PROFESSOR: Today's lecture is about a brand new data structure that you've probably seen before, and we've mentioned earlier in double 06, called a binary search tree. We've talked about binary search. It's a fundamental divide and conquer paradigm. There's a data structure associated with it, called the BST, a binary search tree.

And what I want to do is motivate this data structure using a problem. It's a bit of a toy problem, but certainly a problem that you could imagine exists in all sorts of scheduling problems. It's a part of a runway reservation system that you can imagine building.

And what I'll do is define this problem and talk about how we could possibly solve it with the data structures you've already seen-- so lists and arrays, heaps as well as, which we saw last time-- and hopefully motivate you into the reason behind the existence of binary search trees, because they are kind of the perfect data structure for this particular problem. So let's dive into what the runway reservation system looks like. And it's your basic scheduling problem.

We'll assume an airport with a single runway. Now Logan has six runways. But the moment there's any sort of weather you're down to one. And of course, there's lots of airports with a single runway.

And we can imagine that this runway is pretty busy. There's obviously safety issues associated with landing planes, and planes taking off. And so there are constraints associated with the system, that have to be obeyed. And you have to build these constraints in-- and the checks for these constraints-- into your data structure. That's sort of the summary of the context.

So reservations for future landings is really what this system is built for. There's a notion of time. We'll assume that time is continuous. So it could be represented by a real variable, or a real quantity.

And what we'd like to do is reserve requests for landings. And these are going to specify landing time. Each of them is going to specify a landing time. We call it t.

And in particular, we're going to add t to the set R of landing times if no other landings are scheduled within k minutes. And k is a parameter that could vary. I mean, it could be statically set to 3 minutes, or maybe 4. You can imagine it varying it dynamically depending on weather conditions, things like that. For the most of the examples we'll talk about today, we'll assume k is 3 minutes, or something like that.

So this is about adding to the data structure. And so an insert operation, if you will, that has a constraint associated with it that you need to check. And so you wouldn't insert if the constraint was violated. You would if the constraint was satisfied.

And time, as I said, is something that is part of the system. It needs to be modeled. You have the current notion of time. And every time you have a plane that's already landed, which means that you can essentially take this particular landing time away from the set R. So this removal, or delete-- we remove from set R, which is the set of landing times after the plane lands.

So every once in awhile, as time increments, you're going to be checking the data structure. And you can do this, maybe, every minute, every 30 seconds. That isn't really important. But you have to be able to remove from this data structure.

So fairly straightforward data structure. It's a set, R. We don't quite know how to implement it yet. But we'd like to do all of these operations in order log n time, where n is the size of the set. All right?

So any questions about that? Any questions about the definition of the problem before we move on? Are we good on? OK.

So let's look at a real straightforward example, and put this up here so you get a better sense of this. Let's say that, right now, we are at time 37. And the set R has

41.2, 49, and 53 in it. And that's time.

Now you may get a request for landing time 53. And-- I'm sorry. I want to call this 56.3-- 41.2, 49, and 56.3. You may get a request for landing time 53. And right now the time is 37. It's in the future, and you say OK because you've done the check. And let's assume that k equals 3. And 53 is four ahead of 49, and 3.3 before 56.3, so you're OK.

44 is not allowed. It's too close to 41.2. And 20, just for completeness, is not allowed because it's passed. Can't schedule in the past. I mean, it could be the next day. But then you wouldn't call it 20.

Let's assume that time is a monotonically increasing function. You have a 64-bit number. It can go to the end of the world, or 2012, or wherever you want. So you can keep the number a bit smaller, and do a little constant factor optimization, I guess.

So that's sort of the set up. And hopefully you get a sense of what the requirements. And you guys know about a bunch of data structures already. And what I want to do is list each one of them, and essentially shoot them down with respect to not being able to make this efficiency requirement. And I'd like you guys to help me shoot them down.

So let's talk about an easy one first. Let's say you have an unsorted list or an array corresponding to R. That's all you have. What's wrong with this data structure from an efficiency standpoint? Yeah.

AUDIENCE: Pretty much everything you want to do to it is linear.

