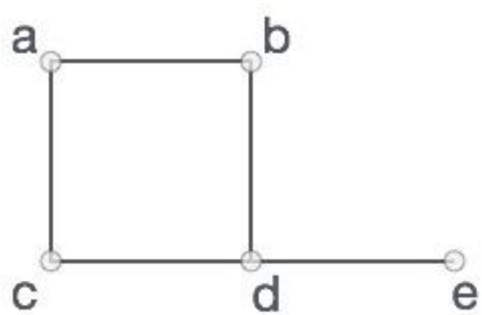
Graphs

Graphs

- Non-linear data structures
  - Trees
  - Graphs
- Tree
  - There is a hierarchical relationship between parent and children.
  - Tree is a special case of graph.
- Graphs
  - No hierarchical relationship.

What is a graph?

- **Definition:**
  - A data structure that consists of a set of **nodes (*vertices*)** and a set of **edges** that relate the nodes to each other.
  - The set of edges describes relationships among the vertices .
  - 

In the graph,

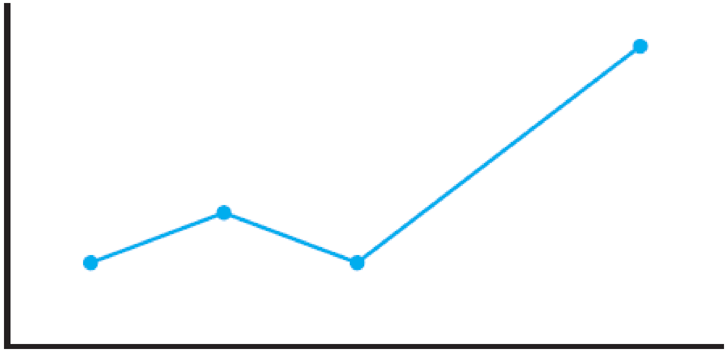
V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

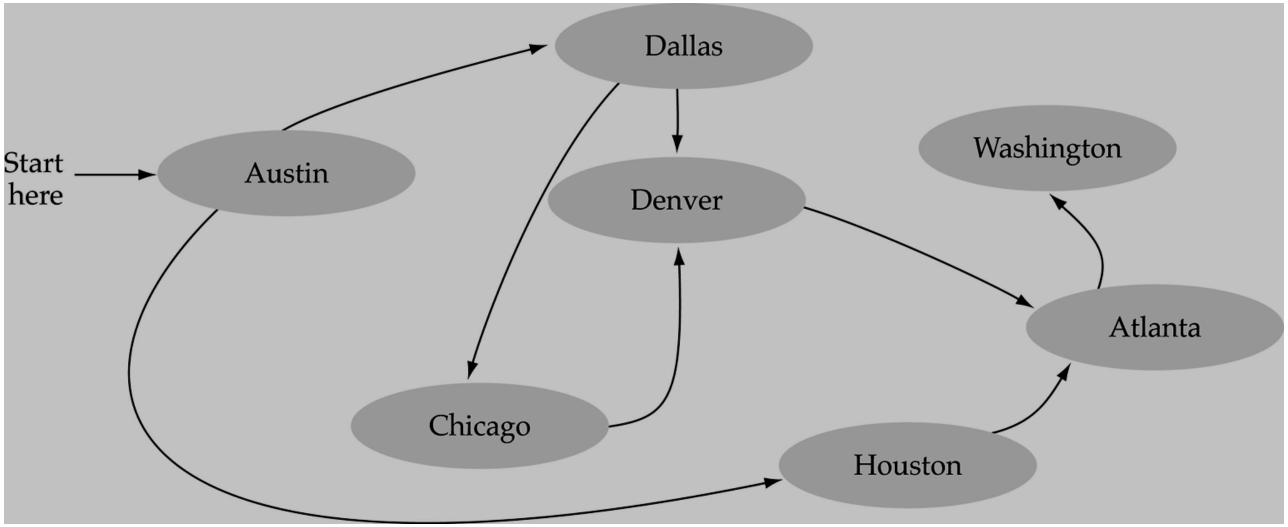
What is a graph?

Graphs consist of

- points called *vertices*
- lines called *edges*



- Edges connect *two* vertices.
- Edges only intersect at vertices.
- Edges joining a vertex to itself are called *loops*



Formal definition of graph

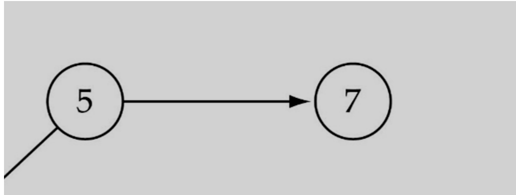
- A graph G consists of two things:

1. A set *V*, called set of all **vertices**(or nodes or elements)
2. A set *E*, called set of all **edges** such that each edge e in E is identified with a unique pair (u,v) of nodes in V, denoted by **e=(u,v)**

- A graph can be represented as **G=(V,E)**

Graph terminology

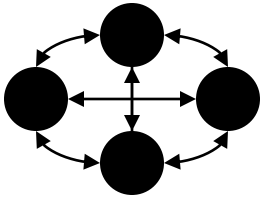
- **Adjacent nodes**: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7

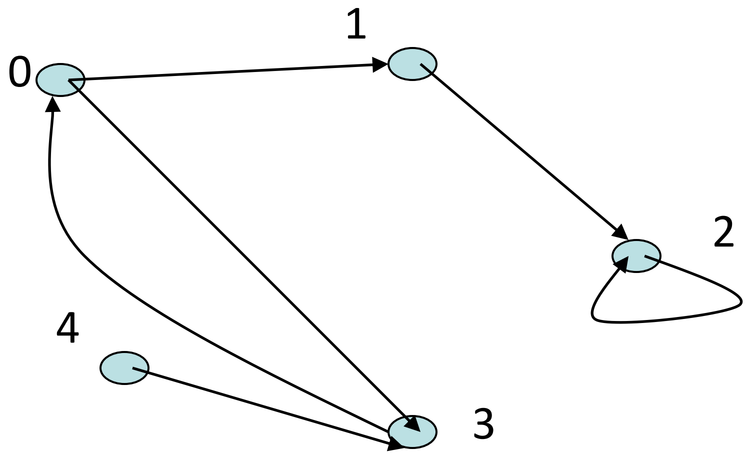
- **Path**: a sequence of vertices that connect two nodes in a graph.

- **Degree** of a node x, deg(x) is the no. of edges containing x.
- **Complete graph**: a graph in which every vertex is directly connected to every other vertex



Examples of Graphs

- V={0,1,2,3,4}
- E={(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)}



When (x,y) is an edge,
we say that x is *adjacent to* y, and y is *adjacent from* x.

0 is adjacent to 1.
1 is not adjacent to 0.
2 is adjacent from 1.

Graph terminology

- Connected graph: a graph is said to be connected, if there is a path from every node to every other node
- The size of a graph is the number of *nodes* in it
- The empty graph has size zero (no nodes)
- Cycle: a path that begins and ends at same vertex
- A directed graph is one in which the edges have a direction
- If a graph does not have any cycle, then it is acyclic graph
- An undirected graph is one in which the edges do not have a direction

•An *undirected graph* is connected if there is a path from every node to every other node

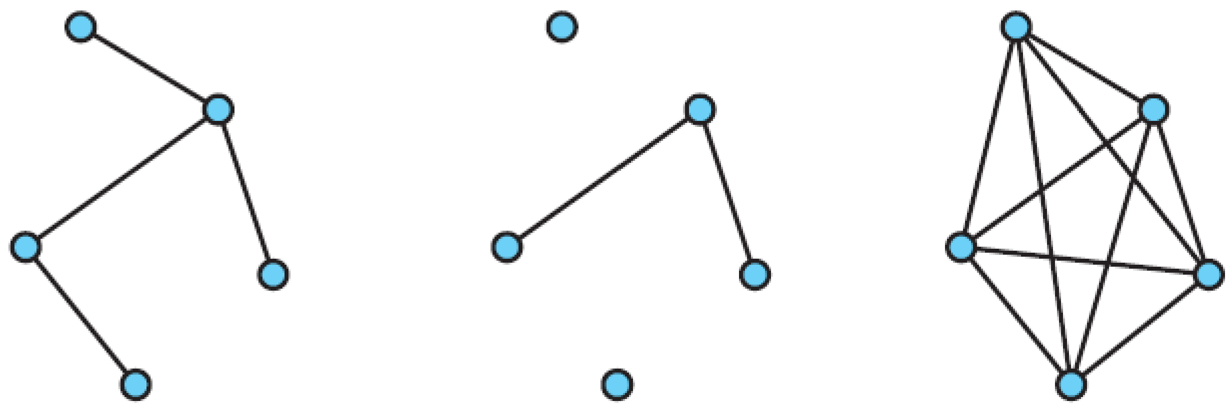
•A *directed graph* is strongly connected if there is a path from every node to every other node

