MITOCW | watch?v=FzS0n_Z8lrk

ERIK DEMAINE: All right. Time for some more geometry, and, in particular, some more fractional cascading, which is a cool topic we saw last lecture. And then we're going to do a different kind of data structure called kinetic data structures, where you have moving data. And that will actually be most of the lecture.

But last time, we saw this nice general black box transformation, fractional cascading, and we didn't really see a lot of applications of it. We saw a simple example in orthogonal range searching. But I want to show you today a much cooler one in the original fractional cascading papers.

So remember, in general, what it lets you do is search for a common element x, find its predecessor and successor in k sorted lists, in order log n plus k time, instead of the obvious k times log n, where n is the length of each list. And the general version that we talked about is, if you're navigating a graph, and at each node of the graph, you have one of these lists, then you can instantly know where x fits in that list in constant time per thing, as long as you spend log n time to get started, provided your graph has bounded degree. That was necessary to do all the fractional cascading stuff of copying half into the next level up.

So we're going to use this result to get a lag factor improvement in our old friend orthogonal range searching. So let's do orthogonal range searching.

So last time, we saw how to do two dimensional orthogonal range searching in log n per query. That was using fractional cascading, or, really, half of fractional cascading, which was just a cross-linking idea. This time we're going to do 3D orthogonal range searching in log n time. Our space is going to go up by a couple log factors. But this is pretty cool. In general, for d dimensions, this gives us log to the d minus two, whereas last class, we saw how to do log the d minus 1.

Now, this is static only. You could probably do poly-log update. But let's not worry about updates. We're just trying to do static.

So 3D orthogonal range search using fractional cascading-- this is kind of a tour de force. There's a really cool result. It's in the "Applications of Fractional Cascading" paper by Chazelle and Guibas. And it proceeds in four easy steps. Each of the steps are easy, but it's kind of

amazing where we get to.

We're going to start out with a two dimensional restricted orthogonal range query. We'll see why at the very end. But this is actually from another paper of Chazelle in the same year.

OK, in general, remember, in 3-D, we're trying to do querying with a rectangle-- a1, b1, a2, b2, a3, b3. We want to know all the points in that rectangle. So I'm going to start out with a very restricted form, which is only two dimensional, for whatever reason. And we'll see why later.

y-coordinate and z-coordinate, skipping x-- and the left end point doesn't exist. So you go all the way up to b2, and you go all the way up to b3. This is a quarter plane in two dimensions.

I want to know all the points in there. It's the same as saying all the points that are dominated by this point. Both y- and z-coordinates are dominated by that yz coordinate. This is b2, b3.

So we can solve this in log n time plus k. But I want to be a little bit more precise about what that log in is, and say that this costs whatever it costs to search for b3, the z-coordinate, in a z list-- in a list of z-coordinates of points-- plus order k. I write it this way because if we have many searches among lists, then we can speed things up. So I don't want to just think of it as log n. I want to think of it as one of these fractional cascading operations.

So this is the time bound I want to get for finding the k points in this search range. Now, here's the fun part. We're going to transform this into a kind of stabbing ray problem, like we saw last class, and in the retroactive stuff, and so on.

So let's suppose I have some points. I'm just going to draw an arbitrary arrangement. Hopefully, it's reasonably interesting. Those are the points I want to be able to query. And if I'm given, say, a query like this one, I want to know all the points in here.

So what I'm going to do is draw a leftward horizontal ray from the query point. And I'm going to draw-- maybe use a color-- upward vertical rays from each of the points. OK. And where there's crossings, those correspond to points that are in the query quarter plane. OK, so same problem-- but now, thinking about rays.

So here's a cool thing you can do with this approach. So we want to preprocess these vertical rays so that, then, we can stab with a horizontal ray. And it's actually easier to think of it as coming from the left, because that's kind of a consistent x-coordinate, and walking to the right. I'd like to find this intersection, then find this one, then find this one. Eventually, I get to the

desired y-coordinate. This is the y direction, and this is z direction. Then I stop.

So if I could get started over here in log n time, and then do a walk in constant time per intersection, I'd be golden. That is possible. And the way that Chazelle did this-- I'm going to erase the query ray-- is to decompose the plane in a pretty simple way. We're going to draw a horizontal segment from each point. We're going to extend it to the right until it hits something, extend it to the left. In this case, it goes off to infinity.

Here, I extend this guy. I extend this guy. I extend this guy. I extend this guy. And extend this one out here. I'm going to add one more point over here-- a little more exciting. So this stops there. This goes to there.

AUDIENCE: One more-- to the right.

ERIK DEMAINE: One more up here-- thanks. OK. So this is kind of decomposition into slabs or bricks or something. It looks good. And so the idea is, over here, there's, at most, n different rays that make it all the way to the left.

Do a search. That's your z search. So that's this search for b3 in the z list. So remember, our goal is to get to here. So we search for that z-coordinate over here. We say, OK, it falls here.

Then I enter this face. I'd like to then navigate to this face, say, OK, that's where I am now. By crossing this edge, I know that this point is actually in my answer. And then I cross this edge, so I know that this point is in my answer. Then I cross this ray, so I know that this point is in my answer. Then I say, OK, I reached my desired y-coordinate. Stop.

So if I can do each of these traversals in constant time, I'd be all set. So I do one search at the beginning, then constant time per query. We know how to solve this problem. We can do it with range trees and log n time preparation. But this is a particular way to solve it that will work with fractional cascading when we do many of them.

OK. Cool. Can I do this in constant time? Maybe not, because it's this sort of scenario. If I draw a whole bunch of points like this, they'll each have a segment. And in general, this face will have large degree. And so I need to sort of find my z-coordinate again, somewhere in here.

It turns out, with, essentially, fractional cascading again, you can avoid that. If you have many segments here, just extend half of them. So maybe I'll extend this one over and this one over. It looks like fractional cascading.

