Problem Set 3 Solutions

Problem 3-1. Heap Operations

For parts (a), (b), (c) and (d) of this problem, you may assume that the *input* heaps are nearly complete binary trees; for parts (a), (b) and (c) your *output* heaps need not be nearly complete. Note that the input heaps are not given to you in the form of arrays and your output heaps need not be in the form of arrays.

(a) Give an algorithm to merge two heaps of size m and n (assume $n \geq m$), into a single heap of size $(m+n)$ in time $O(\log n)$.

Solution: Call the two input heaps, i.e. two binary trees with the heap property, H_1 and H_2. Compare the minimum elements, i.e. the top elements, from H_1 and H_2. Denote the smaller element as x and the greater as y and without loss of generality, assume that x is in H_1. Remove x from H_1 and replace it with an element z from the bottom of H_1. Note that H_1 now satisfies the precondition for the Heapify procedure, namely the binary trees rooted at the children of the root are both heaps but the root may be larger than its children. Thus, we can run the procedure HEAPIFY on H_1. Note that the code in the book can be modified to run on an input that is a binary tree but perhaps not an array. This (also) takes $O(\log n)$ time. Now H_1 and H_2 are both heaps so we can join them in a binary tree rooted at x. Note that since x was the minimum element in H_1, it must be the smaller than the new root of H_1. Furthermore, since x is smaller than y, it is smaller than both it's children. Thus, the new binary tree rooted at x still satisfies the heap property.

(b) Suppose you are given a heap H of size m and n unordered elements, where $n \geq m$. What is the runtime to insert these n elements into the heap H? What is the total runtime required to build a second heap H' of size n and then merge H and H' into a single heap?

Solution: Inserting n elements into a heap of size m takes time:

$$\sum_{i=0}^{n} \log(m+i) < \sum_{i=0}^{n} \log(n+i) = O(n \log n)$$

Building H' would take $O(n)$ and from part (A) merging the two heaps would take $O(\log n)$. So, the second method would take time $O(n + \log n) = O(n)$.

(c) Give an algorithm to merge k heaps each of size n into a single heap of size nk in time $O(k \log n)$.

Solution: Perform the merging from part **(a)** on pairs of heaps, producing $k/2$ heaps of size $2n$. This will take $c \cdot k/2 \cdot \log n$ time. Repeating this process another step will take $c \cdot k/4 \cdot \log 2n$ time. The total amount of work done is given as:

$$\sum_{i=1}^{\log k} \frac{k}{2^{i+1}} \log(2^i n) = \frac{k}{2} \sum_{i=1}^{\log k} \frac{(i + \log n)}{2^i} = \frac{k}{2}(\sum_{i=1}^{\log k} \frac{i}{2^i} + \log n \sum_{i=1}^{\log k} \frac{1}{2^i})$$

Note that:

$$\sum_{i=1}^{\infty} \frac{i}{2^i} \leq 3$$

This is because for $i \geq 3$, each two terms in the sum are less than half of the two preceeding terms.

$$\frac{i+1}{2^{i+1}} + \frac{i+2}{2^{i+2}} \leq \frac{1}{2}(\frac{i}{2^i} + \frac{i-1}{2^{i-1}}) \Rightarrow i \geq \frac{8}{3}$$

Thus, the total amount of work done is $O(k \log n)$.

(d) Given a min-heap of n distinct elements and a real number x, give an algorithm to determine whether the kth smallest element is less than x in $O(k)$ time.
Hint: You do not have to extract the kth element. You only need to find its relation to x.

Solution: A solution is to start at the top of the heap and recursively visit the subheaps keeping a global count of the keys seen that have value less than x. If some key has value at least x, then we do not need to visit any of its children. If the global count reaches k, then we terminate after $O(k)$ steps and output "yes". If we have visited all the nodes with keys of value less than x and the counter is less than k, then we also terminate in $O(k)$ steps and output "no".
We give pseudocode for the procedure KTHLESSTHANX below. The algorithm takes as input a heap H, a real number x, and a positive integer k.

```
KTHLESSTHANX(H, x, k):
count ← 0.
RECURSE(H, x, k):
   if (count < k) and (H! = null):
         if (H.min < x):
            count ← count + 1.
            RECURSE(H.left, x, k).
            RECURSE(H.right, x, k).
return (count == k).
```

Problem 3-2. Political Machinations

Noted actor Ranier Wolfcastle decides to hold a "town hall" meeting as part of his campaign for Governor of California. His political advisor, Rover Karlson, is in charge of hand-picking an audience and asks you for help.

(a) Rover tells you that he only wants middle-class voters in the audience. He decides to eliminate anyone in the top or bottom $(1/k)$th income bracket. Unfortunately, you cannot find out anyone's actual income. You can only compare two people and see who makes more money. Given a list of n potential audience members, give an $O(n)$ time algorithm to find the people with the n/kth largest and smallest incomes.

Solution: Run the linear time SELECT algorithm to find the n/kth highest and lowest order statistics.

