MITOCW | watch?v=DZ7jt1F8KKw

PROFESSOR: All right today we begin a topic dear to my heart. It's a problem that's still pretty open, but I've worked on a lot. Dynamic optimality. Central question here is, is there one binary search tree that's at least as good as all other binary search trees? Is there one binary search tree to rule them all?

It's kind of a fundamental question. Goes back to the '80s. In particular splay trees, which you have probably seen before. We'll review them briefly here. And there's a lot of technology built up, which we'll talk about this lecture and next lecture, about how to tackle this problem. It's still active area of research.

So before I get to-- well the central question here is, is there one best binary search tree? It's a natural question because we use binary search trees all the time. Already in this class, probably every lecture, we've used a binary search tree for something.

And we all know how to do order log in balanced binary search trees. And the question becomes, is this the best you can do?

To make this more formal of a problem, well, yeah. I want to define binary search tree as a model of computation. OK we've talked about pointer machine model. Binary search tree model is a more restricted version of pointer machine. So it's less than pointer machine.

Just so it's clear what the name of this game is, we require the data to be stored in a binary search tree as it's keys. And then we allow basic operations. Let me make sure I get the right list.

You can follow pointers, and we're going to assume our BST has left triad pointers, right triad pointers, and parent pointers. You can do each of these in constant time. And then the interesting one, interesting thing you can do, is rotate a node and its parent.

So sure you've seen rotations before. If this is my node x, and I do rotate x, it's going to be my definition of rotation, x moves to the root. It's in this case because x is to the right of it's parent it remains now to the right of it's, what now becomes it's child. These sub-trees come along for the ride. This preserves the binary search tree property. And you could get back by rotating y.

OK by this definition of rotate we don't need to distinguish between left and right rotations. We always name the child and it's parent is well defined. That's who we rotate with. So it's just flipping this edge. So these are the operations you can do, and in this model really the only thing you can do is a search.

Suppose I want to search for a key. The model is to do a search I start at the root of the tree, of the binary search tree. I can do these walks. I could do rotations. And I must visit-- if I'm going to search for x, must visit the node with key equal to x.

And we're going to assume throughout these lectures that searches are always successful. You're only searching for keys that are in the tree. Doesn't make that much of a difference if you're searching for keys that are not in the tree. Think of it as searching for the predecessor or the successor. Not a big deal. And so for simplicity, we're just going to assume successful searches.

We're also going to assume no insertions and deletions in this model, because they just make problem a little messier. Some papers have considered insertions and deletions, and to the large extent things just work. So, we're going to focus on searches. Searches is interesting enough for this problem.

Suppose I have n items. Now the model is clear. I store them in a binary search tree. These are the only things I'm allowed to do. Could I imagine beating log n? Well, no in the worst case. This is best possible in the worst case.

Why? Because your tree has to have depth at least log n if you're going to store n things, and if it's binary. So your adversary could always be, could always just look at your tree and say, Oh I'm going to pick anybody who's a log in depth or lower, and if you're going to start at the root and walk left and right to get there, you're going to have to pay log n time every single, every single operation.

So actually there's a lot of sequences, in fact, most sequences. You take an average sequence, this will be the case. But we'll see formal ways to prove that next class. But, easy adversary argument tells you you need to go down, log n in the worst case. But, that's the worst case.

The name of the game in dynamic optimality is to consider every case separately, and do as well as you can all the time. In general, the answer to this question is that, is log n best possible? Depends on the access sequence. Depends on what you're searching for.

So, let's say we're, to make things a little clean, let's assume that the key is we're storing are the numbers, integers, one up to n. So in particular there's n of them. And really we just care about their order, so I'm going to relabel them to be one up to n. And say we search for x 1 first, then we search for x 2, and so on, up to x m.

So, these indices are time, basically. Over time, and you can plot this if you like. Here is time. Here is space. So at the first time unit we search at x 1, then some other place, then some other place. We can have repeats, whatever. OK.

Some search sequences are going to be slow, they're going to require log n time per access. In particular, if I always, if the adversary chooses x i to be the deepest node in the tree repeatedly. But some search sequences are going to be fast. Any suggestions, if you don't have the notes in front of you, of access sequences that we can do in less than log n time per operation. Let's A constant.

There's a lot of possible answers, I think. Yeah?

AUDIENCE: N divided by two.

PROFESSOR: N divided by two. What do you mean.

AUDIENCE: Like it's going to be [INAUDIBLE]

PROFESSOR: Oh, if you're searching for the median in a perfectly balanced tree? Yeah, that's true. So if you're always searching for the root of the tree, in particular, that's going to be good. That's hard to, that depends on the tree though. I'd like a particular access sequence that I can always serve as well, with some tree.

AUDIENCE: n order walk.

PROFESSOR: n order walk. Yeah, good. So this has a name. Sequential access property. If you want to access the elements in order, so one through n, this should only cost constant amortized. Because for any tree you can do and in order walk in linear time. And so per search, if you're careful about how you do this, now this is not totally obvious in the model that I've set up, because I said search always starts at the root. If you let your search start where you left off, then you can definitely do this. It turns out you can assume this or not, it doesn't make a difference. That's maybe a fun exercise.

But if, you can essentially reroute the tree to be wherever you left off. I'm not going to prove that here. OK, but in particular, some trees have this property. Some don't. I mean if you use a red black tree, and you search for one, then search for two, then search for n, you're not going to get constant amortized, because it doesn't realize this is happening.

But if you're clever, of course, this can be achieved. Easy way to see how to achieve it is, this is your binary search tree. You access one, then you do a rotation with the right child of one. So then you have this binary search tree, and then you access two, and then you do a rotation. This is basically a way to make a deck. View a, use a link, simulate a linked list in a binary search tree.

