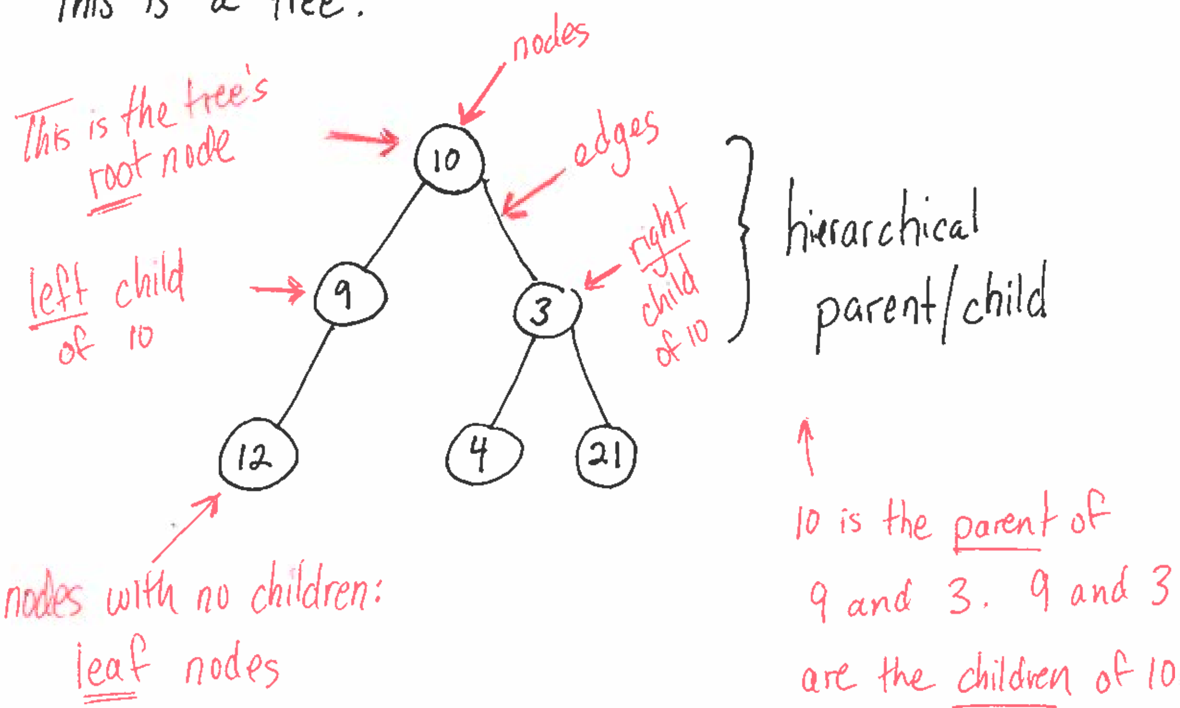
Today's topic: Heaps

We'll focus on minheaps, which form the foundation for priority queues. Analogous data structure: maxheap.

Preliminaries: This is a tree:

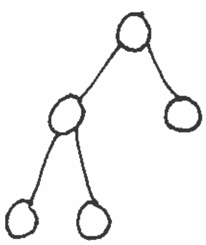


This is a binary tree. Every node has (at most) two children. Each child is either the left child or the right child of its parent.

Definition: A minheap is a binary tree with the following structural and ordering properties:

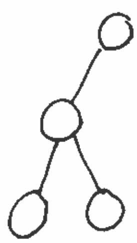
① structural property:
it's a "complete" binary tree: every level fills up from left to right with no gaps before moving on to the next level.

POP QUIZ! Are the following trees complete binary trees?



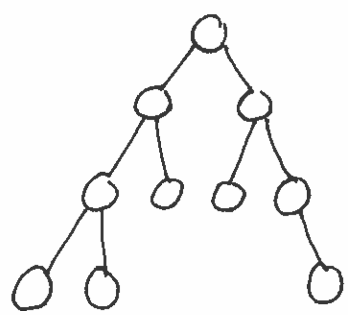
yes!

Nope! We did not fill that final level up from left to right. We left a gap!



Nope! We did not fill up second level before moving down to third level.

yes!

