Lecture 13: Intro to Graphs

CSE 373: Data Structures and Algorithms

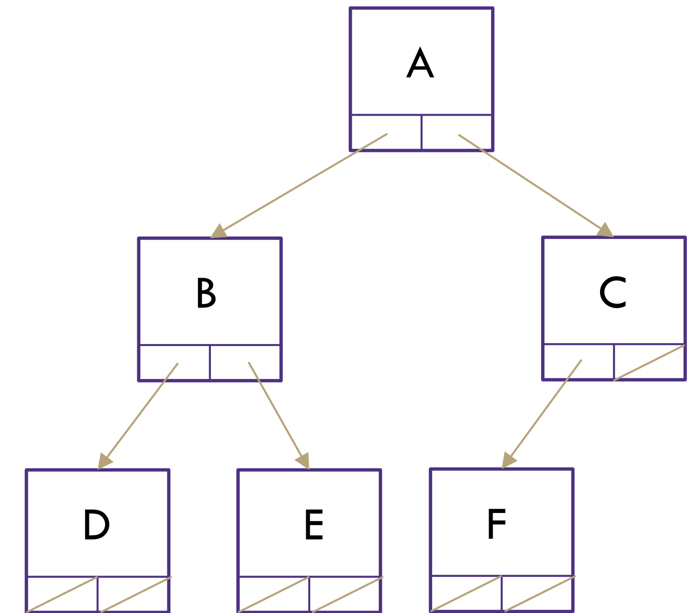
Announcements

- p2 due tonight – office hours all today. Fill out the optional P2 feedback quiz (on Canvas) so we can improve this for future quarters!
- p3 released (pull from skeleton repo as usual) and website instructions are up, due next Wed 11:59pm
- exercise 3 out Friday, due next Friday

minHeap runtimes

Min Priority Queue ADT
state Set of comparable values - Ordered based on "priority" **behavior** **add(value)** – add a new element to the collection **removeMin()** – returns the element with the smallest priority, removes it from the collection **peekMin()** – find, but do not remove the element with the smallest priority

removeMin():
- remove root node
- Find last node in tree and swap to top level
- Percolate down to fix heap invariant

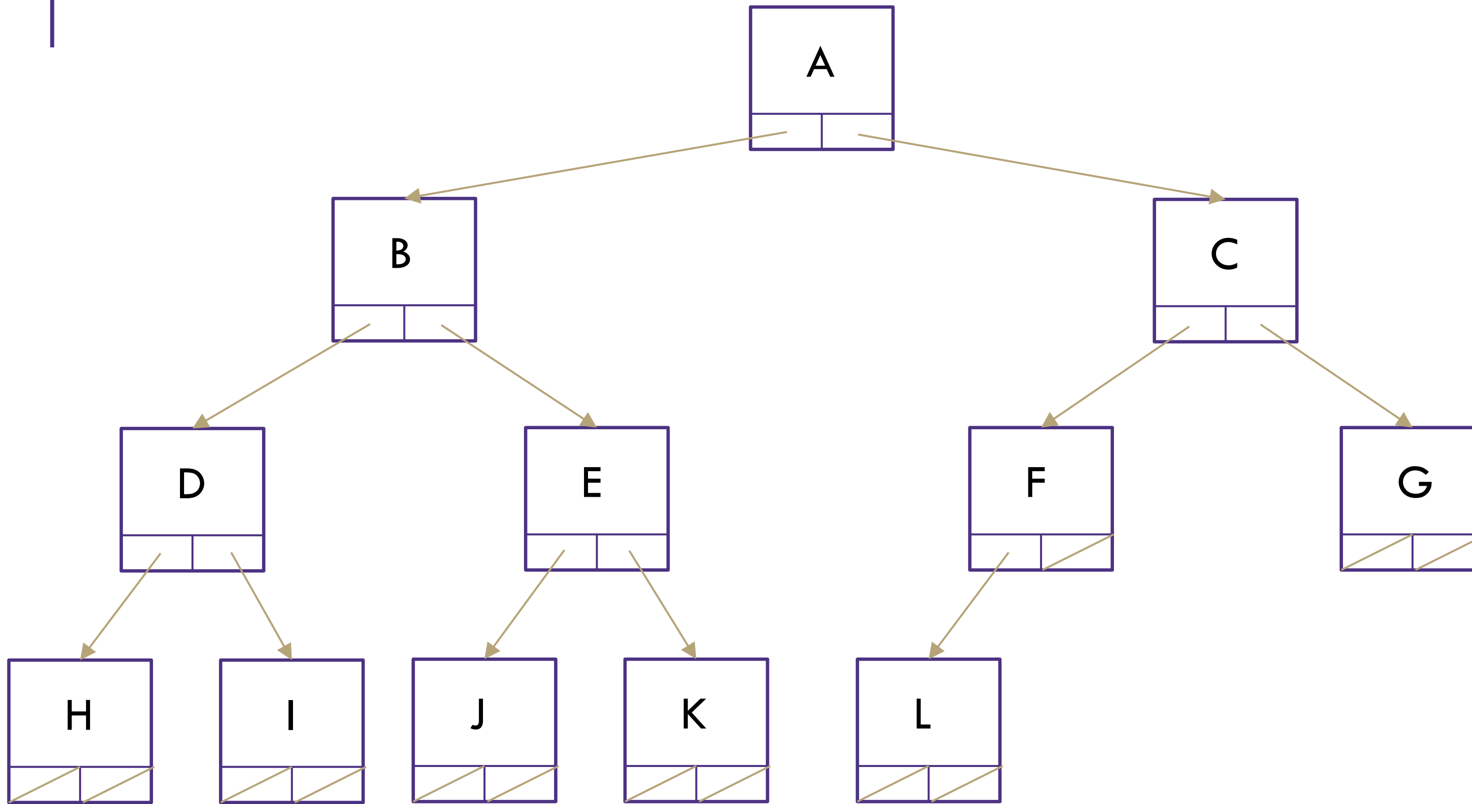


add():
- Insert new node into next available spot
- Percolate up to fix heap invariant

Finding the last node/next available spot is the hard part.
You can do it in $\Theta(\log n)$ time on complete trees, with some extra class variables…
But it's NOT fun

And there's a much better way!

Implement Heaps with an array



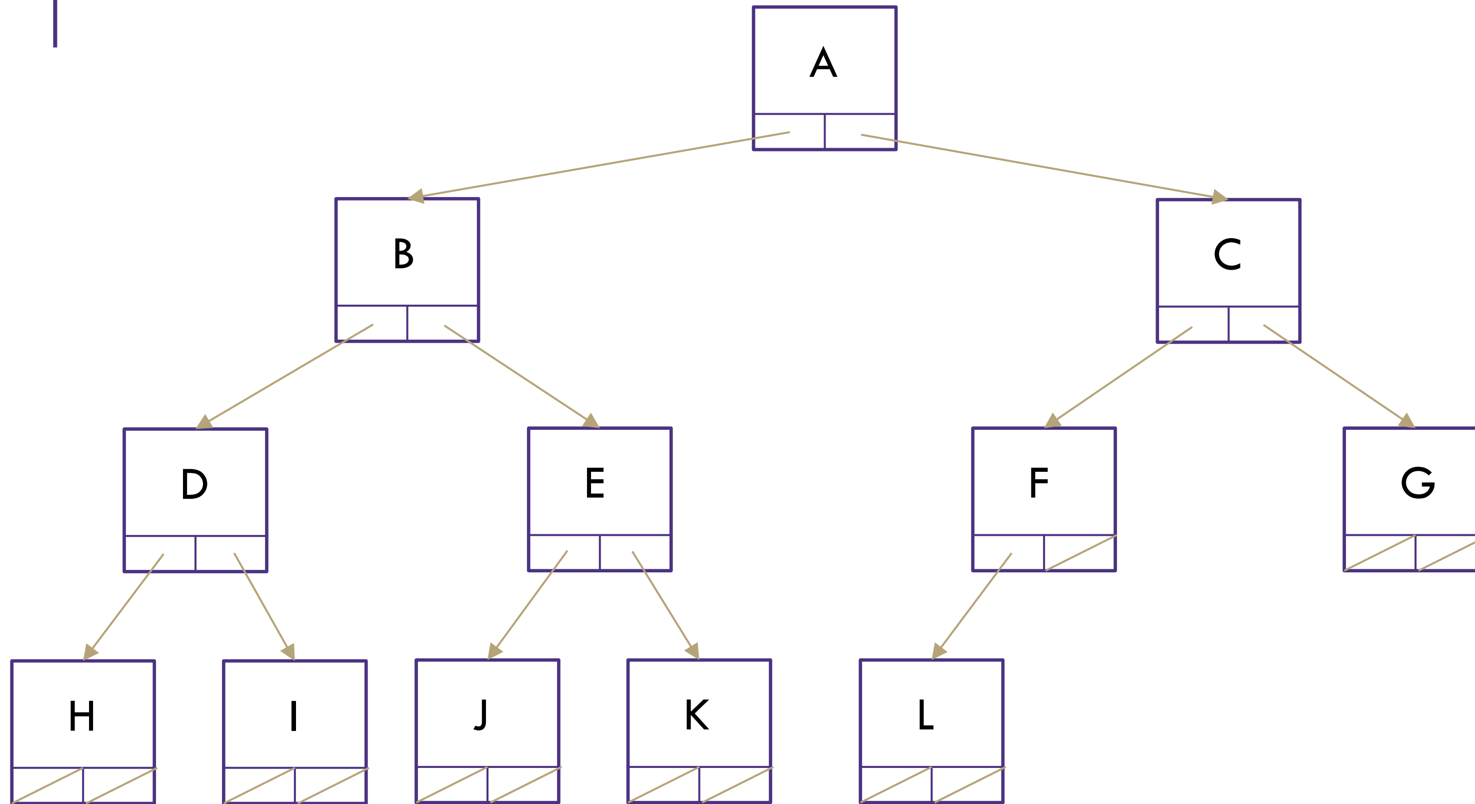
We map our binary-tree representation of a heap into an array implementation where you fill in the array in level-order from left to right.

