ERIK DEMAINE: Today, we continue our theme on integer data structures. And we're going to cover one data structure called fusion trees, which introduces a bunch of cool concepts using, you might call bit tricks, using the word level parallelism to do lots of great things. To do those great things we need to do something called sketching, which is essentially taking w-bit integers and making them less than w-bits, but still having enough stuff that we care about. And then we can actually compute those sketches using integer multiplication in a very clever way. And given a bunch of these slightly less than w bit numbers we can compare several for the price of one operation as a parallel comparison. And then a particularly nifty thing, which is new this time I haven't covered it before, is how to compute the leftmost 1 bit in a w-bit integer in constant time, all very cool.

And so we're going to combine all these techniques and get fusion trees. What do fusion trees give us in the end? Basically, the goal is to get log base w of n, predecessor and successor. So we're again talking about the predecessor problem. We did log of w last time with van Emde Boas and y-fast tries. So then with the two of them together, we get log w and log base w of n. The min of those two things is always at most root log n, but sometimes much better than that.

So fusion trees are great when w is big, van Emde Boas was good when w was smallish. Like poly log n, van Emde Boas is optimal. Here, we're thinking about w being closer to n, or maybe n to the epsilon or something. Then we get constant time, if it's n to the epsilon.

Anyway, the version we're going to cover is static. And it's linear space, and it runs on the good old word RAM, which you may recall is regular C operations on w-bit words. w is at least log n, so you can do random access. And anything else? I think that's the version we will cover. And that is the original version of Fredman and Willard, and it was invented in 1990, which was one year after the cold fusion debacle. So this may be where this name came from. There's other reasons it might be called fusion trees, which we'll get to.

Cool. So now, there's other versions of fusion trees which I will not cover, but just so you know about them. And in some sense we will pseudo cover them. There's a version for an AC0 RAM. This is a model we haven't talked about. It's another version of a trans-dichotomous RAM, somewhere off to the side next to word RAM. AC0 is a circuit model. And it's basically

any constant depth circuit of unbounded fan in and fan out.

And so in particular what AC0 forbids are operations like multiplication. I think this used to be a bigger deal than it is today. I think multiplication used to be in order of magnitude slower than integer addition. I checked on current Intel architecture. It's about 3 times slower than an addition, because of pipelining a lot of things get cheaper. But in some theoretical sense, multiplication is worse than a lot of other operations, because you need to have a log and depth circuit. So you can't quite get as much parallelism as you can with integer addition. If you don't know about circuit models, don't worry about it too much. But AC0 means no multiplication, sort of simpler operations.

The weird thing about the AC0 RAM is it lets you do weird things, which are AC0, but are not in C. So you could look at the intersection of AC0 RAM and word RAM, and that would basically be word RAM without multiplication. But AC0 RAM allows other operations as long as there's some circuit that can compute them. Sometimes they're reasonable operations like most significant set bit is an AC0 operation. So, you don't have to do any work to get this operation. So, in some sense this makes your life easier. In other ways it makes your life harder, because there is no multiplication and we're going to use multiplication to compute sketches and other things.

So it's both better and worse. This is by Anderson and others a few years after the original fusion trees. More interesting is that there are-- so this is a version of fusion trees. More interesting are the dynamic versions of fusion trees. So there's one that the first version-- it's actually published later.

There's a general trick for dynamizing static data structures. We actually saw one. You may recall weight balanced search trees was a way to dynamize a static data structure. It turns out it doesn't work so great on fusion trees. Because the time to build a fusion tree is polynomial. It's not linear. And so that's kind of annoying. You need polynomial time to build this thing. So weight balance is not enough to slow down the rebuilds.

There's another structure, which we won't cover, called exponential search trees. It has this log log n overhead. But other than that, it gives you a nice time dynamization. So these are updates.

There's another version which uses hashing, and achieves log base w expected time for updates. This is by Raman. And so this gives you matching the fusion tree query bound, you

can do inserts and deletes, the same amount of time if you allow expected. And essentially the idea is to do sketches more like hashing. I mean, think of sketching as just hashing down to a smaller universe. And in expectation that will work well, although it's open, actually, whether you can achieve this bound with high probability. So it's an interesting open question.

So that's the various versions of fusion trees. But we're going to just cover the static ones, because they're interesting enough. Dynamic ones are not that much harder than the regular.

So how do we achieve log base w of n? Well we've already seen B-trees which achieve log base b of n. So presumably it's the same idea, and indeed that's what we'll do. What we want is a B-tree with branching factor ideally w. We can't quite achieve w though. So it's going to be w to some small constant power. And 1/5 is the one that I'll use in this lecture. You can improve it or make it worse, up to you. But any constant up here will do. Because then log base w to the 1/5 is going to be five times log base w of n.

So we have a node. So it has branching factor w to the 1/5. Then the height of the tree, of course, is theta log base w of n. So that's all good. But now what we need to do is, doing a predecessor search given a node, we need to decide in constant time which branch has our answer. So normally in a B-tree you would read in all these keys, and then compare your item to all of them, and then decide which way to go. Because in a B-tree we can read all of these items in one operation.

Now here, is that possible? Think about it. You've got each of these keys is w bits long. There's w to the 1/5 of them. So the total number of bits in the node to store all those keys is w to the 1 plus 1/5, which is a lot. There's no way you can read all those bits in constant time. You can only read order w bits in constant time. So we can't look at them all. Somehow, we still have to figure out which way to go correctly in constant time. So this is the idea of a fusion node.

