MITOCW | watch?v=u-HHY1ylhHY

ERIK DEMAINE: All right. Today is all about the predecessor problem, which is a problem we've certainly talked about implicitly with, say, binary search trees. You want to be able to insert and delete into a set, and compute the predecessor and successor of any given key. So maybe define that formally.

And this is not really our first, but it is an example of an integer data structure. And for whatever reason, I don't brand hashing as an integer data structure, just because it's its own beast. But in particular, today, I need to be a little more formal about the models of computation we're allowing-- or I want to be. In particular, because, in the predecessor problem, which is, insert, delete, predecessor, successor, there are actually lower bounds that say you cannot do better than such and such.

With hashing, there aren't really any lower bounds, because you can do everything in constant time with high probability. So I mean, there are maybe some lower bounds on deterministic hashing. That's harder. But if you allow randomization, there's no real lower bounds, whereas predecessor, there is.

And in general, predecessor problem-- the key thing I want to highlight is that we're maintaining here a set of-- the set is called s of n elements, which live in some universe, U-- just like last time. When you insert, you can insert an arbitrary element of the universe. That probably shouldn't be an s, or it will get thrown away.

But the key thing is that predecessor and successor operate not just on the [INAUDIBLE] in s, but you can give it any key. It doesn't have to be in there. And it will find the previous key that is in s, or the next key that is in s. So predecessor is the largest key that is less than or equal to x that's in your set. And successor is the smallest that is larger-- of course, if there is one.

So those are the kinds of operations we want to do. Now, we know how to do all of this n log n time, no problem, with binary search trees, in the comparison model. But I want to introduce two more, say, realistic models of computers, that ignore the memory hierarchy, but think about regular RAM machines-- random access machines-- and what they can really do.

And it's a model we're going to be working on for the next, I think, five lectures. So, important to set the stage right. So these are models for integer data structures.

In general, we have a unifying concept, which is a word of information, a word of data, a word of memory. It's used all over the place-- a word of input. A word is the machine theoretic sense, not like the linguistic sense. It's going to be a w-bit integer.

And so this defines the universe, which is-- I'm going to assume they're all unsigned integers. So this is 2 to the w minus one. Those are all the unsigned integers you can represent with w-bits. We'll also call this number, 2 to the w, little u. That is the size of the universe, which is capital U.

So this matches notation from last time. But I'm really highlighting how many bits we have, which is w. Now, here's where things get interesting.

I'm going to get to a model called a word RAM, which is what you might expect, more or less. But before I get there I want to define something called transdichotomous RAM-- tough word to spell. It just means bridging a dichotomy-- bridging two worlds, if you will. RAM is a random access machine. I've certainly mentioned the word RAM before. But now we're going to get a little more precise about it.

So in general, in the RAM, memory is an array. And you can do random access into the array. But now, we're going to say, the cells of the memory-- each slot in that array-- is a word. Everything is going to be a word. Every input-- all these x's are going to be words. Everything will be a word. And in particular, the things in your memory are words.

Let's say you have s of them. That's your space bound. In general, in transdichotomous RAM, you can do any operation that reads and writes a constant number of words in memory. And in particular, you can do random access to that memory.

But in particular, we use words to serve as pointers. Here's my memory of words. Each of them is w bits-- so s of them, from, I guess, 0 to s minus one.

And if you have, like, the number 3 here, that can be used as a pointer to the third slot of memory. One, two, three. You can use numbers as indexes into memory. So that's what I mean by, words serve as pointers.

So particularly, you can implement a pointer machine, which-- no surprise-- but for this to work, we need a lower bound on w. This implies w has to be at least log of the space bound. Otherwise, you just can't index your whole memory.

And if you've got s minus 1 things, this 2 to the w minus 1 better be at least s minus 1. So we get this lower bound. So in particular, presumably, s is at least your problem size, n. If you're trying to maintain n items, you've got to store them. So w is at least log n.

Now, this relation is essentially a statement bridging two worlds. Namely, you have, on the one hand, your model of computation, which has a particular word size. And in reality, we think of that as being 32 or 64 or maybe 128. Some fancy operations on Intel machines, you can do 128-bit or so.

And then there's your problem size, which we think of as an input. Now, this is relating the two. It's a little weird. I guess you could say it's just a limitation for a given CPU. There's only certain problems you can solve.

But theoretically, it makes a lot of sense to relate these two. Because if you're in a RAM, and you've got to be able to index your data, you need at least that many bits just to be able to talk about all those things. And so the claim is, basically, machines will grow to accommodate memory size. As memory size grows, you'll need more bits.

Now, in reality, there's only about 2 to 256-- what do you call them-- particles in the known universe. So word size probably won't get that much bigger. Beyond 256 should be OK.

But theoretically, this is a nice way to formalize this claim that word sizes don't need to get too big unless memories get gigantic. So it may seem weird at first, but it's very natural. And all real world machines have big enough words to accommodate that.

Word size could be bigger, and that will give you, essentially, more parallelism. But it should be at least that big. All right. Enough proselytizing.

That's the transdichotomous RAM. The end. And the word RAM is a specific version of the transdichotomous RAM, where you restrict the operations to c-like operations. These are sort of the standard-- they're instructions on, basically, all computers, except a few risk architectures don't have multiplication and division. But everything else is on everything.

So these are the operators, unless I missed one, in c. They're all in Python, and pick your-- most languages. You've got integer arithmetic, including mod. You've got bitwise and, bitwise or, bitwise x or, bitwise negation, and shift left, and shift right.

These we all view as taking constant time. They take one or two integer inputs-- words as inputs. They can compute an answer. They write out another word. Of course, there's also random access-- array dereference, I guess.

So that's the word RAM. You restrict to these operations. Whereas transdichotomous RAM, you can do weird things, as long as they only involve a constant number of words. Word RAM-- it's the regular thing. So this is basically the standard model that all integer data structures use, pretty much.