•A directed graph is weakly connected if the underlying undirected graph is connected

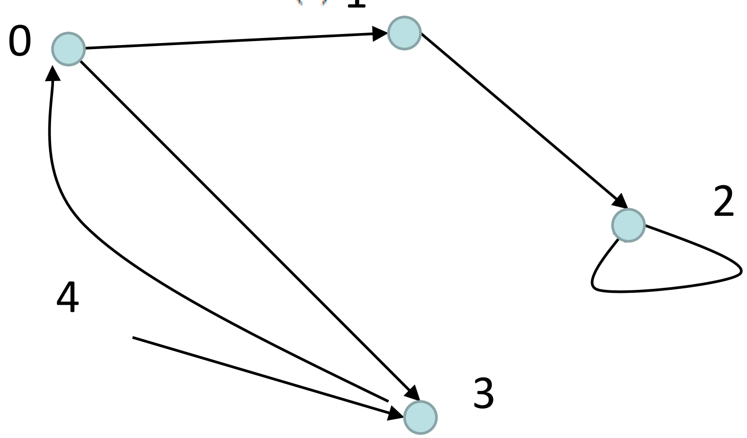
•Node X is reachable from node Y if there is a path from Y to X

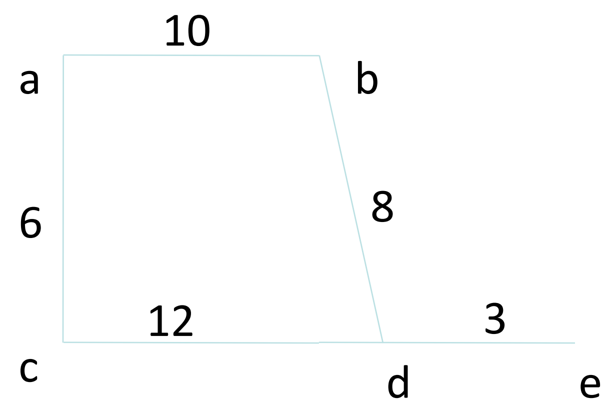
•A weighted graph is a graph in which each edge is assigned a weight.

Terminology



(a) (b) (c)





Graphs that are (a) connected (b) disconnected (c) complete (d)directed (e) weighted graph

Graph representations

- **Sequential representation**
  – Using adjacency matrix

- **Linked list representation**
  - Using adjacency list

- **Set representation**
  - Using edge list

Sequential representation

<u>Adjacency matrix:</u>

- Suppose G is a directed graph with n nodes
- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A

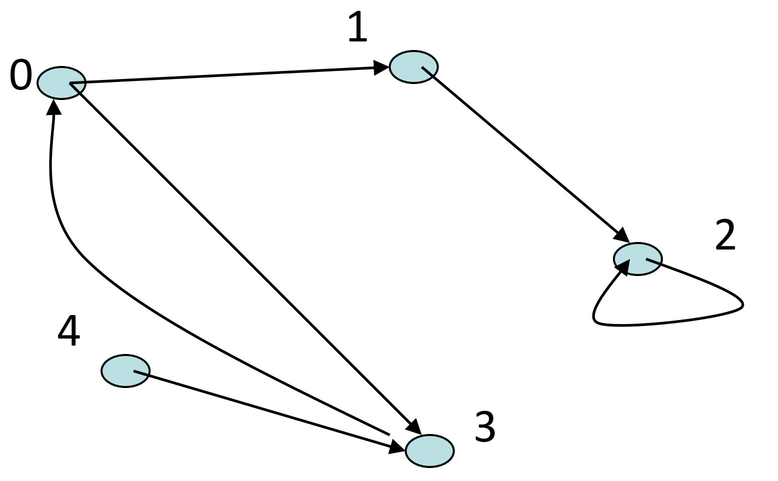
- A[i][j] = 1 if (i,j) is an edge
- A[i][j] = 0 if (i,j) is not an edge

- i.e A_{ij} = 1, if there is an edge from v_i to v_j
  = 0, otherwise

Example of Adjacency Matrix

A =

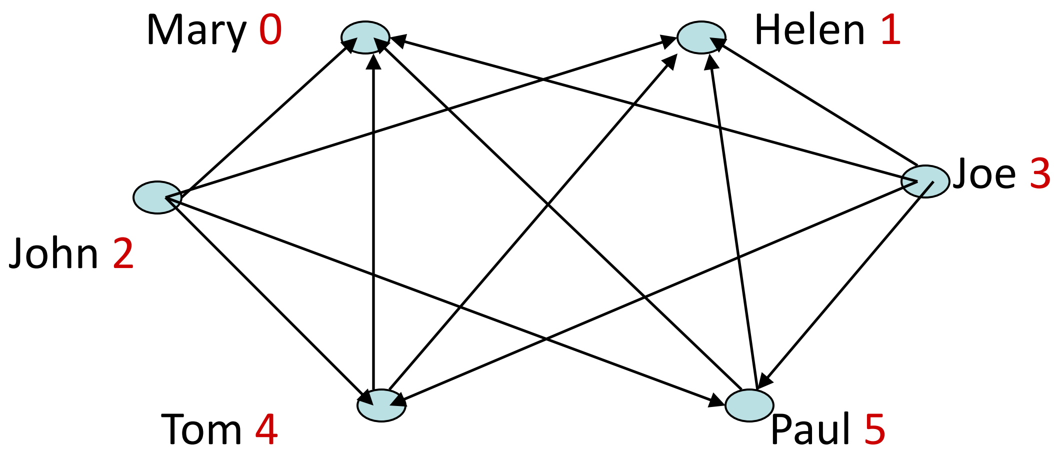
	0	1	2	3	4
0	0	1	0	1	0
1	0	0	1	0	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	0	0	1	0



Another Example of Adj. Matrix

0 1 2 3 4 5

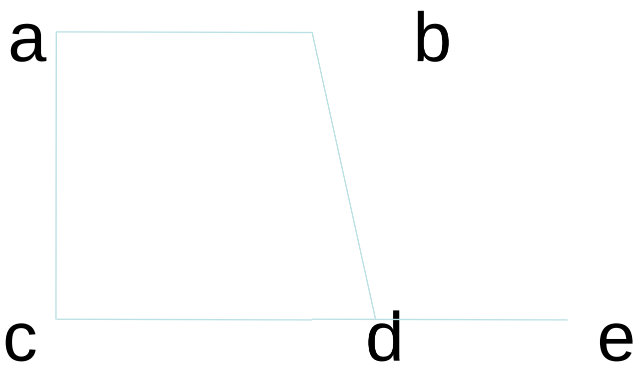
$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency matrix

- Suppose G is an undirected graph.
- Then the adjacency matrix A of G will be a symmetric matrix.
- i.e $a_{ij}=a_{ji}$ for every i and j

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



Exercise

- Consider a directed graph with nodes a, b, c & d. The adjacency matrix of A of G is as follows. Draw G.

$$A = \begin{matrix} a \\ b \\ c \\ d \end{matrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Pros and Cons of Adjacency Matrices

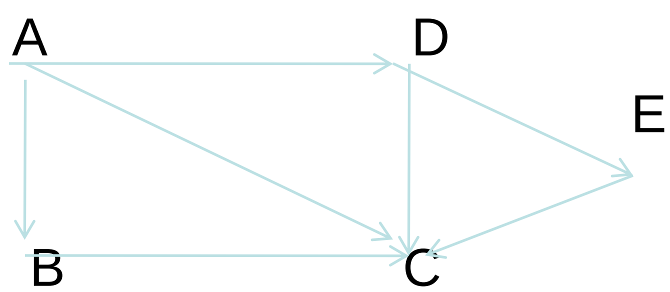
- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i,j) is an edge: simply check if A[i][j] is 1 or 0
  - Easy to **implement dense matrix.**
- Cons:
  - No matter how few edges the graph has, the matrix takes **$O(V^2)$** in memory
  - Memory wastage in case of sparse matrix.
  - Difficult to insert and delete nodes in G

Linked list representation

- **Using adjacency list.**
  - List of adjacent nodes.
  - Adjacent nodes are also called **successor or neighbors**
  - It is **the space saving way** of graph representation.

