MITOCW | watch?v=WqCWghETNDc

ERIK DEMAINE: All right, welcome back to 6.851. We continue our theme of time travel today. Before I get started, I wanted to briefly talk about the project, which I forgot to talk about in lecture one. Project is sort of the focus of your evaluation in the class, other than problems sets inscribing, it's supposed to be the bulk of the work. Project can be theoretical, trying to solve an open problem. You don't have to succeed. If you want to do that, you should come to the open problem sessions.

Could be a survey of things, especially not covered in class, or not covered in class in much detail. Look at a few papers. Could be an implementation of one of these data structures. A lot of the data structures we're talking about here have never been implemented. We don't know how well they perform in practice, and it'd cool to find out. Many of them have though, so you should do one that hasn't been done already. Those are the main project types.

There will in a month or so, there will be a project proposal deadline where you have to pick what you're doing. And then the project itself is due last day of classes. And in the last few lectures, there will be project presentations. So you'll also have to give a talk about it.

You can do group projects. And for more people we expect more stuff to be done, that's all. Otherwise unconstrained.

So in time travel land, I guess the most sort of physically realistic physics land, if time travel is branching universes where if you make a change in the past you get a totally new universe. That's full persistence. That's last class.

Today we're going to do the more science fiction-y approach to time travel, which is you can-- like in *Back to the Future.* You can go back in time, make a change, and then return to the present and see all the intervening-- the result of all the intervening things that happen meanwhile. A fun example of this, if you haven't seen it, is the movie *Time Cop,* where there's this guy going back in time, trying to fight evil time travelers. And when he returns to the present, sometimes, like the whole time travel service is starting to fail. And there's this worry that he'll never be able to return to the present.

So anyway, we want to do that with the data structure. So this is the idea of retroactivity. In general, we think of a data structure as having a timeline.

We're going to maintain one linear timeline of updates. So here's time. And the idea is, well, maybe at time 0 you do an insert of 7. And then at time 1, you maybe did an insert of 3. And then at time 2, you did a delete-min.

OK, what's that spell? Or, at the end you have-- right now, you'd have just the element 7. In general, every data structure you can think of is this. Now normally, we're always appending to the timeline that's working in the present. You're doing operations on the present. But we'd like to allow-- with retroactive data structures you can go back in time, say right at the beginning of time, and say actually I meant to put, at time minus 1, I meant to do something insert 4. Or actually, insert 2 would be a little more interesting.

Insert 2 is interesting, because that actually changes what the delete-min operation would have done. And so I want to go back in time, do this, and then return to the present, say what's my data structure look like now. I want to be able to do the regular queries I can.

So let's say you can do queries at the present. This is now. And you can do queries here. And I want to be able to add and remove operations. Update operations at any time.

So that's what's called a partially retroactive data structure. So like persistence, we have partial and full. So in general, we're going to have insert at a given time, a given update. Delete the operation at a given time. And query.

And I'm going to use capital letters to denote retroactive operations and lower case letters to denote regular operations. So it's intentional. This is an uppercase insert. This is a lowercase insert. So we're inserting an insert at time, the time before 0.

What exactly does time mean here? We're not going to work with that too much directly today. You can think of them as just being integers. But it gets awkward if you just keep inserting new operations to do in between 1 and 2. There aren't very many integers between 1 and 2.

One easy solution to this is use the order maintenance data structure that we mentioned in the last class, where you basically are maintaining a linked list and you're able to insert new times in between any two given times. And you can still do order queries in constant time. But I mean, if you don't mind logs you could just store the times in a binary search tree. That's an easy way to do it. We're not really going to worry about that too much here.

What we're interested here is how do we maintain the sort of side effects of doing this

operation earlier in time, computing all of this, or chain reactions that would have happened meanwhile, sort of butterfly effect.

All right, I want to define partial retroactivity means queries are always done at the present. Let's say t equals infinity or now. This is what you might call the Q model, if you're a Star Trek fan. You can exist in all time simultaneously, do updates whenever you feel like it, do queries whenever you feel like it. All right model clear.

Now as you might expect, this is hard. You can't always do retroactivity super well. But we'll talk about our situations you can do well and what you can do in general.

Go to an easy case, which is still fairly interesting. Suppose you have commutative updates, meaning x followed by y is the same thing as y followed by x. This would be great if I could reorder updates, because then I don't really care about the sequence of updates, as long as I'm only querying the final results.

If I want to do an update in the past-- this would not be an example of commutative-- delete-min and insert are not commutative. But if I was just doing insert and delete for example, delete 7, whatever, then inserting 2 here is the same thing as inserting 2 at the end, as long as I'm only querying the end. So this would be an easy case for partial retroactivity.

If my updates are commutative, then inserting an operation at time t is the same thing as inserting that operation now, at the end of time. That's for inserts. But if I want to also be able to-- for retroactive inserts, I should say, capital I.

If I want to be able to do retroactive deletes, I need something else. I need some way-- I still want to only work in the present, if I can. And for that I need inversion. I have invertible updates.

If there is some way to, given an operation x, compute a new operation x inverse, that does the reverse, and that's equivalent to nothing, then deleting an operation is the same thing as doing the inverse. And if you have also commutativity-- so I'm adding these two constraints-- then delete t some operation is the same thing as inserting now that operation inverse.

So that's-- if I have both of these properties, then partial retroactivity is trivial. I just do everything in the present. May not seem very exciting, but it solves a bunch of problems already for free. So for example, hashing. If you want a dynamic hash table or dictionary, and

you want to be able to make retroactive changes, it really doesn't matter whether you make them in the past or in the present.