We need to be able to store k, which is order w to the 1/5. Keys, I'm going to give them some names, x0 up to xk minus 1. Assume that they've been presorted. We can preprocess and do all those things. I'm going to go constant time, predecessor and successor, and it's going to be linear space, and it's going to require polynomial preprocessing. And this is the annoying part. If this was just k preprocessing, it would be easy to make fusion trees dynamic. But it's k to squared, or k cubed or something, depending on how fancy you are. It's not known how to do it in linear time.

So that's really our goal. If we can implement fusion tree nodes and do constant time predecessor on this small value of n basically, when n is only w to the 1/5. If we can do constant time for that, then by plugging in B-trees we get the log base w of n for arbitrary values of n. So it's really all about a single mode and doing that fast. So, the rest of the lecture will be about that.

So, I want to introduce this idea of sketching. And to do that I'm going to think about what it takes to distinguish this small number of keys. High level ideas, well we've got w to the 1/5 keys. Each of them is w bits. Do you really need all w bits for each of them? It seems a little excessive. If there's only w to the 1/5, you should only need about w to the 1/5 bits of each to distinguish them all. So that's the intuition.

And indeed, you can formalize that intuition by viewing each of the keys as a path in a binary tree. So this represents the bit string 01011101. 0 means left. 1 means right. This is a transformation we'll use a lot. So maybe that's in your set. Maybe this other bit string is in your set. Maybe this bit string-- oh, I've got to make these the same height which is a little challenging. So maybe those are the three. Suppose you just have these three bit strings, w bit strings in your set. So this is a tree of height w. Because each of the keys has w bits, so maybe k is 3. And those are your three keys.

OK. The idea is, look at the branching nodes. Where's a color? So, you've got a branching node here and a branching node here. Because there's three leaves, there's only going to be two branching nodes. So the idea is, well I really only care about these two bits. Or it would be enough to think about these two bits.

OK. Well, we'll look at this more formally in a moment. But by storing this bit, I know whether the key is over here on the left or over here on the right. And then by storing this bit, I don't really care about it for this word. But it will distinguish these two words. So if you just look at the bits that contain branching nodes in this tri-view, then it's enough to distinguish all of the xi's. So this is x0, x1, x2.

OK. Let me formalize that a little bit. So we have k minus 1 branching nodes in this height w tree of the k keys. Because there's k leaves, there's going to be k minus 1 branching nodes, because the k leaves are distinct. So this means there are at most, k minus 1 levels containing a branching node. It might be fewer.

Maybe it's nice to add in another key over here on the left. I mean if I was lucky, there'd be another key over here, and then I'd be using this bit and getting two for the price of one. If I'm less lucky, it will be more like this. So here's another x value. And in this case, I care about this branching node. So I care about another bit here.

OK. But if I have four keys, it will be at most three bits corresponding to these levels. So call these-- these levels correspond to bits. This is the first bit, second bit, third bit, and so on. This is the most significant bit, next, and then the least significant is at the bottom. So these levels correspond to important bits. That's the definition. And we're going to give these bits a name, b0, b1, up to br minus 1. Those are bit indices saying which bits we care about.

And we know that r is less than k, and k is order w to the 1/5. So there are only w to the 1/5 important bits overall among these k keys. So the idea is don't store all w bits for all the keys. I mean you have to store them. But don't look at them. Just look at these important bits for the keys. And then life is good. Because there's only w to the 1/5 bits per key. There's only w to the 1/5 keys. And so the total number of important bits among all k keys is small. It's only w to the 2/5, which is less than w. So it fits in a single word, and we can look at this in constant time.

So that seems like a good thing. Let me tell you what properties this has. Let me also define the notion of a perfect sketch of a word. x is going to be what you get when you extract bits b0 to br minus 1 from x. So in other words, this is a bit string, an r-bit string who's i-th bit equals bit bi of x. So you've got a bit string which is x. You say, oh, the important ones are this one, this one, this one, and this one. Inside here is either a 0 or a 1. And there's other bits which we don't care about, because they're not important. And we just compress this to a 4-bit string. 0110. OK. This is sketch of x.

And to be a little bit more explicit about how I'm labeling things, this is b0. This is b1, b2, and b3. Because you number bits-- I think this is right, we'll see later-- we're going to number bits from the right-hand side. This is 0-th bit, first bit, second, third, fourth; which is the opposite of this picture, unfortunately, sorry. This is a bit 0, bit 1, anyway. This will be convenient.

So, that's perfect sketch. For now, I'm going to assume that we can compute this in constant time. This one answer is it's an AC0 operation. That's not so obvious, but it's true. So on an AC0 RAM, you can just say, oh, this is an operation. Right? It's given one word. And, well OK. It's given these description of bit numbers, but those will also fit in one word. And then does

this bit extraction. We're going to see a reasonable way to do it. But for now, take that as an unreasonable way to do this.

So perfect sketch is good, because it implies the following nice property. If you look at the sketch of x0 that's going to be less than the sketch of x1, and so on, which is going to be less than the sketch of xk minus 1. Sketch preserves order. We assume that-- where do we have it? Over here, x0 is less than x1, is less than xk minus 1. And because we're keeping all the bits where are these xi's get distinguished, this one it doesn't matter whether we kept here, it doesn't matter whether we kept these guys. But in particular, we keep the bits that have all the branching nodes. That will preserve the order of the xi's.

So we know that the order of the xi's is preserved under sketching. The trouble is the following. Suppose you want to do a search, a predecessor search. So, you're given some query q, and you want to know where does q fit among the xi's. Because that will tell you which child to visit from here. So, OK. You compute a sketch of q, seems reasonable, and move into sketch world.

