MITOCW | watch?v=T0yzrZL1py0

ERIK DEMAINE: Welcome to 6.851 Advanced Data Structures. I am Erik Demaine. You can call me Erik. We have two TAs, Tom Morgan and Justin Zhang. Tom's back there. Justin is late.

And this class is about all kinds of very cool data structures. You should have already seen basic data structures like balance binary search trees and things like that, log n time to do wherever you want in one dimension. And here we're going to turn all those data structures on their head and consider them in all sorts of different models and additional cool problems.

Today we're going to talk about time travel or temporal data structures, where you're manipulating time as any good time traveler should. Then we'll do geometry where we have higher dimensional data, more than one dimension. Then we'll look at a problem called dynamic optimality, which is, is there one best binary search tree that rules them all? Then we'll look at something called memory hierarchy, which is a way to model more realistic computers which have cache and then more cache and then main memory and then dish and all these different levels. How do you optimize for that?

Hashing is probably the most famous, and most popular, most used data structure in computer science. We'll do a little bit on that. Integers, when you know that your data is integers and not just arbitrary black boxes that you can compare or do whatever, you can do a lot better with integers. You usually beat log n time. Often you can get constant time.

For example, if you want to do priority queues, you can do square root log log n time. That's the best known randomized. Dynamic graphs, you have a graph you want to store, and the edges are being added and maybe deleted, like you're representing a social network. And people are friending and de-friending. You want to maintain some interesting information about that graph.

Strings, you have a piece of text, such as the entire worldwide web. And you want to search for a substring. How do you do that efficiently? It's sort of the Google problem. Or you searching through DNA for patterns, whenever.

And finally succinct data structures, which is all about taking what we normally consider optimal space or n space and reducing it down to the very bare minimum of bits of space. Usually if you want to store something where there's 2 to the n possibilities, you want to get

away with n bits of space, maybe plus square root of n or something very tiny. So that's the sync data structures.

So that's an overview of the entire class. And these are sort of the sections we'll be following. Let me give you a quick administrative overview of what we're doing. Requirements for the class-- I guess, first, attending lecture. Obviously if you don't attend lecture, there'll be videos online. So that's resolvable. But let me know if you're not going to make it.

We're going to have problems sets roughly every week. If you're taking the class for credit, they have a very simple rule of one page in, one page out. This is more constraint on us to write problems that have easy or short answers. You probably need to think about them a little bit before they're transparent, but then easy to write up.

And then scribing lectures-- so we have a scribe for today, I hope. Here? Yes, good. So most of the lectures have already been scribed in some version, and your goal is to revise that scribe notes that if you don't like handwritten notes, which are also online, then easier for people to read.

Let's see. Listeners welcome. We're going to have an open problem session. I really like open problems. I really like solving open problems. So we've done this every time this class has been offered. So if you're interested in also solving open problems, it's optional.

I will organize-- in a couple of weeks, we'll have a weekly open problem session and try to solve all the things that push the frontier of advanced data structures. So in classes, we'll see the state of the art. And then we'll change the state of the art in those sessions.

I think that's it. Any questions about the class before we get into the fun stuff? All right. Let's do some time traveling.

Before I get to time traveling, though, I need to define our model of computation. A theme in this class is that the model of computation you're working with matters. Models matter. And there's lots of different models of computation. We'll see a few of the main ones in this class. And the starting point, and the one we'll be using throughout today, is called a pointer machine. It's an old one from the '80s. And it corresponds to what you might think about if you've done a lot of object-oriented programming, and before that, structure-oriented programming, I guess.

So you have a bunch of nodes. They have some fields in them, a constant number of fields. You can think of these as objects or strucs in c It used to be records back in Pascal days, so a lot of the papers call them records.

You could just have a constant number of fields. You could think of those numbered, labeled. It doesn't really matter because there's only a constant number of them.

Each of the fields could be a pointer to another node, could be a null pointer, or could have some data in it. So I'll just assume that all my data is integers. You can have a pointer to yourself. You can have a pointer over here, whatever you want.

A pointer machine would look something like this. In any moment, this is the state of the pointer machine. So you think this as the memory of your computer storing. And then you have some operations that you're allowed to do. That's the computation part of the model. You can think of this as the memory model.

What you're allowed to do are create nodes. You can say something like, x equals new node. You can, I don't know, look at fields. You can do x equals y.field. You can set fields, x.field equals y.

You can compute on these data, so you can add 5 and 7, do things like that. I'm not going to worry about-- I'll just write et cetera. This is more a model about how everything's organized in memory, not so much about what you're allowed to do to the data items. In this lecture, it won't matter what you're doing to the data items. We never touch them. We just copy them around.

So am I missing anything? Probably. I guess you could destroy nodes if you felt like it. But we won't have to today, because we don't want to throw anything away when you're time traveling. It's too dangerous.

And then the one catch here is, what are x and y? There's going to be one node in this data structure or in your memory called the root node. And you could think of that as that's the thing you always have in your head. This is like your cache, if you will. It's just got a constant number of things, just like any other node.

And x and y are fields of the root. So that sort of ties things down. You're always working relative to the root. But you can look at the data, basically follow this pointer, by looking at the field. You could set one of these pointers-- I think I probably need another operation here, like x equals y.field1, field2, that sort of thing, and maybe the reverse.

But you can manipulate all nodes sort of via the root is the idea. You follow pointers, do whatever. So pretty obvious, slightly annoying to write down formally. But that is pointer machine.

And what we're going to be talking about today in time travel is suppose someone gives me a pointer machine data structure, for example, balanced binary search tree, linked list. A lot of data structures, especially classic data structures, follow pointer machine model. What we'd like to do is transform that data structure or make a new pointer machine data structure that does extra cool things, namely travel through time. So that's what we're going to do.

There's two senses of time travel or temporal data structures that we're going to cover in this class. The one for today is called persistence, where you don't forget anything, like an elephant. And the other one is retroactivity.

