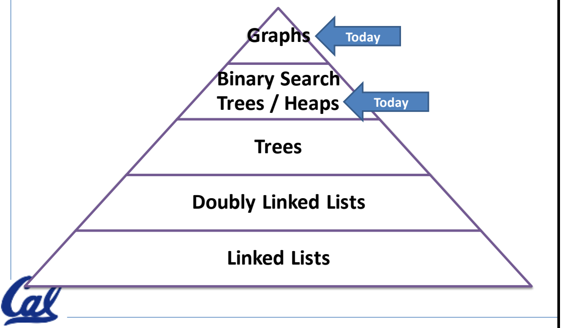
Queues, Priority Queues and Graphs

CS61BL Spring 2011

Tower of AWESOME



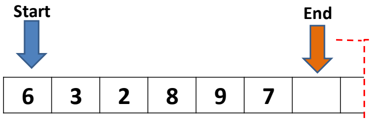
Queues and Priority Queues

Queues (Review)

Queues (Review)

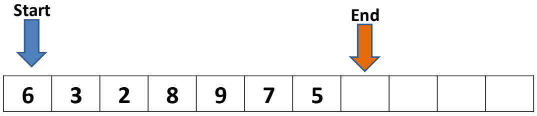
```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public void enqueue(Object item);
  public Object dequeue() throws EmptyQueueException;
  public Object front() throws EmptyQueueException;
}
```

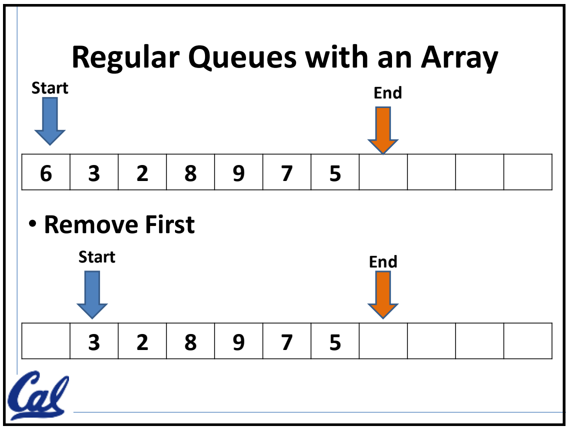
Regular Queues with an Array



Notice the invariant of what “End” and “Start” represent.

- Insert 5





Regular Queues with an Array

Start

End

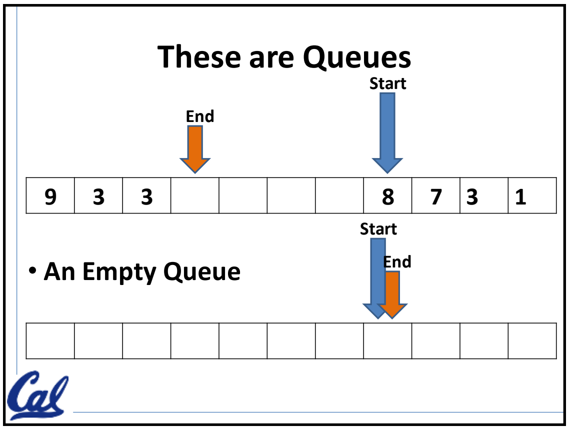
6	3	2	8	9	7	5				

- Remove First

Start

End

	3	2	8	9	7	5				



These are Queues

Start

End

9	3	3					8	7	3	1

- An Empty Queue

Start

End

A Queue Based upon Priority
PriorityQueue

```
public interface PriorityQueue {
    public int size();
    public boolean isEmpty();
    Entry insert(Object k, Object v);
    Entry max();
    Entry removeMax();
    }
```

PriorityQueues Implemented with Sorted LinkedLists

Operation	Run Time	Other Requirements
`size()`		
`isEmpty()`		
`insert(…)`		
`max()`		
`removeMax()`		

PriorityQueues Implemented with Non-Sorted LinkedLists

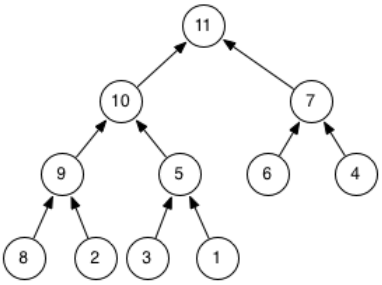
Operation	Run Time	Other Requirements
`size()`		
`isEmpty()`		
`insert(…)`		
`max()`		
`removeMax()`		

The big idea is that there are tradeoffs!

Heaps

This is a Max Heap

What would you guess are the invariants?



