Here 2.

Good morning. Today we're going to talk about

augmenting data structures.

That one is 23 and that is 23. And I look here. For this one, And this is a -- Normally, rather than designing Or, once again, data structures from scratch, you tend to take existing data structures and build your functionality into them.

And that is a process we call data-structure augmentation.

And this also today marks sort of the start of the design phase of the class. We spent a lot of time doing analysis up to this point. And now we're still going to learn some new analytical techniques.

But we're going to start turning our focus more toward how is it that you design efficient data structures, efficient algorithms for various problems?

So this is a good example of the design phase.

It is a really good idea, at this point, if you have not done so, to review the textbook Appendix B. You should take that as additional reading to make sure that you are familiar, because over the next few weeks we're going to hit almost every topic in Appendix B. It is going to be brought to bear on the subjects that we're talking about.

If you're going to go scramble to learn that while you're also trying to learn the material, it will be more onerous than if you just simply review the material now.

We're going to start with an illustration of the problem of dynamic order statistics.

We are familiar with finding things like the median or the kth order statistic or whatever. Now we want to do the same thing but we want to do it with a dynamic set.

Rather than being given all the data upfront, we're going to have a set. And then at some point somebody is going to be doing typically insert and delete.

And at some point somebody is going to say OK, select for me the ith largest guy or the ith smallest guy -- -- in the dynamic set. Or, something like OS-Rank of x. The rank of x in the sorted order of the set.

So either I want to just say, for example, if I gave n over 2, if I had n elements in the set and I said n over 2, I am asking for the median.

I could be asking for the mean. I could be asking for quartile.

Here I take an element and say, OK, so where does that element fall among all of the other elements in the set?

And, in addition, these are dynamic sets so I want to be able to do insert and delete, I want to be able to add and remove elements. The solution we are going to look at for this one, the basic idea is to keep the sizes of subtrees in the nodes of a red-black tree.

Let me draw a picture as an example.

In this tree -- I didn't draw the NILs for this.

I am going to keep two values. I am going to keep the key.

And so for the keys, what I will do is just use letters of the alphabet.

And this is a red-black tree. Just for practice, how can I label this tree so it's a red-black tree?

I haven't shown the NILs. Remember the NILs are all black. How can I label this, red and black? Make sure it is a red-black tree. Not every tree can be labeled as a red-black tree, right?

This is good practice because this sort of thing shows up on quizzes. Make F red, good, and everything else black, that is certainly a solution.

Because then that basically brings the level of this guy up to here. Actually, I had a more complicated one because it seemed like more fun.

What I did was I made this guy black and then these two guys red and black and red, black and red, black and black. But your solution is perfectly good as well. So we don't have any two reds in a row on any path. And all the black height from any particular point going down we get the same number of blacks whichever way we go. Good.

The idea here now is that, we're going to keep the subtree sizes, these are the keys that are stored in our dynamic set, we're going to keep the subtree sizes in the red-black tree.

For example, this guy has size one.

These guys have size one because they're leaves.

And then we can just work up. So this has size three, this guy has size five, this guy has size three, and this guy has five plus three plus one is nine.

In general, we will have size of x is equal to size of left of x plus the size of the right child of x plus one.

That is how I compute it recursively.

A very simple formula for what the size is.

It turns out that for the code that we're going to want to write to implement these operations, it is going to be convenient to be talking about the size of NIL.

So what is the size of NIL? Zero.

Size of NIL, there are no elements there.

However, in most program languages, if I take size of NIL, what will happen? You get an error.

That is kind of inconvenient. What I have to do in my code is that everywhere that I might want to take size of NIL, or take the size of anything, I have to say, well, if it's NIL then return zero, otherwise return the size field, etc. There is an implementation trick that we're going to use to simplify that.

It's called using a sentinel.

A sentinel is nothing more than a dummy record.

Instead of using a NIL, we will actually use a NIL sentinel. We will use a dummy record for NIL such that size of NIL is equal to zero.

Instead of any place I would have used NIL in the tree, instead I will have a special record that I will call NIL.

But it will be a whole record. And that way I can set its size field to be zero, and then I don't have to check that as a special case. That is a very common type of programming trick to use, is to use sentinels to simplify code so you don't have all these boundary cases or you don't have to write an extra function when all I want to do is just index the size of something. Everybody with me on that?

So let's write the code for OS-Select given this representation.

