PROFESSOR: If you guys want me to cover anything in particular, is there anything you didn't understand in lecture? In the last section, I covered the recursion trees because they will be on the Pset, and people said they were a bit unclear, so we can do that and cover less of the stuff that I have here. Or if there's anything else, you can tell me what you want. So there I cover recursion trees because someone said, hey, can you go over that again? Is there any pain points? No? OK.

So then I'm going to give you the same choice that I gave to people last time, and that is we can go over recursion trees again, but if I do that, then I won't have time to go over the code for deleting a node from a binary search tree. So we'll go through the theory and you guys will have to go through the code on your own. But instead, we'll go over recursion trees again and remember how you solve a recurrence using recursion trees. The alternative is we don't do that and we complete the deletions part.

AUDIENCE: I feel like covering deletions, since we didn't do that in lecture, that would probably be more helpful.

PROFESSOR: Let's take a vote. Who wants to do deletions in painstaking detail? So deletions and not recursion? Who wants to do recursion trees and not deletion?

AUDIENCE: It's about equal.

PROFESSOR: It's equal and nobody cares. I'm really sad.

AUDIENCE: Let's do both in half detail.

PROFESSOR: OK, sure. Who remembers merge sort? What does merge sort do really quick?

AUDIENCE: It takes some sort of unsorted array, splits it in half, and then continually splits it, and then once it finally gets to the point where you have arrays of two elements, then it sorts them, and then sorts those, and then sorts those. It's a fun thing. And then it merges [INAUDIBLE].

PROFESSOR: That's so much code. I don't like to write much code because for every line of code that you write, you might have a bug in it, so I like to write less code. So the way I do it is when I get to an array size of one element, I know it's already sorted.

So merge sort. You have an array, it's unsorted. Split it into two halves, call merge sort on each half, assume that magically, they're going to come back sorted, and then you merge the sorted halves. How much time does merging take? OK. So the recursion for the running time of merge sort?

AUDIENCE: Why does it take n time? Just too large?

AUDIENCE: Isn't it the finger thing where you take each element, and you're like, this one, is that greater or less than, then you put it in the array. So you get--

PROFESSOR: Please take my word for it that it's order n.

AUDIENCE: I'll explain it and then I'll be confused.

PROFESSOR: OK, so order n. What's the recursion? Don't give me the solution because then I can't do the trees anymore, so give me the recursion before it's solved. Give me the recurrence formula. So it starts with T of N, right?

AUDIENCE: It starts with N over 2 plus N, I think.

PROFESSOR: Perfect. So you take the array, you split it into two, you call merge sort on the two halves of the arrays. So you call merge sort twice. That's why you have a 2 here. The 2 matters. Without it, you get a different answer. And when you call it, the arrays that you give it are half the size, and then merge takes order and time. Splitting depends on what you're using to store your arrays. Can be constant time or it can be order N. So the time won't change because of split.

How do we solve this recurrence? The recursion tree method says that we're going to draw a call graph. So we start out with a call to merge sort with an array of size N. Then it's going to call merge sort again, but after the array is split. So it's going to call merge sort twice, size is N over 2. This guy gets an array of N over 2, calls merge sort. Two arrays, sizes N over 4, N over 4. This does the same.

So this goes on forever and ever and ever until at some point we reach our base case. So we're going to have a bunch of calls here where the array size is? What's our base case? 1. Excellent. So this is the call graph for merge sort, and let's put the base case here so we know what we're talking about. T of 1 is theta 1.

Now inside the nodes, we're going to put the cost for each call without counting the sub-call, so the children here. That's this guy here, except instead of order N, I will write CN. Remember how sometimes we use CN instead of the order of notation? The reason we do that is if I put in the asymptotic notation, then we're going to be tempted to sum them up. You're allowed to sum terms using asymptotic notation as long as there's a finite number of them, but here, it turns out there's an infinite number of them. Also, if you go this way, you can never go wrong. You always get the right answer, so that's why we switch from order N to CN.

In order to merge sort an array of size N, we're going to merge sort two arrays of size N over 2 and then spend CN time on doing the merge. What are the costs here? To sort an array of N over 2, what's the cost outside the cost to merge?

AUDIENCE: C of N over 2.

PROFESSOR: Perfect. C times N over 2. C times N over 2. How about here?

AUDIENCE: C times N over 4.

PROFESSOR: Perfect. CN over 4. My nodes are really ugly. I should have drawn them like this from the beginning. CN over 4. There you go. How about down here?

AUDIENCE: C of N over 2 to the i.

PROFESSOR: You're going on step ahead. We'll do that right next.

