



Lecture 16: Graphs

CSE 373 Data Structures and Algorithms

Administrivia

Exercise 3 due tonight.

Exercise 4 out today.

Robbie has one-on-one “talk about the midterm/your grade” office hours today (2:30-4 in CSE 330)

Sorting Summary

You have a bunch of data. How do you sort it?

Honestly...use your language's default implementation

- It's been carefully optimized.

In practice, quicksort usually has the best constant factors among those we've discussed. (But remember it's worst case is $\Theta(n^2)$.)

Choose a specific algorithm If you're situation is special

- Not a lot of extra memory? Use an in-place sort.
- Want to sort repeatedly to break ties? Use a stable sort.
- Know your data is integers in a small range? Maybe radix sort.



Graphs

ADTs so far

Queues and Stacks

- We want to process our data in some order (based on when they were inserted)

Lists

- We want to maintain an order, but add or remove from anywhere

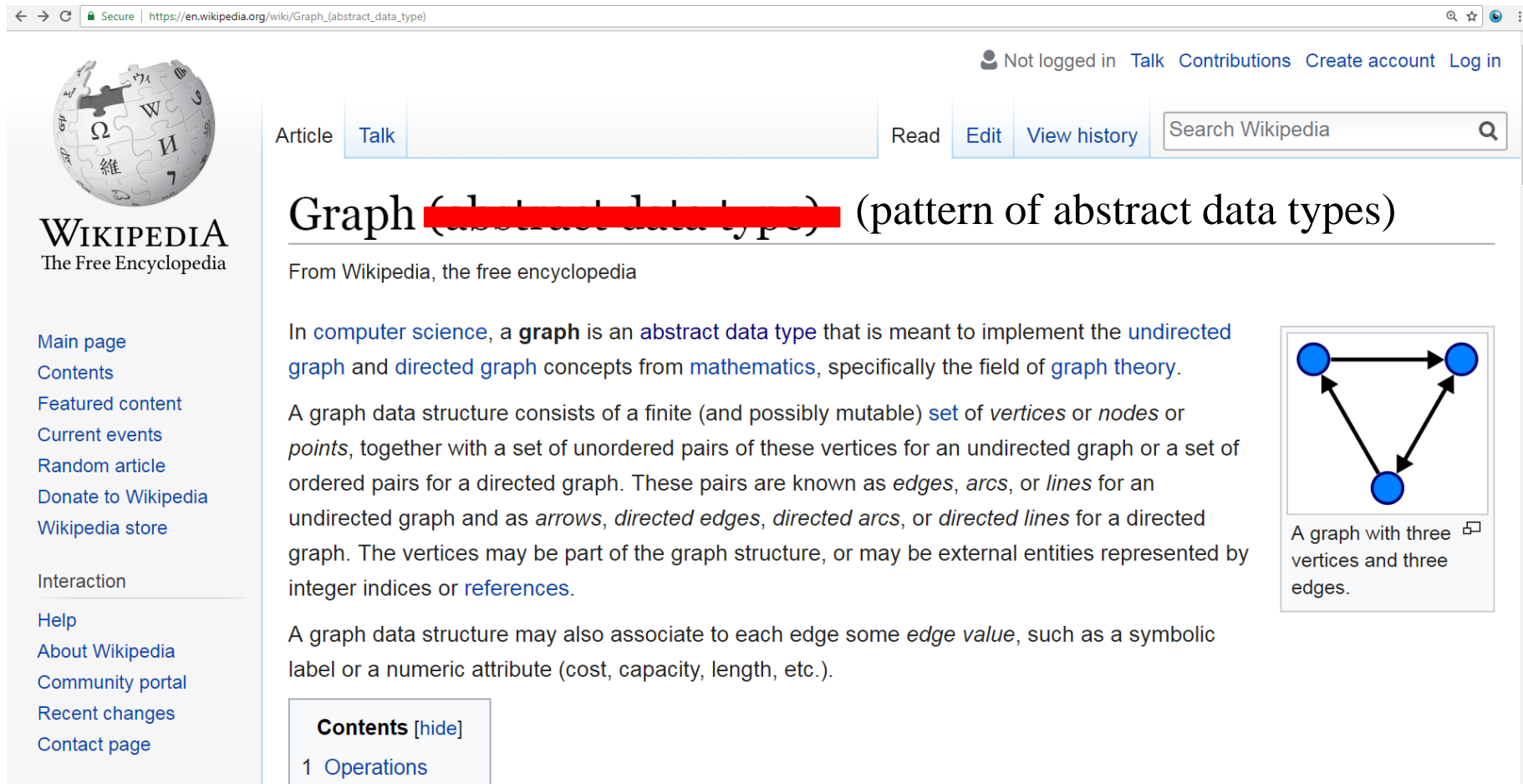
Priority Queues

- Our data had some priority we needed to keep track of, and wanted to process in order of importance.

Dictionaries

- Our data points came as (key, value) pairs.
- Quickly find the value for a key

Graphs



The screenshot shows the Wikipedia page for "Graph (abstract data type)". The page title is "Graph (abstract data type) (pattern of abstract data types)". The page content includes a definition of a graph in computer science, a diagram of a directed graph with three vertices and three edges, and a table of contents with one entry: "1 Operations".

