PROFESSOR: All right, let's get started. Today we have another data structures topic which is, Data Structure Augmentation. The idea here is we're going to take some existing data structure and augment it to do extra cool things.

Take some other data structure there we've covered. Typically, that'll be a balanced search tree, like an AVL tree or a 2-3 tree. And then we'll modify it to store extra information which will enable additional kinds of searches, typically, and sometimes to do updates better.

And in 006, you've seen an example of this where you took AVL trees and augmented AVL trees so that every node knew the number of nodes in that rooted subtree. Today we're going to see that example but also a bunch of other examples, different types of augmentation you could do. And we'll start out with a very simple one, which I call easy tree augmentation, which will include subtree size as a special case.

So with easy tree augmentation, the idea is you have a tree, like an AVL tree, or 2-3 tree, or something like that. And you'd like to store, for every node x, some function of the subtree, rooted at x. Such as the number of nodes in there, or the sum of the weights of the nodes, or the sum of the squares of the weights, or the min, or the max, or the median maybe, I'm not sure. Some function f of x which is a function of that. Maybe not f of x, but we want to store some function of that subtree.

Say the goal is to store f of the subtree rooted at x at each node x in a field which I'll call x.f. So, normally nodes have a left child, right child, parent. But we're going to store an extra field x.f for some function that you define. This is not always possible, but here's a case where it is possible. That's going to be the easy case. Suppose x.f can be computed locally using lower information, lower nodes.

And we'll say, let's suppose it can be computed in constant time from information in the node x from x's children and from the f value that's stored in the children. I'll call that children.f. But really, I mean left child.f, right child.f, or if you have a 2-3 tree you have three children,

potentially. And the .f of each of them.

OK. So suppose you can compute x.f locally just using one level down in constant time. Then, as you might expect, you can update whenever a node ends up changing. So more formally. If some set of nodes change-- call this at s.

So I'm stating a very general theorem here. If there is some set of nodes, which we changed something about them. We change either their f field, we change some of the data that's in the node, or we do a rotation, loosen those around. Then we count the total number of ancestors of these nodes. So this subtree. Those are the nodes that need to be updated because we're assuming we can compute x.f just given the children data. So if this data is changing, we have to update it's parents value of f because it depends on this child value. We have to update all those parents, all the way up to the root. So however many nodes there are there, that's the total cost.

Now, luckily, in an AVL tree, or 2-3 tree, most balanced search structures, the updates you do are very localized. When we do splits in a 2-3 tree we only do it up a single path to the root. So the number of ancestors here is just going to be log n. Same thing with an AVL tree. If you look at the rotations you do, they are up a single leaf to root path. And so the number of ancestors that need to be updated is always order log n. Things change, and there's an order log n ancestors of them.

So this is a little more general than we need, but it's just to point out if we did log n rotation spread out somewhere in the tree, that would actually be bad because the total number of ancestors could be log squared. But because in the structures we've seen, we just work on a single path to the root, we get log n. So in a little more detail here, whenever we do a rotation in an AVL tree. Let's say A, B, C, x, y.

Remember rotations? Been a while since we've done rotations. So we haven't changed any of the nodes in A, B, C, but we have changed the nodes x and y. So we're going to have to trigger an update of y. First, we'd want to update y.f and then we're going to trigger the update to x.f. And as long as this one can be computed from its children, then we compute y.f, then we can compute x from its children.

All right. So a constant number of extra things we need to do whenever we do rotation. And because the rotations lie on a single path, total cost that-- once we stop doing the rotations, in AVL insert say, then we still have to keep updating up to the root. But there's only log n at

most log n nodes to do that.

OK. Same thing with 2-3 trees. We have a node split. So we have, I guess, three keys, four children. That's too many. So we split to two nodes and an extra node up here. Then we just trigger an update of this f value, an update of this f value, and an update of that f value. And because that just follows a single path everything's log n.

So this is a general theorem about augmentation. Any function that's well behaved in this sense, we can maintain in AVL trees and 2-3 trees. And I'll remind you and state, a little more generally, what you did in 006, which are called order statistic trees in the textbook.

So here we're going to-- let me first tell you what we're trying to achieve. This is the abstract data type, or the interface of the data structure. We want to do insert, delete, and say, successor searches. It's the usual thing we want out of a binary search tree. Predecessor too, sure. We want to do rank of a given key which is, tell me what is the index of that key in the overall sorted order of the items, of the keys?

We've talked about rank a few times already in this class. Depends whether you start at 0 or 1, but let's say we start at one. So if you say rank of the key that happens to be the minimum, you want to get one. If you say rank of the key that happens to be the median, you want to get n over 2 plus 1, and so on.

So it's a natural thing you might want to find out. And the converse operation is select, let's say of i, which is, give me the key of rank i.

We've talked about select as an offline operation. Given an array, find me the median. Or find me the n over seventh rank item. And we can do that in linear time given no data structure. Here, we want a data structure so that we can find the median, or the seventh item, or the n over seventh key, whatever in log n time. We want to do all of these in log n per operation.

OK. So in particular, rank of selective i should equal i. We're trying to find the item of that rank. So far, so good. And just to plug these two parts together. We have this data structure augmentation tool, we have this goal we want to achieve, we're going to achieve this goal by applying this technique where f is just the subtree size. It's the number of nodes in that subtree because that will let us compute rank.

So we're going to use easy tree augmentation with f of subtree equal to the number of nodes

in the subtree. So in order for this to apply, we need to check that given a node x we can compute x.f just using its children. This is easy. We just add everything up. So x.f would be equal to 1. That's for x. Plus the sum of c.f for every child c.

