Grado en Ciencia e Ingeniería de Datos, 2018-2019

Unit 6. Graphs

Algorithms and Data Structures (ADS)

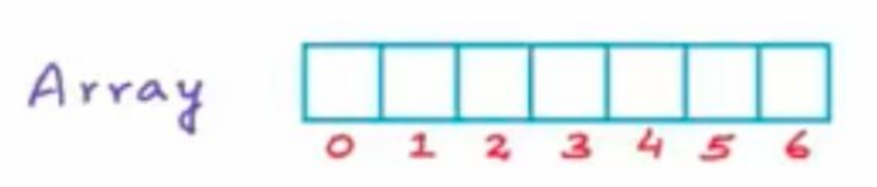
Author: Isabel Segura-Bedmar

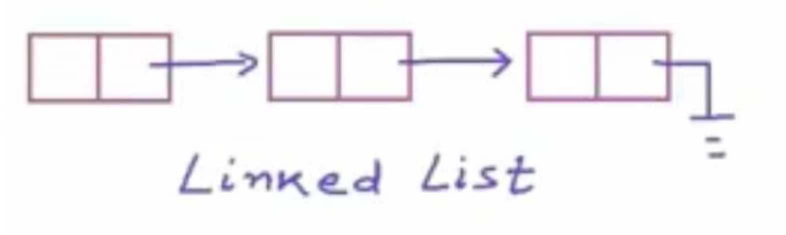
Index

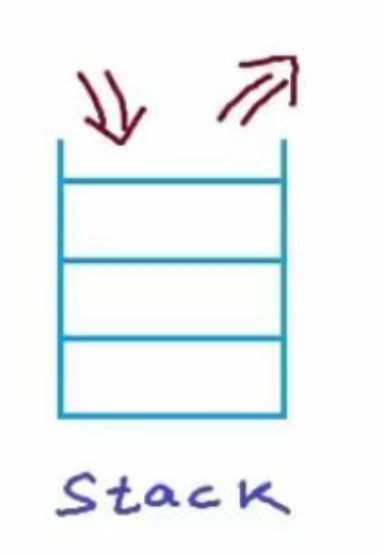
- **Introduction to Graphs**
- Graph properties
- Graph representation:
    - Adjacency Matrix.
    - Adjacency List.
- Graph Traversal

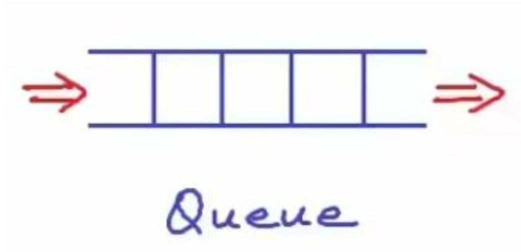
Introduction to Graphs

Linear data structures:



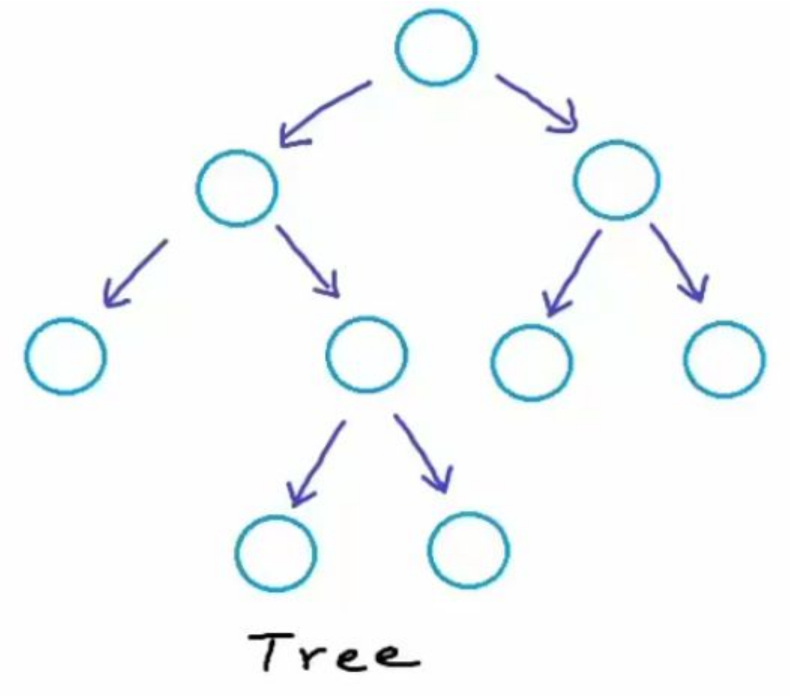






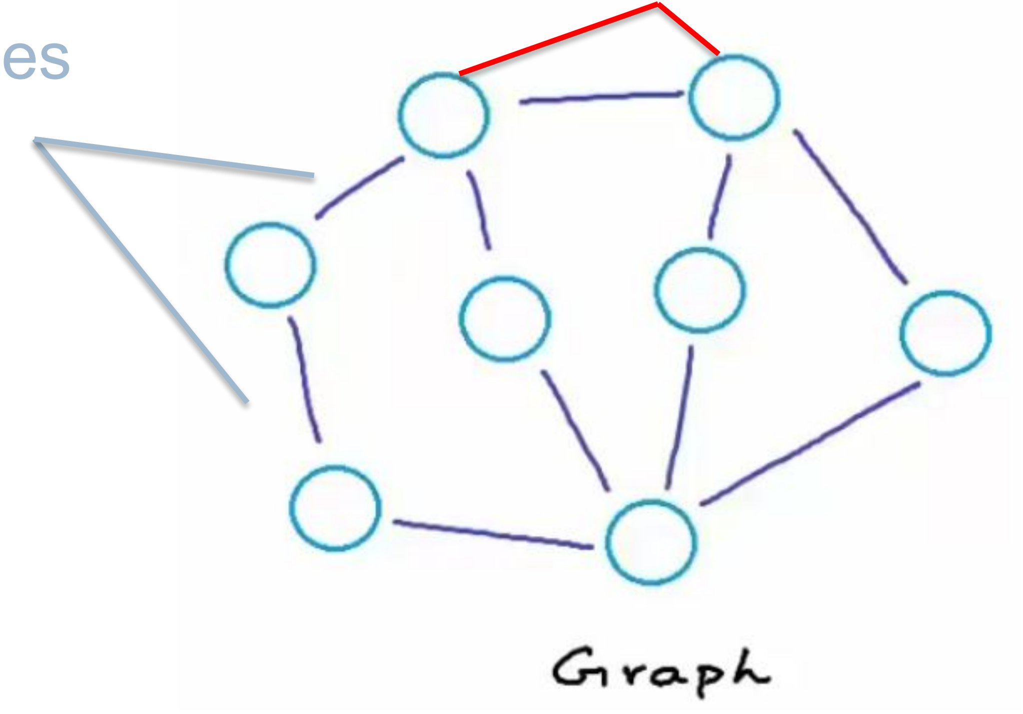
Introduction to Graphs

Non-linear data structures:



Introduction to Graphs

Non-linear data structures:



Nodes or vertices

Edges

No rules for connections

Index

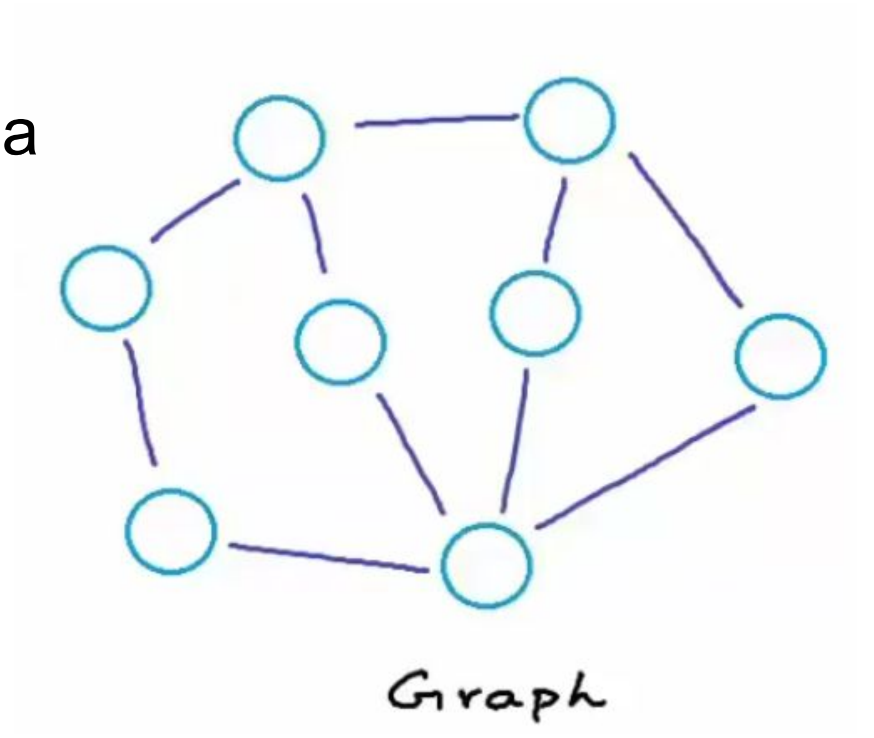
- Introduction to Graphs
- **Graph properties**
- Graph representation:
  - Adjacency Matrix.
  - Adjacency List.
- Graph Traversal

Graph properties

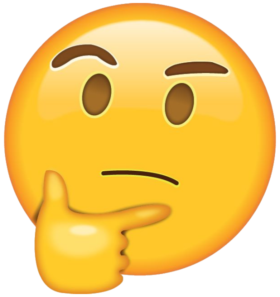
Graph:

A graph G is an ordered pair of a set V of vertices and a set E of edges

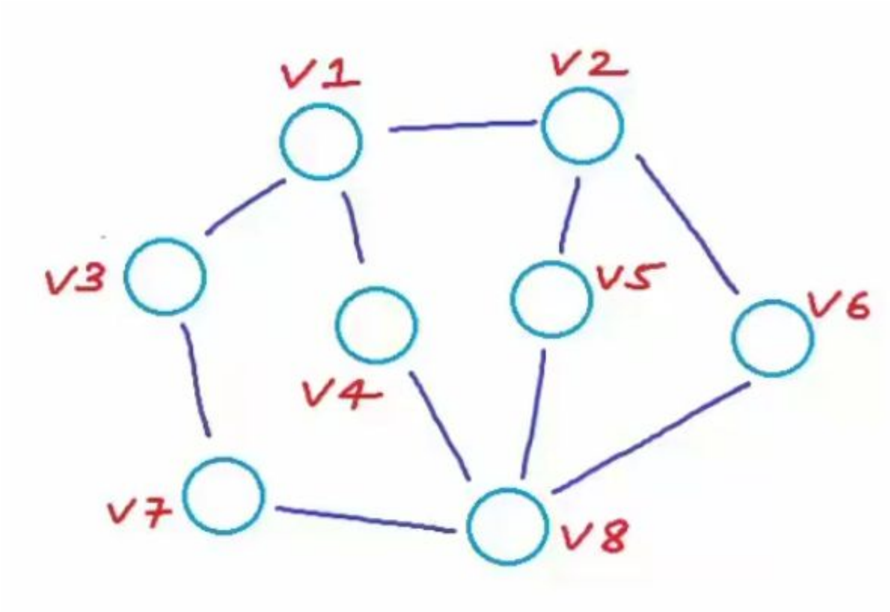
$G=(V,E)$



Graph properties



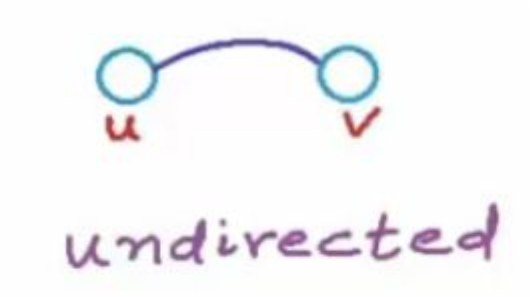
How can we represent an edge?



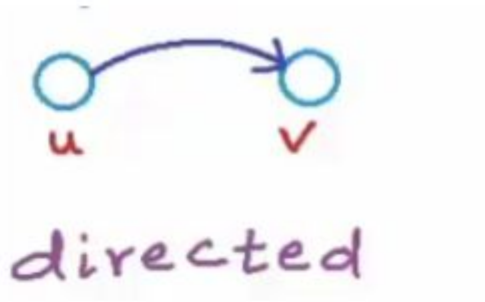
V = { v1, v2, v3, v4, v5, v6, v7, v8}

Graph properties

Types of edges:



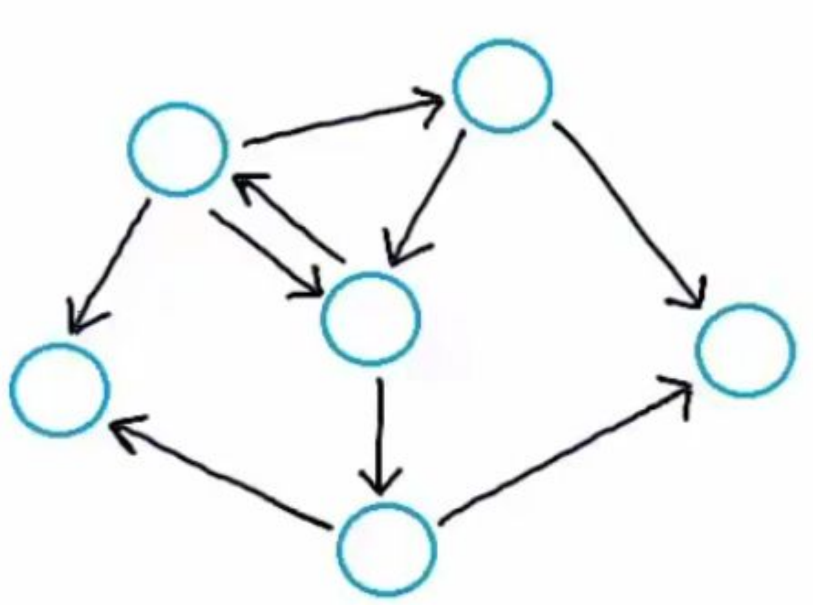
$\{u,v\} = \{v,u\}$



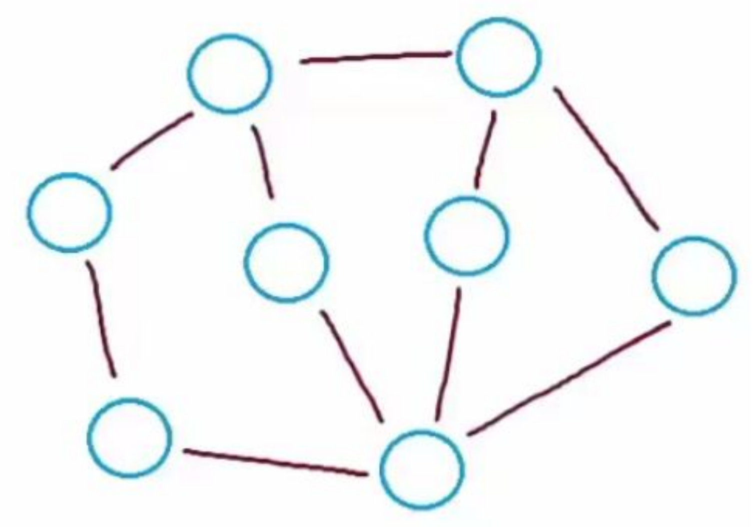
(u,v)!=(v,u) if u!=v

Graph properties

directed vs. undirected

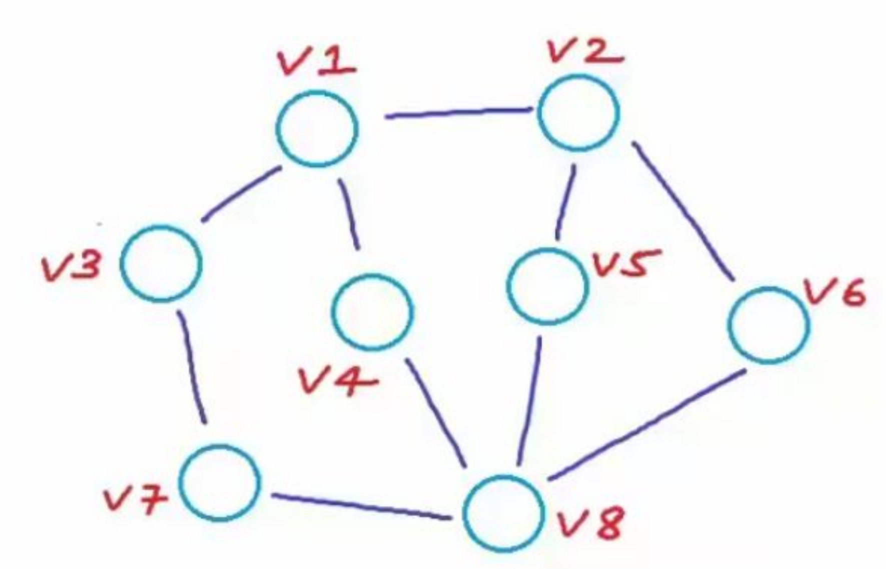


a directed graph
(digraph)



an undirected graph

Graph properties



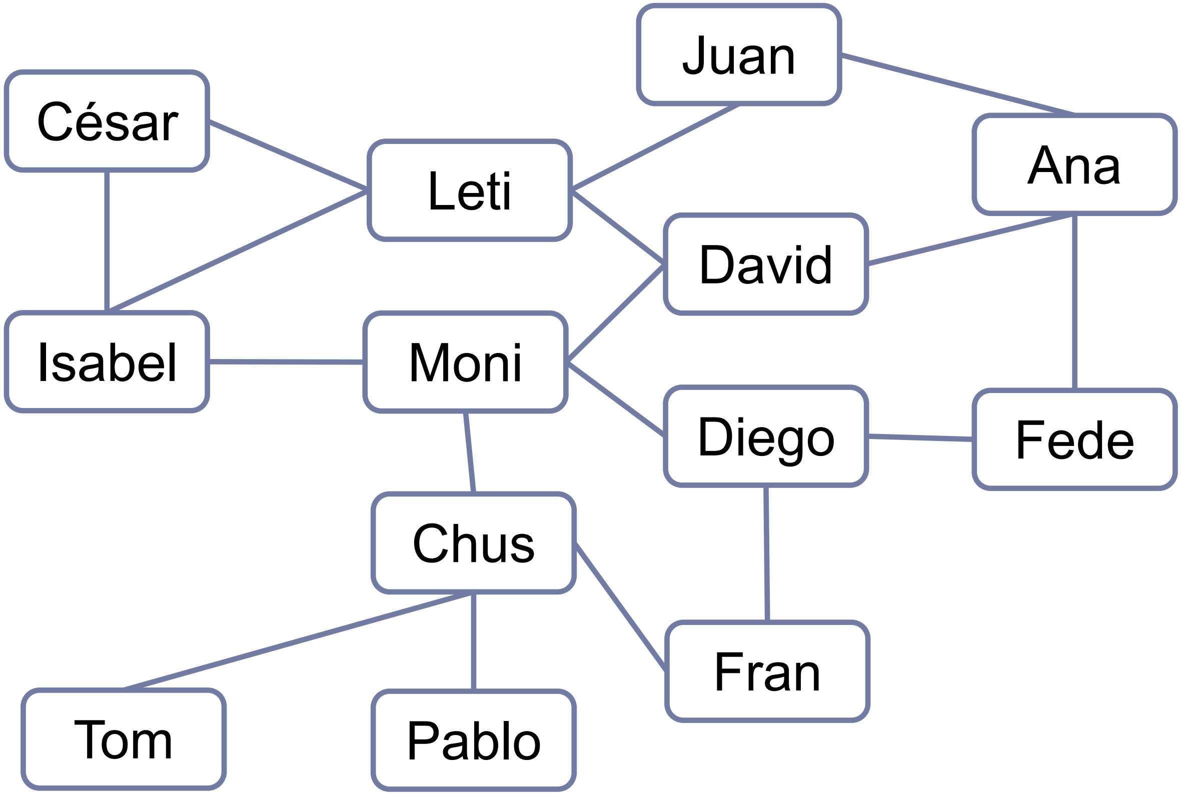
IVI =number of vertices
IEI =number of edges

V = { v1, v2, v3, v4, v5, v6, v7, v8}
E= { {v1, v2}, {v1, v3}, {v1, v4},{v2,v5}, {v2,v6},
{v3,v7}, {v4,v8}, {v5,v8},{v6,v8}, {v7,v8}}

