MITOCW | ocw-6.046-lec10

Good morning. It looks like 9:30 is getting earlier and earlier for everyone. Hello to all the people watching at home. And, it has two black leaves I think there should be a requirement that if you're watching the video, you can only watch it 9:30-11:00 on Sunday, or at least start watching then just so you can all feel our mornings. Today, we're going to talk about balanced search trees. Now, we've hinted at this for a while. Our goal today is to get a search tree data structure, so we can insert, delete, and search all at log n time for operations. So, we want a tree that's guaranteed to be log n in height. So, that's a balanced search tree data structure. And, we want a data structure that can maintain a dynamic set of n elements in log n time for operation. So, we'll say, using a tree of height order log n. Now, if you look very closely, we haven't actually defined what a search tree data structure is. We've defined what a binary search tree data structure is, and that's one particular kind. And that's what we will be focusing on today. In recitation on Friday, we will look at, or you will like that, balanced search trees that are not necessarily binary. Each node can have a constant number of children, not just two. So, I'm defining is generally. You actually see what a search tree is in the general case later on. Today, we will just be focusing on the binary case. So, I won't define this yet. So, there are a lot of different balanced search tree data structures. So, these are the main ones that I know of. The first one was AVL trees. This was invented in 1962. So, that was the beginning of fast data structures. The next three sort of come together and this is what you will cover in recitation this week. So, these are non binary trees. Instead of binary, we have maybe binary and tertiary, or maybe binary and tertiary, or quaternary, over a general concept degree, B. So, that's another way you can get balance. Two-three trees, which were the second trees to be invented, they were invented in 1970 by Hopcroft. The trees we will cover today are called red black trees. These are binary search trees of guaranteed logarithmic height. So then, there's some others. So, skip lists are ones that we will cover next week. It's not exactly a tree, but it's more or less a tree, and one that you will see in your problem set this week are treeps, which I won't talk too much about here. But they are in some sense easier to get because they essentially just rely on the material from last Monday. So, on Monday we saw that if we just randomly built a binary search tree, it's going to have log n height most of the time in expectation. So, treeps are a way to make that dynamic, so that instead of just having a static set of n items, you can insert and delete into those items and still effectively randomly permute them and put them in a tree. So in some sense, it's the easiest. It's also one of the most recent search tree data structures. That was invented in 1996 by a couple of geometers, Rimon Sidell and Aragen. So, those are just some search tree data structures. The only ones we will not cover in this class are AVL trees. They're not too hard. If you're interested, you should read about them because they're fun. I think they are a problem in the textbook. OK, but today, we're going to focus on red black trees, which is a fairly simple idea, red black trees. And, it's a particular way of guaranteeing this logarithmic height so that all the operations can be supported in log n time. So, they are binary search trees. And, they have a little bit of extra information in each node called the color field. And there are several properties that a tree with a color field has to satisfy in order to be called a red black tree. These are called the red black properties. And, this will take a little bit of time to write down, but it's all pretty simple. So once I write them down I will just say what they really mean. There's four properties. The first one's pretty simple. Every node is either red or black, hence the name of red black trees. So, the color field is just a single bit specifying red or black. And red nodes, I'm going to denote by a double circle because I don't have colored chalk here, and black nodes will be a single circle. And you probably don't have colored pens either, so it will save us some grief. Red is double circle; black is single circle. And, we sort of prefer black nodes in some sense. Red nodes are a pain, as we'll see. OK, second property is that the root and the leaves are all black. And, I'm going to pull a little trick here. Treat binary search trees a little bit differently than we have in the past. Normally, you think of the tree as a bunch of nodes. Each node could have zero or one or two children, something like this. I'm going to imagine appending every place where a node does not have a child. I'm going to put a little dot here, an external node, which I call a leaf. So, normally leaves would have been these items. I'm just going to add to every absent child pointer a leaf. And, these will be my leaves. These are really the nil pointers from each of these nodes. So now, every internal node has exactly two children, and every leaf has zero children. OK, so these are those I'm referring to. These are black, and this guy is black according to rule two. Now the properties get a little bit more interesting. The parent of every red node