MITOCW | 15. Static Trees

PROFESSOR: All right. Today, we're going to look at some kind of different data structures for static trees. So we have-- at least in the second two problems-- we have a static tree. We want to preprocess it to answer lots of queries. And all the queries we're going to support today we'll do in constant time per operation, which is pretty awesome, and linear space. That's our goal. It's going to be hard to achieve these goals. But in the end, we will do it for all three of these problems.

So let me tell you about these problems. Range minimum queries, you're given an array of numbers. And the kind of query you want to support-- we'll call RMQ of ij-- is to find the minimum in a range. So we have Ai up to Aj and we want to compute the minimum in that range. So i and j form the query. I think it's pretty clear what this means.

I give you an interval that I care about, ij, and I want to know, in this range, what's the smallest value. And a little more subtle-- this will come up later. I don't just want to know the value that's there-- like say this is the minimum among that shaded region. But I also want to know the index K between i and j of that element. Of course, if I know the index, I can also look up the value. So it's more interesting to know that index.

OK. This is a non-tree problem, but it will be closely related to tree problem, namely LCA. So LCA problem is you want to preprocess a tree. It's a rooted tree, and the query is LCA of two nodes. Which I think you know, or I guess I call them x and y. So it has two nodes x and y in the tree. I want to find their lowest common ancestor, which looks something like that. At some point they have shared ancestors, and we want to find that lowest one.

And then another problem we're going to solve is level ancestor, which again, preprocess a rooted tree and the query is a little different. Given a node and an integer k-- positive integer-- I want to find the kth ancestor of that node. Which you might write parent to the k, meaning I have a node x, the first ancestor is its parent. Eventually want to get to the kth ancestor.

So I want to jump from x to there. So it's like teleporting to a target height above me. Obviously, k cannot be larger than the depth of the node. So these are the three problems we're going to solve today, RMQ, LCA, and LA. Using somewhat similar techniques, we're going to use a nice technique called table look-up, which is generally useful for a lot of data structures. We are working in the Word RAM throughout. But that's not as essential as it has been in our past integer data structures.

Now the fun thing about these problems is while LCA and LA look quite similar-- I mean, they even share two letters out of three-- they're quite different. As far as I know, you need fairly different techniques to deal with-- or as far as anyone knows-- you need pretty different techniques to deal with both of them. The original paper that solved level ancestors kind of lamented on this.

RMQ, on the other hand, turns out to be basically identical to LCA. So that's the more surprising thing, and I want to start with that. Again, our goal is to get constant time, linear space for all these problems. Constant time is easy to get with polynomial space. You could just store all the answers. There's only n squared different queries for all these problems, so quadratic space is easy. Linear space is the hard part.

So let me tell you about a nice reduction from an array to a tree. Very simple idea. It's called the Cartesian tree. It goes back to Gabow Bentley and Tarjan in 1984. It's an old idea, but it comes up now and then, and in particular, provides the equivalence between RMQ and LCA, or one direction of it. I just take a minimum element-- let's call it Ai-- of the array. Let that be the root of my tree. And then the left sub-tree of T is just going to be a Cartesian tree on all the elements to the left of i. So A less than i, and then the right sub-tree is going to be A greater than i.

So let's do little example. Suppose we have 8, 7, 2, 8, 6, 9, 4, 5. So the minimum in this rate is 2. So it gets promoted to the root, which decomposes the problem into two halves, the left half and the right half. So drawing the tree, I put 2-- maybe over here is actually nicer-- 2 at the root. On the left side, 7 is the smallest. And so it's going to get promoted to be the root, and so the left side will look like this.

On the right side, the minimum is 4, so 4 is the right root, which decomposes into the left half there, the right half there. So the right thing is just 5. Here the minimum is 6, and so we get a nice binary tree on the left here. OK. This is not a binary search tree. It's a min heap. Cartesian tree is a min heap.

But Cartesian trees have a more interesting property, which I've kind of alluded to a couple of times already, which is that LCAs in this tree correspond to RMQs in this array. So let's do some examples. Let's say I do LCA of 7 and 8. That's 2. Anything from the left and the right sub-tree, the LCA is 2. And indeed, if I take anything, any interval that spans 2, then the RMQ is 2. If I don't span 2, I'm either in the left or in the right. Let's say I'm on the right, say I do an LCA between 9 and 5. I get 4 because, yeah, the RMQ between 9 and 5 is 4. Make sense?

Same problem, really, because it's all about which mins-- I mean in the sequence of mins-- which mins do you contain? If you contain the first min, you contain the highest min you contain. That is the answer and that's what LCA in this tree gives you. So LCA i and j in this tree T equals RMQ in the original array of the corresponding elements. So there is a bijection between these items, and so I and J here represents nodes, and here corresponding to the corresponding items in A.

OK. So this says if you wanted to solve RMQ, you can reduce it to an LCA problem. Quick note here, which is-- yeah. There's a couple of different versions of Cartesian trees when you have ties, so here I only had one 2. If there was another 2, then you could either just break ties arbitrarily and you get a binary tree, or you could make them all one node, which is kind of messier, and then you get a non-binary tree.

I think I'll say we disambiguate arbitrarily. Just pick any min, and then you get a binary tree. It won't affect the answer. But I think the original paper might do it a different way. OK. Let's see. So then let me just mention a fun fact about this reduction, which is that you can compute it in linear time. This is a fun fact we basically saw last class, although in a completely different setting, so it's not at all obvious. But you may recall, we had a method last time for building a compressed trie in linear time. Basically, same thing works here, although it seems quite different.

The idea is if you want to build this, if you build a Cartesian tree according to this recursive algorithm, you will spend n log n time or actually, maybe even quadratic time, if you're computing min with a linear scan. So don't use that recursive algorithm. Just walk through the array, left to right, one at a time. So first you insert 8. Then you insert 7, and you realize 7 would have would have won, so you put 7 above 8.