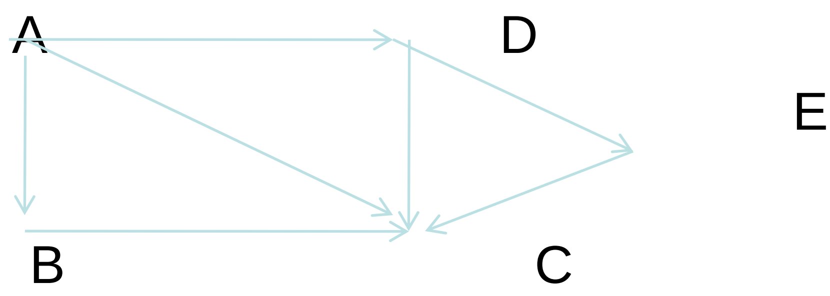
Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

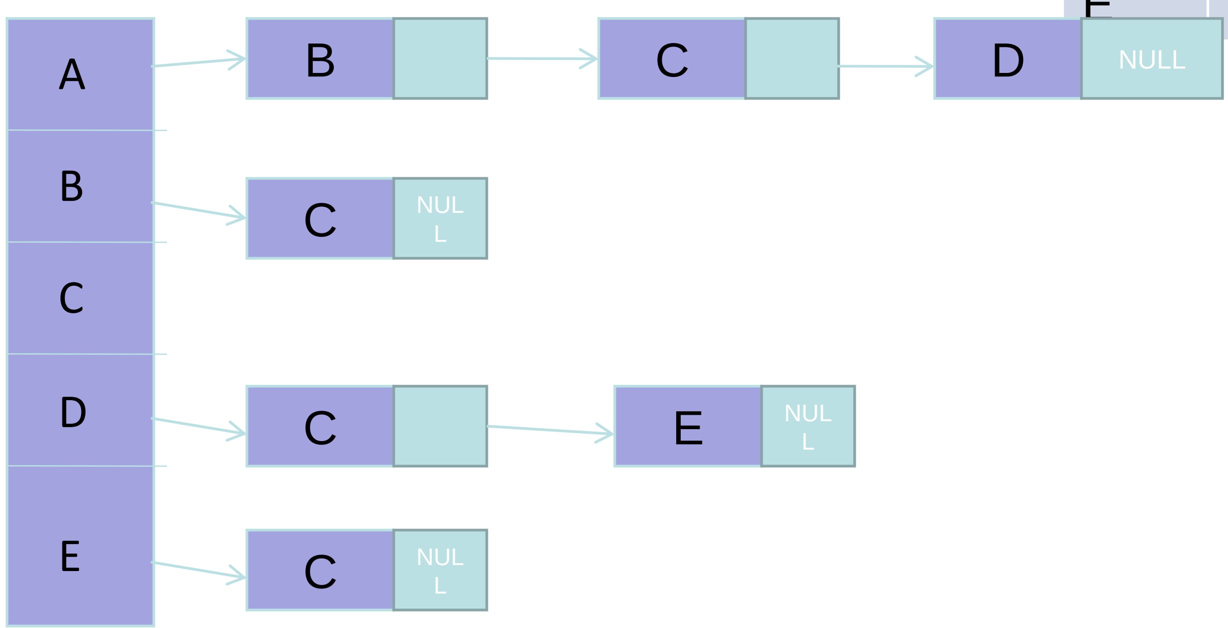


Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

Adjacency Lists Representation



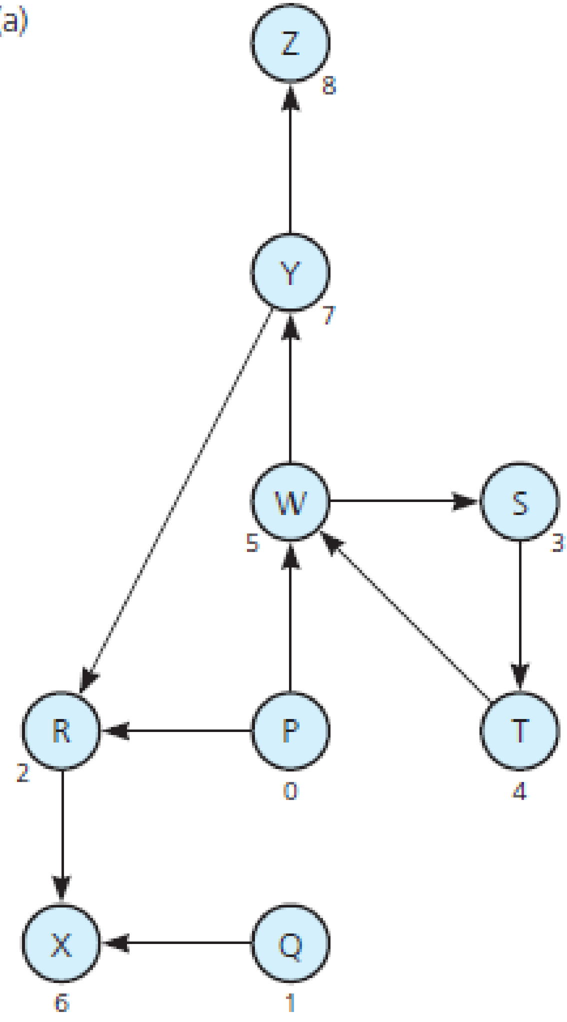
Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C



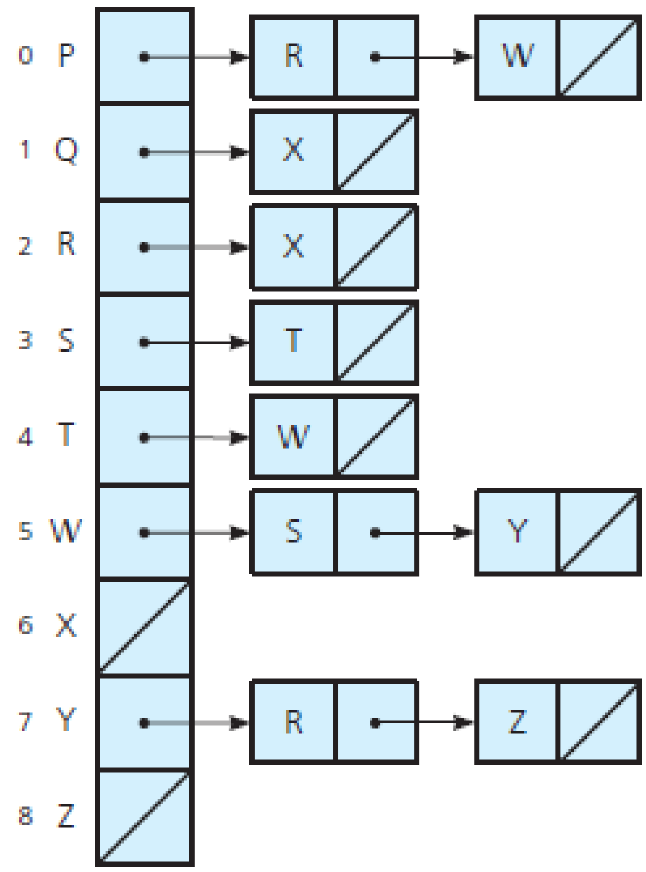
Adjacency Lists example

Eg2

(a)

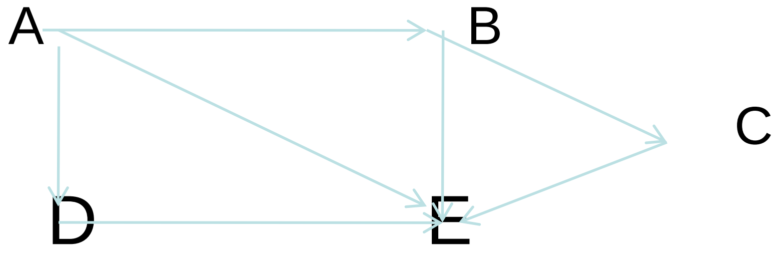


(b)



Set Representation

- Using edge list.
- Straight forward method of representing graph.
- Two sets are maintained
  1. V, the set of vertices
  2. E, the set of edges- which is the subset of V x V in sorted form.

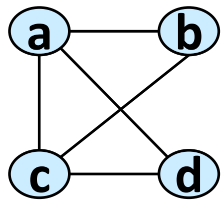


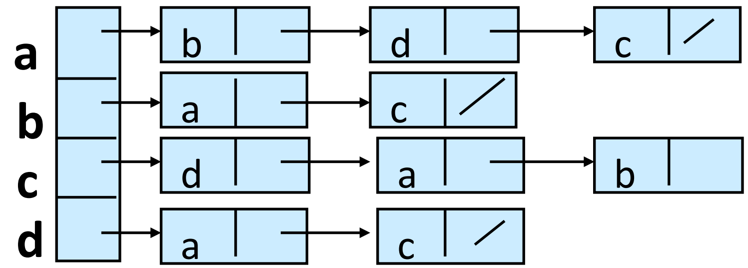
V={ A, B, C, D, E}

E={(A,B),(A,D),(A,E),(B,C),(B,E), (D,E), (C,E)}

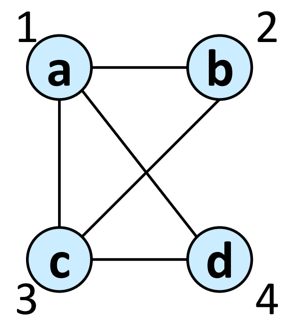
So, Representation of Graphs..

