7/13/20 - Quickselect, Heaps

Monday, June 29, 2020 10:10 AM

Selection problem: Given a list of n items, find the kth smallest item.

One way to do this: sort the data and return array[k-1].
Run time = O(n lg n) - use Merge Sort, for example…

Is there a faster way to do this that doesn't require sorting?

At least for the average case run time, yes!
We can get an average case run time of O(n) using quickselect.

```
// Return the rank smallest item in array[low, high].
int quickselect(int* array, int low, int high, int rank);
```

Consider what partition does…

10, 3, 18, 9, 15, 2, 6, 8, 14, 13, 1

imagine wanting to find the 4th smallest value.

Step 1: Partition the array:

6, 3, 1, 9, 8, 2, 10, 15, 14, 13, 18

Partition tells us that there are 6 elements to the left of 10 and 4 elements to the right of 10.

So this tells me that I can just recursively find the 4th smallest item on the left side of this array (which has low=0, high = 5).

Imagine instead, that I was looking for the 8th smallest value…

There are 6 values less than 10.
10 it the seventh smallest value.

This means the value I am looking for is the 1st smallest value in the subarray [15, 14, 13, 18]. (where low = 7, high = 10)

Other option would be that I was looking for the 7th smallest value. In this case, we know the answer is 10, since 10 is definitely the 7th smallest value!

```
int quickselect(int* array, int low, int high, int rank) {

    if (low == high) return array[low];
    int mid = partition(array, low, high);

    if (rank <= mid-low)
        return quickselect(array, low, mid-1, rank);
    else if (rank > mid-low+1)
        return quickselect(array, mid+1, high, rank-(mid-low+1));
    else
        return array[mid];
}
```

Run-Time:

Best-case - we find the item after the first partition, this is clearly O(n) time.

Worst-case - we do bad partitions, and the first one takes O(n), the next one takes O(n-1) time …downto O(1). $1+2+3+\ldots+n = O(n^2)$.

What about the average case???

Let T(n) be the average case run time of partition…

1/n of the time the split is 0 and n-1
1/n split is 1 and n-2
…

yellow = partition time

$$
\begin{aligned}
T(n) = 1/n*(\ & 1/n*0+0/n*T(0)+(n-1)/n*T(n-1) + \\
& 1/n*0+1/n*T(1)+(n-2)/n*T(n-2) + \\
& 1/n*0+2/n*T(2)+(n-3)/n*T(n-3) + \\
& \ldots \\
& 1/n*0+(n-1)/n*T(n-1)+0/n*T(0)) + \\
& O(n)
\end{aligned}
$$

$$T(n) = 2/n^2 *(\ T(1) + 2T(2) + 3T(3) + 4T(4) + (n-1)T(n-1)\ ) + O(n)$$

It turns out that the solution to this recurrence after even more math than the quick sort analysis is O(n).

Intuition is as follows:

On average, my partition will split the array into an array of size 3n/4 and n/4. In the most probable case, I go into the big array. So my run time is:

$n + 3n/4 + 9n/16 + 27n/64 + \ldots.$

This is an infinite geometric sum. The value of it 4n = O(n).

Next topic is Binary Heaps, which are used for both Priority Queues and Heap Sort.

We want a data structure that will allow us to do the following things:

1) Insert an item
2) Remove the minimum item (or maximum)

