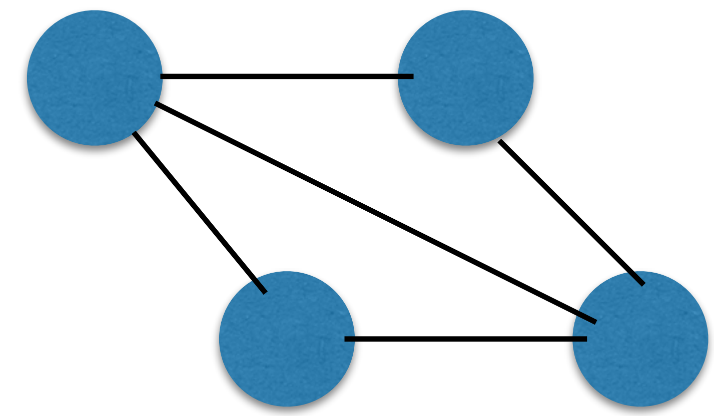
Graphs and Centrality

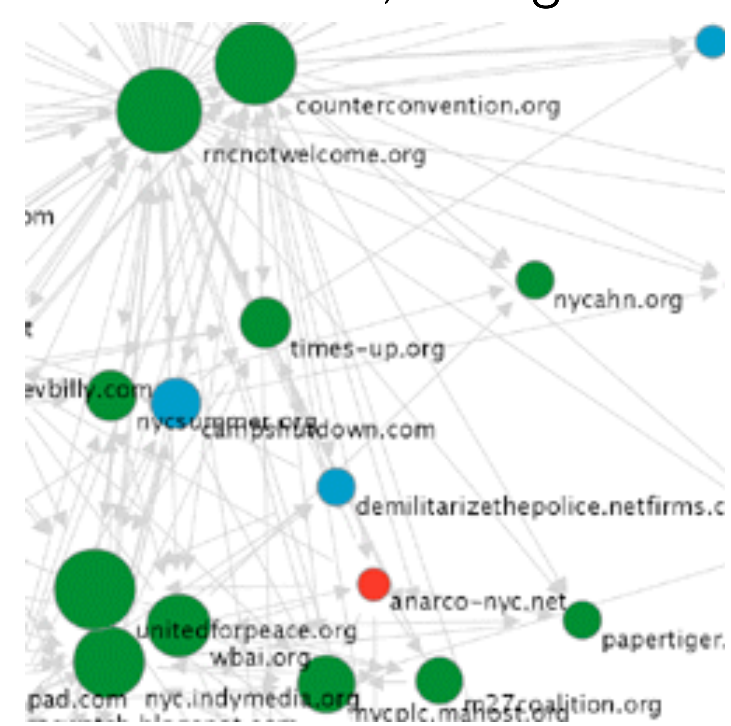
Professor Kevin Gold

The Graph Concept

- A graph consists of things (**vertices**) and connections between those things (**edges**)
- The connections can be asymmetric (directed graph) or symmetric (undirected graph)
  - Asymmetric/directed: Webpage A links to webpage B, but B doesn't link to A
  - Symmetric/undirected: We are Facebook friends — it's symmetric



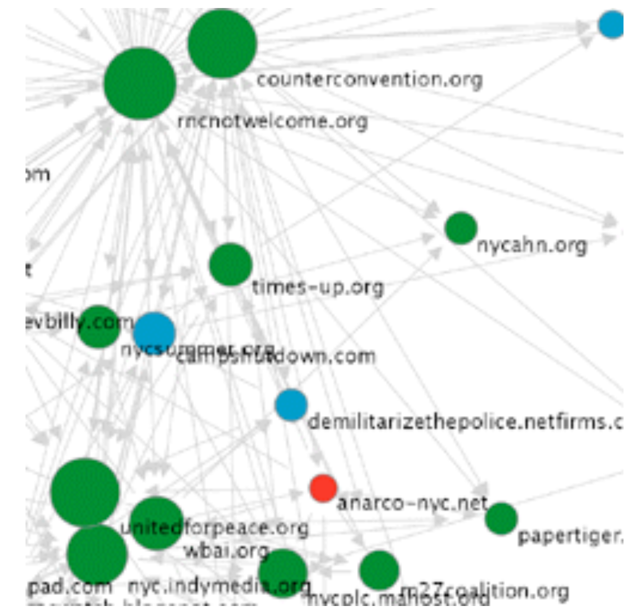
Undirected graph with 4 vertices, 5 edges



Piece of a directed graph of the web

web graph source: http://farrall.org/webgraph/home.html

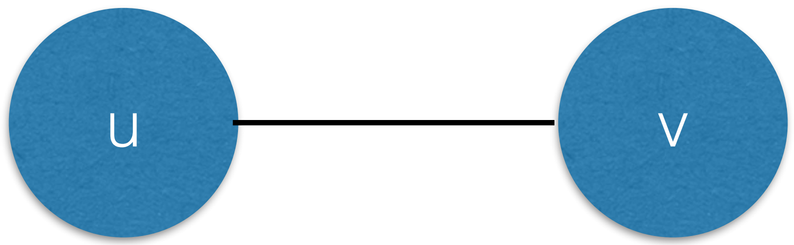
Graph Algorithms Have Many Applications

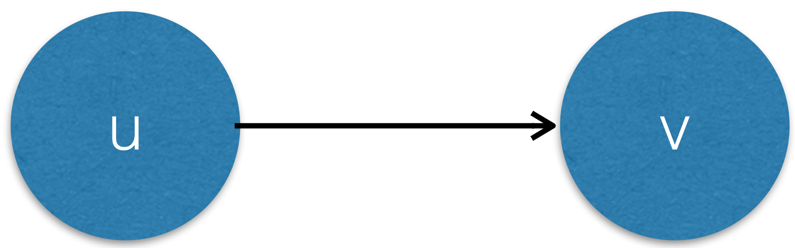


- The World Wide Web - analyze structure to find important pages (PageRank)
- Social networks - Try to find influential individuals
- Computer networks - Find a path for data packets
- Pathfinding - Google Maps
- Graph databases (Neo4j) - for data structured like a graph
- Classic AI - treat puzzles as graphs to pathfind on
- And more!

More Specifically, a Graph is…

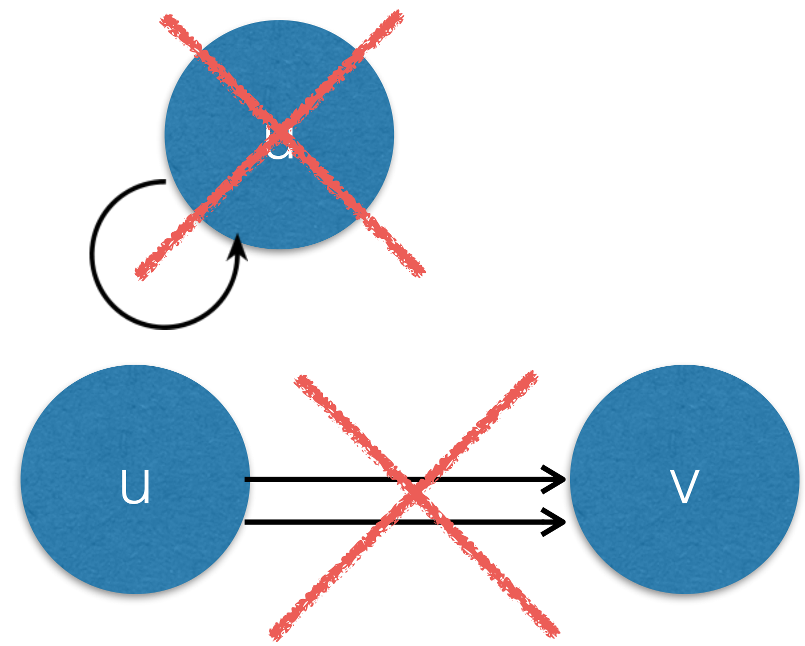
- A graph G consists of a set of vertices V (also called **nodes**) and a set of edges, E
    - The *number* of vertices and edges is referred to as |V| and |E|
    - In an *undirected* graph, the edges are two-element subsets of V: {u, v}. No arrows on the drawing.
    - In a *directed* graph, the edges are ordered pairs of vertices (u, v). We draw an arrow from u to v.
- The **degree** of a vertex is the number of edges it touches ("**in-degree**" and "**out-degree**" for directed)





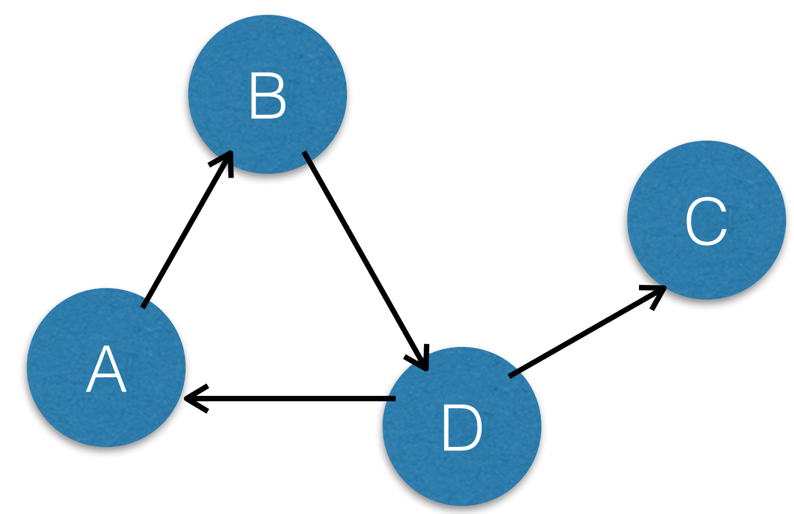
More Specifically, a Graph is…

- Generally assume no self-loops
- Usually not multiple edges connecting the same two vertices — (u,v) is in the graph or not
  - This would be called a "multigraph" (unusual)



Paths and Cycles

- "Is there a way to get from this vertex to that one?"
- A *path* from vertex A to vertex B is a sequence of vertices that leads from A to B — edges connect each vertex (going the right direction, if directed)
  - If all vertices distinct, it is a "simple path"
- A *cycle* is a path from a vertex to itself that contains at least one other vertex and doesn't repeat other vertices



