MITOCW | watch?v=3Y2weLDiUWw

ERIK DEMAINE: All right. Today we start a new section of data structures. This is going to be two lectures long, so this week succinct data structures, where the goal is to get really small space. And first thing to do is to define what "small" means. Most succinct data structures are also static, although there are a few that are dynamic. We'll be focusing here on static data structures.

And so in general, the name of the game is taking a data structure that you're familiar with-- we're going to talk about essentially two today. One is binary tries, which has the killer application of doing binary suffix trees in particular. So it could be a compressed try, whatever.

But we'll assume hear the alphabet is binary. A lot of this has been generalized to larger alphabets. I'll tell you a little bit about that. But to keep it simple in lecture, I'm going to stick to a binary alphabet.

And another data structure we're going to look at is a bit vector, so n bits in a row. And you want to do interesting operations, like find the i-th 1 bit in constant time. These are things that are easy to do in linear space, where linear space means order and words.

Like if you want to implement a try, you have a pointer at every node, two pointers at every node, that's easy. Bit vectors you could store an array of where all the 1s are. And so it's easy to do linear space, but linear space is not optimal.

And there are three senses of small space that we'll be going for. The best version is implicit. An implicit data structure means you use the very optimum number of bits, plus a constant.

But here, we're focusing on bits, not words. And if you translate a linear word data structure, and words is order n times w bits, to store something like an n-bit vector, you should really only use order n bits, not order nw bits.

Now, what would be ideal is if you used n bits plus a constant. The plus a constant is essentially because sometimes the optimum number is not an integer. And it's hard to store a non-integer number of bits, so you add a constant. I mean, I wouldn't mind too much if you added order log n bits or something. But typically, the goal here is constant. Sometimes you can really get zero.

But the next best thing would be a succinct data structure, where the goal is to get opt plus

little o of opt bits. So the key here is to get a constant factor of 1 in front.

And then the worst thing in this regime is still pretty good. It's order opt bits. So for example, if you're storing an n-bit vector, if you use order n bits, that's still better than order nw bits. Usually, compact is a savings of at least a factor of w over what you'd normally call a linear space data structure. So this is the big savings, factor w.

But there's a constant here. Sometimes you like to get rid of that constant, make it 1. And it's not so bad if you have, say, another square root of n or something little o of n bits of extra space. But of course, the ideal would be to have no extra space, and that's implicit.

Now, it's a little confusing. I usually call this area succinct data structures to mean all three. But the middle one is called succinct. That's typically the goal is to go for succinct, because implicit is very hard, and compact is kind of like a warm-up towards succinct. So this is the usual goal in the middle. And we're going to do this for binary tries, and rank and select.

So let me tell you a little bit about what's known. Oh, sorry. One quick example.

You've seen implicit data structures. You may have even heard this term in the context of heaps. Binary heaps are an example of a dynamic implicit data structure. They achieve this bound. They have no extra space in the appropriate model.

And another one would be a sorted array. Sorted array supports binary search. You can't really update it, so it's a static search structure. It achieves the optimal number of bits. You're just storing the data.

And usually, implicit data structures just store the data, though sometimes they reorder the data in interesting ways, like sorting. A sorted array is one way to reorder your data items.

So here's a short survey. This is definitely not exhaustive, but it covers a bunch of the main results.

So one place where this area really got started-- well, there are a few places, actually. My academic father, one of my PhD advisors, Ian Munro is sort of one of the fathers of this field. And he started looking at specific data structures at the very early days.

And one of the problems he worked on was dynamic search trees. So if you want to do static search trees, you can just store the items in a sorted array, easy log n search, no extra space.

What if you want to do inserts and deletes also in log n time per operation, just like a regular search tree, but you want to do it implicitly? Now, this is tricky. And there's an old result that would let you get log squared per update and query, which essentially encoded-- you can't afford pointers at all here.

But the idea was to encode the pointers by permuting enough items. If you take, say, log n items, then the permutations among them is roughly log n, log log n bits. And so you can encode bits by just permuting pairs of items. And so you could read a pointer in like log n operations, you end up with log squared.

And then Ian Munro got down to a little bit less than log squared. And then there's a series of improvements over the last several years.

And the final result is log n worst case insert, delete, and predecessor. And this is by Franceschini and Grossi. And furthermore, it's cache oblivious, so you can get log base b of n cache oblivious. So this has been pretty much completely solved. Implicitly, you can do all the good things we know how to do with search trees.

Now, this is not trying to solve the predecessor problem using Van Emde Boas and such tricks. That's, I believe, open. But for a basic log n performance, it's solved.

Before this got solved, another important problem is essentially the equivalent of hashing. So you want a succinct dictionary. You want to be able to do-- now, this is going to be static, so there's no insert and delete. It's just, is this item in the dictionary?

So I have a universe of size u. I have n items in the dictionary. So the first question is, what is the optimal number of bits? And this is actually usually very easy to compute. You just take, what are the set of possible structures you're trying to represent, which is n items out of a universe of size u? How many different ways are there to have n items in a universe of size u?

Come on. It's easy combinatorics. Somebody?

AUDIENCE: Log u of n.

ERIK DEMAINE: Log u--

AUDIENCE: Sorry. That's how many bits you'll need.

