[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: All right. Welcome, everybody. Everyone ready for the quiz?

AUDIENCE: No.

JASON KU: Quiz next week. Yes. I hope you all are here because you know that there's a quiz next week. OK. So what is this quiz about? It's about what we've talked about so far in this class of course. What is this class about? Someone remember from my first lecture? What is this class about? What are we trying to test you in this class?

AUDIENCE: Algorithims.

JASON KU: Algorithms. Great. Also data structures. Right? That's what the first part of this is. But really it's to get you to solve computational problems. That's the first thing. Be able to argue to someone else that you actually did solve it. Right? It's correct. Right? That you chose something that's better than other things, that it's efficient. Right? And that you can communicate those things to other people. Right?

Those are my big four that I try to get you guys to internalize. And so that's what our quizzes are going to try to evaluate you on. OK. And so aside from some nitty gritty kind of stuff that we do at the beginning of the term like talking about our model of computation. Right? Our model-- and asymptotics. Asym-- sym-- totics. Is that right? Recurrences right?

Aside from these kind of basics, we delve straight into algorithms. Right? These are kind of like almost definitions. We don't rely on these things very much. I mean, we rely on these things all the time. But it's kind of the mathematics that we use to talk about things. Right? How can we even say how long this stuff takes unless we can reason-- we can abstract a way that this stuff is not on a real computer.

This is kind of in our minds, in a computer. And we're reasoning about these things based on the number of constant time operations this magical computer might have, which is a pretty good representation of any of the computers you have for certain assumptions. Right?

We're not-- I mean, there's not a lot of problem set questions we had you talk about. These things-- specifically-- usually they were part of some other problem. Right? You had to describe the running time of this thing. And you might have had to solve a recurrence. And you might have uses master theorem. That kind of thing. Right? Or you use asymptotics all the time. Or you need to remember in our model, oh, it kind of matters how big of an integer I can store and do arithmetic on in constant time. Right?

So for that p-set 3 question you had at the end of the coding question, you wanted to hash things. And you need to argue that those things fit in a constant number of words so that could be done in constant time. A lot of you guys found canonicalizations basically mapping to things that were exponentially large. So maybe multiplying a product of primes or something like that.

And that would not be a good representation. OK. So these things come up, but they're not the main focus of the problems we solve. What are the main focus of the-- how do we solve a computational problem in this class? I gave you two ways at the beginning of the term. Do you guys remember? We can solve-- how to solve a computational-- computational problem.

One's the hard way. One's the easy way.

AUDIENCE: Brute force.

JASON KU: Brute force. OK. So you're describing to me a technique for making your own algorithm. Right? I can just design my new algorithm from scratch. One way I could do that is brute force it. Right? Look at all the possible outputs and see which one works. Right? Or I could reduce to something like divide and conquer or something like that.

That's generally a hard thing to do-- to make your own algorithm. Right? That's why we don't ask you to do it a lot in this class. It's an 046 kind of thing. Right? So the first thing you could do is write design a new algorithm from scratch. Usually it's not from scratch. Right? Usually you're reducing to some kind of algorithmic design paradigm that you've maybe heard of.

You'll talk a lot more about it in 046 and at the end of this term when we talk about dynamic programming. But generally, that's a hard thing to do. Right? You're trying to think of a recursive algorithm. Right? You're trying to prove that it's correct-- all these nitty gritty-- we have actually, throughout the class in lectures, have been showing you the algorithms. But we're not really expecting you to make those algorithms.

What are we expecting you to do most of the time. Yeah? To reduce it to a problem that we showed you how to solve. Right? I'm going to say thing here, but really what I mean is an algorithm that we've taught you to-- basically here-- reduced to a known thing-- to known thing. Usually that means there's a problem or an interface that we've given you.

And in general, we've shown you multiple different ways to solve that problem or interface. Right? So we've shown you many ways on how to sort things. And we've shown you many ways on how to implement sequence and set interfaces. Remember?

And a lot of times the types of problems that we're asking you to do is to just use as a black box some of the things that we did. But you need, as a programmer, as a computer scientist, you need to tell me when I should use what went. Right? Yeah?

AUDIENCE: Could you clarify what you mean by use as a black box?

JASON KU: Use as a black box. Exactly. Right. So this is a phrase that I use and a lot of people in computer science use. It's basically you import a library into your code. Right? What do I have? I have an API. I have a way to interact with that code. I don't actually know what's going on inside of that library. I'm using it as a black box. It's opaque to me. I cannot look inside-- I can actually probably could look inside what their code is, but I'm not going to.

The thing that makes it useful to me is that has this useful API that I trust it to do the things that you told me that it was going to do. Right? And so there's kind of-- I'm going to jump around a little bit here actually, because that's a great question. So here I think of there's three different types of problems that we talk about, that we give you in this class. You might have seen this on the problem set. Right?

I like to categorize them into three different categories here. One is you have to understand the internals of a data structure, an algorithm that we know. You have to be able to look inside and, I don't know, given a node in a balanced binary search tree-- an AVL tree-- how can I do a rotation? Right? Or how do I do an insert? Or something about the structure of this thing-- a binary heap-- where are the top k things in a max binary heap, which is on your problem set.

Those things require me to very much not black box these data structures. It's a white box. Right? I need to know what's inside of that to answer that question. Right? I need to know about the internals of that data structure. Right?

And there are other types of problems where it's like, oh, I don't need to know what the internals of this data structure is. I can just operate with knowledge of the API and try to hook it in to the problem that I need. And that's what I call a reduction type problem. This is how does the core material we presented to you in lecture work? This is how do I apply that core material?

And harder than both of those things is what I might call a modification type of-- these aren't really good names. I came up with these this morning. But it's trying to get at the idea here that it's possible that you need to know what the API is, and you need to know what's going on inside to be able to answer the problem. Things like adapting a divide and conquer algorithm.

Or making-- instead of using a dynamic array that has extra space on one end, maybe I have to put extra space in the middle or something like that. Right? I'm adapting something that was from the core material. It's pretty close. But I have to modify it in some way. Augmentation right?

I have to take the vanilla said AVL tree that I might have given you and put some other property on the nodes and you need to tell me how to maintain that. How can I compute that subtree property from its children? Does that make sense? So this is the harder of the things. Right? If you can identify which one of these-- a problem that you look at on an exam-- [INAUDIBLE] under-- maybe that can help you conceptualize what should I use.

For a reduction type problem-- I'm going to talk about this in a second-- but a lot of times it's useful to reduce it to a problem or an interface rather than an algorithm or a data structure. What does that mean? If I can solve the problem by saying reducing to sorting, I can argue to you that that algorithm is correct. I just use sorting as a black box. Now it might not be efficient.

My choice of sorting algorithm that I chose matters for efficiency. But for correctness, it doesn't matter. Right? For a data structures problem I might reduce to using two set data structures and sequence data structure or something like that. But it will be correct if I reduce it to those things. I can define the operations in terms of those interfaces. I don't have to make that choice until I talk about running time. Right? Until I talk about efficiency.

