[SQUEAKING] [RUSTLING] [CLICKING]

ERIK DEMAINE: All right, welcome back to data structures land. Today we continue and complete our segment on binary trees. So this is part two. If you missed part one, go back and watch part one.

Last time, we talked about binary trees in general. We had each node stored an item, and also a left pointer and a right pointer to other nodes, and a parent pointer to another node. This was an example of a tree. B, and C are A's children. A is the parent of B and C, and also the root of the entire tree.

We defined the height of a node. We didn't use this too much yet. But we're going to use it a lot today. So remember, the height is as drawn in red here. Height of the node is the length of the longest downward path counting edges. So B, for example, has a length 2 paths. So we write a 2 here. You can also think of it as if you just live within the subtree rooted at B, B subtree, then what is the maximum depth of those nodes, if you prefer to think about it that way. Either way is fine. And in particular, we distinguished h, the height of the root node, as the height of the entire tree.

And what we achieved last time was basically all of our operations ran in order h time. So we had subtree insert, subtree delete, subtree first and last. We could compute the predecessor and successor of a node, all in order h time. So as long as h was small, we were happy.

And remember, what does predecessor and successor mean? It's talking about an implicit order in the tree, which is what we call traversal order, which is defined recursively as recursively traverse the left subtree, then output the root, then recursively traverse the right subtree. So in this example, the traversal order is F is the-- if you go all the way left, that was the first in the traversal order. Then we have-- right, I'll make me some space here. Then we have D, then we have B. Then we do the right subtree of B, which is E. Then we have the root, because we finished the left subtree of the root. So that's A. And then we have C.

So there's an implicit linear order encoded by this tree. And the whole point of binary trees is that we can efficiently update the tree much faster than we could explicitly write down an order in an array or something like that. So binary trees let us quickly-- now, quickly is not so quick right now. Because everything is order h. And in the worst case, h is linear. Because we can have a tree like this.

But today, we're going to make-- we're going to guarantee that h is log n. And so the goal of today is to take all of these operations that run in order h time and get them to run an order log n time, just by modifying the data structure we've already seen. So we've done a lot of the hard work, just a little bit more work we need to do today on something called AVL trees or height balance.

But before we get there, I want to talk a little bit more-- at the very end of the last lecture, we talked about once you have these subtree operations-- so I can insert and delete in the subtree-- how do I actually use that to solve the problems that we care about in this class, which are sequence data structure and set data structure? So we talked mostly about the set data structure last time. So in general, we're going to define what traversal order we maintain by a binary tree.

And so for a set, because for the set interface, we're interested in doing queries like find_next and find_previous, given a key, if it's not there, tell me the previous one or the next one, this is something we could do with binary search. And so the big, cool thing that binary trees let us do, if we let the traversal order always be all of the items stored in increasing key order, then we are effectively maintaining the items in order-- in the traversal order sense. Again, we're not explicitly maintaining them in order. But up here, we're maintaining a tree that represents items in key order. And so this lets us do a subtree_find operation-- which you could easily use to implement find, and find_previous, and so on-- as follows.

We start at the root of the tree. So we can say, node equals root initially. And then we can recursively search for a key k as follows. We check, well, if the item at the root has a key that's bigger than k-- let me draw a little picture. So we're at some node here. This is a node. And it has left subtree and a right subtree. And there's some item with some key. So if the key we're looking for is less than the node's item, that means it's down here in the left subtree. And so we recurse on node.left.

If they're equal, that means that this item is the item we're looking for. So we can just return it or the node, depending on what you're looking for. And if the key in here is greater than the key we're looking for, then we'll recurse to the right. If you think about it a little bit, this is exactly binary search on an array. It just happens to be on a tree instead.

If you think of an array like this, what does binary search do? It first looks at the key in the middle. I'm going to draw that as the root. And then, it recurses either on the left chunk, which I will draw recursively, or on the right chunk. And so if you happen to have a perfect binary tree like this one, it is simulating exactly binary search in this array. But this we're going to be able to maintain dynamically-- not perfect any more, but close. Whereas this we could not maintain in sorted order.

