CSE332 Data Abstractions, Spring 2012
Homework 2

Due: **Friday, April 13, 2010** at the beginning of class. Your work should be readable as well as correct.

This assignment has **five** problems.

Problem 1. Binary Min Heaps

This problem gives you some practice with the basic operations on binary min heaps. Make sure to check your work.

(a) Starting with an empty binary min heap, show the result of inserting, in the following order, 17, 9, 3, 8, 5, 6, 14, 1, 12, 11, and 2, one at a time (using percolate up each time), into the heap. By *show*, we mean, "draw the resulting binary tree with the values at each node." Draw the resulting tree after each insertion. In addition, give the array representation of your final answer.

(b) Now perform two `deleteMin` operations on the binary min heap you constructed in part (a). Show the binary min heaps that result from these successive deletions, again by drawing the binary tree resulting from each step. In addition, give the array representation of your final answer.

(c) Instead of inserting the elements in part (a) into the heap one at a time, suppose that you use Floyd's algorithm. Show the binary min heap tree that results from `buildHeap`. (It will help to show some intermediate trees so that if there are any bugs in your solution we will be better able to assign partial credit, but this is not required.) In addition, give the array representation of your final answer.

Problem 2. Merging "Full" Complete Binary Heaps

If we have two binary min heaps h_1 and h_2 of total size n, then Floyd's algorithm lets us combine them to build a new heap with all the elements from the two heaps in time $O(n)$. However, there are special cases where we can do better. Here we consider the case where both h_1 and h_2 are both *perfect trees*: a perfect tree is a binary tree where every level i contains 2^i nodes. That is, the last rows of h_1 and h_2 are full. We also allow the `merge` operation to "re-use/destroy" h_1 and h_2, i.e., only the merged heap is available after the `merge` operation. We assume the heaps are reprsented as pointer-based trees (not arrays) but we can find the "last" element of h_1 and h_2 in $O(1)$ time.

(a) Suppose h_1 and h_2 have the same height. Describe an $O(\log n)$ algorithm to merge the heaps. A concise English answer is sufficient.

(b) Suppose the height of h_1 is the height of h_2 minus one. Describe an $O(\log n)$ algorithm to merge the heaps. A concise English answer is sufficient.

Problem 3. Removing Arbitrary Items From Heaps

One way to remove an object from a binary min heap is to decrease its priority value by ∞ and then call `deleteMin`. An alternative is to remove it from the heap, thus creating a hole, and then repair the heap.

(a) Write pseudocode for an algorithm that performs the `remove` operation using the alternative approach described above. Your pseudocode should implement the method call `remove(int index)`, where `index` is the index into the heap array for the object to be removed. Your pseudocode can call the following methods described in lecture: `insert`, `deleteMin`, `percolateUp`, and `percolateDown`. Like in lecture, you may assume that objects are just priority integers (no other data).

(b) What is the worst case complexity of the algorithm you wrote in part (a)?

Problem 4. `findMax` for Min Heaps

In this problem you will *prove* that any algorithm for finding the maximum element in a binary min heap takes $\Omega(n)$ time in the worst case. For parts (a)–(c), be sure to state clearly what proof technique you are using, such as proof-by-contradiction or proof-by-induction (on what?).

(a) Prove that the maximum element of a min heap must be at a leaf.

(b) Prove that a heap with n elements has $\lceil n/2 \rceil$ leaves. (The notation $\lceil x \rceil$ is for the *ceiling function*; it means round up to the nearest integer.)

(c) Prove that every leaf must be examined to find the maximum.

(d) Conclude that any `findMax` algorithm for the binary min heap data structure takes $\Omega(n)$ time.

Problem 5. Double-Ended Priority Queues

Warning: Parts (c) and (d) may be much more difficult than most homework problems.

