MITOCW | watch?v=XZLN6NxEQWo

PROFESSOR: So today we're going to cover a data structure called link-cut trees. A cool way of maintaining dynamic trees, and it begins our study of dynamic graphs. Where in general, we have a graph usually undirected. You want to support insertion and deletion of edges. So we're going to do that in the situation where at all times our graphs are trees. So in general, we have a forest of trees and we want to maintain them.

So that will be our only data structure today because it's a bit complicated. It's going to need some fancy amortization. And two techniques for splitting trees into paths, one we've seen already in the context of tango trees and lecture's six, preferred paths. And another, which we haven't seen is probably my favorite technique in data structures, actually, heavy like decomposition. The technique and data structure that I've used the most outside of data structures. It's very useful in lots of settings. Whenever you have an unbalanced tree and you want to make it balance, like with like cut trees, It's the go to tool.

So, in, general we want to maintain a forest of trees. And these are going to be rooted unordered trees. So arbitrary degree I this. And we have a bunch of operations we want to do. Basically, insertion and deletion of edges and some queries. And we want to do all of those operations in the log n time per operation. That's the hard part.

So we have some kind of getting started operation make tree. This is going to be make a new vertex in a new tree, return. It. So this is how you add notes to the structure. And we're not going to worry about doing deletions. Once you add something it stays there forever. So you could you could do that. It's no big deal. You could throw away a tree.

Then we have link which is essentially the insertion of an edge. So here we're given two vertices BMW that are in different trees as a precondition. So v has to be at the root of its tree. W can be any node. So here we have another tree and we have some node w inside v And the link operation adds V as a child of W.

So we add this edge. OK, that, of course, combines two trees into one tree. And the requirement is BMW has to be in different trees or otherwise you'd be adding a cycle that doesn't satisfy that were always a forest of trees. Then we have a cut operation, which is basically the reverse. So let me draw it. We have V subtree parent. And there is a whole tree up here. And what we want to do is remove that edge. So that splits one tree into two trees by the leading edge from V to its parent.

OK so those are our updates. Pretty obvious in the setting of dynamic graphs. Of course, it's the first time we see them. Now we have some queries. There are lots of different queries that cut-trees can support. One of the most basic is find the root of vertex. Let me just draw a picture of. It you have some vertex V, you want to return the root of the tree. So find R. And that's pretty easy to do. Well actually see it in next class, next lecture, if you wanted to make tree/link cut and find root, there's a really easy way to do them in logarithmic time. link-cut trees is a harder way to do it in log n time, but they support a lot of other operations.

And in particular, something called path aggregation. This is kind of the essence of link/cut are cool. Let's suppose you have a for a tree, you have a vertex V. There's a root to V path. And let's suppose that every node or every edge, doesn't matter, has a weight on it. Just some real number. Then path aggregation can do things like the min or the max or the sum or the product or whatever of weights on that path.

And this is a powerful thing. And it's actually the reason link/cut trees were originally invented for doing network flow algorithms faster. If you know network flow algorithms. Or maybe you've seen them in advanced algorithms. This is sort of the key thing you need to do. You find, I think, min way edge along one of these paths and that's the max you can push in a push reliable strategy. I don't want to get into that because that's algorithms, not data structures. But this is why they were invented.

And, in general, link/cut trees have a very strong kind of path thing. If you want to know anything about a root to the path for any node v, link-cut trees can do whatever you want in lon n time. This is just sort of an example of that. And Yeah. course.

So that's what link countries are able to do. All these operations in log n And the main point here is that the trees are storing are probably not balanced. It could be a path, could be depth and is this path we're talking about could be length order n is really bad. And yet, we can do all these operations in log n time. And we're going to do that, essentially, by storing the unbalanced tree that you're trying to represent, which we call this the represent tree. We're going to store it using, essentially, a balanced tree. In one version of link-cut trees, indeed, we store all the nodes in a log n height tree.

The version we're going to cover here-- there's several versions of link-cut trees. Original are biased later in Tarjan in 1983. And then Tarjan wrote a book, one of a few books on data structures. Very thin book, usually called Tarjan's little book, in 1984. And it has a different version of link-cut trees, and that's the version we're going to cover it uses splay trees as a subroutine. So it's not going to be perfectly balanced. It's going to be balanced in an amortize sense, or vaguely balanced, roughly balanced, whatever. But there is a version that does everything in log n worse case pre-operation. If there's time, I'll sketch that for you at the end.

But the simplest version I know is this splay tree version. And it's also kind of fun because the version we'll see, doesn't really look like it should work, and yet it does thanks to splay trees, essentially. So it's a little, it's fun and surprising in that sense. As I said, link-cut trees are all about paths in the tree. And so somehow we want to split a tree up into paths and we've already seen a way to do this. The preferred path decomposition. So it's kind of reminding you, but I'm also going to slightly redefine it, if I recall correctly. It's just a minor difference which I can talk about a second.

