ERIK DEMAINE: All right, let's get started. So today, we start out geometry, geometric data structures. There are two lectures on this. This is lecture one.

And we're going to solve two main problems today. One is point location, which is finding yourself on a map. And the other is orthogonal range searching, which is catching a bunch of dots with a rectangular net.

And they're fun problems. And they're good illustrations of a couple of techniques. We're going to cover two general techniques for data structure building. One is dynamizing static data structures, turning static into dynamic using a technique called weight balance, which is really cool. And another one is called fractional cascading, which has probably one of the coolest names of any algorithmic or data structures technique. It's actually a very simple idea, but sounds very scary.

And with point location, we're going to see some fun connections to persistence and retroactivity, which was the topic of the last two lectures, you may recall. And so we'll start out with that. Planar point location, you can do it in higher dimensions as well.

In general, geometric data structures are about going to more than one dimension. Most data structures are about one dimensional ordered data. Now, we have points in the plane.

We might have polygons in the plane. So this is what we call a planar map, got a bunch of line segments and points forming a graph structure. So think of it as a planar graph drawn in the plane where every edge is a straight line segment. And none of the edges cross, let's say.

So this is a planar map. It's also called a planar straight line graph. And the static version of this problem-- so there's two versions, one is static-- you want to preprocess the map.

So I give you a single map up front. And then I want to support dynamic queries, which are which face contains a point p. So that point is going to be given to you as coordinates x and y.

So maybe I mark a point like this one. I give you those x and y coordinates. I want to quickly determine that this face is the one that contains it. I give you another point over here. It quickly determines this face.

This has a lot of applications. If you're writing a GUI and someone clicks on the screen, you need to map the coordinates that the mouse gives you to which GUI element you're clicking on. If you have a GPS device and it has a map, so it's preprocessed the map all at once. And now, given two GPS coordinates, latitude, longitude, it needs to know which city you're in, which part of the map you're in, so that it knows what to display, that sort of thing.

These are all planar point location problems. It comes up in simulation, lots of things. It's actually one of the first problems I got interested in algorithms way back in my oceanography days. So that's planar point location. That's the static version.

The dynamic version-- make things harder-- is the map is dynamic. So here, the map is static. The queries are still coming online. Dynamic version, you can insert and delete edges in your map.

And let's say if you get a vertex down to degree 0, you can delete the vertex as well, add new degrees 0 vertices. As long as you don't have crossings introduced by inserting edges, you can change things. So that's obviously harder.

And we can solve this problem using persistence and using retroactivity in a pretty simple way using a technique which you may have seen before, pretty classic technique in computational geometry. So this is a technique that comes from the algorithms world. And we're going to apply it to the data structures world.

So, sweep line technique, it's a very simple idea. So you have some line segments in the plane, something like this. And I'm going to take a vertical line. So the algorithmic problem is I want to know are there any crossings.

Do any of these segments cross? This is where sweep line technique comes from, I believe. So the idea is we want to linearize or one-dimensionalify the problem.

So just take a slice of the problem with a vertical line. And imagine sweeping that line from left to right. So you imagine it moving continuously. Of course, in reality, it moves discretely. Let me unambiguate this a little bit.

OK. There are discrete moments in time when what is hit by the sweep line changes. Let me maybe label these segments. We've got a, b, c, and d.

So initially, we hit nothing. Then we hit a, then we hit b. Why do we hit b? Because we saw the

left end point of b. Then we see the right endpoint of a which means we no longer-- sorry, at this point, we see both a and b in that order.

Then we lose a, so we're down to b. Then we see c. c is above b. Then we see d. d is above c and b.

Then c and d cross. So c and d change positions. And then we have b. Then we lose b, then we lose c. Then we lose d.

This is a classic algorithm for detecting these intersections. I don't want to get into details how you do this. But you're trying to look for when things change in order in these cross-sections. The way you do that is you maintain the cross-section in a binary search tree, so you maintain the order.

If you hit a left endpoint, you insert into the binary search tree. If you see a right endpoint, you delete from the binary search tree. And you do some stuff to check for crossings. In this problem, there are no crossings. So we don't need to worry about that.

But we're taking this technique. Say, OK, there's a data structure here, which is the binary search tree maintaining the cross-section. OK. So, typically, the cross-section data structure is regular balanced binary search tree.

Our idea is what if we add persistence to that binary search tree? So instead of using a regular binary search tree, we use a partially persistent balanced binary search tree, which we know how to do. This is a bounded n degree structure. We can make it partially persistent, constant overhead.

So if we add partial persistence, what does that let us do? Well, let's just look at a moment in the past, partial persistence about querying the past. So there's a sequence of insertions and deletions that occur from the sweep line. But now, if we can query in the past, that's like going to a desired x-coordinate and saying, what does my data structure look like at this moment?

OK. Now, the data structure, let's maybe look at this one, because it's got three elements, very exciting. So you've got d, then c, then b. So you've got a little data structure that looks something like this.

It understands the order of the cross-section of those segments. And so, for example, if I was given a query point like this one, I could figure out what is the segment above me, what is the

segment below me. That is a successor query and a predecessor query in that binary search tree.

This notation maybe-- a query at time t of, let's say, successor of y is what we call an upward ray shooting query from coordinates t,y. So t, the time, is acting as x-coordinate. Time is left to right here.

And so what's happening is we're imagining, from this point, shooting a ray upward and asking what is the segment that I hit first. That's an upward ray shooting query. And this is from a problem called vertical ray shooting, which is more or less equivalent to planar point location.

So vertical ray shooting, again, you're given a map, planar map. And the queries are like this. What is the first segment that you hit with an upward ray?