AUDIENCE: C of N over log N, right? Because they're log N levels, so--

PROFESSOR: Let's not worry about the number of levels. You're ruining my steps. I was going to get to that two steps after this.

AUDIENCE: Is it just C?

PROFESSOR: Yep. So array size is 1, right? So the cost is C. C, C, C, C. OK, you guys got it if you're thinking of levels already. The next thing I want to do is I want to figure out how many levels I have in this tree. Why do I care about that?

The answer for T of N is the sum of all these costs in here because the cost of merge sorting an array of size N is the merge sort plus the costs for sorting the two arrays. And the nodes here keep track of all the time spent in recursive sub-calls, so if we can add up everything up, we have the answer to T of N. It turns out the easiest way to do that is to sum up the cost at each level because the costs are this guy copied over here. For a level, they tend to be the same, so it's reasonably easy to add them up, except in order to be able to add those up, you have to know how many levels you have.

So how do I know how many levels I have? Someone already told me log N. How do I get to that log N? So when I get to the bottommost level, the number has to be 1, the number next to the node, because that's my base case. When I have a one element array, it's sorted, I'm done. I return.

So I can say that for each level, the number next to the node is something as a function of L. Here, I'm going to say that this is N over 1, which is N over 2 to the 0 power. And this is N over 2, so it's N over 2 to the first power. This is N over 2 to the second, and so on and so forth. It might not be obvious if you only have two levels.

I don't want to draw a lot on the board because I don't have a lot of space and I'd get my nodes all messed into each other. If it takes more than two levels to see the pattern, go for it. Expand for three levels, four levels, five levels, whatever it takes to get it right on a Pset or on a test. So you see the pattern, then you write the formula

for the node size at the level.

And assuming this pattern holds, we see that the size of a node at level l, the size is 2 N over 2 to the l minus 1. Fair enough? You can say N over 2 to the l, and forget that there's a minus 1, and then the asymptotics will save you, so it's no big deal, but this is the real number.

So that means that at the bottommost level, at level l, this size is going to be 1. N over 2 to the l minus 1 equals 1. So now this is an equation, so I can solve for l. I pull this on the right side, N equals 2 to the l minus 1, so l minus 1 equals-- anyone? The inverse of an exponential?

AUDIENCE: I wasn't paying attention. Sorry.

AUDIENCE: Log N.

PROFESSOR: The inverse of an exponential is a logarithm. Keep that in mind for solving 6.006 problems. l minus 1 is log N so l is log n plus 1, roughly log n. I could use log n plus 1 and go through the math. It's a bit more painful and, because we're using asymptotics, it doesn't really matter.

So now we know how many levels we have. Let's see what's the cost at the level. So all the calls at a certain level, what's the sum of the costs? For this level, what's the cost? CN. And That was the easy question. Just the root, right?

How about this level? Because I have two nodes, the cost in each node is CN over 2. How about this level? Four levels, each level CN over 4. How about the bottom level?

AUDIENCE: CN.

PROFESSOR: Why is it CN?

AUDIENCE: Because there are N arrays of size 1.

PROFESSOR: N arrays of size 1. Excellent. A cute argument I heard once is you start out with N,

you split it into N over 2 and N over 2. Then you split this guy in N over 4, N over 4, so this is like conservation of mass. If you start with N and here, you don't end up with N total, then you lost some element somewhere on the way.

So CN. CN, CN, CN, CN. I think I see a pattern. I think it's reasonable to say that for every level, it's CN. And if you write the proof, you can prove that by using math instead of waving hands. So CN times the number of levels, right? The answer for this guy is C of N is CN times l. What's l?

AUDIENCE: N log N.

PROFESSOR: Roughly. OK So order of N log N. C becomes order of, l is order of log N, N stays the same. Any questions? Are people getting it or did I confuse you even more?

AUDIENCE: We got it.

PROFESSOR: OK, sweet. Thank you for the encouragement. So this gets you through problem one of Pset 2. So in this case, the tree is nicely balanced. The cost at each level is the same. When [INAUDIBLE] talked about recursion trees in lectures, he showed two more trees, one where pretty much all the cost was up here-- the cost of the children was negligible-- and one tree where all the cost was concentrated here, so the cost of all the inner nodes was negligible and the leaves were doing all the real work. So don't be scared if your costs aren't the same. Just sum them up and you'll get to the right answer.

Now I'm going to talk about binary search trees, except I will make a five minute general talk about data structures before I do that. So we use the term "data structures." I think we covered it well, and I want to give you a couple of tips for dealing with them on Pset 1.

A data structure is a bunch of algorithms that help you store and then retrieve information. You have two types of algorithms. You have queries, and you have updates.