And now you try to find where sketch of q fits among these guys. So you can do that. And I claim you can do that in constant time. It's again, an AC0 operation. But the nice thing is the sketches all fit in one word. Also this single sketch fits in one word, no big surprise there. So let's say you can find where sketch of q fits among these items in constant time. The trouble is were the sketch of q fits is not necessarily the same as where q fits among the xi's. Because q was not involved in the definition of sketch. q is an arbitrary query. They come online, I mean any word could be a query, not just the xi's.

So you've set everything up to distinguish all the xi's. But q is going to fall off this tree at some point. And that kind of messes you up. Because if q fell off here, you don't have that bit. You won't notice the q fit there. So we have to do some work. And this is what I call de-sketchifying. And I like a big board.

OK, let mean draw a some more methodical and smaller example. I need to make it the right number of levels. A little bigger than my usual tree, and I'll get my red, actually maybe use two colors.

So here's a real example. Now it has four keys. And here I'm in the lucky case, where this is an important bit. And I get two for the price of one. I cared about this branching node, I cared

about this branching node, and so I only have to door two bits in my sketch for these four nodes. In general, it might be three bits. But this will just make the point. So it's actually, life is in some ways harder in this situation.

OK. So what are my bit strings here? Over here I've got 0000, which corresponds to always going left. And I've got 0010. Over here I've got 1100 and 1111. We drew these pictures for van Emde Boas, right? The idea is we're going to use some of the similar perspectives at least. OK, but the important bits were the very first, the leftmost bit I should say. And then two bits after that, so these guys. And so the sketch here is 11, 10, 01, and 00. And you can see this is the minimal number of bits I need to keep them in order. But it does. It works. You can check. This works in general.

OK. Now comes the query. I have a problematic query I'd like to draw. And it is 0101, so 0-1-0-1. So here's my query queue. Let me draw these as white. Query is 0101. 1 If we take the sketch, we get 00. Those are the important bits. So if I search for the query of 00, I will find that it happens to match this key, or it matches the sketch of this key. But that key is neither the predecessor, nor the successor of that query. So this is bad news. I find the predecessor in sketch world, which is the red stuff, I get the wrong answer. In general, they could be very far away from each other. Here, I've got it 1 away, but that's as big an example as I can draw.

So, how do we deal with this? This is the de-sketchification. So when I do this query, I end up finding this guy, x0. I claim that I can still use that for something interesting. OK, let's say we have a sketch of xi as the predecessor of the sketch of q. And so sketch of q is sandwiched between a sketch of xi and sketch of xi plus 1.

First of all, we're assuming that I can compute this in constant time, I can find where sketch of q fits among these guys. Because it just fits into words. And for now, let's just assume all operations on a constant number of words are at constant time. We will see how to do this. This is parallel comparison. So you figure out sketch of q fits here. I want to learn something about where q fits among the xi's. It's obviously, these may be the wrong answer. But I claim I can do something useful by looking at the longest common prefix of those words.

So I want to compare q, not sketch of q but the real value q, and either xi or xi plus 1. And what I want is the longest. So I look at the longest common prefix of q and xi. I look at longest common prefix of q and xi plus 1. Whichever of those is the longest that's my longest common prefix. In the tree, it's the longest common ancestor, or lowest common ancestor.

OK. So let's do it. We found that sketch of q fit between, I guess, these two guys, the way I've written it with the inequalities. It's between x0 here and x1. So in this case, the lowest common ancestor of this node and q is going to be here. Also this node and q happens to also be here. So this is the lowest we can go. And what this means is that these guys, they share the bit string up to here. We were on the blue substructure up till here. This was the node where q diverged. We followed a pointer here along a non-blue edge. That's where we made a mistake.

So this lets us find, in some sense, the first mistake, where we fell off the tree. So that's where we fell off the blue tree. That's useful information. Because now we know, well, we went to the right, whereas all the actual data is over here in the left subtree. There is no blue stuff in the right. So that tells us a lot. If we want to now find the predecessor of q, it's going to be whatever is the max in this subtree. So, I just need to be able to find the max over here.

So this is the idea. Now there's two cases, depending on whether we were in the right or in the left from that node. So let me write this, find the node y where q fell off the blue tree. So this node y, we can think of as a bit number. Here the leftmost bit was still on, but then the next bit was off. And so we look at-- I'll call that bit y plus 1, or maybe size of y plus 1. If that bit equals 1, that's the picture we have. Then what I'm going to do is set a new quantity e, which is going to be-- this is a new word. It's going to be the bit string y, followed by a 0, followed by lots of 1's.

Whereas our bit string q had a 1 here, and fell off the tree. What we're instead going to do is identify this node, the rightmost node in this subtree. That's not necessarily an xi. But it's a thing. And then we're going to do, again, this search and sketch space, but now using e instead of q.

If we do that, what is this node? Let's label it, 0011. If you look at the sketch bits, this has a sketch of 01. So if I did a search here, I would actually find that this is the answer, and that actually is a predecessor of q. In general, this is going to work well. Because essentially some of these bits are going to be sketch bits. This one was not, and we made a mistake there. We went right. We should have got left.

These ones, some of them are going to be sketch bits. Some of them are not. But whichever ones get underlined, it's going to be a 1, which means we're going to do the right thing. We want the very rightmost item in this tree. So if we always go right whenever there's a sketch

bit, and then do a search in sketch space, we will find the rightmost item in this tree. So if we then do a search on e, we're always going to get the right answer.

