PROFESSOR: All right. Welcome back to 6046.

AUDIENCE: Woohoo.

PROFESSOR: Are you guys ready to learn an awesome data structure?

AUDIENCE: Woohoo.

PROFESSOR: Yeah, let's do it. This is a data structure named after a human being, Peter van Emde Boas. I was just corresponding with him yesterday. And he, in the '70s, he invented this really cool data structure. Its super fast It's amazing.

It's actually pretty simple to implement. And it's used a lot, in practice, in network routers, among other things. And we're going to cover it today. So let me first tell you what it does.

So it's an old data structure. But I feel like it's taken us decades to really understand.

Question.

AUDIENCE: You're mic's not on.

PROFESSOR: In what sense? It's not amplified. It's just for the cameras.

So it's taken us decades, really, to understand this data structure, exactly how it works and why it's useful. The problem it's solving is what you might call a predecessor problem. It's very similar to the sort of problem that binary search trees solve. But we're going to do it faster, but in a somewhat different model, in that the elements we're going to be storing are not just things that we know how to compare.

That would be the comparison model. We're storing integers. And the integers come from a universe, U, of size little u. And we'll assume that they're non-negative, so from 0 to u minus 1. Although you could support negative integers without much more effort.

And the operations we want to support, we're storing a set of n of those elements. We want to do insert, delete, and successor. So these are operations you should be familiar with. You should know how to solve these in log n time per operation with a balanced binary search tree, like AVL trees.

You want to add something to the set, delete something from the set, or given a value I want to know the next largest value that is in the set. So if you draw that as a one dimensional thing, you've got some items which are in your set. And then, you have a query. So you ask for the successor of this value.

Then you're asking for, what is the next value that's in the set? So you want to return this item. OK, predecessor would be the symmetric thing. But if you could solve successor, you could usually solve predecessor. So we'll focus on these three operations, although, in the textbook, you'll see there are lots of operations you could do with van Emde Boas.

So far so good. We know how to do this in log n time. We are going to do it in log log u time. Woah, amazing. So an extra log, but we're cheating a little bit, in that we're replacing n with u.

Now in a lot of applications, u is pretty reasonable, like 2 to the 32 or 2 to the 64, depending on what kind of integers you usually work with. So log log of that is usually really tiny, and often smaller than log n.

So in particular, on the theory side, for example, if u is a polynomial in n, or even larger than that, you can support n to the polylog n. Then log log u is the same as log log n, up to constant factors. And so this is an exponential improvement over regular balanced binary search trees.

OK, so super fast, and it's also pretty clean and simple, though it'll take us a little while to get there. One application for this, as I mentioned, is in network routers. And I believe most network routers use the van Emde Boas data structure these days, though just changed in the last decade or so.

Network router, you have to store a routing table, which looks like, for IP range from this to this, please send your packets along this port. For IP range from this to this, send along this port. So if you mark the beginnings of those ranges as items in your set, and given an actual IP address, you want to know what range it's in, that is a predecessor or a successor problem. And so van Emde Boas lets you solve that really fast.

u, for IPV4 is only is only 2 to the 32. So that's super fast and practical. It's going to take like

five operations to do log log 2 to the 32. So that's it. And as you may know, network routers are basically computers.

And so they used to have a lot of specialized hardware. These days it's pretty general purpose. And so you want nice data structures, like the one we'll cover.

OK, so we want to shoot for log log u. We're going to get there by a series of improvements on a very simple idea. This is not the original way that van Emde Boas got to this concept. But it's sort of the modern take on it. It's one that's in the textbook.

So the first question is, how might we get a log log u bound? Where might that come from? That's a question for you. This is just intuition. Any suggestions?

We see logs all the time. So, yeah.

AUDIENCE: You organize the height of a tree into a tree.

PROFESSOR: Ah, good. You organize the height of the tree into a tree. So we normally think of a tree, let's say we have u down here. So the height is log u. So somehow, we want a binary search on the levels of this tree.

Right, if we could kind of start in the middle level, and then decide whether we need to go up or down, I'm totally unclear what that would mean. But in fact, that's exactly the van Emde Boas will do. So you can binary search-- I think we won't see that until the very end-- but on levels of a tree.

So at least some intuition. Now let's think about this in terms of recurrences. There's a recurrence for binary search, which is usually you have k things, t of k is t of k over 2 plus order 1. You spend constant time to decide whether you should go left or right in a binary search, or in this case up and down somehow. And then you reduce to a problem of half the size.

So this solves to log k. In our case, k is actually log u. So we want a recurrence that looks something like t of log u equals t of log u/2 plus order 1. OK, even if you don't believe in the binary search perspective, this is clearly a recurrence that solves to log log u. I'm just substituting k equals log u here. So that could be on the right track.

Now, that's in terms of log u. What if I wanted to rewrite this recurrence in terms u? What

would I get? If I wanted to have this still solve to log log u, what should I write here?

If I change the logarithm of a number by a factor of 2, how does u change?

AUDIENCE: Square root.

PROFESSOR: Square root. OK, So I've changed what the variable is here. But this is really the same recurrence. It will still solve to log log u. The number of times you have to apply square root to a number to get to 1 is log log u.

So this is some more intuition for how van Emde Boas is going to achieve log log u. And in fact, this is the primary intuition we'll be using. So what we would like is to some take our problem, which has size u, and split it into problems of size square root of u, so that we only have to recurse on one of them. And then, we'll get this recurrence.