And the name of the game on the quiz to get points-- we can't give you points for an incorrect algorithm or something that's pretty close to correct. Right? And we can't give you full points unless the correct algorithm you give us is efficient and that you've argued things like correctness and running time and things like that. Your algorithm could be correct and efficient, but you analyzed the running time incorrectly so we mark you points off for there.

Or most of the time what you do is you present us with an inefficient algorithm. And then you analyze the running time as if it's the target running time that we gave you. Right? That's bad on two fronts. Right? OK. So try not to fall into these traps. OK.

So some general test taking strategies when you're looking at your quiz. I really strongly urge you to read through the entire exam before you start because some of the problems will be easier for you than others. And if you're trying to maximize points on here as all of you, I'm sure, are trying to do, it's useful to make that initial pass through the problems to see which ones are easiest for you.

And then you can tackle them in the order in which you have confidence. Now in actuality, the average on say quiz one of this class tends to be around, I don't know, between 60 and 80. I don't think it's ever been 80. But it's not 100. So doing 50% of the problems well is probably going to be better for you in terms of time management and those kinds of things than doing all of the-- attempting all of the problems and not doing great on any of them in terms of point-- you have to, in computer science, you have to be pretty close to a correct answer to get points. Right?

It basically needs to be almost correct or you don't get-- if you've seen how your problem sets are being graded-- sometimes our problem set graders make mistakes. Sometimes they give you points for an incorrect solution. Right? It's really on you to take a look at your problem sets that you gave us and our solutions that we gave you. We spent a lot of time writing good solutions for you guys.

You need to make sure that the material. Don't come up to us at the end of an exam and say, oh, I said the same thing on my problem set. It was marked correct and you guys marked it wrong. Well, yeah. The staff knows a little bit more about algorithms than your graders on your problem sets. And we grade your exams. So unfortunately, that's not an excuse. It's on you to know the material. Yep?

AUDIENCE: So are most of the problems on the exams multiple part things where we need to have a good understanding and do the first few parts in order to get even partial credit on the other parts? Or is it like [INAUDIBLE]?

JASON KU: Yeah. So the question is, are questions built on top of each other so that you're kind of at a wall if you missed the first part? We try not to design exams that way. OK. You can actually take a look at the practice exam that's already been posted. Our problems tend to be self-contained. And if they are multiple parts, the parts are usually independent.

Usually you don't need to have done A correctly in order to do B correctly. All right? And that's how our problem sets try to be written as well. For your last coding question and 4 you had this problem where you had to design this data structure. But C said use that as a black box essentially and solve the problem. Right?

So you actually don't need to show-- we've given you this interface. You can just use that interface to be able to answer C, the algorithms question, without even solving the data structures question correctly. Does that make sense? And actually, the algorithms question was the easier one there, I think, from what I remember. Yeah?

AUDIENCE: Do we have to write code on the exam?

JASON KU: Do you have to write code on the exam? I've never given an exam where you've had to write code. I have written exams where you have to read code. OK? So pseudocode or Python-- since Python's a prerequisite for this class, it's completely fair game that we give you small snippets of Python code and you have to be able to understand what's going on. Yeah?

AUDIENCE: You listed amortization under the modification [INAUDIBLE]. Does that mean we're using an amortized [INAUDIBLE]?

JASON KU: Yeah. What I mean here-- amortization certainly appears in here or here-- these kinds of things. Oftentimes if I'm using a dynamic array, if I'm using a binary heap, if I'm using a hash table, amortization will appear here in our running times for those dynamic operations. What I mean here in amortization-- I mean if I'm asking you to generalize something that we've done like with dynamic arrays where instead of adding additional space at the end, I'm putting additional space in the middle or at the beginning or something like that.

And you're having to do some kind of amortized analysis. Now, often it's unnecessary to do this in-- when we talked about a problem where we did do our own amoritized analysis and making a double ended deck-- I mean a double ended queue-- you could actually solve it by reducing to using two dynamic arrays. Right? So there's a lot of ways in which you could reduce to using things. But you might have to do some additional bookkeeping at the end.

But what this is saying is that these are more-- you're not using things as a black box. You're changing something about the boxes that we gave you. Right? Does that make sense. Yeah?

AUDIENCE: If you tell us to write an algorithm that does something like O(log n) time, and we can think only of an algorithm that does something inefficiently like O(n) at end time.

JASON KU: OK.

AUDIENCE: Then is there any point writing that?

JASON KU: Sure. So let's actually move on for a second. I actually-- I'm going to answer your question very soon. OK. But I'm going to get to it in a second. OK? If I don't answer that question in five minutes, please let me know. OK? So the first thing, when I'm approaching a problem on the exam, I might try to ask some questions about the problem. OK.

It's going to help me decide what to use. Right? Different than your problems sets. Your problem sets-- basically, what do you use is what did we talk about in lecture that week. On a quiz you have eight lectures that you've talked about. And so this is going to be a harder thing for you to do because you don't know which of the eight lecture material is going to apply to this problem. And it could be a combination of them actually.

And so I'm trying to give you ways of answering that question faster. OK? So is this a mechanical reduction or a modification type problem? That's just going to help me determine the difficulty level of what this is. You might not be able to answer it. But it can give you a sense for what kind of problem it is.

Is this a problem about data structures sorting both? Right? If it's about data structures, do I need to support sequence type operations or do I need to store an extrinsic order on something? Or is it a thing where I care about what the objects are? I'm trying to look things up by what they are.

Or maybe both? Or maybe some combination? If I have a bunch of different types of keys that I might want a query on, I might have to use at least two set type of data structures. Right? You could get very complicated with these things. But putting it in terms of well, I'm going to need to do this kind of operation on these names. Then I can think, oh, I need a set data structure there. I'll think about how to implement that later.

Should I use a hash table? Should I use a sorted array? Should I use a AVL tree? But thinking about it first at the abstract level of I need a set data structure here can help you compartmentalize correctness versus efficiency. Does that make sense? OK. If you're stuck, this is your question. If you're stuck, write down a correct algorithm that's inefficient.

We can give you points for a correct algorithm that's inefficient. At least it's a correct algorithm. That's better than other things. Right? Now, if it's exponential time, you might be limited to 10% or 20% of the points. But if it's a log factor-- worse, or linear factor-- worse. Maybe that's OK. On a data structures problem, if any operation takes order n time, that's probably not going to give you a lot of points because the whole point of the data structure is to make those operations fast.

But if it solves the problem, you'll get some points. You won't get 0 points. Yeah?

AUDIENCE: Will we get any questions that are like how fast can you make this? Make this as fast as possible.

JASON KU: Yeah. So a lot of times we'll say, give us an efficient algorithm. OK. It's like, whoa, I don't know if it's efficient or not. Well, that just means that faster running times are going to give you more points. OK? So in questions like that, it's mostly trying to play this game of-- usually, we'll put an efficient one in not in terms of a data structure because usually, the data structures problem-- it's important in your implementation that these data structure operations be fast.

