

CSE 332

Data Abstractions

Priority Queues & Heaps



Outline

1 PriorityQueues

2 Heaps

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

Queue (FIFOQueue) ADT

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

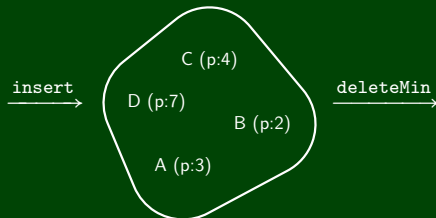
But sometimes we're interested in a `PriorityQueue` instead:
That is, a Queue that prioritizes certain elements (e.g. a hospital ER).
Examples, in practice, include...

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

PriorityQueue ADT

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

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

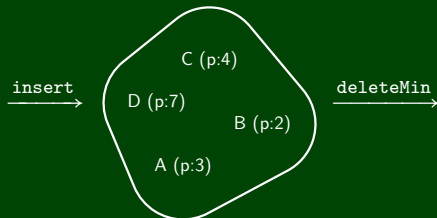


- `findMin`
- `deleteMin`
- `insert(E (p:1))`
- `deleteMin`
- `deleteMin`

PriorityQueue ADT

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

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

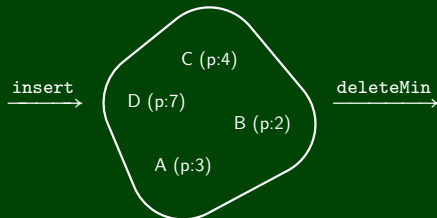


- `findMin` → B
- `deleteMin`
- `insert(E (p:1))`
- `deleteMin`
- `deleteMin`

PriorityQueue ADT

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

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

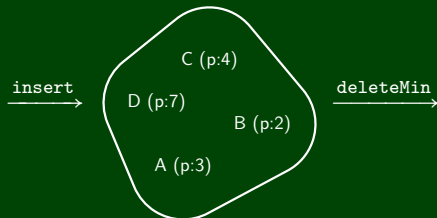


- `findMin` → B
- `deleteMin` → B
- `insert(E (p:1))`
- `deleteMin`
- `deleteMin`

PriorityQueue ADT

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

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

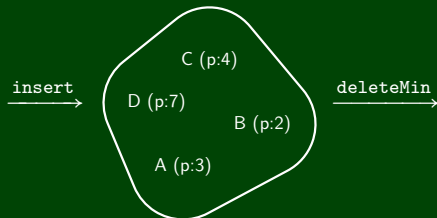


- `findMin` → B
- `deleteMin` → B
- `insert(E (p:1))`
- `deleteMin` → E
- `deleteMin`

PriorityQueue ADT

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

- 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

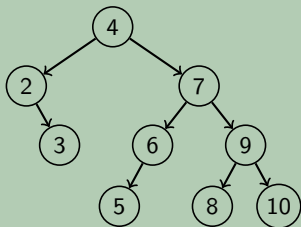
For each of the following potential implementations, what is the worst case runtime for `insert` and `deleteMin`? Assume all arrays do not need to resize.

- 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 $\mathcal{O}(n)$

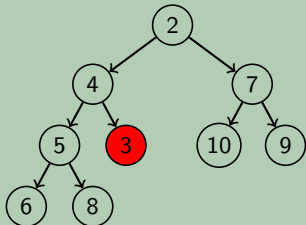
Recall BSTs



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

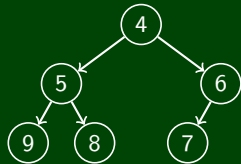
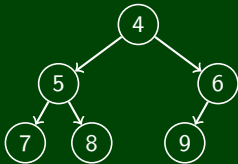
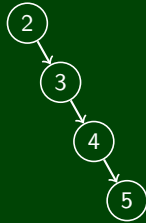
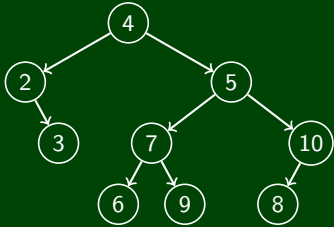


Heap Property:
All Children are larger

Structure Property:
Insist the tree has no “gaps”

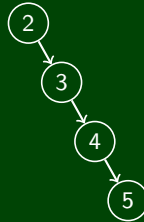
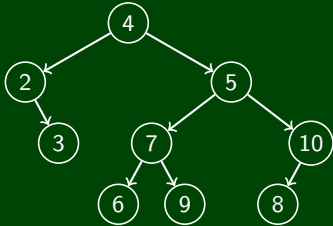
Is It A Heap?