You start out with an empty data structure, like an empty binary search tree or an

empty list, and then you throw some data at it. That's when you update it. Then you ask it some questions, and that's when you query it. Then maybe you throw more data at it, so you do more updates, and you ask more questions, so you do more queries. What are the queries and the updates for the binary search trees that we talked about in lecture?

AUDIENCE: A query would be like, what's your right child, what's your left child?

PROFESSOR: So that's for a node.

AUDIENCE: What are you looking for?

PROFESSOR: I'm looking for something for the entire infrastructure. So for the entire tree, what's a question that you would ask the tree?

PROFESSOR: Max.

PROFESSOR: OK. Min.

AUDIENCE: Next larger.

PROFESSOR: Next larger. Are you looking at the nodes?

AUDIENCE: Is there an are you balanced question?

PROFESSOR: Well, I would say that the most popular operation in a binary search tree is Search, which looks for-- we call it Find in the code because most code implementations call it Find nowadays, but binary search tree. What are you going to do in it? You search for a value. That's why it has the Search in binary search.

So queries are operations where you ask questions to the data structure and it doesn't change. How about updates? What did we learn for updates?

AUDIENCE: Insert.

PROFESSOR: Excellent. So Insert was covered in lecture, and we're doing Delete today. So data structures have this property that's called the representation invariant, RI, or Rep

Invariant. Actually, before I get there, the rep invariant says that the data in the data structures is organized in this way, and as long as it's organized in this way, the data structure functions correctly. Can someone guess for a sorted array what's the representation invariant?

AUDIENCE: It can mean sorted.

PROFESSOR: Yeah. A sorted array should be sorted. Sounds like a very good rep invariant. So the elements should be stored an array. Every element should be smaller than any element after it. And as long as the rep invariant holds, so as long as elements are stored in the right way in the data structure, the queries will return the right results. If the rep invariant doesn't hold, then God knows what's going to happen. What can you do in a storage array as long as the rep invariant holds? Sorted array. What's the reason why I would have a sorted array? What can I do that's fast in a sorted array?

AUDIENCE: Min and Max.

PROFESSOR: I can do that very fast. That's good. What's the running time?

AUDIENCE: A constant.

PROFESSOR: Perfect. Min you look at the beginning, Max you look at the end. Yes?

AUDIENCE: Binary search.

PROFESSOR: Binary search. That's the other reason for that. So binary search runs in order log N time, doesn't have to look at most of the array, tells you whether an element is there are not. Now, what if the array is unsorted? Will the algorithm work? It might say something isn't there when it actually is there. You can do binary search on a non-sorted array. So if the rep invariant doesn't hold, your queries might give you a wrong answer. How about updates? How do you search something in a sorted list?

AUDIENCE: You find where it should go and you move everything.

PROFESSOR: Yep. So you have to move everything, make room for it, and put it there so that the

array is still sorted at the end. You can't just append things at the end, even though that would be faster and lazier and less code. When you do an update to a data structure, you have to make sure that the rep invariant still holds at the end. Sort of a correctness proof for an update algorithm says that if the rep invariant holds at the beginning, the rep invariant is guaranteed to hold at the end.

Why do we care about this rep invariant stuff? Suppose you have a problem, say on the next Pset, that asks you to find the place that's slow in your code and then speed it up. And suppose you recognize the data structure there, and you say that's inefficient, and you want to implement another data structure that would be more efficient.

You're going to implement it. You might have bugs in an update. How do you find the bugs? Queries give you the wrong answers. You might do queries a long time after you do updates, and you're not going to know which update failed.

The right way to do this is you implement the method called Check RI-- that's what I call it-- so check the representation invariant. And that method walks through the entire data structure and make sure that the rep invariant holds, and if it doesn't, it raises an exception because you know that whatever you try to do from there is not going to work, so there's no reason to keep going.

So at the end of every update, you add a call to this Check RI method until you're sure that your code is correct. And after you're done debugging your code, you remove this method and you submit the code. Why do I want to remove the method? It might be painfully slow and inefficient, much slower than the actual queries and updates.

For example, let's take a heap. Do people remember heaps from lecture? What's the query for a heap? Say you have a max heap. What's a query?

AUDIENCE: Where's the max?

PROFESSOR: OK, cool. So for a max heap, a query would be max. Running time?

AUDIENCE: Constant.

PROFESSOR: Perfect. Constant. What do you do? Look at the top?

AUDIENCE: Yeah, exactly.

PROFESSOR: OK. Sweet. So what are the two popular updates in a max heap?

AUDIENCE: There would be Insert as well.

PROFESSOR: OK. Insert. And did we teach you general delete? Usually Extract Max is simpler. That's all you need. What's the running time for Insert? Do people remember heaps?

AUDIENCE: I think it was per N, but I'm not completely sure.