PROFESSOR: That's exactly right. Pretty much everything you want to do to it is linear. And so you want to check the k minute check. You can certainly insert into it, and just add to it. So that part is not linear, that's constant time.

But certainly, anything where you want to go check against other elements of the array, it's unsorted. You have no idea of where to find these elements. You have to scan through the entire array to check to see whether there's a landing time that's within k of the current time t that you're asking for. And that's going to take order n time.

So you can insert in order 1 without a check. But sadly, the check takes order n time. All right?

Let's do something that is a little more plausible. Let's talk about a sorted array. So this is a little more subtle question. Let's talk about a sorted array. What happens with a sorted array? Someone? What can you do with a sorted array? Yeah.

AUDIENCE: Do a binary search to find the [INAUDIBLE].

PROFESSOR: Binary search would find a bad insert. OK, good. So that's good. So if you have a sorted array, and just for argument's sake, it looks like 4, 20, 32, 37, 45. And it's increasing order. And if you get a particular time t, you can use binary search.

And let's say, in particular, the time is, for example, 34. Then what you do is you go to the midpoint of the array, and maybe you just look at that. And you say oh, 34 is greater than 32. So I'm going to go check and figure out if I need to move to the left or the right. And since it's greater I'm going to move to the right.

And within logarithmic time, you'll find what we call the insertion point of the sorted array, where this 34 is supposed to sit. And you don't necessarily get to insert there. You need to look, once you've found the insertion point, to your left and to your right. And do the k minute check.

So finish up the answer to the question, tell me how long it's going to take me to find the insertion point, how long it's going to take me to do the check, and how long it's going to take me to actually do the insertion.

AUDIENCE: Log n in the search--

PROFESSOR: Log n for the search, to find the point.

AUDIENCE: Constant for the comparison?

PROFESSOR: Constant to the comparison. And then the last step?

AUDIENCE: Do the research [INAUDIBLE].

PROFESSOR: Sorry, little louder. Sorry.

AUDIENCE: The insertion is constant.

PROFESSOR: Insertion is constant? Is that right? Do you people agree with him, that insertion is constant?

AUDIENCE: You've got a maximum size up there, right? There must be a maximum. [INAUDIBLE].

PROFESSOR: No, the indices-- so right now the array has indices i. And if you start with 1, it's 1, 2, 3, 4, 5, et cetera. So what do you mean by insertion? Someone explain to me what-- yeah, go ahead.

AUDIENCE: When you put something in you have to shift every element over.

PROFESSOR: That's exactly right. That's exactly right. Ok, good, that's great. I guess I should give you half a cushion. But I'll do the full one, right? And you get one, too.

So the point here is this is pretty close. It's almost what we want. It's almost what we want.

There's a little bit of a glitch here. We know about binary search. The binary search is going to allow us, if there's n elements here, to find the place-- it's going to be able to find-- and I'm going to precise here-- the smallest i such that R of i is greater than or equal to t in order log n time. It's going to be able to do that.

You're going to be able to compare R of i and R of i minus 1-- so the left and the right-- against t in order 1 time. But sadly, the actual insertion is going to require shifting. And that could take order n time, because it's an array.

So that's the problem. Now you could imagine that you had a sorted list. And you could say, hey if I have a sorted list, then the list looks like this, and it's got a bunch of pointers in it. And if I've found the insertion point, then-- the list is nice, because you can insert something by moving pointers in constant time once you've found the insertion point. But what's the problem with the list? Yeah.

AUDIENCE: You can't do binary search [INAUDIBLE].

PROFESSOR: Well you can't do binary search on a list. There's no notion of going to the n by 2 index and doing random access on a conventional list, right?

So the list does one thing right, but doesn't do the other thing right. The array does a couple things right, but doesn't do the shifting right. And so you see why we've constructed this toy problem. It's to motivate the binary search tree data structure, obviously. But you're close, but not quite there.

What about heaps? We talked about heaps last time. What's the basic problem with the heap for this problem? The heaps are data arrays, but you can visualize them as trees. And obviously if we're talking about min heaps and max heaps.

So in particular, what goes wrong with a min heap or a max heap for this problem? What takes a long time? Yeah.

AUDIENCE: You have to scan every element, which [INAUDIBLE].

