[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: Welcome to our fourth problem session. We're going to be talking about binary trees mostly, today. We'll talk a little bit about binary heaps, which is a topic we won't cover until next Tuesday, but it will appear in very small ways on your problem set 4, which will be due next Friday. So I'm going to go over a little bit of that material today. But it's mostly concerning-- the subject material for today is mostly binary trees, specifically, being applied to set data structures and sequence data structures, as Professor Demaine talked to you earlier this week.

But for now-- actually, as of yesterday-- you've seen all of the data structures that we're going to cover to-- that will implement the set interface and the sequence interface. Those nice tables that Professor Demaine has been showing you, those are now complete. We have some data structures that are really good-- constant time operations for some operations. So we might use them for some applications. And this week, we've been describing to you trees, which achieve, really, pretty good, for any type of query operation on my sets or sequences-- pretty good meaning logarithmic time, not quite constant.

But for our purposes, log n is-- I mean, on your computer, practically-- not asymptotically, but practically-- log n is going to be at most what on your computer? Something like 64, right? Any input that you're operating on with, in machine words, is your input. You need to be able to address all those machine words in your input. And on your computer, the size of your machine word addresses is 64 bits, right? And we assume that the word size is at least log the size of your input so that you can address the input.

So for your purposes, on your computer, log n is going to be no more than 64, which means you would get maybe a 50 times overhead, or for smaller instances, it could be more like 10, if you've got 1,000 things that you're working on. It's not that bad, right? It's a constant-- it's not a constant factor for theory purposes, but for your purposes, log n is much better than a polynomial factor-- another factor of n.

You've seen all of the code. You've seen implementations of all of these set and sequence interfaces, right? So I went ahead and wrote a little-- I compiled all of that code from your recitation notes, of all of the different interface implementations. And what I did was, I wrote a little test program to see how they ran on a real machine. I have a little test code here. I have a little folder that lists an array implementing a sequence, a binary tree implementing a sequence, a dynamic array implementing-- all of these kinds of things. Then set things-- a sorted array being a set in a binary tree, and a hash table.

These are our implementations. I'm not using Python dictionaries for hash tables, I'm using the implementations that are in your recitation. And I'm going to run this little test efficiency Python code that basically is just going to free each one. It's going to do a bunch of these different operations and measure to see how much time it took. I'm just logging how much time it took. It's not an asymptotic analysis, but hopefully, we see some separation. So when you press that, it runs a bunch of tests. Let's take a look. OK. I've got a bunch of sequence operations. We've got build, set_at, get-at, insert, delete, at the various places. And these are the actual timings to some scale-- to some resolution that I had for these data structures.

And you can see build-- actually, build, on this machine, just allocating some array and clearing it, is a really efficient thing that Python is going to do for me. And so that's actually-- it's mislabelling that as log n. But these other things, get_at and set_at-- really, really fast, right? That's constant time. And then these other things, I essentially can't do better than loop through the thing, and so it takes linear time. And again, sequence stuff, setting_at and getting_at is slow, but deleting and removing from the first thing, I'm just re-linking the pointer, right? Dynamic arrays. Again, set_at, get_at is fast, because it's just regular arrays. And then inserting- and deleting-last, that's getting, essentially, constant time.

Now, I'm actually-- when I'm running these tests to deal with averages, I'm actually running these things, a lot of times, and testing their performance. And so I'm not seeing the worst case happen here, right? I'm averaging over all of the things, which is exactly what amortization means. That's why I'm getting good performance here. A hash table. Again, really-- oh, so this is what we talked about in problem session last week, implementing kind of a double-ended queue with a hash table. This is that implementation. I just wanted to show it to you. But it's actually pretty good. This is what JavaScript uses for arrays.

And then a binary sequence represented as a binary tree-- a balanced binary tree. This is our AVL code that I had. And all of the other things have been really pretty bad at insert_at and delete_at, but this one does comparable to all of the other things. Now, you see these are a little bit more machine cycles than the other things, but not so bad, actually. And then, on the set side of things, again, we had a sorted array. Sorry, this is a set from an array. Basically, it's an unsorted array. I'm just looking for all of the things-- that's very bad times.

Sorted array does these find operations great, but inserting and deleting is poor. That's why we need binary trees. Hash tables get good dictionary operations, but really bad order operations. And then the binary search tree, a set binary tree, again, does quite good on all of these things. In fact, it's getting really quite good-- it's getting better, for some reason, than the sorted array even did. I don't know why. Our implementations are not optimized at all. But it does pretty well asymptotically. Yeah?

AUDIENCE: Could you explain again why the first data treasury [INAUDIBLE] log n?

JASON KU: This is just labeled based off the timings. It happens to be that, probably, there's a C intrinsic underneath Python that allocates this thing, and so it does it really fast. My program that's looking at these numbers and trying to guess what the asymptotic running time is, these are just labels based on these things, ranges. I just-- it's mischaracterized.

AUDIENCE: [INAUDIBLE]

JASON KU: Yeah. Well, I mean, in actuality, if it was C code-- if all of this stuff was in C, probably, we would see that bar be longer, because it's actually having to go through and touch all of that memory. It's still doing that here, but all of the Python stuff is super crufty. It's, like, 100 times slower than anything that C does. And so you're seeing that disparity. Does that make sense? OK. I just wanted to show you that. We might release this for you to play around with, but just wanted to give you a taste of that. OK, any questions before we move on? How do I turn this off? Up, and off. Shut down. Yes. OK.

Moving on to problems-- working some problems. You have your set of problems here. The first one is, we're going to look at a sequence AVL tree. This is a sequence AVL tree. How do I know that? You don't, necessarily. But these things are certainly not in sorted order of the things I'm storing in them, Yeah? So it had better not be a set AVL tree. Is it an AVL tree? Is it balanced-- height balanced? Yeah, basically. Actually, if you compute the size of each subtree, the left and right subtrees on all of these-- you can confirm for yourself-- are balanced. They're within plus or minus 1 of each other.

Actually, this is about as far away from balanced as you could get for this many nodes while still maintaining height balance-- maintaining AVL property-- which is why this is an instructive example. It's kind of at the limit. And what am I going to do? What's missing in this picture if I'm claiming this is a sequence AVL tree? Any ideas what's missing? What does the sequence AVL tree store that I'm not showing in this picture?

AUDIENCE: Counts.

JASON KU: What?

AUDIENCE: Counts.

JASON KU: Counts. And? It's a sequence AVL tree. Heights, right? Sequence AVL trees, different than set AVL trees, are augmented by two things, right? Because I need to be maintaining balance during rotations, and so I need to store heights. I need to be able to tell what the heights of these subtrees are in constant time when I'm walking up the tree, fixing things. And the sequence requires me to store their subtree numbers there. I don't know-- I'm not going to draw it for all of these things, but how about for number four?