So in the end, we're going do two searches in sketch space, once with q to find this place where we fell off, then once with e where we actually find the right answer. And there's a symmetric case, which is if we went left and we should have gone right, then we go right, and then we put a lot of 0's. Because then we want to find the min in that tree.

So, back to search, we compute sketch of q. We find it among the sketch of the xi's. This gives us this y. So we find the longest common prefix, y equals longest common prefix of q and xi or xi plus 1. Then we compute e, and then we find sketch of e among sketch of xi's. And the claim is that the predecessor and successor of sketch of e among sketch of xi's equals the predecessor and successor of q, our actual query, among the xi's.

So this is a claim. It needs proof. But it's what I've been arguing that e gives us the right structure. It fixes all the sketch bits that are potentially wrong. We found the first sketch bit that was wrong. We fixed that one. And then the remainder, as long as we go all the way to the right, we'll find the max, or in the other case we want to go all the way to the left because we want to find the min. So that's this claim.

We find the predecessor of sketch of e. Run the sketch of the xi's, which is just this thing again. So again, we can do it in constant time. Then we find-- I mean I have to be a little bit more precise here. Of course, we find the predecessor and successor, we get a sketch of the xi. We have to undo that sketch operation. Really the way to think of it is predecessor and successor are really returning a rank. I want to know the i that matters. So if it fits between sketch of xi and sketch of xi plus 1, if sketch of e fits between those, then I know that q will fit between xi and xi plus 1, in terms of that rank, i, the index in the array.

So, that makes sense. This is the end of the-- what's the right way to put it? This is the big picture of fusion trees. At this point you should believe that everything works. And overall, what are we doing? We're building a w to the 1/5 tree. It's not yet clear why w to the 1/5. And so we have to implement these nodes that only have w to the 1/5 keys. So we're looking at a single node. And say, hey look, there's a bunch of keys. Let's just look at the important bits. That defines the sketch operation.

Now if we want to do a search, we do this double search. We compute the sketch, find the sketch among the sketches, find our mistake, compute our proper query, compute the sketch

of that, find that sketch among the sketches, and then that index in the array of sketches will be the correct index of our actual query q among the xi's.

Now, there are several things left to be done in a reasonable way. One is how do we compute sketches. How do we do this kind of operation of taking the bits we care about and bringing them all next to each other? Second thing is, how do we do this find? This parallel comparison. So it's basically all the bullets up here. We have how do we do a sketch, how do we do parallel comparison to find where one sketch fits among many sketches, and there's also a most significant set bit. Where did we do that? In computing the longest common prefix.

So if you have two bit strings and you want to know where did they first differ, the natural way to do that is compute the XOR, which gives you all the differing bits. And then find the first one bit from the left. So this is really most significant set bit. So, we need that operation. So, we have our work cut out for us. But the overall picture of fusion trees should now be clear. It just remains to do these three things. And this is where the engineering comes in, I would say. Any questions about the big picture?

So, the first thing I'm going to do is sketch. And as I've hinted at in the outline here, we're not going to do a perfect sketch. We're going to do an approximate sketch. This will probably be the most work among any of these operations. Parallel comparison is actually quite easy. Sketching is, I think, the biggest insight in Fusion trees. So, perfect sketch takes just the bits you care about that we need. We only want to look at the bits we care about.

But it's easy to look at the bits we care about. We can apply a mask, and just AND out the bits we care about. Everything else we can zero out. So that's easy. The hard part is compression, taking these four bits and making them four consecutive bits. But they don't really need to be consecutive. If I added in some 0's here in a consistent pattern that would still work. I'd still preserve the order among the sketches. And that's all I care about. And this is where I'm going to use the slop I have. Because right now I have w to the 1/5 keys. If I did perfect sketch, the total number of bits would only be w to the 2/5. But I can go up to w.

So what I'm going to do is basically spread out the bits, the important bits, in a predictable pattern of length w to the 4/5. Predictable just means it doesn't depend on what x is. So when there are extra 0's here, you know that's fine. But there's always going to be two 0's here, one 0 here, three 0's here, no matter what x was. As long as it's predictable, I'm going to preserve order. And as long as it's length order w to the 4/5, if I take w to the 1/5 of them, that will still fit

in a constant number of words. Because it will be order w bits total. So that's what I can afford. And now I'm going to do it.

So here's how. First thing, as I said, is we're going to mask the important bits. I just want the important bits. I should throw away all the others. And so this is going to be x prime equals x bit-wise AND. And here's where I'm going to use the notation that the bits count from the right. I want the bi-th bit to correspond to the value 2 to the bi. This thing is just a bit string. It has 1's wherever the important bits are. So if this is the b0, b1, b2, and b3; I just want this bit string. I mean, you can think of this as 1 shifted left bi times. So I get 1's in exactly the positions I care about. And if I bit-wise AND that with x it zeros out all the other bits. This is what we call masking.

So that's the obvious thing to do. And then the second idea is multiplication. And it's just like, well, maybe we could do it with a multiply, and then we'll just work it out. And the answer is yes, you can do it with a multiply. So that I imagine was the big insight was to see that multiplication is a very powerful operation. So we're just going to do x prime times some number m. And we're going to prove that there exists a number m that does what we need. So I'm going to write this out a little bit algebraically. So we can think about what m might be.