PROFESSOR: That's right. I mean, sadly, you know when we talk about min heaps or max heaps, they actually have a fairly weak invariant. It turns out that-- I'm previewing a bit here-- binary search trees are obviously similar to heaps in the sense that you visualize an array as a tree, in the case of a heap. And binary search trees are trees.

But the invariant in a min heap or a max heap, is this kind of a week invariant. It essentially says, look at the min element. And the min element has to be the root, so you can do that one operation pretty quickly. But if you want to look for a k minute check, you want to see if there's an element in the heap that is less than or equal to k, or greater than or equal to k from t, this is going to take order n time.

OK? Good.

And finally, we haven't talked about dictionaries, but we will next week. Eric will talk about hash tables and dictionaries. And they have the same problem. So it's not like dictionaries are going to solve the problem, for those of you who know about hash tables and dictionaries. But you'll hear about them in some detail. They're very good at other things.

So I don't want to say much more about that, because you're not supposed to know about dictionaries. Or at least we don't want to assume you do, though we have talked about them and alluded to dictionaries earlier.

And so that's a story here. Yeah, back there, question.

AUDIENCE: Yeah, can you explain why it's [INAUDIBLE] time?

PROFESSOR: So what is a heap, right? A heap essentially-- a min heap, for example, or we talked about max heaps last time, has the property that you have an element k, and you're going to look at, let's say it's 21. Let's do min heaps, so this has to be less than what's here, 23, and what there, maybe it's 30, and so on and so forth. And you have a recursive definition.

And when you insert into a min heap, typically what happens is suppose you wanted to insert, for argument's sake, I want to insert 25. I want to insert 25 into this. The insertion algorithm for a min heap typically adds to the end of the min heap. So what you do is you would add 25 to this. And let's say that you had something out here.

So you'd add to it. And you'd start flipping things. And you could work with just this part of the array to insert 25 in here.

And you'd be able to satisfy the invariant of the min heap. And you'd get a legitimate min heap. But you'd never check the left part of it, which is 23.

So it's quite possible-- and this is a good example-- that your basic insertion algorithm, which is essentially a version of max heap of i, or min heap of i, would simply insert at the end, and keep flipping until you get the min heap property, would be unable to check for the k minute check during the insertion. But what you'd have to do is to go look elsewhere. That min heap of i we'd never look at-- or the insert algorithm we'd never look at-- and that would require order n time. All right?

AUDIENCE: Thank you.

PROFESSOR: So that's the story for the min heap. Thanks for the question. And it's similar for dictionaries, as I said. And so we're stuck. We have no data structure yet that can do all of the things that I put up on the board to the left, in order log n time.

And as you can see, the sorted array got pretty close. And so if you could just solve this problem, if you could do fast insertion-- and by fast I mean order log n time-- into a sorted array, we'd be in business. So that's what we'd like to do with binary search trees.

Binary search trees are, as you can imagine, enable binary search. But the sorted arrays don't allow fast insertion, but BSTs do. So let me introduce BSTs.

As with any data structure, there's a nice invariant associated with BSTs. The invariant is stronger than the heap invariant. And actually, that makes them a different data structure, not necessarily a better data structure. And I'll say why, but different. For this problem they're better.

So one example of a binary search tree looks like this. And as a binary tree you have a node, and we call it x. Each of the nodes has a key of x. So 30 is the key for this node, 17 for that one, et cetera.

Unlike in a heap, your data structure is a little more complicated. The heap is simply an array, and you happen to visualize it as a tree. The binary search tree is actually a tree that has pointers, unlike a heap. So it's a more complicated data structure. You need a few more bytes for every node of the binary search tree, as opposed to the heap, which is simply an array element.

And the pointers are parent of x. I haven't bothered showing the arrows here, because you could be going upwards or backwards. And you could imagine that you actually have a parent pointer that goes up this way, and you have a child pointer that goes this way. So there's really, potentially, three pointers for each node, the parent, the left child, and the right child.

So pretty straightforward. That's the data structure in terms of what it needs to have so you can operate on it. And there's an invariant for a BST. What makes a BST is that you have an ordering of the key values that satisfy the invariant that for all nodes x if y is in the left subtree of x, we have-- if it's in the left subtree then key of y is less than or equal to key of x. And if y is in the right subtree we have key of y is greater than or equal to key of x.