And this is going to basically give us the ith smallest in the subtree rooted at x. It's actually going to be a little bit more general. If I want to implement the OS-Select i of up there, I basically give it the root n_i. But we're going to build this recursively so it's going to be helpful to have the node in which we're trying to find the subtree.

Here is the code.

This is the code. And let's just see how it works and then we will argue why it works.

As an example, let's do OS-Select of the root and 5. We're going to find the fifth largest in the set. We have OS-Select of the root and 5. This is inconvenient.

We start out at the top, well, let's just switch the boards. Here we go.

We start at the top, and i is the root.

Excuse me, i is 5, sorry, and the root.

i=5. We want to five the fifth largest. We first compute this value k.

k is the size of left of x plus What is that value? What is k anyway?

What is it? Well, in this case it is 6.

Good. But what is the meaning of k?

The order. The rank.

Good, the rank of the current node.

This is the rank of the current node.

k is always the size of the left subtree plus 1.

That is just the rank of the current node.

We look here and we say, well, the rank is k.

Now, if it is equal then we found the element we want.

But, otherwise, if i is less, we know it's going to be in the left subtree.

All we're doing then is recursing in the left subtree.

And here we will recurse. We will want the fifth largest one. And now this time k is going to be equal to what? Two.

Now here we say, OK, this is bigger, so therefore the element we want is going to be in the right subtree. But we don't want the ith largest guy in the right subtree, because we already know there are going to be two guys over here.

We want the third largest guy in this subtree.

We have i equals 3 as we recurse into this subtree.

And now we compute k for here. This plus 1 is 2.

And that says we recursed right here.

And then we have i=1, k=1, and we return in this code a pointer to this node.

So this returns a pointer to the node containing H whose key is H. Just to make a comment here, we discovered k is equal to the rank of x.

Any questions about what is going on in this code?

OK. It's basically just finding its way down. The subtree sizes help it make the decision as to which way it should go to find which is the ith largest. We can do a quick analysis.

On our red-black tree, how long does OS-Select take to run? Yeah?

Yeah, order log n if there are n elements in the tree.

Because the red-black tree is a balance tree.

Its height is order log n. In fact, this code will work on any tree that has order log n the height of the tree.

And so if you have a guaranteed height, the way that red-black trees do, you're in good shape. OS-Rank, we won't do but it is in the book, also gets order log n.

Here is a question I want to pose.

Why not just keep the ranks themselves?

Yeah? It's the node itself.

Otherwise, you cannot take left of it.

I mean, if we were doing this in a decent language, strongly typed language there would be no confusion.

But we're writing in this pseudocode that is good because it's compact, which lets you focus on the algorithm. But, of course, it doesn't have a lot of the things you would really want if you were programming things of scale like

type safety and so forth. Yeah?

It is basically hard to maintain when you modify it.

For example, if we actually kept the ranks in the nodes, certainly it would be easy to find the element of a given rank.

But all I have to do is insert the smallest element, an element that is smaller than all of the other elements.

And what happens? All the ranks have to be changed. Order n changes have to be made if that's what I was maintaining, whereas with subtree sizes that's a lot easier.

Because it's hard to maintain -- -- when the red-black tree is modified.

And that is the other sort of tricky thing when you're augmenting a data structure. You want to put in the things that your operations go fast, but you cannot forget that there are already underlying operations on the data structure that have to be maintained in some way.

Can we close this door, please?

Thank you. We have to look at what are the modifying operations and how do we maintain them.

The modifying operations for red-black trees are insert and delete. If I were augmenting a binary heap, what operations would I have to worry about?

If I were augmenting a heap, what are the modifying operations? Binary min heap, for example, classic priority queue?

Who remembers heaps? What are the operations on a heap? There's a good final question.

Take-home exam, don't worry about it.

Final, worry about it. What are the operations on a heap? Just look it up on Books24 or whatever it is, right?

AnswerMan? What does AnswerMan say?

OK. And? If it's a min heap. It's min, extract min, typical operations and insert. And of those which are modifying? Insert and extract min, OK? So, min is not.

You don't have to worry about min because all that is is a query. You want to distinguish operations on a dynamic data structure those that modify and those that don't, because the ones that don't modify the data structure are all

perfectly fine as long as you haven't destroyed information.

The queries, those are easy.

But the operations that modify the data structure, those we're very concerned about in making sure we can maintain. Our strategy for dealing with insert and delete in this case is to update the subtree sizes -- -- when inserting or deleting. For example, let's look at what happens when I insert k.