The previous problem proved that no $O(\log n)$ algorithm can get the maximum element from a min heap. You could use a max heap instead (highest priority value at the root), but then finding the minimum element is not $O(\log n)$. It turns out there is a different, more complicated data structure that can do `insert`, `deleteMin`, *and* `deleteMax` all in $O(\log n)$ time. We describe the data structure to you and then you need to describe how to perform the operations.

- Give a concise English description for your answers (not pseudocode, which could get a bit long). For example, English for `deleteMin` in a min heap from class would be: *Remove the root from the tree and return it after modifying the remaining heap as follows. Take the last element in the heap (the rightmost element of the deepest nodes) and put it at the root. Suppose it has priority p. Now percolate-down by seeing if p is greater than any child. If not, we are done. Else swap the position of p with the smallest child and then recursively percolate-down from the new position of p.*
- Some of your descriptions will be longer than the example above because how you percolate is more complicated: it depends on what level of the tree you are at (keep reading).

The data structure on which your operations work is as follows:

- The *structure property* is the same as for min heaps: a complete binary tree.
- The *heap property* is different:
  - If a node X is at an even depth (i.e., the root, the root's *grand*children, their *grand*children, etc.), then the element at X is smaller than its parent but larger than its grandparent. (Except the root has no parent/grandparents.)
  - If a node X is at an odd depth (i.e., the root's children, their *grand*children, etc.), then the element at X is larger than its parent but smaller than its grandparent. (Except the root's children have no grandparent.)
- Notice the heap property implies that every node has a value between its parent and its grandparent.

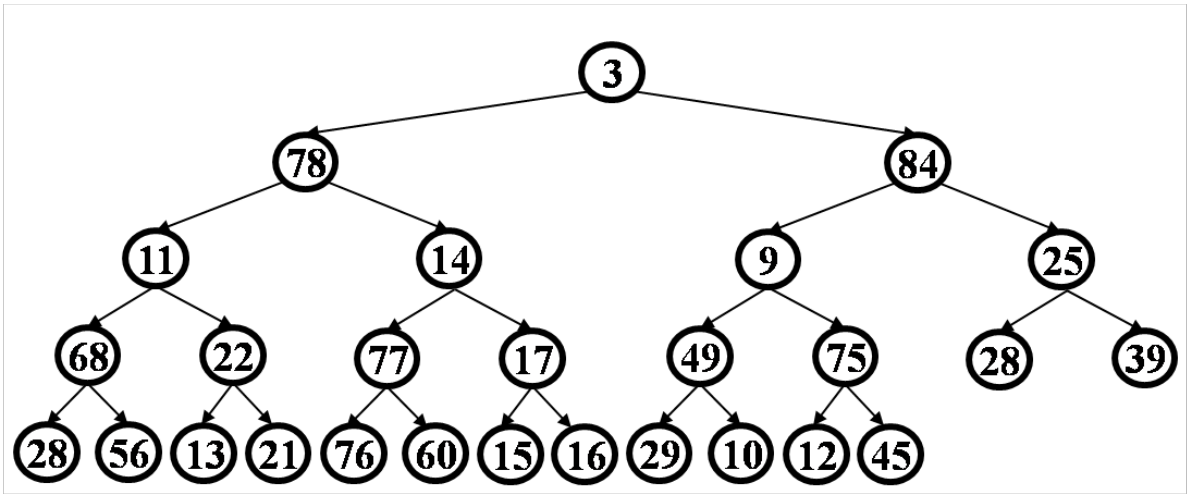
An example of this data structure is on the next page.

(a) Describe an $O(1)$ algorithm for `findMin`. (This is easy.)

(b) Describe an $O(1)$ algorithm for `findMax` operation. (This is only a little more difficult.)

(c) Describe $O(\log n)$ algorithms for `deleteMin` and `deleteMax`. Describe percolating down, which is more complicated.

(d) Describe an $O(\log n)$ algorithm for `insert`. Describe percolating up, which is more complicated.


3
78
84
11
14
9
25
68
22
77
17
49
75
28
39
28
56
13
21
76
60
15
16
29
10
12
45