If they don't use this model, they have to say so. Otherwise this model has become accepted as the normal one. It took several years before people realized that's a good model-- good enough to capture pretty much everything we want.

The cool thing about word RAM is, it lets you do things on w-bits in parallel. You can take the and of w-bits, pairwise, all at once. So you get some speed up. But it's a natural generalization of something like the comparison model.

Comparison model-- I guess I didn't write those. It's more operations-- less than, greater than, and so on. You can compare two numbers in constant time, get a Boolean output via, say, subtraction, and computing the sine. And you think of comparisons as taking constant time, so why not all of these things? Cool.

One more model-- this is kind of a weird one. It's called cell probe model, which is, we just count the number of memory reads and writes that we need to do to solve a data structure or a query. Like, you you're looking at predecessor, and you just want to know, how much of the data structure do I have to read in order to be able to answer the predecessor problem? How much do I have to write out to do an insertion, or whatever?

And so in this model, computation is free. And this is kind of like the external memory model, and the cache oblivious models. There, we were measuring how many block reads and writes there are. Here, our blocks are actually our words. So there is a bit of a relation, except there's no real-- you can either think of there being no cache here, because you're just reading in a constant number of words, doing something, spitting stuff out.

Or in the cell probe model, you could imagine there being an infinite cache for this operation, but no cache from operation to operation. It's just, how much do I have to [INAUDIBLE] information, theoretically, to solve a particular predecessor problem? We'll deal with this a lot

in a couple of lectures-- not quite yet.

This model is just used for lower bounds. It's not a realistic model, because you have to pay for computation in the real world. But if you can prove that you need to read at least a certain number of words, then, of course, you have to do at least that many operations. So it's nice for lower bounds.

In general, we have this sort of hierarchy of models, where this is the most powerful, strongest, and below cell probe, we have transdichotomous RAM, then word RAM, then-- just to fit it in context, what we've been doing-- below that is pointer machine, and below that would be binary search tree. I've mentioned before, pointer machines are more powerful than binary search tree. And of course, we can implement a pointer machine on a word RAM.

So we have these relations. There are, of course, other models. But this is a quick picture of models we've seen so far.

So now, we have this notion of a word. In the predecessor problem, these elements are words. They're w-bit integers, universe-defined. And we want to be able to insert, delete, predecessor, and successor over words. So that's our challenge.

In the binary search tree model, we know the answer to this problem is theta log n. In general, any comparison-based data structure, you need theta log n, in the worst case. It's an easy lower bound.

But we're going to do better on these other models in the word RAM. So here are some results. First data structure is called Van Emde Boas. You might guess it is by van Emde Boas-- Peter. It actually has a couple other authors in some versions of the papers, which makes a little bit confusing. But for whatever reason, the data structure is just named Van Emde Boas.

And it achieves log w per operation. I think I'll rewrite this. This is log log u per operation. But it requires u space. So think of u space as being, like, for every item in the universe I store, yes or no, is it in the set?

So that's a lot of space, unless n and u are not too different. But we can do better. But the cool thing is the running time. This is really fast-- log log u.

If you think about, for example-- I don't know-- the universe being polynomial in n, or even if the universe is something like-- polynomial in n is the same as this-- 2 to the c log n. You can

go crazy and say log to the c power-- so, like, 2 to the log to the fifth power. All those things, you take log twice. Then log log u becomes theta log log n.

So as long as your word size is not insanely large, you're getting log log n performance. So in general, when, let's say, w is is polylog n, then we're getting this kind of performance. And I think on most computers, w is polylogarithmic.

We said it has to be at least log. It's also, generally, not so much bigger than log. So log squared is probably fine most of the time, unless you have a really small problem. OK, so cool.

But the space is giant. So how do we do better than that? Well, there's a couple of answers. One is that you can achieve log w with high probability, and order n space. With a slight tweak, basically, you combine Van Emde Boas plus hashing, and you get that.

I don't actually know what the reference is for this result. It's been an exercise in various courses, and so on. I can talk more about that later.

Then alternatively, there's another data structure, which, in many ways, is simpler. It really embraces hashing. It's called y-fast trees. It achieves the same bounds-- so log w with high probability and linear space. It's basically just a hash table with some cleverness.

So we'll get there. Even though it's simpler, we're going to start with this structure. Historically, this is the way it happened-- Van Emde Boas, then y-fast trees, which are by Willard. And it'll be kind of a nice finale.

There's another data structure I want to talk about, which is designed for the case when w is very large-- much bigger than polylog n. In that case, there's something called fusion trees. And you can achieve log base w of n-- and, I guess, with high probability and linear space.

The original fusion trees are static. And you could just do log base w of n deterministic queries. But there's a later version that dynamic, achieves this using hashing for updates, insertions, and deletions.

Cool. So this is an almost upside-down-- it's obviously always an improvement over just log base 2 of n. But it's sometimes better and sometimes worse than log w. In fact, it kind of makes sense to take the min of them. When w is small, you want to use log w. When w is big, you want to use log base w of n. They're going to balance out when w is 2 to the root log n-- something like that.

The easy thing is, when these balance out is when they're equal. And that will be when this is log n divided by log w. So when log w equals log n divided by log w-- let do that over here. log w is log n over log w. Then this is like saying log squared w equals log n, or log w is root log n.

So I was right. w is 2 to the root log in, which is a weird quantity. But the easy thing to think about is this one-- log w is root log n. And in that case, the running time you get is root log n.

So it's always, at most, this. And the worst case is when these things are balanced, or these two are the same, and they both achieve root log n. But if w is smaller or larger than this threshold, these structures will be even better than root log n. But in particular, it's a nice way to think about, oh, we're doing sort of a square factor better than binary search trees. And we can do this high probability in linear space.