OK, that's where we're going to go. But we're going to start with a very simple data structure for representing a set of n numbers from the universe 0 up to u minus 1. And let's say, initially, our goal is for insert and delete to be constant time. But let's not worry about successor. Successor could take linear time.

What would be a good data structure for storing items in this universe? I want u to be involved somehow. I don't just want to, like, store them in a linked list of items or assorted array of items. I would like u to be involved, insert and delete constant time.

Very simple. Yeah.

AUDIENCE: Simply an array.

PROFESSOR: In an array, yeah. What's the array indexed by?

AUDIENCE: It would be index n.

PROFESSOR: Sorry?

AUDIENCE: By the index of n.

PROFESSOR: The index of n, close.

AUDIENCE: The value.

PROFESSOR: Sorry?

AUDIENCE: The value.

PROFESSOR: The value, yeah. Good. So I want-- this is normally called a bit vector, where I want array of size u, and for each cell in the array, I'm going to write 0 or 1. 0 means absent. 1 means present. It's in the set.

So let me draw a picture, maybe over here. Let me take my example and give you a frisbee. Let me put it in the middle.

So this is an example of a set with-- if I maybe highlight a little bit-- here's 1. Here's a 1, and a one, and a one. So there are 4 elements in the set. The universe size is 16. n equals 4, in this particular example.

If I want to insert into this set, I just change 0 to a 1. If I want to delete from the set, I change a 1 to a 0. So those are constant time. Good.

If I want to do a successor query, not so good. I might need to spend order u time. Maybe I asked for the successor of this item, and the only thing to do is just keep jumping until I get to a 1.

And the worst case, there's almost to u 0's in a row, or u minus n. So that's really slow. But this, in fact, will be our starting point. It may seem really silly. But it's actually a good starting point for van Emde Boas.

So the second idea is, we're going to take our universe and split it into clusters. van Emde Boas, the person, likes to call these galaxies. I think that's a nice name for pieces of the universe. But textbook calls it clusters. Because they used to call it clusters.

So now, it's question of how big the cluster should be. But I gave you this picture, and I want to think about these galaxies as separate chunks, and I ask for the successor of this, how could I possibly speed up the successor search? Yeah.

AUDIENCE: You could form a tree for each cluster and connect--

PROFESSOR: You could form a tree here and store what at the--

[INTERPOSING VOICES]

AUDIENCE: Could store an or between the two bits.

PROFESSOR: Cool. I like this. So I could store the or of these two bits-- clean this up a little bit-- or of these two bits, or of these two bits, and so on. The or is interesting, because this 0 bit, in particular, tells me there's nothing in here. So I should just be able to skip over it.

So you're imagining a kind of binary search-ish thing. It's a good idea. So each node here, I'm just writing the or of its two children. And in fact, you could do this all the way up. You could build an entire binary tree.

But remember, what we're trying to do is a binary search on the levels of the tree. And so, in particular, I'm going to focus on this level. This is the middle level of that tree if I drew out the whole thing.

And that level is interesting, because it's just summarizing-- is there anybody in here, is there anybody in this cluster, is there anybody in this cluster, is there anybody in this cluster. So we call this the summary vector. So we'll come back to your tree perspective at some point. That is a good big picture of what's going on.

But at this level, I'm just going to say, well let's store the bit vector. Let's also store this summary vector. And now, when I want to find the successor of something, first I'll look inside the cluster. If I don't find my answer, I'll go up to the summary vector and find where is the next cluster that has something in it.

And then I'll go into that cluster and look for the first one. OK, that's a good next step. So this will split the universe into clusters.

How big should the clusters be to balance out? There's three searches I'm doing. One is within a cluster. One is in the summary vector. And one is within another cluster. Yeah.

AUDIENCE: Square root u.

PROFESSOR: Square root u. Yeah. That will balance out. If there's square root of u size, then the number of clusters will be square root of u. So the search in the summary vector will be the same as the cost down here.

Also we know that we kind of want to do square root of u recursion somehow. So this is not yet the recursive version. But square root of u is exactly right. And I owe some frisbees, sorry.

Here's one frisbee. And yeah, cool.

And I think also you one. Sorry. So clusters have size square root of u, the square root of u of them. And, cool.

So now, when I want to do an insert or a delete, it's still-- let's not worry about delete. That's a little tricky. To do an insert, it's still easy. If I insert into here, I set it to 1. And I check, if this is already 0, I should also set that to 1.

Now deleting would be tricky. To delete this guy and realize that there's nothing else, eh. Let's not worry about that until we do a lot more work. Let's just focus on insert and successor.

So insert, with this strategy, is still constant time. It's two steps instead of one, but it's good. Successor does three things. First, we look, let's say, successor of x.

First thing we do is look in x's cluster. Then, if we don't find what we're looking for, then we'll look for the next 1 bit in the summary vector, and then, we'll look for the first 1 in that cluster.

So there are two cases. In the lucky case, we find the successor in the cluster that we started in. So that only takes root u time. If we're unlucky, we research in the summary. That takes root u time. And then we find the first 1 bit. That takes root u time.

Whole thing is square root of u, which is, of course, not very good, compared to log n. But it's a lot better than u, which is our first method, the bit vector. So we've improved from u to square root of u. Now of course, the idea is to recurse.