So this is like a generalization of binary search to work on trees instead of on arrays. And for this reason, set binary trees are called binary search trees, because they're the tree version of binary search. So there's many equivalent names. So binary search tree is another name for set binary tree.

And the key thing that makes this algorithm work is the so-called binary search tree property, which is all the keys in the left subtree of a node are less than the root, or of that node, and that key is less than all the keys in the right subtree. And this is true recursively all the way down. And so that's how you prove that this algorithm is correct by this property.

Why is this true? Because if we can maintain traversal order to be increasing key, then that's exactly what traversal order means. It tells you all the things in the left subtree come before the root, which come before all the things in the right subtree. So this property implies this one.

And how do you maintain things in increasing key order? It's pretty easy. If you want to insert an item, where does it belong? Well, you do this search to find where it would belong if it was there. If it's there, you can overwrite the value stored with that key. If it's not, this search will fall off the tree at some point, and that's where you insert a new node in your tree. That was covered in recitation, so I don't want to dwell on it.

What I want to focus on today is the other application. How do we-- this is for representing a set, which is relatively easy. A challenge to sort of set ourselves up for, but we need a little more work, is to make sequence binary trees. So suppose I have a binary tree, and what I would like-- we mentioned at the end of last time-- is that I want the traversal order of my tree to be the sequence order, the order that I'm trying to represent that's changed by operations like insert_at.

So I'd just like to do the same thing. But now, I have to think about how do I do a search, how do I do a insert_at, that sort of thing. And here is an algorithm for what I would like to work. But it's not going to quite work yet.

So suppose I give you a subtree, so specified by a node. So there's all the descendants of that node. And I'd like to know what is in the traversal order of that subtree, which starts here, and ends here, and the root will be somewhere in the middle. Give me the ith node. So if I ask i equals 0, I want to get this leftmost descendant. If I ask for i equals the size of the tree minus 1, I want to get the rightmost descendant. That was the first and last in the subtree that we talked about.

But we know how to find the first and last. Just walk left or walk right. But we don't know how to find the ith node-- in order h time is the goal right now, not log n. And the idea is, well, it seems like size matters. [CHUCKLES] Sorry if you heard otherwise.

So in particular, I mentioned size when I was talking about the last node in the sequence. The index of that node is size of the subtree minus 1. So let's define the size of a node to be the number of nodes in its subtree-- we were calling that subtree(node)-- including the node itself. So if I somehow knew the size, this seems important for understanding indexes. Let's just assume that I knew that magically in constant time.

Then, I claim that the size of the left subtree-- so why don't I expand this diagram a little bit? So we have node as before. But we have left subtree and a right subtree. So this node here is node.left. This node here is node.right. They might not exist, but let's ignore those exceptional cases for now. Let's suppose we knew not only the size of node, but we knew the size of node.left, so that is the size of this tree on the left. I'm going to call that nL.

So let's suppose that there are nL nodes down here. I claim that lets me do the equivalent of a binary search. Because I'm looking for some index i. And if i is less than nL, then I know that it must be down here. For example, if i equals 0, it's going to be in the left subtree, as long as nL is greater than 0, right? So that's exactly this check. If i is less than nL, I'm going to recurse to the left, call subtree at of node.left, i. That's what's written here.

If i equals nL, if you think about it for a second-- so nL is the number of nodes here. And so that means this node has index nL. The index of this node is nL. And so if i equals-- if the one index we're looking for is that one, then we just return this node. We're done.

And otherwise, i is greater than nL. And that means that the node we're looking for is in the right subtree, because it comes after the root. Again, that's what it means. That's what traversal order means. So if we define it to be sequence order, then we know all the things that come after this node, which is index nL, must be over here.