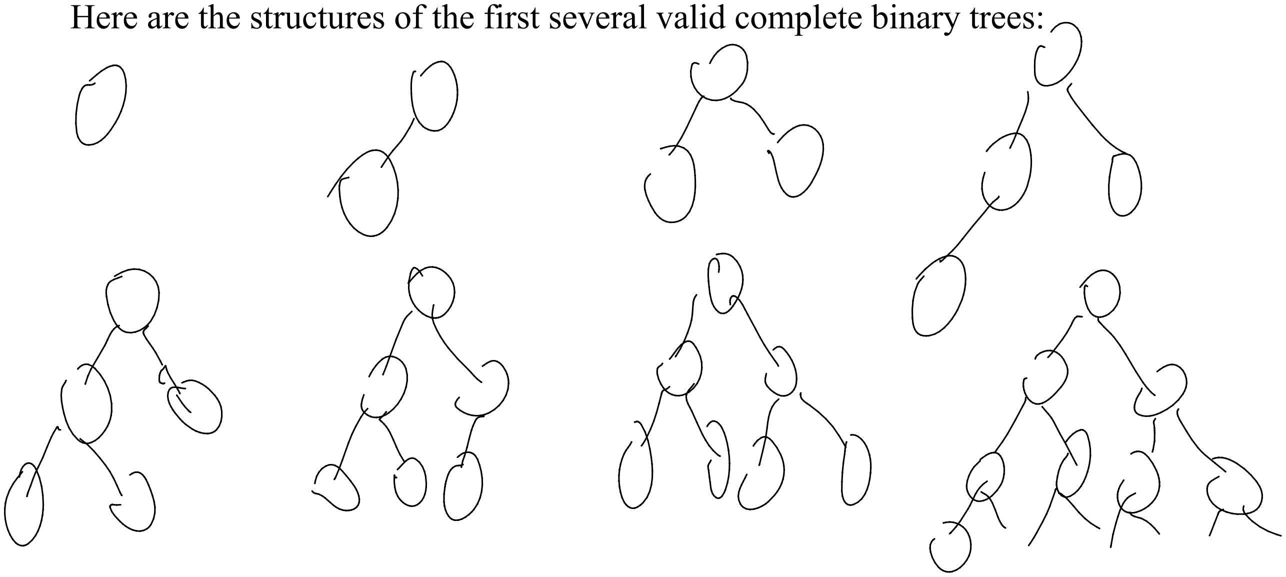
We want to do these things in O(lg n) time, where n is the # of items in the data structure.

Binary Heap is a Compete Binary Tree that satisfies the following property (the heap property):

For each node in the tree, both children store a value greater than it stores.

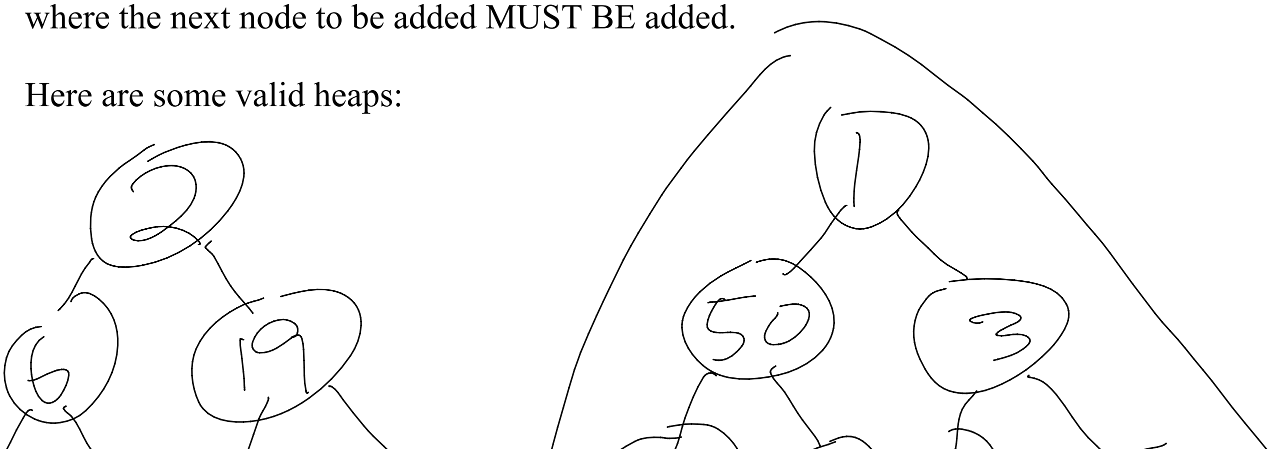
A complete binary tree is a tree that is completely filled in at each level, except possibly the last, and on the last level, the items are filled in left to right.