So that's cool. Turns out it's also pretty much optimal. And that's not at all obvious, and wasn't known for many years. So there's a cell probe lower bound. So these are all in the word RAM model-- all these results. The first one actually kind of works in the pointer machine. I'll talk about that later.

This lower bound's a little bit messy to state. The bound is slightly more complicated than what we've seen. But I'm going to restrict to a special situation, which is, if you have n polylog n space.

So this is a lower bound on static predecessor. All you need to do is solve predecessor and successor, or even just predecessor. There's no inserts and deletes.

In that case, if you use lots of space, like u space, of course, you can do constant time for everything. You just store all the answers. But if you want space that's not much bigger than n-- in particular, if you wanted to be able to do updates in polylog, this is the most space you could ever hope to achieve.

So assuming that, which is pretty reasonable, there's a bound of the min of two things-- log base w of n, which is fusion trees, and, roughly, log w, which is Van Emde Boas. But it's slightly smaller than that. Yeah, pretty weird.

Let me tell you the consequences-- a little easier to think about. Van Emde Boas is going to be optimal for the kind of cases we care about, which is when w is polylog n. And fusion trees are optimal when w is big. Square root log n log log n.

OK-- a little messy. So there's this divided by log of log w over log n. If w is polylog n, then this is just order log log n. And so this cancels. This becomes constant. So in these situations, which are the ones I mentioned over here, w is polylog n, which is when we get log log n performance. And that's kind of the case we care about. Van Emde Boas is the best thing to do. Turns out, this is actually the right answer. You can do slightly better. It's almost an exercise. You can tweak Van Emde Boas and get this slight improvement. But most word sizes, it really doesn't matter. You're not saving much. Cool.

So other than that little factor, these are the right answers. You have to know about Van Emde Boas. You have to know about fusion trees. And so this lecture is about Van Emde Boas. Next lecture is about fusion trees. This result is from 2006 and 2007, so pretty recent.

So let's start a Van Emde Boas. Yeah. Let's dive into it. I'll talk about history a little later. The central idea, I guess, if you wanted to sum up Van Emde Boas in an equation, which is something we very rarely get to do in algorithms, is to think about this recurrence-- T of u is T of square root of u plus order 1. What does this solve to? log log u.

All right, just think of taking logs. This is the same as T of w equals T of w over 2 plus order 1. w is the word size. And so this is log w. It's the same thing.

If we could achieve this recurrence, then-- boom-- we get our bound of log w. So how do we do it. We split the universe into root u clusters, each of size root u. OK, so, if here is our universe, then I just split every square root of u items.

So each of these is root u long. The number of them is square root of u. And then somehow, I want to recurse on each of these clusters. And I only get to recurse on one of them-- so a pretty simple idea.

Yeah. So I'll talk about how to actually do that recursion in a moment. Before I get there, I want to define a sort of hierarchical coordinate system. This is a new way of phrasing it for me. So I hope you like it.

If we have a word x, I want to write it as two coordinates-- c and i. I'm going to use angle brackets, so it doesn't get too confusing. c is which cluster you're in. So this is cluster 0, cluster 1, cluster 2, cluster three. i is your index within the cluster. So this is 0, 1, 2, 3, 4, 5-- up to root u minus 1 within this cluster. Then 0, 1, 2, 3, 4, 5 up to root u minus 1 with in this cluster-- so the i is your index within the cluster, like this, and c is which cluster you are in. OK. Pretty

simple. And there's easy arithmetic to do this.

c is x integer divide root u. And i is x integer mod root u. I used Python notation here. So fine, I think you all know this-- pretty simple. And if I gave you c and i, you could reconstruct x by just saying, oh, well, that's c times root u plus i. So in constant time, you can decompose a number into its two coordinates. That's the point.

In fact, it's much easier than this. You don't even have to do division if you think of everything in binary, which computers tend to do. So the binary perspective is that x is a word. So it's a bunch of bits. 0, 1, 1, 0, 1, 0, 0, 1-- whatever. Divide that bit sequence in half, and then this part is c, this part is i.

And if you assume that w is a power of 2, these two are identical. If they're not a power of 2, you've got to round a little bit here. It doesn't matter. But you can use this definition instead of this one either way.

So in this case, c is-- ooh, boy-- x shifted right, w over 2, basically. So this w over 2-- w over 2. The whole thing is w bits. So if I shift right, I get rid of the low order bits, if I want. i is slightly more annoying. But I can't do it as an and with one shifted left w over 2 minus 1. That's probably how you do it in c.

I don't know if you're used to this. But if I take it a 1 bit, I shift it over to here, and I subtract 1. Then I get a whole bunch of 1 bits. And then you mask with that bit pattern. So I'm masking with 1, 1, 1, 1. Then I'll just get the low order bits.

Computers do the super fast-- way faster than integer division. Because this is just like routing bits around. So this is easy to do on a typical CPU. And this will be much faster than this code, even though looks like more operations, typically.

All right. So fine. The point is, I can decompos x into c and i. Of course, I can also do the reverse. This would be c shifted left w over 2, ord with i. It's a slight diversion.

Now, I can tell you the actual recursion, and then talk about how to maintain it. So we're going to define a recursive Van Emde Boas structure of size u and word size w. And what it's going to look like is, we have a bunch of clusters, each of size square root of u.

So this represents the first root u items. This represents the next root u items. This represents the last root u items, and so on. So that's the obvious recursion from this. So this is going to be

a Van Emde Boas structure of size root u. And then we also have a structure up top, which is called the summary structure. And the idea is, it represents, for each of these clusters, is the cluster empty or not? Does this cluster have any items in it? Yes or no.

If yes, then the name of this cluster is in the summary structure. So notice, by this hierarchical decomposition, the cluster number and the index are valid names of items within these substructures. And basically we're going to use the i part to talk about things within the clusters. And we're going to use the c part to talk about things within the summary structure. They're both numbers between 0 and root u minus 1. And so we get this perspective.