- Three standard ways.
  - Adjacency Lists.





  - Adjacency Matrix.



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

  - Edge list.

V={ a,b,c,d}

E={ (a,b),(a,c),(a,d),(b,c),(c,d)}

Graph traversals

- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)

- *Methods*:

  1. **Depth-First-Search** (DFS) – use Stack for implementation

  2. **Breadth-First-Search** (BFS) – use Queue for implementation

Breadth First Search

Using QUEUE

Graph traversals

- During the execution of DFS or BFS, each node N of G will be in one of three states, called status of N:
  - STATUS =1 (Ready state) – The initial state of the node N.
  - STATUS =2 (Waiting state) – The node is on stack/queue. Waiting to be processed.
  - STATUS =3 (Processed state) – The node N has been processed.

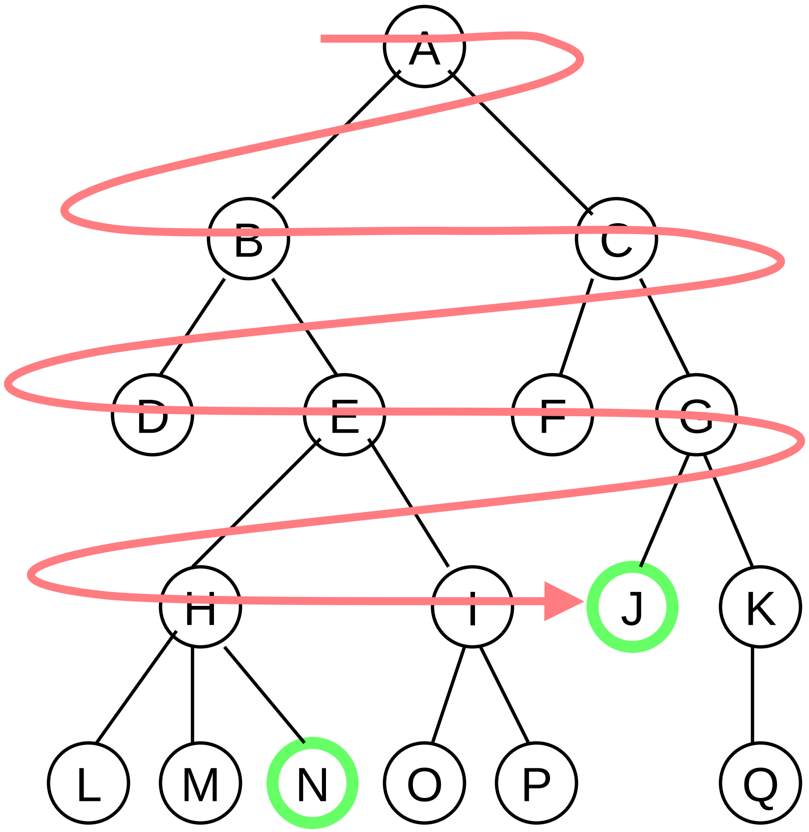
Breadth-First-Search (BFS)

- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
  - Its like ripples in the pond.
- BFS can be implemented efficiently using a *queue*

BFS- rules

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

Breadth-first searching

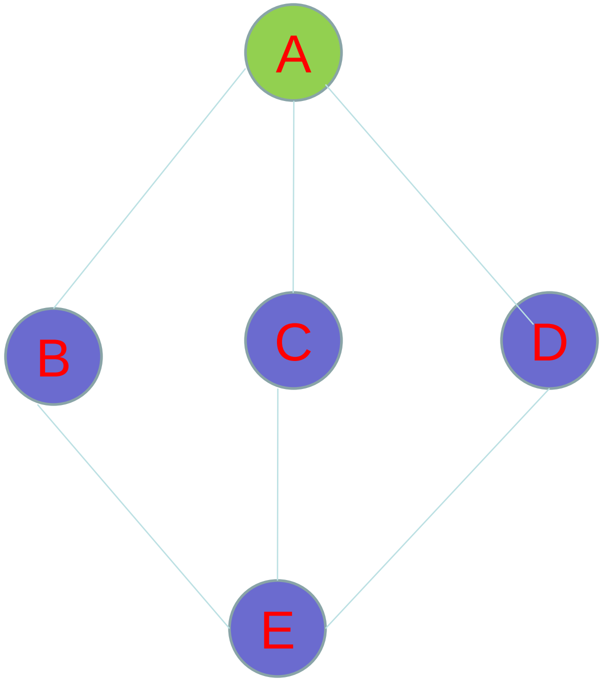


- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

BFS algorithm

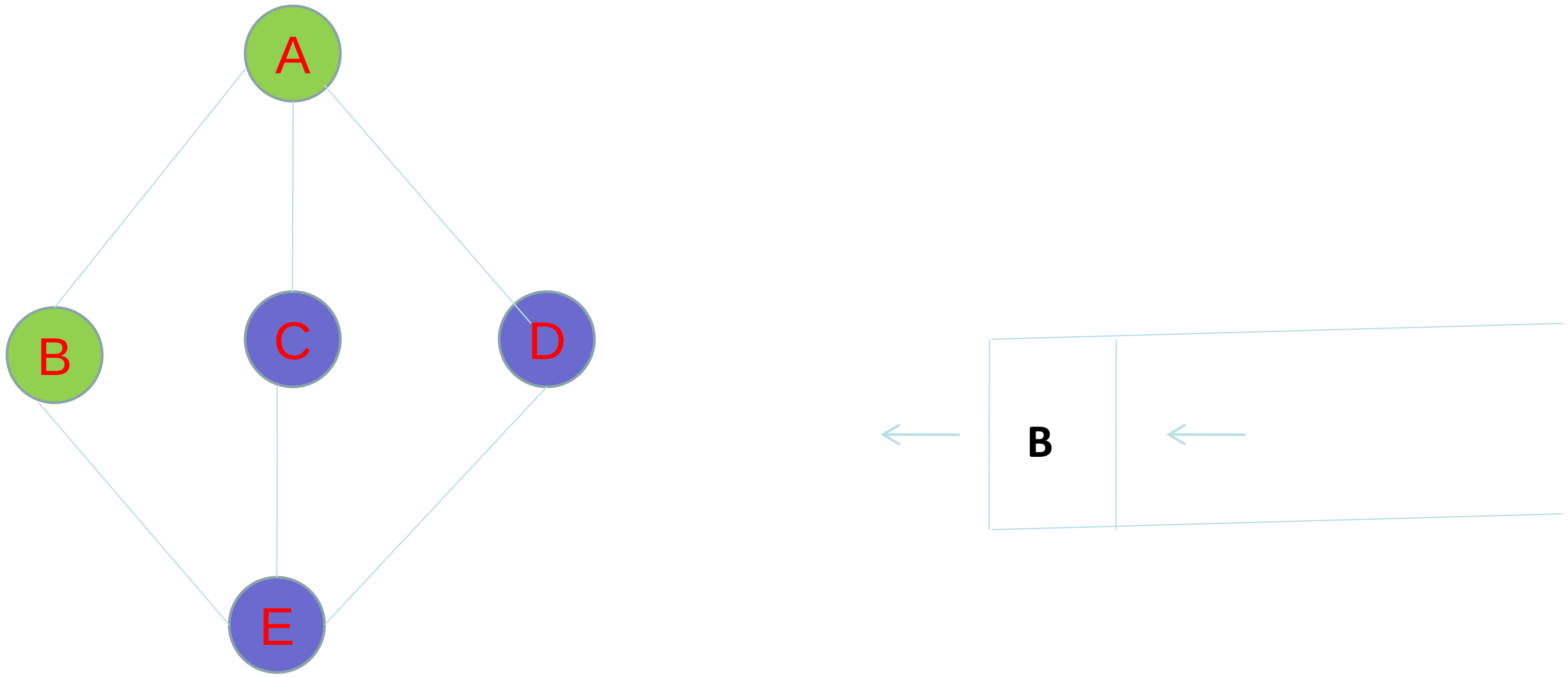
1. Put the starting node in a Queue named OPENQ
2. Repeat until Queue is empty:
3. Dequeue a node
4. Process it
5. Add it’s children to queue

BFS example



- Initially all nodes are in ready state
- Let the starting node be A. Insert in Q
- **Node visited: A**

