ERIK DEMAINE: All right, welcome back to Dynamic Optimality. This is the second of two lectures. And today we're going to focus mainly on lower bounds. So last time we saw this geometric connection to binary search trees. So again, this is about is there one best binary search tree. And we represented binary search trees, or at least the execution of those algorithms, as point sets in time space. And of course a point set corresponded to a valid execution of a BST tree where each of these points represented which nodes got touched during an access. If and only if the point set was arborally satisfied, meaning you take any two points in the point set, if they span a rectangle that is not just a horizontal or vertical line segment, there must be a third point somewhere inside that rectangle, which in the end implies that there's a monotone path between those two points.

And then we saw, on the upper bound side, we saw a greedy algorithm, which was the obvious offline thing to do, which is as these points come along, as you do the accesses, the white dots, you add the necessary red dots in order to make it arborally satisfied row by row. And so that seemed like the obvious offline thing to do. Turns out it could be done online up to constant factors. I sketched that last time. And this is conjectured to be within a constant factor of optimal. We can't prove it. What I'm going to show today is our best attempts at proving this is optimal. In particular, there's something called the signed greedy algorithm, which is almost the same as greedy, but it's a lower bound. And greedy is an upper bound. So all you need to do is show these two things are within a constant factor of each other and we're done.

We're not going to get there, obviously, because we haven't solved that yet. But along the way, we're going to see tango trees, which achieve this log log n competitive bound. So we think greedy is constant competitive. The best we know right now is log log n, this is an improvement over red black trees, which achieve log n. Any balance binary search tree is within a log n factor of optimal. So it's all between constant and log n that we're trying to do.

Another fun consequence of lower bounds is a particular sense in which log n is necessary for some access sequences. So I argued last time that if you take, like, a random access sequence, or-- for example, if you look at a binary search tree and you say, oh, I'll just access the thing that's deepest in the tree, there's always something that's deep in the tree of depth, at least log n. And so for any binary search tree there is an access sequence. No matter what

that binary search tree does, I can choose the next access to force you to take log n per operation.

What we're going to see today in these lower bounds is one access sequence that for all binary search trees, they must spend log n time. Just changing the quantifiers around. So instead of for every binary search tree there is an access sequence, there's going to be there is an access sequence such that for every binary search tree you need log n time. That's something we'll get easily out of these lower bounds.

So let's jump into the lower bounds. And we're going to cover you could say three different lower bounds. The independent rectangles is kind of a generic class of lower bounds. Then we're going to see two specific choices of these independent rectangles, which are actually older than this result. So this is sort of a modern interpretation of two older results and a more general result.

Signed greedy is going to turn out to be the best lower bound. Is It's better than all the ones that we will cover, but each of them has their own uses for analysis. Each of them is going to let us analyze an algorithm that we couldn't or that we don't otherwise know how to analyze.

So let's do the independent rectangle lower bound. The sort of generic one. So these lower bounds are all going to refer to the original point set, the white dots, the accesses. The idea is you're given an access sequence, a sequence x i-- x 1 up to x n, and you want to know some lower bound that every binary search tree requires a certain number of accesses, a certain number of node touches for that access sequence. You know it's at least n. You want something bigger than n. We've got to at least touch the nodes that are being accessed. I'm going to drop this.

So I want the notion of independent rectangles. And general idea of dependent rectangles would be something like this. Ah, I see.

So these are two rectangles. I consider them dependent because one of the corners is inside the other rectangle. This is true no matter where the points are. So, for example, if I take two points, they span a rectangle. If I take these two points, for example, they span a rectangle. This corner is inside that one. So these are considered dependent rectangles in either case. So corner here does not necessarily mean a point-- any of the four corners. Rectangle is defined by two points, but it has all four corners.

And so, in particular, independent rectangles-- for example, they might be completely disjoint. Those are going to be independent. Something like that is independent. But there are some other cases. You can have rectangles that look like this. OK? And it doesn't matter where the points are. Maybe here, here, here, and here. Or the other way. These are independent.

And there's one other kind of special case, which maybe I'll use color to draw the other one because they're right on top of each other. So I've got a point here, a point here, and a point here. These are two rectangles defined on three points. So they both use this point. And if you check, it does satisfy this condition. So no corner strictly inside the other.

But we also need that the rectangles are unsatisfied. So this is saying that there's no other point even on the boundary of the rectangle. So this part says, OK, there's nothing strictly inside. But we also need that on the boundary there's no other points. So this is the only sort of situation other than reflections where you get this working out as independent.

AUDIENCE: Last case is independent?

ERIK DEMAINE: Last case is independent. All right? So this is a definition. If I give you a set of rectangles, they're independent. I mean, I was looking at pairwise. But if they're are pairwise independent, then they will be independent. No corner of any rectangle strictly inside any other rectangle. And there's no points of those rectangles that are inside others. OK. Cool.

So what? Lower bound says the optimal offline binary search tree, or the optimal way to add dots to satisfy your point set, is going to be at least the size of the input-- meaning the number of initial points you have-- plus half the maximum number of independent rectangles.

OK. So this is a max independence set problem. In general, that's NP-complete. Turns out we'll be able to at least approximate the number of independent rectangles within a constant factor by the end of class. That's going to be signed greedy. So signed greedy is going to be the best way up to constant factors to choose independent rectangles. For now, someone magically tells you what's the best way or you just choose some reasonable-- any choice of independent rectangles will be a lower bound. But you get the best lower bound by choosing the max. OK?

So we're going to prove this theorem, and then we're going to see three different ways to choose those independent rectangles. And we'll use them for various things. Wilber 1, Wilber, 2 and signed greedy are going to be the three choices for independent rectangles.