I'm taking half of the elements here, inserting them into the previous list, which is the left side of the face. If this now has too many, well, half of them get promoted. But it decreases exponentially. And so the total amount of extra edges I'm adding here is only linear. So linear space-- I'm not going to prove this formally here, but it's the same idea as fractional cascading. We just need it as a tool to get to 3D.

You can extend these things, and then every face will have bounded degree. And so you can just look at every single rightward edge in constant time, and figure out which one has your z-coordinate. Follow that edge. And so every time you're crossing a ray and getting an output, you can pay only constant time to get it. Pretty cool. That's the first step. Question.

AUDIENCE: So when you say that a particular face has too many crossings, what do you define too many?

ERIK DEMAINE: More than a constant. In general, you just look at the right side of a face, and just take half of those things and propagate it to the left. Just do that right to left in one pass. I think you might also need to do it left to right, if you want to have bounded leftward degree. But I'm not sure that really matters.

I think you just do one pass right to left, and half the guys keep getting promoted. And it's the same thing, where you're promoting from the LI primes, not from the LI. So it's everybody who came from the right plus whatever you originally had. Half of them get promoted to the left, but because it's geometrically decreasing, all is OK. This is earlier than fractional cascading, but I would guess it's what motivated them to then do general fractional cascading.

Other questions? I know this is a little bit vague. But the more exciting stuff, to me, is the next three steps. So let's move on to those. .

So this is a tool for doing two dimensional quarter plane searching. The next thing we're going to do is make it three dimensional. This is actually something we already know how to do. And we're going to do it in exactly the same way we knew how to before.

Suppose you have a three dimensional query, and two of the intervals start at minus infinity, but the new x-coordinate is a regular interval. You can specify both endpoints. I want to do this in log n searches plus k. k is the size of the output. How do I do this using one?

Two words-- range tree. Yep. Easy. Just do 1D range tree on x. And then each node stores that data structure-- one-- on points in the subtree. And so just like before, you get log n

subtrees that represent your x interval. You look at the root of each one, and it stores a one data structure. You query each of them for b2 and b3 for that interval among the y- and z-coordinates.

And each of them costs a search plus order k. The k's sum up to order k. So we end up doing log n searches of that type plus k time. And we can now solve this kind of 3D query. This is really easy. This is what we did last class.

The cool thing, of course, is that, by doing log n searches, it's always searching for the same thing-- b3-- in various c lists. We know by fractional cascading this is actually log n time. We're doing log n searches in k lists-- and slightly different k here, sorry. It's actually log n lists. But we know from this bound, we're going to get order log n plus k.

But I don't want to do fractional cascading yet, because we're not done. This is a sort of three dimensional orthogonal range query. But I want to put a2 here and a3 here. We're going to do that step by step. First step is a2. We're going to do it in exactly the same way, twice.

Again, I want to do it in log n searches plus k. It's the same time bound I want to put into a2. The cost will be a log n factor in space. And it's a cool transformation. It's a general transformation. Whenever you have a data structure has a minus infinity, you can turn it into a lower bound, magically-- almost the same way as we did here, except we're not going to lose a log factor in time-- only in space.

So I'm going to say it's kind of like a range tree on the y-coordinate. y-coordinate is the one that we want to extend. And you may remember there was this brief question last time. In a 1D range tree, what key does a node store? It just has to store something that's in between what's in the left subtree and what's in the right subtree.

So I proposed you could store max of left subtree. And that's enough to do a search. Then you know whether you should go in the left subtree or the right subtree. Just compare with that key. So same as before-- we're going to store that key. Except now, I'm making it explicit, because we really need to know what that key is.

A node v will also store two of these. So it's going to store the two data structure on the points in the right subtree-- not the entire subtree, just the right subtree. And it's going to store a y-inverted two structure on points in the left subtree.

So this is the new part. And what does y-inverted mean? It means that you can search for

boxes like this. So regularly, a two structure is sort of left-centered, and the left end point is undefined, and it goes up to b3. Now, I want something that's right-centered. It goes up to infinity on the right side. But you can start at an arbitrary a2.

How do I make such a data structure do exactly the same thing, but with this inverted, which means, do exactly this, but with this inverted? Easy to do-- just twice as many data structures. So you can think of that as one prime and two prime. I'll call this two prime, explicitly. OK, now, the big question is, why is this enough? So let's do that.

How do I do a query in data structure three? The basic idea is simple. We're going to walk down the tree.

Now, before we walk down the tree, and the interval is represented by log n different subtrees-- we can't afford that anymore. We can really only afford a constant number of calls to this data structure, if we're not going to get an extra log blowup. So I don't want to walk and then fork and then visit all these subtrees. I could do that.

But it turns out, with this structure, I can be a little bit more efficient. Because, essentially, I want to do an interval like this. The queries I'm allowed is, I can do a query that's infinite to the left, and I can do a query that's infinite to the right. So the intersection of those would be what I want. I can't really compute intersection. So it take too much time to list those.

But if, somehow, I could get this left endpoint in another way, using the tree, then I can do a leftward infinity query. So that's what we're going to do. And once I say walk down the tree, it's pretty clear what you have to do.

When you visit a node, if the key of the node is less than-- what are you searching for-- a2, b2-- so a2 is less than b2. And if the key of the node is to the left, that means that the stuff we're interested to is to the right. So walk right. If the key of the node is greater than the interval, then walk left. That's sort of the easy case.

And then the interesting case is this fork. Before, we had to do a lot of work at the fork. Now, I claim we only need to do constant work at the fork.

So if the key falls between a2 and b2, so our interval is stabbed by the key of the node, then I want to query the two data structure-- two is the right number-- yeah-- and I want to query the two prime data structure. So I'm going to do two calls to data structure two. And so I only get a

constant factor blowup.

And what could I possibly search for? Well, two-- I'm able to do-- what is it-- a1, b1, minus infinity b2, and minus infinity b3. We're not fixing the z-coordinate yet. And with two prime, I can do the left endpoint bounded. Once I've set things up, this is, like, the only thing you could possibly do. But I claim it's actually the right answer.

