CS361 Homework #3 Solutions

1. Suppose I have a hash table with 50 locations. I would like to know how many items I can store in it before it becomes fairly likely that I have a collision, i.e., that two items get hashed to the same location. Assume that the hash function is random, and solve this problem in two ways:

   (a) Find the smallest value of n for which, if I store n items, the probability I *don't* get a collision falls below $1/2$. Do this by calculating this probability *exactly* for $n = 1$, $n = 2$, and so on.

   (b) Find the smallest value of n for which the *expected* (or average) number of colliding pairs is equal to or greater than 1. Do this by calculating this average exactly for the relevant values of n.

   Do these two methods give roughly the same answer? Can you explain why they are slightly different?

   Answer. The exact probability that there is no collision is the product of the probabilities, for each item, that it doesn't fall into any of the locations taken by the previous items. For the ith item, there are $50-(i-1)$ locations taken by the previous $i-1$ items, so this is

   $$\frac{50}{50}\frac{49}{50}\frac{48}{50}\cdots\frac{50-(n-1)}{50} \quad .$$

   The first value of n for which this is less than $1/2$ is $n = 9$, for which this probability is roughly 0.4655.

   The expected number of colliding pairs is the number of possible pairs of items, times the probability that a given one of them collides. This is

   $$\binom{n}{2}\frac{1}{50} = \frac{n(n-1)}{100} \quad .$$

   The smallest value of n for which this is greater than 1 is $n = 11$, for which the expected number of pairs is 1.1.

   These values are somewhat different because, first of all, setting the threshold at $1/2$ is arbitrary; secondly, even if we have a collision with probability $1/2$, the expected number of collisions could be just $(0+1)/2 = 1/2$.

2. Consider the following question:

> Input: a min-heap H containing a set of n numbers, an integer k, and a number x
>
> Question: does H contain k or more numbers which are smaller than x?

Show how to answer this question in $O(k)$ time; in other words, in an amount of time that grows linearly with k, and doesn't depend on n.

Answer. Our goal is to go looking for elements less than or equal to x. We will do this by starting at the root and exploring downward; since this is a min-heap, whenever we reach a node greater than or equal to x we don't need to explore that branch any further, since all its descendants are also greater than x. As soon as we find k nodes smaller than x, we halt and return "yes"; if our search ends after fewer than k nodes by hitting "bedrock," i.e., nodes greater than or equal to x, we return "no."

We can do this depth-first or breadth-first, but in either case we never need to explore more than $O(k)$ nodes of the heap. Here's pseudocode for a version that explores depth-first; to do it breadth-first, simply make `s` a queue instead of a stack.

```
are_there_k_smaller_than_x(k,x) {
  t = 0; // number found smaller than x so far
  stack s;
  push root onto s;
  while (t < k && s is not empty) {
    y = pop(s);
    if y < x {
      t++;
      push y's daughters onto s;
    }
  }
  if (t >= k) return "yes"
  else return "no"
}
```

3. A *d-ary heap* is a heap based on a balanced d-ary tree, where each node has d children (except that the bottom row may be incomplete).

   (a) How would you implement a d-ary heap in an array?

   Answer: if the array is labelled $a[1...n]$ where $a[1]$ is the root, the d children of $a[k]$ should be

   $$a[dk - d + 2], a[dk - d + 3], \ldots, a[dk + 1]$$

   This generalizes the implementation for a binary heap with $d = 2$, where the children of $a[k]$ are $a[2k]$ and $a[2k + 1]$. If we instead label the array $a[0...n - 1]$ where $a[0]$ is the root, the d children of $a[k]$ are

   $$a[dk + 1], a[dk + 2], \ldots, a[dk + d]$$

   which looks a little simpler.

   (b) How should `downSift` work in a d-ary heap?

   Answer: to call `downSift` on a node x, find the minimum of x's d children (this takes $d - 1$ comparisons). If x's key is larger than this minimum (which takes 1 more comparison), swap them. This takes a total of d comparisons.

   (c) Analyze the running time of `makeHeap`, `deleteMin`, and `insert` in terms of n and d. Find the dependence on d instead of just absorbing it into the constant hidden in Θ, and discuss whether they are faster or slower than in a binary heap.

   Answer: since the depth of the tree is roughly $\log_d n$ and each step of `downSift` takes d comparisons, `downSift`ing the root, which we have to do for `deleteMin`, takes $d \log_d n = (d/\log d) \log n$ steps at worst. Since $d/\log d$ is an increasing function of d, the worst-case running time of `deleteMin` is worse than for a binary heap.

   For `insert`, we need to `upSift` a new leaf. But this only takes one comparison for each step, so even if we make to go all the way to the root this takes just $\log_d n = (\log n)/(\log d)$ time. Since this is a decreasing funciton of d, `insert` is faster than in a binary heap.

