Graphs

CSE 332 Spring 2025
Lecture 15

Announcements

	Monday	Tuesday	Wednesday	Thursday	Friday
This Week			MIDTERM :O		Ex 5 due Ex 6 out
Next Week	TODAY Ex 7 out				Ex 6 due Ex 8 out

Extra video coming soon on those two sorts we skipped.

See the extra slides (linked in the spec) for more details on Ex 7.

ADTs so far

We've seen:

Queues and Stacks

- Our data points have some order we're maintaining

Priority Queues

- Our data had some priority we needed to keep track of.

Dictionaries

- Our data points came as (key, value) pairs.

Graphs

Secure | https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

Not logged in Talk Contributions Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk | Read Edit View history | Search Wikipedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

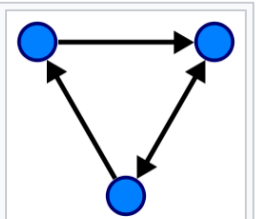
Help
About Wikipedia
Community portal
Recent changes
Contact page

Graph [redacted]

From Wikipedia, the free encyclopedia

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics, specifically the field of graph theory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.



A graph with three vertices and three edges.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Contents [hide]

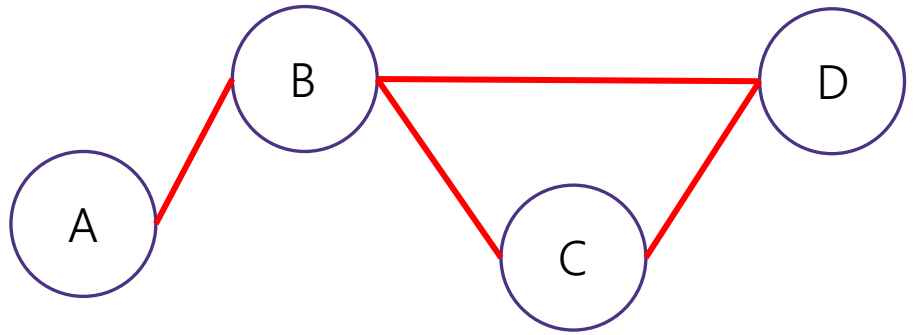
1 Operations

Graphs are too versatile to think of them as only one ADT!

Graphs

Represent data points and the relationships between them.

That's vague.



Formally:

A graph is a pair: G = (V,E)

V: set of **vertices** (aka **nodes**) $\{A, B, C, D\}$

E: set of **edges** $\{(A, B), (B, C), (B, D), (C, D)\}$

- Each edge is a pair of vertices.

Making Graphs

If your problem has **data** and **relationships**, you might want to represent it as a graph

How do you choose a representation?

Usually:

Think about what your "fundamental" objects are

- Those become your vertices.

Then think about how they're related

- Those become your edges.

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

Facebook friendships

Input data for the “6 degrees of Kevin Bacon” game

Course Prerequisites

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

- Vertices: webpages. Edges from a to b if a has a hyperlink to b.

Facebook friendships

- Vertices: people. Edges: if two people are friends

Input data for the “6 Degrees of Kevin Bacon” game

- Vertices: actors. Edges: if two people appeared in the same movie
- Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

- Vertices: courses. Edge: from a to b if a is a prereq for b.