Now when we recurse down here, our numbering system changes. Because for node, 0 is here. And then for node.right is here. So we need to do a little bit of subtraction here, which is when we recurse to the right, we take i minus nL minus 1-- minus nL for these guys, minus 1 for the root node. And that will give us the index we're looking for within this subtree.

So my point is, this algorithm is basically the same as this algorithm. But this one uses keys, because we're dealing with a set, and in sets we assume items have keys. Over here, items don't have to have keys. In fact, we're not touching the items at all. We're just asking, what is the ith item in my sequence, which is the same thing as what is the ith item in my traversal order, which is the same thing as asking what is the ith node in the traversal order? And this algorithm gives it to you exactly the same way in order h time.

Now, I'm not going to show you all the operations. But you can use subtree_at to implement get_at set_at. You just find the appropriate node and return the item or modify the item. Or you can use it to-- most crucially, you can use it to do insert_at delete_at. This is a new thing we've never been able to do before.

What do you do? Just like over here, if I'm trying to insert an item, I search for that item over here. So if I'm trying to insert at i, for example, I look for i. And then for insert_at i, want to add a new item just before that one. And conveniently, we already have-- I didn't mention, but we have a subtree insert. We had two versions-- before and after. I think we covered after, which I use successor before I use predecessor. But we can just call subtree insert before at that node, and boom, we will have added a new item just before it.

And great, so magically, somehow, we have inserted in the middle of this sequence. And all of the indices update, because I'm not storing indices. Instead, to search for an item at index i, I'm using the search algorithm.

But there's a problem. What's the problem? This seems a little too good to be true. I insert in the middle of this tree, and then somehow, I can magically search and still find the ith item, even though all the indices to the right of that item incremented by 1. It's almost true. Answer? Yeah?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Because we have to update the sizes, right. I didn't say, how do I compute the size of the left subtree? So that is the next topic.

We're almost done. This will actually work. It's really quite awesome. But for it to work, we need something called subtree augmentation, which I'll talk about generally. And then, we'll apply it to size.

So the idea with subtree augmentation is that each node in our binary tree can store a constant number of extra fields. Why not? And in particular, if these fields are of a particular type-- maybe I'll call them properties. I'm going to define a subtree property to be something that can be computed from the properties of the nodes' children. So I should say this is of a node.

So children are node.left and node.right. You can also access constant amount of other stuff, for example the node itself. But the point of a subtree property is it's downward looking. If I have a node here, and I want to compute some property about it-- call it, we want to store P of the node-- and suppose we already know P over here, the property computed for the left subtree or for the left node, and we already know the property for the right node, then what I'd like is for this to be computable in constant time.

So I can compute P of this node given P of the left node and P of the right node. That's a subtree property. Now, in particular, size is a substrate property. Why? Because I can write this kind of recurrence, node.size equals node.left.size-- this is very tedious to write-- plus node.right.size, plus? 1, thank you. The size of the entire subtree here, called node, is the size of the left subtree plus size of the right subtree, plus 1 for that node itself.

So this is an update rule. It takes constant time to evaluate. It's two editions. Sorry, my t's look kind of like plus signs. I'll make the pluses a little bigger. So we're just summing those three things. Boom, we can get node.size.

So I claim that as long as my property has this feature, I can maintain it dynamically as I'm changing the tree. Now, this is a little bit of a forward reference, because we haven't said exactly how we're going to change the tree yet. But question?

AUDIENCE: If node.size is recursive, then how is it happening in constant time? Wouldn't it be happening [INAUDIBLE]?

ERIK DEMAINE: Why is this-- OK, good question. So one natural way, you can think of this as a recursion, which gives you a recursive algorithm. So I wrote-- but didn't write it. But I could have written size of node equals this-- size of node.left plus-- and that would give you a linear time algorithm to count the size. And if you don't have any information, that is what you would do. And that would be very painful. So that would make this algorithm really slow. If I'm calling size as a recursive function, it's bad.

What I'm instead doing is storing the sizes on every single node and pre-computing this. So in fact, I'm going to define the size of node in-- so this is the definition mathematically. But the algorithm for this function is just going to be return node.size. So that's constant time.