I'll write this as a python interpolation so it looks a little more like an algorithm. I'm trying to be generic here. If it's a binary search tree you just do x.left.f, plus x.right.f. But this will work also for 2-3 trees. Pick your favorite data structure. As long as there's a constant number of children then this will take constant time. So we satisfied this condition. So we can do easy tree augmentation. And now we know we have subtree sizes. So given any node. We know the number of descendants below that node. So that's cool.

It lets us compute rank in select. I'll just give you those algorithms, quickly. We can check that they're log n time.

Yeah. So the idea is pretty simple. You have some key-- let's think about binary trees now, because it's a little bit easier. We have some item x. It has a left subtree, right subtree. And now let's look up from x. Just keep calling x.parent. So sometimes the parent is to the right of us and sometimes the parent is to the left of us. I'm going to draw this in a, kind of, funny way.

But this funny way has a very special property, which is that the x-coordinate in this diagram is the key value. Or is the sorted order of the keys, right? Everything in the left subtree of x has a value less than x. If we say all the keys are different. Everything to the right of x has a value greater than x. If x was the left child of its parent, that means this thing is also greater than x. And if we follow a parent and this was the right child of that parent, that means this thing is less than x. So that's why I drew it all the way over to the left. This thing is also less than x because it was a, I'll call it a left parent. Here we have a right parent, so that means this is something greater than x. And over here we have a left parent, so this is something less than x. Let's say that's the root.

In general, there's going to be some left edges and some right edges as we go up. These arrows will go either left or right in a binary tree. So the rank of x is just 1 plus the number of nodes that are less than x. Number of keys that are less than x. So there's these guys, there's these guys, and there's whatever's hanging off-- OK. Here I've almost violated my x-coordinate rule. If I make these really narrow, that's right. All of these things, all of these nodes in the left subtrees of these less than x nodes will also be less than x. If you think about these other subtrees, they're going to be bigger than x. So we don't really care about them.

So we just want to count up all these nodes and all of these nodes. So the algorithm to do that is pretty simple. We're just going to start out with--

I'm going to switch from this f notation to size. That's a little more natural. In general, you might have many functions. Size is the usual notation for subtree size. So we start out by counting up how many items are here. And if we want to start at a rank of 1, if the min has rank 1, then I should also do plus 1 for x itself. If you wanted to start at zero you just omit that plus 1. And then, all I do is walk up from x to the root of the tree. And whenever we go left from, say x to x prime. So that means we have an x prime. It's right child is x. And so when we went from x to its parent we went to the left.

Then we say rank plus equals x prime.left.size plus 1 for x prime itself. And maybe x prime.left.size is zero. Maybe there's no nodes over there. But at the very least we have to count those nodes that are to the left of us. And if there's anything down here we add up all those things. So that lets us compute rank.

How long does it take? Well, we're just walking up one path from a leaf to a root-- or not necessarily a leaf, but from some node x to the root. And as long we're using a balance structure like AVL trees. I guess I want binary here, so let's say AVL trees. Then this will take log n time. So I'm spending constant work per step, and there's log n steps. Clear?

So that's good old rank. Easy to do once you have subtree size. Let's do select for fun.

This may seem like review, but I drew out this picture explicitly because we're going to do it a lot today. We'll have pictures like this a bunch of times. Really helps to think about where the nodes are, which ones are less than x, which ones are greater than x. Let's do select first. This you may not have seen in 006.

So we're going to do the reverse. We're going to start at the root and we're going to walk down. Sounds easy enough. But now walking down is kind of like doing a search but we don't have a key we're searching for, we have a rank we're searching for. So what is that rank? Rank is i. OK. So on the other hand, we have the node x. We'd like to know the rank of x and compare that to i. That will tell us whether we should go left, or go right, or whether we happen to find the item.

Now one possibility is we call rank of x to find the rank of x. But that's dangerous because I'm going to have a four loop here and it's going to take log n iterations. If at every iteration of

computing rank of x, and rank costs log n, then overall cost might be log squared n. So I can't afford to-- I want to know what the rank of x is but I can't afford to say rank, open paren, x. Because that recursive call will be too expensive. So what is the rank of x in this case? This is a little special. What's that?

AUDIENCE: Number of left children plus 1.

PROFESSOR: Number of left, or the size of the left subtree plus 1. Yep. Plus 1 if we're counting, starting at one. Very good. I'm slowly getting better. Didn't hit anyone this time. OK.

So at least for the root, this is the rank, and that only takes us constant time in the special case. So we'll have to check that it's still holds after I do the loop. But it will. So, cool. Now there are three cases. If i equals rank. If the rank we're searching for is the rank that we happen to have, then we're done, right? We just return x. That's the easy case.

More likely is that I will be either less than or greater than the rank of x. OK. So if i is less than the rank, this is fairly easy. We just say x equals x.left.

Did I get that right? Yep. In this case, the rank. So here we have x. It's at rank, rank. And then we have the left subtree and the right subtree. And so if the rank were searching for is less than rank, that means we know it's in here. So we should go left. And if we just said x equals x.left you might ask, well what rank are we searching for in here? Well, exactly the same rank. Fine. That's easy case.

In the other situation, if we're searching in here, we're searching for rank greater than rank. Then I want to go right but the new rank that I'm searching for is local to this subtree. I'm searching for i minus this stuff. This stuff is rank. So I'm going to let i be i minus rank.

