**CSci 4041: Algorithms and Data Structures**
**Optimal Binary Search Trees (OBSTs)**
**Part 1**

*Last revision March 24, 2020*

Here's the agenda.

What's an optimal binary search tree?
What does *optimal* mean?
Why an obvious algorithm won't work.

If I were giving this lecture in person, all I would do is say things that are in Cormen on pages 397–399, but explaining them in a little more detail. That's all I'll do here too, but because of our current situation, I'm writing instead of talking. Also, this lecture is prepared with TEX instead of my usual HTML-making program. This is because there's lots of math this time, and my HTML program can't handle that.

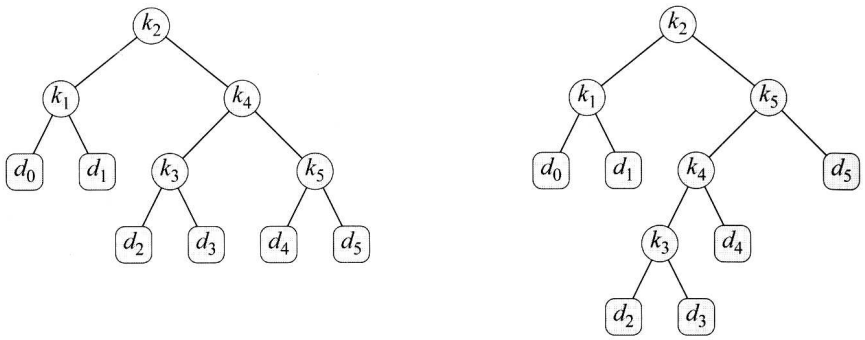
**What's an OBST?** In the binary search trees (BST's) we've seen so far, all the keys have the same probability, so there's an equal chance that we'll find any given key when we search for it. This is unlikely to be true in many cases. We'd like to consider BST's in which the keys have probabilities, so some keys are more likely to be found than others. BST's like these are called *optimal binary search trees,* or OBST's for short. (We'll have more to say about that word *optimal* later.)

Here's how to define OBST's formally. Suppose we have a sequence $K = \langle k_1,\ k_2\ \cdots,\ k_n\rangle$ of $n$ keys. They're called *success keys* because they're the ones that we might search for successfully. We don't care what the success keys are, except that they're totally ordered. We also don't care about their values, because they can be anything. We assume that the success keys are distinct, and in strictly ascending order, so $k_1 < k_2\ \cdots\ < k_n$. Finally, for each success key $k_i$, there's a probability $p_i$ that we'll end up at a node containing it when we search an OBST.

We may also search for a key that's *not* in an OBST. In an ordinary BST, if we search unsuccessfully, looking for a key that isn't there, then we end up at an empty subtree. It will simplify things if we assume this can never happen in an OBST. Instead, unsuccessful searches always end up at external nodes that contain *failure keys* (Cormen calls them *dummy keys*).

The sequence $D = \langle d_0,\ d_1,\ d_2\ \cdots,\ d_n\rangle$ contains all the failure keys. There are $n+1$ of them: we always have one more failure key, $d_0$, than we have success keys. We don't care what the failure keys are, except that they're totally ordered, and we don't care about their values. We assume the failure keys are distinct, and in strictly ascending order, so that $d_0 < d_1 < d_2\ \cdots\ < d_n$. For each failure key $d_i$, there's a probability $q_i$ that we'll end up at a node containing it when we search an OBST.

For example, the following two BST's, a and b, are stolen from page 398 of Cormen. They show success nodes with keys $k_1$ through $k_5$, and failure nodes $d_0$ through $d_5$.




(a) (b)

All success keys are in internal nodes (nodes with children) and all failure keys are in external nodes (nodes without children). Also, since there's always one more failure node than there are success nodes, there are

no empty OBST's. Instead, an empty OBST is represented as a tree with no success nodes; it has only one failure node.

As suggested by these trees, the failure keys represent sets of keys that are not success keys. The failure key $d_0$ represents all keys less than $k_1$, and the failure key $d_n$ represents all keys greater than $k_n$. If $1 \le i \le n-1$, then the failure key $d_i$ represents keys between (but not including) $k_i$ and $k_{i+1}$.

Taken together, the probabilities of the success and failure keys sum to 1. This means that the success and failure keys are the only ones we need to care about. It may also be useful for solving certain mathematical problems about OBST's, like possibly those on an upcoming assignment (hint).

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

As a result, there are some things we can do with BST's that we can't do with OBST's. We can't add a key (and its value) to an existing OBST, because then we'd have to adjust the probabilities of the other keys so they still sum to 1 after the addition. We can't delete a key (and its value) either, because then we'd have to adjust the probabilities of the other keys so they still sum to 1 after the deletion. It's not obvious how to do either of those things. All we can do is make an OBST that has a given set of keys. We'll soon see an algorithm for that.

**What's optimal?** Before we can talk about an algorithm that makes optimal binary search trees, we must first know what *optimal* means. If someone tells you that an algorithm, or a data structure, is optimal, then you should ask: optimal with respect to what? By definition, an OBST is said to be *optimal* because it minimizes the sum of the expected costs of finding all its keys, both its success keys and its failure keys. The relevant words in that sentence are *cost* and *expected*.

