CSci 4041: Algorithms and Data Structures
Optimal Binary Search Trees (OBSTs)
Part 2

Last revision March 27, 2020

Here's the agenda for this lecture.

Balance and height in OBST's.
An overview of the OBST algorithm.
An unusual representation for trees.

Specifically, I'll clarify how a BST is different from an OBST. Then I'll say some general things about how the OBST-making algorithm works. We'll see that algorithm in a later lecture. By the way, it's easily the most complicated algorithm we'll see in this course.

Balance and height. Last time we saw a way to compute $E[S(T)]$, the expected value of the cost of searching a BST T. We also saw that an optimal BST is one that minimizes $E[S(T)]$. This has some possibly unexpected consequences. It means that a good BST is not necessarily an OBST, and vice versa.

Look at the two BST's on top of page 398 in Cormen. Tree (a) has smaller height, and is better balanced, than tree (b). However, (b) is optimal, but (a) is not. This is because (a) has $E[S(T_a)] = 2.80$ and (b) has $E[S(T_b)] = 2.75$. Also note that the keys with the largest probabilities are not necessarily at the top of an OBST. For example, in tree (b), key k_5 has the largest probability of its keys, but key k_2, with a smaller probability, is at the root.

We can conclude that all the nodes in T and their depths contribute to the overall value of $E[S(T)]$. An algorithm that makes OBST's must take all of them into account. This fact might be used in a question on an assignment, so beware.

An overview of the OBST algorithm. As we saw last time, the algorithm that makes OBST's takes a sequence of keys $\langle k_1, k_2 \ \cdots, \ k_n\rangle$ as its input, where $n > 0$. It doesn't know the values of those keys, and it doesn't need to know them. All it knows is that the keys are sorted in strictly increasing order, so that $k_1 < k_2 \cdots < k_n$. It takes the keys and somehow organizes them into an OBST.

How does it do that? It chooses one of the keys k_r to be the root of the OBST, where $1 \leq r \leq n$. Now, since all the keys are sorted, the keys in the root's left subtree must be in the sequence $\langle k_1, k_2 \ \cdots, \ k_{r-1}\rangle$, since they're less than the root key k_r. Similarly, the keys in the root's right subtree must be in the sequence $\langle k_{r+1}, k_{r+2} \ \cdots, \ k_n\rangle$, since they're greater than the root key k_r. Both of these result from the BST property.

It then applies the same procedure recursively to the keys in the sequence $\langle k_1, k_2 \ \cdots, \ k_{r-1}\rangle$. This gives us the OBST's left subtree. It also applies the procedure recursively to the keys in the sequence $\langle k_{r+1}, k_{r+2} \ \cdots, \ k_n\rangle$. This gives us the OBST's right subtree. Since we know its root, its left subtree, and its right subtree, we can use these to make an entire OBST.

The recursion stops when a sequence of keys becomes empty. An empty sequence is one like $\langle k_i, k_{i-1}\rangle$, whose first key has a subscript greater than that of its second key. There are no keys lying between them, for the same reason that there are, say, no integers greater than 2 but less than 1. When the algorithm tries to make a tree from an empty sequence like this one, it uses a node containing the failure key d_{i-1}, which becomes an external node in the OBST. That's what the failure keys are for.

We can summarize this recursive algorithm below in pseudocode. This algorithm isn't in Cormen. I made it up myself, to explain what's going on here. The parameters i and j are subscripts of the k's, so the call CURSY-OPTIMAL-BST(1, n) returns a tree using keys k_1 through k_n.

```
CURSY-OPTIMAL-BST($i$, $j$)
    if $i > j$
        return $d_{i-1}$
    else
        $r$ = a subscript between $i$ and $j$
        $l$ = CURSY-OPTIMAL-BST($i, r-1$)
        $r$ = CURSY-OPTIMAL-BST($r+1, j$)
        return a BST with root $k_r$, left subtree $l$, and right subtree $r$
```