OK, so it definitely can be done by some binary search tree. A stronger version of the sequential access property, it's a little more interesting, is dynamic finger property. This is something that should hold for every access sequence, and it gives you a measure of how expensive an access sequence is. So we're doing x one x two up to x m. And we say, if we measure the key distance between our current access, x i and the previous access, x i minus 1. If that equals k, then ideally we should be able to do log k amortize per operation.

So in the worst case this is going to be log n, but if I look at my space-time diagram here, if for example, if I do sequential access, then this number is always one, and so I get constant time. If I do something like this, where my spatial distance is not changing too much, and this is the reason I numbered the keys like this. What I really mean is, in the sorted order of the keys, how do the ranks differ. But if I number the keys one through n, that's just their absolute difference.

So anything like this, dynamic finger properly tells you you're doing well. Dynamic finger property can be achieved by a binary search tree, but it's quite difficult as you might imagine. Or it's difficult, let's say not necessarily super hard. If you don't need to do it in a binary search tree model, an easy way to do it is with what's called a level linked tree.

So a level link tree looks like this, and I add pointers between adjacent nodes in the same level. This is a pointer machine data structure, it's not a binary search tree, because binary search trees we only allowed to do walk left, walk right, walk parent, But it's an easy way to do dynamic finger. You start from somewhere, you basically walk up to the right level, move over, walk back down, and you can do a search for x i from x i minus one relatively quickly. It turns out this can be stimulated by a binary search tree, but we won't go into it here. Lots of fun puzzles here.

Let's move on to some more bounds or properties. So, next property is called the entropy property, or entropy bound. And it says if key k appears p sub k fraction of the time, then I'd like to achieve a bound of entropy p k log one over p k per operation. This is, everything here today will be amortized.

So, if you haven't seen entropy, that's the definition. It's the entropy of this distribution. We're thinking of these as probabilities, although they're not really probabilities. They're, because this is over a particular sample. We have this sequence x1 to x n, There's no randomness here, but we just measure what is the fraction, what is sort of the observed probability for each of these x i's. And then just say, what is the, what is the entropy of that distribution? In the worst case it's log n, if say everybody's equally likely, everyone's accessed once, or the same number of times.

But if it's highly skewed, like if only one element is accessed, the entropy will be constant. If a small number of elements are accessed, entropy will be constant. So this can be achieved, and in fact, if you disallow rotations, so if I change the model to forbid rotations, all I get to do is set up some binary search tree, and then I have to walk left, walk right, walk parent, from in that binary search tree to do this.

So if I get to see the x i's then I get to build a binary search tree. One binary search tree. This is what's called the static optimal, when you're not allowed to change the tree as you're doing searches. Entropy is the best bound you can get. And roughly speaking, key k appears at height, or depth, log one over p k in the tree, maybe plus one. And you can show there's always a tree where every node ends up at depth log one over p k. Again, not hard, but we won't do it here.

OK, a related property, it's called the working set property. Little bit harder to state. For each search we do x i, we're going to see when that's some key, when was the last time that key was accessed. So, in our space-time diagram we have some access at time i, we want to look backwards in time to the last time that key was accessed. Say that was, well some other time j, and in this interval we want to know, how many distinct keys were accessed. So it's at most, i minus j, but maybe there were some repeated accesses, we only count those as one.

So how many different keys were accessed during that time interval? Claim is, we only have to pay log of that. So what this means, what this implies, is in particular, if you're only accessing say k different elements at all, and you ignore all the other elements, then you'll only pay log k per operation. That's why it's sort of a working set. If you're focusing your attention on k elements for a while, you only pay log k.

But in general, for any access sequence you can compute, what is the working set costs, which is you sum over all the i's of log t i, that is your total cost for the access sequence. Divide by n, that's the amortized cost. OK? That's another nice property, not so obvious is that working set implies the entropy bound, so this is a stronger property.

I guess dynamic finger implies sequential access, but there's no relation between working set and dynamic finger. In fact, they're kind of transposes of each other. Working set is about, if you access a key that was accessed recently, then it's fast. Dynamic finger is saying, if you access a key that is close in space to the previous access, so here we're looking at a key and basically in the very previous time step we're looking at how far away it was vertically, here we're looking at how far away it was horizontally. So they're almost duals of each other.

Be nice if you could have one property that included both dynamic finger and working set. I should mention, again this is not obvious how to do it with a binary search tree, but it can be done. Leave it at that. We'll see eventually some trees that have it.

For now I just want to cover what are the conceivable goals we should aim for. All right, log n is too easy, so we need more challenging goals. So the next property, it was introduced, this is we're getting into more recent territory, 2001, it's called the unified property. Natural name, and it tries to unify dynamic finger with working set. And the rough idea is, that if you access a key that is close in space to a key that was accessed recently in time, then your access should be fast. So, here's the formal statement.

So here's the unified bound. Let's draw a picture. So here we're accessing x i at time i, sorry j. Change of notation. Time j. We want to evaluate the cost of x j, and we're basically going to look in-- I guess one way to think of it is this cone. 90 degree cone, where these are 45 degree angles. And if there's something-- and look at the first thing that you hit in that cone. Is that the right way to think of it? Maybe not.

It's like you're growing a box, this is probably more accurate, and you find the first key that you hit in that box. It's a little more subtle than that because we're only counting distinct keys, and let's say you find this key, and this key is good because it's temporal distance, this time, is only this big. It's spatial distance is only this big. What we're going to pay is, log of the sum of the

[illegible]
spatial distance, and the temporal distance.