ERIK DEMAINE: Yeah. Log u choose n is the number of bits you'll need. The number of different possibilities is u choose n. You take log base 2 of that, that's how many bits you'll need to represent this.

Now, this is not necessarily an integer, because it's got a log. That's why we would have plus order 1 if we were doing implicit. It's not known how to do implicit. It's known how to do succinct, so this is going to be plus little o of that thing.

And actually, I have the explicit bound. Eraser. I don't know how exciting this is, but you get to know how little o it is.

Log log n squared over log n. So this is slightly smaller, so this is roughly something like u log n. It depends. If n is small, this is like u log n. If n is big, this is like-- sorry, n log u, I should say.

You can always encode a dictionary using n log u bits. Just for every item, you specify the log u bits. But when n is big, close to u, then you can just use a bit vector and use u bits.

And so this is little o of n, so it's always smaller than whatever you're encoding over here. It's only slightly smaller than n, log log n squared over log n, but it's a little o of 1.

And that's I believe the best known. So this is Brodnik and Munro, and then improved by Pagh. And the point is you get constant time membership query. In general, the name of the game is you want to do the queries you're used to doing in something like a dictionary, same amount of time, but with less space.

OK. Next one is-- maybe I'll go over here-- binary try, which is what we're going to work on today. There's various results on this, but sort of one of the main ones is by Munro and Raman. Again, now this is a little harder of a question. How many binary tries on n nodes are there?

The answer, I will tell you, is the nth Catalan number, which is a quantity we've seen before. 2n choose n over n plus 1. As we mentioned last time, this is roughly 4 to the n.

This is kind of interesting. We saw the Catalan number in a different context, which is we were doing indirection and using lookup tables at the bottom on all rooted trees on n nodes. Number of rooted trees on n nodes was also Catalan number.

This is a different concept, binary tries. If you've ever taken a combinatorics class, you see Catalan numbers all over the place. There's a zillion different things, all of them the number of

them is Catalan number. And we will actually use that equivalence between binary tries and rooted trees at the end of today's lecture. So you'll see why they're the same number.

OK. So we take log of that, that's 2n bits. So you need 2n bits to represent a binary try. And indeed, you can achieve 2n plus little o of n bits.

So that's a succinct data structure. And our goal will be to be able to do constant time traversal of the tree, so left child, right child, parent. And for fun, another operation we might want to do is compute the size of the current subtree.

Again, think of a suffix tree. You start at the root and you want to be able to go along the left child every time you have a 0 bit in your query string. You want to go to the right child every time you have a 1 bit. Parent we don't really need, but why not?

And then subtree size would tell us, how many matches are there below us? You could count either the number of nodes below you or the number of leaves below you. It's roughly the same for a compact try.

And so this lets you do substring searches. And we'll actually talk more about that next lecture, how to actually do a full suffix tree. But this is a component of a binary suffix tree that has the same performance but uses optimal amount of space.

And this is a big motivator originally for doing compact or succinct data structures. At University of Waterloo, they were doing this new OED project, where Oxford English Dictionary was trying to go online or digital. And this concept of having a CD-ROM that could have an entire dictionary on it was crazy.

And CD-ROMs were really slow, so you don't want to just scan the entire CD to do a search. You want to be able to do a search for arbitrary substrings. That's what suffix trees let you do. But you really can't afford much space. And so that was the motivation for developing these data structures back in the day.

Things are a lot easier now. Space is cheaper. But still, there's always going to be some giant thing that you need to store that if you want to store a data structure, you really can't afford much space.

Cool. So that's static. There is a dynamic version. It's a more recent result from just a few years ago.

You can do constant time, insert, and delete of a leaf. And you can do a subdivision of an edge. These are operations we saw for dynamic LCA.

So same operations as dynamic LCA, you can do these in constant time and still maintain the succinct binary try representation, where you can do all these traversal operations. So we won't cover that. We will cover the static version today.

Then that's for binary alphabet. Let me tell you what's known about larger alphabet. This is a problem I worked on a while ago, though our result is no longer the best.

So here, it's a little complicated. But the number of tries, number of k-ary tries, so k is now the size of the alphabet, is kn plus 1 choose n over kn plus 1. And so it's succinct, meaning we can achieve log of that plus little o of that bits. And the queries we can achieve are constant time, child with label i, and parent, and subtree size. So we can do all the things we were able to do before.

The analog of left child and right child now is I have a character in my pattern, p, and I want to know which child has that label. So it's not the same as finding the i-th child of a node. The edges are labeled by their letter in the alphabet, and you can achieve that. We had an earlier result that achieved like log log or something, and then finally it was brought down to constant by Farzan and Muno.

So couple more. Why don't I just mention what they are? It's getting a little tedious.

There's a lot of other things you might want to store. You can store succinct permutations, although there are some open problems here. So you want to store a permutation using log n factorial bits, plus little o of n.

If you want to achieve succincts, the best known-- oh, and the interesting query is you want to be able to do the k-th power of your permutation, so see where an item goes after k steps. Best query known for that is log n over log log n if you want succinct.

If you only want compact, then it's known how to do constant time. So an interesting open question is, can you achieve succinct constant time queries? If you relax either, then it's known how to do it. There's a generalization of this to functions, where one of those results is known, the other one isn't.