All right. To prove this theorem, we're going to change it a little bit first. And this is kind of the focus of today-- is the idea of signed rectangles. If you look at the rectangles in the world spanned by two points, there are two different kinds. There's the top right, lower left kind. And then there's the top left, lower right kind. These are positive slope or negative slope. Those are the two kinds of rectangles.

And it's helpful to think about just the positive rectangles or the slash rectangles and just the backslash rectangles separately. So we're going to call a point set-- it's a little hard to pronounce-- we used to call this plus satisfied. So maybe it's easiest to pronounce it that way, the symbol formerly known as plus satisfied, if all plus rectangles that are not on a horizontal or vertical line contain another point.

So a point set is arborally satisfied if and only if it is plus satisfied and minus satisfied-- just breaking apart that definition into two parts. But now, we're going to look at point sets that are just plus satisfied or point sets that are just minus satisfied. And then we can look at the optimal solution if you only care about plus rectangles.

So this is the smallest plus satisfied point set containing all the access points, all the given points. So we'll call that the input. OPT was the smallest arborally satisfied. OPT plus is the-- you just look at plus rectangles.

OK. Why are we doing this? Well, for now, we're going to do it to prove this theorem. So lemma, which was what we're actually going to prove, is if you look at this OPT plus thing, it's got to be at least the size of the input-- everything has to at least contain the input-- plus maximum number of independent plus rectangles.

So this is where we're actually going to prove. If you want to get plus satisfied and you've got k independent plus rectangles, you need to add at least that many points-- so at least one point per plus rectangle. If you can prove this, you prove the theorem because this holds for minus just as well as plus by symmetry. And so you take your maximum independent set of rectangles. At least half of them are plus or at least half of them are minus. You apply this bound, and that's where you get the 1/2 here.

So this is stronger, I guess, than the theorem, and this is what we're actually going to prove. And so, in this world, we just are thinking about plus rectangles, which is a little weird. But it works.

And the proof is going to be in three steps. I'm first going to give you an overview of the steps, and then we'll actually do them. So this is like a two-level proof. First thing we do, the top level, is we're going to find a rectangle in the independent set, and we're going to find a vertical line that hits only that rectangle. So we're going to have some rectangle in the independent set, and we want a vertical line stabbing it such that no other rectangle is stabbed by this vertical line. So all other rectangles-- that's independence, so maybe they look something like this-- but nothing like this.

Not obvious that such a thing exists, but it does. Actually, not that hard to find. We just need some rectangle with some line. Then, using that property, we're going to be able to find some points in that rectangle that are also in the optimal plus solution in the rectangle crossing the line. Let me get another color.

So we're going to find a point on the left of the line and a point on the right of the line. And they're horizontally adjacent, meaning there's no other point between them. So we know there's some point in this box. Because this is a plus box, it has got to be satisfied somehow. And I claim there's actually two points on either side of the line. One of them could be equal to this or this, but not both obviously because they're horizontally aligned.

And then what we're going to do is charge the rectangle to those points. And then, basically, we're going to remove that rectangle and repeat. And the claim is this charging sort of only happens once per point. And therefore, the number of points in the optimal solution has to be at least the number of rectangles-- number of plus rectangles in the independent set.

So, basically, this is a way of ordering the rectangles. We're going to take one that has one of these vertical lines, find two points that pay for that rectangle, and therefore argue that OPT has to be at least the number of rectangles. So we have to argue that at least one of these points is not one of the original points. And that's where we're getting the input plus this.

So there's lots of things to check here. Let's do them one at a time. And throughout, I'm going to assume-- let me write that the bottom-- assume all x- and y-coordinates are unique. This is an idea I mentioned last time as well. If you have lots of accesses to the same key, imagine them being accesses to slightly different keys. Just skew them a little bit, and it doesn't change any of the bounds much. I won't to argue that here, but at the least think of this as just a simplifying assumption to make the proofs cleaner.

So how are we going to do step one? I need to find some rectangle and some vertical line that

only stabs that rectangle. And the way we're going to do that is just take the widest rectangle that just has the maximum x extent. There might be more than one, but just take one of them. So it's very wide. What this tells us is that there's no other rectangle like this. This would be independent, but it would be wider. So that's not allowed.

Now, we have to think about all sorts of scenarios. So we've got a point here and a point here. It could still be that we have rectangles like this. They just can't go farther to the right. It could be we have rectangles that go like this-- just can't go too far to the left. These rectangles that are anchored in the lower left and these rectangles that are anchored in the upper right can't touch each other because then one of them would be satisfied. This one's going to have a point down here. This one is going to have a point here. I guess-- yeah, let's see, how would it go if they were touching? We'd have a corner-- hmm, touching is a little weird. Ah, I see. Good. This can't happen because we assume the x-coordinates are distinct. So that's why I did this. That's the reason. So this can't happen.

And I also can't have them go like this because then there's a corner in the strict interior of the other rectangle. Is that clear? This rectangle can't come over here because then that would be not independent. Rectangle can't come right to the same spot because there is no same spot. That would be two points on the same vertical line. And so what we must have is a picture more like this where there's an empty region in between. that not hit by-- there can be many of these rectangles, many of these rectangles. They're independent from each other. That's like this case here.

There can also be some rectangles like this. But by the same argument, these guys can't touch each other and they can't overlap horizontally because then one of the corners would be inside the other. Question?

AUDIENCE: For that picture, you drew a rectangle under the other one.

ERIK DEMAINE: This one or this one?

AUDIENCE: That one.

ERIK DEMAINE: Yeah, this one cannot happen. That's what we claim-- haha, right. So you're right. So we worry about-- interesting. Well, we worry about something like this.

AUDIENCE: Sorry, why can't that happen?

ERIK DEMAINE: Yeah, you're right. I actually drew the wrong picture. Sorry. Kidding.