   For `makeHeap`, the recurrence for makeHeaping a set of n nodes is

   $$f(n) = d \cdot f(n/d) + d \log_d n$$

   since we first `makeHeap` the d subheaps and then `downSift` the root. Since the driving term is $O(\log n)$ is small compared to the homogeneous solution $\Theta(n)$, this gives $f(n) = \Theta(n)$. But, to get the constant hidden in Θ, we need to think about the fraction of nodes at each level of the tree.

   Let's denote $f(h)$ as the fraction of the n nodes that are h levels above the leaves; $f(0)$ is the fraction of nodes that are leaves, $f(1)$ is the fraction of nodes that are one step above a leaf, and so on. In a

balanced d-ary tree, each level has d times as many nodes as the one above it, so

$$f(h+1) = f(h)/d \ .$$

which means that

$$f(h) = A/d^h$$

for some constant A. But, by definition all these fractions have to add up to 1, so in a large tree we can say,

$$1 = \sum_{h=0}^{\infty} f(h) = A \sum_{h=0}^{\infty} (1/d)^h = A \, \frac{1}{1-1/d} = A \, \frac{d}{d-1}$$

where we used the formula for a geometric sum. But then $A = (d-1)/d$, so

$$f(h) = \frac{d-1}{d} (1/d)^h \ .$$

Now, we saw before that each step of `downSift` takes d comparisons, to find whether we should swap keys with one of our daughters, and if so, which one. Since there are $f(h)n$ nodes at each level, and since we might have to `downSift` a node at level h up to h times before it finds its proper place, the total worst-case running time is

$$\sum_{h=0}^{\infty} f(h) n \cdot d \cdot h = nAd \sum_{h=0} h/d^h = n(d-1) \frac{1/d}{(1-1/d)^2}$$

where we used $A = (d-1)/d$ and the formula $\sum_{k=0}^{\infty} k r^k = r/(1-r)^2$ with $r = 1/d$. Simplifying the fraction, this becomes

$$n \frac{d}{d-1}$$

This decreases slightly (towards 1) as d increases — so `makeHeap` is faster for a d-ary heap than for a binary heap, but not by much.

4. Suppose I have a binary search tree. It starts out empty, and I insert three items into it. The shape of the resulting tree depends on which of the 6 possible orders I insert them in. Assume that the 6 possible orders are equally likely; then, give the possible shapes the tree can have, and calculate the probability for each one.

   Now, for each of these shapes, suppose that I search for one of the three items, where each of the three is equally likely. Calculate the average number of steps I need to find this item, i.e., its average depth in the tree, where the root has depth 0.

   Finally, take the average over the possible shapes, weighted by their probabilities, to get the average of this average: in other words, calculate the average depth of a random item in a random tree. If all goes well, you should get 1/3 times the average number of comparisons Quicksort needs to sort 3 items, or 8/9!

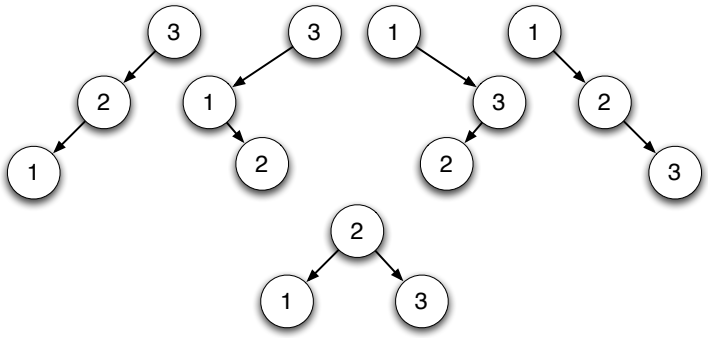

Figure 1: The five possible tree shapes with three items.

Answer. There are 5 possible tree shapes, shown in Figure 2. The four "stringy" ones each result from one of the 6 possible orders, and so they have probability 1/6. The balanced one results from two orders, namely $2, 1, 3$ and $2, 3, 1$, and so its probability is 1/3. For the stringy ones, the average depth is $(0 + 1 + 2)/3 = 1$, while for the balanced one it is $(0 + 1 + 1)/3 = 2/3$. Thus the total average depth is

$$4 \times \frac{1}{6} + \frac{1}{3}\frac{2}{3} = \frac{8}{9}$$

as promised.

5. Let's prove that an AVL tree with n nodes has depth $O(\log n)$. We will do this by solving the opposite problem: finding the smallest number of nodes that can cause an AVL tree to have a certain depth. Call an AVL tree "extreme" if every node except the leaves is unbalanced to the right, and let $f(d)$ be the number of nodes in an extreme tree of depth d. Then $f(1) = 1$, $f(2) = 2$, $f(3) = 4$, and $f(4) = 7$.