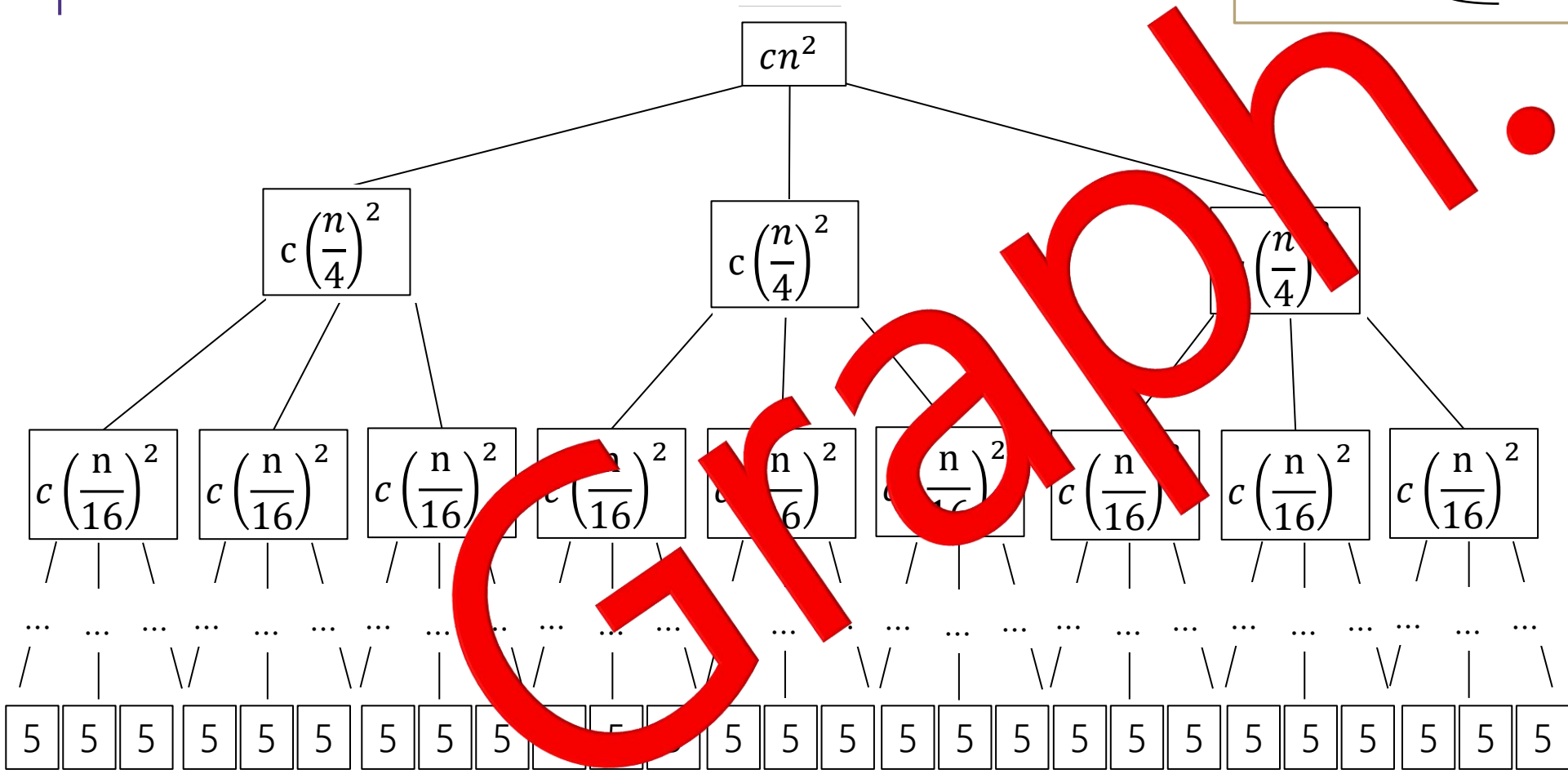
More Graphs

We've already used graphs to represent things in this course:

A LOT

Solving Recurrences III

$$T(n) = \begin{cases} 5 \; when \; n \leq 4 \\ 3T\left(\frac{n}{4}\right) + cn^2 \; otherwise \end{cases}$$



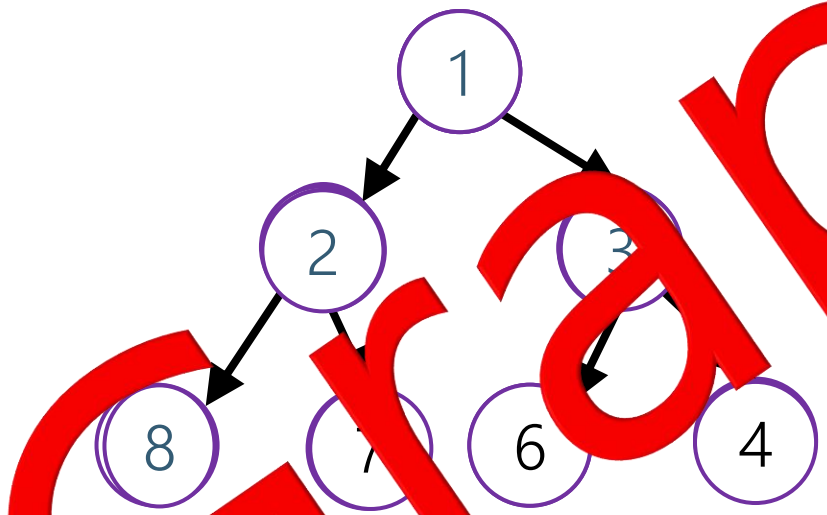
Graph.

Answer the following questions:

1. What is input size on level i?
2. Number of nodes at level i?
3. Work done at recursive level i?
4. Last level of tree?
5. Work done at base case?
6. What is sum over all levels?

BuildHeap: Only One Possibility

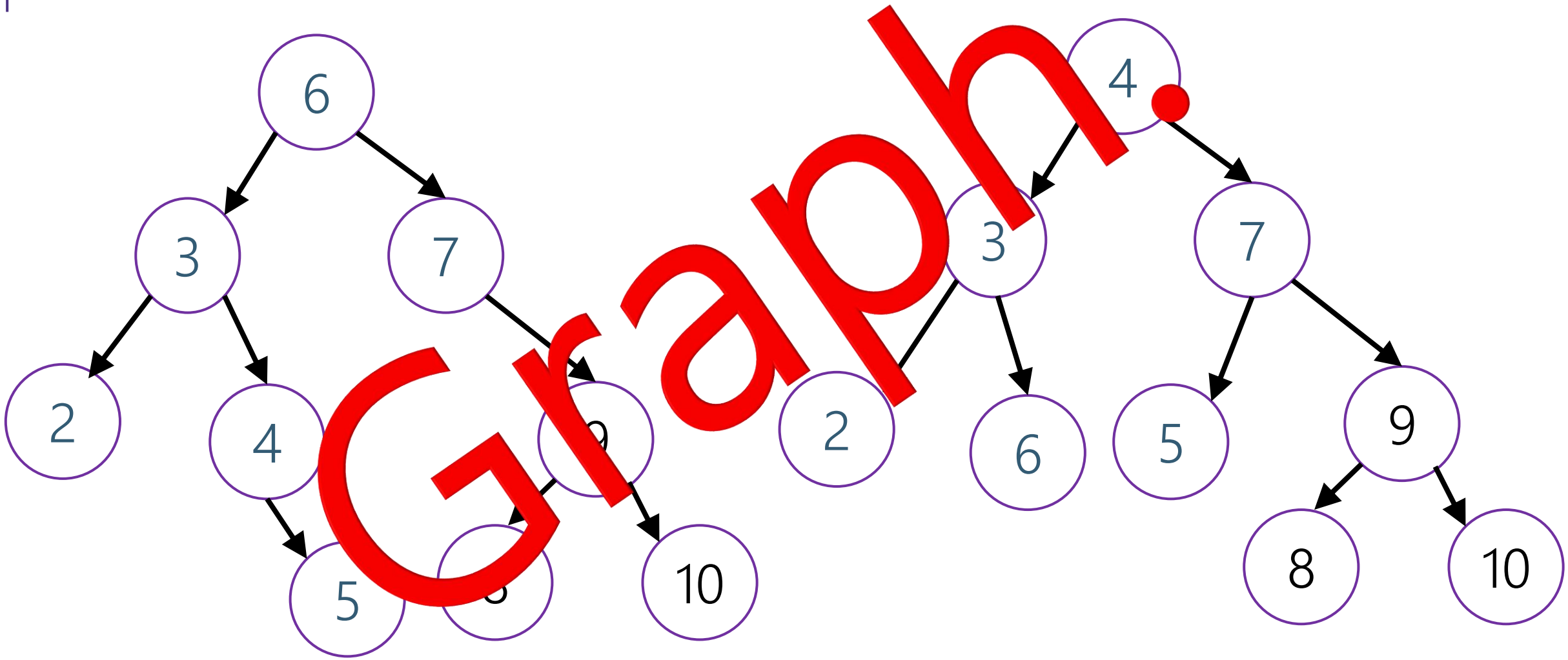
But `StartBottom()` seems to work.

