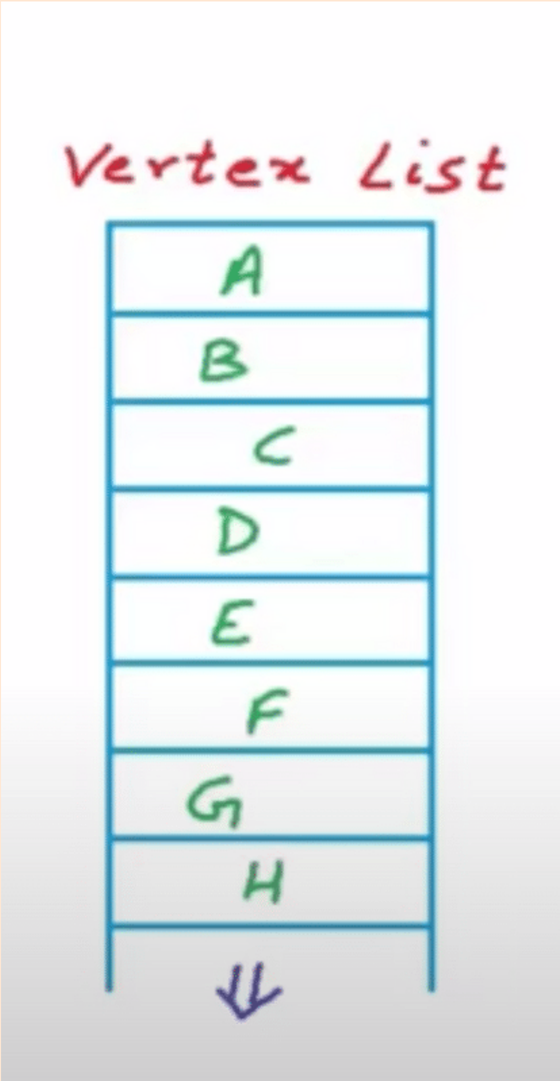
3

Adjacency List

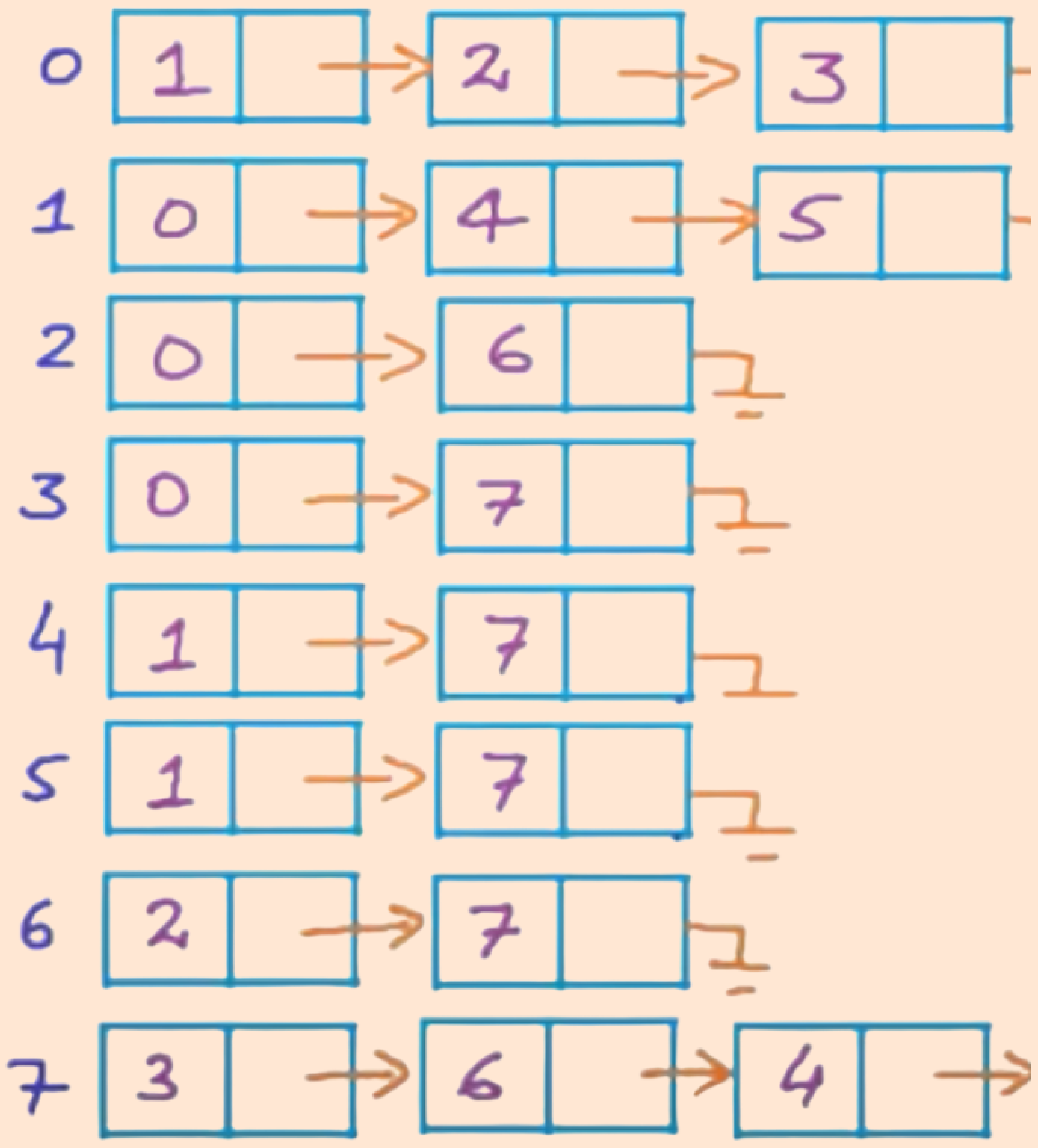


Using Linked List

Using vector



0	1	2	3	
1	0	4	5	
2	0	6		
3	0	7		
4	1	7		
5	1	7		
6	2	7		
7	3	6	4	5



Time and Space Complexity

We will address two questions