WIKIPEDIA
The Free Encyclopedia

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

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Article [Talk](#)

Read [Edit](#) [View history](#)

Search Wikipedia

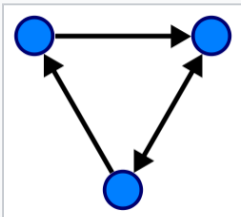
Graph (abstract data type) (pattern of abstract data types)

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



A graph with three vertices and three edges.

Contents [\[hide\]](#)

- [Operations](#)

We'll list Graphs as one of our ADTs...

But don't let that limit your thinking. They are more versatile than any ADT we've seen before.

Graphs

Graphs are versatile enough, I'm not going to show you one of those gold boxes with states and behaviors

The state/behaviors to track change with every new problem we try to solve.

Graphs

Everything is graphs.

Most things we've studied this quarter can be represented by graphs.

- BSTs are graphs
- Linked lists? Graphs.
- Heaps? Also can be represented as graphs.
- Those trees we drew in the tree method? Graphs.

But it's not just data structures that we've discussed...

- Google Maps database? Graph.
- Facebook? They have a "graph search" team. Because it's a graph
- Gitlab's history of a repository? Graph.
- Those pictures of prerequisites in your program? Graphs.
- Family tree? That's a graph

Graphs

Represent data points and the relationships between pairs of them.
That's vague.

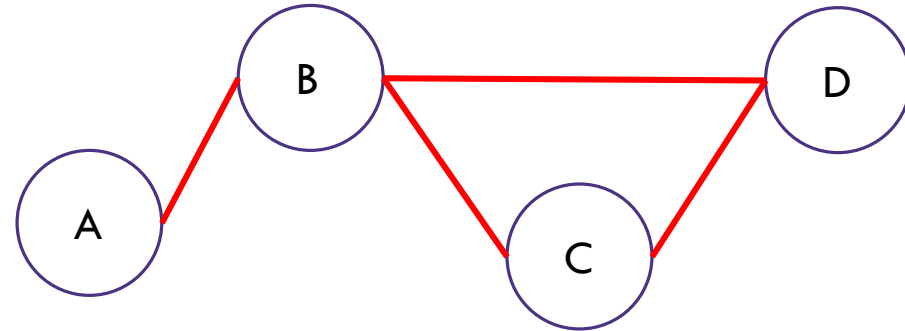
Formally:

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

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

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

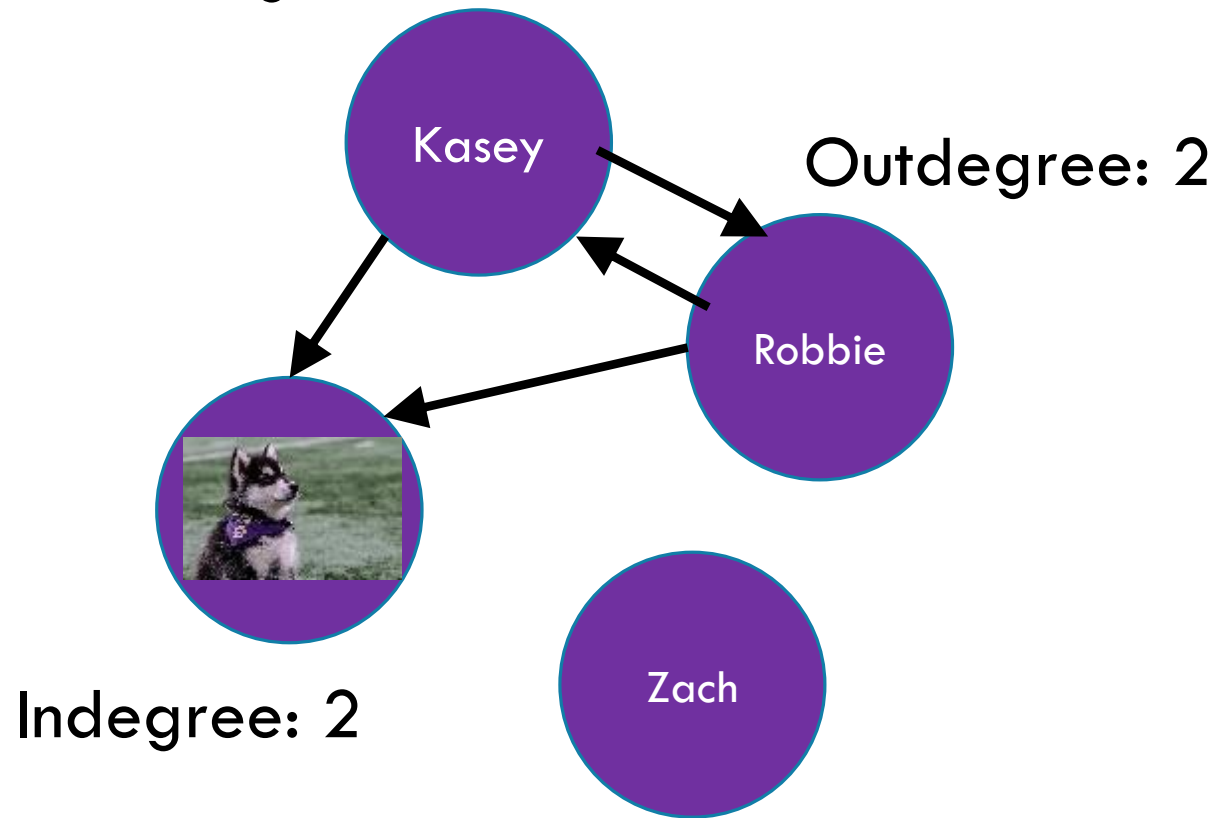
- Each edge is a pair of vertices.