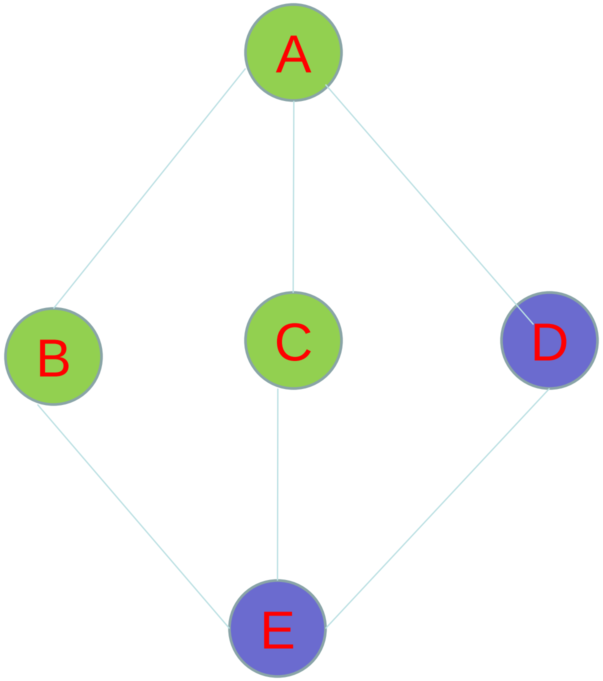
BFS example

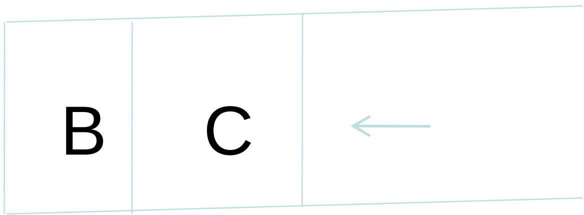


1. Dequeue A
2. Insert the adjacent unvisited vertex of A in queue

Node visited: A B

BFS example

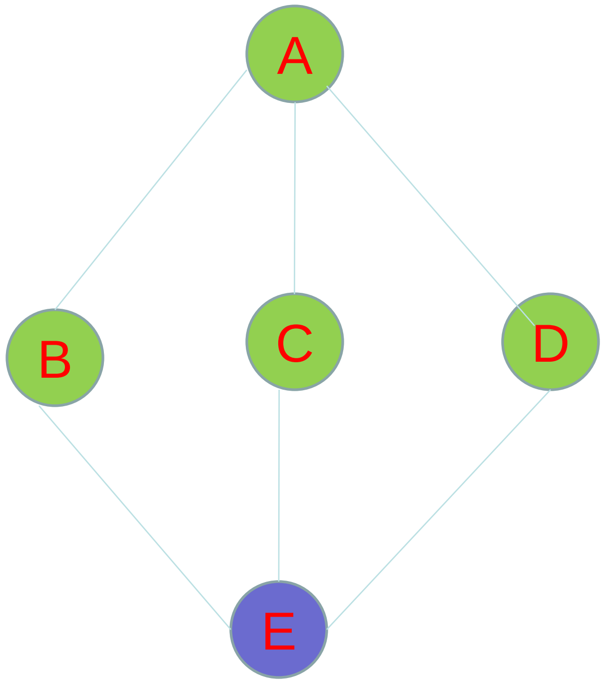


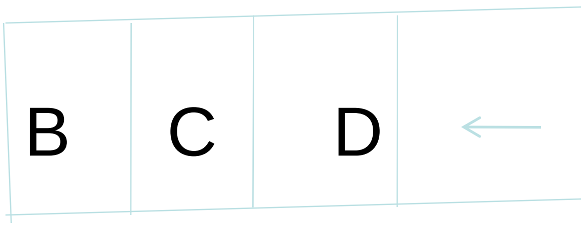


1. Insert the adjacent unvisited vertex of A in queue: C

Node visited: A B C

BFS example

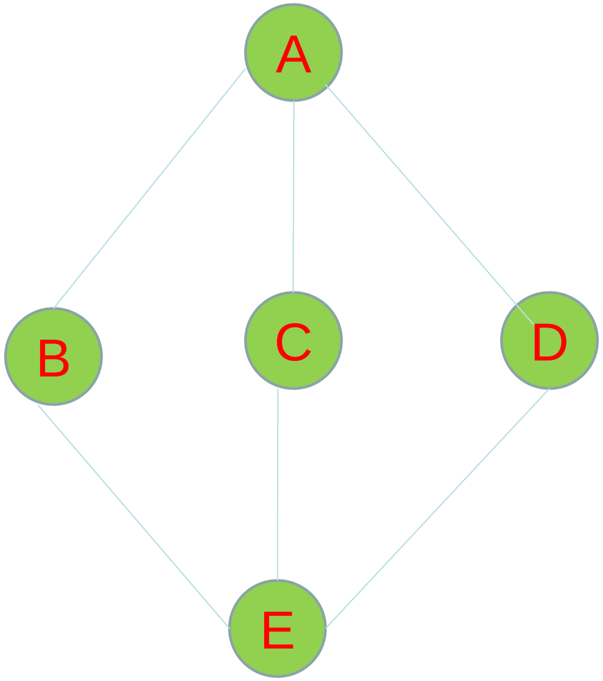


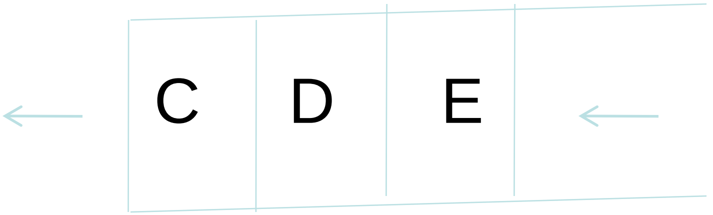


1. Insert the next adjacent unvisited vertex of A in queue

Node visited: A B C D

BFS example





1. Now no unvisited adjacent vertex for A.
2. So dequeue: Remove B and insert its adjacent vertex in queue : E
3. Display E

Node visited: A B C D E

QUEUE: Example 2

a

b c d e

c d e f g

d e f g

e f g h i j

f g h i j

g h i j

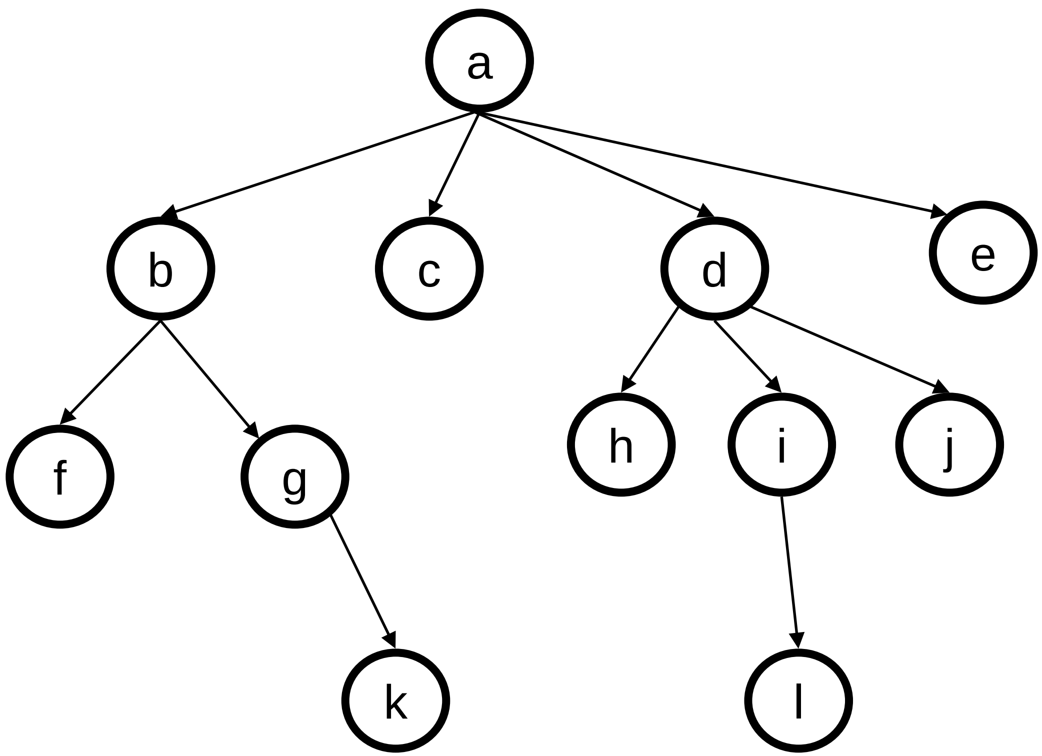
h i j k

i j k

j k l

k l

l



Result: a b c d e f g h I j k l

Depth First Search

Using STACK

- **Depth First Search algorithm(DFS)** traverses a graph in a **depthward motion** and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Travel as far as you can down a path.
  - Nodes are visited deeply on the left-most branches before any nodes are visited on the right-most branches
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
  - It is similar to the inorder traversal of a tree.