Persistence will be today. Retroactivity is next class. Basically, these correspond to two models of time travel. Persistence is the branching universe time travel model, where if you make a change in the past, you get a new universe. You never destroy old universes. Retroactivity is more like *Back to the Future,* when you go back, make a change, and then you can return to the present and see what happened.

This is a lot harder to do. And we'll work on that next class. Persistence is what we will do today. So persistence.

The general idea of persistence is to remember everything-- the general goal, I would say. And by everything, I mean different versions of the data structure. So you're doing data structures in general. We have update operations and query operations. We're mainly concerned about updates here. Every time you do an update, you think of it as taking a version of the data structure and making a new one. And you never want to destroy old versions. So even though an update like an insert or something changes the data structure, we want to remember that past data as well.

And then let's make this reasonable. All data structure operations are relative to a specified version. So an update makes and returns a new version.

So when you do an insert, you specify a version of your data structure and the thing you want to insert. And the output is a new version. So then you could insert into that new version, keep

going, or maybe go back to the old version, modify that.

I haven't said exactly what's allowed here, but this is sort of the general goal. And then there are four levels of persistence that you might want to get.

First level is called partial persistence. This is the easiest to obtain. And in partial persistence, you're only allowed to update the latest version, which means the versions are linearly ordered.

This is the easiest to think about. And time travel can easily get confusing, so start simple. We have a timeline of various versions on it. This is the latest. And what we can do is update that version. We'll get a new version, and then our latest is this one.

What this allows is looking back at the past to an old version and querying that version. So you can still ask questions about the old version, if you want to be able to do a search on any of these data structures. But you can't change them. You can only change the most recent version. So that's nice. It's kind of like time machine on Mac, I guess.

If you've ever seen the movie *Deja Vu,* which is not very common, but it's a good time travel movie, in the first half of the movie, all they can do is look back at the past. Later they discover that actually they have a full persistence model. It takes a while for dramatic effect.

In full persistence, you can update anything you want-- so update any version. and so then the versions form a tree. OK. So in this model, maybe you initially have a nice line of versions. But now if I go back to this version and update it, I branch, get a new version here. And then I might keep modifying that version sometimes. Any of these guys can branch.

So this is why I call it the branching universe model, when you update your branch. So no version ever gets destroyed here. Again, you can query all versions. But now you can also update any version. But you just make a new version. It's a totally new world.

When I update this version, this version knows nothing about all the-- this doesn't know about this future. It's created its own future. There's no way to sort of merge those universes together. It's kind of sad.

That's why we have the third level of persistence, which lets us merge timelines. It's great for lots of fiction out there. If you've seen the old TV show *Sliders,* that would be confluent persistence.

So confluent persistence, you can combine two versions to create a new version. And in this case, again, you can't destroy old versions. In persistence, you never destroy versions. So now the versions form a DAG, directed acyclic graph.

So now we're allowing-- OK, you make some changes, whatever. You branch your universe, make some changes. And now I can say, OK, take this version of the data structure and this version and recombine them. Get a new version, and then maybe make some more changes.

OK, what does combine mean? Well, it depends on your data structure. A lot of data structures have combine operations like if you have linked lists, you have two linked lists, you can concatenate them. That's an easy operation. Even if you have binary search trees, you can concatenate them reasonably easy and combine it into one big binary search tree.

So if your data structure has an operation that takes as input two data structures, then what we're saying is now it can take two versions, which is more general. So I could take the same data structure, make some changes in one way, separately make some changes in a different way, and then try to concatenate them or do something crazy.

This is hard to do, and most of it is an open problem whether it can be done. But I'll tell you about it. Then there's another level even more than confluent persistence. This is hard to interpret in the time travel world, but it would be functional data structures. If you've ever programmed in a functional programming language, it's a little bit annoying from an algorithm's perspective, because it constrains you to work in a purely functional world. You can never modify anything. OK.

Now, we don't want to modify versions. That's fine. But in a functional data structure, you're not allowed to modify any nodes ever. All you can do is make new notes. This is constraining, and you can't always get optimal running times in the functional world. But if you can get a functional data structure, you have all these things, because you can't destroy anything. If you can't destroy nodes, then in particular, you can't destroy versions. And all of these things just work for free.

And so a bunch of special cases are known, interesting special cases, like search trees you can do in the functional world. And that makes all of these things easy. So the rest of this lecture is going to be general techniques for doing partial full persistence, what we know about confluent, and what we know about functional, brief overview. Any questions about those

goals, problem definitions? Yeah.

AUDIENCE: I'm still confused about functional, because--

ERIK DEMAINE: What does functional mean?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Yeah, I guess you'll see what-- functional looks like all the other things, I agree. You'll see in a moment how we actually implement partial and persistence. We're going to be changing nodes a lot. As long as we still represent the same data in the old versions, we don't have to represent it in the same way. That lets us do things more efficiently. Whereas in functional, you have to represent all the old versions in exactly the way you used to represent them. Here we can kind of mangle things around and it makes things faster. Yeah, good question.

So it seems almost the same, but it's nodes versus versions. I haven't really defined a version. But it's just that all the queries answer the same way. That's what you need for persistence. Other questions? All right.

Well, let's do some real data structures. We start with partial persistence. This is the easiest. For both partial and full persistence, there is the following result. Any pointer machine data structure, one catch with a constant number of pointers to any node-- so this is constant n degree.

In a pointer machine, you always have a constant number of pointers out of a node at most. But for this result to hold, we also need a constant number of pointers into any node. So this is an extra constraint.

Can be transformed into another data structure that is partially persistent and does all the things it used to do-- so I'll just say, can be made partially persistent. You have to pay something, but you have to pay very little-- constant amortized factor overhead, multiplicative overhead and constant amount of additive space per change in the data structure.

So every time you do a modification in your pointer machine-- you set one of the fields to something-- you have to store that forever. So, I mean, this is the best you could hope to do. You've got to store everything that happened. You pay a constant factor overhead, eh. We're theoreticians. That doesn't matter. Then you get any data structure in this world can be made partially persistent. That's nice. Let's prove it.