You can try to do Abelian groups. There are finite Abelian groups. There aren't too many

different ones, and you can represent an entire Abelian group on n items using log n bits, which is pretty crazy, order log n bits. And you can represent an item in that group in log n bits and do multiplication, inverse, and equality testing.

There's other results on graphs, which I won't get into. Those are a little harder to state.

And then another interesting case is integers. So you want to store an integer and you want to be able to increment and decrement the integer. And you want to do as few bit operations as possible.

Worst case, for example, if you have lots of 1s in your bit string, you do an increment, you don't want to pay linear costs, linear number of bit updates to do it. And so you can achieve implicit, so just a constant number of extra bits of space. If I have an n-bit integer, then I can do an increment or a decrement in order log n bit reads, and constant bit writes. And this is Raman and Munro from a couple years ago.

So this is pretty good. Of course, ideal would be to do a constant number of-- well, a constant number of bit reads or writes would be optimal, I guess. I personally would be interested in getting a constant number of word reads and writes. But that's an open problem, I believe.

So there's only order n bit reads. If they were all consecutive, that would be a constant number of word reads, but they're kind of spread out. It'd be nice to get constant number of word operations.

So that's a quick survey of what's known.

Let's do some actual data structures now. So we're going to be focusing on this synced binary tries. And we're going to do two versions of it. One of them is level order. And the other will use a balanced parenthesis representation.

So let's start with the level order one. This is very easy. So I'm going to just loop over the nodes in my try in level order, so level by level, and write one bit to say whether there's a left child and one bit to say whether there's a right child.

Let's do a little example. So here is a binary try. And I'm going to write a bit string for it. So first, I'm going to look at the top level. I have the node A. It has a left child of B, right child of C, so I write 1, 1. And this corresponds to B. This corresponds to C.

OK. Then next level is B. It has no left child and it has a right child, which is D. I'm just writing down the labels so I don't get lost.

Then we have node C, which has a left child and a right child, E and F. Then we have node D, which has no left child. It has a right child, which is G.

Node E has no children. Node F has no children. Node G has no children.

OK. So there is a 2n-bit string. This is obviously 2n bits for n nodes, so this is one way to prove there's at most four to the n tries. And well, we'll talk about how useful it is.

I want to give you another representation of the same thing, which is if we take these nodes and add on-- wherever there's an absent leaf, I'm going to add what we call an external node, as you sometimes see in data structures books. One way to represent a null pointer, say, oh, there's a node there that has no children. This unifies things a little bit, because now every node either has two children or no children. Another way to think about the same thing.

And it turns out if you look at this bit string and add a 1 in front-- so I'll put this one in parentheses-- then what this is encoding is just for every node in level order, are you a real node or are you an external node? Are you an internal node or an external node?

A here is internal. B is internal. C is internal. This is an external node. Then this is internal, internal, internal.

Then external, G, external, external, external, external, external, external. The zeros correspond to external nodes because those are absent children.

So same thing. So I'll write equivalently, 1 equals an internal node and 0 equals an external node.

Of course, to do this, we need one more bit. And I'm going to take this view primarily, because it's a little easier to work. It doesn't really make much of a difference, just shifts everything over by 1. And I'd like to write down the indices here, so we have 1, 2, 3, 4, 5, 6, 7 into this array.

Because now our challenge is all right, great. We've represented a binary try. But we want to be able to do constant time, left child, right child, parent. We're not going to be able to do subtree size. Level order is really not good for subtree size. But left child, right child, and parent we can do in constant time.

And the reason that we can do it in constant time is because there's a nice lemma, kind of analogous to binary heaps.

So the claim is if we look at the i-th internal node, so for example C is the third internal node, so internal node number 3, then we look at positions 2 times 3, and 2 times 3 plus 1, so 6 and 7 in this array. And we get the two children E and F. So that worked. 6 and 7 are E and F.

Or this one would be the fourth internal node. D is the fourth internal node. And so at positions 8 and 9 should be this external node and G. 8 is an external node. 9 is g. So it works.

This is a lemma. You can prove it pretty easily by induction on i. So the idea is, well, if you look at let's say the i-th internal node and the i minus first internal node, this one has two left children. Don't know whether they're internal or external.

Between them-- they're on the same level and we're in level order, so anything in between here is an external node. So they have no children, which means if you look at the children of i, they're going to appear right after the children of i minus 1, because we're level order.

So we have these two guys. The next nodes at this level are going to be these two guys. So this one appeared at 2i minus 2, and 2i minus 1. This one will appear at 2i and 2i plus 1.

This is if i and i minus 1 are on the same level, but it also works if they're in different levels. So if I'm i minus 1 is the last node on its level, again it's going to have two children here, which will be the last nodes on this level. And then right after that will come the children of i, again at position 2i and 2i plus 1.

OK. So that's essentially the proof that this works out. There's lots of ways to see why this is true, but I think I'll leave it at that.

OK. So this is good news. It says if we have the i-th internal node, we can find the left and the right children. But these are in different namespaces, right?

On the one hand, we're counting by internal nodes. On the other hand, we're counting by position in the array, which is counting position by internal and external nodes. This counts both 0's and 1's. This only counts 1's.