And we want to tell you how fast is. Right? So data structures questions in general, there's usually a trade-off between running times of these different operations. And it's really important how they relate to each other. And so for data structures problem, it's kind of about getting those running times. OK?

With an algorithms problem where we ask you to do one thing and we try to do it as fast as possible, try to get linear time. Right? Most of the time you can't get better than linear time if you have to read the entire input at some point if I want to find the things in my data. And if you can't think of a linear time algorithm, think of an n squared thing or think of n log n thing. Right?

Maybe that's a little hard for you guys to think of right now. But that's why I'm saying start with any correct algorithm and then maybe you can optimize. Maybe you can use a better data structure to make it more efficient. Does that make sense? Any other questions? OK. Moving right along. OK.

Here's some downsides. OK. If you find yourself doing one of these things three things, take a step back. You're probably doing something wrong. OK? So question yourself if you're trying to compute decimals, rationals, or real numbers. I can't store those things on a-- I mean, I can store decimals to finite precision. But if you're doing finite precision, you might as well multiply your numbers by that fixed precision and deal with the integers. Right?

We only have taught you how to deal with the integers in this class. Right? We haven't even shown you how to efficiently compute on rationals and real numbers. We have told you if you have a denominator and a numerator of a fraction, I can take two fractions and compare them right in constant time by doing cross multiplication. But if I'm trying to actually do this division to arbitrary precision, that's not happy because I can't even represent that on my computer in a finite number of decimal points for some of these things.

If you're trying to use Radix sort for every answer, it's probably wrong. One of the things that we try to do on our quizzes-- we're not just giving you a bunch of problems randomly. We probably are making problems that cover the material in some way. We do want to test you on all of the things. And so if you find that you're using the same thing four or five times on the exam, that might be a sign that you're using it too many times.

It's not always the case. Right? Sometimes hashing is super useful, so you want to use it all the time. But in particular, everyone tries to use Radix sort when it's inappropriate. And they love because it gets linear time. Right? But if you write merge sort for something where Radix sort will apply, you'll get some points because it's correct but not efficient. And it's inefficient by a log factor. Right?

If you're trying to use Radix sort in a situation where comparisons are the answer and you don't have a bound on the integers, if I don't have a bound on the size of the integers, then this may be taking a huge amount of time. So I might not even think of that as being correct because it could be exponential time. I mean, I don't know. I don't know how big my word size is. It could be arbitrarily bad. OK.

And then if you're trying to augment a binary tree with something that's not a subtree property-- something that can't be computed from the augmentations of its two children-- you're doing something bad. Every exam we have 30% of students say augment by my index in the entire tree or augment by-- here's one that's fun-- augment by the size of my left subtree-- the number of nodes in my left subtree.

How can I-- I'm not sure how I can maintain that with rotations and things like that. Let's see. In order for me to keep track of the augmentation of my left tree from the augmentation of my left tree, I have to do a logarithmic walk all the way down the thing to figure out how many things were there. So that's not a maintainable thing in constant time.

If I want to augment something of my left subtree, just augment the thing itself and just look at your left subtree and look at its augmentation. Does that make sense? Yeah?

AUDIENCE: If you had say, augmented it with a subtree and then augmented it again with [INAUDIBLE] subtree and then use that--

JASON KU: You could do that.

AUDIENCE: --does that still count as--

JASON KU: You could do that. Yeah. So you could do that in constant time by augmenting by subtree size. Right?

AUDIENCE: And then have another augmentation in [INAUDIBLE].

JASON KU: You could have another augmentation because then you could just look at-- but then just look at your left subtree please. Yeah. No reason to store it again. Right? You just do one constant time-- look to your left. OK. Don't do that. OK. So that is-- cool. I'm not that far behind. Those are my tips on solving questions. Oh, there's one more page. Yeah?

AUDIENCE: So when defining an augmentation, [INAUDIBLE] do the formula and argue that it's [INAUDIBLE].

JASON KU: Exactly. Right. So the idea is if you give an augmentation that's not a-- we're going to talk about our standard things you can reduce to in a second. If you're saying, I'm going to take a set AVL tree or a sequence AVL tree-- and for example, at the end of lecture, we were talking about how a sequence AVL tree could be modified to support priority queue operations in the same running times as binary heaps. Right?

And that was saying we store our subtree maxes-- the max thing in my subtree. Right? And so that's a different augmentation than what's already augmented on the sequence AVL tree. What are the augmentations on a sequence AVL tree?

AUDIENCE: Size.

JASON KU: Size.

AUDIENCE: Count.

JASON KU: So counts is the same thing as how many nodes are in my subtree. And?

AUDIENCE: Height.

JASON KU: Height. Right because it's an AVL tree. Right? So if I'm augmenting by max in my subtree, that's not part of my standard interface. So you need to tell me that. Even though we've done it before, it should be very easy for you. Just say, I'm augmenting by my max. Max can be computed as the max between me and my left and right subtree if they exist. Done. Right? But just do that.

You have to tell me-- and it takes constant time, so it can be maintained at constant time when I'm doing my stuff. Does that makes sense?

OK. So the last thing, I guess especially on the data structures problems, I would suggest that you approach these things by solving these problems just in terms of the interfaces first. Because then at least you get something that's correct. And then choose the algorithms or data structures that you use to implement those interfaces afterwards.

One gets you to a correct algorithm. The other is for efficiency. Decoupling these might help you in solving the problem. If it doesn't help you, don't. If you're like, whenever I see a set data structure, I'm going to probably use a hash table, that's probably fine. But if we're looking for worst case time bounds, that's probably not fun.

So you just-- I'm suggesting that you separate these things so that you concentrate on solving the problem first and then optimize it later. Yeah?

AUDIENCE: Just a question in regards to worst case time bounds.

JASON KU: Mm-hmm.

AUDIENCE: So for a hash table.

JASON KU: Mm-hmm.

AUDIENCE: Given that's a code one expected.

JASON KU: Mm-hmm.

AUDIENCE: That also implies that's O of n worst case. So could you --technically

JASON KU: It doesn't imply that. It is that. So, I mean, you could have a data structure whose expected time bound is constant, but it's a worst case bound is n log n. It just happens to be the fact that for a hash table, those worst case operations are linear. But if I --had we had a question up here beforehand, if I had a running time bound that I did something to a hash table in constant expected time, I did a look up, and then I queried an a AVL tree for the predecessor of a node or something like that, and I did that in O of log n time, what's the worst case running time of that?

AUDIENCE: O of n.

JASON KU: O of n. What's the expected running time of this thing?

AUDIENCE: Log n.

JASON KU: Log n. Expected log n. Because it's possible that in the worst case it could be higher. Does that makes sense? OK. --so OK. The second bullet is just setting up a data structures problem. There's a lot of moving parts. We're going to do two data structures problem at the end of this session.