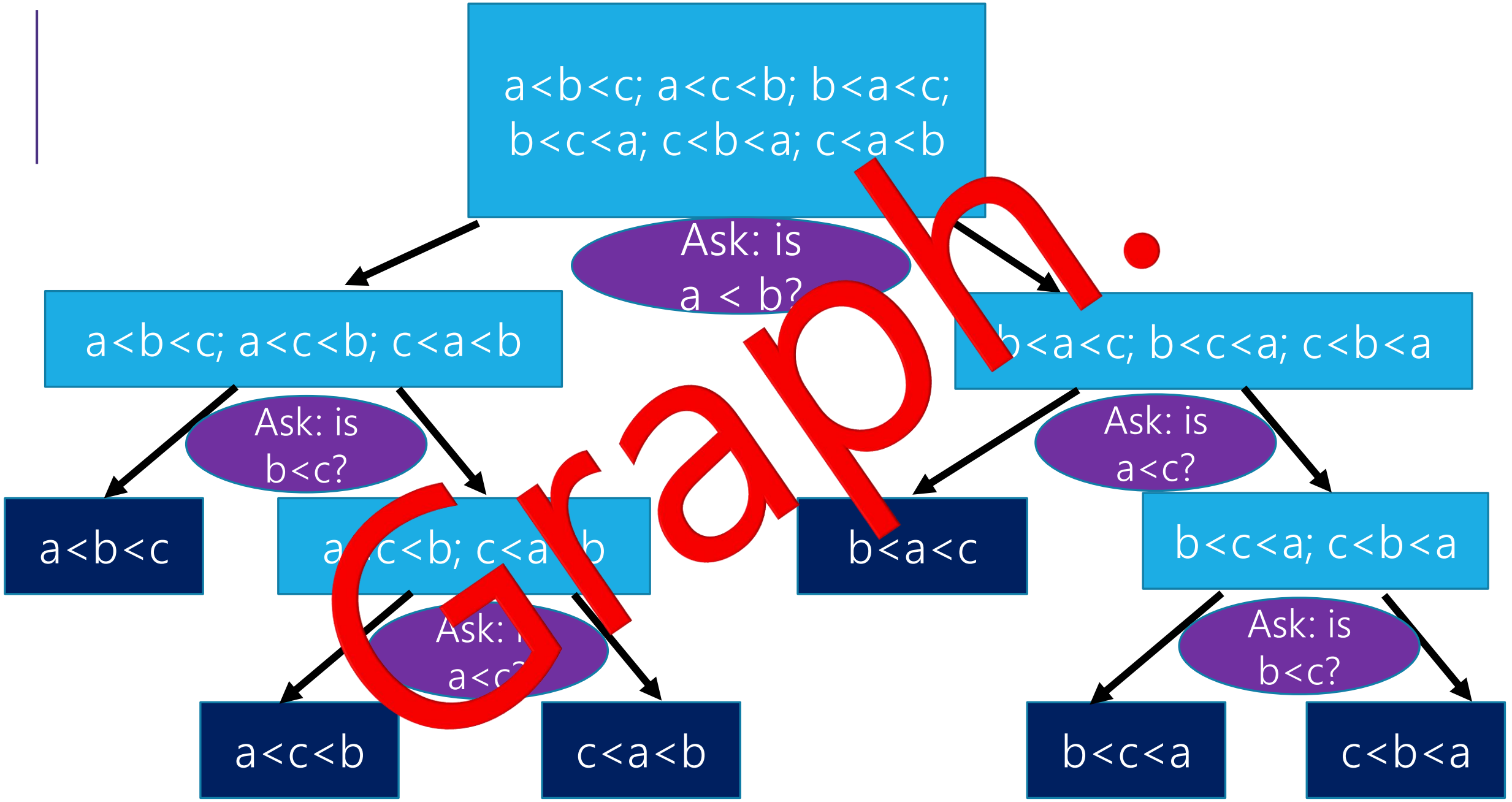


Graph.

Does it always w

Are These AVL Trees?




a<b<c; a<c<b; b<a<c; b<c<a; c<b<a; c<a<b
Ask: is a < b?
a<b<c; a<c<b; c<a<b
b<a<c; b<c<a; c<b<a
Ask: is b<c?
Ask: is a<c?
a<b<c
b<a<c
b<c<a; c<b<a
Ask: is a<c?
Ask: is b<c?
a<c<b
c<a<b
b<c<a
c<b<a
Graph.

More Graphs

EVERYTHING was graphs.

The whole time.

They don't just show up in data structures.

311: NFAs/DFAs and relations
Compilers: Use graphs to figure out valid compilation orders.

Networking: Building a graph
- To the point that some CS people call graphs "networks"

Circuits: represented as graphs

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

Facebook friendships

Input data for the “6 Degrees of Kevin Bacon” game

Course Prerequisites

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

- Vertices: webpages. Edges from a to b if a has a hyperlink to b.