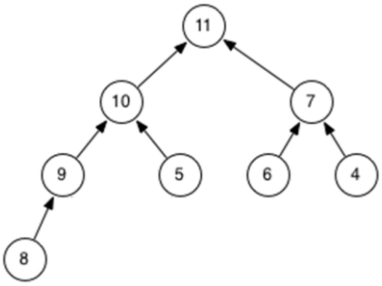
Binary Heaps (for your notes)

- A Binary Heap is a binary tree, with two additional properties
  - **Shape Property:** It is a complete binary tree – a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right
    - MAXIMALLY BALANCED!
  - **Heap Property (or Heap Order Property):** *No child has a key greater than its parent's key.* This property is applied recursively: any subtree of a binary heap is also a binary heap.
    - Nodes are bigger than their descendants
- If we use the notion of *smaller than* in the Heap Property we get a *min-heap.*
- We'll look at *max-heap* in this lecture.

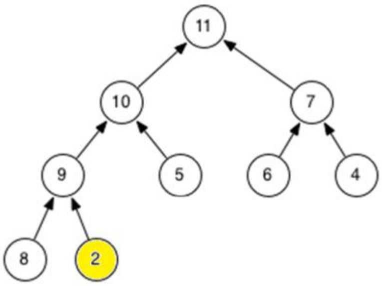
Heap Operations

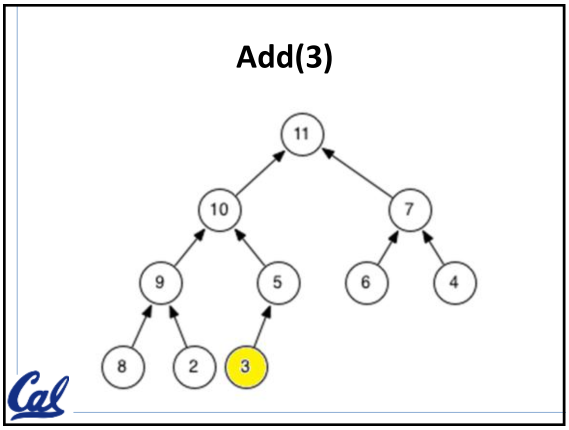
- **Step 1:** Ensure that it is maximally balanced
- **Step 2:** Ensure that every node is bigger than its descendents

Initial Heap

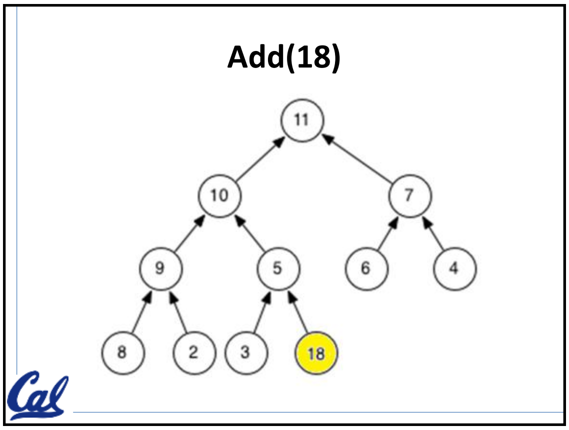


Add(2)

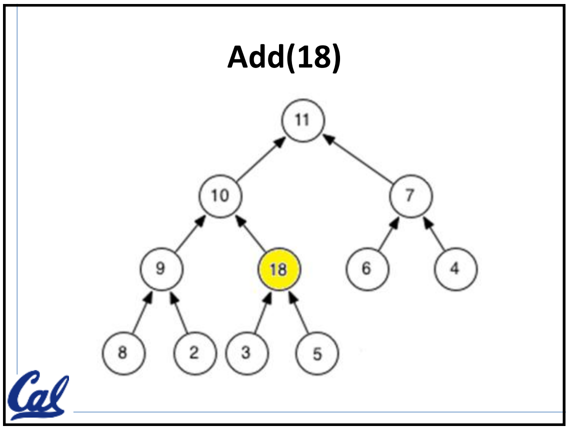




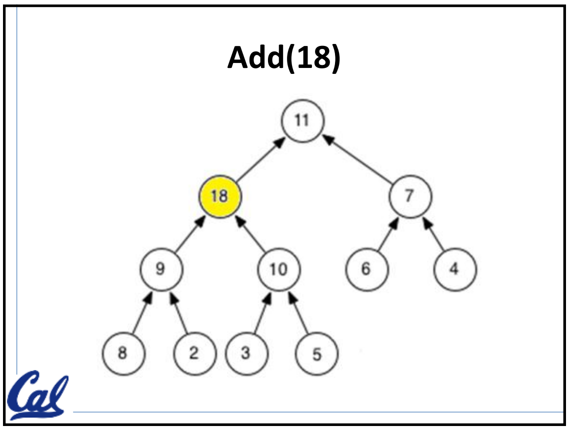
Add(3)