So if we're talking about trees here, subtrees here, everything underneath-- and that's the stronger part of the invariant in the BST, versus in the heap we were just talking about the children.

And so you look at this BST, it is a BST because if I look to the right, from the root I only see values that are greater than 30. And if I look to the left, in the entire subtree, all the way down I only see values that are less than 30. And that has to be true for any intermediate node in the tree.

And the only other nontrivial node here is 17. And you see that 14 is less than 17, and 20 is greater than 17. OK?

So that's the BST. That's the data structure. This is the invariant.

So let's look at why BSTs are a possibility for solving our runway reservation problem. And what I'll do is I'll do the insert. So let's start with the nil set of elements, or null set of elements, R. And let's start inserting.

So I insert 49. And all I do is make a node that has a key value of 49. This one is easy.

Next insert, 79. And what happens here is I have to look at 49, and I compare 79 to 49. And because 79 is greater than 49 I go to the right and I attach 79 to the right child of 49.

Then I want to insert 46. And when I want to insert 46 I look at this, I compare 49 and 46. 46 is less, so I go to the left side and I put 46 in there.

Next, let's say I want to insert 41. So far I haven't really talked about the k minute checks. And you could imagine that they're being done. I'll show you exactly, or talk about exactly how they're done in a second. It's not that hard. But let me go ahead and do one more.

For 41, 41 is less than 49, so I go left. 41 is less than 46, so I go left and attach it to the left child. All right? So that's what I have right now.

Now let's talk about the k minute check. It's good to talk about the K minute check when there's actually a violation. And let's assume the k equals 3 here. And so, same thing here.

You're essentially doing binary search here. And you're doing the checks as you're doing the binary search. So what you're going to be doing is you're going to check that-- you're going to compare 42 with 49, with the k minute check. And you realize they're 7 apart. So that's OK.

And 42 is less than 49, so you go left. And then you compare 42 with 46. And again, it's less than 46, but it's k away, more than 3 away from 46. So that's cool. And you go left.

And then you get to 41. And you compare 42 with 41. In this case is greater. But it's not k more than it.

And so that means that if you didn't have the check, you would be putting 42 in here. But because you have the check, you fail. And you say, look, I mean this violates the safety property, violates the check I need to do. And therefore I'm not going to insert-- I'm not going to reserve a request for you. All right?

So what's happened here is it's basically a sorted array, except that you added a bunch of pointers associated with the tree. And so it's somewhere between a sorted list and a sorted array. And it does exactly the right thing with respect to being able to insert.

Once you've found the place to insert, it's merely attaching this particular new node with it's appropriate key to the pointer. All right?

So what's happened here is that if h is the height of the tree then insertion with or without the check is done in order h time. And that's what BSTs are good for. People buy that? Any questions about how they k minute check proceeded? Yeah, question.

AUDIENCE: So, what's it called? The what check?

PROFESSOR: The k minute check. Sorry, the k was 3 minutes k. I had this thing over here, add t to the set R if no other landings are scheduled within k minutes. So k was just a number. I want it to be a parameter because it doesn't matter what k is. As long as you know what it is when you do the binary search, you can add that in to an argument to your insert, and do the check.

AUDIENCE: OK.

PROFESSOR: So in this case, I set k to be 3 out here. And I was doing a check to see that the invariant, any elements in the BST already, on any nodes that had keys that were within 3 minutes-- because I fixed k to be 3-- to the actual time that I was trying to insert. All right?

AUDIENCE: So there's no way [INAUDIBLE].

PROFESSOR: I'm sorry, there's no way?

AUDIENCE: There's no way you could insert the 42 into the tree then?

PROFESSOR: Well, if the basic insertion method into a binary search tree doesn't have any constraints. But you can certainly augment the insertion method without changing the efficiency of the insertion method.

So let's say that all you wanted to do was insert into a binary search tree, and it had nothing to do with the runway reservation. Then you would just insert the way I described to you. The beauty of the binary search tree is that while you're finding the place to insert, you can do these checks-- the k minute checks. Yeah, question back there.