So I give you a point, x,y. And I ask, if I go up from there, what's the next edge that I get? That's the vertical ray shooting problem. And we just solved the vertical ray shooting problem for static.

If you're given a static map, you run this algorithm once, assume there are no crossings. Then to answer vertical ray shooting query, we just go back in time to time t, do the successor query, which takes log n time, and then we get the answer to this. So we can do this in log n per query static.

This is all two dimensional. I should probably say that. You can generalize. Questions? This is actually really easy. This is the stuff we get for free out of persistence and, at the moment, retroactivity.

I believe this is one of the reasons persistence was invented in the first place. There were a bunch of early persistent data structures. Then there was a general Driscoll paper, which I talked about.

But I think geometry was one of the main motivations. Because it lets you add a dimension. As long as that dimension is time-like, then you get the dimension sort of for free. So that's nice.

OK. What about retroactivity? Again, we're going to use partial retroactivity. And I can tell you for certainty, because I was there, this is why retroactivity was invented.

So retroactivity, so that would mean that we get to dynamically add and delete insertions and

deletions. So that's like adding and deleting segments from the structure. Again, we have a linear timeline.

We always want to maintain a linear timeline, because that is reality. That corresponds to the x-coordinate. And now I want to be able to add a segment like this, which means there was an insertion at this time, a deletion at this time.

Now, this doesn't quite work. Because this point in cross-section, it's actually moving over time. Binary search trees, that's OK, because things are simple. But at the moment, all we know how to do is actually horizontal segments, which are inserted and deleted at the same y-coordinate.

So then we can do insert at time t1, an insertion of some y-coordinate, and then an insert at some later time, the deletion of that y-coordinate. So this is a partially retroactive successor problem. This is equal to dynamic vertical ray shooting.

I guess this insertion corresponds to the insertion of a thing. If you instead do delete here, then you're deleting one of the segments. This is among horizontal segments. So if your map is made of horizontal and vertical segments-- so it's an orthogonal map-- then you can solve the dynamic problem using a partially retroactive successor.

Again, we want to do successor just like before, querying the past. But now, our updates are different. Now, we have retroactive updates. That lets us dynamically change the past, which is like inserting and deleting edges through that algorithm.

But at the moment, we only know how to do this for horizontal segments. So this gives us, if you remember, the retroactive successor result. We haven't seen that, how it works.

It's complicated. But it achieves log n insert, delete successor retroactively. And so we get a log n, which is an optimal solution for dynamic vertical ray shooting among horizontal segments.

There are a bunch of open problems. What about general maps? So for a dynamic vertical ray shooting in general maps, if you want log n query, the best results are log to the 1 plus epsilon insert log to the 2 plus epsilon delete.

There's some other trade-offs. You can get log times log log n query and reduce. Still, we don't know how to delete faster than log squared in any of the general solutions. So you can

do log square for everything.

But the hope would be you could do log for everything even when the segments are not horizontal. But here, retroactivity doesn't seem to buy you things. It'd be nice if you could.

Another fun open problem is, what about non-vertical rays, general rays, non-vertical rays? So I give you a point and I give you a vector, I want to know what do I hit in that direction. This is a lot harder. You can't use any of these tricks.

And in fact, it's believed you cannot get polylog performance unless you have a ton of space. So the best known result is-- I'll just throw this up here. You can get n over square root s polylog and query. If you use, basically, s space.

So you need quite a bit of space. Because if you use n to the 1 plus epsilon space, you can get roughly root n query time. If you use n to 5 space, then you get somewhat better query time, but still not great. You can maybe get down to n to the epsilon if you have very large polynomial space.

But this is conjectured to be roughly optimal, I assume, other than the polylog factors. The belief is you cannot beat this for general ray. This is kind of annoying, because this is a problem we care about. Especially in 3D, this is ray tracing.

You shoot a ray, what does it hit? You bounce it. You shoot another ray. I always want to know what objects am I hitting.

And for special cases, you can do better. But in general, it seems quite hard. This is even in two dimensions. But there are a bunch of papers on 3D and so on.

I just wanted to give you those connections to persistence and retroactivity. And that's point location. And now, I want to go on two orthogonal range searching.

We can do some new data structures, new to us. So first, what is the problem? So it's sort of the reverse kind of problem here. You're given a bunch of points before the query was a point.

And the query, in this case, is going to be, in two dimensions, a rectangle, a window if you will. And you want to know what points are in the rectangle. So given n points and d dimensions, query in general is going to be a box.

So in 2D, it's an interval crossing interval. In 3D, it's three intervals cross-product together. OK.

So in the static version, you get to preprocess the points. In the dynamic version, the points are being added and deleted.

And in all cases, we have dynamic queries, which are what are the points in the box. Now, there are different versions of this query. There is an existence query, which is are there any points in the box? That's sort of the easiest.

Next level up is, how many points are in the box? Which you can use to solve existence. Next level up is give me all the points in the box, or give me 10 points in the box, give me a point in the box. All of these problems are more or less the same. They do differ in some cases. But the things we'll see today, you can solve them all about as efficiently.

But, of course, if you want to list all the points in the box, it could be everything. And so that could take linear time. So in general, our goal is to get a running time something like log n plus k, where k is the size of the output.

So if you're asking how many points are in there, the size the output is a single number. So k is 1, you should get log n time. If you want to list 100 points in there, k is 100. And so you have to pay that to list them.

If you want to know all of them, well, then k is the number of points that are in there. And we'll be able to achieve these kinds of bounds pretty much all the time, definitely in two dimensions. In D dimensions, it's going to get harder.

