ERIK DEMAINE: All right, welcome to my last lecture for the semester. We finish our coverage of dynamic graphs, and also our coverage of lower bounds. We saw one big lower bound in this class in the cell probe model. You may recall cell probe model, you just count how many cells of memory do you touch. You want to prove a lower bound on that.

And today we're going to prove a cell probe from lower bound on dynamic connectivity, which is a problem we've solved a few different times. Our lower bound will apply even when each of the connected components of your graph are just a path. And so in particular, they imply matching lower bounds for dynamic trees.

So here is the theorem we'll be proving today. You want to insert and delete edges, and do connectivity queries between pairs of vertices, vw. I want to know is there a path from v to w, just like we've been considering. These require omega log n, time for operation.

This is the max of updating, query times has to be at least log n time per operation, even if the connected components are paths, and even amortized, and even randomized. Although I'm not going to prove of all of these versions, I won't prove the amortized version. I'm going to prove a worst case log and lower bound, it's just a little more work to prove an amortized lower bound. But same principles.

And so that's going to be today, is proving this theorem. It's not a short proof, but it combines a bunch of relatively simple ideas, and ends up being pretty clean overall, piece-by-piece, but there's just a bunch of pieces, as we will get to. Key concept is an idea introduced in this paper, which is to build a balanced binary tree over time, over your access sequence. And argue about different subtrees within that tree.

This is a paper that maybe came out of this class, in some sense, it was by Mihai Patrascu and myself. Back when Mihai was an undergrad, I think he'd just taken this class. But at that point the class didn't even cover dynamic connectivity, so-- and time here is cell probes.

So this is a very strong model, it implies a lower bound on ram, and implies a lower bound on pointer machine. We know matching upper bounds for trees on a pointer machine link/cut trees in [INAUDIBLE] to our trees. It's kind of fun that this lower bound even applies to paths, because most of the work in link/cut trees is about decomposing your tree into paths.

And so what this is saying is even if that's done for you, and you just need to be able to take paths, and concatenate them together by adding edges, then maintaining the find root property so that you can do connectivity queries, even that requires log n time. So converting a tree into a path is basically free, the hard part is maintaining the paths.

So let's prove a theorem. The lower bound, we get to choose what access sequence we think is bad. And so we're going to come up with a particular style of graph, which looks like the following.

Graph is going to be-- the vertices are going to be a root n by root n grid. And we're going to-- these guys are in, what did I call them? Groups? Columns. Columns is a good name. These are columns of the matrix, or vertices. And what I'd like to have is between consecutive columns, I want to have a perfect matching between these vertices.

So could be, I don't know, this edge, this edge, this edge, and that edge. And I also want a perfect match between these two columns, so maybe this one, this one, this one, this one. And you can have some boring things too. Something like that. So between every pair of columns is a perfect matching, meaning perfect pairing.

OK, this of course results in a collection of paths, square root of n paths. You can start at any vertex on the left, and you'll have a unique way to go to the right. And so that's path 1, this is path 2, this is path 3, and this is path 4. And so if I-- an interesting query. Well, an interesting query is something like I want to know, is this vertex connected to this one? And it's not so easy to figure it out, because you have to sort of walk through this path to figure that out.

We're going to think of each of these perfect matchings as defining a permutation on the vertices, on the column really. So you start with the identity permutation, and then some things get swapped around, that's pi 1. Something gets swapped around again, that's pi 2. Somethings get swapped around here, that's pi 3. And then this position would be pi 3 of pi 2 of pi 1 of vertex 4. We call this vertex 4. Or row, row 4.

So in some sense, we have to compose permutations. I'll call this pi 1, circle pi 2, circle pi 3 of 4. And we're going to show, basically, composing permutations is tough when you can change those permutations dynamically. So what we're going to do is a series of block operations, which change or query entire permutations.

So here, an update is going to be a whole bunch of insertions and deletions of edges. Basically, what we want to do is set pi i equal to pi. So that's what update of i comma pi does. It changes an entire perfect matching to be a specified permutation. So how do you do that? Well, you delete all the edges that are in the existing permutation, then you insert all the new edges. So this can be done in square root of n edge deletions and insertions.

So it's a bulk update of square root of n operations. And so this could only make our problem easier, because we're given square root of n updates that we all need to do at once. So you could amortize over them, you could do lots of different things, but we're sure that won't help. And then we have a query, and the query is going to be a little bit weird, and it's also going to make the proof a little bit more awkward.

But what it asks is if I look at the composition of pi j, from 1 up to i. This is 1. So I want to know is that composition equal to pi? Yes or no? This is what I'll call verify sum, sum meaning composition. But the sum terminology comes from a different problem, which we won't talk about directly here, called partial sums.

Partial sums is basically this problem, you can change numbers in an array, and you can compute the prefix sum from 1 up to i. Here we're not computing it. Why are we not computing it? Because actually figuring out what pi 3, or pi 2 of pi 1 is of something is tricky in this setting. The operations we're given are, given two vertices, are they connected by a path?

So to figure out the other end of this path, that requires-- I mean, it's hard to figure out where the other end is. If I told you is it this one? Then I can answer that question with just a connectivity query. So verify sum can be done with order square root of n connectivity queries. Whereas computing the sum could not be, as far as we know. If I tell you what that composition is supposed to be, I can check does 4 go to 1? Yes or no? Does 3 go to 3? Yes or no? Does 2 go to 4? Yes or no? Does 1 go to 2? Yes or no?