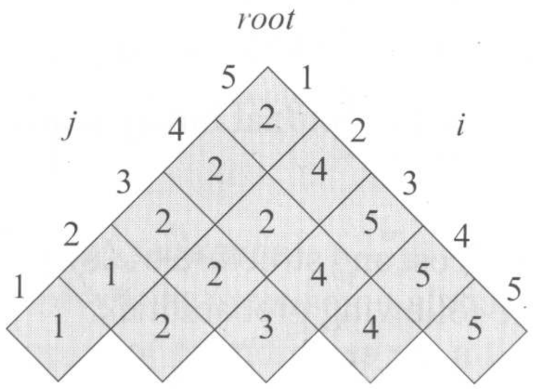
Now, suppose we have some way of choosing r in line 05 so that the resulting BST in line 08 is always optimal. Then we could use this algorithm to make an OBST. However, there's no way to choose r like that, so this algorithm is a fake. Maybe it should have been called FAKE-OPTIMAL-BST instead.

One way to choose r might be to turn CURSY-OPTIMAL-BST into a backtracking algorithm. It might choose r arbitrarily somehow, then test if it got an optimal BST as a result. If it didn't, then it would undo the choice of r, and choose r in a different way instead.

Unfortunately, we can't choose r by backtracking, because then we'd get an algorithm that generates too many trees. The real algorithm must therefore do something different from this. However, some parts of the fake recursive algorithm will help explain how that can be done.

An unusual representation for trees. When we get to the real algorithm, we'll find it uses an array representation for BST's that may be unfamiliar. It uses a two-dimensional integer array called *root*, where $root[i, j]$ is the subscript of the key at the root of the BST that contains keys k_i through k_j.

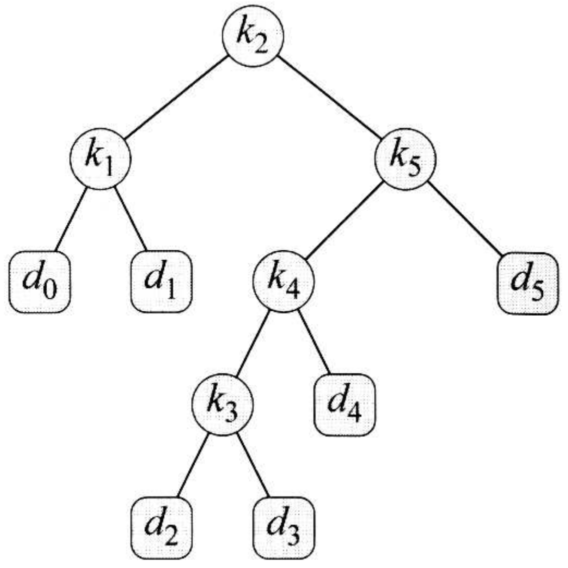
Huh? An example may make things clearer. Page 403 of Cormen shows a possible *root* array, which I'll also show here. We'll see later why it's tipped on its side. Its i indexes run down the right side of the array, and the j indexes run down the left side.



Let's see if we can reconstruct the BST from *root*. The entire tree contains keys k_1 through k_5, so the root of the tree must be given by $root[1, 5] = 2$. That means k_2 is the key at the root. If k_2 is the root, then its left subtree has only the key k_1, so $root[1, 1] = 1$. That means k_1 is the root of the left subtree. Similarly, the root's right subtree has keys k_3 through k_5, so $root[3, 5] = 5$. That means k_5 is the root of the right subtree. Here's a summary that tells us what the entire tree looks like.

$$
\begin{aligned}
root[1, 5] &= 2 \\
root[1, 1] &= 1 \\
root[3, 5] &= 5 \\
root[3, 4] &= 4 \\
root[3, 3] &= 3
\end{aligned}
$$

And here's the tree itself, from page 398 of Cormen. We've seen it before. Sure enough, k_2 is the root of the subtree containing k_1 through k_5, and k_5 is the root of the subtree containing k_3 through k_5, etc. Make sure you know how this works, because a future homework question will be about it.



A tree represented like this, as a *root* array, will be the output of the OBST algorithm that we'll see later.