CSE 347 Lecture 18

Randomized Algorithms I

1. Treaps: randomized search trees
2. Quicksort

Average-case analysis had the pleasing consequence that very simple data structures achieved great performance. Unfortunately, the assumption that the inputs are random and independent is often unrealistic — if the inputs we receive have been "helpfully" sorted, the naive search tree will have a single branch of length n, and correspondingly half of the elements take $\frac{n}{2}$ steps to reach.

Intuitively, if we could only "mix the elements up again," we could use our simple search tree algorithms, and still achive good performance. The key is that we make <u>random</u> choices about the ordering of the elements. This is a <u>randomized algorithm</u>. And, indeed, if we can generate random numbers, we can create fast, simple search trees this way.

The main difference is now that we no longer assume that insertions and queries are uniformly random, so we need to argue more carefully that it's unlikely that <u>any</u> node of the tree has depth greater than

~log n. That is, we'll show:

Theorem If $x_1, x_2, \dots, x_n$ are inserted into a binary search tree in random order, with probability $1-\delta$ every x_i has depth $O(\log \frac{n}{\delta})$ for any $\delta > 0$.

Proof: Consider any x_i; we'll first show that the probability that x_i has depth $O(\log \frac{n}{\delta})$ is at least $1 - \frac{\delta}{n}$. Then we'll invoke the "union bound" that if we have "bad" events $B_1, \dots, B_n$ (i.e., B_i = "x_i has depth $> c \cdot \log \frac{n}{\delta}$") then the probability that any B_i occurs is at most $\sum_{i=1}^{n} \Pr[B_i]$. Since here $\Pr[x_i \text{ has depth} > c \cdot \log \frac{n}{\delta}] < \frac{\delta}{n}$ (we'll show), we conclude that $\Pr[\text{any } x_i \text{ has depth} > c \cdot \log \frac{n}{\delta}] < \delta$, which will finish the proof.

We can generate a random ordering of the elements as follows: we first pick one out of n elements to be the first element, one out of the remaining n-1 elements to be second, and so on: this produces any fixed ordering with probability $\frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{1}{n-2} \cdots \frac{1}{2} \cdot \frac{1}{1} = \frac{1}{n!}$, so it is indeed a uniform distribution on the ordering. Now, let's consider the elements remaining in the subtree containing x_i after each insertion (elements in other subtrees can't appear as ancestors of x_i, so we can ignore them). In particular, let's consider the number of elements greater than x_i and the number of elements less than x_i. Notice, if

$x_j\, x_{j+1} \cdots \underbrace{x_{i-1}}_{1/2}\, x_i \underbrace{x_{i+1} \cdots}_{1/2}\, x_k$ an element is chosen within the half of the range closer to x_i, then the half of the range further from x_i is

placed in a different subtree. And, this occurs with probability $1/2$ (among the rounds in which we insert an element from this range) How many times can this occur before x_i is the only element remaining? There are at most $n-1$ elements greater than x_i, and after we cut in half $\log_2 n$ times, the number remaining is at most $\frac{n-1}{2^{\log n}} = 1 - \frac{1}{n}$, but it's an integer, so it must be 0. The same argument holds for the elements less than x_i, too, so it takes at most $2\log n$ choices of these "good splits". Now, we only need to show that for some C, after choosing $C \log \frac{n}{\delta}$ elements from x_i's subtree, we get fewer than $2\log n$ "good splits" with probability less than $\frac{\delta}{n}$. But, the distribution of whether or not an element is a "good split" is an independent, fair coin toss, so we are simply interested in how many 'heads' (="good splits") we get when we toss k coins. We can use the following standard inequality:

Theorem (Simplified Chernoff Bound): The probability of obtaining fewer than $(1-\varepsilon)pk$ heads when tossing a coin of bias p k times is at most $e^{-\frac{1}{2}\varepsilon^2 pk}$.

So, in particular, if we toss the coin $k = 16 \log_2 \frac{n}{\delta}$ times, $p = \frac{1}{2}$, and taking $\varepsilon = \frac{1}{2}$ gives that the probability of getting fewer than $(1-\frac{1}{2}) \cdot \frac{1}{2} \cdot 16 \log_2 \frac{n}{\delta} = 4\log_2 n + 4\log_2 \frac{1}{\delta}$ heads is at most $e^{-\frac{1}{2}(\frac{1}{2})^2 \frac{1}{2} 16 \log_2 \frac{n}{\delta}} = e^{\log_2 \delta/n} = e^{\ln \delta/n / \ln 2} = (\frac{\delta}{n})^{1/\ln 2} < \frac{\delta}{n}$ (note: $e > 2$ so $\frac{1}{\ln 2} > 1$)

Thus, the probability that x_i has more than $16 \log_2 \frac{n}{\delta}$ ancestors is $< \delta/n$, which is what we needed to show. ▨

So if we are given all of our data up-front, we can

build a tree of depth $O(\log n)$ using the simple algorithm just by randomly ordering the data. Next, we'll show that we can achieve the same balance even if the data arrive one at a time using a slightly more clever algorithm called a "treap."