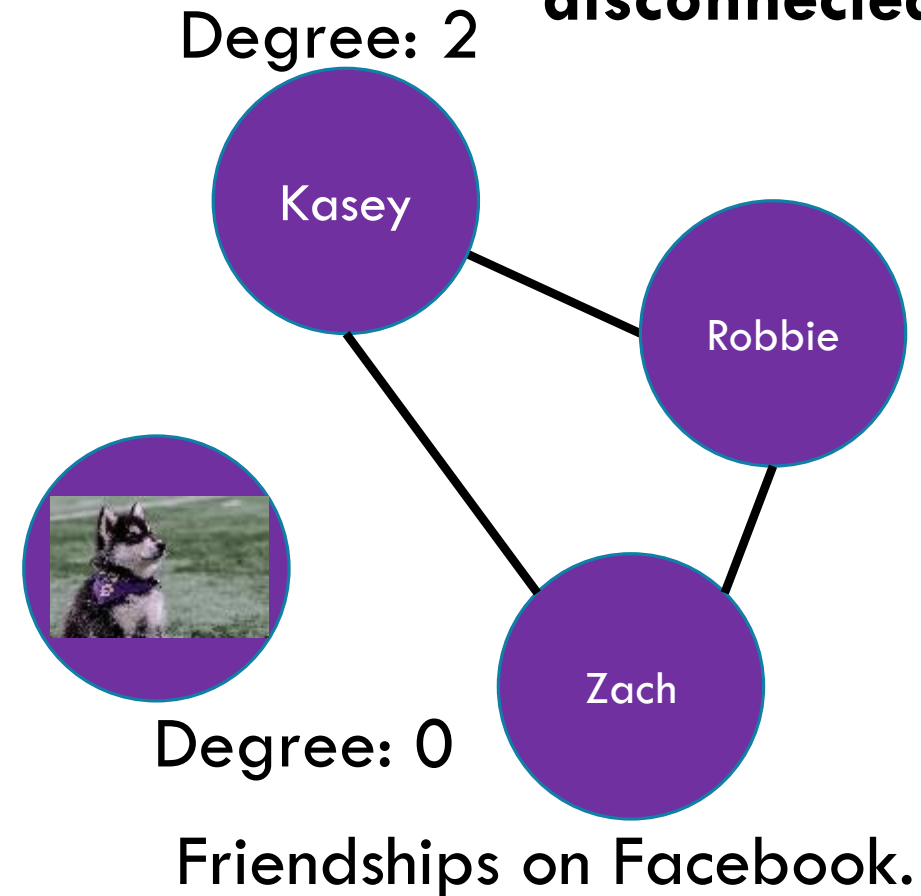
Graph Terms

Graphs can be directed or undirected.

Following on twitter.



This graph is **disconnected**.



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

Family tree

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

Course Prerequisites

pollEV.com/cse373su19

Which of these did you talk most about with your neighbors?

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.

Family tree

- Vertices: people. Edges: from parent to child, maybe for marriages too?

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

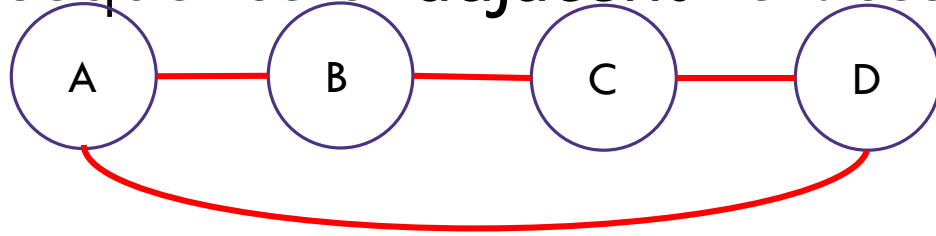
- Vertices: actors. Edges: if two people appeared in the same movie
- Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

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

Graph Terms

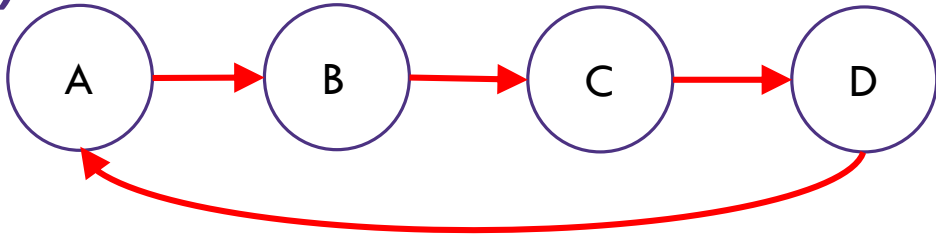
Walk – A sequence of **adjacent** vertices. Each connected to next by an edge.



A, B, C, D is a walk.

So is A, B, A

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



A, B, C, D, B is a directed walk.

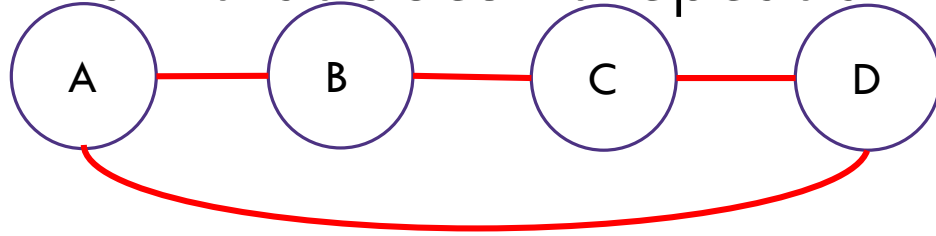
A, B, A is not.