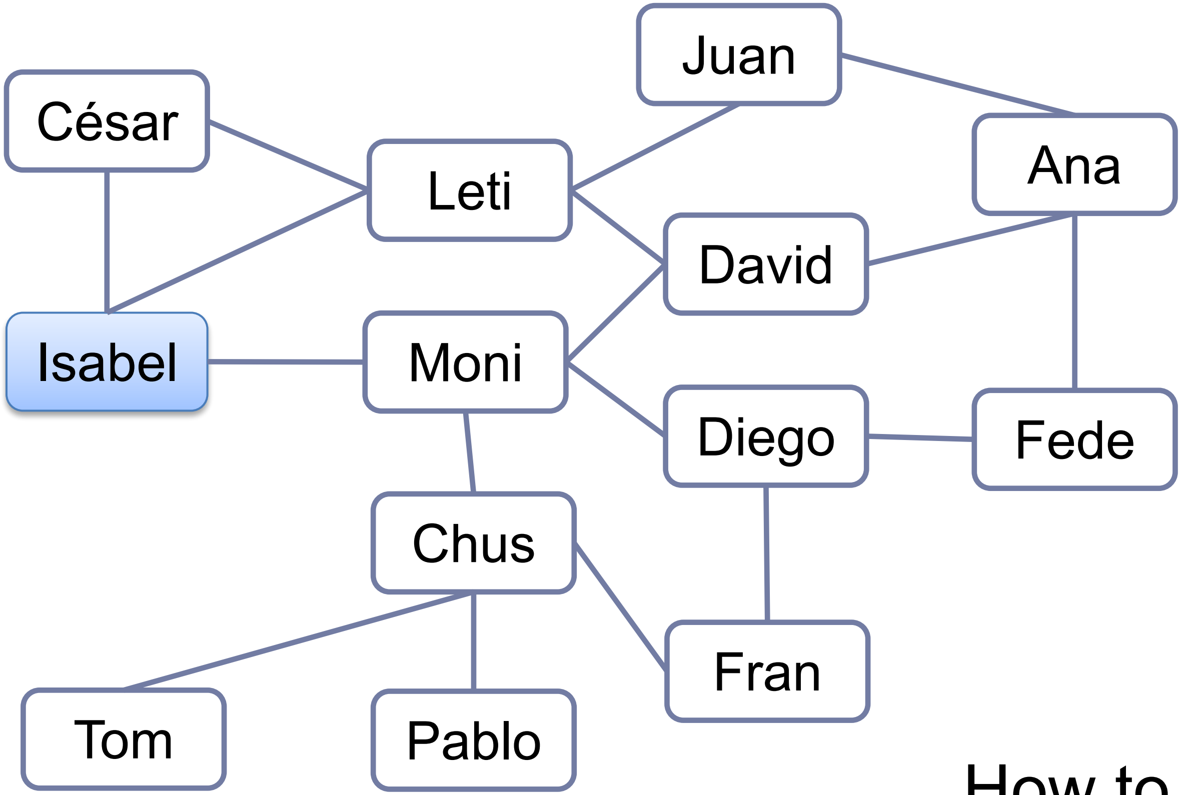
IVI = 8, IEI=10

Graph properties

Social Network (undirected graph)

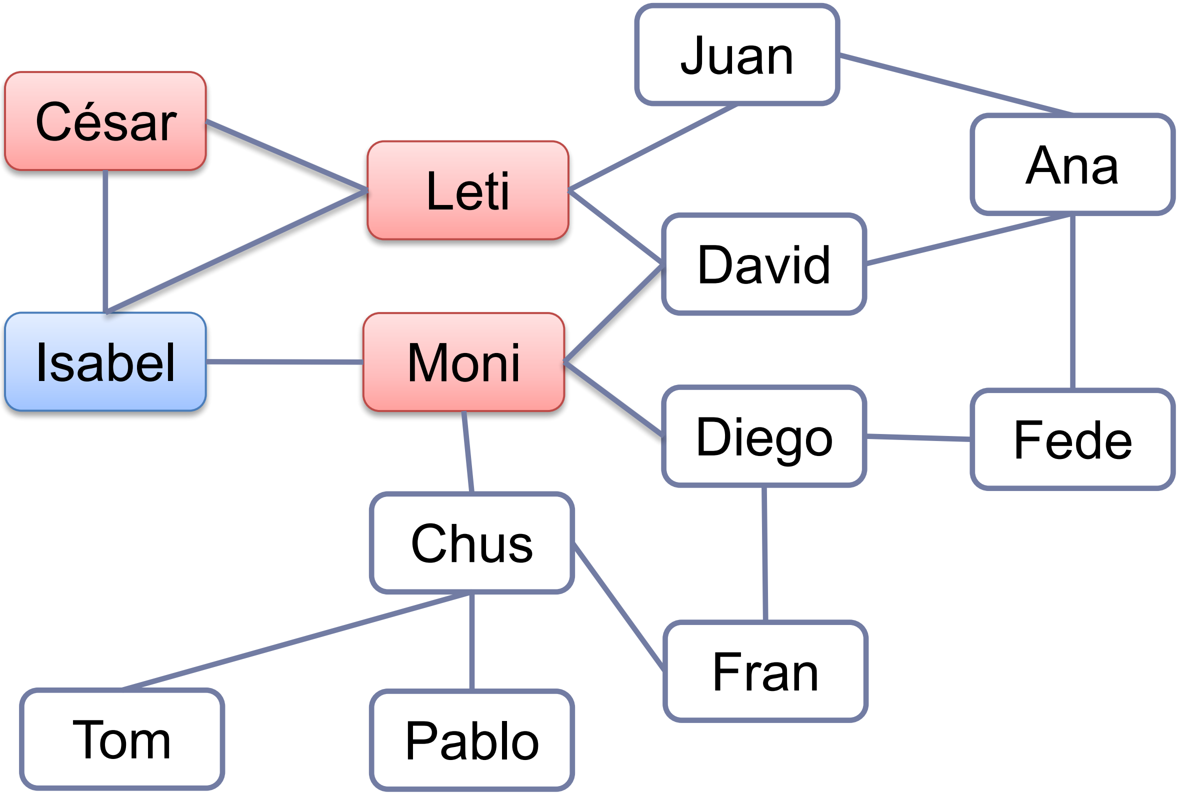


Graph properties

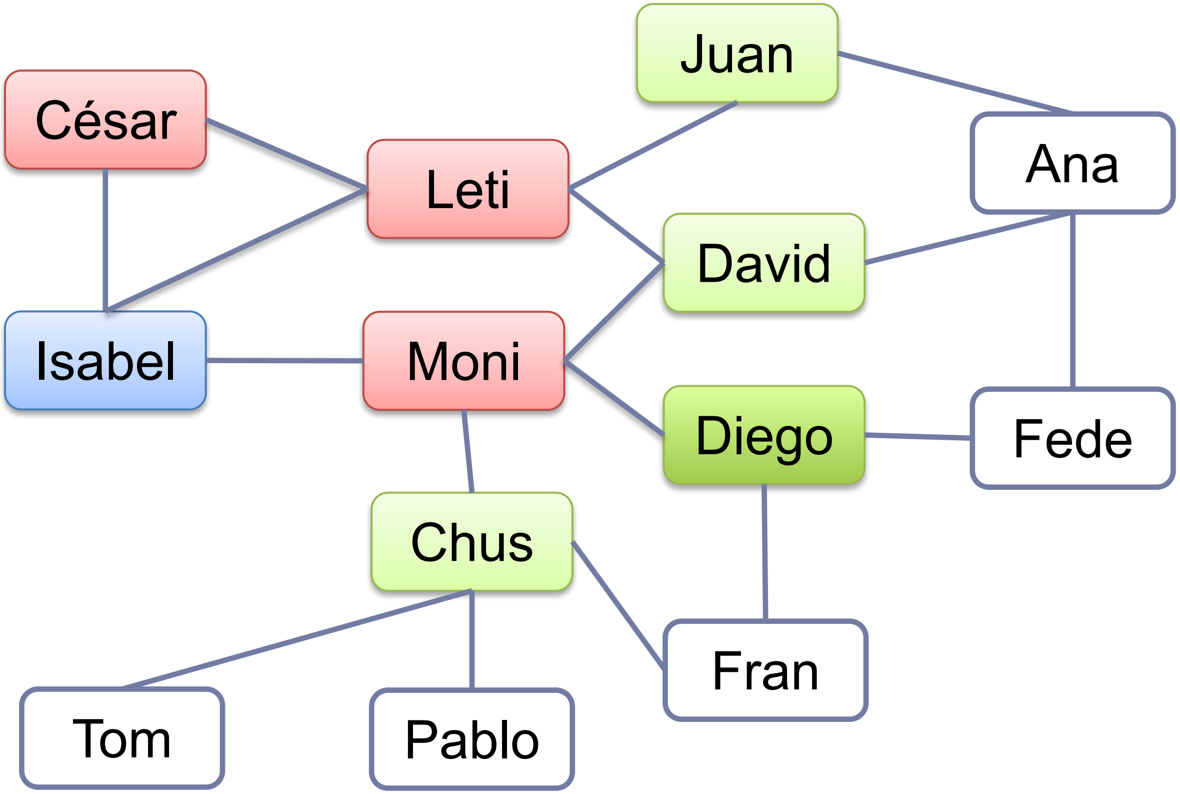


How to suggest some new friends to Isabel?

Graph properties



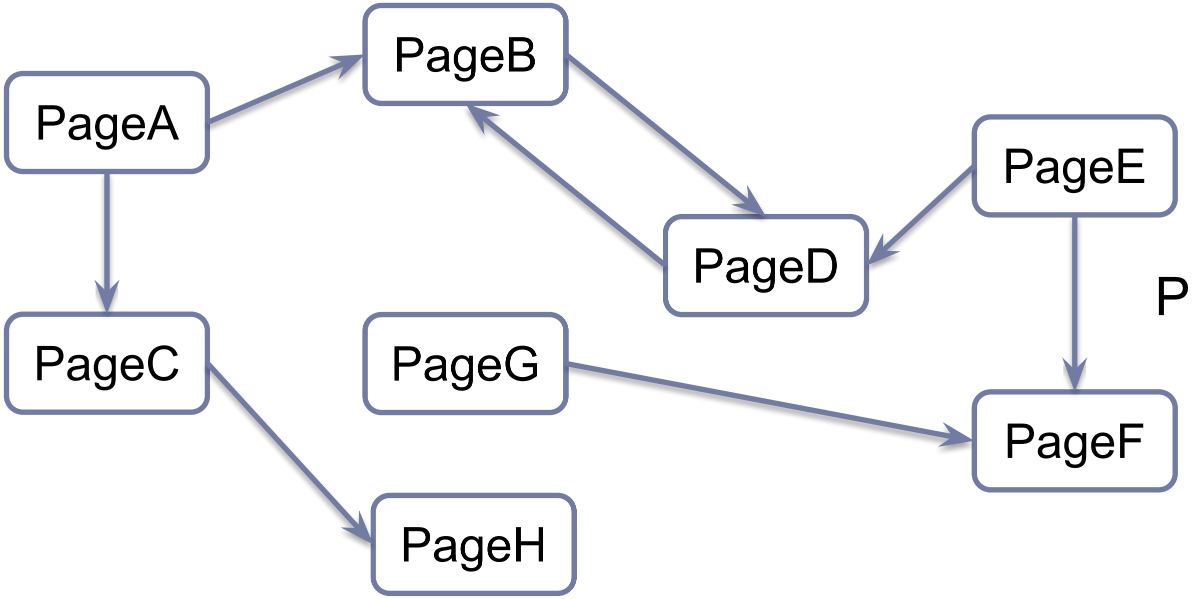
Graph properties



Find all nodes having length of shortest path from Isabel equal to 2

Graph properties

World Wide Web (it’s a directed graph)

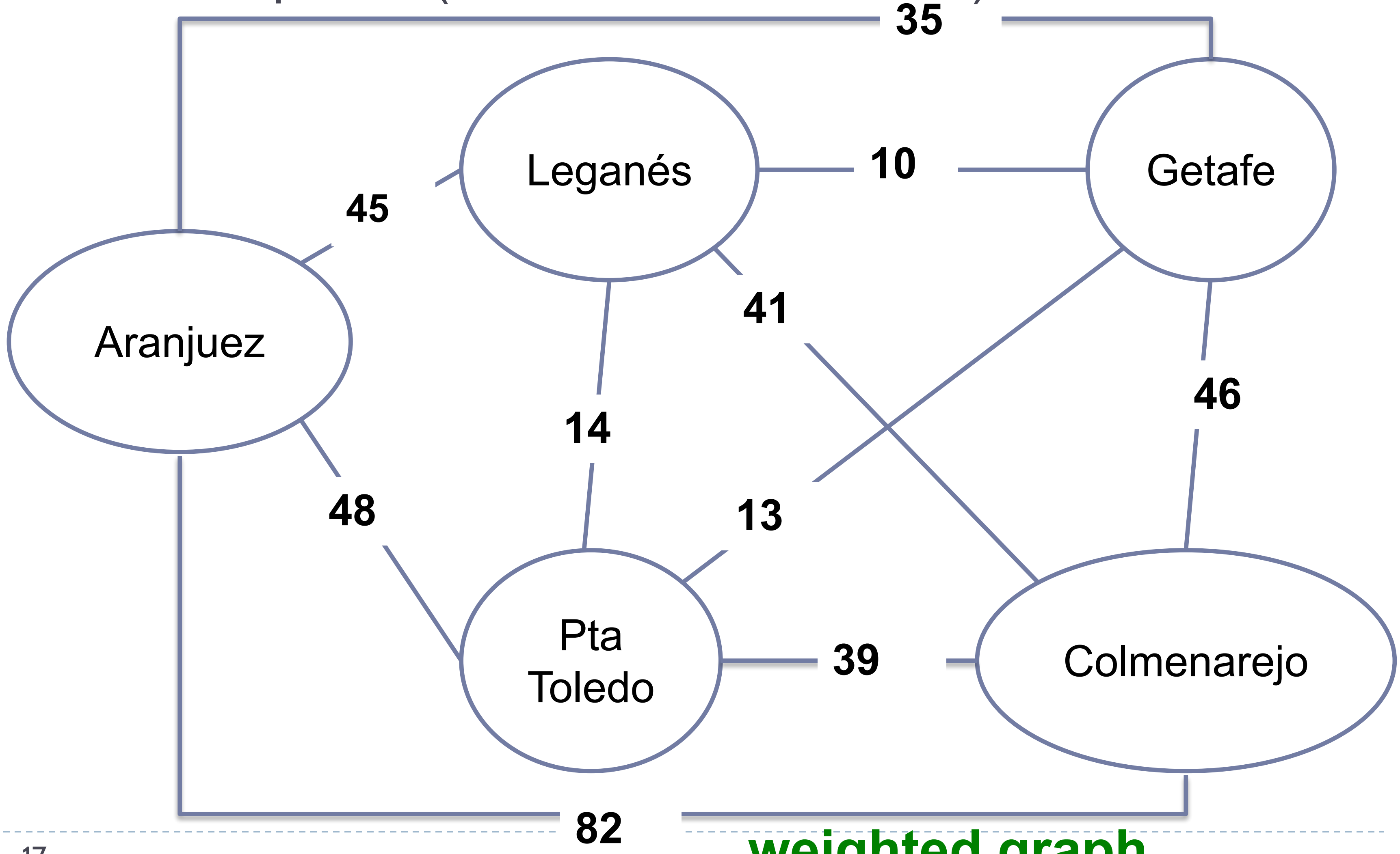


PageE has a link to PageF

Pages as vertices (have a unique URL)

Graph properties

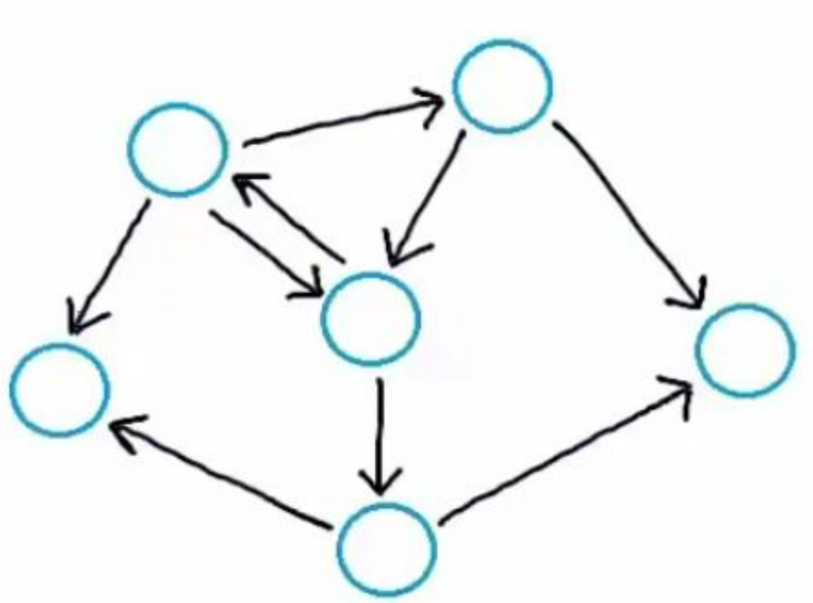
UC3M Campuses (distance in kilometers)



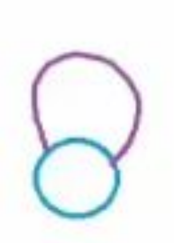
weighted graph

Graph properties

Type of edges



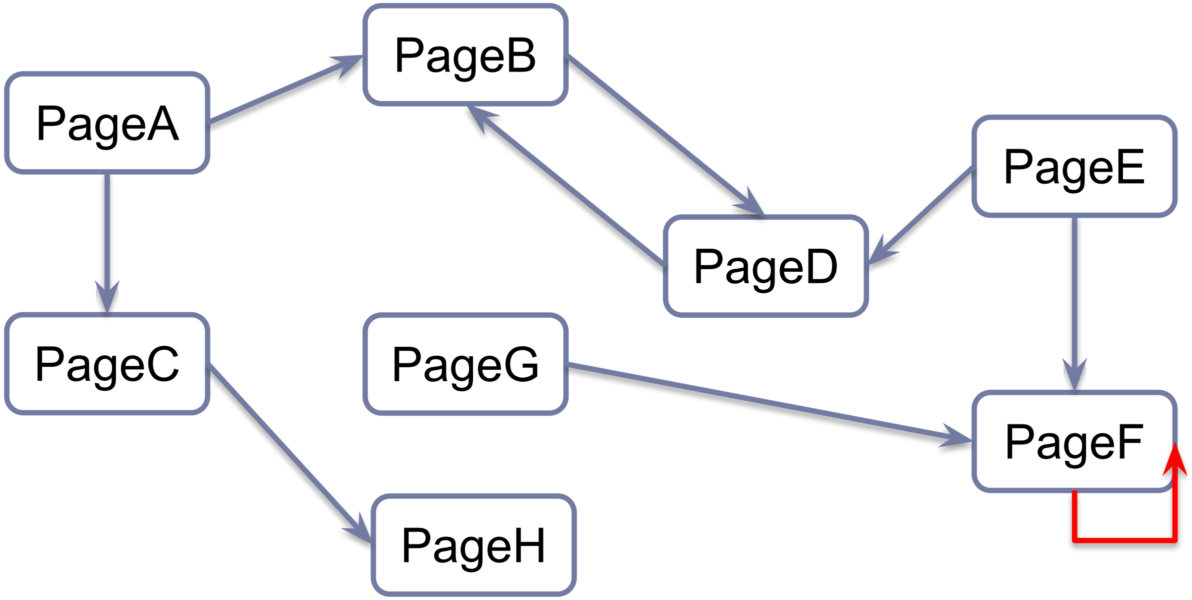
loop



When are they necessary?