Instead of just doing a bit vector at each of these levels, we're going to recursively represent each of these clusters in this way. This is where things get a little magical, in the magic of divide and conquer. And then, we'll get t of square root of u instead of square root of u. And then we'll get a log log cost.

So before I get there, let me give you a little bit of terminology and an example for dealing with clusters. OK, in general, remember the things we're searching for are just integers. And what we're talking about is essentially dividing an integer, like x, by square root of u.

And so this is, whatever, the quotient. And this is the remainder. So I want j to be between 0 and strictly less than square root of u. Then this is unique, fundamental theorem of arithmetic, or something.

And i is the cluster number. And then j is the position of x within that cluster. So let's do an example like x equals 9. So I didn't number them over here. This is x equals 0, 1, 2, 3, 4, 5, 6, 7, 8, 9-- here's the guy I'm interested in-- 10, 11, 12, and so on.

So 9 is here. This is cluster number 0, 1, 2. So I claim 9 equals 2 times square root of u. Here is 4. I conveniently chose u to be a perfect square. And it is item 0,1 within the cluster.

And indeed, 9 equals 2 times 4 plus 1. So in general, if you're given x, and I said, ah, look in x's cluster, what that means is look at x integer divided by square root of u. That's the cluster number. And I'll try to search in there.

And I look in the summary vector, starting from that cluster name, the name of the cluster for this guy, finding the next cluster. Then I'll multiply by square root of u to get here, and then continue on.

In general, because dividing to multiplying-- I don't want to have to think about it too hard. I'm going to say, define some functions to make this a little easier, more intuitive. So when I do integer division by square root of u, which is like taking the floor, I'll call that high of x, the high part of x.

And low of x is going to be the remainder. That's the j up here. And if I have the high and the low part, the i and the j, I'm going to use index to go back to x. So index of ij is going to be i times square root of u plus j.

Now why do I call these high and low? I'll give you a hint. Here's the binary representation of x. In this case, high of x is 2. And low of x is 1. Yeah.

AUDIENCE: So the high x corresponds to the first two, which is the first 2 bit. And the low x corresponds to [INAUDIBLE].

PROFESSOR: Right. High of x corresponds to the high half of the bits. And low of x corresponds to the bottom half of the bits. So these are the high order bits and the low order bits.

And if you think about it, remember when we take square root of u in logarithm, it takes log u and divides it in half. So it's exactly, in the bit factor, which is log u bits long, we're dividing in half here, and looking at the high bits versus the low bits.

OK? So that's another interpretation of what this is doing. And if you don't like doing division,

as many computers don't like to do, all we're actually doing is masking out these bits, or taking these bits and shifting them over. So these are very efficient to actually do.

And maybe get some intuition for why they're relevant. So let's recurse, shall we? I think now we know how this splitting things up works.

So I'm going to call the overall structure v, or a van Emde Boas structure I'm trying to represent is v. And v is going to consist of two parts. One is an array of all of the clusters.

I'm going to abbreviate van Emde Boas as VEB. And recursively, each of those clusters is going to be represented by a smaller VEB structure, of size square root of the given one.

OK, and i ranges from 0 to square root of u minus 1. OK, so there's square root of u of them. Total sizes is u. And then, in addition, we're going to have a summary structure. And this is also a size square root of u VEB. OK, you should think about inserts and successors. Those are the two operations I care about for now. Let's start with insert. That's easier.

So if I want to insert an item, x, into data structure v, then first thing I should do is insert into its corresponding cluster. So let's just get comfortable with that notation. We're inserting into the cluster whose number is high of x. That is where x belongs. The name of its cluster should be high of x.

And what we're going to be inserting recursively into there is low of x. That is the name of x local to that cluster. x is a global name with respect to v. This cluster only represents a small range of square root of u items. So this gets us from the big space of size u to the small space of size square root of u within that cluster. So that's basically what high and low were made for.

But then, we have to also update the summary structure. So we need, just in case-- Maybe it's already there. But in the worst case, it isn't. So we'll just think of that as recursively inserting into v dot summary the name of the cluster, which is high of x.

High of x is keeping track of which clusters are non-empty. We've just inserted something into this cluster. So it's non-empty. We better mark that that cluster, high of x, is non-empty in the summary structure.

Why? So we can do successor. So let's move on to successor. Actually, I want to mimic the successor written here on the bottom of the board.

So what we had in the non-recursive version was three steps. So we're going to do the same thing here. We're going to look within x's cluster. We now know that is the cluster known as high of x.

And either we find, and we're happy, or we don't. Then we're going to look at v dot summary search for this the successor of high of x. Right, finding the next 1 bit, that is successor.

And then, I want to find the first 1 bit in that cluster. Is that a successor also? Yeah. That's just the successor of negative infinity. Finding the minimum element in a cluster is the successor of -1, or 0, or not zero. But -1 would work, or negative infinity, maybe more intuitively.

That'll find the smallest thing here. So each of these is a recursive call. I can think of it as recursively calling successor. So let's do that.

I want to find the successor of x in v. First thing I'm going to do is do the ij breakdown. I'll let i be high of x and j be-- I could do low of x. But what I'm going to try for is to search within this cluster, high of x.

So I'm going to look for the successor of cluster i, which is cluster high of x, of low of x. OK, so that's this first step of looking in x's cluster. This is x's cluster. This is x's name in the cluster.