OK. So I want to start out with one dimension just to make sure we're on the same page. And in general, we're going to start with a solution called range trees, which were simultaneously invented by a lot of people in the late '70s, Bentley, one of the main guys. And in general, we're going to aim here for a log to the d n plus k query time.

So I like this, but now we have a dependence on dimension. And for 2D, this is not great. It's log squared. And we're going to do better. OK. But let's start with d equals 1. How do you do this? How do I achieve log n plus k query?

Sort the points. Yeah. I could sort the points, then do binary search. So the query now is just an interval. That's the one dimensional version of a box.

So if I search for a, search for b in a sorted list, then all the points in between I can count the different indices-- or subtract the two indices into the array. That will give me how many points

there are in the box, all these things. Arrays aren't going to generalize super nicely. Although, we'll come back to arrays later.

For now, I'd like to think of a binary search tree, balanced binary search tree. And I'm going to make it a little different from the usual kind of binary search tree. I want the data to be in the leaves.

So I want the leaves to be my points. And this will be convenient for higher dimensions. It doesn't really matter for one dimension, but it's kind of nice to think about.

So you've got a binary search tree. And then here is the data sorted by the only coordinate that exists, the x-coordinate. And so, of course, I can search for a, here's a maybe, search for b.

And the stuff in between here, that is my result. And in a little more detail, as you search for a and b, at some point, they will diverge. One will go left, one will go right.

At some point, you reach a. Maybe a isn't actually in the structure. You're searching for everything between a and b inclusive, but a may not be there. So in general, we're going to find the predecessor and successor of a. In this case, I'm interested in the predecessor.

And similarly over here, eventually-- this is all, of course, logarithmic time-- I find the successor of b. Those are the two things I'm interested in. And now, all the leaves in between here, that's the result. Question?

AUDIENCE: So if you have the data just on the leaves, what do you have intermediate node?

ERIK DEMAINE: Ah, right. So in the intermediate nodes, I need to know, let's say, if every subtree knows the min and max, then at a node, I can decide should I go left, should I go right? I think every node can store the max of the left subtree if you just want one key per node. But, yeah, good question. Sorry, I forgot to mention that.

You store a representative sort of in the middle that lets you decide whether to go left or right. So you can still do searches. We can find these two nodes.

And now, the answer is basically all of this stuff. I did not leave myself enough space. That's the left child.

OK. So wherever this left branch went left, the right branches in the answer. Whenever this

right branch went right, the left branch is the answer. But from here, there's no subtree that we care about. Because this is all greater than what we care about.

OK. But the good news is there's only log n of these subtrees, maybe two log n. Because there's the left side, the right side. OK. So the answer is implicitly represented. We don't have to explicitly touch all these items. We just know that they live in the subtrees, in those order log n subtrees.

So in particular, if every node stores the size of its subtree, then we can add up these log n numbers. And we get the size of the answer. If we want the first k items, we can visit the first k items here in order k time. So in log n time, we get a nice representation of the answers, log n subtrees.

Of course, we also had a nice answer when we had an array. But this one will be easier to generalize. And that's range trees.

So that was a 1D range tree. The only difference is we put data at the leaves. 2D range tree has a simple idea. We have the data in these subtrees.

These are the matches. Let's think we have an x-coordinate and a y-coordinate. We have an x range and a y range. Let's do this for x.

Now, we have a representation of all the matches in x. So we want this rectangle. But we can get this entire slab in log n time, and we have log n subtrees that we now have to filter in terms of y. There's all these points out here that we don't care about. We want to get rid of those and just focus in on these points.

So we're going to do the same thing on y, but we want to do that for this subtree. And we want to do it for this subtree, and for this subtree, so simple idea. For each subtree, let's call it an x subtree.

So we have one tree which represents all the x data. It looks just like this. And then for each subtree of that x tree, we store, let's say, a pointer to a y tree, which is also a 1D range tree.

So this guy has a pointer to a similarly sized triangle. Except, this one is on y. This one's sorted by x. This one's sorted by y, same points.

This subtree also has one, same data as over here, but now sorted an y instead of x. For

example, there is a smaller tree inside this one. That one also has a pointer to a smaller y tree. Except, now, these are disjoined, because these are completely-- yeah.

This is a subset of that one. But we're going to store a y tree for this one and a y tree for this one. So we're blowing up space. Every element, every point lives in log n y trees.

Because if you look at a point, there's the tiny y tree that contains it bigger, bigger, bigger, bigger until the entire tree also contains it. Each of those has a corresponding y tree. So the overall space will be n log n.

We're repeating points here. But the good news is now I can do search really efficiently, well, log squared efficiently. I spend log time to find these x trees that represent the slabs that I care about. So it's more like this picture.

So there's a bunch of disjoint slabs, which together contain my points in x. And now I want to filter each of them by y. So for each of them, I jump over to y space and do a range query in y space just like what we were doing here. So search for a, search for b, but in y-coordinate.

And then I get log n subtrees in here, log n subtrees in here, log n subtrees in here. So the query gives me log squared y subtrees. It takes me log squared n time to find them. If I have subtree sizes, I compute the number of matches in log squared n time. If I want k items, I can grab k items out of them in order k time.

OK. Pretty easy. Of course, D dimensions is just the same trick. You have x tree. Every subtree links to a y tree. Every y subtree links to a z tree. Every z subtree links to a w tree, and so on.

For D dimensions, you're going to get log to the D query as I claimed before. How much space? Well, every dimension you add adds another log factor of space. So it's going to be n log to the d minus 1 space.