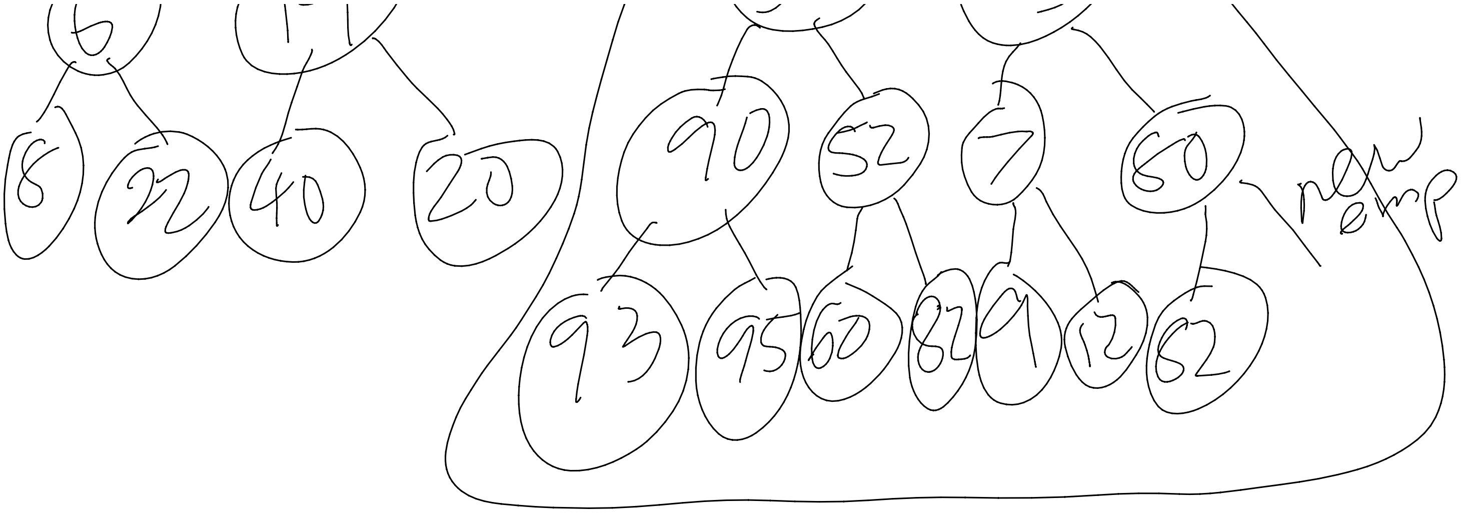
Here are the structures of the first several valid complete binary trees:



The key is that the structure is completely fixed. We ALWAYS know where the next node to be added MUST BE added.

Here are some valid heaps:





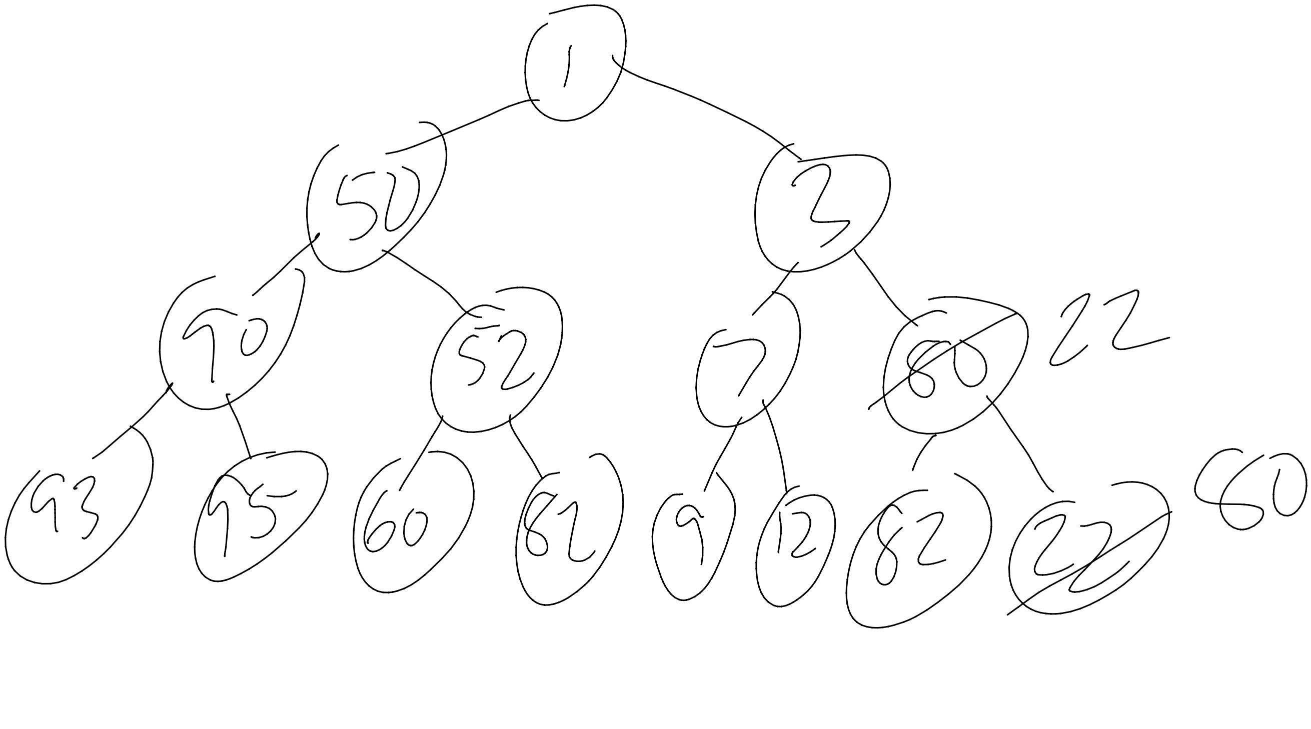
Two ways to store a heap:

1. Like a binary tree with pointers to the left and right child.
2. Like an array, indexed 1, 2, 3, …, where for each node stored in index i, its left child is stored in index 2i and its right child is stored in index 2i+1.

	2	6	19	8	22	40	20
0	1	2	3	4	5	6	7

If you need to get to the parent of the node stored at index i, go to index i/2.

How do I insert something into a heap?
How do I delete the minimum?
For now, we pretend we have a valid heap.



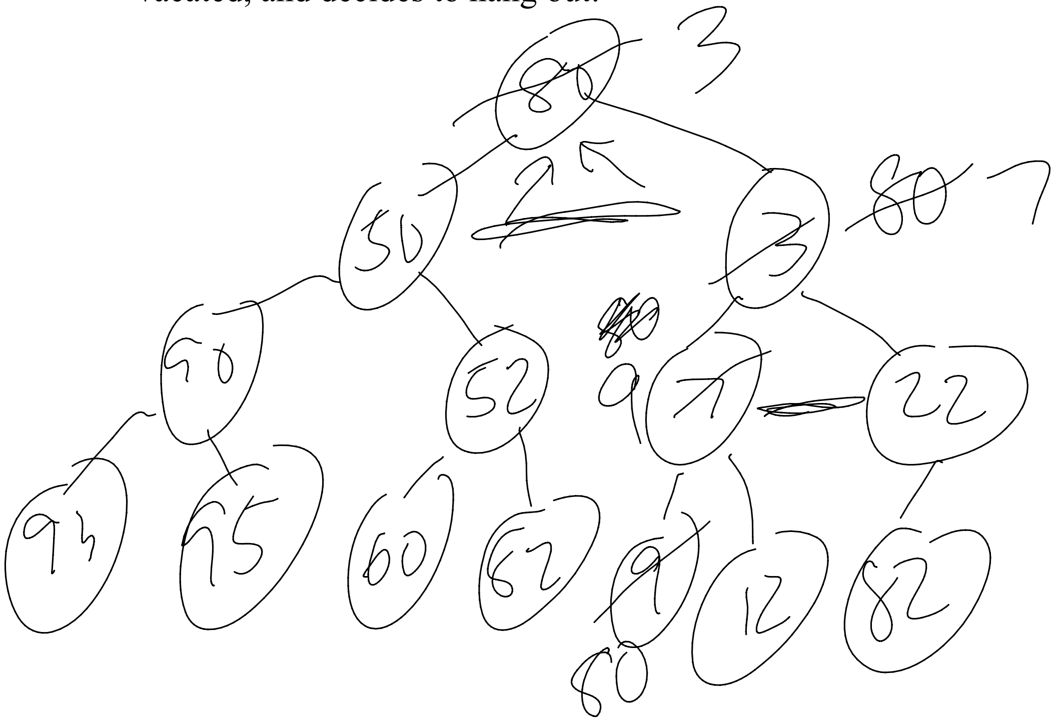
Insert steps:

1. Place the new value in the next valid location, on the last row (or start a new row if you need to).
2. Report to your boss (parent), if you are lower in number (higher in priority), then you ought to swap with them. Continue until, your boss is of higher priority (lower number) than you.

We will call the process in #2 "percolate up".

Delete min steps:

1. Remove min value from top, but keep the node there. This is like the CEO vacating her office.
2. Structurally, the "last node" must be removed…This is where the mailboy is. As he is delivering mail, he discovers the CEO's office vacated, and decides to hang out:



2. Both child nodes of the top node see if the value belongs there, and if not, the better one (lower value) of the two, swaps with it. Process continues down the tree, until the mailboy finds a proper place in the hierarchy.

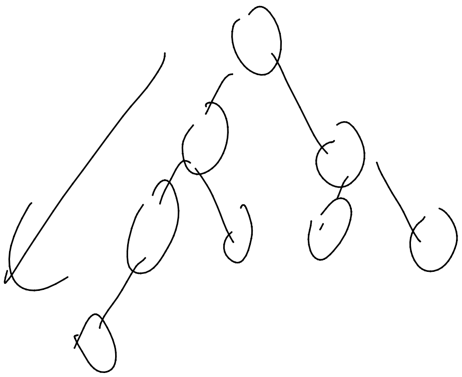
#2 here is called percolate Down.

If I have a complete binary tree of n elements, its height is O(lg n). An easy way to see this is to fill a tree up to level k:

1 + 2 + 4 + 8 + … + 2^{k-1} +1 (first node in next level…)

This sum is 2^k, and the worst amount of work to move all the way down the tree is k steps to that last bottom node.

This sum is 2^k, and the worst amount of work to move all the way down the tree is k steps to that last bottom node.



8 nodes
path of 3
to get to bottom

In the worst case, the height of a tree with 2^k nodes is k. Let $n = 2^k$ and we see that k, the height is $\lg_2 n$.

In both percolate Up and percolate Down, the amount of work is a constant times the height of the tree, since Insert and Delete Min run in O(n) time.

Heap Sort:

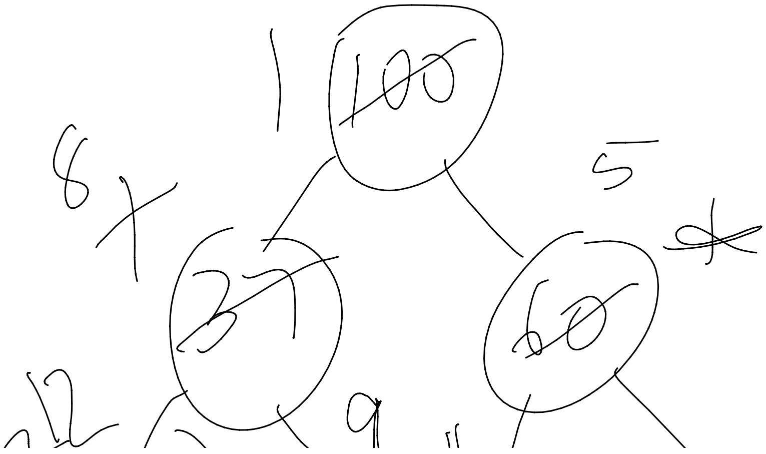
1) Add each item you want to sort in a heap. Run time is n * O(lg n) so at most O(nlg n).
2) Delete the minimum item n times. this will also take n *O(lg n) = O(nlgn) time.