Yeah. I really meant line segment here. I'm sorry. Poor choice of wording. So vertical line is actually just going to go the extent of the rectangle-- something like this. Sorry. We can't forbid rectangles like this. What we can forbid our rectangles like this that also try to cross that segment. We'll see why this is enough in a moment. Sorry about that.

I really only care about the interior of this rectangle. I'm trying to get a vertical line that only stabs this rectangle, nothing else-- inside the rectangle. Sorry, poor wording. I don't care about these guys outside because I can't say anything about them. They could be all over the place in an independent set. I mean, relative to what hits this rectangle, there's stuff on the left. There's stuff on the right. There are these guys. There can also be things like this.

But still remaining are these regions which are not hit by any rectangles, and that's because what I was saying. These guys can't touch each other because then there would be equal x-coordinates. They can't overlap because then one of them would not be independent from the other. So I get my vertical lines. I just need one, but it could be any of these. In general, for example, if you take all of these lower left anchored rectangles and take just to the right of the rightmost one, that will be a valid choice for your line. Because you can argue none of these can overlap it.

So that's step one. We just take a widest rectangle. The one thing we needed to forbid was something going like this all the way across.

Step two. Step two is actually pretty easy. Once you've identified this red line-- inside the rectangle, you know there are some points. And I'm going to take the rightmost. And then among all the rightmost points, I'm going to take the topmost point that is to the left of the line and inside the rectangle. So let p be the topmost, leftmost point-- sorry, rightmost-- that is both in the rectangle and left of the line.

Let me erase this one for a little bit more room. So I'm looking at all of this region to the left of the line in the rectangle. I want to take the rightmost and then topmost point-- something like this. How do I know such a point exists? Because this point is such a point. And this point is to the left of the line. So if there's nothing else in here, that is a valid choice. But in general, we go to the right as much as possible. Then we go up as much as possible. So that's a point, which we will call them p.

AUDIENCE: Couldn't it be on the border of the rectangle?

ERIK DEMAINE: It could be on the border. It could be interior. We don't know.

AUDIENCE: When you said topmost, what is your topmost?

ERIK DEMAINE: Topmost means of maximum y-coordinate.

AUDIENCE: Oh, OK. Got it.

ERIK DEMAINE: So it could be up here. We don't know. First, we go rightmost. Then, among all the things in that column, we go to the topmost one. So it might be on the top. It might not be.

These are points-- sorry, this is a point in OPT plus.

And then q is going to be a similar thing. It's going to be the bottom-most, leftmost point in OPT plus that is in the rectangle and right of the line. Not totally symmetric, though. We're also going to say and not below p.

So now we're looking at this upper region here. Among all the things that are not below p-- should have drawn this more horizontal-- and to the right of the red line-- so that's up to here, I guess-- I want to take the leftmost column that has any points in it and then among those take the bottom-most point in the column. I claim that's actually going to be on this line.

First thing to check is that such a point exists. Such a point exists because, in particular, this is such a point. It is to the right of the line. It's above the blue line, right of the red line, in the rectangle. But I claim that if we take the leftmost, bottom-most one, then they must actually be aligned.

Why? So if it was somewhere else, like up here or like this point, then I claim that is an unsatisfied box. Let me draw that picture, make a little clearer. So something like this. So we've got our red line and we've got this picture. This is p, and then actually we don't know anything about down here. This is q.

I claim that these black regions cannot have any points in them because, by definition, p was in the rightmost column. So there's nothing in this strip and between p and the red line. And it was the topmost within the column, which means there's nothing above p in the column. So that's why all the points are confined to this region over here.

Similarly, for q, if you look at the things that are above or on this horizontal line, which was the blue line over there, then we know that there's nothing in this strip in between because q is leftmost. And then among leftmost, it was bottom-most, so there's nothing down here. So that means, if these guys are not horizontally aligned, there is an unsatisfied box here. Contradiction. It's a plus box. So in OPT plus, there's got to be another point, which was a contradiction. So in fact, p and q must be horizontally aligned. So that was step two.

Finally, we get step three. So the idea with step three-- now we're going to do a charging argument. We want to say, OK, basically, for every independent rectangle, we want to find a point that's in OPT that was not in the original input. And therefore, then OPT plus has to be at least the size of the input plus 1 per each of these plus rectangles.

So the idea is the following. Because of all this set-up-- because we made pq horizontally aligned-- they're inside the rectangle. And furthermore, they're adjacent and they cross this vertical line. And that vertical line is not crossed by any other rectangle. When I say line, I mean line segment. There's no other rectangle that hits this red thing. Therefore, these two points are not going to get charged as a pair ever again. If you remove this rectangle, repeat this process, pq is never going to get charged again. So we charge to pq. And the pair never charged again, never be charged by another rectangle because no rectangle hits the red thing. So no rectangle contains the segment pq, the horizontal segment pq.

So this is almost what we want. We really want a single point which is not in the input. So we have p and q. They're horizontally aligned. Now, if they're horizontally aligned, we know that not both of them are in the original input because all y-coordinates are distinct. This is usually true because you're only accessing one point per row, per time step.

So one of these might be in the input, but the other one is not. So that's the one I want to hold onto. And say, OK, that's a point added to OPT plus that pays for this rectangle.

It's not quite so simple, though, because we might have a whole bunch of horizontally-aligned things. And one rectangle charges to this one. One rectangle charges to this one. One rectangle charges to this pair. That's OK, though, because here we have four points. Again, one of them could be in the input. The other three have to be added. And so you've got three rectangles, three added points, and we're happy. Question?