Describe all of the data structures you're using, including what they store. If you're storing a set data structure, you better tell me what the things you're stored are keyed on. Usually the things that we're storing contain a bunch of information, and if you just say I'm storing all of the toppings of my pizza in the set data structure, and that's all you tell me, I have no idea what you're talking about because I don't know what the semantics of your data structure are. What is it keyed on? I have to say, oh, it's keyed on, I don't know the y's or something like that.

OK. And there are invariants. How we're setting up these data structures problem, usually how I solve these when I'm writing the solutions, I set up a state of what this data structure could be at some instance. I'm going to say, this data structure stores all of the things less than --k with key less than k, blah, blah, blah. And this one stores the extrinsic order of the items based on blah, blah, blah. OK.

So actually, me stating what they store in that way is actually imposing some kind of invariant on these data structures that I'm wanting to maintain. But what I need to do to prove that this thing is correct is that based on the assumption that those invariants held before my operation. Then I can prove that an operation is correct if all of those invariants are maintained before, then after the operation. That's kind of how I'm proving that this thing is correct.

And then when I'm querying, I'm doing some kind of lookup on this data structure, I can rely on those invariants. I know that those things are good, those things have been maintained, and so I can rely on those to look up what's the largest k of this thing. Does that make sense? If this is very abstract, we're going to get a little bit more concrete in just a second.

And then implement every operation. You have no idea how many solutions we read on the quizzes, which we give you three operations to implement and you don't even mention one. And it's usually the easiest one. It's like inserted into your data structure. It's like, come on, just say that. We can't give you points unless you mention that operation. Does that make sense?

And then it's going to help --us happy graders give you more points. Not really, but if your solution is well-organized and well-labeled and things like that, then we're going to be able to comprehend your solution better and we'll be able to give you more points. Remember, part of this class is about communication. If your thing is correct but we can't tell what you're saying, then it's not correct. OK. All right. So now we get --to any questions on that? Yeah.

AUDIENCE: Just a question on invariants. So one of the data structures that we've discussed in the class previously to state like what the invariants are, like the set AVL tree rule.

JASON KU: Right. So if it's the standard things, we were going to talk about what the standard things are now. Then you don't need to re-argue or --restate I mean, you can basically say like, because the set and sequence interfaces are defined that way, these things are correct, like --almost you can --basically you're trying to convince us that why it's correct.

If you're correctly using a set or sequence data structure in these data structures-type problems, then unless you're using it in a way that is unusual, usually you can just rely on the properties of the set and sequence data structures that we gave you. I do want you to mention that you thought about correctness. Like this data structure is correct because.

Just write a sentence saying that and arguing that you're basically maintaining the invariance of your data structure kind of at the upper level. What kinds of things is this data structure storing? What things about the global data structure are we relying on to make query operations? As long as you are convincing that after a dynamic operation when modifying the data structure that those invariant stay the --same that are still satisfied, then that's really all you need to say.

These invariants are satisfied because of the definitions of a set and sequence data structure. A lot of times it doesn't require a lot of thought for --the we're not --asking the reason why we do reduction problems is so you don't have to do a lot of work to prove to us that it's correct. We have these really nice black boxes. They are correct. We proved them to you that they're correct, and so you don't have to redo that work.

So now we're going to go through the core material I like to think about in this class. The first part of this class, aside from these mathematical tools that we developed at the beginning of the course, it's mostly about solving problems involving data structures. And we motivated this problem of sorting by saying that a sorted array is a data structure that's actually pretty useful. But how do we sort those things? Well, we showed you a bunch of ways to do that.

And this is that nice table. Lots of stuff. Why do we show you so many sorting algorithms? Why don't we just give you one algorithm? Hmm? Yeah?

AUDIENCE: --run times.

JASON KU: Different run times.

AUDIENCE: Better for --different

JASON KU: Better for different scenarios.

AUDIENCE: Yeah. They each have their own individual points, then they do better.

JASON KU: Yeah. You'll notice in this table, there's not blue all the way across for any of these things. So some of them are better for different scenarios. And actually, these comments list some special cases where these things might be better. In actuality, this ultra-blue thing is saying like this could be linear time. That's better. But in some cases this is worse than all the other things.

So be a little wary of this blue color here. Generally we're trying to get you further down in this chart if you can. And in general, like for example, merge sort, AVL sort, these are really the same in terms of asymptotic complexity and the way in which you interact with these sorting rooms.

But there are special cases where you might use insertion sort or selection --sort actually, I'm not sure about insertion sort. You had in your recitation two days ago-- I think you guys showed how to do if you had a k-proximate array where things are not more than k away from each other. Insertion sort actually runs in n times k, and so if k is small, then that's really good, that's kind of like linear. But you can actually do even better with a binary heap, which you saw in recitation, hopefully, where you can get that down to n log k by --keeping maintaining a heap as you go across and finding the max that way.

So insertion sort maybe is not so great. But selection sort, the name of the game there is that if my reads are cheap but my writes are expensive, selection sort it actually does pretty well, because I only have-- sorry. Reads are cheap and my writes are expensive. Selection sort does a linear number of swaps. It's looking down, finding the max, swapping it in, and keep it going. And so in such cases, that's actually better than any of these other algorithms that we've got. Yeah?

AUDIENCE: Those heap --sorts

JASON KU: Mm-hmm.

AUDIENCE: --time, worst case are expected.

JASON KU: This is worst case. So it's a little hard to-- there was a lot of moving parts to get to this bound in Tuesday's lecture. Basically what we did, we showed you how to think of an array as a heap-- as a binary tree. Complete binary tree. It's not an AVL-- I mean, it is an AVL tree, but AVL trees are weaker than a complete tree. Height balance is a weaker property than a complete.

The reason why we use complete is because it's unique for a number of nodes. That way, when I give you one array with a fixed length, I know exactly what tree you're talking about, because there's a one-to-one mapping there. If there was some ambiguity on what tree I was talking about, I --wouldn't heaps just wouldn't work.

So what heap sort does is it has this correspondence between arrays and binary trees. And then what it does is it provides these operations that kind of only do operations at the end. And then the in-place optimization is that, well, instead of actually popping it or pushing it onto the back of an array, I'm just going to think of a subset of my array as a heap and then always kind of pooping the max out to the end. Just leaving it behind and thinking of my heap as a smaller subset.

And that's how we --got it didn't actually de-use any amortization. For the time bound, time bounds you could actually do this with the amortized basically dynamic array version of this. The time bound doesn't rely on that. The in-place relies on keys staying all within one array. Does that make sense? You're doing a bunch of amortized operations, so that this actually does achieve worst case n log n. Yeah?

AUDIENCE: It seems that the things we learned--

JASON KU: Mm-hmm.

AUDIENCE: --not like--

JASON KU: Mm-hmm.

AUDIENCE: They tend to be better. So assume that most algorithms than we want them to be--

JASON KU: Uh huh.

AUDIENCE: --faster. It seems like people will not use as often, especially in these [INAUDIBLE] insertion and selection.

JASON KU: Right, yeah. So there are these special cases where they're good, but, I mean, generally these are better general data structures for most situations that you come across. I mean, there are cases where those other things are good, so you don't want to completely ignore them, but generally, yeah, you're trying to be lower bound in this chart.