So with 4 queries, I can check whether the permutation is what it is. If any of those fail, then I return no. So the way this proceeds is first we proved a lower bound on partial sums, which is computing this value when you're not told what the answer is. And then we extended that, and we'll do such a proof here today.

First, we're going to prove a lower bound on the sum operation, which is computing this value that's on our outline over here, sum lower bound. And then we'll extend that and make the argument a little bit more complicated, then we'll get an actual connectivity lower bound, a lower bound on verify sum.

OK, but obviously if we can prove that these operations take a long time to do, we can prove that these original operations take a long time to do. So what we claim is that square root of n updates, these block updates plus square of n verify some queries, require root n times root n log n cell probes. I guess this is the amortized claim.

So if I want to do root n updates and root n queries, and I take root n times root n times log n-- funny way of writing n log n-- cell probes, then if I divide through, I want the amortized lower bound, I lose one of these root ns, because I'm doing different operations. I lose another root n, because each of these updates corresponds to root n operations. Each of the verify sums corresponds to root n operations. So overall per operation, per original operation of edge deletion insertion or connectivity query, I'm paying log n per operation.

So if I can prove this claim, then I get this theorem. All clear? So now we've reduced the problem to these bulk operations, we'll just be thinking about the bulk operations, update verify sum. We won't think about edge deletion insertions and connectivity queries anymore.

OK. So this is just sort of the general set up of what the graphs are going to look like. And now I'm going to tell you what sequence of updates and verify sums we're actually going to do that are bad. This is the bad access sequence. And this is actually something we've seen before in lecture 6, I think, the binary search tree stuff.

We're going to look at the bit reversal sequence. So you may recall a bit reversal sequence. You take binary numbers in order, reverse the bits. So this becomes 000, 100, 010, 110, 001, 101, 011, and 111. So those are the reversed strings.

And then you reinterpret those as regular numbers. So this is 0, 4, 2, 6, and then it should be the same thing, but the odd version. So I have 1, 5, 3, 7.

OK, I claimed, I think probably didn't prove, that this bit reversal sequence has a high Wilber lower bound. And so any binary search tree accessing items in this order requires log n per operation. And we want log n per operation here, so it seems like a good choice, why not?

So we're going to follow this access sequence. And sorry, I've changed notation here, we're going to number the permutations from 0 to root n minus 1 now. And assume root n is a power of 2, so the bit reversal sequence is well defined. And then we are going to do two things for each such i. We're going to do a verify sum operation. Actually maybe it is starting at 1, I don't know. It doesn't matter. And then we'll do an update

OK, so let's see. This pi random is just a uniform random permutation, it's computed fresh every time. So we're just re-randomizing pi i in this operation. Before we do that, we're checking that the sum, the composition of all the permutations up to position i, is what it is. So this is the actual value here, and we're verifying that that is indeed the sum. So this will always return yes.

But data structure has to be correct. So it needs to really verify that that is the case. There's the threat that maybe we gave the wrong answer here, and it needs to double check that that is indeed the right answer. It may seem a little weird, but we'll see why it works.

So this is the bad access sequence. Just do a query, do an update in this weird order in i. OK, and big idea is to build a nice balanced binary tree over time. So we have on the ground here 0, 4, 2, 6, 1, 5, 3, 7. And when I write 5, I mean verify sum of 5, and update permutation 5. And then we can build a binary tree on that.

And for each node in this tree, we have the notion of a left subtree, and we have the notion of a right subtree. And cool thing about bit reversal sequence is this nice self-similarity. If you look at the left subtree of any node and the right subtree of any of node, those items interleave.

If you look at the sorted order, it's 1 on the left, 3 on the right, 5 on the left, 7 on the right. They always perfectly interleave, because this thing is designed to interleave at every possible level. So that's the fact we're going to use. We're going to analyze each node separately, and talk about what information has to be carried from the left subtree to the right subtree.

In particular, we're interested in the updates being done on the left subtree, because here we change pi 1, we change pi 5. And the query's being done on the right subtree, because here we query 3, we query 7. When we query 3, that queries everything, all the permutations, up to 3. It's a composition of all permutations up to 3. So in particular it involves 1. So the claim is going to be that the permutation that we set in 1 has to be carried over to this query. And similarly, a changing permutation 5 will affect the query for 7. Also query, the update for 1, will affect the query for 7. So we need to formalize that little bit.

So here is the claim. For every node in the tree, say it has l leaves in its subtree-- This should be a comma and this should be a colon. Here's what we say. During the right subtree of v, so right subtree corresponds to an interval of time. So we're talking about those operations done during the right subtree of v. Claim is we must do omega l root n cell probes-- sorry, expected cell probes.

We are using some randomness here, right? We said we're going to update each permutation to a random value, so we can only make claims about the expected performance. Fine. But that's actually a stronger thing, it implies a lower bound, even for randomized algorithms. So if you can randomize your input set. And then not just any cell probes, but they're cell probes that read cells last written during the left subtree.

So this is what I was saying at a high level before. We're looking at reads over here, to cells that are written over here. Because we claim the updates over here have to store some information that is-- whatever the updates that happen over here influence the queries that happen over here. So these queries have to read the data that was written over here.