Then you insert 2. You say that's even higher than 7, so I have to put it up here. Then you insert 8 so that you'll just go down from there, and you put 8 as a right child of 2. Then you insert 6. You say whoops, 6 actually would have gone in between 2 and 8. And the way you'd see that is-- I mean, at that moment, your tree looks something like this. You've got 2, 8, and there's other stuff to the left, but I don't actually care. I just care about the right spine.

I say I'm inserting 6. 6 would have been above 8, but not above 2. Therefore, it fits along this edge, and so I convert this tree into this pattern, and it will always look like this. 8 becomes a child of 7-- sorry, 6. 6. Thanks. Not 7. 7 was on the left. This is the guy I'm inserting next because here. So I guess it's a left child because it's the first one. So we insert 6 like this. So now the new right spine is 2, 6, and from then on, we will always be working to the right of that. We'll never be touching any of this left stuff.

OK. So how long did it take me to do that? In general, I have a right spine of the tree, which are all right edges, and I might have to walk up several steps before I discover whoops, this is where the next item belongs. And then I convert it into this new entry, which has a left child, which is that stuff. But this stuff becomes irrelevant from then on, because now, this is the new right spine. And so if this is a long walk, I charge that to the decrease in the length of the right spine, just like that algorithm last time.

Slightly different notion of right spine. So same amortization, you get linear time, and you can build the Cartesian tree. This is actually where that algorithm comes from. This one was first, I believe. Questions? I'm not worrying too much about build time, how long it takes to build these data structures, but they can all be built in linear time. And this is one of the cooler algorithms, and it's a nice tie into last lecture.

So that's a reduction from RMQ to LCA, so now all of our problems are about trees, in some sense. I mean, there's a reason I mentioned RMQ. Not just that it's a handy problem to have solved, but we're actually going to use RMQ to solve LCA. So we're going to go back and forth between the two a lot. Actually, we'll spend most of our time in the RMQ land.

So let me tell you about the reverse direction, if you want to reduce LCA to RMQ. That also works. And you can kind of see it in this picture. If I gave you this tree, how would you reconstruct this array? Pop quiz. How do I go from here to here? In-order traversal, yep. Just doing an in-order traversal, write those guys-- I mean, yeah. Pretty easy. Now, not so easy because in the LCA problem, I don't have numbers in the nodes. So if I do an in-order walk and I write stuff, it's like, what should I write for each of the nodes. Any suggestions?

AUDIENCE: [INAUDIBLE]

PROFESSOR: The height? Not quite the height. The depth. That will work. So let's do it, just so it's clear. Look at the same tree Is that the same tree? Yep. So I write the depths. 0, 1, 1, 2, 2, 2, 3, 3.

It's either height or depth, and you try them both. This is depth. So I do an in-order walk I get 2, 1, 0-- can you read my writing-- 3, 2, 3, 1, 2. It's funny doing an in-order traversal on something that's not a binary search tree, but there it is. That's the order in which you visit the nodes.

And you stare at it long enough, this sequence will behave exactly the same as this sequence. Of course, not in terms of the actual values returned. But if you do the argument version of RMQ, you just ask for what's the index that gives me the min. If you can solve RMQ on this structure, then that RMQ will give exactly the same answers as this structure. Just kind of nifty. Because here I had numbers, they could be all over the place. Here I have very clean numbers. They will go between 0 and the height of the tree.

So in general at most, 0, 2, n minus 1. So fun consequence of this is you get a tool for universe reduction in RMQ. The tree problems don't have this issue, because they don't involve numbers. They involve trees, and that's why this reduction does this. But you can start from an arbitrary ordered universe and have an RMQ problem on that, and you can convert it to LCA. And then you can convert it to a nice clean universe RMQ, just by doing the Cartesian tree and then doing the in-order traversal of the depths.

This is kind of nifty because if you look at these algorithms, they only assume a comparison model. So these don't have to be numbers. They just have to be something from a totally ordered universe that you can compare in constant time. You do this reduction, and now we can assume they're integers, nice small integers, and that will let us solve things in constant time using the Word RAM. So you don't need to assume that about the original values.

Cool. So, time to actually solve something. We've done reductions. We now know RMQ and LCA are equivalent. Let's solve them both. Kind of like the last of the sorting we saw, there's going to be a lot of steps. They're not sequential steps. These are like different versions of a data structure for solving RMQ, and they're going to be getting progressively better and better.

So LCA which applies RMQ. This is originally solved by Harel and Tarjan in 1984, but is rather complicated. And then what I'm going to talk about is a version from 2000 by Bender and Farach-Colton, same authors from the cache-oblivious B-trees. That's a much simpler presentation. So first step is I want to do this reduction again from LCA to RMQ, but slightly differently.

And we're going to get a more restricted problem called plus or minus 1 RMQ. What is plus or minus 1 RMQ? Just means that you get an array where all the adjacent values differ by plus or minus 1. And if you look at the numbers here, a lot of them differ by plus or minus 1. These all do. But then there are some big gaps-- like this has a gap of 3, this has a gap of 2. This is plus or minus 1.

That's almost right, and if you just stare at this idea of tree walk enough, you'll realize a little trick to make the array a little bit bigger, but give you plus or minus ones. If you've done a lot of tree traversal, this will come quite naturally. This is a depth first search. This is how the depth first search order of visiting a tree in order. This is usually called an Eulerian tour. The concept we'll come back to in a few lectures. But Euler tour just means you visit every edge twice, in this case. If you look at the node visits, I'm visiting this node here, here, and here, three times. But it's amortized constant, because every edge is just visited twice.

What I'd like to do is follow an Euler tour and then write down all the nodes that I visit, but with repetition. So in that picture I will get 0, 1, 2, 1. I go 0, 1, 2, back to 1, back to 0, then over to the 1 on the right, then to the 2, then to the 3, then back up to the 2, then down to the other 3, then back up to the 2, back up to the 1, back down to the last node on the right, and back up and back up.