1) How much time will it take to find all nodes adjacent to a given node?

2) How much time will it take to find if two nodes are connected or not?

Space Complexity - O(V) + O(E)

Better because e<<v in most cases

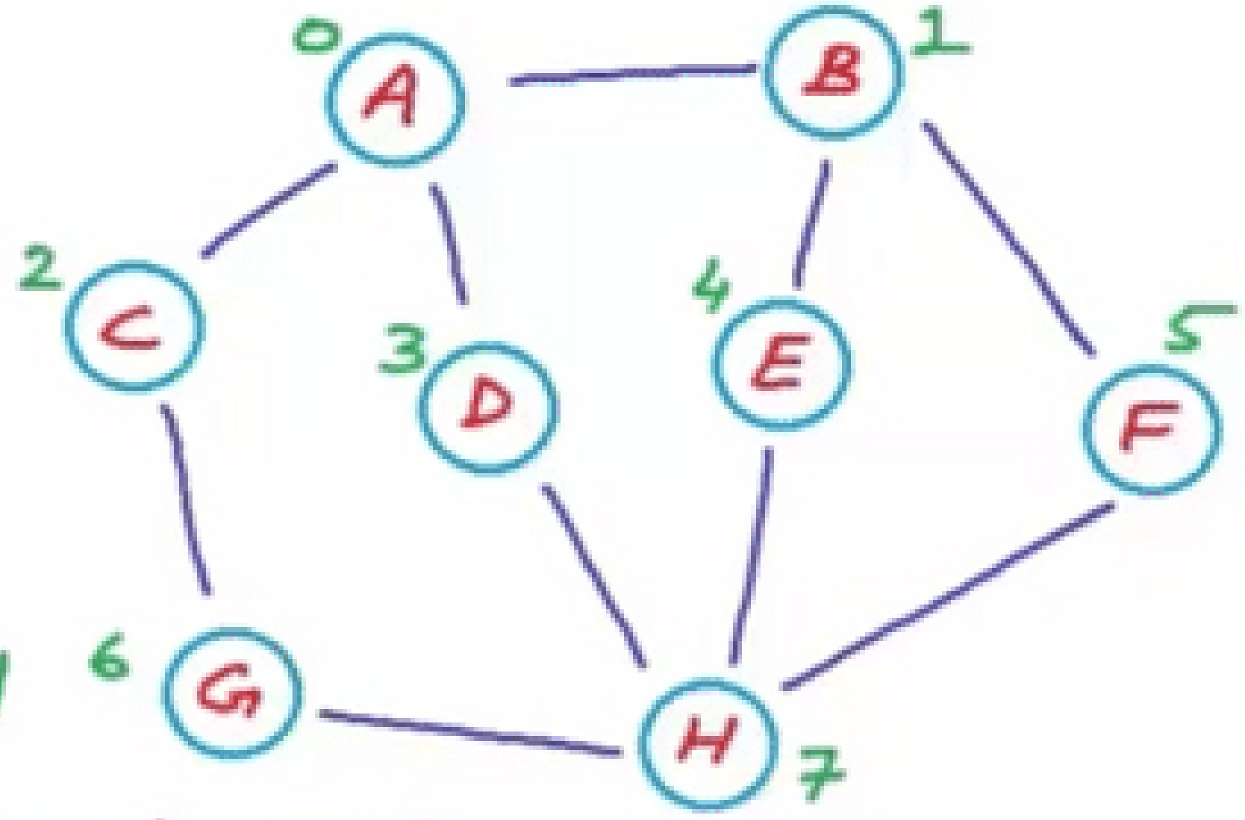
Finding adjacent nodes - O(V)
FInding if 2 nodes are connected - O(V) / O(log V)