So the challenge now is I have to keep these sizes up to date, no matter what I do to the tree. And you could look back at last lecture and see, OK, what were all the changes that I did in a tree? We only did changes during insert and delete. And I will just claim to you, when we did insert and delete, what they ended up doing in the end, they add or remove a leaf of the tree. Remember, a leaf was a node with no children.

So let's just think about if I add a new leaf in a tree-- here's a tree, suppose I add a leaf here-- which subtrees change? Well, which subtrees contain that node? It is its own new subtree. Then it's in its parent subtree, and its grandparent subtree, and the overall subtree. In general, these nodes are called the ancestors of this node that we added. And those are the ones that update.

This subtree over here didn't change. It didn't change size. And because it's a subtree property, no subtree property will change over here, because the subtree was untouched. So when I touch this guy, I just have to update the subtree property here, update the subtree property here, update subtree property here. How many of these are there? Yeah? h-- I'll say order h to be safe. But I think it's exactly h. So also, when I remove a leaf, the same thing-- if I remove this leaf, then the subtrees that change are exactly its former ancestors.

Cool, so we're going to update those order h ancestors in order up the tree. So what do I mean by update? I mean apply this rule. For size, it's this rule. But in general, the subtree property gives me an update rule that takes constant time.

And so I'm going to apply that update rule to this node, which will fix whatever property is stored in there. Maybe there's more than one property. And then I'll apply it to this node. And because this is already correct by induction, and this is already correct because I didn't touch this subtree-- it's unchanged-- then I can update the value at this node-- the property at this node in constant time. Then I update this one. And because this one is already correct by induction, and this one is already correct because this subtree is unchanged, I can update the property correctly here in constant time.

So when I make a change in order h time, because I'm making h calls to this constant time algorithm, I can update a constant number of subtree properties. This is very powerful. Data structure augmentation is super useful. You will use it on your problem set. We will use it again today.

Let me give you some examples of subtree properties. They could be-- common ones are, like, the sum, or the product, or the min, or the max, or sum of squares, or all sorts of things, of some feature of every node in the subtree. In fact, subtree size is an example of such a thing. It is the sum over all nodes in the subtree of the value 1. It's another way to say count the number of nodes.

But you could also say, what's the sum of the keys in these nodes? Or you could say, what's the maximum key in these nodes? Or you could say, what is the maximum value in these nodes? You can take any property. It doesn't have to be key. It doesn't have to be anything in particular. It's very powerful. You can take all sums, products and maintain them as long as they're downward looking-- as long as you're only thinking about the subtree.

Some examples of things you cannot maintain are-- not a nodes index. So if you get a little bit too excited about augmentation, you might think, oh, I could do everything. I needed to support subtree_at, or let's just say, get_at globally, I wanted to know what is the ith node in my tree? Well, I'll just use data structure augmentation and store in every node what is its index, 0 through n minus 1.

I can't maintain that efficiently. Because if I insert at the beginning of my traversal order, then all the indices change. So that's an example of a edit. So if I insert a new node over here, so this guy's index was 0, now it's 1. This guy's index was 1, now it's 2. This was 2, now it's 3, and so on. Every node changes its index. Index is not a subtree property, and that's why we can't maintain it. Because it depends on all of the nodes in the tree. Or it depends on all the nodes to its left-- all the predecessors.

So for example, this guy's index depends on how many nodes are over here on the left, which is not in the subtree of that node. So that's where you have to be careful. Don't use global properties of the tree. You can only use subtree properties.

Another example is depth. Depth Is annoying to maintain, but it's not obvious why yet. We will see that in a moment.

The rest of today is about going from order h order log n, which is what this slide is showing us. So at this point, you should believe that we can do all of the sequence data structure operations in order h time-- except for build and iterate, which take linear time-- and that we can do all of the set operations in order h time, except build and iterate, which take n log n and n respectively. And our goal is to now bound h by log n.