I'm assuming a little bit here that you don't, for example, clobber a key and write to the same key twice. Then the order would matter. But if you store all copies of all keys, then this is true.

Slightly more interesting, if you have an array with the operation add delta to a given array element. So if I had an assignment here, then the order would matter. But if I have only addition or subtraction, that's fine, because addition and subtraction are commutative. So I can do that. Not terribly exciting, as you might imagine. We're going to get to much more interesting examples.

But a natural question here is what about full retroactivity? So partial's nice. But if I have such an easy setup, can I achieve full retroactivity? It's not so obvious because now working in the present, it's not correctly reflecting the past. To do that, we need-- we cannot do it in general, we don't know how.

But there is a situation where we do know how. First I'll define a search problem. The search problem is-- you've seen many examples of this in your algorithmic lifetime-- I maintain a set S of objects-- we don't know what they are-- subject to insert, delete, and some kind of search query. And I want to focus in on the types of queries we allow.

So this in particular, insert and delete are commutative and invertible. They are each other's inverses. So n queries are queries. This is really just a constraint on updates. So search problems fall into this easy class. So in particular, it covers hashing.

Fine, that's a general class of problems. Here's a slightly less general class of problems, which are sort of well-studied. The old geometric data structures literature. Decomposable search problems. This is a concept from 1980 revitalized.

I should mention this retroactive paper is pretty recent. I think the journal version appeared in 2007. I have 2003 and 2007 here. So those are the two versions of the paper.

So what's a decomposable search problem? Same thing, but the query is of a particular flavor, has a particular structure. If I want to do a query on the union of two sets, a and b, it's the same thing as doing the query on the two sets individually and then combining them with some function.

This is a constraint on what the query is. The problem is still this. I still want to do a query on the entire set. But I just know this fact, that if I could somehow solve the query on smaller sets, I could combine them into the union using this f function. For some, let's say this can be computed in constant time.

That's a decomposable search problem. For example, suppose I have a set of points in the plane. And I want to know, given a query point, what's the nearest point in my set? So it would be dynamic nearest neighbor.

If I could compute the nearest neighbor among one subset of the points, and the nearest neighbor among a different subset, I can compute the nearest neighbor among the union just by taking the min of the two. Doesn't matter whether a and b overlap. That will still give the right answer. Sort of a min over all things. I have some other examples.

Successor on a line. If I'm given a query point, I want to know what the next, the nearest item after it is. That's sort of like one-sided nearest neighbor. Yeah, point location, we'll get to that in the future. Don't want to time travel too much.

OK, so some property-- some queries have this property. Some don't. If they do-- and we have commutative and invertability-- I claim full retroactivity becomes easy. Relatively easy. Log factor overhead. So that's what I wanted to show you.

Full retro for decomposable search problems. Decomposable search problems are automatically computed of an invertible, so I don't need to write that constraint. In log-- I have n here, but I think I mean m-- overhead. It's a multiplicative factor. m here is the number of retroactive operations you're doing. So it's the size of the timeline.

OK, we're going to do that via something called a segment tree, which was introduced in the same paper that dealt with decomposable search problems by Bentley and Saxe in 1980. They weren't thinking about time travel. Retroactively their result is useful.

So what's a segment tree? It's a tree. Let me draw my favorite picture. Draw this many times in this class, I'm sure.

I would like to build a balanced binary search tree on time. Leaves will be time. Here's time.

And the idea is, well, if I have my operations, my updates-- which are what the things that live on time are-- are inserts and deletes. And so if I look at an element, at some point it might be

inserted. At some point later it might be deleted. I'm going to assume every element is inserted and deleted only once. Otherwise, if it happens multiple times, think of them as different elements.

Then every element exists for an interval of time. So for example, maybe I have an element that exists for this interval of time. Maybe it gets inserted here and then gets deleted here. How should I represent that interval in my tree? Let's call this element a.

Well, if I draw a complete binary tree, there there's an obvious way to represent that interval, which is as the union of these four intervals. If you think of this node as representing the interval of all of its leaves, descendant leaves, then I could put a here. And I could put a here, and here, and here. I do that for all my elements.

This picture's going to get messy quickly. Let me just do one more example. So maybe I have an element from here to here. Then I'm going to call that element b. Then I'll put b here. And I'll put b here. So always putting it is as high as I can.

I don't want to put b or a in all the leaves it appears in, because that would be a linear number of leaves. If I do it like this, there's only log n subtrees I care about, representing the interval as a union of log n subtree aligned intervals.

OK, now the real picture is slightly messier than this. I drew a nice, complete, perfect tree. In reality, you're inserting new operations in here. And so you've actually got to maintain this as a balanced binary search tree. So use a red-black tree, AVL tree, or pick your favorite. As long as it stays balanced, you can still do this. It'll be a little less obvious how to do rotations, but let's not worry about rotations right now.

Now I want to do a query. What's a query look like? Well, I care about all the things done up to a certain time. Or really I care about who exists. Let's say I want to do a query on an array here. That's maybe not so exciting. Let's do a query here.

I want to know at this time who exists, and perform the query on that time. So really, I'm asking about everything done in this interval, from the beginning of time to my query time t. So here's query of t.

Now we don't know what the query is. We just know it's on the set. And we know that is decomposable in this way by a set union. So the idea is, well I'll do the query on this data structure, because that represents whatever existed at that time. And then I'll do the query on

this data structure, because that represents the things that existed before that. That doesn't look so great.