What's its height? 1, 2, 3. That's the longest path from the root subtree of that. So this is height equals 3. That came from height equals 2 and height equals 1. Does everyone see that? Yeah? And then the size here, how big is that? That's 1, 2, 3, 4, 5, 6, 7. This is-- I'm going to put size equals 7 here. And that's coming from this guy, 1, 2, 3, 4. And this guy is 2. So how do I compute the subtree size? It's my left subtree size plus my right subtree size plus 1. And my height is taking the max of the 2 plus 1. All right. So we did all that yesterday. I'm just labeling these things.

And what I'm asking of you is to perform a delete operation. This is a sequence tree. So I'm finding things by their index in the tree. So I'm going to ask you to delete the eighth thing in my sequence. What is the 8th thing in my sequence? Yeah?

AUDIENCE: Just to clarify, since delete-8 is not delete the number.

JASON KU: Correct. Well, delete_at 8. Do you see that? It's a sequence operation.

AUDIENCE: Oh, OK. [INAUDIBLE]

JASON KU: Yeah. So this is very important, that you differentiate between sequence and set semantics. If I'm dealing with the sequence, I had better not be looking up intrinsic things on this data structure, because it's not an intrinsic data structure. It doesn't support that. If I wanted to support find, say, the index of key 8, or something like that, then all I could do is-- it's similar to an array. I would just have to loop through the whole sequence and tell me if the thing is in it. Can't really do better than linear time. This data structure is not designed for that. What is it designed for? It's designed to find things by their index in the sequence.

So how do I find the 8th index? Well, I mean, I'm looking at the tree, and I can just count along in-order traversal. What's the in-order traversal? 0, 1, 2, 3, 4, 5, 6, 7. OK, found 8. But what does a sequence AVL tree do? I'm storing subtree sizes, and when I'm here, I don't know what index I'm at.

How can I find out what index I'm at from the root? I look at my left subtree, see how many it is. There are seven things here. 1, 2, 3, 4, 5, 6, 7. Yeah. Because I'm looking for the 9th item by index 8. This is saying that I'm the 8th item. I'm the guy at index 7. Does that make sense? Because I'm looking at the subtree size here. So what do I know? I know that the index that I'm looking for is to my right.

I go down over here and I happen to know-- what index am I looking for in this subtree? 0, right? I want the first thing in the subtree. My search index has changed now, because I essentially dealt with all of those eight items. Here, I'm looking for the 0th thing in my index. I look to my left-- if I didn't have a left subtree, I would be the 0th thing, and I would return me. But there is stuff in here. So I'm looking for the 0th thing in here, which is just him, and I return it. And actually, what I'm doing is, I'm deleting it. So I delete it. Yuck. What's the problem here?

AUDIENCE: [INAUDIBLE]

JASON KU: Not height balanced. What's not height balanced here?

AUDIENCE: The left subtree is-- or, sorry. [INAUDIBLE]

JASON KU: This guy is not height balanced, right?

AUDIENCE: --right subtree of the right [INAUDIBLE].

JASON KU: This guy's subtree is not height balanced, right? This guy's 2. This guy's 1. So how do we fix it?

AUDIENCE: Rotate.

JASON KU: We do some rotations. This is actually the bad case that-- the third bad case that we talked about yesterday. If I tried to just left-rotate this guy, what would it look like? It would put 12 here. It would put 10 here. And 8 would be attached to that. Now it's height balanced wrong in the other direction, right? That's no good. So the way to handle this case, where I am badly skewed to the right but my right subtree is skewed to the left, I have to do a rotation here, right rotation, and then do a rotation. That's the formula. Here, we first do a right rotation at 10, which gives me something that looks like 8, 10.

Now, obviously, this is not better than what was before, but it was an intermediate step so that we can fix it. We right-rotate here and then we left-rotate here. The default is that we would left-rotate here, but because this had the skew in the wrong direction, I need to right-rotate this one first and then we can do it. So now I rotate all of these guys over and put 12 down here, 8 here, 10 here. Everyone see that that's what a rotation looks like? OK. It takes a little while to get your mind wrapped around what the transformation is, but hopefully, you guys all followed that transformation. There was a little magic while I was trying to draw. Yeah?

AUDIENCE: I still don't feel like this tree is height balanced.

JASON KU: It's not. Good observation. Why is that? This thing still has height 3. What is the height of this thing? 1, right? This is height 1. And actually, when I was doing that rotation, I needed to update all of these augmentations. Which augmentations did I really need to-- which subtrees have changed during those things? I don't remember what the thing looked like. What did the thing look like? 10 had 8 in its subtree, so its subtree definitely changed. 8's subtree changed.

AUDIENCE: [INAUDIBLE]

JASON KU: 12 didn't change-- eventually.

AUDIENCE: These are 10 and 8 [INAUDIBLE].

JASON KU: OK. So there's the case analysis that's in your lecture notes and was done in recitation. It tells you that these A, B, C, D kind of subtress, the ones that could change in these things, those subtrees don't change. The only subtree that changed during one of these fix operations, when you do one or two rotations, is either two nodes or three nodes whose subtree has changed. Here, it could have been the case that three subtrees have changed. But in the easy case, only two nodes-- x and y, I think, in the notes-- could have changed. And so when I do that, I have to recompute their augmentations from their augmentations of their children, but it's only a constant number of those, so I just recompute them, because the subtrees below me haven't changed. OK. So we have a height mismatch here. Yeah?

AUDIENCE: [INAUDIBLE]

JASON KU: Right. So, originally, in the picture, 12 has a bunch of things in its subtree-- 10 and 8, and we just deleted 7. So its subtree definitely changed. There used to be three--

AUDIENCE: [INAUDIBLE]

JASON KU: Oh, no, sorry, it did. Yeah. So here, three node subtrees have changed. But that's actually the most. I'm showing you the worst case. Only three nodes possible in doing one of these double-rotation things could have changed their subtrees. And so we just have to fix the augmentation of those three things. In the easy case, it's just two things. All right. We have unbalanced. How can we fix this? I could have been mean. I want to be able to right-rotate here, to re-balance I could have been mean and switched these two.

If I switched those two, then I would have to do two rotations to fix this thing, because the middle one is heavier than the left one against what I'm doing. But I'm not that mean, so I'm going to right-rotate. How do I do that? Well, a right-rotate at 6 is going to bring all of this down below 4 and stick this subtree as the left child of 6. Does that makes sense? Yuck. That's going to be fun to draw. I'm just going to redraw it. That makes more sense, right? 4, 11, 3, 2, 1, and then 6, 5, 9, 8, 12, 10. That's the right rotation at 6. Is everyone cool with this?

The rotation-- my x is 6, my y is 4. I have A, B, C subtrees. What I'm doing is kind of switching which of x and y is the root here. So now y is the root, and B and C subtrees here now become the children of x underneath y. And notice that, hopefully, through all of that process, my in-order traversal has not changed. We had to update our augmentations along the way, but it's a constant every time we walk up the tree. And we walk up the tree only a logarithmic number of times. Yeah? Yes.