OK. This is what we call Euler tour. So multiple visits-- for example, here's all the places that the root is visited. Here's all the places that this node is visited, then this node is visited 3 times. It's going to be visited once per incident edge. I think you get the pattern. I'm just going to store this. And what else am I going to do? Let's see. Each node in the tree stores, let's say, the first visit in the array. Pretty sure this is enough. You could maybe store the last visit as well. We can only store a constant number of things.

And I guess each array item stores a pointer to the corresponding node in the tree. OK. So each instance of the 0 stores a pointer to the root, and so on. It's kind of what these horizontal bars are indicating, but those aren't actually stored. OK. So I claim still RMQ and here is the same as LCA over there. It's maybe a little more subtle, but now if I want to compute the LCA of two nodes, I look at their first occurrences.

So let's do-- I don't know-- 2 and 3. Here, this 2 and this 3. I didn't label them, but I happen to know where they are. 2 is here, and it's the first 3. Now here, they happen to only occur once in the tour, so it's a little clearer. If I compute the RMQ, I get this 0, this 0, as opposed to the other 0s, but this 0 points to the root, so I get the LCA.

Let's do ones that do not have unique occurrences. So like, this guy and this guy, the first 1 and the first 2 It'd be this 1 and this 1. In fact, I think any of the 2s would work. Doesn't really matter. Just have to pick one of them. So I picked the leftmost one for consistency. Then I take the RMQ, again I get 0. You can test that for all of them. I think the slightly more subtle case is when one node is an ancestor of another.

So let's do that, 1 here and 3 there. I think here you do need to be leftmost or rightmost, consistently. So I take the 1 and I take the second 3. OK. I take the RMQ of that, I get 1 which is the higher of the two. OK. So it seems to work. Actually, I think it would work no matter which guy you pick. I just picked the first one.

OK, no big deal. You're not going to see why this is useful for a little bit until step 4 or something, but we've slightly simplified our problem to this plus or minus 1 RMQ. Otherwise identical to this in-order traversal. So not a big deal, but we'll need it later. OK. That was a reduction. Next, we're finally going to actually solve something. I'm going to do constant time, n log n space, RMQ. This data structure will not require plus or minus 1 RMQ. It works for any RMQ. It's actually a very simple idea, and it's almost what we need. But we're going to have to get rid of this log factor. That will be step 3.

OK, so here's the idea. You've got an array. And now someone gives you an arbitrary interval from here to here. Ideally, I just store the mins for every possible interval, but there's n squared intervals. So instead, what I'm going to do is store the answer not for all the intervals, but for all intervals of length of power of 2. It's a trick you've probably seen before.

This is the easy thing to do. And then the interesting thing is how you make it actually get down to linear space. Length, power of 2. OK. There are only log n possible powers of 2. There's still n different start points for those intervals, so total number of intervals is n log n. So this is n log n space, because I'm storing an index for each of them. OK. And then if I have an arbitrary query, the point is-- let's call it length k-- then I can cover it by two intervals of length a power of 2. They will be the same length. They will be length 2 to the floor of log k, the next smaller power of 2 below k.

Maybe k is a power of 2, in which case, it's just one interval or two equal intervals. But in general, you just take the next smaller power of 2. That will cover more than half of the thing, of the interval. And so you have one that's left aligned, one that's right aligned. Together, those will cover everything. And because the min operation has this nifty feature that you can take the min of all these, min of all these, take the min of the 2. You will get the min overall. It doesn't hurt to have duplicate entries. That's kind of an important property of min. It holds for other properties too, like max, but not everything.

Then boom, we've solved RMQ. I think it's clear. You do two queries, take the min of the two-- actually, you have to restore the arg mins. So it's a little more work, but constant time. Cool. That was easy. Leave LCA up there. OK. So we're almost there, right. Just a log factor off. So what technique do we have for shaving log factors?

Indirection, yeah, our good friend indirection. Indirection comes to our rescue yet again, but we won't be done. The idea is, well, want to remove a log factor. Before we removed log factors from time, but there's no real time here, right. Everything's constant time. But we can use indirection to shave a log factor in space, too. Let's just divide. So this is again for RMQ. So I have an array, I'm going to divide the array into groups of size, I believe 1/2 log n would be the right magic number. It's going to be theta log n, but I need a specific constant for step 4.

So what does that mean? I have the first 1/2 log n entries in the array. Then I have the next 1/2 log entries, and then I have the last 1/2 log n entries. OK, that's easy enough. But now I'd like to tie all these structures together. A natural way to do that is with a big structure on top of size, n over log n, I guess with a factor 2 out here. n over 1/2 log n. How do I do that? Well, this is an RMQ problem, so the natural thing to do is just take the min of everything here. So the red here is going to denote taking the min, and take that-- the one item that results by taking the min in that group, and promoting it to the next level.

This is a static thing we do ahead of time. Now if I'm given a query, like say, this interval, what I need to do is first compute the min in this range within a bottom structure. Maybe also compute the min within this range, the last bottom structure, and then these guys are all taken in entirety. So I can just take the corresponding interval up here and that will give me simultaneously the mins of everything below. So now a query is going to be the min of two bottoms and one top.

In other words, I do one top RMQ query for everything between, strictly between the two ends. Then I do a bottom query for the one end, a bottom query for the other end. Take the min of all those values and really, it's the arg min, but. Clear? So it would be constant time if I can do bottom in constant time, if I can do top in constant time. But the big win is that this top structure only has to store n over log n items. So I can afford an n log n space data structure, because the logs cancel.