AUDIENCE: What I mean is that if I don't have any --bound like I'm using them all except for selection sort--

JASON KU: So there's too many things on here for us to test all of them on an exam. So don't be afraid if not everything is covered. Worry when things are covered 18 times on the exam, that's not good. OK. So for radix sort, there are situations where you get linear time. It's when you're polynomial-bounded. Are there times when I want to use radix sort when I'm not polynomial-bounded in my integers?

AUDIENCE: Well, if you're not polynomial-bounded, then that could take a really long time.

JASON KU: Sure, yeah. But like where's --the this will be worse than n log n when? It's definitely better than n log n when u is polynomially bounded. Because it's linear. Yeah?

AUDIENCE: [INAUDIBLE] n to the n.

JASON KU: n to the n. OK. So if I put n to the n in here, I get an n factor that comes out here. That gives me a quadratic running time, which is not great. But when will this be better then n log n? Yeah?

AUDIENCE: n to the c, because it's less n to the c.

JASON KU: n to the c-- so that'll definitely give us linear time. That's what this is saying. But we can actually do better than n log n if this u is n to the c times log n for some c. If --it's like if it's n to the c log log n, that's smaller than log n. So that's a better algorithm. That's a faster algorithm.

Do you guys see why we wrote these things this way? It's so that we give you a more --precise this is more important for you understand what u means here than that this thing sometimes runs in linear time. We want to know when it runs in linear time. Does that makes sense? Or when it runs faster than merge sort. Does that makes sense?

OK. So that's sorting. We have sequence-type data structures. We have linked lists, we have dynamic arrays, we have sequence AVLs. Sequence AVLs are great. I don't know why no one teaches it. They're great. They don't teach it probably because they're actually not that useful. You don't actually use the insert in the middle a lot in coding.

So it's-- you can usually get around with like shifting something to the end and doing dynamic operations there, and that's a lot of the games that you try to play so you don't have to make your own data structures, and you can just use the Python list that's in your-- like native to your thing. But it has a theoretical interest because it gets these kind of balanced bounds if you need to insert in the middle of this sequence.

Now some of you look at me and had a question before that was like, but Jason, how does linked list operations on the end of a linked list, why does that take linear time? It's because in lecture, we presented you with what? A singly-linked list. It just had pointers to the next thing.

And if I only have pointers to the next thing and a pointer to the head, for me to find the end, I need to walk all the way down the list. Now let's say I keep my pointer to the tail, then finding that and n1 is fine, but removing it still takes linear time, because I don't know what came before me. And so that's why in p-set whatever, 1, 2, I don't remember, you stored a pointer back to your previous one that gave you doubly-linked lists. Yay, right?

So actually, expanding this table out, you can reference a doubly-linked list here and get this one as constant time. Does that make sense? And this one's still linear time. OK. But this one's still linear time. We actually also showed you how to do this in constant amortized. Do you guys remember that? That was in problem session 2 or 1? I don't remember. It was whatever we talking about amortized stuff.

We --got we got actually both of these to one amortized using the concepts of the dynamic array. And then we actually did it one more time where we got a good doubly-ended thing. What was that? Does anyone remember? We went to problem session 3. Yeah?

AUDIENCE: Oh. I'm probably wrong, this is--

JASON KU: OK.

AUDIENCE: --problem session 3. But like q dq type stuff?

JASON KU: So q dq, those are talking about doubly-ended things. Those are implemented in a certain way, which is actually one of these things. It's actually, I think, this one in Python. But there's --a we used a different data structure to get something that had really good at. It had really good pop and append and first and last dynamic operations. Do you --guys anyone remember from session 3?

AUDIENCE: [INAUDIBLE]

AUDIENCE: Isn't it dynamic array [INAUDIBLE]?

JASON KU: That's what this was. OK. We got one that had expected bounds. Does that help? What? I heard someone say it. Hash table. Yeah. So basically what you did, instead --of this is a sequence thing. These things don't have keys. But I could identify with each item as I stick it in, a key representing its index. And I could use a hash table that way.

Now there was some difficulty if I removed the first thing. Because actually, how --those are all of my indices have now changed. But if I just store what the smallest index in my thing is, then I'm golden, because I can compute what that index should be I change things at the front.

OK. So there's actually three different ways we showed you of getting constant time at the front and the back of this thing. So actually, you can think of that as standard material that you can reduce to. That's the one exception of the things that I'm not showing on this chart, but is a bonus if you're watching this problem session. Yeah?

AUDIENCE: So isn't a hash table a set data structure, though?

JASON KU: It is. But we used it to implement a sequenced data structure. So I will refer you to that problem session if you want to learn more about it. So, any questions on sequenced data structures? Yeah?

AUDIENCE: I--

JASON KU: Yeah, uh huh.

AUDIENCE: Should we be-- or-- yeah. Should we be able to look prove the tables that we're given?

JASON KU: So I would hope-- hope-- that if I gave you a blank table, that you would be able to fill it out. That's how well I want you to know how these things are implemented. We're not going to have that on the exam. That's a boring kind of question.

AUDIENCE: But it is important to know?

JASON KU: Yeah. I would-- it's good for you to think, like, oh, if I'm going to use an AVL tree, operations are generally going to be log n. That's a really useful thing-- or, if I'm going to use a hash table, dictionary-type operations, finding, inserting, and deleting, those are fast. Doing order-type operations on hash tables is bad, that's hard to do because I have to basically look through all my things. Knowing that dynamic operations on a sorted array is bad.

Or knowing that-- you have to think about here what we mean when we say linked list and dynamic array in this table, because--

AUDIENCE: Singly-linked.

JASON KU: Yeah, exactly. We are implying singly-linked here because that's what we presented to you in lecture. And so those are the standard things that we want you to reduce to. Yeah?

AUDIENCE: --three modified standard things are the doubly-linked lists, the doubly-ended--

JASON KU: Yeah. Basically you can assume that you have-- it's this one that you probably want to use, because you get constant indexing. And then pretty good on both ends. But if you need a double-ended Q, you can also reduce it to having two dynamic arrays back-to-back when going this way, when going this way.

And so there's lots of-- we're not going to give that to you as a standard method because there's literally like four methods we showed you how to do. So you choose one. OK. Yeah?

AUDIENCE: In the interest of saving time on the exam, if we want to say, like, we do this thing with the sequence AVL and it takes log n time. Do we have to say the sentence like, because this is sequence AVL, it takes a longer time because this? Or can we just like say like, for the table that we know?

JASON KU: Yeah, no. If you told me that you're storing things in a sequence AVL and you just say-- you basically say what you do to it, and you say, which takes blah, blah, blah time, you don't need to say because it's a sequence AVL, because you already told us it's a sequence AVL. I believe that you probably wrote the chart on your cheat sheet and you looked it up. All right. Any other questions on this? No. Yeah?

AUDIENCE: Just make sure, the doubly-linked lists will make the insert on [INAUDIBLE]. Is that the only change?