OK, the idea is pretty simple. Pointer machines are all about nodes and fields. So we just need to simulate whatever the data structure is doing to those nodes and fields in a way that we don't lose all the information and we can still search it very quickly. First idea is to store back pointers. And this is why we need the constant n degree constraint.

So if we have a node-- how do I want to draw a node here? So maybe these are the three fields of the node. I want to also store some back pointers. Whenever there is a node that points to this node, I want to have a back pointer that points back so I know where all the pointers came from.

If there's only p pointers, then this is fine. There'll be p fields here. So still constant, still in the pointier machine model. OK, I'm going to need some other stuff too.

So this is a simple thing, definitely want this. Because if my nodes ever move around, I've got to update the pointers to them. And where are those pointers? Well, the back pointers tell you where they are. Nodes will still be constant size, remain in pointer machine data structure.

OK. That's idea one. Idea two is this part. This is going to store something called mods. It could stand for something, but I'll leave it as mods. So these are two of the fields of the data structure.

Ah, one convenience here is for back pointers, I'm only going to store it for the latest version of the data structure. Sorry. I forgot about that. We'll come back to that later.

And then the idea is to store these modifications. How many modifications? Let's say up to p, twice p. p was the bound on the n degree of a node. So I'm going to allow 2p modifications over here. And what's a modification look like?

It's going to consist of three things-- get them in the right order-- the version in which something was changed, the field that got changed, and the value it go changed to. So the idea is that these are the fields here. We're not going to touch those. Once they're set to something-- or, I mean, whatever they are initially, they will stay that way.

And so instead of actually changing things like the data structure normally would, we're just going to add modifications here to say, oh, well at this time, this field changed to the value of 5. And then later on, it changed to the value 7. And then later on, this one changed to the value 23, whatever. So that's what they'll look like.

There's a limit to how many-- we can only store a constant number of mods to each node. And our constant will be 2p. OK. Those are the ideas, and now it's just a matter of making this all work and analyzing that it's constant amortized overhead.

So first thing is if you want to read a field, how would I read a field? This is really easy. First you look at what the field is in the node itself. But then it might have been changed. And so remember when I say read the field, I actually mean while I'm given some version, v, I want to know what is the value of this field at version v, because I want to be able to look at any of the old data structures too.

So this would be at version v. I just look through all the modifications. There's constantly many, so it takes constant time to just flip through them and say, well, what changes have happened up to version v? So I look at mods with version less than or equal to v. That will be all the changes that happened up to this point.

I see, did this field change? I look at the latest one. That will be how I read the field of the node, so constant time. There's lots of ways to make this efficient in practice. But for our purposes, it doesn't matter. It's constant.

The hard part is how do you change a field? Because there might not be any room in the mod structure. So to modify, say we want to set node.field equal to x. What we do is first we check, is there any space in the mod structure? If there's any blank mods, so if the node is not full, we just add a mod. So a mod will look like now field x.

Just throw that in there. Because right at this moment-- so we maintain a time counter, just increment it ever time we do a change. This field changed that value. So that's the easy case. The trouble, of course, is if the node is full-- the moment you've all been waiting for.

So what we're going to do here is make a new node. We've ran out of space. So we need to make a new node. We're not going to touch the old node, just going to let it sit. It still maintains all those old versions. Now we want a new node that represents the latest and greatest of this node. OK.

So make a new node. I'll call it node prime to distinguish from node, where with all the mods, and this modification in particular, applied.

OK, so we make a new version of this node. It's going to have some different fields, whatever

was the latest version represented by those mods. It's still going to have back pointers, so we have to maintain all those back pointers.

And now the mod, initially, is going to be empty, because we just applied them all. So this new node doesn't have any recent mods. Old node represents the old versions. This node is going to represent the new versions. What's wrong with this picture?

AUDIENCE: Update pointers.

ERIK DEMAINE: Update pointers. Yeah, there's pointers to the old version of the node, which are fine for the old versions of the data structure. But for the latest version of the data structure, this node has moved to this new location.

So if there are any old pointers to that node, we've got to update them in the current version. We have to update them to point to this node instead. The old versions are fine, but the new version is in trouble. Other questions or all the same answer? Yeah.

AUDIENCE: So if you wanted to read an old version but you just have the new version, [INAUDIBLE]?

ERIK DEMAINE: OK--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: The question is essentially, how do we hold on to versions? Essentially, you can think of a version of the data structure as where the root node is. That's probably the easiest. I mean, in general, we're representing versions by a number, v. But we always start at the root.

And so you've given the data structure, which is represented by the root node. And you say, search for the value 5. Is it in this binary search tree or whatever? And then you just start navigating from the root, but you know I'm inversion a million or whatever. I know what version I'm looking for. So you start with the root, which never changes, let's say.

And then you follow pointers that essentially tell you for that version where you should be going. I guess at the root version, it's a little trickier. You probably want a little array that says for this version, here's the root node. But that's a special case. Yeah. Another question?

AUDIENCE: So on the new node that you created, the fields that you copied, you also have to have a version for them, right? Because [INAUDIBLE]?

ERIK DEMAINE: These--

AUDIENCE: Or do you version the whole node?

ERIK DEMAINE: Here we're versioning the whole node. The original field values represent what was originally there, whenever this node was created. Then the mods specify what time the fields change. So I don't think we need times here. All right.

So we've got to update two kinds of pointers. There's regular pointers, which live in the fields, which are things pointing to the node. But then there's also back pointers. Because if this is a pointer to a node, then there'll be a back pointer back to the node. And all of those have to change.

Conveniently, the back pointers are easy. So if they're back pointers to the node, we change them to the node prime. How do we find the back pointers? Well, we just follow all the pointers and then there will be back pointers there.

Because I said we're only maintaining backed pointers for the latest version, I don't need to preserve the old versions of those backed pointers. So I just go in and I change them. It takes constant time, because the constant number of things I point to, each one as a back pointer. So this is cheap. There's no persistence here. That's an advantage of partial persistence.

The hard part is updating the pointers because those live in fields. I need to remember the old versions of those fields. And that we do recursively.