The array implementation of a heap is what people actually implement, but the tree drawing is how to think of it conceptually. Everything we've discussed about the tree representation still is true!

Fill array in **level-order** from left to right

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	C	D	E	F	G	H	I	J	K	L		

Implement Heaps with an array



Fill array in **level-order** from left to right

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	C	D	E	F	G	H	I	J	K	L		

How do we find the minimum node?

$$peekMin() = arr[0]$$

How do we find the last node?

$$lastNode() = arr[size - 1]$$

How do we find the next open space?

$$openSpace() = arr[size]$$

How do we find a node's left child?

$$leftChild(i) = 2i + 1$$

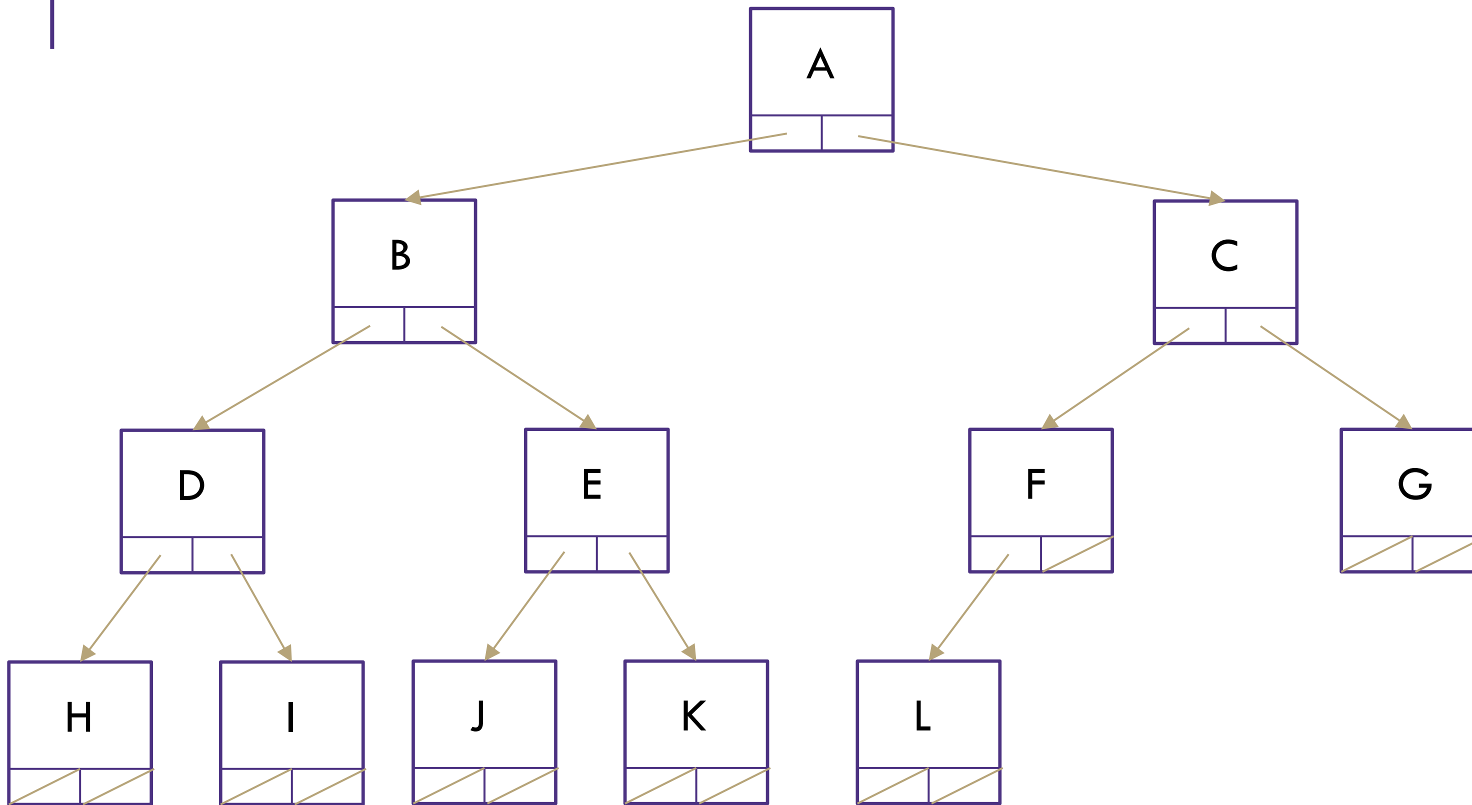
How do we find a node's right child?

$$rightChild(i) = 2i + 2$$

How do we find a node's parent?

$$parent(i) = \frac{(i - 1)}{2}$$

Implement Heaps with an array



Fill array in **level-order** from left to right

0	1	2	3	4	5	6	7	8	9	10	11	12	13
/	A	B	C	D	E	F	G	H	I	J	K	L	

How do we find the minimum node?

$$peekMin() = arr[1]$$

How do we find the last node?

$$lastNode() = arr[size]$$

How do we find the next open space?

$$openSpace() = arr[size + 1]$$

How do we find a node's left child?

$$leftChild(i) = 2i$$

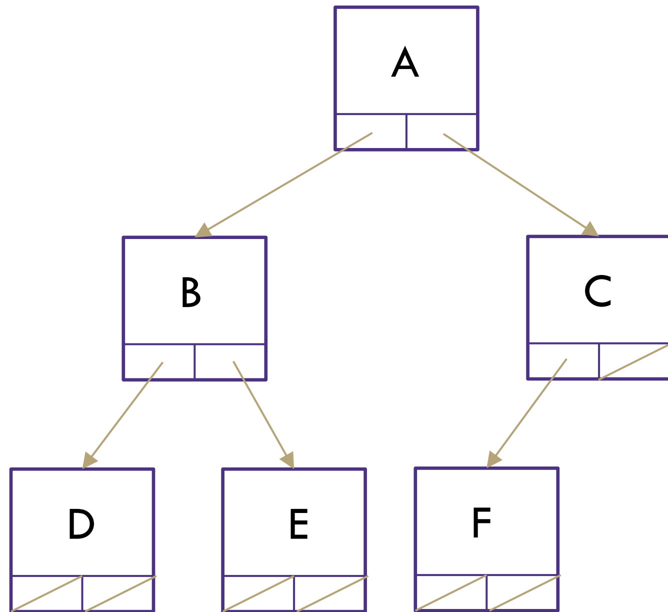
How do we find a node's right child?

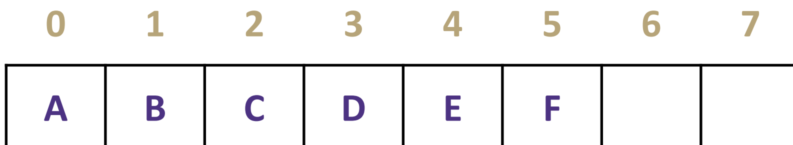
$$rightChild(i) = 2i$$

How do we find a node's parent?

$$parent(i) = \frac{i}{2}$$

Heap Implementation Runtimes





Implementation	add	removeMin	Peek
Array-based heap	worst: $\Theta(\log n)$ in-practice: $\Theta(1)$	worst: $\Theta(\log n)$ in-practice: $\Theta(\log n)$	$\Theta(1)$

`add()`

- worst – the item added is the new minimum and has to traverse all the way to the top of the tree
- in-practice – most nodes are near the bottom of the tree, so in practice new values rarely travel further than a level or two

`removeMin()`

- worst – the item pulled from the bottom of the tree is large and has to percolate all the way back down
- in-practice – because we pull an item from the bottom level to replace the top node, that is probably where it belongs and has to percolate all the way back down

We've matched the **asymptotic worst-case** behavior of AVL trees, but we're actually doing better!
- The constant factors for array accesses are better.
- The tree can be a constant factor shorter because of stricter height invariants.
- In-practice case for add is really good.
- A heap is MUCH simpler to implement.

Are heaps always better? AVL vs Heaps

- The really amazing things about heaps over AVL implementations are the constant factors (e.g. 1.2n instead of 2n) and the sweet sweet Theta(1) in-practice `add` time.

- The really amazing things about AVL implementations over heaps is that AVL trees are absolutely sorted, and they guarantee worst-case be able to find (contains/get) in Theta(log(n)) time.

If heaps have to implement methods like contains/get/ (more generally: finding a particular value inside the data structure) – it pretty much just has to loop through the array and incur a worst-case Theta(n) runtime.