PROFESSOR: Anyone else? It's not. Life would be bad if it would be N.

AUDIENCE: N squared?

PROFESSOR: No. It's better than N, so you guys are doing a binary search over the few running times that I gave you earlier.

AUDIENCE: [INAUDIBLE] add to the N, and then you compare your neighbor, and then you [INAUDIBLE].

AUDIENCE: If it's an array, there isn't--

PROFESSOR: So conceptually, a heap looks like this. And yeah, it becomes an array eventually, but let's look at it this way. It is a full binary tree. Binary tree means that each node has at most two children, and full means that every level except for the last level is completely populated. So every internal node has exactly two children, and in here, every node except for some nodes and then some nodes after it will not have. Everything to the left is fully populated, and then at some point, you stop having children.

It turns out that this is easy to store in an array, but I will not go over that. Instead, I

want to go over inserting. What's the rep invariant for a heap?

AUDIENCE: The max in the top, right? Well, for max heap, and then the two children are less than the next node.

PROFESSOR: All right. So the guy here has to be bigger than these guys, then the guy here has to be bigger than these guys, and so on and so forth. And if you use induction, you can prove that if this is bigger than this, it has to be bigger than these guys, and bigger than these guys, and bigger than everything, and it's a max. That's the reason why we have that rep invariant.

So the way we insert a node is we add it at the bottom, the only place where we could add it. And then if this guy is bigger than this guy, the rep invariant is violated, so we swap them in order to fix that. Now the guy is here. If this is bigger than this, we do another swap. If this is bigger than this, we do another swap. So you're going to go from the bottom of the heap potentially all the way up to the root. So the running time of insert is order of the height of the heap.

Now, the heap is a full binary tree. I said "full." I keep saying "full." The reason I care about full is that the full binary tree is guaranteed to have a height of log N. It's always log N, where N is the number of nodes. So inserting in a heap takes log N.

AUDIENCE: I have a question. Didn't they say that because it's in an array, then to find it-- oh no, I guess because you can still do the swaps.

PROFESSOR: You can still do the swaps when you have it serialized in an array. You know that given an item's index, the parent is that index divided by 2. So you add an element at the end of the array, and then you know what the parent is, and then you keep swapping and swapping and swapping towards the [INAUDIBLE].

AUDIENCE: You don't ever have to put it in and shift everything over. You're only swapping.

PROFESSOR: Yep. You only swap. That's important. Thanks for asking. That's important. So log N. Extract max, take my word for it, also log N. What's the running time for checking the invariant in a heap? So to make sure that this guy is a heap, if I had numbers

here, what would you have to do?

AUDIENCE: You'd have to look at every node.

PROFESSOR: Yep. So running time?

AUDIENCE: Theta of N.

PROFESSOR: Yep. So if I'm going to submit code for a heap where the operations are our order of log N, or order 1, but then each of these calls Check RI, that's going to be painfully slow because I'm making the updates be order N instead of log N. So you're putting Check RI calls in every update. You debug your code. When you make sure it's correct, you remove those, and then you submit the Pset. Make sense? Sweet. And we looked a little bit at heaps, which is good.

Binary search trees. So a binary tree is a tree where every node has at most two children. When we code this up, we represent a node as a Python object, and for a node, we keep track of the left child, of the right child, parent, and then this is a hollow tree. It's not very useful. This becomes useful when you start putting keys in the nodes so that you can find them and do other things with them. So each node has a key.

Let me draw a binary search tree. Can people see this? So this is a binary tree. Can someone say something a bit more specific about it?

AUDIENCE: It's unbalanced.

PROFESSOR: OK. It's imbalanced. So that means that finding things all the way at the bottom is going to be expensive. What else? So I said it's a binary tree. Give me something more specific.

So binary tree just means that every node has two children. There's a bit more structure in this guy. So if I look at the root, if I look at 23, all the nodes to the left are smaller. All the nodes to the right are bigger. Now, if I look at 8, all the nodes to the left are smaller, all the nodes to the right are greater.

This additional rep invariant defines a binary search tree. This is what we talked about in class. BST. Why would I want to have this rep invariant? It sounds like a pain to maintain nodes with all these ordering constraints. What's the advantage of doing that?

AUDIENCE: Search is fast.

PROFESSOR: Yep. Search is fast. How do I do search?

If you're looking for 42 or for 16, you'd be like, oh, it's less than 23. I'll get on this path.

PROFESSOR: So start at the root, compare my key to the root. If it's smaller, go left. If it's bigger, go right. Then keep doing that until I arrive somewhere or until I arrive at a dead end if I'm looking for 14. This is a lot like binary search. Binary search in an array, you look at the middle. If your key is smaller, go left. If your key is bigger, then go right.