OK, so I need to put each element in log n different nodes in here, such that I can answer a query by looking at log n different nodes. And a query is asking about what are all the things that exist at this time. So here it would be ab. Over here it would be nothing. Over here it would be nothing. Yeah?

AUDIENCE: If you're querying, the time correspond to particular [INAUDIBLE]. Then you can query all the data structures--

ERIK DEMAINE: OK, maybe I was just doing the queries wrong. Good, I think that's a good idea. I guess I'm asking about querying this time. So which node should I do? All the ancestors. I see. Claim that does it. Gotta think about it.

AUDIENCE: And then you carry [INAUDIBLE]

ERIK DEMAINE: That definitely gets a and b. That looks good. All right, exercise to prove that this works. I think that is right. So it's an asymmetry between queries and updates. For updates, you basically partition the interval.

AUDIENCE: What happens to a very short-lived element at the very end of the time travel? [INAUDIBLE]

ERIK DEMAINE: So maybe this works. I'll think about it offline and I'll send email if it doesn't work. But I think this is right.

So for query, you walk up time. And I'm going to query each of these data structures. Now this is a logarithmic factor overhead, both in time and space, because each element might be stored in log n different structures.

Or what are in these nodes, by the way? That is your data structure, the non-retroactive version. So we're storing the non-radioactive data structure on these guys in order to-- when I add-- if I modify a's interval, I delete it from all the places that used to be in and then reinsert it into all the places it's now in. So I'm doing inserts and deletes on log n of those non-retroactive data structures.

OK, and then I combine with f which is constant time. So only log factor overhead on the queries. OK, a little harder than I thought.

So full retro, full retroactivity, but in this relatively simple special case. Let's talk about the general case.

There's an obvious way to do full retroactivity, which is-- let's say partial retroactivity. I want to know what happened in the present. The obvious thing to do is write down all the changes you made. And then if you want to make a retroactive change, you roll back, do your change, and then replay everything that happened meanwhile. So this is the rollback method.

So if I need to go back in time by a factor of r, by r time units, I pay a factor of r overhead in time. And space is reasonable. You just have to remember everything that happened. This is pretty obvious. Also old. Unfortunately, it's about the best we can do in general, depending on your exact model.

So we add the lower bound.

This is kind of a fun result from a philosophy standpoint, philosophy of time travel. It's essentially saying that *Back to the Future* is impossible. I know, it's sad but true. And there's a good movie about this called *Retroactive.* This is a very unknown movie, from '97, or so, where they play through this sequence of events. And then there's a time machine which brings them, I forget, an hour into the past. And then they have to relive those events. And they can make changes.

But they have to spend all that time until they get to the time machine again. They still didn't get it right, so they go back in time. And the movie repeats like six times or so, until they finally get it right. Or I shouldn't, whatever, forget the ending actually.

So that's realistic in that we know from persistence, we can remember the past. And we can jump back into the past and then just relive it. And maybe make changes along the way. But the claim is you can't do that any more efficiently than replaying all of those events, in the worst case, and I think even in real life. Should include the worst case. So let me tell you why. All we need to live in our universe for this lower bound to hold is a very simple computer.

It's a computer with two registers, x and y. They start at zero.

AUDIENCE: Question.

ERIK DEMAINE: Yeah?

AUDIENCE: When you say that the overhead can be necessary, does that mean that it's sometimes not--

ERIK DEMAINE: The can be is relative to the data structure. So there is a data structure where you're guaranteed to need order r in the worst case. And probably the average case in all sorts of bad things. Yeah, so there are some data structures, like these ones, where you can do better. And this is for partial retroactivity. But then there are some data structures, like the one I'm going to describe to you, that you cannot do better.

So here are my operations. I can set x to a value. I can add a value to y. I can compute the product of x and y and put it into y. And then I have a query, which is what is y. So this is one of the simplest computers ever designed. A very basic calculator. But it lets us do--

Here's what the operations I'm going to do. First operation-- this is in order of time. First I will add a into y. Then I will compute the product. Then I will compute-- then I will add a and minus 1 to y. Then I will compute the product. And then at the end, I will add a zero to y. What does that do? Mathematicians.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Polynomial evaluation. This is equivalent to computing a n, x to the n-- well, I guess capital X to the n, a n minus 1, x to the n minus 1, a0. OK, I didn't set X at the beginning. But whatever X was, this sequence of operations will evaluate this polynomial. This is called Cramer's rule for evaluating that polynomial. Just keep multiplying by X in each round, adding on the next coefficient.

OK, so the result-- this I'm just going to do at the beginning. It's like regular operations. And now I'm going to do things like insert at time 0 that X equals some value. I'm just going to keep-- first I'll put it here, then right after, then right after-- keep doing those insertions. I completely change what X is. And so I change where I'm evaluating this polynomial.

And claim is you really can't do that any faster than reevaluating the polynomial. In a history independent algebraic decision tree model-- which I don't want to define precisely, but maybe computation tree would be better.

Anyway, this is just you draw a tree of all the possible things you could do. And the things you can do are additions, multiplications, subtractions, divisions, all field operations. So for any field, this would work with reals, would also, should work for integers module LP. Any fields. And the interesting part is independent of preprocessing of the coefficients, you can stare at

those coefficients however long you like. In the worst case, you need mega n field operations, where n is the degree of the polynomial.

So this is a known result from 2001. And so it tells you that each of these retroactive operations, it's n time units in the past. And you need order n time to complete it. Even though these operations can be solved in constant time each. So it's a factor n loss.