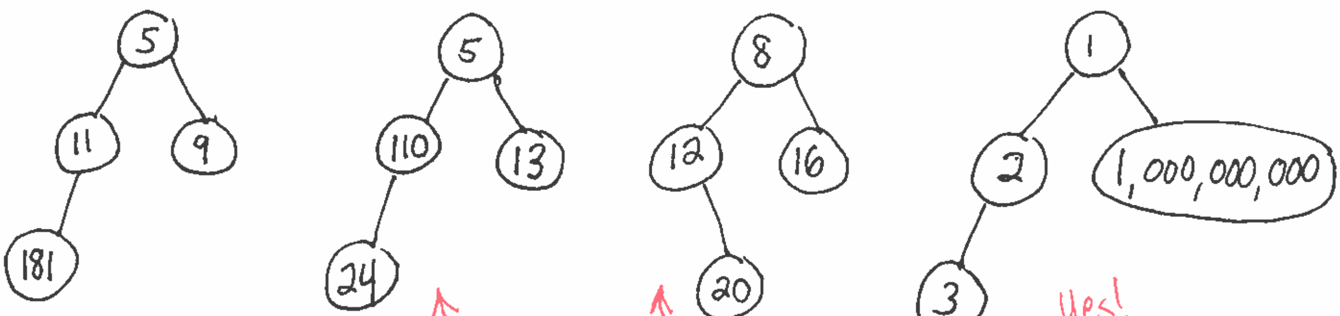


Nope! We didn't fill the bottom level from left to right without leaving gaps.

② ordering property:

every node's value ("priority") must be less than or equal to the values of any/all of its children

POP QUIZ! Are the following trees valid minheaps?



Yes!

No! 110 is larger than its child, 24.

No! The ordering property is satisfied, but not the structural property. (It is ~~not~~ complete.)

Yes!

*Side note (not particularly important for this class): Statements about the elements of an empty set are said to be "vacuously true."

Minheap operations:

	Best-Case Runtime		Worst-Case Runtime	
insertion (enqueue)	$O(1)$	too large to percolate up at all!	$O(\log n)$	percolates up entire height of tree!
deleteMin (dequeue)	$O(1)$	see following page for example	$O(\log n)$	percolate all the way down entire height of tree
findMin / getMin (peek)	$O(1)$	just return value at root!	$O(1)$	

... and other standard fare: size(), isEmpty(), clear(), ...

* Minheaps only support deletion of the <u>minimum</u> value in the heap, not of arbitrary values. Same for the find/search/get operation.

(*) The runtimes on this page were presented <u>after</u> exploring those operations on subsequent pages of these notes.

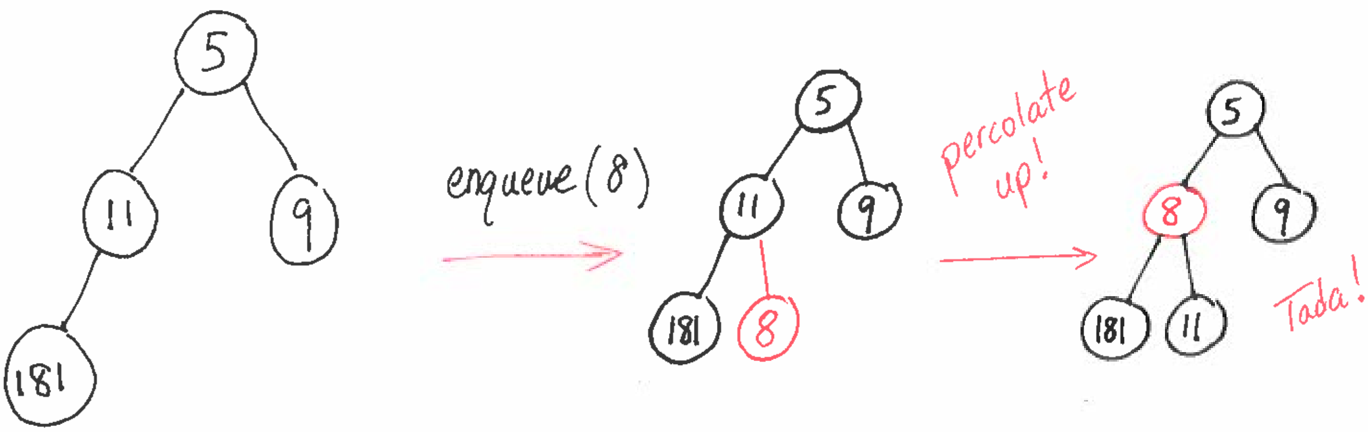
Note:

The $O(\log n)$ worst-case insertion runtime assumes we are not triggering the expansion of the heap's underlying array representation, which would actually be an $O(n)$ operation.