All right. So formally, or some notation, cluster i-- so we're going to have an array of clusters. It is Van Emde Boas thing of size square root u, and word size w over 2. This is slightly weird, because the machine, of course, its word size remains w. It doesn't get smaller as you recurse. We're not going to try to spread the parallelism around or whatever.

But this is just a notational convenience. I want to say the word size conceptually goes down to w over 2, so that this definition still makes sense. Because as I look at a smaller part of the word, in order to divide it in half, I have to shift right by a smaller amount. So that's the w that I'm passing into the structure.

OK, and then v dot summary is same thing. It's also Van Emde Boa's thing of size root u. Then the one other clever idea, which makes all of this work, is that we store the minimum element in v dot min. And we do not store it recursively.

So there's also one item here, size 1, which is the min. It's just stored off to the side. It doesn't live in these structures. Every other item lives down here. And furthermore, if one of these is not empty, there's also a corresponding item up here. This turns out to be crucial to make a Van Emde Boas work. And then v dot max, we also need-- but it can be stored recursively. So just think of it as a copy of whatever the maximum element is.

OK, so in constant time, we can compute the min and compute the max. That's good. But then I claim also in log w time-- log log u time-- we can do insert, delete, predecessor, successor. So let's do that.

This data structure-- the solution is both simple and a little bit subtle. And so this will be one of the few times I'm going to write explicit pseudocode-- say exactly how to maintain this data structure. It's short code, which is good. Each algorithm is only a few lines. But every line

matters. So I want to write them down so I can talk about them. And with this new hierarchical notation, I think it's even easier to write these down. Let's see how I do.

OK, so we'll start with the successor code. Predecessor is, of course, metric. And it basically has two cases. There's a special case in the beginning, which is, if the thing you're querying happens to be less than the minimum of the whole thing, then of course, the minimum is the successor.

This has to be done specially, because the min is not stored recursively. And so you've got to check for the min every single level of the recursion. But that's just constant time. No big deal.

Then the interesting things is, we have recursions in both sides-- in both cases-- but only one. The key is, we want this recurrence-- T of u is 1 times T of root u plus order 1. That gives us log log u. If there was a 2 here, we would get log u, which is no good. We want the one.

So in one case, we call successor on a cluster. In the other case, we call successor on the summary structure. But we don't want to do both. So let's just think about, intuitively, what's going on.

We've got this-- I guess I can do it in the same picture. We've got this summary and a bunch of clusters. And let's say you want to compute, what's the successor of this item? So via this transformation, we compute which cluster it lives in and where it is within the cluster. That's i. So it's some item here.

Now, it could be the successor is inside the same cluster. Maybe there's an item right there. Then want to recurse in here. Or it could be, it's in some future cluster.

Let's do the first case. If, basically, we are less than the max of our own cluster, that means that the answer is in there. Figure out what the max is in this structure-- the rightmost item in s that's inside this cluster c. This is c.

If our index is less than the max's index, then if we recurse in here, we will find an answer. If we're bigger than the max, then we won't find an answer down here. We have to recurse somewhere else. So that's what we do.

If we're less than the max, then we just recursively find the successor of our index within cluster c. And we have to add on the c in front. Because successor within this cluster will only give an index within the cluster. And we have to prepend this c part to give a global name. OK,

so that's case 1. Very easy.

The other case is where we're slightly clever, in some sense. We say, OK, well, if there's no successor within the cluster, maybe it's in the next cluster. Of course, that one might be empty, in which case, it's in the next cluster. But that one might be empty, so look at the next cluster. We need to find, what is the next non-empty cluster? For that, we use the summary structure.

So we go up to position c here. We say, OK, what is the next non-empty structure after c? Because we know that's going to be where our answer lives for successor. So that's going to give us, basically, a pointer to one of these structures-- c prime, which-- all these guys are empty. And so there's no successor in there. The successor is then the min in this structure.

So that's all we do. Compute the successor of c in the summary structure. And then, in that cluster, c prime, find the min, which takes constant time, and then prepend c prime to that to get a global name. And that's our successor. Yeah, question.

AUDIENCE: Could you repeat why min is not recursive? Because looking at this, it looks like all these smaller [INAUDIBLE] trees have [INAUDIBLE]

ERIK DEMAINE: Ah, OK. Sorry. The question is, why is the minimum not recursive? The answer to that question is not yet clear. It will have to do with insertion. But I think what exactly this means, I maybe didn't state carefully enough.

Every Van Emde Boas structure has a min-- stores a min. In that sense, this is done-- that's funny-- not so recursively. But every one stores it. The point is that this item doesn't get put into one of these clusters recursively-- just the item.

But each of these has its own min, which is then not stored at the next level down. And each of those has its own min, which is not stored at the next level down. Think of this as kind of like a little buffer.

The first time I insert it into the structure, I just stick it into the min. I don't touch anything else. You'll see when we get to the insertion algorithm. But it sort of slows things down from trickling.

AUDIENCE: So putting that min, is that what prevents from--

ERIK DEMAINE: That will prevent the insertion from doing two recursions instead of one. So we'll see that in a moment. At this point, just successor is very clear. This would work whether the min is stored

recursively or not. But we need to know what the min is of every structure, and we need to know the max of every structure. At this point, you could just say that min and max could be copies-- no big deal-- and we'd be happy. And of course, predecessor does the same thing.

So the slight cleverness here is that we use the min here. This could have been a successor operation with minus infinity as the query. But that would be two recursions. We can only afford one. Fortunately, it's the min item that we need. So we're done with successor.

That was the easy case-- or the easy one. Insert is slightly harder. Delete is just slightly messier. It's basically the same as insert. So insert-- let me write the code again.

Insertion also has two main cases. There's this case, and the other case. But there's no else here. This happens in both cases. And then there's some just annoying little details at the beginning.