Time Complexity

It's better because we can easily add a new edge and can model real world really well!!
Let's have a look at real world example

A real world scenario

Graph Representation



For a social network with a billion(10^9) users:

if maximum no. of friends = 10,000

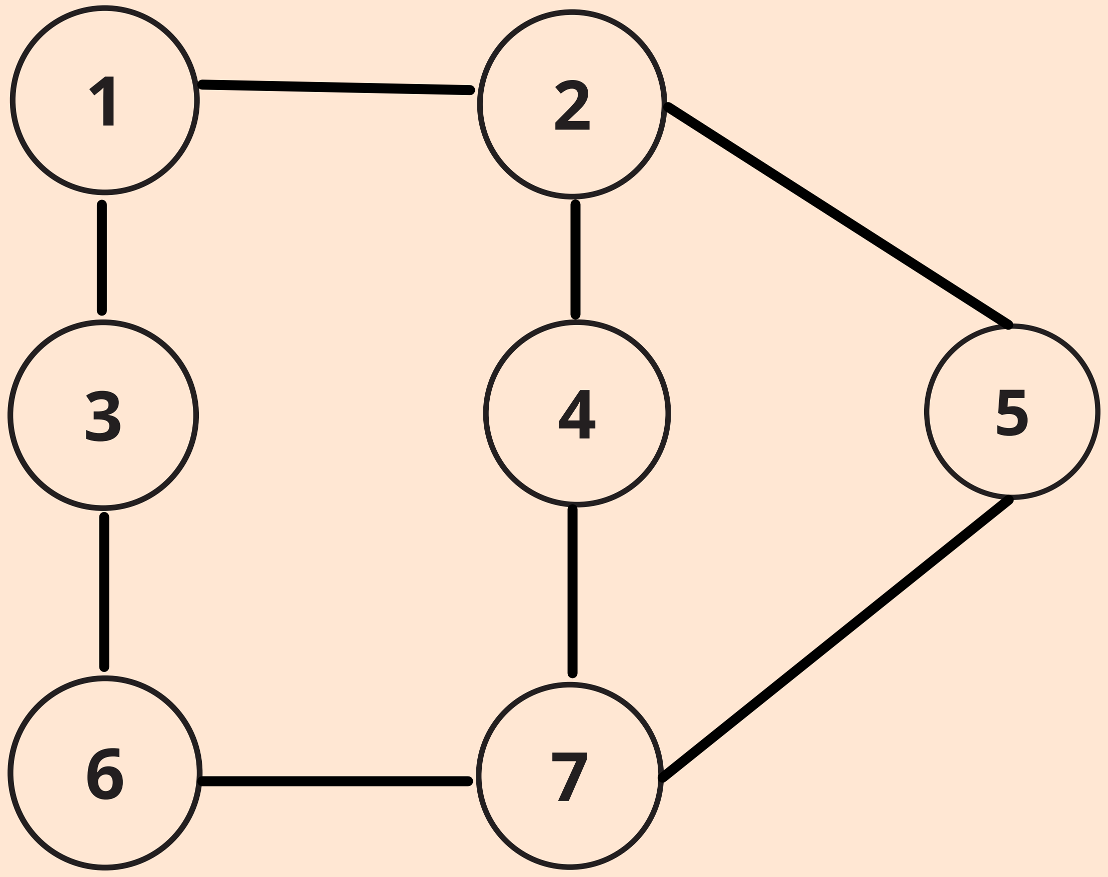
if machine can scan 10^6 cells/second

	Structure-1 (Matrix)	Structure-2
finding adjacent nodes	$\frac{10^9}{10^6} = 1000$ sec $= 16.66$ min	$\frac{10^4}{10^6} = 10^{-2}$ sec $= 10$ ms
finding if two nodes are connected	$\frac{1}{10^6}$ sec $= 1 \mu s$	10 ms

Credits : MyCodeSchool

Breadth-First Search (BFS)

BFS is an algorithm for traversing Graph data structures. The neighbours of vertices are visited in order of occurrence starting from an arbitrary node "breadth-wise".