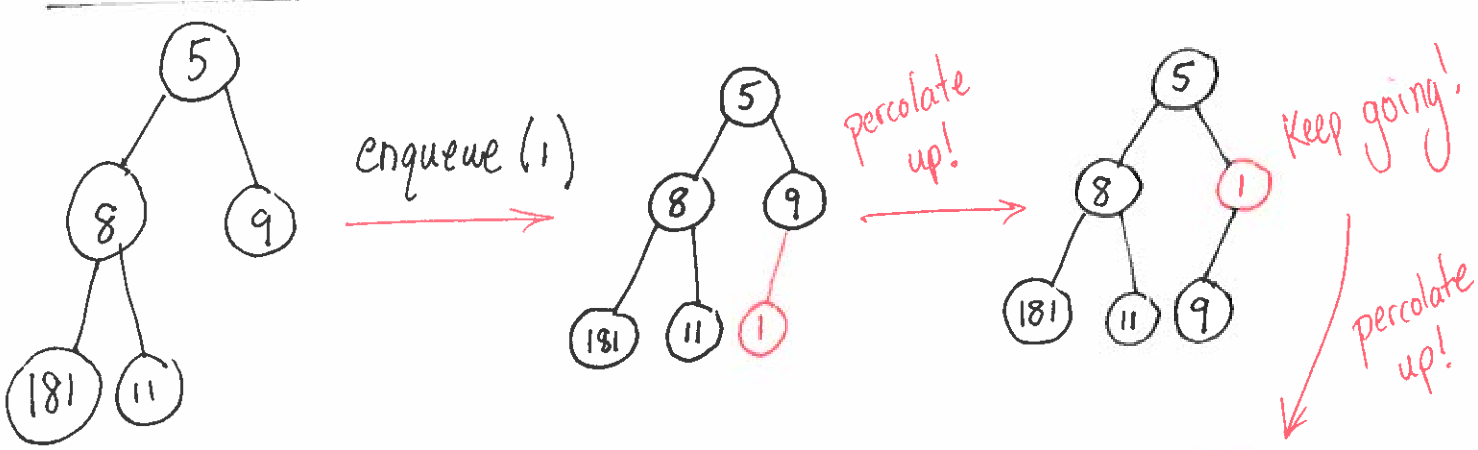
Here's how insertion works:

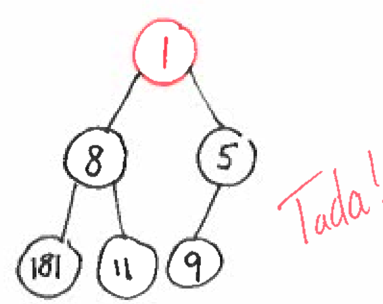
1. Insert at the next open position.
2. Percolate up! ("sift up" / "bubble up")

Example:



Another Example:





Best case scenario?

$O(1)$ — New value too large to percolate up.

Worst case scenario?

$O(\log n)$ — New value percolates up entire tree.
(See derivation of tree height on following page.)

How tall is a minheap?

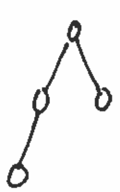
This matters because the worst-case runtime for percolateUp() — and therefore enqueue() — is $O(\text{height})$.

* Height is # edges from root to a farthest leaf node.

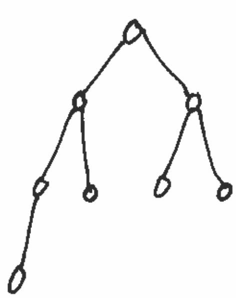
n = 1, height = 0

n = 2, height = 1

n = 3, height = 1



n = 4, height = 2



n = 8, height = 3

	n	height
	1	0
$2^1 =$	2	1
	3	1
$2^2 =$	4	2
	5	2
	6	2
	7	2
$2^3 =$	8	3
	...	...
$2^4 =$	16	4

for n that are powers of 2, we have: $n = 2^{\text{height}}$

So: $\log_2 n = \log_2 2^{\text{height}}$

Take the floor (which performs integer truncation) to deal with n not a power of 2. → $\lfloor \log_2 n \rfloor = \text{height}$