- DFS can be implemented efficiently using a *stack*

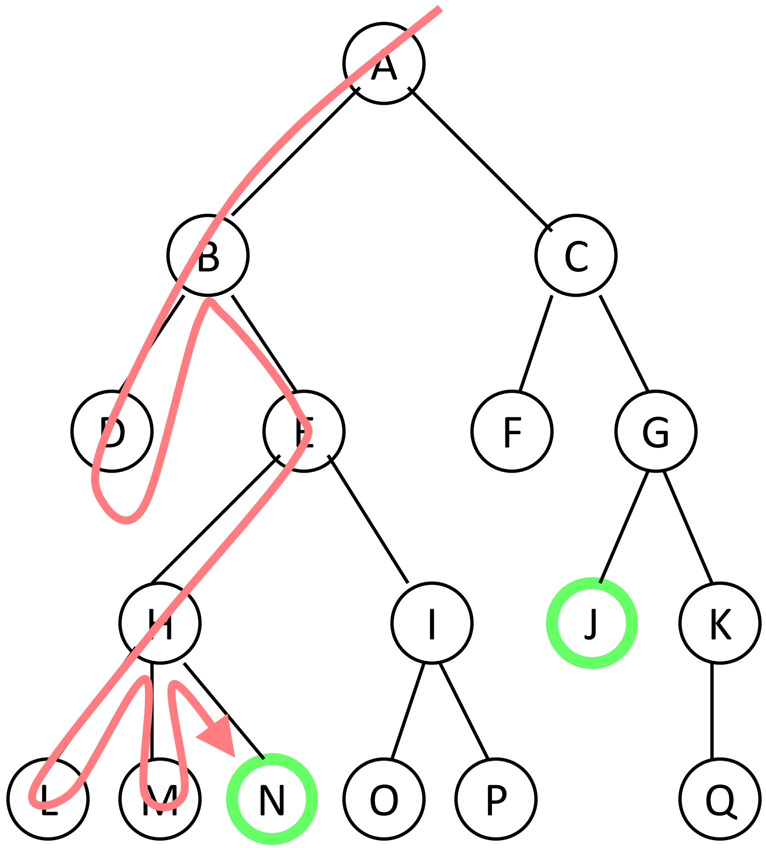
DFS- rules

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

DFS Algorithm

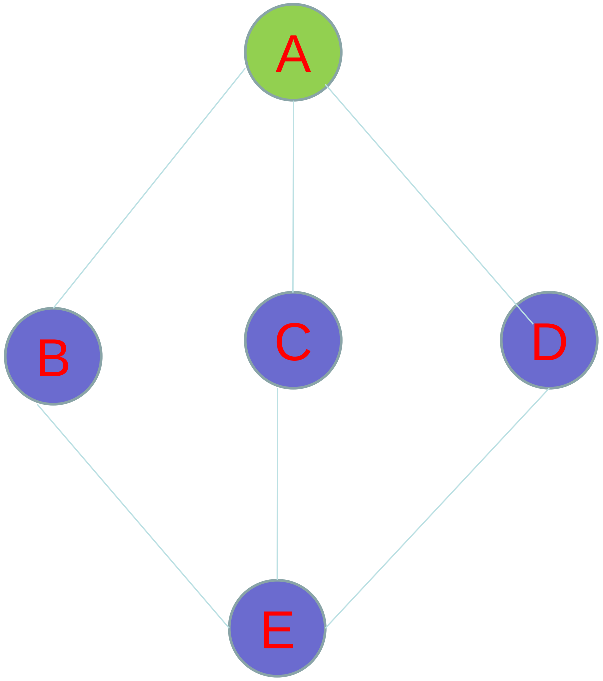
1. Push the starting vertex into the stack OPEN
2. While OPEN is not empty do
3. POP a vertex v
4. If v is not in VISIT
5. Visit the vertex v
6. Store v in VISIT
7. Push all the adjacent vertices of v onto OPEN

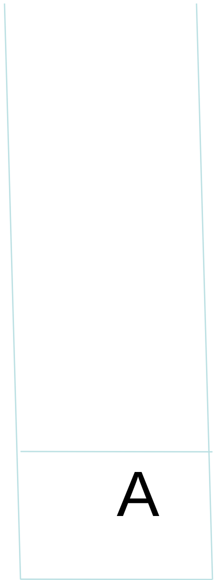
Depth-first searching



- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

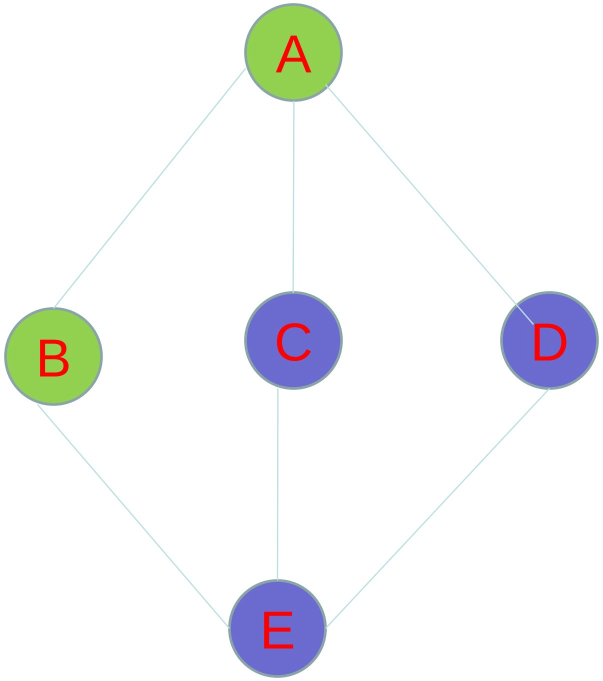
DFS example

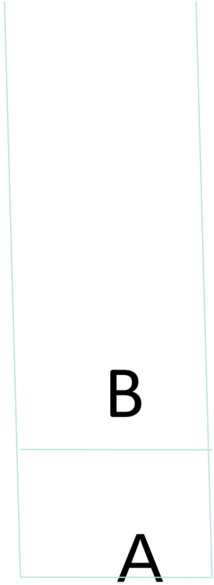




-
- Initially all nodes are in ready state
- Let the starting node be A. Push it on to stack & display it
- **Output: A**