Length – The number of edges in a walk

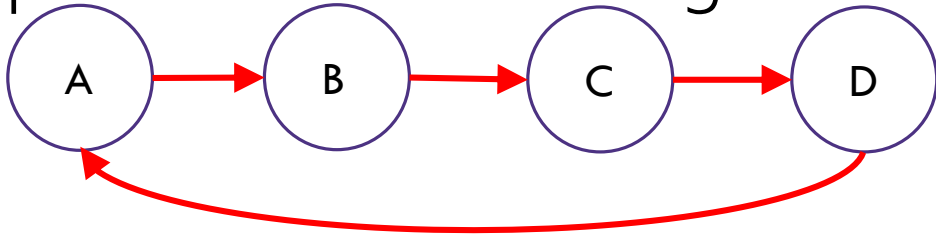
- (A, B, C, D) has length 3.

Graph Terms

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



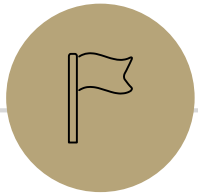
Cycle – path with an extra edge from last vertex back to first.



Be careful looking at other sources.

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

Worst-case Time Complexity

($|V| = n$, $|E| = m$):

Add Edge: $\Theta(1)$

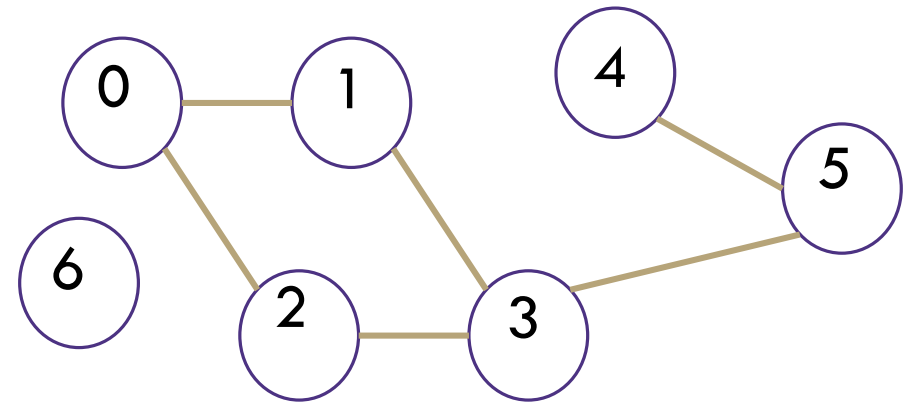
Remove Edge: $\Theta(1)$

Check edge exists from (u,v) : $\Theta(1)$

Get outneighbors of u : $\Theta(n)$

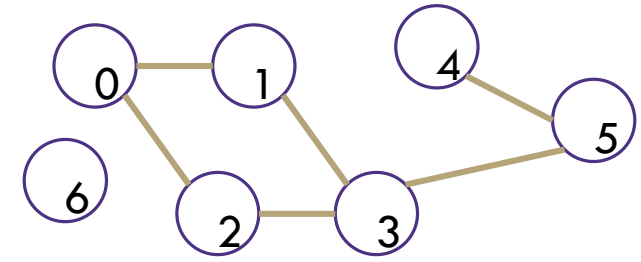
Get inneighbors of u : $\Theta(n)$

Space Complexity: $\Theta(n^2)$



	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: list of out-neighbors ($a[u]$ has v for all (u,v) in E)