The notion of a preferred child of a node, and it's going to be two cases. There isn't a preferred child, if the last access in these subtree is v. So we have some node v. Now, this is relative to the represented tree. So we have some node to v. We don't care about accesses outside of that subtree, but of the last access within the subtree was v itself, then there's no preferred child. Otherwise, the last access within the tree was in one of these child subtrees. And whichever one it was, we say w is the preferred child. So if the last access in v subtree is in child w's subtree.

OK, I think last time we defined preferred children in the context of tango trees, we just had the second part. So if there is an access to v, we ignored it and we just consider what the last child access. Was I don't think there's actually would affect tango trees, but I think it does effect Link-cut trees. So I want to define it this way. It'll make life cleaner, basically. Probably not a huge difference either way.

OK, once you have preferred children, that gives you one outgoing edge, downward edge, from every node. And so we get preferred paths by repeatedly following preferred child pointers. Now, we may stop at some point because we reach a none. I'm going to use none as like null pointers in this lecture.

But, in general, we have some tree and you follow for a while and then it's going to be a bunch of things like this. We decompose the nodes and the represented tree, partition the nodes.

Every node belongs to some preferred path. So basic idea is we're going to store the preferred paths in kind of balanced binary search trees, splay trees, and then we're going to connect them together because that's not all the edges in the tree.

There's some edges that sort of connect them, connect the paths together. That's where the tree part comes in. That's what we're going to do it this thing is unbalanced. We're going to take each of these white lines and compress it into a splay tree, and preserve the red pointers. I'll write that down.

These are called auxiliary trees. This was exactly the idea and tango trace and I'm using tango tree terminology to keep it consistent with that notation. It wasn't called preferred paths in the link-cut tree papers. Or it wasn't called auxiliary trees, but same difference. So auxiliary trees were going to store represent each preferred path by a splay tree. Tango trees we use red-black trees or whatever.

Could us splay trees, actually, if you use tango trees with splay trees for each preferred path. It's called a multi-splay tree. Multi-splay trees are almost identical to link-cut trees but, they're used for a different setting. The use for proving log n competitiveness. Here we're doing it to represent a specific tree. With tango trees, the tree we're representing was just a perfect binary tree, p. So you may recall, here it has some meaning. And it's going to be changing, and we care about how it's changing.

OK, another difference is we're going to key the nodes by depth. This is what we wanted to do with tango trees, but we weren't allowed to because we had to be a binary search trees. And the keys were given to us. Here, there are no keys. So we need to specify them and the most convenient one is by depth. So we're just taking a path like this, 0, 1, 2, 3 4. And then we're turning it into a nice balance tree like this, 0, 1, 2, 3, 3, 4. OK, Now, it's a splay trees so it's not guaranteed to be balanced at any moment, but amortized is going to have log n performance preparation.

OK, and then this is sort of that's representing each of these white paths, but then we also need the red pointers. So what we're going to do is say the root of each auxiliary tree, I'll abbreviate ox tree, stores what I'll call the path parent, which are these red pointers. The definition is this is, the paths, the top nodes, parent, and represented tree.

OK, very soon it's going to get confusing and I'm going to always make it explicit. Am I talking about the represented tree the thing we're trying to represent? Or am I talking about auxiliary trees because notion of parent is different in the two? So in an auxiliary tree a parent is just whatever it happens to be stored in the splay tree. When you splay operations you care about those parents. When you're thinking about the tree you're trying to represent, you care about parents in the represented tree.

Now, how do you see that here? Well following parent is like doing predecessor over here. So we kind of know how to do predecessor in a binary search tree. But when we get to 0, which is the left most node in this tree, corresponds to the top of the path, if we want to do the parent that's going up from this path, that's going to be the path parent.

And it saying the path's top node, that's this guy 0, its parent and the representative tree are just going to store that pointer. The only weird thing is we're not going to store the pointer here, we're going to store the pointer here. That's the path parent. We're going to put it at the root of the splay tree, basically because it's really easy to get to the root of a tree. You could probably start here, you just have to go left every time and it's kind of annoying. Analysis will get messier. So we'll put at the root.

And so, in particular, if I do a rotation, like if I rotate the tree like this and make one the route, that path parent pointer has to move to 1. But it's still representing essentially for this whole splay tree representing this path, what was the parent pointer from there. So this is the represented tree, and this is the ox tree.

OK, now if I take all the ox trees plus these parent pointers, I get something called the tree of ox trees. And this is our representation. So there's the represented thing versus our representation. I'm not going to use the word representation because it sounds almost like represented. So it's going to be represented versus tree of ox trees. Tree of ox trees is what we store, represented is what we want to store. That's what we're trying to represent. So that's the terminology, Basically, we want the tree of ox trees to be balanced, and if there's splay trees, they'll be kind of balanced. Whereas, the represented tree is on balance.

Cool so now I want to give you some code and how we're going to manipulate these things. And then we'll be able to analyze the code and that both will take a little while. The main operation I'm going to talk about is actually just a helper operation.

So first I want to tell you what it is, how it works. It's like an access in a tango tree. With tango trees we just wanted to touch item xi and then go onto the next item, xi plus 1. So I'm going to think about that world, while all I'm trying to do is touch nodes. What's interesting about touching nodes is it changes this notion of preferred edges.