AUDIENCE: Just to make the argument formal, wouldn't you want to say that only when your saying assume that x and y are always distinct-- but then, if you have the same either x or y--

ERIK DEMAINE: Ah, good point. So this is distinct in the input is what I meant. Obviously, in OPT, any satisfied set is not going to have this property. Yeah, good. So I want to assume x- and y-coordinates are only distinct in the input. OPT will not have that property. And that's why p and q can exist and have the same y-coordinate. Another question?

AUDIENCE: Does this still [INAUDIBLE] the special case where your two points are the points of the rectangle?

ERIK DEMAINE: OK. So the question is can p and q be the points of the rectangle? One of them can be. Like, p could be here, and then another point is over here. So then that will be the segment that you are using, between p and q. Or it could be q is here, and p is over here. Then that's the segment. You can't have them both equal because p and q have to be horizontally aligned and also because there's got to be another point in there.

Yeah, so that should work. You have to check that this boundary case is still OK. But the claim is no other rectangle touches this red line even on the endpoint. And therefore, no other rectangle will wholly contain p and q. And so that means you're only charging to this pair once. And then this pair charging is OK because, luckily, there's three edges here, four vertices. One of those vertices we can't charge to, so there's exactly the right number of things for the edges, and we're OK.

Yeah, this can really happen. In fact our favorite example of the pinwheel-- if instead of doing the greedy addition, we do this addition-- these are supposed to be horizontally aligned. A little hard without a grid-- a graph blackboard would a great. So this is not quite satisfied. You've got to add some more points here or something.

But it has the feature that-- here's an independent set of rectangles. I can do this one, this one, and this one. So this is three independent rectangles. As the white points go, they're independent rectangles. The corners are not strictly inside each other, and none of the white points satisfies any of the other rectangles.

And indeed, if you applied this argument, first you take the widest rectangle and say, OK, here is my vertical red segment. I'm going to charge to these two guys, this segment, and then eventually this guy will charge to this segment and this guy will charge to this segment. And luckily, there are three added points for exactly the three segments for the three rectangles. There had to be another point. So that's a lower bound. A lot of work-- but in the end, it says,

look, just find an independent set of plus boxes, plus rectangles. That's a lower bound on OPT.

So now, the question remains, how do we find a good independent set of plus boxes? And now we'll go through the three different ways we know how to do it.

I'll start actually with Wilber 2. It's called Wilber 2 because it was in a paper by Wilber, and I think he called it lower bound number 1 and lower bound number 2. But for pragmatic reasons, I'm going to start with number 2. It's from 1989, so it's actually an old paper. And it was sort of lost for a long time. I don't think Wilber wrote any other papers. It was in *SICOMP,* a big journal. so a few years after splay trees and then sort of rediscovered in the early 2000s and turns out to be really useful for a lot of theorems.

So here's the lower bound. Again, we're looking at the input point set-- no added points, just the original points. Look at every point, and look at all the points that you can see from this point downward. What does see mean? I'm interested in points below p that when I draw the rectangle contain no other points.

So this is sort of like a lower envelope. It's going to look something like this-- and maybe some points like this. So all of these rectangles have to be empty.

So these are the downward visible points from p. And now, among these points, you can sort them by y-coordinate. And I want to see how many times do they cross this vertical line. So if I order them by y-coordinate-- so I start here, and maybe I go to here. Then the next one is over here, so that's across. Then I go over here. Then I cross. Then I go here. Then I cross. Go here, here, cross. So if I visit them in order, I want to know how many times do I cross this vertical line.

So this is the past of p, all of the accesses before p. Think of this is how many times you alternate between accessing on the left of the line and accessing on the right of the line. So count number of alternations left or right of p. And again, if we assume that no key is ever accessed more than once, then there will always be left or right, never exactly on.

And then I want to sum over all points. And I claim this is a lower bound. Why is it a lower bound? Essentially, I take each of these red lines that cross the p vertical line and I turn them into a box. So there's one there, one there, one there, and one there. I claim if I do this for all p, all those boxes will be independent. All those rectangles will be independent from each

other. I won't prove that formally here, but you can check it.

So it's obvious for one p because each of these has a different vertical span. If you do it for all p-- all the points p-- these won't conflict. So by the independent rectangle lower bound, this is a lower bound on OPT up to a factor of 2.

So what? Wilber 2 is quite interesting. For a long time, we've conjectured that it is the right answer. So conjecture-- I know it's a weird lower bound to even think of. It's a very hard paper to read. Without the geometric view, it's even harder to imagine the definition of this bound. It's sort of an algorithm. It's a way to assign boxes. It gives you a lower bound. It's a little weird. We conjecture that it's proportional to the optimal solution. We can't prove it. We've tried many times. It's a pain to work with, but it is what it is.

There's one theorem that uses it, so I want to tell you about that theorem. But I don't want to go into it in too much detail. It's a neat theorem. And it's in a paper by Iacono, 2002. And it was the first paper to revitalize the Wilber stuff. So it's like, hey, there's this Wilber 2 bound. We can use it to solve a new problem, which is called key independent optimality.

Briefly, the idea with key independent optimality is, suppose you've heard about dynamic optimality. You know, it's really cool because splay trees and whatnot seem to really adapt to whatever your inputs are. But suppose your inputs really don't have keys. They're just arbitrary objects labeled however, just randomly. In fact, let's assume that they're labeled randomly. Suppose the keys are generated randomly because they're meaningless or just arbitrary things. So you figure, oh, maybe I'll make it better and just randomize them completely.

If keys are random, then dynamic OPT is the same thing up to constant factors as the working set bound. That's the theorem. So this is cool because it means splay trees are actually optimal in the setting where keys are random. This is in expectation over the randomized keys.