Now x prime only has the important bits. So we can write that as a sum i equals 0 to r minus 1 of xbi 2 to the bi. So I am introducing some notation here. xbi, is that important bit bi 1 or 0? This is just a de-reference of the bit vector or a bit string. And so you multiply that by that position. I mean this the definition of binary notation, right? But we only care about the important bits. Because only those are set. So that's x prime.

And then we're multiplying that by m. Now m could have any bit set. So I'm going to-- but I'm going to write it like this. I'm going to assume that m only has r bits set, same as the number of important bits, r is a number of important bits. But I don't know where they are. So I'm just going to suppose they're at positions m0, m1, up to mr minus 1. I've got to find what these mi's should be, or mj's. And now just taking this product, so we can expand out the product algebraically and see what we get.

So, what's this product? Sum i equals 0 to r minus 1 sum j equals 0 to r minus 1 of xbi-- I mean just the product of these. So 2 to the bi plus mj. That's the algebraic product of those two things. That's why I wrote it out this way. So I can see what's going on. The point is when you do multiplication, you're doing these pairwise products.

Now the guys that are going to survive are the ones where the xbi's are 1, of course. But they survive in multiple places. Essentially the mj's shift all of those bits by various amounts. So it used to be at this position, 2 to the bi. But now we're shifting it by mj for all j. So some of those bits might hit each other. Then they add up. That's really messy. We're going to avoid that, and design the mj's so that all of these values are unique. Therefore, bits never hit each other. That's step one. And then furthermore, what we care about or what we're trying to do is to get the xbi's to appear in a nice little window, consecutive interval of w to the 4/5 bits, somehow by setting the mj's. So let me tell you the claim, which we will prove by induction.

So we're given these bi's that we can't control. Those are the important bits. And the claim is we can choose the mi's such that three properties hold. First one is that bi plus mj are distinct for all i and j. So that was that these bits don't collide with each other. So there's no actual summation here. These sums could then be replaced by ORs, which makes it very easy to keep track of where the bits are going, if we can achieve this property.

Property b is that it turns out that the bits I'm going to end up caring about our b0 plus m0, b1 plus m1, and general bi plus mi. In general, we have bi plus mj for different values of i and j. I claim the ones I care about are the ones where i and j are equal. So I'm going to look at these bits, and in particular I want them to appear in order in the bit string.

And then third property-- I need some more space-- is that if I look at the span of those bits, so I look at br minus 1 plus mr minus 1 minus b0 plus m0 that is the interval that these bits span. I want that to be order r to the fourth power. Because r was w to the 1/5. So this would be order w to the 4/5. That's what I need for everything to fit in. So this is guaranteeing that these bits are the sketch that I need. They appear in order, and they don't span a very large interval, just w to the 4/5. This is what I need to prove. If I can prove this, I have approximate sketching. So let's prove it.

Proof happens in two steps. First thing I'm going to worry about is just getting these guys distinct. Then I'll worry about the order property. So here's how we get them distinct. And these are going to be the mi primes, not quite the mi's that we want. They're all going to be integers less than r cubed, greater than or equal to 0, and they're going to have the property that the bi's plus mj primes are distinct mod r cubed. So this is a stronger version of a. We really just need them to be distinct. But to make it easier for the other steps, we're going to force them to be distinct mod r cubed.

How do we do this? By induction. So let's suppose that we've picked m0 up to mt minus 1. So suppose by induction that we've done that. And now our goal is to pick mt prime. So how do we choose mt prime? Well, what can't it be? mt prime has to avoid basically mi prime, and believe minus bj plus b-- we're going to call it k? I guess so.

If it avoids all expressions like this, then mt prime plus bj will be different from mi prime plus bk. In other words, all of these things will be distinct. So it has to avoid this modulo r cubed. If I can avoid all of these things-- so this is for all ijk-- if I can choose mt prime to avoid all those, then I'm happy. Because then these things will continue to be distinct, and then I apply induction.

Well, how many choices are there for i, j, and k? For i, there's I guess t choices. Because mi can be any of the previous values. For j, let's call it r choices for k, there's r choices. That's how many important bits there are. So total number of choices is tr squared. But t here is always less than r. So this is going to be less than r cubed. So that means there is less than r cubed things we have to avoid. But I have r cubed allowable choices on working modulo r cubed. So I just pick any one that avoids the collision. This is basically deterministic hashing, in a certain sense. We are choosing these values deterministically to avoid collisions in this simple hash function.

OK. It takes time. It's going to take polynomial time to compute this thing. And you can imagine if you just plug in hashing, this will work with some probability, and blah, blah, blah. But I want to make it always work deterministically. Because we know what the xi's are here.

All right. So we've avoided collisions. There's enough space. That's all. That was step one. Step two, and this will solve property a, even modulo r cubed. Now we have a little bit of space. We're allowed to go up to r to the fourth. And now we just need to spread out these bits. So that's step two.

Basically we're going to set mi to be these values that we chose plus this weird thing, w minus bi plus ir cubed rounded down to a multiple of r cubed. So I guess you could put this in parentheses if you want. Rough idea is, we want to take mi prime plus ir cubed. Because these mi primes, they're all values between 0, and r cubed minus 1. We got everything working modulo r cubed.

If we could just add ir cubed to each of these values that we'll spread them out. Because each of these values used to fall just in this tiny range ir cubed. So we can move the next one to the next position, move the next one to the next position, and so on. Spread them out to the left by

adding on multiples of r cubed, then that will achieve property b.