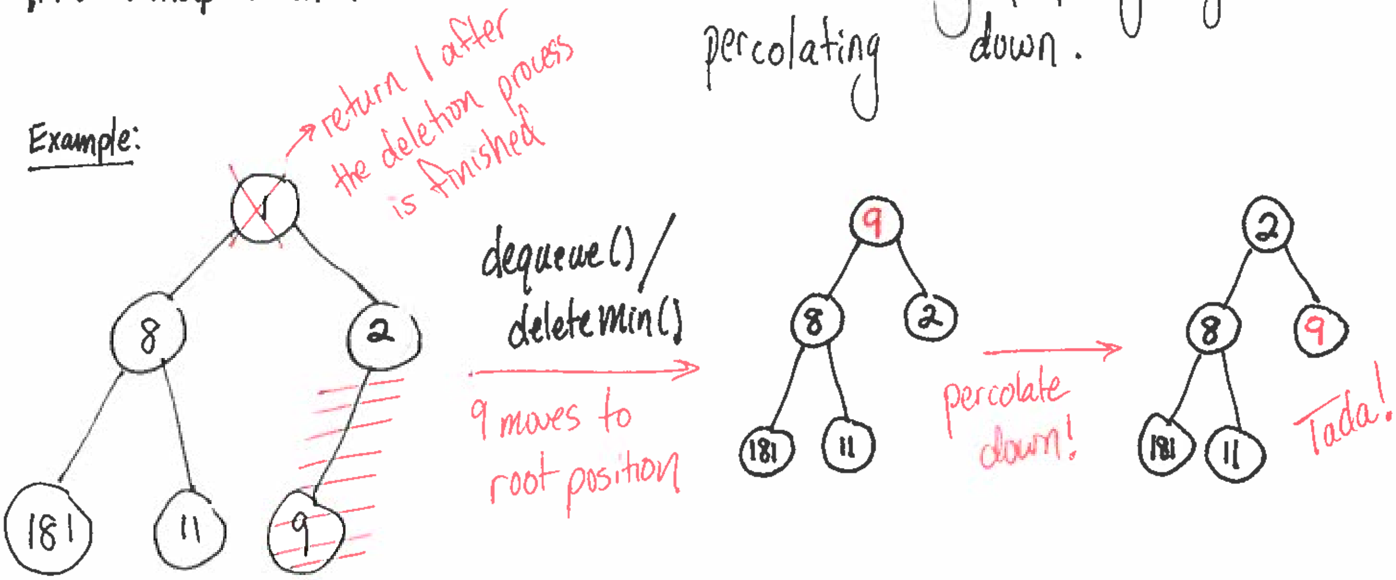
Tada!

Here's how deletion works:

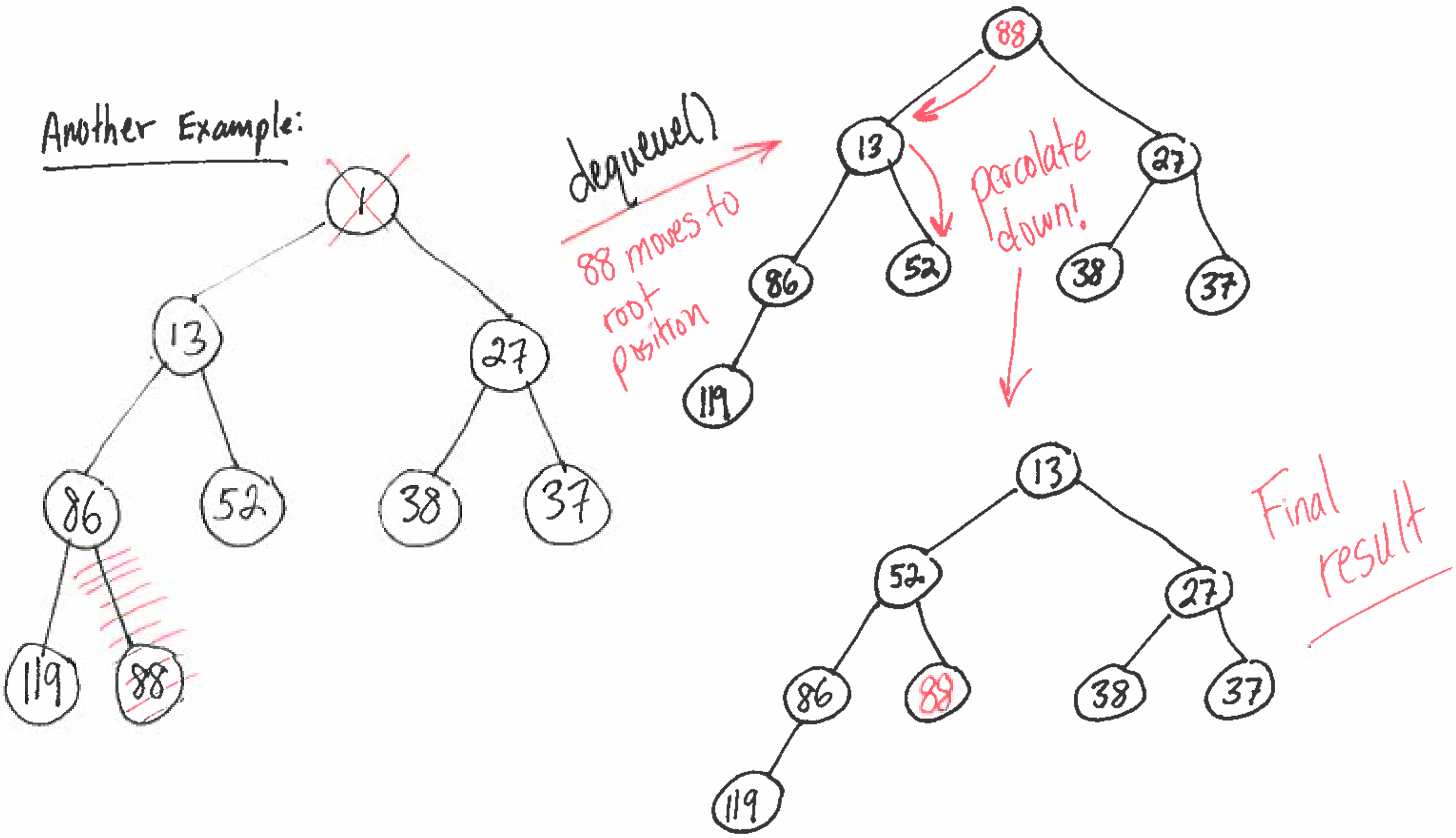
(Note: We will see shortly that we can get to the bottom-right node in a minheap in O(1) time.)

1. Preserve the structural property.
   The last value (bottom-right) moves up to root position.
2. Restore ordering property by percolating down.

Example:



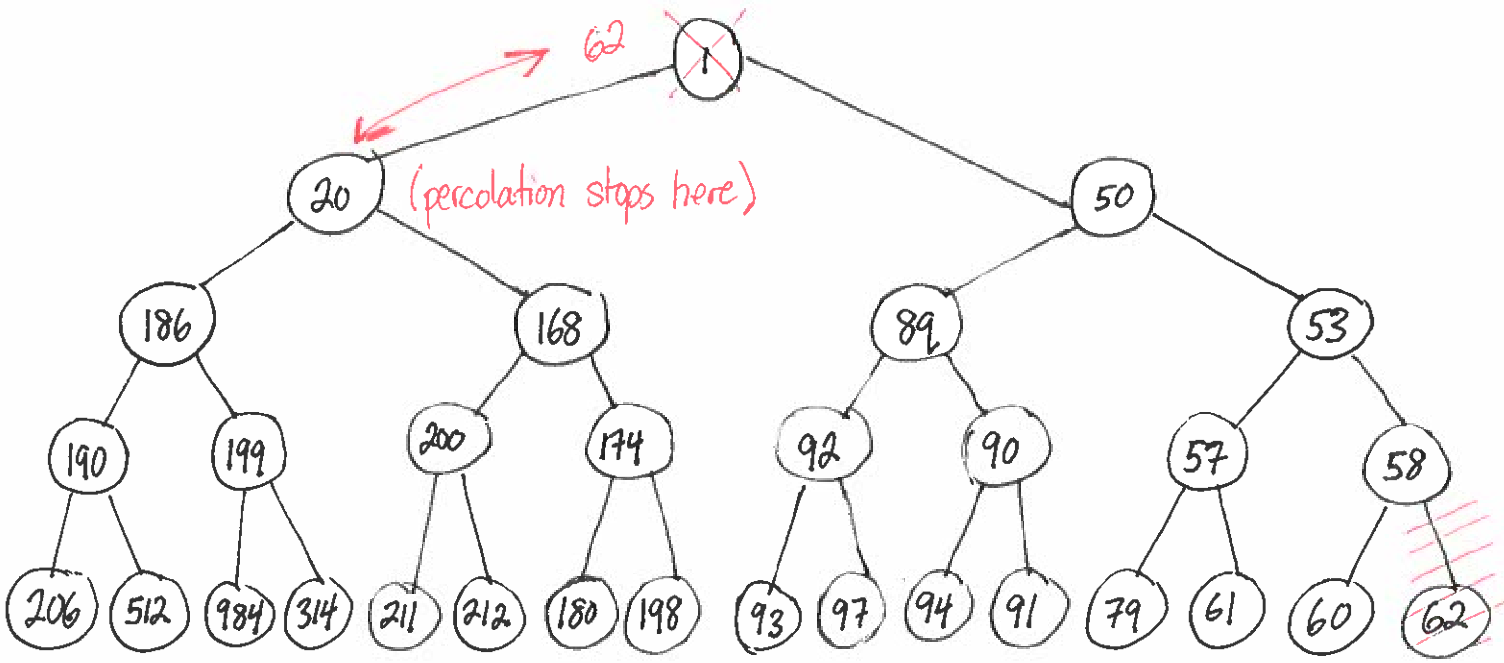
Another Example:



Best Case Scenario? See next page!

Worst case scenario? O(log n) — percolate all the way down the tree!

Example of best-case runtime for dequeue():



In this minheap, if we dequeue, 62 moves to the root, percolates down to the left, then stop because it is less than both 186 and 168. Regardless of the height of the tree, it is possible to encounter a situation like this that only involves a single swap in the percolation. This is an O(1) deletion!

(Similarly, if all the values were <u>equal</u>, percolateDown() would not perform <u>any</u> swaps.)

An application: HeapSort!

Worst-case Runtime:

Step 1: Insert n elements into minheap — $O(n \log n)$

Step 2: Remove all elements from minheap — $O(n \log n)$

Place them in a vector as they come out. They're coming out in sorted order!

Total: $O(n \log n)$

Pretty awesome!

Also: Priority Queues!

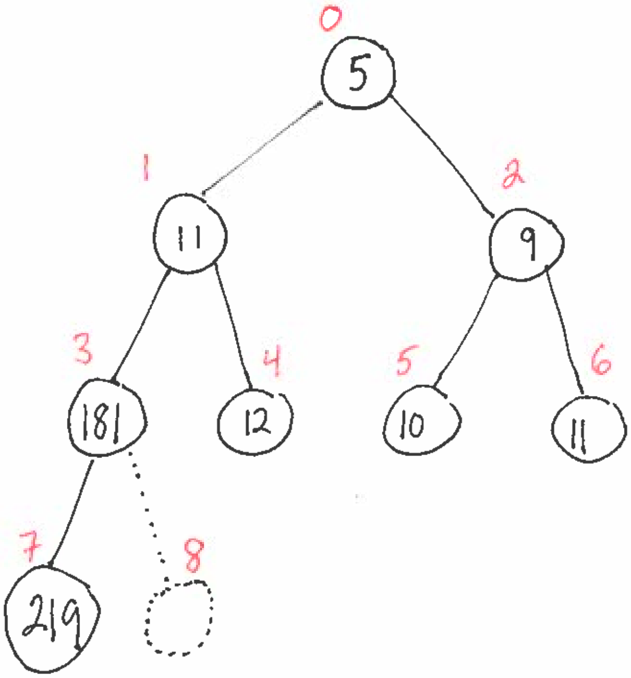
Bundle some data up with a priority that determines where it ends up in the queue!

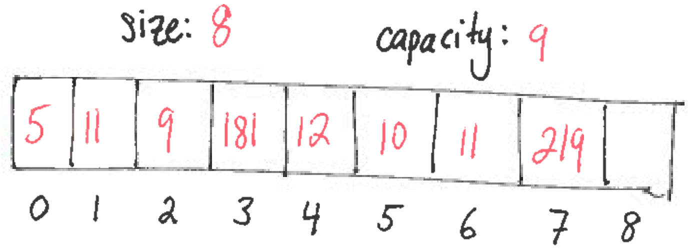
Examples: Emergency room triage
Printer queues

imagine a struct or class where we have a priority field to indicate how severe someone's situation is and a HealthRecord field with their personal information. We would use the priority for percolation, and the health record would come along for the ride. (Lower "priority" = higher urgency.)

Priority could be # of pages to print (bundled up with print job data). A priority queue would process smaller jobs first rather than making everyone wait for someone who wants to print 500 pages.