Because to change those pointers, that's a field update. That's something exactly of this form. So that's the same operation but on a different node. So I just do that. I claim this is good. That's the end of the algorithm. Now we need to analyze it.

How do we analyze it? Any guesses?

AUDIENCE: Amortize it.

ERIK DEMAINE: Amortized analysis, exactly the answer I was looking for. OK. [INAUDIBLE] amortization. The most powerful technique in amortization is probably the potential method. So we're going to use that. There's a sort of more-- you'll see a charging argument in a moment.

We want the potential function to represent when this data structure is in a bad state. Intuitively, it's in a bad state when a lot of nodes are full. Because then as soon as you make a

change in them, they will burst, and you have to do all this crazy recursion and stuff.

This case is nice and cheap. We just add a modification, constant time. This case, not so nice because we recurse. And then that's going to cause more recursions and all sorts of chaos could happen.

So there's probably a few different potential functions that would work here. And an old version of these nodes I said should be the number of full nodes. But I think we can make life a little bit easier by the following. Basically, the total number of modifications-- not quite the total, almost the total.

So I'm going to do c times the sum of the number of mods in latest version nodes. OK. So because we sort of really only care about-- we're only changing the latest version, so I really only care about nodes that live in the latest version.

What do I mean by this? Well, when I made this new node prime, this becomes the new representation of that node. The old version is dead. We will never change it again. If we're modifying, we will never even look at it again. Because now everything points to here.

So I don't really care about that node. It's got a ton of mods. But what's nice is that when I create this new node, now the mod list is empty. So I start from scratch, just like reinstalling your operating system. It's a good feeling.

And so the potential goes down by, I guess, c times 2 times p. When I do this change, potential goes down by basically p.

AUDIENCE: Is c any constant or--

ERIK DEMAINE: c will be a constant to be determined. I mean, it could be 1. It depends how you want to define it. I'm going to use the CLRS notion of amortized cost, which is actual cost plus change in potential. And then I need a constant here, because I'm measuring a running time versus some combinatorial quantity. So this will be to match the running time that we'll get to.

OK. So what is amortized cost? There's sort of two cases modification. There's the cheap case and the not so cheap case. In general, amortized cost-- in both cases, it's going to be at most-- well, first of all, we do some constant work just to figure out all this stuff, make copies, whatever. So that's some constant time. That's the part that I don't want to try to measure.

Then potentially, we add a new mod. If we add a mod, that increases the potential by c. Because we're just counting mods, multiplying by c. So we might get plus 1 mod. This is going to be an upper bound. We don't always add 1, but worst case, we always had 1, let's say.

And then there's this annoying part. And this might happen, might not happen. So then there's a plus maybe. If this happens, we decrease the potential because we empty out the mods for that node in terms of the latest version. So then we get a negative 2cp, change in potential. And then we'd have to pay I guess up to p recursions.

Because we have to-- how many pointers are there to me? Well, at most p of them, because there are at most p pointers to any node. OK.

This is kind of a weird-- it's not exactly algebra here. I have this thing, recursions. But if you think about how this would expand, all right, this is constant time. That's good. And then if we do this-- I'll put a question mark here. It might be here. It might not.

If it's not here, find constant. If it is here, then this gets expanded into this thing. It's a weird way to write a recurrence. But we get p times whatever is in this right hand side. OK.

But then there's this minus 2cp. So we're going to get p times 2c here. That's the initial cost. So that will cancel with this. And then we might get another recursion. But every time we get a recursion, all the terms cancel. So it doesn't matter whether this is here or not. You get 0, which is great. And you're left with the original 2c. Constant. OK.

[INAUDIBLE] potential functions are always a little crazy. What's happening here is that, OK, maybe you add a mod. That's cheap. But when we have to do this work and we have to do this recursion-- this is up to 2p updates or recursions-- we are charging it to the emptying of this node. The number of mods went from 2p down to 0.

And so we're just charging this update cost to that modification. So if you like charging schemes, this is much more intuitive. But with charging schemes, it's always a little careful. You have to make sure you're not double charging. Here it's obvious that you're not double charging. Kind of a cool and magical.

This is a paper by Driscoll, Sarnak, Sleator, Tarjan from 1989. So it's very early days of amortization. But they knew how to do it. Question?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: What happens if you overflow the root? Yeah, I never thought about the root before today. But I think the way to fix the root is just you have one big table that says, for a given version-- I guess a simple way would be to say, not only is a version a number, but it's also a pointer to the root. There we go. Pointer machine.

So that way you're just always explicitly maintaining the root copy or the pointer. Because otherwise, you're in trouble.

AUDIENCE: Then can you go back to [INAUDIBLE].

ERIK DEMAINE: So in order to refer to an old version, you have to have the pointer to that root node. If you want to do it just from a version number, look at the data structure. Just from a version number, you would need some kind of lookup table, which is outside the pointer machine. So you could do it in a real computer, but a pointer machine is not technically allowed. So it's slightly awkward. No arrays are allowed in pointer machines, in case that wasn't clear. Another question?

AUDIENCE: [INAUDIBLE] constant space to store for [INAUDIBLE]. And also, what if we have really big numbers [INAUDIBLE]?

ERIK DEMAINE: In this model, in the pointer machine model, we're assuming that whatever the data is in the items take constant space each. If you want to know about bigger things in here, then refer to future lectures. This is time travel, after all. Just go to a future class and then come back. [LAUGHS] So we'll get there, but right now, we're not thinking about what's in here.

Whatever big thing you're trying to store, you reduce it down to constant size things. And then you spread them around nodes of a pointer machine. How you do that, that's up to the data structure. We're just transforming the data structure to be persistent. OK, you could ask about other models than pointer machines, but we're going to stick to pointer machines here.

All right. That was partial persistence. Let's do full persistence. That was too easy. Same paper does full persistence. Systems That was just a warm up. Full persistence is actually not that much harder. So let me tell you basically what changes.

There are two issues. One is that everything here has to change and not by much. We're still going to use back pointers. We're still going to have my mods. The number of mods is going to be slightly different but basically the same. Back pointers no longer just refer to the latest

version. We have to maintain back pointers in all versions. So that's annoying. But hey, that's life. The amortization, the potential function will change slightly but basically not much.

