MITOCW | watch?v=NinWEPPrkDQ

ERIK DEMAINE: All right. Today's lecture's full of tries and trays, and trees. Oh, my. Lots of different synonyms all coming from trees. In particular, we're going to cover suffix trees today and various representations of them, and how to build them in linear time. Now, they are good things. Some of you may have seen suffix trees before, but hopefully, haven't actually seen most of the things we're going to cover, except for the very basics.

So the general problem we're interested in solving today is string matching. And in string matching we are given two strings. One of them we call the text T and the other one we call a pattern P. These are both strings over some alphabet. And the alphabet we're going to always call capital Sigma.

Think of that. It could be binary-- 0 and 1. Could be ASCII, so there's 256 characters in there. Could be Unicode-- pick your favorite alphabet. Then it could be ACGT for DNA. And their goal is to find the occurrences of the pattern in the text. Could be we want to find some of those occurrences or all of them, or count them.

And in this lecture, we're only interested in substring searches. So the pattern is just a string. You want to know all the places where P occurs. P might appear multiple times, even overlapping itself-- in those two positions, whatever. You want to find all the shifts of P where it's identical to T.

Now, there are lots of variations on this problem which we won't look at in this lecture, such as when the pattern has wildcards in it, or you could imagine it being a regular expression, or you don't want to match exactly, you want to match approximately, you could have some mismatches, or it could require some edits to match T. We're not going to look at those problems.

This is both an algorithmic problem and a data structures problem. If I give you this text in the pattern, I just want to know the answer. You can do that in linear time-- it's famous Knuth-Morris-Pratt, or Boyer-Moore, or Rabin-Karp. Lots of linear time algorithms for doing that.

We're interested in the data structure version of the problem, static data structure. So we're given the text up front, given T. We want to preprocess T. And then the query consists of the pattern. Imagine T being very big, P being not so big. And we'd like to spend something like order P time to do a query. That would be ideal because you have to at least look at the query and you don't really want to spend time looking at the text.

You'd also like something like order T space. We don't want the space of the data structure to be much bigger than the original text. So these are goals which we will more or less achieve, depending on exactly the problem you want. Sometimes we'll achieve this, sometimes we'll achieve almost this. But these are really nice running times and space. It's all optimal.

Before we get to that problem, I want to solve a simpler problem which is necessary to solve this one. We'll call it a warm up. And that's a good friend-- the predecessor problem, but now among strings.

Let's say we have k strings-- k texts-- T1 to T k. And now the query is you're given some pattern P and you want to know where P fits among these strings in lexical order. So a regular predecessor, but now comparison is string comparison.

Of course, you could try to solve that using our existing predecessor data structures. But they won't do very well. Even a binary search tree is not going to do well here because comparing two strings could take a very long time if those strings are long. So we don't want to do that. Instead, we're going to build a trie.

Now, tries we've seen in fast sorting lecture, when w is at least logs at two plus epsilon event. We used tries in a particular setting there. We're going to use them in their more native setting today a lot. In this setting-- again, a trie is a rooted tree. The children branches are labeled. And in this case, they're labeled with letters in the alphabet-- Sigma.

So you have a node. And let's say, we have the English alphabet-- a, b, up to z. Those are your 26 possible children. Some of them may not exist, they are null pointers. Others may point to actual nodes. That is a trie in its native setting, which is when the alphabet is something you care about.

Now, when we used tries before, our alphabet just represented some digit in some kind of arbitrary representation. The digit was made up of log to the epsilon bits. We were just using it as a tool. But this is where tries actually come from-- they come from trying to retrieve strings out of some database, in this case.

We're doing predecessor-- this is a practical problem. Like a lot of library search engines, you type in the beginning of the title of a book and you want to know what is the preceding and succeeding book title of what you query for. So this is something people care about. Although really they want us-- typically, we want to solve this problem because it's harder.

So that's a trie. Now, to make this actually work, what we'd like to do is represent our strings. So how do we use this structure to represent strings T1 to T k? We're going to represent those strings in the obvious way, which we've done many times in the past when we were doing integer data structures-- as root to leaf paths. Because any root to leaf path is just a sequence of letters, and that's a string. So we just throw them in there.

Now, to do that, we need to change things a little bit. We're going to add a new letter, which we usually present as $ sign, to the end of every string. I have an example. We're going to do four strings-- various spellings of Anna and Ann. And say, we'd like to throw these into a trie.

They all start with a. So at the root, there's going to be four branches corresponding to $ sign, a, e, and n. I'm supposing my alphabet is just a, e, n because that's all that appears here. But everything will be on the a branch. And then from there we're going to have-- let's see-- they all go to n next. So they all follow this branch.

Then one of them goes to a. These all go to n afterwards. So we've got $ sign, a we use, e, n we use. And on the a branch, we are done. This corresponds to and a, n, e. We're finished. And we imagine there being $ sign at the end of this string. So we follow the $ sign child. The others are blank. And this leaf here corresponds to a, n, a.

On the other hand, if we could do a, n, n, there's three options. We could be done. Or there could be an a or an e to follow. So if we're done, that would correspond to the $ sign pointer. That's going to be a leaf corresponding to this string here. Or it could be an a and then we're done. And then we have a leaf corresponding to Anna, a, n, n, a. Or could be we have an e next and then we're done.

OK. Not very exciting but that is the tri representation of a, n, a; a, n, n; a, n, n, a; a, n, n, e. And you can see there is exactly one leaf per word down here. And furthermore, if you take in order traversal of those leaves, you get these strings in order. And typically, if you're going to store a data structure like this, you would store these actual pointers. So once you get to a leaf, you know which word matched. So that's a trie. Seems pretty trivial. Trievial?

But it turns out there's something already pretty interesting about this data structure. How do you do a predecessor search? If I'm searching for something like, I don't know, a, n, e-- because I made a typo-- then I follow a, n, and then I follow this e branch here and discover-- whoops-- there's nothing here.

But right at that node I see, OK, well, my predecessor is going to be the max in this subtree, which happens to be a, n, a. My successor is going to be the min in this subtree, which happens to be a, n, n. And so I find what I want.

