Fundamental Algorithms - Spring 2018
Homework 6

Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

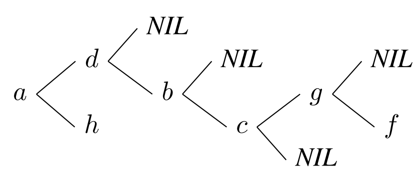
1. (a) The successor of c is f. Referring to the SUCCESSOR function, we find ourselves in the first case, where c has a right child (in this case g). Once that has been established, we simply have to find the minimum element of the tree rooted at g using TREE-MIN, which yields the desired value of f.

   (b) The minimal element is h. To find it, we run TREE-MIN starting at a (the root of the tree), and just keep walking the tree always going to the LEFT child. In this case, there is only one step that takes us from a to h.

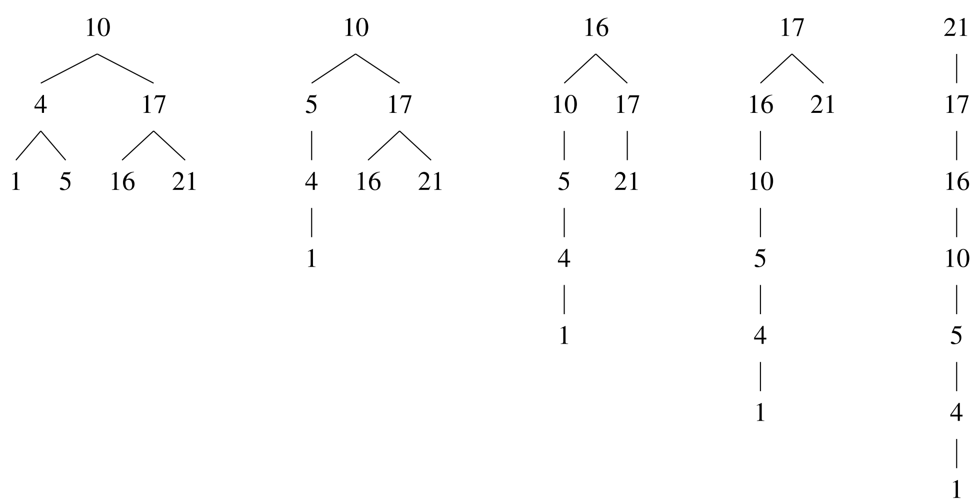
   (c) To perform DELETE[e] we notice that e has two non-NIL children, and therefore we are under the third case of the algorithm, which will perform the following steps:

```
DELETE[e]:
    b = SUCCESSOR[e]
    PARENT[b] ← d
    LEFT[d] ← b
    LEFT[b] ← c
    PARENT[c] ← b
    RIGHT[b] ← NIL
```

   After performing these operations, the tree will look as follows:



2. The following trees, from left to right, have heights $2, 3, 4, 5, 6$:

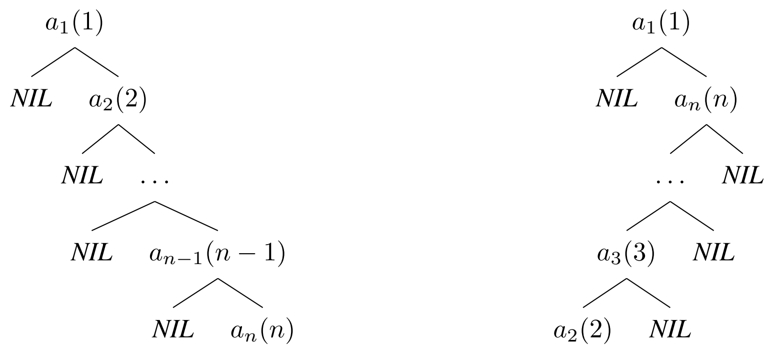


3. The binary search tree has the following properties for any node x:
   - $key(y) \leq key(x)\ \forall\ y \in$ `Tree[Left[x]]`
   - $key(z) > key(x)\ \forall\ z \in$ `Tree[Right[x]]`

   Whereas a max heap A has the following properties:
   - $key(i) \leq key($`PARENT[i]`$)\ \forall\ i \neq$ `Root[A]`
   - All rows of the heap (excepting maybe the last) must be fully filled.