Now this is a particular model of computation. It implies something like an integer or a random access machine, where you can have arrays, I don't really care. But as long as the way that you're manipulating these objects are with field operations, you're in trouble.

OK, but there are conceivably other things you could do. If these numbers are integers maybe I could split their bits into parts and do crazy things with them. That's something we'll get to later on in the class. Even then, you can't do so well. I guess I'll briefly tell you this because there's an interesting lower balance. Oh sorry, interesting open problem.

Cell-probe model, something we'll get into in a deeper way later on in this class. But it just says how many objects, how many words of memory do you need to read in order to solve your problem? So that's a lower bound on time. No matter how you're computing with those numbers, how many of the objects, how many of the integers do you need to read?

Not necessarily just the ai's You could split them into parts, do all kinds of preprocessing, make a hash table, build whatever data structure you want. How many words of that data structure do you need to read? And the claim is at least square root of r divided by log r. So roughly square root of r, open problem, is can you get something like r again in this model?

So this is a very powerful model, not at all realistic. But if you can prove a lower bound in this model, then it holds basically all models you could ever think of.

So here's a data structure-- or here's the proof of this lower bound. Data structure maintains n words. I should say typically in this class, a word is an integer of let's say log n-- I'll be more precise. w bits. w is at least log n. We'll talk about this a lot more later in the class when we get to integer data structures. But if you were wondering what a word means, there it is.

And the operations, the updates you are allowed to do are arithmetic operations. So it's generalization of this problem. You can do addition, multiplication, division, just basic stuff. So that's the data structure we're trying to do. And at the end you can query what is the value of

this word.

And now what are we going to do? Before we did polynomial evaluation in that model. Now what I'm going to do is compute FFT, fast Fourier transform. If you know how to do that great. If not, it's an n log n algorithm. So it's an n log in operations you do. I'm just going to write them down as my sequence of updates. At the end I have the fast Fourier transform.

And the retroactive thing I'm going to do is set all of these words, all the xi's, and name them. Which changes the input to the fast Fourier transform. And it turns out that dynamic fast Fourier transform, where changing one of the words, xi, you can show, or it has been shown-- we're not going to prove this here-- requires at least square root of n cell-probes.

So this is about-- if you change one thing in a fast Fourier transform, it doesn't change everything, necessarily. But the claim is that you need to look at least root n of the things again, in order to figure out what happened, how the FFT changed. So I mention this mostly as an interesting question of whether you can come up with a better lower bound in this data structure. Ideally taking some existing lower bound, plugging it in to this kind of retroactive framework, and getting what you want.

Enough about the bad news. I want to go to positive results again. So we had a very simple thing here. Let's try something that's somewhere in between, where there are chain reactions, but it's not impossible. So in general, if you're building-- if you have a data structure that represents an arbitrary computer, you're basically hosed. If you have a data structure where the operations can be arbitrarily reordered, pretty easy. So our intermediate setting will be priority queues, the example I showed at the very beginning.

So on a priority queue, I'm going to think about two operations, insert and delete-min. This is interesting because delete-min depends on what the current state is. So if I insert something early on, it can change every delete-min that happens thereafter, and completely change what happens at the end of the data structure.

So the claim is we can solve this with log n time per operation, per partially retroactive operation. It's pretty good because in this model, which-- what model are we in here? I guess pointer machine would be a reasonable model? Log n is the best you can hope for a priority queue. I guess comparison model is more relevant then.

So you can use priority queues to sort. So log n is the best you can do for regular priority

queues. And so to do it also partially retroactive is pretty interesting.

How do we do this? Well, I don't want to draw a picture of what these data structures look like. My picture is keyspace is going to be the y-axis, and time is the x-axis. So if someone comes along and does an insertion, they do the insertion at some time with some key value. And so I'm going to think of that as a rightward pointing ray, because at that point that element exists. And if nothing happens, it will exist for the end of time.

The problem would be if someone comes along and does a delete-min. A delete-min is going to be a ray shooting up from infinity, because you always want to delete the smallest possible thing. So whatever this ray hits first, that is the thing that gets deleted.

So I want to keep-- I want to draw all those rays for the delete-mins. So this is a delete-min. And this is an insert. So I have rightward rays for insertions. They start at some point. And I have upward rays from infinity. In general, I'm going to get a bunch of L shapes. And the L's will be non crossing in a legitimate picture of an execution of the algorithm.

AUDIENCE: The vertical stuff is when you're taking the min out?

ERIK DEMAINE: Yeah the vertical lines are delete-mins. They're coming from below. They're taking whatever the smallest thing is that exists. So think about like this time right here. Right now these four elements are in the data structure. You do a delete-min, you delete the smallest one. So this guy dies. But the three others continue.

Maybe there'll be some elements that just exist. They don't get deleted. So it's not entirely L's. That's sort of a half of an L. So you might have some L's like that that just go off to infinity. But everyone that's in an L, this is the insertion that made it. These are the delete-mins that deleted it. We're not going to do arbitrary deletes, just delete-mins. It's hard enough. OK, problem clear?

So I want to maintain this. Let me show you what goes wrong, the hard part about this data structure. Let's say I do an insert here. New insertion. Well, that means that this delete-min would have returned this guy instead of this one. So this one would actually continue to exist.

So then this delete-min would have deleted this guy instead of that one. So this guy continues to exist. So this continues to the right. This starts exist until this delete-min, and now this one exists. OK, and then this one exists for the rest of time. In general, you do a single retroactive insert, everything could change. Or a linear number of guys could change.