OK, what does this mean? Two prime is-- we're doing rightward infinity searches in the left subtree. So here's v. Here is the left subtree of v, and the right subtree of v. So this is a bunch of points down here, a bunch of points down here. We know that the interval looks something like this. It straddles this node. That's what this means.

And so what we'd really like is this stuff plus this stuff. That looks good. Because this goes to infinity on the right. That's the two prime search. And this goes to infinity on the left. That's the two search.

As long as you restrict to this subtree, which is what we've always been doing, and, in particular, here, too, it's only on the points that are in the right subtree. Two prime is only on the points in the left subtree. And so that's it. We're golden. Actually, we only need to do a rightward infinity search, and a leftward infinity search. So that is data structure three.

And it's really easy. Once you have the ability to do a 3D search-- even if this one was minus infinity b1-- if we just could do-- would you call that-- octants-- octant search in 3D, then we could just sit there and apply this transformation, and turn each one of them into double-sided, with a log n penalty each time in space, but no extra query time. We do a single log n traversal of this tree. So this part costs log n. And then we reduce to the previous problem-- it's constant number of calls.

So I could have written minus infinity here. But I pay an extra log factor in space. So this one we get for free, because we're using range trees. We do one transformation. We get this one. And now we're basically done, because we just do this again on the z-coordinate, and we get a1, b1, a2, b2, a3, b3, as desired. For our queries, just ditto on z, and using three in place of two.

So we want to add in this a3. We build a three data structure. We build three prime data structures in exactly the same way. Three primes are on the left. Threes are on the right. Every node in this range tree on z. And we do a log n search at the beginning. Then we do a

constant of calls to the data structure three. Data structure three does a log n search at the beginning-- constant number of calls to data structure two.

Data structure two does our usual range tree thing-- identifies log n different calls to data structure one. Data structure one is a search and a list, plus every time I walk to the right and spend constant time, that is an element of my output. How much time has this taken total? Normally, there's log n of these, plus order of k. Normally, that would be log squared n plus k. But with fractional cascading-- we have to check this is valid within fractional cascading. We've got a graph of data structures here, but each node has only constant degree.

You can come from your parent. You can go to your left child. You can go to your right child. And you can go to previous dimension data structure, or the inverted version of it. So the degree five-- if you count in and out degree. And so fractional cascading applies. We're always searching for the same thing-- not quite, actually-- a little bit of a cheat.

Because of this inversion thing, we'll sometimes be searching for b3 in the z list, but we'll also sometimes be searching for a3 in the z list. But hey-- just a factor of two. So we have two fractional cascading data structures-- one for searching for b3, one for searching for a3. It's the inverted versions of one, and the uninverted versions of one-- z-inverted.

But that's fine. And in the end, we get log n to do the first search, and then plus the number of searches. Number of searches is log n plus k, where k is the size of the output. So total time is log n plus k. This is pretty amazing. We can do 3D orthogonal range queries, still in log n.

And if you go to higher dimensions, the best we know, basically, is to use range trees. And so you get, in general, log to the d minus two n plus k. So last class, we could do one. Now, we improved it by one more. Questions?

AUDIENCE: Yeah, I have a quick question. [INAUDIBLE] question. So we do three, and then we do four. And it's not hard to imagine that same [INAUDIBLE], when you might try to do the same argument for step three, step four, step five.

ERIK DEMAINE: Why can't we keep doing this for all dimensions? So what three is, and also four, is using the same transformation, which is, if I have a minus infinity something interval, I can transform it into an a2, b2 interval. So I can make a one-sided interval into two-sided. The trouble is actually getting the one-sided interval.

So we started, in one and two, just getting up to three dimensions. And that's where things are

hard. So fine, in two dimensions, we can do all sorts of fancy tricks. We saw one way to do it last time. This is a particularly cute way to do it that lets you fractionally cascade. We could add a dimension.

To add a dimension, we just use range trees. This is kind of pathetic. Every time we do this, we're going to pay a log n factor. So we could afford to do it once and get to 3D. We paid a log n factor here, but we were lucky fractional cascading will remove one log factor, but only one.

If we had to go to four dimensions, we'd have to use another level of range trees. And then we'd get a log squared searches. And then we have to pay unit cost for every search. So we'll get log squared for four dimensions. So it's just getting up to the right number of dimensions that's hard.

What you're seeing here is that one-sided intervals are just the same as two-sided intervals, if you don't mind extra space. The space here is, I think, log cubed n log cubed n. Data structure one is linear space, but every other level, we lost a log factor. So one of those was to get up a dimension by range trees. The other two were to convert the one-sided intervals into two-sided intervals.

And that generalizes. You could do that as many times as you want. The hard part is just getting the right number of intervals in the first place. And that's where we pay log per dimension. So kind of annoying you can't do log n for any dimension, but-- pretty sure that's impossible. There are models under which it's impossible, but we're not going to get into that. Yeah?

AUDIENCE: So when our query is log n plus k, k is actually the number of points that are coming back, because--

ERIK DEMAINE: Yeah. Good question. Here, k has to be-- this is for what we call range reporting, where you really want to list everybody in there. And if we wanted to do range counting queries, which just give me the number of elements that match, I don't think this will work. In particular, our seed data structure up here had to pay for everything. It doesn't know how many times it's going to have to walk to the right. It's got to actually do it. So range counting-- not so much.

Our previous data structures could do range counting, without the plus k, just paying log to the d minus one. But this is just for range reporting. Good question. I don't think anyone knows how to do range counting faster.

AUDIENCE: And the reason we're only hitting points we know about is because the one data structure is on the bottom, so we never actually--

ERIK DEMAINE: Right. So you need to check, why is it only order k, where k is the actual output size? Because by the time we get down to the one data structural level, we're guaranteed that x already matches. It's already in our interval a1, b1 by the range tree. And we're guaranteed that these two open intervals-- the minus infinity actually is a2 here, and it actually is a3 here, or, actually, something bigger than it.