And temporal distance is measured as the number of distinct keys accessed in this time interval. So, as long as there is some key in the recent past that is close in space, you take the min over all such keys, magically a unified structure has to find what is the most recent close item and search from there, and achieve log of that. There's a plus 2 just to make sure if these things are zero, you still get a constant. Always have to pay constant time. OK, so that's a unified property.

Fairly natural generalization of dynamic finger and working set. I mean you could change this plus to a product or a max, doesn't make a difference. Yeah. Sadly we don't know whether there is any binary search tree that achieves the unified property. What we know is that this can be done, this property can be achieved by a pointer machine data structure, which is more powerful than a binary search tree. But we don't know whether it's possible to achieve by a binary search tree.

Best bound so far, is that you can achieve this bound plus log log n. So as long as this never gets too small, as long as this quantity never gets smaller than log n, then this thing will be log log n, and it's fine. But in particular, if this is constant, then it's only achieving log log n, so it's not-- there's an additive log log n per operation loss. That's the best binary search, unified binary search tree known.

OK, so this is all good. And this is sort of where the, what we call analytic bounds, here ends. There are various attempts to-- in general we want to characterize, what is the optimal thing we could hope for. When can we do better than log n? These are all specific cases where we can do better than log n, but it's not a complete characterization. There are sequences that have large unified bound, but have no-- yet they can be accessed more quickly by the optimal binary search tree.

And so, while it would be nice to characterize with some clean algebraic, whatever, most of the work, beyond what I've told you about here, is just trying to figure out what opt is. Instead of trying to write it down, try to compute it. Instead of trying to define something that happens to match the optimal, let's just go for optimal. And this is the notion of dynamic optimality.

And in modern terminology this would be called constant competitive, but the paper then introduced dynamic optimality preceded competitive analysis, so it has another name. What we'd like, is that the total cost of all your accesses, so this is like amortized cost, is within a constant factor of the optimal. What's the optimal? This is the-- over all binary search trees, I mean to be precise I should talk about binary search tree algorithms, meaning you specify somehow how to do a search, and it may involve rotations and walks.

So, you know, it could be red black tree, it could be an AVL tree, could be a [? BB ?] alpha tree, anything that can be implemented in this way. Those are all kind of boring. More sophisticated is something like a splay tree. Ideally, you take the min over all-- min over all binary search tree algorithms. The cost of that algorithm, on that access sequence x, and you want the total cost of your algorithm on x to be within a constant factor of the optimal algorithm on x. x is a vector.

So this is what you call the offline optimal, because here you basically get to choose the binary search tree algorithm to be customized to x, and yet somehow you want to achieve dynamic optimality. Open question, is this possible?

And, of course, we're only interested in online solutions. So, you want to build a binary search tree that doesn't know the future, it doesn't know what accesses are to come, but it has to be within a constant factor of the offline solution that does know. And we don't know whether this is possible. Another interesting question, is whether it's possible for a pointer machine, because save for the unified property we know how to get it for a pointer machine.

And there's two versions of this question. You can consider, is there a pointer machine that matches the optimal binary search tree, or you could ask, is there a pointer machine that matches the optimal pointer machine.

That's a little less defined, although there are some-- I have some ideas on how to define that problem, maybe will work on it. But all versions of this problem are open, basically. So, this may seem rather depressing. What else am I going to do for a lecture and a half? But actually, there's a lot of study of this problem, and we do know some good things. For example, we can get log log n competitive. So, not within a constant factor, but we can get within a log log n factor of the optimal. So, that's pretty good.

An easy result is to get log n competitive. Any balanced binary search tree is log n competitive, because best case you could hope for is constant, the worst case you can hope for is log n. So at least within the log factor, but we can do exponentially better. We'll do that next class.

Before we go there, I want to tell you about two structures, to binary search trees that we conjecture are dynamically optimal, but we can't prove it.

So first one, and the classic one, is splayed trees. I don't want to spend too much time on splay trees, but just to let you know what they are if you don't already know. If you want to search for x, you search for x. You do a binary search for x in the tree. You locate x, and then you-- splay trees always move x to the root.

So, this is what we call a self-adjusting tree. It changes the structure of the tree as you do searches, not just when you're doing updates, and there are two cases. If you look at x, and its parent and its grandparent, if they're oriented the same way. So, here it's two left pointers, could be two right pointers. Then you flip them. So we rotate y, and then rotate x, so it's in this model. And we get x y z in the other order.

There's pretty much only one way to move x to the root in that picture. Then the other case is the zigzag case, y, w, x. So, here the two parent pointers are in different directions, one is left, one is right. There's a symmetric picture. In this case, we rotate in the other order. So, we rotate x, and then we rotate y. And in that case you get x nice, x, w, y, and the subtrees hang off. I'm not labeling the subtrees, but they have to be labeled in the same order, left to right.

OK, this is splay trees. You do this for x. x is the thing you search for, now x is up here, then you repeat. You look at it's two parents, it's one of these two cases, you do the appropriate two rotations. Until x is either the root, and you're done, or it's one child from the root, and then you do one rotation to make it the root.

Seems simple enough. It's slightly more sophisticated than an algorithm known as move to root, which is just rotate x, rotate x, rotate x, which would eventually propagate it up. Move to root is a really bad algorithm. It can show its can be a factor of square root of n. It can be square root of n per operation if you're unlucky. Splay trees are always, at most log n amortized per operation, although that's not at all obvious.

