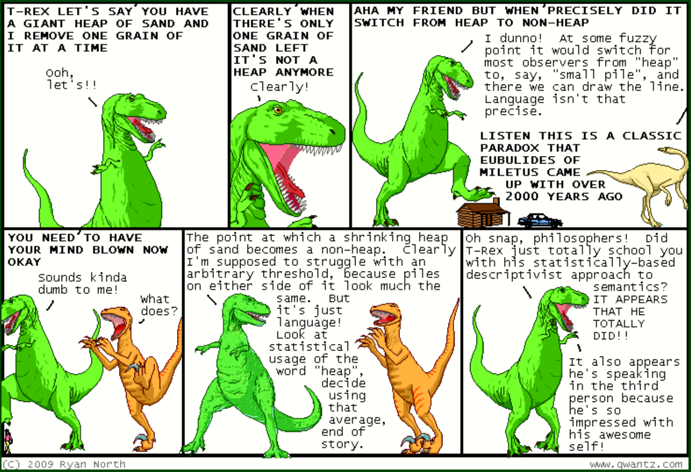
CSE 332

Data Abstractions

Priority Queues & Heaps



Outline

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...

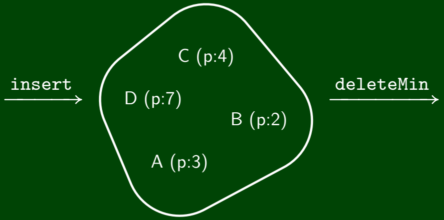
- OS Process Scheduling
- Sorting
- Compression (You did this already!)
- **Greedy** Algorithms (e.g. "shortest path")
- Discrete Event Simulation (priority = time step the event happens)

PriorityQueues!

Queue (FIFOQueue) 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**!

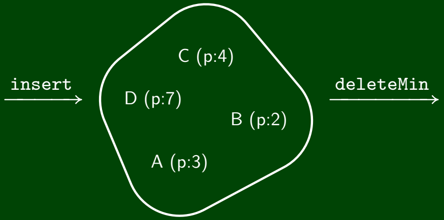


- findMin
- removeMin
- insert(E (p:1))
- removeMin
- removeMin

`Queue (FIFOQueue)` 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**!

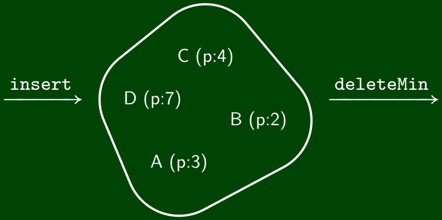


- `findMin` → B
- `removeMin`
- `insert(E (p:1))`
- `removeMin`
- `removeMin`

Queue (FIFOQueue) 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**!

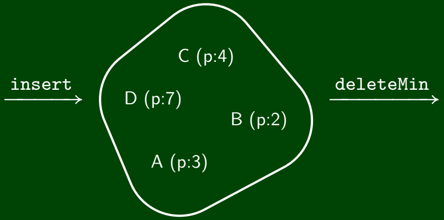


- `findMin` → B
- `removeMin` → B
- `insert(E (p:1))`
- `removeMin`
- `removeMin`

Queue (FIFOQueue) 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**!

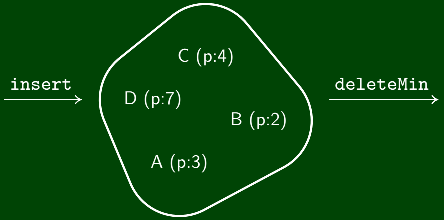


- `findMin` → B
- `removeMin` → B
- `insert(E (p:1))`
- `removeMin` → E
- `removeMin`

Queue (FIFOQueue) 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
- `removeMin` → B
- `insert(E (p:1))`
- `removeMin` → E
- `removeMin` → 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.

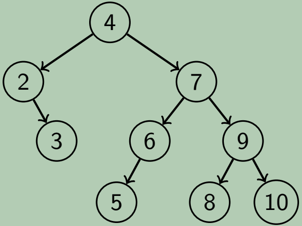
- Unsorted Array
- Unsorted Linked List
- Sorted Circular Array List
- Sorted Linked List
- Binary Search Tree

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`

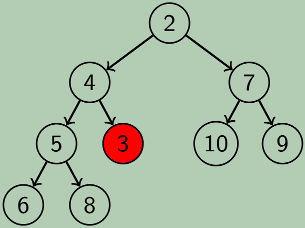
Recall BSTs



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`

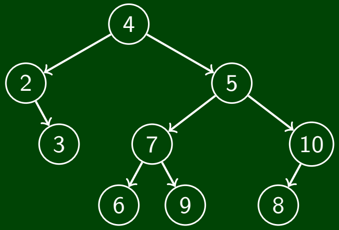


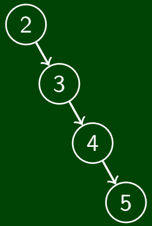
Heap Property:
All Children are larger

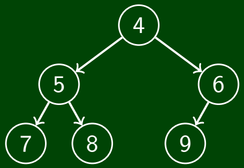
Structure Property:
Insist the tree has no “gaps”

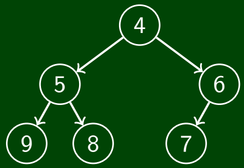
Is It A Heap?