How long does it take me to do that? Well, if I store subtree mins and maxs, then I just have to walk down the tree. That will take order P time to walk down. And then, once I'm at a node, I've got to do a predecessor or successor in the node.

So there are two issues. One is, given a node, how do you know which way to walk down? And then, when you're done, how do you do predecessor in a node? It's the fundamental question. Now, this is something we spent a lot of time doing in, say, fusion trees. That was the big challenge. So this is not really so trivial-- how do I represent a node?

One way to make it trivial is to assume that the alphabet is constant size, like two. Then, of course, there's nothing to do. It's a binary trie. You look at 0, you look at 1. You can figure out anything you need to do if the alphabet is constant. But things get interesting if you imagine, well, the alphabet is some parameter, we don't know how big it is. It might be substantial.

So let's think about how you might represent a trie or the node of a trie. Let's call this trie node representation. Any suggestions? What are the obvious ways to represent the node of a trie? Nothing fancy. I have three obvious answers, at least.

AUDIENCE: Array.

ERIK DEMAINE: Array. Good. That was my number one. Any more? That's I think the most obvious.

AUDIENCE: Tree.

ERIK DEMAINE: Tree. Good. Do a binary search tree. Or?

AUDIENCE: Hash table.

ERIK DEMAINE: Hash table. Good. So for each of them we have query time and space. If I use an array, meaning I have-- let's say, for a through z-- I have a pointer that either is null or points to the child.

This is going to be really fast because they're at a node. If I want to know, I just look at that i-th letter in my pattern P. I say, oh, it's a j. So I look at the j position and I follow it.

You might wonder, how do I do predecessor and successor? Well, this is a static data structure. So for every cell, if it's null, I can store the predecessor and successor in the node.

With no more space. This is Sigma space per node. So the amount of space is T Sigma, which is not so great. But the query is fast, query is order P time.

BST. The idea of the BST is instead of having a node that has some pointers, some of which may be absent, let's expand it out into something like this. Actually, I'll use colors. This will make life a little bit cleaner in a moment. Because we are going to modify this approach.

So let's say that the pointers you care about are red. Those are the actual letter pointers you want to do. So the idea is to expand out this high degree node into binary nodes. You put appropriate keys in here so you can do a binary search. And then, eventually, you get down to where you need to go.

This structure has high log Sigma. So the query time is going to be P log Sigma. So that goes up a little bit, not perfect. But the space now becomes linear, so that's an improvement.

Ideally, we'd like the best of both of these-- optimal query time, optimal space, linear space. And hash tables achieve that. They give you order P query and order T space.

Again, the issue is some of these cells are absent so don't use an array. That's like a direct mapped hash table. Use a hash table, use hashing. That way you can use linear space per node, however many occupied children there are.

What is T here, by the way? T is the sum of the lengths of the T i's-- because here we're storing multiple T i's. Or it's the number of nodes in the tree, which, if your strings happen to have a lot of common prefixes, the number of nodes in the trie could be smaller than that, but not in general.

What's the problem with the hash table? Question.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Yes. For the BST, we need to store some keys in this node. That lets you do a binary search.

For example, every node could store the max in the left subtree-- just within this little tree, though.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: It is order T. Sorry, I see-- why is it not O T Sigma space? You're only storing one letter here, so that fits in a single word, and two pointers. So every node only takes constant space. It's only T space, not T Sigma. Other questions? Or answers?

There's a problem with hashing-- doesn't actually solve the problem we want to solve. It doesn't solve predecessor. Because hashing mixes up the order of the nodes. This is the problem we had with-- what's it called-- signature sort, which hashed, it messed up, it permuted all the things in the nodes and so you didn't know-- I mean, in a hash table, you can't solve predecessor. That's what the predecessor problem is for.

I guess you could try to throw a predecessor data structure in here. Actually, I hadn't thought of that before. So we could use y-fast tries or something. And we would get order T space and-- I guess, with high probability, this is also with high probability-- we get order P log log Sigma, I guess. Because I use Van Emde Boas. I'm going to have to call it 3.5, Van Emde Boas. So that would be another approach. So hashing will not do a predecessor.

We'll do exact search, which is still an interesting problem. Might give you some strings I want to know-- is this string in your set? But it won't solve the predecessor problem. So this is an interesting solution-- hashing-- but not quite what we want.

And Van Emde Boas doesn't quite do what we want either. It improves over the BST approach but we get another log in there. But it's still not order P. I kind of like order P. Or at least, instead of order P times log log Sigma, I kind of like order P plus log Sigma.

And order P plus log Sigma is known. So that's what I want to tell you about. And this is normally done with a structure called trays, which is a portamento, I guess, of tree and array. Somewhere in there there's a tree and an array, so it's a bit of an awkward word. But Those are developed by Koplowitz and Lewenstein, in 2006, a fairly recent innovation.

I'll have this number 6-- trays, achieve order P plus log Sigma and order T space. So this is pretty good. And they will do predecessor and successor-- definitely an improvement over the BST.

It's open whether you could do order P plus log log Sigma. This is as far as I can tell, no one has worked on this. Maybe we will work on it today. So something to think about-- whether you could get the best of all of these worlds for predecessor.

There's a lower bound-- you need to spend at least log log Sigma time. Because even if you try as a single node, you have the predecessor lower bound. And we know log log universe size is the best you can do in this regime. So that's where we're going.

Instead of describing trays, though, I'm going to describe a new way to do it, which has never been seen before in any class or anywhere. Because it's brand new. It's developed by Martin Farach-Colton, who did the LCA and the level ancestor structures that we saw in last class. And he just told it to me and it's really cool so we're going to cover it. A simpler way to get this same bound of trays.

And the first thing we're going to do is use a weight balanced BST. This will achieve P plus log k query and linear space. k, remember, is the number of strings that we're storing, so it's the number of leaves in the trie. So it's not quite as good as P plus log Sigma but it's going to be a warm up. We're going to to do this and then we're going to improve it.

Remember weight balanced trees, we talked about them way back in lecture 3, I believe. There is an issue of what is the weight. And typically, you say, the weight of a subtree is the number of nodes in the subtree.