Let's deal with *cost* first. Remember how to search a BST? We start at the root, comparing its key with the key we're looking for. Depending on the result of that comparison, we move one level down the tree, either left or right, and repeat the process. We stop when we find a node with the key we want, or when we run out of nodes. (In an OBST, we never run out of nodes—we stop when we reach a failure node.)

We determine the cost of a search by counting how many key comparisons it makes. Since we make a key comparison every time we visit a node, the cost of finding a node is just the number of nodes on the path from the root to that node. Now, suppose we're searching a tree $T$ for a key $k$. Cormen defines $depth_T(k)$ to be the number of links on the path from the root to a node containing key $k$. The number of nodes on that path is therefore $depth_T(k) + 1$. This is also the cost of finding the key $k$.

Now we'll deal with the word *expected*. Suppose $X$ is a random variable, and $X(s)$ is its value in some situation $s$. (Random variables aren't random, nor are they variables, they're really just functions in disguise.) Also suppose that the set of all possible situations is $S$, and the probability that $s$ will occur is $Pr\{s\}$. Then the *expected value* of $X$ is $E[X]$, defined like this.

$$E[X] = \sum_{s \in S} X(s)\ Pr\{s\}$$

Recall that we used a similar definition when we analyzed hash tables.

Let's put the pieces together. Suppose that $S(T)$ is a random variable: the cost of searching the tree $T$. (Cormen uses a different, and in my opinion uglier, notation than mine.) Then we can write its expected value $E[S(T)]$ like this.

$$E[S(T)] \ = \ \sum_{i=1}^{n} (depth_T(k_i) + 1)\ p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1)\ q_i$$

Here the situations we're interested in are when we find specific keys, and we're interested in finding not only the success keys, but also the failure keys. For each key, we multiply its cost by the probability that we'll find it.

At this point we need to do some algebra. Cormen's usual habit is to omit some steps, but I'll show more of them here, at the risk of being boring or obvious. (I'm less likely to make mistakes in algebra while

writing than I am while lecturing, but not much less, so beware.) We first distribute the multiplications over the additions, and split the resulting summations into two parts.

$$= \sum_{i=1}^{n} depth_T(k_i)\ p_i + \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} depth_T(d_i)\ q_i + \sum_{i=0}^{n} q_i$$

Then we'll regroup, so the two sums of probabilities are together. Of course addition associates and commutes.

$$= \left(\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i\right) + \left(\sum_{i=1}^{n} depth_T(k_i)\ p_i + \sum_{i=0}^{n} depth_T(d_i)\ q_i\right)$$

But the first parenthesized term is the sum of all the probabilities. We know that's 1, so we get this.

$$E[S(T)] \ = \ 1 + \sum_{i=1}^{n} depth_T(k_i)\ p_i + \sum_{i=0}^{n} depth_T(d_i)\ q_i$$

By definition, a binary search tree $T$ is *optimal* if its nodes are arranged so $E[S(T)]$ takes on a minimum value. (There may be several ways to do this.) We call such a tree an *optimal binary search tree*, or OBST for short.

But wait a minute. Is the tree really optimal? We've just *assumed* that if we minimize $E[S(T)]$, we'll get an optimal tree $T$. It seems plausible that $T$ will be optimal—but how do we know for sure? In particular, how do we know that if we minimized some other function of $T$, that we wouldn't get an even *better* tree? All we've really shown here is that some trees are better than others. Cormen doesn't address this, and I won't either, but it's something to think about if it amuses you.

**Why the obvious algorithm won't work.** Our goal here, in part 2 of this lecture, is to find an algorithm that takes a sequence of success keys $K$, a sequence of failure keys $D$, and probabilities for each key, then makes an OBST out of them. To do that, we'll use the technique of *memoization* we discussed before, along with some tricks from dynamic programming.

At this point, some students may wonder why we can't use a much simpler algorithm. For example, maybe we could write an algorithm that simply generates all possible BST's, computes an expected cost for each one, and then prints the ones having the smallest expected costs at the end. An algorithm that generates possible solutions and tests them is (unsurprisingly) called a *generate-and-test* algorithm. Generate-and-test algorithms are used in some areas of artificial intelligence, but that's beyond the scope of this course.

A generate-and-test algorithm won't work for finding OBST's. This is because there are just too many trees to generate. Suppose that $b_n$ is the number of distinct binary trees with $n$ nodes. In an exercise on page 307, Cormen claims that $b_n$ can be defined like this.

$$b_n = \frac{4^n}{\sqrt{\pi}\ n^{3/2}}\ (1 + O(1/n))$$

We don't have to care how $b_n$ was derived. The important thing is that there are exponentially many binary trees with a given number of nodes, and the number of those trees grows very fast as $n$ grows larger.

How fast does it grow? By some estimates, there are about $10^{80}$ elementary particles (electrons, neutrons, and protons) in the visible universe. They're what you see when you go outside on a dark clear night and look up at the sky. Now, suppose we want to make a list of all binary trees with $n$ nodes. How big does $n$ have to be before that list doesn't fit in the visible universe, even if we could somehow represent each tree as a single elementary particle?

I wrote a little Python program to determine this. Python can perform integer arithmetic with arbitrary numbers of digits, so numbers with 80 digits and more are no problem for it. It turns out that $b_{137} \geq 10^{80}$, assuming my program has no errors. In other words, the universe isn't big enough to hold a list of all possible binary trees with 137 nodes. That's not a particularly large tree. We clearly need a better algorithm than generate-and-test, even if we generate only one tree at a time.