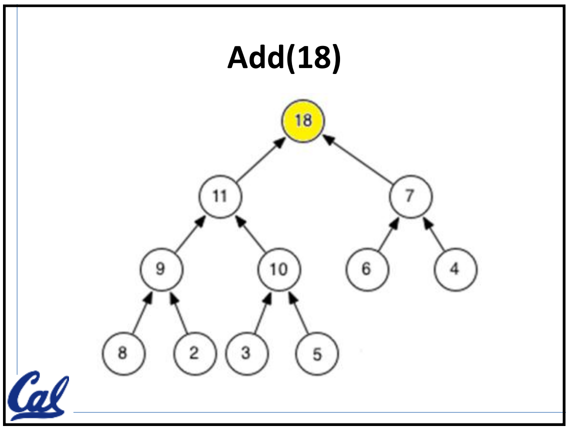
Add(18)



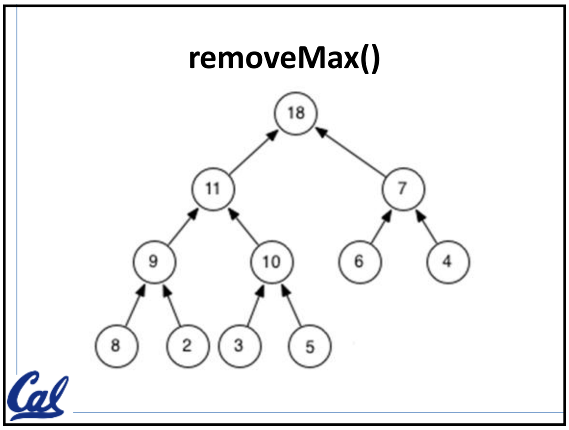
Add(18)



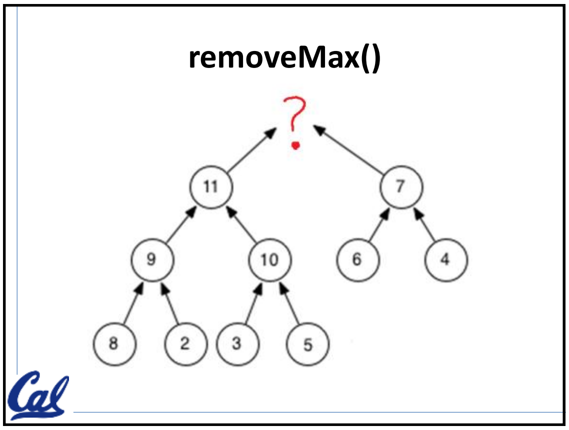
Add(18)



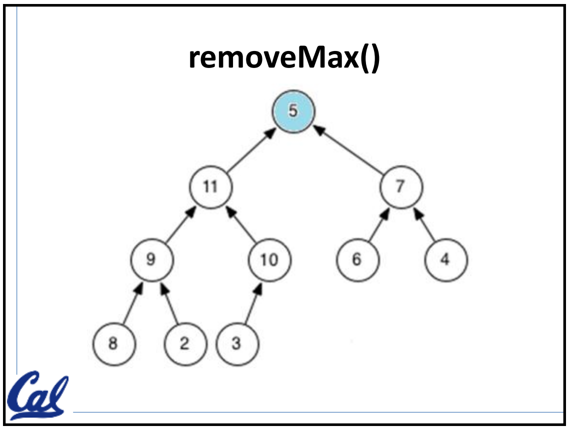
Add(18)



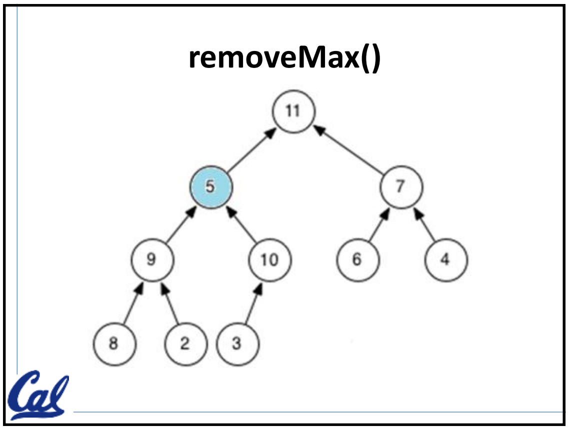
removeMax()