And the way this theorem is proved is basically-- so what this is saying is, if we take a point set-- arbitrary point set-- but then we re-randomize the x-coordinates-- leave the y-coordinates as they are-- then you can compute how Wilber 2 behaves. Because now you have a bunch of points, and you're randomly shifting their x-coordinate. So it's like if you're randomly bouncing around an x and you're interested in this envelope on the left and the right, you want to know basically how many times-- I guess since I last accessed p, which is here. We didn't do that here, but in the working set bound that's part of the deal. If you look on the left side, it's like how many times does the max change.

And you may know if you n random numbers and you want to know how many times does the max changes as I go left to right, as I take larger and larger prefixes of those n numbers, the answer is log n in expectation. Because the more points you have, the less and less likely it is for the max to change.

So basically, you show the expected Wilber 2 of a point over this randomization is theta log ti, where ti is the working set bound. And so, that gives you the theorem. This gives you a lower bound of the working set bound. We have upper bounds of the working set bound, and therefore that's OPT. So that's just a very quick sketch. If you're interested, check out the paper.

That's unfortunately all we know what to do with Wilber 2. But there's this other bound, Wilber 1, which seems less good yet we can do a lot more with it. So let me go to that.

It's a lot easier to analyze algorithms with respect to Wilber 1. What's Wilber 1? We're going to fix something called a lower bound tree. I'm going to call it because it's basically going to be a perfect binary tree on my keys. This tree never changes. That's why I say fix. It is not the binary search tree you're looking for. It is not the binary search tree that you're interested in. It's just a thing to think about.

Now, for each node of that tree-- let's look at this node, I'll give the node a name, y. So here's y. There's the left subtree of y, and there's the right subtree of y. These are a bunch of keys. There's keys that are to the left of y. There's keys to the right of y. There's keys outside the subtree. We're going to ignore those. I want to look at the accesses to these keys and accesses to these keys and see how many times do I switch between left and right.

So count the number of alternations-- so very similar in spirit to Wilber 2, it's just relative to this weird tree, which is kind of arbitrary-- in the access sequence-- which is x1 up to xn-- between left and right subtrees of y So we're going to ignore accesses to y itself. We're going to ignore accesses to keys outside of y. Just look at how many times do I switch between x and y. That's a lower bound. That's the claim.

It's a lower bound for the same reason we use the independent rectangle lower bound. And the claim is, if you look at these alternations, draw the corresponding rectangles-- so over here, we had a vertical line which corresponded to the key, and we see how many times do we cross the line.

Basically, the same thing over here except now there's one big vertical line that corresponds to the root node, then there's some vertical lines that correspond to this node and this node, and you're interested in the access sequence. How many times-- let's do some kind of access sequence like this-- these are our points-- and you just look at what lines are you crossing. So like this crosses the big line. So that's going to be one alternation between left and right here. Here's another alternation between left and right. Here is another alternation between left and right. Here's another alternation between left and right. And one more.

So for the big line, for the root node, that's how many times you cross between left and right relative to the root. Then, for the left subtree the root, there's one crossing here. There is one crossing here, one crossing here. These are touching, but they're not satisfied. So it's OK. The claim is all these rectangles will be independent. Again, I won't prove that formally, but it's true. OK? Rough sketch.

So that's Wilber 1. It's, again, an independent rectangle lower bound. It's a little weird because it depends on this tree. You could choose it to be a nice perfect tree. You could choose it to be a different tree. You'll get a different lower bound each time. So of course, you take the max over all trees. That will give you the biggest Wilber 1 lower bound. We don't know much about that biggest Wilber 1 lower bound.

I guess you could ask the following open question. Is it true that for every access sequence there exists a tree p such that Wilber 1 is theta OPT? Or is theta Wilber 2 or something? Wilber 2 is a single quantity. You compute it. It gives you a bound. Wilber 1, it depends on this p. Maybe if you choose the best p for your sequence you get the right answer. But it's definitely the case that Wilber 1 for a fixed p is not the right answer.

I recall that's easy to prove. Well, maybe we'll come back to that. Yeah, question?

AUDIENCE: So how do you construct this lower bound tree? Like, is it just--

ERIK DEMAINE: I'll tell you what we're going to use-- the question is how do we construct p. You can make it whatever you want. What we're going to use is the perfect tree, which is sort of unique. It's kind of arbitrary, but it works. It has the property that its height is log n. That's all we need. We're going to use that to get tango trees. Other questions?

All right. Let me briefly mention a fun access sequence. You may recognize this sequence. This would be in-order traversal in binary. But if I take these bit sequences and read them

backwards, then I get 0, 4, 2, 6, 1, 5, 3, 7. This is the number 0 through 7 in a funny order. It's called the bit reversal sequence. If you access 0, 4, 2, 6, 1, 5, 3, 7 in a perfect binary tree, it maximizes Wilber 1.

So in-order traversal-- 0, 1, 2, 3, 4, 5, 6. Ignore 7. There's not 7 in this tree. I do 0, 4-- if you look at the root, alternate left, right, left, right, left, right. Because the high-order bit is switching every time, and so whether I go to the left of the tree here or the right of the tree, it's switching every time. And also, if you look in any subtree, like when I'm accessing things within the subtree of one, it alternates 0, too. It's too small a tree to really see that happening, but it's true.

And so, if you do this for k bits, n equals 2 to the k roughly. And Wilber 1, the lower bound, is log n per [INAUDIBLE] because the every access alternates. So if you look at a subtree, whatever the size of that subtree is, that's how many alternations there are. And so, number of alternations is theta n log n because it's the sum over all nodes of their subtree sizes. And so OPT is theta n log n.

We know we can achieve n log n-- this is to do n accesses-- we know we can n log n with a red-black tree or whatever, but there's actually a lower bound of n log n, meaning all binary search trees-- if you're given this access sequence, doesn't matter what you're doing-- you have to pay n log n. It's kind of cool.

