CSE 332

Data Abstractions

Priority Queues & Heaps

T-REX LET'S SAY YOU HAVE A GIANT HEAP OF SAND AND I REMOVE ONE GRAIN OF IT AT A TIME
Ooh, let's!!
CLEARLY WHEN THERE'S ONLY ONE GRAIN OF SAND LEFT IT'S NOT A HEAP ANYMORE
Clearly!
AHA MY FRIEND BUT WHEN PRECISELY DID IT SWITCH FROM HEAP TO NON-HEAP
I dunno! At some fuzzy point it would switch for most observers from "heap" to, say, "small pile", and there we can draw the line. Language isn't that precise.
LISTEN THIS IS A CLASSIC PARADOX THAT EUBULIDES OF MILETUS CAME UP WITH OVER 2000 YEARS AGO
YOU NEED TO HAVE YOUR MIND BLOWN NOW OKAY
Sounds kinda dumb to me!
What does?
The point at which a shrinking heap of sand becomes a non-heap. Clearly I'm supposed to struggle with an arbitrary threshold, because piles on either side of it look much the same. But it's just language! Look at statistical usage of the word "heap", decide using that average, end of story.
Oh snap, philosophers! Did T-Rex just totally school you with his statistically-based descriptivist approach to semantics?
IT APPEARS THAT HE TOTALLY DID!!
It also appears he's speaking in the third person because he's so impressed with his awesome self!
(C) 2009 Ryan North
www.qwantz.com

Outline

~~Queue~~ FIFOQueue vs. PriorityQueue

The Queue we've seen thus far is a FIFO (First-In-First-Out) Queue:

Queue (FIFOQueue) ADT

enqueue(**val**)	Adds **val** to the queue.
dequeue()	Returns the **least-recent** item not already returned by a dequeue. (Errors if empty.)
peek()	Returns the **least-recent** item not already returned by a dequeue. (Errors if empty.)
isEmpty()	Returns true if all inserted elements have been returned by a dequeue.

But sometimes we're interested in a PriorityQueue instead:
That is, a Queue that prioritizes certain elements (e.g. a hospital ER).
Examples, in practice, include...

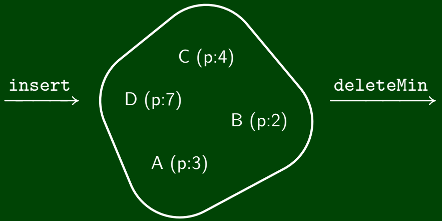
- OS Process Scheduling
- Sorting
- Compression (You did this already!)
- **Greedy** Algorithms (e.g. "shortest path")
- Discrete Event Simulation (priority = time step the event happens)

PriorityQueues!

PriorityQueue ADT

`insert(`**val**`)`	Adds **val** to the queue.
`deleteMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`findMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`isEmpty()`	Returns true if all inserted elements have been returned by a `deleteMin`.

- Data in `PriorityQueues` **must be comparable** (by priority)!
- Highest Priority = Lowest Priority Value
- The ADT **does not specify how to deal with ties**!



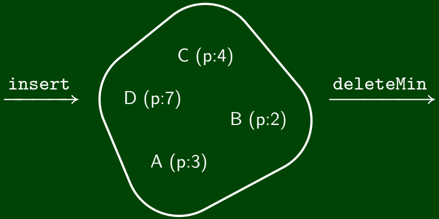
- `findMin`
- `deleteMin`
- `insert(E (p:1))`
- `deleteMin`
- `deleteMin`

PriorityQueues!

PriorityQueue ADT

`insert(val)`	Adds **val** to the queue.
`deleteMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`findMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`isEmpty()`	Returns true if all inserted elements have been returned by a `deleteMin`.

- Data in `PriorityQueues` **must be comparable** (by priority)!
- Highest Priority = Lowest Priority Value
- The ADT **does not specify how to deal with ties**!



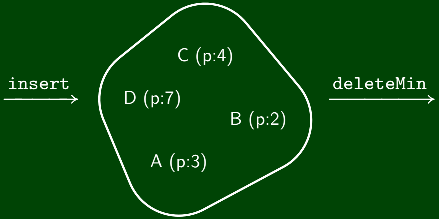
- `findMin` → B
- `deleteMin`
- `insert(E (p:1))`
- `deleteMin`
- `deleteMin`

PriorityQueues!