So we're guaranteed whatever this thing outputs is a result. And we're never doing overlapping intervals. So we never double charge. Yeah. You do need to check that. There's a lot of pointers to follow here, but it works.

All right. You look convinced. Let's move on to kinetic data structures.

The idea with kinetic data structures is, you have moving data. Deal with it. So normally, we're thinking of data that-- at best, it's dynamic data, meaning we can delete something and then reinsert it. But what if everything is constantly changing?

So normally, OK, I've got some points in my data structure. But now, what if they also have velocities? And maybe some guy's just sitting there stationary, but some of them are moving relative to it. And my operations-- this is kind of like time travel, but now we're going to time travel into the future, which we do all the time. But we'd like to do it really quickly, and say, OK, advanced time by five units. And now, in that frame, do an orthogonal range query or something. Do some kind of query.

We're always going to be doing queries in the present. So this is not fancy time travel. This is regular, forward time travel. We just want to quickly say, jump forward 10 time units, do some queries, jump forward some other time units.

There's actually another operation, which is, ah, this point is no longer moving in that direction. Now, it's moving in this direction. So the operations are, advance to time t-- so this is like setting now equal to t-- and change a point x to have some new f of t trajectory.

AUDIENCE: I thought it was supposed to be arbitrary.

ERIK DEMAINE: Well, arbitrary trajectory is not-- not quite. I'm going to restrict what those trajectories are. But

that is the remaining question. What kind of f of t's do we allow?

I'm drawing the picture in d dimensions, let's say. And most of the work in kinetic structures in 2D-- a little bit in 3D-- but I'm going to focus today mostly on 1D. Because it's easy to analyze, clean. There's a lot of open questions in 2D. So we can also think of these-- there's points on a line. They have velocities, accelerations, who knows what. That would be-- OK, let's say-- models of trajectories.

The simplest one that would be affine-- f of t equals a plus bt. a and b here would be points. a and d dimensions are just values in one dimension. So the motivation is, maybe, you have cell phones or cars or something. You have some current estimates on which way they're going and at what speed. Then this would be the simple model.

And if either those two things change, you have to do a change operation. So you pay for that every time they change their trajectory. But otherwise, it's going to be super efficient, because advance is going to be super fast. That's the plan.

But maybe it's not just position and speed. Maybe also have acceleration and stuff. And then the extension of that would be bounded degree algebraic, which is, you have some polynomial of bounded degree-- sorry, that should be c-- but bounded. And the reason we care about bounded is really the following. There's an even more general model which we call pseudo algebraic.

So we would like to bound the cost of this advance operation. What we'd like is that when we advance time a large amount, stuff doesn't change crazy number of times. And pseudo algebraic says that if you look at anything you care about-- we call this a certificate of interest. We'll be talking a lot about certificates today.

Certificate is something like, is this point left of that point. It's a Boolean question about the moving points. It's either true or false.

And what I'd like is that-- I have some point. It has some crazy trajectory. I have another point. It has some crazy trajectory. I don't want, is this point left of this point, to change an unbounded number of times. I'd like it to be constant, as long as no change operations are called.

So for a single trajectory, I'd like these to flip a constant number of times. Now, if you're algebraic with bounded degree, then that will be the case. But more generally, as long as you

sort of switch between left and right bounded number of times, then that particular certificate will only change unbounded number of times. But in general, anything where all the certificates I care about change a constant number of times is just as good as algebraic.

This is really why we like this. I think I've talked enough about trajectory models. So it's not totally generic. But it covers a lot of things you might care about.

AUDIENCE: [INAUDIBLE] certificate of interest?

ERIK DEMAINE: Certificate of interest is just a Boolean function on a constant number of data points over time-- a constant number of trajectories, I should say. I'll probably actually define certificate right now.

AUDIENCE: Good.

ERIK DEMAINE: Certificates-- I'm going to define them more as a tool that we use to build data structures, but they're really the same thing that I mean here. So pretty much all kinetic data structures follow this unified approach. Kinetic data structures are actually a pretty new thing, introduced in 1999. So as data structures go, that's new. And over the last however many years that is-- 13-- there's been a bunch of kinetic data structures.

So what we're going to do is store the data structure that is accurate now. So this will make queries about the present really easy. It just is . You look at the data structure, you do a query. And so the hard part becomes, how do I do advance? How do I advance time and make the new data structure, which is correct, about the new now?

The way we do that is with certificates. So what I'm going to do is additionally store, basically, a proof that this data structure is valid. We can say conditions which are currently true-- and as long as they remain true, the data structure remains valid. So these conditions are true now, and-- sorry-- under which data structure is accurate or correct. And those conditions are true now.

OK, so for example-- well, we'll get to examples in a moment. We'll just keep abstract for a little bit longer. Each of these certificates, you can figure out a failure time. So you have some certificate, like, this point is to the left of that point. And you just look ahead in time. If you have some constant defined trajectories, you just see, when is the time when they will be aligned vertically? After that time, they're going to switch who is left of whom.

So just compute that failure time. I'm going to assume-- that's another assumption about the trajectory model-- that this takes constant time for trajectory. I do that for every certificate, and then put those failure times into a priority queue. Do you want to ask your question?

AUDIENCE: So is it required that a certificate needs to be-- like, that we've checked that a certificate is true. It just needs to be bounded by order one, or--

ERIK DEMAINE: Yeah. So I'm assuming checking a certificate takes constant time, whether it currently holds, and computing the failure time takes constant time. Yeah. I mean, you could assume it, or maybe you pay a little more. I mean, this is how we're going to build the data structure. So whatever it costs, it will cost. But I think everything we'll talk about and pretty much every certificate out there is sort of a constant size thing.

