

## Building a heap in $O(n)$ (fast)

Last lecture we looked at the `downHeap()` method. We used this method to sort an existing heap which itself was built using an `upHeap()` method. Let's now use the `downHeap` method to build a heap. We will see that this new build heap method is much faster than the one that used `upHeap`.

ASIDE: Having a fast algorithm for building a heap would be useful if  $n$  is large and if we wanted to change the definition of how keys/nodes are compared from time to time. We would need to rebuild the priority queue (heap), based on the new comparison definition. For example, the comparison might be based on a calculation of a monetary cost, which in turn is based on the price of something, which fluctuates over time.) Anyhow, here is the algorithm:

```
Input:  an array a of size n (with elements in any order)
Output: a heap
```

```
for i = n to 1          // backwards from n
    downHeap( i, n )
```

At first glance, you might think that this algorithm is  $O(n \log n)$  since you seem to have a loop over  $O(n)$  elements and it seems that each time through the loop you may (worst case) be performing  $\log n$  operations, namely in the worst case you may have to make  $\log n$  swaps. But this is too pessimistic. Why?

First, note that half the nodes in the tree are already leaves (and, for such nodes, `downHeap` does nothing). Why are half the nodes already leaves? Consider node  $n$  which is the last node. Its parent node is node  $\lfloor \frac{n}{2} \rfloor$ . But by inspection, any node with index greater than this parent node will not have any children. (Draw a heap example to convince yourself.)

For the  $n/2$  internal nodes, half of these have height 1 (i.e. depth  $h - 1$ ) and so are `downHeap`-ed a distance of at most 1. There are only  $n/4$  nodes remaining. Half of these have height 2 and so are `downHeap`-ed a distance of at most 2. Although nodes near the root may need to be `downHeap`-ed a distance of roughly  $\log n$ , there are very few nodes near the root.

Let's show how this works more formally. Consider again the  $n$  nodes placed in a complete binary tree and numbered from 1 to  $n$ . There are  $2^l$  nodes at level  $l$  (except possibly for the nodes at level  $h$  since this level might not be full). Each of the nodes at level  $l$  is the root of a subtree of maximum height  $h - l$ , and hence each will be `downHeap`-ed a maximum distance of  $h - l$ .

height	number of nodes of that height
-----	-----
$h$	1 (namely the root of the tree)
$h-1$	2 (namely the children of the root)
:	:
$i$	$2^{\{h-i\}}$
:	:
1	$2^{\{h-1\}}$
0	$2^h$ (or less, if last level is not full)

Thus, the maximum number of swaps is at most  $\sum_{l=0}^h (h - l) 2^l$ .

[If you do not have time/interest/patience to go through the following derivation, then just jump down to the last few lines where the result is given. ]

We can evaluate this upper bound by changing variables:  $j = h - l$ , which gives

$$\sum_{l=0}^h (h-l)2^l = \sum_{j=0}^h j 2^{h-j} = 2^h \sum_{j=0}^h j 2^{-j} = 2^h \sum_{j=0}^h j 2^{-(j-1)-1} = 2^{h-1} \sum_{j=0}^h j \left(\frac{1}{2}\right)^{j-1}.$$

Now we use a trick. We let  $x = \frac{1}{2}$  and rewrite the summation on the right:

$$\sum_{j=0}^h j x^{j-1} = \sum_{j=0}^h \frac{d}{dx} x^j = \frac{d}{dx} \sum_{j=0}^h x^j < \frac{d}{dx} \sum_{j=0}^{\infty} x^j = \frac{d}{dx} \left(\frac{1}{1-x}\right) = \frac{1}{(1-x)^2}$$

Note, we can only write  $\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$  when  $|x| < 1$  but that's ok since  $x = \frac{1}{2}$ .  
From

$$\sum_{j=0}^h j x^{j-1} < \frac{1}{(1-x)^2}$$

we substitute  $x = \frac{1}{2}$  to get

$$\sum_{j=0}^h j \left(\frac{1}{2}\right)^{j-1} < 2^2$$

Thus,

$$\sum_{l=0}^h (h-l)2^l < 2^{h-1} 2^2 = 2^{h+1}$$

But

$$2^h \leq n < 2^{h+1} < 2n$$

and so *the maximum number of swaps is bounded above by 2n*. Thus, the algorithm for building a heap using `downHeap` is  $O(n)$ .

### Notes on Assignment 3

See lecture slides for a brief discussion of casting primitive types in Java, and an introduction to prefix codes. For a more thorough introduction see the assignment pdf, which points to my COMP 423 lecture notes (or google other resources covering the same material).