So, as I said, the last access in that subtree was blah, blah, blah. So access. In reality, what I mean is the last time that node was used as a link or cut or find operation. But in fact I'm going to make that explicit. Every operation is going to start by calling this function access on its arguments. Question?

AUDIENCE: I still don't understand. When you say that this [INAUDIBLE] by depth--

PROFESSOR: Yes.

AUDIENCE: How do you go from that path to the actual splay tree?

PROFESSOR: OK so the transformation from this path in the represented tree to splay tree is just these nodes have depth within that path, I mean they're just stored along the path. What I mean is, I want to store them in as a binary search tree ordered by that value. So that's what-- yeah. So splay trees have some key on the nodes, they maintain the binary search tree property. My key is just going to be the depth. The position along the path.

AUDIENCE: So if you do an in order traversal you just get the path.

PROFESSOR: Right if I do it in order traversal here, I'll get the path back. Yep, no problem. That's going to be important to understand. OK, so to do an access, access is going to do two things. The first thing it has to do, and the other thing is going to make our life easy, you'll see once we've defined access, all operations become trivial. So it's going to be a really cool helper function.

The first thing we have to do, when we access a node v, is say well that then by definition the path to v from the root --v to root I guess, sort of, but probably more root to v-- should be preferred. That's the definition of preferred because now it's the latest thing accessed. So we've got to fix that.

But we're going to do a little bit more we're also going to make v the root of its ox tree. Which will mean, because that ox tree is now the top most ox tree because that actually contains the root and it contains v. So in fact, v is the root of the tree of ox trees. It's the overall root for this tree of our streets. Why do we do that? Well, it's sort of going to just be a consequence of using splay, because whenever you splice something you move it to the root in this zigzaggy way. But it will actually be really helpful to have this property, and that's why these are the simplest link-cut trees I know.

OK, so how do we do this? So you're given the node v somewhere in this world, and we've basically got to fix a lot of preferred child pointers. And, whereas, tango trees basically walked from the top down, navigating one preferred path until it found the right place to exit and then fixing that edge, were going to be working bottom up. Because we already know where the vertexes, were given it, and we need to walk our way up the tree and kind of push v to the root overall.

So the first thing to do is splay v. So that, now, when I say splay v, I mean within its ox tree. Doesn't make sense to do it in any global sense. In some sense access is going to be our global version of splay. Whenever I say splay a node, I mean just within its ox tree. That's going to be the definition of splay tree. So you may recall from Lecture Six there's the zig-zig case and zig-zag case. You do some rotation, double rotation, whatever. In the end, maybe one more rotation, and then v becomes the root.

So what does the picture look like? We've got v at the root of it's ox tree. Then we've got things in the left subtree which have smaller depth. And then we've got things in the right subtree of tree which have bigger depth. Depth bigger than v. So we just pulled to the root. So in the represented tree what we have is a path v. These are things of smaller depth these things of larger depth. Now, if you look at the definition of preferred child, when we access v, v now no longer has a preferred child. This preferred child is now none. If you look at this picture, this is a preferred path, currently, v has a preferred child. We want to get rid of that. That's our first operation.

First thing we want to do is get rid of that edge. It's no longer a preferred edge. So that essentially corresponds to this edge because that's the connection from v to deeper things. I mean in fact, this node right below us let's call it x is right here. It's the smallest depth among the nodes with larger depth than v.

But if we kind of kill that connection, then, now, this path is its own thing, the part below v, separate from v and above. So that's what we're going to do first. Remove these preferred child. And I'm going to elaborate exactly the point arithmetic that makes that happen. This is the kind of tedious part of, or really any point of machine data structure. This is going to all work on a point of machine, by the way.

OK, so basically I'm obliterating the edge. So I'm going to make this vertex here, have a path parent pointer to v and otherwise obliterate this edge. It will no longer has a parent, v no longer has a right child. So it's going to be v has a left thing and no right pointer. But this separate tree is going to a parent pointer to v.

This is, of course, if it exists. If v.right is nothing already, then you don't do any of this. But if there is a right pointer then you've got to do this. So this is the place where we're going to set path parents, pretty obvious stuff. The only thing to check is that this is really in the right place. This thing is the root of this ox tree, which is where the parent pointer path parent supposed to be, and it points to v meaning, that the parent of x is v in the represented tree. And that's exactly what this picture is. This is the represented tree parent of x is v, and so that's the correct definition of pat parent.

Cool. OK, now the fun part. Now, we're going to walk up the tree. We've seen everything on the outline except this heavy light decomposition. We'll come to that soon. OK, so now we're going to walk up the tree and add new preferred child pointers. This is the one that we had to remove because it's the stuff below v. But up the path, if we walk up to the root here, to the left most node of v then there's a path parent from there, we want to make that not a path parent but a regular parent. Because that right now, is living in its own little preferred path that we want that preferred path to extend all the way to the root, by the definition of a child after doing an access.