Some of them are bigger. But we'll get to the costs involved. Maybe we won't worry about time too much.

So priority queue is going to take log n time, where n is the number of certificates in the data structure, to find, when is the next failure? So if I want to do an advance, basically, I'm going to do discrete event simulation. I find the next event when something changes, i.e. a certificate fails. I'm going to fix that event-- fix that certificate-- make the data structure correct again. Then advance to the next failure. Repeat. As long as there aren't too many certificates to mess me up, this advance will be fast. So yeah.

So here's how we're going to implement advance. Basically in all kinetic data structures, we just say, while t is greater than or equal to the next failure of the priority queue, we advance to that moment in time, when something interesting happens. We do an event. I'll write it this way. And then we set now to t.

And this event thing has to somehow fix the data structure and fix the certificates. So that is the challenge, is, how do you deal with an event when one of your certificates breaks? So if you have a data structure, and you want to make it, kinetic you just first write down some certificates under which it's guaranteed to be valid, and then see how to fix them as things happen. Then there's the analysis issue, which we will get to.

Let's start with an example before we get to analyzing, so this becomes a little more concrete.

AUDIENCE: I have another question.

ERIK DEMAINE: Yeah.

AUDIENCE: So are the change operations sort of online, or do you have the sequence of changes that you're going to [INAUDIBLE]

ERIK DEMAINE: All of these operations are online. So at any moment, someone says advance, someone says change, or someone says query. And query is respect to now.

So we have no idea which of these are coming, in what order, whatever. I don't think anyone's studied the case where you know up front all the things that are going to happen. Though there are, presumably, applications for that. I mean, time can just be a euphemism for a dimension, or whatever. But a lot of the kinetic people really want to be tracking points and maintaining what's happening.

So let's do a 1D problem-- a simple one-- sort of the most basic problem, which is a predecessor problem. Insert, delete. We won't worry too much about insert and delete here. It's hard enough, because the points are moving around-- and predecessor and successor.

So I want to know, on the line, I have some points. They're moving. Now, query is, at the current time, who's to the left and who's to the right of this query? OK.

How do we maintain this? This is a problem we'll be studying a lot in this class. But the basic structure for solving predecessor, insert, delete, predecessor, is--

AUDIENCE: Binary search tree.

ERIK DEMAINE: Binary search tree-- balanced binary search tree. OK, so let's use a log n high AVL trees-- whatever. So what do we need for certificates? I was going to use this as an example, but it's actually a little tricky to think about what the certificates are. Because the binary search tree property is x plus or equal to x greater or equal to x.

That's kind of a lot of certificates. If I want to compare x to every single guy in here and compare x to every single guy in here, that would be a, I think, quadratic number of certificates, in general. Almost everyone has a relation. I'd prefer to get away with fewer certificates. Because then, less certificates will fail.

So cute idea-- I really only need to compare x with this one and this one. -- the max in the subtree and the min in the subtree. In general, if I look at the data in sorted order, it has to

stay sorted order, where it's not going to be sorted. But this is an inorder traversal.

Inorder traversal is something we can understand without knowing what the data is. Because remember, data is constantly changing. We can't really use the keys here. But we can use the abstract shape of the tree and do an inorder traversal, and say, look, as long as x i is less than or equal to x i plus one in the inorder traversal for all i, then this is a valid binary search tree. If an inorder traversal stays sorted, we're golden.

So those are my certificates. There's only n of them. So that's nice.

And we need to check that we can compute a failure time. This is usually really easy. But we'll go through the exercise of writing it down.

So I want to know, among all times greater than or equal to now, when will x i-- am I doing strict here? This should probably be greater than. Yeah, so that's why I have an infimum.

OK, I take the earliest moment when x i of t is greater than x i of plus one, which is the opposite of what I want, and take the infimum of those times. And so that will be the moment of transition when they're equal. And then-- boom-- it's going to jump over. I'm assuming these things are continuous.

AUDIENCE: That's why you take the infimum?

ERIK DEMAINE: Yeah. So it would be an infimum, because these guys are going to cross. I mean, I don't care about this kind of happening. But if it's going to actually go across, then there'll be the moment of transition where they're equal. And that's going to be this infimum.

OK. How do you compute that? Well, it depends what these trajectory functions are like. If it's algebraic, then this is just a polynomial thing. You can do it in bounded degree. You can do it in constant time. That's our model.

OK, so you put them into a priority queue. Do this advance. And now, the question is, how do you process an event? When one of these things happens, you're about to transition to x i being bigger than x i plus one. What do you do?

So that's the real heart of the data structure. Although, really, the heart of a kinetic data structure is the choice of certificates. If you choose certificates well, then you're going to be efficient. We haven't defined efficient yet. We will. Otherwise, you're not going to be so fast. So

it's all about using certificates right. The rest is kind of straightforward.

So let's suppose that this certificate is about to fail. And we're guaranteed by this algorithm that it fails now. We have advanced to the time when it is about to fail. We process that event. So now is the time when these two things are equal. Right after now, we will get to greater than.

So here's what I do. Swap them in the binary search tree. So right now, maybe just in general, it's going to look something like this. We have x i, x i plus one. Or it could be the reverse scenario, where x i plus one is a leaf, and x i is the predecessor. Right now, they're equal in value. So I'm just going to interchange them, move x i up here.

So it's a little confusing. But this is x i plus one, and this is x i. Replace those. These are really pointers to the trajectories, however they're described. Interchange them. It's still valid as a binary search tree. Actually, this binary search tree never becomes invalid, because at this moment, they're equal. And after now, things will continue to be OK, because this guy will be bigger than this one. That's the assumption.

We just need to fix the certificates. Fixing the data structure was pretty trivial-- constant time. So what certificates do we need to do? We need to add this new certificate. I'm going to call it x i prime is less than or equal to x i plus one prime. So this is actually x i plus one, formerly known as x i plus one, formerly known as x i. But it'd be really confusing.