JASON KU: Yes. That doubly-linked list gives this guy constant time. And actually, there are two operations here, insert, delete, and-- I guess there's a find here as well. If I just store the tail pointer, that gets defined to constant time. but it doesn't get the dynamic ones to constant time. I need to store the previous pointers as well at each node. Does that make sense? OK.

Lastly-- or I guess second to lastly, set data structures. There are a little bit more of these. Not so many more. But yeah. We had a sorted array, gets good find, but is not dynamic. We set AVL tree which does pretty good find and is dynamic. Again, you get this n log n overhead to build, because essentially what you're doing with both of those data structures is to sort.

But if I'm looking on a theory question, like I'm not asking you something specifically about a sorted array, if you have a choice between this data structure and this data structure, which one do you choose? Well, I don't quite know, because it's almost like this one's better at everything except for this one. But can anyone tell me how to make this one constant time as well?

Augmentation. I could just store in my subtrees the max or min. And I can make this one strictly better than this one. So in theory problem, you probably just want to use this one. Here, hash tables, direct access arrays, even better for these operations. That's great. But they suck at these. So if you need these, don't use these.

And in actual coding, especially if you're coding in a language that's not Python, something that doesn't automatically give you a hash table, if you're in C in a microcontroller lab at MIT here, you're taking 6115 or whatever, and you're doing assembly, usually what you're doing is a direct access stuff. Because that's giving you the jumps that you need in machine language to actually go and access this in constant time. That's generally if you have control over the keys that you're putting-- that you are putting in your data structure, you don't want this overhead of running your keys through a hash function to look these things up. You just store the things in an array.

You use the hash table when you don't have control over the keys or your keys are like strings or something. So that's when you use the hash table. Now for our purposes, usually a hash table is just as good unless we're asking you for worst case bounds. And we're going to do this when we talk about the data structures problem. If we give you a situation where we don't care if you achieve worst case are expected or amortized or any of these things, we'll just say, make sure you state which one you achieve. And as long as you analyzed it correctly with respect to your data structures, then you're fine.

But if we say you better do worst case here, I'm going to slap you around. Then please get those bounds. Don't use the hash table in that case. That makes sense? OK. Lastly, we've got priority queues which we talked about. I'm not going to go through this one very much. It's basically just adding this to the thing. But in actuality, you can get all of these bounds with a set AVL-- I mean a sequence AVL tree with max or min augmentation, which isn't on this list because we didn't really talk about it, but hopefully you can-- if you need these bounds without the amortization, then you could achieve them.

All right. So that's basically everything that we've talked about in the class. We're going to spend the rest of the time working a couple of data structures problem. I'm not-- there's a number of different types of questions you'll actually see on the quiz-- the practice quiz that we gave you from last term. There are some what I call mechanical-type questions up at the front. Then usually some reduction-type problems where you're reducing to using some sorting algorithms or some data structures.

And then usually the latter ones are ones where you have to do some kind of something additional, like some augmentation or some divide and conquer or something like that. OK. So we're going to go ahead and spend the rest of the time working a couple of these problems. These were from spring of 2019 on the exam. And actually, one of the TAs who's TAing for us now was also TAing for us in spring 2019. Was grading problem number 2 here, the rainy research problem, and just hated me, because no one did it right. All right. So, let's try to solve these problems.

So problem 1 is about restaurant-- restaurant, yes, OK. All right. So basically, what's happening? A popular restaurant, Criminal Seafood. What's the reference? Legal Seafood. Yes, opposite. Does not take reservations, but maintains a wait list where customers who have been on the wait list longer are seated earlier. Sometimes customers decide to eat somewhere else, so restaurant must remove them from the wait list. OK.

Assume that a customer has a different name. No two customers are added to the wait list at the exact same time. So there's a kind of an ordering at which people are being added to this wait list. Does that makes sense? Design a database to help Criminals Seafood maintain its wait lists supporting the following operations, each in constant time.

OK, so here, we've refactored the running time up to the top. And it's a-- oh, sorry. I added the build here. I guess that's still constant time. That's fine, OK.

AUDIENCE: I have time d--

JASON KU: Yeah. State whether each running time operation is worst case, amortized, expected. So, when you see that statement, you're saying, OK, I'm allowed to use a hash table if I want to. I just have to make sure if I use one, I label my operations expected and amortized when they occur.

Which operations are expected? Basically all of them. Which operations are amortized? The ones that changed what's in the data structure, insert, delete. OK. So we've got some operations, building an empty thing. Adding a name-- so this name is x to the back of the wait list. What do I know about x? What do I know about the names per our assumptions in this class?

AUDIENCE: They're unique, so they can be a key.

JASON KU: They're unique, so they can be a key. And they fit in a constant number of words by our assumption. So I can compare two of them in-- or I can hash one in constant time. That's kind of the assumption we make on these inputs, they're strings. And I didn't assign you a bound on their length, so it's probably not something you need to worry about.

OK remove a name. So already I'm feeling like, I need to be able to find things by their name. OK. And then seat the next person in line. Does that makes sense? So what kinds of things do I need to maintain here? I have people. They have names and they have kind of places, the time that they came in. But am I given the times? No. I'm not given times anywhere. It's not on the inputs to my operations. So it's not like I'm going to be able to key on time. That makes sense?

What's the important part about the times? The order.

AUDIENCE: And you're given that.

JASON KU: Yeah. Basically whenever-- I'm basically trying to maintain a sequence on these guys. There's a front one and a back one and people in the middle. And I want to make sure that order stays the same, or else people are going to getting angry at me, because ah, they came here after me and I got a-- yeah. You've been in that situation. OK.

So, we're trying to maintain some kind of sequence, an extrinsic order on these things. But we're also needing to be able to look up people by their name because I want to be able to change this thing. Does this maybe sound familiar to some other problem we had on this term's problem sets? Yeah, I think there was a problem where--

AUDIENCE: The chat.

JASON KU: The chat. You had to store a sequence. But you also had this dictionary that you had to look things up. Now the nice thing about that situation is that the things that you needed to look up on was static. And so what could I use for my dictionary, my sets data structure for that? Anyone remember? You could just use a sorted array. Because it's static, these things aren't updating all the time, and so it was fine for me to just use a static array.

Here-- and we gave you lookup times that were worst case logarithmic. Here, I'm asking for constant time. Sorted array is not going to cut it, set AVL's not going to cut it, so what am I going to use?

AUDIENCE: Dynamic--

JASON KU: A dynamic array or a hash table. Now dynamic array might not be great because I don't actually have a numeric bound on how big these keys are, I just know that they fit in words. So I can't actually make a direct access array because those words-- while they fit in a constant number of words, I don't know if the integer representation of those are polynomial-bounded. Does that make sense? So I do want to use hashing in this case.

And so what do I want to maintain? I want to maintain a sequence data structure on customers. Customers. Is there a U in there somewhere? No. This is right, right? OK. And a set mapping. So this is usually how I do it. Like I want to say a set keyed on something. If it's just a set, then I just have it key. I can look up whether that thing is there or not.