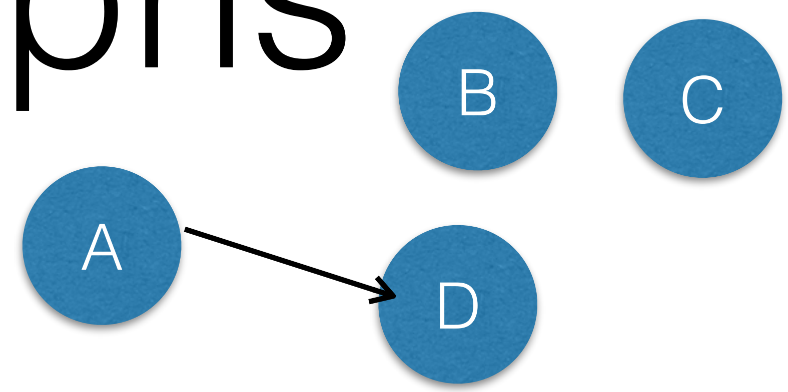
A,B,D,C: simple path from A to C

A,B,D,A,B,D,C: non-simple path from A to C

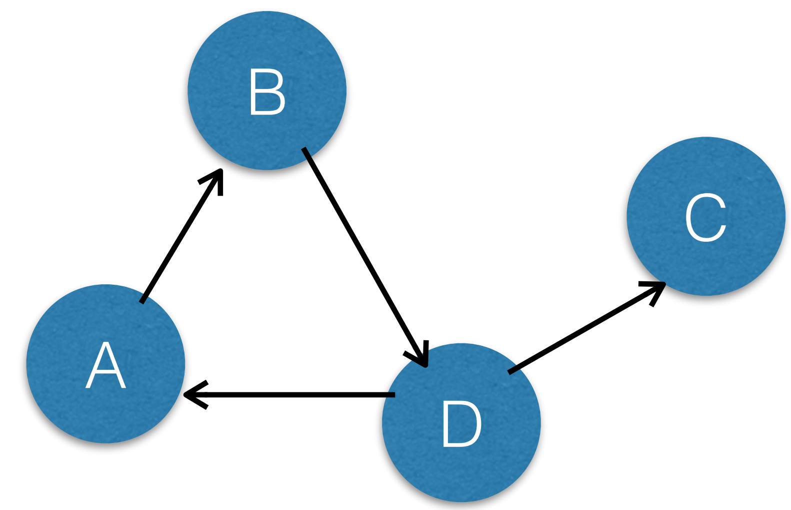
A,B,D,A: cycle

Connected Graphs

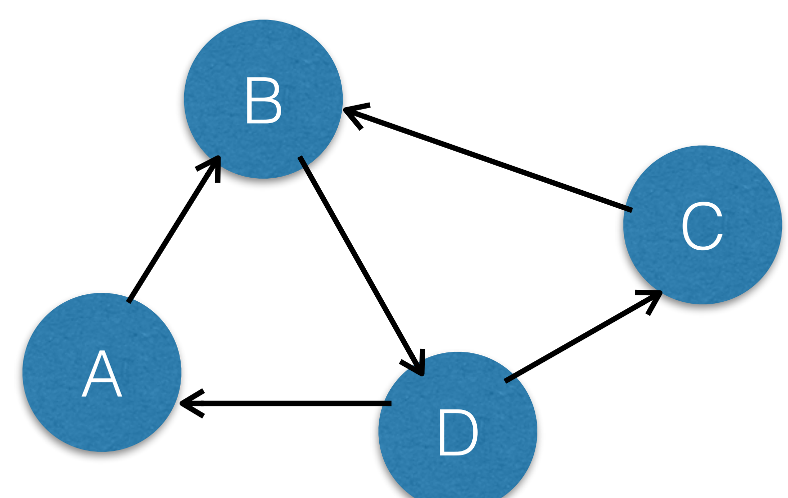
- An undirected graph is **connected** if, for every pair of nodes u and v, there is a path from u to v.
- In directed graphs, this has two varieties.
    - A directed graph is **weakly connected** if it would be connected if it were undirected (the paths can “go the wrong way”)
    - A directed graph is **strongly connected** if there is a path (obeying the directions) from every node u to every other node v.
- Some algorithms assume graphs are connected in one of these ways.



not connected (B,C)

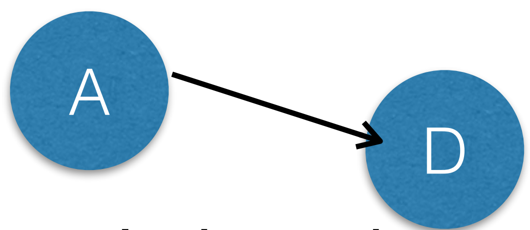


weakly connected (C)

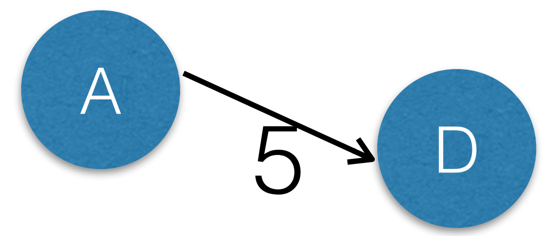


strongly connected

Weights



unweighted graph



weighted graph

- By default, graphs are "unweighted": the distance between two **neighbors** (nodes connected by an edge) is assumed to be 1.
  - And the *shortest path* between two nodes is the path between those nodes that uses the *fewest edges*
- **Weighted graphs** can represent distances and costs of paths between vertices (each edge gets a number that is its weight)
  - Shortest path is path with smallest sum of weights.

Trees

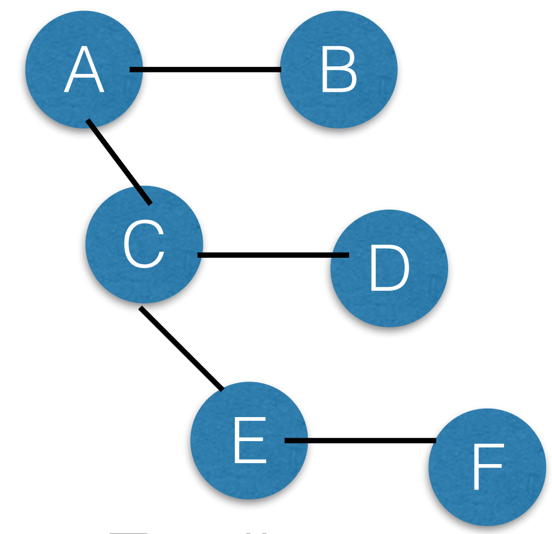
- A **tree** is any connected graph that has no cycles. It doesn’t necessarily need to be hierarchical.

- You can choose one node to be the **root** — grab the tree and let the rest of the graph “hang” from that.

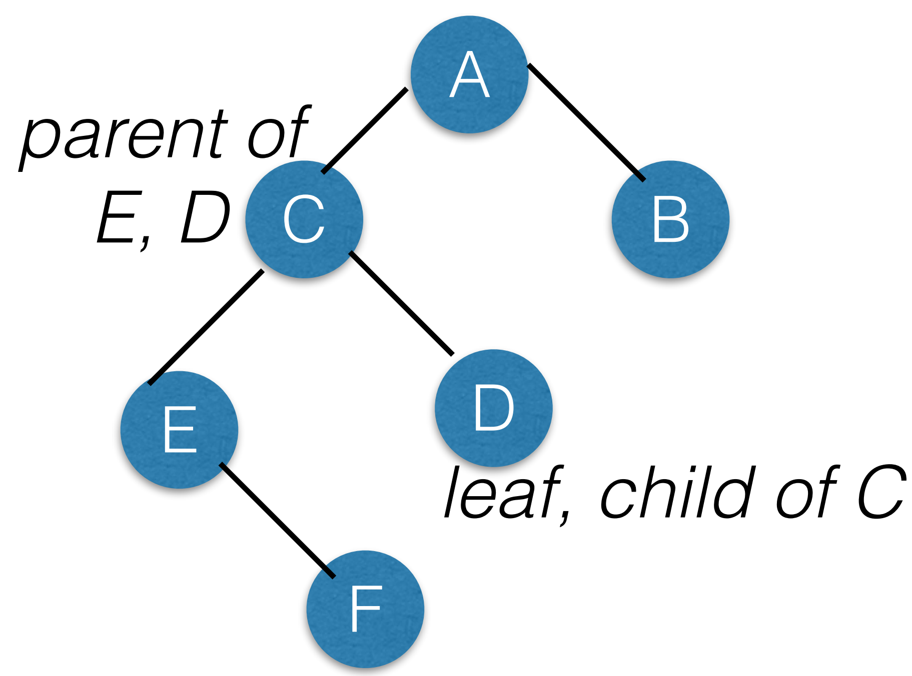
    - Each node besides the root will have one neighbor closer to the root — its **parent**

        - …So a tree has exactly how many edges?

    - The **depth** of a node is its distance from the root



Totally a tree



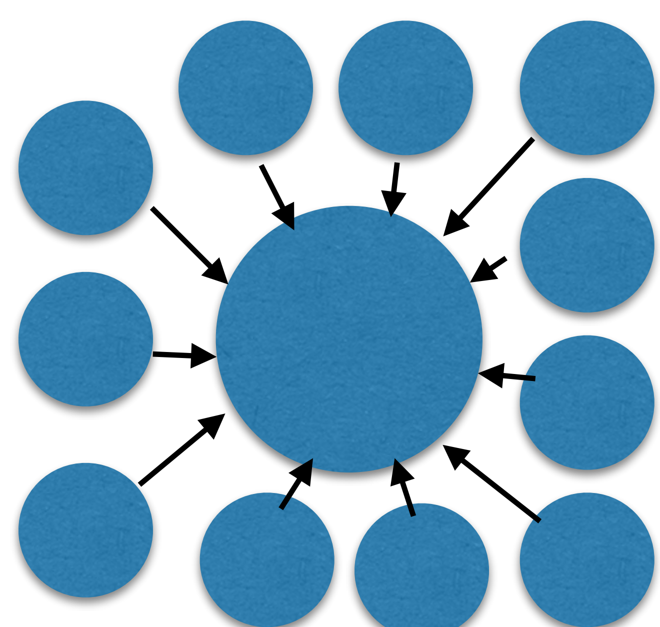
Same tree, if A is root

Who's Important in this Network? **Centrality**

- Data science task: here is a graph of a social network; who is most influential?
- Influence could come in several forms:
  - Degree centrality - most direct friends
  - Closeness centrality - just a few hops from anyone
  - Betweenness centrality - a "gatekeeper" between subgraphs
  - Eigenvector centrality - considered important by important nodes