Edges have direction

Facebook friendships

- Vertices: people. Edges: if two people are friends

Edges don't have direction

Input data for the "6 Degrees of Kevin Bacon" game

- Vertices: actors. Edges: if two people appeared in the same movie
- Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

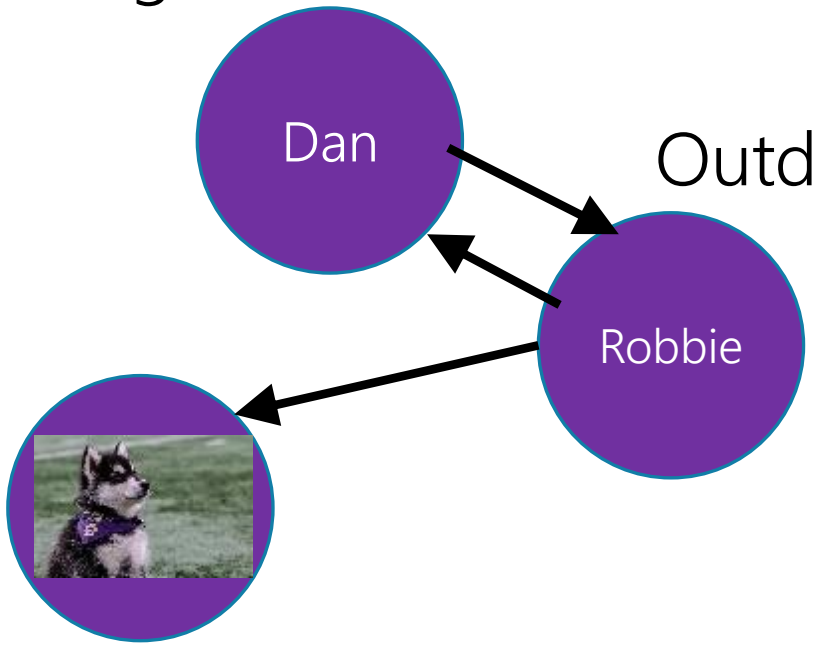
- Vertices: courses. Edge: from a to b if a is a prereq for b.

Edges have direction

Graph Terms

Graphs can be directed or undirected.

Following on twitter.

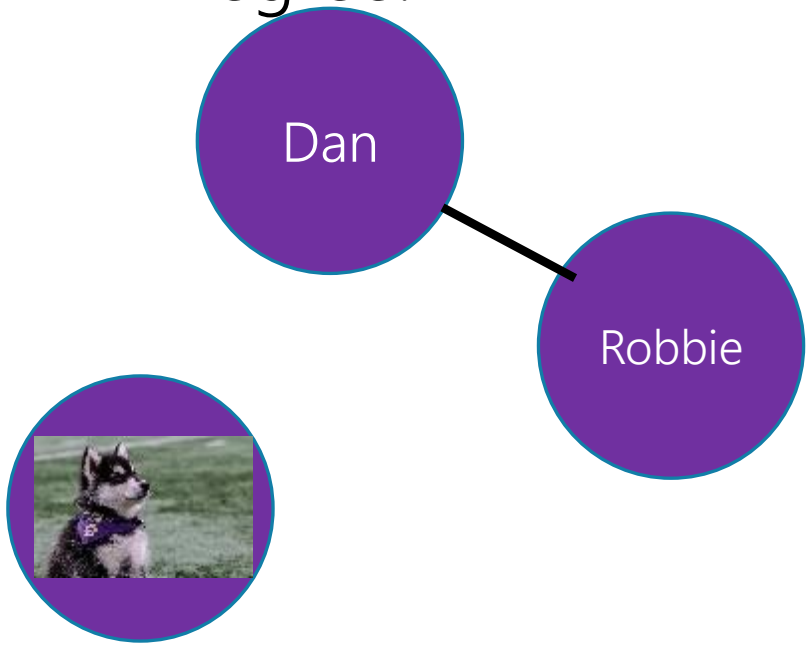


Outdegree: 2

Indegree: 1

This graph is **disconnected**.

Degree: 1

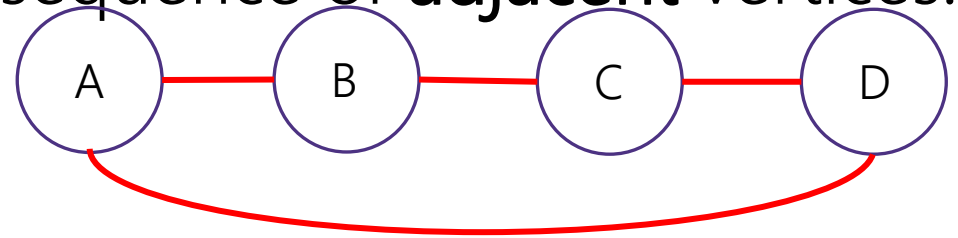


Degree: 0

Friendships on Facebook.

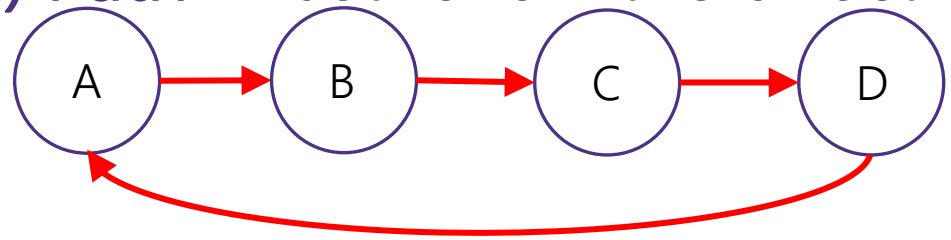
Graph Terms

Path – A sequence of **adjacent** vertices. Each connected to next by an edge.



A,B,C,D is a path.
So is A,B,A

(Directed) Path–must follow the direction of the edges



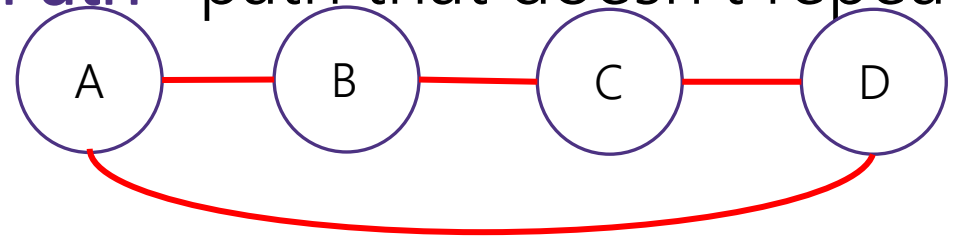
A,B,C,D,A is a directed path.
A,B,A is not.