Sort of the bigger issue you might first wonder about, and it's actually the most challenging technically, is versions are no longer numbers. Because it's not a line. Versions are nodes in a tree. You should probably call them vertices in a tree to distinguish them from nodes in the pointer machine.

OK, so you've got this tree of versions. And then versions are just some point on that tree. This is annoying because we like lines. We don't like trees as much. So what we're going to do is linearize the tree. Like, when in doubt, cheat.

How do we do this? With tree traversal. Imagine I'm going to draw a super complicated tree of versions. Say there are three versions. OK. I don't want to number them, because that would be kind of begging the question. So let's just call them x, y, and z.

All right. I mean, it's a directed tree, because we have the older versions. This is like the original version. And we made a change. We made a different change on the same version.

What I'd like to do is a traversal of that tree, like a regular, as if you're going to sort those nodes. Actually, let me use color, high def here. So here's our traversal of the tree. And I want to look at the first and the last time I visit each node.

So here's the first time I visit x. So I'll write this is the beginning of x. Capital X. Then this is the first time I visit y, so it's beginning of y. And then this is the last time I visit y, so it's the end of y. And then, don't care. Then this is the beginning of z. And this is the end of z. And then this is the end x.

If I write those sequentially, I get bxbyeybzez, because this is so easy, ex. OK, you can think of these as parentheses, right? For whatever reason I chose b and e for beginning and ending, but this is like open parens, close parens.

This is easy to do in linear time. I think you all know how. Except it's not a static problem. Versions are changing all the time. We're adding versions. We're never deleting versions, but we're always adding stuff to here. It's a little awkward, but the idea is I want to maintain this order, maintain the begin and the end of each you might say subtree of versions.

This string, from bx to ex, represents all of the stuff in x's subtree, in the rooted tree starting at

x. How do I maintain that? Using a data structure.

So we're going to use something, a data structure we haven't yet seen. It will be in lecture 8. This is a time travel data structure, so I'm allowed to do that. So order maintenance data structure.

You can think of this as a magical linked list. Let me tell you what the magical linked list can do. You can insert-- I'm going to call it an item, because node would be kind of confusing given where we are right now. You can insert a new item in the list immediately before or after a given item.

OK. This is like a regular linked list. Here's a regular linked list. And if I'm given a particular item like this one, I can say, well, insert a new item right here. You say, OK. Fine. I'll just make a new node and relink here, relink there. Constant time, right?

So in an order maintenance data structure, you can do this in constant time. Wow! So amazing. OK, catch is the second operation you can do. Maybe I'll number these. This is the update. Then there's the query.

The query is, what is the relative order of two notes, of two items? x and y. So now I give you this node and this node. And I say, which is to the left? Which is earlier in the order? I want to know, is x basically less than y in terms of the order in the list? Or is y less than x?

And an order maintenance data structure can do this in constant time. Now it doesn't look like your mother's linked list, I guess. It's not the link list you learned in school. It's a magical linked list that can somehow answer these queries. How? Go to lecture 7. OK. Forward reference, lecture 8, sorry.

For now, we're just going to assume that this magical data structure exists. So in constant time, this is great. Because if we're maintaining these b's and e's, we want to maintain the order that these things appear in. If we want to create a new version, like suppose we were just creating version z, well, it used to be everything without this bz, ez. And we'd just insert two items in here, bz and ez.

They're right next to each other. And if we were given version x, we could just say, oh, we'll look at ex and insert two items right before it. Or you can put them right after bx. I mean, there's no actual order here. So it could have been y first and then z or z first and then y.

So it's really easy to add a new version in constant time. You just do two of these insert operations. And now you have this magical order operation, which if I'm given two versions-- I don't know, v and w-- and I want to know is v an ancestor of w, now I can do it in constant time. So this lets me do a third operation, which is, is version v an ancestor of version w?

Because that's going to be true if and only if bv is an ev nest around bw and ew. OK. So that's just three tests. They're probably not all even necessary. This one always holds. But if these guys fit in between these guys, then you know-- now, what this tells us, what we care about here, is reading fields.

When we read a field, we said, oh, we'll apply all the modifications that apply to version v. Before that, that was a linear order. So it's just all versions less than or equal to v. Now it's all versions that are ancestors of v. Given a mod, we need to know, does this mod apply to my version?

And now I tell you, I can do that in constant time through magic. I just test these order relations. If they hold, then that mod applies to my version. So w's the version we're testing. v is some version in the mod. And I want to know, am descendant of that version? If so, the mod applies. And I update what the field is.

I can do all pairwise ancestor checks and figure out, what is the most recent version in my ancestor history that modified a given field? That lets me read a field in constant time. Constants are getting kind of big at this point, but it can be done.

Clear? A little bit of a black box here. But now we've gotten as far as reading. And we don't need to change much else. So this is good news

Maybe I'll give you a bit of a diff. So full persistence, fully persistent theorem-- done. OK. Same theorem just with full persistence.

How do we do it? We store back pointers now for all versions. It's a little bit annoying. But how many mods do we use? There's lots of ways to get this to work, but I'm going to change this number to 2 times d plus p plus 1. Wait, what's d? d is the number of fields here. OK. We said it was constant number fields. I never said what that constant is. d for out degree, I guess.

So p is in degree, max in degree. d is max out degree. So just slightly more-- that main reason for this is because back pointers now are treated like everyone else. We have to treat both the out pointers and the in pointers as basically the same. So instead of p, we have d plus p. And

there's a plus 1 just for safety. It gets my amortization to work, hopefully.

OK. Not much else-- this page is all the same. Mods are still, you give versions, fields, values, reading. OK, well, this is no longer less than or equal to v. But this is now with a version, sort of the nearest version, that's an ancestor of v.

That's what we were just talking about. So that can be done in constant time. Check it for all of them, constant work. OK. That was the first part.

Now we get to the hard part, which is modification. This is going to be different. Maybe you I should just erase-- yeah, I think I'll erase everything, except the first clause.