A little side effect-- that's Wilber's paper ended. It's like, hey, cool, can find one access sequence that is bad for everybody. But now we're going to use Wilber 1 to get one binary search tree that's pretty good for all access sequences. Pretty good meaning within a log log n factor of optimal.

And this is tango trees, which would be log log n competitive online binary search trees. Why are they called tango trees? People made up all sorts of reasons, but I can tell you-- because I was there-- they were invented mostly on a flight from New York to Buenos Aires, which is the center of tango. I bought this T-shirt I think the day after. And then that week, we wrote the paper, and that was tango trees. So no particular reason, but it sounds good. Always good to have a cool name. So the secret is revealed. The true meaning of tango trees is nothing, but you we'll see.

So how do they work? It's very simple. Basically, we take Wilber 1 and we simulate it. So let

me be more precise. There's one key idea, which is to look at the preferred child of a node. I'm going to say the preferred child is left. Let's see, node y in p. It's left if we accessed some node in the left subtree of y most recently. It's the right child if we accessed something in the right subtree most recently. So we're just looking at left and right subtree accesses, what was most recent?

There is a special case in the beginning, which is you don't have a preferred child because you haven't accessed either left or right yet. So this is if no access to the left or right yet. So that just happens in the beginning. Once you've touched everything, everybody will have a left or right preferred child. So this is just what was your most recent child. This is like a parent with a very short memory. Just whichever child I most recently talked to, that is my preferred child at the moment. It's kind of like I don't know when you're going to job interviews. You know, the most recent interview is the one you remember most fondly and so, ah, you like that one the best independent of which is the coolest.

So let me draw a picture. And I guess I'm going to draw a big picture-- my favorite-- a perfectly balanced binary search tree with eight leaves. And so now, suppose that every node has a preferred child. Let's say they all do just because it makes a more interesting picture. I'm going to draw that with a big fat arrow.

And now, what does that do? It decomposes our tree. This is the perfect tree. p is going to be perfectly balanced, log n height. It could be any log n height tree, but we'll make it perfect. And it decomposes that tree into paths. And there's a path here. You just keep following parent pointers, you get a path-- not parent pointers, preferred pointers. It's also true if you follow parent pointers you get a path, but they'll overlap each other. You follow preferred child pointers, you get non-overlapping paths.

There they are. We also get these singleton paths at the leaves. Some of the leaves are in singleton paths. These are called preferred paths. Why do I care? So this tells me the most recently accessed element was somebody on this path. I don't quite know who. It could have been this one, and that would say, OK, this is the most recent direction we went through all of them. Let's say it's that one.

Now suppose I access this node. What does that tell me? Well, if I most recently accessed left here and now I'm accessing the right, if you look at this node, the Wilber 1 bound goes up by 1. Because I just accessed left. Now I accessed right. Also, if I access this node, this guy, his

Wilber 1 bound goes up by 1 because now he's going to the right, whereas last time he went to the left. Also, this node previously went to the right and went to the left. So Wilber 1 went up because of this edge, and it went up because of this edge. In general, following non-preferred edges, I can pay for because Wilber 1 goes up by 1 every time I use a non-preferred edge. This is another way to state the Wilber 1 bound. This is the cool thing.

As long as I can go through a path quickly-- ideally, if I could do it in constant time, this would be a dynamically-optimal binary search tree. If I could instantly transport to where I need to go on a path and then jump off the path to the next path, that I can pay for-- I can spend constant time to do that-- then I'd be OK.

I'm not going to be able to do it in constant time, but I'm going to be able to do it log log n time. I'm going to be able to jump through a path in log log n time, and then jump-- figure out where I need to diverge from the path because maybe I'm accessing this guy. Jump to the next path. Do that in log log n time. I've got to update the path structure because now the preferred child is to the right. It used to be to the left. So I've got to do something that will only cost log log n time. If I can do that, the lower bound is the number of edges. The upper bound is the number of non-preferred edges times log log n.

So we get a lower bound Wilber 1, which is going to be equal to the number of non-preferred edges. And we're going to get an upper bound through tango trees, which is going to be order number of non-preferred edges times log log n.

OK. Why is it log log n? Because each of these paths has length only log n. So put them in a balanced binary search tree, and it has height log log n. So take these paths, squish them into a tree-- it's hard, I don't know which way you're squishing. It says log n depth. It's a path. I'm going to fold it into a tree. So it has height only log log n. Then I can jump around it in log log n time. That's the idea with tango trees. You're basically done.

A few details in how they work. I don't want to spend too much time on them, but let's go through some of them. So we're going to store each preferred path as an auxiliary tree, which is just-- I don't know-- a red-black tree, say.

What is the red-black tree sorted by? I don't have a choice. Whatever I do has to be a binary search tree among the original keys. So if I take these items and I just throw them into a red-black tree, they will be sorted by whatever their x-coordinate is. So this is the max, this is the min. This is somewhere in between. This is to the left of that. So the order is a little weird. I'd

really like to store them sorted by depth, but I can't do that. They are sorted by their key values.

Now, what do I need to do with these auxiliary trees? I mean, the basic thing I do is a search, right? I'm searching for a key. It's a binary search tree, so I can still do a search. I can figure out this tree gets represented as something more like this. That would be a nicely balanced version of these four nodes. So if I called them, I don't know, a, b, c, d. That's their sorted order. It's going to be a, b, c, d. That's also their sorted order over here.

So if I search for my key, I'll figure out, oh, do I fall off here, here, here, here, or here? Now, each of those corresponds to another path I need to visit. So if I fall off the left side of a, then I should have a pointer to this structure. If I fall off the-- I guess these two are empty. Those correspond to these two places. If I fall off here, the right side of c, which is now here, this is going to be a pointer to my new structure which corresponds to this one. And then this one is going to correspond to all this stuff-- well, in particular this one. It's a little hard to draw this picture, but you get the idea. You just rebalance each of these things. Keep that the pointers between the preferred paths just as they were. This is uniquely defined how to do this because it's a binary search tree.