Graph properties

World Wide Web

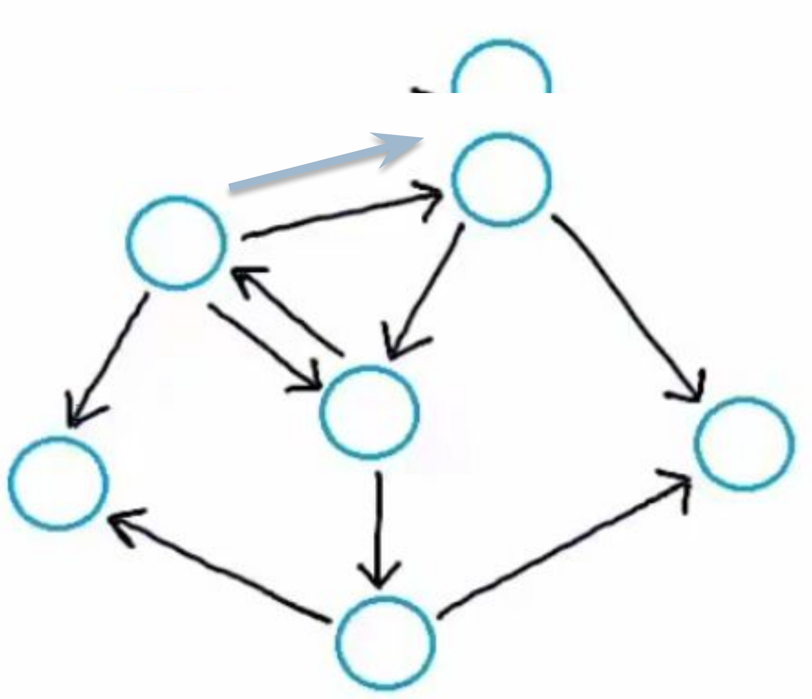


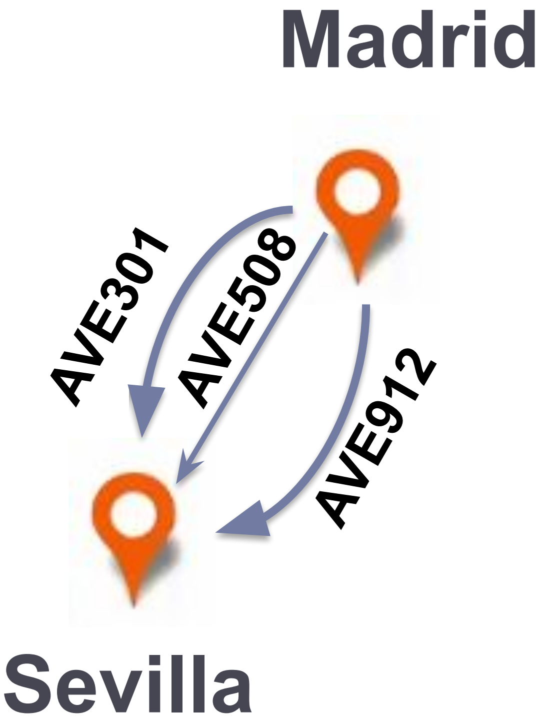
A web page may contain a link to itself

Graph properties

Type of edges

Multi-edge (parallel edges)

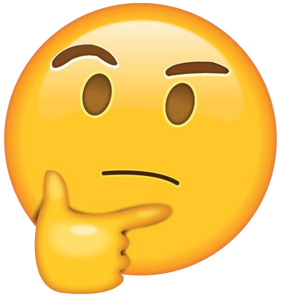




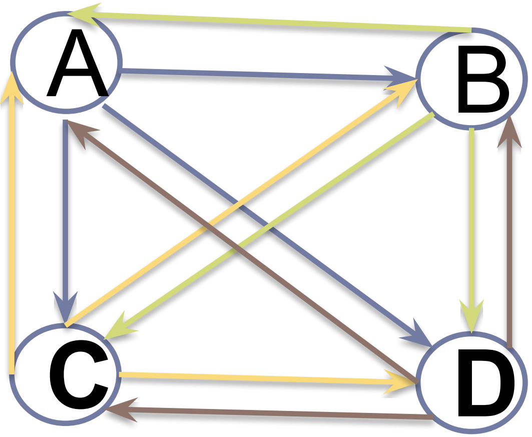
Graph properties

- Loops and parallel edges lead to complicate graph algorithms
- A graph is simple if it has no loops or parallel edges.

Graph properties



What is the maximum possible number of edges in a simple directed graph?



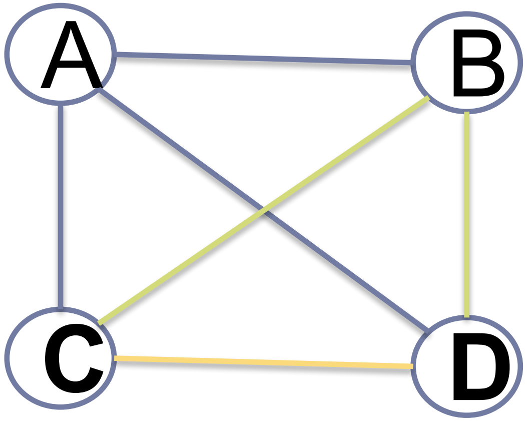
|V| = 4
|E| = 0 (minimum)

|V| = 4
|E| = 12 (maximum)

If /V/ = n, each vertex may have n-1 edges.
Therefore, 0<= /E/<=n(n-1), if directed

Graph properties

What is the maximum possible number of edges in a simple **undirected** graph?



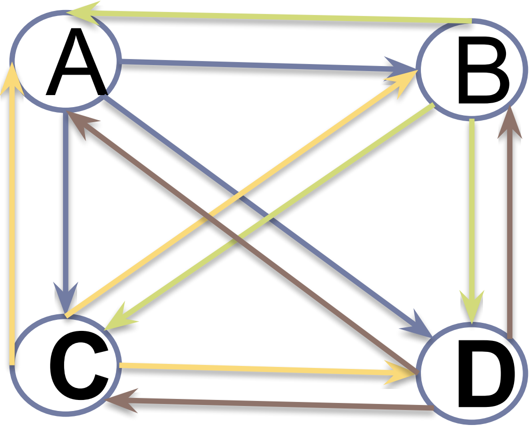
|V| = 4
|E| = 0 (minimum)

|V| = 4
|E| = 6 (maximum)

If /V/ = n, each vertex may have n-1 edges.
Therefore, 0<= /E/<=n(n-1)/2, if directed

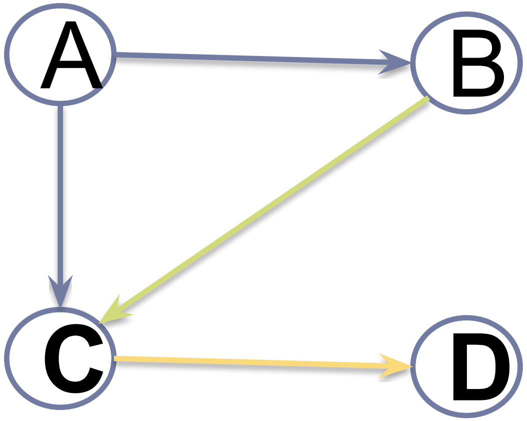
Graph properties

- A graph is dense if the number of its edges is close to its maximum possible number (≈ $|V|^2$)



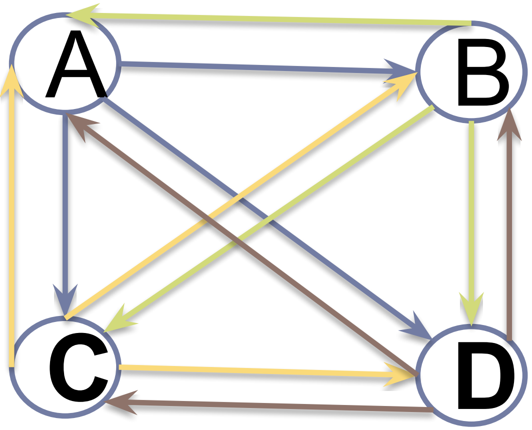
Graph properties

- A graph is sparse if the number of its edges is close to its number of vertices (≈|*V*|)

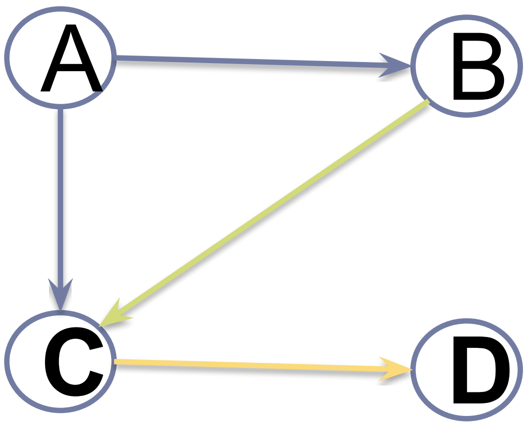


Graph properties

- Knowing if a graph is dense or sparse can help us to select the most appropriate data structure to represent it.



dense

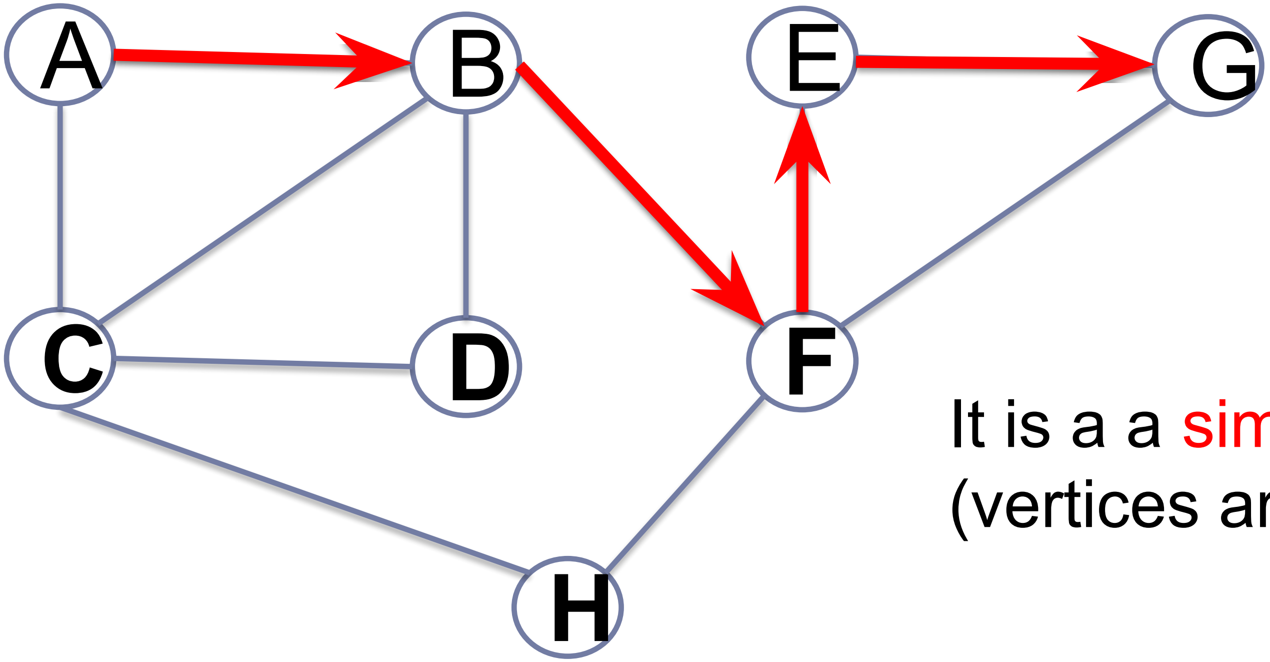


sparse

Graph properties

- Path is a sequence of vertices where each adjacent pair is connected by an edge

<A,B,F,E,G>

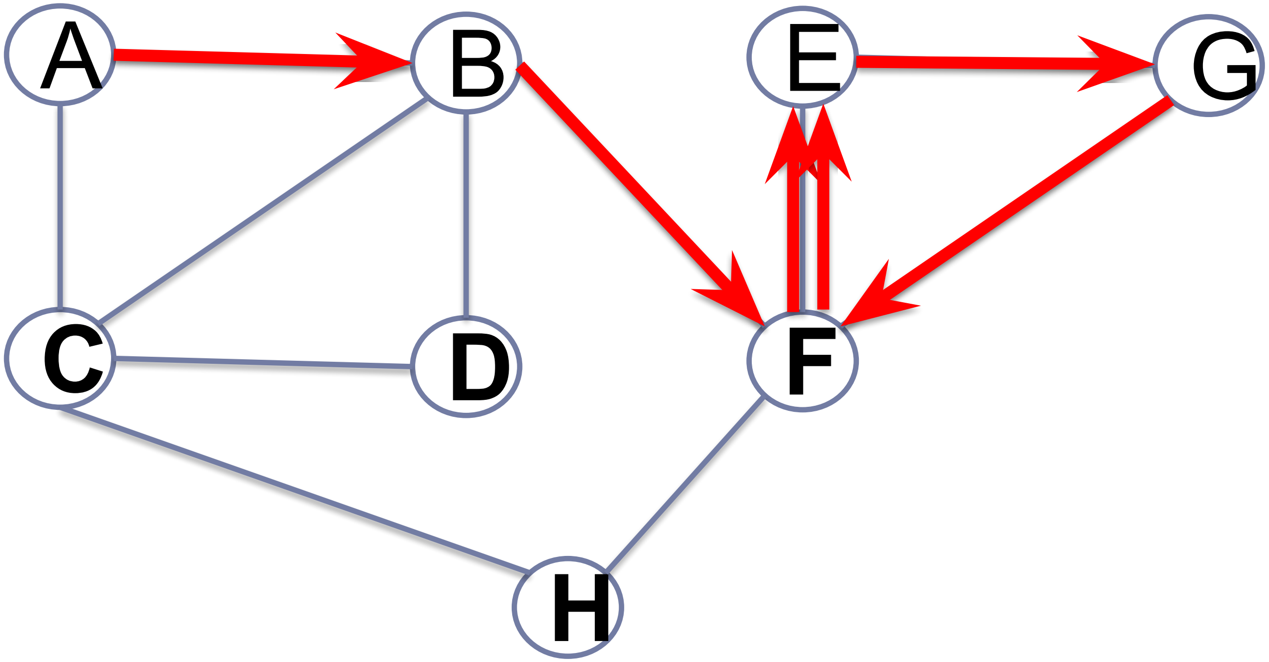


It is a a simple path
(vertices are not repeated)

Graph properties

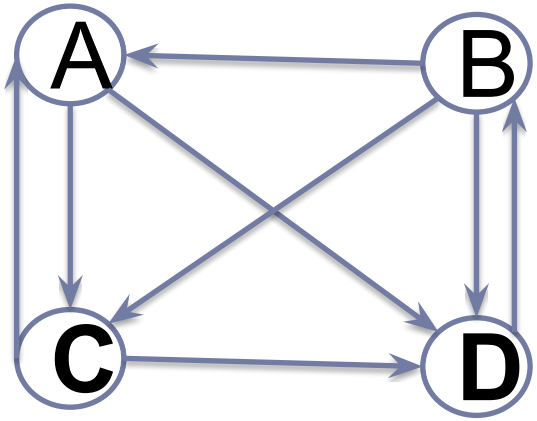
<A,B,F,E,G,F,E>

This is not a simple path (two repeated vertices and one edge)

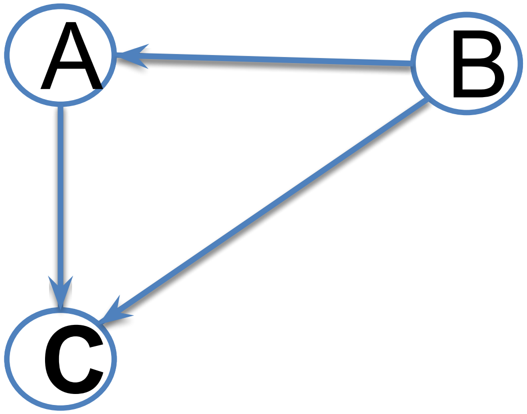


Graph properties

- A graph is strongly connected if there is a path from any vertex to any other vertex.



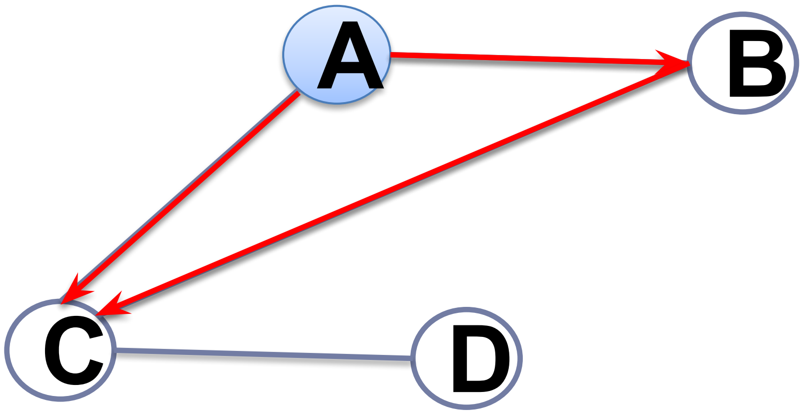
strongly connected



Weakly connected

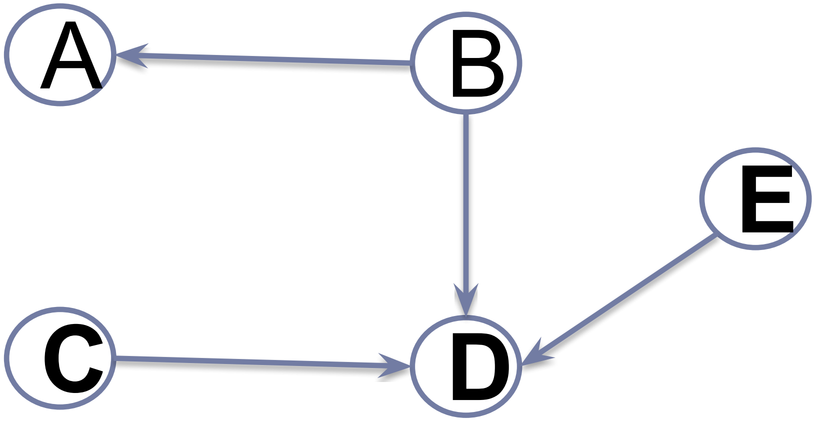
Graph properties

- Simple cycle is a close walk with no repetition other than start and end.

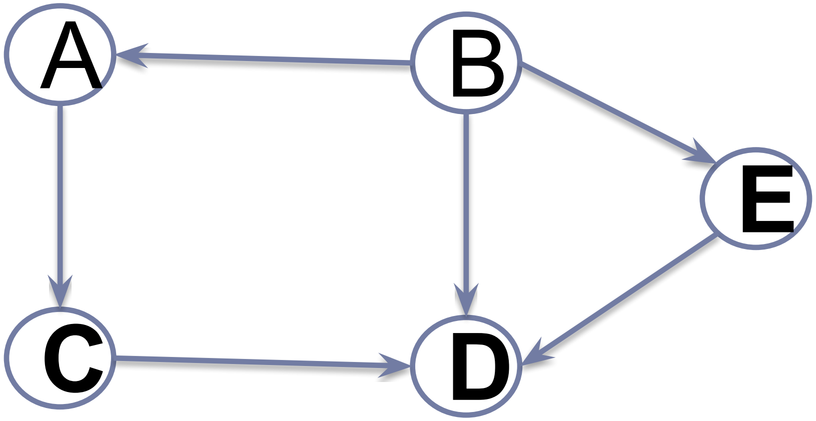


Graph properties

Acyclic graph is a graph with no cycles.



Undirected acyclic graph

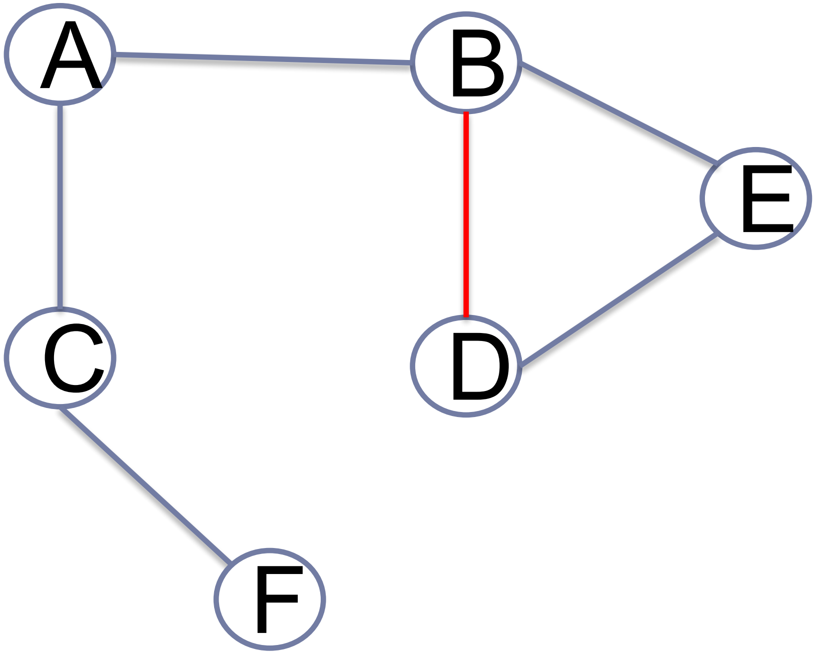


directed acyclic graph (DAG)

Index

- Introduction to Graphs
- Graph properties
- **Graph representation:**
  - Adjacency Matrix.
  - Adjacency List.
- Graph Traversal

Graph representation



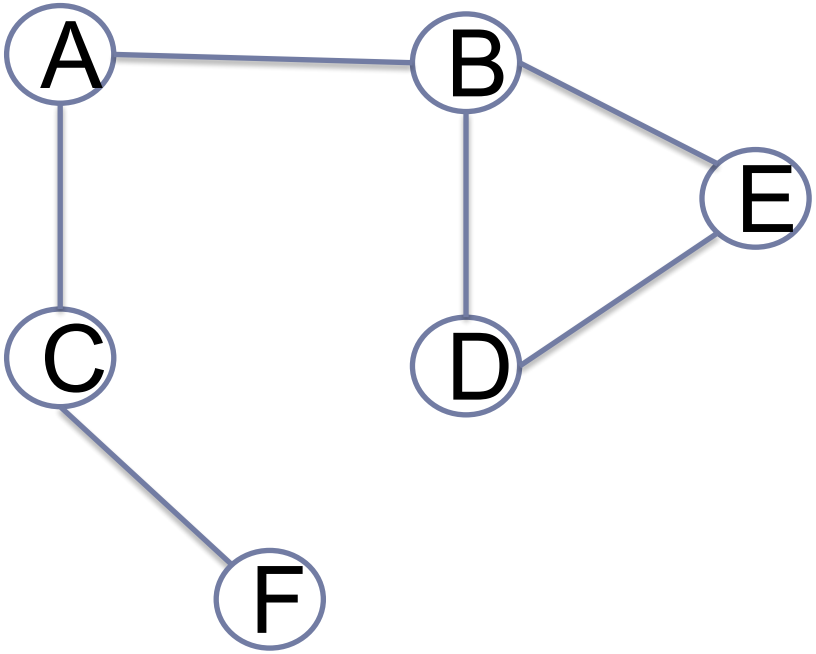
How can we create and store a graph in computer memory?