Heaps are stuck at Theta(n) runtime and we can't do anything more clever…. aha, just kidding.. unless…?

More Operations

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

add(value) – add a new element to the collection

removeMin() – returns the element with the smallest priority, removes it from the collection

peekMin() – find, but do not remove the element with the smallest priority

We'll use priority queues for lots of things later in the quarter.

Let's add them to our ADT now.

Some of these will be **asymptotically** faster for a heap than an AVL tree!

BuildHeap(elements $e_1, \ldots, e_n$ **)**
Given n elements, create a heap containing exactly those n elements.

Even More Operations

BuildHeap(elements $e_1, \ldots, e_n$ **) –** Given n elements, create a heap containing exactly those n elements.

Try 1: Just call insert n times.

Worst case running time?

n calls, each worst case $\Theta(\log n)$. So it's $\Theta(n \log n)$ right?

That proof isn't valid. There's no guarantee that we're getting the worst case every time!

Proof is right if we just want an $O()$ bound

- But it's not clear if it's tight.

BuildHeap Running Time

Let’s try again for a Theta bound.

The problem last time was making sure we always hit the worst case.

If we insert the elements in decreasing order **we will!**

- Every node will have to percolate all the way up to the root.

So we really have $n\ \Theta(\log n)$ operations. QED.

There’s still a bug with this proof!

BuildHeap Running Time (again)

Let's try once more.

Saying the worst case was decreasing order was a good start.

What are the actual running times?

It's $\Theta(h)$, where h is the **current** height.

- The tree isn't height $\log n$ at the beginning.

But most nodes are inserted in the last two levels of the tree.

- For most nodes, h is $\Theta(\log n)$.

The number of operations is at least

$$\frac{n}{2} \cdot \Omega(\log n) = \Omega(n \log n).$$

Can We Do Better?

What's causing the n `insert` strategy to take so long?

Most nodes are near the bottom, and they might need to percolate all the way up.

What if instead we dumped everything in the array and then

tried to percolate things down to fix the invariant?

Seems like it might be faster

- The bottom two levels of the tree have $\Omega(n)$ nodes, the top two have 3 nodes.
- Maybe we can make "most nodes" go a constant distance.

Is It Really Faster?

Assume the tree is **perfect**

- the proof for complete trees just gives a different constant factor.

percolateDown() doesn't take $\log n$ steps each time!

Half the nodes of the tree are leaves

- Leaves run percolate down in constant time

1/4 of the nodes have at most 1 level to travel

1/8 the nodes have at most 2 levels to travel

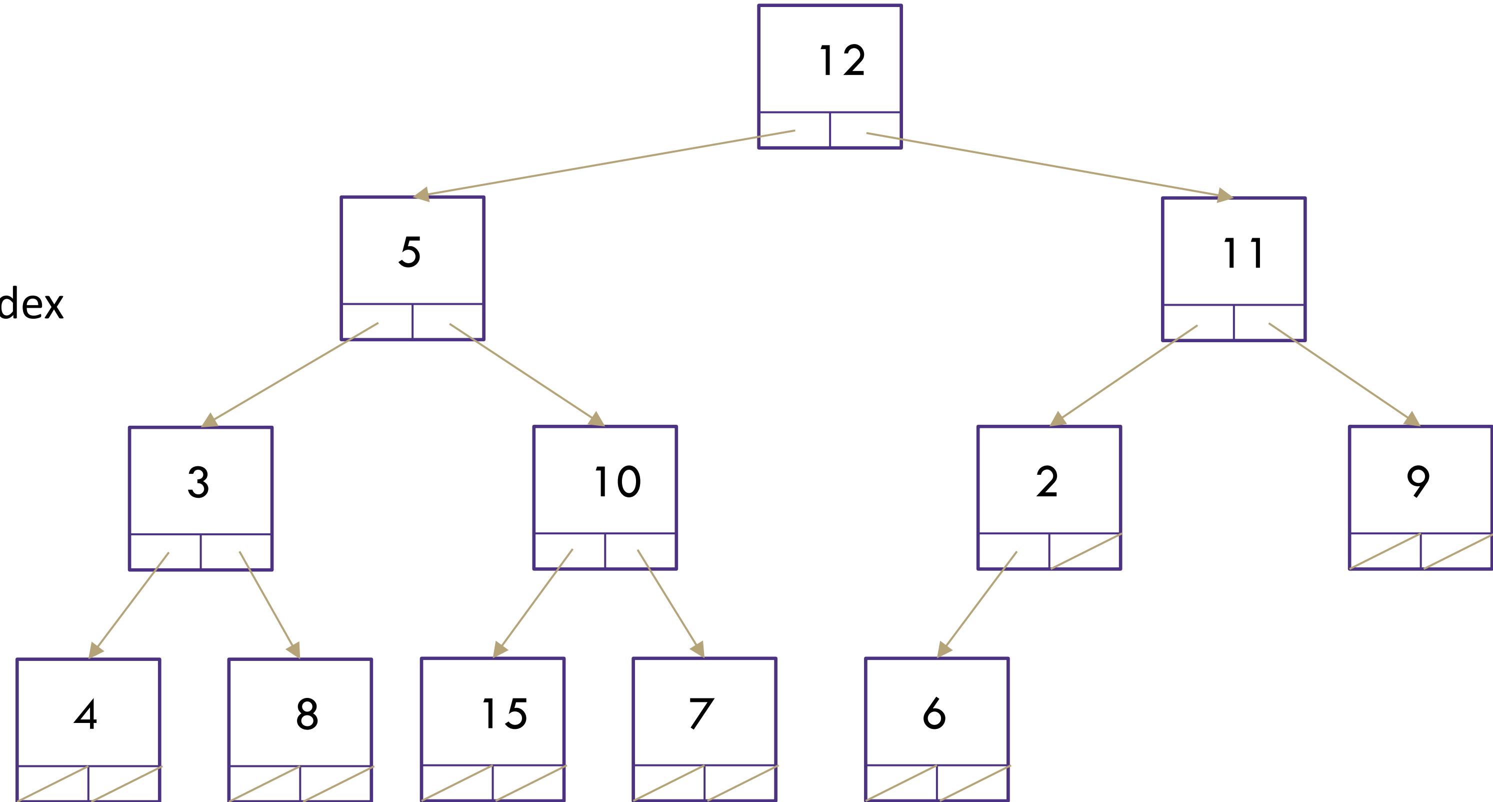
etc...

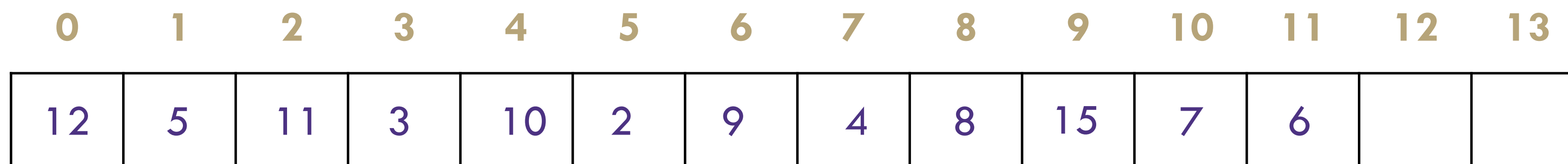
work(n) ≈ $\frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 1 \cdot (\log n)$