So we're going to do a loop until v path parent is none. I'm going to let w be v's path parent. save some writing. And then we're going to splay w. OK, I think at this point, I should draw a picture. So many pictures to draw. So we have, let's say, a node w as a child v. This is the represented tree. That's some other child x and I can have many children.

Let's suppose that, right now, the preferred child from w is x. It's not v because we just followed a path parent pointer to go from v's path, which is something here, to w's path. So w's path is going to be something like this. So it goes to some other guy x, we want change that that pointer. Instead of being that, we want to go here. That's in the represented tree. Now what does this look like in the tree of ox trees? W lives in some thing. w's there, x's its successor in there, so maybe it's like a right child or whatever, somewhere else in the tree.

And then separately we've already built this thing. v it has a left child, it has no right child because there's nothing deeper than v in its own preferred path. And then we have a path parent pointer that goes like this. I want to fix. That the first thing I'm going to do is splay w, so the w is in the root of its own tree. So then it's going to look like w.

It's going to have a left child, left subtree, right subtree. X is going to be its successor so it's the leftmost thing in there. And we still have v with its left child and this pointer. It's not in the tree. So if two ox trees, now I want to basically merge these two ox trees, but also get rid of this stuff. Just like we did up here actually. First, I have to destroy this preferred child and then make a new one, which is v.

So how do I do it? I just replace the right pointer from w to be v instead of whatever this thing is that's it. So say, switch w's preferred child to be v. And what we do is say w, first we clip off the existing guy, we say this node's w.rights path parent is now going to be w.

We still want to have some way to get from here back up to w, just like we did up here. This code is going to look very similar to these two lines. We set it's parents to none. And we set w.right. Up, here we set it to none. Now we know it actually needs to be v. And then we set the reverse pointer so v's parent is now w and v no longer has a path parent.

So essentially the reverse of these operations. So we remove this preferred child pointer and we add this one in. So the new picture will be, we have w has its left subtree just like before. It's right subtree is now v with its left subtree. And then we also have this other tree hanging out. Not directly connected except by path parent pointer. So whereas before, v was linked in with the path parent pointer, now this thing is linked to in the parent pointer and sort of the primary connection, the white connection is direct from w to v. And that's exactly what corresponds to clipping off this portion of the preferred path and concatenating on this portion of the preferred path.

More or less clear? You can double check this at home but that's what we need to, do. To go back and forth between these two worlds. The represented world and the tree of our world is doing the right thing. Again you can check that this path parent is, indeed, the right thing.

It's essentially saying the parent of x equals w, which is, indeed, the case. That left most thing here, its parent is w. Cool last thing we do is kind of a lame way to say it, but I want to splay v within its ox tree Now, v is a child of the root. So this just means rotate v. So what it's going to look like is v becomes the root, it's left child is w, that left child is whatever, that right child is whatever.

V will still have no right child because v is the deepest node in it's preferred path. So there's nothing deeper than it. There's nothing to the right of v. So these two triangles go to here. Of course, there's still the old triangle with x in it, and it's still going to prefer it's still going to point to w. But we want v to eventually end up to be the very, very root so I'd like to make the root again after I did this concatenation.

And so there you go you can think of this as a second splay if you want or as a rotation, either way. And now we're going to loop. OK, the one thing I didn't make explicit is that when we do a splay, or when we do a rotation, you have to carry along the path parent pointer. So like right now, sorry, in this picture, w has some path parent pointer because it's the root of its ox tree. After we do the rotation, v is the root and so it's going to have the parent path parent point. Or you can just define rotate to preserve that information.

So now v has some new path parent and we do the loop. As long as it is not there's something there, we're going to splay it stick them together. Repeat, the number of times we repeat, is equal to the number of preferred child changes. So that's how we're going to analyze the thing. How many times does a child have to change in this world.

OK, clear, more or less that is that's the hard the hard operation access. One thing to note is v will have no right child at the end, and it will be the root of the tree of ox trees. Why the tree of ox trees? When we stop, we have no path parent. That means we are the overall root. So that's the end.

Cool now let me tell you how to do-- first, I'll do the queries, and then I'll do the updates, link-cut. Cut Make tree, I'm pretty sure, if you all think you know how did you make tree. So let's start with find root. So for find root, first thing we're going to do is access v.

So what that gives us is v, no right child, some left child. This is v's ox tree, but the roots of the overall tree is right here. This is r. Because, in the end, we know that, the mean the root ox tree always contains the root node of the tree, and the access operation makes it also contain v So what's the highest node in that path from r to v? Well, it's the left most node in the thing. So you just block left to find the root r.

And then, we are going to do one more thing, which splay r. If we didn't splay r, we'd be in trouble because this is a splay tree. If you say this r might be extremely deep in the tree. And so if you just repeatedly said, find root of v, find v for the same v, you don't want to have to walk down linear length path every single time. So we're going to splay it every time we touch the root, so that very soon r will be near the root of the tree of ox trees, and so this operation will become fast. So amortized it will always be order log n. But we need to do that.