PriorityQueue ADT

`insert(`**val**`)`	Adds **val** to the queue.
`deleteMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`findMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`isEmpty()`	Returns true if all inserted elements have been returned by a `deleteMin`.

- Data in `PriorityQueues` **must be comparable** (by priority)!
- Highest Priority = Lowest Priority Value
- The ADT **does not specify how to deal with ties**!



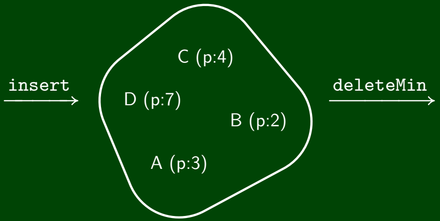
- `findMin` → B
- `deleteMin` → B
- `insert(E (p:1))`
- `deleteMin`
- `deleteMin`

PriorityQueues!

PriorityQueue ADT

insert(**val**)	Adds **val** to the queue.
deleteMin()	Returns the **highest priority** item not already returned by a deleteMin. (Errors if empty.)
findMin()	Returns the **highest priority** item not already returned by a deleteMin. (Errors if empty.)
isEmpty()	Returns true if all inserted elements have been returned by a deleteMin.

- Data in PriorityQueues **must be comparable** (by priority)!
- Highest Priority = Lowest Priority Value
- The ADT **does not specify how to deal with ties**!



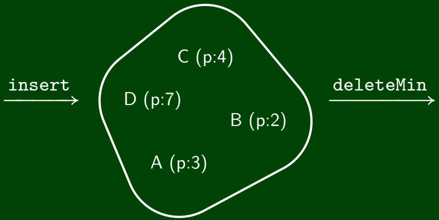
- findMin → B
- deleteMin → B
- insert(E (p:1))
- deleteMin → E
- deleteMin

PriorityQueues!

PriorityQueue ADT

`insert(val)`	Adds **val** to the queue.
`deleteMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`findMin()`	Returns the **highest priority** item not already returned by a `deleteMin`. (Errors if empty.)
`isEmpty()`	Returns true if all inserted elements have been returned by a `deleteMin`.

- Data in `PriorityQueues` **must be comparable** (by priority)!
- Highest Priority = Lowest Priority Value
- The ADT **does not specify how to deal with ties**!



- `findMin` → B
- `deleteMin` → B
- `insert(E (p:1))`
- `deleteMin` → E
- `deleteMin` → A

Implementing A Priority Queue

For each of the following potential implementations, what is the worst case runtime for `insert` and `deleteMin`? Assume all arrays do not need to resize.

- Unsorted Array
- Unsorted Linked List
- Sorted Circular Array List
- Sorted Linked List
- Binary Search Tree

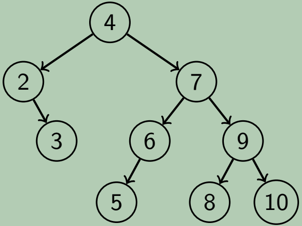
Implementing A Priority Queue

For each of the following potential implementations, what is the worst case runtime for `insert` and `deleteMin`? Assume all arrays do not need to resize.

- Unsorted Array
  Insert by inserting at the end which is $\mathcal{O}(1)$
  deleteMin by linear search which is $\mathcal{O}(n)$
- Unsorted Linked List
  Insert by inserting at the front which is $\mathcal{O}(1)$
  deleteMin by linear search which is $\mathcal{O}(n)$
- Sorted Circular Array List
  Insert by binary search; shifting elements which is $\mathcal{O}(n)$
  deleteMin by moving front which is $\mathcal{O}(1)$
- Sorted Linked List
  Insert by linear search which is $\mathcal{O}(n)$
  deleteMin by remove at front which is $\mathcal{O}(1)$
- Binary Search Tree
  Insert by search which is $\mathcal{O}(n)$
  deleteMin by findMin which is $\mathcal{O}(n)$

A New Data Structure: `Heap`

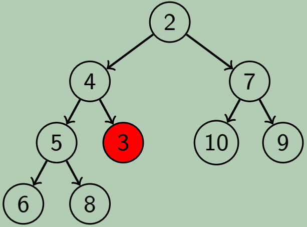
Recall BSTs



BST Property:
Left Children are smaller
Right Children are larger

For a `PriorityQueue`, how could we store the items in a tree?