AUDIENCE: [INAUDIBLE]. So every time we do a rotation, do you just update the augmentation via the [INAUDIBLE] before we do any other rotation?

JASON KU: Exactly.

AUDIENCE: The second part. Updating the augmentation just means updating the count, and the height, and just, the properties that stay--

JASON KU: Yeah. Basically what we did, we defined-- when we augmented-- Professor Demaine yesterday defined for you what a subtree property is. It meant a property that I can compute only by looking at my children-- the augmentations of my children, recursively. So here, instead of trying to increment or try to think about, locally, what this augmentation should be, I'm going to throw away my old augmentation and just recompute it from my children, because those, recursively, had better be correct. Does that make sense? Yeah?

AUDIENCE: So just looking at how the rotation works. I'm having trouble wrapping my head around. So basically, you're swapping 4 and 6, and that way, 4 becomes the parent node and 6 becomes the right node.

JASON KU: I'm going to draw this picture. It's just something you've got to memorize. This is x, B, C, and A. Can you see that picture?

AUDIENCE: [INAUDIBLE]

JASON KU: What?

AUDIENCE: [INAUDIBLE]

JASON KU: Yeah. It's in your notes. It's not a big deal. But if you've got this structure, where x has a left child-- and these subtrees may be empty or not. Doesn't really matter. What I can do is, I can move from here to there-- has the same in-order traversal order, but it's got a different shape. And in particular, subtree heights have changed, which means it can help us re-balance the tree. And that's the whole point of AVL. Does that make sense?

AUDIENCE: That's the right rotation?

JASON KU: This one is-- this is a right rotation. This is a left rotation. Any other questions? Yeah?

AUDIENCE: [INAUDIBLE]

JASON KU: As I'm walking up the tree, every node I might have to fix with a re-balance, but that re-balance does at most two rotations, and there is at most log n ancestors that I have, because my tree was height balanced at 2 log n or something like that. Which means that, at max, I might have to do four log n rotations, because each one could do two rotations. Does that makes sense? Now, in actuality, you can prove that, in a delete operation, it's possible that you have to do a logarithmic number of these rotations up the tree.

This was that bad case. The original tree I gave you is called a Fibonacci tree. It's the few-- it's the highest height balanced tree you can have on a given number of nodes. Yeah, the fewest nodes for a certain height. You can think of it either way. And if you generalize that to a large enough thing, then that thing will take a logarithmic number of rotations going up. Now, actually, with an insertion, you can actually prove-- you can go through the case analysis. An insertion operation will always re-balance the tree after one re-balance operation, which could include two rotations. Does that makes sense? Yeah?

AUDIENCE: [INAUDIBLE]

JASON KU: Yeah. Right rotation, this guy becomes a right child.

AUDIENCE: Yeah. So can you not-- are there certain rotations that you can't perform, depending on whether you had a child--

JASON KU: Yeah. If I didn't have a left subtree, can't perform a right rotation there. A right rotation necessitates that I have a left child. So if you're doing it-- and you'll see, our code actually checks to make sure you have a left child. That's an assertion that you might want to fire before you ever do one of these rotations. Anything else? Yeah?

AUDIENCE: Just to reiterate, so an insertion may take two rotations at most [INAUDIBLE]?

JASON KU: Mhm. A constant number of rotations, and the deletion could take a logarithmic number of rotations. Now, that's not something you need to know. It's not something I'm proving here to you. Just something that's interesting. There are rebalancing schemes, like in CRS. They introduce a red-black tree to introduce balance. And those trees actually have a weaker bound on-- it's not as tightly balanced as an AVL tree is. It allows higher than skew 2. And because it's kind of a weaker restriction, they get away with only doing a constant number of rotations-- that they can afford that before they fix the tree. But a little more complicated.

AUDIENCE: [INAUDIBLE]

JASON KU: Very nice. OK. Any questions on this? OK, so now, this is more of a mechanical question you'll get on your problem sets. And now we get more on to the theory-type questions. These are going to be a reduction-type questions. OK. This first problem, Fick Nury. This is-- anyone? Nick Fury, right? So it's an *Avengers* reference. So basically, what happens in this thing, he's got a list of superheroes that each have an opinion on whether they should go fight Sanos. And their opinion can be strongly positive or strongly negative.

And so what Fick is trying to do is find, from among his Revengers, what the log n most extreme-opinion Revengers are so that he can talk to them. He doesn't want to talk to everybody. He wants to talk to a logarithmic number of them. OK. It's kind of-- whatever. Basically, we have a classified situation where you're given, as a read-only input data store of these things in an array. And I want to find the log n ones with the strongest opinions. Does that make sense? And I want to do it-- and the first problem is in linear time.

You actually don't know how to do this yet. You'll know how to do it with material that you cover in-- well, they teach you one way to do it in 046, but we're not going to get you there right now. We'll teach you another way to do it on Tuesday, which is via binary heaps. Binary heaps are an interesting thing. It implements a subset of the set interface. Really, it just-- you can build on some iterable x. I collect a bunch of things.

These items have keys. It's a key data structure in the same way. It's implementing what we call a priority queue interface. I can build these things. I can insert things, but I'm not going to do that here. All I really need here, for this situation, is a delete_superlative kind of operation-- in this case, probably max. Delete_max. So this is like-- I've got a data structure, I'm calling these things. Does that makes sense? Yeah?

AUDIENCE: What's a priority queue?

JASON KU: Yes. A priority queue is essentially something that implements these two things. Actually, there's a third one where I can insert a new thing, but I'm not going to need that right now. So that's what a priority queue is. And actually, a set-- this is a subset of the set interface. Right? The nice thing about a heap-- which, I won't to show you how it's done, but what a heap can do-- if I had both of these operations implemented using a set AVL tree, how long would these things take me?

How long does it take to build a set AVL tree? n log n, right? Because essentially, I'm getting a sorted order out of this thing if I'm inserting these things one at a time-- or I could sort them, and then put them in a tree in linear time, like you saw a couple of days ago in recitation. But I have to sort them at some point, right? I'm kind of-- I need to take at least n log n time, because if I'm going to be able to return their traversal order in linear time, and I have this lower bound of n log n on sorting, I kind of needed to spend n log n time here, right? And how long would this delete_max take?

AUDIENCE: It's sorted, so log n.

JASON KU: Log n, right? So it's a set AVL tree. Where is my max? It's the right-most thing. I can just walk down the thing, take it off. Maybe I have to re-balance. But that's a log n operation. It's the same as insert-last in my subtree. For a set AVL tree, this is n log n. This is log n. Now, there's another data structure that does better for one of these operations. And the same for the other one that we've learned earlier. Anyone remember? Set AVL tree didn't actually give us anything over a sorted array in a dynamic array.