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

Add Edge: $\Theta(1)$

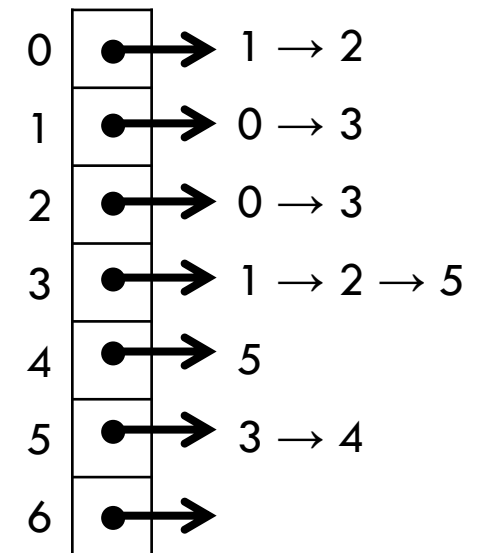
Remove Edge (u,v) : $\Theta(\deg(u))$

Check edge exists from (u,v) : $\Theta(\deg(u))$

Get neighbors of u (out): $\Theta(\deg(u))$

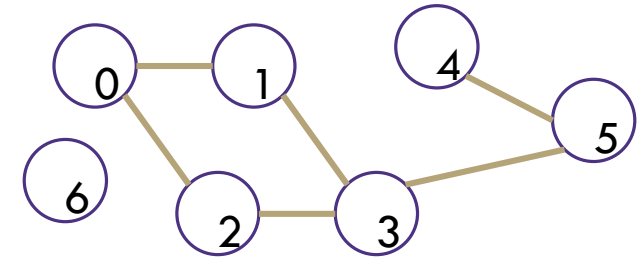
Get neighbors of u (in): $\Theta(n + m)$

Space Complexity: $\Theta(n + m)$



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: list of out-neighbors ($a[u]$ has v for all (u,v) in E)

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

Add Edge: $\Theta(1)$

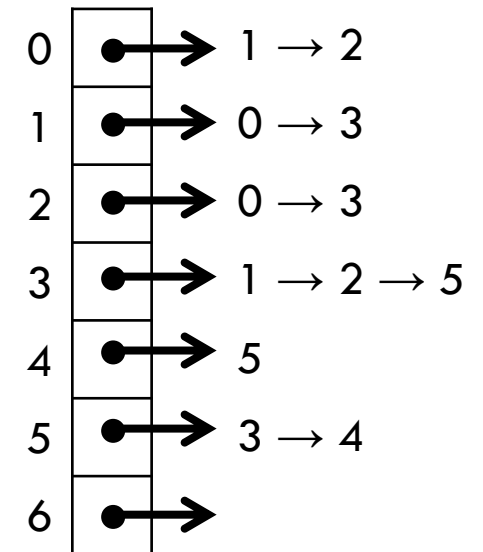
Remove Edge: $\Theta(1)$

Check edge exists from (u,v) : $\Theta(1)$

Get neighbors of u (out): $\Theta(\deg(u))$

Get neighbors of u (in): $\Theta(n)$

Space Complexity: $\Theta(n + m)$



Switch the linked lists to hash tables, and do in-practice analysis.

Tradeoffs

Adjacency Matrices take more space, and have slower $\Theta()$ bounds, why would you use them?

- For **dense** graphs (where m is close to n^2), the running times will be close
- And the constant factors can be much better for matrices than for lists.
- Sometimes the matrix itself is useful ("spectral graph theory")

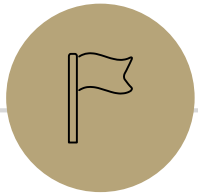
What's the tradeoff between using linked lists and hash tables for the list of neighbors?

- A hash table still *might* hit a worst-case
- And the linked list might not
 - Graph algorithms often just need to iterate over all the neighbors, so you might get a better guarantee with the linked list.

For this class, unless we say otherwise, we'll assume the hash tables operations **on graphs** are all $O(1)$.

- Because you can probably control the keys.

Unless we say otherwise, assume we're using an adjacency list with hash tables for each list.