So I'm going to use the primes to be the new data structure. Because this is always the kind of certificate we want. We also need to update certificates.

So there used to be an x i minus one less than or equal to x i. We want to turn that into x i minus one less than or equal to x i prime. And we used to have an x i less than or equal to x i plus one. We want to turn that into x i prime less than or equal-- oh, that one, we already did. Sorry.

So I want x i plus one to x i plus two. And I want to do, now, x i plus one prime to x i plus two. OK, basically, wherever the primes happen, which is x i and x i plus one-- whatever certificates they're involved in, you have to update them-- meaning, rip out the old one, put in the new one.

And the main issue here is that you have to maintain the priority queue. So you've got to take them out of the priority queue, recompute their failure times, put them back in the priority

queue with that failure time. But the point is, this is a constant number of updates.

In general, pretty much, as long as the things you're messing with-- the items you're changing-- are involved in a small number of certificates each, then this will be cheap. I'm writing this explicitly. But in general, it's just, update whatever certificates those points were involved in. The issue is how many things you change, and how many certificates each of those things is in.

In this case, changing a constant number of things-- each of them is in a constant number of certificates. So it's not constant time, but a constant number of calls to priority queue updates-- so log n time.

AUDIENCE: Question.

ERIK DEMAINE: Yeah, question.

AUDIENCE: So but how do you find the things that you just changed in the priority queue of the failure times.

ERIK DEMAINE: How do you find them in the priority queue? How do you find these x i's and x i plus ones? Every point knows what certificates it's in, meaning it has a point or two-- a list of certificates it's in. There's only two of them. And each certificate has a pointer into its existence in the priority queue. So then you know where it is in the priority queue and you can rip it out. Yeah. Lots of pointers and cross-linking. Yeah?

AUDIENCE: So in something that's pseudo algebraic, any certificate of interest changes o of one times. But here, there's o of n squared-- even though we're only taking n certificates at a time, there's o of n squared possible certificates in each of them.

ERIK DEMAINE: You're asking about efficiency, which we'll go to now. This looks disconcerting, because even though pseudo algebraic gives us that each certificate won't change too many times, there are quadratic number of potential certificates. They could all become relevant. And indeed, that is the case, if you have these points, spaced out a lot, and then you have these points, spaced out a little-- four and four.

And these guys are all moving constant velocity this way. And the white guys are stationary. Then what happens? Well, there's event, event.

You're going to get n squared events. You get, like, n over two each time you cross a white point.

I call this OK. Why? Because of efficiency.

The claim is, this is sort of the best you could hope to do. So in that sense, it's as good as you can hope to do. If you want to maintain predecessors-- let's put it this way. If we need to know-- "know" is a sort of vague thing-- the sorted order of the points, then you need an event-- if we're going to keep a data structure that is always accurate now-- so this is sort of an assumption-- then you need an event every time you have an order change. That's sort of a tautology. But it's a perspective.

And what's happening here is that we have an event every time there's an order change. So sometimes, yeah, it's going to be bad. Worst case here is quadratic. You can actually prove for pseudo algebraic, it is order n squared events. So this was really a lower bound. But it is also order n squared all the time, because, if you look at any pair of guys, they're only going to change a constant number of times who's above whom, for pseudo algebraic, if there are no change events.

So for efficiency, because it's really hard to analyze if points are changing their trajectories all the time, assume there's no changes. We basically just advance to infinity. How much could that possibly cost? That's how kinetic people like to analyze things.

And for this problem, for maintaining sorted order at all times, the worst case answer is theta n squared. This data structure achieves theta n squared, so we consider it worst case optimal, in this weird sense.

OK, this is if you really want to maintain the sorted order. Now, we didn't say we wanted to maintain the sorted order. We said we want to maintain a predecessor data structure. But it feels kind of like those are the same thing, maybe. I don't know if there's a formal sense in which this is the case.

And efficiency, in general, is the vaguest part of kinetic. And for each problem, you have to think hard to understand, what does efficient mean for this problem? But maybe don't even worry about efficiency-- what does a lower bound mean-- but bottom line is-- worst case, n squared events, if there are no change operations.

So you can think of the running time, if you want, of jumping to infinity as order n squared in the worst case. That's how we analyze these things. There are other things we'd like to analyze. Efficiency is one of them.

We have three others, which I sort of hinted at. There's responsiveness, which is time spent to do an event. So when a certificate is invalidated, how much time does it take to do that?

There's locality, which is closely related. This is the number of-- oh, I see-- number of certificates per data object. I said over here, this is good, because each item x i is only involved in a constant number of certificates. So we say the locality is constant-- constant number of certificates per object.

Usually, locality implies responsive. As long as you can update the data structure, you can also update the certificates in whatever-- you pay a log factor, because you're updating a priority queue. But you can update all those events, or redo all the certificates in an event, provided you are local. So locally usually implies responsive, more or less.

And then-- what's the other one-- compact. This is about space. And we just like the number of certificates to be small. So in this case, number of certificates is linear. That's that up here. So we consider the structure to be optimally compact. You need at least n.

So in our case, we're getting order n here, order one here, order log n here. And efficiency I guess you'd call constant, kind of. Efficiency is the ratio of how many events you do divided by how many events you need to do. And here, it is optimal, in a certain vague sense. Sorry, that's a little unsatisfying. But it's unsatisfying. That's the literature for that problem.

We're going to do another problem, which is more satisfying, I would say. And it kind of shows you why kinetic problems are pretty interesting. Data structures are often pretty darn simple, and the fun part is the analysis. Or, where it can get intricate is the analysis. So let's solve kinetic heap. Our goal is to be able to do find min.

And yeah, there's also insertions and deletions. But I'm not going to think about them. Because they're not really that different. They're kind of like a change. Let's not worry about it.

The goal is to maintain the minimum. Now, if you look at this point set and their velocities, how many times does the minimum change? Once. Initially, the min is this. Eventually, the min will be this-- one change to the minimum.