OK. If a node is not full, we'll just add a mod, just like before. What changes is when a node is full. Here we have to do something completely different. Why? Because if we just make a new version of this node that has empty mods, this one's still full. And I can keep modifying the same version. This new node that I just erased represents some new version. But if I keep modifying an old version, which I can do in full persistence, this node keeps being full. And I keep paying potentially huge cost.

If all the nodes were full, and when I make this change every node gets copied, and then I make a change to the same version, every node gets copied again. This is going to take linear time per operation. So I can't do the old strategy. I need to somehow make this node less full. This is where we're definitely not functional. None of this was functional, but now I'm going to change an old node, not just make a new one in a more drastic way.

Before I was adding a mod. That's not a functional operation. Now I'm actually going to remove mods from a node to rebalance. So what I'd like to do is split the node into two halves. OK. So I had some big node that was-- I'll draw it-- completely full. Now I'm going to make two nodes. Here we go.

This one is going to be half full. This one's going to be half full of mods. OK. The only question left is, what do I do with all these things? Basically what I'd like to do is have the-- on the one hand, I want to have the old node. It's just where it used to be. I've just removed half of the mods, the second half, the later half. What does that mean? I don't know. Figure it out. It's linearized. I haven't thought deeply about that.

Now we're going to make a new node with the second half of the mods. It's more painful than I

thought. In reality, these mods represent a tree of modifications. And what you need to do is find a partition of that tree into two roughly equal halves. You can actually do a one third, 2/3 split. That's also in a future lecture, which whose number I forget.

So really, you're splitting this tree into two roughly balanced halves. And so this 2 might actually need to change to a 3, but it's a constant. OK. What I want is for this to represent a subtree of versions. Let me draw the picture.

So here's a tree of versions represented by the old mods. I'd like to cut out a subtree rooted at some node. So let's just assume for now this has exactly half the nodes. And this has half the nodes. In reality, I think it can be one third, 2/3. OK. But let's keep it convenient.

So I want the new node to represent this subtree and this node to represent everything else. This node is as if this stuff hasn't happened yet. I mean, so it represents all these old versions that do not, that are not in the subtree. This represents all the latest stuff.

So what I'm going to do is like before, I want to apply some mods to these fields. And whatever minds were relevant at this point, whatever had been applied, I apply those to the fields here. And so that means I can remove all of these mods. I only cared about these ones. Update these fields accordingly. I still have the other mods to represent all the other changes that could be in that subtree. OK.

So we actually split the tree, and we apply mods to new nodes. Anything else I need to say? Oh, now we need to update pointers. That's always the fun part. Let's go over here.

So old node hasn't moved. But this new node has moved. So for all of these versions, I want to change the pointer that used to point to old node should now point to new node. In this version, it's fine. It should still point to old node, because this represents all those old versions. But for the new version, that version in the subtree, I've got to point here instead.

OK. So how many pointers could there be to this node that need to change. That's a tricky part in this analysis. Think about it for a while. I mean, in this new node, whatever is pointed to by either here or here in the new node also has a return pointer. All pointers are bidirectional. So we don't really care about whether they're forward or backward.

How many pointers are there here? Well, there's d here and there's p here. But then there's also some additional pointers represented over here. How many? Well, if we assume this magical 50/50 split, there's right now d plus p plus 1 mods over here, half of them.

Each of them might be a pointer to some other place, which has a return pointer in that version. So number of back pointers that we need to update is going to be this, 2 times d 2 times p plus 1.

So recursively update at most 2 times d plus 2 times p plus 1 pointers to the node. The good news is this is really only half of them or some fraction of them. It used to be-- well, there were more pointers before. We don't have to deal with these ones. That's where we're saving, and that's why this amortization works. Let me give you a potential function that makes this work-- is minus c times sum of the number of empty mod slots.

It's kind of the same potential but before we had this notion of dead and alive nodes. Now everything's alive because everything could change at any moment. So instead, I'm going to measure how much room I have in each node. Before I had no room in this node. Now I have half the space in both nodes. So that's good news.

Whenever we have this recursion, we can charge it to a potential decrease. Fee goes down by-- because I have a negative sign here-- c times, oh man, 2 times d plus p plus 1, I think. Because there's d plus p plus 1 space here, d plus p plus 1 space here. I mean, we added one whole new node. And total capacity of a node in mods is 2 times d plus p plus 1. So we get that times c.

And this is basically just enough, because this is 2 times d plus 2 times p plus 2. And here we have a plus 1. And so the recursion gets annihilated by 2 times d plus 2 times p plus 1. And then there's one c left over to absorb whatever constant cost there was to do all this other work.

So I got the constants just to work, except that I cheated and it's really a one third, 2/3 split. So probably all of these constants have to change, such is life. But I think you get the idea. Any questions about full persistence? This is fun stuff, time travel. Yeah?

AUDIENCE: So in the first half of the thing where the if, there's room you can put it in.

ERIK DEMAINE: Right.

AUDIENCE: I have a question about how we represent the version. Because before when we said restore now [INAUDIBLE]. It made more sense if now was like a timestamp or something.

ERIK DEMAINE: OK. Right, so how do we represent a version even here or anywhere? When we do a modification, an update, in the data structure, we want to return the new version. Basically, we're going to actually store the DAG of versions. And a version is going to be represented by a pointer into this DAG. One of the nodes in this DAG becomes a version.

Every node in this DAG is going to store a pointer to the corresponding b character and a corresponding e character in this data structure, which then lets you do anything you want. Then you can query against that version, whether it's an ancestor of another version.

So yeah, I didn't mention that. Versions are nodes in here. Nodes in here have pointers to the b's and e's. And vice versa, the b's and e's have pointers back to the corresponding version node. And then you can keep track of everything. Good question. Yeah?

AUDIENCE: [INAUDIBLE] question. Remind me what d is in this.

ERIK DEMAINE: Oh, d was the maximum out degree. It's the number of fields in a node, as defined right here. Other questions? Whew. OK, a little breather.

That was partial persistence, full persistence. This is, unfortunately, the end of the really good results. As long as we have constant degree nodes, in and out degree, we can do all. We can do for persistence for free.

