This little paper argues that the heap building operation, which appears to be $\mathcal{O}(n \log n)$ on a list of length n, is actually $\mathcal{O}(n)$. Unfortunately, the heap sort algorithm is still $\mathcal{O}(n \log n)$. A standard reference for this material is *Introduction to Algorithms* by Cormen, Leiserson, and Rivest.

1 A Fact from Calculus

In order to carry out our heap building running time analysis we need a little fact from mathematics: Equation (1) below. The proof of this fact uses calculus. If you've taken calculus at the level of Calculus II, then you should be able to follow it. If you haven't studied such stuff, then don't worry; skim this section and just remember that Equation (1) is true, and that the mathematical argument for it is pretty short and fairly elementary.

In calculus, it is well known that the geometric series converges for any real number x such that $|x| < 1$. Its value is

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

The function defined by the series is differentiable on $(-1, 1)$; let's differentiate it:

$$\sum_{k=1}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}.$$

If we multiply that equation by x, we obtain

$$\sum_{k=1}^{\infty} k x^k = \frac{x}{(1-x)^2}.$$

Now plug in $x = \frac{1}{2}$:

$$\sum_{k=1}^{\infty} \frac{k}{2^k} = 2.$$

The quantity $k/2^k$ is 0 when k is 0, so we can add a $k = 0$ term to the left-hand side without any harm:

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = 2. \tag{1}$$

Okay; that's what we needed.

2 A Fact about Binary Heaps

We need another ingredient for our running time analysis. It doesn't use any mathematics beyond algebra, but it's more difficult than the preceding argument.

Recall that in any binary tree the height h and the number n of nodes are related by

$$n \leq 2^{h+1} - 1. \tag{2}$$

Equality holds if and only if the binary tree is full. In a binary heap, all of the levels of the tree are full except perhaps for the last one. Using Equation (2) twice, it is easy to see that a binary heap's height h and number of nodes n satisfy

$$2^h - 1 \leq n \leq 2^{h+1} - 1. \tag{3}$$

Unfortunately we need some terminology. When we talk about the *height* of a tree node, we mean the height of the subtree based at that node. For example, the leaves have height 0, the nodes immediately above the leaves have height 1, and the root has height h. For any number k such that $0 \leq k \leq h$, let $n_{\geq k}$ denote the number of nodes in the tree that are of height k or greater. For example, $n_{\geq 0} = n$ and $n_{\geq h} = 1$. In any binary heap, for any given node-height k, the nodes of height greater than or equal to k form a binary heap of height $h - k$. Therefore, by Equation (3),

$$2^{h-k} - 1 \leq n_{\geq k} \leq 2^{h-k+1} - 1. \tag{4}$$

Now, for any number k such that $0 \leq k \leq h$, let $n_{=k}$ denote the number of nodes in the heap that are of height exactly k. For example, $n_{=h} = 1$. Then for any k,

$$\begin{aligned} n_{=k} &= n_{\geq k} - n_{\geq k+1} \\ &\leq \left(2^{h-k+1} - 1\right) - \left(2^{h-(k+1)} - 1\right) \\ &= 2^{h-k+1} - 2^{h-k-1} \\ &= 3 \cdot 2^{h-k-1} \\ &= 3 \cdot 2^h \cdot 2^{-k-1}. \end{aligned}$$

(The inequality follows by applying Equation (4) in two different ways.) Henceforth assume that $h \geq 2$. Then $3 \cdot 2^h \leq 4 \cdot \left(2^h - 1\right)$, so

$$\begin{aligned} 3 \cdot 2^h \cdot 2^{-k-1} &\leq 4 \cdot \left(2^h - 1\right) \cdot 2^{-k-1} \\ &\leq 4 \cdot n \cdot 2^{-k-1} \\ &= \frac{2n}{2^k}. \end{aligned}$$

It wasn't easy, but we've proved that in any binary heap of n nodes and of height $h \geq 2$, for any integer k such that $0 \leq k \leq h$,

$$n_{=k} \leq \frac{2n}{2^k}. \tag{5}$$

Intuitively, this says that there are many more low-height nodes than high-height nodes.

3 The Running Time of Heap Building

In this section we use our two facts (Equations (1) and (5)) to prove that building a heap is $\mathcal{O}(n)$ on a list of length n.

The first step in the heap building algorithm is to copy the given list into the heap's internal list. This is $\mathcal{O}(n)$.

The next step in building a heap is to percolate down all of the heap nodes. (Commonly we percolate only half of the nodes, but in this argument we'll pretend that we percolate all of them — this can only worsen the running time.) If the height h of the heap is less than 2, then the number n of nodes is less than 8 and all operations are constant-time; henceforth assume that $h \geq 2$. Some of the nodes that we percolate are at height 0, some are at height 1, and so on, up to the full height h. The number of nodes at any given height k is at most $2n/2^k$ by Equation (5). The cost of percolating a node at height k is proportional to k. Therefore the total cost of percolating all of the nodes is no greater than proportional to

$$\sum_{k=0}^{h} \frac{2n}{2^k} k \quad \leq \quad 2n \sum_{k=0}^{\infty} \frac{k}{2^k} \quad = \quad 2n \cdot 2,$$

by Equation (1). So the percolation step is $\mathcal{O}(n)$, and so is the entire heap building algorithm.

4 Heap Sort

Unfortunately, the heap sort algorithm is still $\mathcal{O}(n \log n)$ because the process of dequeueing everything from the heap is still $\mathcal{O}(n \log n)$.