So if you want to maintain this picture, you're in trouble. You get an omega r lower bound. So we're not going to maintain this picture. But we're going to keep it in our minds, vaguely, that we wanted to know something like this picture.

What we really need to know is that this element now exists in the data structure. If I'm just doing partial retroactivity, what I care about is who exists at the end of time. So I'm doing an insert at a given time t, with a given key x. And if life was good, you know, if I was commutative, then that would be equivalent to inserting that key x at the end of time. But it's not.

Instead I have to figure out what key gets inserted at the end when I insert this key at this time. And similarly for adding a delete-min. I won't try to draw that, but exactly the same kind of chain reactions can happen. OK, so we're going to do this in log n time.

How do we do it? So I'm going to focus on the insertion case, inserting-- the double insertion case-- inserting an insert operation. I guess I'm going to switch notation and call the keys k, instead of x. It doesn't matter.

OK, what I'd like to know is that when I do this insertion, what gets inserted into the final Q? I'm going to call that Q now, being the the present, the future, whatever you want to call it.

I claim this is what gets inserted.

If you look at all the keys that get deleted from now on, and you look at the key you're inserting, you take the max of all those, that will be the one that survives. That's pretty obvious. The annoying thing is this is really hard to think about. Because who gets deleted is something that is a chain reaction effect. What we'd really like to have is the word inserted there, because we know what things get inserted when. Those are the operations. We just maintain the list of operations, we'd know that.

Deleted is a lot harder to think about. So we need a Lemma that tells us-- there's a nice equation that simplifies this formula for us, or this computation. For that we need the notion of a bridge. A very simple idea. A bridge at time t is a time when everything that's currently in the queue will finally be in the queue.

So this is a bridge. This is a bridge. Because these elements exist for the rest of time. This is a bridge. OK, these are basically-- interesting stuff happens between bridges. Bridges are the

points of rest when things are kind of boring. We'll need this for the next statement.

And the other claim is it's easy to maintain where the bridges are. That's not a big surprise. Because there's no chain reactions at the bridges.

So if I have a time t that I care about-- I want to compute this formula for time t-- what I do is find the preceding bridge. Call it t prime.

This is like-- an arbitrary time t like here, we're in the middle of some action. I want to know what that action is. So I want to go to the preceding blue line. So that's t prime. If my query is here, t prime will be there.

Then, what I want to know is this max of k prime where k prime is deleted after time t. I should say greater equal to t, instead of after, a little more precise. Claim this equals the max over all k prime not in the final queue, because those guys obviously still exist. Where k prime is inserted at time greater than or equal to t prime.

So basically I can turn the word deleted into the word inserted, if I change t to t prime. And I also have to exclude the guys at the top. So we're looking at a particular time like here. I want to know who are the things that are deleted after that. Well, this is deleted. This is deleted. All these guys are deleted after.

I'm going to rewind to t prime and say, well, you know there are these guys that they survive. They obviously don't get deleted. But if you ignore the guys that end up in the final queue, the guys that get deleted after time t are the same as the ones that get inserted after t prime. That's the claim.

Or actually, that set of elements is not the same, but the max will be the same. That's all I care about. I only want to know the max. The max among all the ones deleted here is this upper uppermost segment. And that's the same as the max among all the guys that are inserted after here. So it's really just a claim about this one being inserted after this bridge.

I won't prove this Lemma. It's just, you think about it for long enough, it becomes obvious. Leave it at that. Now we can do some data structures, because now we have something that we can reasonably hope to maintain.

So maintaining bridges, one thing-- Yeah, actually kind of an important thing. So here's where we're going to store. We're going to store the current Q as a balanced binary search tree. OK,

that's not very exciting. Though the hard part is to figure out how Q now is changing at each step. But this will allow us to do arbitrary queries on the final result

We want to store a balanced binary search tree whose leaves represent insertions. So lowercase insertions. So that's all of the rightward arrows. And they're are going to be sorted by time. Not by a key value.

OK, and I'm going to augment that balanced binary search tree with a fun value for every node in the balanced binary search tree. I want to know max of all the keys not currently in the queue, or not presently in the queue at the end of time. That changes is very slowly, so this is easy to maintain. Among all the keys that are inserted in the subtree rooted at x.

OK, that's basically this value, which I want to maintain, but split up over a subtree. And so that lets me easily compute this value by taking the max of log n of these values at any time.

OK, then we store another balanced binary search tree where the leaves are all the updates. Again they will be ordered by time. And they will be augmented. In each node, I'm going to store a number which is 0 plus 1 or minus 1. It's going to be 0 for an insert. This is at the leaf level because the leaves correspond to updates.

If that key ends up in the final queue. So those we just ignore. It's going to be plus 1 for any other kind of insert. Inserting a key where k is not in Q now. And it's going to be minus 1 for a delete-min.

I'm going to store this. And we're going to have subtree sums. So this is fun with augmentation.

Why do I do all that? Because if I store all these numbers, it becomes really easy to detect bridges. I claim.

AUDIENCE: Professor?

ERIK DEMAINE: Yeah?

AUDIENCE: So, what we're looking for is basically we know how the cascade went in any given--

ERIK DEMAINE: We want to compute how one of these cascades will end.

AUDIENCE: And so we just always maintain that information?