Rough intuition, is if you look at the path, the route to x path, half of the nodes, at most half of the nodes go down when you splay. So why is that? Here, see y stays at the same level. z goes down, but x goes up. Here, w stays at the same level, x goes up, y goes down by one, x goes up by two, so you might call that a net improvement. But in general you don't mess up the path too much. Half the items stay where they are, and so it's something like your bisecting-- if you repeatedly search for x, well I'm not really searching for x, but if you repeatedly search for things in that path, you're kind of cutting the path and half repeatedly.

It'll look kind of logarithmic, that's a very vague argument. And in the advanced algorithms, it's proved why this is log n amortized. Question?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Rotate x, rotate x, I think you're right. Yeah, y would go somewhere else, thanks. Yeah, so here it looks like move to root, here it doesn't. Good, and there's lots of things known about splay trees. So for example, they satisfy the sequential access property. There's an entire paper, combinatoric on 1985 Tarjan I believe, proving splay trees have the sequential access property. Not at all obvious, but it's true.

They have the working set property, and therefore they have the entropy property. That's in the original splay tree paper. It's not that hard to prove. Once you prove log n, with a little bit more effort you can prove the working set property. We won't do it here. It's a cool amortization.

Splay trees also have the dynamic finger property. This is a series of two papers, over 100 pages long. Very difficult to prove. In general, splay trees are hard to work with, because it's hard to keep track of what's going on. You can define potential functions, and that will prove this, and with a lot of effort it will prove this. We don't know whether splay trees satisfy the unified property, and more to the point, we don't know whether splay trees are dynamically optimal. This is the big question, and in the original splay tree paper it asked this question, are splay trees dynamically optimal. Still open, I wish I knew the answer, but we don't.

So what the rest of today's lecture is going to be about is a geometric view, which I've kind of been hinting at with this picture, but it's, there's more to it than just plot the points of searches. There's a way, not only to see what the searches are, but to see what the binary search tree algorithm is. So, at this point binary search tree algorithms are kind of abstract. I mean, we've kind of drawn pictures, and how to do specific cases, you know how red black trees or something work. But, you know, it's complicated to write down one of these algorithms.

How would we think about all of them? What would you think about the optimal one? Turns out there's a geometric view, in which it's actually very easy to see what the algorithm is, but it takes a little bit of work to prove that it's the right thing. And it suggests a totally obvious offline optimal algorithm, which we don't know is optimal, but we think is offline optimal. It's like the obvious thing to do, which we'll get to, and we can actually make it online. So, this is-- this gives us a binary search tree that is so obviously optimal, where splay trees is kind of vaguely feels good, this is really obvious, yet we can't prove it. So it's like we're one step closer, but not quite there.

So that's where we're going.

So this geometric view was the topic of [? Deion ?] [? Harmon's ?] PhD thesis, here at MIT. And it's also with John Iacono, [? Danielle ?] [? Caine, ?] who was an undergrad here, and [? Mihal ?] [? Piotrowski, ?] who was an undergrad and a PhD here. It was published three years ago, 2009. So, here's the idea. Access sequence, this is the sequence of searches you do. That's very easy, you just map it to the set of points where, I guess I'm going to switch things on you. Sorry about that. X-axis is going to be space, y-axis is going to be time. And I'm going to look at a specific example of this. Sort of pinwheel. Four accesses at-- the first time we access key three, then we access key one, then key four, then key two. We've been drawing this picture throughout the lecture.

Now the interesting part, is if you look at a binary search tree that actually accesses these elements, here's the exciting part. We're going to map this to a point set, which is which nodes get touched during a search, during each search.

OK, this is the fun part. So, what does touched mean? I erased the model, but in general with a binary search tree, we only have one pointer into the tree. It starts at the root, it can go left, it can go right, it can go to the parent. Every node that gets visited during-- it can do rotations, every node that gets visited by walking, that's what I call a touched node. Just look at all of them.

So for example, in this picture-- color-- suppose, maybe we're lucky, and key three is at the root. So that's the only thing I touch. If that's the case, when I access one, it's definitely not the root. So, in particular, I definitely have to have touched that node. And I'm just going to fill in a reasonable choice.

I happen to know, this is a valid binary search tree. Not so obvious, but you can find one. I'm using here, the greedy algorithm actually, I'm cheating. We'll get to what that is. But in general, some point sets, like this point set, are not valid. There's no binary search tree that has this touch pattern. You just can't jump around, teleport, without getting somewhere. But, this one is valid. So, which ones are valid? Oh sorry I need one more point.

Which ones are valid, which ones are invalid? How did I know that that was invalid, and now this one is valid? Well, there's a theorem tell you. Then what I was just doing there will become clear. So, I want to know when is the point set a valid BST execution, and I claim it is, if and only if, in every rectangle spanned by two points not on a common horizontal or vertical line, in every rectangle, there there's another point.

So, let me draw a picture. I have some point set. We'll look at this one in a moment. In general, I take any two points that are not horizontally or vertically aligned, that spans a rectangle. In other words, the rectangle with those two as opposite corners. There's got to be another point in there, somewhere. Could be on the boundary, could be interior. OK, in fact, you can pump this a little bit.

So once I find some point, let's say on the interior, then my picture looks like this. Maybe I'll just draw it as a circle. Well, here's another rectangle. That one has to have a point inside it. Here's another rectangle, that one has to have a point inside it.

You could keep going, until you know maybe I'd put one here, maybe I put one here. OK if I update the picture then, I still have a little rectangle like that, little rectangle like that, little rectangle like this. At this point, this point is no longer involved in any rectangles essentially, because there's no rectangle between these two points because they're horizontally aligned. I could finish this off, finally, by adding points, let's say at the corners. Once I add points at the corners this will be satisfied. We call this property arborally satisfied. Gotta look up the spelling. Arboral satisfaction. Arboral means having to do with a tree.