I'm going to try to find the successor. But it might say infinity. I didn't find anything. And then I'll be unhappy if j equals infinity. So that's line one.

Well, then we're in the wrong cluster. High of x is not the right cluster. Let's find the correct cluster, which is going to be the next non-empty cluster. So I'm going to change i to be the successor in the summary structure of i.

So i was the name of a cluster. It may have items in it. But we want to find the next non-empty thing. Because we know the successor we're looking for is not here. OK.

So this is the cluster we now belong in. What item in the cluster do we want? Well, we want to find the minimum item in that cluster. And we're going to do that by a recursive call, which is j is the successor within cluster i of minus infinity, I'll say. -1 would also work.

So this will find the smallest item in the cluster. And then, in both cases, we get i and j, which together in this form describe the value x that we care about. So I'm just going to say, return index of ij. That's how we reconstruct an item name for the structure v.

We knew which substructure it's in. And we know its name within the substructure, within the cluster. Is this algorithm clearly correct? Good. It's also really bad.

Well, it's better than everything we've done so far. The last result we had was square root of u. This is going to be better than that, but still not log log u. Why? Both of these are bad.

Yeah.

AUDIENCE: You make more than one call to [? your insert. ?]

PROFESSOR: Right. I make more than one recursive call to whatever the operation is here. Insert calls insert twice. Here, successor calls successor potentially three times. This is a good challenge for me.

Let's see. Eh, not bad. Off by one. OK, that's a common problem in computer science, right? Always off by one errors.

OK, so let's think of it in terms of recurrences, in case that's not clear. Here we have t of u is 2 times t of square root of u. Right, to solve a problem of size u, I solve two problems of size square root of u plus constant. Because high of x and low of x, I'm assuming, take constant time to do.

It's just, I have an integer. I divide it in half. Those are cheap. What does this solve to? It's probably easier to think of it in terms of log u. Then we could apply the master method.

Right, this is the same thing as t prime of log u is 2 times t of log u divided by 2 plus order 1. This is not quite the merge sort recurrence. But it's not good.

One way to think of it, is we start with the total weight of log u. We split into log over 2, but two copies of it. So we're not saving anything. And we didn't reduce the problem strictly.

In terms of the recursion tree, we have, you know, log u-- well, it's hard to think about because we have constant total cost. You could just plug this in with the Master method, or see that essentially we're conserving mass.

We started with log u mass. We have two copies of log u over 2. That's the same total mass. So how many recursions do we do? Well we do do log log u recursions. The total number of leaves in that recursion tree is log u.

Each of them, we pay constant. So this is log u, not log log u. To get log log u, we need to

change this 2 into a 1. We can only afford one recursive call. If we have two recursive calls, we get logarithmic performance.

If we have three recursive calls, it's even worse. Here, I would definitely use the Master method. It's less obvious. In this case, we get log u to the log base 2 of 3 power, which is log u to the 1.6 or so, so both worse than log n.

This is strictly worse than log n. This is maybe just a little bit worse than log n, depending on how u relates to n. OK, so we're not there yet. But we're on the right track. We have the right kind of structure. We have a problem of size u.

We split it up into square root of u sub problems of size u. From a data structures perspective, this the first time we're using divide and conquer for data structures. It's a little different from algorithms. So that's how the data structure is being laid out.

But now we're worried about the algorithms on those data structures. Those, we can only afford t of u equals 1 times [? t of ?] squared of u plus order 1. Then we get log log u. So, here we have two recursive calls. Somehow we have to have only one.

Let's start by fixing insert. Insert? No. Let's start by fixing successor. I think that will be more intuitive. Let's look at successor.

Because successor is almost there. A lot of the time, it's just going to make this call, and we're happy. The bad cases is when we need that make both of these calls. Then there's three total, very bad.

How could I get rid of this call? I was being all clever, that the minimum element is the successor of negative infinity. But that's actually not the right idea. Yeah.

[? AUDIENCE: Catching ?] the minimum element in cluster i.

PROFESSOR: Store the minimum element of cluster i. Yeah. In general, for every structure v, let's store the minimum. Why not? We know how to augment structures.

Here in 006, you took an AVL tree, and you augment node to store the sub-tree size of the node. In this case, we're doing a similar kind of augmentation. Just for every structure, keep track of what the minimum is. So that will be idea number four.

I'm going to add something here. But for now, let's store the minimums. So to do an insert into

to structure v, item x, first thing we'll do is just say, well, if x is-- let's see if it's the new minimum. Maybe x is smaller than v dot min.

If that's the case, let's just set v dot min to x. OK? And then, the rest is the same, same insertion algorithm as over here, these two recursive calls. I just spent constant additional time. And now every structure knows it's minimum.

Again, ignore delete for now. That's trickier. OK, now every structure knows its minimum, which means we can replace this call with just v dot cluster i dot min. One down.

OK, so if we look at successor, of v comma x. I'm going to replace the last line, or next to last line with j equals v cluster i dot min.

So now, we're down to log u performance. We only have, at most, two recursive calls. So that's partial progress. But we need another idea to get rid of the second one. And the intuition here is that really, only one of these call should matter.

OK, let's draw the big picture. Here's what the recursive thing looks like. We've got v dot summary. Then we've got a cluster 0, cluster 1, cluster square root of u minus 1. Each of those is a recursive structure.