Make sure I don't have any off by 1 errors. That seems to be right. OK. And then I do a loop. So I'll write repeat.

So then I'm going to go up here and say, OK. Now relative to this thing. What is the rank of the root of this subtree? Well, it's again going to be that node .left.size plus 1. And now I have the new rank I'm searching for, i. And I just keep going. You could write this recursively if you like, but here's an iterative version.

So it's actually very familiar to the select algorithm that we had, like when we did deterministic linear time median finding or randomized median finding. They had a very similar kind of

recursion. But in that case, they were spending linear time to do the partition and that was expensive. Here, we're just spending constant time at each node and so the overall cost is log n. So that's nice. Any questions about that?

OK. I have a note here. Subtree size is obvious once you know that's what you should do. Another natural thing to try to do would be to augment, for each node, what is the rank of that node? Because then rank is really easy to find. And then select would basically be a regular search. I just look at the rank of the root, I see whether the rank I'm looking for is too big, or too small, and I go left or right, accordingly.

What would be bad about augmenting with rank of a node? Updates. Why? What's a bad example for an update?

AUDIENCE: If you add new in home element.

PROFESSOR: Right. Say we insert a new minimum element.

Good catch, cameraman. That was for the camera, obviously. So, right. If we insert, this is off to the side, but say we insert, I'll call it minus infinity. A new key that is smaller than all other keys, then the rank of every node changes. So that's bad. It means that easy tree augmentation, in particular, isn't going to apply. And furthermore, it would take linear time to do this. And you could keep inserting, if you insert keys in decreasing order from there, every time you do an insert, all the ranks increase by one. Maintaining that's going to cost linear time per update.

So you have to be really careful that the function you want to store actually can be maintained. Be very careful about that, say, on the quiz coming up, that when you're augmenting something you can actually maintain it. For example, it's very hard to maintain the depths of nodes because when you do a rotation a whole lot of depths change.

Depth is counting from the root. How deep am I? When I do a rotation then this entire subtree went down by one. This entire subtree went up by one. In this picture. But it's very easy to maintain heights, for example. Height counting from the bottom is OK, because I don't affect the height of a, b, and c. I affect it for x and y but that's just two nodes. That I can afford. So that's what you want to be careful of in the easy tree augmentation.

So most the time easy tree augmentation does the job. But in the remaining two examples, I want to show you cooler examples of augmentation. These are things you probably wouldn't

be expected to come up with on your own, but they're cool. And they let us do more sophisticated operations.

So the first one is called level linking. And here we're going to do it in the context of 2-3 trees, partly for variety. So the idea of level linking is very simple. Let me draw a 2-3 tree.

Not a very impressive 2-3 tree. I guess I don't feel like drawing too much. Level linking is the idea of, in addition to these child and parent pointers, we're going to add links on all the levels. Horizontal links, you might call them.

OK. So that's nice. Two questions-- can we do this? And what's it good for? So let's start with can we do this. Remember in 2-3 trees all we have to think about are splits and merges. So in a split, we have, for a brief period, let's say three keys, four children. That's too many. So we change that to--

I'm going to change this in a moment. For now, this is the split you know and love, maybe. At least know. And if we think about where the leveling pointers are, we have one before. And then we just need to distribute those pointers to the two resulting nodes. And then we have to create a new pointer between the nodes that we just created. This is, of course, easy to do.

We're here. We're taking this node. We're splitting it in half. So we have the nodes right in our hands so just add pointers between them. And key thing is, there's some node over here on the left. It used to point to this node, now we have to change it to point to the left version. The left half of the node. And there's some node over on the right. We have to change it's left pointer to point to this right half of the node. But that's it. Constant time.

So this doesn't fall under the category of easy tree augmentation because this is not isolated to the subtree. We're also dealing with it's left and right subtrees. But still easy to do in constant time.

Merging nodes is going to be similar. If we steal a node from our parents or former sibling, nothing happens in terms of level links. But if we have, say, an empty node and a node that cannot afford any stealing. So we have single child here, two children, and we merge it into--

We're taking something from our parent. Bringing it down. Then we have three children afterwards. Again, we used to have these level pointers. Now we just have these level pointers. It's easy to maintain. It's just a constant size neighborhood.

Because we have the level links, we can get to our left and right neighbors and change where the links point to. So easy to maintain in constant time. I'll call it constant overhead. Every time we do a split or merge we spend additional constant time to do it. We're already spending constant time. So just changes everything by constant factor. So far, so good.

Now, I'm going to have to tweak this data structure a little bit. But let me first tell you why. What am I trying to achieve with this data structure? What I'm trying to achieve is something called the finger search property.

So let's just think about the case where I'm doing a successful search. I'm searching for key x and I find it in the data structure. I find it in the tree. Suppose I found one-- I search for x, I found it. And then I search for another key y. Actually I think I'll do the reverse. First I found y, now I'm searching for x. If x and y are nearby in the tree, I want this to run especially fast. For example, if x is the successor of y I want this to take constant time. That would be nice.

In the worst case x and y are very far away from me in the tree then I want it to take log n time. So how could I interpolate between constant time for finding the successor and log n time for finding the worst case search. So I'm going to call this search of x from y. Meaning, this is a little imprecise, but what I mean is when I call search, I tell it where I've already found y. And here it is. Here's the node storing y. And now I'm given a key x. And I want to find that key x given the node that stores key y. So how long should this take? Will be a good way to interpolate between constant time at one extreme. The good case, when x and y are basically neighbors in sorted order, versus log n time, in the worst case.

AUDIENCE: Distance along the graph.