Length – The number of edges in a path

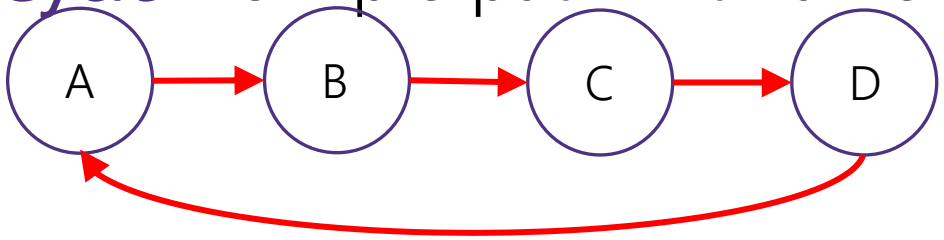
- (A,B,C,D) has length 3.

Graph Terms

Simple Path –path that doesn't repeat a vertex. A,B,C,D is a simple path. A,B,A is not.



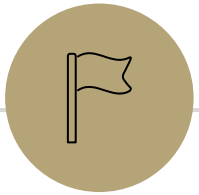
Simple Cycle – simple path with an extra edge from last vertex to first.



Be careful looking at other sources.

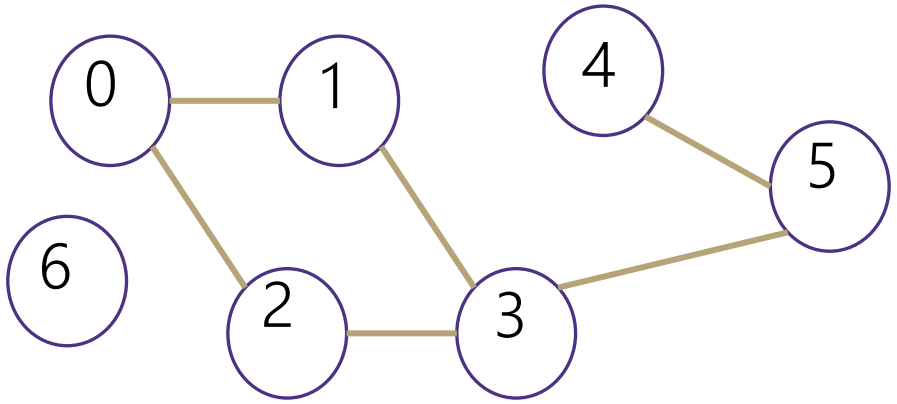
Some people call our "paths" "walks" and our "simple paths" "paths"

Use the definitions on these slides.



Representing and Using Graphs

Adjacency Matrix



In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

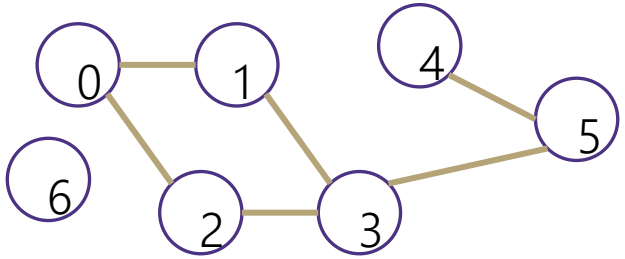
Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get neighbors of u (out): O(n)
- Get neighbors of u (in): O(n)

Space Complexity: $O(n^2)$

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

Adjacency List



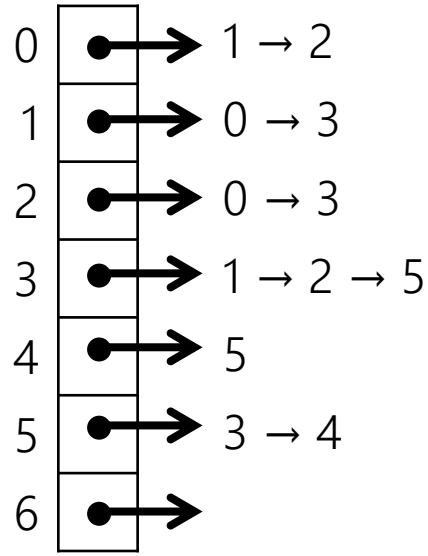
An array where the u'th element contains a list of neighbors of u.

Directed graphs: put the out neighbors (a[u] has v for all (u,v) in E)

Time Complexity (|V| = n, |E| = m):

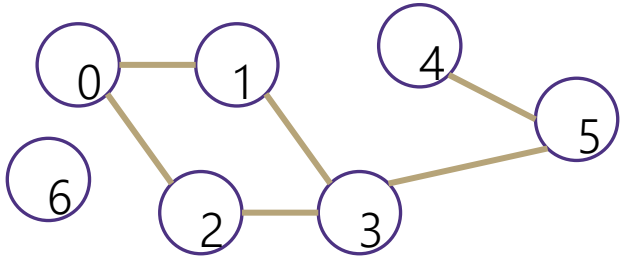
- Add Edge: O(1)
- Remove Edge: O(d)
- Check edge exists from (u,v): O(d)
- Get neighbors of u (out): O(d)
- Get neighbors of u (in): O(n + m)

Space Complexity: O(n + m)



Suppose we use a linked list for each node.

Adjacency List



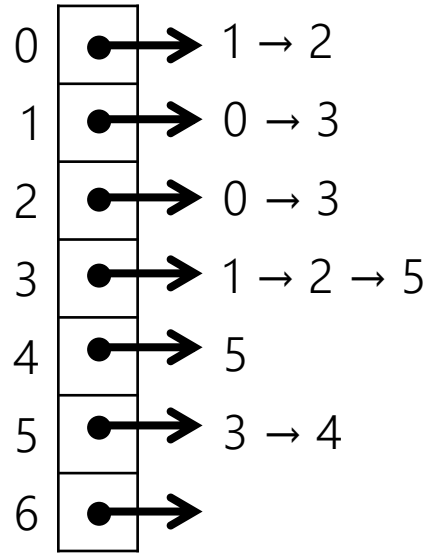
An array where the u'th element contains a list of neighbors of u.