AUDIENCE: What about 45?

PROFESSOR: What about 45? So this is after-- we haven't inserted 42 because it violated the check. So when you do 45, then what happens is you see that 45 is less than 49 and you pass, because you're more than 3 minutes away. We'll stick with that example.

And then you get here and then you see that 45 is less than 46, and you'd fail right here. You would fail right here if you were doing the check, because 45 is not 3 away from 46. All right? So that's the story.

And so if you have h being the height of the tree, as you can see you're just following a path. And depending on what the height is you're going to do that many operations, times some constant factor. And so you can say that this is order h time. All right?

Any other questions? Yeah, question back there.

AUDIENCE: In a normal array [INAUDIBLE].

PROFESSOR: Well, it's up to you. In a conventional binary search tree, or the vanilla binary search tree, typically what you're doing is you're doing either find or insert. And so what that means is that you would just return the pointer associated with that element.

So if you're looking for find 46, for example, on the tree that I have out there, typically 46 is just the key value. And there may be a record associated with it. And you would get a pointer to that record because it's already in there.

At that point you can say I want to override. Or if you want, you could have duplicate values. You could have this, what's called a multiset. A multiset is a set that has duplicate elements. In that case, you would need a little more sophistication to differentiate between two elements that have the same key values.

So you'd have to call it 46a and 46b. And you'd have to have some way of differentiating. Any other questions? Yeah.

AUDIENCE: Wouldn't it be a problem if the tree's not balanced?

PROFESSOR: Ah, great question. Yes, stay tuned.

So I was careful, right? I guess I kind of alluded to the fact that we'd solved the runway reservation system. Did I actually say that we'd solved the problem? Did I say we had solved the problem? OK, so I did not lie. I did not lie.

I said that the height of the tree was h. And I said that this was accomplished in order h time, right? Which is not quite what I want, which is really your question. So we'll get to that. So we're not quite done yet.

But before we do that, it turns out that today's lecture is really part one of two. You'll get a really good sense of BST operations in today's lecture. But there's going to be a few things that-- we can't cover all of double 6 in the lecture, right? We'd like to, and let you off for the entire fall, but that's not the way it works, all right?

So it's a great question. I'll answer it towards the end.

I just wanted you to say a little bit about other operations. There's many operations that you can do on a binary search tree, that can be done in order h time, and some even in constant time. And I'll put these in the notes. Some of these are fairly straightforward.

Find min can be done in heap, in a min heap. If you want to find the minimum value, it's constant time. You just return the root.

In the case of a binary search tree, how do you find the min? Someone? Worth a cushion. Yep.

AUDIENCE: Keep going to the left?

PROFESSOR: Keep going to the left. And how do you find the max?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Keep going to the right. All right great, thank you. And finally, what complexity is that? I sort gave it away, but I want to hear it from you.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Hm?

AUDIENCE: It's the height

PROFESSOR: It's the height, order h. All right, it's order h complexity. Go to the left until you hit a leaf, and until leaf order h complexity. Same thing for max.

And then you can do a bunch of things. I'll put these in the notes.

You can find things like next larger x, which is the next largest value beyond x. And you look at the key for x and you say, for example, if you put 46 in there, what's the next thing that's larger and that? In this tree here, it's 49. But that's something which was trivially done in this example.

But in general you can do this in order h time as well. And you can see the pseudocode. And we'll probably cover that in section tomorrow.

What I want to do today, for the rest of the time I have left, is actually talk about augmented binary search trees, which are things that can do more and have more data in them than just these pointers. And that's actually something which should give you a sense of the richness of the binary search tree structure, this notion of augmentation.

And those of you, again, who have taken double 05, you know about design amendments. And so specifications never stay the same. I mean, you're working for someone, and they never really tell you what they want. They might, but they change their mind.

So in this case, we're going to change our mind. And so we've done this to the extent that we can cover all of these in order h time. And let's say that now the problem specification changed on us. There's an additional requirement that we're asked to solve.

And so you sort of committed to BST structures. But now we have an additional requirement. And the new requirement is that we be able to compute rank t. And rank t is how many planes are scheduled to land at times less than or equal to t.