But when I'm actually having it mapped to something else, I'll say, mapping a key, space, to something else, usually the item that I'm storing or maybe some property of the items that I'm storing. Does that make sense? OK. So mapping, what do I map here? Names. To-- oh. To what? Do I want to map?

AUDIENCE: To time--

JASON KU: The time. The time that they entered. Do I want to do that?

AUDIENCE: Well, you can't exactly do it with the time that they entered, but the sequence on the customers will show what's next in line.

JASON KU: Yeah. I want to store it to where it is in this sequence. So I'll just store the index where it is in the sequence and I can just look it up. That's sounds-- yeah? Is that what--

AUDIENCE: I thought the indices change.

JASON KU: Yeah. Indices change every time I add or remove things.

AUDIENCE: [INAUDIBLE]

JASON KU: Ah, yeah. Store a pointer. Pointer to place in sequence. Now, that's a little weird to say, because I haven't told you how I represented this sequence. But conceptually I can say I'm drawing a pointer to someplace in this that represents where it is, I'll deal with the details of that later.

But generally I call this a linked data structure, because I'm linking between two data structures so that I can do a query in one, say, and find out where it is and the other. Or do a query in the other, that kind of thing. So, everyone kind of understand why I chose these things? How I approached this problem? Yeah?

AUDIENCE: I'm not sure how a pointer fixes the problem of just storing an index. The place that a person is at in the sequence is going to change over time.

JASON KU: Mm-hmm.

AUDIENCE: --assuming that a pointer updates, would that change?

JASON KU: So-- OK. So let's say if every time-- so let's say I'm storing-- let's say I'm storing this sequence in a linked list, let's just say, because I know that I'm going to have to be inserting and deleting things from the middle of this thing. Does that make sense? All right. So I'm going to go ahead and say that this is going to be a linked list. Now I have this linked list.

So if I said, all right, the set data structure in my set stores-- or stores where it is in the linked list, let's say it's stored at k, all right? Cool. Now if I seat this guy, I stick him at his table, every index has changed. So for me to update the indices stored in this set data structure, I have to change every single one of them.

AUDIENCE: Yeah.

JASON KU: Does that make sense?

AUDIENCE: Yeah, no, this makes way more sense.

JASON KU: Yeah. So really what I want, not to store a number here, but an actual pointer to the node that's containing this thing, because the node isn't changing unless that thing left my data structure. The node, the address of this thing in memory, the node is just a little container that contains an item and the next pointer, and actually we're going to need a previous pointer here. We're going to need a doubly-linked list.

Because what we're going to do when we remove something, we need to stitch it back together in constant time, which means we need to know what's the one in front of us and the one behind us. Yeah? No? All right. So linked list here. We already said maybe using a hash table here.

And so, OK, great. This is basically enough for me to say, this is my data structure, these are the invariants. What are the invariants? It's storing all the customers. That's an invariant. I mean, it's not a very strong invariant, but it's-- yeah, I mean, I should say that. I'm storing all the customers because I'm going to have to make sure that I'm maintaining that when I do a query.

And then here, set mapping names to pointers. The place in this thing. And as I remove things, I need to make sure that invariant stays the same. It's still mapping all the names of customers that I have in my thing to their nodes. So when something leaves, I better need to make sure that both of these things are updated. Does that make sense?

OK. So how do I maintain these operations? We've got a build-- build just sets up empty of these things. It's easier to say that. I build an empty linked list, doubly-linked lists on the customers, and I build a set-- a hash table mapping nothing to nothing right now. OK. So that's build.

I'm going to be precise here and actually write down the thing because I told you not to ignore a operation, or else we can't give you points for it. The next one-- yeah?

AUDIENCE: By emptying-- by building an empty linked list--

JASON KU: Mm-hmm.

AUDIENCE: --it just has like the head and the tail, it's actually none, but, like, it exists?

JASON KU: Yeah, right. It's a thing in memory that we store. It has a pointer to a head and a tail. Those are none right now. But we will add things to it. OK. So the second one, add name. What do we have to say? We have to update this data structure so it maintains it's a variance. I have to-- usually I start with one of them. I get to a point where, oh, I really should have updated the other one first. So sometimes it's hit or miss.

What do I want to do here? Well, I have no idea where in this sequence this is. So I kind of have to go here first. Oh, sorry. Adding a guy, I could do either way. I know-- where is he going to go? End of the list. So I just stick it there. So add x to end of sequence.

So I come to a trick that I like. Sequence, set. OK. In this instance I only have one sequence and one set. And so calling them sequence and set, probably fine. I'm not going to confuse the greater, I'm not going to confuse myself. But when I have more than one of these things, or even for brevity on an exam where I'm time-constrained, give these things a name. Say this is a sequence C and M. I don't know. I see customer here, I see a map here, so maybe. I don't know. But just give them a letter, then we can follow you much more clearly and you can refer to these things more precisely. So end of C.

Then what do we have to do? I fix this guy, this guy's good. Now I have to fix this guy. So I add x to the set and I map it back to that node that I just came from. I could store it in a temporary variable. Add x to M pointing to n node v to v. OK. Cool?

So, I added my x to the end of C into a node v. I'm kind of-- I labeled it so I can reference it later. And in code, I would probably remember what that node was. Add x to M pointing to that node, and now I've maintained my invariant. Great.

AUDIENCE: Maybe [INAUDIBLE].

JASON KU: Add key x. Or add x to M having the key of x point to that node. Does that make sense? There's a subtlety there. 3 3, we have remove-- I don't remember the name, whatever. OK. So here, we have to do-- order matters. I don't know where it is in the sequence, but I'm going to look it up in the set, and I'm going to remove it from the set, and I'm going to look at whatever it points to and remove it from the sequence. I'm running out of time a little bit here, so I'm not going to write all that down. You guys understand what that means.

Here, we're using the fact that it's a doubly-linked list so that we can relink things together. You don't have to tell me the three points-- the previous pointer of my next thing to the next-- the-- right? You don't have to tell me how to relink those pointers because we did that already in problem set 1 or whatever.

AUDIENCE: Just in general, as you're in a situation for where you're using not a doubly-linked list--

JASON KU: Mm-hmm.

AUDIENCE: where just a linked list would be better than a doubly linked list because it seems like it always solves problems.

JASON KU: Yeah. Doubly-linked lists is almost always strictly better than a singly-linked in theory problems. So yeah, use it. OK. And then seat, last one. That's just take the front of the sequence, remove it, change the head around, but I get that you can delete first. You're reducing to the interface that we had. You're deleting first on the sequence.

But now, we have a situation. I deleted the first guy. How do I know who the first guy is? Well, I'm storing its name in there. I'm storing the names of these customers. So I know who is at the front, I look in this set data structure, and I remove that entry. Does that makes sense? Because now, I no longer need to maintain where he goes.

Now in actuality, I could just not update this set data structure. But if I do that, then, well, my running times are still linear time. I'm not giving you a bound on space. I'm still constant time, sorry. So you don't actually have to do that removal, but if that customer comes back and wants to get on the wait list again, there are things to consider.