G=(V,E), V vertices, E edges

Index

- Introduction to Graphs
- Graph properties
- **Graph representation:**
  - **Adjacency Matrix.**
  - Adjacency List.
- Graph Traversal

Graph representation: Adjacency Matrix

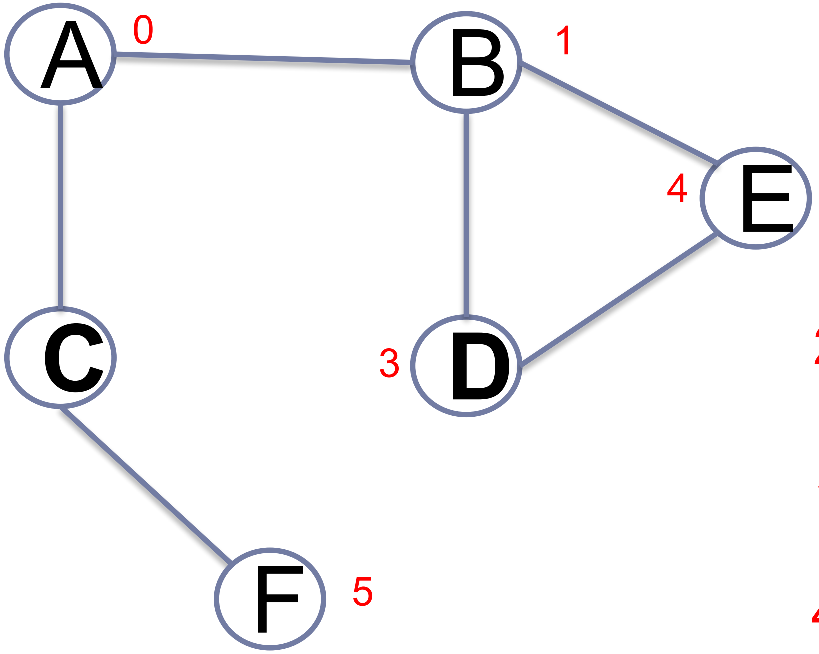


Vertex list

0	A
1	B
2	C
3	D
4	E
5	F

We can use a Python list to store the vertices.
Each vertex is represented by an index.

Graph representation: Adjacency Matrix

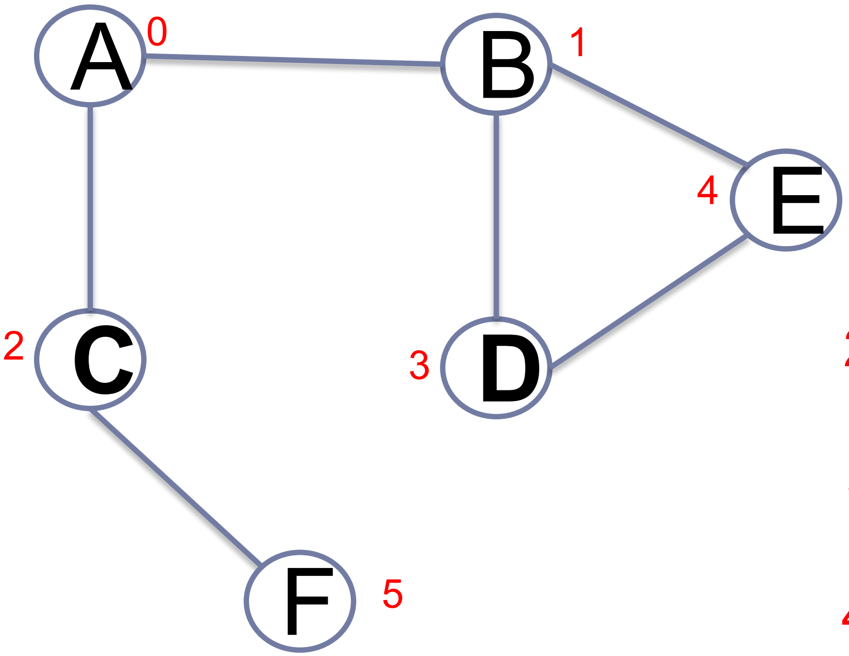


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

$$Mij = \begin{cases} 1, & \text{if } \{i, j\} \text{ is an edge} \\ 0, & \text{otherwise} \end{cases}$$

Graph representation: Adjacency Matrix

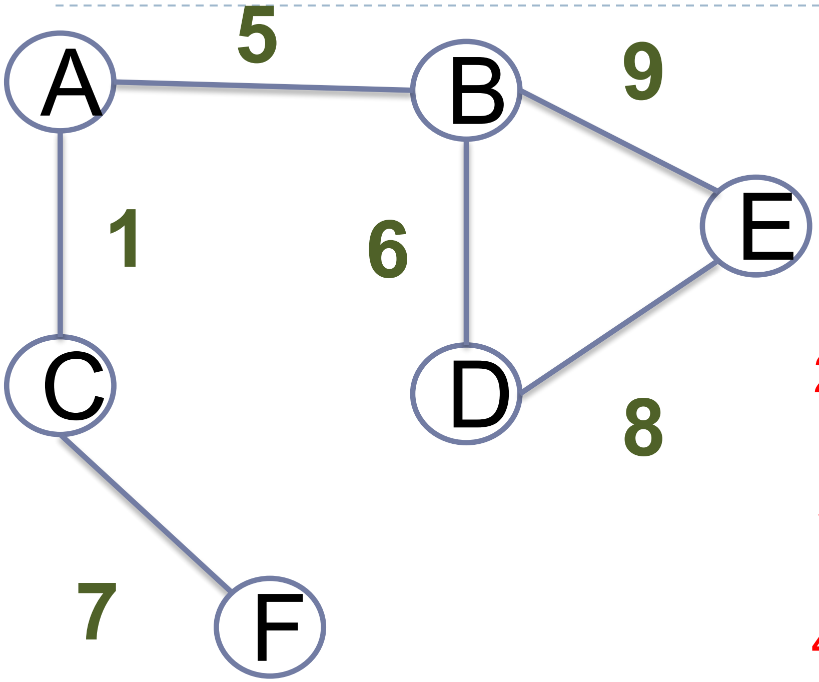


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

undirected graph
Mij = Mji

Graph representation: Adjacency Matrix

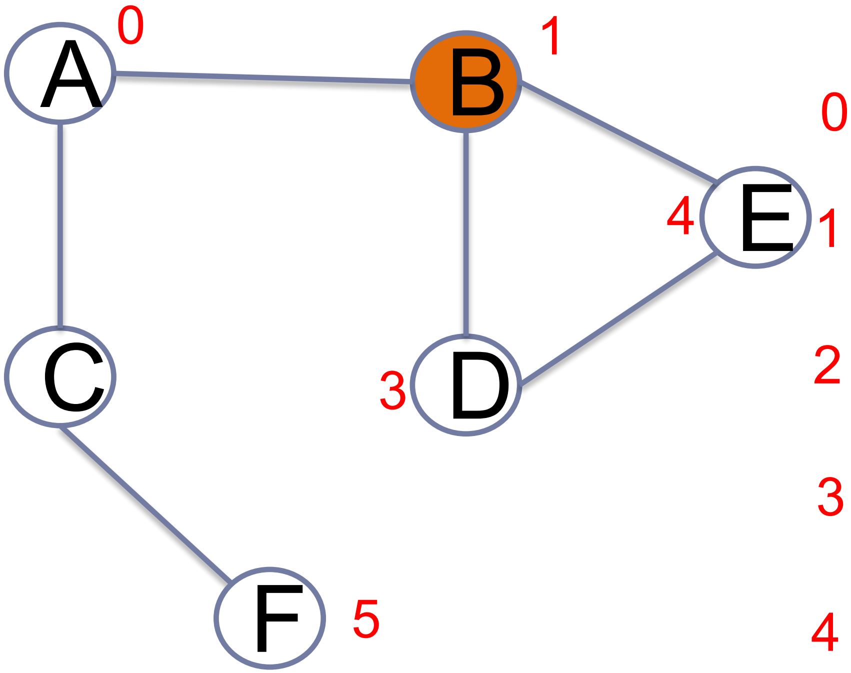


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	∞	5	1	∞	∞	∞
1	5	∞	∞	6	9	∞
2	1	∞	∞	∞	∞	7
3	∞	6	∞	∞	8	∞
4	∞	9	∞	8	∞	∞
5	∞	∞	7	∞	∞	∞

Representation of weighted graph

Graph representation: Adjacency Matrix



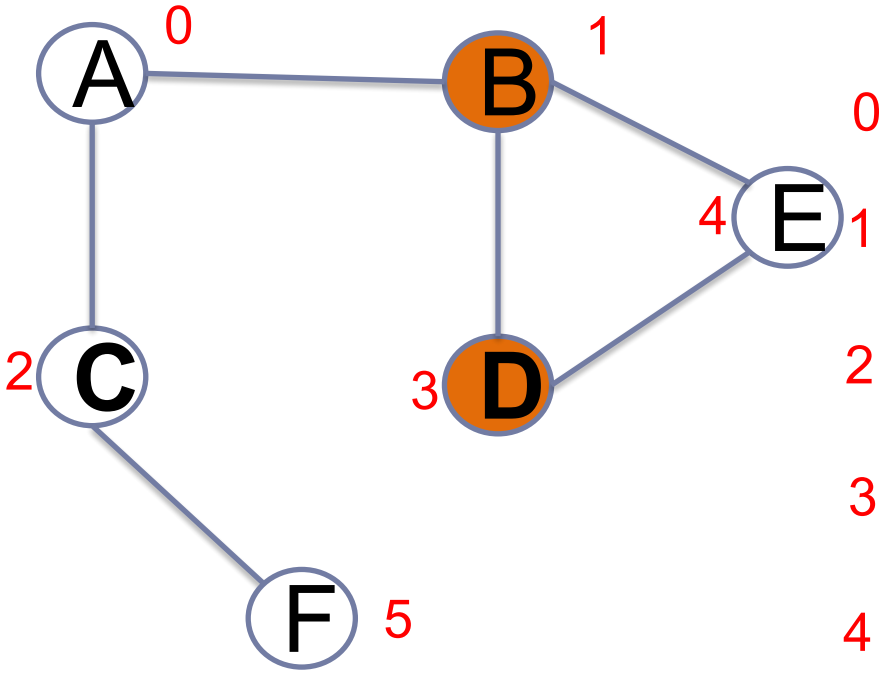
0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

|V| = n

Operations:	**Time complexity**
Finding adjacent nodes	O(n)

Graph representation: Adjacency Matrix



|V| = n

0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

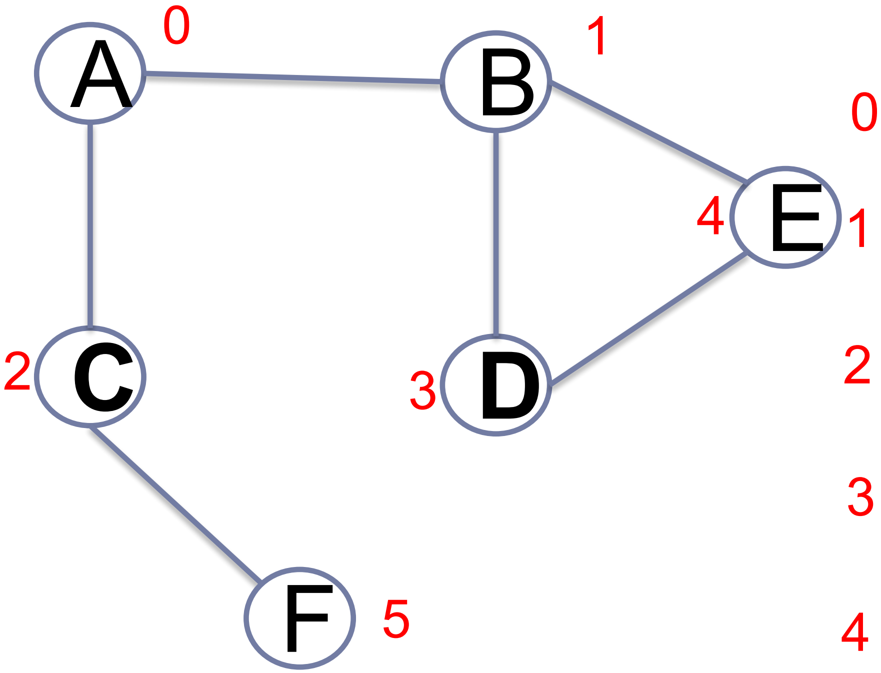
Operations:

Checking if two given nodes are adjacent (M(1,3)?)

Time complexity

O(1)

Graph representation: Adjacency Matrix



0	A	0	0	1	1	0	0	0
1	B	1	1	0	0	1	1	0
2	C	2	1	0	0	0	0	1
3	D	3	0	1	0	0	1	0
4	E	4	0	1	0	1	0	0
5	F	5	0	0	1	0	0	0

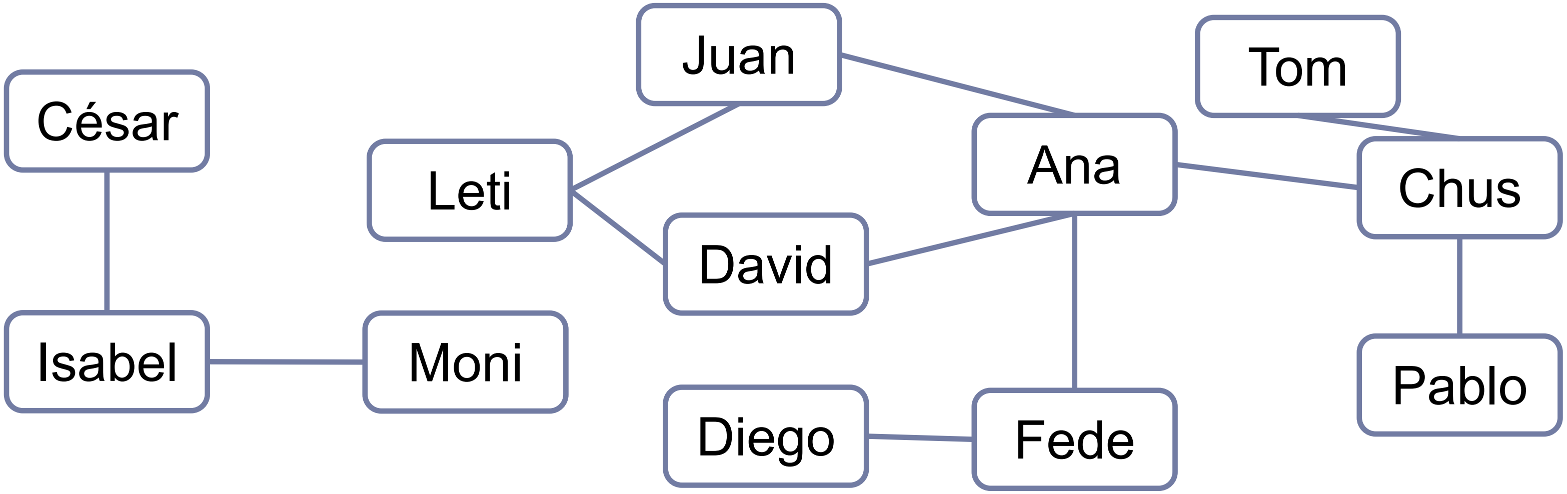
Space complexity:

If |V| = n, $O(n^2)$

Graph representation: Adjacency Matrix

- In terms of time complexity, adjacency matrix is an efficient data structure.
- However, in terms of space complexity, it is too costly.
- Adjacency matrix is a good representation when n^2 is small or the graph is dense.
- However, most real graphs are sparse (for example, WWW).

Graph representation: Adjacency Matrix



If $|V| = 10^9$ space= 10^{18}

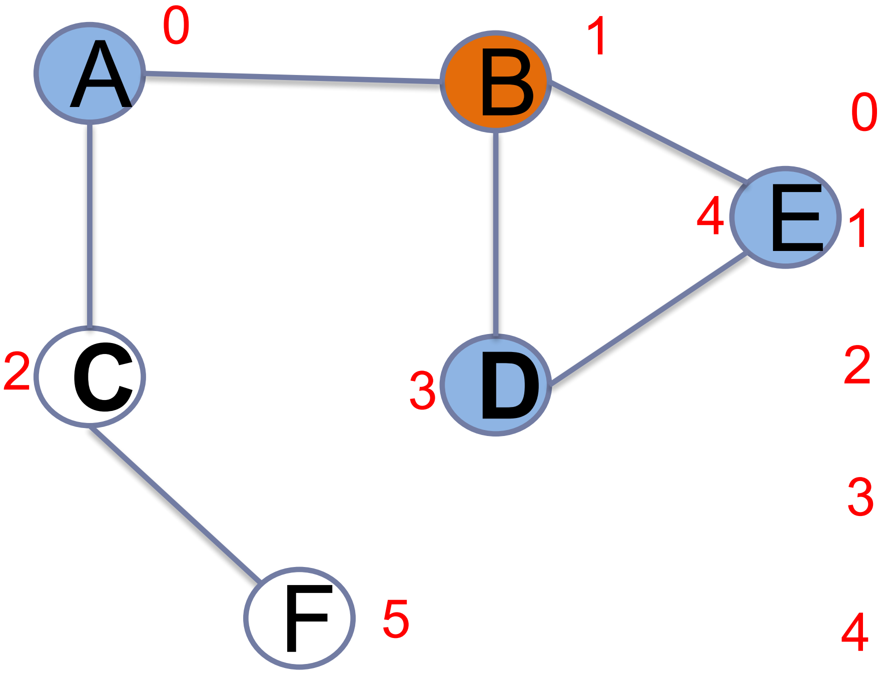
Suppose avg. number of friends ≈ 1000

$|E| = (10^9 * 10^3)/2 = 10^{12}/2^{<<} 10^{18}$

Index

- Introduction to Graphs
- Graph properties
- **Graph representation:**
  - Adjacency Matrix.
  - **Adjacency List.**
- Graph Traversal

Graph representation: Adjacency Matrix



|V|=6, matrix=6x6

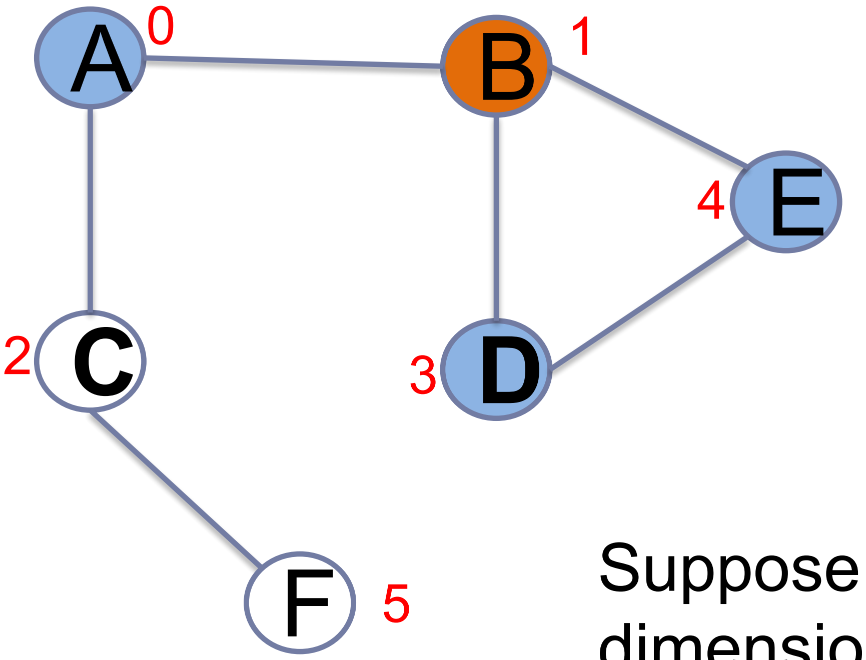
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

Adjacent vertices for B?