OK that's fine root. Let's do half aggregate. Of aggregates basically the same. Actually, even easier. First thing we do like all of our operations, is access v, so we get that picture. So remember what this corresponds to? This is an ox tree that represents a path ending in v So this is in the represented tree.

And the goal of a path aggregate is to compute the min or the max or to some of those things. So I basically have this tree that represents exactly the things I care about, and I just need to do a sum or a min or a max or whatever. So easy way to do it is augment all the ox trees to have subtree size, subtree sums, or mins or maxes, whatever operations you care about. And so then it's just a return v.subtree min max, whatever.

You have to have to check when we do link-cuts that it's easy to maintain augmentation like this, but it is. And now this, the subtree aggregations are relative to your own ox tree. You don't go deeper. In fact, there's no way to go deeper because if you look at the structure, there's no way to go down. We store these path parents, but we can't afford to store path children because a node may have a zillion path children. So it's kind of awkward to store them. we don't have to because this aggregation is just within the ox tree. That's exactly what we care about because it represents that path and nothing more.

So you see, access makes our life pretty darn easy. Once that path is preferred, we can do whatever we need to with that path. We can compute aggregations, the root, anything pretty much instantly once we have access. OK, I claim also link and cut are really easy. So let me show you that.

So let's start with cut. First thing we do, you guessed it, access v. So think about this picture a little bit. So we have v, have this stuff. What this corresponds to, this is the ox tree in the represented tree. This is a path from the root to v. And our goal in the cut operation is to separate v from its parent in the represented tree. So we want to remove this edge above v. That basically corresponds to this edge. Connection from v to all the things less deep than it. So this is the preferred path.

Of course, in reality, there is some subtree down here, and that's going to correspond to things that are linked here by path parent pointers. So they'll come along for the ride, because what they know is they're attached to v. And so all we need to do is delete this edge and we're done. It's kind of crazy, but it works. So what we do, say, v left the parent is none. The left is none. That's it. Gone. That edge has disappeared. What will be left with is v all by itself. / Has no left child, no right child. It still has some things that link into it via path parent pointers.

But v is alone in its ox tree after you do a cut. This thing will now live in a separate world. In particular, this node becomes the new root of the tree of ox trees for this tree. after we do the cut, there's two trees. So there's two trees of ox trees. There's the one with v, v will remain the root of its tree of ox trees. And the other one, this thing called x, becomes its own root of its own tree of ox trees. So there's nothing else to do. It doesn't need a parent pointer because it's not linked to anything above it. The end.

OK, I corresponds to some node up here. Kind of the median node. So there you go. That's a cut. It's like super short code. You have to stare at it for a while and make sure it does all the right things, but it does. How about a link? Well first thing we do on a link, is access v and access w.

These don't interfere with each other because precondition is that v and w are in different trees of ox trees. They're indifferent represented trees. So they're completely independent. The result will be that we have-- should be consistent --so ox trees are on the left. We're going to have w, is going to have a left thing.

We're going to have v claim all by itself because remember what a link does? It's right here, v is assumed to be a root node. So if you do an access on the root node, then the path from the root to v is a very short path. It is just v itself. So the ox tree containing v, will just be v itself when you do x as v. Yeah, so this is the picture in represented space.

And so we access v, we're going to have this. Of course, there's stuff pointing into it. We're at access w, so it's going to look like this. And then this path is going to be what's over here. And there's, of course, more stuff linked from below into those. Our goal is to add this edge between v and w, which corresponds to adding this change. So that's what we're going to do. V.left left equals w. W.parent equals v.

If you want, you could instead make v the right child of w. And that looks much more sane. I like it this way because it looks kind of insane. This is not unbalanced or anything. But splay trees will fix it so you don't have to worry. This is the carefree approach to data structuring. And you just leave it to the analysis to make sure. Everything here is going to be log n amortized.

But you can check this is doing the right thing, because v is deeper than w, right? So we had this path from the root over here to w. We're extending the path by one node, v, and so v should be to the right of everything. So either it goes that here is that right child w that would also work, or would make it the parent of w on the right that works. It's in the correct order, binary search tree order. OK?

So you see length and links and cuts are easy. In fact, the most complicated was find root where we had to do a walk and splay, but basically, everything reduces to access. If access is fast, all these operations will be fast because they spend essentially constant time plus a constant number of accesses. Except for find root. It also doesn't splay. But we're going to show displays are efficient as, well as part of access, because access does a ton of splays. So this is log n amortize, surely one splay is log n amortized. And indeed, that will be the case.

AUDIENCE: Splay as access.

PROFESSOR: Here, I'm treating splaying not as an access. So I'm going to define access to mean calling this function.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK so over here, access first splays v and displays various things. It might not splay r.

AUDIENCE: No, I'm saying, do you call access r [INAUDIBLE]?

PROFESSOR: Oh good. That might simplify my analysis. I'll just change this line to access r. Good, why not? Yeah, that seems like a good way. OK, I think you could do that. It might simplify, conceptually, what's going on. The one thing I find annoying about it's just-- it's an aesthetic, let's say.