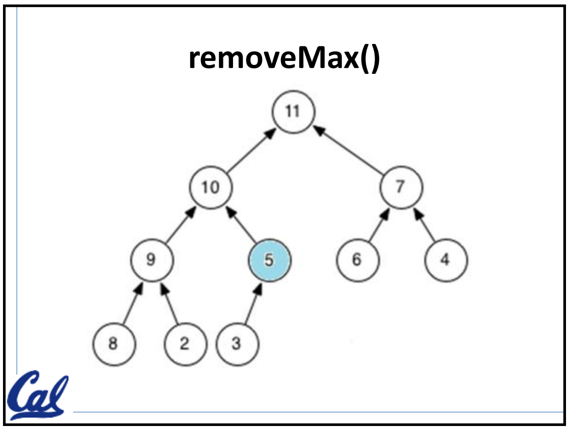
removeMax()



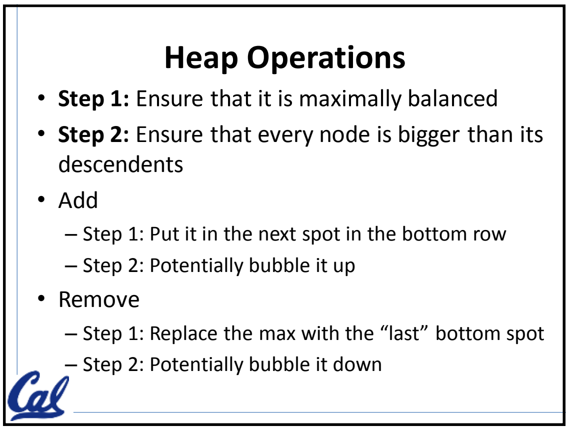
removeMax()



removeMax()

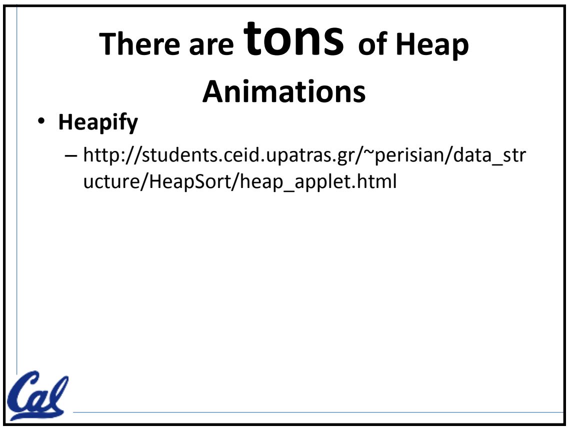


removeMax()



Heap Operations

- **Step 1:** Ensure that it is maximally balanced
- **Step 2:** Ensure that every node is bigger than its descendents
- Add
  - Step 1: Put it in the next spot in the bottom row
  - Step 2: Potentially bubble it up
- Remove
  - Step 1: Replace the max with the “last” bottom spot
  - Step 2: Potentially bubble it down



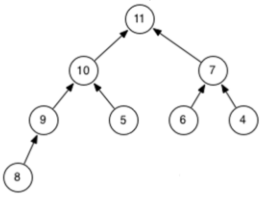
There are **tons** of Heap Animations

- **Heapify**
  - http://students.ceid.upatras.gr/~perisian/data_structure/HeapSort/heap_applet.html

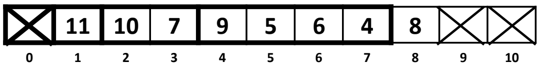
Representing Trees (Review)

Array representations are common for Heaps

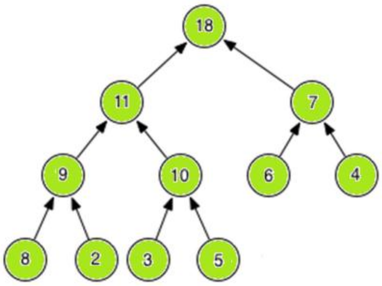
(don't use the 0 index)



Left Child at 2n
Right Child at 2n + 1



Heapify Demo



Heapify Demo Array Version

6	9	7	8	3	18	4	11	2	10	5
6	9	7	**11**	3	18	4	**8**	2	10	5
6	9	7	11	**10**	18	4	8	2	**3**	5
6	**11**	7	**9**	10	18	4	8	2	3	5
6	11	7	9	10	18	4	8	2	3	5
6	11	**18**	9	10	**7**	4	8	2	3	5
18	11	**6**	9	10	7	4	8	2	3	5
18	11	**7**	9	10	**6**	4	8	2	3	5