And we're also just storing the min over here as a copy. So when I do a query for, I don't know, the successor of this guy, there's kind of two cases. One situation is that I find the successor somewhere in this interval.

In that case, I'm happy. Because I just need this one recursive call. OK, the other case is that I don't find what I'm looking for here. Then I have to do a successor up here. And then I'm done.

Then I can teleport into whatever cluster it is. And I've stored the min by now. So that's constant time to jump into the right spot in the cluster. So either I find what I'm looking for here, or I find what I'm looking for here.

What would be really nice is if I could tell ahead of time which one is going to succeed. Because then, if I know this is not going to find anything, I might as well just go immediately up here, and look at the successor in the summary structure. If I know I'm going to find something here, I'll just do the successor here. And I'm done.

If I could just get away with one or the other of these calls, not both, I'd be very happy. How could I tell that? Yeah.

AUDIENCE: Store the max.

PROFESSOR: Store the max. Store the min and the max. Why not? OK, I just need a similar line here. If x is bigger than v dot max, change the max. So now, I've augmented my data structure to have the min and max at every level.

And what's going on here is, I won't find an answer if I am greater than or equal to the maximum within this cluster. That's how I tell. If I'm equal to the max, or if I'm beyond the max, if all the items are over here, the max will be to my left.

And then I know I will fail within the cluster. So I might as well just go up to summary and do it there. On the other hand, if I'm less than the max, then I'm guaranteed I will find something in this cluster. And so I can just search in there.

So all I need to do-- I'll probably have to rewrite this slightly. If x is-- not x, close. I'm going to mimic this code a little bit, at least the first line is going to be i equals high of x. And now, that's the cluster I'm starting in.

And I want to look at the maximum of that cluster. So I'm looking at v dot cluster i dot max. And I want to know, is x before that? Now within that cluster, x is known as low of x.

So I compare low of x to cluster i's maximum element. If we're strictly to the left, then there is a successor guaranteed within that substructure. And so, I should do this line. I wish I could copy paste. But I will copy by hand.

Successor within v dot cluster i, of low of x. OK, then I've found the item I'm looking for. Else, I'm beyond the max, I know this is the wrong cluster. And so I should immediately do these two lines, well, except I've made the second line use the min.

So it will only be one recursive call, followed by a min. OK, so this is going to be i equals the successor within v dot summary of high of x. And then j is that line successor within-- oh, sorry-- the line that I used to have here, which is going to be v cluster i dot min.

OK, and then, in both cases, I return index of ij. OK, so we're doing essentially the same logic as over here. Although I've replaced the step with the min, to get rid of that recursive call. But

I'm really only doing one or the other of these, using max to distinguish.

If I'm left of the max, I do the successor within cluster high of x. If I'm right of the max, then I do the successor immediately in summary structure. Because I know this won't find anything useful. And then I find the min within that non-empty structure.

And in both cases, ij is the element I'm looking for. I put it back together with index. Clear? What's the running time of successor now? Log log u.

Awesome. We've finished successor. Sadly, we have not finished insert. Insert still takes log u time. But, b progress. Maybe your routing table doesn't change that often, so you can afford to pay some extra time for insert, as long as you can route packets really fast, as long as you can find where something belongs, the successor in log log u time.

But for kicks, let's do insert in log log u as well. This is going to be a little harder, or I would say a more surprising idea. This may be--

I don't have a great intuition for this step. I'm thinking. But again, most of the time, this should be fine, right? Most of the time, we insert into cluster high of x, low of x, and we're done. As long as there is something already in that cluster, we don't need to update the summary structure.

As long as high of x has already been inserted into the summary structure, we can get away with just this first step. The tricky part is detecting. How would we know? Well, that's not enough just to detect it.

If high of x is not in v dot summary, we have to do this insert. We can't get away with it. But that's kind of rare. That only happens the very first time you insert into that cluster. Every subsequent time, it's going to be really cheap. We just have to do this.

It's easy enough to keep track of whether a cluster is empty. For example, we're storing the min. We can say v dot min is none, special value, whenever the structure v is empty.

But we still have this problem, that the first time we insert into a cluster, it's expensive. Because we have to do this. And we have to do this. How could we avoid, in the case where a cluster is empty-- remember, an overall structure looks like this. We can tell that it's empty by saying min equals none, let's say.

What could I do? Sorry, there's also a max now. What could I do to speed up inserting into an empty cluster? Because I'm first going to have to insert into the empty cluster. Then I'm going to have to answer into the summary.

I can't get away from this. So I'd like this to become cheap, in the special case when this cluster is empty. Yeah.

AUDIENCE: Lazy propogation.

PROFESSOR: Lazy propagation-- you want to elaborate?

AUDIENCE: Yeah. We mark the place we want to insert in. And then we will take it down whenever we [? insert ?] there.

PROFESSOR: Good. So when I insert into an empty structure, I'm just going to have a little lazy field, or something. And I'll put the item in there. And then the next time I insert into it, maybe I'll carry it down a little bit. That actually works. And that was the original van Emde Boas structure, [? I ?] [? learned ?] [? recently. ?]

So that works. But it's a little more complicated than the solution I have in mind. So I'm going to unify that lazy field with the minimum field. Say, when I insert into a structure, if there's nothing here, I'm just going to put the item there, and not recurse. I just am not going to store the minimum item recursively.

Definitely frisbee. So that's the last idea, pretty much. Idea number five is, don't store the min recursively. This is effectively equivalent to lazy.