So I'm going to use structure 2 for the top. That will give me constant time up here, linear space. So all that's left is to solve the bottoms individually. Again, similar kind of structure to [INAUDIBLE]. We have a summary structure and we have the details down below. But the parameters are way out of whack. It's no longer root n, root n. Now these guys are super tiny because we only needed this to be a little bit smaller than n, and then this would work out to linear space.

OK. So step 4 is going to be how do we solve the bottom structures. So step 4. This is where we're going to use technique of lookup tables for bottom groups. This is going to be slightly weird to phrase, because on the one hand, I want to be thinking about an individual group, but my solution is actually going to solve all groups simultaneously, and it's kind of important. But for now, let's just think of one group.

So it has size n prime and n prime is 1/2 log n. I need to remember how it relates to the original value of n so I know how to pay for things. The idea is there's really not many different problems of size 1/2 log n. And here's where we're going to use the fact that we're in plus or minus 1 land. We have this giant string of integers. Well, now we're looking at log n of them to say OK, this here, this is a sequence 0, 1, 2, 3. Over here a 0, 1, 2, 1. There's all these different things. Then there's other things like 2, 3, 2, 3.

So there's a couple annoying things. One is it matters what value you start at, a b, and then it matters what the sequence of plus and minus 1s are after that. OK. I claim it doesn't really matter what value you start at, because RMQ, this query, is invariant under adding some value x to all entries, all values, in the array. Or if I add 100 to every value, then the minimums stay the same in position. So again, here I'm thinking of RMQ as an arg min. So it's giving just the index of where it lives.

So in particular, I'm going to add minus the first value of the array to all values. I should probably call this-- well, yeah. Here I'm just thinking about a single group for now. So in a single group, saying well, it starts at some value. I'm just going to decrease all these things by whatever that value is. Now some of them might become negative, but at least now we start with a 0. So what we start with is irrelevant.

What remains, the remaining numbers here are completely defined by the gaps between or the difs between consecutive items, and the difs are all plus or minus 1. So now the number of possible arrays in a group, so in a single group, is equal to the number of plus or minus 1 strings of length n prime, which is 1/2 log n. And the number of plus or minus 1 strings of length n prime is 2 to the n prime.

So we get 2 to the 1/2 log n, also known as square root of n. Square root of n is small. We're aiming for linear space. This means that for every-- not only for every group, there is n over log n groups-- but actually many of the groups have to be the same. There's n over log n groups, but there's only root n different types of groups. So on average, like root n over log n occurrences of each.

So we can kind of compress things down and say hey, I would like to just like store a lookup table for each one of these, but that would be quadratic space. But there's really only square root of n different types. So if I use a layer of indirection, I guess-- different sort of indirection-- if I just have, for each of these groups, I just store a pointer to the type of group, which is what the plus or minus 1 string is, and then for that type, I store a lookup table of all possibilities. That will be efficient.

Let me show that to you. This is a very handy idea. In general, if you have a lot of things of size roughly log n, lookup tables are a good idea. And this naturally arises when you're using indirection, because usually you just need to shave a log or two. So here we have these different types. So what we're going to do is store a lookup table that says for each group type, I'll just say a lookup table of all answers, do that for each group type. Group type, meaning the plus or minus 1 string. It's really what is in that group after you do this shifting.

OK. Now there's square root of n group types. What does it take to store the answers? Well, there is, I guess, 1/2 log n squared different queries, because n prime is 1/2 log n, and a query is defined by the two endpoints. So there's at most this many queries. Each query, to store the answer, is going to take order log log n bits-- this is if you're fancy-- because the answer is an index into that array of size 1/2 log n, so you need log log n bits to write down that.

So the total size of this lookup table is the product of these things. We have to write root n look up tables. Each stores log squared n different values, and the values require log log n bits. So total number of bits is this thing, and this thing is little o of n. So smaller than linear, so it's irrelevant. Can store for free.

Now if we have a bottom group, the one thing we need to do is store a pointer from that bottom group to the corresponding section of the lookup table for that group type. So each group stores a pointer into lookup table.

I'm of two minds whether I think of this as a single lookup table that's parameterized first by group type, and then by the query. So it's like a two-dimensional table or three-dimensional, depending how you count. Or you can think of there being several lookup tables, one for each group type, and then you're pointing to a single lookup table. However, you want to think about it, same thing. Same difference, as they say.

This gives us linear space. These pointers take linear space. The top structure takes linear space linear number of words, and constant query time, because lookup tables are very fast. Just look into them. They give you the answer. So you can do a lookup table here, lookup table here. And then over here, you do the covering by 2, powers of 2 intervals. Again, we have a lookup table for those intervals, so it's like we're looking into four tables, take the min of them all, done.

That is RMQ, and also LCA. Actually it was really LCA that we solved, because we solved plus or minus 1 RMQ, which solved LCA, but by the Cartesian tree reduction, that also solves RMQ. Now we solved 2 out of 3 of our problems. Any questions?

Level ancestors are going to be harder, little bit harder. Similar number of steps. I'd say they're a little more clever. This I feel is pretty easy. Very simple style of indirection, very simple style of enumeration here. It's going to be a little more sophisticated and a little bit more representative of the general case for level ancestors. Definitely fancier. Level ancestors is a similar story we solved a while ago, but it was kind of a complicated solution. And then Bender and Farach-Colton found it and said hey, we can simplify this. And I'm going to give you the simplified version.

So this is level ancestors. Says originally solved by Berkman and Vishkin in 1994, OK, not so long ago. And then the new version is from 2004. Ready? Level ancestors. What was the problem again? Here it is. I gave you a rooted tree, give you a node, and a level that I want to go up, and then I level up by k, so I go to the kth ancestor, or parent to the k.