Just like over here, we had to check for the min specially, here, we've got to update the min and max. And there's a special case, which I haven't mentioned yet. v dot min-- special case is, it will be this value, none, if the whole structure is empty. So this is the obvious way to tell whether a structure is empty and has no min. Because if there's any items in there, there's going to be one in the min slot.

So first thing we do is check, is our structure empty? If it's empty, the min and the max become the inserted item. We're done. So that's the easy case.

We do not store it recursively in here. That's what this means. This element does not get stored in any of the clusters.

If it's not the very first item, or it's not the min item, then we're going to recursively insert it into a cluster. So if we have x in cluster c, we always insert index i into cluster c, except if it's the min.

Now, it could be where a structure is non-empty. There is a min item there. But we are less than the min. In that case, we're the new min, and we just swap those. And now, we have to recursively insert the old min into the rest of the structure. So that's a simple case. Then we also have to update v dot max, just in the obvious way. This is the easy way to maintain v dot max in variant, that is the maximum item.

OK, now we have the two cases. I mean, this is really the obvious thing to get to do insertion.

We have to update the summary structure, meaning, if the cluster that we are inserting into-- cluster c-- is empty, that means it was not yet in the summary structure. We need to put it in there.

So we just insert c into v dot summary-- pretty obvious. And in all cases, we insert our item into cluster c. This looks bad, however, because there's two recursions in some cases. If this if doesn't hold, it's one recursion. Everything's fine.

So if the cluster was already in use, great. This is one recursion. This is constant work. We're done. The worry is, if the cluster was empty before, then this insertion is a whole recursion. That's scary, because we can't afford a second recursion.

But it's all OK. Because if we do this recursion, that means that this cluster was empty, which means, in this recursion, we fall into this very first case. That structure, it's min is none. That's what we just checked for.

If it's none, we do constant work and stop. So everything's OK. If we recursed in the summary structure, this recursion will be a shallow recursion. It just does one thing. You could actually put this code into this if case, and make this an else case. That's another way to write the code. But this will be a very short recursion. So either you just do this recursion, which could be expensive, or you just do this one, in which case, we know this one was cheap.

If this happens, we know this will take constant time. So in both cases, we get this recursion-- square root of u plus constant. And so we get log log u insertion.

Do you want to see delete? I mean, it's basically the same thing. It's in the notes. I mean, you do the obvious thing, which is, you delete in the cluster. And then if it became empty, you also have to delete in the summary structure. So there's, again, a chance that you do two recursions. But-- OK, I'm talking about it. Maybe I'll write a little bit of the code.

I think I won't write all the code, though-- just the main stuff. So if we want to delete, then basically, we delete in cluster c, index i. And then if the cluster has become empty as a result of that, then we have to delete cluster c from the summary structure, so that our predecessor and successor queries actually still work.

OK, so that's the bulk of the code. I mean, that's where the action happens. And the worry would be, in this if case, we're doing two recursive deletes. The claim is, if we do this second delete, which is potentially expensive-- this one was really cheap-- the claim is that emptying a

Van Emde Boas structure takes constant time-- like, if you're deleting the last element.

Why? Because when you're deleting the last element, it's in the min right here. Everything below it-- all the recursive structures-- will be empty if there's only one item, because it will be right here. And you can check that from the insertion.

If it was empty, all we did was change v dot min and v dot max. So the inverse, which I want right here, is just to clear out v dot min and v dot max. So if this ends up happening, this only took constant time. You don't have to recurse when you're deleting the last item.

So in either case, you're really only doing one deep recursion. So you get the same recurrence, and you get log log u. So for the details, check out the notes.

I want to go to other perspectives of Van Emde Boas. This is one way to think about it. And amusingly, and this is probably the most taut way to do Van Emde Boas. It's, in CLRS, described this way, because in 2001, when I first came here, I presented Van Emde Boas like this in an undergrad algorithms class with more details. You guys are grads, so I did it like three times faster than I would in 6046.

So now, it's in textbooks and whatnot. But this is not how Van Emde Boas presented this data structure-- just out of historical interest. This is a way that I believe was invented by Michael Bender and Martin Farach-Colton, who are the co-authors on "Cache-oblivious B-trees." And around 2001, they were looking at lots of old data structures and simplifying them. And I think this is a very clean, simple way to think about Van Emde Boas.

But I want to tell you the other way, which is the way it originally appeared in their papers. There's actually three papers by van Emde Boas about this structure. Many papers appear twice-- once in a conference, once in a journal-- this one, there's three relevant papers. There's conference version, journal version. The only weird thing there is that the conference version has one author-- van Emde Boas. The journal version has three authors-- van Emde Boas, Kaas, and Zijlstra.

And they're acknowledged in the conference version, so I guess they helped even more. In particular, they, I think, implemented this data structure for the first time. It's a really easy data structure to implement, and very fast.

Then there's a third paper by van Emde Boas only in a journal which improves the space a

little bit. So we'll see a little bit what that's about. But what I like about both of these papers is they offer a simpler way to get log log u, successor, predecessor. Let's not worry about insertions and deletions for a little bit, and take what I'll call the simple tree view.

So I'm going to draw a picture-- 0, 1, 0, 0, 0, 0, 0-- OK. This is what we call a bit vector, meaning, here's item zero, item one, item two. And here is u minus 1. And I'll put a 1 if that element is in my set, and a 0 otherwise. OK, so one is in the set, nine-- I think-- is in the set, 10, and 15 are in the set.

I kind of want to maintain this. This is, of course, easy to maintain by insertions and deletions. I just flip a bit on or off. But I want to be able to do successor queries. And if I want the successor of, say, this 0, finding the next 1-- I don't want to have to walk down. That would take order u time-- very bad. So obvious thing to do is build a tree on this thing.

And I'm going to put in here the or of the two children. Every node will store the or of its children. And then keep building the tree.