0	1	2	3	4	5
1	0	0	1	1	0

List of size 6

Graph representation: Adjacency List



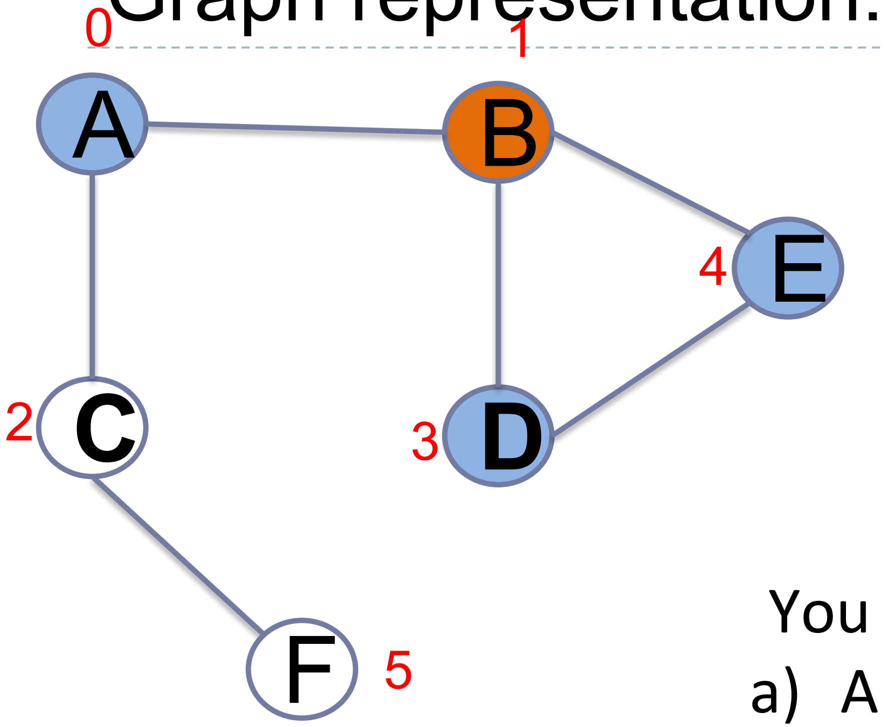
Connections for B:

0	1	2	3	4	5
1	0	0	1	1	0

Suppose Facebook has 10^9 users
dimension of row is 10^9

If B has 1000 friends:
Numbers of 1: 1000 **≈ 1 KB**
Numbers of 0: 10^9- 1000 **≈ 1 GB**

Graph representation: Adjacency List



Connections for B:

0	1	2	3	4	5
1	0	0	1	1	0

You can use:

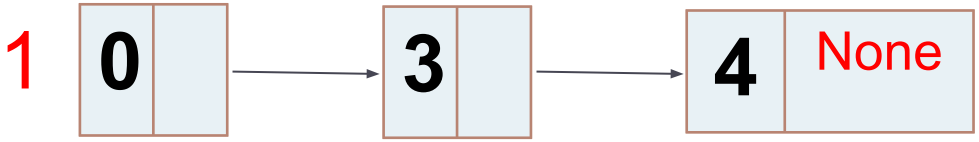
a) A Python List, or

b) A Linked List

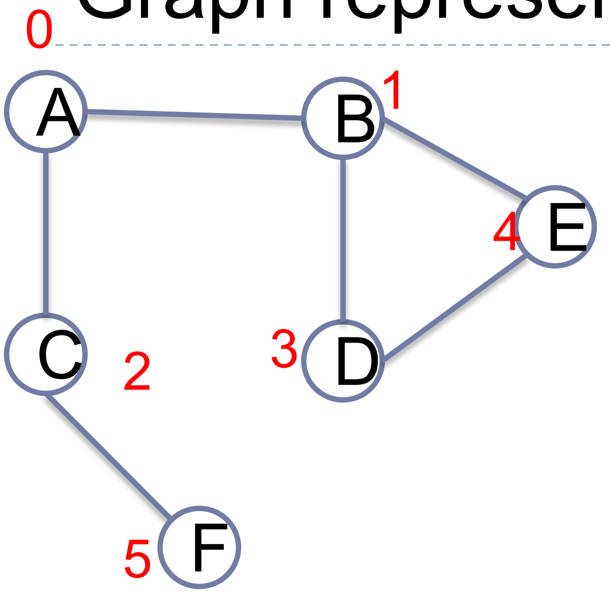
Python List

0	3	4

Linked List



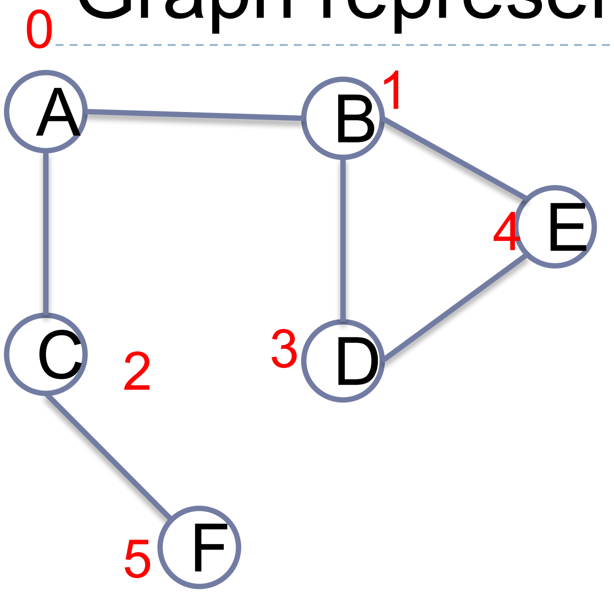
Graph representation: Adjacency List

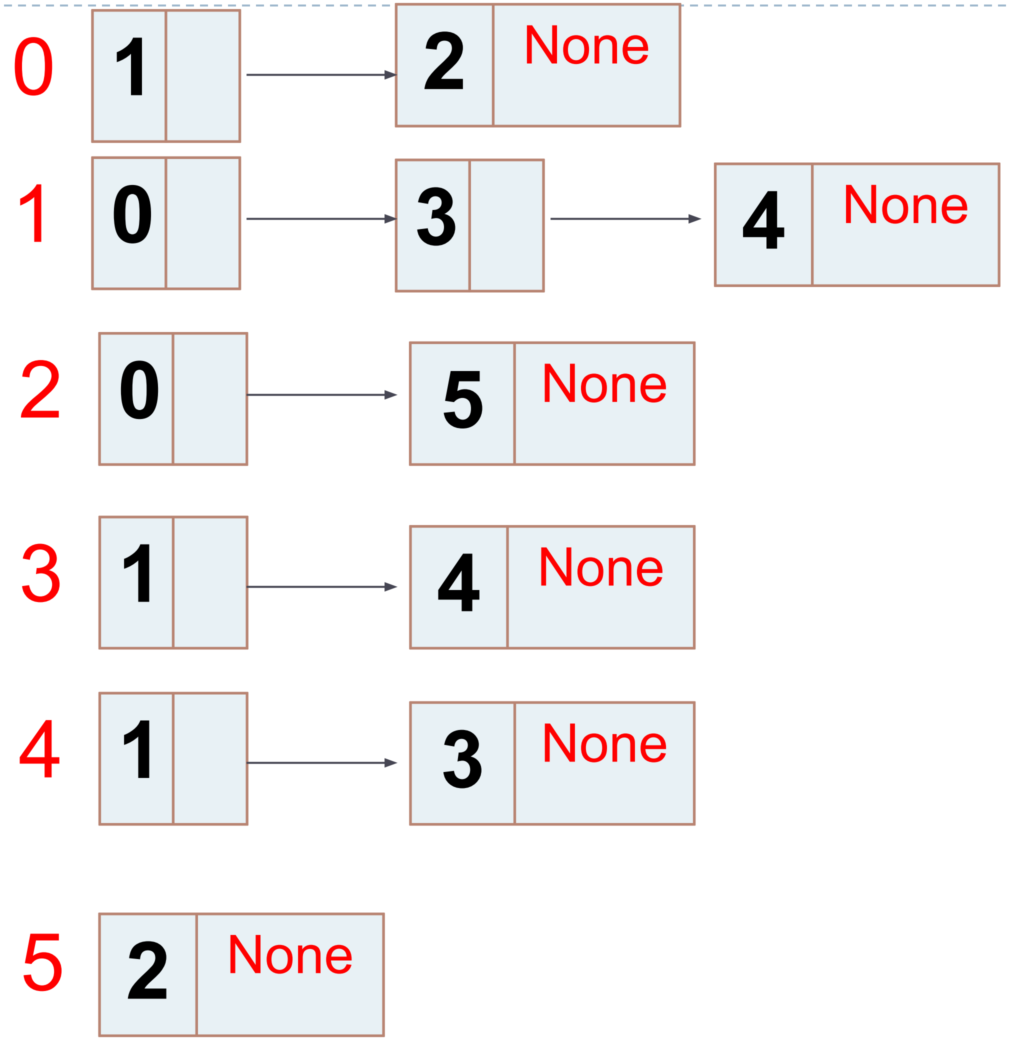


0	A	0	1	2	
1	B	1	0	3	4
2	C	2	0	5	
3	D	3	1	4	
4	E	4	1	3	
5	F	5	2		

Adjacency list can be represented as a list of lists

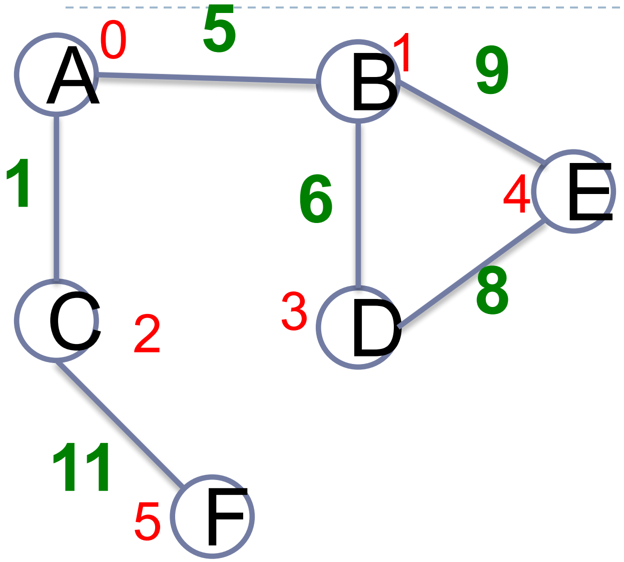
Graph representation: Adjacency List





Adjacency list
= List of Linked Lists

Graph representation: Adjacency List

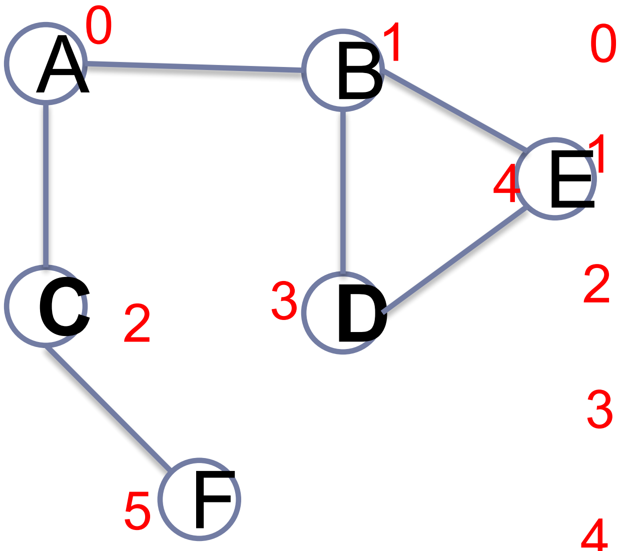


0	(1,5)	(2,1)	
1	(0,5)	(3,6)	(4,9)
2	(0,1)	(5,11)	
3	(1,6)	(4,8)	
4	(1,9)	(3,8)	
5	(2,11)		

Adjacency list
(weighted graph)

Each adjacent vertex is represented with a pair (i,j) where i is the index of the vertex and j the related weight.

Graph representation: Matrix versus List



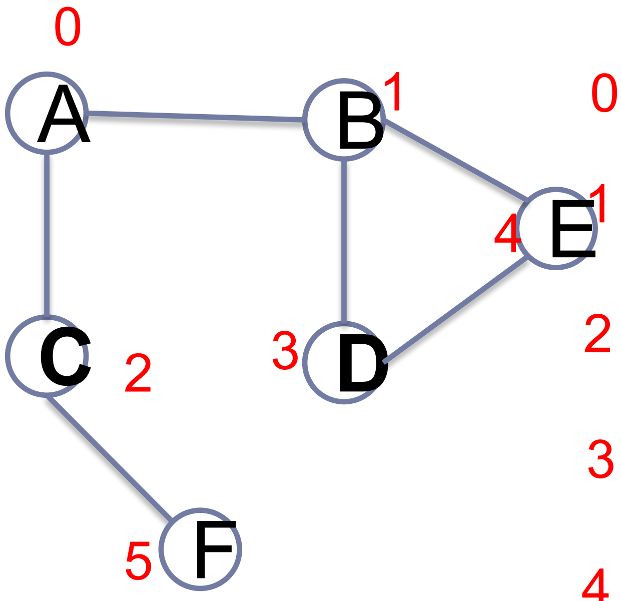
	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

0	1	2	
1	0	3	4
2	0	5	
3	1	4	
4	1	3	
5	1		

Operations:

✔ adjacent nodes for i?	O(n)	O(1)
✔ (i,j) is an edge?	O(1)	O(n)

Graph representation: Matrix versus List



	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

0	1	2	
1	0	3	4
2	0	5	
3	1	4	
4	1	3	
5	1		

Space = O(n^2)

Space = O(e)

Most real graphs are sparse ($|E| \approx |V| << |V|^2$)

Graph representation

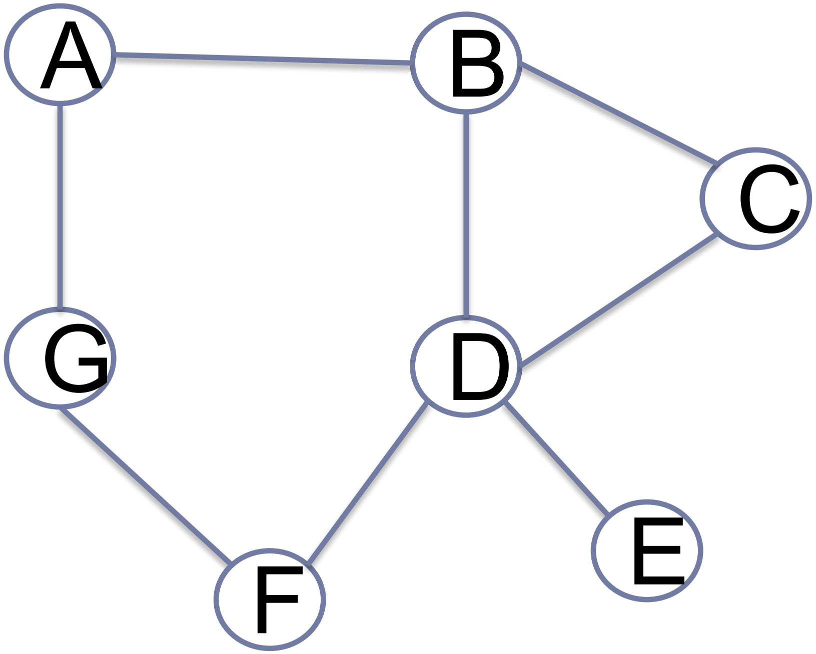
- Most real graphs are sparse (|E|≈ |V|<<$|V|^2$)
- Adjacency matrix, space complexity O($|V|^2$), time complexity O(|V|) (sometimes O(1)). It is a good solution when the graph is dense or n^2 is small.
- Adjacency list, space complexity: O(|E|) <<O($|V|^2$) (if graph is sparse). Time complexity: O(|V|)

Index

- Introduction to Graphs
- Graph properties
- Graph representation:
  - Adjacency Matrix.
  - Adjacency List.
- **Graph Traversal**
  - Breadth-first Traversal
  - Depth-first Traversal

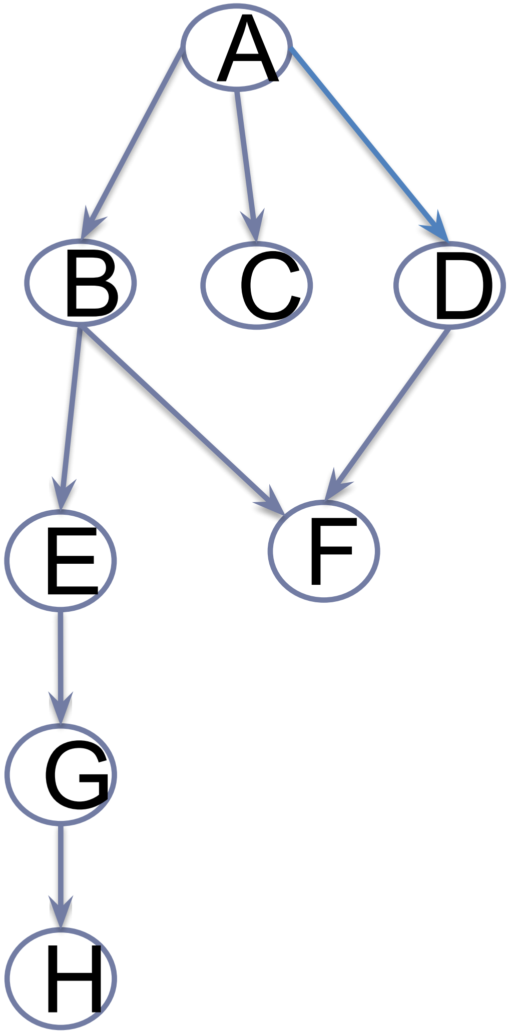
Graph traversal

Visiting all the nodes of the graph



Traveling Salesman Problem (TSP)

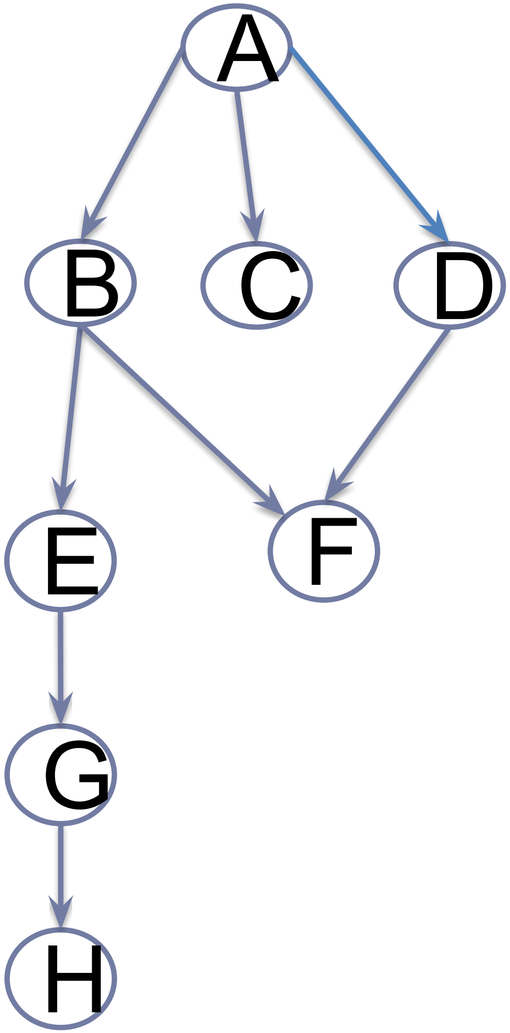
Graph traversal



Visiting all the nodes of the graph

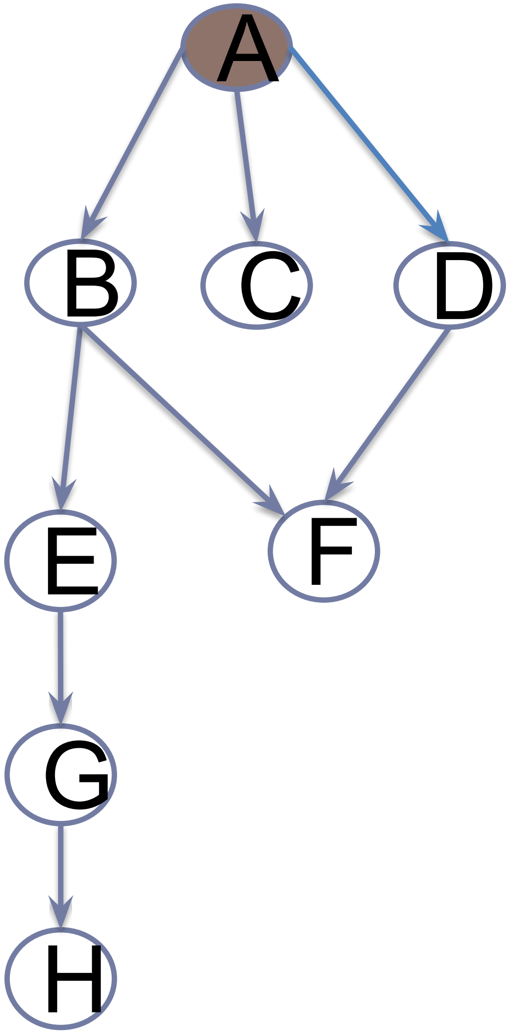
1) Breadth-first traversal (BFS)
2) Depth-first traversal (DFS)