ERIK DEMAINE: We're always going to maintain what's here. We can't afford to maintain this whole picture because it changes too much. But we want to-- we're given this. We want to know the result of the cascade, without having to do all that stuff in the middle. Yeah, that's our goal. And I claim-- this is a lot of text. But I claim that this information is enough to do that in log n time. So let me try to convince you.

AUDIENCE: So are you storing several binary searches?

ERIK DEMAINE: I'm storing three. I'm storing one that represents the final result. I'm storing one that's just looking at the insertions. I'm storing one that looks at all the updates. I maintain all three, and you can cross-link them all you want.

OK, I claim a bridge is now a prefix of updates summing to zero, according to this measure. Because inserts and deletes, when they all cancel out-- and if we ignore the things that finally end up there, the periods of rest, when no one is active that will eventually get deleted, that's exactly when everyone to the left has been deleted or will survive to the end. So these guys counts as zeros, so we ignore those. Each of these corresponds to plus 1. Each of these corresponds to a minus 1. So these have all canceled out by now. And when this prefix sum, which is everything up to a given point, is zero, then that is a bridge.

What this lets you do is if you have some-- you have this binary search tree. You have some query time t, which corresponds to a leaf. And now I want to know what is the preceding bridge? I want to know t prime, the bridge proceeding time t. Then basically I walk up this tree and find the preceding moments-- I actually have to first walk down to compute all the sums.

So hanging off on the left of this path are all these times in the past. And I say well, if the sum of all of those subtree sums, sum of all the values in those leaves, if that's 0 then we're already at a bridge. If not, I basically compute the preceding point where this sum up to that point is 0. And that should be relatively obvious how to do. It's a regular kind of walking up the tree.

So log n time I can find the preceding bridge. Because I have all these subtrees sums I can do that by a tree walk.

OK, then it just says here, "And then you can compute the change to Qnow." How do we compute the change to Qnow? We have to compute this max of all the keys that are inserted after time t prime.

So this was the tree of all updates. And we have another tree, which is just the tree of insertions. And so if we cross-link time-- so we find t prime, where the prefix sum is zero. We map over to t prime here. And again we get a root to leaf path. And now we look at all of the elements hanging off the right side. Those are all the elements that are inserted after time t prime. And I just take the max of those.

So I guess I didn't write it, but I need to store our subtree-- oh no, I store this. Right, this is what I want. For every subtree, I want to know the max of the things in that subtree, but ignoring the guys that are in Qnow.

I take all those maxes on all those subtrees. And I can find what element gets inserted into Qnow in log n time. Then, of course, I have to update all this information. And I don't want to spend any more time on this, no pun intended. You have to-- then Qnow changes. And so you've got to update all these values and these values. But the claim is that you can do all that in log n time. It's not hard.

OK, that's priority queues. So I know that was a little bit tedious. But this is sort of-- it's not quite the coolest thing we know how to do, but it was the first cool thing we knew how to do in the retroactive world. It's this nice borderline where it looks scary. There are linear-sized chain reactions. Yet we can still deal with them in log n time. So not quite the worst case, but also not easy.

So in general, if you want a retroactive data structure for your favorite data structure, you're going to have to think along these lines, see whether you can maintain something reasonable. Let me tell you briefly about other problems, other data structures that we've looked at, and other people have looked at.

So a simple one is a queue, first in, first out first in, last out. One of those. I've got stacks too, probably. You can do constant time partial retroactivity. Log n time full retroactivity for a deque. I mentioned this last time where you can insert and delete from either end. We can do log n full retroactivity for union-find.

You have a bunch of sets. You want to be able to take their union. And given an element you want to know which set it's in. It's sometimes taught in 6046.

The best thing we know is log m fully retroactive. Priority queue we already talked about. But--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Is there a question? Priority queue, we already did partial retroactivity. But what if you want full retroactivity? This is pretty much an open problem. Best thing we know is square root of m to m's log n. This is via-- we have a general result that says if you have anything that's partially retroactive, you can make it fully retroactive at a huge cost of square root of m factor. So you could take what we have, make it fully retroactive. Full.

AUDIENCE: Is there any difference when you're talking about n [INAUDIBLE]

ERIK DEMAINE: n, I think it's supposed to be the current size of the data structure. I'm not actually sure what that means with a retroactive operation. I'll just make them all m's. I think that's safer. Other question or same one? OK good, thank you.

OK, let me tell you about the most important problem in this field. This is actually what motivated us to define retroactivity, which is a particular problem, retroactive successor. So successor problem is I want to be able to insert and delete keys in one dimension. And given a query key, I want to know what is the next key greater or equal to it. This is something you can do in log n time with a balanced binary search tree. There's lots of other ways to do it, which we will get to later in this class.

But what if I want to do it fully retroactive? Partially retroactive is really easy, because this is a search problem. So I can definitely do log m partial retroactivity. In fact, it is a-- I just use a regular binary search tree, or whatever. In fact, it is a decomposable search problem. So I get full retroactivity, no problem. But I pay a log factor, so I get log squared m, full retroactivity.

And so that's where we left it, back in 2003. But there's a new result, which is that you can actually get log m without the square full retroactivity. This is a complicated result. It's by Goran and Kaplan, 2009. It uses fun techniques like fractional cascading, which is next lecture, and van Emde Boas, which is lecture 11, and various other tools.

So it's a little too advanced to cover now. But it's cool. And it has lots of applications in geometry, which we will be talking about next class. So hold your breath for that, I guess. Yeah. I want to go to one last topic, which is a different kind of retro activity

Nonoblivious retroactivity is introduced by Acar, Blelloch, Tangwongsan in 2007. And it's basically answering the question "What about my queries?" So we have this timeline of updates. And we've done various operations on our data structure, which are the updates.