But we're actually just never going to get around to moving this guy down. Just leave it there. First, if the min field is blank, store the item there. Yeah.

AUDIENCE: What do you mean by moving the guy down?

PROFESSOR: Don't worry about moving the guy down. We're not going to do it.

AUDIENCE: [INAUDIBLE]

PROFESSOR: But in general, moving down means, when I want to insert an item, I have to move it down into its sub cluster. So I want to insert x into the cluster, high of x with low of x, that recursive call. That's moving it down. I'm not going to do that.

If the structure is empty, I'm going to set v dot min equal to x, and then stop. Let me illustrate with some code, maybe over here.

Here's what I mean. If v dot min is special none value-- use Python notation here-- then I'm just going to set v dot min equal to x. I should also set v dot max equal to x. Because I want to keep track of the maximum. And then, stop. Return.

That's all I will do for inserting into an empty structure, is stick it in the max field. OK, this may seem like a minor change. But it's going to make this cheap. So the rest of the algorithm is going to be pretty similar.

There's a couple annoying special cases, which is, we have to keep the min up to date. And we have to keep the max up to date, in general. This one is easy. We just set v dot max equal to x. Because we're not doing anything fancy with max.

Min is a little special. Because if we're inserting an item smaller than the current minimum, then really x belongs in the slot. And then whatever was in here needs to be recursively inserted. OK, so I'm going to say swap x with v dot min.

So I'm going to put x into the v dot min slot. And I'm going to pull out whatever item was in there and call it x now. And now my remaining goal is to insert x into the rest of the structure. There's only one item that gets this freedom of not being recursively stored. And it's always going to be the minimum one.

So this way, the new value x goes there. Whatever it used to be there now has to be recursively inserted. Because every item except the minimum, we're going to recursively insert.

So the rest is pretty much the same. But we're going to, instead of always inserting into the summary structure, we're going to see whether it's necessary. Because we know how to do that. We just look at a cluster high of x. And we see, is it empty?

Cluster high of x-- and empty means its minimum is none. So we're going to-- in fact, the next line after this one is going to be insert v cluster high of x, comma low of x. All right, that's this line. We're always going to do that.

And in the special case, where there was not previously nothing in v cluster high of x, we need to update the summary structure. And we do that with this line. So I'm going to insert into v dot

summary high of x.

But I'm only doing that in the case when I need to. If it was already non-empty, I know this has already happened. So I don't need to bother with that insertion. OK, this is a weird algorithm. Because it doesn't look much better.

In the worst case, we're doing two recursive calls to insert. But I claim this runs in log log u time. Why? Yeah.

AUDIENCE: Because when we update the v dot summary, we [? just ?] [? have the ?] [? first ?] [? line. ?]

PROFESSOR: Good. Yeah. In the case when I have to do this summary insertion, I know this guy was empty. Cluster high of x was empty. So this call is just going to do these two lines.

Because I optimized the case of empty-- when a structure is empty, I spend constant time, no recursive calls. That means in the case when cluster high of x is empty, and I have to pay to insert into the summary structure, I know my second call is going to be free, only take constant time.

So either I do this, in which case this takes constant time, or I don't do this, in which case I make one recursive call. In both cases, I really am only making one recursive call.

OK, so this runs in log log u. Because I get the t of u equals 1 times square root of t of u plus order 1 recurrence. All the work I'm doing here is constant time, other than the recursive calls. Question?

AUDIENCE: So when we insert the first time, we don't update v dot summary?

PROFESSOR: When I insert into a completely empty structure, we don't update summary at all. That's right. We just store it in the min, and we're done.

AUDIENCE: Oh. So then, if you were to [? call ?] the successor, and you--

PROFESSOR: Good. Yeah. The successor algorithm is currently incorrect. Thank you. Here's some frisbees for that question and the last answer. Yeah. This code is now slightly wrong.

Because sometimes I'm storing elements in v dot min. And successor is just completely ignoring them. So it's not going to find those items. Luckily, it's a very simple fix.

Out of room, but please insert right in here. If x is less v dot min, return v dot min. That's all we

need to do.

The min is special. Because we're not storing it recursively. And so, we can't rely on all of our recursive structures. We can't rely on cluster i. We can't rely on summary, on reporting about v dot min.

v dot min is just a special item sitting there. It's represented nowhere else. But we can check. Because it's the minimum element, and we're looking for successors, it's really easy to check for whether it's the item we're looking for. Because it's the smallest one.

If we're smaller than it, then that's clearly the successor. OK, so in that case, we just spent constant time. So it actually speeds up some situations for successor. We're not exploiting that here.

It doesn't help much in the worst case. But now, it should be correct. Hopefully, you're happy. Any other questions?

So at this point, we have what I will call a van Emde Boas. This last version-- we can do insert and successor in log log u time. Yeah, sorry. I modified the wrong successor algorithm, didn't I?

I meant to modify this one. This is the fast one. So please put that code here. That's the log log u version of successor. We just added this constant time check. And now this runs in log log u time.

The key idea here was if we store the max, then we know which of the two recursive calls we need to do. If we store the min, this doesn't end up being a recursive call. So that's very clean.

With insert, we needed this trickier idea that the min, we're not even going to recursively represent. We'll just keep it there. That requires this extra little check for successor. But it allows us to do insert cheaply in all cases-- cheap meaning only one recursive call.