Element key k. I am going to want to insert it in here, right? What is going to happen to this subtree size if I am inserting k in here?

This is going to increase to And then I go left. This one is going to increase to 6. Here it is going to increase to

And then I will put my k down there with a 1.

So I just updated on the way down.

Pretty easy. Yeah?

But now it's not a red-black tree anymore.

You have to rebalance, so you must also handle rebalancing. Because, remember, and this is something that people tend to forget so it's always, I think, helpful when I see patterns going on to tell everybody what the pattern is so that you can be sure of it in your work that you're not falling into that pattern. What people tend to forget when they're doing red-black trees is they tend to remember the tree insert part of it, but red-black insert, that RB insert procedure actually has two parts to it.

First you call tree insert and then you have to rebalance.

And so you've got to make sure you do the whole of the red-black insert. Not just the tree insert part.

We just did the tree insert part.

That was easy. We also have to handle rebalancing. So there are two types of things we have to worry about. One is red-black color changes.

Well, unfortunately those have no effect on subtree sizes.

If I change the colors of things, no effect, no problem. But also the interesting one is rotations. Rotations, it turns out, are fairly easy to fix up. Because when I do a rotation, I can update the nodes based on the children.

I will show you that. You basically look at children and fix up, in this case, in order one time per rotation.

For example, imagine that I had a piece of my tree that looked like this.

And let's say it was 7, 3, 4, the subtree sizes.

I'm not going to put the values in here.

And I did a right rotation on that edge to put them the other way. And so these guys get hooked up this way. Always the three children stay as three children. We just swing this guy over to there and make this guy be the parent of the other one.

And so now the point is that I can just simply update this guy to be, well, he's got 8, 3 plus 4 plus 1 using our formula for what the size is. And now, for this one, it's going to be 8 plus 7 plus 1 is 16, or, if I think about it, it's going to be whatever that was before because I haven't changed this subtree size with a rotation.

Everything beneath this edge is still beneath this edge.

And so I fixed it up in order one time.

There are certain other types of operations sometimes that occur where this isn't the value.

If I wasn't doing subtree sizes but was doing some other property of the subtree, it could be that this was no longer 16 in which case the effect might propagate up towards the root. There is a nice little lemma in the book that shows the conditions under which you can make sure that the re-balancing doesn't cost you too much.

So that was pretty good. Now, insert and delete, that is all we have to do for rotations, are therefore still order log n time, because a red-black tree only has to do order one rotations. Do they normally take constant time? Well, they still take constant time. They just take a little bit bigger constant. And so now we've been able to build this great data structure that supports dynamic order statistic queries and it works in order log n time for insert, delete and the various queries. OS-Select.

I can also just search for an element.

I have taken the basic data structure and have added some new operations on it. Any questions about what we did here? Do people understand this reasonably well? OK.

Then let's generalize, always a dangerous thing.

Augmenting data structures. What I would like to do is give you a little methodology for how you go about doing

this safely so you don't forget things. The most common thing, by the way, if there is an augmentation problem on the take-home or if there is one on the final, I guarantee that probably a quarter of the class will forget the rotations if they augmented red-black tree. I guarantee it.

Anyway, here is a little methodology to check yourself.

As I mentioned, the reason why this is so important is because this is, in practice, the thing that you do most of the time.

You don't just use a data structure as given.

You take a data structure. You say I have my own operations I want to layer onto this.

We're going to give a methodology.

And what I will do, as I go along, is will use the example of order statistics trees to illustrate the methodology. It is four steps.

The first is choose an underlying data structure.

Which in the case of order statistics tree was what?

Red-black tree.

And the second thing we do is we figure out what additional information we wish to maintain in that data structure.

Which in this case is the subtree sizes.

Subtree sizes is what we keep for this one.

And when we did this we could make mistakes, right? We could have said, oh, let's keep the rank. And we start playing with it and discover we can do that. It just goes really slowly.

It takes some creativity to figure out what is the information that you're going to be able to keep, but also to maintain the other properties that you want.

The third step is verify that the information can be maintained -- -- for the modifying operations on the data structure.

And so in this case, for OS trees, the modifying operations were insert and delete.

And, of course, we had to make sure we dealt with rotations.

And because rotations are part of that we could break it down into the tree insert, the tree delete and rotations.

And once we've did that everything was fine.