Graph traversal: Breadth-first traversal (BFS)



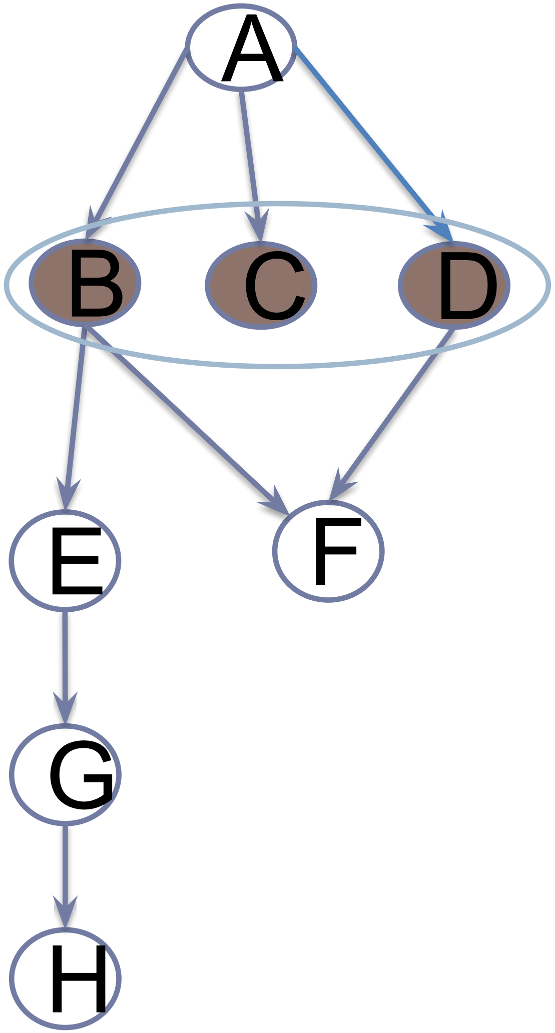
Idea: visit nodes in layers (levels). It’s similar to Level-order traversal in trees

Graph traversal: Breadth-first traversal (BFS)



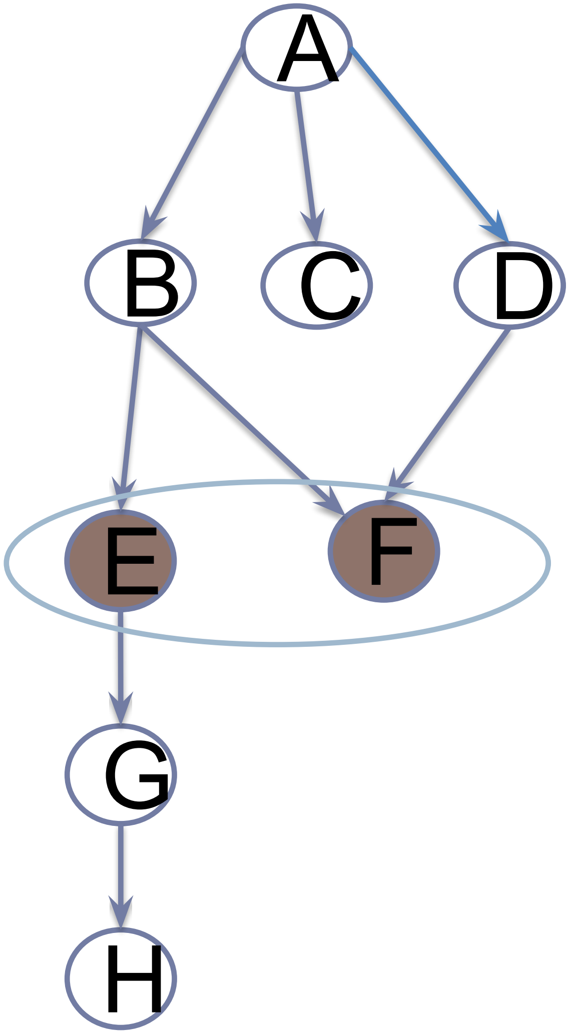
Output: A

Graph traversal: Breadth-first traversal (BFS)



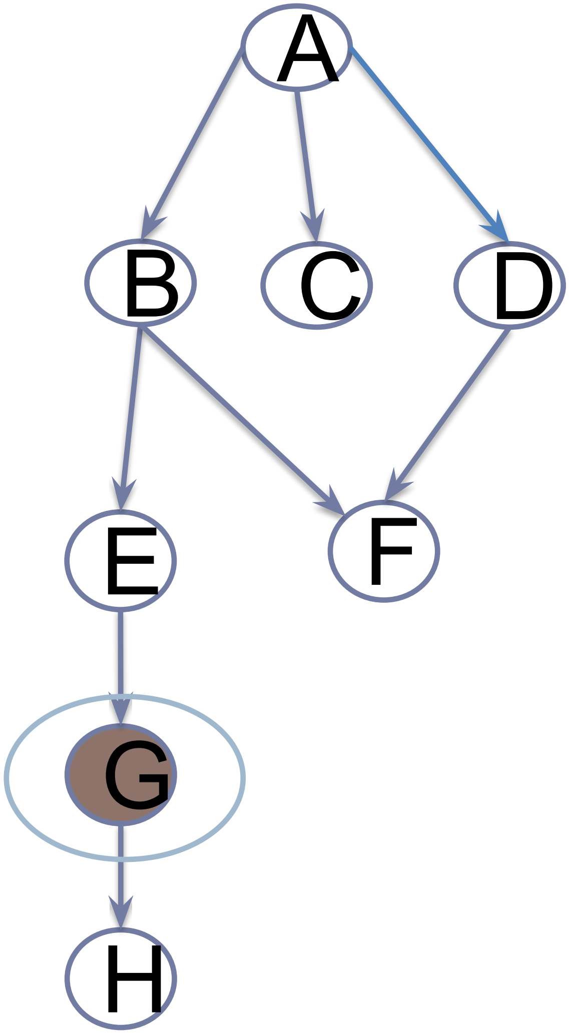
Output: A B C D

Graph traversal: Breadth-first traversal (BFS)



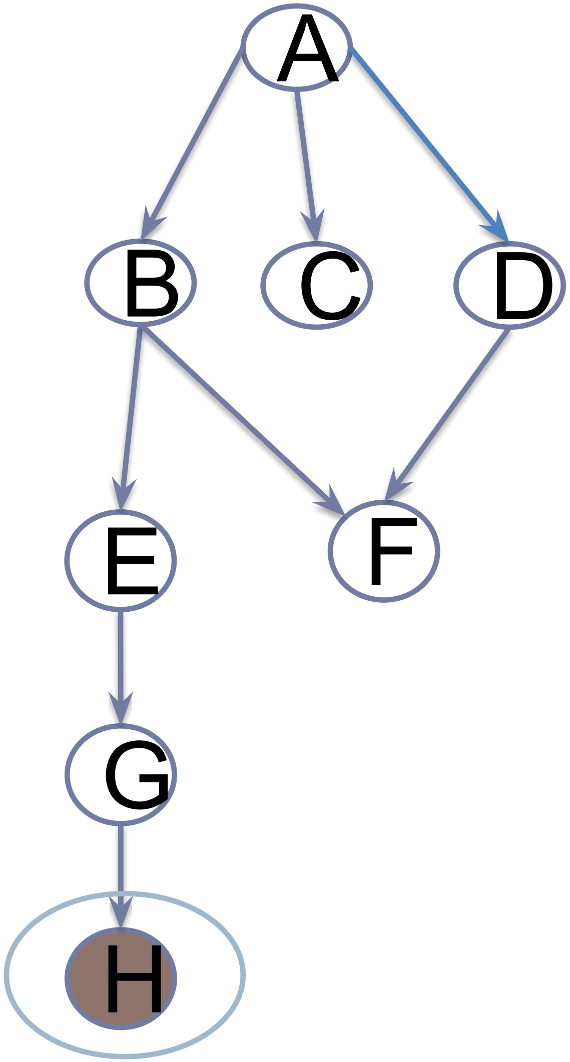
Output: A B C D E F

Graph traversal: Breadth-first traversal (BFS)



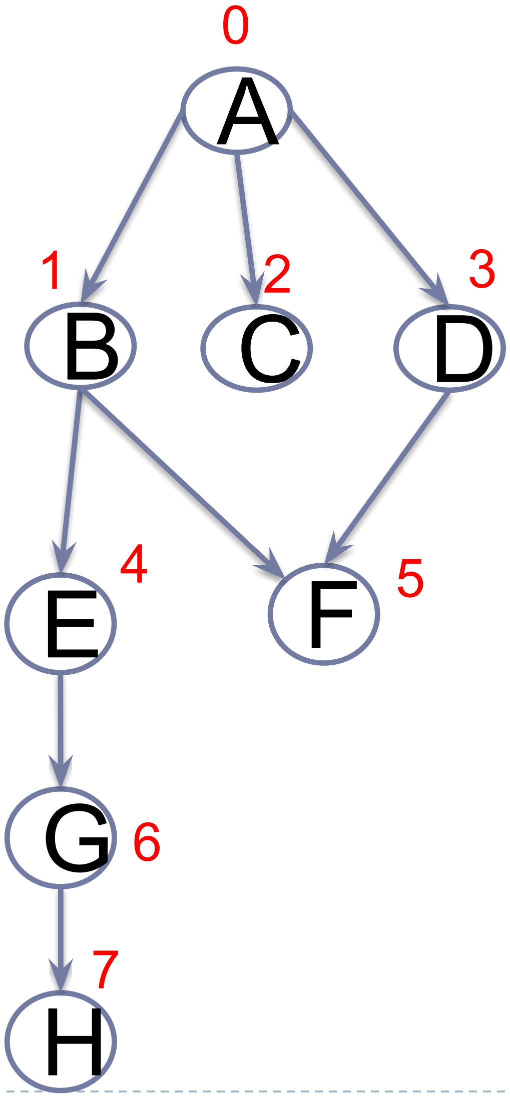
Output: A B C D E F G

Graph traversal: Breadth-first traversal (BFS)



Output: A B C D E F G H

Graph traversal: Breadth-first traversal (BFS)

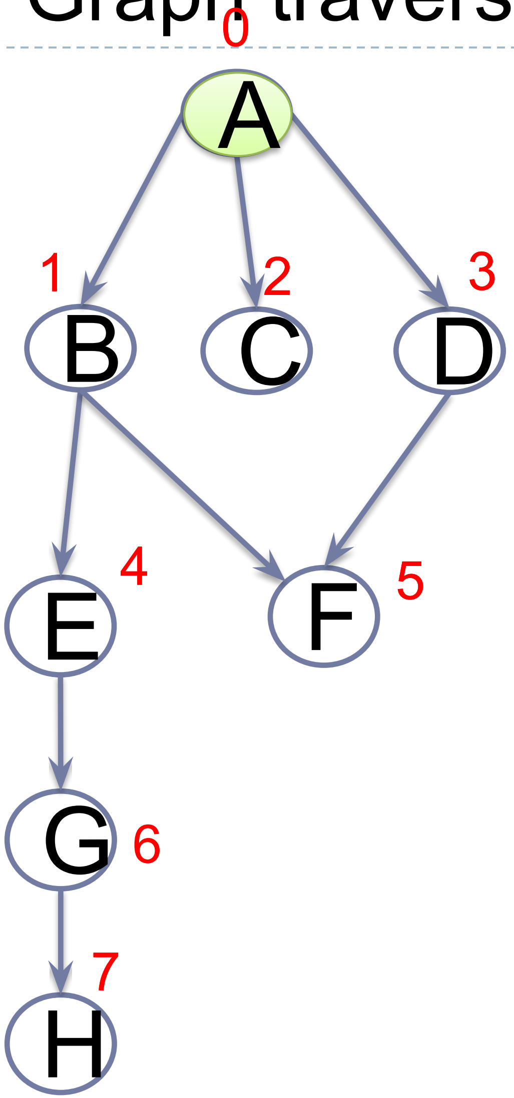


- Queue
- A list to store the visited nodes

q

visited

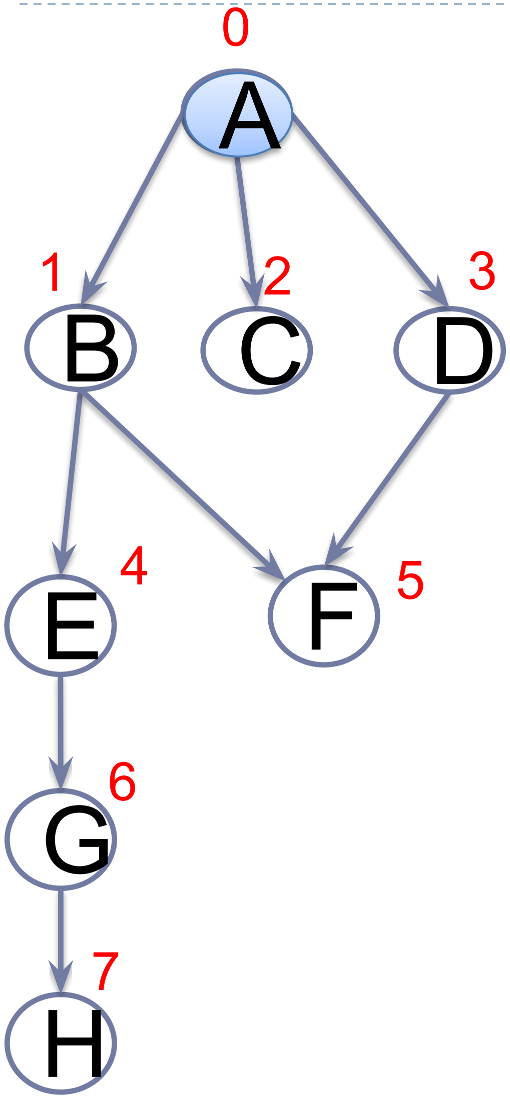
Graph traversal: Breadth-first traversal (BFS)



Start by a node (for example, vertex=0) , and put it into the queue

q | **0**

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

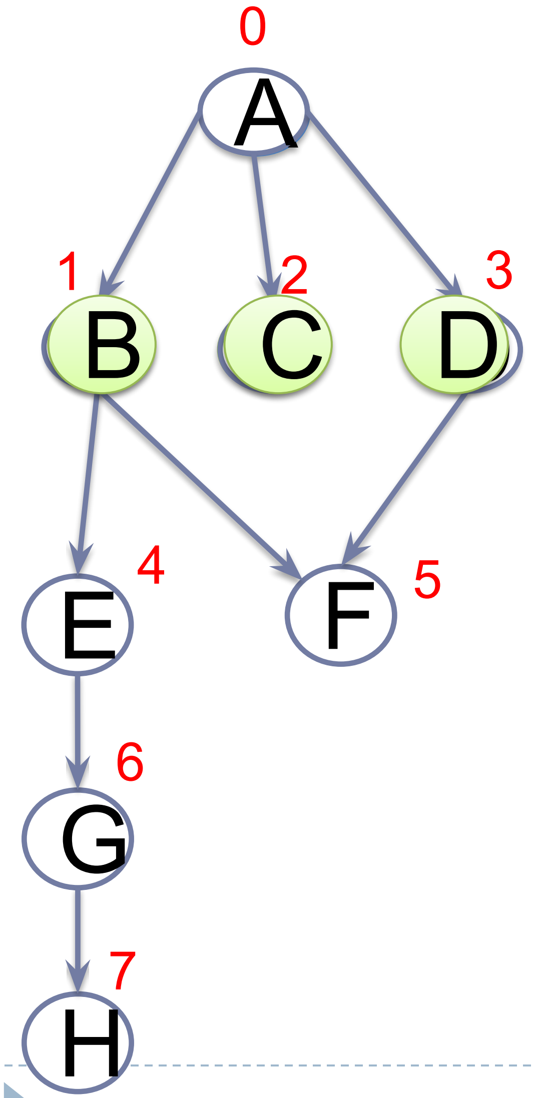
1. Remove the head from the queue
2. Print it and save it into the visited list

V=0 **q** 0

visited 0

output: 0

Graph traversal: Breadth-first traversal (BFS)

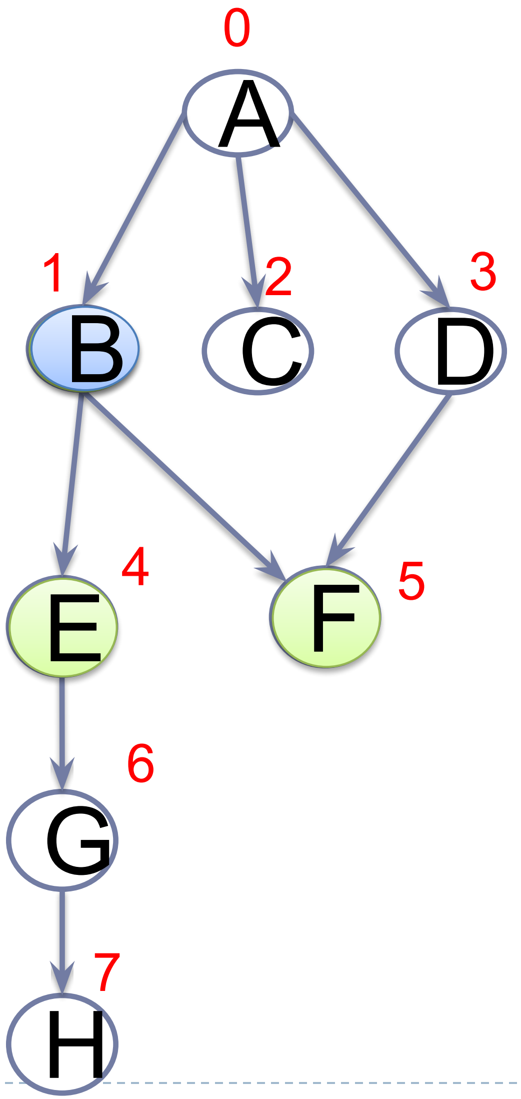


While the queue is not empty, repeat:

1. Remove the head from the queue
2. Print it and save it into the visited list
3. **Get its adjacent nodes and put them into the queue (only not visited)**

q | 1 2 3

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)

V=1 **q**

1	2	3	4	5

B (1) has as adjacent nodes: E(4), F (5)

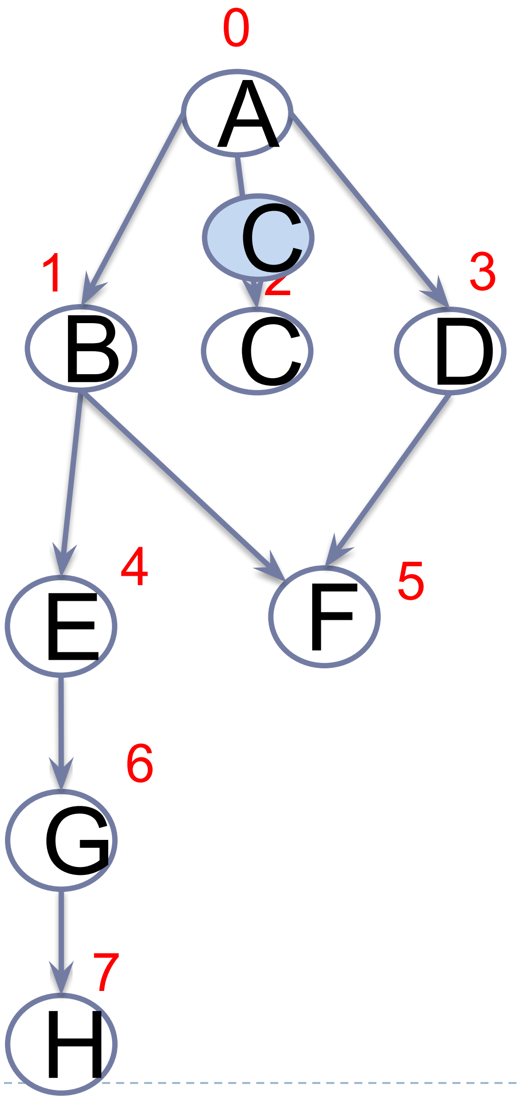
visited

0	1

output:

0	1

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)

