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

Suppose you want to build a priority queue very quickly. For example, you might already have a priority queue, but now you want to change the definition of how any two elements are compared. This would require that you build a new priority queue based on the new comparison definition. (Recall the discussion of methods `compareTo()` and `compare()` in lecture 18.)

We are implementing priority queues with a heap, so let's now look at a faster algorithm for building a heap. We start off with a complete binary tree with n nodes and make no assumption on the ordering of the elements.

The `makeHeap()` algorithm for last class built a heap *from the top down* by swapping elements *up the heap* from child to parent. Given that the first i elements already define a heap, it compares the $i + 1^{th}$ element to its parent (and grandparent, etc, if need be) and carries out a sequence of swaps. Worst case is that the element climbs up to the root.

Today we go in the opposite direction(s). We build a heap *from the bottom up*, and *swapping elements down the heap* toward the leaves.

```
Algorithm:  buildHeapFast(a)
Input:  an array a of size n (with elements in any order)
Output: a heap
                            //  We could start at n rather than n/2
for i = floor(n/2) to 1     //  but this is not necessary, since
  downHeap( a, i )          //  i > n/2 implies the node is a leaf.
```

This algorithm calls an operation `downHeap()`. In fact we used this operation in lecture 25 (though we did not explicitly state it). The operation `downHeap` takes a binary tree whose two children (if they exist) are heaps, and pushes the root node of this binary tree down (by swapping with the smaller of its children) until the heap property is satisfied, namely parent is smaller than children.

```
Algorithm:  downHeap(a , i)
Input:  an array a and index i such that the children of node i
        are heaps
Output: an array a such that the subtree rooted at node i is a heap

left = 2 * i;
right= 2 * i + 1;
if (left <= size) & ( a[i] <= a[left] )
    min = i;
else
   min = left;
if (right <= size) & ( a[min] > a[right] )
   min = right;
if (min != i){
   swap( a, i, min)
   downHeap(a, min)
}
```

See comment in the code: if node i is a leaf, then its `left` and `right` child will be greater than n. So this is a base case of the `downHeap` recursion. No swap is done in the base case.

Analysis of `buildHeapFast` algorithm

At first glance, you might think that this algorithm is $O(n \lg n)$ since you seem to have a loop over $O(n)$ elements and it seems that each time through the loop you are performing $O(\lg n)$ operations, namely in the worst case you may have to make $\lg n$ swaps. Interestingly, this reasoning is false. In fact, the algorithm for building a heap runs in $O(n)$ time.

The intuition is as follows. First, notice that the loop is only over $n/2$ nodes, since half the nodes are leaves and there is nothing to be done (they are already heaps). For the $n/2$ internal nodes, half of these have height 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 two. Thus, although nodes near the root may need to be `downHeap`-ed a distance of roughly $\lg n$, there are extremely few of these nodes. Let's show how this works formally.

Consider again the n nodes placed in a complete binary tree and numbered from 1 to n. Each of these nodes defines a subtree with this node at the root. 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 $h - l$.

Thus, the maximum number of swaps is at most $\sum_{l=0}^{h}(h-l)2^l$. We can evaluate this by changing variables: $j = h - l$, which gives

$$\sum_{l=0}^{h}(h-l)2^l = \sum_{j=0}^{h} j2^{h-j} = 2^h \sum_{j=0}^{j} i2^{-j}$$

The summation on the right is probably a mystery to you but notice it can be written

$$\sum_{j=0}^{h} jx^j$$

where $x = \frac{1}{2}$.

$$\sum_{j=0}^{h} jx^j \;=\; x\sum_{i=0}^{\infty} jx^{j-1} \;=\; x\sum_{j=0}^{h}\frac{d}{dx}x^j \;=\; x\frac{d}{dx}\sum_{i=0}^{h}x^j \le x\frac{d}{dx}\sum_{j=0}^{\infty}x^j \;=\; x\frac{d}{dx}(\frac{1}{1-x}) \;=\; \frac{x}{(1-x)^2}$$

Now substitute $x = \frac{1}{2}$, and we get

$$\sum_{i=0}^{h} i(\frac{1}{2})^i \le 2$$

Thus, the maximum number of swaps $\sum_{l=0}^{h}(h-l)2^l$ is bounded above by $2^h \cdot 2$. But $2^h \le n$ and so the maximum number of swaps is bounded above by $2n$ which obviously is $O(n)$.