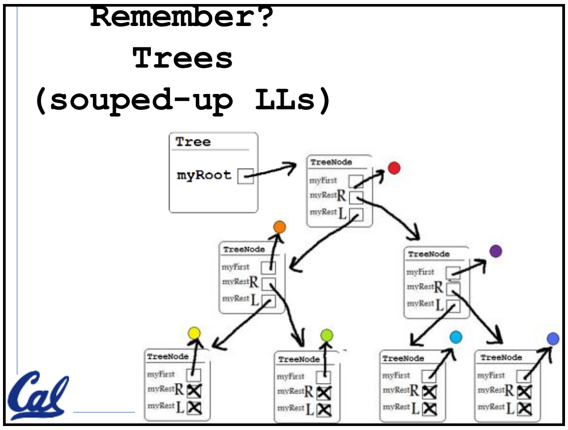
Bottom-Up Heap Construction

- Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them.
- What's the obvious way
  - Apply `insert` to each item in *O(n log n)* time.
- A better way: `bottomUpHeap()`
  1. Make a complete tree out of the entries, in any *random* order.
  2. Start from the last internal node (non-leaf node), in reverse order of the level order traversal, *heapify down* the heap as in `removeMax`().

Cost of Bottom Up Construction

- If *each* internal node bubbles *all the way* down, then the running time is proportional to the sum of the heights of all the nodes in the tree.
- Turns out this sum is less than n, where n is the number of entries being coalesced into a heap.
- Hence, the running time is in *O(n)*, which is better than inserting n entries into a heap individually.

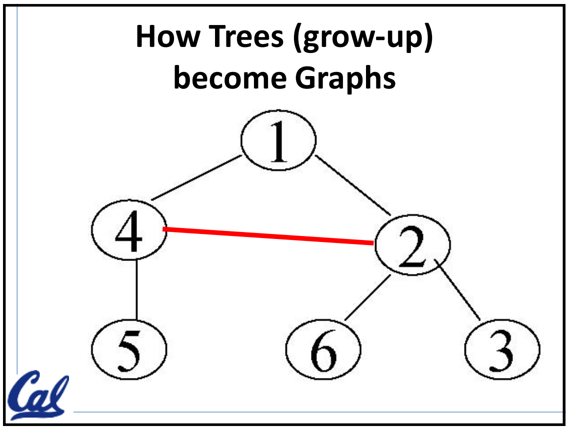
Graphs



Remember? Trees (souped-up LLs)

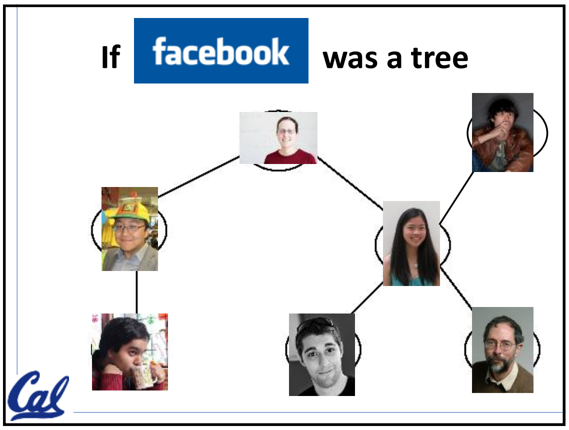
Tree Definitions: Overview

- A ***tree*** consists of a *set of nodes* and a *set of edges* that connect *pairs of nodes*.
- There is *exactly one* path between any two nodes of the tree.
- A ***path*** is a connected sequence of zero or more edges.

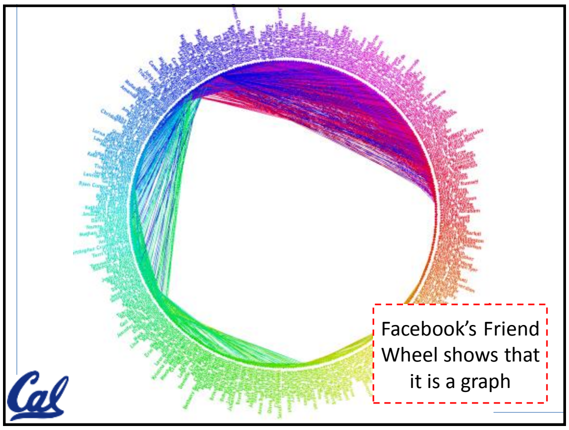


How Trees (grow-up) become Graphs

facebook

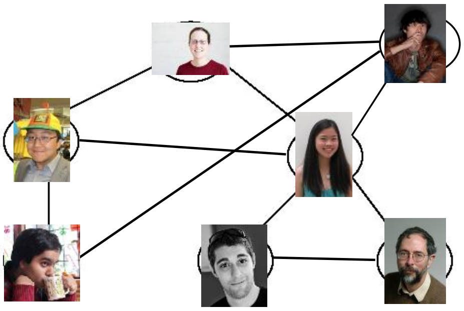


If facebook was a tree



Facebook's Friend Wheel shows that it is a graph

facebook Is a graph



Graph Vocab in facebook Terms

Graph Vocab in facebook Terms

- **Undirected Graph:**
  - If I am friends with Mike, Mike is friends with me.
- **Directed Graph:** (non-mutual friendships)
  - I can be friends with Kaushik even if he is not friends with me
  - Matt can be friends with me even if I am not friends with him.
  - I can be friends with Courtney AND Courtney can be friends with me.
- **Weighted Graph:**
  - I can give each friendship (link) a value based upon how good of friends we are.