What that did was, we have a-- we could sort it in n log n time using merge sort or something like that. And then we could just pop off the last one n times. That would be an amortized-- I mean, if I didn't care about taking up that size, I could do it, in worst case, constant time. I just read off the first-- the last one. I don't need to resize the array, ever. I can just ignore that. Does that makes sense? OK. But that's-- OK. If I had a data structure that implemented these two operations, can someone tell me an algorithm to generate fixed lists-- don't worry about running time right now-- but that just uses these two operations? Yeah?

AUDIENCE: So we build this data structure. It's ordered from least to greatest toward absolute value of x.

JASON KU: Don't worry about where things are ordered or anything like that. I don't tell you how these things are implemented, right? All I'm saying is, I can accept a bunch of these things, and I can remove the maximum and return it. OK?

AUDIENCE: I think we just build it-- make sure that you build it such that the opinion levels are the absolute value of the opinion levels, not--

JASON KU: Sure. OK. So that's a nice thing. What I'm going to do, as your colleague is saying, is, I'm going to look through all of the things in my input. I'm going to copy it over to some writable memory store. That read-only thing is not relevant to this part of the problem. What I'm going to do is-- right. Sorry. I'm thinking about your problem set that we're writing. I'm mixing it up. OK, so we copy it over to our new linear-sized array. But instead of putting their values there, I'm going to put the absolute values of their values. Does that make sense? I just check if it's negative. If it is, I put the positive thing there. OK? And then I stick that array in this build. I put that there.

That will take some-- whatever this build time is. And then I can delete max k times. Or I can delete max some number of times, however many things that I need. Right? If I want log n highest things, I can just do that log n times, right? So for this-- if I had such a data structure, I could do this in one run of this operation and log n runs of this operation. Does that make sense? I could solve this problem reducing to this data structure. Now, for a sorted array or a set AVL tree, this operation kind of kills me already. It takes n log n time. The nice thing about a binary heap is, it does this operation in linear time. You will see that on Tuesday. And it does this operation in log n time.

What's the running time if I use a binary heap to implement this data structure? Order of n times order log n times log n. How big is log n squared-- log squared n-- compared to n? It's smaller, right? So if I add those two running times together, it's still linear. That's how you solve the first problem. I didn't have to tell you what a binary heap was or how it did what it did. All I needed to tell you is that it did this operation in linear time and it did this operation log n time. All right. The magic will be shown to you on Tuesday.

Part B says, now, suppose Fick's computer is only allowed to write to, at most, log n space. Well, OK. That's a problem here. Because before, we copied over the entire array, filtered it out, and then did some operations. But we couldn't even afford this if we couldn't store the whole thing externally in writable memory. So we can't do that. So in some sense, this is a more restrictive environment. I can do less things. It's less powerful than my previous situation, where I had as much space as I wanted to use. So it kind of makes sense that I, maybe, couldn't get the running time bound that we had before. Maybe I have to sacrifice something because I'm in a more restricted computational setting.

Now, this is something you could solve with binary heaps, but you don't have to. You can solve it with set AVL trees. Does anyone have an idea of how you could solve this using a set AVL tree? I'm limited by my number of-- my space is, at most, log n.

AUDIENCE: So how much space does a set AVL tree take?

JASON KU: Right. Space-- there's constant number of pointers for each one of these nodes. And I'm storing in notes and space. Basically every data structure we've shown you takes space-- the order of the things that we're storing. It's not using additional space. It might take more time to do certain things, but the space takes the number of items that we're storing plus, maybe, a constant factor. So I'm going to draw my input here, which I can only read-- I can't write. Do I give it a-- I'm just going to call it A. So this is my list of all the Revenger opinions. I can only read it. But my computer can only write to this logarithmic amount of space. What can I put in that space?

AUDIENCE: The log [INAUDIBLE]?

JASON KU: Well, I can certainly put log n things in there. So if I'm given that restriction, I probably want to build a data structure of that size, containing that number of things. Does that make sense? Because what else are you going to do? So I gave you an idea. Maybe we could use a set AVL here. I see a logarithm in my answer. It's very possible that we might have sorted arrays or set AVL things. Those things give me a log somewhere in my running times, right? So kind of makes sense that I might have, maybe, a set AVL tree here. Why would a set AVL tree be helpful for me? Yeah?

AUDIENCE: Because it's sorted and you don't have the traversal order, you can calculate the traversal order and insert [INAUDIBLE]?

JASON KU: Sure. I can do all of those things. But in particular, it's going to help me be able to find a large one quickly, right? If I have a set of things, it's going to be-- and I'm maintaining this data structure by adding things incrementally to it, I can find out what the biggest one is-- or the smallest one-- pretty fast in log n time. So if I have log n things in a tree here, what's the height of that thing?

AUDIENCE: [INAUDIBLE]

JASON KU: Log log n. That looks familiar. So what can I afford? I can afford a linear number of opt set AVL tree operations on this data structure. OK, you had a question?

AUDIENCE: [INAUDIBLE]

JASON KU: OK, I'm sorry. Yeah?

AUDIENCE: For that to be an AVL tree, does it have to be a BTS tree?

JASON KU: Uh, BTS-- BSTs. So when I talk about-- someone likes Korean K-Pop. OK. So BST-- but in natural-- kind of in the lingo that you're probably used to hearing in other contexts, what we mean, in this class, is a set AVL tree. Now, sometimes, what people refer to as a binary search tree doesn't have balance semantics-- so what we might call, in this class, a set binary tree. But really, they're useful because they're balanced. So we're going to usually just assume that we're talking about balanced things here.

Now, a set AVL tree has these binary search tree semantics where the keys are ordered. These items have keys and they're ordered. It's a set interface. Whereas we also presented to you a sequence interface, for which these things don't even have keys. How could I store set semantics there? So that's the distinction that we mean when we say binary search tree versus, really, a set AVL tree versus [INAUDIBLE]. Yeah?

AUDIENCE: So if we look at it to make an AVL tree out of this, would that mean that, when we make a node, we tell it, we are keying on the absolute value of [INAUDIBLE]?

JASON KU: OK, when you're making a set AVL tree, you've got to tell us what-- if you're storing objects, you've got to tell me what their key is. You're just storing some numbers, like what I'm doing here. Now, this isn't a set AVL tree. But if I'm just storing numbers, I have to tell you the items that I'm storing are the keys. And then everything follows. But if you've got an object that you're trying to sort, like the students in this room, you've got a lot of properties.

I want all of the people with phone number-- maybe I want to key you on phone number for some reason. That's going to help me find out where you live? I don't-- this is getting a little-- I don't want to go there. But if I give you a set AVL tree, I've got to tell you what it's keyed on. If I give you a sequence AVL tree, it's obvious what my traversal order is going to be because I'm giving you a sequence. That's what the input was. Does that make sense? All right, so I've got this set AVL tree of size log n. What should it be keyed by?

AUDIENCE: Absolute value.

JASON KU: The absolute value of their preference-- or of their opinion. I don't remember what this is called. But what log n things do I put in here? I don't know. I don't know anything about these things. What makes one better than another? Let's just put the first log n things. Does that make sense? All right. What could that tell me? Now, I've put this thing in it. How long did that take?