We know it's possible at some level, because there are trees that have logarithmic height. That's like this perfect tree here. But we also know we have to be careful, because there are some bad trees, like this chain. So if h equals log n, we call this a balanced binary tree. There are many balanced binary trees in the world, maybe a dozen or two-- a lot of different data structures. Question?

AUDIENCE: [INAUDIBLE] you said not to think about things on a global level so we'll think of them [INAUDIBLE]. Can you explain what that means a little more?

ERIK DEMAINE: OK, what does it mean for a property to be local to a subtree versus global? The best answer is this definition. But that's maybe not the most intuitive definition. This is what I mean. Something that can be computed just knowing information about your left and right children, that is the meaning of such a property. And those are the only things you're allowed to maintain. Because those are the only things that are easy to update by walking up this path.

And the contrast is that global property like index, it's global, in particular, because I can do one change, add one node, and all of the node's properties change. So that's an extreme example of global. We want this very particular notion of local, because that's what we can actually afford to recompute. Hopefully that clarifies. Yeah?

AUDIENCE: Doesn't size not work with that [INAUDIBLE]?

ERIK DEMAINE: You're right that if we added-- oh, no. OK, let's think about that. If we added a new parent to the tree-- this is not something that we've ever done. But even if we did that, which subtrees change? Only this one.

This node, it's a totally new subtree. But the subtree of this node is completely unchanged. Because subtrees are always downward looking, if I added a new root, I didn't change any subtrees except for one. So size is a subtree property.

Now, there are-- I mean, I could completely redraw the tree. And that's an operation that requires everything to be recomputed. So it is limited exactly what I'm allowed to do in the tree. But I claim everything we'll do, last class and today, we can afford this augmentation. So it's a feature, not of all binary trees necessarily, but of the ones that we would cover. Yeah?

AUDIENCE: What is a min?

ERIK DEMAINE: What is a min?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Binary tree, yeah. OK, this will make a little more sense in a moment when I say what we're actually going to do with trees. We need a new tool for manipulating a tree. What we've done so far, we've done some swapping of items. And we did adding and removing a leaf. That's not enough. We're going to need something else to let us guarantee logarithmic height. And that something else is called a rotation.

What does this something else need to do? This is just a tool for rebalancing the tree. So it should not change the data that's represented by the tree. What is the data represented by the tree? Traversal order. Traversal order is sacrosanct. We're not allowed to touch it. It's already defined two different ways, depending on whether you're using a set or sequence.

So we want to modify the tree in a way that doesn't modify the traversal order. So we're exploiting redundancy. If you wrote down the traversal order in an array, there's exactly one representation of a given order. But in a tree, there's many representations. It could be a long line. It could be a balance thing. They could represent the exact same order on the nodes if you label them right.

In fact, there are exponentially many different representations of the same thing. And we're going to exploit that, the same order, and define-- this is just a thing you need to know. Let me label, A, X, B, Y, C. You can tell I've drawn this diagram before many, many times. This is a very powerful tool in all tree data structures, which are most of data structures. And they are called right rotate of y and left rotate of x.

So if I have this tree-- which I'm just black boxing some of the subtrees into little triangles. If I have a node, and it has a left child, then I'm allowed to right rotate this edge, which means take this edge and go like this-- 90 degrees, kind of. Or you could just think of it as rewriting it this way.

Now, you might also have keeping track of the parent pointer. Parent pointer moves around. Before, this was the parent of y. Now it's the parent of x. So x and y are switching places. But we couldn't just swap these items around, because that would change traversal order. In this picture, x comes before y, because x is in the left subtree of y in traversal order.

And over here, now y is in the right subtree of x. So it comes after x. So in both cases, x comes before y. And indeed, in all of these pictures the traversal order-- I mean, not just for x and y, but also for A, B, and C, the traversal order is consistent. It is A, X, B, y, C, where, when I write a triangle, I mean recursively the traversal order of all the things in the triangle. So if you just apply the traversal order algorithm here versus here, you get the same output, which means these operations preserve traversal order.