So we need a mechanism for translating between those two worlds, translating between

indices that only count 1's and indices that count 0's and 1's. And this is the idea of rank and select.

So in general, if I have a string of n bits, I want to be able to compute the rank of a bit, which is the number of 1's at or before position i. So I'm given a position like 6 and I want to know how many 1's are there up to 6, which would be 5 in the full array here. Or I'm given a query like 8, number of 1's is 6 up to position 8. And then the inverse of rank is select, which gives you the position of the j-th 1 bit.

So this lets you translate between these two worlds of counting just the 1's, which is rank, or going to the j-th 1 bit, that's select. So this lets you compute the left child as just being at position twice the rank, because the rank tells you this value i, which is which internal node are you. That's your rank. You multiply by 2 and that was the position of the left child.

Right child is going to be that plus 1. Parent is going to use select. So if we want the parent of i, this is going to be select of i over 2 with a floor. So that's just the inverse of left and right child. If I divide by 2 and take the floor, I get rid of that plus 1, I get the rank. And then I do select, sub 1, and that's the inverse of rank.

So that lets me implement. If I have rank and select in constant time, now I can do left child, right child, parent in constant time. The remaining challenge is, how do I do rank and select? And that's what we're going to do next. Any questions about that before we go on?

All right. So now we do real data structures. This is going to be some integer data structures, some fun stuff. It's going to use some techniques we know, but in a different setting, because now our goal is to really minimize space. We're going to use indirection and lookup tables in a new kind of way.

These are going to be word RAM data structures. I want to do both rank and select in constant time. And the amount of space I have is little o of n.

I want succinct. And I'm going to store the bit vector. So then in addition to the bit vector, all I'm allowed for rank and select is little o of n space. That's the cool part.

So rank is one of the first succinct data structures. It's by Jacobson, 1989. So a first observation is, what can we do with a lookup table? Suppose I wanted to store all the answers, but I can't afford much space.

Well, let's do sort of a worksheet. If I had space x, or if I looked at all bit strings of length x and then I wanted to store them, that's going to cost-- or store a lookup table for each of them, it's going to cost-- well, there's 2 to the x different bit strings of that length. Then for each of them, I have to store all possible answers to rank and select queries. So there's order x different queries. You could query every bit.

And then for each of them, I have to write down an answer. So this is going to be log x bits, because the answer is a value between 0 and x minus 1, so it takes log x bits to write it down. So this is how much space it's going to be to store the answer for all bit strings of length x.

So what should I set x to? I'd like this to be little o of n, so anything that's a little bit less than log n is going to be OK. And in particular, 1/2 log n is a good enough choice. If we use 1/2 log n bits, this is going to be root n, log n log log n, which is little o of n, quite small as succinct data structures go. We're going to use more space than root n.

The point is, if we could get down to bit strings of logarithmic size, we'd be done. But we have a bit string of linear size. So how do we reduce it? Something like indirection.

So the funny thing here is we're going to do indirection twice, kind of recursively, but stopping after two levels. The first level of indirection is going to reduce things down to size log squared. So we're going to take our n bit string, divide into chunks of size log squared n, so there's n over log squared n chunks. They look something like this, log squared n.

And the idea is right now we're trying to just do rank, so rank number of 1's at or before a given position. So what I'm going to do is that each of these vertical bars, I'm going to store the cumulative rank so far.

Why log squared? Basically because this is what I can afford. To store that cumulative rank is log n bits. I mean, this rank is going to get very big. By the end, it will have most of the 1 bits, so it could be potentially linear. So I'm going to need log n bits to write that down.

But how many of these vertical bars are there? Well, only n over log squared n of them. So I have n over log squared n things I need to write down. Each of them is log n bits.

So do some cancellation. This cancels with that. We have n over log n bits overall, which is

slightly little o of n. And that's the bound we're going to achieve.

OK. Of course, now we have to solve the problem within a chunk. But we've at least reduced to something of size log squared. Unfortunately, we need something of size 1/2 log n before we can use a lookup table.

So there's a bit of a gap here, so we're going to use indirection a second time. This time, we can go all the way to 1/2 log n. So I'll use red vertical bars to denote the subchunks.

Each of these is 1/2 log n. Overall size of a chunk here is log squared n. So every one of these chunks gets further divided.

Now, how could this help? Why didn't I just subdivide into chunks of size 1/2 log n before? I mean, why I couldn't do it is clear. If I did n over log n of them, each of them stores log n bits, I'd have a linear number of bits. I can't afford a linear number of bits. That would only be compact, not succinct.

How does it help me to first reduce to this and then reduce to this? Well, what I want to do at each of these red vertical bars is store the cumulative rank, but not the overall cumulative rank. I only need the cumulative rank within the overall chunk, not relative to the entire array.

Why does that help me?

AUDIENCE: Need less bits.

ERIK DEMAINE: Need less bits. These ranks can't get too big, because the overall size of a chunk is just log squared. Log of log squared is log log n. So I only need log log n bits to write down those cumulative ranks.

And so total size here is going to be n over log n times log log n bits, because there's n over log n of these red vertical bars. Each one I only need to write log log n bits. And this is slightly little o of n. It's actually a little bit bigger than this, but still little of of n. So we can still afford this.