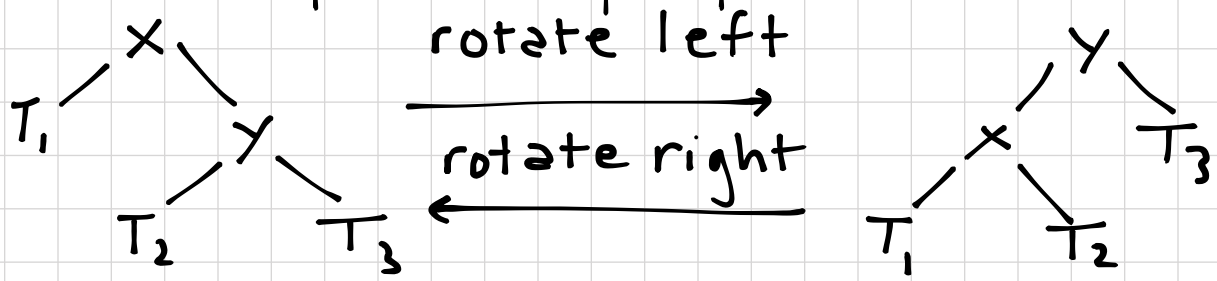
The idea is that when we insert an item, we assign it a random priority from some large range $0-(N-1)$ independently. We argued last time that as long as $n < O(\sqrt{N\delta})$, then with probability $1-\delta$, the priorities will all be distinct. So, if we can obtain the same tree as we would have gotten by inserting the elements in order of increasing priority, then since this is a random ordering of the elements (when the priorities are distinct) we'll obtain our balanced tree. Observe that each child's priority is then greater than that of its parent. It turns out that this determines the tree:

Lemma: If $x_1, \ldots x_n$ are assigned distinct priorities $p_1, \ldots, p_n$, there is a unique binary search tree in which each parent's priority is less than its children's.

Proof? Suppose WLOG that $p_1 < p_2 < \cdots < p_n$. Observe that if we consider the first i elements, that since none of the final $n-i$ can be an ancestor of the first i, that the first i must be contained in a connected component containing the root of the tree. We now argue by induction on i that this component is uniquely determined. For $i=1$, it must simply

be the root. Supposing that the subtree containing i is uniquely determined, we now observe that $i+1$ must be a child of the unique tree containing the first i. But, since this is a binary search tree, its location is uniquely determined as the location it would be inserted ▨

So, how do we maintain the ordering by priorities? Roughly: we insert a new element as usual first, and then "float" it up to its proper location using rotations:



Observe: since $T_1 < X < T_2 < Y < T_3$ in both trees, both are binary search trees. Moreover

Lemma: If y is the only element with a higher priority ancestor before rotate-left, only y can have a higher priority ancestor afterwards (and similarly for x and rotate-right).

Proof: We're given that priority(x) > priority(y), so after the rotation, x has higher priority than its only new ancestor, y. Moreover, since T_1 already had x as an ancestor, every node in T_1 had priority greater than x, and hence also than y as well. We added no other ancestor relationships. ▨

Once a node reaches the root, it can't have a higher-priority parent, so we must eventually restore the ordering property. Since the tree has height $O(\log n)$, inserts (and lookups) still take $O(\log n)$ time.

Quicksort

Here is a well-known sorting algorithm that is very fast in practice, that uses the same analysis:

```
Quicksort (A[1...n], i, j)
  Choose p at random in the range i, ..., j
  Initialize r ← j
  For l = i, ..., r
    If A[l] ≥ A[p]
      decrement r until A[r] < A[p] or l = r
      If A[r] < A[p] Swap A[l] and A[r]
      If l = p put p ← r
  Swap A[p] and A[l]
  if i < l-1  Quicksort(A, i, l-1)
  if j > l+1  Quicksort(A, l+1, j)
```

Notice, in the recursion tree for Quicksort, if we label the node with the "pivot" element A[p], the first call only contains elements less than A[p] and the second only contains elements greater than A[p], so this is a binary search tree on A, in which the elements were chosen in random order. Thus, its height is $O(\log \frac{n}{\delta})$ with probability $1-\delta$; since the array is partitioned at each level and the algorithm takes time $O(j-i)$ on the range $i-j$, each level takes $O(n)$ time in total. Thus, the algorithm runs in time $O(n \log \frac{n}{\delta})$ with probability $1-\delta$. Also, each element is examined once at each of its ancestors,

so the sum of the depths of each node is another way of writing the total time; we saw last time that the sum of the depths is $C(n) = (n+1)(2H_{n+1} - 3) + 1$ on average when the order is random. Thus, the running time is $O(n \log n)$ on average as well.