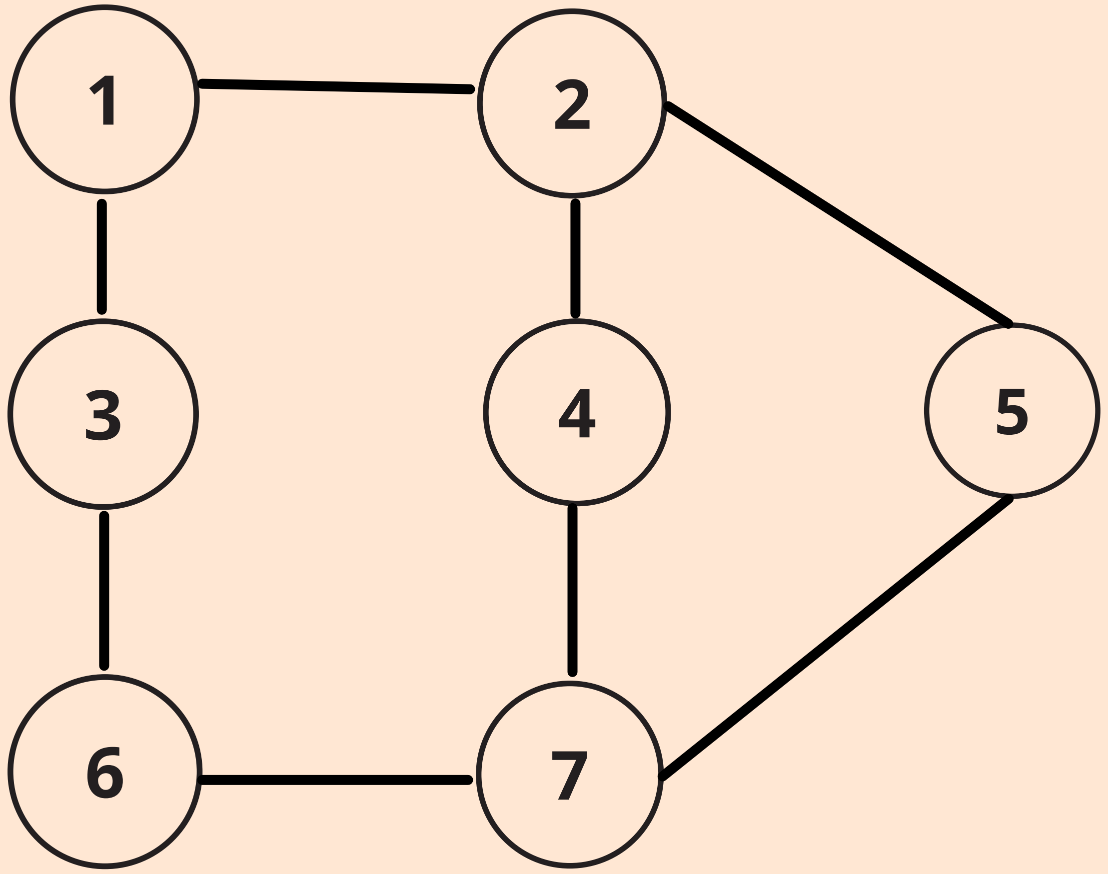


1, 2, 3, 4, 5, 6, 7

Algorithm

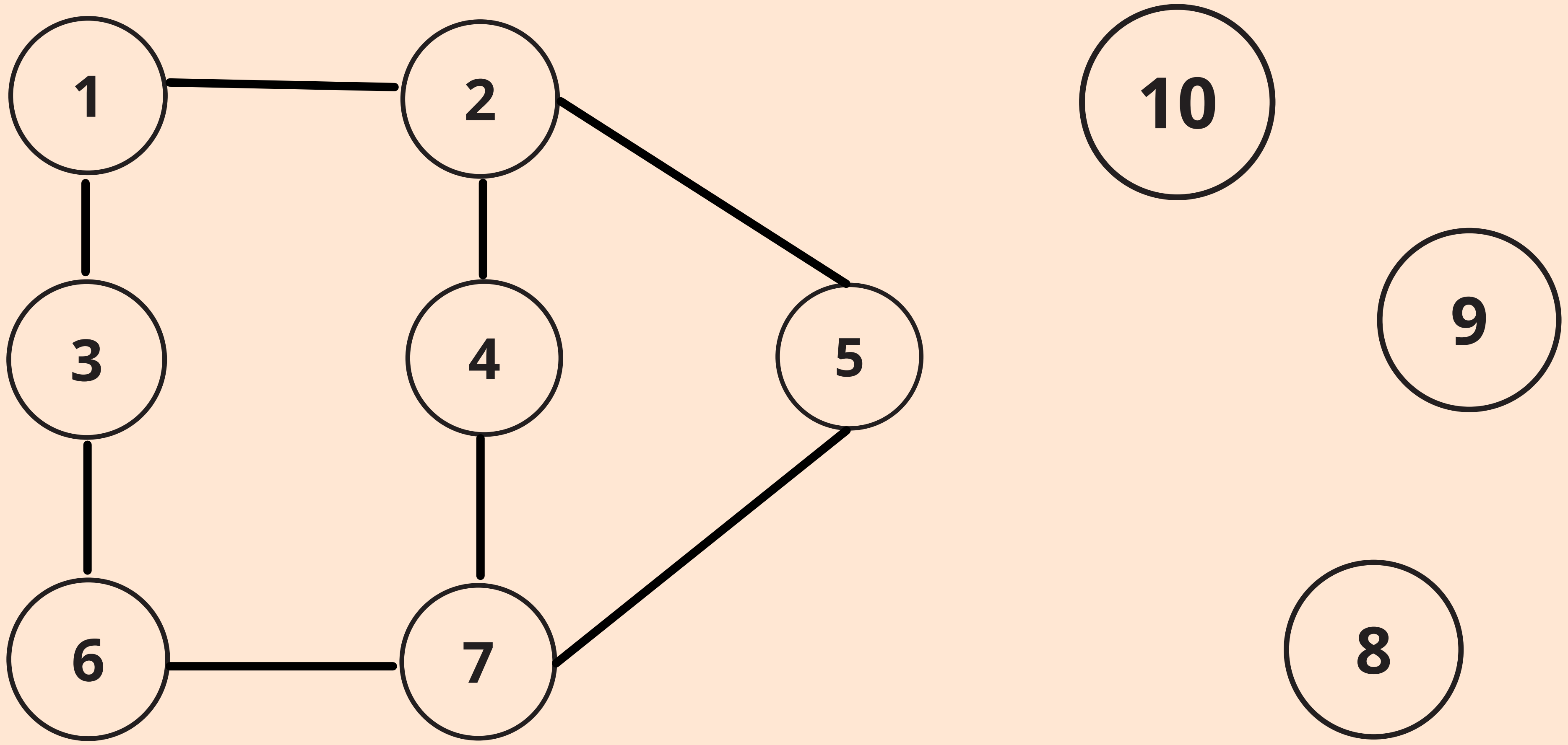
```
BFS (G):
    while vertex is not explored:
        add to queue, Q(any vertex)

        while Q is not empty:
            p = remove front element from Q
            if p is not visited:
                print "p"
                mark p as visited
                add to queue, Q(all adjacent vertices of p)
```



