CSCI 4041: Algorithms and Data Structures
Unbalanced Binary Search Trees

Last revision March 19, 2020

Here's the agenda for this lecture.

Binary search trees (BST's).
Definition of BST's.
The BST property.
Finding a value in a BST using BST-GET.
Changing a value in a BST using BST-PUT.
Run times for BST-GET and BST-PUT.
How BST's fail, and how to fix them.

In the next week or so, we'll discuss AVL trees (these aren't in Cormen) and optimal binary search trees (in Cormen on pages 397–403). Both are special kinds of binary search trees (BST's). That means we need to talk about BST's first. If you've already taken a data structures course (and if you haven't, then you shouldn't be in my class) then much of this material will be review. The Early section has already heard this lecture, on the Thursday before Spring Break, but the Late section has not, so I need to post it here.

Binary search trees. Cormen (286–307) discusses BST's, but his are different from mine, and are more complicated than what we need for this course. You might want to read Cormen for some background about BST's, but you don't have to know what he says about them.

Here's what you do need to know. A BST is a data structure that maps a set of *keys* to a set of *values.* Given a key, we can get its value, we can change its value, and we can delete the key and its value (but we won't discuss deletion here). BST keys are *totally ordered,* which means that for any two keys k_1 and k_2, exactly one of the following is true.

$k_1 < k_2$
$k_1 = k_2$
$k_1 > k_2$

For example, integers, real numbers, and strings are totally ordered in this way, so they could be BST keys. Complex numbers and sets are unordered, so they can't be BST keys. There are also things outside mathematics that are unordered, such as Human preferences. For example, I like oranges better than bananas, and I like bananas better than apples—but I don't know if I like oranges better than apples.

For our purposes, a BST is a linked binary tree, made from nodes with four slots. If p points to a node in a BST, then the slot $p.key$ is a key, and the slot $p.value$ is the value that corresponds to $p.key$. The slot $p.left$ points to p's left subtree, and the slot $p.right$ points to p's right subtree. Both the left and right subtrees are also BST's. And any BST can be NIL, which means it's empty, with no nodes.

The BST property. BST's satisfy the *BST property.* It says that if k is the key at the root of some subtree, then all the keys in the left subtree are strictly less than k, and all the keys in the right subtree are strictly greater than k. There is no such restriction on values—they can be anything. (Cormen defines the BST property in a different way than I just did, but ignore him.)

For example, Fig. 1 shows a BST that satisfies the property. Its nodes are drawn as circles, its keys are integers, and it doesn't show values at all. Since the values can be anything, we don't care what they are.

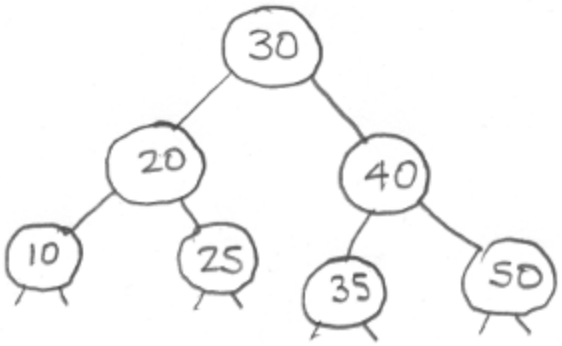

Fig. 1

A BST with given keys isn't necessarily unique—there can be many different BST's with the same keys. Also, some of those BST's are better than others, because they can be searched more efficiently. This will become important when we discuss AVL trees and OBST's later in the course.

Getting the value of a key. The procedure BST-GET is shown below, in Cormen's pseudocode. It takes a pointer *r* and a key *k* as its parameters. The parameter *r* points to the node at the root of the BST, or to NIL. BST-GET returns the value associated with the key *k* in the tree *r*. If *k* doesn't appear as a key in *r*, then BST-GET asserts an error.

```
01 BST-GET(r, k)
02   while r ≠ NIL
03     if k < r.key
04       r = r.left
05     else if k > r.key
06       r = r.right
07     else
08       return r.value
09   error "Can't find k!"
```

How does BST-GET work? Here's an example. Suppose that *r* points to the root node of the tree shown in Fig. 1, whose root has the key 30. Also suppose that *k*, the key whose value we want, is 25. We enter the loop at line 02. It runs because *r* is not NIL, the empty tree.