Obviously there are practical constants involved here. But in theory, you can do this perfectly. Before we go on to confluence, there is one positive result, which is what if you don't like amortize bounds. There are various reasons amortize bounds might not be good. Maybe you really care about every operation being no slower than it was except by a constant factor. We're amortizing here, so some operations get really slow. But the others are all fast to compensate.

You can deamortize, it's called. You can get constant worst case slowdown for partial persistence. This is a result of Garret Brodle from the late '90s, '97. For full persistence-- so it's an open problem. I don't know if people have worked on that. All right. So some, mostly good results.

Let's move on to confluent persistence where things get a lot more challenging. Lots of things go out the window with confluent persistence. In particular, your versions are now a DAG. It's a lot harder to linearize a DAG. Trees are not that far from pads. But DAGs are quite far from pads, unfortunately. But that's not all that goes wrong.

Let me first tell you the kind of end effect as a user. Imagine you have a data structure. Think of it as a list, I guess, which is a list of characters in your document. You're using vi or Word, your favorite, whatever. It's a text editor. You've got a string of words.

And now you like to do things like copy and paste. It's a nice operation. So you select an interval of the string and you copy it. And then you paste it somewhere else. So now you've got two copies of that string.

This is, in some sense, what you might call a confluent operation, because-- yeah, maybe a cleaner way to think of it is the following. You have your string. Now I have an operation, which is split it. So now I have two strings. OK. And now I have an operation, which is split it. Now I have three strings.

OK. Now I have an operation which is concatenate. So I can, for example, reconstruct the original string-- actually, I have the original string. No biggie. Let's say-- because I have all versions. I never lose them. So now instead, I'm going to cut the string here, let's say. So now I have this and this. And now I can do things like concatenate from here to here to here. And I will get this plus this plus this.

OK. This guy moved here. So that's a copy/paste operation with a constant number of splits and concatenates. I could also do cut and paste. With confluence, I can do crazy cuts and pastes in all sorts of ways. So what?

Well, the so what is I can actually double the size of my data structure in a constant number of operations. I can take, for example, the entire string and concatenate it to itself. That will double the number of characters, number of elements in there. I can do that again and again and again.

So in u updates, I can potentially get a data structure size 2 to the u. Kind of nifty. I think this is why confluence is cool. It's also why it's hard. So not a big surprise. But, here we go.

In that case, the version DAG, for reference, looks like this. You're taking the same version, combining it. So here I'm assuming I have a concatenate operation. And so the effect here, every time I do this, I double the size.

All right. What do I want to say about confluent persistence? All right. Let me start with the most general result, which is by Fiat and Kaplan in 2003.

They define a notion called effective depth of a version. Let me just write it down. It's kind of like if you took this DAG and expanded it out to be a tree of all possible paths. Instead of point to the same node, you could just duplicate that node and then have pointers left and right.

OK. So if I did that, of course, this size grows exponentially. It explicitly represents the size of my data structure. At the bottom, if I have u things, I'm going to have 2 to the u leaves at the bottom.

But then I can easily measure the number of paths from the root to the same version. At the bottom, I still label it, oh, those are all v. They're all the same version down there. So exponential number of paths, if I take log, I get what I call effective depth. It's like if you somehow could rebalance that tree, this is the best you could hope to do. It's not really a lower bound. But it's a number. It's a thing. OK.

Then the result they achieve is that the overhead is log the number of updates plus-- this is a multiplicative overhead, so you take your running time. You multiply it by this. And this is a time and a space overhead.

So maximum effective depth of all versions, maybe even sum of effective depths, but we'll just say max to be safe. Sorry-- sum over all the operations. This is per operation. You pay basically the effective depth of that operation as a factor.

Now, the annoying thing is if you have this kind of set up where the size grew exponentially, then number of paths is exponential. Log of the number of paths is linear in u. And so this factor could be as much as u, linear slowdown.

Now, Fiat and Kaplan argue linear slowdown is not that bad, because if you weren't even persistent, if you did this in the naive way of just recopying the data, you were actually spending exponential time to build the final data structure. It has exponential size. Just to represent it explicitly requires exponential time, so losing a linear factor to do u operations and now u squared time instead of 2 to the u. So it's a big improvement to do this.

The downside of this approach is that even if you have a version DAG that looks like this, even if the size of the data structure is staying normal, staying linear, so this potential, you could be doubling the size. But we don't know what this merge operation is. Maybe it just throws away one of the versions or does something-- somehow takes half the nodes from one side, half the

nodes from the other side maybe. These operations do preserve size.

Then there's no great reason why it should be a linear slowdown, but it is. OK? So it's all right but not great. And it's the best general result we know.

They also prove a lower bound. So lower bound is some effect of depth, total bits of space. OK. What does this mean?

So even if this is not happening, the number of bits of space you need in the worst case-- this does not apply to every data structure. That's one catch. They give a specific data structure where you need this much space. So it's similar to this kind of picture. We'll go into the details.

And you need this much space. Now, this is kind of bad, because if there's u operations, and each of these is u, that's u squared space. So we actually need a factor u blow up in space. It looks like.

But to be more precise, what this means is that you need omega e of v space, and therefore time overhead per update, if-- this is not written in the paper-- queries are free. Implicit here, they just want to slow down and increase space for the updates you do, which is pretty natural. Normally you think of queries as not increasing space.

But in order to construct this lower bound, they actually do this many queries. So they do e of v queries and then one update. And they say, oh well, space had to go up by an extra e of v. So if you only charge updates for the space, then yes, you have to lose potentially a linear factor, this effect of death, potentially u.

But if you also charge the queries, it's still constant in their example. So open question, for confluent persistence, can you achieve constant everything? Constant time and space overheads, multiplicative factor per operation, both updates and queries. So if you charge the queries, potentially you could get constant everything.

This is a relatively new realization. And no one knows how to do this yet. Nice challenge. I think maybe we'll work on that in our first problem session. I would like to.