I'm going to change that slightly and say, the weight of a subtree is the number of descendant leaves in the subtree, not the number of nodes, because it's log k. We really care about the number of leaves down there. There could be long paths here which we are not so excited about. We really care about how many leaves are down there. Like the weight of this node here is three-- there's three leaves below it.

You may recall weight balanced BSTs trying to make the weight of the left subtree within a constant factor of the weight of the right subtree. Because we're static, we can be even simpler and say, find the optimal partition.

So we're thinking about this approach-- idea of expanding a large degree node into some binary tree. We have a choice of what binary tree to use. With three nodes it may be not many choices-- that could be this or it could be a straight this way or a straight line that way. Those are different. And if one of these guys is really heavy, one of these children is really heavy, you want to put it closer to the root. So that's what we're going to do.

Let me draw it this way. That's kind of an array. But what this array represents is for a node-- so here's my node, it has lots of children. Some of these are heavy, some of them are light, lighter than others. We don't know how they're distributed. But they're ordered, we have to preserve the order.

What this is supposed to represent is the total number of leaves in this subtree. So the total number of leaves here. And then I'm going to partition this rectangle into groups corresponding to these sizes. So these are small, medium, small, little less than medium, big, and then small. Something like that. So these horizontal lengths correspond to the number of leaves in these things, correspond to the weight of my children.

So I look at that and I say, well, what I'd really like to do is split this in the middle, which is, maybe, here. I say, OK, well, then I'll split here. That's pretty close to the middle. So my left subtree will consist of these guys, my right subtree will consist of these guys. And then I recurse-- over here I've split at the middle, I find the thing that's closest to the middle. Over here I've split at the middle, I find the thing that's closest to the middle. It's pretty much determined.

So my root node corresponds to this one. It's going to partition here. So over on the right, there's going to be-- here's going to be the big tree and then here is the small tree. So this small tree corresponds to this one. This big tree corresponds to this interval. Then on the left we've got four things we need to store. So these are the red pointers that we had before.

Then over on the left, we're going to have a partition. And then there's going to be two guys here. It doesn't really matter how we store them. It's something like this. There is medium and small. And then over on the left, we also have two guys. So it's going to be, again, something like this. You got medium and small. So you see how that worked.

Our main goal was to make this big guy as close to the root as possible. It was the biggest and that's basically what happened. This one is really big. And we couldn't quite put it as a child of the root because it appeared in the middle, but we could put it as a grandchild at the root. In general, if you have a super heavy child, it will always become a child or grandchild of the root. So in constant number of traversals you'll get there.

Now again, you fill in these nodes with some keys so you can do a binary search. But now the binary search might go faster than log Sigma, which is what we had before. And indeed, you can prove that this really works well.

So what's the claim? Claim is every two edges you follow either advance one letter in P-- these are the red edges that we want to follow. So if we follow a red edge, then we've made progress to the next node. So this would be following a red edge. Or we reduce the number of candidate to T i's by 2/3 or, I guess, to 2/3 of its original value. So we lose a third of the strings. That's what I'd like to claim.

And it's not too hard to see this. You have to imagine all of these possible partitions. It's a little bit awkward. The idea is the following. If you take one of these arrays-- this view of all the leaves just laid out on the line-- you say, well, I'd like to split in half and half. But that will never happen unless I'm really lucky.

So let's think about this one third splitting. If I were able to cut anywhere in here, then in one step, actually, I would achieve this 2/3 reduction. I'd lose a third of the nodes. If I end up cutting here, for example, then either I go to the left and I lost almost 2/3 of the nodes, or I go to the right and I at least lost this one third of the notes or one third of the leaves, I should say.

So that would be a good situation if I got some cut in here. But it might be there is no possible cut I can make in here because there's a giant child in here that has more than one third of the nodes. It would have to span all the way across here. So I can't make any cuts, I can only cut between child boundaries.

In that situation, you make this-- well, this is when I need to follow two edges, not one. When there's a super big child like that, as we said, it will become a grandchild of the root. So it will be-- there's the root and then here is the giant tree. And then there's going to be the other stuff here and here.

So after I go down to either one step or two steps, I will either get here-- sorry, more red chalk, this was a red point. Now, this is going to a child. So if I went there, I'm happy in two steps. I advance one letter in P. Or in two steps, I went here or I went here. And this was a huge amount of the nodes, this is at least a third of the nodes. Again, if I end up here or end up here, I lost 2/3 of the candidate leaves. I mean, I lost one third of the candidate leaves, leaving 2/3 of them.

If this happens, I charged to this order P term. And if the other situation happens, I charge the log k term-- because I can only reduce k by a factor of 2/3-- order log k times. This implies order p plus log k search.

So a very simple idea. Just change the way we do BSTs. And we get, in some cases, a better bound. But not in all cases because maybe P plus log k might be bigger than P times log Sigma. And k and Sigma are kind of incomparable, so we don't know.

That's where method 5 comes in, which is our good friend from last class-- leaf trimming and indirection. So we're going to use this idea of finding-- we're going to cut below maximally deep nodes with the right number of descendants in them. So we need at least Sigma descendants. It could just be descendants or descendant leaves, doesn't actually matter. Let me draw a picture, maybe.

This is pretty much what we did before, except before this magic number was log n that we needed or 1/2 log n or something. Now it's going to be Sigma that we need. So it is we find these maximally deep nodes-- these dots-- that have at least-- I guess, there is really multiple things hanging off here. In general, it could be several things hanging off. But the total number of descendants of each of these nodes is at least Sigma.

So what that implies is that the number of these dots, the number of the leaves in the top tree-- so up here-- number of leaves is at most T over Sigma. Because we can charge each of these nodes to Sigma descendants in each of them.

So that's good because it says we can use method 1-- the simple array method-- which is fast but spacious. But if our new size of the trie gets divided by a Sigma factor, then this turns out to be linear. So up here we use method 1.