The first time through the loop, we see that *r*'s key is 30. At line 03, we see that 25 < 30. If 25 is anywhere in the tree, then it must be in *r*'s left subtree, so at line 04 we reset *r* to the left subtree, and go around the loop again.

The second time through the loop, we see that *r*'s key is now 20. At line 05, we see that 25 > 20. If 25 is anywhere in the tree, then it must be in *r*'s right subtree, so at line 06 we reset *r* to the right subtree, and go around the loop again.

The third time through the loop, we see that *r*'s key is 25. At line 03, we see that $25 \not< 25$, and at line 05, we see that $25 \not> 25$. Because keys are totally ordered, the only other possibility is that *r*'s key is equal to 25, so at line 08 we stop the loop and return *r*'s value.

Now suppose that we search for a key that's not in Fig. 1, like 37. I won't go through this example in as much detail, because it's boring. However, the first time through the loop, we visit the node containing 30. The second time, we visit the node containing 40. The third time, we visit the node containing 35. After that, *r* becomes NIL, the loop stops, and we assert an error.

How fast is BST-GET? Suppose that *r* has *n* nodes, and that we measure run time by counting key comparisons. Every time we go through the loop, we divide the size of the tree about in half, using operations that run in constant time, and we use only one of the halves. That means we can use the master theorem to compute BST-GET's run time, where $a = 1$, $b = 2$, and $f(n) = \Theta(1)$. The master theorem says BST-GET should run in $\Theta(\log_2 n)$ comparisons. We'll see later, however, that if the tree has a different shape, then BST-GET may run much slower than this.

Changing the value of a key. Given a set of keys and a set of values, we can build a BST using the procedure BST-PUT. We can also use BST-PUT to change the value of a key in an existing BST. It's shown below in Cormen's pseudocode.

BST-PUT takes three parameters. The parameter *r* points to the node at the root of a BST, or to NIL. It's prefixed by the symbol **var**, so it's *variable parameter*. That means if we change *r* inside the procedure, then the change is also visible to the procedure's caller. The parameter *k* is a key, and the parameter *v* is the key's corresponding value.

```
01 BST-PUT(var r, k, v)
02   if r == NIL
03     r = NEW-NODE(k, v)
04   else
05     p = r
06     while TRUE
07       if k < p.key
08         if p.left == NIL
09           p.left = NEW-NODE(k, v)
10           return
11         else
12           p = p.left
13       else if k > p.key
14         if p.right == NIL
15           p.right = NEW-NODE(k, v)
16           return
17         else
18           p = p.right
19       else
20         p.value = v
21         return
```

How does BST-PUT work? At line 02, we first test if *r* is NIL, the empty BST. If the test succeeds, then at line 03, we call NEW-NODE to make a new node containing the key *k* and the value *v*. The new node's left and right subtrees are empty. We set *r* to the new node and return. This effectively adds a new root node to an initially empty BST.

The rest of the algorithm assumes that *r* is not empty. At line 05, we set a local variable *p* to point to the same node as *r*. We then use *p* instead of *r*. If we used *r*, then changes to *r* would be visible outside BST-PUT. When I discussed this algorithm in the Early section, before Spring Break, I mistakenly didn't use *p*. *Students from the Early section: please correct your notes!*

In lines 06–21, we move *p* down the tree, searching for a node that contains the key *k*, using the same algorithm as BST-GET. If we find such a node, then we end up at line 19. At line 20, we change *p*'s value to *v*, the new value that corresponds to *k*, stop the loop, and return at line 21.

What if we don't find a node that contains *k*? When that happened in BST-GET, the BST we searched became empty, which stopped the **while** loop and asserted an error. However, if that happens here in BST-PUT, then it's not an error. Instead, it means we've found the place in the BST where a new node should go, so we make a new node and add it there. Lines 08–10 handle the case where we add the new node to a left subtree, and lines 14–16 handle the case where we add it to a right subtree. Either way, after we've added a new node, we stop the loop and return.