We didn't, for this particular problem, have to worry about color changes. But that's another thing that under some things you might have to worry about.

For some reason the color made a difference.

Usually that doesn't make a difference.

And then the fourth step is to develop new operations.

Presumably that use the info that you have now stored.

And this was OS-Select and OS-Rank, which we didn't give but which is there. And also it's a nice little puzzle to figure out yourself, how you would build OS-Rank.

Not a hard piece of code. This methodology is not actually the way you do this. This is one of these things that's more like a checklist, because you see whether or not you've got -- When you're actually doing this maybe you developed the new operations first.

You've got to keep in mind the new operations while you're verifying that the information you're storing can be here.

Maybe you will then go back and change this and sort of sort through it. This is more a checklist that when you're done this is how you write it up.

This is how you document that what you've done is, in fact, a good thing. You have a checklist.

Here is my underlying data structure.

Here is the addition information I need.

See, I can still support the modifying operations that the data structure used to have and now here are my new operations and see what those are. It's really a checklist.

Not a prescription for the order in which you do things.

You must do all these steps, not necessarily in this order.

This is a guide for your documentation.

When we ask for you to augment a data structure, generally we're asking you to tell us what the four steps are.

It will help you organize your things.

It will also help make sure you don't forget some step along the way. I've seen people who have added the information and developed new operations but completely forgot to verify that the information could be maintained.

So you want to make sure that you've done all those.

Usually you have to play -- -- with interactions -- -- between steps. It's not just a do this, do this, do this. We're going to do now a more complicated data structure. It's not that much more complicated, but its correctness is actually kind of challenging.

And it is actually a very practical and useful data structure. I am amazed at how many people aren't aware that there are data structures of this nature that are useful for them when I see people writing really slow code.

And so the example we're going to do is interval trees.

And the idea of this is that we want to maintain a set of intervals. For example, time intervals. I have a whole database of time intervals that I'm trying to maintain.

Let's just do an example here.

This is going from 7 to 10, 5 to 11 and 4 to 8, from 15 to 18, 17 to 19 and 21 to 23.

This is a set of intervals. And if we have an interval i, let's say this is interval i, which is 7,10.

We're going to call this endpoint the low endpoint of i and this we're going to call the high endpoint of i.

The reason I use low and high rather than left or right is because we're going to have a tree, and we're going to want the left subtree and the right subtree.

So if I start saying left and right for intervals and left and right for tree we're going to get really confused.

This is also a tip. Let me say when you're coding, you really have to think hard sometimes about the words that you're using for things, especially things like left and right because they get so overused throughout programming.

It's a good idea to come up with a whole wealth of synonyms for different situations so that it is clear in any piece of code when you're talking, for example, about the intervals versus the tree, because we're going to have both going on here. And what we're going to do is we want to support insertion and deletion of intervals here.

And we're going to have a query, which is going to be the new operation we're going to develop, which is going to be to find an interval, any interval in the set that overlaps a given query interval.

So I give you a query interval like say 6, 14 and you can return this guy or this guy, this guy, couldn't return any of these because these are all less than 14.

So I can return any one of those.

I only have to return one. I just have to find one guy that overlaps. Any question about what we're going to be setting up here? OK.

Our methodology is we're going to pick, first of all, step one. And here is our methodology.

Step one is we're going chose underlying data structure.

Does anybody have a suggestion as to what data structure we ought to use here to support interval trees?

What data structure should we try to start here to support interval trees? Anybody have any idea?

A red-black tree. A binary search tree.

Red-black tree. We're going to use a red-black tree.

Oh, I've got to say what it is keyed on.

What is going to be the key for my red-black tree?

For each interval, what should I use for a key?

This is where there are a bunch of options, right?

Throw out some ideas. It's always better to branch than it is to prune. You can always prune later, but if you don't branch you will never get the chance to prune. So generation of ideas.

You'll need that when you're doing the design phase and doing the take-home exam. Yeah?

We're calling that the low endpoint.

OK, you could do low endpoint. What other ideas are there?

High end point. Now you can look at low endpoint, high endpoint. Well, between low and high which is better? That one is not going to matter, right? So doing high versus low, we don't have to consider that, but there is another natural point you want to think about using like the median, the middle point. At least that is symmetric.

What do you think? What else might I use?

The length? I think the length doesn't feel to me productive. This is just purely a matter of intuition. It doesn't feel productive, because if I know the length I don't know where it is so it's going to be hard to maintain information about where it is for queries. It turns out we're going to use the low left endpoint, but I think to me that was sort of a surprise that you'd want to use that and not the middle one.