Traversals

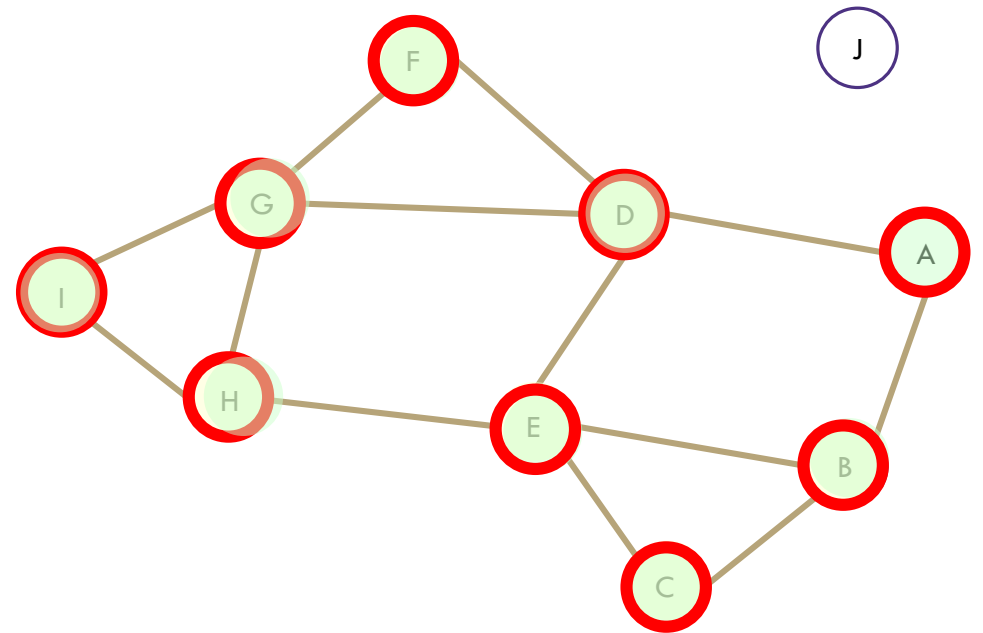
Breadth First Search

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

Current node: I ,

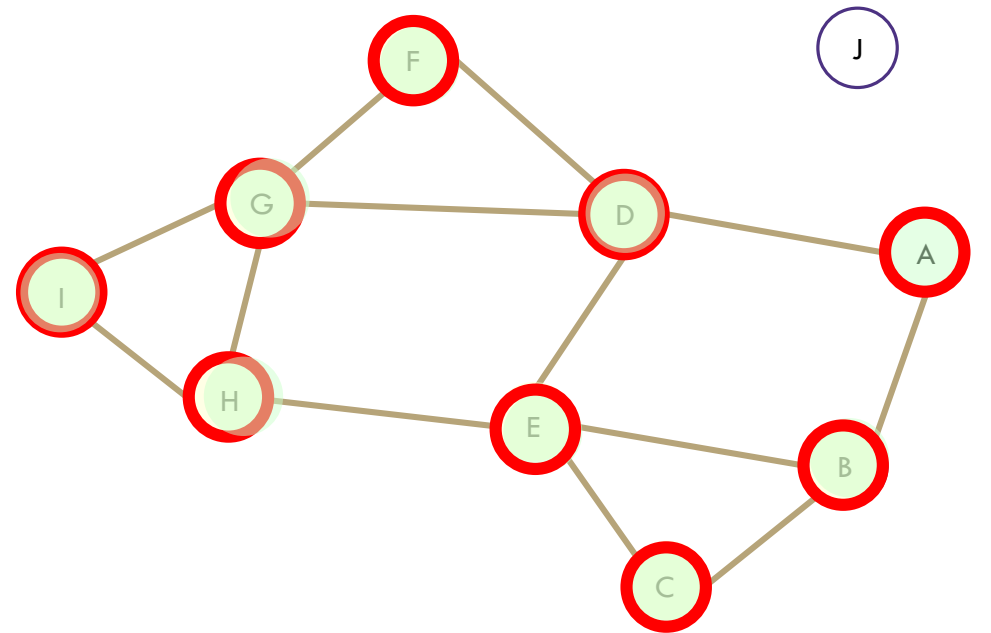
Queue: B D E C F G H I

Finished: A B D E C F G H I



Breadth First Search

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



Hey we missed something...

We're only going to find vertices we can "reach" from our starting point.

If you need to visit everything, just start BFS again somewhere you haven't visited until you've found everything.

Running Time

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

This code might look like:
a loop that goes around m times
Inside a loop that goes around n times,
So you might say $O(mn)$.

That bound is not tight,
Don't think about the loops, think about
what happens overall.
How many times is `current` changed?
How many times does an edge get used
to define `current.neighbors`?

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

Implementation

What does “mark as seen” mean?