AUDIENCE: [INAUDIBLE]

JASON KU: Log n times log log n time, right? But that's much less than our running time that we're looking for, so I don't really care. I mean, I want you to say how long it took, but for my purposes, I know that it's lower than the running time I'm looking for. And I did that operation once. I don't really care about it anymore. Yeah?

AUDIENCE: How did you get the log n times log log n?

JASON KU: Because the number of things I'm storing in this thing is log n. And so if I pattern match the build time of an AVL tree and I stick log n in there, then it's log n times log n. OK.

AUDIENCE: So that's just for one iteration?

JASON KU: Well, right now, I've just built this thing. Maybe-- I just built it once. I'm asserting, too, that maybe I don't need to build it again. What could I-- so, by now, I know-- I haven't filtered my data at all. I'm just storing these things in sorted order in some way. What can I do to maybe start processing the rest of the data? Yeah?

AUDIENCE: [INAUDIBLE] try to scroll through the list A and try to find someone that's bigger than-- try to keep the maximum [INAUDIBLE].

JASON KU: Sweep this guy over inserting things, and always maintaining-- if I do that, and I keep sticking things n I'll have this sorted of thing at the end. And now I can just read off the biggest k things. However, as I'm inserting things across here, my thing's growing.

AUDIENCE: Well, just delete the smallest one.

JASON KU: Oh, delete small stuff. I like that idea.

AUDIENCE: So basically replace it.

JASON KU: Yeah, basically replace it. Right. What I'm going to do-- here's a proposal. We're going to take the next guy, stick it in. Awesome. Which one don't I care about now? The smallest one there. So kick the smallest one out. Now, this one that I stuck in may. Be the smallest. So I just kind of passed it through this thing, but how long did that take me? It took me the height of this tree. What's the height of this tree?

AUDIENCE: [INAUDIBLE]

JASON KU: Log log n. So I put one in, I popped one out. That's the smallest, right? And I keep doing that all the way down the thing. How long did that take me?

JASON KU: [INAUDIBLE]

JASON KU: Yeah. Processing n minus log n things, which is basically n. And each one of those operations took me height of the tree time. That gives me the running time that we're looking for, n log log n. sense?

AUDIENCE: Is this reminiscent of a sliding window technique?

JASON KU: Yeah. It's kind of a sliding window technique. You may have been using one recently. OK. Everyone OK with this? Yeah?

AUDIENCE: Could you just remind me of the context that we're talking about is log log n, like tree and where--

JASON KU: So this thing-- the size of this thing is log log n?

AUDIENCE: Yeah.

JASON KU: I mean-- sorry, log n. And the height of this thing is a log of the size.

AUDIENCE: I'm sorry, in relation to our little log n size [INAUDIBLE] small log log n trees, or--

JASON KU: No, so-- sorry. I'm taking this stuff-- there's no intermediate data structure here-- I'm just sticking all of these things into set AVL. Yeah?

AUDIENCE: [INAUDIBLE]

JASON KU: Into one set AVL of size log n. I'm sticking a guy in, popping the worst guy out. Going through all of the things. I need to make sure, when I'm sticking it in, I'm keeping track of which Revenger it is, and that I'm taking the absolute value, and all of those nitty gritty kind of things. But that's the basic idea. I'm just taking this, I'm sliding the window in, putting something in, taking something out that may or may not-- probably is not-- the same thing.

And at the end of this procedure, the invariant I'm maintaining here is that my thing always has the k largest opinions of the ones that I've processed so far. That's obviously true at the beginning, when I build this thing. And when I get to the end, I've processed all of the things, and this has size log n, and so I have the log n largest-- highest-- extremist opinions. And then I can just do an in-order traversal of this thing and return. Does that make sense? And I've only used logarithmic space. Yeah?

AUDIENCE: Wait, I don't get it. Are all of the opinions in that AVL tree?

JASON KU: Are all of the opinions in that AVL tree? All of these opinions are in the AVL tree. And at some point, I will insert every opinion into this AVL tree. But I'll be removing the ones that I don't care about as I go. Does that makes sense? I'm always maintaining the invariants that this thing, before I insert something, has exactly log n items in it, and then I'm maintaining that invariant by sticking one in, taking one out.

AUDIENCE: Oh, OK, so then which one are you deleting?

JASON KU: Always the min, because I'm wanting the largest ones.

AUDIENCE: And the min of the absolute value.

JASON KU: Yeah. I'm keying by the absolute value of these opinions. Yeah?

AUDIENCE: [INAUDIBLE]?

JASON KU: Total runtime here? It's bookkeeping. It took me log n times log log n time to build this data structure at the beginning, plus n times log log n. I did, basically, n operation-- asymptotically, n operations. This way, it's actually n minus log n operations. And each one of those tree operations-- doing one insert, one delete-- each one of those took the height of the tree time. And so this is that. Good? Yeah?

AUDIENCE: If, instead of one, we just insert and delete, can you do a comparison and then--

JASON KU: The inserting and deleting a set AVL tree is actually doing comparisons within its data structures.

AUDIENCE: Just compare with the min. And then--

JASON KU: Sure, you could do that. I could do it the other way. I could remove the smallest element here, to start with, right? And then I compare it with this guy, and then whichever is bigger, I stick it back in. Same thing. It's just, am I doing the delete first and then the insertion, or am I doing the insertion first and then the deletion? Any other questions? Lots of questions. All right. Well, I'm probably going to have to skip a problem. We're going to move on to CS-- no, SCLR. What's the reference here?

AUDIENCE: [INAUDIBLE]

JASON KU: Yeah, CLRS. These are four academics who wrote a popular textbook in computer science. This is the same kind of k kind of thing. They found first editions and they want to auction them off. People can go on to their website. They have a bidder ID. It's a unique identifier. And they can place a bid for one of these books. And they can change it during the bidding period, but at the end of the bidding period, the academics want to know who the-- what is the expected revenue I'll get by selling to the k highest bidders? Does that makes sense? Yeah? OK.

Note that, before I build this data structure, I know what k is. k is a fixed thing. Because my running time of this get-revenue depends on this k, it's not an input to that operation. So k is kind of I don't know what it is, a priori. It could be n over 2. It could be log n. It could be 1. But the data structure I build needs to satisfy these running time properties, no matter what choice of k that the academics told me. Does that make sense?

What I need to do is, as far as time is going on, people are placing new bids and updating their bids. And those updates can take log n time. But as soon as I close the window, I want to be able to tell, in constant time, what the k highest bidders are. Any ideas on how to do this? What are the operations that I have to do? I have to be able to place a new bid. Associated with a bidder is an idea and a bid, which is also an integer-- how many dollars I'm going to pay for this book. Update the bid.

In some sense, I need to find whether that person placed the bid before, in my data structure. So at some point, I'm going to need a find on bidder ID. Does that seem possible? So I might want to have some kind of dictionary on bidder IDs. When I say that I want to have a dictionary on something, right I'm not specifying to you yet how I'm going to implement that dictionary. What are my usual options? A hash table. But what if I need worst-case time?