And if you want to do this statically, you can also build the data structure in n log to the d minus 1 n time, except for d equals 1 where you need n log n time to sort. But as long as d is bigger than 1, this is the right bound for higher dimensions. It takes a little bit of effort to actually build the structure in that much time, but it can be done.

OK. That's the very simple data structure. Any questions about that before we make it cooler? You may have seen this data structure before. It's kind of classic. But you can do much better,

well, at least a log factor better.

AUDIENCE: Question.

ERIK DEMAINE: Yeah.

AUDIENCE: So when your storing one pointer for each subtree, you essentially have a pointer for each root, like for each node?

ERIK DEMAINE: Yeah. Right. Every node. So I know these are the nodes that the stuff below them represents my answer in x. And so I teleport over to the y universe from the x universe.

AUDIENCE: So, basically, it has all the same nodes of that subtree, but [INAUDIBLE]

ERIK DEMAINE: Yeah. All the points that are in here also live in here, except these ones are sorted by x. These ones are sorted by y. If I kept following pointers, I get to z and w and so on. Other questions? Yeah.

AUDIENCE: So if we were doing the dynamic case, how would we implement rotations in the [INAUDIBLE]?

ERIK DEMAINE: OK. Dynamic is annoying. Yeah. Rotations are annoying. I think we'll come back to that. We can solve that.

I thought it was easy, but you're right. Rotations are kind of annoying. And we can solve that using this dynamization trick. So we don't have to worry about it till we get there.

It's going to get even harder to make things dynamic. And so then we really need to pull out the black box. Well, it's not a black box. We're going to see how it works. But it's a general transformation that makes things dynamic.

OK. Before we get a dynamic, stick with static. And let's improve things by a log factor. This is an idea called layered range trees. It's also sometimes called fractional cascading, which is the technique we're going to get to later.

I would say it involves one half of fractional cascading. Fractional cascading has two ideas. And the one that it's named after is not this idea. So idea one is basically to reuse searches. The idea is we're searching in this subtree or, I guess, this subtree with respect to y.

We're also searching for the same interval of y in this subtree. Completely different elements, but if there was some way we could reuse the searches for y in all of these log n subtrees, we

could save a log factor. And it turns out we can. And this is one idea in fractional cascading, but there will be another one later.

OK. So, fun stuff. This is where I want to change my notes. So we're searching in x with a regular 1D range tree. I also want to have a regular 1D range tree-- range tree? Sure.

Actually, it doesn't matter too much. I want to have an array of all the items sorted by y-coordinate. And we're going to simplify things here. Instead of pointing to a tree, I'm going to point to an array sorted by y. This is totally static. And this is where dynamic gets harder, not that know how to do it over there yet.

So for each x subtree, we're going to have a pointer to the same elements sorted by y. So all the leaves that are down here are, basically, also there, but by y coordinate instead of x. Obviously, we can still do the same thing we could do before, spend log n time to search in each of these log n arrays corresponding to these log n subtrees. And in log squared n, we'll have our answers.

But we can do better now. I only want to do one binary search in y. And that will be at the root. So the root, there's an array representing everything sorted by y. I search for the lower y-coordinate. I search for the upper y-coordinate, some things.

It's hard to draw this, because it's in the dimensional orthogonal to this one. I guess I should really draw the arrays like this. So this guy has an array. We find the upper and lower bounds for the y-coordinate in the global space. This takes log n time to do two searches. Question.

AUDIENCE: Those are the upper and lower bounds from the predecessor [INAUDIBLE] successor [INAUDIBLE]?

ERIK DEMAINE: Yeah, right. So we're doing a predecessor and successor search, let's say, in this array. Binary search we find-- I didn't give them names, but in the notes they're a1 through b1 and x. And they're a2 through b2 and y. So that's my query, this rectangle. I'm doing the search for a2 and for b2 in the top array.

Now, what I'd like to do is keep that information as I walk down the tree. So that in the end, when I get to these nodes, I know where I am in those arrays in y. So let's think of that just step by step.

So imagine in the x tree, I'm at some node. And then I follow, let's say, a right pointer to the

right child. OK. Now, in y space-- maybe I should switch to red for y space. This guy has a really big array representing all of the nodes down here, but sorted by y-coordinate.

This guy has a corresponding array with some subset of the nodes. Which subset? The ones that are to the right of this x-coordinate. So there's no relation. I mean, some of the guys that are here-- let me circle them-- some of these guys exist over here. They'll be in the same relative order.

So here's those four guys, then one, and two. So some of these guys will be preserved over here. Some of them won't, because their x-coordinate smaller. It's an arbitrary subset. These guys will also live here.

OK. The idea is store pointers from every element over here to, let's say, the successor over here. So store these red arrows. let's say, these guys all point to this node. These guys point to that node. I guess these guys just point to some adjacent node, either the predecessor or the successor.

So the result is if I know where a2 and b2 live in this array, I can figure out where they live in this array. I just follow the pointer. Easy. Done.

OK. Let's think about what this means. So I'm going to store pointers from the y array of some x node. Let's call that node v in the x tree to the corresponding places, corresponding points, let's say, in the y arrays of left child of v and the right child of v.

So, actually, every array item is going to have two pointers, one if you're going right in the x tree, one if you're going the left in the x tree. But we can afford a constant number of pointers per node. This only increases space by a constant factor.

And now, it tells me exactly what I need to know. I start at the root. I do a binary search. That's the slow part. I spend log n time, find those two slots. Every time I go down, I follow the pointer. I know exactly where a2 and b2 live in the next array.

In constant time, as I walk down, I can figure this out. I can remember the information on both sides here. And every time I go to one of these subtrees, I know exactly where I live-- it's no longer a tree-- now, in that array.