This may seem superficially like LCA, but it's very different, because as you can see, RMQ was very specific to LCA. It's not going to let you solve level ancestors in any sense. I don't think. Maybe you could try to do the Cartesian tree reduction, but solution we'll see is completely different, although similar in spirit. So step 1. This one's going to be a little bit less obvious that we will succeed. Here we started with n log n space which is shaving a log, no big deal. Here, I'm going to give you a couple of strategies that aren't even constant time, they're log time or worse. And yet you combine them and you get constant time. It's crazy.

Again, each of the pieces is going to be pretty intuitive, not super surprising, but it's one of these things where you take all these ingredients that are all kind of obvious, you stare at them for a while like, oh, I put them together and it works. It's like magic. All right, so first goal is going to be n log n space, log n query. So here's a way to do it with a technique called jump pointers.

In this case, nodes are going to have log n different pointers, and they're going to point to the 2 to the ith ancestor for all i. I guess maximum possible i would be log n. You can never go up more than n. So I mean, ideally you'd have a pointer to all your ancestors in array, boom. In the quadratic space, you solve your problem in constant time. But it's a little more interesting. Now every node only has pointers to log n different places so it's looking like this. This is the ancestor path. So n log n space, and I claim with this, you can roughly do a binary search, if you wanted to.

Now we're not actually going to use this query algorithm for anything, but I'll write it down just so it feels like we've accomplished something, mainly log n query time. So what do I do? I set x to be the 2 to the floor log kth ancestor of x. OK, remember we're given a node x and a value k that we want to rise by. So I take the power of 2 just below k-- that's 2 the floor log k. I go up that much, and that's my new x, and then I set k to be k minus that value. That's how much I have left to go.

OK. This thing will be less than k over 2. Because the next previous power of 2 is at least, is bigger than half of the thing. So we got more than halfway there, and so after log n iterations, we'll actually get there. That's pretty easy. That's jump pointers to two logs that we need to get rid of, and yes, we will use indirection, but not yet. First, we need some more ingredients.

This next ingredient is kind of funny, because it will seem useless. But in fact, it is useful as a step towards ingredient 3. So the next trick is called long path decomposition. In general, this class covers a lot of different treaty compositions. We did preferred path decomposition for tango trees. We're going to do long path now. We'll do another one called heavy path later. There's a lot of them out there. This one won't seem very useful at first, because while it will achieve linear space, it will achieve the amazing square root of n query, which I guess is new. I mean, we don't know how to do that yet with linear space. Not so obvious how to get root n.

But anyway, don't worry about the query time. It's more the concept of long path that's interesting. It's a step in the right direction. So here's what here's how we're going to decompose a tree. First thing we do is find the longest route to leaf path in the tree, because if you look at a tree, it has some wavy bottom. Take the deepest node. Take the path the unique path from the root to that node. OK.

When I do that, I could imagine deleting those nodes. I mean, there's that path, and then there's everything else, which means there's all these triangles hanging off of that path, some on the left, some on the right. Actually, I haven't talked about this, but both LCA and level ancestors work not just for binary trees. They work for arbitrary trees.

And somewhere along here-- yeah, here. This reduction of using the Euler tour works for non-binary trees, too. That's actually another reason why this reduction is better than in-order traversal by itself. In-order traversal works only for binary trees. This thing works for any tree. In that case, in an arbitrary tree, you visit the node many, many times potentially. OK, but it will still be linear space and everything will still work.

Here also, I want to handle non-binary trees. So I'm going to draw things hanging off, but in fact, there might be several things hanging off here, each their own little tree. OK, but the point is-- where's my red. Here. There was this one path in the beginning, the longest path, and then there's stuff hanging off of it. So just recurse on all the things hanging off of it. Recursively decompose those sub-trees.

OK. Not clear what this is going to give you. In fact, it's not going to be so awesome, but it will be a starting point. Now you can answer a query with this, as follows. Query-- oh, sorry. I should say how we're actually storing these paths. Here's the cool idea with this path thing. I have this path. I'd like to be able to jump around at least-- suppose your tree was a path. Suppose your tree were a path. Then what would you want to do? Store the nodes in an array ordered by depth, because then if you're a position i and you need to go to position i minus k, boom. That's just a look up into your array.

So I'm going to store each path as an array, as an array of nodes or node pointers, I guess, ordered by depth. So if it happens, so if my query value x is somewhere on this path, and if this path encompasses where I need to go-- so if I need to go k up and I end up here-- then that's instantaneous. The trouble would be is if I have a query, let's say, over here. And so there's going to be a path that guy lives on, but maybe the kth ancestor is not on that path. It could be on a higher up path. It could be on the red path, and I can't jump there instantaneously.

Nonetheless, there is a decent query algorithm here. All right. So Here's what we're going to do. If k is less than or equal to the index i of node x on its path. So every node belongs to exactly one path. This is a path decomposition. It's a partition of the tree into paths. Not all the edges are represented, but all the nodes are there. All the nodes belong to some path, and we're going to store, for every node, store what its index is and where it lives in its array.

So look at that index in the array. If k is less than or equal to that index, then we can solve our problem instantly by looking at the path array at position i minus k. That's what I said before. If our kth ancestor is within the path, then that's where it will be, and that's going to work as long as that is non-negative. If I get to negative, that means it's another path. So that's the good case.

The other case is we're just going to do some recursion, essentially. So we're going to go as high as we can with this path. We're going to look at path array at position 0. Go to the parent of that. Let's suppose every node has a parent pointer. That's easy, regular tree, and then decrease k by 1 plus i. So the array let us jump up i steps-- that's this part-- and then the parent stepped us up one more step. That's just to get to the next path above us.

OK, so how much did this decrease k by? I'd like to say a factor of 2 and get log n, but in fact, no, it's not very good. It doesn't decrease k by very much. It does decrease k, guaranteed by at least 1, so it's definitely linear time. And there's a bad tree, which is this. It's like a grid. Whoa. Sorry. OK, here's a tree. It's a binary tree. And if you set it up right, this is the longest path.