A set AVL tree, right? That's going to be your go-to for a dictionary, because that's going to give me log n time to find things via a key. It's the only thing-- well, except for a sorted-- you could also use a sorted array, but that's going to not be dynamic. And here, we're updating who's in my data structure all the time. People are going in and placing bids-- new people placing bids. So my set of things that I care about is changing all of the time. So that's probably going to steer me away from sorted arrays, because they're not good with dynamic operations.

So I'm going to need some kind of dictionary on bidder IDs, but I'm also going to need to maintain the sum of the k highest bidders. Does that makes sense? And so, in some sense, I need to keep track of an ordered notion of the bidders, the bids, that are in my data structure. Does that make sense? So order is going to be important on the bids. I'm going to need a look-up on bidder ID. And that's about it, right? OK. Yeah?

AUDIENCE: Just checking. So if something that's worst-case runs at worst-case time, runs expected [INAUDIBLE].

JASON KU: Yes. Correct. Yeah. That's a very good observation. If it runs in worst case time, it also runs in expected at time, right? Because there's essentially no randomization that I'm talking about here.

AUDIENCE: [INAUDIBLE]?

JASON KU: Yeah. And so there's a stronger notion which we want you to specify, which is that, actually, there is no randomization here. We're not using a hash table. In this class, really, that's the only situation where that's going to be an issue. But if it is, what this problem is saying for each [INAUDIBLE], whether your running time is worst case expected and/or amortized, what we're really trying to get you to say is what's the-- evaluate the running time of your algorithm with the proper qualifications.

If it took worst case, I want you to say that it took worst case. If it took-- if you used a hash table, I want you to say expected. And if these operations were sometimes really bad, but on average, they're really good-- if I did a lot of them, that's amortized. Or if I reduced to using a dynamic array, or if I reduced to using a hash table, those dynamic operations would still be amortized. OK.

The dynamic ones. The nice thing about linked data structures is that dynamic operations aren't amortized. So we're going to be able to get one. Now, for this problem, we can actually get worst case bounds, so we're going to try for that. You can also do it in expected using some hash tables for that dictionary. When you approach a data structures problem in this class, you want to tell me what it is you're storing, first off.

Tell me what's supposed to be in those things-- some invariants on this data structure to make sure that, when I do queries later, that these things are being maintained, so that if I'm maintaining a sorted array, and I'm supporting an operation to find the maximum, I had better-- anything I do to this data structure had better be maintaining the invariants that these things are in sorted order and the last thing has the maximum item. Because my max-return thing is going to look there and return that. Does that makes sense?

So you want to tell me what is being stored at a generic point in time during your data structure, what is being maintained-- so that, when I support the dynamic operation or a query, in a dynamic operation, where I'm inserting and leading things from this thing, I need to make sure that I'm maintaining those invariants. And when I'm querying, I can actually rely on those invariants to answer my query. Does that makes sense?

So, for this problem-- this is 4-3. Any ideas? I have two kind of keys that I might have to deal with. One's a bid ID and one's a bid, right? So how could I, if I have two keys that I might want to, maybe, order on one and look up on another, how many data structures do you think I'm going to use?

AUDIENCE: Two.

JASON KU: Two. That's a pretty good guess. So one of them-- let's just guess, right? I need to be able to look up on bid, so let's store these bidders in some kind of dictionary that's going to be able to look up those things fast. So two data structures. One is a dictionary keyed on bidder ID. What else am I going to want? What's up? The other way around, a dictionary stored on the bids? Is a dictionary what I want here?

AUDIENCE: [INAUDIBLE] set up AVL tree [INAUDIBLE]?

JASON KU: I want to maintain order somehow. Because I want to maintain the biggest things that I've seen so far. Right now, if I have-- at some point in time, what's going to happen? If I'm maintaining the k largest at any point in time, it's possible that one of those bidders maybe decreases his bid so it's no longer in the highest. I'm going to also need to keep track of the other guys to see who I should add back into that set, for example. Here's an idea. I'm going to keep not just one other data structure, but two other data structures. Maybe this is a leap. You don't have to do this. There's a way to do it with just one other.

But I'm going to store two more. One is kind of an-- a data structure to store bidders with a-- store the k highest bidders, and a data structure to store the n minus k highest bidders. Does that make sense? This separates my problem quite nicely, right?

Every time someone does an interaction with this data structure, I can check to see whether it's bigger than the smallest thing in here. If it is, I can do the same kind of trick I did before. I can remove it and stick my new one in there. And we're going-- but I removed it. I have to maintain this property. So I stick it in here.

There's another case. What's the other case? It's smaller. In which case, I don't do anything to this data structure, and I just stick it into here. Does that makes sense? What are the operations these data structures need to maintain? Finding the minimum or the maximum of these two sets. Does that make sense? Actually, really, the-- where are those operations? I don't have them anymore. But they were the priority queue operations. They had delete_max-- and also insert-- were things that it did well on. So any priority queue, anything that can deal with maxes and mins, is good.

And what's a data structure you know that can deal with maxes and mins pretty efficiently? The set AVL, right? So instead of data structure here, I'm going to say, set AVL. And obviously, it's going to be cheered by bid. Because that's the thing that I'm going to want to find maxes and mins over. Everyone following the logic here of why I'm maintaining these things? This is the level of an invariants that I want to maintain, because when I go to, for example, do this query, get-revenue, I can just run through and sum all of these things.

Oh, wait. How much time do I have? Do I have k time? No, I don't have k time. So I don't-- I can't afford to sum up all of these things at the end of my thread. I have to return it to you in constant time. Any ideas? Yeah, just compute-- update a sum. Along with this data structure, I'm going to keep a fourth thing, which is just total of their bids. I'm going to call it t. And that's something I'm maintaining. It's part of my data structure. You can think of it as, I'm augmenting this thing with a number.

And the point of augmenting this thing with a number is that I can just-- if I need to know what the total of this stuff is, I can just look at that number. Does that make sense? All right. So now, I think, we're almost done. We're basically done, right? How do we do this? Someone walk through to me how I would get revenue with this data structure. I basically kind of told you. Look at this number. Return it. Because that's the invariant that I've maintained on my data structure. I'm relying on this invariant. Now, I'd better make sure this is good when I do dynamic operations. I make sure I maintain it.

But if I, by induction, I ensure that all of this stuff is good, and when I do a dynamic operation, all of that stuff is maintained, then I'm all good. So get-revenue, after I did all this extra work, is very easy. I just look at this number and return then. When we're grading a data structures problem, usually we give you some points, first, for setting up your data structure, separately from the operations, and then we give you points per operation that you successfully deal with, and then some points for correctness and running time. Yeah, you had a question?

AUDIENCE: So would total be a thing that we update whenever we mess around with the highest bidder tree and then n minus k bidder tree?