So this data structure is a bad way to maintain the minimum. Yes, it does maintain the minimum. But it's going to spend a quadratic number of events in this situation, where the min only changes once. So can we get a-- ideally, you have one certificate, which is like, this point is the minimum. But that's not such a good certificate, because then, every point is involved in it.

And how do you know when it's going to be violated? I guess you could compute it in, like, linear time or something. It's not a good plan. We need to break down the certificates that only involve a constant number of things and maintain the minimum. And somehow, I want to get below quadratic number of events, in the worst case. And you can. And you do it with a heap.

Remember heaps? Store min heap. So min heap has this property. You have x, y, and z. Then x is less than or equal to y, and x is less than or equal to z. This is nice, because-- local property. These are my certificates. For every node x, I have to have this property.

So I have order n certificates. That's good. I had order n certificates before. But I claim these, magically, are easier to maintain than those, even though they look almost the same-- kind of crazy.

I don't know if I really need to write it, but how do we do an event? If x and y are about to invert in order-- so currently, x lets me go to y. In a moment, x will be bigger than y. I swap x and y, update all the certificates-- constant for each.

There's two items I'm moving. Each one is involved in a constant of certificates-- three now, instead of two-- but constant number of certificate changes and stuff--

AUDIENCE: Question.

ERIK DEMAINE: --log n time. I update the priority queue. Question?

AUDIENCE: How is it three?

ERIK DEMAINE: Why is it three? x is involved in a certificate with its two children, and with its parent. Its parent has a relation, as well. Yeah. So you have to be careful in counting locality. How many certificates in each object in?

OK so we're responsive-- log n time, local, constant, certificates per object, compact, linear number certificates-- all these are the same. Big issue is about efficiency. How many events is

it, in the worst case? How many events do I need?

OK, here, for kinetic heap, where I want to maintain the minimum at all times, there's actually a very clear lower bound. The number of events-- or let's say, the number of changes to the min-- I think this is what some people call external events. Like, you can't control when the min changes. The user controls that by how they set up the trajectories. Changes to the min is at least n in the worst case.

It's pretty easy to set this up. You have a point, another point moving at constant velocity to the left, it will overtake. You have another point farther away, with a bigger velocity, to the left. It will overtake after this one overtakes-- and, you know, something like this. You've got to make sure they don't all cross at the same moment. But this one will go first, and then this guy will cross over. And eventually--

So it's easy to set up. The min can change a linear number of times. I claim in this data structure, well, of course, the min will change, at most, a linear number of times. Well, it's not clear it's at most. But that's true.

What we care about though, is, we're storing way more certificates than the min. We have a linear number of certificates. I claim the total number of events in this data structure, if there are no change operations, if we just advance to infinity-- number of events is order n log n. And so we call the efficiency log n, because you're log in factor away from this lower bound. This is the interesting part. And this requires a proof.

So why is the number of events order n log n? We're going to prove this by amortization, with a somewhat tedious potential function. But it's sort of the obvious one.

At a time t, I wanted to find my potential at time t to be, how many events will happen in the future? This is the thing I want to bound. So I want to prove this is order n log n at time zero. But let's think about how it changes over time.

So I'm going to rewrite this as follows. I'm going to Sum over all items x. And we're thinking about the potential at time t. So I look-- x is in some node. And which node it's in changes over time. But at time t, it's in some node.

I look at all the descendants of that node. I look at those items. And I see which of those items will overtake x in a future time. Those are the ones that I care about. When you overtake an ancestor, that's when an event happens.

I claim these two things are equal. I probably don't even need to prove that. So this is in parentheses. Don't worry about this being equal. Think of this as the potential function.

Now, I want to look at an event and see how this changes. I need a little bit more notation. I'm going to call this thing phi of t comma x. So phi of t is the sum over all x of this thing, which is phi of t comma x-- number of descendants of x at this time that will, in the future, overtake x, meaning their key will get larger than x.

OK. I can also expand phi of t, x in a simple way. It's kind of trivial. But if I look at each child of x-- there's two of them-- call that y-- and then measure how many descendants of y at time t will overtake x? So it's almost the same words, but I changed one of the x's to a y-- but only one of them. Sorry-- "future" I write greater than "t."

This is a different quantity-- slightly different-- which I'm going to call phi of t comma x comma y. I just need to be able to talk about this. So this is mostly to introduce notation.

OK, and descendants of y-- this is including y itself. So there's y and all of its children-- descendants-- whatever-- which are those that will overtake x. If I add that up for both children, that is the total number of descendants of x that will overtake x. So this equality is trivial. Mainly, I wanted to introduce that.

OK. So what now? Let's look at an event, and see what changes.

So-- try a little bigger-- x, y, z. Let's say we need to change it to y, x, z. Because y is about to overtake x. How does the potential change?

OK, well, potential is the sum of all these phi t, x's. So which of these phi t, x's change? I claim that phi of-- I'm going to call it t-plus, which is the moment right after now-- infintesimally larger-- I claim it does not change for most vertices. It only changes for x and y. If x and y are the guys that are switching order, this will not change for any others.

Why is that? Because we're looking at number of descendants to some vertex that will overtake that vertex. So do the descendants change? No. Descendants of x and y change, but no one else, their descendants change. This is just a swap of two adjacent elements.

Will overtaking change? No. I mean, overtaking doesn't change. If you're going to be overtaken by x before, you still be overtaken by x. And it's still a descendant-- unless it was y.

y is the only one for which it changed. OK, so we only have to think about x and y and how their potentials are changing. OK.

So let's look at x. x is pretty simple. x went down. So we care about now the descendants of x. Those are what used to be the descendants of y.

So what are the descendants of y that will overtake x? That is phi of t, x, y. That's the definition here. The descendants of y that will overtake x-- we had that written down. So that is its new potential, except as a minus 1, because y used to be a descendant that would overtake x. But it just overtook x-- not going to happen again. Not gonna happen again.