And specifically, we're claiming at least l root n cell probes have to be read over here, from cells that were written-- that were basically just written in the left subtree. If we could prove this, then we get our other claim-- this one over here, that root n updates, and root n verifies sums that require this much time. The difference is-- well, here we have an l, for an l leaf tree. And so what I'd like to do is sum this lower bound over every node in the tree. I need to check that that is valid to do. So let's do that.

OK, for every node v, we are claiming there's a certain number of reads that happen over here, that correspond to writes over here. But let's say you look at the parent of v, which is over here. This thing is also in the right subtree, and we're claiming there's some number of reads on the right subtree, that read things that are written over on the left.

The worry would be that the reads we counted here, we also count at the next level up. We don't want to double count in our lower bounds. If we're able to sum them up, we can't be double counting. But the claim is we're not double counting, because if you look at any particular-- any read-- so here's time, and suppose you do a read here. You're reading a cell that was written sometime in the past, if it was never written, it's a not very interesting read, it communicates no information.

So there's some write in the past that changed the cell that's just read. And we are going to count this read at a particular node, namely the lca of those two times. So if you look at the lca of the times of the reads and the writes, that is the single note that we'll think about that read that happened in the right subtree, that was written in the left subtree. So no double counting, because we only count at the lca.

The other thing that we need to be able to do is, because this is an expected lower bound, we need linearity of expectation. But expectation is indeed linear, so we're all set. OK, so all that's left is a little bit of common [INAUDIBLE] if we take l root n, where l is the size of the subtree below a given node, we sum that up over all nodes, and it's a balanced binary search tree-- or a balanced binary tree, I should say, not a search tree.

What do we get? Well, every leaf appears in log n subtrees. So we get the total size of the tree times log n for this, and we get another root n over here. The total size of the tree is root n. So we get this root n log n, that's when you sum up the l part. Then everything gets multiplied by root n, and that becomes our lower bound, and that's exactly what we need over here. So now this claim is done. Maybe I should do a check mark. Provided we can prove this claim.

So now our goal is to prove this thing. And now we're in a more local world, looking at a single node, counting reads over here, the corresponding rights over there. And then you just add up those lower bounds, you get what you want. So this is where the log comes from, because it's a balanced tree. And there's log n levels in a balanced tree, that's where we're getting our lower bound. The root n's are just keeping track of the size of the things we're manipulating. All right. So it remains to prove this claim. Prove that claim, we get that claim, and then we get this theorem.

So proof of claim. We're going to do an information theoretic argument, so let me set it up. It's again, it's making this claim I said before, that the permutations that get written over here somehow have to be communicated to the queries over here, because they matter. Because the permutations that get said over here changed the answers to all the queries over here, because of the interleaving between left and right.

So how are we going to formalize that? Well, left subtree does l/2 updates with l/2 random permutations, uniform random permutations, because every node does an update. And so the information theoretic idea is that if we were to somehow encode those permutations, That encoding must use omega l log l-- l? No, I'm sorry. It's not right. Off by some root n factors here, l root n log n. OK, each permutation must take root n log root n bits to encode. If you have a random permutation, expected number of bits have a very high probability. Almost every permutation requires root n log root n bits. I'm not going to worry about constant factors, put an omega here, so the root n turns into an n. And then we've got l over two of them, so again, ignoring constant factors, that's l root n log n bits.

And this is just information, theoretic fact, our common [INAUDIBLE] theory fact. And once we know that, the idea is let's find an encoding that's better than this, and get a contradiction. Of course we shouldn't get a contradiction unless this claim is false. So either this claim is true and we're happy, but if somehow the word not enough cell reads on the right, that did things that were written on the left, then we will, from that, get a smaller encoding of the update permutations that happen on the left. If we could somehow do that, then we can get a contradiction, and therefore conclude the claim is in fact true.

So, if the claim fails, we'll find a smaller encoding, which will give us a contradiction. All right, so let's set up this problem a little bit more. I'm going to-- because we're really just interested in this subtree v stuff on the left, stuff on the right, but this of course lives in a much bigger tree, there's stuff that happens over here. This I will call the past. I'm just going to assume we know everything about the past.

Everything to the left of the subtree, we can assume that we know. When I say we know, what do we know? We know all the updates, we know all the queries that happen, and we know, at this moment in particular, what is the state of the data structure. Because this claim has nothing to do with this stuff, it's all about reads here that corresponds to writes here. So we can just assume we know everything up to this point. In our encoding, this is a key point.

One way to say this in a probabilistic sense is we're conditioning on what happened over here on the left, what updates happened. And if we can prove that whatever we need to happen here holds no matter what the condition is, then it will hold overall. So that's probabilistic justification for why we can assume we know the past OK. So then our goal is to encode, this is a little bit different from this goal.

What we really want to do is encode the update permutations on the left. That's a little awkward to think about, because this is a claim about how many probes happen on the right. So instead, what we're going to do is encode the query permutations on the right. So there are updates over here, that's what we want to encode, but we're instead going to encode the queries over here. I claim if you know what the results of the queries were over here, then you know what the updates were over there. Basically because of this interleaving property. So I can write that down a little more formally.