Iterative Depth First Traversal

```
Stack<Vertex> fringe;
```

Put the root on the stack to be processed

Keep going till the fringe (Stack) is empty

Iterative Depth First Traversal

```
Stack<Vertex> fringe;
fringe = stack containing the root;
while (! fringe.isEmpty()) {
```

Get Something off the stack and do something with it

Iterative Depth First Traversal

```
Stack<Vertex> fringe;
fringe = stack containing the root;
while (! fringe.isEmpty()) {
     Vertex v = fringe.pop ();
```

```
Do something with the node ("visit")
   Add the neighbors to the stack
}
```

Iterative Depth First Traversal

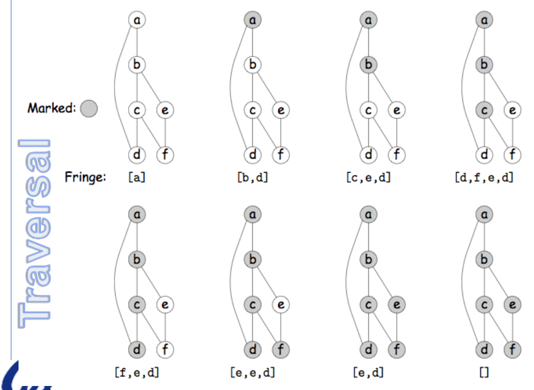
```
Stack<Vertex> fringe;
fringe = stack containing the root;
while (! fringe.isEmpty()) {
     Vertex v = fringe.pop ();
     mark(v);
     VISIT(v);
```

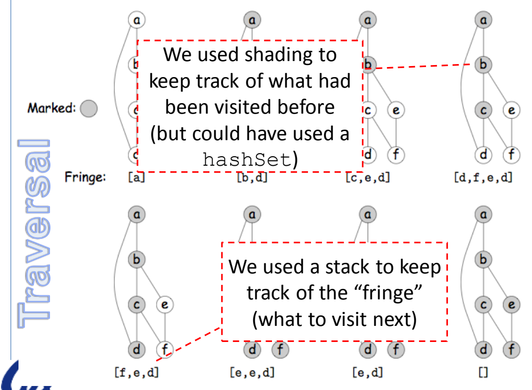
Deal with children

```
}
```

Iterative Depth First Traversal

```
Stack<Vertex> fringe;
fringe = stack containing the root;
while (! fringe.isEmpty()) {
     Vertex v = fringe.pop ();
     mark(v);
     VISIT(v);
     for each edge (v,w) {
          if (! marked (w))
               fringe.push (w);
     }
}
```





We used shading to keep track of what had been visited before (but could have used a `hashSet`)

We used a stack to keep track of the “fringe” (what to visit next)

Recursive Depth First Traversal

```
void traverse (Graph G) {
  For each vertex v in G {
          traverse (G, v);
  }
}
static void traverse (Graph G, vertex v) {
```

Call on EVERY vertex, so complete traversal even if the graph is not “connected”

Keep going till we can’t find anything that is unmarked

Recursive Depth First Traversal

```
void traverse (Graph G) {
  For each vertex v in G {
          traverse (G, v);
  }
}
static void traverse (Graph G, vertex v) {
  if (v is unmarked) {
```

Call on EVERY vertex, so complete traversal even if the graph is not “connected”

If we haven’t seen this node yet,
- `Do something with it`
- `Do recursion on neighbors`

Recursive Depth First Traversal

```
void traverse (Graph G) {
  For each vertex v in G {
          traverse (G, v);
  }
}
static void traverse (Graph G, vertex v) {
  if (v is unmarked) {
          mark(v);
          VISIT(v);
```

Call on EVERY vertex, so complete traversal even if the graph is not “connected”