Disconnected Graph

BFS can not be applied



V * adj V
SUM(1 * 2= 2
1*2 = 2
1 * 2 =2)

= 2*Edges

Time   Complexity of BFS : O(V + E)

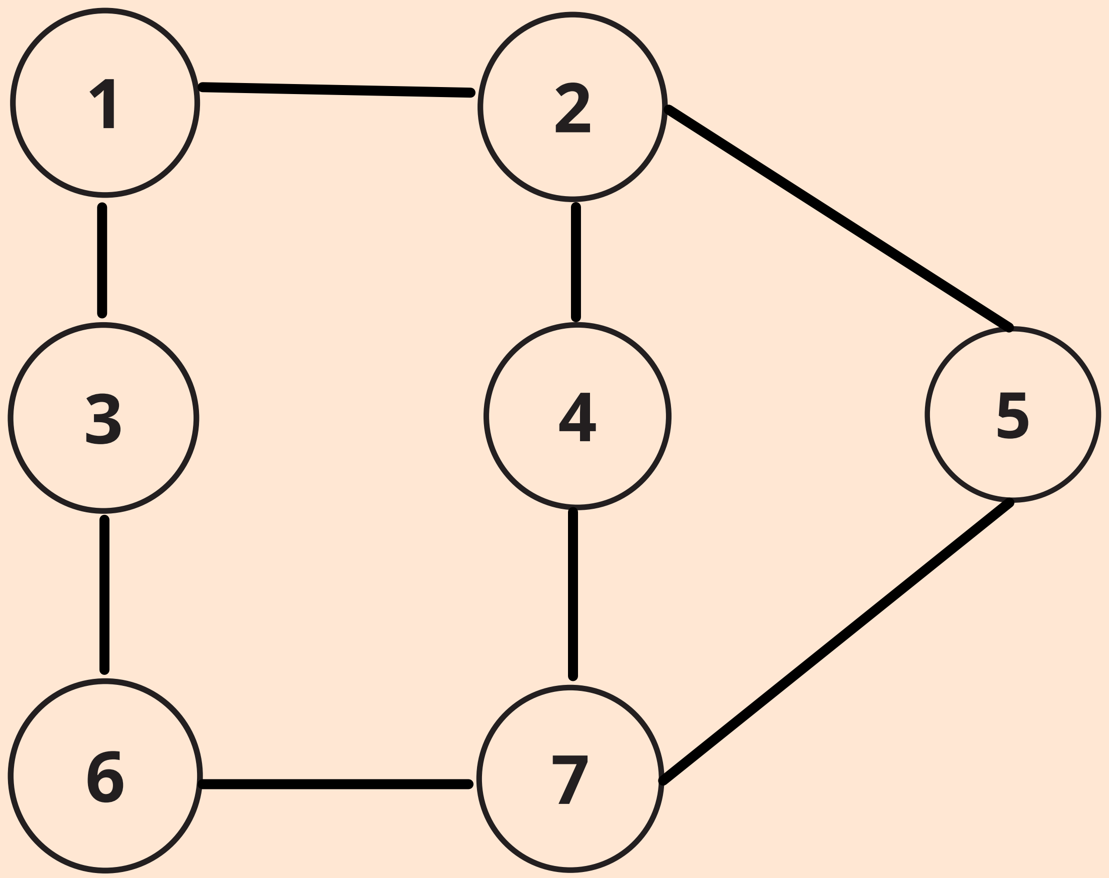
Space Complexity of BFS : O(V + E)

```
BFS (G):
    while vertex is not explored: -----------------------------------------------> O(V)
        add to queue, Q(any vertex)  ------------------------------------------->O(1)
                                                                                          O(V+E)
        while Q is not empty:  ---------------------------------------------> O(V)        O(E)

            p = remove front element from Q -------------> O(1)                     O(E)
            if p is not visited:    ------------------------------> O(1)
            print "p"   -------------------------------------------> O(1)   O(adj V)
            mark p as visited  ------------------------------------> O(1)
            add to queue, Q(all adjacent vertices of p)  --> O(adj V)
```

Depth-First Search (DFS)

DFS is an algorithm for traversing Graph data structures. It starts at some arbitrary node and explores as far as possible along each edge before backtracking.