So if we look at time here, over, let's say this is v's subtree. Then what we have are a sequence of updates and a sequence of queries. These are queries, and these are updates. This is what the sequence looks like-- sorry, this is v's subtree, this is the pi is, I should say. I mean, these operations are all happened during time, but now I'm sorting by i. A little confusing.

There are two orders to think about, right? There's the sequence over time, we're now looking at such a left subtree where we do say 1, 5, and 3, 7. What that means-- so you're imagining here, this is 1, this is 5, this is 3, this is 7. Here we're sorted by the value written down there, we're sorting by the i, the pi i that they're changing or querying. And so all the read things are in the right subtree of v. And all the updates are in the left subtree of v. This is the interleaving property that I mentioned earlier.

So I claim that if I encode the results of the queries, namely I encode these permutations, these are like summary-- partial sums. These are prefixed sums of the permutation list. Then I can figure out what the updates were. Why? Because if I figure out what this query, what it's permutation is, that's the sum of all of these permutations.

Now only one of them changed in the left subtree, the rest all are in the past. They were all set before this time over here, and I know everything about the past, I'm assuming. So most of these I already know, the one thing I don't know is this one, but I claim if I know this sum, and I know all the others, then I can figure out what this one is, right?

It's slightly awkward to do, if I give you this, I give you the sum of pi j from j equals 0 to i, or something. I've got to-- I want to strip away all these, strip away all these. So I'm going to multiply by sum of pi j inverses over here, and multiply by sum pi j-- when I say multiply, I mean compose. Sum pi j inverse is here, maybe let's not worry about the exact indices here. But the point is, this is all in the past, and this is all in the past, so I know all these pi js, I know they're inverses. So if I have this total sum, and I right multiply with these inverses, left multiply with these inverses, I get the one that I want. This gives me some particular pi k, if I set the indices right.

OK? So if I know this query, I figure out what this update is. Now once I know what this update is, and I know this query, then in this sum, I know everything except this one thing. And so by using the same trick, I can figure out what this update is. So now I know the first two updates, if I then know the answer to this query, I can figure out what this update is. If I know the answer to this query, I can figure this update. Because they're perfectly interleaved, I only need to reconstruct one update at a time.

So if I'm given-- if I've somehow encoded all of the queries results, all of these prefix sums, and I'm given the past, then I can reconstruct what all the updates were. So that's basically saying these two are the same issue. If I can encode the verified sums in the right subtree, using less than l root n log n bits, then I'll get a contradiction, because it implies that from that same encoding, you can also decode the update permutations in the left subtree. So that's our goal.

OK. So we'd like to prove this for verify sum. But the first thing I'm going to do is consider an easier problem, which is sum. So suppose, basically, this was not an input to the query. Suppose the query was, what is the sum of i? Like this. I just want-- this is the partial sum problem. I'm given an index i, I want to know what is the permutation from pi 0 up to pi i. Now that is not-- that doesn't correspond to dynamic connectivity, it's a new problem. We'll first prove a lower bound for that problem, and then we'll put the verify word back in.

OK, so that's-- we're now here at sum lower bound. Where should I go? Different-- so this is a lower bound on the operation sum, as opposed to here, where we're adding up lower bounds. Sorry for the conflation of terms. Let's go here.

So I'll call this a warm up. Suppose a query is sum of i, which is supposed to give you this prefix sum of pi j again, sum means composition. So this is going to be relatively easy to prove, but it's not the problem we actually want to solve, we'll use it to then solve the real problem. And this is the order in which we actually solve things.

First, we prove a lower bound of partial sums. OK, so let me give you some notation, so we can really get at this claim. Reading on the right, writing on the left. So let r be all the cells that are read during the right subtree, which is an interval of time. And let w be the cells written in the left subtree.

OK, so what we're talking about over here is that r intersects w, those are cells that are read during the right subtree, that were at some point written during the left subtree, should be large. So we want to prove a lower bound on the size of r intersect w. So if the lower bound doesn't hold, that means that r intersect w is relatively small. So imagine a situation where r intersect w is very small, there's not very much information passed from the left subtree to the right subtree. If r intersect w is small, then presumably I can afford to write it down, I can encode it. So that's what we're going to do, and we'll compute-- we'll figure out that this is indeed something we can afford.

I'm going to encode r intersect w explicitly. Meaning-- and this is a set of cells in memory. So for every cell, I'm going to write down what it's address is, and what the contents of the cell are. So write down the addresses and the contents for every such cell.

So how many bits does that take? I'm going to say that it's r intersect w times log n bits. Here's where I need to mention an assumption. I'm assuming that the address space is order log n bits long, that's like saying that the space of your data structure is order-- is polynomial in n. And if you want any hope of having a reasonable update time, you need to have polynomial space at most. So assuming polynomial space, each of those addresses only takes order log n bits to write down. The contents, let's say, also take order log n bits to write down.

OK, so fine. That's-- I mean, yeah. We don't really need to make those assumptions, I don't think, but we will for here to keep things simple. So if r intersect w is small, meaning smaller than this thing, then this will be small, smaller than l root log n. OK.

So on the other hand, we know that every encoding should take l root n log n bits. And so this will be a contradiction, although we haven't quite encoded what we need yet, or we haven't proved that, but we're getting to be at the right point. These log ns are going to cancel in a moment.