Now we have a binary tree, with bits on the vertices. And I claim, if I want to compute the successor of this item, I can do it in a pretty natural way in the log log u time. So keep in mind, this height here is w-- log u. So I need to achieve log w.

So of course, you could try just walking down this tree, or walking up and then back down. That would take order w time. That's the obvious BST approach. I want to do log w.

So how do I do it? I'm going to binary search on the height. How could I binary search on the height?

Well, what I'd really like to do, in some sense-- if I look at the path of this node to the route-- where is my red chalk? So here's the path to the root. These bits are saying, is there anybody down here? That's what the or gives you. So it's like the summary structure.

If I want to search for this guy-- well, if I walked up, eventually, I find a 1. And that's when I find the first nearby element. Now, in this case it's not the successor I find. It's really the predecessor I found. When you get to the first one-- the transition from 0 to 1-- you look at your sibling-- the other child of that one. And down in this subtree, there will be either the predecessor or the successor. In this case, we've got the predecessor, because it was to the left. We take the max element in there, and that's the predecessor of this item.

If instead, we had found this was our first one, then we look over here, take the min-- there's, of course, nothing here. But in that situation, the min over there would be our successor. So we can't guarantee which one we find. But we will find either the predecessor or the successor if we could find the first transition from 0 to 1. And we can do that via binary search, because this string is monotone. It's a whole bunch of zeros for awhile, and then once you get a 1, it's going to continue to be 1, because those are or. That one will propagate up.

So this is the new idea to get log log u, predecessor, successor is to-- let's say-- any root-to-leaf path is monotone. It's 0 for awhile, and then it becomes 1 forever. So we should be able to binary search for the 0 to 1 transition. And it either looks like this, or it looks like this.

So our query was somewhere down here in the 0 part. I'm assuming that our query is not a 1. Otherwise, it's an immediate 0 to 1 transition. And that's a special case. It's easy to deal with.

And then there's the other tree-- the sibling of x-- the other child of the 1. And in this case, we want to take the min. And that will give us our successor of x. And in this case, we want to take the max over here, and that will give us the predecessor of x.

So as long as we have minimax of subtrees, this is constant time. We find either the predecessor or the successor. Now, how do we get the other one? Pretty easy. Just store a linked list of all the items, in order.

So I'm going to store a pointer from this one to this one, and vice versa-- and this one or this one. This is actually really easy to maintain. Because when you insert, if you can compute the predecessor and the successor, you can just stick it in the linked list. That's really easy. We know how to do that in constant time.

So once you do this, it's enough to find one of them, as long as you know which one it is. Because then you just follow a pointer-- either a forward or a backward pointer-- and you get the other one. So whichever one you wanted-- you find both the predecessor and successor at the cost of finding either one. So that's a cute little trick.

This is hard to maintain, dynamically, at the moment. But this is, I think, where the Van Emde Boas structure came from. It's nice to think about it in the tree view.

So we get log log you u, predecessor, and successor. I should say what this relies on is the ability to binary search on any route-to-node path. Now, there aren't enough pointers to do that.

So you have a choice. Either you realize, oh, this is a bunch of bits in a complete binary tree, so I can store them sequentially in array. And given a particular node position in that array, I can compute, what is the second ancestor, or the fourth ancestor or whatever, in constant time. I just do some arithmetic and I can compute from here where to go to there. It's like the regular old heaps, but a little bit embellished, because you have to divide by a larger power of two, not just one of them.

So that's one way to do it. So in a RAM, that all works fine. When van Emde Boas wrote this paper, though, the RAM didn't-- it kind of existed. It just wasn't as well-developed then. And the hot thing at the time was the pointer machine, or I guess at that point, they called it the Pascal machine, more or less. Pascal does have arrays. And the funny thing is, Van Emde Boas does use arrays, but mostly it's pointers. And you can get rid of the arrays from their structure.

And essentially, in the end, Van Emde Boas, as presented like this, is in a pointer machine. Let me tell you a little bit about that.

So original Van Emde Boas, which I'll call stratified trees-- that's what he called it-- is basically this tree structure with a lot more pointers. So in particular, each leaf-- or every node, actually, let's say-- stores a pointer to 2 to the ith ancestor, where i is 0, 1, up to log w. Because it was the 2 to the-- here.

So once you get the ancestor immediately above me, two steps above me, four steps above me, eight steps above me, that's what I really need to do this binary search. The first thing I need is halfway up. And then if I have to go down, I'm going to need a quarter of the way up. And if I have to go down, I want an eighth of the way up.

Whenever I go up, from-- if I decide, oh, this is a 0. I've got to go above here. Then I do the same thing from here. I want to go halfway up from here-- from this node. So as long as every node knows how to go up by any power of 2, we're golden. We can do a binary search.

The trouble with this is, it increases space. This is u log w space, which is a little bit bigger than u. And the original van Emde Boas paper, conference and journal version, achieves this bound-- not u. Little historical fun fact-- not terribly well known. Cool. So that's stratified trees. Anything else? All right. Stratified tree. Right.

At this point, we have fast search, but slow update let. Me tell you about updates in a second. Yeah, question.

AUDIENCE: So once you do binary search to find the first 1, how do you walk back down the tree--

ERIK DEMAINE: Oh, I didn't mention, but also, every node stores min and max. So that lets me do the teleportation back down. Every node knows the min and the max of its subtree.

Right. One more thing I was forgetting here-- when I say, this a lot of pointers to store. You can't store them all in one node. And in the van Emde Boas paper, it's stored in an array. But it doesn't really need that its an array. It could just as well be a linked list. And that's how you get pointer machine.

So this could be linked list. And then this whole thing works in pointer machine, which is kind of neat. And it's a little weird, because if you used a comparison pointer machine, where all you can do is compare items, there's a lower bound of log n, because you only have branching factor constant.