I'm assuming here affine motion, I suppose. It could really happen a constant number of times. But let's keep it simple. OK. That was easy. Next one's a little harder.

The other one is y. What is the new potential of y? y has new descendants now. It used to just have this many descendants. It still has those descendants. It now has x, which doesn't matter, if we assume it will never be overtaken again. And now, it has this whole subtree of z. So we have whatever it used to have, which is phi of t, y. That's all it's old descendants. And now we have this new thing phi of t, y, z.

If you plug that into here, that is the number of descendants of z that will overtake y. That's the new thing that we add on. So this is an increase. This was a decrease and this is an increase. We hope that they're about the same. In fact, they will be basically equal. So first claim is this is, at most, this other thing-- phi of t, y-- phi of t, x, z. So I'm just replacing this y with an x.

Why is this true? Because this is overtaking y. If you're going to overtake y at this time, you would overtake x at this time. Because x and y are equal right now. I'm pretty sure this is actually equal. I don't know why I wrote my notes "less than or equal to." Less than or equal to is all I need. But I think, at this moment in time, when they're actually equal to each other, it doesn't matter whether you have x and y here. You will overtake one if and only if you overtake the other at this moment.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: No, I see. Right. Actually, this is about future time. In future time, we know that y is always going to be less than x, because it's moving up. In future time, if you overtake y, you'll certainly overtake x, because-- no, the other way around. If you overtake x, meaning you go below x, one of these ways-- overtaking is actually going up in the tree, but it's actually getting smaller

in value.

If you get smaller than y, you rise above y. That means you will have already risen above x. You already be a value smaller than x. So hopefully, that is the way that it's less than or equal to. Because that's what we need. OK, good. Now, I understand.

AUDIENCE: So at this point, we're assuming affine?

ERIK DEMAINE: We're assuming affine. Yeah. Sorry. Forgot to mention that. I think this works for pseudo algebraic, but this proof does not, assuming that x and y only switch places once.

OK, then we have this simple equation, which is phi of t, x equals phi of t, x, y plus phi of t, x, z, where you just sum over the two children. So this is phi of t, x, z. So-- I need more space, I guess.

So phi of t, x, z, that is phi of t, x minus phi of t, x, y. So we have, let's say, on the left-hand side, is less than or equal to phi of t, y plus-- this is what we had before-- t of x, z. But now, that is the same as phi of t, x minus the other child. Sorry-- y. OK, that was just that equation.

Now, do things simplify? Hope so. Phi of t, y-- these two things-- OK. Not so pretty. What we care about is the sum of these things. We care about sum-- the overall potential.

So we want phi of t-plus is phi of t plus something. How it changes is, well, how does phi of t prime x change? Well, it went down a lot. It went all the way down to phi of t, x, y. Well, actually, we just add these together. Add them up. This is messy. Sorry.

Let's look at this sum-- phi of t plus x and y. So we get this thing-- phi of t, x, y minus 1. And then we get this thing-- so phi of t, y, plus phi of t, x, minus phi of t, x, y. It's an elaborate way of saying this cancels with this.

So we are left with the old value-- phi of t, x, plus phi of t, y, minus 1. So that means phi-- sorry, this is less than or equal to. This means phi of t-plus is less than or equal phi of t minus 1. In other words, every time an event happens, the potential goes down by at least one. This is basically confirming this intuition, but roughly.

But our definition was this-- the number of descendants of x that overtake x in the future. How big could this thing be? It is, at most, the sum over all nodes of the number of descendants of that node. So it is, by definition, at most, n log n. If we look at phi of 0, it's order n log n.

And what we're doing is using a balanced heap here. And so for order n log n, initially, in every event that happens, we go down by one. And we're never negative. That thing is always non-negative. Then the number of events is, at most, order n log n. Cool? OK, I only have one more page of notes to cover in zero minutes-- about 15 seconds.

Let me quickly tell you about what's on this page, and you can read it at home. It's a little survey of lots of different kinetic data structures, in particular, for more than one dimension. So there's a lot of work on 2D convex hull. You have moving points in two dimensions. You want to maintain the convex hull.

The number of events-- the best we know how to achieve is order n to the two plus epsilon-- a little bit bigger than n squared. And there's a lower bound that it can change n squared times. So it's almost optimal-- unknown how to do that in 3D.

A problem that we solved in the open problems in this class two years ago is smallest enclosing disk. So you have points moving in the plane, and you want to maintain, what is the smallest enclosing disk of those points? Number of events we got was n to the 3 plus epsilon, which is a little bit painful. Best lower round is n squared-- so still a gap there.

Those closely related to another open problem, which is, you want to maintain something called a Delaunay triangulation. If you know what that is, great. If not, it's a big open problem-- how many times can the Delaunay triangulation change if you have just points moving along straight lines? Best upper bound is n cubed. Best lower bound is n squared-- so similar linear gap.

If you just want to maintain some triangulation on your points, you can do better. Best upper bound is n to the 2.33333. Best lower bound is n squared. So a smaller gap-- just cube root of n gap.

Collision detection is probably the most obvious application here. You have a video game. Things are moving with known velocities. You update those velocities-- like, you have bullets, and people running around, and whatever you want to know when they bounce into each other-- walls, which are stationary. When does that happen? There's pretty good algorithms for kinetic collision detection. Although, it's very unclear what efficient means.

Because you want to optimize for the case when not many collisions happen. And these algorithms do-- in a certain sense, they're optimal. But it's much harder to state what that

sense is. Minimum spanning tree is another one I have. This is tough. Easy thing to do is, you do kinetic predecessor on the entire sorted order of the edge lengths. And then you just run a [? stem ?] thing that processes the edges in order. So that gets a quadratic number of events. It's unknown whether you can do any better for minimum spanning tree.

So that was your quick survey of kinetic. And that ends geometry. The end.