So what we need to do is, I claim this is actually enough to encode what we need. And so all that's left is a decoding algorithm for the sum queries in the right subtree. So how are we going to do that? So this is my encoding, these are the bits that I have written down. So now what I know, as a decoder, is I know everything about the past. I don't know what these updates are, that's my whole goal, to figure out what they are. I don't know what the results of the queries are, but magically, I know that r intersect w. Well, not magically. I wrote it down, kept track on a piece of paper. So that's what I know. And so the idea is, well, somebody gave us a data structure, tells you how to do an update, tells you how to do a query.

Let's run the query algorithms over here. Run that query, run that query, or whatever. It's a little hard to run them, because we don't know what happened in this intermediate part. But I claim r intersect w tells us everything we need to know. So the decoding algorithm is just simulate sum queries, simulate that algorithm.

And let's go up here.

How do we simulate that algorithm? Well, the algorithm makes a series of cell reads, and maybe writes, but really we care about the reads. Writes are pretty easy to simulate.

There are three cases for reads. It could be that the thing you're trying to read was written in the right subtree, it could be that it was written in the left subtree, or it could be it was written in the past, before we got to v subtree. Now we don't necessarily know which case we're in, but I claim we'll be able to figure it out. Because any cells that are written in the right subtree, we've just been running the simulation algorithm, so every time we do it right, we just can store it off to the side.

So when we're doing simulations, we don't need that the simulation takes low space. We just need that the input-- these decoding algorithms doesn't have to be low space, we just need that the encoding was small. We've already made the encoding small. And so the decoding algorithm can spend lots of time and space, we just need to show that decoding algorithm can recover what it's supposed to recover. It's like a compression algorithm, to show there's some way to decompress, could take arbitrarily amount of time and space.

So when we're simulating the right subtree, and we simulate not only the sum queries, but also the updates. So whatever gets written during that simulation, we just store it, and so it's easy to reread it. If it was written in the left subtree, well, that is r intersect w. And we've written down r intersect w. So we can detect that this happened, because we look at r intersect w, we see, oh that word was in there, that address was in there, and so we read the contents from the encoding.

If it was in the past, it's also easy. We already know it. OK, so basically what we do-- what the simulation algorithm is doing is it says, OK, let's assume that main memory was whatever it was at this point. That data structure, I mean we know everything about the past, so we know what the data structure looked like at this moment, store that. Update all of the cells that are in r intersect w given by our encoding. And then just run the algorithm.

So we're sort of jumping into this moment in time with a slightly weird data structure. It's not the correct data structure. It's not what the data structure will actually look like at this point, but it's close enough. Because anything that's read here, either was written here, in which case it's correct, or was written here, in which case it's correct because r intersect w had it. Or isn't it written here, in which case-- maybe it's always correct. No, no.

See there could be some writes that happened here, where there's no corresponding read over here. So the data structure may have been changed in ways here that don't matter for this execution of the right subtree. So any rights that happened here to some cell probe, to some cell, where that cell is not read over here, we don't care about, because they don't affect the simulation. So we have a good enough data structure here, it may not be completely accurate, but it's accurate enough to run these queries.

Once we run the queries, the queries output the sums. That's what we're assuming in this warm up, we run the query, we get the sum. Once I have that sum, as I argued before, once you know what the results of these queries were, I can figure out what the arguments to the updates were, by doing that inverse multiplication stuff. So that's actually it.

What this implies is that this is a correct encoding, which means that this order, r intersect w times log n bits that we use to encode, must be at least this big. Because we know any encoding is going to require at least that many bits, l root n log n. And so the log ns cancel, and we're left with r intersect w is at least l root n. And this is exactly the quantity we cared about for this claim. So same thing, r intersect w is at least l root n.

OK, so warm up done. Any questions about the warm up? So in this weird problem, which does not correspond to dynamic connectivity, because it's this other problem, prefix sums computation. We get the intended lower bound, you need log n per operation. Or you need root n log n per block operation.

OK, but this is not what we really want, we really want a lower bound on verify sum. Where you're given as an argument the permutation that we're talking about over here. So this goal is not the right goal for verify sum, in some sense. Well, sort of the right goal. It's a little awkward though, because they're given as inputs to the queries. So what is there to encode? Well, we can still set it up in a useful way. Same goal, slightly restated.

So this is the last step to verify sum lower bound.

So here's the set up.

OK, so slightly different set up here. Here I assumed that we just knew the past. I also basically assumed these two things, that we didn't know what the update permutations were in the left subtree, and we didn't know what the answers to the queries were in the right subtree. Now I'm going to assume we don't even know what we're passing into the queries, because that is the information we're trying to figure out.

These two things are basically the same, if you knew all the update permutations, you could figure out all the query permutations. If you knew all the query permutations, you could figure out all the update permutations. That's what we argued over here, it's enough to figure out query permutations, then we could figure out the update permutations.

It's just a little more awkward, because now there are arguments to queries. And so if we did this simulation, right? We'd simulate-- we don't know how to simulate the query algorithm, because it's supposed to be, given the argument, which is what we're trying to figure out. So we can't simulate the query algorithm. It's kind of annoying, but otherwise the set up is roughly the same.