And Now, `Heaps`

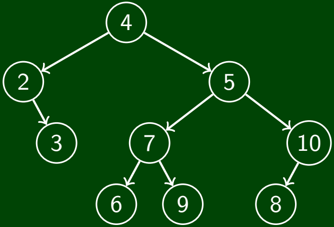


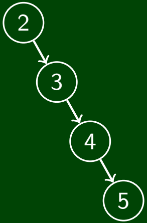
Heap Property:
All Children are larger

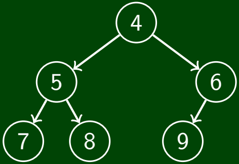
Structure Property:
Insist the tree has no “gaps”

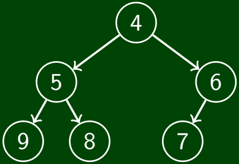
Is It A Heap?

For each of the following, is it a heap?



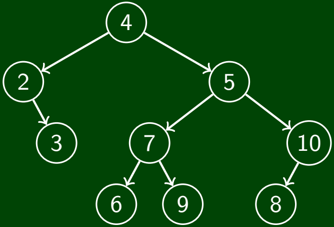


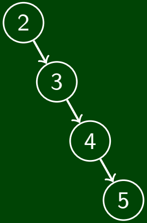




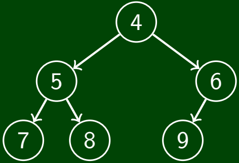
Is It A Heap?

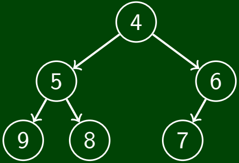
For each of the following, is it a heap?





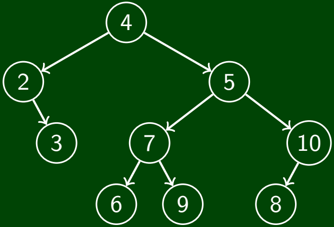
No, it fails both properties.



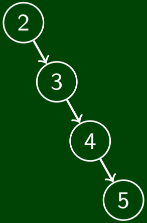


Is It A Heap?

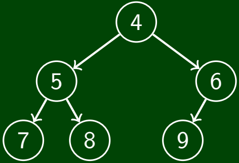
For each of the following, is it a heap?

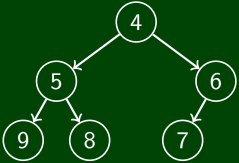


No, it fails both properties.



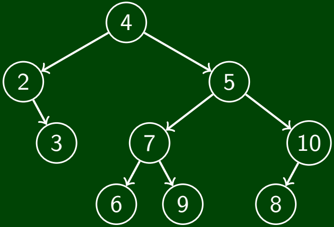
No, it fails the structure property. But ⑤ is.



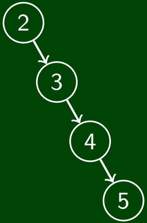


Is It A Heap?

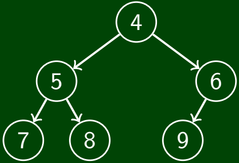
For each of the following, is it a heap?



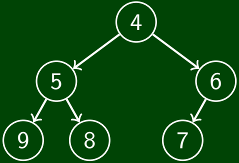
No, it fails both properties.



No, it fails the structure property. But $\textcircled{5}$ is.

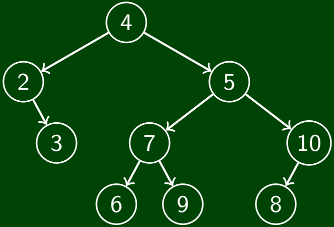


Yup! It's a heap.

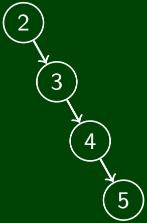


Is It A Heap?