PROFESSOR: Distance along the graph. That would be one reasonable definition. So I have a tree which you could think of as a graph. Measure the shortest path length from x to y. Or we have a more sophisticated graph over here. Maybe that length. The trouble with the distance in the graph, that's a reasonable suggestion, but it's very data structure specific. If I use an AVL tree without level links, then the distance could be one thing, whereas if I use a 2-3 tree, even without level lengths, it's going to be a different distance. If I use a 2-3 tree with level lengths it's going to be yet another distance. So that's a little unsatisfying. I want this to be an answer to a question. I don't want to phrase the question in terms of that data structure.

AUDIENCE: Difference between ranks of x and y?

PROFESSOR: Difference between ranks between x and y. That's close.

So I'm going to look at the rank of x and rank of y. Let's say, take the absolute difference. That's kind of how far away they are in sorted order. Do you want to add anything?

AUDIENCE: Log?

PROFESSOR: Log. Yeah. Because in the worst case the difference in ranks could be linear. So I want to add a log out here to get log n in that worst case.

Add a big o for safety. That's how much time we want to achieve. So this would be the finger search property that you can solve this problem in this much time. Again, difference in ranks is at most n. So this is at most log n. But if y is the successor of x this will only be constant and this will be constant.

So this is great if you're doing lots of searches and you tend to search for things that are nearby, but sometimes you search for things are far away. This gives you a nice bound.

On the one hand, we have, this is our goal. Log difference of ranks. On the other hand, we have the suggestion that what we can achieve is something like the distance in the graph.

But we have a problem with this. I used to think that data structure solved this problem, but it doesn't. Let me just draw-- actually I have a tree right there. I'm going to use that one. Suppose x is here and y is here. OK. This is a bit of a small tree but if you think about it long enough, this node is the predecessor of this node. So their difference in ranks should be 1.

But the distance in the graph here is two. Not very impressive. But in general, you have a tree of height log n. If you look at the root, and the predecessor of the root, they will have a rank difference of one by definition of predecessor. But the graph distance will be log n. So that's bad news, because if we're only following pointers there's no way to get from here to there in constant time. So we're not quite there.

We're going to use another tweak that data structure, which is store the data in the leaves. Tried to find a data structure that didn't require this and still got finger search. But as far as I know, there is none. No such data structure. If you look at, say, Wikipedia about B-trees, you'll see there's a ton of variations of B-trees. B+-trees, B*-trees. This is one of those. I think B+-trees.

As you saw, B-trees or 2-3 trees, every node stored one or two keys. And each key only existed in one spot. We're still only going to put each key in one spot, kind of. But it's only going to be the leaf spots. OK. Good news is most nodes are leaves, right? Constant fraction of the nodes are going to be leaves. So it doesn't change too much from a space efficiency standpoint. If we just put data down here and don't put-- I'm not going to put any keys up here for now.

So this a little weird. Let me draw an example of such a tree. So maybe we have 2, and 5, and 7, and 8, 9, let's say. Let's put 1 here. So I'm going to have a node here with three children, a node here with two children, and here's a node with two children. So I think this mimics this tree, roughly. I got it exactly right.

So here I've taken this tree structure. I've redrawn it. There's now no keys in these nodes. But everything else is going to be the same. Every node is going to have 0 children if it's a leaf, or two, or three children otherwise. Never have one child because then you wouldn't get logarithmic depth. All the leaves are going to be at the same depth.

And that's it. OK. That is a 2-3 tree with the data stored in the leaves. It's a useful trick to know. Now we're going to do a level linked 2-3 tree. So in addition to that picture, we're going to have links like this.

OK. And I should check that I can still do insert and delete into these structures. It's actually not too hard. But let's think about it.

I think, actually, it might be easier. Let's see. So if I want to do an insert-- OK. I have to first search for where I'm inserting. I haven't told you how to do search yet. OK. So let's first think about search.

What we're going to do is data structure augmentation. We have simple tree augmentation. So I'm going to do it and each node, what the functions I'm going to store are the minimum key in the subtree, and the maximum key in the subtree. There are many ways to do this, but I think this is kind of the simplest. So what that means is at this node, I'm going to store 1 as the min and 7 as the max.

And at this node it's going to be 1 at the min and 9 at the max. And here we have 8 as the min and 9 as the max. Again min and max of subtrees are easy to store. If I ever change a node I can update it based on its children, just by looking at the min of the leftmost child and the max

of the rightmost child. If I didn't know 1 and 9, I could just look at this min and that max and that's going to be the min and the max of the overall tree. So in constant time I can update the min and the max of a node given the min and the max of its children. Special case is at the leaves. Then you have to actually look at keys and compare them. But leaves only have, at most, two keys. So pretty easy to compare them in constant time. OK.

So that's how I do the augmentation. Now how do I do a search? Well, if I'm at a node and I'm searching for a key. Well, let's say I'm at this node. I'm searching for a key like 8. What I'm going to do is look at all of the children. In this case, there's two. In the worst case there's three. I look at the min and max and I see where does 8 fall? Well it falls in this interval. If I was searching for 7 1/2 I know it's not there. It's going to be in between here. If I'm doing a successor then I'll go to the right. If I'm doing predecessor I'll go to the left. And then take either the maximum item or the minimum item.

If I'm searching for 8 I see, oh. 8 falls in the interval between 8 and 9, so I should clearly take the right child among those two children. In general, there's three children. Three intervals. Constant time. I can find where my key falls in the interval. OK.