And now we're done, because these subchunks are of size 1/2 log n, so I can use this lookup table and solve my problem. So let me step forward, just putting everything together.

To compute the rank of a query, first thing you do is figure out which chunk you fall into, which you can do by division, integer division. These things are stored in an array, so you just

compute, what is that cumulative rank? So you take the rank of that chunk, you add on the rank of the subchunk within the chunk, and then you add on the rank of the element in the subchunk.

So rank of the chunk is stored in the array known as 2. The rank of the subchunk within the chunk is stored in the array, in the array known as 3. And then to compute the rank of the element in the subchunk, you use the lookup table, which is essentially telling you for every possible subchunk what the answers are. So 3 times a constant is a constant. And we get rank, constant time, and n log log n over log n space.

If you're concerned that n times log log n over log n is not very sublinear, you can do a little bit better using fancier tricks. Namely, you can achieve n over log to the kn space. This is the result of Patrascu from 2008.

I'm not going to go into how it's done. But if you're interested, it's a little bit less. It would be nice to do better. But my guess is there should be a lower bounds, that with constant-- so this is for any constant k.

It would be nice to do better, like square root of n or something. But my guess is there's a matching lower bound. I don't think that's known.

OK. So that was rank. Our next challenge is to do select, the inverse. And select is a little bit harder, I would say. Don't have a great intuition why. But it is.

And we're going to be able to use the same kind of technique. So again, we can use a lookup table and-- I'll do that first.

So just like before, if we have bit strings of length at most 1/2 log n, then we're only going to need something like root n space. It's root n again times log n log log n space, just like rank.

There are at most n possible queries, actually fewer, because there may be fewer 1 bits. But at most, there are n 1 bits to query. An answer is now an index, which is within a thing of size 1/2 log n. So I just have to write down an index of that size, Soit's it's log log n bits to write it down.

Cool. So that's the same. Now the challenge is about getting down to 1/2 log n bits. We're going to use the same technique of two levels of indirection. But they work differently. There's

an extra thing we need to deal with in select.

There will be two cases, depending on whether your array has lots of 1's or not so many 1's. And those two cases can vary throughout the string. So what we do, first of all, is-- actually, maybe I'll go over here. I'll stick here. Whatever.

So we're back to an n-bit string. So we're looking at we want the analog of this structure, this structure of chunks. Now, we can't just say, oh, take the bit string, divide it into chunks of equal size, because then given a query, we want to do select of j, we need to know which of these chunks j belongs to.

So instead of making them equal size, we're going to make them have an equal number of 1 bits. So then we can just take j, divide by the size of these chunks, which is log n log log n. You could probably do log squared as well, but log n log log n is a slightly better choice.

And so we just divide every log n log log n 1 bit, put a vertical bar. That way, given j, we divide by this thing, take the floor, that tells us which chunk we belong to. So it's different. Decomposing by 1 space instead of 01 space.

And so for those guys, we just store an array. If your query happens to have a 0 mod this, then you have your answer. Otherwise, you still need to query within the chunk. In some sense, the array has gotten divided something like this, so the number of 1 bits in here is always the same, log n log log n 1's.

So you can now teleport to the appropriate chunk. And the issue is, how do I solve a chunk? But now chunks have different sizes, which is kind of annoying.

That's why we need this extra step, which is within a group of log n log log n 1 bits-- I'm calling them groups now, instead of chunks. So each of these groups has different size. Let's suppose it has size r, so say it's r bits long. R is going to be different for each chunk, but we'll do this for every chunk, every group.

Then there's two cases. If r is big, we're done. How big? Well, if it's at least the square of the number of 1 bits, that means it's very sparse. Only square root of the bits are 1's. The rest are all 0's.

But then, I can afford to just store all the answers. I'm just going to store a lookup table of all the answers if it's very sparse, because then I claim I only need this many bits in order to store

all of these answers.

So if I do this for all groups that have a large number of bits, I store this lookup array, how many-- if I sum up the size of all of these arrays, how much do I pay? Well, the lookup array has this kind of size. There are log n log log n 1 bits. And each of them I need to store an index for them.

Now, this could cost log n bits, because potentially one of these groups is very large. It could be almost linear size. So I need log n bits to write down a position in there. There's log n log log n 1 bits to write down to position for. So this is the size of one of these arrays.

Now, how many of these could I possibly need to store? Well, I know that this group has log n log log n squared bits in it, so the maximum number of such groups is n over that, n over log n log log n squared.

And now we get to do some cancellation. So this 2 cancels with this log n log log n. And then this log n cancels with this log n. And so we get n over log log n bits, which is slightly little of of n.

OK. Again, it is possible to get n over log to the k space. But we won't do that here. We'll be happy enough with n over log log n.

OK. But we're not done, unfortunately. So we've now reduced in two groups, and I've only given you one case. This is when r is large.

The other cases, r is small, meaning the number of bits in the group is at most log n log log n squared. That's a good case for us, because that's pretty similar to rank. Here we got chunks of size log squared. Here it's slightly larger than log squared, but only by a poly log log factor.

And that would correspond to this step 2 in rank. You do step 2 and step 3 here. Then we get step 2 of rank. We've reduced to poly log size chunks by getting rid of this case.