And then when you decompose, this is the longest path, and this is the longest path, this is the longest path. If you query here, you'll walk up to here, and then walk up to here, and walk up to here, and walk up to here. So this is a square root of n lower bound for this algorithm. So not a good algorithm yet, but the makings of a good algorithm. Makings of step 3, which is called ladder decomposition.

Ladder decomposition is something I haven't really seen anywhere else. I think it comes from the parallel algorithms world in general. And now we're going to achieve linear space log n query. Now this is an improvement. So we have, at the moment, n log n space, log n query or n space root n query. We're basically taking the min of the two. And so we're getting linear space log n query. Still not perfect. We want constant query. That's when we'll use indirection, I think. Yeah, basically, a new type of indirection, but OK.

So linear space log n query. Well, the idea is just to fix long paths, and it's a crazy idea, OK. Let me tell you the idea and then it's like, why would that be useful. But it's obvious that it doesn't hurt you, OK. When we have these paths, sometimes they're long. Sometimes they're not long enough. Just take each of these paths and extend them upwards by a factor of 2. That's the idea. So take number 2, extend each path upward 2 x. So that gives us call a ladder.

OK, what happens? Well, paths are going to overlap. Fine. Ladders overlap. The original paths don't overlap. Ladders overlap. I don't really care if they overlap. How much space is there? It's still linear space, because I'm just doubling everything. So I've most doubled space relative to long path decomposition. I didn't mention it explicitly, but long path decomposition is linear space. We're just partitioning up the tree into little pieces. Doesn't take much.

We have to store those arrays, but every node appears in exactly one cell here. Now every node will appear in, on average, two cells in some weird way. Like what happens over here? I have no idea. So this guy's length 1. It's going to grow to length 2. This one's length 2, so now it'll grow to length 4. This one's length 3-- and it depends on how you count. I'm counting nodes here. That's going to go here, all the way the top. Interesting. All the others will go to the top.

So if I'm here, I walk here. Then I can jump all the way to the top. Then I can jump all the way to the root. Not totally obvious, but it actually will be log n steps. Let's prove that. This is again something we don't really need to know for the final solution, but kind of nice, kind of comforting to know that we've gotten down a log n query. So it's at most double the space. This is still linear. Now-- oh, there's one catch.

Over in this world, we said each-- I didn't say it. I mentioned it out loud. Every node stores what array it lives in. Now a node lives in multiple arrays, OK. So which one do I store a pointer to? Well, there's one obvious one to store a pointer to. Whatever node you take lives in one path. In that long path decomposition, it still lives in one path. Store a pointer into that ladder.

So node stores a pointer you could say to the ladder that contains it in the lower half. That corresponds to the one where it was an actual path. And only one ladder will contain a node in its lower half. The upper half was the extension. I guess it's like those folding ladders you extend.

OK. Cool. So that's what we're going to do and also store its index in the array. Now we can do exactly this query algorithm again, except now instead of path, it says ladder. So you look at the index of the node in its ladder. If that index is larger than k, then boom, that ladder array will tell you exactly where to go. Otherwise you go to the top of the ladder and then you take the parent pointer, and you decrease by this.

But now I claim that decrease will be substantial. Why? If I have a node of height h-- remember, height of a node is the length of the longest path from there downward-- it will be on a ladder of height at least 2h. Why? Because if you look at a node of height h-- like say, I don't know, this node-- the longest path from there is substantial. I mean, if it's height h, then the longest path from there is length at least h. So every node of height h will be on a path of length at least h, and from there down.

And so you look at the ladder. Well, that's going to be double that. So the ladder will be height at least 2h, which means if your query starts at height h, after you do one step of this ladder search, you will get to height at least 2h, and then 4h, and then 8h. You're increasing your height by a power of 2, by a factor of 2 every time. So in log n steps, you will get to wherever you need to go. OK You don't have to worry about overshooting, because that's the case when the array tells you exactly where to go.

OK. Time for the climax. It won't be the end, but it's the climax in the middle of the story. So we have on the one hand, jump pointers. Remember those? Jump pointers made small steps initially and got-- actually, no. This is what it looks like for the data structure. But if you look at the algorithm, actually it makes a big step in the beginning. Right? It gets more than halfway there. Then it makes smaller and smaller steps, exponentially decreasing steps. Finally, it arrives at the intended node.

Ladder decomposition is doing the reverse. If you start at low height, you're going to make very small steps in the beginning. As your height gets bigger, you're going to be making bigger and bigger steps. And then when you jump over your node, you found it instantly. So it's kind of the opposite of jump pointers. So what we're going to do is take jump pointers and add them to ladder decomposition. Huh.

This is, I guess, version 4. Combine jump pointers from one and ladders from three. Forget about two. Two is just a warm up for three. Long paths, defined ladders. So we've got one way to do log n query. We've got another way to do log n query. I combine them, and I get constant query. Because log n plus log n equals 1. I don't know.

OK, here's the idea. On the one hand, jump pointers make a big step and then smaller steps, right. Yeah, like that. And on the other hand, ladders make small steps. It's hard to draw. What I'd like to do is take this step and this step. That would be good, because only two of them. So query is going to do one jump, plus 1 ladder, in that order.

See, the thing about ladders is it's really slow in the beginning, because your height is small. I really want to get large height. Jump pointers give you large height. The very first step, you get half the height you need. That's it. So when we do a jump, we do one step of the jump algorithm. What do we do? We reach height at least k over 2 above x. All right, we get halfway there. So our height-- it's a little-- let's say x has height h. OK, so then we get to height-- this is saying we get to height h plus k over 2.

OK, that's good. This is a big height. Halfway there, I mean, halfway of the remainder after h. Now ladders double your height in every step. So ladder step-- so this is the jump step. If you do one ladder step, you will reach height double that. So it's at least 2 h plus k, which is bigger than what we need. We need h plus k. That's where we're trying to go. And so we're done. Isn't that cool?