Because you're favoring one endpoint over the other.

It turns out that's the right thing to do, surprisingly.

There is another strategy. Actually, there's another type of tree called a segment tree. Actually, what you do is you store both the left and right endpoints separately in the tree. And then you maintain a data structure where the line segments go up through the tree on to the other. There are lots of things you can do, but we're just going to keep it keyed on the low endpoint. That's why this is a more clever data structure in some ways.

Now, this is harder. That is why this is a clever data structure. What are we going to store in the -- I think any of those ideas are good ideas to throw out and look at.

You don't know which one is going to work until you play with it. This one, though, is, I think, much harder to guess.

You're going to store in a node the largest value, I will call it m, in the subtree rooted at that node.

We'll draw it like this, a node like this.

We will put the interval here and we will put the m value here.

Let's draw a picture.

Once again, I am not drawing the NILs.

I hope that that is a search tree that is keyed on the low left endpoint. 4, 5, 7, 15, 17, 21. It is keyed on the low left endpoint. If this a red-black tree, let's just do another practice. How can I color this so that it is a legal red-black

tree? Not too relevant to what we're doing right now But a little drill doesn't hurt sometimes. Remember, the NILs are not there and they are all black. And the root is black.

I will give that one to you.

Good. This will work.

You sort of go through a little puzzle.

A logic puzzle. Because this is really short so it better not have any reds in it.

This has got to be black. Now, if I'm going to balance the height, I have got to have a layer of black here.

It couldn't be that one. It's got to be these two.

Good. Now let's compute the m value for each of these. It's the largest value in the subtree rooted at that node. What's the largest value in the subtree rooted at this node? And in this one? In this one?

In general, m is going to be the maximum of three possible values. Either the high point of the interval at x or m of the left of x or m of the right of x.

Does everybody see that? It is going to be m of x for any node. I just have to look, what is the maximum here, what is the maximum here and what is the high point of the interval.

Whichever one of those is largest, that's the largest for that subtree.

The modifying operations.

Let's first do insert. How can I do insert?

There are two parts. The first part is to do the tree insert, just a normal insert into a binary search tree.

What do I do? Insert a new interval?

Insert a new interval here? How can I fix up the m's?

That's right. You just go down the tree and look at my current interval. And if it's got a bigger max, this is something that is going into that subtree.

If its high endpoint is bigger than the current max, update the current max. I just do that as I'm going through the insertion, wherever it happens to land up in every subtree that it hits, every node that it hits on the way down. I just

update it with the maximum wherever it happens to fall.

Good. You just fix them on the way down.

But we also have to do the other section.

Also need to handle rotations.

So let's just see how we might do rotations as an example.

Let's say this is 11, 15, 30.

Let's say I'm doing a right rotation.

This is coming off from somewhere.

That is coming off. This is still going to be the child that has 30, the one that 14 and the one that has 19. And so now we've rotated this way, so this is the 11, 15 and this is the 6, I just use my formula here. I just look here and say which is the biggest, 14, 15 or 19?

Which is the biggest? 30, 19 or 20?

it turns out, not too hard to show, that it's always whatever was there, because we're talking about the biggest thing in the subtree.

And the membership of the subtree hasn't changed when we do the rotation. That just took me order one time to fix up.

Fixing up the m's during rotation takes O(1) time.

So the total insert time is O(lg n).

Once I figured out that this is the right information, of course we don't know what we're using this information for yet. But once I know that that is the information, showing you that it works in certain delete continuing work in order log n time is easy.

Now, delete is actually a little bit trickier but I will just say it is similar. Because in delete you go through and you find something, you may have to go through the whole business of swapping it. If it's an internal node you've got to swap it with its successor or predecessor.

And so there are a bunch of things that have to be dealt with, but it is all stuff where you can update the information using this thing. And it's all essentially local changes when you're updating this information because you can do it essentially only on a path up from the root and most of the tree is never dealt with. I will leave that for you folks to work out. It's also in the book if you want to cheat, but it is a good exercise.

Any questions about the first three steps?

Fourth step is new operations.

Interval search of i is going to find an interval that overlaps the interval i. So i here is an interval.

It's got two coordinates. And this, rather than writing recursively, we're going to write as, it's sort of going to be recursive, but we're going to write it with a while loop. You could write it recursively.

