Solutions to Problem Set 3

Solution to Problem 1

If the number of items greater than \(x\) is at least \(k\), then the \(k\)th largest item must be greater than \(x\). Otherwise if the number of items greater than or equal to \(x\) is at least \(k\), then the \(k\)th largest item must be equal to \(x\). Otherwise the \(k\)th largest item must be less than \(x\).

So, first we will count the number of items greater than \(x\), stopping the count at \(k\). Here is the pseudocode. This code assumes that the heap is stored as a binary tree.

```plaintext
count := 0;
S := {root};
while (count < k) and not empty(S) do
    begin
        Extract some element p from S;
        if (p <> nil) and (p^.key > x) then
            begin
                inc(count);
                Insert(S, p^.left);
                Insert(S, p^.right)
            end
    end
end
```

How much time does this code take? Well, count is always at most \(k\). Hence count is incremented at most \(k\) times. Hence the if condition is true at most \(k\) times. Hence at most \(2k\) items are inserted into \(S\) after it was initialized. Hence at most \(2k + 1\) items were ever in \(S\). Hence, because an item is extracted from \(S\) during each iteration of the while loop, the number of iterations is at most \(2k + 1\).

Let us implement \(S\) as either a stack or a queue. Then each Extract and Insert operation runs in constant time. Hence the code above will take time \(O(k)\).

The code above determines whether the number of items greater than \(x\) is at least \(k\). If that condition is not true, then we need to count the number of items greater than or equal to \(x\), stopping the count at \(k\). But we can make that count by modifying the code above: Replace “> \(x\)” with “\(\geq \) \(x\)”.

Solution to Problem 2

(a) Store the root in location 0. For the node at location \(i\), store its children in locations \(di + 1\) through \(di + d\). The parent of the node at location \(i\) will be at location \(\left\lfloor \frac{i-1}{d} \right\rfloor\).
(b) The complete $d$-ary heap of height $h - 1$ has exactly $(d^h - 1)/(d - 1)$ nodes. So, given $d$ and $n$, the height we desire is the largest integer $h$ such that

$$\frac{d^h - 1}{d - 1} < n,$$

which is equivalent to

$$d^h < (d - 1)n + 1.$$

Because both sides are integers, this inequality is equivalent to

$$d^h \leq (d - 1)n,$$

which is equivalent to

$$h \leq \frac{\log((d - 1)n)}{\log d}.$$  

Because we are looking for the largest such integer $h$, the height is

$$h = \left\lfloor \frac{\log((d - 1)n)}{\log d} \right\rfloor.$$  

Asymptotically, this height is $\Theta((\log n)/(\log d)).$

(c) An easy adaptation of the “floating-down” algorithm for binary heaps. The worst-case time is proportional to $d$ times the heap height, which is $\Theta(d(\log n)/(\log d))$.

(d) An easy adaptation of the “floating-up” algorithm for binary heaps. The worst-case time is proportional to the heap height, which is $\Theta((\log n)/(\log d))$.

(e) Increasing a key can make an item larger than its parent, in which case it must move higher up in the tree. The algorithm is very similar to insertion. The worst-case time is again $\Theta((\log n)/(\log d))$.

Solution to Problem 3 We will keep track of the (remaining) capacity of each bin that we have used. Imagine a “tournament” among these bins. That is, place the bins at the leaves of a binary tree, in order from first to last. (If it helps, create some extra bins of capacity 1 to make the number of bins a power of 2.) Each internal node will have a “capacity” $c$, that is the maximum capacity of its two children.

Now suppose we are ready to pack the next item. Start at the root of the tree. If the capacity of the left child is big enough to hold the item, then we go left; otherwise, we go right. Repeating this rule, we will end up at a leaf, corresponding to the first bin that can hold our item.

Next we need to update the capacities in the tree. The leaf (bin) found will decrement its capacity by the size of the item. Then we follow a path from this leaf to the root, updating the capacities so that each internal node has a capacity equal to the maximum of the capacities of its two children.

What is the running time of our algorithm? Well, the tree has height $O(\log n)$. Each insertion of a new item involves going down the tree from the root to a leaf, and then going back up to the root. Hence each insertion takes time $O(\log n)$. Thus to pack all $n$ items will take time $O(n \log n)$.  

2