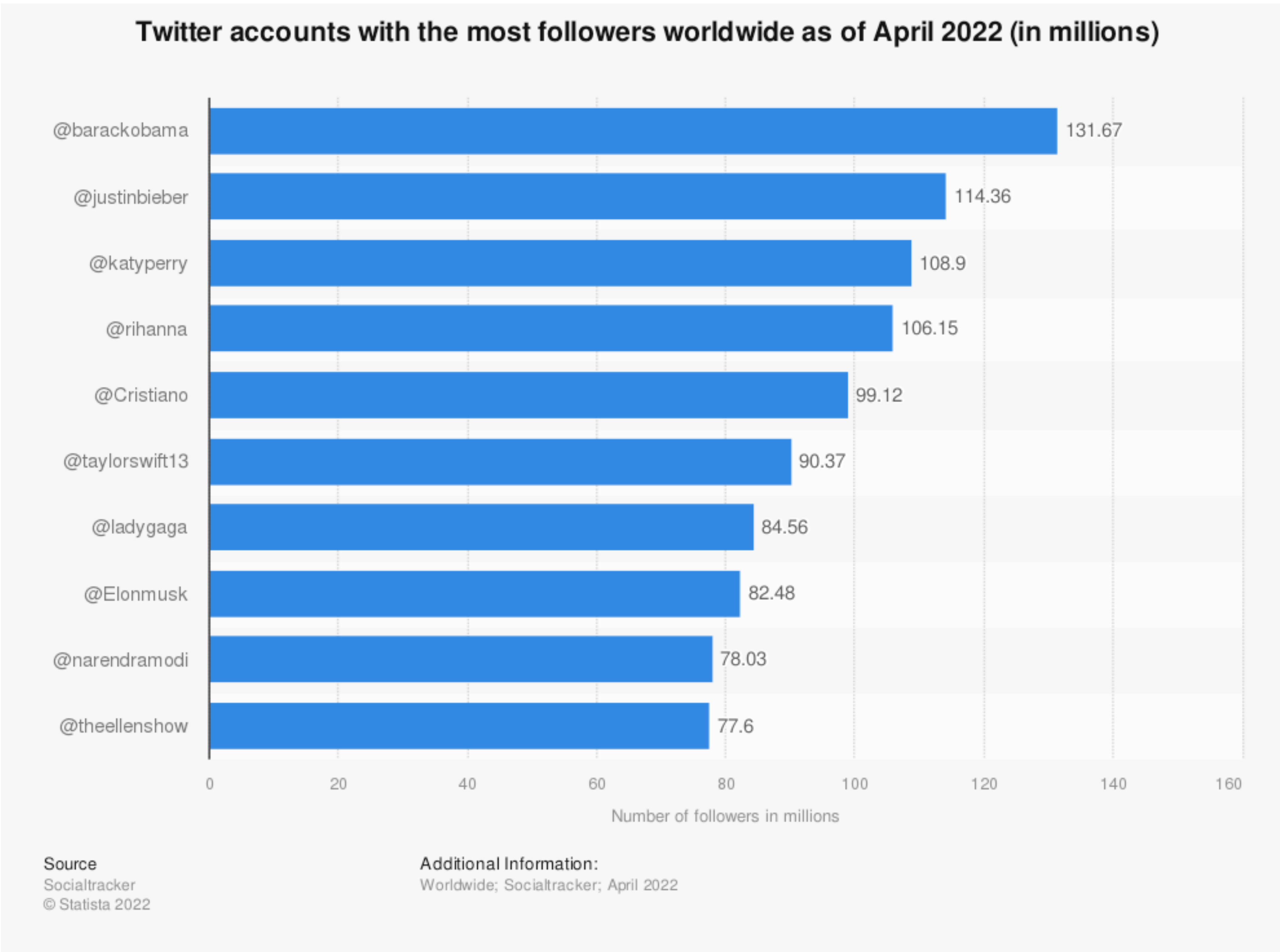
Degree Centrality

- Simply the degree (number of neighbors) of the vertex
- If graph is directed, could focus on in-degree or out-degree as appropriate
- Example: someone with a lot of Twitter followers is important
    - A little easy to game with bots and fake accounts



High in-degree centrality

Degree Centrality

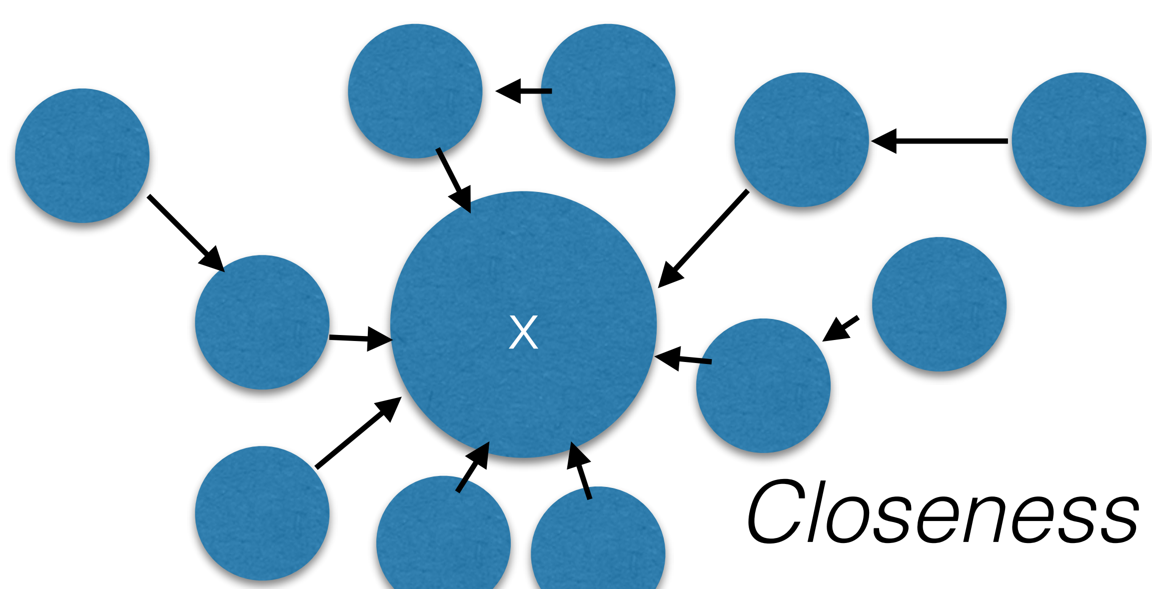


Closeness Centrality

- The inverse of the average number of steps in a closest path to each other node

- $C(x) = \frac{N-1}{\sum_y d(y,x)}$ where N-1 is a count of all the other nodes besides *x* and d(y,x) is a shortest path distance

- Larger for nodes that are just a few hops away from the whole graph - a broadcast message has "reach"

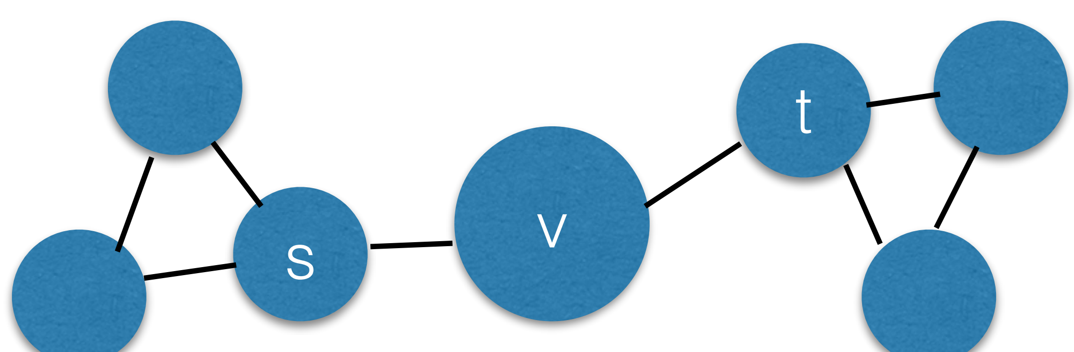


*Closeness 11/(7*1+4*2)=0.73*

Betweenness Centrality

- Finds nodes that connect one community to another

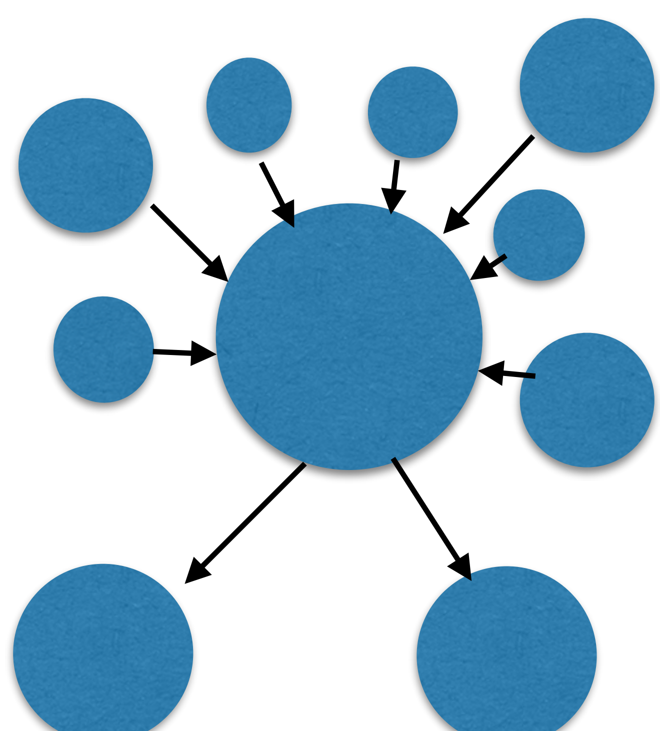
- Betweenness of v is sum over start and end locations of proportion of shortest paths that pass through v:
$$\sum_{s \neq v \neq t} \frac{\sigma_{svt}}{\sigma_{st}}$$ where $\boldsymbol{\sigma}_{svt}$ is a count of shortest paths from s to t through v and $\boldsymbol{\sigma}_{st}$ is a count of shortest paths from s to t (maybe including v, maybe not)



s,t on opposite sides of v contribute 1, on same side contribute 0

Eigenvector Centrality

- A term for what Google's PageRank does
- With some linear algebra, calculates the proportion of time a random walk through the graph would pass through any particular node
- Being linked to by popular sites does more to increase your popularity than links from small sites - thus harder to "game"



Finding Shortest Paths: Breadth-First Search

- Closeness and betweenness centrality both need to find shortest paths to calculate them

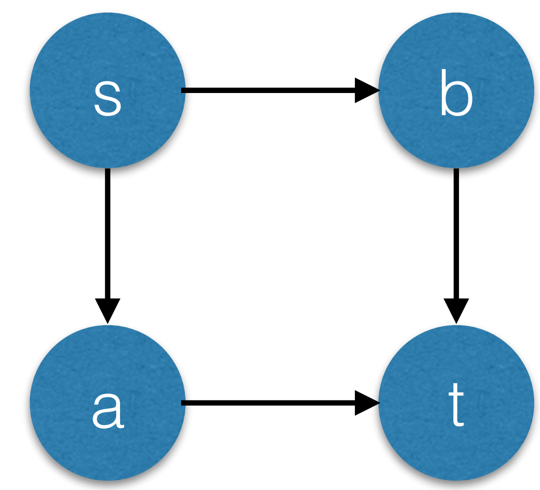
- There are two fundamental algorithms for finding a path from one node to another on an **unweighted** graph: **breadth-first search** and **depth-first search**

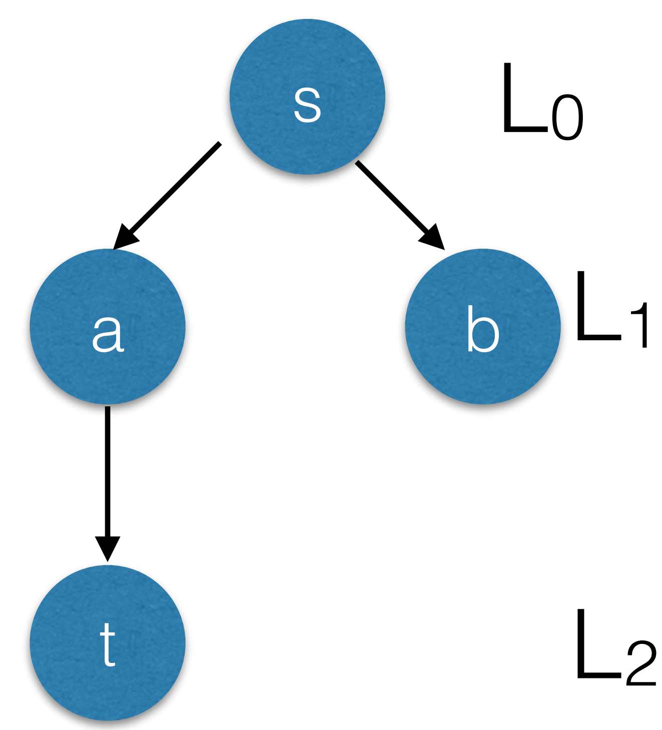
- But breadth-first search finds the *shortest* path, and depth-first search generally doesn't.