And we've considered how those can be changed. And then with queries, either we're querying at the end or querying in the middle. But we're always getting sort of instantaneous, as the timeline exists right now, here's what the result of your query is.

But what if I want to do-- I mean, if I really-- normally when I use a data structure, I do some updates. I do some queries. I do a combination of these things.

And to make life worse, when I do a query, the result of that query probably influences what updates I do in the future. So real use, I'll just call that the algorithmic use of a data structure. Results of queries influence updates, what updates are going to do.

OK, and now this gets a little dicey, because what does influence mean? We don't know, because that's the user somehow depends on these results. So nonoblivious retroactivity is trying to deal with this issue. Can we get something reasonable?

So what we'd like to do is say, oh I now add an update over here, retroactively. So their updates are the same, in some sense. You can retroactively insert and delete updates and queries now. I'd like to know well, did the result of this query change? Maybe you can show-- at the data structure level we can tell did that query result change. Maybe this one changed, but this one did not.

What we'd like to report to the user is what is the first query that changed, because that's like the first mistake that you made. Like oh you know, retroactively oops I should have deposited $100 back here. And then these queries which we're checking whether the balance was positive or negative, some of their results might change, some of them might not.

If they change, then the algorithm has to-- I mean, it doesn't have to rerun these operations, because that's what retroactivity buys for you. But it has to say, oh well, now this query has changed. And then you have to rerun the algorithm from that point.

But not necessarily entirely. I mean, you look at that and say, oh, did that query actually change anything? This is now the algorithm. Oh yeah, instead of this update here, I would have made a different update u prime. Well, that's good, we know how to do that. That's a retroactive update. So I can delete this operation retroactively and insert this one.

So in general, what we assume the algorithm does is it looks at the first error, which is reported by our data structure. And then it then may make various changes from that point onward, but always monotonically left to right, as if it was rerun in the algorithm, but just

changing what needed to change, until all the errors are resolved. All the queries have been corrected to the right answers. Then it can do whatever it wants. Then it can do retroactive operation anytime in the past.

OK, so this is an assumption. Assume once there's an error, that the algorithm makes changes, retroactive updates, from left to right. And also going to assume that it does it at times less than or equal to all errors.

So in general, you have a timeline. And then there are some errors where bad queries were done. Naturally, you would go right here and fix the first error, because that may influence other errors. And maybe you have-- so then you make that change. Maybe then that causes some other changes right after or somewhere in between. But then I don't want to make a change here, because that would be depending on this result, which was incorrect. So first you have to fix this. So you keep going to the right and you visit all the stars. That's this assumption.

This is a very different kind of retroactivity. I would say it's more about maintaining this exact picture of what's happening, but being able to teleport over the uninteresting stuff. So in this picture, if we're doing priority queues like this, when I make this change, well, that query changes. That query changes.

The delete-mins, the mins that they're deleting change. So here they were just updates. But if there's also a query here, which is what is the min, then all of these query values would change. So you're forced now to pay for that. So in some sense, it gets easier. Let's say.

For a lot of problems this kind of retroactivity is easier. But for many problems, it's also more useful. So their example is what if you want to do a dynamic Dijkstra? So you have Dijkstra's algorithm, and then you say, oh, actually this edge weight was wrong. So you run Dijkstra's algorithm once. You've got a priority queue in there telling you which vertex is next. And now you say, OK, I changed this weight at the beginning of time. I don't want to have to rerun Dijkstra entirely if nothing changes. Maybe that weight was completely irrelevant.

If you just do a retroactive update and see did any of the query results change, if they didn't change, then great you have the right answer. If they did change, the results change. And so you have to know for every wrong query that oh, I really had to change the shortest pathway. It's not this, it's now this. But hopefully, you will only have to change a very small amount if

your graph didn't change in a big way.

So it's hard to proof results about that, but at least this would let you dynamize almost any algorithm that's using the data structure. So it's easier, but useful for that kind of transformation.

OK, let me give you an example, which is priority queues. There's a bunch of results in their paper, but for symmetry. Tell you about priority queues. This will look completely different, so don't worry.

Operations-- well, the visual picture is the same, but the data structuring is all completely different. They can do all these things in log m per retroactive operation. Now there's no notion of full and partial. It's just nonoblivious retroactivity. I didn't write it here, but the data structure always maintains what is the earliest error currently.

Maintain the earliest in time error. That error is a query that used to return one result, now returns a new result.

OK we have a similar picture. So I cheated-- I've changed things a little bit. I didn't cheat. It's just a slight discrepancy. For retroactive, regular retroactive, insert and delete-min were more interesting, because then they were chain reactions. In this world you don't have to combine delete and min. You could consider delete separate from min.

Because now, updates are allowed to depend on queries. So maybe you compute a min and then you decide to delete that thing. If the min changes, well then, the thing that got deleted might also change. That's the algorithm's choice. What to do.

So in this new world of insert, delete, and min, slightly more general, if you look at an item, it gets inserted at some time. It might get deleted at some time. Sometimes it might go off to infinity, if it never gets deleted. OK, this is our new picture of the priority queue over time. So again, this is the time axis. And this is the key value. So have some inserts, deletes. They're all corresponding points in here.

And then you have min queries, which maybe I will draw in red. So min query would be something like this. Give me what is the lowest segment at this time.