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index

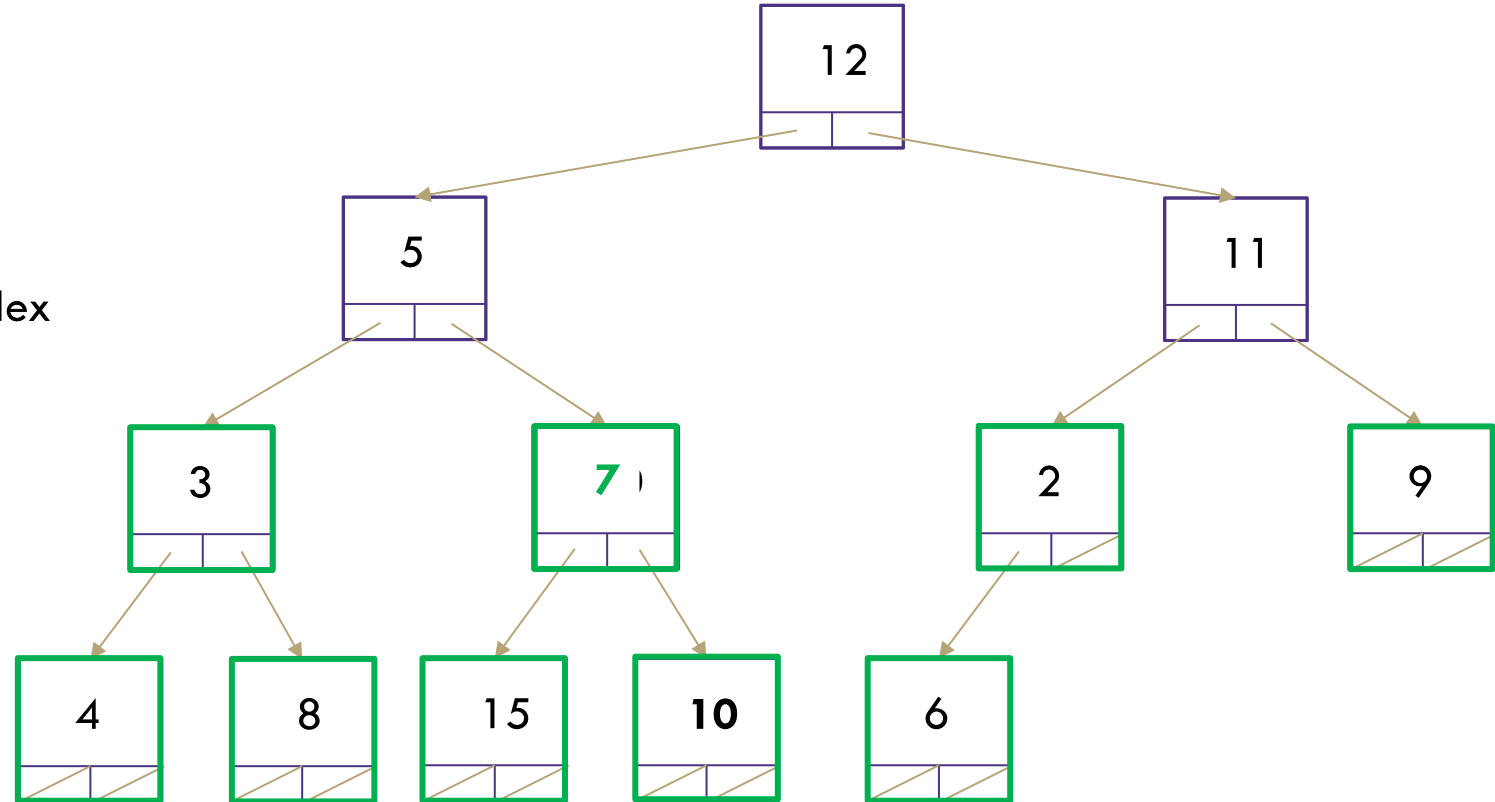


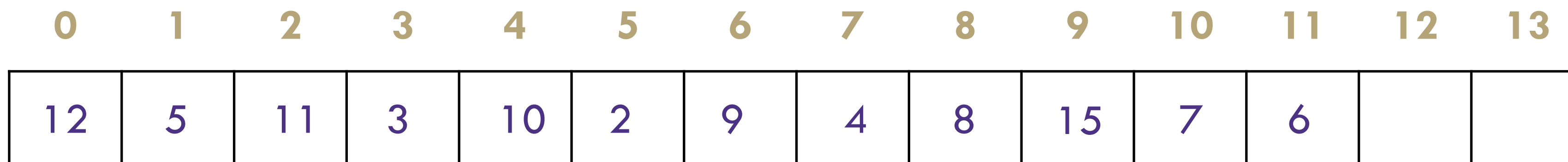


Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
    1. percolateDown level 4
    2. percolateDown level 3



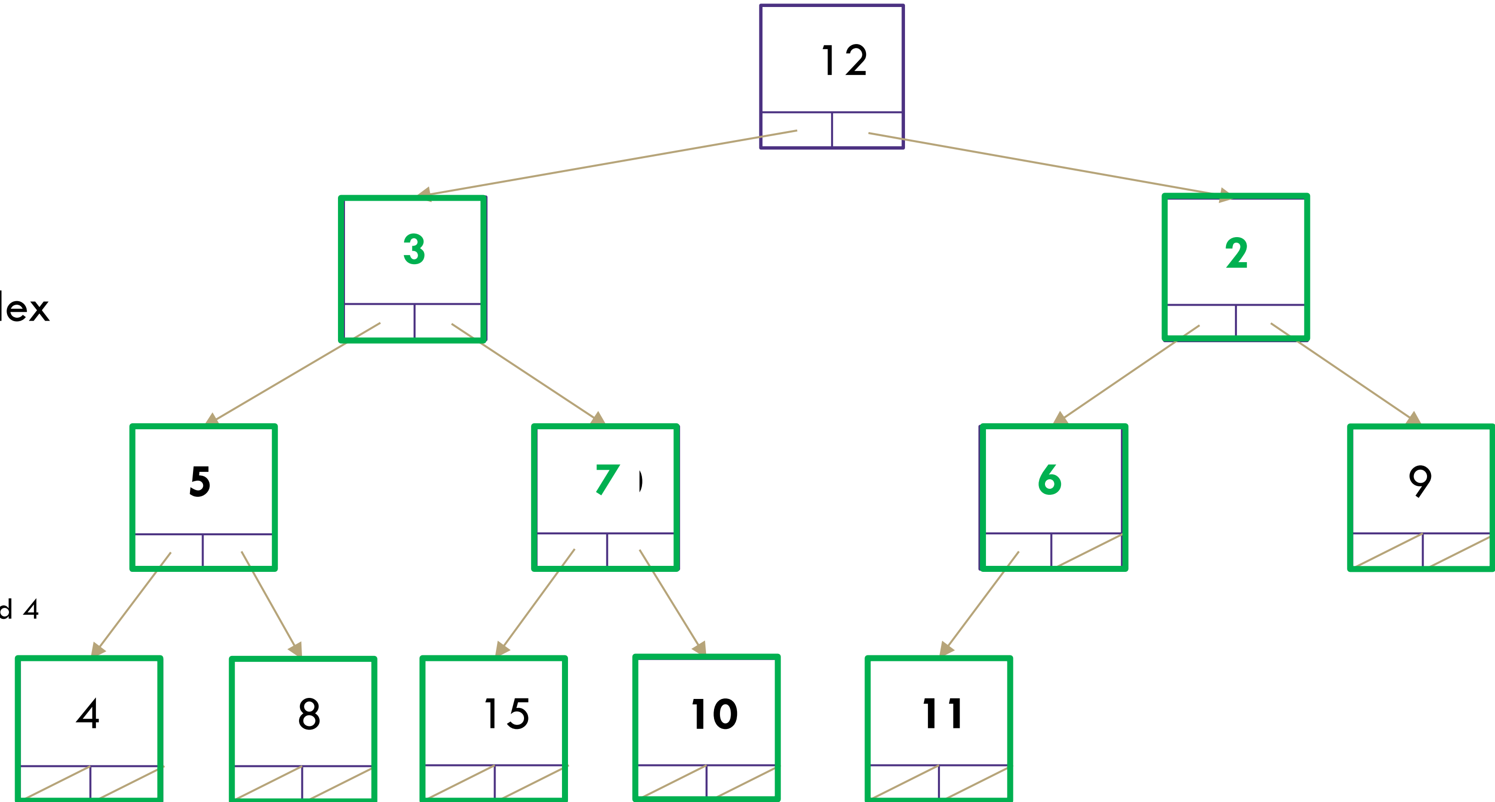


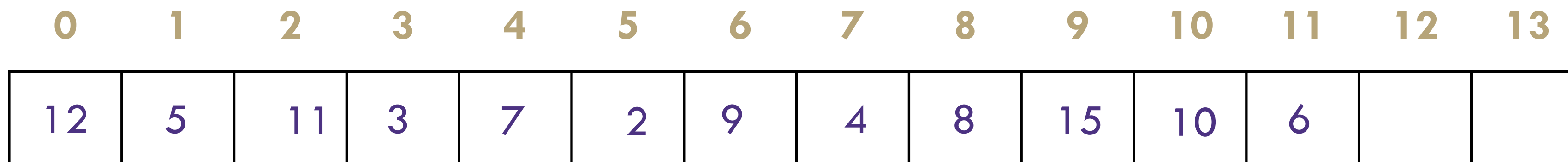
Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2