For each of the following, is it a heap?



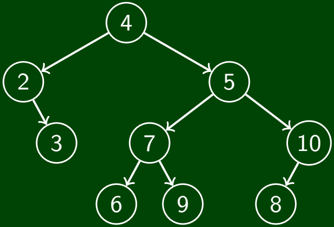


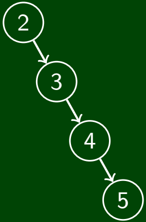




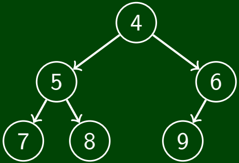
Is It A Heap?

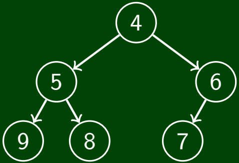
For each of the following, is it a heap?





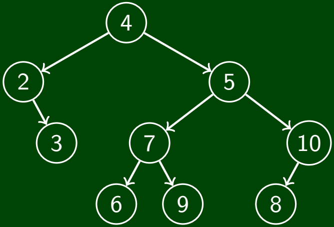
No, it fails the structure property. But if we replace the 6 with anything larger and take the piece from 5 down, it is.

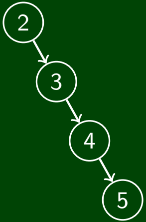




Is It A Heap?

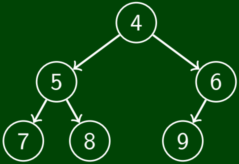
For each of the following, is it a heap?

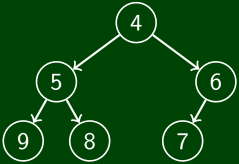




No, it fails the structure property. But if we replace the 6 with anything larger and take the piece from 5 down, it is.

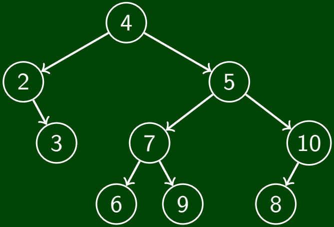
No, it fails the structure property. But 5 is.



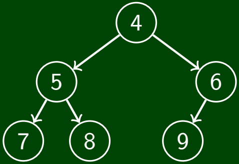


Is It A Heap?

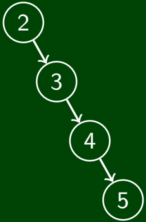
For each of the following, is it a heap?



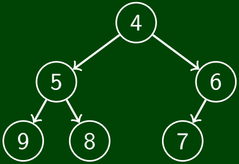
No, it fails the structure property. But if we replace the 6 with anything larger and take the piece from 5 down, it is.



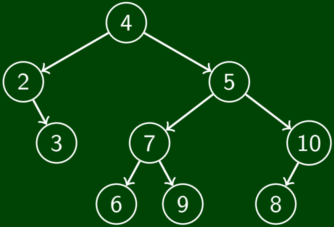
Yup! It's a heap.



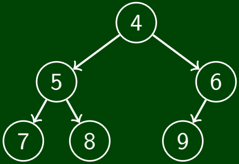
No, it fails the structure property. But 5 is.



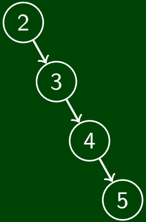
For each of the following, is it a heap?



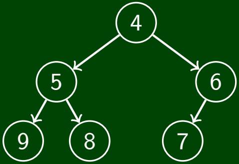
No, it fails the structure property. But if we replace the 6 with anything larger and take the piece from 5 down, it is.



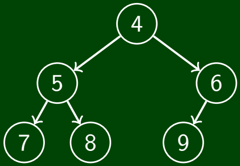
Yup! It's a heap.



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



Yup! It's a heap.



- Where is the minimum item in a heap?

  It's at the top!

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

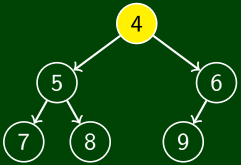
  Note that all but the last row is full. So, $n \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$. So, $\lg n \leq k$.

  So, $k = \lfloor \lg n \rfloor$.

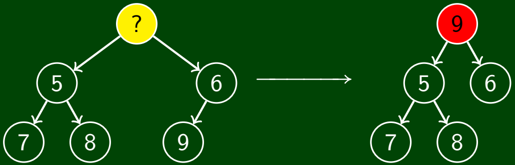
- How do we implement a `PriorityQueue` as a `Heap`?
  `findMin` is easy, but ... `deleteMin`? `insert`?