So I can identify the regions in these arrays. that correspond to these matching subrectangles with no extra time. So I save that last log factor. If you generalize this to D dimensions, it only

works in the last dimension. You can use this trick in the last dimension and improve from log to the d query to log to the d minus 1.

In the higher dimensions, we just use regular range trees. And when we get down to the two dimensional case, it's a recursion. Before we were stopping at the one dimensional case. We use a regular binary search tree. Now, we stop at the two dimensional case, and we use this fancy thing.

I call this cross-linking. A lot of people call it fractional cascading. Both are valid names. It's a cool idea, but simple once you can see both dimensions at once, which I know it's hard to see in two dimensions. But it can be done. All right. Questions?

I guess the obvious question is dynamic. Now, we're going to go to dynamic. This is a very static thing to be doing. How in the world would we maintain this if the point set is changing?

All these pointers are going to move around. Life seems so hard. But it's not. In fact, updates are a lot easier than you might think.

Some of you may believe this in your heart. Some of you may not. But if you've ever seen an amortization argument that says, basically, when you modify a tree, only a constant number of things happen. And they usually happen near the leaves. I'm thinking of a binary search tree.

The easiest way to see this is in a B-tree if you know B-trees. Usually, if you do insertion, you're going to do maybe one or two splits at the bottom, and that's it. Constant fraction at a time, that's all there is. So it should only take constant time to do an update.

This structure is easy to update at the leaves. If you look at one of these structures, a constant number of items, there's a constant size array. You could update everything in constant time. If we're only up to hitting near the leaves, then life is good.

Occasionally, though, we're going to have to update these giant structures. And then we're going to have to spend giant time. That's OK.

The only thing we need out of this data structure is that it takes the same amount of space and pre-processing time, n log to d minus 1 space, and time to build the static data structure. If we have this, it turns out we can make it dynamic for free. This is the magic of weight balance trees.

In general, there are many kinds of weight balance trees. We're going to look at one called BB alpha trees, which are the oldest and sort of the simplest. Well, you'll see. It's pretty easy to do.

You've already seen height balance trees. AVL trees, for example, you keep the left and the right subtree. You want their height to be within an additive constant of each other, 1. Red black trees are multiplicative factor 2. Left and right subtree, the heights will be roughly the same.

Weight balance trees, weight is the number of nodes in a subtree. Weight balance trees, they want to keep the size of the left subtree and the size of the right subtree to be roughly the same. So here's the definition of BB alpha trees.

For each node v, size of the left subtree of v is at least alpha times the size of v. And size of the right subtree of v is at least alpha times the size of v. Now, size, I didn't define size. It could be the total number of nodes in the subtree. It could be the number of leaves in the subtree. Doesn't really matter.

What else? What's alpha? Alpha is a half, you're in trouble. Because then it has to be perfectly balanced. But just make alpha small, like 1/10 or something. Any constant less than a half will do.

Right. The nice thing about weight balance is they imply height balance. If you have this property that neither your left nor your right subtree are too small, then as you go down, every time you take a left or a right child, you throw away an alpha fraction of your nodes. So initially, you have all the nodes. Every time you go down, you lose an alpha fraction.

How many times can that happen? Log base alpha, basically, so log base 1 over alpha. The height is log base 1 over alpha of n.

So this is really a stronger property than height balance. It implies that your heights are good. So it implies the height of the left and right subtree are not too far from each other.

But it's a lot stronger. It lets you do updates lickety fast, basically. So how do we do an update?

The idea is, normally, you insert a leaf, do a regular BST, insert a delete. You add a leaf at the bottom or delete a leaf. And so you have to update like that node and maybe its parent. As

long as you have weight balance, you're just making little constant sized changes at the bottom. Everything's good.

OK. The trouble is when one of these constraints becomes violated. Then you want to do a rotation or something. OK. So when a node is not weight balanced, it's a pretty loose algorithm.

But it's easy to find nodes. You just store all the weights, all the subtree sizes, which we were doing already. You can detect when nodes are no longer weight balanced.

And then we just want to weight balance it. How do we weight balance it? We rebuild the entire subtree from scratch.

This is sort of the only thing we know how to do. We have a static data structure. This is a general transformation, dynamization when you have augmentation. We have this data structure. It's got all these augmented things. It's complicated.

But at least it's sort of downward looking. I mean, you only need to store pointers from here down, not up. I mean, your parent points into you. But you have a nice local thing.

So if this guy's not weight balanced, if this left subtree is way heavier than the right subtree by this alpha factor, one over alpha factor, then just redo everything in here. Find the median. Make a perfect binary search tree.

Then the weights between the left and the right will be perfectly balanced. We'll have achieved the one half, one half split of weight. How long before it gets unbalanced again? A long time.

If I start with a one half, one half split, and then I have to get to an alpha 1 minus alpha split, a lot of nodes had to move from one side to the other. The alpha gets messy. So let me just say when this happens, rebuild entire subtree.

I guess it's like a 1/2 minus alpha had to move. 1/2 minus alpha times the size of the subtree had to be inserted or deleted, had to happen, or maybe half of that, some constant fraction. I don't really care. Alpha's a constant.

I'm going to charge to the theta k updates that unbalance things. k here is the size of the subtree. k So when I see a node is on balance, just fix it. Make it perfect.

And if I started out perfect, the subtree started out perfect, I know there were theta k updates

that I can charge to. The only catch is I'm actually double charging quite a bit, actually. If you look at a tree, if I do an insert here, it makes this subtree potentially slightly unbalanced.