The one thing we know is that the query is supposed to return yes, because if you look at this bad access sequence, it is designed to always return yes. So that is a thing we know, but we don't know the arguments to the updates on the left, we don't know arguments to the updates on the right. We'll assume we know everything else, basically, up to this time.

Again, this is a probabilistic statement, that conditioned on the past, conditioned on the queries on the left, which probably don't matter, conditioned on the updates on the right, which do matter, but they're sort of irrelevant to this r intersect w issue. Conditioned on all those things will prove that the expected number of operations you need to-- or expected encoding size, for this problem, is at least what it is, l root n log n bits. And from that lower bound, you can then take the sum over all possible setups, over all conditions. And that implies a lower bound on the overall setting without these assumptions. OK?

So all I'm saying is in this set up, it still takes a lot of bits to encode these updates, because we don't have the queries which would tell us the answers. So we get a lower bound on encoding these updates, or a lower bound on encoding these queries, because we assume we don't know them. The rest of the-- all the remaining operations don't tell us enough about this. OK.

So how the heck are we going to do-- prove a lower bound in this setting, when we can't simulate the query algorithm? There's one cool idea to make this work. You may recall our last cell probe lower bound for the predecessor problem. Use this idea of round elimination. The idea with round elimination was-- Alice is sending a message, Bob was sending a response. But that first m-- we set things up, we set up the problem so the first message sent by Alice had, on average, less than 1-bit of information to Bob, or very little information to Bob. And so what Bob could do is basically guess what that message was. And that would be accurate with some probability.

Now here, we're not quite allowed to do that, we're not allowed to change the accuracy of our results, because of our particular setting. So we can't afford to just guess by flipping coins what we were supposed to know. What we're supposed to know here is-- we're trying to simulate a query operation, and so we need to know the argument, that whole permutation to the queries.

It's hard to run it without that permutation. So instead of guessing by flipping coins, we're going to guess in the dynamic programming sense, which is we're going to try all the possibilities. Run the simulation over all possible queries, all possible second arguments to the query. We don't know what the presentation is, so just try them all. Cool thing is, only one argument here should return yes. That's the one we're looking for. So if you try them all, find which one says yes, we'll be done.

So this is called the decoding idea. Simulate verify sum of i comma pi, for all pi. And take the one that returns yes, that is our permutation. And so if we figure out what those query permutations are, then we figure out what the update permutations are, and we get our lower bounds just like before.

OK. This is easier said than done, unfortunately. We'd like to run the simulation just like here, so simulate inquiry algorithm. They said, OK, still the case, that if you're reading a cell that's either in the left subtree, in the right subtree, or in the past. And we said this was easy, this was known. And the hard part is this case, because if we're running this query, and it reads something that was written in the left subtree, it may not be in r intersect w. Why is that? Little puzzle for you. So we're running one of these queries for sum pi. And I claim that when we read something in the left subtree, we don't know if it's in r intersect w, it might not be.

Let's see if we're on the same page. So r is the set of cells read during the right subtree when executing these operations. OK? But what we're doing now is simulating some executions that didn't necessarily happen. We're doing a verify sum of i comma pi, but in the bad access sequence, we did verify sum of i comma something specific, not any pi, but the correct pi. So we only ran the yes verify sums, and that's what r is defined with respect to. r is the set of things that get read during these operations, where the verify sum is always output yes.

If you now run a verify sum where the answer is no, it may read stuff that the other verify sum didn't read maybe. Shouldn't matter, but it's awkward, because now it's not just r intersect w we need to encode. We need to encode some more stuff. It's basically a new r prime that may happen during these reads, and we just can't afford to encode that r prime, because it's not the thing we care about. We care about what happens in the actual access sequence, not in this arbitrary simulation.

So this is the annoying thing. Trouble. If you look at an incorrect query, meaning the wrong pi, this is like a no query, the output's no. Reads some different set of cells, r prime, which isn't the same thing as r. And so if-- we have some good news, which is if we can somehow detect that this happened, that we read something that is in r prime, but not r, then the answer must be no.

So that's our saving hope, is that either we're reading something at r intersect w, in which case it's been written down, we know how to do it. What's not written there, and if it's not written there, then it should be, hopefully, in r prime minus r. So the answer should be no. Maybe.

Slight problem, though, because we used r intersect w to detect what case we were in. If we were in r intersect w, then we knew we should read from those encoded cells. If we weren't, we were either in the past or in the right subtree, these things were easy to detect, because they got written during the simulation. But we need to distinguish between-- did we read something that was in the left subtree, or did we read something that was known? This is a little tricky, because this gets at exactly the issue. Left subtree might write some stuff that didn't get read by verify sum. So now you go to read it, you need to know, am I reading something that was not in r intersect w? And therefore-- Yeah.

Basically the issue is, is it in w? If it's in w, but not in r intersect w, then I know the answer is no, and I should stop. If it's not in w though, that means it was in the known past, and then I should continue. How do I know if I should stop or continue? So this is the tricky part.

We can't tell whether there's the weird notation. We want to know whether r is in w minus r or past minus r intersect w. OK, we can tell whether it's in r intersect w, if it is, we're happy. If it's not in r intersect w, it could be that's because it was just in some past thing we were reading, that didn't get read otherwise. Or it could be we're reading something that was written in the left subtree, but not read in the right subtree.