(b) Rover decides to sort the list of potential audience members into k equal-sized (to within 1) groups by income. Denote these groups as $G_1, \ldots, G_k$. The groups must have the property that $\forall i < j$, every person $g_a \in G_i$ has a lower income than every person $g_b \in G_j$. Rover doesn't care about the ordering within any particular group G_i. Give an $O(n \log k)$ time algorithm to produce k such groups.

Solution: We use a recursive algorithm GROUP, which takes an array and a value k, and works as follows: if $k = 1$, return. Otherwise, SELECT and PARTITION around the median of the array. Then call GROUP on the lower half of the array with $k/2$, and again on the top half with $k/2$.

To see that this works, note that after partitioning, all grades in the upper half of the array are greater than those in the lower half. By induction, the two recursive calls divide each half into $k/2$ groups, for a total of k groups. Finally, note that the base case satisfies the problem statement.

We now analyze the running time: the recurrence describing the algorithm's running time is $T(n, k) = 2T(n/2, k/2) + \Theta(n)$ because SELECT and PARTITION are linear-time. The base case of the recurrence is $T(n, 1) = \Theta(1)$ for any n. Therefore the recurrence tree does $\Theta(n)$ work at each level, and has $\lg k$ levels, for a total running time of $\Theta(n \log k)$.

Some students correctly observed that this solution is essentially an '‘early quitting" QUICKSORT, where the pivot is always chosen to be the median, and the algorithm terminates once the recursion depth reaches $\lg k$.

Problem 3-3. Set Operations

Suppose you are given two sets of integers, S and T. Let $m = |S|$ and $n = |T|$ and suppose $n \geq m$. You may assume any comparison or arithmetic operation (e.g. addition, multiplication) on two integers takes unit time.

(a) Give a deterministic algorithm for finding $S \cap T$ in $o(n^2)$ time.

Solution: Sort each set using Merge Sort. Merge the results, discarding elements that appear in only one of the sets. This algorithm runs in $\Theta(n \log n)$ time.

(b) Give a randomized algorithm for finding $(S \cap T)$ in expected $O(n)$ time.

Solution: Insert each element of S into a sufficiently large $O(n^2)$ hash table using a universal hash function. Then lookup every element in T in the hash table. For each element in T, if no collision occurs, discard the element. If a collision occurs, check to make sure the element is actually the same as the one in the hash table. If it is, output it. If not, discard. Each insertion and lookup takes constant expected time, so the entire algorithm will take $O(n)$ expected time.

Problem 3-4. **$d(v)$-ary Search Trees**

A $d(v)$-ary Search Tree ($d(v)$-ST) is a Search Tree in which each non-leaf node has $d(v)$ children. The function d maps nodes to integers.

(a) Let $d(v) = c$ for some integer c. Suppose we build a complete $d(v)$-ary Search Tree on n nodes. What is the height of the tree in terms of n and c? What is the runtime of a SEARCH operation?

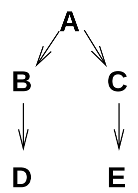
Solution: Each non-leaf node will have c children. The height of the tree will be $O(\log_c n) = O(\log n)$. At each level of the tree, we must search at most c nodes. Therefore, searching takes $O(c \log_c n) = O(\log n)$ in the worst case.

(b) Let $size(v)$ be defined as "the number of nodes in the subtree rooted at v, not including v". For a leaf node v, $size(v) = 0$. Suppose $d(v) = \lfloor\sqrt{size(v)}\rfloor$. The figure below depicts such a tree in which the function d has the following values:

$d(A) = \lfloor\sqrt{size(A)}\rfloor = \lfloor\sqrt{4}\rfloor = 2,$

$d(B) = d(C) = \lfloor\sqrt{size(B)}\rfloor = \lfloor\sqrt{1}\rfloor = 1$

$d(D) = d(E) = \lfloor\sqrt{size(D)}\rfloor = \lfloor\sqrt{0}\rfloor = 0$



What is the height of a complete $d(v)$-ary Search Tree with n nodes? What is the runtime of a SEARCH operation? For convenience, you may assume that $\sqrt{size(v)}$ is always an integer.

Solution: For a tree of size n rooted at v, the each subtree has size $(n-1)/d(v) = (n-1)/(\lfloor\sqrt{n-1}\rfloor)$. By using the Sloppiness theorem, an the height of the tree may be upper bounded by the following recurrence relation: $H(n) = H(\sqrt{n}) + 1$. Suppose that $n = 2^m$. By changing the variables of the recurrence, the height is equivalent to: $H'(m) = H(\frac{m}{2}) + 1$. The solution to this recurrence is $H'(m) = O(\log m) = O(\log\log n)$.

An upper bound of a recurrence relation for SEARCH is as follows: $S(n) = S(\sqrt{n}) + \sqrt{n}$. Using a change of variables as above, this is equivalent to $S'(m) = S'(m/2) + 2^{m/2} = \Theta(2^{m/2}) = \Theta(\sqrt{n})$. Thus the time to search is $O(\sqrt{n})$.