JASON KU: I'm sorry, say that again?

AUDIENCE: Are we treating the total to an augmentation that we update every time we do something?

JASON KU: Yeah. Yeah. So it's just one number. It's not really a data structure, it's just one number that I'm storing with my database. All right. How do I implement a new bid operation? Yeah?

AUDIENCE: I have a question. Can we assume that the bids will also be unique?

JASON KU: Can you assume that the bids may be unique? No. That's actually something that is a really useful observation. We've been talking about set data structures as requiring unique keys. How can I deal with non-unique keys? It actually turns out that, hash table, it's really important that these be unique keys. Because I need to check whether it's in there. I'm looking for that single key. When I find it, I have to return. If I had multiple things with that key, I might not return the one that I'm looking for. Doesn't even make sense. But you can generalize the set infrastructure to deal with multi-sets.

How can I do that? Well, with each key-- again, I'm storing unique keys. With each key, I can link it to a sequence set of structure, or any other data structure. And what I'm going to do is, I'm going to do-- anything that has that key, I'm going to stick it in that data structure. So instead of storing one item there, I have the possibility of storing many things there. Now, I have to change the semantics here.

If I'm saying, find on this key, well, now, I could say, I'm going to return all of the things with that key, or I'm going to store some thing with that key. But you get the idea. All I have to do is map it to some other data structure to maintain [INAUDIBLE]. Like, maybe, I want all of the things with that key. I want to find the one with this other key. So maybe I link to a set data structure that can search on other things, right? But the idea here is, we maintain this uniqueness key property. I have to relax my semantics so that I'm storing multiple things at that key location. Does that makes sense? Yeah?

AUDIENCE: Why does it matter whether the set AVL tree has unique keys or not?

JASON KU: It's going to matter here because I have bids. And the bids could be non-unique. Two people could have the same bid. And by our definition of a set data structure, it had to have unique keys. So if I stuck in all of these things keyed by bidder, you've got a problem. Now, in actuality, we can get away with that by storing, basically, a linked list of all of the things with that key, and we would be fine. And then, whenever I want to return one, I could just do it. But actually, a binary tree actually is flexible enough that, in most implementations, you can just store a bunch of those things. But, actually, our run times do worse.

What does it mean to find-next in my sequence? What does it mean to return the next larger thing above this key? Doesn't really make sense, because there could be multiple ones. Which one do I return? And if I repeatedly do find-next on this data structure, I might not loop through all of the things. So some stuff breaks down in our interface. So I would prefer you use unique keys in this kind of situation. Next Tuesday, I think, with binary heaps, we'll deal with non-unique keys. That's fine. But if you're going to use non-unique keys in here, you've just got to be a little bit careful about the semantics. Yeah?

AUDIENCE: [INAUDIBLE]?

JASON KU: You would get the same running time-- you have to change the semantics on what you mean by "find something." I just want to return anything with this key.

AUDIENCE: What if everything has the same key. Then--

JASON KU: Then it takes constant time. I just return the first thing. I mean, these are special cases that you have to think about, right? I don't like thinking about them. So I just like having unique keys. And if I want a situation where I have non-unique keys, I'm going to basically put collisions at that key into a new data structure. It's just easier for me to separate out in my head on what's happening. Because, all of the running times that we proposed, there's very strong definitions for unique key. When you're dealing with a multi-set, it's a little bit more prevalent. Any other questions? We really need to kind of move on here, right?

Dictionary keyed on bidder. We still haven't implemented any dynamic operations. New bid. What do I do? What am I going to need for my update? I'm going to be able to need to essentially find, in each of these data structures, where that bidder is. And if I just have a thing keyed on their bid, the interface doesn't tell me what their old bid was. It just tells me what their bidder ID is.

So if I just had their bidder ID and their new bid, how the heck am I going to find out which of these data-- where in these data structures they are? What I can do is, I can store, in this dictionary-- which I can look up in some amount of time-- a pointer to where it exists in these things. Does that make sense? This is called cross-linking. You may have done that a little bit in problem set 2, or something like that. Yeah?

AUDIENCE: Restoring a pointer to a specific bidder?

JASON KU: Yeah, exactly. The invariant we have is that all of the bidders we've processed so far exist in these data structures-- in one of these data structures-- because we've used a set AVL tree. In particular, it exists in a node of one of these data structures. What we can do is, in this thing, maintain pointers mapping each of the bidder IDs to their location in these data structures. And why is that going to be a useful thing?

Say I map this dictionary. What could I use for this dictionary to get the running time we're looking for? I could use a hash table or a set AVL. If it's a set AVL, I'm going to get logarithmic time, worst case. With a hash table, I'm getting constant time, but it's expected. So it could be linear time in the worst case. We're going to use a set AVL tree, because that's what we do right now. And that's going to give us worst case. What I'm going to do is, for each one of these things, I'm going to store that pointer.

What I'm going to do is, first, I'm going to do that operation we had. If I'm adding a new bidder, I'm going to take the D and B-- these two values, that object, that bidder object, or whatever-- I'm going to look at the smallest thing in this data structure, see if its bid is bigger than the thing I'm inserting. If it is, then I'm not going to touch this data structure. I'm just going to insert it in here. And now, after I insert in here, I know exactly where it is in the data structure. I just inserted it. So now, holding that in my hand-- the node-- I can go and insert that bidder into here by bid ID. So it's going to take logarithm time.

And now I can store, with that node, my pointer to this data structure. Does that makes sense? And in the other case, I kind of do the same thing. If it's bigger than the smallest thing here, I pop that smaller thing out, stick it in there, and I stick my new guy in here, cross-linking each of those pointers along the way. Does that make sense, hopefully? Kind of? Kind of? OK. And for update, very similar. If I want to update a certain bidder, I look in this data structure, find the bidder, traverse that pointer to wherever it is in one of these AVL trees, right?

If it's in this one, I just remove it from the tree, or I remove it from the tree and then I re-insert with whatever the new bid is. And if it's in this one, again, I remove it from the tree, re-insert in whichever of these things is, and then I might have to swap a constant number of things back and forth here to maintain that this has the k highest. And when I do those dynamic operations, I'm always removing some constant number of nodes in each of these trees and adding back in a constant number of things. And while I do that, I just make sure to update this total as I go. This total was the sum of all of the bids in here. And if I insert a new bid in here, I have to add to that total. And if I remove one, I have to remove from that total.

But again, it's a constant number of things I'm moving in and out of these data structures, and so it can update this in constant time. Does that makes sense? Now, the lookup here, and the insertion and deletion in here, those each took logarithmic time, worst case. But I did a constant number of them. So again, logarithm time. Does that makes sense? That's, essentially, this problem.

It's difficult, right? There's a lot of moving parts here. But if you just break it up and to describe to me-- like, you really do a good job on this part, describe well to me what your data structure has, then those descriptions of those algorithms can be pretty brief, actually. In this one, you tell me these three data structures, you tell me this guy's mapping to its location and these things, I'm maintaining this guy, and then you just maintain those things with dynamic operations and then use those things for query operations. Does that make sense? Wow, we have 10 more minutes? I'm going to briefly do 4-4 for you.