keep percolating down
like normal here and swap 5 and 4



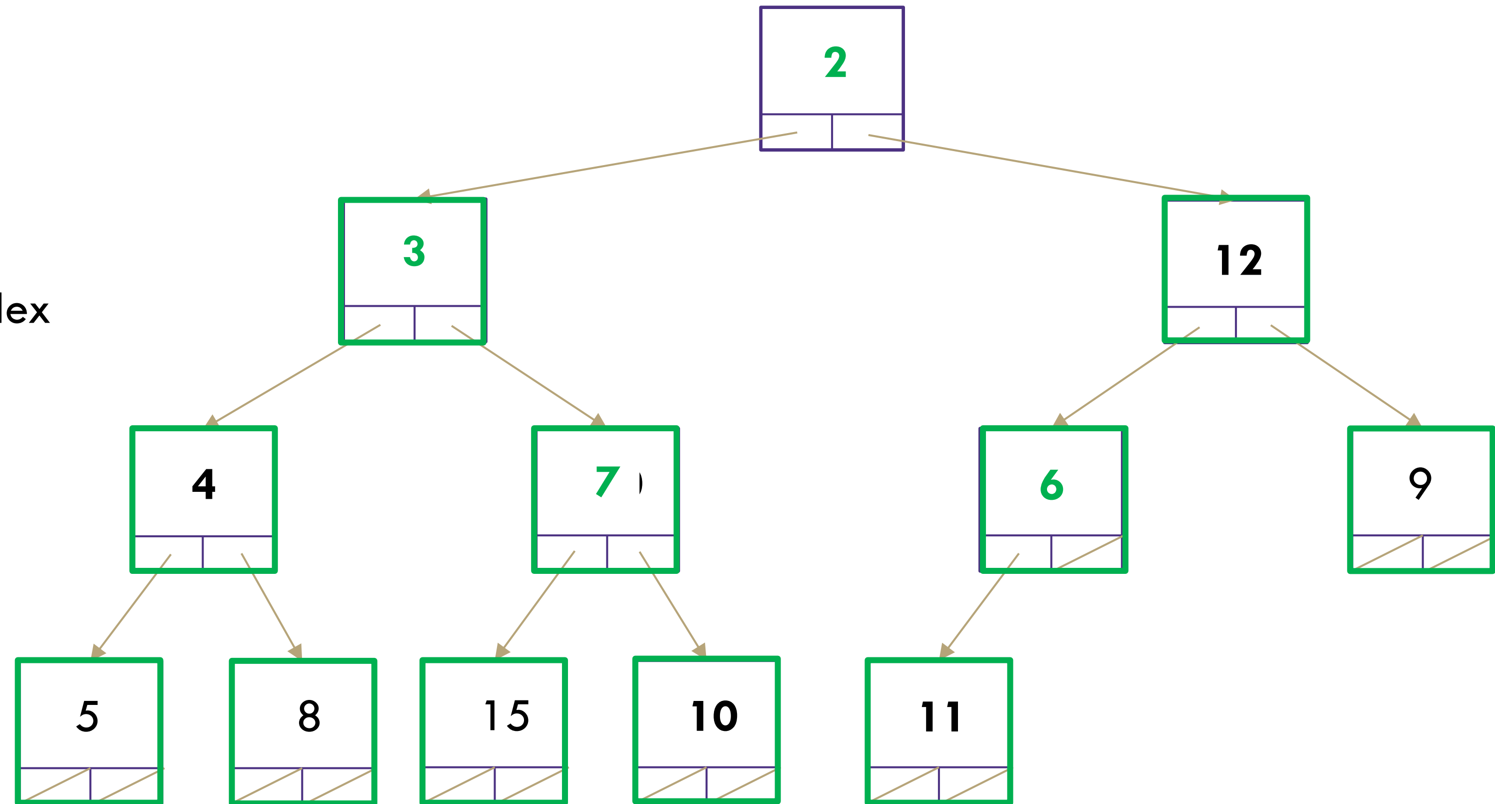


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	7	2	9	4	8	15	10	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2
   4. percolateDown level 1

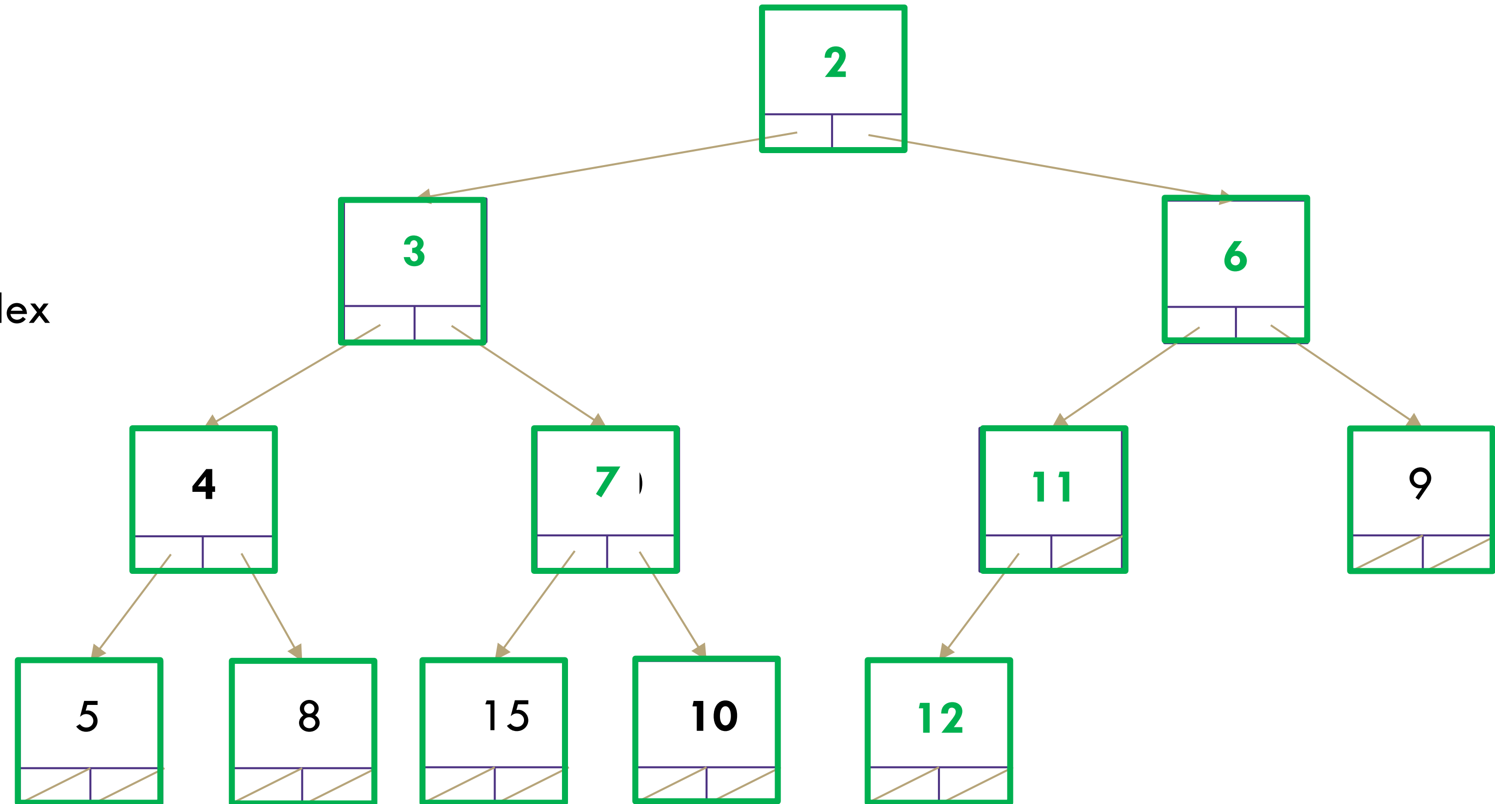


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	3	2	4	7	6	9	5	8	15	10	11		

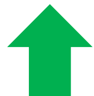
Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2
   4. percolateDown level 1



0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	6	4	7	11	9	5	8	15	10	12		

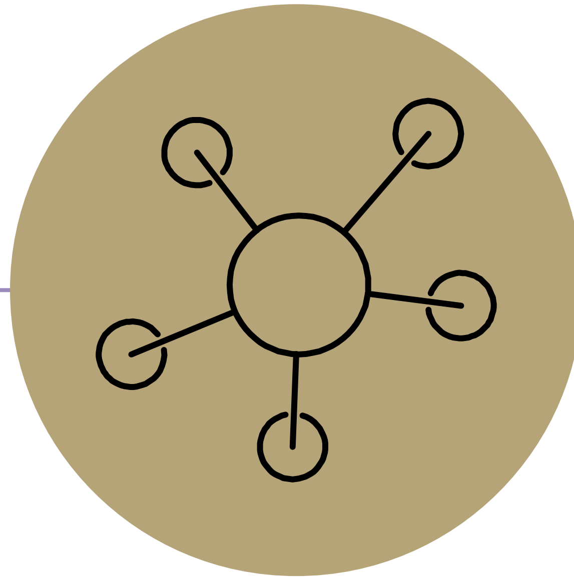


Relevant hint for project 3:

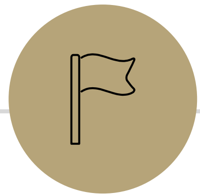
- When coming up with data structures, we can actually combine them with existing tools to improve our algorithms and runtimes. We can improve the worst-case runtime of get/contains to be a lot better than Theta(n) time depending on how we have our heap utilize an extra data-structure.

- For project 3, you should use an additional data structure to improve the runtime for `changePriority()`. It does not affect the correctness of your PQ at all (i.e. you can implement it correctly without the additional data structure). Please use a built-in Java collection instead of implementing your own (although you could in-theory).