OK. So that's that question. Next question, last 10 minutes or so. Raniy Research. This is a problem that people had nightmares about. OK, so basically we've got a Meather Wan. He's a weatherman. A scientist who studies global rainfall. And he's got a bunch of sensors everywhere. And each one can post to the cloud or something a measurement that's of the form a triple of integers r, l, and t where r is a positive amount of rainfall, an integer; a latitude, an integer again; and at a time. We got three things to deal with here. Yuck.

But they're all integers. And don't be like, oh, well Jason, latitudes are pretty small. So I can assume these integers are small and these things take constant time. I'm not specifying to you a resolution at which I'm measuring these integers. And I haven't given you a bound between what that resolution is compared to the number of measurements that I have, so I don't play those games. OK.

The peak rainfall at a given latitude since a particular time is the maximum rainfall at any measurement at that latitude measured at a time greater than or equal to that time. Does that make sense? Or 0 if there aren't measurements at that latitude. OK. To score after the time-- or before the time or whatever.

Describe a database that we can build it in constant time. Is an empty one-- I added this one because we weren't good about that last spring. Record data. We give you a triplet. And then-- so record data, for it to be correct, I just have to maintain that information. For these kind of updates, I don't-- it's really hard for me to argue that this thing is correct. Because I just-- I throw it at the database, the database doesn't have to give anything back to me.

So the important thing here about correctness is that peak rainfall gives it to me and it gives it to me in the time bound that I'm looking for. And peak rainfall is returning the peak rainfall at a particular latitude since t. So we have three things. Yep?

AUDIENCE: Given that you never have to return a single measurement, is that [INAUDIBLE] have a record of it?

JASON KU: There's the potential that you don't need to store all of the information because all we're doing is giving you back R's, essentially. It's possible that you don't need to store the latitudes or times at all. You don't even need to store the triplets. Now in reality, I'm querying on the latitudes and the times. So I should store them somewhere, but I might be able to compress them.

In particular, many things could be stored at the same latitude. That's kind of the whole point of the query. And so we want-- I mean, we may only need to store that latitude once. Does that makes sense? OK. I'm going to wait for questions until after because I want to get to a solution to this problem.

All right. So what do we need to do? We need to be able to add things. And I want to return. So return, I'm going to have to query something, and then I'm going to return something. So return peak rainfall at latitude l since time t. What do I care about in a particular query? I only care about all the things at l, at a latitude l.

So really, this isn't such an interesting thing, but I want to be able to have maybe many data structures, one associated with each L. Does that make sense? And how can I find one in each L quickly? Put it in a dictionary. What's my time bound? Worst case log n. So what data structure do I use for that set data structure? A set AVL.

So you're going to first have a set AVL-- say, L-- mapping latitudes to-- well, now we have more data structures. I want to store a lot of the things that have the same latitudes in another data structure. The ones probably storing the times in the rainfalls of all those measurements. Yeah? Yeah.

AUDIENCE: A hash table.

JASON KU: A hash table. OK, so what kinds of query am I going to want to do on the things at the same latitude?

AUDIENCE: You're going to want to get the times.

JASON KU: I'm going to want to get the times, but more than that, I'm doing kind of an ordered query. I need the things less than a certain time. Greater, sorry.

AUDIENCE: So it's really--

JASON KU: Just a second.

AUDIENCE: Could you do like an AVL for the time and an AVL for the rain?

JASON KU: OK. Do I care about an AVL for the rain? I mean, looking up on the rainfall.

AUDIENCE: No.

JASON KU: No. So I'm going to go ahead and store these things in a time-sorted AVL to mapping latitude l to-- I'm going to call this data structure a time data structure. I'm going to say it's t of l. That kind of looks like a recurrence, so it's a little irking me right now, but I don't have anything better.

All right. So now each one of these time data structures is a set AVL mapping time to the rainfall measurement. All right. So that's going to-- if my query was, return the peak rainfall-- sorry. Return the rainfall of the thing with latitude l and time t, we'd be done, kind of. You'd know how to support that query. To insert things, I insert things into both data structures, and I just look it up.

The one complication here is that I'm not asking for what is the rainfall at a particular time. I want to know what the maximum rainfall is up to this time. OK. So max heap's good if I want to know the global max. But here, I want to know the max bounded by a certain range. So we're going to-- you can ask me questions after this, we're running out of time a little bit. So anyone have an idea for how-- yeah?

AUDIENCE: Could you just augment the AVL with the max?

JASON KU: OK.

AUDIENCE: And you can just look at the right child and then just look at the max at that point?

JASON KU: Ah, OK. So what your colleague is saying, if we augment by the max r in my subtree, maybe we can use that to figure out this query. Because we're ordered on t, right we have this nice monotonic property that everything that's going to be in my query-- everything to the right of a certain time-- if my time is above t at a particular node, in everything in the right subtree is also above that t because of the order of my set's data structure, because I'm ordered on times.

So there's maybe the possibility that if I look to my right subtree, I can not do work all over here by just looking at the max in that subtree. so That's an idea of, let's say, augmented by subtree max r. You probably want to give this a name as well. So like v max where v is a node in my thing.

And I want to show how to support this, how I can compute that from its children. So how do I actually support this query, I can think of it recursively. I have a couple of cases. If I'm at a v here, I want to define a recursive function that's called peak rainfall of a given node lower bounded by a t.

So if I'm here, there's two cases. Either my t, my time is bigger or smaller-- is in my range or out of my range. If it's out of my range, what do I do? It's lower than my time bound. I can just recursively call this function on this node. Because I know that anything is going to be down here that I care about.

And that's just one recursive call down the tree. And so if I only limit myself to one recursive call down the tree, I'm always going down each time. This is going to take logarithmic time. So that's the first case, that's the easy case. This thing is not in range, I return recursively the thing to my right.

What's the other case? I'm in my range. Well now I could return, recursively call both sides. Because that's what this peak is talking about. What's my peak rainfall? But if I do that, if I call it here recursively and I call it here recursively, that could take linear time. I might touch every node in my tree.

So why did I do this augmentation? So I don't have to do work on this node. I just return the maximum rainfall in this subtree, and then recurse on this side. So I did constant work on this side, I did one recursive call down here which you could go to the bottom, but that's OK, I can afford to go to the bottom of the tree. Does that makes sense?

And if I don't have either subtree, then I'm done. If at any time. I don't have the node that I'm supposed to recurse on, I take the max of this subtree, myself, and whatever the recursive return value is here. And comparing three values, returning their max. Does that make sense? OK. So that's what we call a one-sided range query. So in the-- I think problem session 4 that I didn't get to, it shows you a way to do this for a two-sided range query where I need to know the maximum of all things between two things.

But it's really no more difficult than this to finding a recursive function that uses an augmentation so that you don't have to do recursive call on both sides. Does that makes sense? OK. That's going to be it for now, and I can take questions after. Thanks for coming.