Here's an example. Suppose we want to add a new key *k* and value *v* to the BST *r* shown in the diagram above. Also suppose the key is 38; we don't care what its value is. We visit nodes with the keys 30, 40, and 35, in that order. After we visit the node with the key 35, we'll try to descend into that node's right subtree (line 13) because 38 > 35, but its right subtree is empty. That means we'll add a new node in the right subtree of the node that contains key 35 (line 15).

How fast is BST-PUT? Again, suppose that *r* has *n* nodes, and that we'll measure run time by counting key comparisons. By a argument similar to the one we used for BST-GET, we can show that BST-PUT's run time should be $\Theta(\log_2 n)$.

How BST's fail. Unfortunately, this argument isn't complete, either for BST-GET or BST-PUT. This is because there may be many BST's with the same keys, and some can be searched more efficiently than others. For example, Fig. 2 shows a BST with the same keys as in Fig. 1, but it's shaped very differently.

Fig. 2

Suppose we want to search Fig. 2 for the key 10 using BST-GET. Now we must visit nodes with keys 50, 40, 35, 30, 25, 20, before we finally get to the one we want, with key 10. This is no better than linear search, which needs $\Theta(n)$ comparisons to search n keys. We say that this tree is a *degenerate binary search tree,* not because it's morally objectionable, but rather because it has *degenerated* into a simpler data structure: a linear list of nodes.

A degenerate tree like this one results if we start with an empty BST (so r = NIL) and then call BST-PUT repeatedly with keys in strictly decreasing order: 50, 40, 35, 30, 25, 20, 10. (Again, we don't care what the values are.) We get a different tree, but still degenerate, if we call BST-PUT with keys in strictly increasing order: 10, 20, 25, 30, 35, 40, 50.

For this reason, we use a procedure like BST-PUT only when we know that it will be called with sequences of keys in *random order.* If the keys arrive at random, then we're about as likely to move left as we are to move right as we descend through the tree, trying to find where we should add new nodes. This usually gives us a BST with a small height (like Fig. 1) which can be searched in approximately $\Theta(\log_2 n)$ time for n keys. This happens often enough that usually a simple algorithm like BST-PUT is all we need.

Balanced vs. unbalanced BST's. Okay, so what if it doesn't happen? What if we must build a BST using a sequence of keys that's (approximately) in increasing or decreasing order? Then we must stop using the *unbalanced* BST's we've discussed before, and use *balanced* BST's instead.

A balanced BST is built using a procedure similar to BST-PUT, but that monitors the height of the tree it builds. If the height becomes too large, then the procedure reorganizes the tree so it still satisfies the BST property, but so its height is minimized. The balanced BST's we'll discuss are called *AVL trees,* named for the two Soviet Russian mathematicians who invented them in 1962, Georgy Adelson-Velskii and Evgenii Landis. (The authors of at least one web site wrongly claims these are three people, but they were confused by the hyphenated surname.) AVL trees will be discussed in a future lecture.

Optimal BST's. In the previous discussion, we assumed that keys will be given to BST-GET with equal probabilities. But what if some keys are more likely than others? This is probably true if the keys are English words, so BUT, IS, and THE will have higher probabilities than BAIZE, MARZIPAN, and SYGYZY.

If we know that some keys have higher probabilities than others, then we may be able to make a better BST by placing those keys nearer the root, so BST-GET can find them with fewer comparisons. The resulting BST might not necessarily have minimum height. For example, suppose $\Pr\{k = x\}$ is the probability that a key k has the value x. Also suppose that we know the following:

$$\Pr\{k = 50\} > \Pr\{k = 40\} > \Pr\{k = 35\} > \Pr\{k = 30\} > \Pr\{k = 25\} > \Pr\{k = 20\} > \Pr\{k = 10\}$$

Then the degenerate tree in Fig. 2 might be better than the balanced tree in Fig. 1! This is because keys with higher probabilities are nearer the root, even though the entire tree has greater height.

BST's constructed according to the probabilities of their keys are called *optimal binary search trees,* or OBST's for short. We'll discuss OBST's in a future lecture, after AVL trees. For impatient students who can't wait, OBST's are also discussed on pages 397–404 of Cormen.