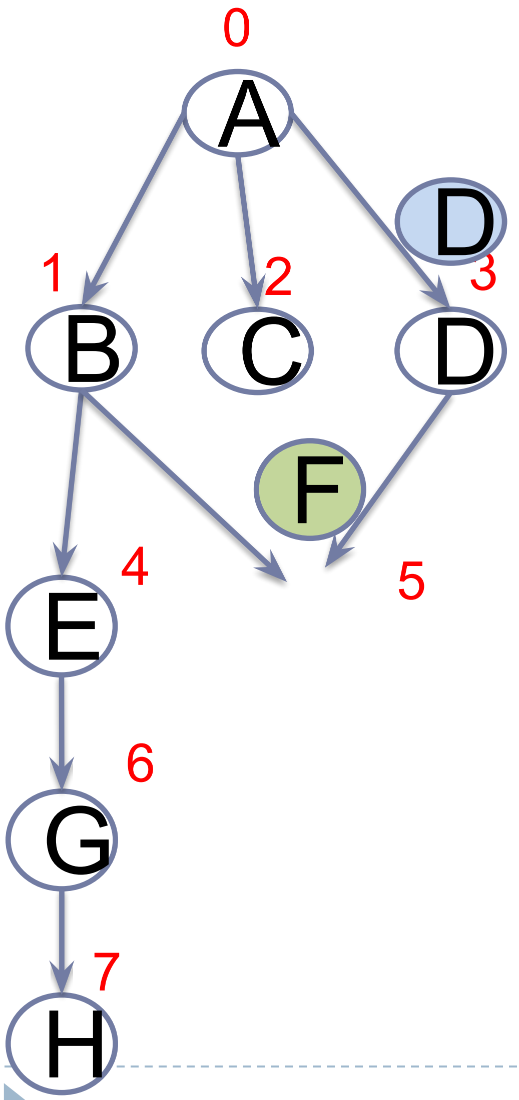
V=2 **q** 2 3 4 5

C(2) has no adjacent nodes

visited 0 1 2

output: 0 1 2

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)

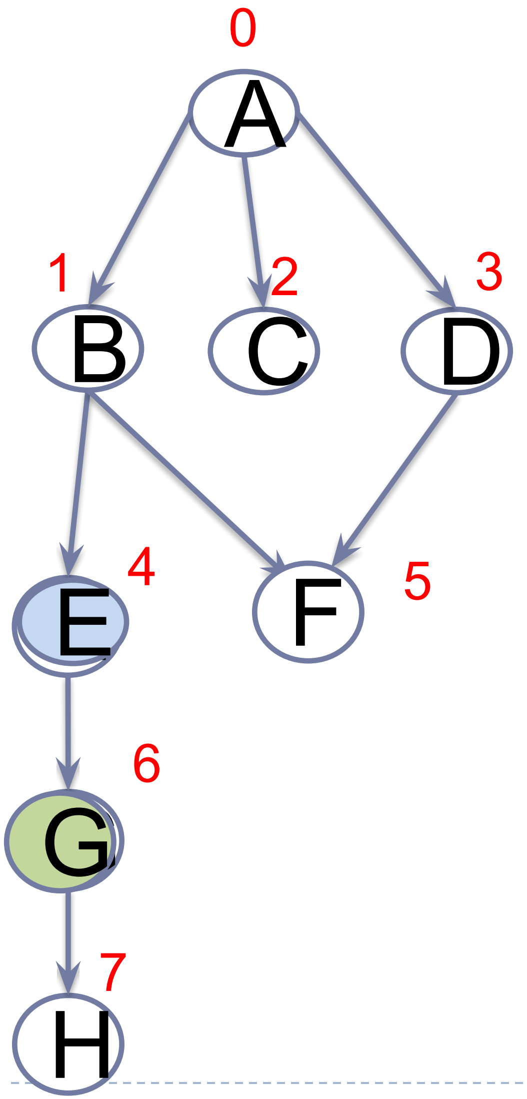
V=3 **q** 3 4 5

D(3) has one only adjacent node, F(5), which is already in the queue

visited 0 1 2 3

output: 0 1 2 3

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)

V=4

q

4 5 6

E(4) has one only adjacent node, G(6)

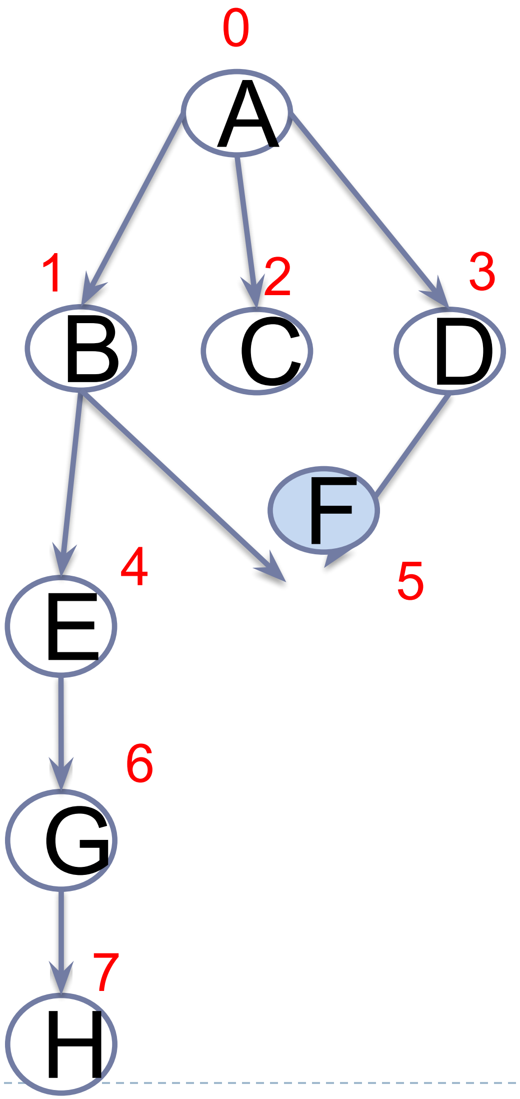
visited

0 1 2 3 4

output:

0 1 2 3 4

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)

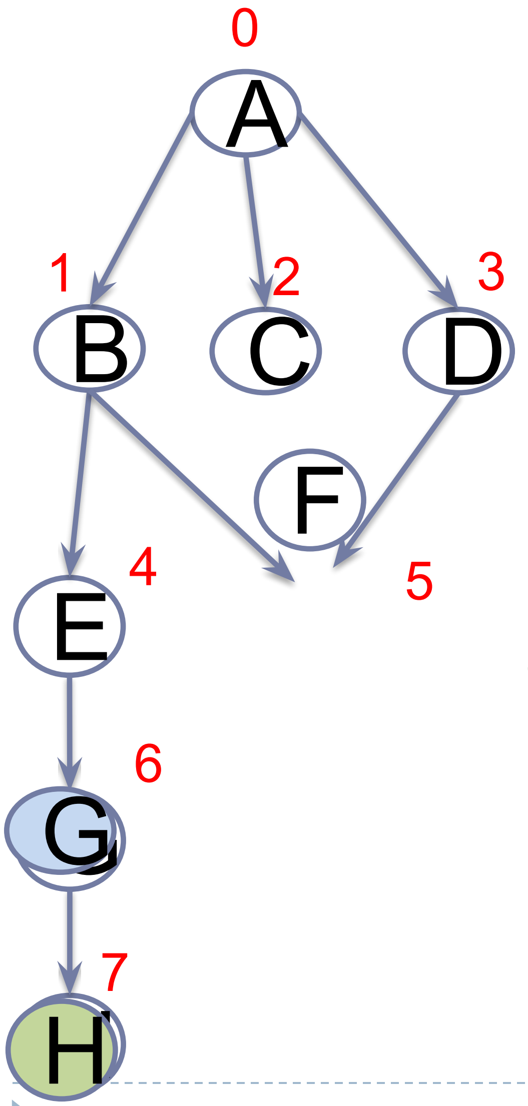
V=5 **q** 5 6

F(5) does not have any adjacent node

visited: 0 1 2 3 4 5

output: 0 1 2 3 4 5

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)

V=6 **q**

6	7

G(6) has one only adjacent node, H(7)

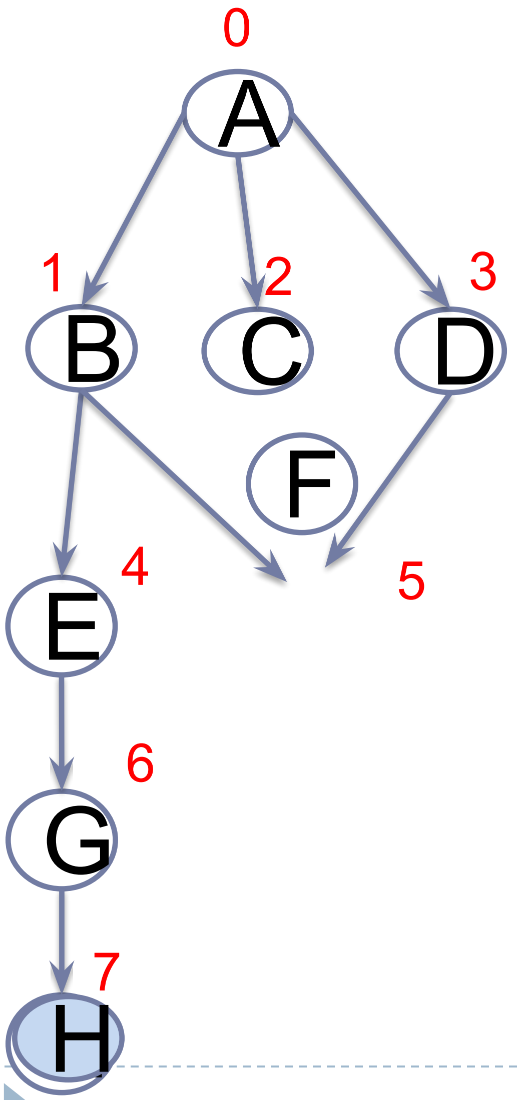
visited:

0	1	2	3	4	5	6

output:

0	1	2	3	4	5	6

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)∑

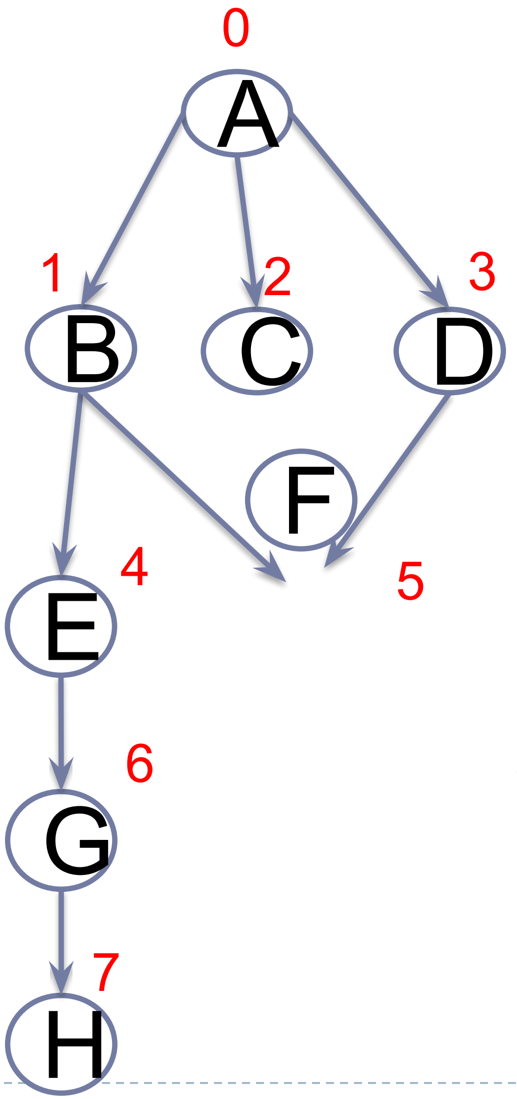
V=7 **q** 7

H(7) does not have any adjacent node

visited 0 1 2 3 4 5 6 7

output: 0 1 2 3 4 5 6 7

Graph traversal: Breadth-first traversal (BFS)



While the queue is not empty, repeat:

1. Remove the head from the queue.
2. Print it and save it into the visited list
3. Get its adjacent nodes and put them into the queue (only not visited)∑

q

The queue is empty and all the nodes have already visited!!!

output: 0 1 2 3 4 5 6 7

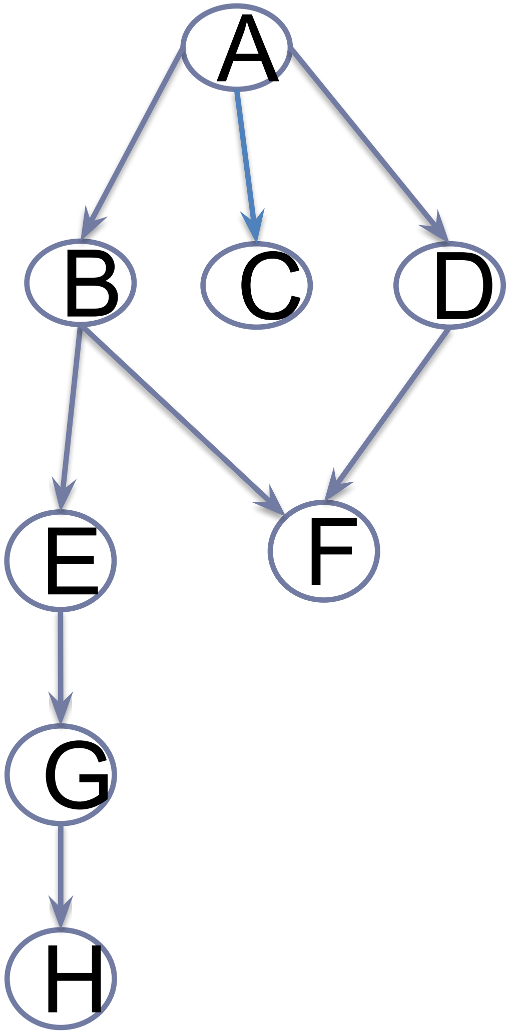
Graph traversal: Breadth-first traversal (BFS)

```
**Algorithm** bst(vertex):
   q=Queueu() #queue for adjacent vertices
   visited=[]
   q.enqueue(vertex)
   while q.isEmpty()==False:
      current=q.dequeue()
      print(current)
      visited.append(current)
      adjLst=getAdjacents(current)
      for v in adjLst:
         if v not in visited:
            q.enqueue(v)
```

Index

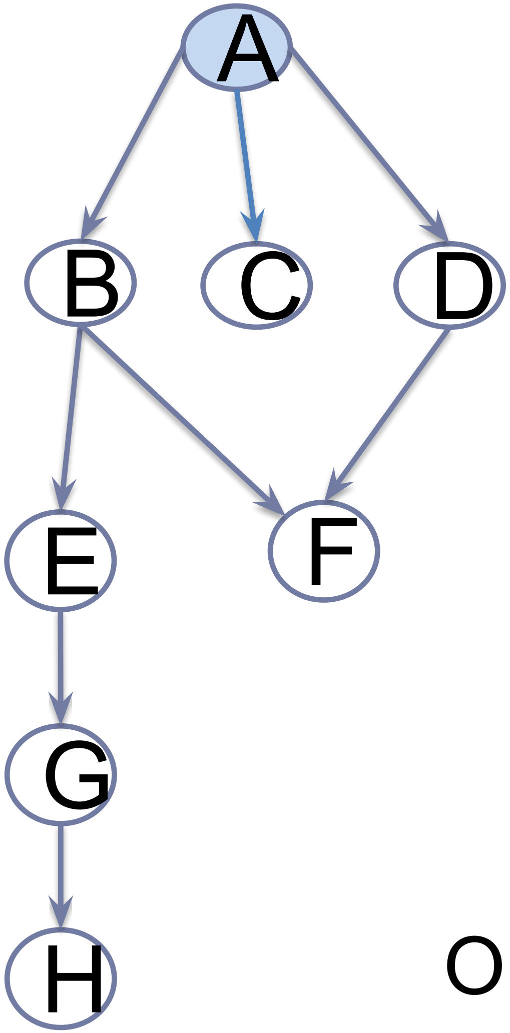
- Introduction to Graphs
- Graph properties
- Graph representation:
  - Adjacency Matrix.
  - Adjacency List.
- **Graph Traversal**
  - Breadth-first Traversal
  - **Depth-first Traversal**

Graph traversal: Depth-first traversal (BFS)



Select a node and go forward as far as possible along a branch, if not then, backtrack

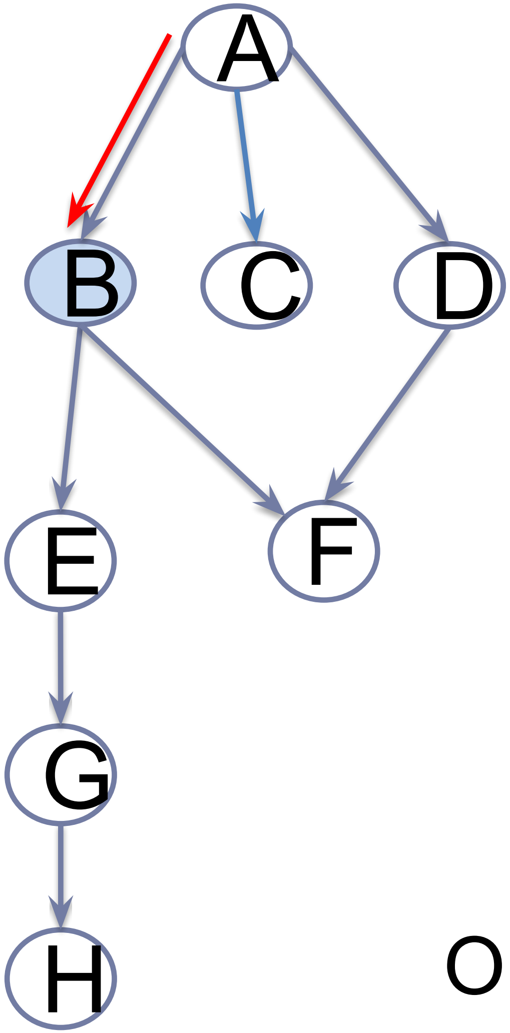
Graph traversal: Depth-first traversal (BFS)



Select a node and go forward as far as possible along a branch, if not then, backtrack

Output: A,

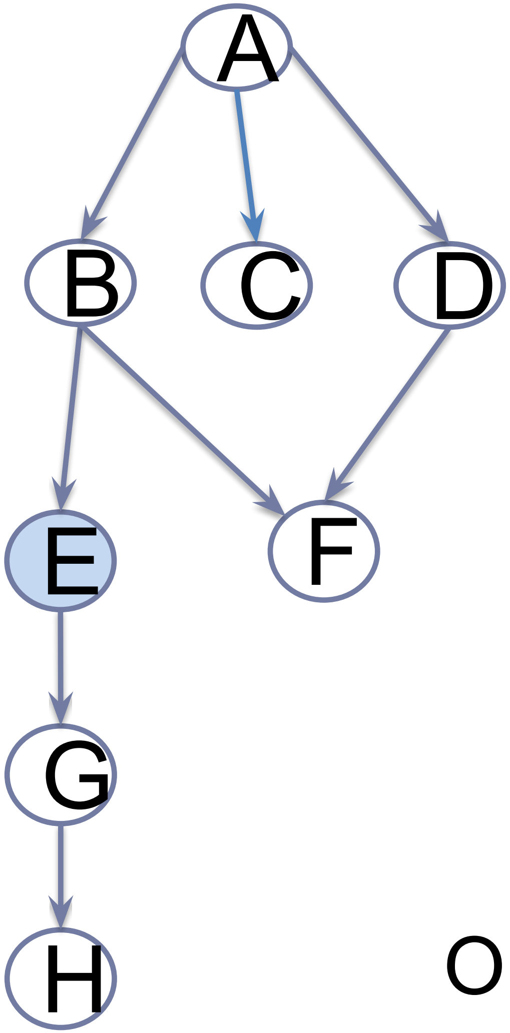
Graph traversal: Depth-first traversal (BFS)



Select a node and go forward as far as possible along a branch, if not then, backtrack

Output: A,B

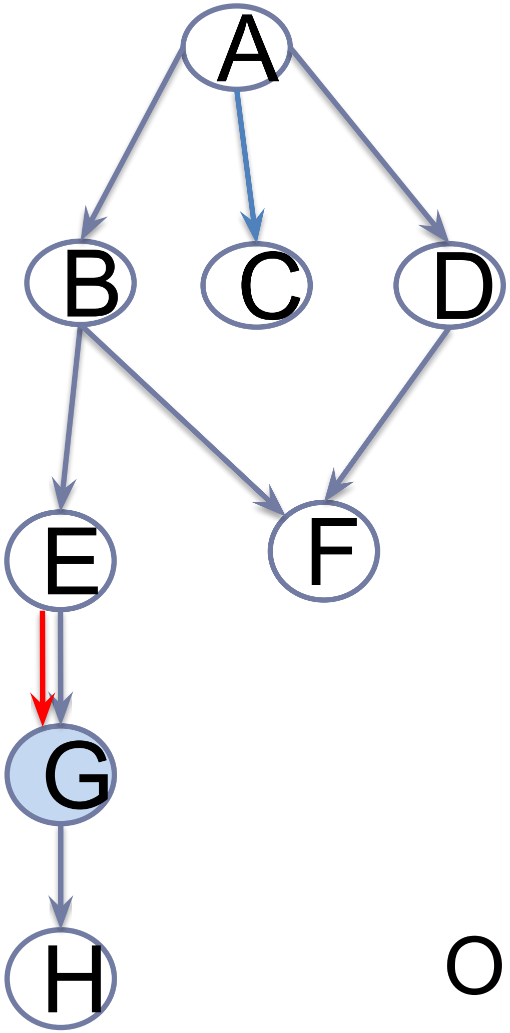
Graph traversal: Depth-first traversal (BFS)