- For project 3, feel free to try the following development strategy for the `changePriority` method
  - implement `changePriority` without regards to efficiency (without the extra data structure) at first
  - then, analyze your code's runtime and figure out which parts are inefficient
  - reflect on the data structures we've learned and see how any of them could be useful in improving the slow parts in your code



Questions



Introduction to Graphs

Inter-data Relationships

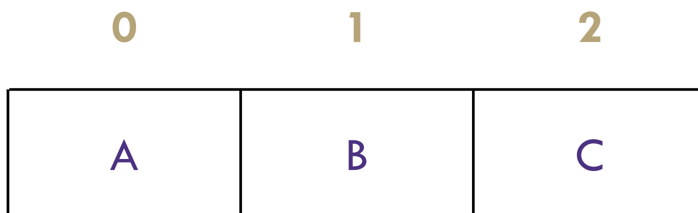
Arrays

Categorically associated

Sometimes ordered

Typically independent

Elements only store pure data, no connection info



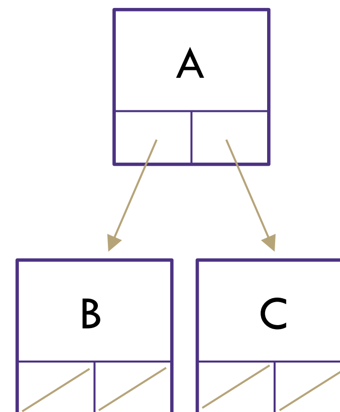
Trees

Directional Relationships

Ordered for easy access

Limited connections

Elements store data and connection info



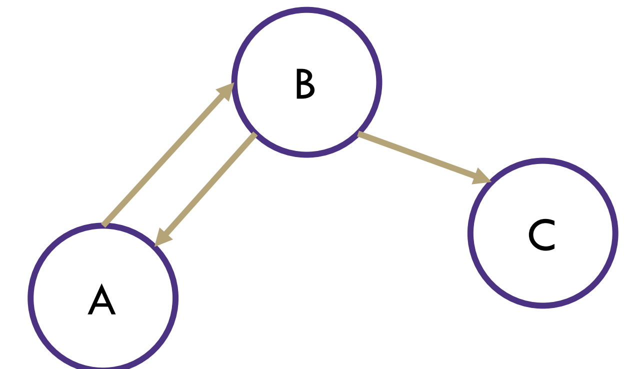
Graphs

Multiple relationship connections

Relationships dictate structure

Connection freedom!

Both elements and connections can store data



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

Graph: Formal Definition

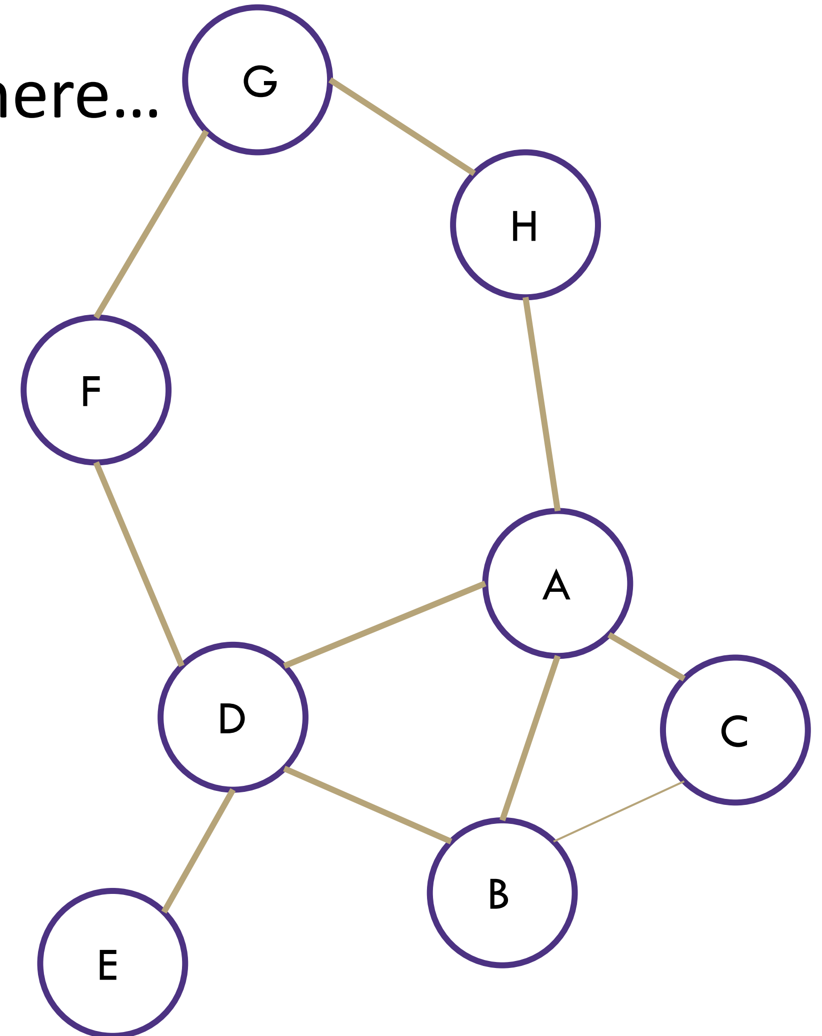
A **graph** is defined by a pair of sets G = (V, E) where…

- V is a set of **vertices**
  - A vertex or “node” is a data entity

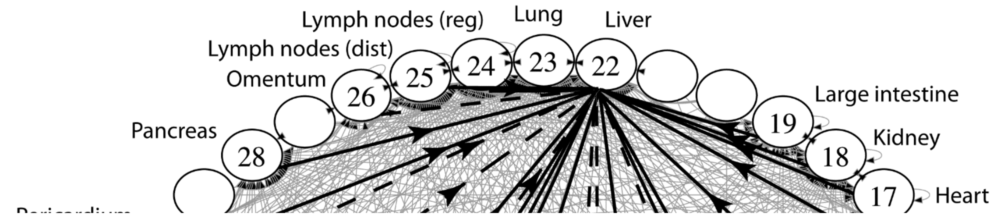
V = { A, B, C, D, E, F, G, H }

- E is a set of **edges**
  - An edge is a connection between two vertices

E = { (A, B), (A, C), (A, D), (A, H),
(C, B), (B, D), (D, E), (D, F),
(F, G), (G, H)}

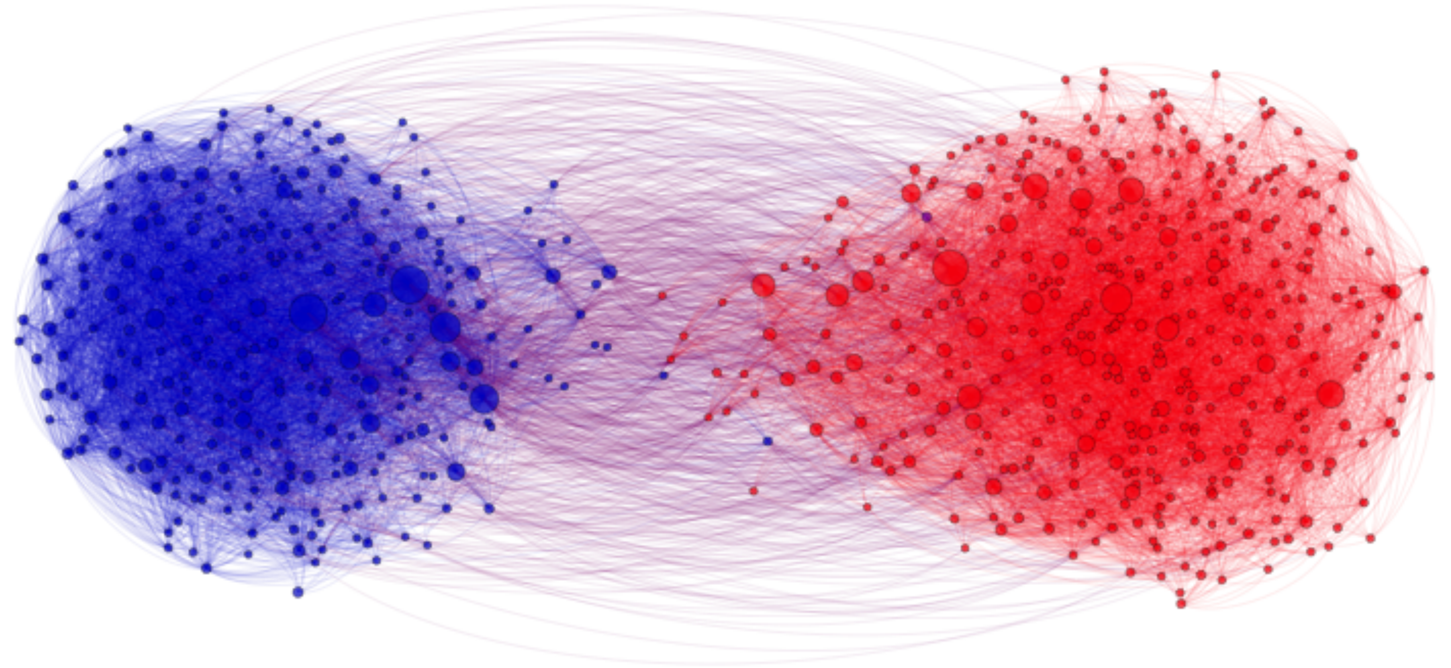


Applications



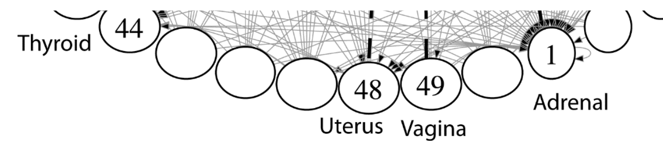
Physical Maps
- Airline maps
  - Vertices are airports, edges are flight paths
- Traffic
  - Vertices are addresses, edges are streets

Relationships
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Code bases
  - Vertices are classes, edges are usage



Influence
- Biology
  - Vertices are cancer cell destinations, edges are migration paths