OK, so this is saying that point set is treelike, if it has this property. OK this picture is getting a little messy, but if I have any monotone path, like this, this is satisfied. If you look at any rectangle, say this one, it has points-- other points in it. OK, if I look at, I don't know, this one, it has another point in it. But look at this one, that one doesn't count because it's just a vertical line. So it has to have non-zero area.

If you look at any two points, this one and this one, they span a rectangle, there's another point in there. So, monotone paths are good. This point set. If I drew it correctly, is good. I look at any rectangle here, it's got another point in it. Takes some practice to be able to see all the rectangles, but check it. Eventually this one is good. This one's obviously bad. It has lots of wrecked-- we call these empty rectangles, unsatisfied rectangles, that have no extra points in them. So, this is a valid BST execution, this is not.

Let's prove this theorem it's actually not that hard. The consequence of the theorem, is it tells us what we need to do. We are given a point set like this, we want a point set like this. We can define the cost of a binary search tree to be how many nodes does it touch. That's within a constant factor of how many operations you do. If you don't want to, it's pointless to touch a node more than a constant number of times. You can prove that. So, really we just care about how many nodes get touched.

So, what we want to do is find a minimum superset of this point set that is satisfied. We're given a set to represent or access sequence that is not satisfied. We need to act-- we need to touch those points. That's the definition of search. And now we'd just like to also touch some other points that make it satisfied. This is a geometric interpretation of dynamic opt. Can you find the offline optimal, is-- what's the minimum number of points to add in order to make your point set arborally satisfied.

We don't know whether that problem is NP hard. Probably it is. We don't know how to compute it. We don't know how to find a constant factor approximation, unfortunately. Yeah?

AUDIENCE: So, this is for and access sequence, so every time corresponds to just seeking like one individually thing in the tree?

PROFESSOR: Right.

AUDIENCE: OK, that makes more sense.

PROFESSOR: Yes, so the input-- there is a unique thing, a key, that gets accessed at each time. So if you draw any horizontal line, in the input point set there's only one point in each horizontal line. There can be multiple points in each vertical line, which would mean that key gets accessed more than one time. That's the obvious interpretation terms of binary search tree, turns out you don't need to assume either of those things. You could allow a sort of multi-search, where I say during this round you have to access key 5 and 7, I don't care in what order. Whereas normally I say, you have access key 5, then you have to access key 7.

You could do a multi-search, where there's multiple points in a single row. All of the things I'll say work, it doesn't really make a difference as the claim. You could also assume that no key is accessed more than once, so then there's only one-- those would be the opposite extreme. You can assume there's only one key per column. That turns out not to make much difference.

If there's multiple keys in the same position, just spread them out a little and you'll get roughly the same cost.

So, good question. But in the natural interpretation there's one per row, multiple per column, but it doesn't, neither one matters. OK. So this becomes the problem, go from this to this, with a minimum number of added points. We do know a log log n approximation to that problem. That's the best we know. You just take this binary search tree, and apply this transformation, and it tells you if you have a binary search tree, you can turn it into a way to satisfy a point set. But this is actually the best approximation algorithm that we know.

But as you'll see, the geometric view offers a lot of insight. Gives us a lot of power, and in some sense we use it to construct this. OK, so let's prove the theorem. So there's two directions. We'll start with the easy direction. If you have a binary search tree, then it must be arborally satisfied. Then we'll have to do the reverse, actually build a binary search tree out of a point set. That's kind of the harder step.

So, let's say we have two points. Let's say this is that time i, we access key x, and this one is at time j, we access key y. Let's suppose y is greater than x, there's the symmetric picture. We want to argue that there is some other point in this rectangle. OK, here's the plan, let a be the lowest common ancestor of x and y. x is a key, y is a key, they are nodes in the tree.

This is a changing quantity. As the tree wiggles around those rotations, the least common ancestor changes. But at every time, there is, it has some of these common ancestor. Might be x, might be y, might be some other node. OK? I want to look at a couple of things. Let's say, right before x is touched at time i, or I guess right before time i would be the proper phrasing.

So we're at this moment, I want to know does a equal x? If a does not equal x, then I'm going to use the point a comma i. OK? Least common ancestor has the property that x is less than or equal to a, is less than or equal to y. Common ancestor, least common ancestor will be between x and y. And we also know that it's an ancestor of x, and it's an ancestor of y. So, if you're going to touch x, you must touch a. If you're going to touch y, you must touch a. So a comma i is going to be some point here, which is what we need. Except, if a equals x, sorry, what the heck did I do?

Transposed again. OK, this is time i, time j, x, y, sorry about that. OK, here's time i, here is a. So, if a does not equal x, we know at time i, we must access all ancestor of x, we must touch all ancestors of x, before we touch x. Therefore, this is a point and we're done, unless a equals x at this time, because then it's the same point. We didn't get a third point, we at least need to find some different point other than these two.

OK, so that would be good. We can-- so it must actually be that at this time, at time i, a is right here. Let's then look at time j. Same deal, at time j, here's y that gets access. We must also access a. Now it could be its equal to y, or it could be somewhere else. It could be here, or could be still here. In those cases we're happy, but this one also could have been at the corner. The case we're not happy is when b equals x, sorry, a equals y. Getting my letters mixed up. As long as a does not equal y, then we can use the point a comma j, that must be, these are always in your execution, and if a does not equal y, then we're, that's a third point and we're done.

So we're left with one more case, which is that at time i, a is here, here's a, and at time j, a is here. Question is, what happened in between? a changed. For a to change, something in here has to get rotated. That's the [? clam. ?] I mean here's the picture, you have a, actually-- picture is at the beginning a is x. So, here we have x, and then y is an ancestor, sorry, y is a descendant of x, because this is a. And then somehow we have to transition to the reverse picture, which is that y is the least common ancestor, x.