- General BFS idea: fan out from the start, methodically trying all closer vertices before all farther ones

Breadth-First Search

- Basic idea: Visit all nodes one step away, then all nodes two steps away, then …
- In exploring, we can build a tree, where the nodes at depth *i* are *i* hops away from the start using a shortest path
- We can also use a **queue** to track which nodes to explore next — a **first-in, first out** list where new elements “go to the end of the line”

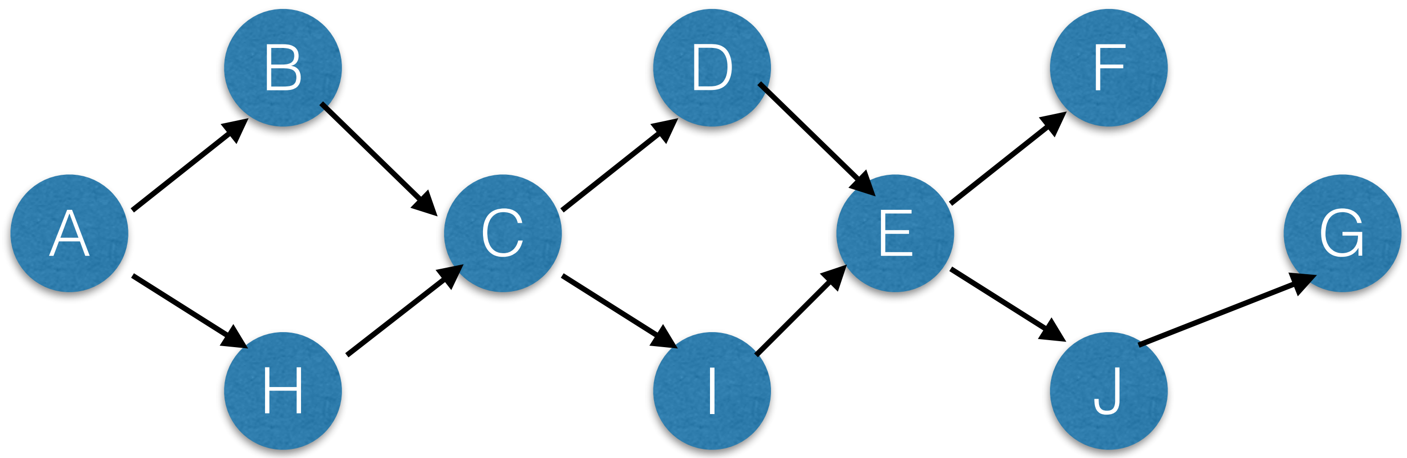




Breadth-First Search

- Add start node **s** to the **queue** of nodes to explore
- Initialize “**discovered**” set, tracking which nodes have been seen before, to **s**
- Initialize distance table with d[s] = 0 and create an empty parent table (representing the tree)
- While the queue of nodes to explore is not empty:
  - Pull off the node *p* at the **front** of the queue (oldest node in queue)
  - For each neighbor *n* of *p*:
    - If *n* is not in discovered set:
      - Add *p* as *n*'s parent in parent table; set d[n] to d[p]+1
      - If *n* is the target **t**, return the path to **t** (following the parent table from **t** back to **s**)
      - Add *n* to the **end** of the queue and add to “discovered” set
- Return "no path"