Let's look at the code for a little bit. Look at the BST Node Class, and you'll see that it has the fields that we have up here. And look at the Find method, and this is pretty much the binary search code. Lines 8 and 9 have the return condition when you're happy and you found the key, and then line 10 compares the key that you're looking for with the key in the node that you're at, and then lines 11, 14, 16, and 19 are pretty much copy pasted, except one of them deals with the left case, the other one deals with the right case. What is the running time for Find?

AUDIENCE: Wouldn't it be log N, right?

PROFESSOR: I wish. If this is all you have to do to get log N, then I would have to write a lot less code. So not quite log N. We will have to go through next lecture to get to log N. Until then, what's the running time?

AUDIENCE: Order h.

PROFESSOR: Yep. So you told me at the beginning it's unbalanced.

AUDIENCE: Yeah.

PROFESSOR: So then it's not going to be fast. OK, so order h. The reason why we care about h, and the reason we don't say order N, is because next lecture after we learn how to balance a tree, there's some magic that you can do to these binary search trees to guarantee that the height is order of log N. And then we'll go through all the running times that we have and replace h with log N.

Now, it happens that in this case, if you would have told me order N, I couldn't argue with you because worst case, searches are order N. Can someone give me a binary search tree that exposes this degenerate case? Yes?

AUDIENCE: If it's completely unbalanced and every node is greater than the parent nodes.

PROFESSOR: So give me some inserts that create it.

AUDIENCE: Insert 5.

PROFESSOR: 5.

AUDIENCE: Insert 10.

PROFESSOR: 10.

AUDIENCE: Insert 15.

PROFESSOR: 15.

AUDIENCE: Insert 20.

PROFESSOR: Yep. And I could keep going. I could say, 1, 2, 3, 4, 5. I could say 5, 10, 15. As long as these keep growing, this is basically going to be a list, so searching is order N. This is a degenerate case.

Turns out it doesn't happen too often in practice. If you have random data, the height will be roughly log N. But in order to avoid those degenerate cases, we'll be doing balanced trees later on. So we covered Find. We know it's order h. How do

you insert, really quickly?

AUDIENCE: Do you mean in searching when it's balanced or unbalanced?

PROFESSOR: This guy. So the trees look exactly the same. If it's balanced, it's going to look more like that than like this. Actually, this is balanced. This is perfectly unbalanced. This is somewhere in the middle. If it's balanced, it's just going to look more like this, but it's still a binary search tree. How would you insert a node? Yes?

AUDIENCE: Can't you start at the root and find your way down, and then the first open child that you see that's applicable to your element, state it then?

PROFESSOR: Yep. So if I wanted to insert 14, which way do I go?

AUDIENCE: So you'd look at 23, and you'd say, it's less than 23, go left. You'd look at 8. You'd say, it's greater than 8. You'd go right. Look at 16. You'd say it's less, so you go left. 15, it's less. Then you have an open spot so you stick it there.

PROFESSOR: Excellent. Thank you. Yes?

AUDIENCE: I have a question. What if we want to insert 5? Then--

PROFESSOR: So if you want to insert who?

AUDIENCE: 5. Or actually no, we can't. I'm thinking, is there any case in which need to move a node?

PROFESSOR: How would you insert 5? Let's see. What would you do for 5?

AUDIENCE: For 5, then we'd insert it to the right of 4, right?

PROFESSOR: Smaller, smaller, greater, 5. Right?

AUDIENCE: So there would be no case in which we'd need to swap nodes or something?

PROFESSOR: No. You're thinking ahead. We'll talk about that a little later when we get to deleting. As long as you follow a path in the tree, the path that finding would get you to, as soon as you hit a dead end, that's where your node belongs. Because you know

next time you're going to search for it, the search is going to follow that path and find the node. Yes?

AUDIENCE: If you have values are the same, like two nodes at the same number, does it matter which side you put it on?

PROFESSOR: You don't.

AUDIENCE: Oh, I see. It's more like you would only have four 1's in the tree.

PROFESSOR: Yes. So if you're trying to store keys and values, then what you'd have to do if you want to allow multiple values for the same key is you have a linked list going off of this, which node becomes an array of values aside from the key. Smart question. Thank you. That trips you up every time you do actual code, so that's the right question to ask yourself when you're implementing this. Will I have duplicates? How do I handle them? We don't. We take the easy way out.

So if you look at Insert, on the next page, you will see that the code is pretty much the Find code copy pasted, except when Self Left is None or Self Right is None, instead of returning, it creates a new node. Does that make sense to people? All right.