And so we have to do it again, because we have poly log size groups. But we need to get down to groups of size log, 1/2 log. So we need to do another layer of indirection.

So we get to do steps 2 and 3 again. This is what I'll call step 4. Repeat steps 2 and 3 on-- oh, sorry. I didn't say I need an else clause. Else I'm going to call this bit vector a reduced bit vector. So I've reduced to order log n log log n squared bits.

And so step 4 is on all reduced strings, all reduced bit strings. I want to do steps 2 and 3 again. Let me do it quickly. My goal is to further reduce to poly log log n. I took n bit strings and I got down to log poly log bits.

I do it again, I should get down to poly log log bits. And indeed, I can. And this is plenty small. Poly log log is way smaller than 1/2 log, so we don't even need that much of the lookup table.

Fine. So I'll call this 2 prime. I want to make this explicit, because they are slightly different, because now everything's relative to the reduced string, which is poly log.

This gets hard to pronounce, but every log log n square-th 1 bit, we're going to write down the relative index within the reduced string of size log n. So writing down the relative index only costs log log n bits, because we're in something of size log n, so writing down that index is short. We write it down for all of these, so we end up paying n over log log n squared. That's the maximum number of these indices that we need to store. Each of them we pay log log n.

So here I'm summing over all the reduced bit strings. This is an overall size. It's at most n over log log n squared that we need to store. Could be fewer if there aren't many reduced bit strings. But worst case, everything ends up being reduced, so we have this many times that many times that many bits. And we get n over log log n bits.

This is roughly following the pattern of step 2 over here. Step 2 over here didn't just have the log term. It also had an auxiliary log log term. So if you felt like it, you could make this log log n times log log log n. But it will actually give you worse space bound, so this is slightly better.

OK. Then we apply step 3 prime, which is we look at each of the groups that we've identified. And either it's big and it has lots of 0 bits, or it's not big. And in either case, we're going to be happy.

So if a group of log log n squared 1 bits has r bits, we look at each of them individually. And if r is at least the square of that, so log log n to the fourth power-- so we're losing constants in the exponents, but it's not a big deal-- then store relative-- I mean, store all the answers, but now as relative indices.

OK. Let's go over here. So how much do these relative indices cost? Again, it's at most order log log n bits to write them down. We don't know that a group is any smaller than log n, but it's

at most the original size of log n. It's only log log n bits to write each of them down.

And now we get to say, oh, well, we had to write down log log n squared 1 bits. But this can only happen n over log log n to the fourth many times. So the space is n over log log n to the fourth.

That's the maximum number of these I guess you call them sparse bit vectors, sparse groups there could be, because each of them is at least this big. The total number of them is at most n divided by that. For each of them, we have to write down log log n squared different indices for our array. And each of those indices cost log log n bits to write down.

So this is log log n to the third power. This is log log n to the fourth power. So again, this is n over log log n. You can tell I've tuned all of these numbers to come out to n over log log n bits.

OK. That was the if case. There's the else case, which is that you have reduced to poly log log size, namely then in the dense case, you have r is at most log log n to the fourth. So at this point, else you are further reduced.

When you're further reduced, you have at most log log n to the fourth bits. And at most, log log n squared of them are 1 bits. But we don't really care about that. Once we're down to a bit vector of poly log log size, we can use our lookup table and we're done.

So that's select. If you want to do a select on an index, first you figure out which group it's in by dividing by log n log log n, taking the floor. You teleport to the appropriate group using this array. Then within that group, there's a bit saying whether it was sparse or dense. If it was sparse, so lots of 0's in it, then you have a lookup table that gives you all your answers for the remainder of your query.

If it's dense, then you go over here. You know that this thing will be stored, and so you figure out which subgroup you belong to by dividing by log log n squared, taking the floor. There's an array, this thing, that teleports you to that group-- sorry, to that subgroup.

And then you apply-- then there's a bit there saying whether it was sparse or dense. If it was sparse, there's a lookup table giving you the answer. If it was dense, there's an index into the number 1 lookup table that tells you what this bit string is, because there is only log log n to the fourth bits. In fact, that is the index. Just what those bits are lets you look up into this table and solve your query in constant time, in all cases constant time. But here's a little bit more branching, depending on your situation.

As I said, select is a little more complicated than rank. But in the end, constant time, little of of n space, n over log log n, which can again be improved by Patrascu, n over log to the k for any constant k. Question?

AUDIENCE: Can you just quickly remind us how the 2 and 3 changed [INAUDIBLE]?

ERIK DEMAINE: OK. How did 2 and 3 change? You don't actually really need to change them. The big change is that you're storing only relative indices, not indices.

So before, we were storing an array of indices of every log log nth 1 bit. These were global pointers. But now after 2 and 3, we've reduced to something of size log n or poly log n. We need to exploit that here, so that we were only storing log log n bits. If we didn't do that, this would be order n bits and it would be too big.

That's really the only thing you need to change. The other thing I changed was this value. If you follow that plan, it would be-- it was also a square. So we did have to add a square here. You could also add a log log log n term here, but it won't matter. Basically you do something that works.