It makes this subtrees slightly unbalanced. It makes this subtree slightly unbalanced. There are log n subtrees that contain that item. Each of them may be getting worse.

So if I say, well, yeah, there are these theta k updates, but actually there are log n different subtrees that will charge to the same update. So I lose a log n factor in this amortization. But it's not so bad. I get log n amortized update.

This is if a rebuild costs linear time. This is pretty nifty. I don't have to do rotations per se. I just take all the notes in the subtree, write them down.

I do an in order traverse. I have them sorted, take the median, build a nice perfect binary search tree on those items. I can easily do that in linear time. And so this is like the brain dead way to make this weight balanced tree dynamic.

The original BB alpha trees use rotations. But you don't have to. You can do this very simple thing and still get a log n amortized update.

And the good news is, if you have augmentation as well-- because with this subtree, there's tons of extra stuff, all these arrays and pointers and stuff, it's easy to build from scratch. But it's hard to maintain dynamically. The point is, now, we don't have to.

If ever we need to change a node, we just rebuild the entire subtree. And we can afford it at the loss of a logarithmic overhead. So we had n log to the d minus 1 n time to build the structure.

So for a structure of size k, it's going to be k times log to the d minus 1 of k. We're going to lose an extra log factor. So this d minus 1 is going to turn into a d minus 2 for updates.

So that was the generic structure. And now, if we apply this to layered range trees, we get log to the d n amortized update. Because we had k times log to the d minus 1 of k pre-processing to rebuild node.

And just to recall, we still have log to the d minus 1 of n query. So this was regular range trees. And we've made them dynamic, the same time as range trees.

And still, the query is a log factor faster. So for 2D, we get log n query log squired n update

insertion and deletion of points. Questions about that?

Cool. Well, that is range searching, orthogonal range searching. Let's see. There are more results, which I don't want to cover in detail here. But you should at least know about them. And then we're going to turn to fractional cascading a little more generally.

So where is this result? Somewhere here. So for static orthogonal range searching, range searching is a big area. We're looking at the orthogonal case. There's other versions where you're querying with a triangle or a simplex.

You can query with two-sided box, which goes out to infinity here. All sorts of things are out there. But let me stick to rectangles. Because that's what we've seen and we can relate to.

You can achieve these same bounds-- sorry, no update. You can achieve the log to the d minus 1 n query using less space. So I can get log to the d minus 1 n query and n log to the d minus 1 n space-- that's what we were getting before-- divided by log log n. Slight improvement.

And in a certain model, this is basically optimal, which is kind of even crazier. This is an old result by Chazelle. That's in '86. OK. This is 2D-- sorry, not 2D, just in general.

Turns out this query time is not optimal. If you allow the space to go up a little bit, you can get another log improvement. So I can get log to the d minus 2 and query if I'm willing to pay-- I didn't this is space-- n log to the d n space.

So if I give up another log factor in space, I can get another log factor in query. I don't think you can keep doing that. But for one more step, you can. I believe this is conjectured optimal for query. I don't know if it's proved.

And this was originally done by Chazelle and Guibas using fractional cascading. And we'll see. If there's time next class, I'll show you how this works. But for now, I want to tell you in general how fractional cascading works in generality.

This is part of fractional cascading, this idea of cross-linking from a bigger structure to a smaller one, so that you don't have to keep researching. You just reuse where you were. But there's another idea. I want to show you that idea. So, fractional cascading.

AUDIENCE: Would that work for d equals 2?

ERIK DEMAINE: For d equals 2, no it does not work. So I should say this is for 2D and higher. D has to be bigger than 1. Because you can never be log n.

So for 2D and higher, we could use the trick that we just did. For 3D and higher, you can improve by another long, thanks. Other questions?

AUDIENCE: But you said you can never beat log n.

ERIK DEMAINE: We can never beat log n. In this model, which is basically comparison model, we're comparing coordinates. In that model and many other models, you can't beat log n query. Because in particular, you have to solve the search problem in 1D.

So we're always hampered by that. But the question is, how does it grow with d? And the claim is we can get log n all the way up to three dimensions. Only at four dimensions do we have to pay log squared. It's pretty amazing I think.

OK. Fractional cascading-- super cool name, kind of scary name. I was always scared when I heard about fractional cascading. But it turns out, it's very simple. Goal today is to not be scared.

Let's start with a warm up problem. And then I'll tell you its full generality. But simple version of the problem is not geometry, per se. It's kind of 1 and 1/2 dimensions, if you will. Suppose I have k lists and each has size n.

They're sorted lists, think of them. So we have n items come from an ordered universe. Here's list one. Here's list two. Here's a list three. There's k of them. Each of them has n items.

I would like to search the query. We'll just do static here. Original fractional cascading was just static. And these results are just static.

You can make it dynamic, but there is some overhead. And I don't want to get into that. It's even messier, or it is messy. Fractional cascading by itself is a very simple idea.

Query is search for x in all lists. OK. So I want to know what is the predecessor and successor of x in this list. I want to know what is the predecessor and successor in this list. I want to know what's the predecessor and successor in this list, all of them.

It's more information. If I just merged the lists and searched for x, I would find where x fits globally. But I want to know how it fits relative to this list and relative to this list and relative to

this list. How do I do it?

I could just do k binary searches. So this is an easy problem to solve. You get k times log n. But, now, fractional cascading comes in. And we can get the optimal bound, which is k plus log n.

I need k to write down the answers. I need log n to do the search in one list. It turns out I can search on all k lists, simultaneously get all k answers in k plus log n time.