The annoying issue here is we don't want to mess things up modulo r cubed. So we need to round things down to be a multiple of r cubed so that this is congruent to mi prime. That's what we want. We want it to stay congruent to mod r cubed. Well, why do we need to round down to a multiple of r cubed? We were adding on ir cubed. Well, it's not quite mi that we care about. It's mi plus bi. Those are the bits that we want to be nicely ordered. And so we kind of need a minus bi here, so that when we take mi plus bi, those cancel.

But then bi is not a multiple of r cubed. So you've got to do this rounding down to r cubed. Also negative bi is a negative number. And we can't really deal with negative numbers. Because you can't go left of 0. So we have to add on this w just to make things work out. So it's a little messy, and I don't want to spend too much time on why this formula works. But I think you have the essence of what's working.

This is just to avoid negative numbers. This negative bi is so that when you add it to mi that cancels. And so you get these r cubes separations. In the end, let me draw a picture, perhaps.

In the end, if you look at the bit space, so this is w bits. And you divide it up into multiples of r cubed. All of the mi primes are over here. So these are mi primes. You don't know in what order or anything. They're just kind of randomly in there, and chosen pretty much arbitrarily on the low end of the spectrum, from 0 to r cubed minus 1. And then what we want is for x0 plus m0 to fall somewhere in this range, and then x1 plus m1 to fall somewhere in this range, and x2 plus m2 to fall somewhere in this range.

If I do that, and it's weird because the bits are numbered from 0 to the left here. Then I will have this property. I claim this assignment does that. It's an exercise you can check that indeed xi plus mi will fall in this range.

So this gives us property b. It also gives us property c. Because we've been fairly tight here. There's r of these guys, and r of these intervals of size r cubed. And so the total range of these bits is going to be r to the fourth. We started at x0 plus m0. We end at xr minus 1, plus, mr minus 1. That's going to be somewhere here. But if you look at just that interval of bits-- so there are more bits actually over here, in particular, because of this w bit part. This whole picture basically starts at bit w. Then there's all this stuff to 0. So this is a more accurate picture.

You're doing this multiplication. Garbage happens here. We have no idea. Garbage happens here. We have no idea. Actually, garbage happens all over here. But what we know is that these bits are the bits we care about. These are the xi plus mi bits. If you look at xi plus mi, they give you xbi. They exist in other places. But these bits will have the important bits. Now the bits are also all over everywhere else. But none of the bits hit each other. So these bits remain correct, because nothing else collides with it. And so if I just mask out those bits, again, so I have to do another mask.

I did one mask here. I did a multiplication, and then I have to do another mask. So why don't I write it over here? So we AND with sum i equal 0 to r minus 1 of 2 to the bi plus mi. Those are the circled bits. So if we grab those things, and then we shift right by-- why am I writing x? Sorry. These are all b's. Too many letters. We shift right by b0 plus m0. Because we don't care about all those leading bits. So we shift this over to the left. We did the mask, then we will just have the important bits and they will occupy over here an interval of size at most-- I'll say order r to the fourth. Clear?

So this is approximate sketching. This is definitely a bit complicated, but it works. Let me review briefly. So, our algorithm was simple. We have a bit string, x. We just want to get the important bits and compress them to a thing of size r to the fourth. So first of all, we threw away all the non-important bits with this mask. That was easy. Then we just did an arbitrary multiplication, and we proved that there was a multiplication that avoided collision. So the sums basically turned into ORs or XORs. I mean you never get two 1 bits hitting each other, so you don't have to worry about that. And we did that with the simple inductive argument.

And then we also wanted the bi's plus mi's to be linearly ordered. Because we need to preserve the order of the important bits. We can't just permute them. And we needed them to occupy a small range. And we did that basically by adding ir cubed to each of them. But it was a little messy and we had to add w, and blah-blah-blah. But in the end, we got our important bits to be nicely spaced out here by pretty much putting an r cubed in between each one. So those were our bi plus mi bits. They occupied this range of r to the fourth. We'll mask out all the rest of the garbage. Because this multiplication made a quadratic number of bits. We only want these r bits, the r squared 1 bits in here. We'll mask away all the others. Take these bits, shift them over. Now they occupy a nice interval at the beginning size order r to the fourth. And that's our approximate sketch.

So sketch should only take r, but we're being sloppy. With this multiplication trick, the best we

know is to get down to r to the fourth. And that's good enough. And that's why I set everything to w to the one fifth. Because this is w to the 4/5. We're going to have w to the 1/5 of them. And so if you take these sketches and you concatenate them, fuse them together if you will, and that's fusion trees. Then the sketches of all of the keys x0 up to xk minus 1 will occupy order 1 words. Because it's order w bits, w to the 4/5 times w to the 1/5.

Which brings us to parallel comparison. I have all of these approximate sketches. So you could start forgetting approximate sketching. Somehow, we get these w to the 4/5 bits. We want to concatenate them together, and then in parallel compare all of them to the sketch of q. The sketch of the xi's we can preprocess. We can actually spend a lot of time finding the sketch function. But then we have to fix the sketch function. We have to be able to compute a sketch of q in constant time. That's what we just did. Sketch of q is one AND one multiplication and another AND. So computing sketches is fast. That's the steps of computing sketch of q.

Now, next step is find it among the sketch of the xi's. So this is the next thing we want to make fast. It's actually pretty easy. You probably know you can compare two integers by subtracting one from the other. So we're just going to do that, but in a clever way, so we can do k subtractions for the price of one.