Great, so this is a thing that we can do in a tree that won't affect any of the stuff we've done so far. It's a tool that we can use to rebalance. Notice how deep things are in the tree changes. Our problem with this linear tree is that there are some nodes of linear depth. We want to get rid of those.

How? Well, we could take these edges and start rotating them up. If you look at depths, in this picture, A and B are deeper than C. And in this picture, B and C are deeper than A. So it's a trade off. This one moved up. This one moved down. This one stayed at the same depth. So hopefully, if A is too deep and C is too shallow, they can trade off like this.

It may sound difficult, but in fact, there's a pretty simple way, which are called AVL trees, that maintain balance in a particular way called height balance. This is if we take the height of node.left-- actually, I'd prefer to-- node.right, minus height of node.left, so this thing is called skew of the node. I want this to always be minus 1, 0, or plus 1.

So this is saying that if I have any node, and I look if its left subtree and its right subtree, I measure their heights-- remember, that's downward distance, maximum distance to a leaf-- I measure the height of this tree-- maximum height-- and I measure the maximum height of this subtree, I want these to be within 1 of each other. Ideally, they're equal. That would be the perfect case. But let's let them differ by 1.

So maybe this is k and this is k plus 1. Or maybe this is k and this is k minus 1. In this picture, what is the height of this node? It's good practice. k plus 2, good. What's the longest path from this node to a leaf? Well, it could go through this subtree. And that would be length k plus 1, because it's k in here plus 1 for this edge. Or it could be through here, and that's k plus 1 plus 1. So the biggest is to go to the right. So the height-- if I told you the height of these subtrees, we can derive the height of this node. We're going to use that a lot in a moment.

So, the first claim is that if I could maintain height balance, then I will guarantee that h equals log n. So in other words, height balance implies balance. So let's prove that first quickly. And then, the interesting part is how do we actually prove-- or how do we actually maintain the balance property? We're going to do that using rotations. But how is a big question.

So why does height balance imply balance? So what this is saying is that all height balanced trees have logarithmic height. So what I'd like to think about is sort of the least balanced height balanced tree. The least balanced one is going to have every node a mismatch. Let's say the left subtree is shallower than the right subtree by 1, and recursively all the way down. So every node has a gap here, a-- what do we call it-- a skew of 1, which I'm going to write-- I'm going to introduce some notation. I'll write a dissenting rightward arrow of this one is higher than the left subtree.

So the easy way to think about this is this is sort of our worst case. This is going to be the fewest nodes for the maximum depth. Let's just count how many nodes are in this tree. I'm going to write that as a recurrence, which is the number of nodes in a tree of height h. So if this whole tree has height h, as we said in this picture, if I just subtract 2 from all these numbers, then this one has height h minus 2, and this one has height h minus 1.

So how many nodes are in here? Well, this is a recurrence I'm going to write. So this will be N sub h minus 2. This will be N sub h minus 1. And then I just count how many nodes are in this picture. It is Nh minus 1 plus Nh minus 2 plus 1, or this node.

Now you might ask, what is Nh a recurrence for? But it is the number of nodes in this sort of worst case if the worst case has total height h. So you can also think of it as what is the minimum number of nodes I could have in an AVL tree, which is a height balanced tree, that has a height h in a height balanced tree?

OK, so now I just need to solve this recurrence. This recurrence look familiar-ish? It's like Fibonacci numbers. If I remove the plus 1, it's Fibonacci. And if you happen to know the Fibonacci numbers grow as, like, a golden ratio to the n, then we know that this is exponential, which is what we want. Because if Nh is exponential in h, that means h is logarithmic in N, because log is inverse of exponential.

But maybe you don't know about Fibonacci numbers. And so we can still easily show that this is exponential as follows. I want to prove that it's at least an exponential, because that gives me that h is at most logarithmic. So we need a lower bound.