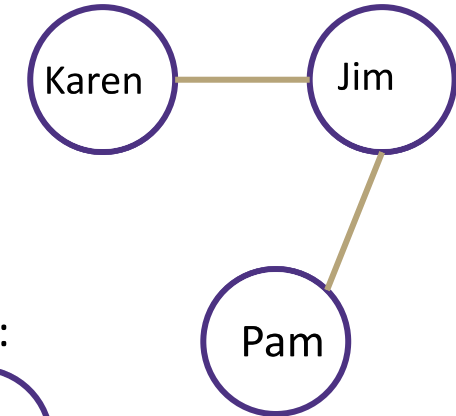
Related topics
- Web Page Ranking
  - Vertices are web pages, edges are hyperlinks
- Wikipedia
  - Vertices are articles, edges are links



SO MANY MORREEEE

www.allthingsgraphed.com

Graph Vocabulary



Graph Direction

- **Undirected graph** – edges have no direction and are two-way

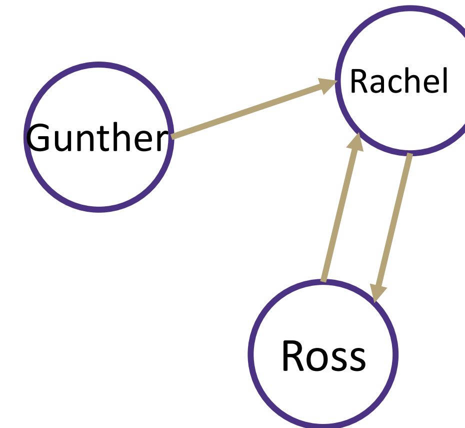
  V = { Karen, Jim, Pam }

  E = { (Jim, Pam), (Jim, Karen) } *inferred (Karen, Jim) and (Pam, Jim)*

- **Directed graphs** – edges have direction and are thus one-way

  V = { Gunther, Rachel, Ross }

  E = { (Gunther, Rachel), (Rachel, Ross), (Ross, Rachel) }



Degree of a Vertex

- **Degree** – the number of edges connected to that vertex

  Karen : 1, Jim : 1, Pam : 1

- **In-degree** – the number of directed edges that point to a vertex

  Gunther : 0, Rachel : 2, Ross : 1

- **Out-degree** – the number of directed edges that start at a vertex

  Gunther : 1, Rachel : 1, Ross : 1

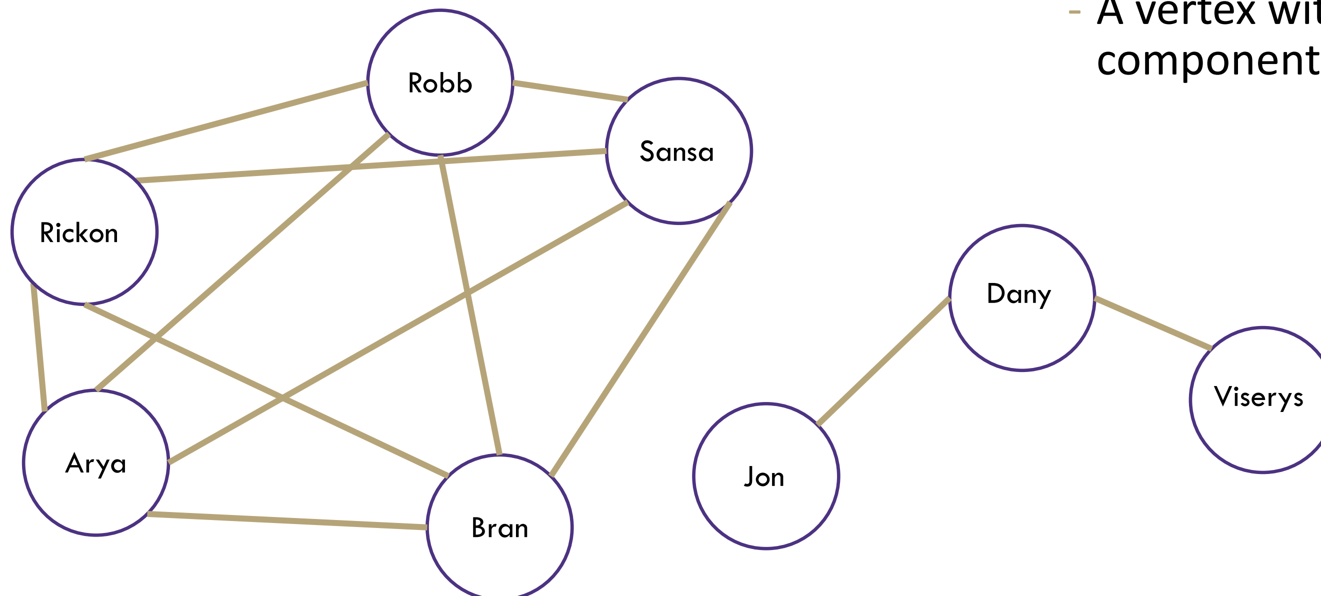
Connected Graphs

Connected graph – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex

There exists some way to get from each vertex to every other vertex

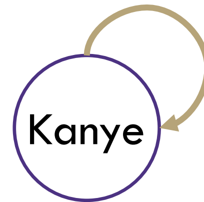
Connected Component – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*

- There exists some way to get from each vertex within the connected component to every other vertex in the connected component
- A vertex with no edges is itself a connected component

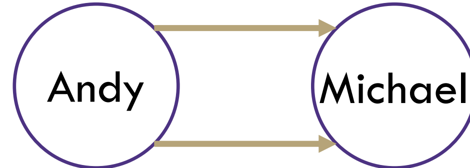


Graph Vocabulary

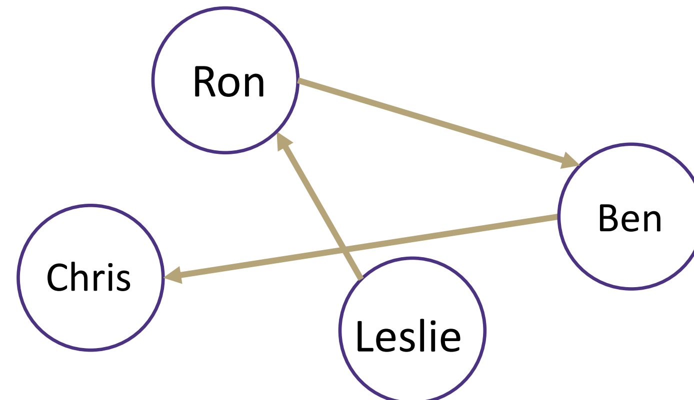
Self loop – an edge that starts and ends at the same vertex



Parallel edges – two edges with the same start and end vertices

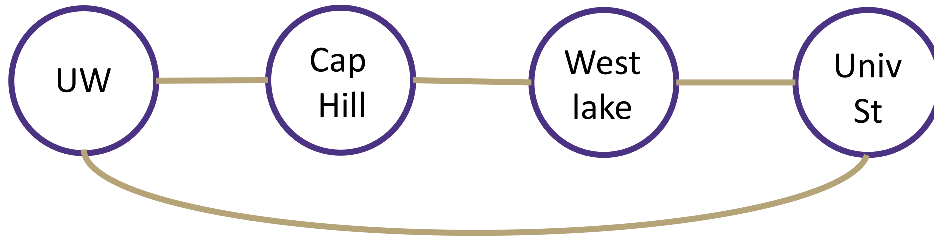


Simple graph – a graph with no self-loops and no parallel edges



Graph Terms

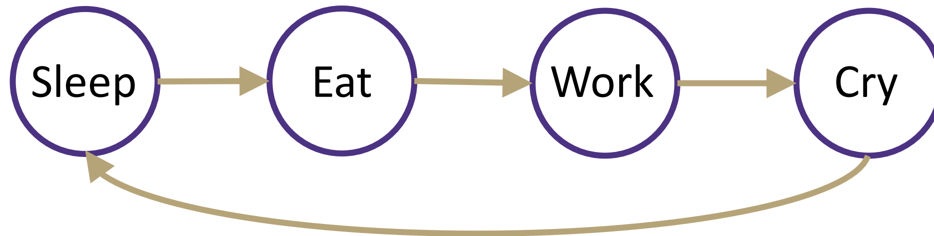
Walk – A sequence of **adjacent** vertices. Each connected to next by an edge.



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

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

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



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

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

Length – The number of edges in a walk/path/cycle. (A,B,C,D) has length 3.

Implementing a Graph

Implement with nodes…

Implementation gets super messy

What if you wanted a vertex without an edge?

How can we implement without requiring edges to access nodes?

Implement using some of our existing data structures!

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

Poll Everywhere!
Pollev.com/cse373activity

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

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?