So search is going to take log n time again, provided I have these mins and maxs. If you stare at it long enough, this is pretty much the same thing as regular search in a 2-3 tree. But I've put the data just one level down. OK. Good.

That was regular search. I still need to do finger search, but we'll get there. And now, if I want to do an insert into this data structure, what happens. Well I search for the key let's say I'm inserting 6. So maybe I go here. I say because 6. Is in this interval. 6 is in neither of these intervals. But it's closest to the interval 2, 5, or the interval 7. Let's say I go down to 2, 5. And well, to insert 6 I'll just add a 6 on there. Of course, now that node is too big.

So there's still going to be a split case at the leaves where I have let's say, a,b,c, too many keys. I'm going to split that into a,b and c. This is different from before. It used to be I would promote b to the parent because the parent needed the key there. Now parents don't have keys. So I'm just going to split this thing, roughly, in half. It works. It's still the case that whoever was the parent up here now has an additional child. One more child. So maybe that node now has four children but it's supposed to be two or three. So if I have a node with four children, what I'm going to do, I'm suppose to use these fancy arrows. What do I do in this case? It's just going to split that into two nodes with two children. And again this used to have

a parent. Now that parent has an additional child, and that may cause another split.

It's just like before. Was just potentially split all the way up to the root. If we split the root then we get an additional level. But we could do all this and we can still maintain our level links, if we want.

But everything will take log n. I won't draw the delete case, as delete is slightly more annoying. But I think, in this case, you never have to worry about where is the key coming from, your child or your parent? You're just merging nodes so it's a little bit simpler. But you have to deal with the leaf case separately from the nonleaf case. OK.

So all this was to convince you that we can store data in the leaves. 2-3 trees still work fine. Now I claim that the graph distance in level link trees is within a constant factor of the finger search bound. So I claim I can get the finger search property in 2-3 trees, with data in the leaves, with level links. So lots of changes here. But in the end, we're going to get a finger search bound. Let's go over here.

So here's a finger search operation. First thing I want to do is identify a node that I'm working with. I want to start from y's node. So we're supposing that we're told the node, a leaf, that contains y. So I'm going to let v be that leaf.

OK. Because we're supposing we've already found y, and now all the data is in the leaves. So give me the leaf that contains y. So that should take constant time. That's just part of the input.

Now I'm going to do a combination of going up and horizontal. So starting at a leaf. And the first thing I'm going to do is check, does this leaf contain what I want? Does it contain the key I'm searching for, which is x? So that's going to be the case. At every node I store the min and the max. So if x happens to fall between the min and the max, then I'm happy.

Then I'm going to do a regular search in v's subtree. This seems weird in the case of a leaf. In the case of a leaf, this is just to check the two keys that are there. Which one is x. OK. But in general I gave you this search algorithm which was, if I decide which child to take, according to the ranges, that's a downward search. So that's what I'm calling regular search here. Maybe downward would be a little better.

This is the usual log n time thing. But we're going to claim a bound better than log n. If this is not the case, then I know x either falls before v.min or after v.max.

So if x is less than v.min then I'm going to go left. v equals v. I'll call it level left to be clear. You might say left is the left child. There's no left child here, of course. But level left is clear. We take the horizontal left pointer. And otherwise x is greater than v.max. And in that case I will go right. That seems logical.

And in both cases we're going to go up. x equals x.parent Whoops. v equals v.parent. X is not changing here. X is a key we're searching for. v is the node. V for vertex. So we're always going to go up, and then we're going to go either left or right, and we're going to keep doing that until we find a subtree that contains x in terms of key range. Then we're going to stop this part and we're just going to do downward search. I should say return here or something. I'm going to do a downward search, which was this regular algorithm. And then whatever it finds, that's what I return.

I claim the algorithm should be clear. What's less clear is that it achieves the bound that we want. But I claim that this will achieve the finger search property. Let me draw a picture of what this thing looks like kind of generically. On small examples it's hard to see what's going on. So I'm going to draw a piece of a large example.

Let's say we start here. This is where y was. I'm searching for x. Let's suppose x is to the right. 'Cause otherwise I go to the other board. So x is to the right. I'll discover that the range with just this node, this node maybe contains one other key. I'll find that range is too small. So I'm going to go follow the level right pointer, and I get to some other node.

Then I'm going to go to the parent. Maybe the parent was the parent of those two children so I'm going to draw it like that. Maybe I find this range is still too low. I need to go right to get to x, so I'm going to follow a level pointer to the right. I find a new subtree. I'll go to its parent. Maybe I find that this subtree, still the max is too small. So I have to go to the right again. And then I take the parent. So this was an example of a rightward parent. Here's an example of a leftward parent. This is maybe the parent of both of these two children.

Then maybe this subtree is still too small, the max is still smaller than x. So then I go right one more time. Then I follow the parent. Always alternating between right and parent until I find a node whose subtree contains x. It might have actually, x may be down here, because I immediately went to the parent without checking whether I found where x is.

But if I know that x is somewhere in here then I will do a downward search. It might go left and then down here, or it might go right, or there's actually potentially three children. One of these

searches will find the key x that I'm looking for because I'm in the case where x is between v.min and v.max, so I know it's in there, somewhere. It could be x doesn't exist, but it's predecessor or successor is in there somewhere.

And so one of these three subtrees will contain the x range. And then I go follow that path. And keep going down until I find x or it's predecessor or successor. Once I find it's predecessor I can use a level right pointer to find its successor, and so on.

So that's kind of the general picture what's going on. We keep going rightward and we keep going up. Suppose we do k up steps. Let's look at this last step here. Step k.