Deal with the kids

Recursive Depth First Traversal

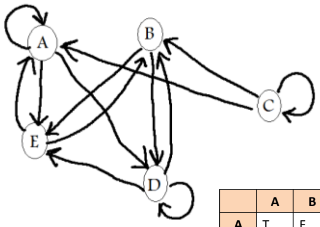
```
void traverse (Graph G) {
  For each vertex v in G {
          traverse (G, v);
  }
}
static void traverse (Graph G, vertex v) {
  if (v is unmarked) {
          mark(v);
          VISIT(v);
          For each edge (v,w) in G {
               traverse(G,w)
          }
  }
}
```

Call on EVERY vertex, so complete traversal even if the graph is not “connected”

Ways to Represent Graphs…

Adjacency Matrix
Adjacency List
Hash Table of Adjacent Pairs
Array of Adjacency Lists

Adjacency Matrix



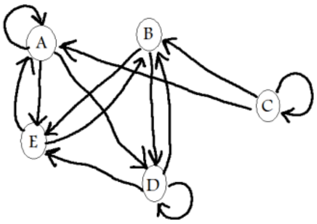
	A	B	C	D	E
A	T	F	F	T	T
B	F	F	F	T	T
C	T	T	T	F	F
D	F	T	F	T	T
E	T	T	F	F	F

Adjacency Matrix

- insert(u, v)
- remove(u, v)
- member (u, v)
- printEdges()

	A	B	C	D	E
A	T	F	F	T	T
B	F	F	F	T	T
C	T	T	T	F	F
D	F	T	F	T	T
E	T	T	F	F	F

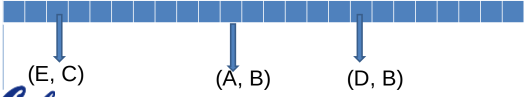
Edge List



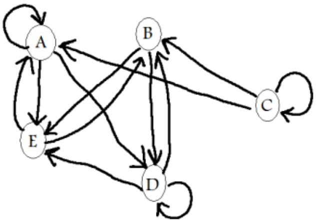
{(B D), (E B), (C A), (C C), (A A), (A E), (D B), (D D), (B E), (D E), (E A), (C B), (A D)}

HashSet of Adjacent Pairs

- insert(u, v)
- remove(u, v)
- member (u, v)
- printEdges()



Array of Adjacency Lists
(Array of Neighbors)
(Neighbors are stored in an unordered LL)



From	To
A	A, D, E
B	D, E
C	A, B, C
D	B, D, E
E	A, B

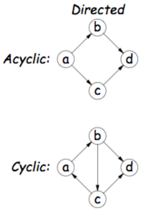
Array of Adjacency Lists
(Array of Neighbors)
(Neighbors are stored in an unordered LL)

From	To
A	A, D, E
B	D, E
C	A, B, C
D	B, D, E
E	A, B

- insert(u, v)
- remove(u, v)
- member (u, v)
- printEdges()

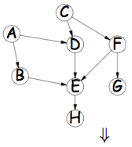
Cycles

- A *cycle* is a *path* without repeated edges leading from a vertex back itself.
- A graph is called ***cyclic*** if it has a a cycle, else *acyclic.*
- DAG: *directed acyclic graph*.



Topological Sort

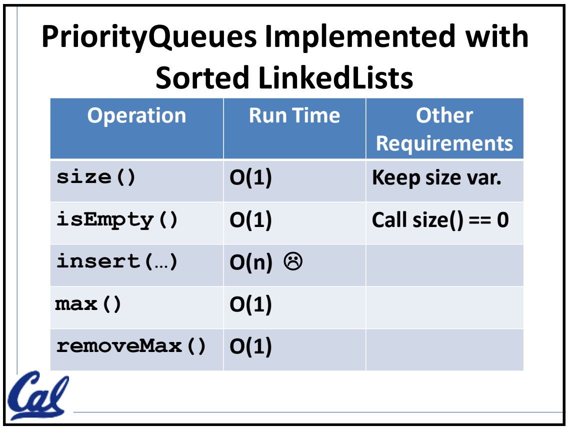
- Problem:
  - given a DAG, find a *linear ordering* of all the *vertices* that's consistent with the edges: if *(u,v)* is an edge, then *u* appears before *v* in the ordering.
- A topological sort of a DAG can be viewed as an *ordering* of its vertices along a horizontal line so that all directed edges go from *left to right.*



⇓

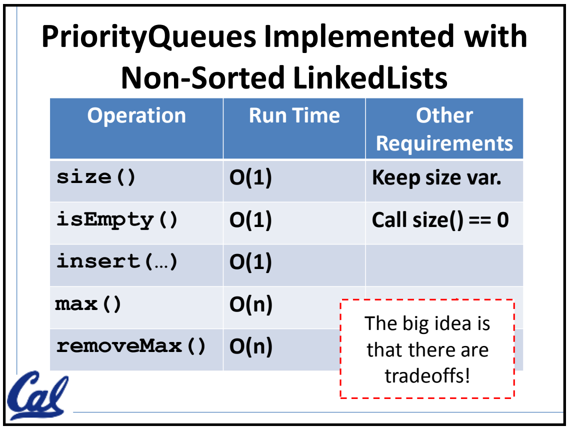
`[A,B,C,F,D,G,E,H]`, or
`[A,C,B,D,F,E,G,H]`, or
`[A,B,C,F,D,E,H,G]`, or
⋮