And for this to happen, somebody here has to get rotated. I guess, in particular x. At some point x had to overtake y, had to be moved up. So, at some point x was rotated, and that would correspond to a point here. So else x must be rotated at some time k, where k is between i and j.

And if we set it up right, I guess because here a was still x, and here a is y, then it must be strictly between. Maybe like this would be what we want. And so then we use the point k comma x. Is that more or less clear?

This was the easy case. I guess it depends what you consider easy. Here we're just, we're given a search tree and the short version is, look at the least common ancestor. It's got to move around at some point, or not, either way you're happy. So the least common ancestor gives you the points you care about. So for that we just needed the least common ancestor idea. For the other direction, if we're given a point set that corresponds to, that has this satisfaction property, we have to somehow build from scratch a binary search tree. How the heck are we going to build a binary search tree?

Well, with treaps. So, this is the other direction. Treap is a portmanteau of tree and heap. Underline it correctly, tree, heap. So, it's simultaneously a binary search tree and a min heap, in this case. It's a binary search tree because it has to be. It's a binary search tree on the keys.

It's going to be heap on a different set of values. So, binary search tree on the keys, and it's heap ordered, min heap ordered, by next access. I didn't say I was going to give you an online algorithm, this is an offline algorithm. So it looks-- if I look at a key, and it's going to be accessed next, it better be at the root.

Next access will always be at the root, if you're heap ordered. If you're a min heap by next access. This is great, that means the thing you're searching for is always at the root, and it's basically free to touch. But you may choose to touch other nodes. And in fact, we're told how to touch notes. We're given a pattern that says, well, at this time you have-- you will touch this node, this node, and this node.

By that definition, this one will be at the root, but you're going to touch these nodes, and possibly you could rotate them. Change the tree. You'll have to, actually, because next access time is constantly changing. And as soon as you access an item, it's next access goes in sometime in the future, possibly infinity, and so you'd like to start pushing it down in the tree.

So, what we need to do is show that we can maintain-- it's always going to be a binary search tree, because we only do rotations. We have to somehow use rotations to maintain this heap order property, and only touch the nodes that we're supposed to touch. That's the challenge.

So, I should mention, this is not a uniquely defined tree. Usually treaps are unique, if you specify a key order, and you specify a heap order, there is exactly one tree that satisfies both of them. But here it's not quite unique, because the next access time-- there are many keys that are going, sorry, next touch. Next touch time.

So, for example, at this moment all three of these nodes are going to be accessed at the same time. And so you don't know, or I'm not specifying how they're supposed to be heap ordered in the tree. Just break ties arbitrarily, it doesn't matter. OK. So, let's look at a time i. When we reach that time i, the nodes to touch form and connected subtree containing the root.

Because according to heap order, they're all, they all want to be at the root. So we break ties arbitrarily, somehow, you know all the nodes that we're supposed to touch, live in some connected subtree of the root. Sorry, some connected subtree containing the root. Everything down here has a later next touch time, and so they're below, by definition of heap order.

OK, now one of these is the one we actually want to access. But we need to touch all of them, so touch all of them, you know, navigate. Walk left, right, whatever. The big question is, what should we change this tree into? I'd like to change it to some other top structure. I can't touch anything down here, I'm only allowed to touch these points. I only want to rotate these somehow. There's a convenient theorem, if you have one tree and you want to convert it into another, you can always do it in a linear number of rotations.

So, rotations are basically free and this model, it's just about how many nodes we touch. We're told which nodes to touch. We want to somehow rearrange them to restore this heap order property. Right? As soon as we touch all these nodes, their next touch time will be sometime in the future. We need to rearrange the tree to still be heap ordered by that new next touch time.

OK, here's what we do. I mean there's only one thing to do. Rearrange those nodes in this connected subtree to be a local treap by the new next touch time. These are the only nodes to get a new next touch time, so it's more or less unique how to rearrange them. Do that.

Now the hard part is to argue that the whole thing is now a treap. Why was it enough to only modify these nodes? Maybe you set one of these nodes to have a very large next touch time, so it's got to be really deep down there. And you can't afford to push it down deep, because you're not allowed to touch any of these nodes. Looks worrisome, but turns out, it just works.

So, if there were a failure, picture would be like this. We rearrange these nodes perfectly, in particular, let's look at some node x that has a child y, that was not touched. So x was touched, all of it's ancestors were touched, but y was not touched. So we know that the next touch time of x is greater than or equal to the next touch time of its parent, of it's ancestor, up to the root. The worry would be that the next touch time of x is greater than the next touch time of y.

So suppose next touch of x is greater than next touch of y. This would be a problem, because then you would not be a heap. So, claim is we get an unsatisfied rectangle based on x and it's next touch time, and y and it's next touch time. So let's draw the picture. Here's time, here's space, I'm going to assume by symmetry x is to the left of y and [? keyspace, ?] and now we're supposing the next touch time of y is earlier than the next touch time of x.

So it looks like this. So this is next touch time of x, this is the next time of y, and I claim that there are no other points in here. That would be a contradiction, because we assume that the thing is satisfied.

To prove this, I need to go back a little bit to the definition, over here. There are actually a couple of different ways to think about satisfaction, which I was getting at here, but I didn't solidify. So, I said OK, if you have two points and that rectangle is satisfied, there is some point, possibly in the interior, in that rectangle. But if it's interior, then I can keep going. If I keep going, in the end I can conclude that not only is this rectangle non-empty, but there has to be a point on one of these two sides, because if I choose any other point, I get a smaller rectangle.