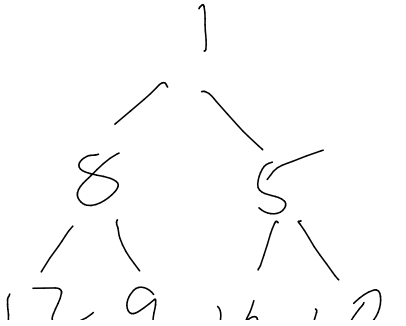
One question you might have - can we do better than O(nlgn) to create a heap???

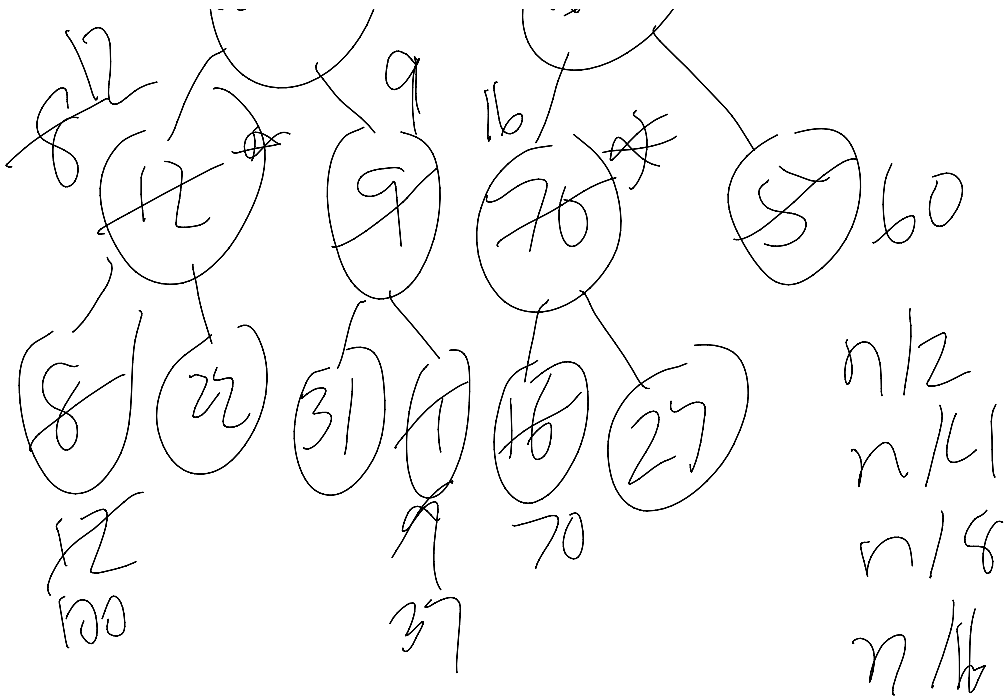
YES - there is a function called "heapify" that takes random values and turns them into a heap in O(n) time:

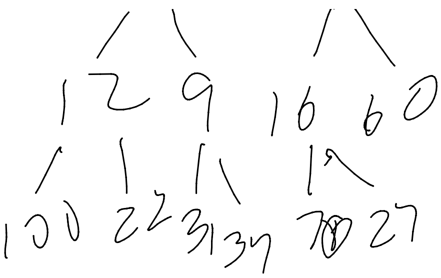
Pseudocode for Heapify(heap of size n):

```
for (i=n/2; i>0; i--)
    percolateDown(heap, i);
```









$n/2 \rightarrow 1$

$n/4 \rightarrow 2$

$n/8 \rightarrow 3$

$n/16 \rightarrow 4$

$$S = (1)\frac{n}{2} + 2\left(\frac{n}{4}\right) + 3\left(\frac{n}{8}\right) + 4\left(\frac{n}{16}\right) + \cdots$$

$$-\frac{S}{2} = 1\left(\frac{n}{4}\right) + 2\left(\frac{n}{8}\right) + 3\left(\frac{n}{16}\right) + \cdots$$

$$\frac{S}{2} = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} +$$

$$\frac{S}{2} = n \qquad O(n)$$

$$S = 2n$$

Also, to see a good heap example, please go here:

http://www.cs.ucf.edu/
~dmarino/ucf/transparency/cop3502/sampleprogs/heapexample.c

The input for it is here:

http://www.cs.ucf.edu/
~dmarino/ucf/transparency/cop3502/sampleprogs/heapexample.in