BFS Example

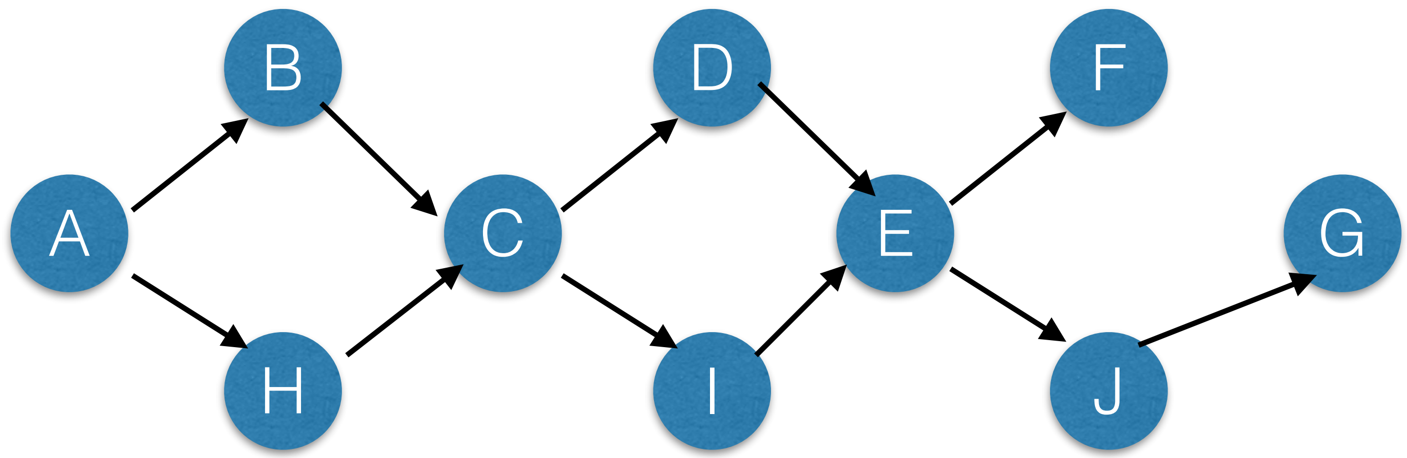


Queue

A

Tree

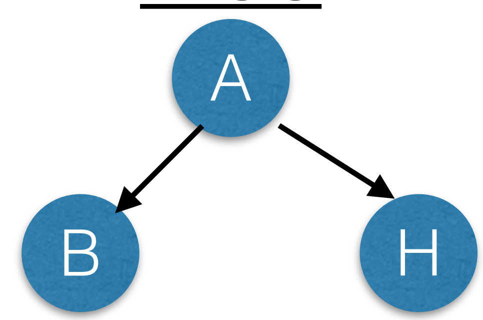
BFS Example



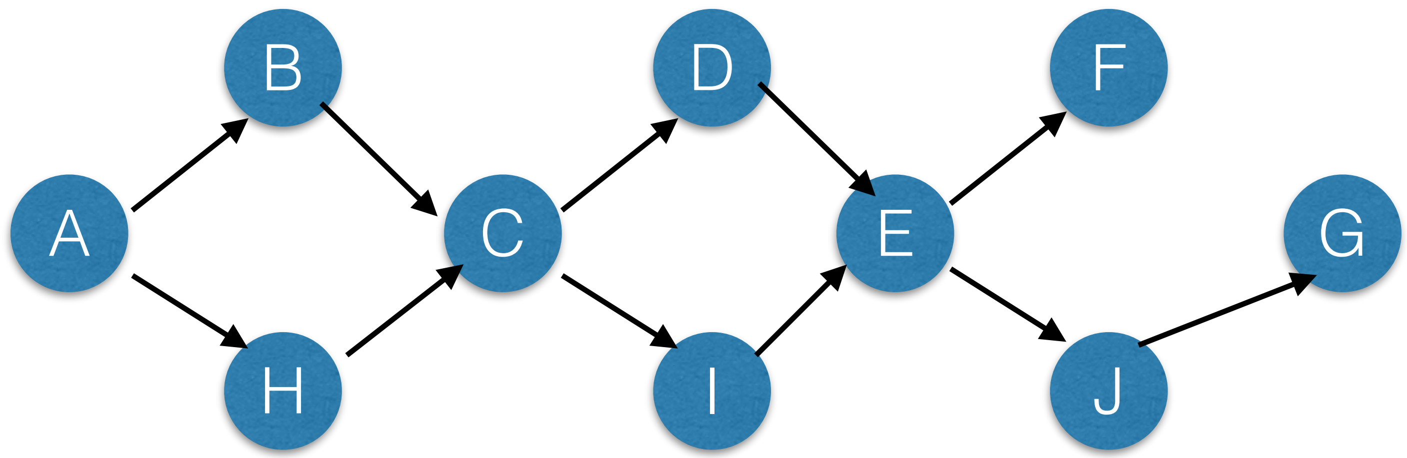
Queue

B, H

Tree



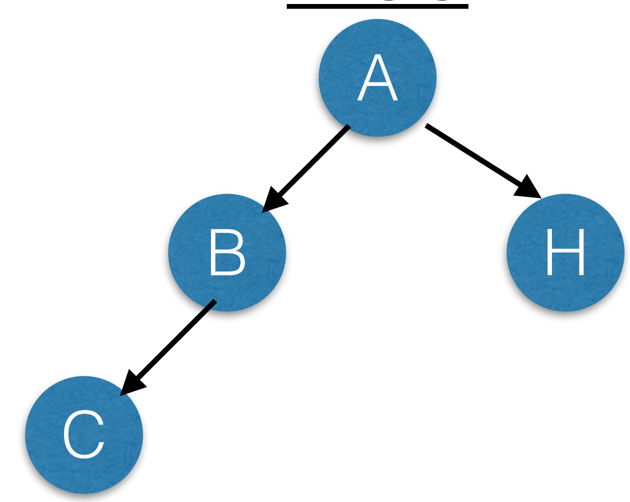
BFS Example



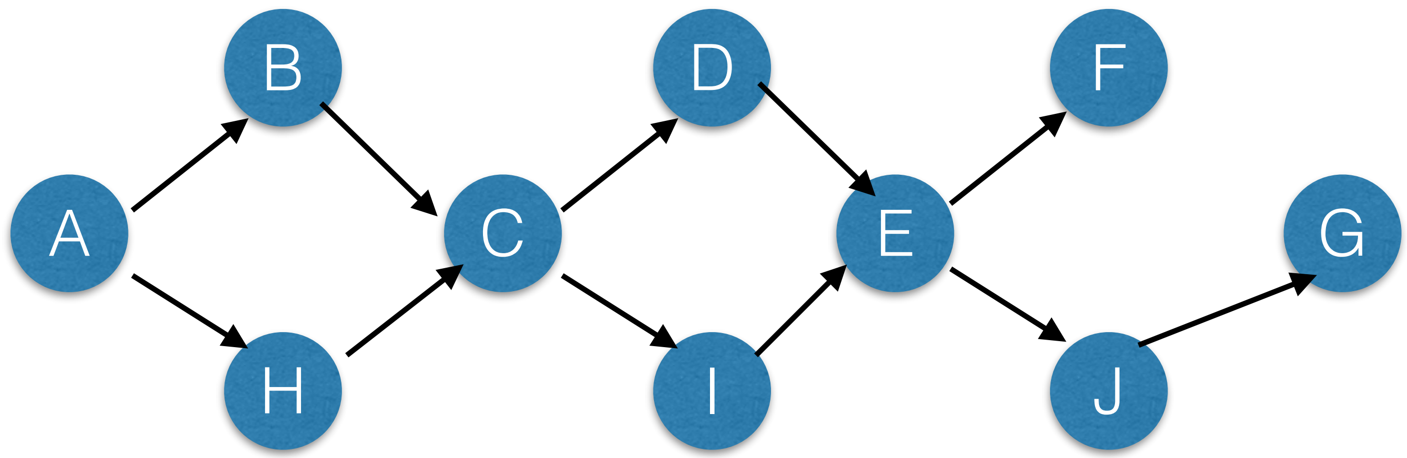
Queue

H,C

Tree



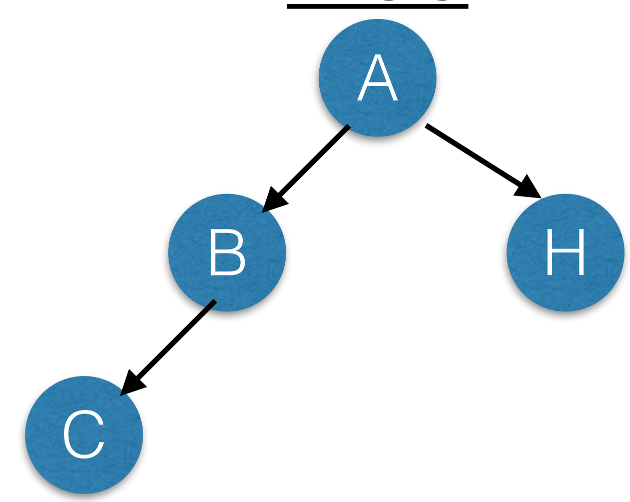
BFS Example



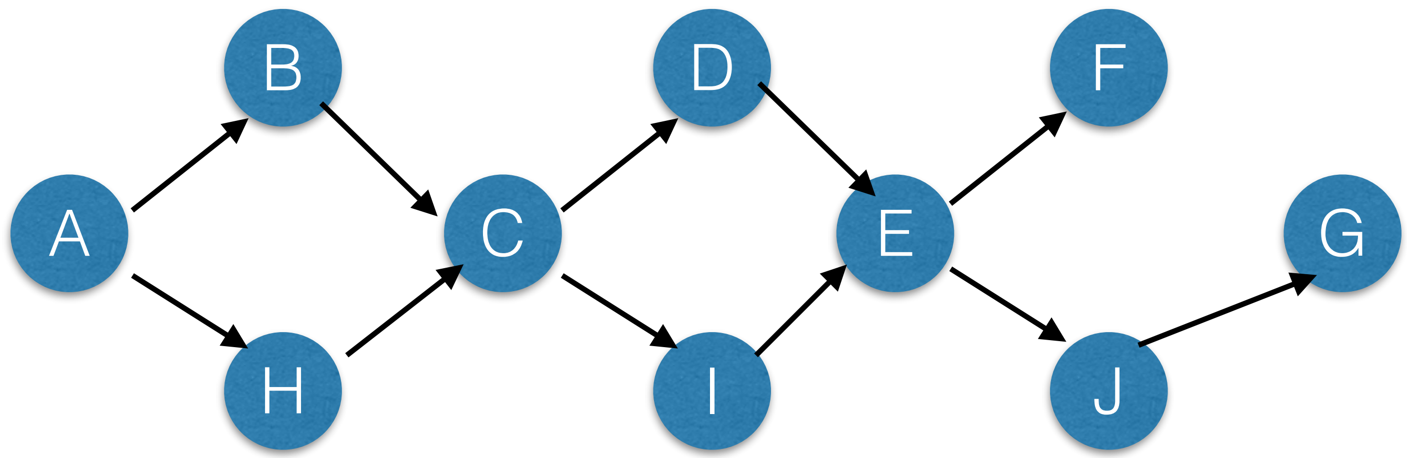
Queue

C

Tree



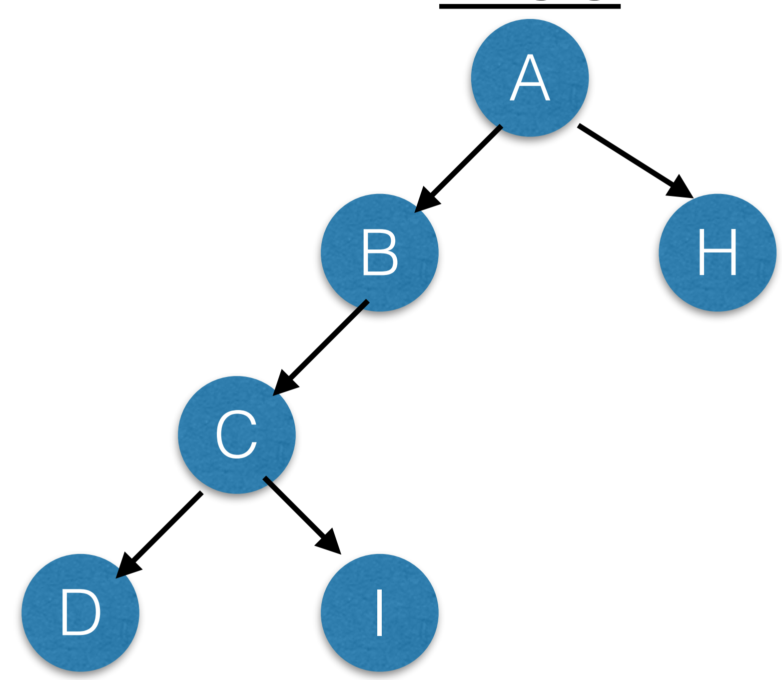
BFS Example



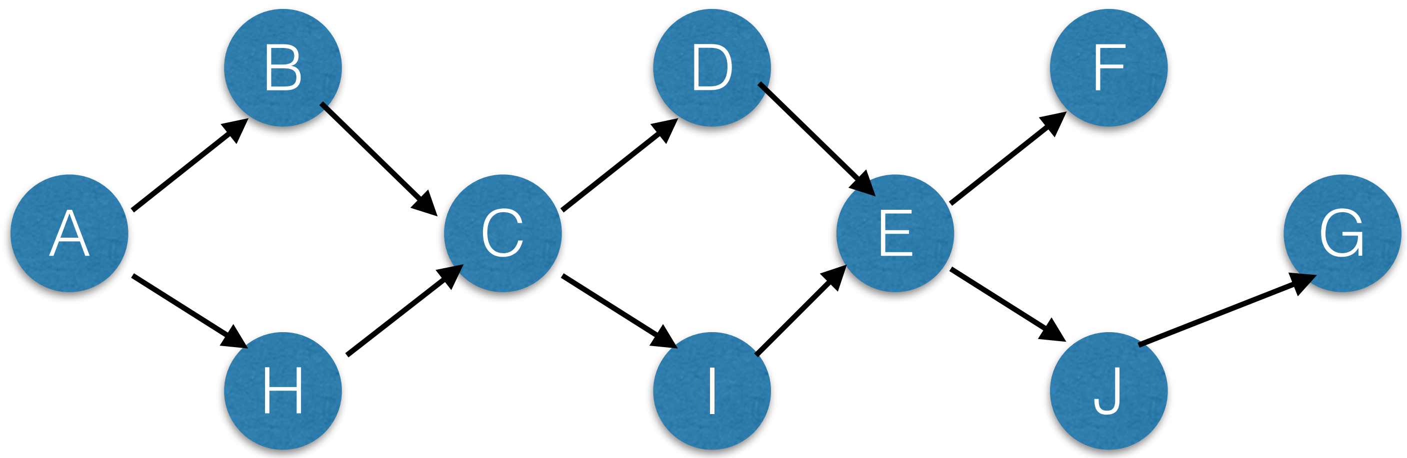
Queue

D,I

Tree



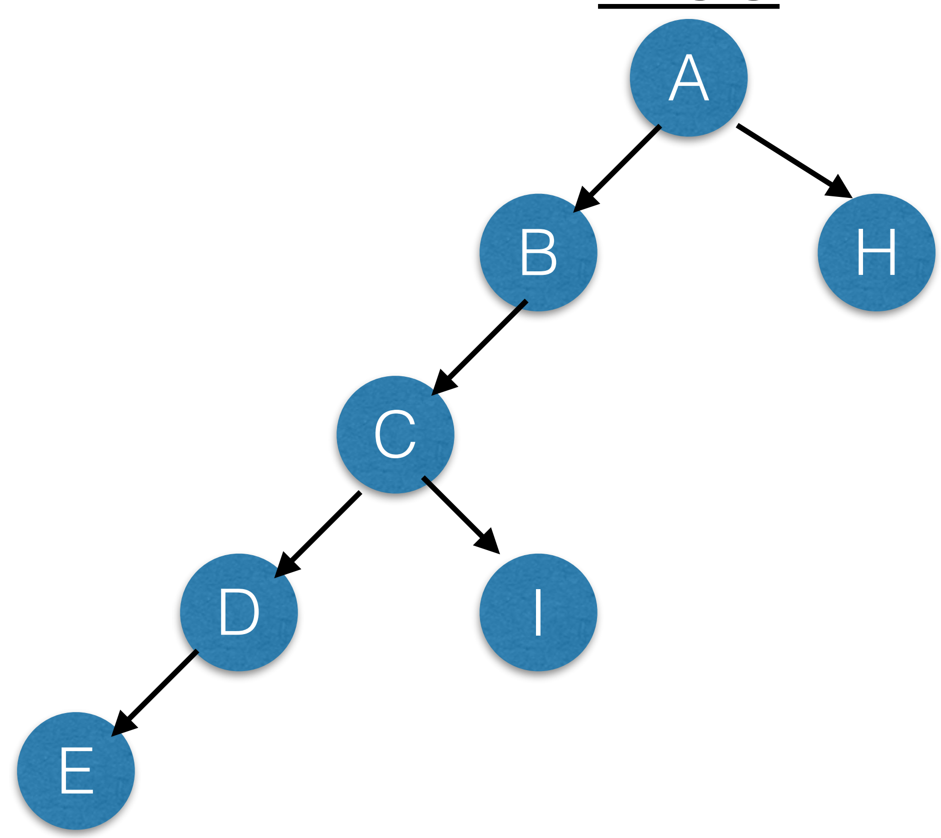
BFS Example



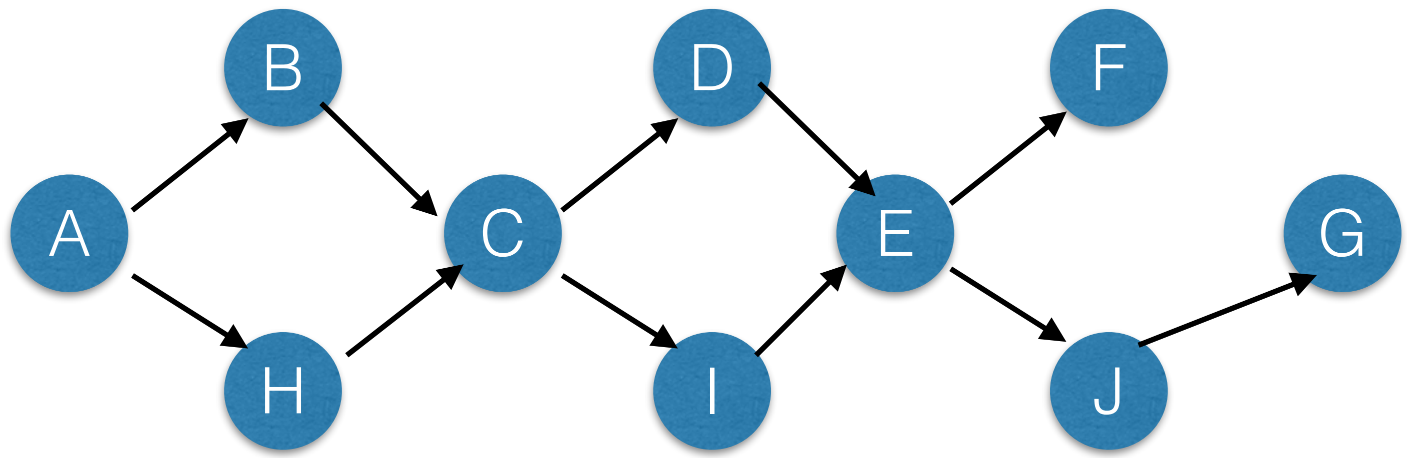
Queue

I,E

Tree

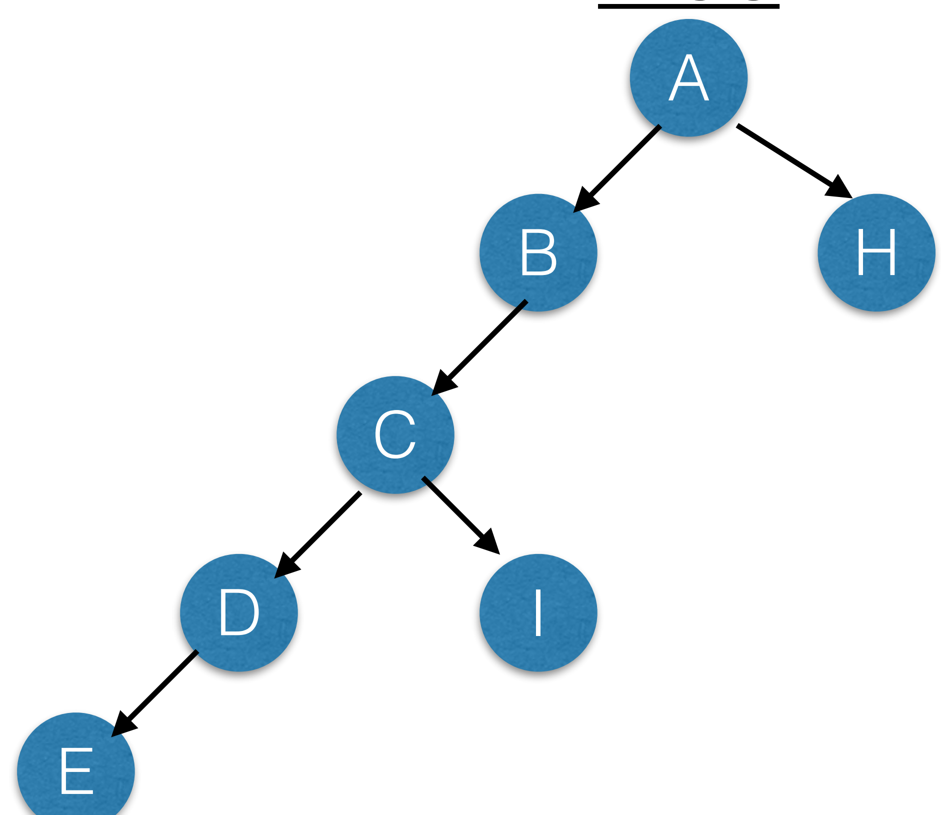


BFS Example

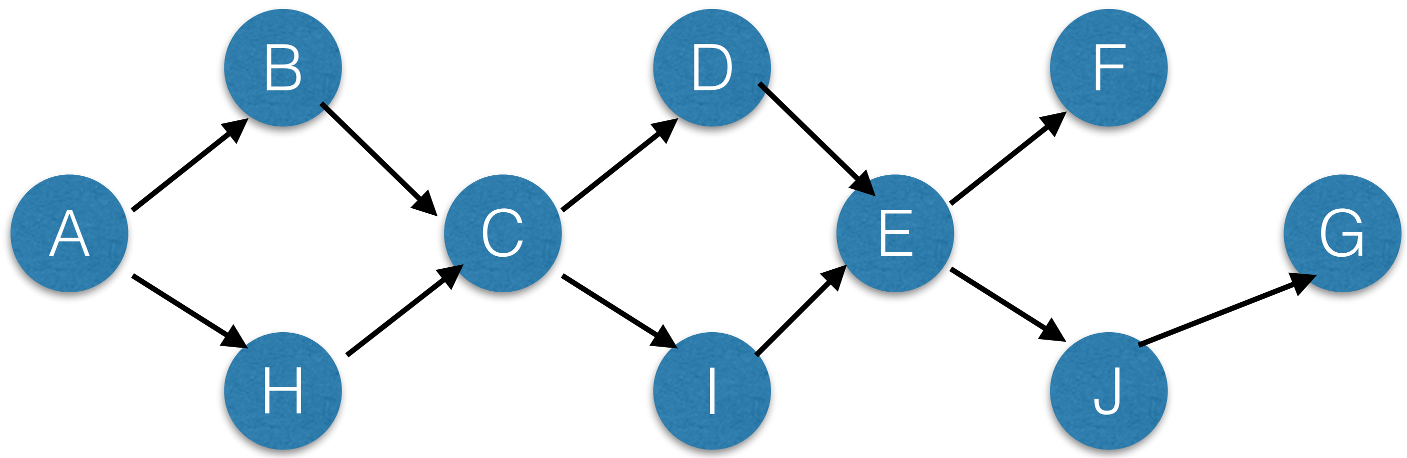


Queue
E

Tree



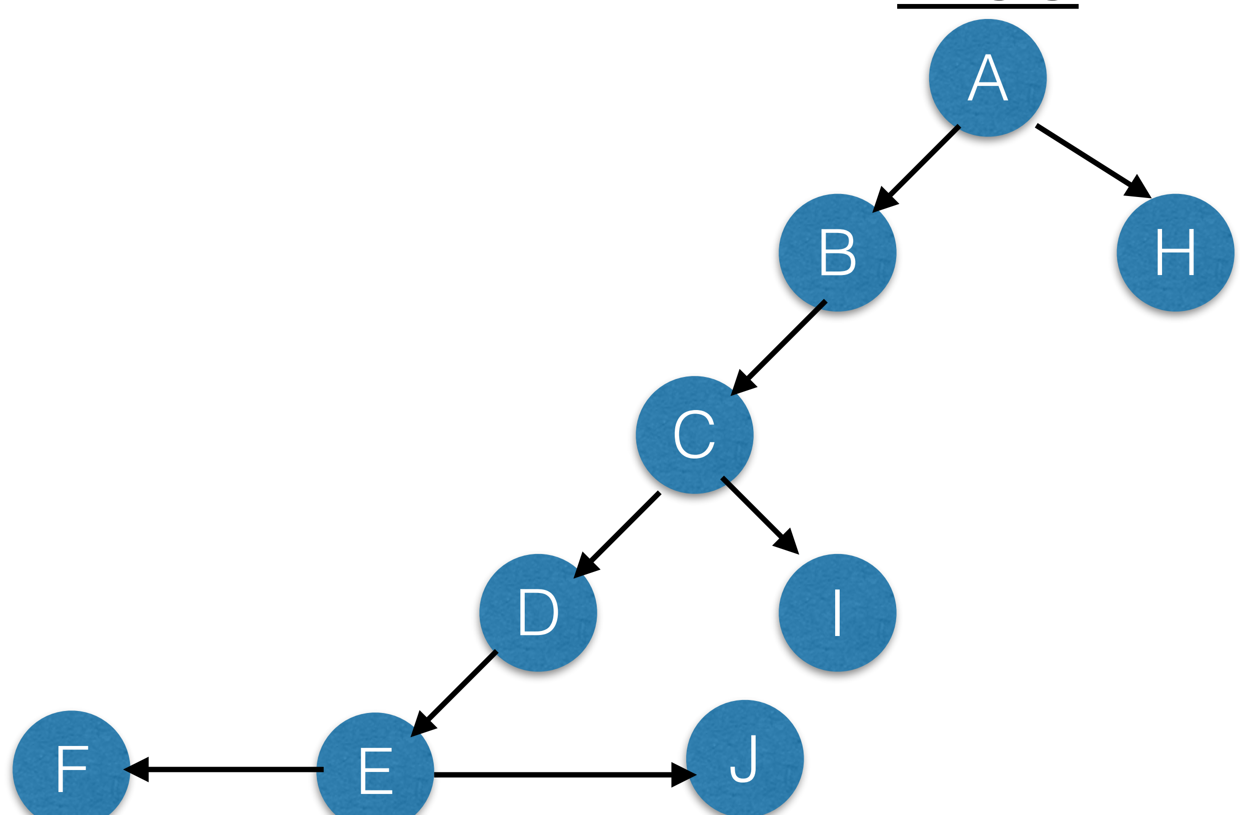
BFS Example



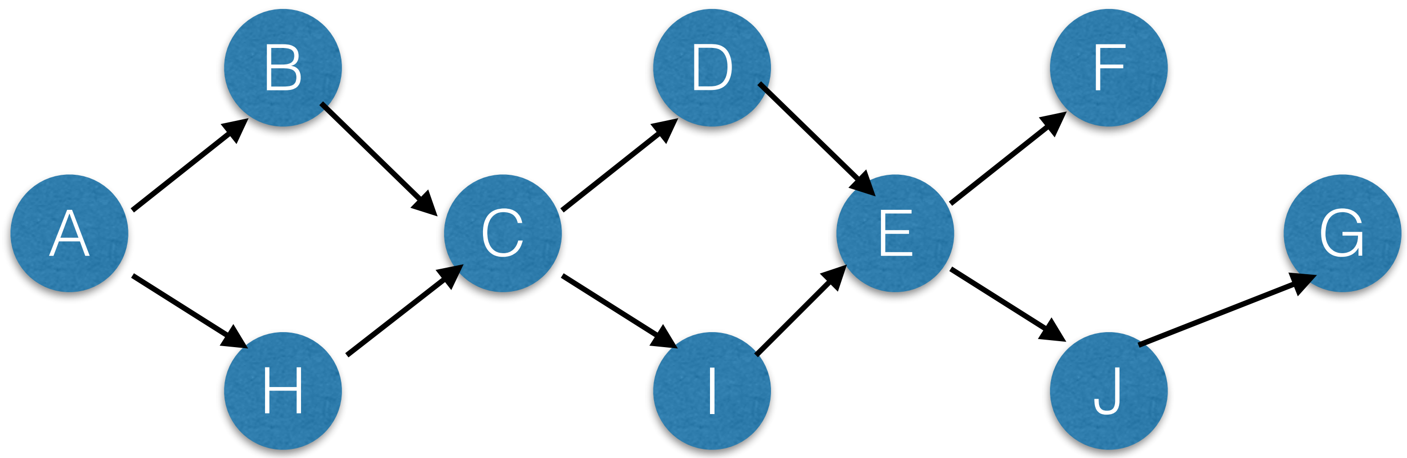
Queue

F,J

Tree



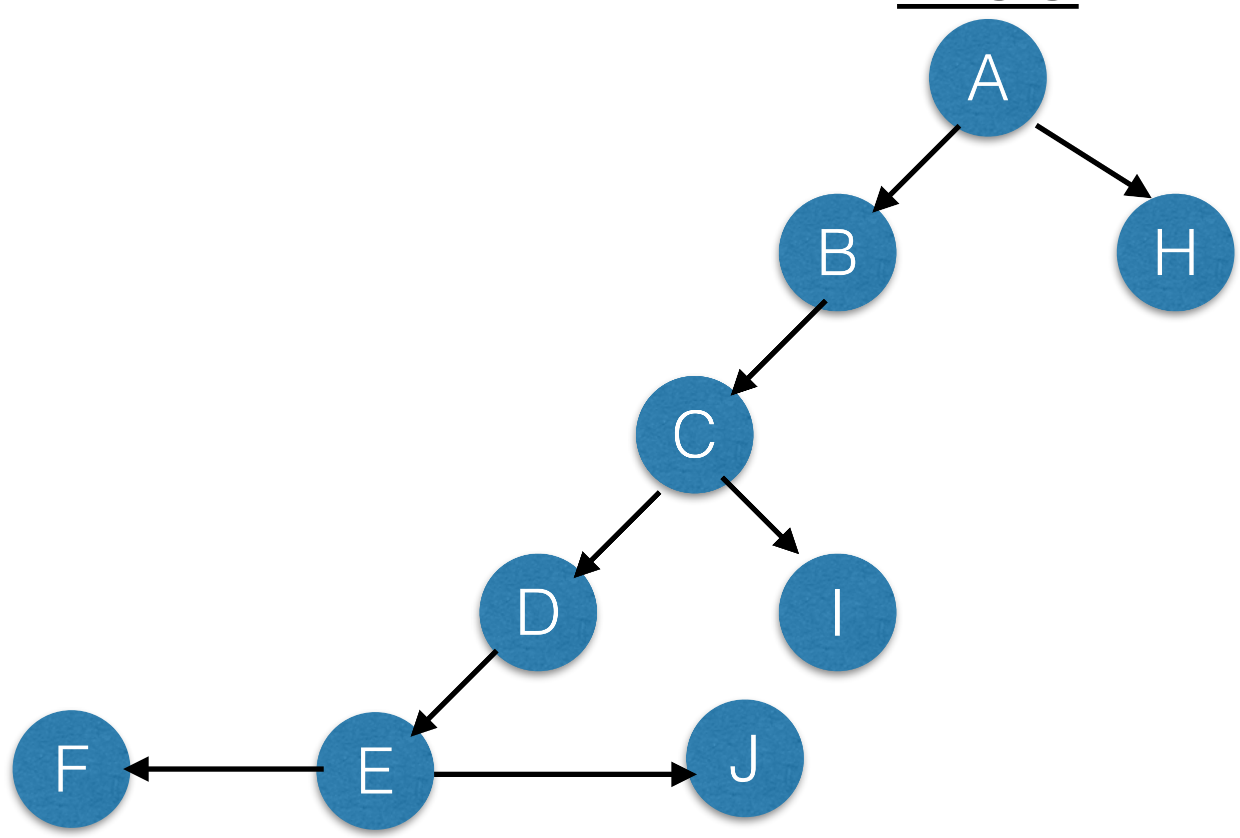
BFS Example



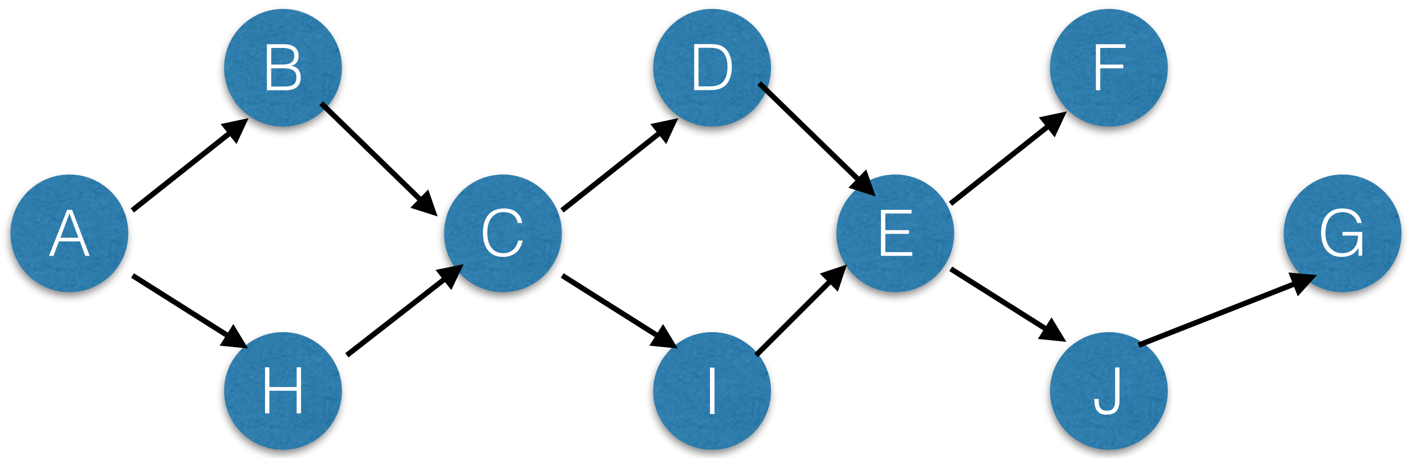