   Find a recurrence for $f(d)$ and solve it within Θ. Then argue that $n \geq f(d)$ and therefore $d \leq f^{-1}(n)$ where f^{-1} is the inverse function of f. End by proving that $d = O(\log n)$. What is the base of the logarithm? How much deeper are AVL trees than perfectly balanced binary trees?

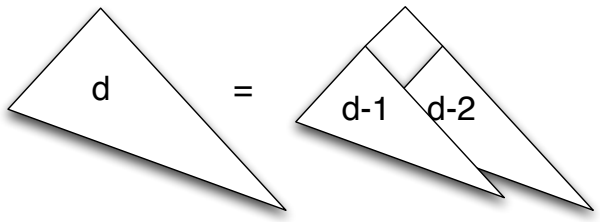

Figure 2: The recurrence for extreme AVL trees.

Answer. A little thought and some doodling reveals that the extreme tree of depth d consists of the extreme trees of depth $d-1$ and $d-2$. Adding the root, this gives the recurrence

$$f(d) = f(d-1) + f(d-2) + 1 \ .$$

Except for the tiny driving term of $+1$, this is the Fibonacci recurrence. Thus $f(d) = \Theta(\varphi^d)$ where $\phi = (\sqrt{5}+1)/2 \approx 1.618$ is the golden ratio, and inverting this gives $d = O(\log_\varphi n)$. (To be more precise, since $f(d) = \Theta(\varphi^n)$ we have $n \geq C\varphi^d$ for some constant C, and so $d \leq \log_\varphi(n/C) = \log_\varphi n - \log_\varphi C$.) Since $\log_\varphi n = \log_2 n / \log_2 \varphi \approx 1.44 \log_2 n$, this is about 44% deeper than a balanced binary tree.

Extra Credit: We analyzed the version of Quicksort in which the pivot is a randomly chosen element. We found that the average number of comparisons is

$$f(n) = 2n \ln n$$

We did this in the following way. Let x be the position in the list that the pivot ends up in, and let $P(x)$ be the probability of a given value of x. Then the recurrence for the average number of comparisons is

$$\begin{aligned} f(n) &= \underbrace{n-1}_{\text{partitioning}} + \sum_{x=1}^{n} P(k) \underbrace{(f(x-1)+f(n-x))}_{\text{recursion}} \\ &\approx n + 2\int_0^n P(x) f(x)\,\mathrm{d}x \end{aligned} \tag{1}$$

In the second line we assumed that $P(k)$ is symmetric, i.e., that $P(x) = P(n-x)$.

Now, if the pivot is random, its rank is equally likely to take any value from 0 to $n-1$, so every value of x occurs with equal probability $1/n$:

$$P(x) = \frac{1}{n} \text{ for all } 0 \le x < n$$

Then Equation (1) becomes

$$f(n) = n + \frac{2}{n}\int_0^n \mathrm{d}x\, f(x)$$

By substituting a solution of the form $f(n) = An \ln n$ into this equation and solving for A, we get $A = 2$ as before.

Now, an *improved* version of Quicksort uses the *median of three random elements* as the pivot. In this case, $P(x)$ is the probability distribution of the *median of three random numbers* in the unit interval $[0, 1]$. Calculate $P(x)$ by asking, given a random number x between 0 and n, what is the probability that if I choose two more random numbers y and z, then x is greater than y and less than z? Then, how many ways are there for this to happen? (You might want to check your work by confirming that the total probability $\int_0^n P(x)\,\mathrm{d}x$ is 1.) Intuitively, rather than being uniform, $P(x)$ should be peaked at $x = n/2$.

Finally, use this expression for $P(x)$ in Equation (1) and again try a solution of the form $f(n) = An \ln n$. You should be able to derive that A is now somewhat smaller than 2, indicating that this method of choosing the pivot improves the constant in the number of comparisons.

Answer. In the median-of-three version, the probability distribution of the median of three random numbers between 0 and n is (when n is large)

$$P(x) = \frac{6x(n-x)}{n^3}$$

To see this, note that once we choose x, the probability that another random number y is less than x is x/n, and the probability that a random number z is greater than x is $(n-x)/n$. The probability that both of these things occur, i.e., that $y < x < z$, and that we chose x in the first place, is $x(n-x)/n^3$. But, there are 6 ways for this to happen: the median could be any of x, y, or z, and we can switch which of the other two is greater and which is smaller.

Then, with a little help from my computer, I get

$$2\int_0^n P(x) \cdot Ax \ln x \, \mathrm{d}x = 2\int_0^n \mathrm{d}x \, \frac{6x(n-x)}{n^3} \, x \ln x = An \ln n - \frac{7}{12} An$$

and so Equation (1) becomes

$$An \ln n = n + An \ln n - \frac{7}{12} An$$

and solving for A gives $A = 12/7 \approx 1.7$, roughly 14% faster than the original version. Indeed, taking the median of three elements is often used in practice.

Even more extra credit: What happens if we choose the pivot by taking the median of 5, 7, ... elements? I leave this to you...