Either we need to update the summary structure, in which case that thing was empty, and so we can think of that cluster-- so we have this special case of inserting into an empty cluster, which is super cheap, or most of the time, you imagine that the cluster was already non-empty. And so we don't need to update the summary structure. And then we just do this recursion.

So in all cases, everything is cheap. Now the one thing I've been avoiding is delete. Yeah, question.

AUDIENCE: [INAUDIBLE] If x is greater than [? v ?] max, [? we ?] [? swap ?] [? x ?] [? with ?] [? v ?] [? max? ?]

PROFESSOR: So if x is greater than v max, I'm just going to update v max. V max is stored recursively. We're not doing anything fancy with v max. And we had, at some point, a similar line.

So this is just updating v max. Yeah, nothing special there. In your problem set, you'll look at a more symmetric version, where you don't recursively store min and max. It works about the same.

But in some ways, the code is actually prettier. So you'll get to do that. Other questions?

All right. So, delete. We have insert and successor. And through all these steps, it would actually be very hard to do delete. It turns out, at this point, delete is no problem.

So let me give you some delete codes. It's a little bit long. Maybe I'll start with a high level picture, sort of the main cases.

Deleting the min is a little bit special, as you might imagine. That element is different from every other element. So if x equals min, we're going to do something else. But let me specify that later.

Let's get to the bulk of the code, which is we're going to delete low of x from cluster high of x. That's the obvious recursion to do. This is essentially the reverse of insert over here. The first thing we do is undo this.

In all cases, insert was doing that. So in all cases, delete has to do that, other than the special case of the min. And then, we need to do the inverse of this. So if that was the last item, then we need to delete from the summary structure.

So it's actually pretty symmetric, other than the tiny details. So after we delete, we can check, is that structure empty. Because then, the min would equal none. OK. If that's the case, we delete from the summary structure.

OK. Cool. And there is a bit of a special case at the end, which is when we deleted the maximum element. OK, so I need to fill these in.

And it's important that these are filled in right. Because in some situations here, we are making two recursive calls. But again, we'd like it to be, when we do both calls, we want one of them to be cheap. Now this one's hard to make cheap.

So when we delete from the summary structure, we want this to delete to have taken only constant time, no recursions. And that's going to correspond to this case. Because if we made the cluster empty, that means we deleted the last item. What's the last item?

Has to be v dot min. If you have a size 1 structure, it's always because that item is in v dot min, everything else is empty. So that's the case of deleting v dot min. So we want this case to take constant time when it's the last item we're deleting.

So let's fill that in a little. Let's see if I can fit it in. This is code that turns out to work in this if x equals v dot min.

It's a little bit subtle. But the key thing to check here is, we want to know, is this the last item. And one way to do that is to look at the summary structure, and say, do you have any non-empty clusters?

If you don't have any non-empty clusters, that means your min is none. And that means, the only thing keeping the structure non-empty is the minimum element. That's stored in v dot min. So in that case, that's the one situation when v dot min becomes none.

We never set v dot min equals none in the other algorithms. Because initially everything is none. But when we're inserting, we never empty a structure. Now we're doing delete. This is the one situation when v dot min becomes none from scratch.

In that case, no recursive calls. So that means this algorithm is efficient. Because if I had to delete from the summary structure, this only had a single item, which is this situation. Then I just set v dot min equals to none. And I'm done. So this will, overall, run in log log u time.

Now, it could be we're deleting the min, but it was not the only item. So that's this situation. In that situation, we want to find out what the min actually is. Right?

We just deleted the min. We want to put something in v dot min. We can't set it to none. Because that indicates the whole structure is empty. So we have to recursively rip out the new minimum out item. Because it should not be recursively stored anymore.

And then we're going to stick it into v dot min. So now, finding minimum items is actually pretty easy. We just looked at the first non-empty structure. And we looked at the-- I think I'm missing-- oh, v dot cluster i min, I guess, closed parenthesis.

That is the minimum item in the first cluster. So I want to recursively delete it. So I'm setting x to that thing. And then I'm going to do all this code, which will delete x from that structure. And then-- I mean, I'm doing it all right here.

But then, I'm going to set v dot min to be that value. So then v dot min has a new value. Because I deleted the old one. And it's no longer recursively stored. I don't want two copies of x floating around.

So that's why I do, even in this if case, I do all these steps. Cool? You can see delete-- is that a question?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, why did I set v dot max to none?

AUDIENCE: Because [? that's the ?] all [? these ?] [INAUDIBLE] [? x ?] equals v dot max, the last time.

AUDIENCE: [? Do you ?] [? find v dot max? ?]

PROFESSOR: Oh, right. I'm not done yet. I haven't specified what to do here. OK, you really want to know? OK. Let's go somewhere else. I have enough room, I think.

Eh, maybe I can squeeze it in. It's going to be super compact. So, when x equals v dot max, there are two cases. So max is a little different. We just need to keep it up to date. So it's not that hard. We don't have to do any recursive magic.

Well, I need another line. Sorry. Let me go up to the other board. OK, I think that's the complete delete code. You asked for it. You've got it.

So, at this point, we have just deleted the max, which means we need to find, basically, the predecessor of x. But we can't afford a recursive call. I mean, that's OK. It's just, we're trying to find the max in what remains.