Queue

J

Tree



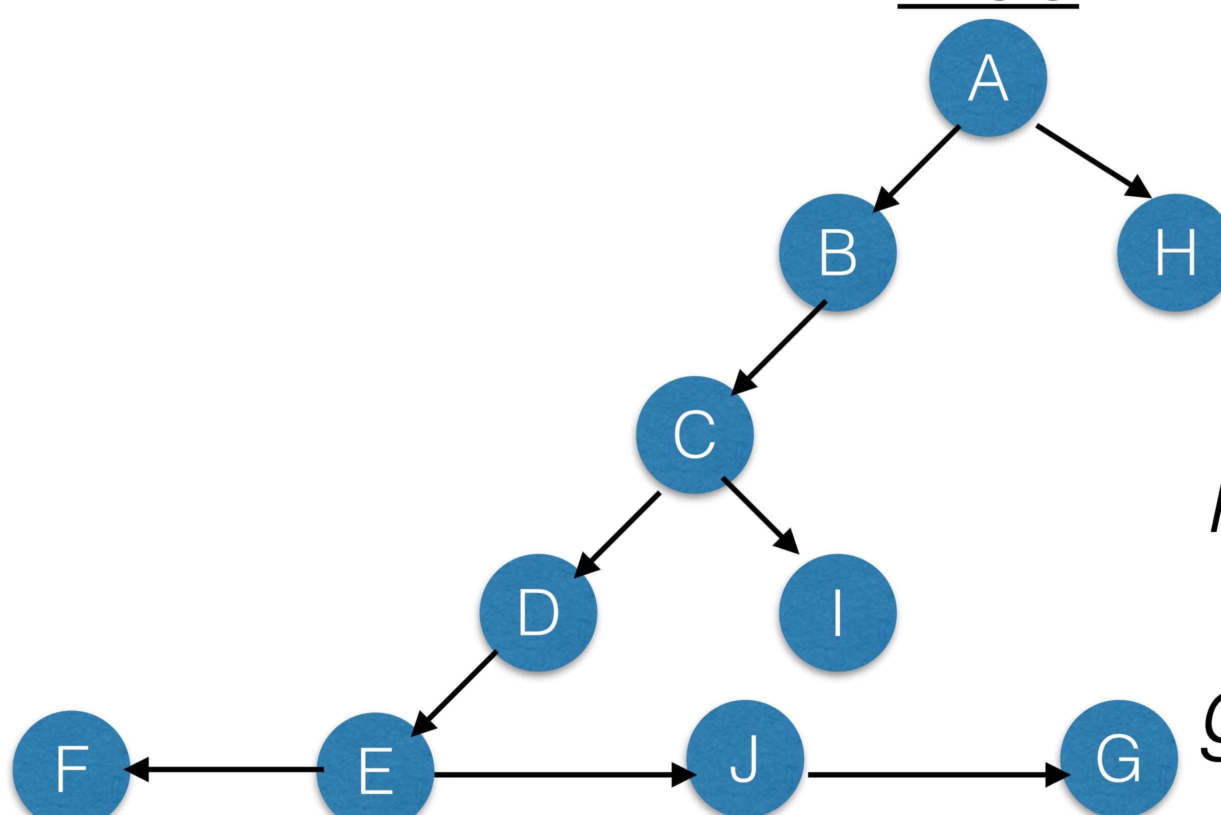
BFS Example



Queue

~empty~

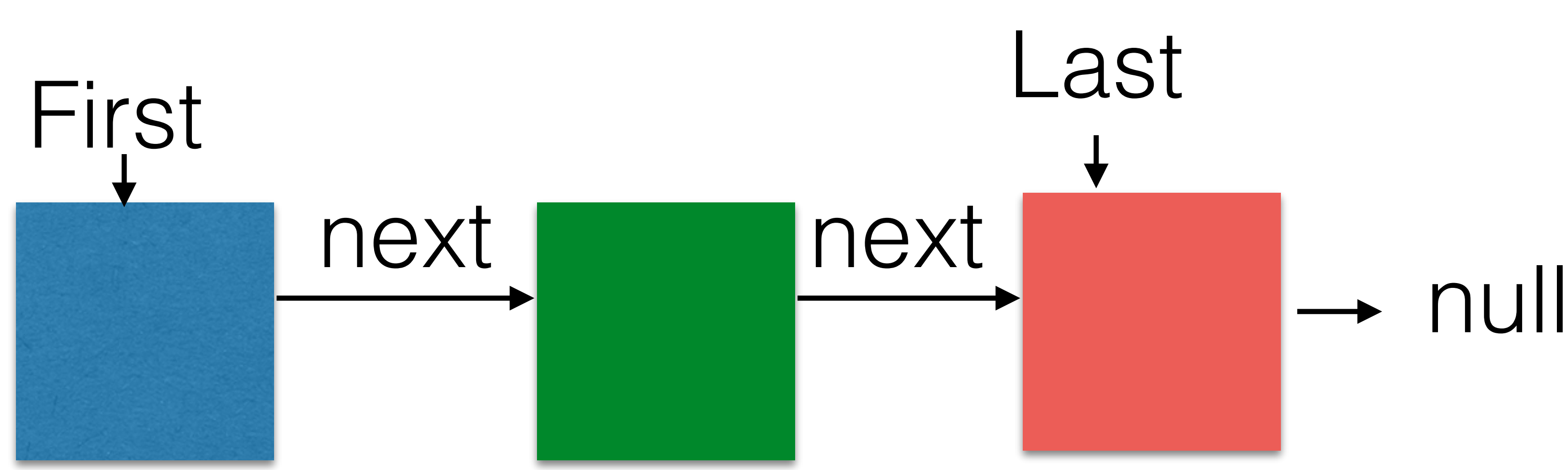
Tree



Follow the parent links from G to get the path

More on Queues

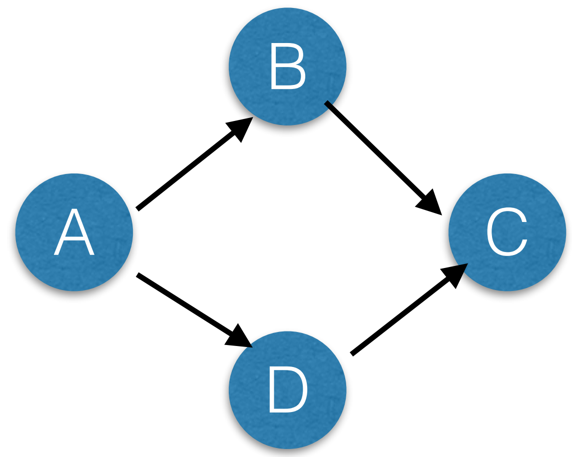
- In an efficiently implemented queue, adding new elements to the back or pulling elements from the front is fast.
- This is possible with 2 additional pointers (or references) in a linked list, “First” and “Last.” Last, in particular, allows direct access to the end.
- In Python, a **deque** data structure offers the quick access to the front and back that a basic list lacks
    - In basic list, only append is fast; dequeue requires shifting elements



Graph Representations

- Two approaches:
  - *Adjacency matrix* : A |V| x |V| array that is 1 at u,v if there is an edge from u to v
  - *Adjacency list*: |V| lists of vertices, where each list contains the vertices a particular vertex has an edge to