So in this case, we want to abort. And in this case, it's known, and so we just continue. So that's what the simulation would like to do, if we could distinguish between these two cases. But right now, we can't distinguish between these two cases, because we don't have enough information. So we're going to make our encoding a little bit bigger.

What we're going to do-- this is here-- is encode a separator for r minus w and w minus r. So let's-- over here.

What does this mean? Separators going to call, called S. So I want this picture, r minus w sits inside S. And w minus r sits outside S. This is my universe of cells. These are the things that are read in the right subtree, but not written in the left subtree. Those are the things I care about-- well, no quite this, the other ones. So things that are read in the right subtree and that are not written in the last, this is the past essentially, that's useful over there. Over here, I have w minus r, these are things that are written in the left subtree, but not read in the right subtree. These are the things that I worry about, because those ones I need to detect that that was changed, and say whoops, you must have an answer of no.

OK? So I can't afford to store these sets exactly, so I'm going to approximate them, by saying, well, let's store the separator out here. And if you're in S, then you're definitely not in w minus r. If you're definitely not in w minus r, then you can run-- you can treat it as if it was known. OK, so if you're in s, this would be-- why don't I write it here. For the decoding algorithm, if you want to read a cell that is written, or last written in the right subtree, in the past, these are the two easy case. Sorry-- I don't want to write what's in the past, because the whole point is to figure out what's in the past.

The other easy case is if it's in r intersect w, then it's written down for us. So this is encoded.

This is easy, because during the simulation we did those rights, and so we know what they were. r intersect w, we've written down, so it's easy to know. Then the other cases are either you're in S, or you're not in S. OK. I claim if you're in S, you must be in the past, that cell must have been written in the past, and so you know what the value was. And so you can continue writing the simulation, just like in this situation.

The other situation is you're not in S, then you don't know, it could have been written or might not have been. But what you know is that you're definitely not in r. Because if you're not in r minus w, and you're not in r intersect w, then you're not in r. If you're not in r, then we're in this situation. If you read something not in r, that means you're running the wrong query. Because the correct query does r-- only reads from r. So if you're not an S, you must not be in r. And so in this case, you know you can abort and try the next pi.

So we're going to do this for all pi, run the simulation according to this way of reading cells. At the end, the queries are either going to say yes or no, or it may abort early. So if it says no or it aborts early, then we know that was not the right pi. Only one of them can say yes, that tells us what the pi is, that tells us what the queries were. Once we know what the queries were in the right subtree, we can use the same multiplying by inverses trick, figure out what the updates were in the left subtree. But those permutations require l root n log n bits. Which used to be on this board, it's been erased now. That's what we use for this argument.

And so what we get is overall, encoding must use l root n log n bits. OK, but our encoding's a little bit bigger now. The big issue is how do we store the separator? We need to do store this separator with very few bits, otherwise we haven't really proved anything. We want encoding to be small. So we get that the encoding must use omega l root n log n bits in expectation, because this is a valid decoding algorithm, it will figure out what the permutations were. And they require at least this many bits, so encoding must use this many bits in expectation.

Now the question is how many bits does the encoding use? Then we'll get either a contradiction or we'll prove the claim. So let's go over here.

So here's a fun fact about separators. I'm not going to prove it fully, but I'm going to rely on some hashing ability. So given some universe U, in this case it's going to be the cells in our data structure. But speak a little bit more generally of the universe U, I have some number m, which is our set size. And what we're interested in is in defining our separator family. Kind of like a family of hash functions, closely related, in fact. Call it S. And it's going to work for size m sets. And so S is a separator family if, for any two sets, A and B, in the universe of size, at most, m, and disjoint.

So A intersect B is the empty set. So of course what we're thinking about here is r minus w, and w minus r. These are two subsets of the universe. Hopefully they're not too big, because if this one is huge, that means you read a huge amount of data in the right subtree. If this one is huge, it meant you wrote a huge amount of data in the left subtree. And then we get lower bounds in an easier way. Or they're not so big, let's say they're size at most m. They're disjoint for sure, by definition, r minus w's disjoint from w minus r. It removes the intersection. So that's our set up.

Then, what we want, is that there is some set C in the separator family, such that A is contained in C, and B is outside of C. So B is in the universe minus C. So this is exactly our picture from before, we have A. A contains C, and we have B over on the right. And this is the whole universe U, and so B is outside of C, A is entirely inside C. OK.

This is what we want to exist, because if a separator family exists, then we know whatever our r minus w, and w minus r sets were, as long as they're not too big, they're definitely disjoint, we can find one of these separators that encodes what we need to encode, which is the set C. Which is called s over there. Cool. How do we encode it? Well, if the number-- if the size of the separator family is something, then we need log of that bits to write down the separator set. So as long as this is small, we're happy.

So let me tell you what's to know about separators There exists a separator family S, with size of S at most 2 to the order m plus log log U. Now this is getting into an area that we haven't spent a lot of time on, but-- so I'm going to give you a sketch of a proof of this claim. Relying on perfect hash functions. So the idea is the following, we want to know, basically, which elements are in A, which elements are in B. But it's kind of annoying to do that, it can't start that for all universe elements.