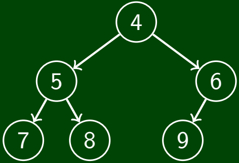
For each of the following, is it a heap?



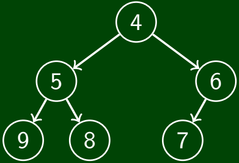
No, it fails both properties.



No, it fails the structure property. But is.

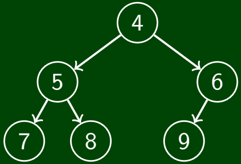


Yup! It's a heap.



Yup! It's a heap.

`Heap` Properties?



- Where is the minimum item in a heap?

  It's at the top!

- What is the height of a heap with n items?

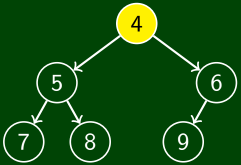
  Suppose that there are h levels in the heap.

  Then, $n \approx \sum_{i=0}^{h-1} 2^i = 2^h - 1$. So, $\lg n \approx \lg(2^h - 1) \approx \lg(2^h) = h$. So, $h \approx \lfloor \lg n \rfloor$.

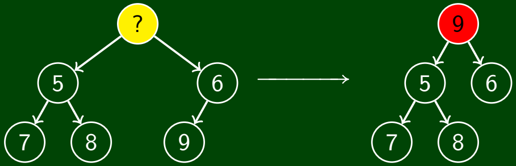
- How do we implement a `PriorityQueue` as a `Heap`?
  `findMin` is easy, but . . . `deleteMin`? `insert`?

Implementing `deleteMin` For a `Heap`

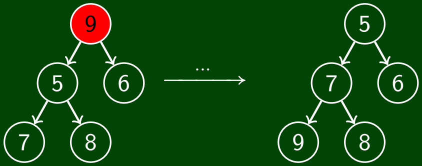
- Find the `min`:



- Remove the `min` and fill the hole with the last child

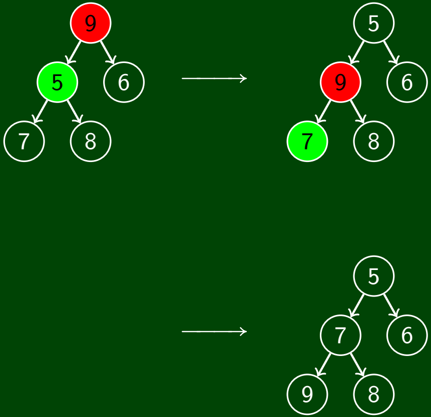


- "Percolate Down" to fix the invariant:



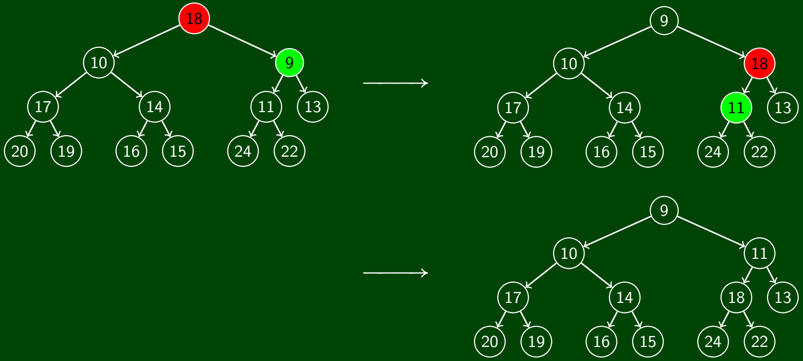
“Percolate Down”?

```
percolateDown(node) {
   while (node.data is greater than either child) {
      swap data with smaller child
   }
}
```



“Percolate Down” (Another Example)

```
percolateDown(node) {
   **while** (node.data is greater than **either** child) {
      swap data with **smaller** child
   }
}
```



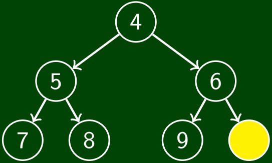
Runtime Analysis?

The height of the heap is $\lfloor \lg n \rfloor$. So, the runtime is $\mathcal{O}(\lg n)$.