Select a node and go forward as far as possible along a branch, if not then, backtrack

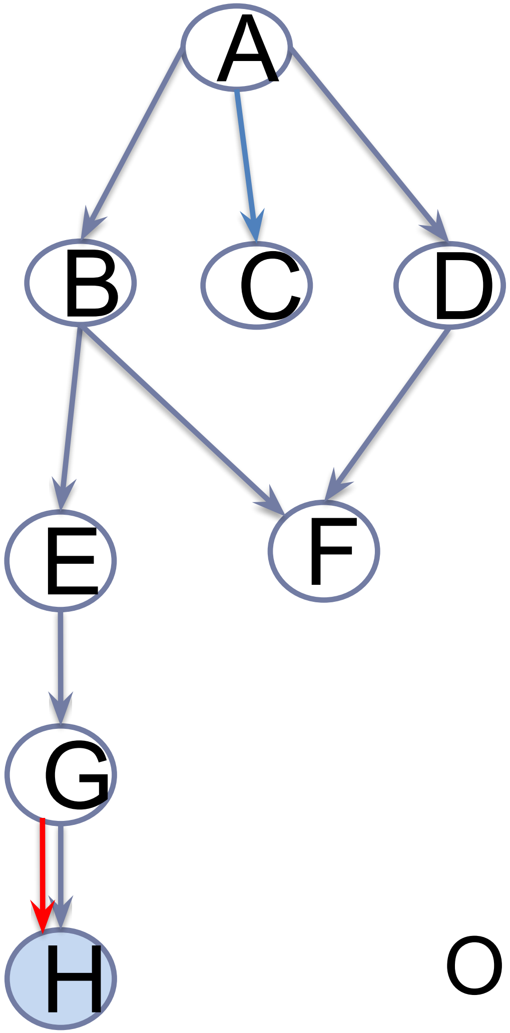
Output: A, B, E

Graph traversal: Depth-first traversal (BFS)



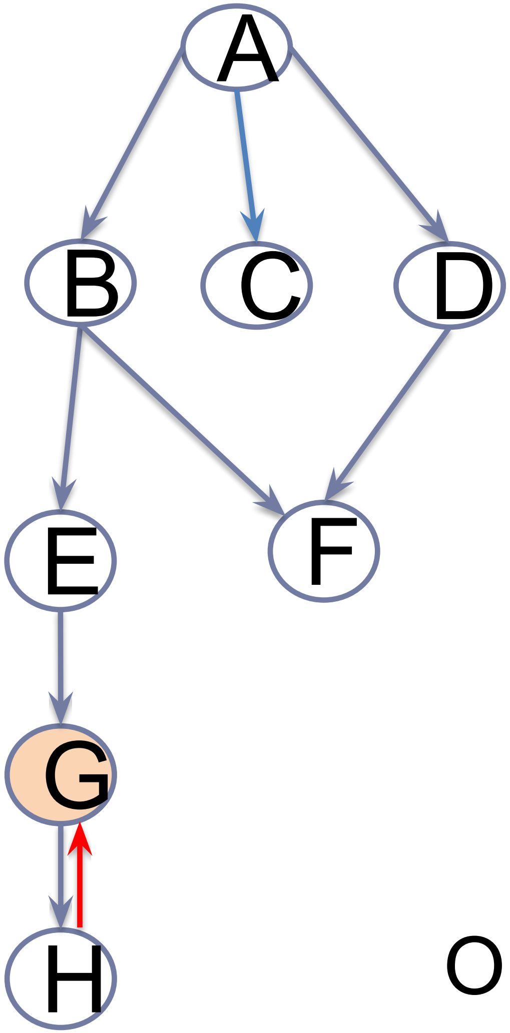
Output: A, B, E, G

Graph traversal: Depth-first traversal (BFS)



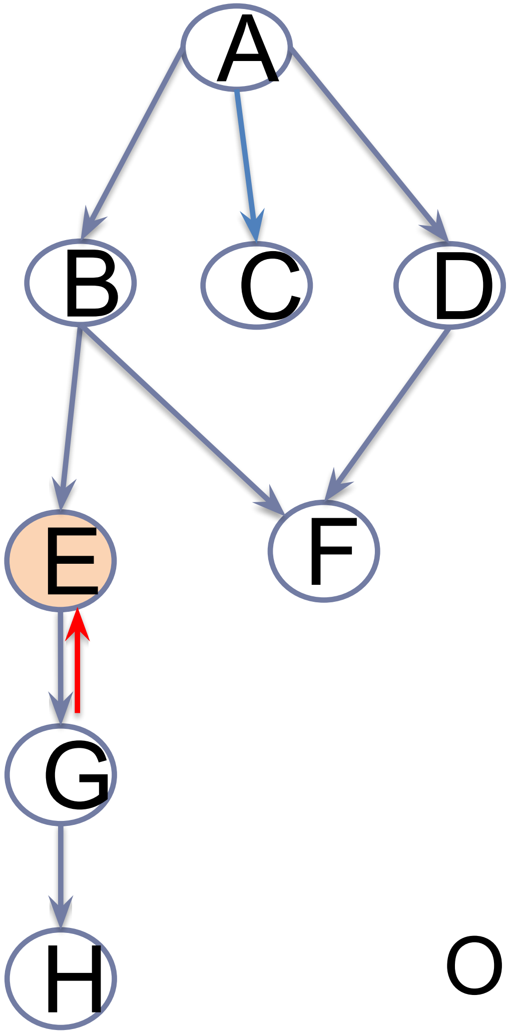
Output: A, B, E, G, H

Graph traversal: Depth-first traversal (BFS)



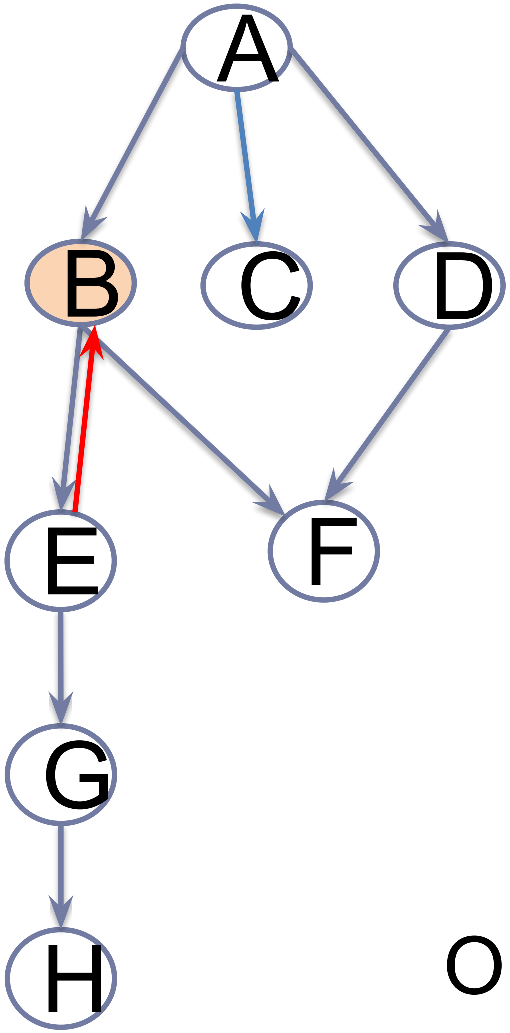
Output: A, B, E, G, H

Graph traversal: Depth-first traversal (BFS)



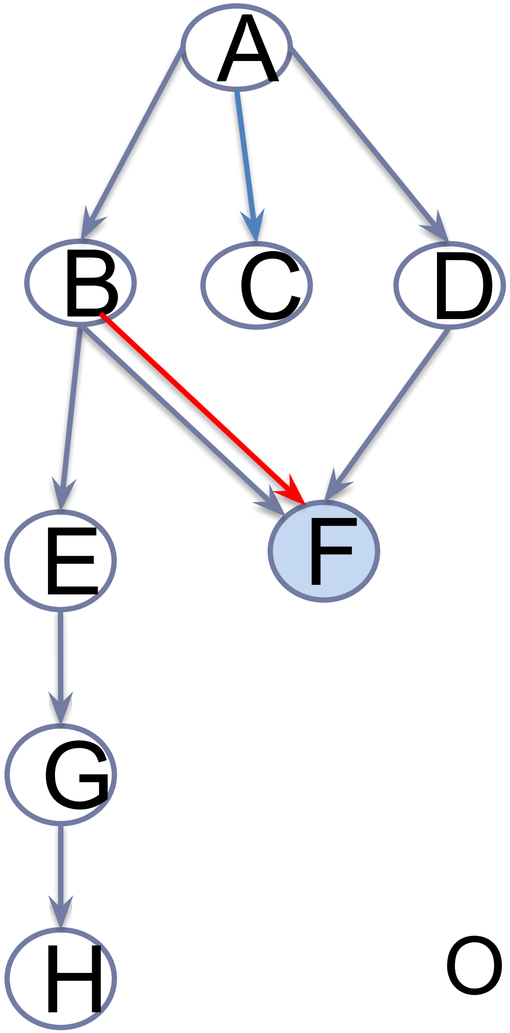
Output: A, B, E, G, H

Graph traversal: Depth-first traversal (BFS)



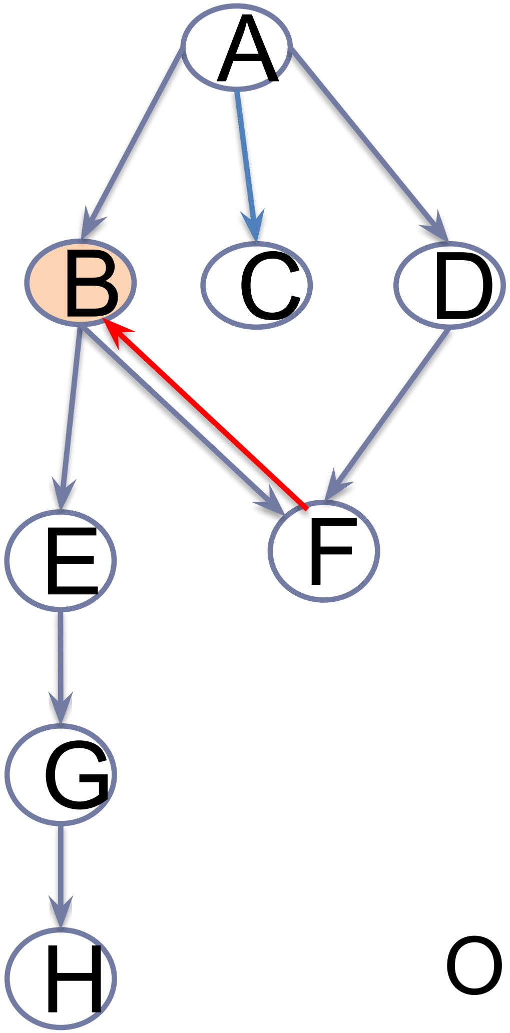
Output: A, B, E, G, H

Graph traversal: Depth-first traversal (BFS)



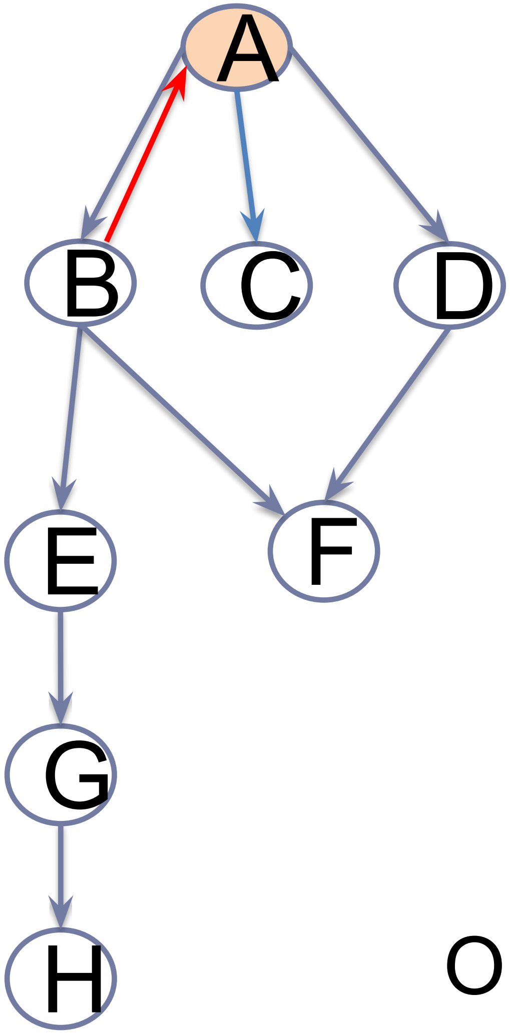
Output: A, B, E, G, H, F

Graph traversal: Depth-first traversal (BFS)



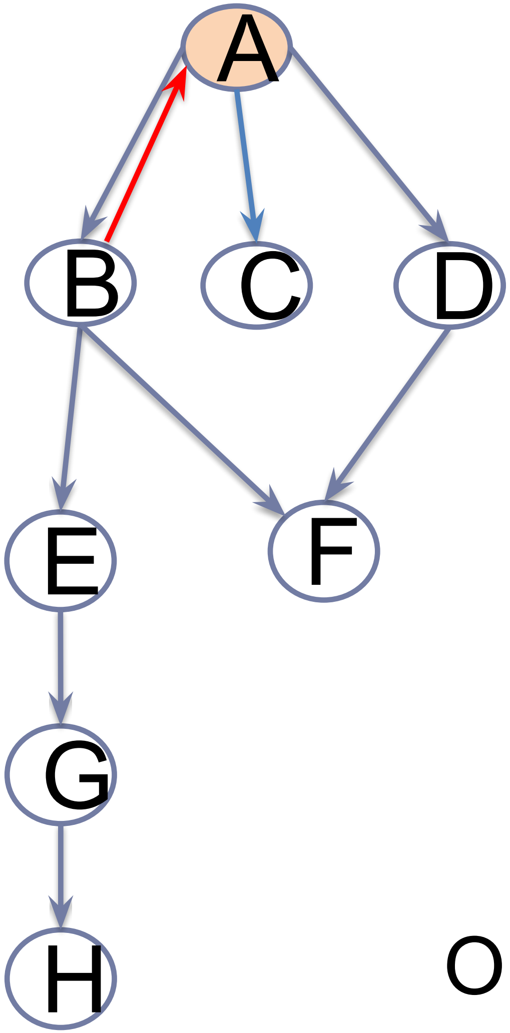
Output: A, B, E, G, H, F

Graph traversal: Depth-first traversal (BFS)



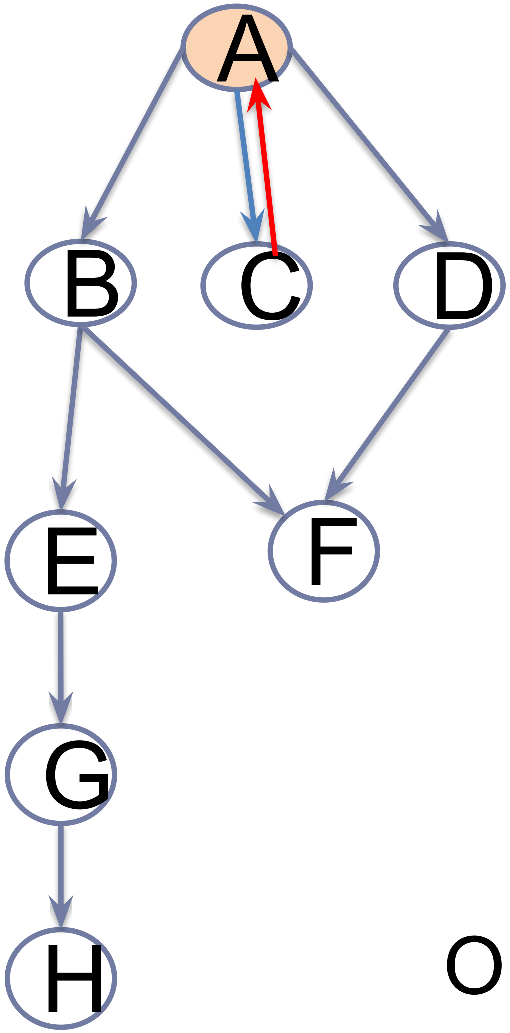
Output: A, B, E, G, H, F

Graph traversal: Depth-first traversal (BFS)



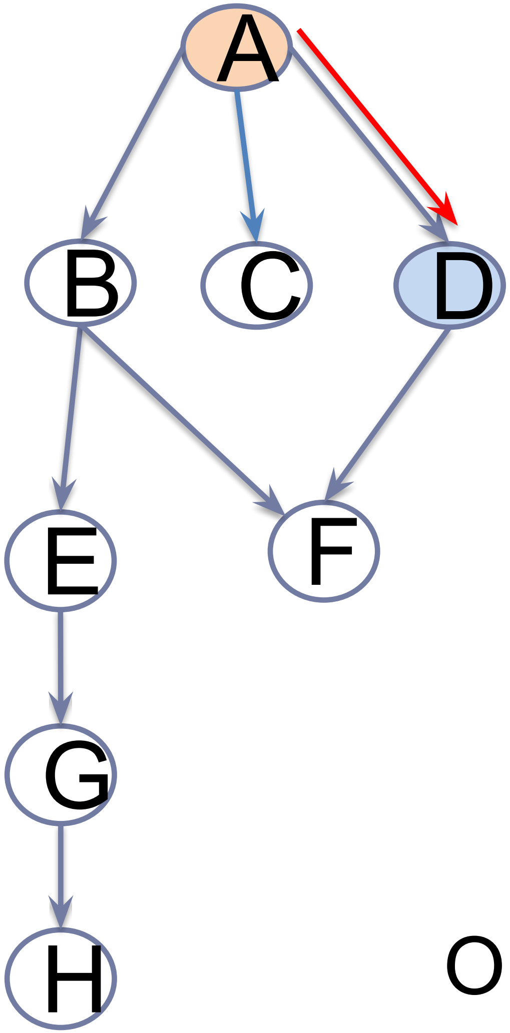
Output: A, B, E, G, H, F

Graph traversal: Depth-first traversal (BFS)



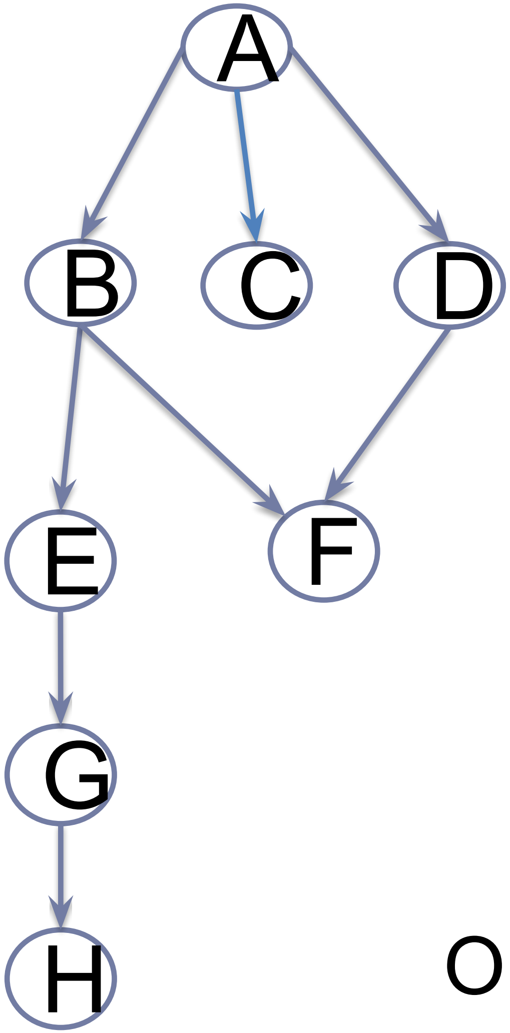
Output: A, B, E, G, H, F, C

Graph traversal: Depth-first traversal (BFS)



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

Graph traversal: Depth-first traversal (BFS)



D has an adjacent node F, which is visited.

We have finished because all nodes are already visited!!!

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

Graph traversal: Depth-first traversal (BFS)

```
Algorithm depth(vertex, visited):
   print(vertex)
   visited.append(vertex)
   for v in getAdjacents(vertex):
      if v not in visited[v]:
         depth(v,visited)
```

Note: visited is a list to store the nodes that we visit.