So Delete is going to be the hardest operation for today. Before we do that, let's do a warm up operation. Let's say I want to implement Find Next Larger, also called Successor in some implementations. So I have a node. Say I have node 8, and I want to find the next key in the tree that's strictly larger than 8 but smaller than anything else. So if I would take these nodes and write them down in order, I want to find the element that would go right after it. How do I do that? Don't cheat. Don't look at the code, or make my life easier and do searches.

AUDIENCE: Go down one to the right, and you try to get down left as far as you can.

PROFESSOR: OK. Very good. So I have a node, and it has some subtree here, so I can go to the right here, I can go all the way left. We have an operation that does this, and it's called Min for a tree. In order to find the minimum in a binary search tree, you keep

going left. For example, in this case, you get 4, which is good.

So the way you would code this up is if you have Min, you go to the right if you can, and then you call Min on the subtree. And you can see that lines 3 and 4 do exactly that. Good guess. But you can line 1 says case one, so you have the right answer for one case.

Now we have to handle more difficult cases. What if instead, I go down a bunch of nodes, and I want to find the successor for this guy, for example, and there's nothing here. What do I do? So if I want to find the successor for 8, what do I do? Sorry. It has an answer. What if I want to find the successor for 4?

AUDIENCE: Go up one.

PROFESSOR: OK. Go up one. Why does that work?

AUDIENCE: You know it's going to be greater.

PROFESSOR: So I'm going up right. So I know that everything here is guaranteed to be smaller, everything here is guaranteed to be greater than this guy. This guy is up right, so this is guaranteed to be greater than this, and everything here is guaranteed to be greater than this, and so on and so forth for the entire tree. So if I go up right, I'm happy. I definitely found my answer. Now, what if I have something that looks like this, and I want to find the successor for this guy?

AUDIENCE: There is none.

PROFESSOR: In this case, there is none if there's nothing else here. What if I have this, but then I have this? So I came down this way.

AUDIENCE: Are you saying you're calling on that last node?

PROFESSOR: Yep.

AUDIENCE: Find the larger? I guess you'd just trace back up.

PROFESSOR: And where do I stop?

AUDIENCE: It affects the tree, so you go up one from there. You don't stop there.

PROFESSOR: Why can't I stop here?

AUDIENCE: Because you know that that-- not necessarily.

AUDIENCE: You know that everything in that long branch right there is less than that node [INAUDIBLE].

PROFESSOR: This is to the left of this guy, so this guy has to be greater than everything here, and then you can repeat the argument that we had before. So here, we could stop right away because we could branch left. In this case, you have to go up until you're able to go left and up. If you get to the root, then what happened? Then we're in this case, and you have no successor.

So take a look at the code. The next larger, lines 1 through 9. Case two, 6 through 8, does exactly that. If I can't go to my right and find the tree there, then I go up through my parent chain, and as long as I have to go up to the left, so as long as I'm the right child of my parent, I have to keep going. The moment I find the parent where I'm the left child, I stop. That's my successor.

What if I would have to find the predecessor instead? So the element that's smaller than me but bigger than everything else in the tree. What would I do?

AUDIENCE: It's just the opposite.

PROFESSOR: Just the opposite. So how do I do the opposite?

AUDIENCE: You can take the max of the left side tree, or traverse up, and if that's less than--

PROFESSOR: OK, so if I have a left subtree, fine. Call max on it and get the rightmost node there. If not, I go up, and when do I stop? When I go left or right?

AUDIENCE: You'd have to go right. Is that right?

PROFESSOR: Yep. So last time, in this case, when I was going up, if I was going left, I had to keep

going, and the moment I went right, I was happy and I stopped. What if I want to find the predecessor? It's the opposite, right? So I will go this way, and the moment I can go this way, I'm done. How do you do this in code? Slightly tricky. Just slightly, I promise.

AUDIENCE: [INAUDIBLE].

PROFESSOR: It's hard. What I would do is copy paste the code, replace "left" with "right" everywhere, and replace "min" with "max." You get it done. So we talked about how the tree is symmetric, right? So every time, instead of saying "left," you say "right," and instead of saying "min," you say "max." That's how you do this.

How do we do deletions? So suppose I'm in this tree and I want to delete 15. What do I do?

AUDIENCE: Kill it.

PROFESSOR: Kill it. Very good. What if I want to delete 16? What do I do?

AUDIENCE: You need to put 15 where 16 is.

PROFESSOR: OK. So I would put 15 here. So I had 16. Suppose I have a big tree here. Actually, let's go for an easier case. Let's say I have this tree here. So you're here, you have a big tree here, you don't have anything here, and you want to delete this guy.

AUDIENCE: You know that everything less than the top node is going to be less than it, so you can just move that up.

