Faster algorithm for building a heap

Last lecture I showed you an \( O(n \log_2 n) \) algorithm for building a heap. I will next present an algorithm that runs in time \( O(n) \). The faster algorithm is based on the `downHeap()` method from last lecture, where the two parameters are `startIndex` and `maxIndex` in the heap array. The input is a list with `size` elements. The output is a heap.

```java
buildHeapFast(list){
    create new heap array // size == 0, length > list.size
    for (k = size/2; k >= 1; k--)
        downHeap( k, size )
}
```

The algorithm begins at node \( k = \lfloor n/2 \rfloor \) and decrements the index down to the root node \( k = 1 \). For each \( k \), it `downHeaps`, that is, it swaps the element from starting position \( k \) with the smaller of its children and repeats this until it is less than both its children (if it has any children).

The reason that the algorithm starts at \( k = \lfloor n/2 \rfloor \) is that the nodes \( \lfloor n/2 \rfloor + 1 \) to \( n \) have no children to compare with. So we don’t bother `downHeaping` them.

Example

An initial arrangement of \( n = 6 \) keys is shown on the left. I show the state of the tree before the \( k \)th node is `downHeaped`, and the final state.

<table>
<thead>
<tr>
<th>( k=3 ) (c)</th>
<th>( k=2 ) (x)</th>
<th>( k=1 ) (w)</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>w</td>
<td>w</td>
<td>b</td>
</tr>
<tr>
<td>/ \</td>
<td>/ \</td>
<td>/ \</td>
<td>/ \</td>
</tr>
<tr>
<td>x c --&gt; x b --&gt; p b --&gt; p c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ \ / \ / \ / \ / \</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p r b p r c x r c x r w</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case analysis for `buildHeapFast`

For each \( k \) of the `buildHeap` algorithm, the worst case number of swaps done by `downHeap()` is the height of the node \( k \) in the tree. Thus the total number of swaps that we need to do is the total of the heights of the nodes in the tree. Recall that the height of a node in a tree is the maximum path length from the node to a leaf.

Let \( h \) be the height of the tree i.e. the height of the root node. Let’s assume for mathematical analysis that we have a complete binary tree of height \( h \) and that level \( h \) is full. (All other levels are full by definition.) In this case, you can see by inspection that the height of every node at level \( l \) will be \( h - l \). That is, the height of the root node (level 0) is \( h \), the height of the two children of the root are \( h - 1 \), etc, and the height of all leaf nodes is \( h - h = 0 \).
Define $t_{\text{worst case}}(n)$ be the sum of heights of all nodes. We write it in terms of $h$ and sum over levels $l$:

$$t_{\text{worst case}}(h) = \sum_{l=0}^{h} (h - l) \ 2^l$$

$$= h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l \ 2^l$$

The first term is $h(2^{h+1} - 1)$. The second term is the sum of the depths (or levels) of all the nodes. It is a bit trickier to solve.

I show in the Appendix (next page) that:

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

Plugging into the term terms above, we get

$$t_{\text{worst case}}(h) = h(2^{h+1} - 1) - (h - 1)2^{h+1} - 2$$

which we can simplify to

$$t_{\text{worst case}}(h) = 2^{h+1} - h - 2$$

To write $t_{\text{worst case}}(n)$ in terms of $n$ rather than $h$, we recall that we are assuming all levels of the tree are full, i.e. including level $l = h$ which is the height of the tree. So,

$$n = 2^{h+1} - 1$$

and so

$$h = \log(n + 1) - 1.$$  

Substituting for $h$, we get

$$t_{\text{worst case}}(n) = n - (\log(n + 1)).$$

Remarkably, this is less than $n$. In particular, $t_{\text{worst case}}(n)$ is $O(n)$.

The intuition here is that most of the nodes in the tree are near the leaves, since the height of the tree is $\lfloor \log n \rfloor$, most of the leaves have depth which is either $\lfloor \log n \rfloor$ or very close to it.
Appendix

Here I will give a slightly simpler derivation than what I gave in the lecture and slides. The idea for this derivation was pointed out to me by a student after class and is indeed simpler.

\[ t_{\text{sumlevels}}(h) = \sum_{l=0}^{h} l \ 2^l \quad (*) \]
\[ = \sum_{l=0}^{h-1} (l + 1)2^{l+1} \quad (**) \]

Multiplying both sides of (*) by 2 gives

\[ 2 \ t_{\text{sumlevels}}(h) = \sum_{l=0}^{h} l \ 2^{l+1} \quad (***) \]

and taking the difference (***) - (**) gives

\[ t_{\text{sumlevels}}(h) = h2^{h+1} - \sum_{l=0}^{h-1} 2^{l+1} \]
\[ = h2^{h+1} - 2 \sum_{l=0}^{h-1} 2^l \]
\[ = h2^{h+1} - 2(2^h - 1) \]
\[ = (h - 1)2^{h+1} + 2 \]