DFS example

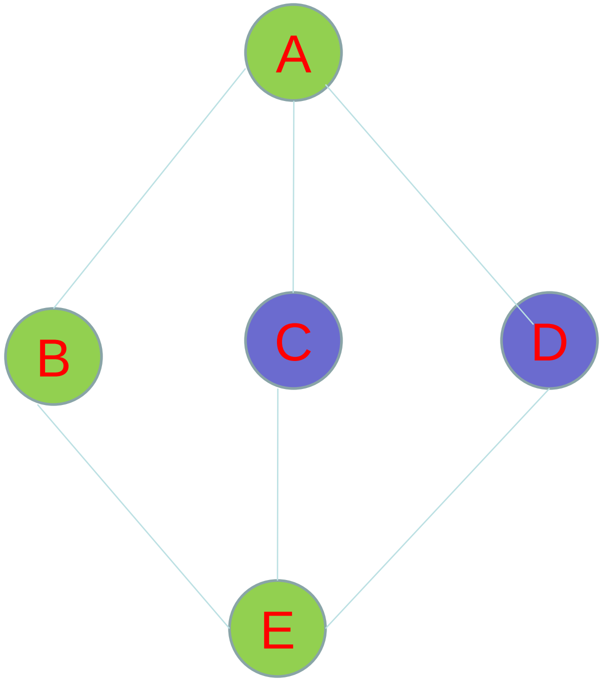


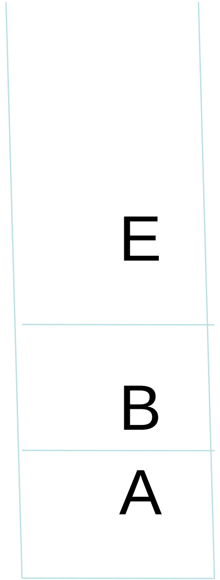


1. Push the adjacent unvisited vertex B onto stack and print it

Output: A B

DFS example

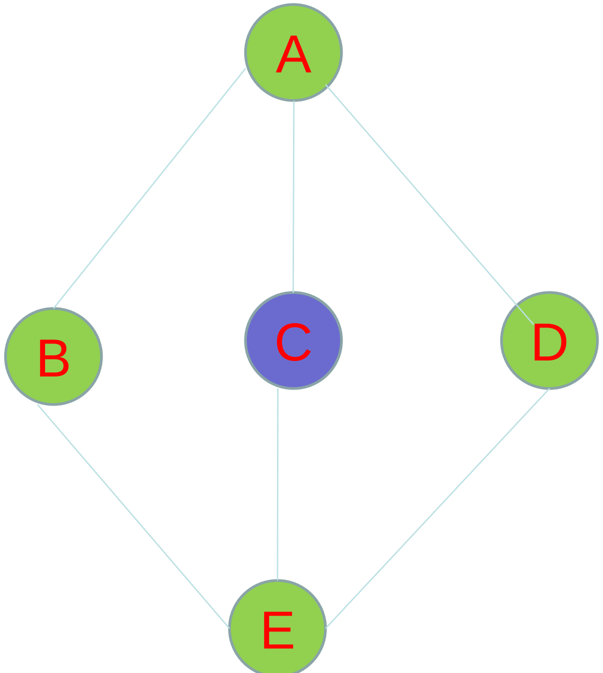


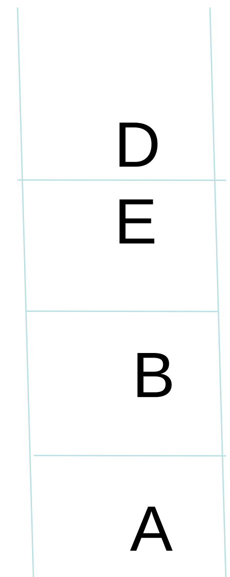


1. Push the adjacent unvisited vertex E onto stack and print it

Output: A B E

DFS example



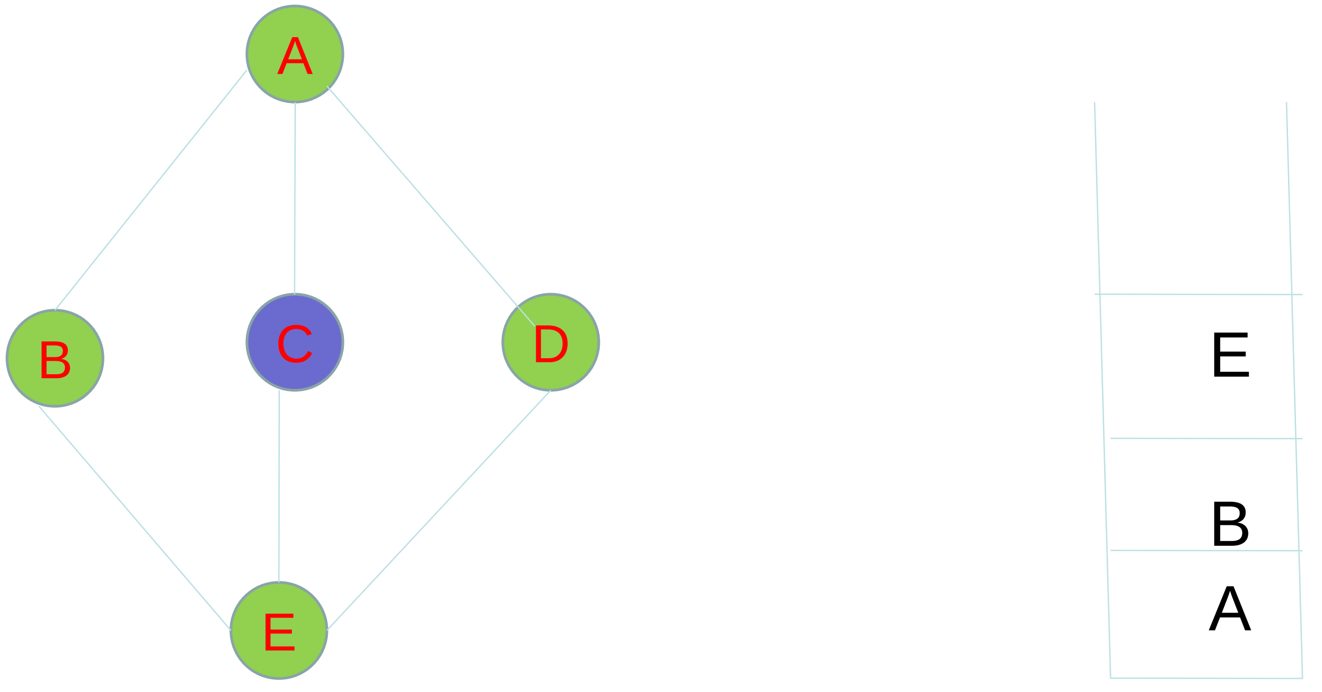


-

1. Push the adjacent unvisited vertex D onto stack and print it

Output: A B E D

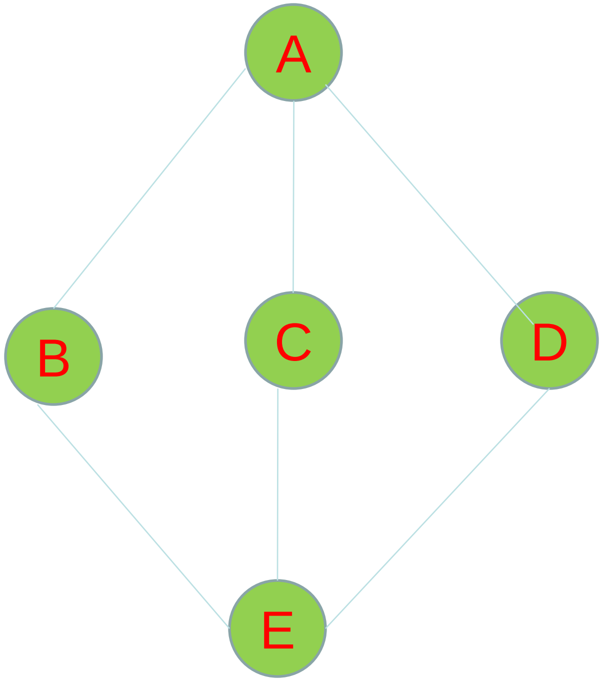
DFS example

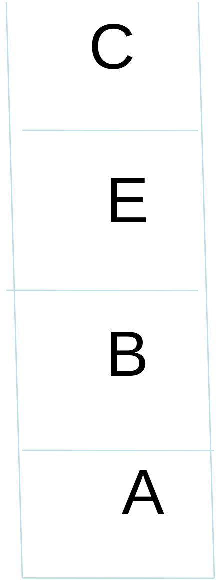


1. Now no adjacent unvisited neighbor for D
2. Pop it and find the unvisited adjacent vertex of stack top

Output: A B E D

DFS example

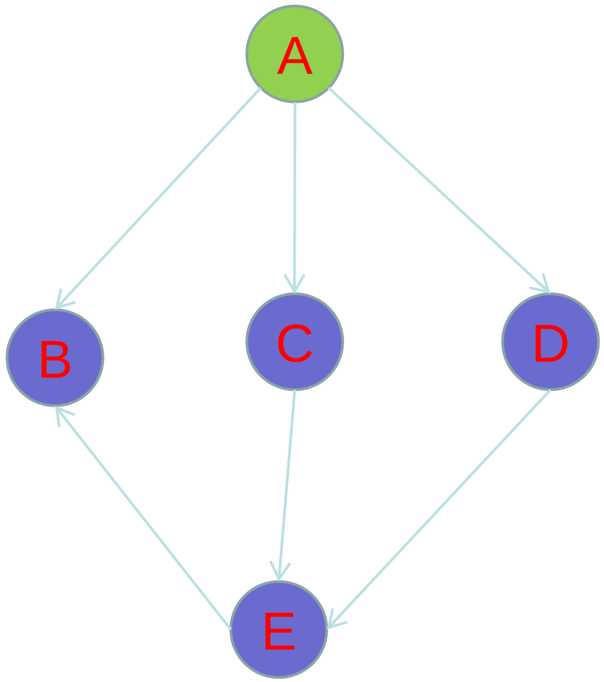


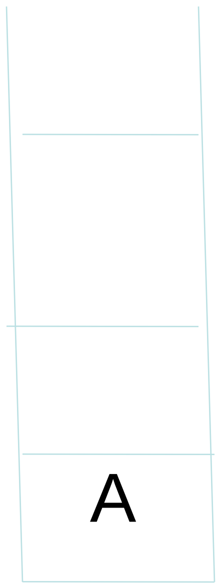


1. Push C on stack and print it
2. Now no unvisited vertex!!

Output: A B E D C

DFS example:2