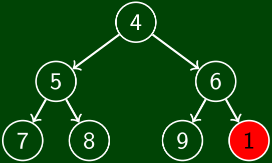
Implementing `insert` For a `Heap`

Let's try `insert(1)`:

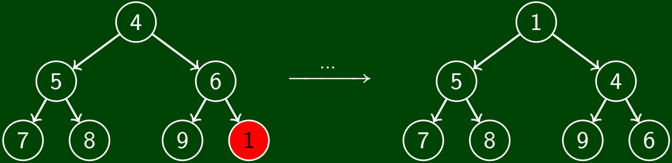
- Where do we put a new item?



- Fill our new hole with 1:

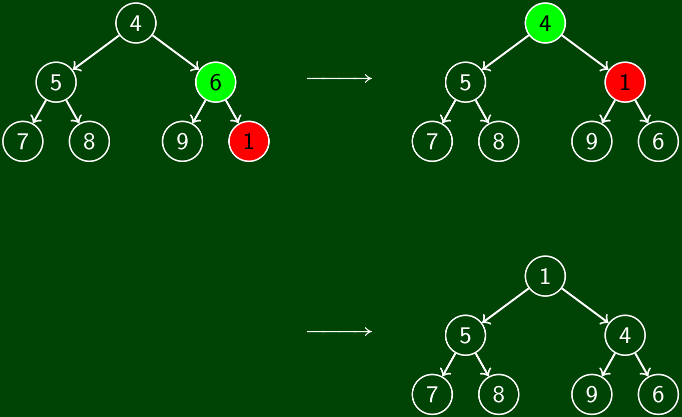


- "Percolate Up" to fix the invariant:



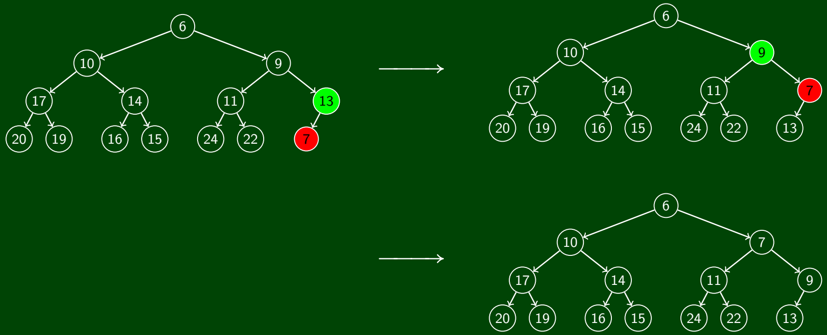
“Percolate Up”?

```
percolateUp(node) {
   while (node.data is smaller than parent) {
      swap data with parent
   }
}
```



“Percolate Up” (Another Example)

```
percolateUp(node) {
   while (node.data is smaller than parent) {
      swap data with parent
   }
}
```

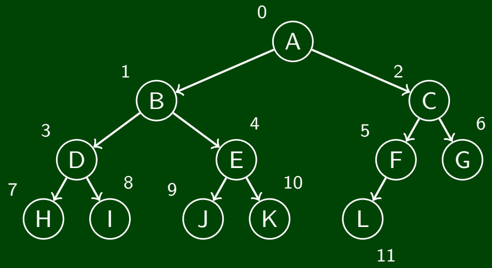


Runtime Analysis?

The height of the heap is $\lfloor \lg n \rfloor$. So, the runtime is $\mathcal{O}(\lg n)$.

And...how do we implement `Heap`?

We've insisted that the tree be complete to be a valid `Heap`. Why?



Fill in an array in **level-order** of the tree:

heap:	h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]	h[9]	h[10]	h[11]	h[12]	h[13]	h[14]
	A	B	C	D	E	F	G	H	I	J	K	L	0	0	0

If I have the node at index i, how do I get its:

- Parent? $3 \to 1,\ 4 \to 1,\ 10 \to 4,\ 9 \to 4,\ 1 \to 0$

  This indicates that it's approximately $n/2$. In fact, it's $\frac{n-1}{2}$.
- Left Child? $2(n+1)-1$
- Right Child? $2(n+1)$