And so we have these two terms which are hard to compare-- Nh minus 1 and Nh minus 2. It's kind of ugly. But if we're allowed to be sloppy-- and we'll see if we're not too sloppy-- and still get an exponential answer, let's just make them equal like so. So this is a true statement, in fact, strictly greater than.

Why? Because I removed the plus 1. That should only make something smaller. And I replaced Nh minus 1 with Nh minus 2. Here, I'm implicitly using a fact, which is obvious by induction, that this tree on height-- if I take this tree versus this tree, this one has more nodes than this one. If I have larger height, this construction is going to build a bigger tree, at least as big. It doesn't even need to be strictly bigger.

So certainly, Nh minus 1 is greater than or equal to Nh minus 2. Now, this is 2 times Nh minus 2. And this is an easy recurrence. This is just powers of 2. I keep multiplying by 2, and subtracting 2 from h. So this solves to 2 to the h over 2, maybe with a floor or something. But I'm using a base case here, which is N sub 0 equals 1. Maybe it's a ceiling then.

But the point is this is exponential. So this implies that the height is always, at most, 2 times log n. This 2 corresponds to this 2. If you just invert this formula, this was a number of nodes is going to be at least 2 to the h over 2. And so h is, at most, 2 log n. So it's not log n. That would be perfect. But it's within a factor of 2 of log n. So AVL trees are always quite balanced. Number of levels is at most double what you need to store n nodes. Great.

We're left with the main magic-- not domain magic. That's different. And let's see, we're going to use subtree augmentation. Keep that. Big remaining challenge is how do we maintain this high balance property using rotations? We have all the ingredients lined up for us.

We have subtree augmentation. What does that let me do? It's relevant to AVL trees. Well, it lets me store height. I need to be able to compute the height of a node. That, in general, takes linear time, because I have to look at all the downward paths-- all the leaves within that subtree. But height is a subtree property-- so, yes-- height. Why? Because-- let me just write it here-- node.height equals 1 plus max of node.left.height and node.right.height and of max. Let me put this in a box.

This equation, or I guess it's an assignment operation-- this is a 1-- is the thing we've been doing over and over. When I said what is the height of this node, you just figured that out, right? You took the height of the left subtree maxed with the height of the right subtree and added 1 to account for these edges.

So this is a general update rule. It matches this subtree property pattern. If I have the property of left and right, I can compute it for node. And this takes constant time to do. And so it's a subtree property. And so I can maintain, through all the things I'm doing, the height of every node.

Oh by the way, whenever I do a rotation, I'm also going to have to update my subtree properties. When I rotate this edge, A does not change, B does not change, C does not change. So that's good. But x's subtree changes. It now has y. It didn't before. So we're going to have to also update the augmentation here in y. And we're going to have to update the augmentation in x. And we're going to have to update the augmentation of all of the ancestors of x eventually.

So rotation is locally just changing a constant number of pointers. So I usually think of rotations as taking constant time. But eventually, we will have to do-- this is constant time locally. But we will need to update h ancestors in order to store all of-- keep all of our augmentations up to date. We'll worry about that later.

All right, so great. Now we have the height of all the nodes. We can compute the skew of all the nodes, cool. We have this rotation operation. And we want to maintain this height balance property. Height of left node-- left and right of every node-- is plus or minus 1, or 0.

Cool, so I said over here somewhere, whenever we-- so the only things that change the tree are when we insert or delete a new node. And the way that we implemented those so far is to add or remove a leaf. So we should still be thinking about adding or removing a leaf. The problem is, when I add a new leaf, now maybe this tree is higher than it used to be. So some node here may no longer be height balanced.

But because height is a subtree property, the only nodes we need to check are the ones up this ancestor path. And there's only log n of them, because now height is log n. That's what we just proved as long as we have this property. Now, we right now don't have it for, like, maybe these few nodes. But it was long n before. It's at most log n-- 2 log n plus 1 right now, because we just added a node.