How high am I in the tree? I started at the leaf level. Remember in a 2-3 tree all the leaves have the same level. And I went up every step.

Sorry. I don't know what this is, like the 2-step dance where, let's say every iteration of this loop I do one left or right step, and then a parent step. So I should call this iteration k. I guess there's two k steps, then.

Just to be clear. So in iteration k, that means I've gone up k times and I've gone either right or left k times. You can show if you start going right you keep going right. If you initially go left you'll keep going left. Doesn't matter too much.

At iteration k I am at height k, or k minus 1, or however you want to count. But let's call it k. So when I do this right pointer here I know that, for example, I am skipping over all of these keys. All the keys down-- the keys are in the leaves, so all these things down here, I'm jumping over them. How many keys are down there? Can you tell me, roughly, how many keys I'm skipping over when I'm moving right at height k? It's not a unique answer. But you can give me some bounds.

Say again. Number of children to the k power. Yeah. Except we don't know the number of children. But it's between 2 and 3 Closer one should be easy but I fail. So it's between two and three children. So there's the number-- if you look at a height k tree, how many leaves does it have? It's going to be between 2 to the k and 3 to the k. Because I have between 2 and 3 children at every node. And so it's exponential in k. That's all I'll need.

OK. When I'm at height k here, I'm skipping over a height k minus 1 tree or something. But it's going to be--

So in iteration k I'm skipping, at least, some constant times 2 to the k. Maybe to the k minus 1, or to the k minus 2. I'm being very sloppy. Doesn't matter. As long as it's exponential in k, I'm happy. Because I'm supposing that x and y are somewhat close. Let's call this rank difference d. Then I claim the number of iterations I'll need to do in this loop is, at most, order log d. Because if, when I get to the k-th iteration, I'm jumping over 2 to the k elements. How large does k have to be before 2 to the k is larger than d? Well, log d. Log base 2

The number of iterations is order log d, where d is the rank difference. d is the absolute value between rank of x and rank of y. And I'm being a little sloppy here. You probably want to use an induction. You need to show that they're really, these items here that you're skipping over that are strictly between x and y. But we know that there's only d items between x or y. Actually d minus 1, I guess. So as soon as we've skipped over all the items between x and y, then we'll find a range that contains x, and then we'll go do the downward search.

Now how long does the downward search cost? Whatever the height of the tree is. What's the height of the tree? That's the number of iterations. So the total cost. The downward search will cost the same as the rest of the search. And so the total cost is going to be order log d. Clear?

Any questions about finger searching with level linked data at the leaves, 2-3 trees?

AUDIENCE: Sir, I'm not sure why [INAUDIBLE] d, why is that?

PROFESSOR: I'm defining d to be the rank of x minus rank of y. My goal is to achieve a log d bound. And I'm claiming that because once I've skipped over d items, then I'm done. Then I've found x. And at step k I'm skipping over 2 to the k items. So how big is k going to be? Log d. That's all. I used d for a notation here. Cool.

Finger searching. It's nice. Especially if you're doing many consecutive searches that are all relatively close to each other. But that was easy. Let's do a more difficult augmentation.

So the last topic for today is range trees. This is probably the coolest example of augmentation, at least, that you'll see in this class. If you want to see more you should take advanced data structure 6851.

And range trees solve a problem called orthogonal range searching. Not orthogonal search ranging. Orthogonal range search.

So what's the problem? I'm going to give you a bunch of points. Draw them as fat dots so you

can actually see them. In some dimension. So this is, for example, a 2D point set. OK. Over here I will draw a 3D point set. You can tell the difference, I'm sure.

There. Now it's a 3D point set. And this is a static point set. You could make this dynamic but let's just think about the static case. Don't want the 2D points and the 3D points to mix. Now, you get to preprocess this into a data structure. So this is a static data structure problem. And now I'm going to come along with a whole bunch of queries. A query will be a box. OK. In two dimensions, a box is a rectangle.

Something like this. Axis aligned. So I give you an x min, x max, a y min, and a y max. I want to know what are the points inside. Maybe I want you to list them. If there's a lot of them it's going to take a long time to list them. Maybe I just want to know 10 of them as examples.

Maybe this is a Google search or something. I just get the first 10 results in the first page, I hit next then want the next 10, that kind of thing. Or maybe I want to know how many search results there are. Number of points in the rectangle. Bunch of different problems.

In 3D, it's a 3D box. Which is a little harder to draw. You can't really tell which points are inside the box. Let's say these three points are all inside the box. I give you an interval in x, an interval in y, and an interval in z, and I want to know what are the points inside. How many are there? List them all. List 10 of them, whatever. OK.

I want to do this in poly log time, let's say. I'm going to achieve today log squared for the 2D problem and log cubed for the 3D problem, plus whatever the size output is. So let me just write that down. So the goal is to preprocess n points in d dimensions.

So you get to spend a bunch of time preprocessing to support a query which is, given a box, axis aligned box, find let's say the number of points in the box. Find k points in the box. I think that's good. That includes a special case of find all the points in the box. So this, of course, we have to pay a penalty of order k for the output. No getting around that. But I want the rest of the time to be log to the d.

So we're going to achieve log to the d n plus size of the output. And you get to control how big you want the output to be. So it's a pretty reasonable data structure. In a certain sense we will understand what the output is in log to the d time. If you actually want to list points, well, then you have to spend the time to do it.

All right. So 2D and 3D are great, but let's start with 1D. First we should understand 1D

completely, then we can generalize. 1D we already know how to do. 1D I have a line. I have some points on the line.