The square was necessary to cancel out this guy, for example. If you didn't do the square-- well, so instead of this, you could have done log log n times log log log n without the square. Then here you would have gotten n over log log log n. So you could have followed the same pattern. You'd just get a slightly worse space bound.

I tuned it here. Here we needed-- here we could not have afforded to go to log squared, if I recall correctly, though you can check that. Maybe it's a good pset question.

There's lots of choices that work here. But this is the one I find the cleanest that gets a decent bound, not the best bound, but reasonable. Other questions?

I think that's all that I changed. The sparsity definition was still a squared thing, so it was squared over here and it was squared over here. It's just the thing we were squaring was a little different.

OK. One more thing I want to talk about. So at this point, we just finish this level order representation of binary tries, because we already saw left child, right child, and parent, reduced to rank and select. We just solved rank and select in little o of n bits, so at least

statically we can do left child, right child, parent in a binary try now in constant time per operation, 2n plus little o of n bits of space. The 2n bits are to store those 2n bits that we wrote down before. So that's succinct binary tries. Done.

One mention, there are some dynamic versions. In particular, there are dynamic versions of rank and select. But the best versions that are known to do dynamic rank and select achieve something like log over log log time per operation, if you're interested in dynamic.

So this is kind of annoying. If you want to go to dynamic, either you pay more time or you don't use rank and select. But I'm not going to worry too much about dynamic. Stick to rank and select.

But there's one more thing on this list, which is a different way to do succinct binary tries. And this different way is going to be more powerful, more useful for things like suffix trees, which is what we're going to do next class. So I want to tell you a little bit about this.

The level order representation is kind of like a warm-up. It motivates rank and select. But it does not let us do subtree size. Subtree size would be nice to do, because you care about how many matches you have after you do a search down a suffix tree.

Level order just ain't going to cut it for that, so we're going to use different representation. We're still going to use rank and select a lot. And I'll generalize forms of rank and select. But it's going to be a little bit more handy.

Essentially I want to do more like a depth-first search of the try, less like a-- less level order, so more depth-first. OK. Here's our friend the binary try, same one as before. We had our binary representation of it. I'm not going to draw that here.

First thing I want to do is say, hey, look, this is the same thing as a rooted ordered tree. I already mentioned that there's the same number of them. There's Catalan of these and there's Catalan of these.

So a rooted ordered tree has a node, it has some number of children, then more nodes. The children are ordered, but they don't have labels on them. So it's a tree, not a try.

So I claim these two things are equivalent. And there's a nice combinatorial bijection between them, which you may have seen before. It's kind of a classic. But here we're going to use it for handy stuff.

Basically so binary tries distinguish between left and right. Rooted order trees do not. They just have order.

So to clean that up, I'm going to look at the right spine of the try, distinguish that, because right and left make the difference here, and then recurse. So this is the right spine of down here. This is the right spine of this subtree. Now every node lives in some right spine.

And then I'm just going to rotate 45 degrees counterclockwise. So I have A, E, G. That's my first right spine. I'm going to think of them as children of a new root node.

And then they have children below that which correspond to the right spines that hang below. So A, for example, has this right spine, B, C, D. So we have B, C, D here. E has a right spine of F hanging off of it. G has no right spine hanging off of it.

So you need to prove that this is a real bijection. Every binary try can be so converted into a rooted order tree. And it's unique, so if it's different over here, it will be different over here.

And you can convert backwards as well, if you just delete the super-root and turn all the children into a right spine or recurse. They're really the same thing. This is why there's Catalan of each of them.

OK. Now what I'd really like to get to is balanced parentheses. And while it's a little unclear how to represent a binary try with balanced parentheses, these things it's really clear how to represent a binary-- represent with balanced parentheses. Here I just do an Euler tour, which was a depth-first search visiting these things.

And every time I start a node, I'll write an open paren. Every time I finish a node, I write a close paren. Similar to representation we talked about before.

So this would be-- I'm going to need more space. This is going to be an open paren for star. Why don't I make that one really big? We start here.

Then we open the A chunk. Then we do B, which has no children. Then we do C, which has no children. Then we do D, which has no children. And that finishes A.

OK. Then we start E. Then we do F. Then we finish F. We finish E. Then we do G. And then we're done with star.

So that's a very easy transformation. Again, there are Catalan many of these balanced parens. You think, oh, there's 2 to the n of them, because each paren could be open or closed. But they have to be balanced, so it's a little bit more constrained than that. And so it ends up being Catalan of n over 2 if there's n parens, because there's 2 parens here for every node over here.

This is going to be our bit string. Open parens are 0's. Close parens are 1's. This has roughly 2n bits, 2n plus 2, I guess, for the star, relative to this n.

So basically, nodes here correspond to nodes here, which correspond to an open paren, close paren pair over here. Now, we can't afford to store these labels. Those are just guidelines to think about what you need.

So let's think about there are three things we really want here, left child, right child, and parent. This is the thing that we care about, but this is what we're going to store. So I want to translate from here to here to here. This is an exercise in translation.

So what does a left child mean here? Left child over here corresponds to-- well, I guess it goes-- in general, the left child goes to this branch, which is like all of these children pointers from A. But really, if you follow the left child, you get to B, not any of the other things on the right spine. You always get to the top of the right spine.

