Homework 1 on Heaps

Edoardo Alessandroni

Exercises 1 and 2
The implementation of the binary heap data structure, together with the functions to work with it, are present in the file `binheaps.c`, and have been implemented like it was showed during the lectures.

Exercise 3
For this first version of the binary heap data structure (see next homework sheet and report for an optimized version of it, which avoids swapping elements in the array) some tests have been conducted, that can be summarized in the following table.
The numbers showed are in seconds and refer to the execution time needed to build a data structure (which size increases during the run) and keep (finding and) removing the minimum until it's empty. The data structures compared are (plain) arrays and binary heaps (implemented through arrays).

size	heaps	arrays
0	0.000016	0.000004
1820	0.157658	1.337558
3640	0.334076	5.314786
5461	0.512298	12.021184
7281	0.726907	21.388118
9102	0.835522	33.352266
10922	1.065924	49.985731
12743	1.429940	69.775816
14563	1.668459	92.275951
16384	3.274522	117.251829

Exercise 4:
If a node has no children, then it's a leaf. The nodes with an index in $S = \{\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, .., n\}$ have no children in the array, since we know that the children of the node m have indexes $2m$ and $2m + 1$ and

$$2(\lfloor n/2 \rfloor + 1) > 2(n/2) = n$$

therefore the first children of the first node in S is already out of the array. For this reason all the possible children of the nodes in S are not contained in the array of length n.
Conversely, all the leafs must have an index in S. This is because, since they are leafs, they must have an index m such that $2m > n$ and therefore $m \in S$.

Exercise 5:
In the worst-case scenario, calling HEAPIFY on the root node of a heap will cause a recursive call on *every* node on a simple path from the root node down to a leaf node. Call $t(h)$ the computational cost of executing 2 comparison and a swap on a node at height h; since this does not depend on h, $t(h) = k$, where k is a constant. Call $T(m)$ the computational time of executing HEAPIFY on a node which is the root of an m-sized subtree. Then:

$$T(n) \geq \sum_{h=1}^{\lfloor log\ n \rfloor} t(h) = k\ \lfloor log\ n \rfloor \in \Omega(log\ n)$$

Exercise 6:
Let's call $f(h) = \lceil n/2^{h+1} \rceil$ and let's consider at first the simpler case of a *full* binary tree of height H, that is one for which the number of nodes is $n^* = 1 + 2 + 4 + \ldots + 2^H = 2^{H+1} - 1$.
Let's call $N(h)$ the number of nodes at height h. For such a heap the condition $N_F(h) \leq f(h)$ holds, where $N_F(h)$ has the same meaning of $N(h)$ in the case of a full tree. Let's show that $N_F(h) \leq f(h)$.
Starting from the bottom (height $h = 0$):

$$N_F(0) = 2^H = \lceil 2^H - \frac{1}{2} \rceil = \lceil \frac{2^{H+1} - 1}{2} \rceil = \lceil \frac{n^*}{2} \rceil = f(0)$$

By induction, going up one level on the heap the number of nodes will be halved:
$N_F(h) = N_F(h-1)/2 = N_F(0)/2^h = f(0)/2^h \leq f(h)$.
So, the relation holds for a full binary tree.

Now let's consider the general case, comparing a full tree with another tree with same height H and with number of nodes $n \leq n^*$.
Clearly, $N(h) \leq N_F(h) \leq f(h)$.