So that's my sequence of operations is the projection onto time, so there's this thing. Then there's this query. Then there's this insertion. Then there's this deletion. Then there's this

deletion. Then there's this query. Min query, then there's this insertion. Then there's this query. Then there's this deletion. You get the idea. But this is a way to two-dimensionalize that picture.

AUDIENCE: Can you delete without the query?

ERIK DEMAINE: Sorry?

AUDIENCE: Can you delete without the query?

ERIK DEMAINE: You can delete without a query in this model. Make our life a little bit harder.

Now what we need to support are capital insert, and capital delete, retroactive insert and delete, of all three of these operations. Before it was just updates. Now we can insert and delete queries. A little harder to think about. But it can be done.

So for example, let's just-- I'm not going to do all the cases, because there's a lot of them. Let's say I delete this deletion. OK, that means that this will just go to the right. OK, that's bad, because it changes all of these queries. All of these queries are now incorrect. We're just going to keep the picture like this and remember that there are errors. And we'll be able to maintain at every moment what is the next error.

The errors are these crossings. Crossings are bad. That means you have the wrong picture. But we'll wait for the algorithm-- we'll tell the algorithm this is the next crossing. And it's going to have to fix it by say deleting that query and then reinserting it. Then it will get the correct result. But it might do other updates meanwhile. So let's not try to fix them all now because they may change. Maybe they'll decide to re-insert a deletion here to do that, and then erase all of those crossings, if we're lucky.

You might also insert an insertion. That's basically the same as deleting a deletion. You get a whole bunch of crossings. Inserting an insertion and deleting a deletion are about the same. So the other case is inserting a deletion or deleting an insertion.

So inserting a deletion. Let's say that I insert a deletion here. So now suddenly this thing ceases to exist. All of this is gone. We get not a crossing error, but we get what I'll call a floating air, which is that these guys are currently returning something that there's no segment there. OK, so we can get crossing errors and floating errors.

I want to give you enough of this data structure to show that actually you use these data structures to solve it. So in that sense, it's actually not that much easier. I said it was easier, but we're going to need a retroactive successor in order to solve this problem.

OK, some variants.

Man, who used this colored chalk? It's so hard to erase.

Let's say I want to maintain the lowest leftmost crossing.

So in this picture, where I have a bunch of crossings, this is the one I want to maintain, the bottom left one. Now there's a bit of a subtlety here, which is do you do you minimize y-coordinate and then x-coordinate? Or vice versa? It turns out these are the same thing.

I don't want to prove that here, but because of this assumption that we're always proceeding monotonically left to right, we have a nice invariant that if you look at all the crossings, they involve segments that start to the left of all errors. Because all the changes we've made are to the left of all errors by this property. And so it's really just a whole bunch of rays coming in from negative infinity on the left, and a whole bunch of rays coming from minus infinity on the bottom. Sometimes the rays stop. But if you look at where they cross, there's a uniquely defined bottom left corner.

OK, so that's this picture. Those rays might go different amounts from either side, but there is a single lower left corner. So we're going to maintain that.

And we're also going to maintain the leftmost floating point, floating error, on each row separately. So in each horizontal line, like here, we'll maintain that there's a floating error here for this row. But we won't figure out that one till later. That just saves the work we have to do.

OK, now let me show you an operation let's do insert min. So I want to-- I'm going to use x for time here, because it's a little bit more intuitive. X-coordinate is time.

Suppose I want to insert a query. This is actually an easy case because there's no chain reactions here. It's going to be correct because I'm doing the query right now. This is basically a fully retroactive query. I'm adding a new time, a new query, and I want to know what is the first ray that I hit. How do I solve that?

So what's going on here is I'm basically manipulating these line segments. I mean, the red

lines don't really affect this query. They just might be incorrect. But I don't care if they're incorrect. I just want to know for this query, what is the lowest segment?

So really, I have segments. They're changing. Inserting and deleting endpoints of the segments. And I want to know from below, what is the first segment that I hit? This is called upward ray-shooting. Or vertical ray-shooting, I guess, would be the normal phrasing, but I want to be a little more specific. Not downward upward. Among dynamic segments.

This is a well-studied problem. And conveniently, it can be solved in log n time preparation. So you can modify segments and do upward ray-shooting queries in log n type preparation. But you already knew that. Good, you're nodding your head. Because that is retroactive successor, if you think about it. A retroactive insertion operation in the past is like its fully retroactive successor. That's this result.

Inserting an insertion, that's like creating a new left endpoint point of a segment. Inserting a deletion is like creating a new right endpoint. Deleting a deletion or deleting an insertion, you could use that to move the endpoint. So that's exactly dynamic segments, horizontal segments.

And then shooting a ray is-- in general-- a general ray-shooting query is like this. I can be not from minus infinity, be at an arbitrary point, and ask, what do I hit upward next? That is like at this time doing a successor query.

OK, now this is a special kind of successor query, because it's always from minus infinity. So you might be able to solve it easier. But in particular, you can solve it using this result. So regular retroactivity helps you do nonoblivious retroactivity. Cool.

We're out of time. So I will just say, the other cases are similar. There's one other thing you need to do, which is in addition to upward ray-shooting, you have to do rightward ray-shooting because of things like this. If I delete this deletion, I want to know what do I hit next. So that's a rightward ray-shooting among dynamic segments, which all happen to start at minus infinity. So it's again-- it's dynamic ray-shooting, or successor queries, but in x instead of y. So with those tricks, you can do all of the cases of inserting deleting, deleting insertions, inserting, all combinations of those things.