But here, the formulation of the problem is, when I say, give me the successor of this, I actually give you a pointer to this item. And then from there, you can do all this jumping around, and find your predecessor or successor. So in this world, you need at least u space, even to be able to specify the input. So that's kind of a limitation of the pointer machine.

And you can actually show in the pointer machine log, log u is optimal for any predecessor data structure in the pointer machine. So there's a matching lower bound log log u in this model. And you need to use space. So it's not very exciting.

What we like is the word RAM. There, we can reduce space to n. And that's what I want to do next, I believe-- almost next. One more mention-- actual stratified trees-- here, we got query fast, update slow. Stratified trees actually do update fast, as well. Essentially, it's this idea, plus you don't recursively store the min, which, of course, makes all these bits no longer accurate, as it gets much messier.

But in the end, it's doing exactly the same thing as this recursion. In fact, you can draw the picture. It is this part up here-- the top half of the tree-- this is summary. And each of these bottom halves is a cluster. And there's root u clusters down here. So those are smaller structures. And there's one root u sized Van Emde Boas structure, which is a summary structure.

These bits here is the bit vector representation of the summary structures. It's, is there anyone in this cluster? Is there anyone in this cluster, and so on? This, of course, also looks a lot like the Van Emde Boas layout. Take a binary tree, cut it in half, do the top, recursively do the bottom. So that's why it was called the Van Emde Boas layout, is this picture.

But if you take this tree structure, and then you don't recursively store mins, and then the bits are not quite accurate, it's messy. And so stratified trees-- you should try to read the original paper. It's a mess. Whereas this code-- pretty clean.

And so once you say, oh, I'm just going to store all these clusters as an array and not worry about keeping track of the tree, it actually gets a lot easier. And that was the Bender/Farach-Colton cleaning up, which never appeared in print. But it's appeared in the lecture notes all over the place-- and now CLRS. Cool.

I want to tell you about two more things. It's actually going to get easier the more time we spend with this data structure. All right. Let me draw a box.

At this point, we've seen a clean way to get Van Emde Boas. And we've seen a cute way in a tree to get search fast, but update slow. I want to talk a little more about that. Let's suppose I have this data structure. It's achieves log w query, which is fast, but it only achieves w update, which is slow.

How do you update the structure? You update one bit at the bottom, and then you've got to update all the bits up the path. So you spend w time to do an update over here.

If updates are slow, I just want to do less updates. We have a trick for doing this, which is, you put little things down here of size theta w. And then only one item from here gets promoted into the top structure. We only end up having n over w items up here, and about 1 over w as many updates.

If I want to do an insertion, I do a search here to figure out which of these little-- I'll call these "chunks--" which little chunk it belongs in. I do an insert there. If that structure gets too big-- it's bigger than, say, 2 times w, or 4 times w, whatever-- then I'll split it. And if I delete from something, and it gets too small, I'll merge with the neighbor, or maybe re-split-- just like B-trees. We've done this many times, by now.

But only when it splits, or I do a merge, do I have to do an update up here. Only when the set

of chunks changes do I need to do a single insertion or deletion up here-- or a constant number. So this update time goes down by a factor of w.

But I have to pay whatever the update cost is here. So what I do with this data structure? I don't want use Van Emde Boas, because this could be a very big universe. Who knows what? I use the binary search tree. Here, I can afford a binary search tree, because then it's only log w. log w is the bound we're trying to get. So you can do these binary search trees. It's trivial. Just do insert, delete, search. Everything will be log w.

So if I want to do a search, I search through here, which, conveniently, is already fast-- log w-- and then I do a search through here, which is also log w. So it's nice and balanced. Everything's log w.

If I want to do an insertion, I do an insertion here. If it splits, I do an insertion here. But that order w update cost, I charge to the order w updates I would have had to do in this chunk before it got split. So this our good friend indirection, a technique we will use over and over in this class. It's very helpful when you're almost at the right bound.

And that's actually in the follow-up van Emde Boas paper. A similar indirection trick is in there. So we can charge the order w update in top to-- that's the cost of the update-- to the order w updates that have actually been performed in the bottom. Because when somebody gets split, it's nice in its average state-- or when it gets merged, it's going to be close to its average state. You have to do a lot of insertions or deletions to get it out of whack, and cause a split or a merge.

So-- boom. This means the updates become log w. Searches are also log w. So we've got Van Emde Boas again, in a new way. Bonus points-- if you take this structure-- even this structure, if we did it in the array form-- great. It was order u space. If we did it with all these pointers, and we wanted a pointer machine data structure, we needed u log w space.

But this indirection trick, you can also get rid of the log w in space factor. It's a little less obvious. But you take this-- here, we reduced n by a factor of w. You can also reduce u by a factor of w. I'll just wave my hands. That's possible.

So u gets a little bit smaller. And so when we pay u log w space, if you got smaller by a factor of w, this basically disappears. So you get, at most, order u space.

But order u is not order n. I want order n space, darn it. So let's reduce space.

As I said, this is going to get easier and easier. By the end, we will have very little of a data structure. But still, we'll have log log u. And you thought this was easy, but wait, there's more.

Right now, we have two ways to get log log u-- query and order u space. There's the one I'm erasing, and there's this-- take this tree structure with the very simple pointers. Add indirection.

So admittedly, it's more complicated to implement. But conceptually, it's super simple. It's like, do this obvious tree binary search on the level thing. And then add indirection, and it fixes all your bounds, magically.

So conceptually, very simple-- practically, you definitely want to do this-- much simpler. Now, what about saving space? Very simple idea-- which, I think, again, comes from Michael Bender and Martin Farach-Colton. Don't store empty structures.

So in this picture, we had an array of all the clusters. But a cluster could be entirely empty, like this one-- this entirely empty cluster. Don't store it. It's a waste.

If you store them all, you're going to spend order u space. If you don't store them all-- just don't store the empty ones-- I claim your order n space. Done.