Imagine v dot max is just wrong. So we've got to set it from scratch. It's not that hard to do. Basically, we want to take the last non-empty structure. That would v dot summary dot max,

and then find the last item in that cluster.

OK, so cluster i is the last one for v dot summary. And then we look v dot cluster of i dot max. And we combine it with i. That gives us the name of that item in the last cluster, the last non-empty cluster.

But there's a special case, which is maybe this returns none. Maybe there actually is nothing in v dot summary. That means we just deleted the last item, I guess. Or there's only one left. We deleted the next to last time.

Now there's only one item left, namely v dot min. So we set v dot max equal to v dot min. So that's a special case. But most the time, you're just doing a couple dot max's, and you're done. So that's how you maintain the maxes, even when you're deleting.

And unless I made an error, I think all these algorithms work together. You're going to insert, delete, and successor. And symmetrically, you can do predecessor in log log u time per operation, super fast.

Let me tell you a couple other things. One is, there's a matching lower bound. Log log-- maybe you wonder, can I get log log log time, log log log log time, or whatever? No.

In most reasonable choices of parameters-- it's a little bit more complicated than this-- but for most of the time that you care about, log log u is the right answer. This was proved in 2007. So it took us decades to really understand. It's by a former MIT student.

So I'll give you some range where it holds, which will raise another issue. But, OK. So this range is the range I talked about before. This is when log log u equals log log n. So that's kind of the case where you care about applying it.

If log log u is more like log n, it's not so interesting. But as long as u is not too big, this is a little bit bigger than polynomial n. Then this is the right answer.

Now technically, you need another assumption, which is the space of your data structure is not to super linear. Now this is a little awkward. Because the space of this data show structure is actually order u, not n. So the last issue is space.

Space is order u. Let me go back to this binary tree picture. So we had the idea of, well, there's all these bits at the bottom. We're building a big binary tree above those. The leaves

are the actual data. And then we're summarizing, by for every node, we're writing the or of the two nodes below it, which is summarizing whether that thing is non-empty.

What van Emde Boas is doing-- so first of all, you see that the total number of nodes in this tree is order u. Because there's u leaves. The total size of a binary tree with u leaves is order u, 2u minus 1, right? And you can kind of see what van Emde Boas is doing here.

First, it's thinking about the middle level. Now it's not directly looking at these bits. It says, hey look, I know my item, the thing I'm doing a successor of, let's say, is three. I want to know the successor of this position.

First, I want to check, should I recurse in this block, or should I recurse in the summary block-- which I didn't draw. But it's the part of the tree that would be up here. And that's exactly what we're doing with successor.

Should we recursively look within cluster i? Or should we look within the summary structure? We only do one or the other. And that's the sense in which we are binary searching on the levels of this tree.

Either we will spend all of our work recursively looking for the successor within the summary structure, which is like finding the next 1 bit in this row, the middle row, or we will spend all of our time doing successor in here. And we can do that. Because we have the max augmented.

OK, but that's the sense in which, kind of, you are binary searching in the levels of this tree. So that's that early intuition for van Emde Boas is kind of what we're doing. The trouble is, to store that tree takes order u space.

We'd really like to spend order n space. And I have four minutes. So you'll see part of the answer to this. My poor microphone. Let me give you an idea of how to fix the space bound.

Let's erase some algorithms. The main idea here is only store non-empty clusters, pretty simple idea. We want to spend space only for the present items, not for the absent ones. So don't store the absent ones.

In particular, we're doing all this work around when clusters are empty, in which case we can see that just by looking at the min item, or when they're non-empty. So let's just store the non-empty ones. That will get you down to almost order n space, not quite, but close.

To do this, v dot cluster is no longer an array. Just make it a hash table, a dictionary in Python. So v dot cluster-- we were always doing v dot cluster of i. Just make that into dictionary instead of an array.

And you save most of the space. You only have to store the non-empty items. And you should know from 006, hash table is constant expected. We'll prove that formally in lecture eight, I think. But for now, take hashing as given.

Everything we did before is essentially the same cost, but an expectation, no longer worst case. But now the space goes way down. Because if you look at an item, when you insert an item, it sort of goes to log log u different places, in the worst case.

But, yeah. We end up with n log log u space, which is pretty good, almost linear space. It's a little tricky to see why you get log log u. But I guess if you look at the insert algorithm, even though we had two recursive calls in the worst case.

One of them was free. When we do both of them, we insert here. This one happens to be free. Because it was empty. But we still pay for it.

We set v dot min equal to x. And so that structure went from empty to non-empty. So this costs 1. And then we recursively call insert v dot summary on high of x.

So we might, when we insert one item x, if lots of things were empty, actually log log u structures become non-empty, and that's why you pay log log u for each item you insert. It's kind of annoying.

There is a fix, which is in my notes. You can read it, for reducing this further to order n. But, OK, I have 30 seconds to explain it. The idea is-- you're not responsible for knowing it. This is just in case you're curious.

The idea is, instead of going all the way down in the recursion, at the very bottom, you say, well, normally if you stop the recursion when you have u equals 1, just stop the recursion when n is very small, like log log u.

When I'm only storing log log u items, put them in a linked list. I don't care. You can do whatever you want on log log u items in log log u time. It's just a tiny tweak. But it turns out, it gets rid of that log u in the space.

So it's a little bit messier. And I don't know if you'd want to implement it that way. But you can reduce to linear space. And that's van Emde Boas.