Now, you got to be a little careful because we can't use method 1 on all the nodes. We can definitely use it on the leaves because there aren't too many leaves. That means we can also use it on the number of branching nodes. Number of branching nodes is also going to be, at most, T over Sigma because it's actually one fewer branching node than there are leaves.

So great, I can use arrays on the leaves, I can use arrays on the branching nodes. I can't use it on the non-branching nodes. Non-branching nodes are nodes with a single descendant and everything else is null.

What do I do for those nodes? Very difficult. I just store that one pointer in a storage label. I guess you could think of that as method 2 in a very trivial case. You see-- is this the right label? Yes or no. So this is the non-branching nodes. Non-branching top nodes-- I will use method 2. So I guess this is really-- well, for these guys I use method 1, for these guys I use method 1.

So I can afford all this. This will take order T space. And it will be fast because either I'm using arrays or I really don't have any work to do, and so it doesn't really matter what I do. But except I can't use arrays because they would be too spacious.

So that handles the top. Now, the issue is, what about these bottom structures? The bottom structures-- what do we know? They have to have less than Sigma nodes, less than Sigma descendants. Also less than Sigma leaves.

So in other words, in these trees we have k less than Sigma. Well, then we can afford to use method 4. Because our whole goal is to get k down to Sigma in this bound. So in the bottom trees, we use method 4.

Method 4 was always linear space. And the issue was we paid P plus log k. But now in here, k is less than Sigma in these trees. So that means we get order P plus log Sigma query time. And that's the best we know how to do if you want predecessor at the nodes.

So it matches this tray bound in pretty easy way. Just to apply weight balanced, clean things up a little bit. But only do that at the leaves and everywhere up here, basically. Except the non-branching nodes use arrays. So for the most part arrays and then, at the bottom, you use weight balance.

This is how you ought to represent a trie. If you want to preserve the order of the children, this is the best we know how to do. If you don't want to preserve order, just use a hash table. So it depends on the application.

One fun application of this is string sorting. It's not a data structures problem so I don't want to spend too much time on it. But you use this trie data structure to sort strings. You just throw in a string and then throw in a string. We didn't talk about dynamic tries but it can be done.

And if you throw it, you just sort of find where you fall off and then add the thing. Now, you have to maintain all this funky stuff but weight balanced trees can be made dynamic and indirection can be made dynamic.

So you end up with this sort of simple incremental scheme. You end up with T plus k log

Sigma to sort k strings of total size T with alphabet size Sigma. This is good. If I used, for example, merge sort to sort strings, it's going to be very bad. It's going to be something like T times k times log something. We didn't really care about the log. T times k is bad. That's because comparing strings could potentially take T time. And then there's k of them.

But this is linear. This is the sum of the lengths of the strings. There's this extra little term. But most of the time that's going to be dominated by the length of the strings. So that's a good way to sort strings using tries. Tries by themselves, I mean this is about all there is to say.

So let's move on to suffix trees and compressed tries. Now, we actually did compressed tries in the signature sort lecture. Actually, why don't I go over here?

So tries-- branches were labeled with letters. That's still going to be true for a compressed trie. But as we saw in that lecture, in compressed trie we're going to get rid of the non-branching nodes.

So idea with the compressed trie is very simple-- just contract non-branching paths into a single edge. This is our example of a trie. We're just going to modify it to make a compressed trie.

Here we have a non-branching path. We have to follow an a, and then we have to follow an n. There's no point in having this node. You might as well just have a single edge that says a-n on it.

So we go from here, from the root. We're going to have an edge that says a-n. And in some sense, the key of this child is a. If you're starting up here and you want to know which way should I go, you should only go this way if your first letter is a. After that, your next letter better be n, otherwise you fell off the tree. So that's the compression we're doing.

Now, here we have-- this is a branching node, so that node we keep intact. This is an n, this is an a here. But here it's non-branching. Let me draw this a little bit longer. In reality, it's just a single edge. And again, the key is a, and then you must have a $ sign on afterwards. Then you reach a leaf, the first leaf.

If we follow the n branch-- this is branching, so that node is preserved. If I go this way, it's a $ sign and I reach a leaf. If I go this way it's an a that must be followed by a $ sign, so that's a leaf. And if I go this way, it must be an e, followed by a $ sign, which is a leaf. Again, these four leaves can point to these places.

That's a compressed trie. Pretty obvious. The nice thing about the compressed trie is the number-- here we knew the number of non-branching nodes, it was at most the number of leaves. Over here, the number of internal nodes is at most the number of leaves.

So this structure has order k nodes in total because we got rid of all the non-branching nodes. I guess except the root, the root might not be branching. We've got a big O there to cover us. And all the things we said about representing tries here, you can do the same thing with a compressed trie. I need to write down that 3.5 here.

And in fact, these results get better because before order T meant the number of nodes in the trie. Now order T will be the number of nodes in the compressed trie, which is actually order k. So life gets really good in this world.

I did it in the trie setting because it's just simpler to think about. But really, you would always store a compressed trie. There's no point in storing a trie. You can still do the same kinds of searches.

But really, compressed tries are warm up for suffix trees. So let's talk about suffix trees. Suffix trees are a compressed trie. So really they should be called suffix tries. And occasionally, people will call them suffix tries. But most people call them suffix trees, so for consistency I'll call them trees as well. But they are tries. I'm going to introduce some notation here.

With tries, we are thinking about lots of different strings. In this case, we're going back to our string matching problem. We have a single text and we want to preprocess that text. But we're going to turn it into multiple strings by looking at all suffixes of the string. This is Python notation for everything from letter i onwards. And we do that for all i, so that's a lot of strings. And we build the compressed trie over them.

That's the idea. And to make it work out-- because you remember, with tries we had to append $ sign to every string. In this case, we'd just have to append $ sign to T, and then all suffixes will end with a $ sign. So that covers us. $ sign, again, is a character not appearing in the alphabet. And that's it. So that's a definition. Let's do an example.

At this point, we going for this goal of order P query, order T space. Suffix trees will be a way to achieve that goal. Let's do my favorite example which is banana. I had a friend who said, I know how to spell banana, I just don't know when to stop. There's nice pattern to it and a lot of repeated letters and so on.