The other one that we wrote, we could have written as a while loop as well and not had the recursive call.

Here we're going to basically just start x gets the root.

And then while -- That is the code. Let's just see how it works.

Let's search for the interval 14, 16 -- -- in this tree. Let's see.

x starts out at the root. And while it is not NIL, and it's not NIL because it's the root, what is this doing?

Somebody tell me what that code does.

Well, what is this doing? This is testing something between i and int of x. Int of x is the interval stored at x. What is this testing for?

I hope I got it right.

What is this testing for? Yeah?

Above or below? I need just simple words.

Test for overlaps. In particular test whether they do or don't?

Do? Don't?

If I get to this point, what do I know about i and int of x? Don't overlap.

They don't overlap because the high of one is smaller than the low of the other. The high of one is smaller than

the low of the other. They don't overlap that way.

Could they overlap the other way?

No because we're testing also whether the low of the one is bigger than the high of the other.

They're saying it's either like this or like this.

This is testing not overlap. That makes it simpler.

When I'm searching for 14, 16, I check here.

And I say do they overlap? And the answer is, now we can understand it without having to go through all the arithmetic calculations, no they don't overlap.

If they did overlap, I found what I want.

And what's going to happen? I am going to drop out of the while loop and just return x, because I will return something that overlaps. That is my goal.

Here it says they don't overlap.

So then I say, well, if left of x is not NIL, in other words, I've got a left child and low of i is less than or equal to m of left of x, then we go left. What happens in this case if I'm searching for 14, 16?

Is the low of i less than or equal to m of left of x?

Low of i is 14. And I am searching.

And is it less than 18? Yes.

Therefore, what do I do? I go left and make x point to this guy. Now I check.

Does it overlap? No.

I take a look at the left guy. It is 8.

I compare 8 with 14, right?

And is it lower? No, so I go right.

And now I discover that I have an overlap here and it overlaps.

It returns then the 15, 18 as an overlapping one.

If I were searching for 12,

I would go up to the top. And I look, 12, 14, it doesn't overlap here. I look at the 18 and it is greater so I go left. I then look here.

Does it overlap? No.

So then what happens? I look at the left.

It says I go right. I look here.

Then I go and I look at the left.

It says, no, go right.

I go here, which is NIL, and now it is NIL.

I return NIL. And does 12, 14 overlap anything in the set? No.

So, therefore, it always works.

OK? OK.

We're going to do correctness in a minute, but let's just do our analysis first so we don't have to do it because the correctness is going to be a little bit tricky.

Time = O(lg n) because all I am doing is going down the tree.

It takes time proportional to the height of the tree.

That's pretty easy. If I need to list all overlaps, suppose I want to list all the overlaps, how quickly can I do that? Can somebody suggest how I could use this as a subroutine to list all overlaps?

Suppose I have k overlaps, k intervals that overlap my query interval and I want to find every single one of them, how fast can I do that?

How do I do it?

How do I do it? If I search a second time, I might get the same value.

Yeah, there you go. Do what?

When you find it delete it. Put it over to the side.

Find the next one, delete it until there are none left. And then, if I don't want to modify the data structure, insert them all back in.

It costs me k lg n if they are k overlaps.

That's actually called an output sensitive algorithm.

Because the running time of it depends upon how much it outputs, so this is output sensitive.

The best to date for this problem, by the way, of listing all is O(k+lg n) with a different data structure.

And, actually, that was open for a while as an open problem. OK. Correctness.

Why does this algorithm always work correctly?

The key issue of the correctness is that I am picking one way to go, left or right.

And that's great, as long as it is in that subtree. But how do I know that when I pick I decide I'm going to go left that it might not be in the right subtree and I went the wrong way?

Or, if I went right, that I accidentally left one out on the left side? We're always going just one direction. And that's sort of the cleverness of the code. The theorem is let's let L be the set of intervals i prime in the left of a node x.

And R be the set of i primes in the right of x.

And now there are two parts I am going to show.

If the search goes right then the set of i prime in L, such that i prime overlaps i is the empty set.

That's the first thing I do. If it goes right then there is nothing in the left subtree that overlaps.

It's always, whenever the code goes right, no problem, because there was nothing in the left subtree to be found. Does everybody understand what that says? We are going to prove this, but I want to make sure people understand.

Because the second one is going to be harder to understand so you've got to make sure you understand this one first.

Any questions about this? OK.