Receiver roster. We've got a coach. She's got a bunch of football players-- receivers. And wanting to start on her team, some number of players that have the highest performance. And by performance, we mean the average number of points they've played in games that they have logged in their system. But actually, their data is incomplete. They don't know which games, and how much they scored, and all of these things. There could be errors. And so these interns, they're constantly updating this database with queries like, oh, never mind, this person didn't play in this game, or actually, they did, and they scored this number of points. That's the-- clear and record things.

And then, at some point in time, like when we want to play a game, I want to be able to return the jersey with the kth highest performance in log n time. This is kind of a rank query, right? The kth highest. Now, in actuality, I might want to return all k highest players so that that might be my roster. But this is a more generalized query. It's more specific, more-- it's not really comparable. But you get an idea for why that might be useful to the coach. I don't know. Maybe not.

So what's the idea here? We have a lot of different things floating around. We've got games. They have IDs-- unique IDs. We've got receivers. They have unique IDs. And each receiver could play in many games. Oh, that's kind of worrisome. And many receivers could play in the same game. These kind of many-to-one mappings are a little confusing. And then we've got each player-- receiver-- having a certain number of points per game. And we're trying to sort them, kind of, by their performance, which is a rational number. Ugh, right? Which has to do with the number of games they've played and the total number of points.

Now, I see rational number, I can't compute that. That's what we're talking about last problem session, right? But what I can do is, I could store the total number of games they played and the total number of points they have. And you can imagine, by augmentation similar to this, every time I add a game, one of these small operations, I can update that information for each player. If one of these dynamic operations is affecting only one receiver, I can update whatever it is in constant time, probably, if I just store with the player what their total number of games is as recorded by the database and how many points they've scored.

Then, if I have a data structure that needs to sort the receivers by their performance so I might be able to find the kth one-- the kth largest-- then I can't compute that performance. But what can I do? I can compare two players based on their performance using cross-multiplication. Because I have the numerator and denominator of each of these rationals and I can cross-multiply and figure out whether one's bigger or smaller. And as long as I have a comparitor, I can do set AVL stuff. Does that makes sense? OK.

I'm just going to outline kind of the components of this data structure. Well, first off, I'm going to need to record a receiver. And a receiver could have a lot of games. But the important-- this is kind of a receiver-centric kind of problem. Does that makes sense to you guys? I'm not ever wanting to filter on all of the receivers playing a game. I'm never removing a game from the system, I'm removing a receiver from ever playing in a specific game. Does that makes sense?

So if I'm storing a receiver, and each receiver has some games associated with them, kind of makes sense I might want to have a nested data structure where with-- maybe I have a dictionary on receivers. And for each one, I store all the games that they've played in some other data structure. With each receiver, I store another-- its own data structure containing all of its games. Does that makes sense?

OK, so that's the idea. We have some kind of-- I need to be able to look up receivers, because I'm clearing them or I'm recording them. So I'm going to have a dictionary or-- here, I'm looking for worst-case log n time. So I'm going to skip the dictionary abstraction and go straight for the set AVL. AVL keyed on receivers. I before E, except after C. It is I-- E-I? That rule never works. OK. Set AVL tree on receivers, and each one of those nodes with each one of those receivers, I'm going to store-- for each, store a set AVL on games.

Why do I store a set AVL on games? Why don't I just store a list of all of the games? Because if I want to remove this game from a receiver, I need to do that in log n time. And here, what we're saying is that n is the number of games, but that the number of receivers on the team is always less than the number of games. If I search in this AVL tree and I search in its AVL tree, I can be assured that those two searches was only log n time. Because I need to remove game, right? So there you go.

Then what am I doing? I'm returning the kth highest performance. Well, I need-- with each one of these guys, I also store-- what was this augmentation? The sum of the points stored in these games. Sum of points and-- what was it-- number games. Because if I store both of those things in constant time, I'm going to be able to compute their performance, where I'm going to be able to have the data I need to compare performances.

AUDIENCE: [INAUDIBLE]?

JASON KU: Yeah, it is. Just numbers. These are data structures. This is a data structure. These are just numbers. And I'm storing that with each receiver. But that's not going to help me find the kth highest player. None of these things are sorted by performance. So I need a last data structure. Five, I need to store something dynamically sorted by performance.

AUDIENCE: Set AVL?

JASON KU: Set AVL, yeah. Set AVL storing receivers keyed on performance. Now, when I say keyed on performance, you want to mention something about the cross-pot multiplication. Like, I'm storing, with each one of these things, this augmentation, and when I'm comparing two things, I'm using cross-multiplication. But other than that, then we can abstract it away, right? We've abstracted that function call. And I can imagine comparing two keys. I can do this. This is a theory thing. I'm not asking you to implement that. But that's sufficient for me, as a reader of your solution, to be able to say, yeah, you know what you're talking about. All right.

How do I connect these things? The thing is, I'm going to need to be-- I need to update these things when I insert or remove a game. So how do I know where are these receivers are in this thing? I store a pointer into this data structure, right? So up here, I store a pointer to where it is in the data structure. Again, I'm storing all of the receivers. This has the same size as the number 1 data structure up there-- has the same number of receivers.

But we're not quite done yet, because I'm not wanting to know who has the best performance. I want to know who has the kth best performance. Ugh. How do I find the kth best thing in this tree? I've got a tree. Set AVL tree. It's mapped on performance. I know where the last one is. But if I want to find the kth one from the end, how do I do that? It's an AVL tree-- a set AVL tree. All I'm storing is heights. Is there an operation that you've thought about?

AUDIENCE: [INAUDIBLE] you're not storing the size of each.

JASON KU: A set AVL tree, by default, does not store sizes, right? That's what a sequence does. But you think maybe that would be helpful in this situation? Yeah. So, actually, if I decided to augment by sizes also, I could do the exact same kind of sequence find_at operation, and I could be able to look up the n minus kth item in here using the exact same function for subtree at that I had in the sequence AVL tree stuff.

Actually, in CLRS, they don't even bother with sequence AVL trees. They go straight to, if I wanted this rank-find functionality on a sorted order of things, then I could augment the subtree sizes. But it's actually a much more useful general property, so we decided to present it to you in the context of sequence AVL trees, because then I can just basically reduce to it when I get to here. So that's kind of a structure of a data structure that work on this problem. I leave it to you as an exercise to implement all of these operations for yourself, or take a look at the solutions.

The last one is going to be put online-- the solution. It's pretty complicated. It's what's called-- you can think of the size augmentation finding-rank as a one-sided range query. It's basically, how many things are to the right of this value? What the last problem does is walks you through a two-sided range query, where I want to know how many nodes are between these two values. So it's a walkthrough. All right. Thanks, guys.

AUDIENCE: Thank you.