Eventually, I have to get one on one of those two sides. Could be at the corner, or this corner, but one of those two sides has to have a point on it. Also, one of these two sides has to have a point on it. It could be both of these constraints are met at once by having one point in the corner. Well that's a somewhat stronger formulation, but equivalent formulation of the satisfaction property. So in particular over here, it should be the case that there's a point, either here or here, and there should be a point, either here or here. I claim that one of those is violated.

You think I'd know this stuff, I wrote the paper, but-- it's all these subtle details, easy to get wrong. I think what I want to look at is, now, which is the moment we're drawing this diagram, versus the next time of y. Sorry, so this is x comma now. That's what my diagram looks like there, so I think that's what I mean. So ignore this picture.

What we learn, what we know is the next access to x, is sometime in the future. That's what we're told here. Next touch of x is greater than next touch of y. So next touch of x is up here, which means this is empty. OK, if that's empty, this better not be empty. All right that's what we claim, one of these two has to have a point in it. This one's empty, so this better not be empty, but I claim this is empty. Suppose it's not, suppose it is, suppose it's not empty-- Wait, no, one of these. Suppose it's not empty, should be the correct scenario. Look at the left most point in this range. This guy.

It's a point between x and y, in terms of key value. So, in this picture, where could it be? Where are the points between x and y in this diagram? They have to be in the left subtree of y. Right? The only points in a binary search-- if x and y are-- if y is a child of x, the only points in the binary tree that are between x and y, or the left subtree of y, or there's a symmetric case, but in this picture left subtree of y.

But I put an x here. That means that whoever we're looking at, some point between x and y, has to be in this top tree. Contradiction, done, OK? If this guy is in there, then y was also in there, which meant there was a point here, and that's what we're assuming did not happen. This is supposed to be an interface between inside the set of touched nodes, and outside the set of touched nodes. So there can't be any points in here, which means this is empty, which means you weren't satisfied. OK, maybe I should write down the words to go with that argument, but--

So, this part is empty by next touch of x being greater than next touch of y. And this part is empty else, or it's empty because any key between x and y is in the left subtree of y, which would imply if it's touched then so is y. But y cannot be touched, by assumption.

So that's the end of that proof. A little bit longer, but hopefully it's pretty clear at this point. Question?

AUDIENCE: Can't there be something like a descendent of x, but an ancestor of y, instead of being in a subtree with y?

PROFESSOR: Would, so you could say it could be in between here. It could be a descendant of x, but an ancestor of y, but we're assuming here that this was a child relation. y was a child of x. This was a-- I didn't say that at the beginning. On a claim that's a global treap, if it's not a global treap, there is some edge that's violated, that does not have heap order property. So assuming this was an edge, x is OK, it was in the local treap, y somehow is going to be bad, because it's next touch time was, should be higher. Should be above. Other question?

AUDIENCE: So that [INAUDIBLE] you drew, and all the everything in that subtree root that gets taken to-- Like the picture on the right is also, it's the same subtree right? But after you change their [INAUDIBLE]

PROFESSOR: These two subtrees have the same set of nodes, they've just been rotated somehow.

AUDIENCE: OK, but then how do you-- but didn't we change the next touch times of all the nodes in there? So how you do know if the root is still going to be some guy in that subtree?

PROFESSOR: Buy this argument. So the question is say, after we do this, we make some root, which is going, the root is going to be among all the nodes that just got touched now, who is going to be touched next? That's who we put at the root. And the claim is, that is globally the node that will be touched next. Why? By this argument.

If there were some other node down here that has a smaller next access time, then-- but we know that it was heap ordered before, so all of these guys are heap ordered. So, this guy would then have the minimum. And then we look at that interface, and the claim is by the satisfaction property, actually this point should have been in the set.

So, what this tells you is the guy that has to be accessed next, in particular, must be in your set. Must be touched now. Kind of magical. Well actually, it's because they're tie-breaking I think, that this works out. Anyway, another question?

AUDIENCE: So, like when you're rearranging the [INAUDIBLE] in the sort of subtree on top [INAUDIBLE]

PROFESSOR: Yeah, we're doing this transformation of the top tree into the local tree by rotations. I mentioned--

AUDIENCE: [INAUDIBLE] It's always--

PROFESSOR: Right, so we're staying within the binary search tree model, and in particular then we stay a binary search tree. So, we can't mess things up. And there's a nice theorem that if you have two binary search trees on the same, keys there's a way to get there with the linear number of rotates, so that's for free.

So, our new cost model is just to count points. The cost of this access is one, the cost of this access is two, two, three. If we just count how many nodes are touched, then the cost of the binary search tree is equal to the cost of the minimum satisfied superset of your point. So this is the problem now. Sadly we don't know how to solve the problem. We do know some things.

I don't think I'll go through an example of this, it's not very exciting. You could run through this picture, and see how the binary [? structure ?] changes. Actually the binary [? structure ?] won't change it all here, so it's a little anticlimactic of an example. You can see it in the notes. I want to get to the greedy algorithm, so let's go here. Last bullet.

The idea is to imagine your points are added one at a time, bottom up. So I'm going to do the same example. First we add this point. That's a satisfied set, done. In general add necessary points on the same row as the search. OK, there's nothing to add here.

Next point we add is over here. I've got the orientation correct, yeah. So this was three, this was one. Is this satisfied? No, there's a bad rectangle here. There's two obvious ways to satisfy the rectangle, I could add a point here, or I could add a point here. I'm going to add a point here, because that's the row, this is current time. OK, it's like a sweep line algorithm. We go up, next point is over here. That's the Next position to sweep line. Now there's a bad rectangle. We fix it by adding this point. Now we're good, all rectangles are satisfied.