For each of the following, is it a heap?

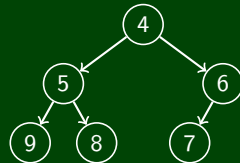
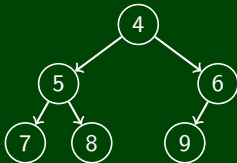


Is It A Heap?

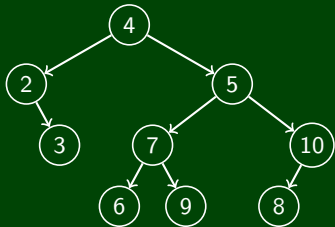
For each of the following, is it a heap?



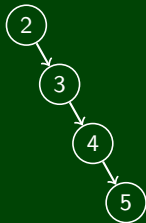
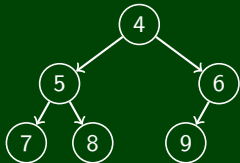
No, it fails both properties.



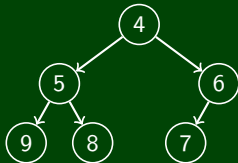
For each of the following, is it a heap?



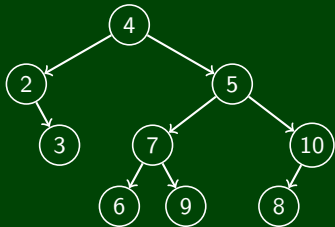
No, it fails both properties.



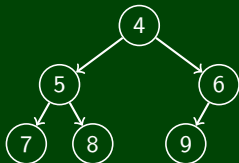
No, it fails the structure property. But **5** is.



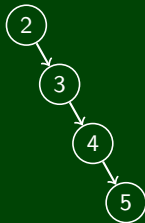
For each of the following, is it a heap?



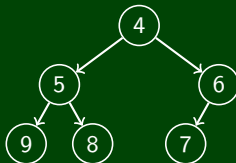
No, it fails both properties.



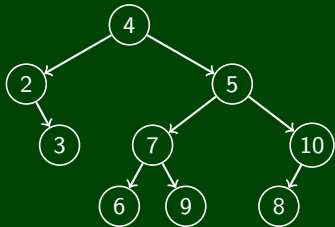
Yup! It's a heap.



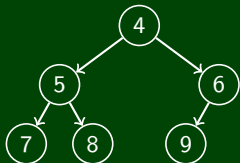
No, it fails the structure property. But **5** is.



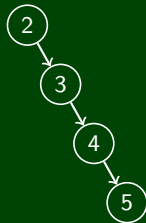
For each of the following, is it a heap?



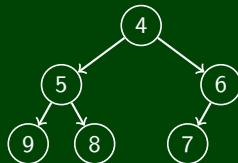
No, it fails both properties.



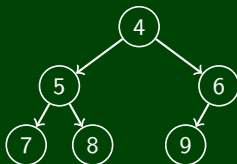
Yup! It's a heap.



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



Yup! It's a heap.



- Where is the minimum item in a heap?

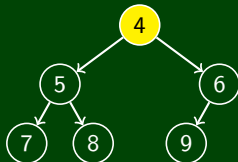
It's at the top!

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

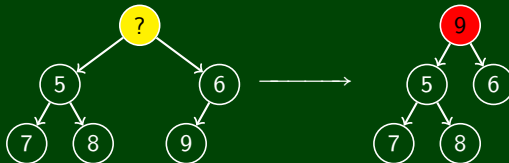
Note that all but the last row is full. So, $n \leq \sum_{i=0}^k 2^i = 2^{k+1} - 1$. So, $\lg n \leq k + 1 - 1$. So, $k = \lceil \lg n \rceil$.

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

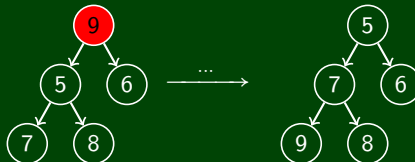
- Find the min:



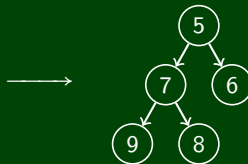
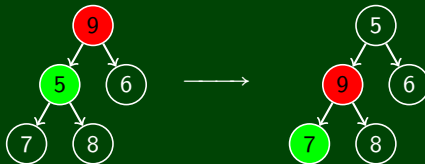
- Remove the min and fill the hole with the last child



- “Percolate Down” to fix the invariant:

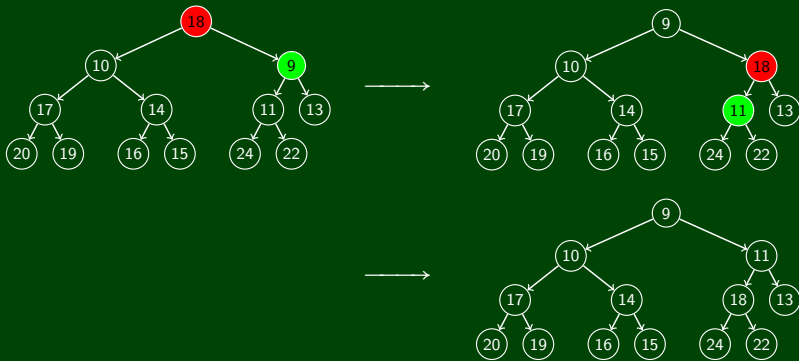


```
1 percolateDown(node) {  
2   while (node.data is greater than either child) {  
3     swap data with smaller child  
4   }  
5 }
```



“Percolate Down” (Another Example)

```
1 percolateDown(node) {  
2   while (node.data is greater than either child) {  
3     swap data with smaller child  
4   }  
5 }
```

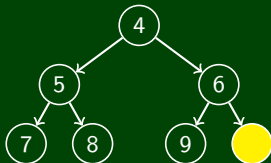


Runtime Analysis?

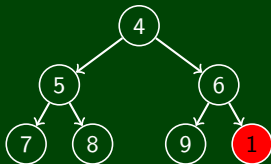
The height of the heap is $\lceil \lg n \rceil$. So, the runtime is $\mathcal{O}(\lg n)$.

Let's try insert(1):

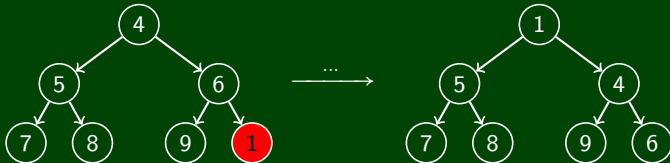
- Where do we put a new item?



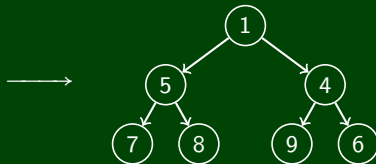
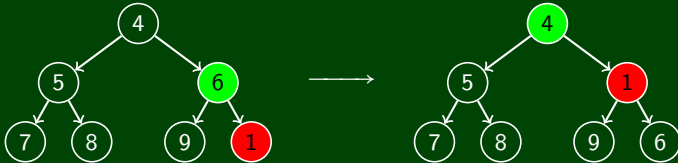
- Fill our new hole with 1:



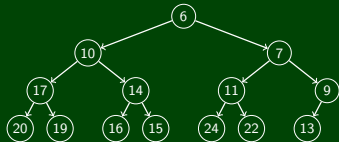
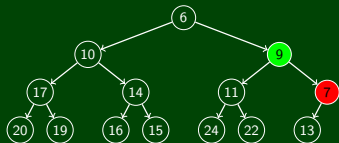
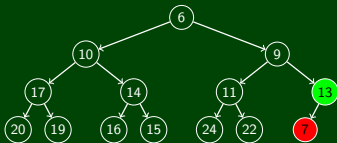
- “Percolate Up” to fix the invariant:



```
1 percolateUp(node) {  
2   while (node.data is smaller than parent) {  
3     swap data with parent  
4   }  
5 }
```



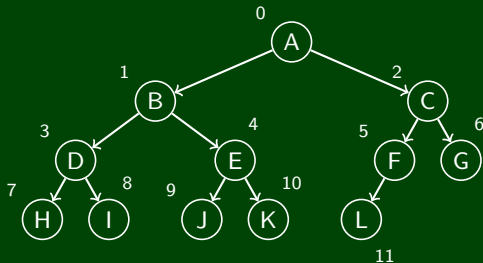
```
1 percolateUp(node) {  
2   while (node.data is smaller than parent) {  
3     swap data with parent  
4   }  
5 }
```



Runtime Analysis?

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

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? $3 \rightarrow 1, 4 \rightarrow 1, 10 \rightarrow 4, 9 \rightarrow 4, 1 \rightarrow 0$

This indicates that it's approximately $n/2$. In fact, it's $\frac{n-1}{2}$.

- Left Child? $2(n+1) - 1$
- Right Child? $2(n+1)$