And I'm given, as a query, some interval. And I want to know how many points are in the interval, give me the points in the interval, and so on. So how do I do this? Any ways?

If d is 1. So I want to achieve log d, sorry, log n, plus size of output. I hear whispers. Yeah?

AUDIENCE: Segment trees?

PROFESSOR: Segment tree? That's fancy. We won't cover segment trees. Probably segment trees do it. Yeah. We know lots of ways to do this. Yeah?

AUDIENCE: Sorted array?

PROFESSOR: Sorted array is probably the simplest. If I store the items in a sorted array and I have two values, I'll call them x1 and x2, because it's the x min and x max. Binary search for x1. Binary search for x2. Find the successor of x1 and the predecessor of x2. I'll find these two guys. And then I know all the ones in between. That's the match. So that'll take log n time to find those points and then we're good.

So we could do a sorted array. Of course, sorted array is a little hard to generalize. I don't want to do a 2D array, that sounds bad. You could, of course, do a binary search tree. Like an AVL tree. Same thing. Because we have log n search, find successor, and predecessor, I guess you could use Van Emde Boas, but that's hard to generalize to 2D.

You could use level links. Here's a fancy version. We could use level linked 2-3 trees with data in the leaves. Then once I find x min, I find this point, I can go to the successor in constant time because that's a finger search with a rank difference of 1. And I could just keep calling successor and in constant time per item I will find the next item. So we could do that easily with the sorted array.

BST is not so great because successor might cost log n each time. But if I have the level links then basically I'm just walking down the link list at the bottom of the tree. OK. So actually level linked is even better. BST would achieve something like log n plus k log n, where k is the size of the output. If I want k points in the box I have to pay log n. For each level linked I'll only pay log n plus k. Here I actually only need the levels at the leaves. Level links.

OK. All good. But I actually want to tell you a different way to do it that will generalize better. The pictures are going to look just like the pictures we've talked about.

So these would actually work dynamically. My goal here is just to achieve a static data structure. I'm going to idealize this solution a little bit. And just say, suppose I have a perfectly balanced binary search tree. That's going to be my data structure. OK. So the data structure is not hard, but what's interesting is how I do a range search.

So if I do range query of the interval, I'll call it ab. Then what I'm going to do is do a binary search for a, do a binary search for b, trim the common prefix of those search paths. That's basically finding the lowest common ancestor of a and b.

And then I'm going to do some stuff. Let me draw the picture. So here is, suppose here's the node that contains a. Here's the node that contains b. They may not be at the same depth, who knows. Then I'm going to look at the parents of a. I just came down from some path here, and some path down to b. I want to find this branching point where the paths to a and the paths to b diverge.

So let's just look at the parent of a. It could be a right parent, in which case there's a subtree here. Could be a left parent in which case, subtree here. I'm going to follow my convention again. That x-coordinate corresponds roughly to key. Left parent here. Maybe right parent here. Something like that.

OK. Remember it's a perfect tree. So, actually, all the leaves will be at the same level. And, roughly here, x-coordinate corresponds to key. So here is a. And I want to return all the keys that are between a and b. So that's everything in this sweep line.

The parents of the LCA don't matter, because this parents either going to be way over to the right or way over to the left. In both cases, it's outside the interval a to b. So what I've tried to highlight here, and I will color it in blue, is the relevant nodes for the search between a and b. So a is between a and b. This subtree is greater than a and less than b. This node, and these nodes. This node, and these nodes. This node and these nodes. The common ancestor. And then the corresponding thing over here. All the nodes in all these blue subtrees, plus these individual nodes, fall in the interval between a and b, and that's it.

OK. This should look super familiar. It's just like when we're computing rank. We're trying to figure out how many guys are to our left or to our right. We're basically doing a rightward rank

from a and a leftward rank from b. And that finds all the nodes. And stopping when those two searches converge. And then we're finding all the nodes between a and b. I'm not going to write down the pseudocode because it's the same kind of thing. You look at right parents and left parents.

You just walk up from a. Whenever you get a right parent then you want that node, and the subtree to its right. And so that will highlight these nodes. Same thing for b, but you look at left parents. And then you stop when those two searches converge. So you're going to do them in lock step. You do one step for a and b. One step for a and b. And when they happen to hit the same node, then you're done. You add that node to your list. And what you end up with is a bunch of nodes and rooted subtrees.

The things I circled in blue is going to be my return value. So I'm going to return all of these nodes, explicitly. And I'm also going to return these subtrees. I'm not going to have to write them down. I'm just going to return the root of the subtree, and say, hey look. Here's an entire subtree that contains points that are in the answer. Don't have to list them explicitly, I can just give you the tree.

Then if I want to know how many results are in the answer, well, just augment to store subtree size at the beginning. And then I can count how many nodes are down here, how many nodes are down here, add that up for all the triangles, and then also add one for each of the blue nodes, and then I've counted the size of the answer in how much time? How many subtrees and how many nodes am I returning here? Log.

Log n nodes and log n rooted subtrees because at each step, I'm going up by one for a, and up by one for b. So it's like 2 log n. Log n.

So I would call this an implicit representation of the answer. From that implicit representation you can do subtree size. Augmentation to count the size the answer. You can just start walking through one by one, do an inter traversal of the trees, and you'll get the first k points in the answer in order k time. Question?

AUDIENCE: Just a clarification. You said when we were walking up, you want to get all the ancestors in their right subtrees. But you don't do that for the left parent, right?