Next level is, there's a point here. Now there's two bad rectangles. This one, I'm going to add a point here, and this one, I'm going to add a point here. Now we're good. And that should be what I did here. Yeah. So that's how I found that point set, and in general claim is this is a good algorithm. It seems, in fact, pretty obvious. There was a choice of course, we could satisfy this corner or this corner. Or some monotone path in between. But the claim would be that-- doesn't make that big a difference. Doing things later is always better. It's kind of a lazy property. Or you could think of this--

Originally actually this algorithm goes back in tree land. And in tree land, if you follow through this reduction, which we did, you know if you convert this into a treap, what this is saying is, look you search for an item, you follow a path. Take all the nodes on the path that you follow, rearrange them optimally for the future, build a heap based on the next access time, next touch time, whenever that happens to be. Next access time actually, in that case. That is equivalent to this algorithm.

That seems like the right thing to do offline. You visit your item, you do the minimum amount you have to do, which is following the search path. You rearrange those items to be optimal for the future. It's an offline algorithm, seems like a really good one. The only thing it's missing out on is maybe you should go off the path and bring other guys closer to the root. But if you believe, which we don't know how to prove, if you believe that there's no-- it doesn't really buy you anything to go off the path now, you can always do it later, then this algorithm is optimal offline.

And in terms of the point set it's-- in the point set of view it's kind of nice because it almost looks online, right. You only have to look at each time and add the points at that time that are useful, in terms of the past. You only had to satisfy the rectangles of the past. So this is where things get interesting, because in terms of a tree view, with this transformation, this looks like an offline tree because it needs to know the future. And it does if you want to build a heap. OK, but if you look at in the geometric view, suddenly it looks online, because you're only looking at the points and all the points you've accessed in the past.

So, in fact, there's a way to make this online. This is where things get interesting. There is a transformation that if you have an online satisfying point set, meaning you can decide what points to add based only on the past, not on the future. So it's a fancier version of this transformation, you get an actual online binary search tree.

So, this algorithm goes back to the '90s I think. Actually '88, it's in a thesis 1988 as rediscovered by my PhD adviser in 2000. It's totally natural algorithm, but they thought it was an offline algorithm. With this view, it's an online algorithm. Let me quickly convince you, or sketch to you, how we make this online.

So we do the same thing, except we don't know how to heapify, because we don't know the next access times. But we know whatever we touch, it's some connected subtree of the root. We know what we touch. We touch whatever greedy tells us to touch. We'll touch all these guys, whatever.

How do we rearrange them? We don't rearrange them. We store them into something called a split tree. Split tree has the feature, it's a tree. Binary search tree, and if you ask for some item x, you can move x to the root. So then you have a left subtree of x, right subtree of x, and then delete x, and now you're left with things that are less than x, and things that are greater than x. Things are greater than x. And you can do all this in constant amortize time.

This is what we need, if you think about here. I'm going to take all these items that were touched, throw them into a split tree. My resulting structure will be a tree of split trees. So think of it as, when I touch all these items I just sort of throw them all in the root, but that root is represented by a split tree. And then hanging off here, there are other split trees which may have several keys in them.

When I do, when I now touch a node, you can show that if I'm trying to touch some node here, I can't just magically touch a node here. I've got to, I had to have followed in the actual tree, whatever the optimal tree is, I had to follow some route to leave path. So in fact, the predecessor and successor here had to have been touched before I touched this one. Which means I'm going to split those nodes.

So, when I actually-- this is basically lazy evaluation. When I actually access something in here, that means I want to pull it to the root. I wanted to pretend that it was at the root, that's what the treap would have done. So basically pull it up, be a root, split, because now there's two structures left where I don't know their orders, but I know that this item was first. And so I end up with-- I don't actually remove it, but I remove it from the split trees, so I make it look like this. And here's the guy where I wanted to access something.

So, I split this root into two split trees, and I have an individual node up here. If I can do this in constant amortize time, it's as if this node was at the root in the first place. And so I simulate this perfect treap order, but using a data structure, split trees, which can actually be solved. How do you solve split trees? You just do the obvious thing, more or less. Ideas, you know red black trees, take your favorite balanced binary search tree, red black trees work fine.

If you're given a node, you can split there. You basically just carve it in half. How much does it cost to split? Well if you're a little bit clever, let me go to a board of cleverness. The one thing we have to optimize for is, what if you're splitting like right here, very close to the left? If you're clever, you'll search from here, and cut off this part in time basically proportional to the height of that tree. OK, you might get some propagation up here, but that's very small amortized.

OK, on the other hand, if you search over here, you'd like to spend only time proportional to this. So in general, if you just search in parallel here, you can split in-- let's say this has size n one, and the rest has size n two. You can split in order log the min of n one and n two. And you can show that if you just-- that's straightforward splitting, that's really easy to do.

You can show that that implies constant amortized, because either you're cutting off a very little nibble and the cost is small, or you're cutting things more or less in half, but that can't happen very much. Then you charge to the fact that log n is going down by one, and overall you get a linear cost to splitting nodes. It's an amortization. Now the trouble is this is not a binary search tree. How in the heck do I have to pointers that in parallel search?

Well you have to take these two halves of the tree and interleave them to make them a binary search tree. It's kind of awkward, but just put it, mash it all together. Fold it in half, basically and turn it upside down, and you've got a binary search tree. So there's some messy stuff there, but that's just a hand-wavy argument that you can make greedy online. Next class we'll talk about lower bounds, which almost proved the greedy is optimal. But not quite.