It’s up to you!

My lectures are going to assume that each vertex/edge is an object, and that we’ve added whatever field we want into it.

Alternatively, you can make “seen” a dictionary (keys are vertices, values are the variable we want to set)

- Again, you control the keys, so you should be able to get $\Theta(1)$ time without too much trouble.

Regardless, you can do those operations in $\Theta(1)$ time.

In general our graph code will be a little more “abstract”

I won’t try to precisely say what fields each object has (it’s more trouble than it’s worth)

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)
```

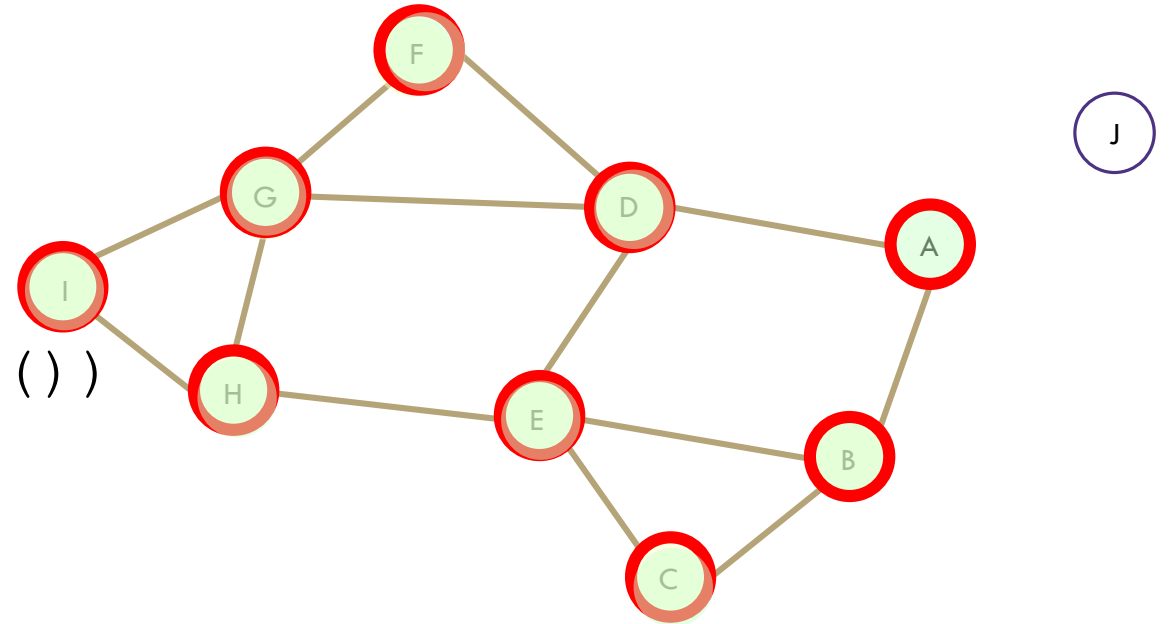
```
  toVisit.enqueue(first vertex)
  mark first vertex as seen
  while (toVisit is not empty)
    current = toVisit.dequeue()
    for (v : current.neighbors())
      if (v is not seen)
        mark v as seen
        toVisit.enqueue(v)
```

```
dfs (graph)
```

```
  toVisit.push(first vertex)
  mark first vertex as seen
  while (toVisit is not empty)
    current = toVisit.pop()
    for (v : current.neighbors())
      if (v is not seen)
        mark v as seen
        toVisit.push(v)
```

Depth First Search

```
dfs(graph)
  toVisit.push(first vertex)
  mark first vertex as seen
  while(toVisit is not empty)
    current = toVisit.pop()
    for (v : current.neighbors())
      if (v is not seen)
        mark v as seen
        toVisit.push(v)
```



Current node: **D**

Stack: **D E H G**

Finished: **A B E H G F I C D**

DFS

Worst-case running time?

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

Indicently, 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.

BFS vs. DFS

So why have two different traversals?

For the same reason we had pre/post/in –order traversals for trees!

BFS and DFS will find vertices in a different order, so they can let you calculate different things.

We'll see an application of BFS next week.

And an application of DFS the week after.