So what I want to do is check all of these ancestor nodes in sequence from bottom up, and find one that's out of balance. So let's take the lowest out of balance node. I'm going to call that x. Now, because we just insert or deleted a single leaf, it's only out of balance by 1, because we only changed height-- one height went up by 1, or one height went down by 1. And before, all of our skews were plus or minus 1, or 0.

So now, it's-- the bad case is when it's plus or minus 2. If it happens to still be in this range for all the nodes, we're happy. But if it's outside this range, it's only going to be out by 1. So this means the skew is n plus 2 or minus 2. And let's say that it's 2 by symmetry. So my picture is-- I'm going to draw double right arrow to say that this subtree is 2 higher than this subtree. OK, so that's bad and we want to fix it.

The obvious thing to do is to rotate this edge. Because that'll make this-- this is too high and this is too low. So if we rotate, this should go down by 1 and this should go up by 1. And that works most of the time. So case one is the skew of y. What is y? I want y to be the right child of x. Because we have a positive skew, we know there is a right child.

Now, because this was the lowest bad node, we know that y is actually good. It's either right heavy-- or even the two subtrees have the same height-- or left heavy. The easy cases are when skew of y is either 1 or 0, which I will draw. So a double right arrow, let's say single right arrow-- so I'm just going to add some labels here to make this picture consistent-- k plus 1, k plus 2. I'm riding the heights.

So this is an example where C is taller than B. A and B are the same height. And then if you compute the heights up here, indeed this one is right leaning. This one is doubly right leaning. Because this one has height k plus 1. This one has height k minus 1. That's bad. But if we do this right rotation on x, we get exactly what we want.

So I'm just going to copy the labels on A, B, C-- we have k minus 1, k minus 1, and k-- and then recompute. That means this guy has height k, this one has height k plus 1. And now, all the nodes in this picture that I've highlighted-- A, B, and C haven't changed. They were height balanced before. They still are. But now, x and y-- x wasn't height balanced before, y was. Now, both x and y are height balanced. That's case one.

In case two, the skew of y is flat, which means that this is a k, and this is a k, and this is a k plus 1, and this is a k plus 2. But still, all the nodes are balanced-- height balanced. They're still plus or minus 1. So those are the easy cases.

Unfortunately, there is a hard case-- case three. But there's only one, and it's not that much harder. So it's when skew of y is minus 1. In this case, we need to look at the left child of y. And to be alphabetical, I'm going to rename this to z. So this one, again, is double right arrow. This one is now left arrow. And this is letter y.

And so we have A, B, C, and D potential subtrees hanging off of them. And I'm going to label the heights of these things. These are each k minus 1 or k minus 2. This one's k minus 1. And now, compute the inside. So this is going to height k for this to be left leaning. So this is k plus 1, and this is k plus 2.

But the problem is this is 2 higher than this. The height of z is 2 higher than the height of A. This case, if I do this rotation, things get worse, actually. I'll just tell you the right thing to do is-- this is the one thing you need to memorize. And let me draw the results. You can also just think of it as redrawing the tree like this. But it's easier from an analysis perspective to think about it as two rotations. Then we can just reduce.

As long as we know how rotations work, then we know that this thing works-- "works" meaning it preserves traversal order and we can maintain all the augmentations. So now, if I copy over these labels-- the height labels-- I have k minus 1. I have, for these two guys, k minus 1 or k minus 2. The biggest one is k minus 1. This is k minus 1. And so this will be k. This will be k. This will be k plus 1. And lo and behold, we have a nice, height balanced tree in all three cases for this one node.

Now, this was the lowest node. Once we update this one, it could be that we changed the height of the root. Before it was k plus 2, now it's k plus 1. Or sometimes, we keep it the same, like over in this case.

And so now, we have to check the parent. Maybe the parent is out of balance. And we just keep walking up the node, and also maintain all the augmentations as we go. Then, we'll keep track of height and subtree size if we want them, or any other augmentations. And after order h operations, we will have restored height balanced property, which means all the way through, h equals order log n. And so all of our operations now are magically order log n.