I've got to number the characters. He said that when he was like six, not when he was older. It's a little harder when you're writing it on the board but we all know how to spell banana, I hope. I'd got it right, right? It should be 7 letters, including the $ sign.

There they are. So there's a suffix which is the whole string. There's a suffix which is a, n, a, n, a, $ sign. There is a suffix which is n, a, n, a, $ sign. There's a suffix which is a, n, a, $ sign. Suffix n, a, $ sign. a, $ sign. And $ sign. And empty, I suppose, but we're not going to store that one. You don't need to. Cool.

I'm going to cheat a little bit and look at my figure because it is a little bit of thinking. One The final challenge of this lecture will be construct this diagram in linear time. But I'm, just for now, going to cheat because it's a little tricky to do it and get all the nodes in sorted order. So that should give it to us. And then the suffixes. Here is another color. 6, 5, 3, 1, 0, 4, 2. Cool.

This I claim is a suffix tree of banana. You see the banana substring. Than the next one is a, n, a, n, a, $ sign. Then the next one is n, a, n, a, $ sign. Then the next one is a, n, a, $ sign. Next one is n, a, $ sign. Next one is a, $ sign. And then $ sign. So that's a nice, clean representation of all the suffixes.

And you can see that if you wanted to search from the middle of this string-- suppose I want to search for a nan-- then it's right there. Just do n, a, n, then I'm done. This virtual node in the middle here along the one third of the way down the edge, that represents n-a-n. And indeed, if you look at the descendant leaf, that corresponds to an occurrence of n-a-n.

If I was going to look for a-n, I would do a, n, so halfway down this edge. And then this subtree represents all the occurrences of a-n. Think about it. There's two of them-- One that starts at position 3, one that starts at position 1. Here's one occurrence of a-n, here's another occurrence of a-n.

This works even when they're overlapping. If I search for a-n-a, I would get here. And then these are the two occurrences of a-n-a and they actually overlap each other-- this one and this one. So this is a great data structure, it solves what we need. It's all substrings searching.

Applications of suffix trees. Just do a search in the trie for a particular pattern. We get subtree representing all of the occurrences of P and T. So this is great. In order P time, walking down this structure, I can figure out all the occurrences. And then, if I want to know how many there were, I could just store subtree sizes-- number of leaves below every node. If I wanted to list them, I could just do an in-order traversal. And I'll even get them in order.

So in particular, if I wanted to list the first 10 occurrences, I could store the left-most leaf from every node, teleport down to the first occurrence in constant time. And then I could just have a linked list of all the leaves. So once I find the first one, I can just follow until I find, oh, that's not an occurrence of P.

So I can list the first k of them in order k time once I've done the search of order P time. So this is really good searching. And It's the ideal situation. You can list any information you want about all of the answers in the optimal time and size of the output.

How big is this data structure? Well, there are T suffixes, so k is the size of T. And when we look at our trie representations, our general goal was to get-- here, capital T was the sum of the lengths. Well, sum of the lengths is not good-- that would be quadratic-- sum of the lengths of the suffixes.

But we also said, or the number of nodes in the trie. And we know the number of leaves in this trie is exactly the size of T. And so because it's a compressed trie, the number of internal [INAUDIBLE] is also less than the size of T. So the total number of nodes here is order T And so if we use any of the reasonable representations, we get order T space.

Now, there's one issue which is, how long does a search for P cost? And it depends on our representation, it depends how quickly we can traverse a node. If we use hashing-- method 3-- use hashing, then we get order P time.

But the trouble with hashing is it permutes the children of every node. So in that situation, the leaves will not be ordered in the same way that they're ordered in the string. So if you really want to be able to find the first five occurrences of the pattern P, you can't use hashing. You can find some five occurrences but you will find the first in the usual ordering of the string.

So if you really want the first five and you want them in order, then you should use trays-- this method 6 that we used. 6? 5. If we use trays, then it will be order P times log Sigma-- sorry, order P plus log Sigma. That was our query time. Here, P plus log Sigma. Small penalty to pay but the nice thing is then your answers are represented in order. No permutation, no hashing, no randomization.

This is the reason suffix trees were invented-- they let you do searches fast. But actually, they let you do a ton of things fast. And I want to quickly give you an overview of the zillions of things you can do with the suffix tree. And then I want to get to how to build them in linear time, which has some interesting algorithms/data structures.

I already talked about if you want to find the first k occurrences, you can do that in order k time. If you want to find the number of occurrences, you can do that in constant time, just by augmenting the subtree sizes.

Here's another thing you could do. Suppose you have a very long string. I mean think of T as an entire document. You know, it could be the Merriam-Webster dictionary or it could be the web. We're imagining T to be the huge data structure. And then we're able to search for substrings within that data structure very fast. So that's cool.

Here's an interesting puzzle. What is the longest substring-- what is the longest string that appears twice on the web? This is called the longest repeated substring. Could be overlapping, maybe not. Well, you take the web, you throw it in the suffix tree-- not sure anyone could actually do that-- but small part of the web. Dictionary-- this would be no problem. Wikipedia would be feasible. You take Wikipedia, you throw it in the suffix tree.

And what I'm interested in is, basically, a node that has two, at least two descendant leaves. And if I'm counting the number of leaves at every node, I could just do one pass over this data structure and find what are all the nodes that have at least two descendant leaves. That's all the internal nodes.

And then among them I'd also like to know how deep is it. Because the depth corresponds to how long the string is. This one is a-n-a so this one has, I call it, a letter depth of 3. This one has a letter depth of 1. This one has a letter depth of 2. So I just want to find the deepest node that has at least two descendant leaves.

In linear time, I could find the longest repeated substring. Or I could find the longest substring that appears five times or whatever. I just do one pass over this thing, find the deepest node that has my threshold of leaves. So that's kind of a neat thing you can do in linear time on a string.

Here's another fun one. Suppose I have this giant string. And I just want to compare two substrings in it. So here's my giant string. And suppose I want to measure how long is the repeated substring. So I say, well, I've got position i, I've got position j. Let's say I already know that they match for a little bit. I want to know, how long do they match? How far can I go to the right and have them still match?