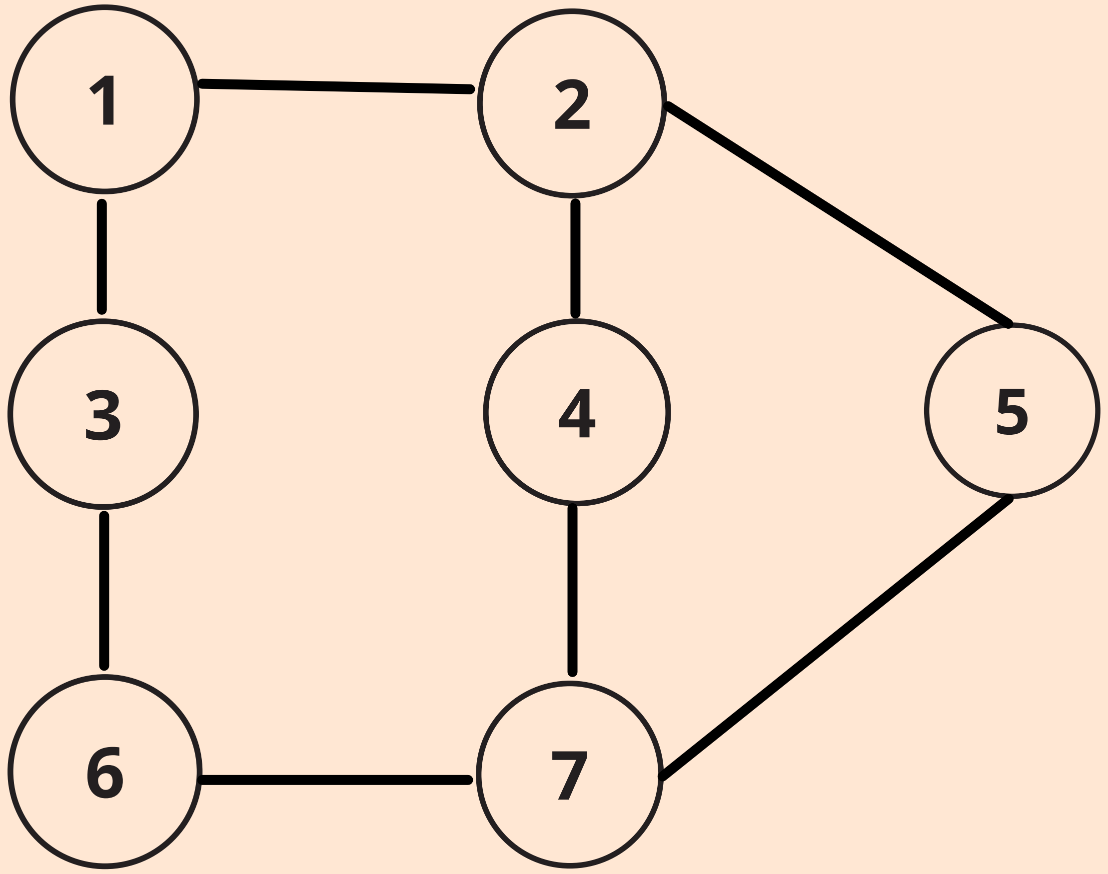


1, 3, 6, 7, 4, 2, 5

Algorithm

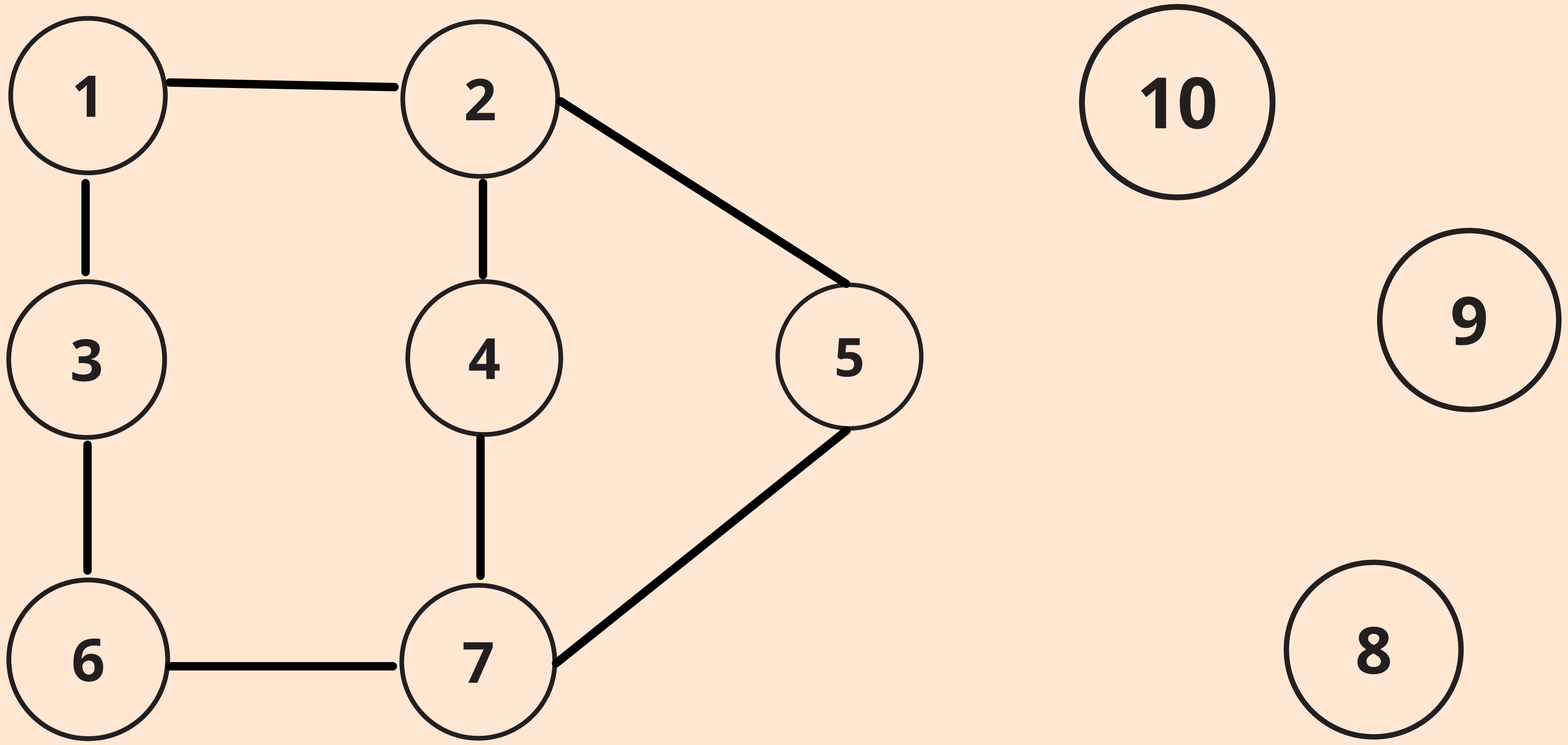
```
DFS (G):
    while vertex is not explored:
        push to stack, S(any vertex)

        while stack is not empty:
            p = pop from S
            if p is not visited:
                print "p"
                mark p as visited
                push to stack, S(all adjacent vertices of p)
```



Disconnected Graph

DFS can not be applied



Time   Complexity of DFS : O(V + E)

Space Complexity of DFS : O(V + E)

```
DFS (G):
    while vertex is not explored: ---------------------------------------------------> O(V)
       add to stack, S(any vertex)  ----------------------------------------------->O(1)
                                                                                              O(V+E)
       while S is not empty:  --------------------------------------------------> O(V)
                                                                                          O(E)

           p = remove front element from S --------------> O(1)                          O(E)
           if p is not visited:    -----------------------------------> O(1)
           print "p"   ---------------------------------------------------> O(1)   O(adj V)
           mark p as visited  --------------------------------------> O(1)
           add to stack, S(all adjacent vertices of p)  --> O(adj V)
```

BFS vs. DFS

	BFS	DFS
How it works?	It visits Breadth-First	It visits Depth-First
Internal Data Structure used:	Stack	Queue
Space Complexity	O(V+E)	O(V+E)
Time Complexity	O(V+E)	O(V+E)
Applications	- Find Shortest paths - Find 2nd & 3rd connections	- Topological Sort - Solving puzzles like maze