So here, I was talking about the last access. And if you define lost access to mean access, that's fine. But also another intuitive notion of the last access, is the last time it was given to any of these functions cut link find root path aggregate. And so r is not really given to the function. Of course, it's the output of find roots, so maybe you think of that as an access. You could say find root is accessing the root. Either way, this should work either way, but I think I like that. You just have to redefine things a little.

OK, let's do some analysis. This is the data structure. Algorithms are all up there in they're gory detail. This is. Of course. goriest. I can now erase the, API. Makes me think of Google versus Oracle. And then following that case? Yeah, it's interesting. OK pretty clear this implementation is much more significant than the API, but anyway. If

First goal is to prove log squared. This is just like a warm up. This is actually trivial at this point. We've done all the work to do a lot of squared bound. In fact, you just replace the slay tree with a red-black tree, and you know that each of these operations is Log n. All displays that we do, I mean, you can do essentially like a splay in a balance binary [INAUDIBLE] tree, you can still move it to the root. You can still maintain it temporarily and maintain the height is order log n.

That's one way to view it. Or you can observe the splay tree analysis that gives you log n, which we haven't actually covered in this class. Still applies in this scenario. Even though it's not one splay a tree. It's a bunch of trees. You can show each of the splays is order log n.

So what that gives you is that-- so we have lets just say it's order log n amortized per splay, a few splay trees or use regular balancer trees is definitely log n. So then if we do m operations, it's going to cost --we're not actually going to be done-- order log n times m plus total number of preferred child changes.

This bound should be clear, because every operation reduces to accesses plus constant amount of work. Maybe one more splay a splay is just another log n. And the total number of preferred child changes comes from this access thing. We're doing one splay per preferred child change. So that thing is reasonable, you just take it, multiply by log n, we're done.

So the remaining thing is, at this point, just claim. Total number of preferred trial changes is order m log n. So if you take this whole thing, divide by m, you get log squared amortize. So that sounds kind of lame, log square is not such a good bound, but it's a warm up. In fact, we need to prove this anyway. We need this for the log n analysis. So first thing I'm going to do is prove total number of preferred child changes is log squared. Before I do that, I need heavy-light decomposition. So to prove, this we're going to use heavy-light decomposition. And this I think, is where things get pretty cool.

So heavy-light decomposition, this is another way to decompose a tree into paths. So heavy-light decomposition is, again, going to apply to the represented tree. Not the tree of ox trees. It's like an intrinsic thing. It's very simple we define the size of the node to be the number of nodes in that subtree. We've done this many times, I think.

And then, we're going to call an edge from v to its parent heavy or light. It's heavy if the size of v is more than half of the size of its parent. And, otherwise, it's called light. OK, so we have parent and sub child v has loads of other children.

I just want to know, is the heaviest of all your children? That's one way to define it. But, in, particular is a bigger than half of the total weight of p? So there might not be any heavy child according to this definition. Maybe it's nicely evenly balanced. Everybody's got a third. But if somebody has got bigger than 1/2, I call it heavy. Everybody else is light. So there's going to be, at most, one heavy edge from every node. Therefore, heavy edges decompose your world into path's. Heavy paths decompose the tree.

Nodes, every node lives in some heavy path. It may be node has no heavy childs, but, at most, one. This may seem kind of silly, but in fact, because your maybe all edges are light. This might not do anything, every node is in its own path, that's actually a really good case, light edges are good.

Why are they good? Because then the size of v is at most half the size of its parent. I mean, every time you follow a light edge, the size of your subtree went down by a factor of 2. How many times is going to happen? Log n times. Start with everything at the root, as you walk down, if you're decreasing your side effect or two every time, you can only follow log . Light edges.

This is what we call the light depth of a node. This is the number of light edges on a root to the path. And it is always, at most, log n. Now, remember heavy edges could be huge. Maybe you follow, maybe your tree is a path. Then every edge is heavy. And you n of them to get from the root of v. So we can't bound the number of heavy edges.

But the number of light we can bound as log n. This is where heavy light to composition is useful. So you've got preferred path composition. Our Or data structure, we're not going to change it. It's still following the preferred path the decomposition. But our analysis is going to think about which edges are heavy and which are light.

In general, an edge can have four different states. It can be preferred or not preferred. And it can be preferred or not preferred, it could be heavy or light. All four of those combinations are possible, because one of them has to do with the access sequence, the other has to do with the structure of the tree. These are basically orthogonal to each other.

OK, so let's maybe go here. So next thing I'm going to do is use heavy light decomposition to analyze total number of preferred child changes, by looking at not only whether an edge is preferred or not, but whether it is heavy or light. We're going to get an order m log n bound on preferred child changes.

So here's the big idea. In order for a preferred child to change, I mean when you change for a child of one node from this to this, I'm going to think of this edge as being destroyed from its preferredness, and this one is being created in it's preferredness.

OK, so of course, if we could count preferred edge equations, that would be basically the same as preferred child changes. Or we could count preferred edge destructions, that would be basically, the same as the number of changes. OK, so that's the idea. If you ignore this part, or look at pilferage equations or prefer it is destructions. But then I also care about whether that edge is light or heavy.