PROFESSOR: That's right. As I'm walking up the tree, if it's a right parent then I take the right subtree and include that in the answer. If it's a left parent just forget about it. Don't do anything. Just keep

following parents. Whenever I do right parent then I also add that node and the right subtree. If it's a left parent I don't include the node, I don't include the left subtree. I also don't include the right subtree. That would have too much stuff.

It's easy when you see the picture, you would write down the algorithm. It's clear. It's left versus right parents.

AUDIENCE: Would you include the left subtree of b?

PROFESSOR: I would also-- thank you. I should color the left subtree of b. I didn't apply symmetry perfectly. So we have the right subtree of a and the left subtree of b. Thanks. I would also include b if it's a closed interval.

Slightly more general. If a and b are not in the tree then this is really the successor of a and this is the predecessor of b. So then a and b don't have to be in there. This is still a well defined range search. OK. Now we really understand 1D. I claim we've almost solved all dimensions. All we need is a little bit of augmentation. So let's do it.

Let's start with 2D. But then 3D, and 4D, and so on will be easy. Why do I care about 4D range trees? Because maybe I have a database. Each of these points is actually just a row in the database which has four columns, four values. And what I'm trying to do here is find all the people in my database that have a salary between this and this, and have an age between this and that, and have a profession between this and this. I don't know what that means. Number of degrees between this and this, whatever.

You have some numerical data representing a person or thing in your database, then this is a typical kind of search you want to do. And you want to know how many answers you've got and then list the first hundreds of them, or whatever. So this is a practical thing in databases. This is what you might call an index in the database.

So let's start. Suppose your data is just two dimensional. You have two fields for every item. What I'm going to do is store a 1D range tree on all points by x. So this data structure makes sense if you fix a dimension. Say x is all I care about. Forget about y. So my point set. Yeah. So what that corresponds to is projecting each of these points onto the x-axis. And now also projecting my query.

So my new query is from here to here in x. And so this data structure will let me find all these points that match in x. That's not good because there's actually only two points that I want, but

I find four points in this picture. But it's half of the answer. It's all the x matches forgetting about y.

Now here's the fun part. So when I do a search here I get log n nodes. Nodes are good because they have a single key in them. So I'll just check for each of those log n nodes. Do they also match in y? If they do, add it to the answer. If they don't forget about it. OK.

But the tricky part is I also get log n subtrees representing parts of the answer. So potentially it could be that your search, this rectangle, only has like five points. But if you look at this whole vertical slab there's a billion points. Now, luckily, those billion points are represented succinctly. There's just log n subtrees saying, well there's half a billion here, and a quarter billion here, and an eighth of a billion here.

Now for each of that big chunk of output, I want to very quickly find the ones that match in y. How would I find the ones matching in y? A range tree. Yeah. OK. So here's what we're going to do. For each node, call it x. x is overloaded. It's a coordinate. So many things. Let's call it v. In the, this thing I'm going to call the x-tree. So for every node in the x-tree I'm going to store another 1D range tree. But this time using the y-coordinate on all points in these rooted subtree.

At this point I really want to draw a diagram. So, rough picture. Forgive me for not drawing this perfectly.

This is roughly what the answer looks like for the 1D range search. This is the x-tree. And here I've searched between this value and this value in the x-coordinate. Basically I have log n nodes. I'm going to check those separately. Then I also have these log n subtrees. For each of those log n sub trees I'm going to have a pointer-- this is the augmentation-- to another tree of exactly the same size. On exactly the same data that's in here. It's also over here. But it's going to be sorted by y. And it's a 1D range tree by y. Tons of data duplication here. I took all these points and I copied them over here, but then built a 1D range tree in y. This is all preprocessing. So I don't have to pay for this. It's polynomial time. Don't worry too much.

And then I'm going to search in here. What does the search in there look? I'm going to get, you know, some more trees and a couple more nodes. OK. But now those items, those points, match in x and y because this whole subtree matched in x and I just did a y search, so I found things that matched in y.

So I get here another log n trees that are actually in my answer. And for each of these nodes I have a corresponding other data structure where I do a little search and I get part of the answer.

Every one. Sounds huge. This data structure sounds huge, but it's actually small. But one thing that's clear is it takes log squared n time, because I have log n triangles over here. For each of them I spend log n to find triangles over here. The total output is log squared n nodes, for each of them I have to check manually. Plus, so over here, there's log n, different searches I'm doing. Each one has size log n. So I get log squared little triangles that contain the results that match in x and y.

How much space in this data structure? That's the remaining challenge. Actually, it's not that hard, because if you look at a key. So look at some key in this x-tree. Let's look at a leaf because that's maybe the most interesting.

Here's the x-tree. x-tree has linear size. Just one tree. If I look at some key value, well, it lives in this subtree. And so there's going to be a corresponding blue structure of that size that contains that key. And then there's the parent. So there's a structure here. That has a corresponding blue triangle. And then its parent, that's another triangle. That contains-- I'm looking at a key k here. All of these triangles contain the key k. And so key k will be duplicated all this many times, but how many sub trees is k in? Log n. Each key, fundamental fact about balanced binary search trees, each key lives in log n subtrees. Namely all of its ancestors.

Awesome. Because that means the total space is n log n. There's n keys. Each of them is duplicated at most log n times. In general, log to the d minus 1. So If you do it in 3D, each of the blue trees, every node in it has a corresponding pointer to a red tree that's sorted by z. And you just keep doing this, sort of, nested searching, like super augmentation. But you're only losing a log factor each dimension you add.