How could I do that? Well, I could look at the suffix starting at i. That corresponds to a leaf over here. And I could look at the suffix starting at j. That corresponds to some other leaf. And what is the length of the longest common prefix of those two suffixes in the suffix tree?

Three letters-- LCA. If I take the LCA of those two leaves-- for example, I take these two leaves-- the LCA gives me the longest common prefix. Then they branch. So longest common prefix of these two suffixes is the letter a, so it's just length 1. And again, if I label every node with the letter depth, I can figure out exactly how long these guys match, even if they overlap.

So in constant time-- because we already have a constant time LCA query. Linear space, constant time query. Given any two positions i and j, I can tell you how long they match for in constant time. Boom-- instantaneously. It's kind of crazy. So you can do tons of these queries instantly. That's one reason why people care about LCAs, there are other reasons. But mostly LCAs were developed for suffix trees to answer queries like that.

Got some more. Why don't I just write-- LCP of one suffix and another suffix is equivalent to an LCA query. And so we can do that in constant time after pre-processing.

Here's another one. Suppose I want to find all occurrences of T i to j. So I give you a substring and I want to know where does that occur. The substring is restricted to come from the text.

Now, this is a little subtle. Of course, I could solve it in j minus i plus 1 time. I just do the search. But what if I want to do it in constant time?

Maybe this is a really big substring. But I still know it appears multiple times. I want to know how many times does it appear. I claim I can do this in constant time. How?

This is a level ancestor query. Why is it a level ancestor query? If I look at the suffix starting at i, and then I just want to trim off, I want to stop. Or I don't care about the entire suffix, I just want to do that j.

It's like saying, well, suppose I'm looking for occurrences of a-n-a. So I go and I start at the first occurrence of a-n-a, which is a-n-a-n-a-$, so this is the leaf corresponding to a-n-a. And then if I want to find all occurrences of a-n-a, I just need to go up to the ancestor that represents a-n-a.

This is what I call a weighted level ancestor. That's not quite the problem we solved in the last lecture, lecture 15, because now it's weighted. So it's level ancestor j minus i of the T i suffix leaf. So I find this leaf, which I just have an array of all the leaves. Given a suffix, tell me what leaf it is in the suffix tree.

And then I want to find the j minus i-th ancestor, except the edges don't just have unit length. So here I want to find the third ancestor, except it's really the ancestor in the compressed trie. I want to do the j minus i-th ancestor in the suffix in the trie, but what I have is a compressed tree. And so these edges are labeled with how many characters are on them and I got to deal with that.

Fortunately, the data structure we gave for a level ancestor-- which was constant time query, linear space-- can be fairly easily adapted to weights. Not quite in constant time though. It can be solved in log log n time.

And I think that's optimal. Because if your thing is a single path with maybe the occasional branch, then finding your i-th ancestor here is like solving a predecessor problem. Because you say, well, from the i-th position up, I want to know what is the previous-- I want to round up or round down. So I want to do a predecessor or a successor on this straight line. And so for a predecessor you need log log time for the right parameters and this can be achieved.

And the basic idea is you use ladder decomposition, just like before. But now a ladder can't be represented by an array because there are lots of absent places in the array. Now instead, use a predecessor, use a Van Emde Boas to represent a ladder. So that's basically all you do. Van Emde Boas represents a ladder.

That's what you do in the top structure. Remember, we had indirection, leaf trimming, top was this thing, ladder decomposition. You Bottom was look up tables. The other problem is you can't use lookup tables anymore because in one of these tiny trees you could have a super long path. It's non-branching, they got compressed. And you can't afford to enumerate all possible situations. It's kind of annoying.

So instead of using lookup tables-- this was actually an idea from some students in this class last time I taught this material-- they said, oh, well, instead of using a lookup table, you can use ladder decomposition. So down here, in the compressed tree, we have log n different nodes. If you use ladder decomposition on that thing-- but not the hybrid structure. Remember, we used jump pointers plus ladders. Jump pointers still work here, just you have to round them to a different place.

Down here, I'm not going to try to do jump pointers plus ladders. I'll just do ladders. And remember, just ladders gave us a log n query time. But now n is log T. And so we get log log T query time. And that's, basically, all you have to do.

So you're always jumping to the top of a ladder. You'll only have to traverse log log T ladders. The very last ladder you might have to do a predecessor query that will cost you log log log T. But overall, it will be log log T time just by this kind of tweak to our level ancestor data structure. So I thought that was kind of a fun connection.

This is the reason, essentially, level ancestors were developed. And people use them because you can do these kinds of things in nearly constant time, even if the substring is huge. So maybe I know ahead of time all the queries I might want to do. I just throw them into the text, just add them in there. Then I've cut these substrings, they're now represented in the suffix tree.

Now any substring I want to query in log log n time, I can find all the occurrences of that string, even if the substring is huge. So if you know what queries you want, you can preprocess them and solve them even faster than order P time. Cool.

Another thing you can do is represent multiple documents. And that's what I was sort of getting at there. If you have multiple documents-- say, you're storing the entire web or Wikipedia. Like there's multiple pages. You want to separate them.

All you need to do is say, OK, I'll take my first string and then put a special $ sign after it. Then take my second string, put a special $ sign after it. And take my k-th string and put a special $ sign after it. Just concatenate them with different $ signs in between them. Then build the suffix tree on this thing which I'll call T

So you can use the same suffix tree data structure, but now, in some sense, you're representing all of these documents and all the ways they interweave. Because there are some shared substrings here that are shared by this, and this, and whatever. And those will be represented in the same structure. Or I can do a search and then I've effectively found all the documents that contain it.

One issue, though. Suppose, I want to find all the documents containing the word MIT or something. Maybe all k of them match, maybe one document matches, maybe two documents match. Suppose, two documents match.

The first document mentions MIT a billion times. The second document has MIT in it once. Then suffix trees are kind of annoying because they will find that billion and one matches as a subtree. But if I just want to know the answer, oh, these two documents match, I'd like to do that in order 2 time, not order billion time, to use technical terms.