So leaves point to other-- let's call them child auxiliary trees. It uniquely defines which ones they have to point to in order to still navigate the whole structure. So it's a weird way of rebalancing your tree. And the point is each of these red-black trees has height log log n because the number of nodes in it is only log n. And that gives us the bound.

Now, key thing to think about is what happens when you change-- I said I have to be able to achieve number of non-preferred edges times log log n. So fine. I do a log log n search in here. Maybe I decide I have to go off here. Then I do a log log n search in here. And then maybe I have to go this way. So number of non-preferred edges was 2. I did two, maybe three searches. Fine. It's going to be number of non-preferred edges plus 1 time log log n. No big deal.

Now I have to update. Now this is the preferred edge from the root, and this is the preferred edge from this node. How do I update preferred edges? That's something to think about. So I've got a path represented by a red-black tree. And now I fall off here, and there's another path here. I need to convert this into a path that goes like this and then does this. And separately, a path that does this. That's the new version.

How do I do that? Conceptually, it's pretty simple. I want to cut the path here and then rejoin along there, like that. So conceptually, if things were stored by depth, this is what we'd call a split and a concatenate. You should know this from regular binary search trees. This is a standard exercise for red-black trees. Given a query, x, you can cut this tree into two halves and get two red-black trees, which represent everything to the left of x and everything to the right of x.

Similarly, given two trees that are sorted like this where all the elements are less than all the elements over here, I can concatenate them into one red-black tree. And all of these take log n time, where n is the number of nodes. Here, that would be log log n time.

In this world, it's not quite so simple because things are not sorted by depth. They're sorted by key value. But it's not so bad. Because, if you look at some path and you want to say, OK, I want everything that's below this key value or something, then that's the same as saying, well, take everything that is within this interval of keys. So it's strictly between here and here. Let me redraw this slightly.

So if you look at the nodes of depth greater than d, I want to cut off everybody that's deeper than a particular spot in order to do this kind of change. These are equal to nodes in subtree of that. So let me give it a name. Let's say I want to cut here. So I'm going to look at this node y. This is nodes in the subtree of y. All of the nodes that are below y are obviously going to have smaller depth than that path. This is nodes in a path. And nodes in a subtree are equal to nodes with keys in the min of that subtree to the max of that tree. It's an interval.

So what do I do? I split at min of y. I split at max of y. That gives me the interval. So here's the picture. I have a tree. I want to cut out this interval of nodes. This is like range queries kind of in 1D. So I split here. I split here. What I will have are the things I care about, the things to the left of it and the things to the right of it. What I wanted was this and everything else. How do I do that? I concatenate-- this is y. This is in the interval min of y to max of y.

So I wanted those guys. Those are the nodes that are deeper than d. I also want the nodes all together that are less deep than d. That's these nodes and these nodes. So I concatenate these together, get one big tree that represents things with depth less than d. These are the things of depth greater than d. OK?

So I do two splits, one concatenate, and that simulates this kind of cut operation. Similarly, if I

want to do a joint operation, it's a constant number of splits and concatenates, and I'm done. Just dealing with the fact that things are in the wrong order here, but it's not so bad.

One more thing, which is-- I basically described the overall structure as a tree of auxiliary trees. In reality, we're in the binary search tree model. We can only have one tree. Not so hard, though. I mean, basically, you want one tree that represents lots of trees that are kind of pasted together.

So to do that, you just mark each node that transitions from one tree to the next. So each node will say, I am the root of a new auxiliary tree or just say, no, I'm part of the same auxiliary tree as my parent. And then you have to define these kinds of split and concatenate operations in this setting where you have a tree embedded inside a tree. But you just ignore all the nodes that are claimed to be part of another tree. Just pretend they weren't there, and it works. So a little hand-wavy there, but it's kind of a tedious detail. You can stick all these trees into one tree just by marking these roots.

And that's tango trees. I already spoiled the climax, which is this log log n thing, but it's pretty obvious how to get there. It's just a lot of details to actually do it. We're just taking the Wilber 1 bound, recasting it in terms of this preferred path thing where it's just the non-preferred edges. Or the non-preferred edges are what Wilber 1 counts, and so we can afford to spend log log n time for each of them. And the paths themselves only have log n nodes, so you can search through them in log log n time pretty easy.

This also shows you why Wilber 1 is not a good bound with a fixed tree. Because here are log n nodes. I can just sit there all day bouncing around all of them in random order. I'm definitely going to need log log n time to access them, but Wilber 1 is not changing at all. So Wilber 1 stays constant, like 0. I had to warm it up, but after I test everything, I can just sit there and bounce around these guys randomly. I've got to spend log log n time to do that, but Wilber 1 doesn't justify it for me. Wilber 2 will go up, but Wilber 1 with this tree? It's kind of lame.

So this is the best tango trees could hope to do using Wilber 1. I would guess that tango trees are a log log factor away from optimal, though we don't know that for sure. But greedy we're still pretty sure is good. It should be a constant factor away from optimal. So I want to talk a little bit about that.

There's one thing on this outline we haven't talked about. We did independent rectangles. We did Wilber 1 and 2. We saw applications of them in particular tango trees. One thing we

haven't done is Signed Greedy. So let's do Signed Greedy.

Still left here is we have two ways to choose rectangles, independent rectangles. They're different. It would be kind of nice to know what the best way to choose rectangles is. And we actually know that-- Signed Greedy. So there's two kinds of Signed Greedy. There's the plus sign greedy, and there's the minus sign greedy.

