CSE 347 Analysis of Algorithms March 6, 2019

Lecture 14 Summary

Instructor: Brendan Juba

1. In *average-case analysis*, we assume that the inputs to our algorithms are drawn independently and at random from a probability distribution, and that we know what this probability distribution is.
   (a) For example, in a dictionary data structure problem, the inputs are integers from some large range 0–$(N-1)$, and they are drawn independently and at random from a probability distribution on this range.
   (b) But, what probability distribution is appropriate? Uniform? Binomial? It depends on the application, and unfortunately the outcome of the analysis will depend on what distribution we choose. This is the Achilles' heel of average-case analysis. Nevertheless, let's choose the uniform distribution today.
   (c) It will essentially be enough for most purposes to consider obtaining n distinct inputs in a random order.
      i. As long as we don't draw the same element twice, these distinct elements will be in a random order.
      ii. We show that as long as n is asymptotically less than $\sqrt{N \ln \frac{1}{1-\delta}}$, that the elements will be distinct with probability $1-\delta$ (for any $\delta > 0$ we wish). Since N is assumed to be very large, this won't bother us.

2. A naïve implementation of a binary search tree will have $O(\log n)$ time lookup and insert operations on average when the inputs are chosen uniformly at random
   (a) We analyze the time to lookup a random element in the tree by calculating the total time to lookup all elements in the tree (equiv., the total number of recursive calls made to insert the elements into the tree) and dividing by the number of elements in the tree.
      i. This gives a nice recurrence: observe that the root is in a uniform-random position among the n elements in the tree. Given it appears in a fixed position j in the tree, we know that every element first passes by the root, and then we pay the cost to build the left subtree on $j-1$ elements, and the right subtree on $n-j$ elements. We just average over the n possible values j could take.
      ii. Writing out the above recurrence, we observe that the difference between the cost for inserting n elements and $n-1$ elements, appropriately rescaled, can be written in terms of the cost for inserting $n-1$ elements and quantities depending only on n (most of the common terms in the expression cancel).
      iii. So we can iteratively substitute this expression on smaller values for n to solve the recurrence.
   (b) We can similarly analyze the time to lookup a random element *not* in the tree by adding the total time to reach each of the $n+1$ null children in the tree and dividing by $n+1$. This yields a similar recurrence to the above, that is solved similarly.

(c) The time to insert a random element, distinct from the first n, is the same as the time to lookup a random element not in the tree.

(d) Note that if the inputs were sorted instead of appearing in random order, the naïve tree takes time n to insert – so our analysis is totally invalid in such a case.

3. Hashing with a table of size m also obtains $O(1)$ lookup and insert operations after n elements as long as we use a table of size $m = \Omega(n^2)$.

   (a) For simplicity we assume that m divides N; then the simple hash function $h(x) = x \bmod m$ takes value i with probability $1/m$, i.e., is a uniform random element in the range 0-$(m-1)$.

   (b) So, by the calculation we did in the beginning of lecture, as long as n is less than $\sqrt{m \ln \frac{1}{1-\delta}}$, the probability that *any* of the elements collide is at most δ. When there are no collisions, we can use any simple method (e.g., linked lists) to resolve collisions and still obtain $O(1)$-time lookup and insert.

   (c) This relies on the inputs being taken uniformly at random, again, and does not address what happens in general, what happens when there are more than $\sqrt{m}$ inserts, etc.

4. If only we could make our inputs random again!

   (a) Simple algorithms get excellent running times on uniform random inputs.

   (b) We will see: by making random choices, i.e., using *randomized algorithms*, we can ensure that arbitrary (worst-case) inputs obtain the kind of behavior we need

   (c) In particular, if we randomly reorder the inserts to a naïve search tree, we obtain a tree that is balanced on average; next time, we will see that it is actually balanced *with high probability,* so that *worst-case* insert and lookup operations take $O(\log n)$ time "with high probability" (i.e., $1 - \delta$ for any $\delta > 0$).

   (d) We will also see that we can do this reordering as elements arrive by assigning a uniform random "priority" to the element, and ensuring that we obtain the same tree as if we had inserted in this priority order. As long as we don't see the same priority twice (make fewer than $\sqrt{N}$ inserts), this will be a random ordering, which is enough.