Top of the right spine is the left-most node in the spine here. In other words, it is the first child of a node. First child of a node, if there is one, is going to be the left child over here.

Right child is like following the spine. That's like going this way. So right child is what I would call next sibling. The next sibling to the right, if there is one, that's going to correspond to the right child, because we're just following a right spine.

OK. Parent is a little trickier. Parent is the reverse of these, so either you take your previous sibling-- but if you're here and there is no previous sibling, then you take your actual parent, because parent should walk up here. This was like going left, previous sibling, previous sibling. Parent of this guy, though, is the actual parent over here. So this is going to be previous sibling if there is one, or if there isn't one, you go to the parent.

OK. So that's easy translation. Now we need to convert these pictures into balanced parentheses pictures, which is also going to be easy in itself. But to jump all the way from binary tries to balanced parens would be pretty confusing, so that's why we have this

intermediate step.

So we want first child here. If I have a paren-- like I'm looking at A. So A corresponds to this paren and this paren.

I'm going to represent the node let's say by the first paren. Then the first child is just the very next character. You put the first child right after that open paren.

So this is really the next character if we want to find the first child. This is if it's an open paren. It could be the very next-- like if you're doing B, the very next character is a close paren, that means there are no children.

But that's how you can tell whether there's a child. If there's an open paren right after your open paren, that's your next child. That's your first child, I should say.

Now what about next sibling? So let's say again I'm at A. And I want to know the next sibling. Next sibling is E. So that's like I go to the close paren for A, and then I go to the next character.

So this would be go to the close paren for where you are right now, and then go to the next character. This is, again, if it's an open paren. If it's a close paren, then you have no next sibling. So again, you can tell whether this operation fails.

What we need is an operation given a bit string representing balanced parentheses and given a query position of a left paren, I need to know what is the matching right paren. And I'll just wave my hands and claim that can be done with the same techniques as rank and select. It's not easy. It's quite a bit harder. But you do enough of these recursions, eventually you can solve it.

OK. Last operation is parent over here, which corresponds to previous sibling, or parent over here, which corresponds to-- there are two cases. We want to move backwards, so here we're always ending with next character. So first thing we do is go to the previous character.

And there are two cases. If it's a close paren-- so let's say we're here at E, we go to the previous character. If it's a close paren, then A is our previous sibling, and so we want to do the previous sibling situation. We again find the match. We hit percent and vi and-- what's the corresponding thing in Emacs? I forget.

You go to the matching close paren-- sorry, open paren. And then there you go. You've got your previous sibling.

So if it's a close paren, then you go to the corresponding open paren. If the previous character is an open paren, then you're done. That's your parent.

So like here if you're at A, you go to the previous character and it's open paren, then you've just found the parent of A. There was no previous sibling. So in either case you end up with an open paren, corresponding to either your previous sibling or your parent. So that's left child, right child, parent. If you have this matching paren operation, you can do all of these in constant time, and little of n space beyond the 2n bits to write down that bit string.

That's not so exciting, because we just reinvented the same results we had before of doing left, right, and parent in constant time. But what we buy out of this representation is we can now do subtree size. So this is a little bit trickier.

Let me go to another board. But whereas with level representation it was impossible, now it is possible. And we're going to use subtree size I think next class, when we do compact suffix trees. So subtree size is what we want in the binary try.

In the rooted ordered tree it's a little tricky, because subtrees no longer correspond to subtrees. For example, the subtree of C consists of this subtree and this subtree, so it's really C-- over here, it's C and all of its right siblings. So this is size of the node plus size of right siblings, however many you have.

OK. So in the rooted ordered tree, it's actually kind of messy. Turns out in the balanced parenthesis it's pretty clean, because all your right siblings correspond to paren groups that just follow each other. And you want to know-- so these are a bunch of siblings here of varying size.

And we're given, say, this sibling. We want to know for this sibling, up to all the ones to the right-- so there's an enclosing parenthesis here for our parent in this representation. We want to know the length of these, so it's just-- we want to take the-- here we are at this left paren. We want to compute the distance to the enclosing close paren. So that's here. That's our enclosing close paren.

So here's a new operation. Given a paren pair, I want to compute the enclosing paren pair, these guys. That can also be done in constant time with rank-and-select-like techniques. And

then you just measure this distance and you divide by 2.

That will give you the number of nodes in here. It's half the number of paren. That will give you subtree size.

And we have a couple extra seconds, so another bonus is suppose you want to know the number of leaves in a subtree. If I recall correctly, that's something like-- instead of doing this distance to the enclosing paren, you do something like rank of-- just rank of that. Those are the number of leaves of the enclosing close paren-- this is getting notationally confusing-- minus the rank of here.

OK. So I just want to compute how many open parens, close parens are there from here to here. And so I just take the rank here, subtract by the rank here. That gives me the number of leaves in that range.

So this is a generalization of rank. Before we did rank of just a single bit. This is rank of a two-bit pattern. But two bits is not much harder than one bit. You can very easily adapt the rank structure we saw to do any two-bit pattern instead of just the one bit.

So that gives you the number of leaves in the subtree, which corresponds to the number of matches. So you can do lots of fun things like this. This representation is super powerful and we'll use it next time.