Directed graphs: put the out neighbors (a[u] has v for all (u,v) in E)

Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get neighbors of u (out): O(n)
- Get neighbors of u (in): O(n)

Space Complexity: O(n + m)



Switch the linked lists to hash tables, and do average case analysis.

Which do we use?

If your graph is **dense** (Close to n^2 edges) or the graph is changing a lot, the adjacency matrix is usually the better choice.

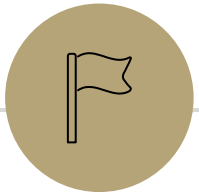
Otherwise, adjacency list is the default choice:

- Memory savings are **huge** for the majority of "real world" graphs
  - Most graphs are "sparse" ($O(n)$ edges), think about a map (vertices are intersections, edges are roads; most vertices touch 4 roads, not all of them).
- The running times look worse, but…
  - If you make the graph class, you can just call the vertices 0,1,2,…n-1, which mitigates bad hashing concerns if you use that approach.
  - Most graph algorithms don't actually use "is edge (a,b) there?" frequently; "iterate over all the edges" is much more common, which linked list does in O(1) time.

Which do we use?

Unless otherwise noted, assume we are using adjacency list.

For the problems in this class, it's usually best to design assuming the operation you need will turn out to be constant time…and then check at the end which data structures will let you do that.



Graph Search

What do we do with graphs

So many things!

- That's why we said graphs are more general than a single ADT---they don't have a standard set of operations.

As a starting point---how could we process the entire graph? Examine every vertex and every edge?

Called a "search" of the graph or a "traversal"

Two algorithms (with different purposes)

BFS

Start somewhere…

For every vertex (in some order)

Do whatever you want on that vertex

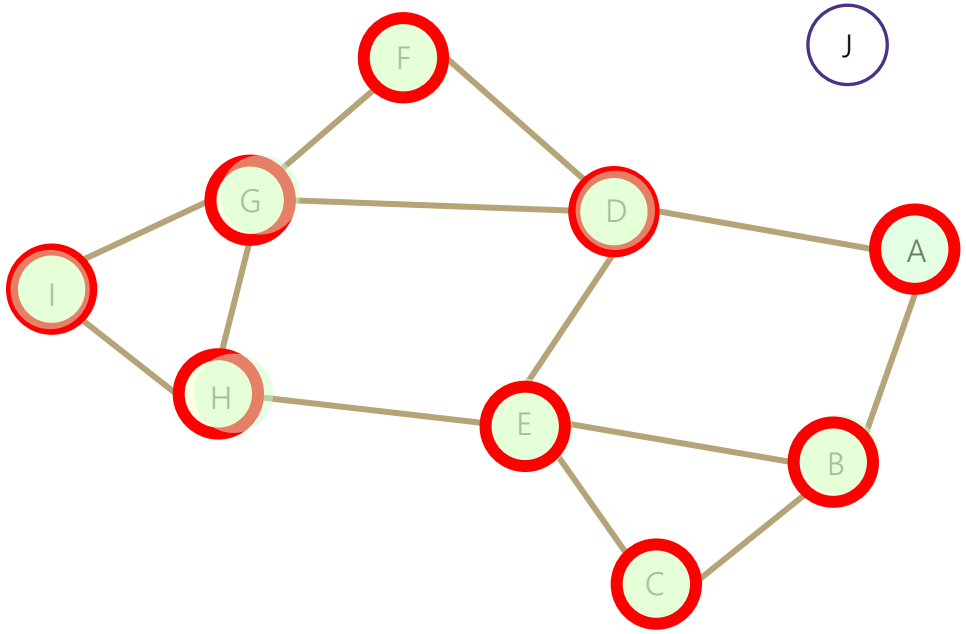
- Sometimes, record some information, store something in there, etc.
- At least, we'll mark it as having been "visited"

You need to process all of its neighbors…store them in some data structure to process them. Then back to the top of the loop.

If you use a Queue for your storage structure, you get BFS.

Breadth First Search

```
search(graph)
    toVisit.enqueue(first vertex)
      mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (V : current.neighbors())
            if (v is not visited)
                toVisit.enqueue(v)
                    mark v as visited
        finished.add(current)
```



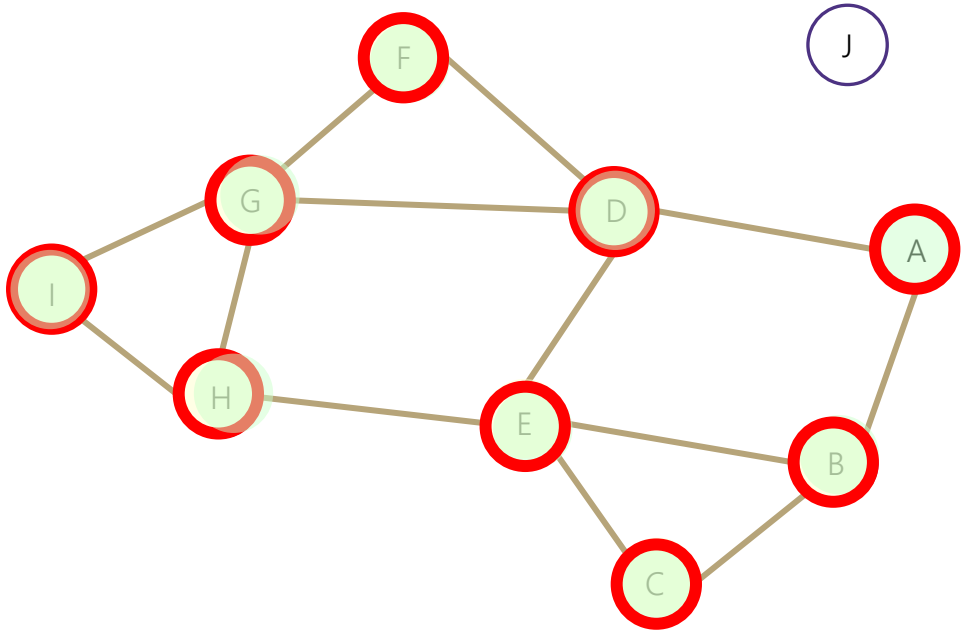
Current node: I

Queue: B D E C F G H I

Finished: A B D E C F G H I

Breadth First Search

```
search(graph)
   toVisit.enqueue(first vertex)
     mark first vertex as visited
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (V : current.neighbors())
         if (v is not visited)
            toVisit.enqueue(v)
                mark v as visited
      finished.add(current)
```



What's the running time of this algorithm?