Representation:





We use an array/vector to represent our minheap. This gives us O(1) access to the next available position (before calling percolateUp).

Formulas:

leftChild(i) = $2i+1$

rightChild(i) = $2i+2$

parent(i) = $(i-1)/2$ ← assuming integer truncation!

* Not having gaps in the various levels of our minheap allows us to maximize the utilization of our array! There are no gaps to cause wasted space as our data structure expands.

This also helps to enable our elegant and consistent numbering scheme and formulas.

Heapify! An algorithm for turning an arbitrary array / complete binary tree into a minheap in place (without creating a new array).

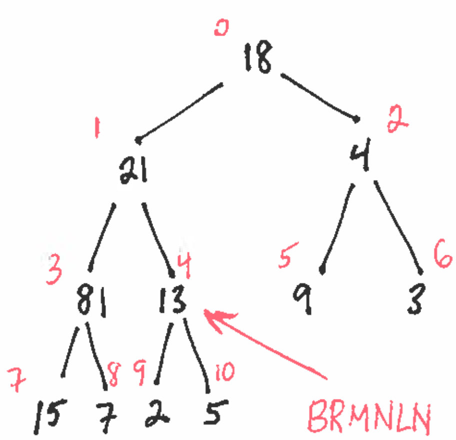
Let BRMNLN be the index of the bottom-rightmost non-leaf node in the complete binary tree represented by the given array.

Call percolateDown() starting at index BRMNLN and down through index 0 (the root):

```
for (int i = BRMNLN;  i >= 0;  i--)
    percolateDown(i);
```

This percolates larger values down while squishing smaller elements up in the heap.

For example:

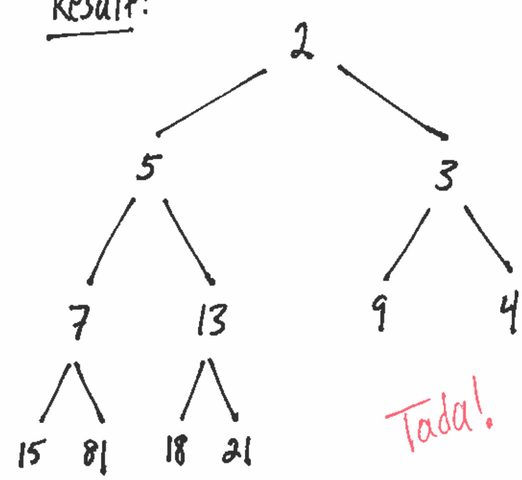


Given the tree to the left, we would call:

percolateDown(4)
percolateDown(3)
percolateDown(2)
percolateDown(1)
percolateDown(0)

These are the indices where we start our percolateDown operations.

Result:



Tada!

Runtime for Heapify:

Note that in a complete binary tree, approximately half our nodes are leaf nodes. So, heapify calls percolateDown() — which has a worst-case runtime of $O(\log n)$ — approximately $n/2$ times.

It might be tempting, then, to say that the runtime for heapify must be $O(n \log n)$.

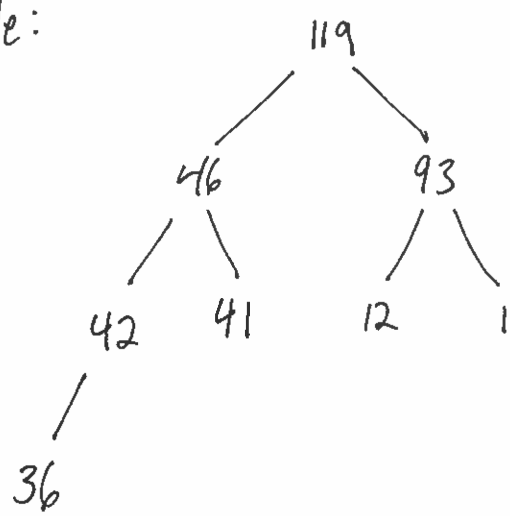
It turns out the runtime for heapify is even <u>better</u> than that, though! Do you see why $O(n \log n)$ is actually an overestimate?

The actual runtime is $O(n)$. This is an awesome surprise! We will explore this briefly in an upcoming lecture.

Supplementary:

A **maxheap** has the same structural property as a minheap, but the ordering property is inverted: every node's value (priority) must be greater than or equal to the values of any/all of its children.

For example:



Can you see how we would modify our enqueue/dequeue operations to implement a maxheap?