So it turns out for the light edges, it's going to be easier to talk about the creations or to bound the creations. For heavy edges is going to be easier to bound the destructions. If I do both of those and add them up, that in total is, basically, the number of preferred edge changes according to whether there is light or heavy.

You need to add in a little bit more because and edge might get created but never destroyed. And if it's heavy, then it won't get counted here. Or an edge might get destroyed because it exists originally, never got created but it got destroyed over the operations. And if it was light, then it wouldn't be counted here. So just add on n plus 1 for all the edges might get a bonus point. But, otherwise, this will bound it, it's going to look at light edge creations-- creations, light preferred edge creations, heavy preferred edge distructions.

And it can be destroyed because it becomes no longer heavy or no longer preferred. And it can be created because it becomes light or it becomes preferred and was already the other one. OK, so all we need to do is think about, in all these operations, access link-cut, just the updates. So access link/cut. How can this change?

So let's start with access. So access. The hard one. Well, there's all this implementation, which is dealing with the tree of ox trees, but really, an access does a very simple thing. It makes the path from the root to be preferred. That's its goal. So there's the root, it's to v, this becomes preferred whether it was before or not. Some of it was before some of it wasn't. So some of these might be on newly preferred.

It does not change which edges are heavy or light. It does not change the structure of the tree when you do an access. Only link/cut changed the structure of the tree. So it makes some of these edges preferred. What that means, of course, is that some other edges used to be preferred and are no longer preferred. Those are the preferred child changes. So if we look at this, first concern is that we're creating new preferred edges, so maybe we create some new light preferred edges. How many new light preferred edges could we create along this path? Most log n. Whatever the light depth of v is as an upper bound.

So make, at most, log and, like preferred edges, because they all live on a single path. And it would be really hard to bound how many heavy preferred edges we make. But we don't have to. We don't have to worry about heavy preferred edges being destroyed. Now, those edges could be these ones. These edges used to be preferred, now they're not. If they were heavy, then I have to pay for them.

But the number of these edges is equal to, or is at most, the number of these edges. I mean, if this edge was heavy, every node has only one heavy down pointer. So if this edge is heavy hanging off, then this one is light. We know there's only log n light edges here. So the number of heavy edges coming off here can also be most log n,

So destroy, at most, log n heavy preferred edges. Yeah In some sense, we don't even really are about the word preferred here. But, of course, we're not making them, light so anyway. Is that clear? Now, there's actually one more edge that could change which is, there used to be a preferred edge from v, now there isn't. We destroyed that in the very beginning of the operation. So maybe plus 1 that might have destroyed one more heavy edge. But overall order log n. So, actually, really easy to analyze access. Any questions about that?

OK, link/cut not much to say here. Yeah, so link/cut don't really change what's preferred. I'm analyzing-- I'm going to think about what link does after it accesses v and w. Because access we've already analyzed. So link is just these two operations which add a single pointer.

So if you look at what that's doing in the represented tree, which was v was the route and we made it to be a new child w, which lived in some tree over here. That's in the represented tree, what happens. What happens is that w gets heavier. I guess also this get's heavier. All the nodes on this route to w path get heavier. That's all that happens in terms of heavy, light. Preferred paths don't change. It's just about heavy and light changing.

OK, in fact, this will be a preferred path from the root to v because we just accessed those guys. So we're heavying edges, which means you might create new heavy preferred edges. But we don't care about heavy preferred edges being created. We only care about them being destroyed.

So has anything changed? Well, there might have been a heavy edge hanging off of here. Is that possible? Yeah, so this might turn light, because this one became heavier. So this edge might become light, which means we might have potentially lost-- this could have been preferred before. It wasn't preferred, so who cares? If we've already done the accesses, the edges are already not preferred. So not a big deal. So can anything happen? I think not.

Some of the edges on the path might become heavy, but we don't care because they are preferred. Some of the edges off the path might become light, but we don't care because they're not preferred. Done. So this actually costs zero in just analyzing number of preferred child changes. Cuts, not quite so simple. When we do a cut, we're lightning stuff.

The path from the root to v in the represented tree, when we line up there, becomes lighter, because we cut off that whole subtree containing v. So we might create light preferred edges on that path. And that's something we actually want to count. we count number of light preferred edges created. But, again, they're on a path so it's, at most, log n

Most log n light preferred edges created. We don't care about edges hanging off that path because they're not preferred anymore, so it's nothing to talk about. So that's it. That proves m log n preferred child changes. It's amortized because we're doing this creation/destruction business. This thing is worst case log n per operation, this quantity. But when you sum it up, then you actually get about the number of preferred child changes overall.

So if you plug that into this bound, we get a log squared m bound. Big deal. But with a little bit more work, we can actually get a log n bound. For this, we need to actually know a little bit about splay tree analysis. I didn't cover in Lecture Six. You probably would have forgotten it by now anyway. So let me give you what little you need to know about splay analysis. First, I need to define a slightly weird quantity. It's kind of like the weight of a node. It's a capital W, but a little bit weird.