Implementing `deleteMin` For a `Heap`

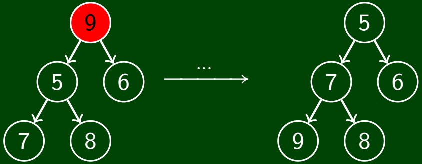
- Find the `min`:



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

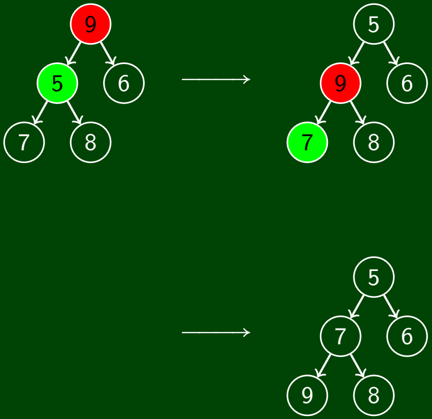


- "Percolate Down" to fix the invariant:



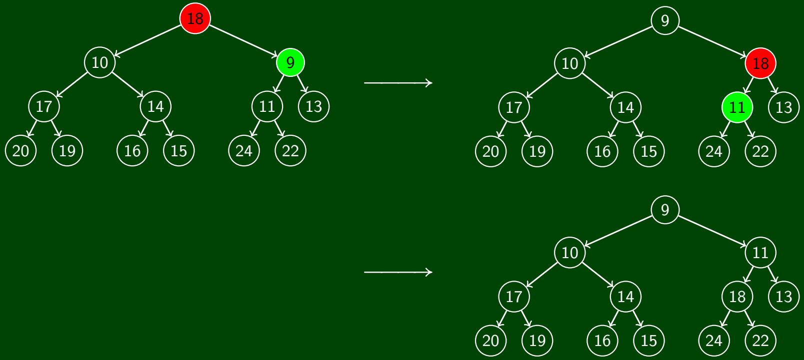
“Percolate Down”?


```
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
   }
}
```



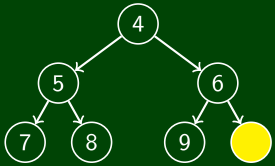
Runtime Analysis?

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

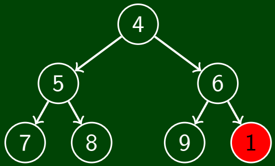
Implementing `insert` For a `Heap`

Let's try `insert(1)`:

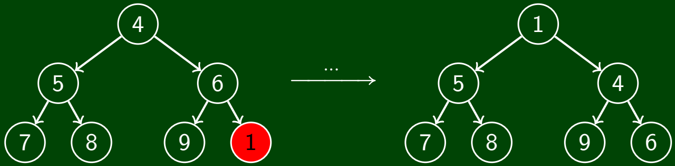
- Where do we put a new item?



- Fill our new hole with 1:

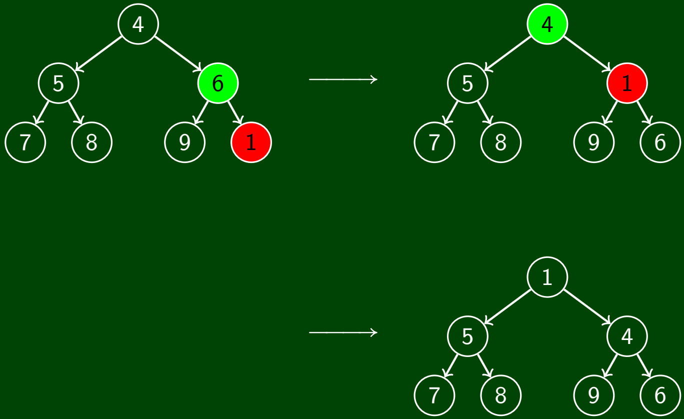


- "Percolate Up" to fix the invariant:



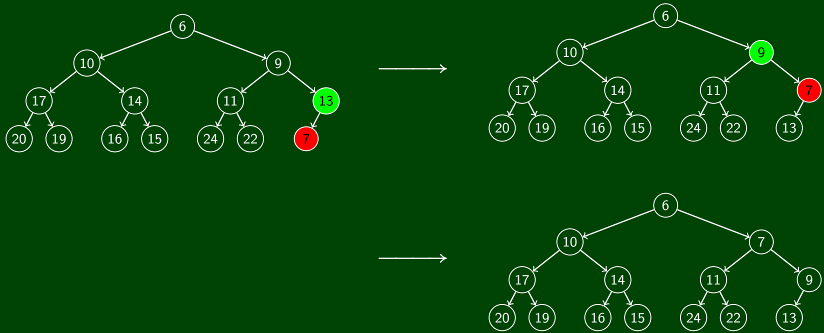
“Percolate Up”?


```
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
   }
}
```

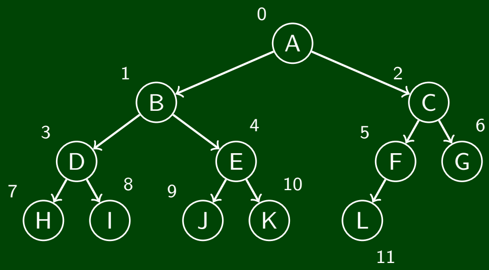


Runtime Analysis?

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:	A	B	C	D	E	F	G	H	I	J	K	L	0	0	0
	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]

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 $\dfrac{n-1}{2}$.
- Left Child? $2(n+1)-1$
- Right Child? $2(n+1)$