I'm going to define the sketch of a node to be 1 bit followed by the sketch of x0 dot, dot, dot, 1 sketch of xk minus one. And I'm going to define a sketch of q to the k-th power, so to speak, to be a 0 bit followed by sketch of q, dot, dot dot, zero bit sketch of q. This is aligning things, so that if I did this subtraction and this one, I would basically be comparing q with all the xi's at once.

The point is these sketches-- this is the thing that fits in order 1 words. These sketches are w to the 4/5 bits, and there's w to the 1/5 of them. So this whole thing is order w bits. So it fits in one word. This thing also. It happens to be the same bits repeated many times but also it fits in one word. How do I compute this thing? I can do it with multiplication. It's sketch of q times 0000001, 000000001.

So, ahead of time, I'll just pre-compute this bit string that has 1's at the rightmost slot for each of these k fields. If I just take that and multiply it by sketch of q, then I get this. So this is easy to do in one multiplication. Now, I take this thing minus this thing. I take the difference. And the key thing is because I put these 1 bits here, I'm taking this minus this. The point is either this 1 bit will get borrowed when I do binary subtraction, or it won't. It gets borrowed when this is

bigger than this, otherwise it doesn't get borrowed. So I'm going to get either a 0 or 1 here, and then some garbage which I don't care about, and a 0 or a 1 here, and then some garbage.

And I'll just mask that out. I'm ANDing with 10000, 100000. And so I end up just with 01 bits and the rest 0's. And these bits, if I get it right, it's 1 if the sketch of q is less than or equal to the sketch of xi. And it's 0 if the sketch of q is greater than the sketch of xi. Because when it's greater that's when the borrow happens. And then the 1 turns into a 0. So 1's indicate the query is too small or they're just right. And 0's indicate that they're greater.

Now the xi's were in order. So probably x0 is too small. And so this bit will end up being a 0. Probably xk this plus 1 is too big. So this bit will be a 1. In general, it's going to be a monotone sequence of bits. If you look at these bits, these 01 bits, they are going to be monotone. They'll be 0 for a while, and then at some point they'll switch to being 1's. And that transition from 0 to 1 that's what we want to find. These keys are too small. These keys are too big. This key is just right.

So we fit between-- this would be position i and position i plus 1. And we fit between xi and xi plus 1. Well, not actually xi and xi plus 1. We fit between sketch of xi and sketch of xi plus 1. That's what we need to find. Now that is again the problem of finding the most significant 1 bit. But in this case, I don't need that operation. I can do it in a simpler way.

But we're almost done, right? We've done all of this parallel comparison. We just need to find that transition between 0's and 1's. Turns out there's a cool way to do it. The cool way is multiply that word times our good friend, this thing, 000001, 000001. This is a little harder to think about. But take this bit string and multiply it by this.

What that does is it takes this string. It includes it. Because there's a 1 right there. It shifts it over by one field, and includes it, shifts it over by another field, includes it. So this repeats this thing. And now collision happens, because they're perfectly aligned. If these 1 bits ever hit each other, they'll be summing. Now, some of them are 0, some of them are 1. Instead of computing the position of the 0 to 1 transition, we could equivalently just count how many 1's are there. I mean that's counting from the right, whereas this is counting of from the left, whatever, same thing.

So if I could count how many 1's I'd be all set. And in this case, if you look at right here, this will be the number of 1's I claim. Because if this one was there, it will stay there. And then all the

other bits get shifted over and fall right here on top of this bit. So as they get added up, you'll get some carries and things will move over. But this is not very big. Because we're talking about k bits. So this is only going to be with log k. I mean there's tons of room here before we get to the next shift.

So I just look at these bits. I mask them out. I shift them over. And that gives me the number of 1's. This is a cute way to count the number of ones in a bit string when the bits are spread out nicely. They have to be at least log k away from each other. Otherwise you get collision. It doesn't work for an arbitrary bit string. But for a bit string like this, we're all set. We can count how many 1's there are. Then we figure out where this transition is. That is parallel comparison.

One more thing to do, which is most significant set bit. The place we needed this, was we were taking the XOR of q with xi. And then we wanted to find the first bit where they were differing. So after you take the XOR, you've got some bit string that looks like this. And you want to find this bit, because that's the place you diverged. Then we would turn that to 0 and change the rest to 1's. That's easy to do if we know where this bit is. And this is a generally useful operation. It's used all over computer science, I would say. So much so that most CPUs have it as an instruction, so on Intel it's called CLZ. And it has many names. They're in the notes.

Most compilers provide this to you as an operation on architectures that have it, otherwise they simulate it. They probably don't simulate it as well as I'm going to tell you. Because we're going to do this in constant time on a regular word RAM, just C operations, which does not seem to have made it into popular culture. It's slightly complicated, which is why. But what's cool is we're going to use-- I'm going to do this relatively quickly. Because I don't have a ton of time.

We're going to use all the things that we just did again, quickly. Most of them just as black boxes. All right. So, here's what we're going to do. Maybe I should go somewhere new. So, I'm going to use sketches, not approximate sketches, but I'm going to sketches. I'm going to use multiplication. I'm going to use parallel comparison. And in some sense I'm going to use most significant set bit. All of these things I'm going to use to solve the most significant set bit problem.

So here's what we do. We split the word into root w clusters of root w bits. Sound familiar? This is exactly what we did in van Emde Boas. So van Emde Boas did this recursively. We're

going to do it once. We can only afford constant time. So here's an example. x is 0101, 0000, 1000, 1101. So each of these is root w bits. There's root w of them. It's approximate. It doesn't it to be exactly. But we'll assume x is a nice power of two, so that works cleanly.