So the annoying part is there's this extra part here. This is the h part and we start at some level. We don't know where. This is x. The worst case is maybe when it's very small, but whatever it is, we do this step and this is our target up here. This is height h plus k. In one step, we get more than halfway there with the jump pointer. And then the ladder will carry us the rest of the way.

Because this is the ladder. We basically go horizontally to fall on this ladder, and it will cover us beyond where we need to go, beyond our wildest imaginations. So this is k over 2. Because not only will it double this, which is what we need to double, it will also double whatever is down here, this h part. So it gets us way beyond where we need to go. I mean, could be h 0. Then it gets us to exactly where we need to go.

But then the ladder tells us where to go. So two steps constant time. Now one annoying thing is we're not done with space. So this is the anticlimax part. It's still going to be pretty interesting. We've got to shave off a log factor in space, but hey, we're experienced. We already did that once today. Question?

Yeah. Why is it OK to go past your target?

The question was why is it OK to go past our target? Jump pointers aren't allowed, because they only know how to go up. They can't overshoot. That's why they went less than halfway, or more than halfway, but less than the full way. Ladder decomposition can go beyond, because as soon as-- the point is, as soon as-- here's you, x, and here's your kth ancestor. This is the answer.

As soon as you're in a common ladder, then the array tells you where to go. So even though the top of the ladder overshot, there will be a ladder connecting you to that top of the ladder. So as long as it's somewhere in between, it's free. Yeah, so that's why it's OK this goes potentially too high. So it's good for ladders, not good for jumps, but that's exactly where we have it Other questions? Yeah.

AUDIENCE: [INAUDIBLE] jump pointers, wouldn't you be high up enough in the tree so that just the long path would work?

PROFESSOR: Oh, interesting question. So would it be enough to do jump pointers plus long path? My guess is no. Jump pointers get you up to-- so think of the case where h is 0. Initially you're at height 0. I think that's going to be a problem. You jump up to height k over 2 with a jump pointer.

Now long path decomposition, you know that the path will have a length at least k over 2, but you need to get up to k. And so you may get stuck in this kind of situation where maybe you're trying to get to the root and you jumped to here, but then you have to walk. So I think the long path's not enough. You need that factor of 2, which the ladders give you.

You can see where ladders come from now, right? I mean we got up to height k over 2. Now we just need to double it. Hey, we can afford to double every path, but I think we need to. Are there questions?

OK. So last thing to do is to shave off this log factor of space. Now, we're going to do that with indirection, of course, constant time and log n space. But it's not our usual type of indirection.

Use this board. Indirections. So last time we did indirection, it was with an array. And actually pretty much every indirection we've done, it's been with an array-like thing. We could decompose into groups of size log n, the top thing was n over log n. So it was kind of clean.

This structure is not so clean, because it's a tree. How do you decompose a tree into little things at the bottom of size log n and a top thing of size n over log n? Suppose, for example, your tree is a path. Bad news. If my tree were a path, well, I could trim off bottom thing of size log n. But now the rest is of size n minus log n, not n divided by log n. That's bad. I need to shave a factor of log n, not an additive log n.

Can you tell me a good thing about a path? I mean, obviously, when we can put in an array. But can you quantify the goodness, or the pathlikedness of a tree? I erase this board. Kind of a vague question. Good thing about a path is that it doesn't have very many leaves. That's one way to quantify pathedness. Small number of leaves, I claim life's not so bad. I actually need to do that before we get to indirection.

Step 5 is let's tune jump pointers a bit. I want to make them-- so they're the problem, right? That's where we get n log n space. They're the only source of our n log n space. So what I'd like to do is in this situation where the number of leaves is small-- we'll see what small is in a moment-- I would like jump pointers to be linear size.

OK, here's the idea. First idea is let's just store jump pointers from leaves. OK. So that would imply l log n space, I guess, plus linear overall. Instead of n log n, now we just pay for the leaves, except we kind of messed up our query. First thing query did was at the node, follow the jump pointer. But it's not so bad.

Here we are at x. There's some leaves down here, and we want to jump up from here, from x. How do I jump from x? Well, if I could somehow go from x to really, any leaf, the ancestors of x that I care about are also ancestors of any leaf descendant of x. So all I need to do is store for each node any leaf descendant, single pointer-- this'll be linear-- from every node.

OK so I start at x. I jump down to an arbitrary leaf, say this one. And now I have to do a query. Jump down, and let's say I jumped down by d. Then my k becomes k plus d, right. If I went down by d, and I want to go up by k from my original point, now I have to go up by k plus d.

But hey, we know how to go up from any node that has jump pointers. So now we have a new node, a leaf. So it has a jump pointer, has jump pointers, upward. So we follow that one jump pointer to get us halfway there from our new starting point.

We follow one ladder thing, and we can get to the level ancestor k plus d from the leaf, and that's the level ancestor k from x. OK, this is like a reduction to the leaf situation. We really don't have to support queries from arbitrary nodes. Just go down to a leaf and then solve the problem from the leaf. OK.

OK, so now, if the number leaves is small, my space will get small. How small does l have to be? n divided by log n. Interesting. If I could get the top structure to not have n over log n nodes, that's not possible. I can, at best, get to n minus log n nodes. But if I could get it down to n over log n leaves, that would be enough to make this linear space, and indeed, I can.

This is a technique called tree trimming, or I call it that. I don't know if anyone else does. But I think I've called it that in enough papers that we're allowed to call it that. Originally invented by [? Al ?] [? Strip ?] and others for a particular data structure.

There's many versions of it. We will see other versions in future lectures, but here's the version you need for this problem. OK, here's the plan. I have a tree and I want to identify all the maximally deep nodes that have at least log n nodes below them. This will seem weird, because we really care about leaves, and so on. So there's stuff hanging off here, whatever. I guess I'm thinking of that as one big tree. No, actually I'm not. I do need to separate these out.