ANSWERS



PriorityQueues Implemented with Sorted LinkedLists

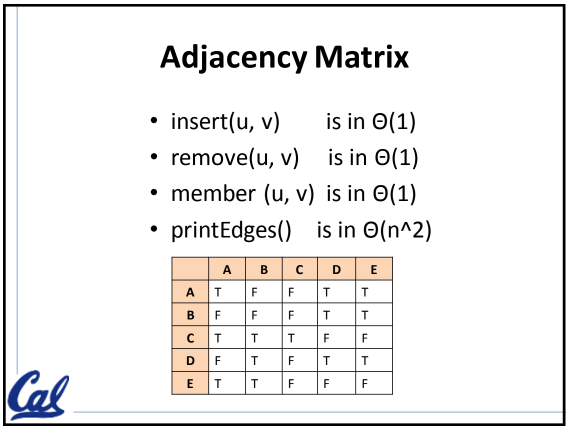
Operation	Run Time	Other Requirements
`size()`	O(1)	Keep size var.
`isEmpty()`	O(1)	Call size() == 0
`insert(…)`	O(n) ☹	
`max()`	O(1)	
`removeMax()`	O(1)	



PriorityQueues Implemented with Non-Sorted LinkedLists

Operation	Run Time	Other Requirements
`size()`	O(1)	Keep size var.
`isEmpty()`	O(1)	Call size() == 0
`insert(…)`	O(1)	
`max()`	O(n)	
`removeMax()`	O(n)	

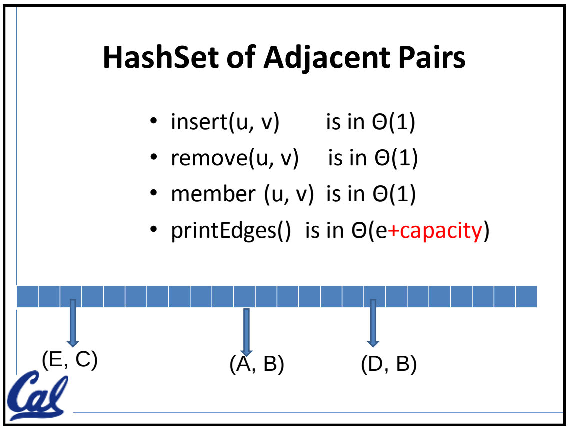
The big idea is that there are tradeoffs!



Adjacency Matrix

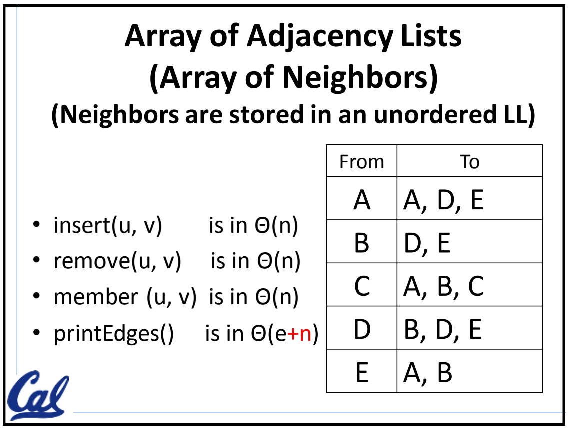
- insert(u, v) is in Θ(1)
- remove(u, v) is in Θ(1)
- member (u, v) is in Θ(1)
- printEdges() is in Θ(n^2)

	A	B	C	D	E
A	T	F	F	T	T
B	F	F	F	T	T
C	T	T	T	F	F
D	F	T	F	T	T
E	T	T	F	F	F



HashSet of Adjacent Pairs

- insert(u, v) is in Θ(1)
- remove(u, v) is in Θ(1)
- member (u, v) is in Θ(1)
- printEdges() is in Θ(e+capacity)



Array of Adjacency Lists (Array of Neighbors)

(Neighbors are stored in an unordered LL)

- insert(u, v) is in Θ(n)
- remove(u, v) is in Θ(n)
- member (u, v) is in Θ(n)
- printEdges() is in Θ(e+n)

From	To
A	A, D, E
B	D, E
C	A, B, C
D	B, D, E
E	A, B