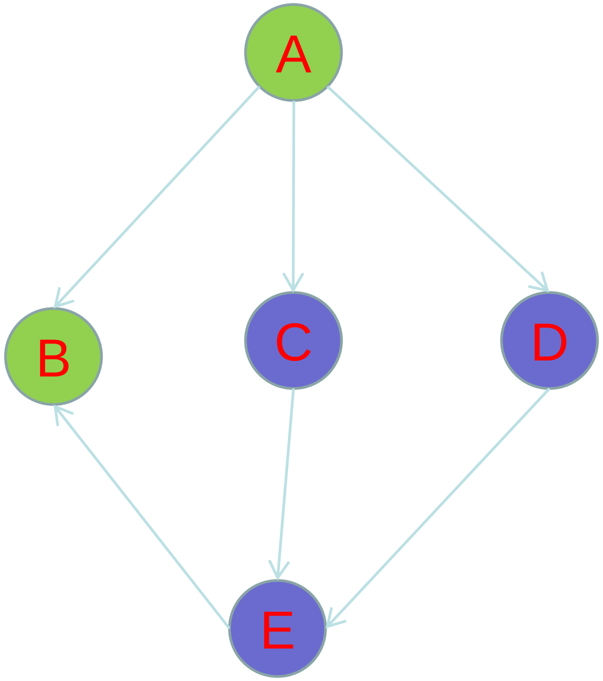


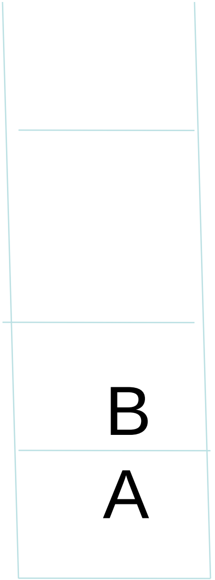


1. Push A onto stack

Output: A

DFS example:2

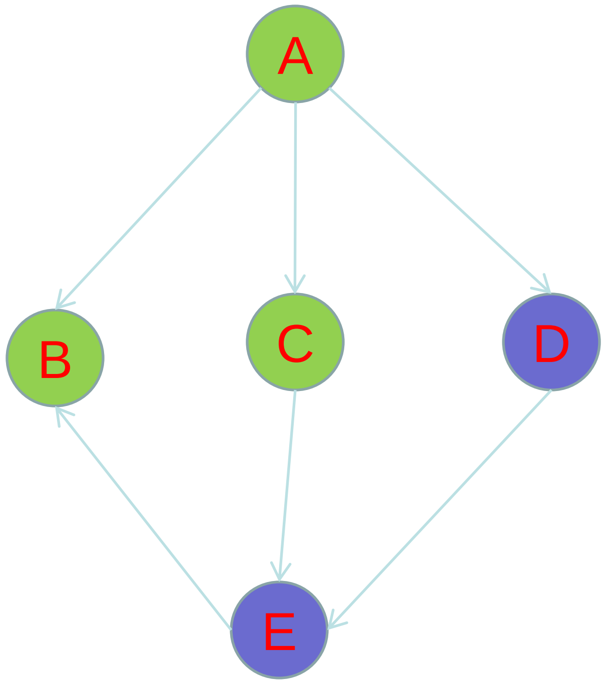


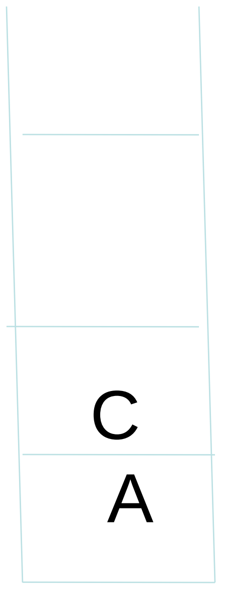


1. Push one of the unvisited adjacent vertex B on to stack

Output: A B

DFS example:2

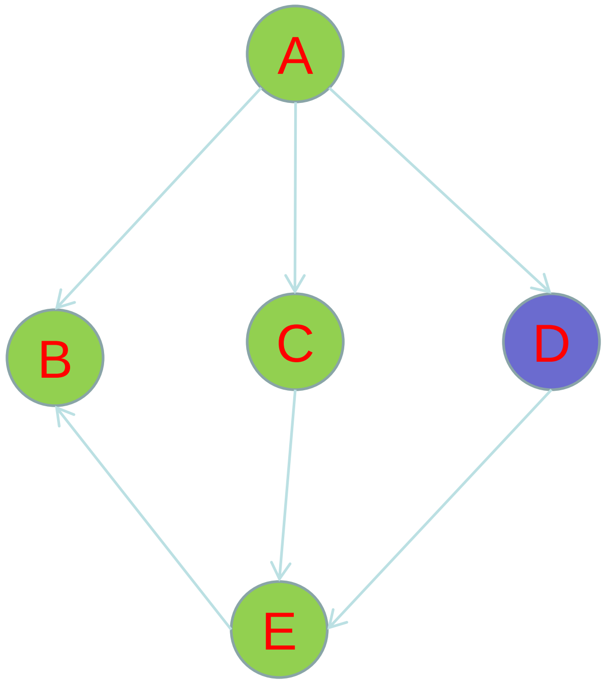


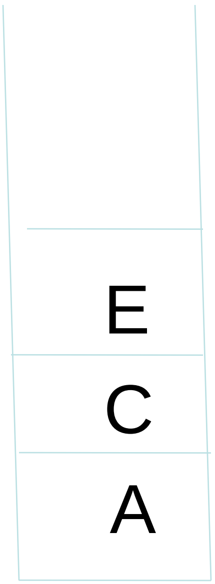


1. Now no adjacent neighbor for B.
2. So pop B and push another adjacent vertex of A onto stack

Output: A B C

DFS example:2

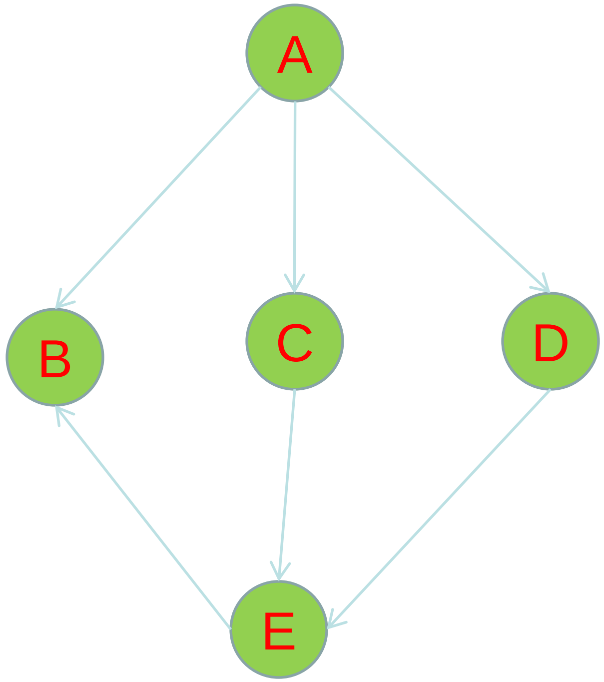


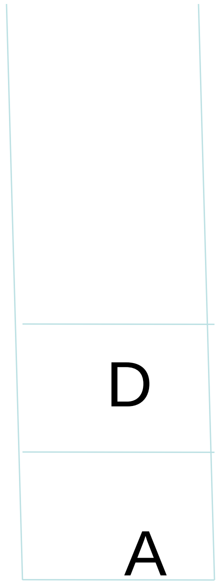


1. Push E on stack

Output: A B C E

DFS example:2

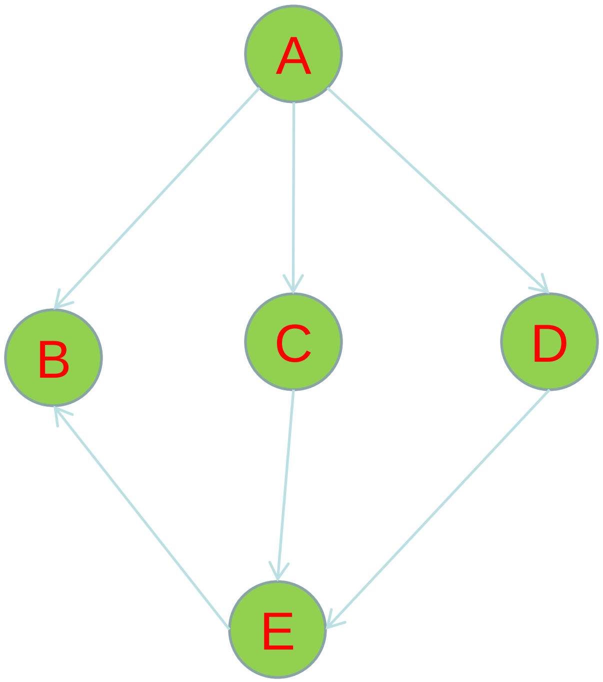




1. No unvisited vertex for E.
2. pop E
3. No unvisited adjacent vertex for C also. So pop C
4. Push the unvisited vertex D on to stack and print it

Output: A B C E D

DFS example:2



1. Now no unvisited vertex

Output: A B C E D