We visit each vertex at most twice, and each edge at most once: $O(|V| + |E|)$

Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing “frontier” movement across graph

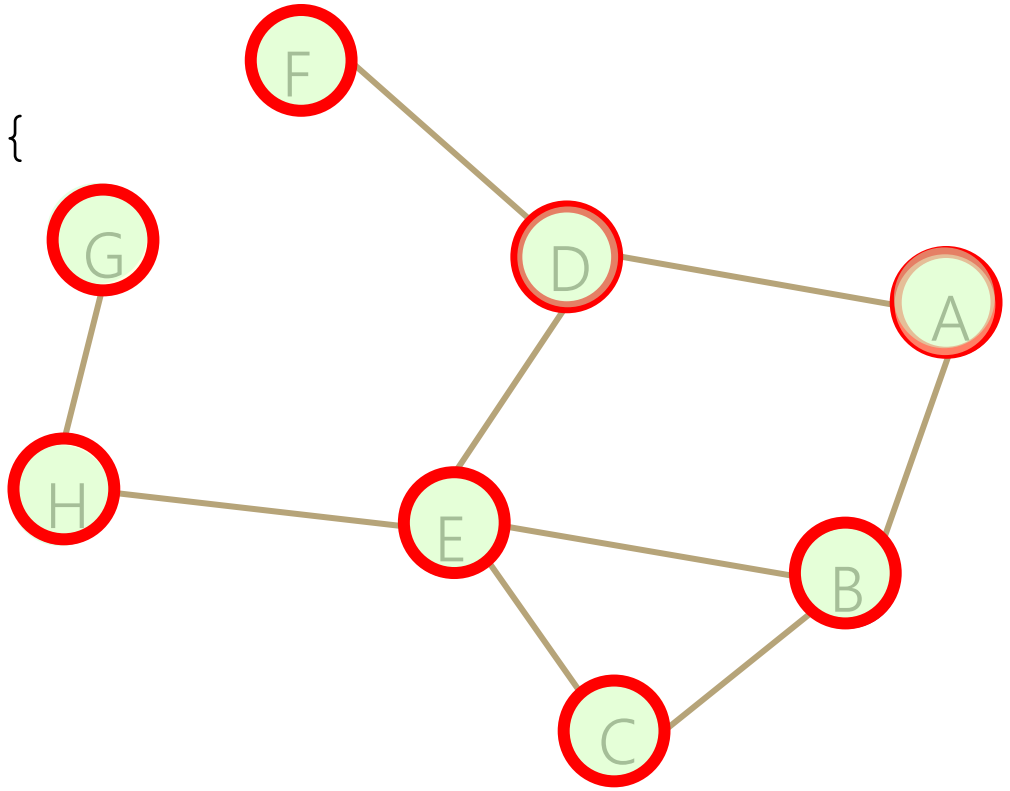
Can you move in a different pattern? What if you used a stack instead?

```
bfs(graph)
   toVisit.enqueue(first vertex)
      mark first vertex as visited
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (V : current.neighbors())
         if (v is not visited)
            toVisit.enqueue(v)
                  mark v as visited
      finished.add(current)
```

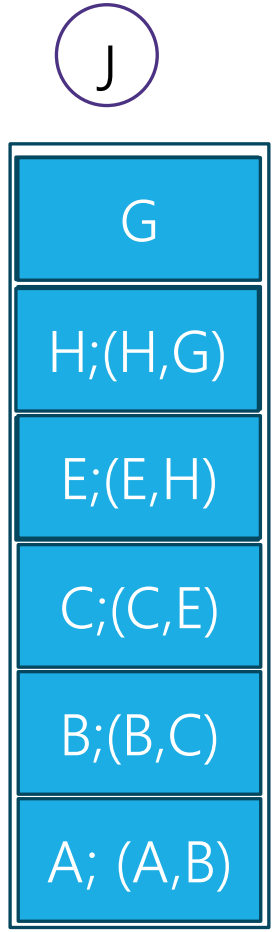
```
dfs(graph, curr)
   mark curr as visited
   for(v : curr.neighbors()){
      if(v is not visited){
         dfs(graph, v)
      }
   }
   mark curr as “done”
```

Depth First Search

```
dfs(graph, curr)
    mark curr as visited
    for(v : curr.neighbors()){
        if(v is not visited){
            dfs(graph, v)
        }
    }
    mark curr as "done"
```



Finished: F D G H E C B A



Stack

DFS

Running time?

- Same as BFS: $O(|V| + |E|)$

You can rewrite DFS to be a recursive method.

Use the call stack as your stack.

No easy trick to do the same with BFS.

Next week: Using BFS, DFS and other algorithms to solve problems!

DFS for applications

Applications for DFS (and BFS) are often:

Run [D/B]FS, and do some extra bookkeeping.