So the first thing, so what the high level idea is I need to find the first non-empty cluster. Here it happens to be the first cluster. And then I need to find the first 1 bit within the cluster. Hard part is finding the first non-empty cluster. Actually, the hard part or the messy part is finding which clusters are empty and which clusters are not. This cluster is not empty. This cluster is empty. These are non-empty. So I want the summary vector which is 1011. I claim if I can do that, everything else is easy. So let's spend some time on identifying non-empty clusters.

First thing I do is I take x, ANDed with this thing, which I'm going to call F, 1000, 1000, 1000, 1000; F for first. So I'm just seeing which of these first bits in each cluster are set. So the result is I get 0000, 0000, 1000, and 1000. So in particular that tells me this cluster and this cluster are non-empty, because they have the first bit set. What about all those other bits? Well, the other bits I'm going to do in a different way. Just the first bits, I need a little bit of room.

I need this bit of room. I want to put these 1's in. So I've got to get rid of some bits to make room for that. So this deals with the first bits. Now I'm going to clear those out. So I'm going to take x XOR this. And that will give me everything with the first bits cleared. So I've got 0101, 0000, 0000, and 0101. These are the rest of the bits I've got to figure out. This one is non-empty and this one's non-empty. How do you do it? With subtraction.

I take F minus that thing. This F has 1's, and they're going to get borrowed. When I take F minus this, this 1 will get borrowed because there's something here. This one will not get borrowed because this is 0. This one will not get borrowed because this is 0. This one will get borrowed because there's something here. That's it. We're comparing with 0 everything. So we're going to get, in this case, 0 and some garbage, 1 and 0's, 1, and 0 with some garbage. I just care about these bits. These are the bits that tell me which ones were empty. The 0's are empty. The 1's are non-empty.

So I do a mask. I get 0, and some 0's, 1 and some 0's, 1 and some 0's 0 and some 0's. OK. Then I do an XOR with F. Because I really want 1 for these guys, and 0 for these guys. 1 means it's not empty. 0 means it's empty. I got that right. So I'm just inverting the 0 bits to 1 bits, and vice versa. So 1 means this one's not empty. 1 means this one's not empty. Those are the non-empty guys.

I take this and I OR it with this thing. This was the thing that told me which ones had that first bit set. So if I take the OR of those two I learn, or any bit set. Because this was dealing with all of the other bits. I threw away this bit, but I had to remember that it was non-empty. OK. So I take that OR. Now, this tells me those three blocks were not empty. This one was empty. So now here I have the bits that I care about.

Sadly they're spread out. I'd really like them compressed. So I do that with sketch. I want to compress them to 1011. It would fit in one little thing here. Because this is root w. There's root w of them. Sadly, I can't use approximate sketch. Because I don't have enough space. This is w to the 1/2. If I used approximate sketch I get w-- I'd lose this factor of 4 and be bigger than w. I really need it to be perfectly sketched.

Conveniently, you can do perfect sketch in this regime. Before the bi's were arbitrary things. We had no idea how they were spread out. Here bi is root w minus 1-- that's the first one-- plus i times root w. They're nicely uniformly spaced by i root w. In this case-- I'm running out of time-- I claim you can use mj equal to w minus root w minus 1 minus j root w plus j. And I won't go to the proof. There's a sketch in the notes.

If you do this, this is a nice setting of mj. It turns out you will get bi-- if we look at bi plus mi, this cancels, this cancels, because i equals j. You're left with w plus j. So in other words, if you look at bi plus mi, you get from bit w to bit w plus root w minus 1. These bits will be exactly the bits you care about. So you take those. You mask out the others. You shift it over to the right, and you have exactly your perfect sketch.

The thing you need to prove here is that bi plus mj are all distinct. So there's no collisions. But in this case it's easy to avoid collisions. You've got all your bits nice and consecutive. Now you've got it down to this thing. OK, not quite done though. Only one more minute. Let's say-- well, that was step one. Identify non-empty clusters. Step two was sketch.

Step three is find the first non-empty cluster. I claim this is easy. So I take this sketch vector. It only has root w bits. So I use parallel comparison. What do I compare to? I'm going to compare many copies of this thing to 0001, 0010, 0100, 1000; the powers of 2. So I take this. I put them in a vector like the sketch of a node. And I take the k, or I guess root w copies of the sketch of the summary vector. That's this 1011. So I compare four copies of this to each of these, and I learn which power of 2 it is greater than. In other words, what is the most significant set bit.

That's why when I told you how to do over here, when I told you how to do parallel comparison, I didn't want to use most significant bit as a subroutine. Because this is a subroutine to most significant bit. Over here, we could just do this multiplication and boom, we found what the most significant set bit was as long as there was room to fit all this stuff in a word. And because I've reduced everything to size root w, and then only there's w of these things to compare to, because that's the width of one of these fields. This all fits in a word. I can do this parallel comparison. Boom, I find the first 1 bit in this bit string, which happens to be the first bit.

That tells me that this cluster is a cluster I care about. So I take those bits out. I mask them out, shift them over, and I find the first 1 bit in that cluster. How do I do it? In exactly the same way, clusters again, root w bits. I can use parallel comparison to compare it to all these things in constant time. I find where the first 1 bit is there. And then I take this cluster C, I take this bit D, and my answer is C root w plus D. That is the final index of the most significant 1 bit in constant time, using all those fusion tricks once again.

And that in the end gives you fusion trees on a word RAM static. It's complicated, probably impractical, but pretty cool. And we're going to use these bit tricks again.