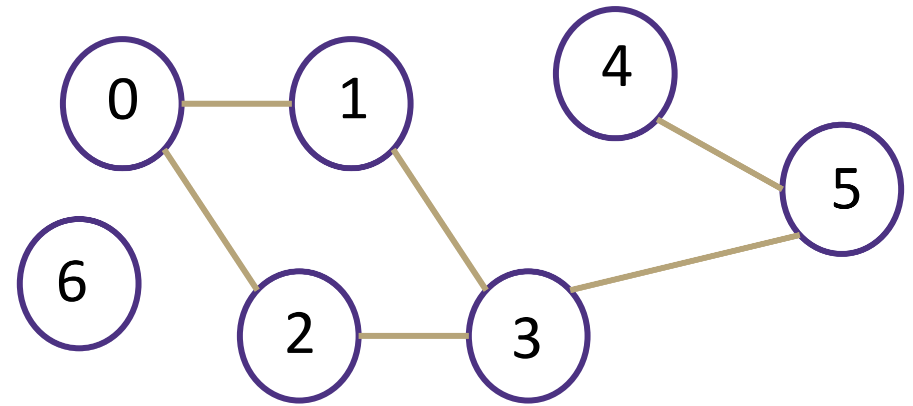
Input data for the "6 Degrees of Kevin Bacon" game

- **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.

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

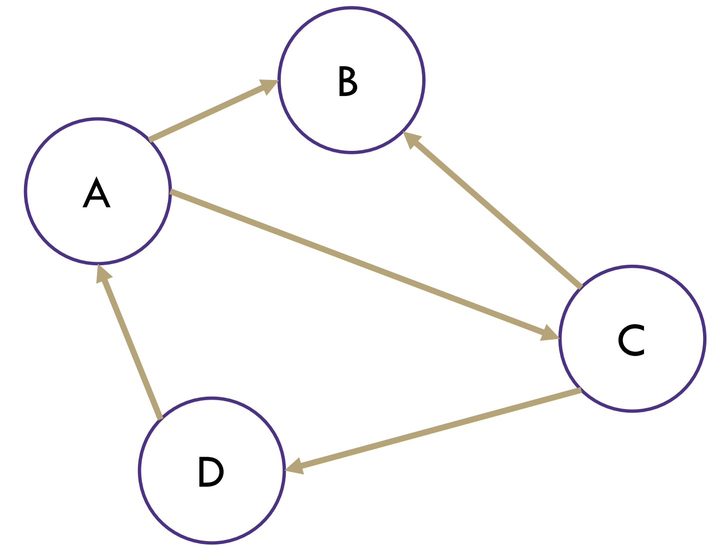
Create a Dictionary of size V from type V to Collection of E

If (x,y) ∈ E then add y to the set associated with the key x

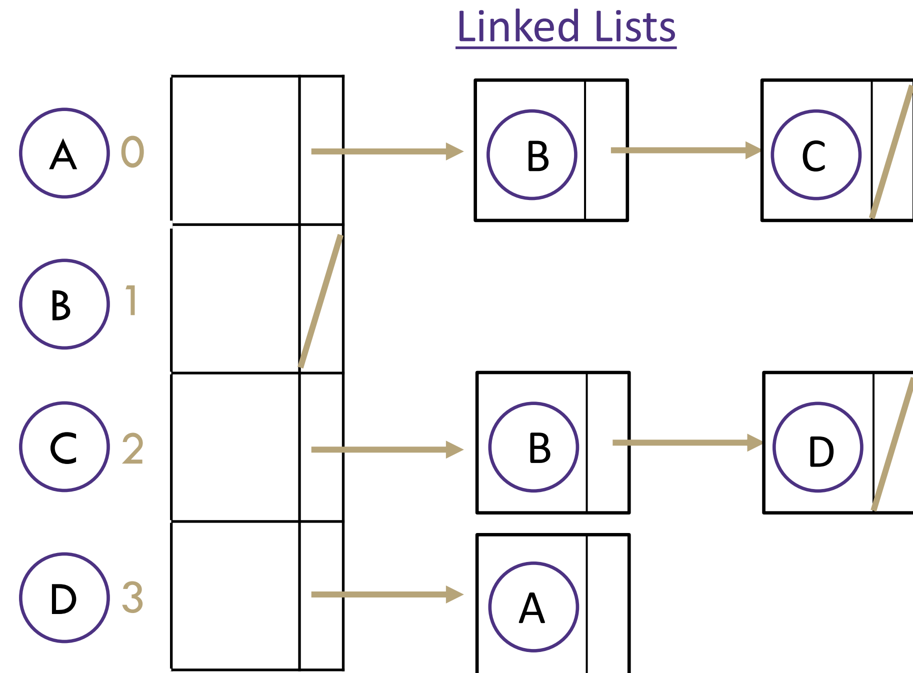
An array where the uth element contains a list of neighbors of u.
Directed graphs: list of out-neighbors (a[u] has v for all (u,v) in E)
Time Complexity (|V| = n, |E| = m):

- Add Edge: $\mathbf{\Theta(1)}$
- Remove Edge (u,v): $\mathbf{\Theta(\ deg(\mathit{u})\ )}$
- Check edge exists from (u,v): $\mathbf{\Theta(\ deg(\mathit{u})\ )}$
- Get neighbors of u (out): $\mathbf{\Theta(deg(\mathit{u}))}$
- Get neighbors of u (in): $\mathbf{\Theta(\mathit{n} + \mathit{m})}$

Space Complexity: $\mathbf{\Theta(\mathit{n} + \mathit{m})}$



Linked Lists



Adjacency List

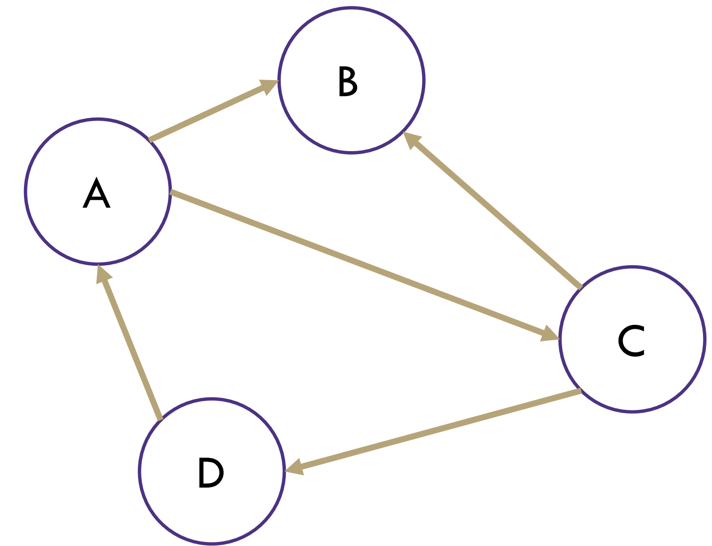
Create a Dictionary of size V from type V to Collection of E

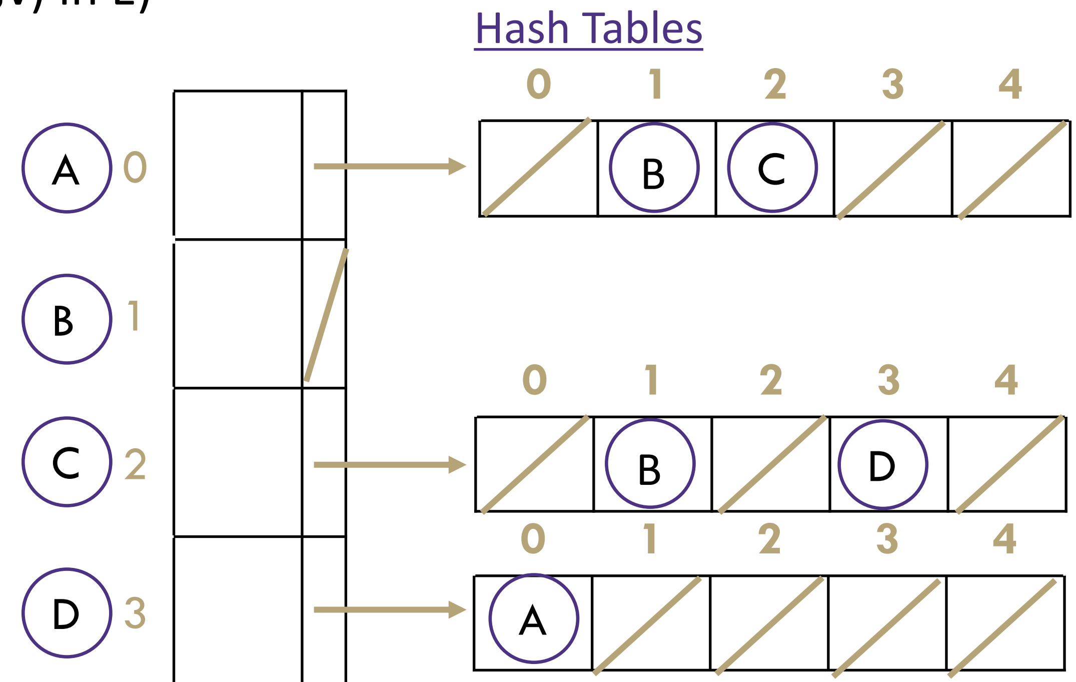
If (x,y) ∈ E then add y to the set associated with the key x

An array where the uth 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: $\boldsymbol{\Theta(1)}$
- Remove Edge (u,v): $\boldsymbol{\Theta(1)}$
- Check edge exists from (u,v): $\boldsymbol{\Theta(1)}$
- Get neighbors of u (out): $\boldsymbol{\Theta(\deg(u))}$
- Get neighbors of u (in): $\boldsymbol{\Theta(n)}$

Space Complexity: $\boldsymbol{\Theta(n + m)}$





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.