Questions about that result? I'm not going to prove the result. But it is a fancy rebalancing of those kinds of pictures to get this log. There are other results I'd like to tell you about.

So brand new result-- that was from 2003. This is from 2012-- no, '11, '11, sorry. It's SOTO,

which is in January, so it's a little confusing. Is it '11? Maybe '12. Actually now I'm not sure. It's February already, right? A January, either this year or last year.

It's not as general a transformation. It's only going to hold in what's called a disjoint case. But it gets a very good bound-- not quite constant, but logarithmic. OK, logarithmic would also be nice. Or log, log n, whatever n is. Pick your favorite n, number of operations, say.

OK. If you assume that confluent operations are performed only on two versions with no shared nodes-- OK, this would be a way to forbid this kind of behavior where I concatenate the data structure with itself. All the nodes are common.

If I guarantee that maybe I, you know, slice this up, slice it, dice it, wherever, and then re-emerge them in some other order, but I never use two copies of the same piece, that would be a valid confluent operation over here. This is quite a strong restriction that you're not allowed. If you try to, who knows what happens. Behavior's undefined. So won't tell you, oh, those two versions have this node in common. You've got to make a second copy of it.

So somehow you have to guarantee that control and operations never overlap. But they can be reordered. Then you can get order log n overhead. n is the number of operations.

I have a sketch of a proof of this but not very much time to talk about it. All right. Let me give you a quick picture. In general, the versions form a DAG. But if you make this assumption, and you look at a single node, and look at all the versions where that node appears, that is a tree. Because you're not allowed to remerge versions that have the same node.

So while the big picture is a DAG, the small picture of a single guy is some tree. I'm drawing all these wiggly lines because there are all these versions where the node isn't changing. This is the entire version DAG. And then some of these nodes-- some of these versions, I should say-- that node that we're thinking about changes.

OK, whenever it branches, it's probably because the actual node changed, maybe. I don't know. Anyway there are some dots here where the version changed, some of the leaves, maybe, that changed. Maybe some of them haven't yet.

In fact, let's see. Here where it's change, it could be that we destroyed the node. Maybe it's gone from the actual data structure. But there still may be versions down here. It's not really a tree. It's a whole DAG of stuff down there. So that's kind of ugly.

Where never the node still exists, I guess that is an actual leaf of the DAG. So those are OK. But as soon as I maybe delete that node, then there can be a whole subtree down there.

OK. So now if you look at an arbitrary version, so what we're thinking about is how to implement reading, let's say. Reading and writing are more or less the same. I give you a version. I give you a node, and I give you a field. I want to know, what is the value of that field, that node, that version?

So now where could a version fall? Well it has to be in this subtree. Because the node has to exist. And then it's maybe a pointer. A pointer could be to another node, which also has this kind of picture. They could be overlapping trees.

In general, there are three cases. Either you're lucky, and the version you're talking about is a version where the node was changed. In that case, the data is just stored right there. That's easy. So you could just say, oh, how did the node change? Oh, that's what the field is. OK, follow the pointer.

A slightly harder case it's a version in between two such changes. And maybe these are not updates. So I sort of want to know, what was the previous version where this node changed in constant time? It can be done. Not constant time, actually, logarithmic time, using a data structure called link-cut trees, another fun black box for now, which we will cover in lecture 19, far in the future. OK.

Well, that's one case. There's also the version where maybe a version is down here in a subtree. I guess then the node didn't exist. Well, all these things can happen. And that's even harder. It's messy.

They use another trick, which is called fractional cascading, which I'm not even going to try to describe what it means. But it's got a very cool name. Because we'll be covering it in lecture 3. So stay tuned for that. I'm not going to say how it applies to this setting, but it's a necessary step in here.

In the remaining zero minutes, let me tell you a little bit about functional data structures.

[LAUGHTER]

Beauty of time travel. Functional-- I just want to give you some examples of things that can be

done functionally. There's a whole book about functional data structures by Okasaki. It's pretty cool.

A simple example is balanced BSTs. So if you just want to get log n time for everything, you can do that functionally. It's actually really easy. You pick your favorite balance BST, like red black trees. You implement it top down so you never follow parent pointers. So you don't need parent pointers.

So then as you make changes down the tree, you just copy. It's called path copying. Whenever you're about to make a change, make a copy of that node. So you end up copying all the change nodes and all their ancestors. There's only log n of them, so it takes log n time. Clear? Easy.

It's a nice technique. Sometimes path copying is very useful. Like link-cut trees, for example, can be made functional. We don't know what they are, but they're basically a BST. And you can make them functional. We use that in a paper.

All right. Deques. These are doubly ended queues. So it's like a stack and a queue and everything. You can insert and delete from the beginning and the end. People start to know what these are now, because Python calls him that.

But you can also do concatenation with deques in constant time per operation. This is cool. Deques are not very hard to make functional. But you can do deques and you can concatenate them like we were doing in the figure that's right behind this board.

Constant time split is a little harder. That's actually one of my open problems. Can you do lists with split and concatenate in constant time-- functionally or confluently, persistently, or whatever?

Another example-- oh, you can do a mix of the two. You can get log n search in constant time deque operations, is you can do tries. So a try is a tree with a fixed topology. Think of it as a directory tree. So maybe you're using Subversion. Subversion has time travel operations. You can copy an entire subtree from one version and stick it into a new version, another version. So you get a version DAG. It's a confluently persistent data structure-- not implemented optimally, because we don't necessarily know how. But there is one paper.

This actually came from the open problem section of this class four years ago, I think. It's with Eric Price and Stefan Langerman. You can get very good results. I won't write them down

because it takes a while. Basically log the degree of the nodes factor and get functional, and you can be even fancier and get slightly better bounds like log log the degree and get confluently persistent with various tricks, including using all of these data structures.

So if you want to implement subversion optimally, that is known how to be done but hasn't actually been done yet. Because there are those pesky constant factors. I think that's all. What is known about functional is there's a log n separation. You can be log n away from the best. That's the worst separation known, between functional and just a regular old data structure. It'd be nice to improve that. Lots of open problems here. Maybe we'll work on them next time.