   The properties of a heap are not enough to print the keys of an n-node tree in sorted order in $O(n)$ time. The simple intuition behind this is that if we want to sort a heap A, we need to use a `HEAP-SORT[A]` algorithm, which takes $O(n \log_2 n)$ for a heap (tree) with n elements.

4. The following are the two trees that would be built under the conditions described:



   As we can see, both trees will end up being very unbalanced.In the first case, all the insertions will be made in the `RIGHT` child of the previous element, basically creating a "list-like" structure. In the second case, only the first element n will be inserted as the `RIGHT` child of 1, and after that all insertions will go to the `LEFT` child of the previous one.

5. (a) Step 4 executes `EXTRACT-MAX[A]`, which we know it takes time $O(\log_2 n)$ for a heap of size n. In this case, the original heap has size N, and with each iteration inside the `while` loop the size is reduced by one (since we are extracting the max element each time). Therefore, at the I^{th} iteration step 4 will take time:

   $$T = O(\log_2(N - I + 1))$$

   (b) The `while` loop will execute as long as $I \cdot I \leq N$, or what is equivalent $I \leq \sqrt{N}$. Inside the loop, we have the `EXTRACT-MAX` operation, whose execution time we already know, and the `I++` operation, which takes constant time. Therefore, the total time for the `while` loop will be:

   $$T = O\left(\sum_{I=1}^{\sqrt{N}} (\log_2(N - I + 1) + 1)\right)$$

   $$T = O\left(\log_2\left(\prod_{I=1}^{\sqrt{N}} (N - I + 1)\right) + \sqrt{N}\right)$$

   $$T = O\left(\log_2\left(\frac{N!}{\left(N - \sqrt{N}\right)!}\right) + \sqrt{N}\right)$$

   If we look only at the argument of the logarithm and apply Sterling's formula to both numerator and denominator we get:

   $$\frac{N!}{\left(N - \sqrt{N}\right)!} \sim \frac{N^N}{\left(N - \sqrt{N}\right)^{\left(N - \sqrt{N}\right)}}$$

Now we look only at the denominator and we notice that if we expand it, it will yield a polynomial expression on N of order $N - \sqrt{N}$. Since we are doing asymptotic analysis, we can replace the whole polynomial with its highest order term:

$$O\left(\frac{N^N}{\left(N-\sqrt{N}\right)^{\left(N-\sqrt{N}\right)}}\right) = O\left(\frac{N^N}{N^{\left(N-\sqrt{N}\right)}}\right) = O\left(N^{\sqrt{N}}\right)$$

Finally, we replace this result in the original expression for the time complexity of the `while` loop and we get:

$$\begin{aligned}
T &= O\left(\log_2\left(N^{\sqrt{N}}\right) + \sqrt{N}\right) \\
T &= O\left(\sqrt{N}\log_2 N + \sqrt{N}\right) \\
T &= O\left(\sqrt{N}(\log_2 N + 1)\right) \\
T &= O\left(\sqrt{N}\log_2 N\right)
\end{aligned}$$

(c) The total time for the whole `KAUSHIK` algorithm will be given by the time it takes to run `BUILD-MAX-HEAP`, plus the total time for the `while` loop:

$$\begin{aligned}
T &= O\left(N + \sqrt{N}\log_2 N\right) \\
T &= O(N)
\end{aligned}$$

The linear term dominates the asymptotic analysis because the logarithmic function is always negligible when compared to polynomial functions.

(d) After completing its execution, `KAUSHIK` returns the value $z = A[1]$. Since the first thing the algorithm does is build a max heap out of A, and the heap property is preserved throughout the algorithm, we can guarantee that the value returned by `KAUSHIK` will be the maximum of the (remaining) elements in A by the end of the execution. Out of the N elements that A had originally, the $\sqrt{N}$ largest will be extracted and lost inside the `while` loop, leaving the array with only $N - \sqrt{N}$ elements by the time the return statement is reached. Therefore, when the algorithm returns the max value of the final version of A, it is in fact returning the $(\sqrt{N}+1)^{th}$ largest element of the original input array.