And that is called the document retrieval problem or a document retrieval data structure. This is a problem considered by M. Krishnan in 2002. Document retrieval you can do an order P plus number of documents matching.

So if I want to list all the documents that match, I could do an order the number of documents that match, not the order of a number of occurrences of the string. So I still got to do the P search in the beginning, and then this is better.

And the funny thing is the solution to this data structure uses RMQ, range minimum queries, from last lecture. So let me tell you how it works. It's actually very simple. And then I think we'll move on to how to build a suffix tree.

So document retrieval. Here's what we're going to do. Remember, these different $ signs i represent different documents. I want to remember which suffixes came from the same document. So at every $ sign i, I want to store the number of the previous $ sign i.

Let's suppose, the suffixes, when they get to one of the $ signs, I can just stop, I don't have to store the rest, I'm going to throw away. Whenever I hit a $ sign, I will stop the suffix tree. That way, the $ signs really are leaves, all of them now become leaves. So I don't really care about a suffix that goes all the way through here. I just want the suffix to the $ sign, as it represents the individual documents.

So $ sign i's are leaves. And I want each of them just to store a pointer, basically, to the previous one of the same type, the same $ sign i. It came from the same document.

Now, here's the idea. I did a search, I got down to a node, and now there's this big subtree here. And this subtree has a bunch of leaves in it, those represent all the occurrences of the pattern P. And let's suppose that those leaves are numbered. I'm numbering the leaves from 1 to n, I guess.

Then in here, the leaves will be an interval-- interval l, comma, n. And the trouble is a lot of these have the same label $ sign i. And I just want to find the unique ones. How do I do that?

What we do is find the first occurrence of $ sign i for each i. I could just find the first occurrence of $ sign i for each i. I'd then only have to pay order number of distinct documents, then we'll have to pay for every match within the document.

Now, one way to define the first $ sign i is-- that's a $ sign i whose stored value-- we said we store the leaf number of the previous $ sign i-- whose stored value is less than l. So we find some position here. If the previous guy is less than l, that means it was the first of that type. If we store this, that's definition of being first.

So in this interval, I want to find $ sign i's that have very small stored values. How would I find the very best one? Range minimum query. So we do a range minimum query on l, comma, n. If there's any firsts in there, this will find it.

Find, let's say, a position m with the smallest possible stored value. If the stored number is less than l, then output that answer. And then recurse on the remaining intervals. So there's going to be from l to m minus 1 and m plus 1 to n.

So we find the best candidate, the minimum. That's minimum sorted value. If anything is going to be less than l, that would be it. If it is less than l, we output it, then we recurse over here and we recurse over here. At some point this will stop finding things. We're going to do another RMQ over here. Might not find anything, then we just stop that recursion.

But the number of recursions we have to do is going to be equal to the number of documents that match, maybe plus 1. So we achieved this bound using RMQ because RMQ we can do in constant time with appropriate pre-processing.

Now, the RMQ is over an array. It's over this array of stored values indexed by leaves. And this idea of taking the leaves and writing them down in order is actually something we need. It's called a suffix array. We're going to use this alternate representation of suffix trees in order to compute them.

Suffix arrays in some sense are easier to think about. The idea with the suffix array is to write down all the suffixes, sort them. This is conceptual. Imagine you take all these suffixes. Their total size is quadratic in T so you'd never actually want to do this. But just imagine writing them down, sorting them lexically using our string sorting algorithms.

And then we can't represent them explicitly because it would be too big. Just write down their index, just store the indices. Let's do this for banana. Banana's over here. It'll make my life a little harder.

Actually, they're already here in sorted order. If dollar sign, I'm supposing, is first, first suffix is $, then a-$, then a-n-a-$, then a-n-a-n-a-$, then banana, then n-a-$, then n-a-n-a-$. I'll just write that down over here. $, a-$, a-n-a-$, a-n-a-n-a-$, then banana, then n-a-$, then n-a-n-a-$. If you look at these, they're indeed in sorted order-- $, a's, b's, n's. Everything is sorted here lexically.

Now, I can't store this because it's quadratic size. Instead, I just write down the numbers that are down there. This was the sixth suffix, it was starting at position 6. Then 5, then 3, then 1, then 0-- that's everything-- then 4, then 2. This thing is the suffix array. It also has linear size. It's just a permutation on the suffix labels, suffix indices. I still want to tell you about it.

There's some other information that's helpful to write down about the suffix array. It's called longest common prefix information, LCP. The idea is to look at adjacent elements in the suffix array. In some sense, this represents the same information, right?

Our whole goal is to sort the suffixes. If we could do this, then, as we'll see, we can also build this. And this is sort of what we really want. The suffix array by itself is pretty good if you add in LCP information.

LCP is-- what is the longest common prefix of these two suffixes? In this case, 0. In this case, one letter. In this case, three letters match. So here the value is 3. And the next one, zero letters match. Next one, zero letters match. Next one, two letters match. So this is another array you could store here-- 0, 1, 3, 0, 0, 2.

AUDIENCE: Longest common prefix?

ERIK DEMAINE: Longest common prefix of the suffixes. Because each of these is a suffix but here we're interested in how long they match for.

I claim if you have this suffix array and this LCP information, you can build this structure. Anyone wants to tell me how to build this using this? It's a one word or two word answer that we saw, I think, last class. But we saw a lot of things last class, so it's maybe not obvious. Magic words are Cartesian tree.

Cartesian tree was how we converted RMQ into LCA, I think. Yeah? Which was you take the minimum value in the array, make that the root, and then recurse on the two sides. So a Cartesian tree of the LCP array, basically, gives you this transformation.

The minimum values here are the 0's. Now, before we just broke ties, we picked an arbitrary 0, put it at the root. Now I want to take all the 0's, put them at the root. If I do that, I get three 0's at the root and then I have everything in between. So there's nothing left of the first 0. Then next one, there's these guys and the mins are going to be 1 and then 3. So here I'm going to get a 1 when I recurse and then 3. There's nothing in between these 0's. And after the last 0, there's a 2.