For DFS, it's common to classify based on "start" and "finish" times

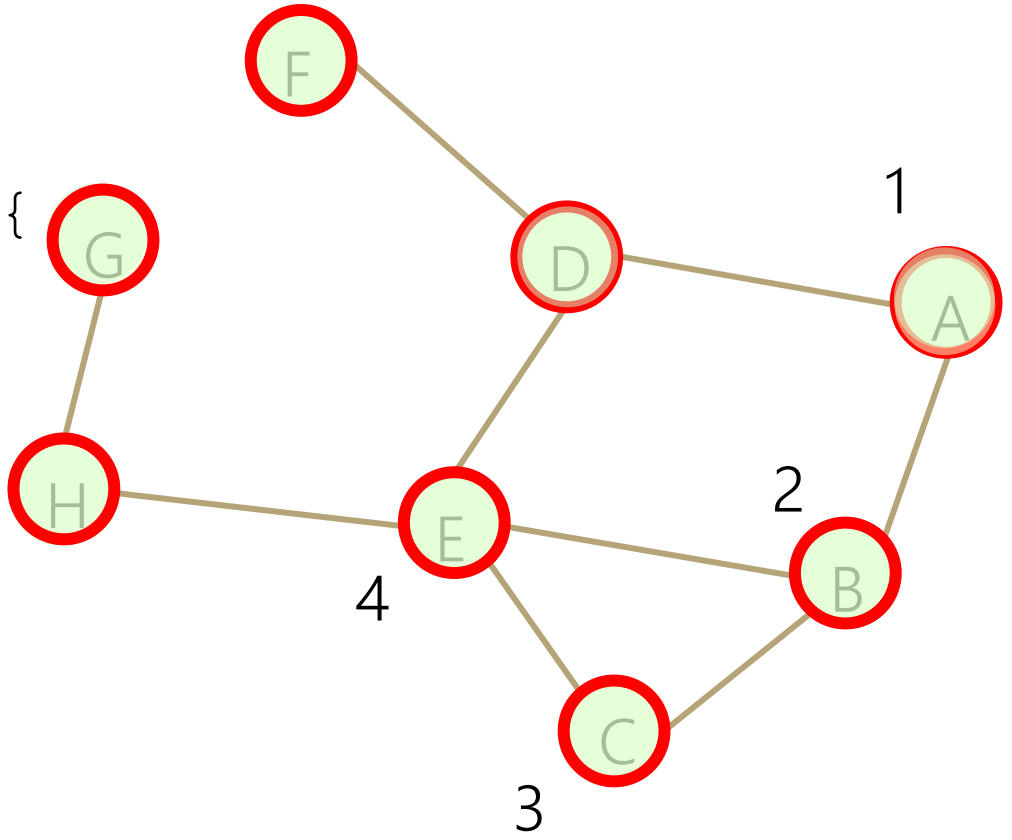
When vertices go on the (call) stack, and when they come off.

Depth First Search

```
dfs(graph, curr)
    mark curr as visited
    record curr.start
    for(v : curr.neighbors()){
        if(v is not visited){
            dfs(graph, v)
        }
    }
    mark curr as "done"
    record curr.end
```

Finished: F D G H E C B A

(A,B), (B,C), (C,E) cause a new vertex to go on the stack.
(E,B) goes **"back"** to an edge that's already on the stack, but not finished.

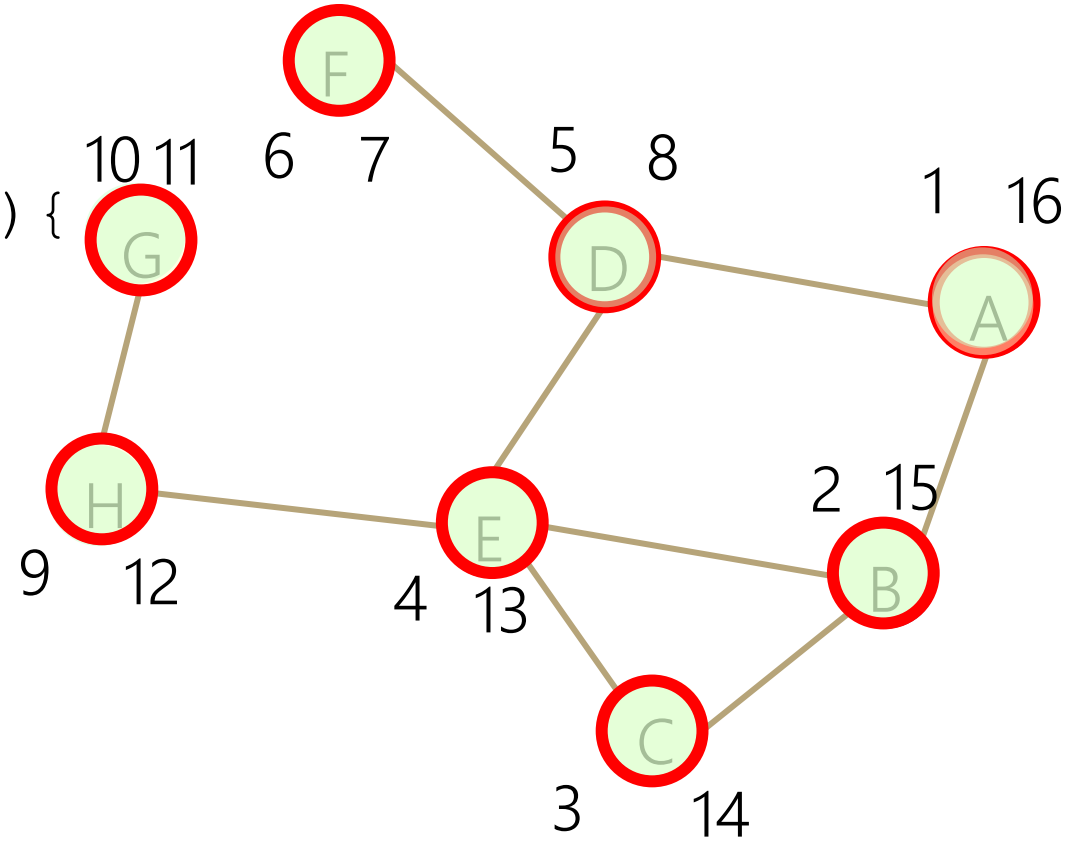


Stack

Depth First Search

```
dfs(graph, curr)
    mark curr as visited
    record curr.start
    for(v : curr.neighbors()){
        if(v is not visited){
            dfs(graph, v)
        }
    }
    mark curr as "done"
    record curr.end
```

(A,B), (B,C), (C,E) cause a new vertex to go on the stack.
(E,B) goes **"back"** to an edge that's already on the stack, but not finished.



Finished: F D G H E C B A

Stack