PROFESSOR: Everything less than this guy is also going to be less than this guy. So you're saying move the whole tree up.

AUDIENCE: Yep.

PROFESSOR: So the way we do that is we'd take this node's left link and make it point here, and take this guy's parent link and make it point here, and this guy sort of goes away. So we have two cases for deleting. We have if you're a leaf, we'll take you out. Sorry. I

got confused. If you have one child and that child is in the same direction as your parent, then you can do this. What if you have one child, but it's a zigzag like this? What do you do?

AUDIENCE: It's still greater than, so you do the same thing.

PROFESSOR: Exactly. Same thing. Just change this guy, change this guy, and I'm happy. So it doesn't matter if you have a zigzag or a straight line. It might help you think about it to convince yourself that the code is correct, but in the end, you do the same thing. Now, what if I want to delete node 8? So what if I have a nasty case where I want to delete this guy and it has children both on the left and on the right?

AUDIENCE: You have to take 8, compare it to its parent and compare it to its right child, and see which one is greater in order to figure out which node gets replaced in its spot.

PROFESSOR: OK. So there is replacing that's going to happen. The answer is really tricky. I always forget this when coding. Try to understand it, and if it doesn't work, refer to the textbook. When you forget it, because you will, refer to the textbook or to the internet.

So what you do is I can't just magically replace this node with one of the subtrees, but we talked right before this about Next Greater, so finding a node's successor. If this node has both a left subtree and a right subtree, then I know that if I call Find Successor on it, I'm going to go somewhere inside here, and I'm going to find a node somewhere in here all the way to the left that is this guy's successor.

So what I'm going to do is I'm going to delete this node instead, and then I'm going to take its key and put it up here. So if I want to delete 8, what I do is I find its successor, then I delete it, then I take the 15 that was here-- you can see it, right? It's still there. Put it here.

So the reason this works is that everything here is greater than this guy. Everything here is smaller than this guy. This is the next node that's greater than this guy, but everything else is bigger than it, right, because we wanted it to be a successor. So if I take this value and I put it up here, everything in here is still going to be greater

than it. This is a successor of this guy, so everything here is still going to be smaller than the successor.

Great. In order to do a delete, I find the successor, and then I call Delete on it. How do I know that this will end? How do I know that I'm not going to go into a loop that runs forever?

AUDIENCE: Because it's not--

AUDIENCE: It's acyclic, right?

PROFESSOR: OK. First answer, good. Eventually, worst case, I'm going to get to the maximum, and then not going on have to delete the successor anymore.

Now, another thing to note here is that if this guy is the successor of this guy, it can't have anything on the left, because if it would, then whatever is down here has to be bigger than this, and whatever's to the left of this node has to be smaller than this. But we said that this is the successor of this, so there's nothing here.

So this will be one of the easy cases that we talked about. The successor either has no kids, or it has only one child, only one subtree. So then I can delete it using one of the easy cases. So in fact, worst case that happens in a delete is my node has two subtrees. Then I find the successor that's only going to have one subtree, I change my links there, and I'm done. What is the running time for Delete?

AUDIENCE: Is it order h, because you should do it all the way down to the bottom of the tree, right?

PROFESSOR: You have the right answer. Let's see why it's order h. It has to be order h, right? Otherwise, the tree would be too slow. If it's order N, then it's bad. So why would Delete be order h? This was a heap, right, so I can't use this. I'm going to write "delete" here again.

So the first thing you do is you have to search for the key, right? That's order h. Now, if it's a happy case, if it's case one or two, you change some links and you're

done. What's the time for that?

AUDIENCE: Constant.

PROFESSOR: Constant. So happy case, order h for sure. Now sad case. If you have two children, what do you have to do after you realize that you have two subtrees?

AUDIENCE: Find the successor.

PROFESSOR: OK. What's the running time for finding a successor?

AUDIENCE: Order h.

PROFESSOR: Order h. Once I find the successor, what do I do? Call Delete on that, and what happens? It's a happy case or a sad case?

AUDIENCE: It's a happy case.

PROFESSOR: Happy case, a few links get swapped, constant time. So worst case, order h plus order h. Order h. So insertions are order h, deletions are order h.

AUDIENCE: The first one. Because the second one is from finding the successor. What is the first one for?

PROFESSOR: Finding the node for a key in the tree. So if I say Delete 8, then you have to find 8. If I give you the node, then you don't have that. Good question. It's a good question. Thank you. So that's insertion. That's deletion.

Let's look at the code for Delete. Looks kind of long. So lines through 21, happy case or sad case? Try to do it by looking at the "if" instead of looking at the comments. So lines through 21 for Delete.

AUDIENCE: On this tree? Which tree, because there are two deletes?