OK, now we're thinking about the tree of ox trees. For the analysis we need that. It's a tree of splay trees. And what I'm looking at, is in that tree of ox trees, how many nodes are in the subtree of v? That's what I'm going to call wv . Whereas size of v was thinking in the represented tree. So this is a totally different world.

Here, we're thinking about the tree of ox trees. So one way you can rewrite this count as the sum of all nodes w in the ox tree containing v. It's a weird notation, but look at all the other nodes in the ox tree-- is that right? Sorry. No. I should say w in v's subtree in v's ox tree. It's awkward to say. What I mean is there is an ox tree containing a node v, and I want to look in that subtree I'll know as w. Just within the ox tree though. And for all those nodes, I take 1 plus the size of ox trees hanging off.

So total number of nodes. And when I say hanging off I mean via pathed parent pointers. So down here, there are trees, add up all their sizes total number of nodes in them, add one for w itself, that's another way of writing this. I'll just mention that this part is what's normally considered in splay trees. This is a bonus thing that, basically, doesn't matter. I'll justify that in a second. So we define a potential function for our amortization fee, which is sum over all nodes v of log this quantity wv, this thing.

So just think of this as an abstract quantity, which for every node it has some number associated with it. Splay trees allow you to assign arbitrary weight to every node, use this potential function, and prove abound --which is called the access lemma The access lemma says is that, for this potential function, the amortized cost of doing a splay operation splay of v, is, at most, three times log w of root of v's ox tree minus log w of v plus 1.

OK, so this is something called the access Lemma It's used to prove, for example, that splay trees have log n performance amortize. It's also used to prove that they have a working set bound, which you may recall from Lecture Six. But we never actually mentioned the access lemma It's a tool. It's an analysis tool.

This works no matter how the w's are defined. And remember, we're thinking of splaying just within a single ox tree. So the size of the ox trees hanging off don't change during a splay. They just come along for the ride. So the old analysis of splay trees I haven't proved this lemma to you. But it still applies in this setting. And given my lack of time.

I will just say in words, the way you prove this lemma is very simple. You do you analyze each operation of display separately there's a zig-zag case in a zigzag case. And you argue that every time you do such an operation, you pay three times a log of w of v after the operation minus log of wv before the operation. So you just see how wv changes when it goes up after you do display operation, and turns out it's the most three times log of that. And it has to do with concavity of log n. It's just basic checking.

Once you have that you get a telescoping sum. Each operation is log w new minus long w old. Those cancel, in turn, until you get the final log w of v, minus the original log w of v. The final log w is whatever-- I mean, v becomes the root and so it has everybody below it. So that's the axis lemma.

So assuming the access lemma, I want to prove to you a log n bound. Maybe over here OK, another thing to note, when we change preferred children, it does not affect w. W defined on the tree of ox trees. If you turn a path parent pointer into a regular parent pointer or vice versa, It doesn't care. It looks the same from the tree of ox trees' perspective. All that changes, all that matters is when you do splays this stuff happens. But the splay analysis tells us how splays behave. So we're kind of good.

If you look at what we did over here, when we splayed v, then we splayed w, then we did a little bit of manipulation which doesn't matter in this analysis. And then we splayed v one more time. How much does that cost according to the access lemma. It's basically going to cost you order log w of little w minus log w of v plus 1, because well is it clear? When we do the first splay of v going up to the root that cost log of w of the whole tree containing b minus longer wv.

But if you just look at log, I mean, w is higher than the root of v. So if we take a log of w this the whole thing from w downwards minus log wv, that includes the cost that we did for the initial splay of v. Then when we played w, well that's basically the same thing but the next level up.

So if you look at log of w minus log of w of next level up, that's what it's going to cost this splay w. To splay v again, will again cost this bound. The point is, we sum this up over all the preferred child changes. And what we get is a telescoping scum again. Same as in the display analysis. So we end up with order log w of everything, which an n minus, I guess, the log of w of the original v, -but we don't really care about that-- plus the number of preferred child changes.

And now we're golden because before, the obvious bound was number preferred child changes times again, now it's number of preferred time changes plus log n for an entire access. And so we pay log n here. We already know the amortize number preferred child changes is log n per operation. So overall the amortize cost per operation is log n. And we're done.

So I'll just mention the worst case version of link-cut trees instead of the amortize splay based version. They actually store the heavy light decomposition. They don't use preferred path to decomposition at all, which makes all the algorithms messy. But you can just maintain the heavy [? likely ?] composition position dynamically as the tree is changing, and then you use a kind of weight balanced trees, like we saw in the strings lecture, where the depth of a node is sort of related to log of its size or inversely related I guess.

So you try to put all the heavy things near the root, because they're more likely to be accessed. And then you guarantee that the overall tree of ox trees has long n depth by skewing each of the end of the ox trees to match. So it can be done but all the operations are messier, because you no longer have the convenience of preferred paths to make it easy to link/cut things.