So perfectly reasonable question. It wasn't part of the original spec. You now have built your BST data structure, you thought you were done. Sorry, you aren't. You've got to do this extra stuff.

So that's the notion of augmentation, which we're going to use this is an example of how we're going to augment the BST structure. And oh, by the way, I don't want you to change the complexity from order h. And we eventually will get to order log n, but don't go change something that was logarithmic to linear. That would be bad.

So let's talk about how you do this. And I don't think we need this anymore. So the first thing we need to do is add a little bit more information to the node structure. And what we're going to do is augment the BST structure. And we're going to add one little number associated with each node, that looks at the number of nodes below it.

So in particular, let's say that I have 49, 46, let's just say 49, 46 for now. And over here I have 79, 64, and 83.

I'm going to modify-- I'm going to have an extra number associated with each of these nodes. And I'm just going to write that number on the outside of the node. And you can just imagine that now the key value has two numbers associated with it-- the thing that I write inside the node, and what I write outside of it.

So in particular, when I do insert or delete I'm going to be modifying these numbers. And these are size numbers. And what do I mean by that? Well these numbers correspond to subtree sizes. So the subtree size here is 1, 1, 1.

So as I'm building this tree up I'm going to create an augmented BST structure, and I've modified insert and delete so they do some extra work. So let's say, for argument's sake, that I've added this in sort of a bottom up fashion. And what I have are these particular subtree sizes.

All of these should make sense. This has just a single node, same thing here. So this subtree sizes associated with these nodes are all 1.

The subtree size associated with 79 is 3, because you're counting 79 and 64 and 83. And the subtree size associated with 49 is 5, because you're counting everything underneath it.

How did we get these numbers? Well you want to think about this as you started with an empty set, and you kept inserting into it. And you were doing a sequence of insert and delete operations. And if I explain to you how an insert operation modifies these numbers, that is pretty much all you need. And of course, analogously, for a delete operation.

So what would happen for, let's say you wanted to insert 43? You would insert 43 at this point. And what you'd do is you follow the insertion path just like you did before. But when you're following that path you're going to increment the nodes that you're seeing by 1.

So you're going to add 43 to this. And you'd add 5 plus 1, because you see 49. And then you would go down and you'd see 46. And so you'd add 1 to that. And then finally, you add 43 and you assign-- since it's a leaf-- you'd assign to value corresponding to the subtree size of this new node that you put in there, to be 1.

It guess a little, teensy bit more complicated when you want to do the k minute check. But from a complexity standpoint, if you're not worried about constant factors, you can just say, you know what? I'm going to first run the regular insert, ignoring the subtree sizes. And if it fails, I'm done. Because I'm not going to modify the BST, and I'm done. I'm not going to have to modify the subtree sizes.

If it succeeds, then I'm going to go in, and I know now that I can increment each of these nodes, because I know I'm going to be successful. So that's sort of a trivial way of solving this problem, that from an asymptotic complexity standpoint gives you your order h augmented insert. That make sense?

Now you could do something better than that. I mean, I would urge you, if you had wrote something that-- we asked you to write something like this, to create a single procedure that essentially uses a recursion appropriately to do the right thing in one pass through the BST. And we'll talk about things like that as we go along in sections, and possibly in lectures.

So that's the subtree insert delete. Everyone buy that? Yeah, question back there.

AUDIENCE: If I wanted to delete a number, like let's say 79--

PROFESSOR: Yep?

AUDIENCE: --would we have to take it out and then rewrite the entire BST?

PROFESSOR: What you'd have to do is a bubble up pointers. So you'd have to actually have 64 connected to-- what will happen is 83 would actually come up, and you would essentially have some thing-- this is not quite how it works-- but 83 would move up and you'd have 64 to the left. That's what would happened for delete in this case.

So you would have to move pointers in the case of delete. And we're not done with binary search tree operations from a standpoint of teaching you about them. We'll talk about them not just in today's lecture, but later as well.

So there's one thing missing here, though, which is I haven't quite figured out-- I've told you how these subtree sizes work. But it's not completely clear, this is the last thing we have to do, is how are you going to compute rank t from the subtree sizes?