So if we could just find a nice perfect hash function that maps the elements of a and the elements would B to different slots in some hash table, then for every slot in the hash table we could say, is it in A, or is it in B? Now if you are not in A union B and you hash somewhere, you'll get some bit, who knows what that that stores. I don't care. For the things outside of A union B, they could be in C or not in C, I don't really care. And so all I care about is if A and B have no collisions between each other, I don't want any A thing to hash to B thing. Then I can store a bit in every cell in the hash table, and that will tell me, in particular, A versus B. And then the rest of the items are somehow categorized, but I don't care how they're categorized.

So we're going to use this fact that there is a set of perfect hash functions of the same size. Sorry, that should be H. This is what's really true, size of H is 2 the order m plus log log U. OK, I'm not going to prove this, but this is about succinct hash functions. It may be hard to find such a hash family, but the claim is that they exist. Or it's hard to find the hash function of the family that has no collisions, but the guarantee is, as long as you have, in total, two items, out of your universe of size U, you can get a collision-free hash function, 2 to the order m plus log log U.

OK. So this is going to-- Yeah. Maps, say A union B, to an order m sized table. And here, there are no collisions. So then what we also store is an A or B bit for each table entry.

So that's our encoding. We store a perfect hash function, that's going to cost log H bits for this part, and log of H is just m plus log log U. And then we're going to store this A or B bit for every table entry. Number of table entries is order m, so this is going to take 2 to the order m bits. Or sorry, not-- sorry, in term of bits, its order m bits, I should say.

In terms of functions, it's 2 to the order m possible choices for this bit vector. And so the easy way is to just sum up these bits, you use log of H bits plus order m bits. This already had an order m term, and so you get this. The log of S is order m plus log log U. So that's the end of proof sketch of the claim. If you believe perfect hash functions can be written down in a small way, then we're done.

Now first with separators, now let's apply this separator theorem claim to this setting. So now we can compute the size of our encoding, our encoding involved writing down r intersect w. That takes r intersect w times log n, just like before. It also involves writing down the separator. Separator takes order m bits, m is r plus w. Things that are-- it's order r plus w. These are all the things-- I'm trying to write down r minus w and w minus r, so that you add up those sizes, basically r plus w. Plus log log U. U is some small thing, size of memory, number of cells in memory. We're assuming that polynomials, so you take log log of a polynomial, that's like log log n.

So let's finish this off. So before this was our equation, r intersect w times log n, that was the size of our encoding. We still have that term. Sorry, r intersect w, size of that, times log n. So we still do that. Now we also pay, for this separator, we're going to pay r plus w, that's the m part. Plus log log n. This is the number of bits in our encoding. And I claim, or what we've proved over here, is that any encoding must use l root n log n bits.

So this thing must be at least this thing. So we have a little bit more work to prove. There are now two cases. It depends-- there's basically-- and log log n is unlikely to dominate. We're doing a block operation on root n things, probably need to use at least log log n steps. So it's not really relevant. What will dominate is either this term, as it used to, or this term. These are two different cases, call them case one, case two.

In case two, r plus w is at least l root n log n. That's the lower bound we want. If we can-- in case two, r plus w is omega l root n log n. What that means is in this subtree, the amount of reading we did in the right subtree, plus the amount of writing we did in the left subtree, is at least l root n log n. That's our goal over here. We want to prove-- sorry, it's a previous claim, that's by now erased. Is the easier claim, we just want to show that the total amount of time spent in v's subtree is at least log n per operation. We're doing l root n things here. So this is a ton of reading and writing. So in that case, we're happy, because we get an actual lower bound on time.

Otherwise, we don't-- I mean, these are actual reads and writes, or total number of reads and writes. Here we're getting-- in the other case, we get r intersect w log n is at least l root n log n, just like before. So again, the log ns cancel. So here we lose the log n factor, but it's OK, because this is only talking about r intersect w. This we use the LCA charging, to say, well, if you look at a particular read, it's only gets charged by the LCA. So then we can afford to sum up large amounts.

So it's a little bit weird. In this situation, we add up all the lower bounds. Each of them doesn't give us a log n, but in aggregate, we get a log n, because every leaf appears in log n levels. In this case, we don't need to aggregate, because we just say, well, the number of operations in the subtree is at least log n per operation. This time spent, cell probe's done, is at least log n per operation. So in that case, we don't need to sum the lower bounds, which is done. So in either case, we're happy.

Little weird, because you could have a mix of cases, one vertex v could be in case two, then you just ignore all the things below it. The rest of the tree might be in case one, but you can mix and match one and two, as long as you don't use a one below a two, you're OK, you won't double count. And so in either case, we're happy, we get a log n lower bound, either on time per operation, or on this kind of time per operation. Add up all those lower bounds, you get log n per operation, or get root n log n per block operation, which implies log n per insert delete edge, or connectivity query. And that proves right there, more or less on time.

You can use the same technique to do a trade off between updates and queries. This is just log n, worst case of the two. I mentioned what the bound was last time. Same trick works, you just do more updates than queries, or more queries than updates. So we get link/cut trees are optimal, other [? tour ?] trees are optimal. And we've got lots of other points on the trade off curve, as you may recall last class. Like our log squared update is optimal for a log over log log query.

And that's the end of dynamic graphs, the end of advanced data structures. Hope you had a fun time, we got to see lots of different topics. And I hope you'll enjoy watching on the videos, and let me know if you have any comments, send an email or whatever. Yay.

[APPLAUSE]