22C:44 Homework 4

Due in class on Tuesday 3/20

---

1. As you know, a heap is typically implemented as an array and the problem with this is that we may run out of slots in the array, but have yet another element to insert into the heap. Here I will lead you though a solution to this problem.

   Suppose that H is an array with m slots that we are currently using for the heap. Further suppose that H is full, that is, it contains m elements. If an element comes along that we need to insert into H, we would have to request an allocation of an array H' of size $m' > m$, copy the elements in H into H' (without changing their order) and then deallocate H. This takes a total of $\Theta(m + m')$ time.

   (a) Suppose that we started with an array of size 1. Further suppose that whenever we ran out of array slots, we requested an array of size one more than the old array. Under this scheme, what would be the total running time of a sequence of n heap-insert operations?

   (b) Suppose that we started with an array of size 1. Further suppose that whenever we ran out of array slots, we requested an array of size that is twice the size of the old array. Under this scheme, what would be the total running time of a sequence of n heap-insert operations, in the best and in the worst case?

2. A *3-ary heap* is just like the heap data structure we studied, except that each node now has at most 3 children.

   (a) What is corresponding heap property for a 3-ary heap?

   (b) Calculate the height of a 3-ary heap with n nodes.

   (c) Calculate the number of leaves in a 3-ary heap with n nodes?

3. Problem 7-1 on Page 152.

4. Problems 8.1-1 and 8.1-2 on Page 155.

5. Write a function `BETTER-PARTITION(A, p, s)` that takes as input the array $A[p..s]$ and partitions it into three blocks $A[p..q]$, $A[q+1..r]$, and $A[r+1..s]$ such that all elements in the middle block, $A[q+1..r]$ are identical, all elements in the first block, $A[p..q]$, are strictly smaller than those in the middle block, and all elements in the third block, $A[r+1..s]$ are strictly larger than those in the middle block. `BETTER-PARTITION(A, p, s)` returns the indices q and r as output.

   Note that if we partition $A[p..s]$ in this manner, we could then recursively sort $A[p..q]$ and $A[r+1..s]$ and ignore the block $A[q+1..r]$.

   Here I sketch a partitioning algorithm that solves the above problem and your task is to write pseudocode based on my sketch. Suppose that the block $A[p..j]$ has been processed thus far and we have computed indices q' and r', $p \leq q' < r' < j$, that yield the partition $A[p..q']$, $A[q'+1..r']$, and $A[r'+1..j]$, such that the first block contains elements strictly smaller than the elements in the middle block, the last block contains elements strictly larger than the elements in the middle block, and all the elements in middle block are identical. Our algorithm now processes $A[j+1]$ and moves it into the first, middle, or last block depending on how the elements $A[j+1]$ and $A[q'+1]$ compare.

   - If $A[j+1] > A[q'+1]$, then $A[j+1]$ should be in the last block and so there is no need to move $A[j+1]$.
   - If $A[j+1] = A[q'+1]$, then $A[j+1]$ should be moved into the middle block and one swap is sufficient to achieve this.

- If $A[j+1] < A[q'+1]$, then $A[j+1]$ needs to move into the first block and four assignment operations get this done.

When the entire array $A[p..s]$ is processed, the indices q' and r' is what the algorithm returns.