    - In Python, can be a dictionary of lists

Examples of graph representations



Adjacency Matrix

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	0	0	0	0
D	0	0	1	0

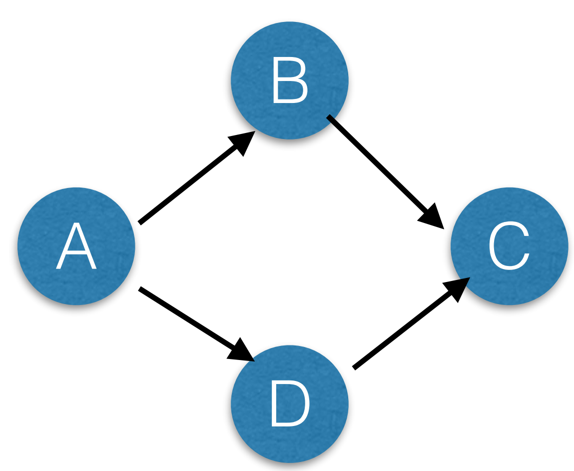
Adjacency Lists

A: [B, D]
B: [C]
C: []
D: [C]

Graph Representation Tradeoffs: Adjacency Matrix

- Pro: Direct lookup of connectivity of two vertices
- Cons:
  - V^2 space when there aren't necessarily that many edges
  - Need to iterate through V entries to check all neighbors of a node

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	0	0	0	0
D	0	0	1	0

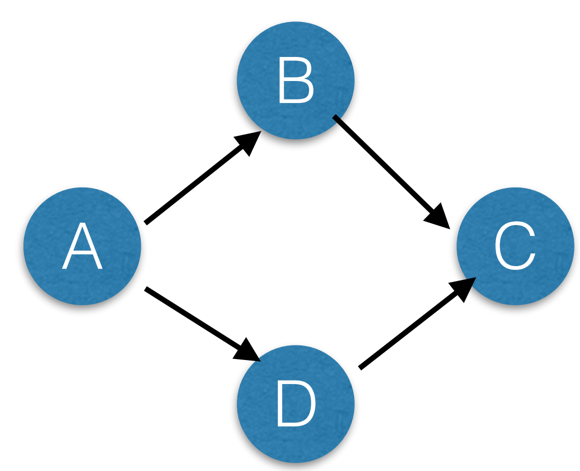


Graph Representation Tradeoffs: Adjacency Lists

- Pros:
    - Checking all edges will not waste time on non-existent edges
    - Takes less memory
- Con: Checking connectivity of a particular pair of vertices requires traversing a list

Adjacency Lists

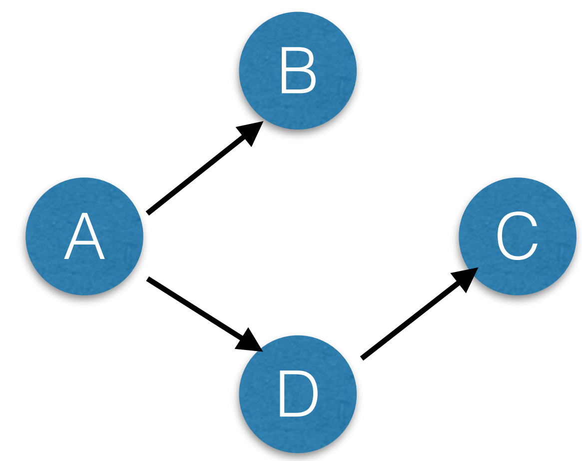
A: [B, D]
B: [C]
C: []
D: [C]