It's kind of cool and, actually, quite easy to do. We want to use this concept. If I could search for my item, for x, in here, and then basically follow a pointer to where I want to go in here, I'd be done. Sadly, that can't be done.

Why? Because who knows what elements are in here? All of these elements could fit right in this slot. And so how do I know where to go in this giant list?

If these all fit in here and, recursively, these all fit in here, then by searching up here, I learn nothing about where x fits in here. I have to do another search. And then I learn nothing about where x fits in here.

So it doesn't work straight up. But if we combine this idea with fractional cascading, then we can do it. So I can erase this now.

So what do we do? Idea is very simple. So I'm going to call these lists L1, L2, L3 up to Lk. I want to add every other item in Lk to Lk minus 1 and produce a new list Lk minus 1 prime.

So I take every second item here, just insert them into this list. [INAUDIBLE] it's a constant fraction bigger. And then repeat. This is the fractional part. Here, a fraction is one half. You can make it whatever fraction you like less than one.

In general, I'm going to add every other item-- this is in sorted order in Lk, of course-- that's in Li prime-- the prime is the important part here-- to Li minus 1 to form Li minus prime. So I've got this new larger version of L2. I take half the items from here.

Some of them may be items that were in L3. Some of them are items that were originally in L2. But all of them get promoted. Or half of them get promoted to L1. So I keep promoting from the bottom up.

How big do my lists get? What is the size of Li prime? Well, it started with Li. And then I added

half of the items that were in the next level down, Li plus 1.

OK. So this is n. And so this is going to be half of n plus half of another n plus half of-- I mean, it's going to be n plus a half n plus a quarter n plus an eighth n. It's a geometric series.

This is just a constant factor growth. I'm assuming all the lists have the same size here for simplicity. You can generalize.

So I didn't really make the lists any bigger, per se. But I fixed this problem. If all of the elements in L2 fit right here in L1, it's no longer a problem. Because, now, half of the items from L2 now live in L1.

So when I search among L1, I'm not quite doing a global search. But I'm finding where I fit in L1. I didn't contaminate it too much from L2. And then now, it's useful to have pointers from L1 to L2. Let me draw a picture maybe.

So here's L1, L2, L3. So half of the items here have been inserted into here. Now, we don't really know-- maybe many of them went near the same location.

But they went there. And I'm going to have pointers in both directions. Let's say I need them down. So that if I search in here, I can figure out where I am down here.

Then half of these guys-- maybe I'll use another color-- get promoted to the next level up. So maybe this one gets promoted. Maybe this one gets promoted. I guess half would be that one, that one, that one, that one, that one. These guys get promoted to the next level.

OK. I claim this is enough information. This is fractional cascading in its full generality. We have the cross-linking that we had in the layered range trees. But, now, we also have the fractional cascading part, which is you take a fraction, you cascade it into the next layer.

The cascading refers to those guys continue to get promoted. Half of them get promoted up recursively. That's where the name comes from.

So now, how do we do a search? We're going to start at the top. And we're going to do a regular binary search at the top. Because we can afford log n once.

So we do the binary search at the top. So maybe we find that our item fits here in this search. So that tells us, oh, well, this is where item x fits in this list. Great. Now, I need to know where it fits in the next list in constant time.

Well, I need some more pointers for this. So for each item in here, I'm going to store a pointer to the previous and next, let's say, red node, the previous and next node that was promoted from the list below. OK. So, now, I basically know where it fits here. Not quite, because this is only half the items.

So I know that it fits between this guy and this guy in list 2 prime, technically. So the only thing I don't know is, is it here or here? So I compare with this one item. And in general, if it's not a half, if the fraction is some other constant, I spend constant time to look at a constant number of items, figure out where it fits among those items.

Now, I know where it fits in L2 prime. Then I, again, follow pointers to the next items. In this case, they're the white items. So let's say it fits here, basically.

I have a pointer to the previous and next white item from that item. Follow those pointers down. And now, I know it's either, basically, here, here, here. It's somewhere in that little range of either equaling this or being between these two items or on this item or between those two items or on this item.

And, again, constant number of things to look at. I figure out where I belong. In the primed list, which is not quite the original list, maybe I determine that x falls here. And what I really want to know is it's between this item and that item of the original list.

I don't care so much about the promoted lists. So I need more pointers, which tell me if it happens that I fall here, basically, every promoted item has a pointer to the previous and next unpromoted item from the original list. This is static. I can have all these pointers. Let's write them down.

So every promoted item in Li prime-- that means it came from a promotion from below-- has a pointer to the previous and next non-promoted item. So that's an item in Li. OK. That's two pointers.

And that's what we just use. So I found where I was among the entire L1 prime, which was almost like a global search, not quite. And then I follow these points to figure out where it was in the original L1.

Well, so if I found that I was in the middle of this big white region, I need to find the next red region. So it's basically the reverse. Every non-promoted item, every item in Li, has a pointer.

So this is basically Li prime minus Li, if you will. And then these guys need a pointer to the next and previous in that, so previous and the next item in Li prime minus Li. So these are the promoted items.

These are the unpromoted items. So it's actually just two pointers per item. If you're promoted, you store the previous and next. Unpromoted, if you're unpromoted, you store the previous and next promoted.

It's nice and symmetric. It's pretty clean, a lot of pointers, hard to draw, but quite simple in the end. There's two main ideas. One is to promote recursively up, just a constant fraction, so the lists don't get much bigger.

Because it's a constant fraction, the gaps when you walk down are constant size. And so you basically get free relocalization within each list with the help of some pointers to walk down and jump left and right between the two colors. OK. That's basic fractional cascading in that we solved this problem, searched within k lists each of size n and k plus log n time, which is kind of amazing I think, pretty cool.