So I'm going back to the structure I erased. Ignore the tree perspective for awhile. Don't store empty clusters. OK, now, this sounds easy. But in reality, it's a little bit more annoying. Because we wanted to have an array of clusters. So we could quickly find the cluster.

If you store an array, you're going to spend at least square rot of u space. Because at the very beginning, you say, here are my root u clusters. Now, some of them might be null pointers. But I can't afford to store that entire array of clusters. So don't use an array. Use a perfect hash table.

So v dot cluster, instead of being an array, is now, let's say, a dynamic perfect hashing. And I'm going to use the version which I did not present. The version I presented, which used universal hashing, was order 1 expected. But I said that it can be constant with high probability per operation. It's a little bit stronger.

So now, everything's fine. If I do an index v dot cluster c, that's still constant time, with high probability now. And I claim this structure is now order n's space. Why is it order n's space? By

simple amortization-- charge each table entry in that hash table to the min of the cluster.

We're only storing non-empty ones. So if one of these guys exists in the hash table, we had to store a pointer to it, then that means the summary structure is non-zero. It means this guy is not empty. So it has an item in its min. Charge the space up here to store the pointer to that min guy.

Then each item-- each min item-- only gets charged once. Because it only has one parent that has a pointer to it. So you only charge once. And therefore-- charge and table entry-- only charge each element once.

And that's all your space. So it's order n space. Done. Kind of crazy.

I guess, if you want, there's also the pointer to the summary structure. You could charge that to your own min. And then you're charging twice. But it's constant per item.

So this is kind of funny. Again, it doesn't appear in print anywhere, except maybe as an exercise in CLRS now. But you get linear order n space, just by adding hashing in the obvious way.

Now, for whatever reason, Willard didn't see this, or wanted to do his own thing, and so he found another way to do order n space log log u query with hashing. Well, I guess, also, you had to think of it in this simple form. It's harder to do this in the tree. It can be done, I think. But this is a simpler view than the tree, I think. And then boom-- order n space.

But it turns out there's another way to do it. This is a completely different way to do Van Emde Boas-- actually, not that completely different. It's another way to do this with hashing. And we're going to start with what's called x-fast trees, and then we will modify it to get y-fast trees. That's Willard's terminology.

OK, so x-fast trees is, store this tree, but don't store the zeros. So don't store zeros. Only store the ones in the-- we call this the simple tree view. This is why I, in particular, wanted to tell you about the simple tree view, because it is really what x fast trees do.

So what do I mean by only store the ones? Well, each of these ones has sort of a name. What is the name of this item? Its name is one-- or in other words, 0, 0, 0, 1. Each of these nodes, you can think of, what is the path to get here?

Like, the path to get to this one is 1, 0, 0. 1 means right. 0 means left.

Those names give you the binary indicator of where that node is in the tree, in some sense. So store the ones as binary strings in a hash table-- again, a dynamic perfect hash table. Let's say I can get constant with high probability.

OK. And if you're a little concerned-- so what this means-- the ones are exactly the prefixes of the paths to each of the items. This was item one. And so I want to store this one, which is empty string, this one, which is 0, this one, which is 00, this one, which is 000, this one, which is 0001.

So I take 0001, which is the item I want to store. And there's all these prefixes, which are the items I want to store. And for this really to make sense, you also need the length of the string. Strings of different lengths should be in different worlds. So the way, actually, x-fast trees originally did it in the paper is, have a different hash table for strings of different lengths. So that's probably an easier way to think about it.

You store all the items themselves in a hash table. You store all the prefixes of all but the last bit in a separate hash table, all but the last two bits in a separate hash table, and so on. Now, what does this let you do? It lets you do this-- binary search for the 0 to 1 transition.

What we did here was-- I look at the bit, is it 0 or 1? Instead of doing that, you do a query into the hash table, and say, is it in the hash table? It's in the hash table if and only if it is one. So looking at a bit in this conceptual tree is the same thing as checking for containment in this hash table. But now, we don't have to store the zeros, which is cool.

We can now do search, predecessor or successor, fast, in log w time, via this old thing. Again, you have to have min and max pointers, as well. So in this hash table, you store the min and the max of your subtree. Or actually, from a 1, you actually need the max of the left subtree, and you need the min of the right subtree. But it's a constant amount of information per thing.

This is not perfect, however, in that it uses nw space. And also, updates are slow. It's order w updates.

But we're almost there. Because we have fast queries, slow updates, not optimal space. Take this. Add indirection-- done. And that's y-fast trees. y-fast trees-- you take x-fast trees, you add this indirection right here, and you get log w per operation order and space. Of course, this is a high probability because we're using hashing.

Because we have a factor w bad here, we have factor w bad here. You divide by w. You're done.

Up here, you have n over w space. n over w times w is n. Queries, just like before, remain log w. But now-- boom-- updates, we pay log w because of the binary search trees at the bottom, but pretty cool. Isn't that neat?

I've never seen this before. OK, I've seen x-fast trees and y-fast trees. But it's really just the same-- we're taking Van Emde Boas, looking at it in the tree view. You can see where Willard got this stuff. It's like, oh, man I really want to store all these bits, but hey, it's way too big. Just don't store the zeros. That means we should use a hash table. Ah , hash table just gives you whether the bit is in or out. Great.

Now use indirection. And indirection was already floating around as a concept at the time-- slightly different parameters. Van Emde Boas had his own indirection to reduce the space from u times log w to u. But Willard did it, and-- boom-- it got down to n space in this way.

But as you saw, you can also do it directly to Van Emde Boas. All these ideas can be interchanged. You can combine any data structure you want with any space saving trick you want, with indirection, if you need to, to speed things up and reduce space a little bit.

So there's many, many ways to do this. But in the end, you get log w per operation, and order n space. And that sort of result one. And it's probably the most useful predecessor data structure, in general. But next time, we'll see fusion trees, which are good for when w is huge.