But one of these nodes could have arbitrarily many children. We have no idea. It's a arbitrary tree. OK, and what I know is that each of these triangles has size less than 1/4 log n. Because otherwise, this node was not maximally deep. So if this had size greater or equal than 1/4 log n, then that would have been the node where I cut, not this one. So I'm circling the nodes that I cut below, so meaning I cut these edges.

OK, so these things have size less than 1/4 log n, but these nodes have at least 1/4 log n nodes below them. So how many of these circle nodes are there? Well, at most, 4 n over log n such nodes, right, because I can charge this node to at least 1/4 log n nodes that disappear in the top structure. But these things become the leaves, right. If I cut all the edges going down from there, that makes it a leaf. And they're the only leaves. Are they the only leaves? Yeah.

If you look at a leaf, then it has size less than 1/4 log n. So you will cut above it somewhere. So every old leaf will be down here, and the only new leaves will be the cut nodes. OK. So we have order n over log n leaves. Yes, good.

So it's funny. We're cutting according to counting nodes, descendants, not leaves. Won't work if you cut with leaves-- cut with nodes. But then the thing that we care about is the number of leaves went down. That will be enough. Great. So up here, we can afford to use 5, the tuned jump pointer, combined with ladder structure. Because this only costs l log n. l is now n over log n, so the log n's cancel. So linear space to store the jump pointers from these circled nodes.

So if our query is anywhere up here, then we go to a descendant leaf in the top structure. And we can go wherever we need to go. If our query is in one of the little trees at the bottom, which are small, they're only 1/4 quarter log n, so we're going to use a lookup table. Either answer is inside the triangle, in which case, we really need to query that structure. Or it's up here. If it's up here, we just need to know, basically, if every node down here stores a pointer to the dot above it.

Then we can first go there and see, is that too high? If it's too high, then our answer is in here. If it's not too high, then we just do the corresponding query in structure 5.

OK, so the last remaining thing is to solve a query that stays entirely within a triangle, so a bottom structure, and that's where we use lookup tables. Again, things are going to be similar to last time except for now, to step 7. But it's a little bit messier because instead of arrays, we have trees. And here it's like we graduate from baby [INAUDIBLE] which is how many plus or minus 1 strings there are-- power of 2-- to how many trees are there.

Anyone know how many trees on n nodes there are? One word answer. No. Nice. That is a correct one word answer. Very good. Not the one I had in mind, but anyone else? Nope. You're thinking end to the end. That would be bad. We could not afford that, because log n to log n is super polynomial. Fortunately it's not that big. Hmm?

AUDIENCE:

PROFESSOR: It's roughly 4 to the n. The correct answer-- I mean the exact answer-- is called the Catalan number, which didn't tell you much. I didn't write it down, but I'm pretty sure it is 2 n prime choose n prime 1 over n prime plus 1 ish? Don't quote me on that. It's roughly that. Might be exactly that. Someone with internet can check. But it is at most 4 to the n prime. The computer science answer is 4 to the n. Indeed.

It's just some asymptotics here. Why is it 4 to the n? 4 to the n you could also write as 2 to the

2 n prime, which is-- first, let's check this is good, and then I'll explain why this is true in a computer science way. So we got 1/4 log n up here. So the one 2 cancels with one 2 up here. So we have 2 to the 1/2 log n. This is our good friend root n. Root n is just something that's n to the something, but is n to the something less than 1. So we can afford some log factors.

Why are there only 2 to the 2 n prime trees? One way to see that is you can encode a tree using 2n bits. If I have an n node tree, I can encode it with 2n bits. How? Do an Euler tour. And all you really need to know from an Euler tour to reconstruct the tree is at each step, did I go down or did I go up?

Those are the only things you can do. If you went down, it's to a new child. If you went up, it's to an old node. So if I told you a sequence of bits for every step in the Euler tour, did I go down or did I go up, you can reconstruct the tree. Now how many bits do I have to do? Well, twice the number of edges in the tree, because the length of an Euler tour is twice the number of edges in the tree. So 2 n bits are enough to encode any tree. That's the computer science information theoretic way to prove it.

You could also do it from this formula, but then you'd have to know why the formula's correct, and that's messier. Cool. So we're almost done. We have root n possible different structures down here. We've got n over log n of them or-- maybe. It's a little harder to know exactly how many of them there are, but I don't care.

There's only root n different types, and so I only need to store a lookup table for each type. The number of queries is order log squared n again, because our structures are of size order log n, and the answer to a query is again, order log log n bits, because there's only log n different nodes to point to.

And so the total space is order root n log n squared, log log n for the lookup table. And then each of these triangles stores a pointer, or I guess, every node in here stores a pointer to what tree we're in, or what type of tree we have, and also what node in that tree we are in.

So every guy in here-- because that's not part of the query-- has to store, not only a little bit more specific pointer into this table. It actually tells you what the query part is, or the first part of the query, the node x. Then the table also is parameterized by k, so one of these logs is which node you're querying.

The other log is now the value k, but again, you never go up higher than log n. If you went up higher than log n, then you'd be in the 5 structure, so if you just do a query up there, you don't need a query in the bottom. OK. So there's only that many queries, and so space for this lookup table is little o of n again. And so we're dominated by space for these pointers and for the space up here, which is linear. So linear space, constant query. Boom. Any questions?

I have an open question, maybe. I think it's open. So what if you want to do dynamic, 30 seconds of dynamic? For LCA, it's known how to do dynamic LCA constant operations. The operations are add a leaf-- we can add another leaf-- given an edge. Subdivide that edge into that, and also the reverse.

So I can erase a guy, put the edge back, delete a leaf, those sorts of things. Those operations can all be done in constant time for LCA. What about level ancestor? I have no idea. Maye we'll work on it today. That's it.