How does plus greedy work? It's the same as greedy, you just only look at plus rectangles. Remember plus rectangles and minus rectangles. So let's look at our favorite example here. With greedy, I would sweep up, and every rectangle that was unsatisfied, I would satisfy it. Now I'm going to ignore minus rectangles, only look at plus rectangles. So I see this rectangle, and I say, oh, I don't care because that's a minus rectangle. Then I see this one and this one. I say, oh, those are plus rectangles. So I'm going to add a point here. I'm going to add a point here.

Then I go up to here. I see this rectangle, which is a plus rectangle. That's bad. So I've got to add a point. I see this minus rectangle I don't care about. This is plus greedy. It does not satisfy the set. This rectangle never got satisfied. But it plus satisfies the set. If I do plus greedy, it will be plus satisfied. Every rectangle you draw here, if it's plus rectangle, it's got another point in it.

What's kind of nice, also, is if you actually draw the rectangles you are satisfying-- maybe I'm use another color. There was one rectangle here. There was one rectangle here. And there was one rectangle here. That's a little awkward because they're not on the original points. So I can change them a little bit, maybe move this one down to here and move this one over to here. You could say that those rectangles came from those points. Then this is a set of independent rectangles on the original points.

Maybe not totally obvious, but plus greedy always gives an independent set of plus rectangles. So it's a lower bound. It's not an upper bound because it's not satisfying the point set, but it's a lower bound. I claim it's a very good lower bound.

It by itself might not be great, but you have to consider both of them. So theorem is if I take the max of plus greedy and minus greedy-- each of them is lower bound, so the max is a lower bound on optimal-- then this is within a constant factor of the biggest possible independent rectangle lower bound.

And so this is the way you should choose independent rectangles. Run plus greedy. Run minus greedy. Take the best of the two. That will always be within a constant factor of the best independent set of rectangles, factors like 4 or something in the worst case.

So let me prove this to you. It's a kind of a weird argument. I'm going to define a new quantity. Let's call this OPT x, I guess. It's sort of like if you consider plus rectangles separately from minus rectangles, which is what we're doing. So I would like a point set-- first, I'd like a plus satisfying point set, and then I'd also like a minus satisfying point set. And then I take their union. And I say the cost of that pair of plus satisfying and minus satisfying is the size of the union. So I get a bonus point if they happen to overlap. Not a big deal, just a factor of 2. So this is not a core concept, but it turns out to be basically what we were doing over here.

Let me give you a sequence of crazy inequalities. First one is that this OPT thing is greater than or equal to size of the input. Each of these inequalities is totally obvious, but the conclusion is kind of crazy.

The independent rectangle lower bound, which we proved, says that if you look at plus satisfying things that's going to be at least size of the input plus the max number of independent rectangles. If you look at the minus satisfying things, that's also going to be at least size of the input plus maximum number of minus independent rectangles. So we already proved this. That, if you look at this union, it's going to be at least the size of the input plus half the overall max. So that's what we proved at the beginning a lecture.

Now, this is the best way to use independent rectangles. This kind of Signed Greedy, which is the max of the two signs, is a way to find independent rectangles. So it's only going to be worse. It's going to be smaller. So we can say is greater than or equal to half the max of plus greedy and minus greedy. This was the max. So this is another way to do it, so it must be smaller.

Now, greedy computes a plus satisfying assignment. So I could say, well, if you looked at the optimal plus satisfying assignment-- this is something we defined at the beginning of lecture-- and the optimal minus satisfying assignment, that's only going to be smaller than greedy because greedy is an algorithm for solving OPT plus. It can't be better than the optimum. Greedy again has to be bigger than the optimum.

Now I just want to turn this max into a plus because the max is always at least the average. So if I take the average, which turns it into 1/4 OPT plus plus OPT minus. Then that holds. You

turn the max into a plus. If I look at the optimal plus satisfying plus the optimal minus satisfying, that's only going to be bigger than this thing because this can only save like a factor of 2 or whatever over just adding them up. I don't even need to factor of 2 thing. I just need that if you add them up, that's only going to be worse than just counting them as the union.

So we get what I call a sandwich. On the one side, we have OPT x. On the other side, we have 1/4 OPT x. I really don't care about OPT x personally. I mean, it's kind of interesting to see that it's here. But the point is these are within a constant factor. Therefore, all of these things in between are within a constant factor of each other.

So in particular, this thing, max of the two greedys, is within a constant factor of this thing. This is the independent rectangle lower bound, the best one. It also tells you that OPT x is basically what we're computing here.

So this is weird. I'm going to draw one more picture, which is greedy versus Signed Greedy. Remember greedy from last lecture. Greedy says, look, I'm going to fix plus rectangles, and I'm going to fix minus rectangles. It does them both at the same time. Signed Greedy says, look, I'm going to do the plus rectangles separately, and then I'm going to the minus rectangles separately, and then add them up or take the union or take the max. It doesn't matter. It's a constant factor. Just add them separately.

This one is an upper bound. It is a binary search tree. This thing is a lower bound. All binary search trees must take at least this. Are they equal up to constant factors? We don't know. That's the big question. They look almost identical. But what greedy has to deal with is sort of the interrelations. When I fix some plus rectangles, I might get new minus rectangles that I have to fix with greedy.

Signed Greedy doesn't have to deal with that. It's just the plus rectangles. They might make more plus rectangles, but that's all I have to deal with. It doesn't deal with the interaction between plus and minus rectangles. Seems like the interaction kind of fades away as a geometric series. And therefore, these things are the same up to constant factors. But we have no way to prove that. It could be the interaction blows you out of the water somehow.

That's the best we know for dynamic optimality. Maybe next time I teach this class we'll have a final answer and it'll be constant, but that's where we are today.