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 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.

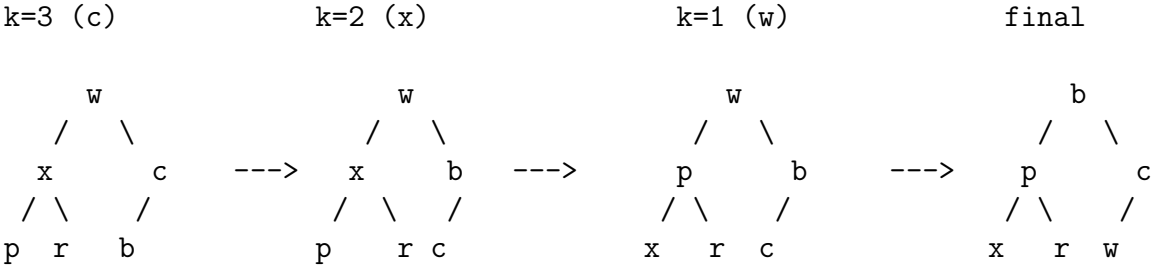
```
buildHeapFast(list){
  create new heap array //  size == 0,  length > list.size
  for (k = size/2; k >= 1; k--)
     downHeap( k, size )
}
```

The algorithm begins at node `k = n/2` 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 = n/2` is that the nodes `size/2+1` to `size` 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.



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 $t_{worstcase}(n)$ be the sum of heights of all nodes. We write it in terms of h and sum over levels l:

$$\begin{aligned} t_{worstcase}(h) &= \sum_{l=0}^{h} (h-l)\ 2^l \\ &= h\sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\ 2^l \end{aligned}$$

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 (end of today's lecture notes) that:

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

Plugging into the term terms above, we get

$$t_{worstcase}(h) = h(2^{h+1}-1) - (h-1)2^{h+1} - 2$$

which we can simplify to

$$t_{worstcase}(h) = 2^{h+1} - h - 2$$

To write $t_{worstcase}(n)$ 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

$$t_{worstcase}(n) = n - \log(n+1).$$

Remarkably, this is less than n. In particular, $t_{worstcase}(n)$ is $\Theta(n)$.

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.

Heapsort

A heap can be used to sort a set of elements. The idea is simple. Just repeatedly remove the minimum element by calling `removeMin()`. This naturally gives the elements in their proper order.

Here I give an algorithm for sorting "in place". We repeatedly remove the minimum element the heap, reducing the size of the heap by one each time. This frees up a slot in the array, and so we insert the removed element into that freed up slot.

The pseudocode below does exactly what I just described, although it doesn't say "`removeMin()`. Instead, it says to swap the root element i.e. `heap[1]` with the last element in the heap i.e. `heap[size+1-i]`. After `i` times through the loop, the remaining heap has `size - i` elements, and the last `i` elements in the array hold the smallest `i` elements in the original list. So, we only downheap to index `size - i` .

```
heapsort(list){
  buildheap(list)
  for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
  return reverse(heap)
}
```

The end result is an array that is sorted from largest to smallest. Since we want to order from smallest to largest, we must `reverse` the order of the elements. This can be done in $\Theta(n)$ time, by swapping i and $n+1-i$ for $i = 1$ to $\frac{n}{2}$.

Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted elements (on the right).

```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   d   b   e   l   u   k   f   w |
b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
w   u   l   k   f   e   d   b   a   (removed w, and done)
```

Note that the last pass through the loop doesn't do anything since the heap has only one element left (`w` in this example), which is the largest element. We could have made the loop go from `i = 1 to size - 1`.

Appendix

$$
\begin{aligned}
\sum_{l=0}^{h} l\ 2^l &= \sum_{l=0}^{h} l\ (2^{l+1} - 2^l) && \text{(trick)} \\
&= \sum_{l=0}^{h} l\ 2^{l+1} - \sum_{l=0}^{h} l\ 2^l \\
&= \sum_{l=0}^{h} l\ 2^{l+1} - \sum_{l=0}^{h-1} (l+1)\ 2^{l+1} && \text{second goes to h} - 1 \text{ only} \\
&= h\ 2^{h+1} + 2\sum_{l=0}^{h-1} (l - (l+1))\ 2^l \\
&= h\ 2^{h+1} - 2\sum_{l=0}^{h-1} 2^l \\
&= h\ 2^{h+1} - 2(2^h - 1) \\
&= (h-1)2^{h+1} + 2
\end{aligned}
$$