So this would be the Cartesian tree, a slightly different version where we don't break ties, we take all the mins simultaneously, put them at the root. Now, does that look like this thing? Yeah. Everything except the leaves. [INAUDIBLE] are missing at the leaves. The leaves are represented by these values. Just visit them in order here, do an inner traversal of the missing pointers here. We're going to get 6, and then 5, and then 3 and then 1, and then 0, and then 4, and then 2.

Now, the meaning of these values is slightly different. Maybe I should circle them in red. These leaves are just like these leaves. They're exactly the labels we wrote down in the same order. These numbers are slightly different. What they represent are letter depths. The letter depth of this node is 0, letter depth of this node is 1, letter depth of this node is 3. That's what I wrote here-- 1, 3, 2. This one says, 2. These LCPs are exactly the letter depth. That's how far down the tree you are.

Once you have this structure and the letter depth, you can very easily put in these labels. I won't say how to do that. But in linear time, if I could build the suffix array plus the LCPs, I could build suffix tree. So our real goal is to build this information, these two arrays. If we could do it in linear time, we'd get a suffix tree in linear time. So that is what remains to be done.

We're going to do-- not quite linear time. If you want a nicely sorted suffix tree where all the children are labeled here-- so in particular, if I just had a single node, I have to be able to sort the letters in the alphabet. However long that takes. Maybe it's a small alphabet and you can do linear time sorting by radix sort or whatever. However long that takes, we do it once. Then the rest will be order T time.

Here's how we do it. First step-- sort the alphabet. This will turn out to be more interesting than you might think. I'll come back to it. Second step-- replace each letter by its index in the sorted order. This sounds boring but it will be useful for later. Third step-- the big idea. This is an algorithm by Karkkainen and Sanders, from 2003.

The problem was first solved in this running time by Martin Farach-Colton, our good friend. But then it got simplified. So I'll tell you a little bit about that in a moment. And there going to be a lot of writing here.

The idea here is we're going to take the 3i-th letter, 3i plus first, 3i plus second letter, concatenate them into a single triple letter-- think of it as a single letter. And then just do that for all i. So it's like I take these guys, make them one letter, these guys, make them one letter.

Now, I could start at 0, or I could start at 1, or I could start at 2. Do them all. So this is going to be 3i plus 1, 3i plus 2, 3i plus 3. And this one is going to be 3i plus 2, 3i plus 3, 3i plus 4. We're going to do this to recurse.

But the point is, if I want to represent all the suffixes of T, suffix could start at a position 0 mod 3, or position 1 mod 3, or position 2 mod 3. So if I could sort all the suffixes of these guys, I would effectively sort all the suffixes of the original T. This tripling up doesn't really change things, up to like plus 1 or 2.

Next, I believe, is recursion. I'm going to take T0 and T1 and concatenate them. This thing has size 2/3 n. It has number of characters 2/3 n because each of them has a third of the number of characters. Of course, all the information is still there, which is kind of weird.

But if we treat this as a single character, which then has a 1/3 n, we can't afford to recurse on all three. We can only afford to recurse on two out of the three because then we're going to get a recurrence of the form T of n is T of 2/3 n plus order n. And this is geometric, so it's order n. That's how we're going to get linear time after the first sort. If this was 3/3 n, then this would be n log n. We don't want to do that.

So that's what I can afford. Now I've got to deal with it. What this tells me is, the sorted order of all the suffixes of T0 and T1, all the suffixes starting at positions that are 0 or 1 mod 3.

Next thing we'd like to do is sort the suffixes of T2. We can do that, I claim, by radix sort. How do we do that? Well, if you look at a suffix T 2i, this is the same thing as T from 3i plus 2 onwards. Which we can think of as that first character, comma, the next character onwards. Sorry, that's the angle bracket.

And this thing is, basically, T0 of i plus 1 onwards. So if I strip off the first letter, then I get a suffix that I know about. I know the sorted order of all the T0 suffixes. So this is really just a-- you can think of this as a two character value.

There's a single character from Sigma here, which we've already reduced down to-- this is an integer between 0 and Sigma minus 1. This thing you can do the same thing with these recursive values. So you've just got two values. Small. You can radix sort them in linear time. And then we will have sorted T2 suffixes because we already knew the order of these guys.

One more thing, which we have to merge suffixes of T0 and T1 with suffixes of T2. And this is where we use the fact that there are three of these things and not two of them. This is a weird case where three way divide and conquer works. Two way divide and conquer is what Farach-Colton did originally. It's much more complicated because of this merge step. Merge gets painful.

I claim this merging is easy because merging is linear time, provided your comparison is constant time. So if I need to compare a T0 suffix with a T2 suffix, if I want to do that comparison, I strip off the first letter from this one. It turns into a T1 suffix, the first character plus a T1 suffix. If I strip out the first character of this one, it turns into the first character and then a T0 suffix. And these things I know how to compare because I already sorted T0, comma, T1.

If I need to compare T1 suffix with the T2 suffix, how do I do it? I strip off the first two letters of this one, I get a T0 suffix. I strip off the first two letters of this one, I get a T1 suffix.

I can't strip off one letter because this would turn it into a T2 and I don't know how to compare T2 to other things, that's the whole point. I guess, it's a T2 versus a T0, if I did that, which is this case. But here, I strip off two letters, I get something I know how to compare. This technique does not work if you only have two things. It only works if you have three things because they're sort of these situations.

So constant time. By comparing these little tuples, the first character or two plus the remaining suffix, I can do the comparator and merge. And then if I can do that, everything is linear time.

The only interesting thing is how do I sort the alphabet when I recurse? And for that, you use radix sort. So the first time, you pay sort of Sigma. We don't know how long that takes, depends on your alphabet. But every following recursion it's a radix sort because you have a triple of values, each of which is small. And so you can do it in linear time. Because there's only three digits to the thing you're sorting.

So overall, this is a recursive algorithm. It gives you linear time because you're making one recursive call of 2/3 the size. Pretty clever and simple. And that's suffix trees and how you build them. Versus you get suffix arrays, you can do the same thing and get LCP information at the same time, it's written in the nodes. Then you get suffix trees. And then you're done.