So everyone understand subtree sizes? It's just the number of nodes that are underneath you. And you remember to count yourself, all right?

Now what is rank t? Rank t is how many planes are scheduled to land at times less than or equal to t. So now I have a BST structure that looks like the one and I just ended up with. So I've added this 43. And so let me draw that out here, and see if we can answer this question. This is a subtle question.

So I got 49, and that subtree size is 6. I got 46, subtree size is 2. 43, 79, 64. and 83. So what I want is what lands before t?

And how do I do that? Give me an algorithm that would allow me to compute in order h time. I want to do this in order h time. What lands before t? Someone? Yeah.

AUDIENCE: So first you would have to find where to insert it, like we did before.

PROFESSOR: Right, right.

AUDIENCE: And then because we have the order of whatever it was before-- not the order, the--

PROFESSOR: The sizes? The sizes? Yeah.

AUDIENCE: And then we can look what's more than it on the right, we can subtract it and we get--

PROFESSOR: What is more than it on the right. Do you want to say--

AUDIENCE: Because, like--

PROFESSOR: OK.

AUDIENCE: --on the right--

PROFESSOR: Right.

AUDIENCE: --and then we can take this minus this and we get what's left.

PROFESSOR: That's great, that's excellent. Excellent. So I'm going to do it a little bit differently from what you described. I'm going to actually do it in a, sort of, a more positive way, no offense intended.

What we're going to do is we're going to add up the things that we want to add up. And what you have to do is walk-- your first step was right on. I mean, your answer is correct. I'm just going to do it a little bit differently.

You walk down the tree to find the desired time. This is just your search. We know how to do that.

As you walk down you add in the nodes that is the subtree sizes-- you're just adding in the notes here. So if you see-- depending on the number of nodes that you see as you're going deeper in, you want to add in the nodes. And you're going to add one to that, corresponding to the nodes that are smaller. And we're going to add in the subtree sizes to the left, as opposed to subtracting.

That may not make a lot of sense. But I guarantee you it will once we do an example.

So what's going on here? I want to find a place to insert. I'm not actually going to do the insert. Think of it is doing a lookup.

And along the way, I need to figure out the less than operator. I want to find all of the things that are less than this value I'm searching for. And so I have to do a bit of arithmetic.

So let's say that I'm looking for what's less than or equal to 79. So t equals 79. So I'm going to look at 49. I'm going to walk down, I'm going to look at 49.

And because I say I'm looking at 49-- and 49 is clearly less than 79. So I'm going to add 1. And that's this check over here.

I move on and what I need to do now is move to the right, because 79 is greater than 49. That's how my search would work. But because I've moved to the right, I'm going to add the subtree sizes that were to the left. Because I know that all of the things to the left are clearly less than 79.

So I'm going to add 2, corresponding to a subtree 46. So I'm not actually looking there. But I'm going to add all of that stuff in. I'm going to move to the right, and now I'm going to see 79.

At this point 79 is less than or equal to 79. So I'm going to see 79 and I'm going to add 1. And because I've added 79, just like I did with 49, I have to add the subtree size to the left of 79.

So the final addition is I add 1 corresponding to the subtree 64. And at this point I've discovered where I have to insert, I've essentially found the location, it matches 79. And there was no modification required in this algorithm. So if that was 78 you'd essentially do the same things.

But you're done because you found the value, or the place that you want to insert. And you've done a bunch of additions. And you go look at add 1, add 2, add 1, add 1, and you have 5. And that's the correct answer, as you can see from this example.

So what's the bad news? The bad news was what this lady said up front, which was we haven't quite solved the problem. Because sadly, I could easily set things up such that the height h is order n, h could be order n.

And if, for example, I gave you a sorted list, and I said insert into binary search tree that's originally null 43, and you put 43 in there. Then I say insert 46. And then I say instead of 48. And then I say insert 49, et cetera. And, you know, these could be any numbers.

Then you see that what does this look like? Does it look like a tree? It looks like a list. That's the bad news.

And I'll let Eric give you good news next week. We need to have this notion of balanced binary search trees.

So everything I've said is true. I did not lie. But the one extra thing is we need to make sure these trees are balanced so h is order log n. And then everything I said works. All right? See you next time.