But there's a more general form of this, uses the exact same ideas. But I just want to tell you how they generalized it. This is Chazelle and Guibas.

So in general, fractional cascading, if you look at what cascading is happening here-- here being here-- we essentially have a path of cascades. We start at the bottom and push into the predecessor in the path. We push into the predecessor in the path.

In the general case, we do it on a graph instead of a path, arbitrary graph. So we have a graph. The input, in some sense, you can think of this as a transformation. But it's for a specific kind of data structure.

The data structure is represented by a graph. And each vertex of the graph has a set of elements or a list of elements. That's what we had here.

We had a path here. And each node in the path had a corresponding list of elements. And we wanted to search among those lists. Before I tell you exactly what search is, let me tell you about the rest of the graph.

And this is, sorry, in an ordered universe. So it's one dimensional. Edge is labeled with a range

from that ordered universe a, b. Every edge has some range. You can think of it as a directed graph. It's probably cleaner.

So when I follow this edge, there's a range here. And, basically, I'm only allowed to follow that edge if the range contains the thing I'm searching for. So here, I was searching for some item x in all the lists.

There's no ranges here. But in general, you get to specify a range. Why do we want to specify a range? We need a sort of bounded degree constraint.

We want to have bounded n degree. So here we had n degree 1 for every node. In general, we don't want to have too many nodes pointing in. Because we want to take half the nodes in here, or a constant fraction of the items here, and promote them into all the nodes that point to it, so that when we follow the pointer, we get to know where we belong here. That's the general concept.

So, ideally, we have bounded n degree. If we do, we're done. We can have a slightly weaker condition, which is called locally-bounded n degree where the number of incoming edges for a node whose labels are ranges

The labels have to have a common intersection, x. So we're searching for some item x. And if all the possible ways we can enter this node given item x-- so this x has to fall in all those ranges-- that should be bounded, so, at most, some constant c.

If it's always at most c for all nodes and for all x's, then this is locally-bounded degree. And these range labels help you achieve this property. If you can constrain that you're only going to follow this edge in certain situations and there aren't too many ways you could have gotten to a node, then you have this property.

AUDIENCE: [INAUDIBLE] bound to x?

ERIK DEMAINE: Contain x is a backwards containment. Let me put it this way. You have a node. You have all these edges coming into it.

I want x to be a valid choice for each of these edges. Meaning, the range, each of them is some interval on the line. All those intervals should contain x.

It's basically, if you laid out all the intervals incoming into this node, what is the maximum

depth of those intervals? What's the maximum intersection between all those intervals? That is your local degree. And as long as that's the constant, we're happy. All right.

So now, let me specify what a search means. This is the problem that fractional cascading can solve. Goal is to find x in some k vertex sets.

So k vertices, each of them has a set. I want to find x in k of them. Not all of them, k of them. That's the general problem. I have a constraint on how those sets are found.

They're found by navigating this graph starting from any vertex. And we navigate by following edges whose labels contain x. So we started some vertex in the graph.

We can follow some edges that contain x. x is a valid choice here that's inside the interval. Then from here, maybe we follow some more where x is a valid choice, and so on. It could look like anything.

It doesn't have to be depth first or breadth first. It's just you follow some tree from some node where all of the edges are valid for x. At some point, you decide that I've seen enough.

And now, the goal is to find in this set, where is x? In this set, where is x? In this set, where is x? In each of these lists, what is a predecessor and successor of x? Question.

AUDIENCE: So there's generally some root node from which all queries start?

ERIK DEMAINE: I believe you do not need a single root node. Each search could start from a different point.

AUDIENCE: OK. So it's [INAUDIBLE].

ERIK DEMAINE: But you're told where. So imagine this is like an interaction between two parties. So the input basically says, look, I'm searching for x. And I'm going to start at this node.

And then the fractional cascading data structure says, OK, here's where x is in that node. It tells you immediately. Why not? Then it says, OK, I'd like to follow this edge and go to this node.

And fractional cascading says, OK, here's where x is in this node in constant time. OK. Then now these two guys are active. And now, the adversary, the input, whatever, can decide, OK, I'm going to follow this edge, or this edge, any order. It can build this tree in any order.

And every time it says here's the edge I want to follow, the fractional cascading data structure

in constant time tells you here's where x is among all the items in that node. How does it do that? With fractional cascading.

You just take half the items. Half doesn't work anymore. Now, it depends on that bounded n degree. But you take some function of that degree c, take some constant fraction of the items, promote them to all the things, keep going. It's a little trickier, because now you have cycles.

So you could actually promote back into yourself, eventually, by chain reactions. But if you set the constant low enough, it's like radioactive decay. Eventually, it all goes away, right? I wish.

So it's much better than radioactive decay. Radioactive is logarithmic. This is exponential. So it's decreasing very quickly. After log n steps, all your items are gone.

So, yeah, maybe you go in a short loop for a while. But after log n steps, it's all gone. So you're, at most, increasing by a log factor. In fact, you just increase by a constant factor, because the number of items that remain gets so tiny very quickly.

So I'm not going to go into the details, but you just take this list idea, apply it to your graph. It works. It gets messier. But in this very general scenario, you can support these searches in k plus log n where n, let's say, is the maximum size of any vertex set.

So just it directly generalizes. And this is the thing that you can use to get this log factor improvement and many other things. Actually, this was such a big thing at the time.

There were two papers on fractional cascading, part 1 and part 2. Part 1 is what is solving this. And part 2 is applications. They solved a ton of problems that no one knew how to solve using this general fractional cascading technique. That's it for today.