If the search goes left -- -- then the set of i prime in L such that i prime overlaps i empty set implies that i prime -- OK. What is this saying?

If the search goes left, if the left was empty, in other words, if you went left and you discovered that there was nothing in there to find, no overlapping interval to find then it is OK because it wouldn't have helped me to go right anyway because there is nothing in the right to be found.

So it is not guaranteeing that there is nothing to be found in the left, but if there happens to be nothing to find in the left it is OK because there was nothing to be found in the right either. That is what the second one says. In either case, you're OK to go the way. So let's do this proof.

Does everybody understand what the proof says?

Understanding the proof is tricky.

It's logic. Logic is tricky.

Suppose the search goes right. We'll do the first one.

If left of x is NIL then we are done since we proved what we wanted to prove. If we go right there are two possibilities, either we have left of x be NIL or left of x is not NIL. So if it is NIL we are OK because we said if it goes right I want to prove this, that the things in the left subtree that overlap is empty.

If there is nothing there, there is clearly nothing there that overlaps. Otherwise, the low of i is greater than m of the left of x.

If I look at x here, either x was NIL in the while statement here or this is true. We just said it is not NIL so let's take a look at, excuse me.

I'm on the wrong line. I am in this loop.

Left of x was not NIL and the low of i was this.

Which way am I going here? I am going right.

Therefore, this was not true. So either left of x was not NIL, which was the first one, or low of i is greater than m of left of x if I am going right.

If I'm going right one of those two had to be true.

The first one was easy. Otherwise, we have this, low of i is greater than m of left of x.

Now this has got to be that value.

m of left of x is the right endpoint, is the high endpoint of some interval in that subtree.

This is equal to the high of j for some j in L.

So m of left of x must be equal to the high of some endpoint because that's how we're picking the m's.

For some j in the left subtree. And no other interval in L has a larger high endpoint -- -- than high of j. If I draw a picture here, I have over here i and this is the low of i.

And I have j where we say its high endpoint is less than the low of i. This is j, and I don't know how far over it goes. And this has high of j -- -- which is the highest one in the left subtree.

There is nobody else who has got a higher right endpoint.

There is nobody else in this subtree who could possibly overlap I, because all of them end somewhere before this point.

This point is the highest one in a subtree.

Therefore, i prime in L such that i prime overlaps i is the empty set. And now the hard case.

Everybody stretch. Hard case.

Does everybody follow this? The point is that because this is the highest guy everybody else has to be left, so if you didn't overlap the highest guy you're not going to overlap anybody. Suppose the search goes left -- -- and that there is nothing to overlap in the left subtree.

I went left here but I am not going to find anything.

Now I want to prove that it wouldn't have helped me to go right. That's essentially what the theorem here says. That if I assume this it wouldn't have helped to go right.

I want to show that there is nothing in the right subtree.

So going left was OK because I wasn't going to find anything anyway. Similarly, we go through a similar analysis. Low of i is less than or equal to m of the left of x, which once again is equal to the high of j for some j in L. We are

just saying if I go left these things must be true. I went left.

Since j is in L it doesn't overlap i, because the set of things that overlap i in L is empty set.

Since j doesn't overlap i that implies that the high of i must be less than the low of j.

Since j is in L and it doesn't overlap i, what are the possibilities? We essentially have here, if I draw a picture, I have j and L and I have i here. The point is that it doesn't overlap it, therefore, it must be to the left because its low endpoint is less than this.

But it doesn't overlap it, therefore its high endpoint must be left of the low of this one.

Now we will use the binary search tree property.

That implies that for all i prime in R, everything in the right subtree, we have a low of j is less than or equal to low of i prime, so we're sorted on the low endpoints. Everything in the right subtree must have a low endpoint that starts to the right of the low endpoint of j because j in the left subtree.

And everything in the whole tree is sorted by low endpoints, so anything in the right subtree is going to start over here. Those are other things.

These are the i primes in R. We don't know how many there are, but they all start to the right of this point.

So they cannot overlap i either, therefore, there is nothing. All the i primes in R is also nobody.

Just to go back again, the basic idea is that since this guy doesn't overlap the guy who is in the left and everybody to the right is going to be further to the right, if I go left and don't find anything that's OK because I am not going to find anything over here anyway.

They are not going to overlap. Data-structure augmentation, great stuff. It will give you a lot of rich, rich data structures built on any ones you know, hash tables, heaps, binary search trees and so forth.