BFS With Adjacency Matrices is Slower

- With adjacency matrices, in adding all neighbors of a node to queue, need to check all entries of the matrix for that node's row

- That could be many more operations than just checking the edges that exist



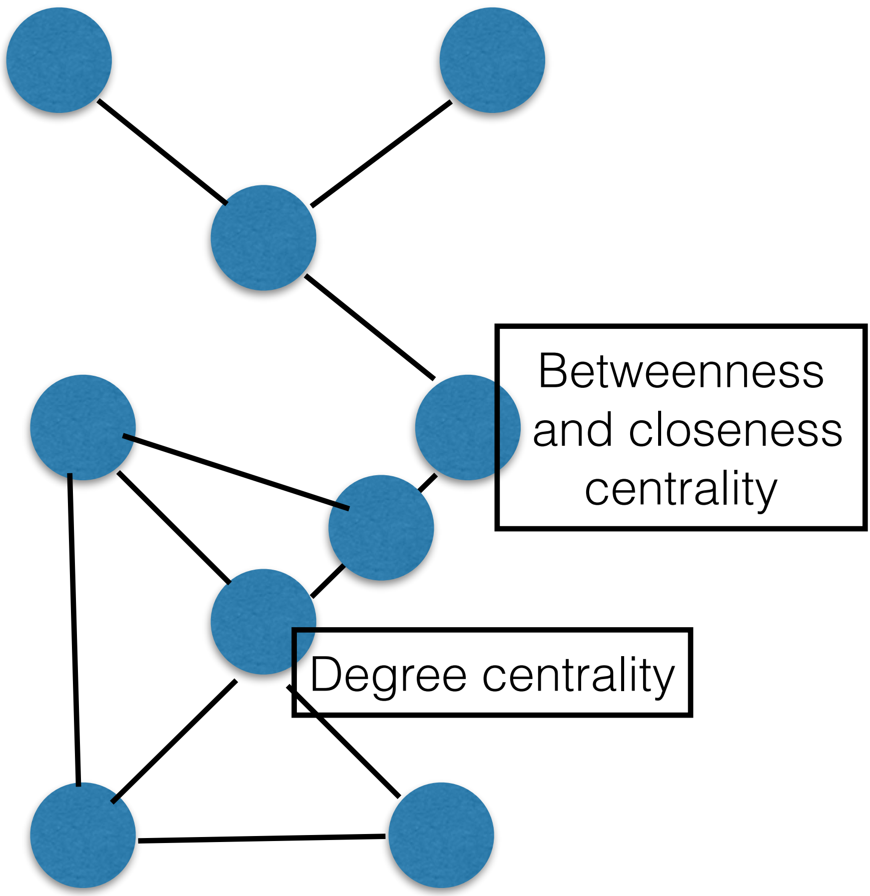
	A	B	C	D
A		1		1
B				
C				
D			1	

Calculating Centrality With BFS

- Closeness centrality requires 1 BFS with no "target" to find shortest path lengths to all nodes

- Betweenness centrality can be estimated by choosing random *s*,*t* pairs and running BFS

- Degree and eigenvector centrality don't need BFS



Summary — Graphs

- Graphs are an extremely versatile way of abstractly reasoning about relationships and connections
- One thing we can do with a social network graph is calculate different kinds of centrality - find the most important people
- Breadth-first search is an efficient tool for finding shortest paths on graphs - which can be used for centrality