PROFESSOR: Oh really? Sorry. Why do we have two deletes?

AUDIENCE: There's BST Delete and then there's BST Node Delete.

PROFESSOR: So BST Delete. Finds the node, and then calls Delete on the node. And then if the node is a tree's root, then it updates the tree's root. So let's look at the nodes delete. Oh, I see. I think I was looking at the wrong one. Thank you. My Delete was much longer than yours. So lines 3 through 12, happy case or sad case? Look at the "if" on line 3 and tell me, what case is it going for?

AUDIENCE: [INAUDIBLE].

PROFESSOR: If it doesn't have a left child or it doesn't have a right child, is that the happy case or the sad case?

AUDIENCE: Happy.

PROFESSOR: Happy case. So lines 4 through 12 handle the happy case. Lines 14 through 16 handle the sad case. Do lines 14 through 16 make sense? Find the successor, then swap the keys, then delete that successor.

Now, lines 4 through 11 are pretty much what we talked about here, except I can't draw arrows on the board and instead I have to change left and right links. Line 4 has to see if we're a left child or a right child, and then lines 5 through 7 and 9 through 11 are pretty much copy paste, swap left with right. And they changed the links like we changed them here. Do we have any questions on Deletes?

AUDIENCE: So if the successor had a right child, then all you do, you just do the workaround thing where you just--

PROFESSOR: Yep. So the case that it doesn't have two children. As long as it doesn't have both children, you're in the happy case and you can do some link swapping. Are you guys burned out already? Fair enough.

I left a part out. What I left out is how to augment a binary tree. So binary trees by default can answer the question, what's the minimum node in a tree in order h. You go all the way to the left, you find the minimum. That's the minimum.

It turns out that if you make a node a little bit fatter, so if instead of storing, say, 23 in this node, I store 23, and I store the fact that the minimum in my left subtree is 4,

then it turns out that I can answer the question in constant time, what's the minimum? Oh gee, if you store the minimum here, of course you can retrieve it in constant time, right?

The hard part is, how do you handle insertions and updates in the same time? So the idea is that if I have a node and I have a function here, say the minimum of everything, if I have two children, here they're 15 and 42, and say the minimum in this tree is 4 and the minimum in this tree is. So if I already computed the function for these guys, how do I compute the function for this?

AUDIENCE: [INAUDIBLE] and compare it?

PROFESSOR: Yep. Take the minimum of these two guys, right? There are some special cases if you don't have a child. If you don't have a left child, then you're the minimum. But you write down those special cases, and you can compute this in how much time?

AUDIENCE: Order h, right?

PROFESSOR: What if I already computed the answer for the children? How much time does it take to compute the answer for a single node?

AUDIENCE: Constant.

PROFESSOR: Constant. OK.

AUDIENCE: For a tree, though.

PROFESSOR: For a tree, it's order h. Yeah. You're getting ahead. You're rushing me. You're not letting me finish.

AUDIENCE: Are you saying that we store the minimum value?

PROFESSOR: So for every--

AUDIENCE: Each node has a field that says what the minimum value is in that tree.

PROFESSOR: Yep, exactly. So for each node, what's the minimum in the subtree. So if I add a

node here, suppose I add three and I had my minimums, what changed? This subtree changed, this subtree changed, this subtree changed, and then this subtree changed.

So I have to update the minimums here, here, here, here. Nothing else changed. Outside the path where I did the Insert, nothing changed, so I don't have to update anything. So what I do is after the Insert, I go back up and I re-compute the values. So here, I'll have 3. I go back up 3, 3, 3.

AUDIENCE: You could when you're passing down, though. When you're going down that column, you can just compare it on the way down. You don't have to go back up, right?

PROFESSOR: Yep. So the advantage of doing it the way I'm saying it is that you can have other functions instead of minimum. As long as you can compute the function inside the parent in constant time using the function from the children, it makes sense to compute the function on the children first. There's an obvious function that I can't tell you because that's on the Pset, but when you see the next Pset, you'll see what I mean.

So if you have a function where you know the result for the children and you can compute the result for the parent in constant time, then after you do the Insert, you go up on the path and you re-compute the function. When you delete, what do you do?

AUDIENCE: Same thing.

PROFESSOR: Same thing. If this goes away, then this subtree changed, and then if there would be something else here, then this subtree changed, but nothing else changed. So whenever you do an Insert or a Delete, all you have to do is go back up the path to the parent and re-compute the function that you're trying to compute. And that's tree augmentation. Does this make sense somewhat?

That's it. So what you'll find in lecture notes is a harder way of doing it that works for minimum, but what I told you works for everything. So don't tell people I told you

how to do this for everything. Sure nobody's going to know.