is black. So, whenever I have a red node, its parent has to be black, a single circle. OK, so in other words, if you look at a path in the tree you can never have two red nodes consecutive. You can have, at most, red, black, red, black. You can have several black nodes consecutive, but never two red nodes. OK, and then one more rule. It says a little bit more about such paths. So, if we take a simple path, meaning it doesn't repeat any vertices from a node, x, to a descended leaf of x, all such paths to all descendant leaves have the same number of black nodes on them. So, let me draw a picture. We have some tree. We have some node, x, in the tree. And, I'm looking at all the paths from x down to some descendant leaf down here at the bottom of the tree. All of these paths should have the same number of black nodes. So, here I'll draw that each one has four black nodes, the leaf, and three above it. We know that from property three, at most, half of the nodes are red because whenever I have a red node, the parent must be black. But I want all of these paths to have exactly the same number of black nodes. One subtlety here is that the black height, I didn't really leave room. So I'll write it over here. This should be the same for all paths, but in particular, the count I'm interested in does not include x itself. OK, so if x is black, I'm only calling the black height. So, the black height of x is this count four. And even if x is black, the black height is four. So, these are just some minor details to get all of the algorithms a bit clean. So, let's look at an example of a red black tree. So, yeah, I'll show you an example. Then I'll say why we care about these properties. OK, so this tree has several properties. The first thing is that it's a binary search tree. OK, and so you can check an [in order traversal?]. It should give these numbers in sorted order: three, seven, eight, ten, 11, 18, 22, 26. So, it's a valid binary search tree. We've appended these leaves with no keys in them. They are just hanging around. Those are the nil pointers. So, each of these, you can call them nil. They are all just marked there, wherever there is absent

child. And then, I've double circled some of the nodes to color them red. OK, if I didn't, the black heights wouldn't match up. So, I have to be a little bit careful. From every node, we'd like to measure the number of black nodes from that node down to any descendent leaf. So, for example, the nil pointers, their black height is zero. Good. That's always the answer. So, these guys always have black height zero. I'll just represent that here. Black height equals zero. OK, what's the black height of three? Zero? Not quite, because these nodes are black. So the black height is one. You're right that we don't count three even though it's black. It's not included in the count. But the leaves count. And there's only two paths here, and they each have the same number of black nodes as they should. Over here, let's say eight also has black height one even though it's red. OK: same with 11, same with 26. Each of them only has two paths. Each path has one black node on it. Ten: what's the black height? It's still one, good, because we don't count ten. There's now four paths to leaves. Each of them contains exactly one black node, plus the root, which we don't count. 22: same thing, hopefully. This is getting a little more interesting. There's one path here which has one black node. There are other paths here, which are longer. But they still only have one black node. So, if we just sort of ignore the red nodes, all these paths have the same length. OK: 18 should be bigger hopefully, black height of two because each of these paths now has one black node here, one black node in leaves, or one black note here, and one black node in the leaves. And finally, the root should have a black height of two. It's easier to see over here, I guess. Each of these paths has two black nodes. Same over here. OK, so hopefully these properties make sense. We didn't check all of them. Every red node has a black parent. If you look at all of these paths, we sort of alternate, red, black at most. Or we have just a bunch of blacks. But we never repeat two reds in a row. The root and the leaves are black that we used pretty much by definition. Every node is red or black. OK, that's easy. This is a particular set of properties. It may seem a bit arbitrary at this point. They will make a lot more sense as we see what consequences they have. But there are a couple of goals that we are trying to achieve here. One is that these properties should force the tree to have logarithmic height, order log n height. And, they do, although that's probably not obvious at this point. It follows mainly from all the properties. Three and four are the main ones. But you pretty much need all of them. The other desire we have from these properties is that they are somehow easy to maintain. OK, I can create a tree in the beginning that has this property. For example, I could make, I have to be a little bit careful, but certainly if I take a perfectly balanced binary tree and make all of the nodes black, it will satisfy those properties. OK, this is a red black tree. OK, so it's not too hard to make any these properties hold just from the beginning. The tricky part is to maintain them. When I insert a node into this tree, and delete a node for this tree, I want to make it not too hard. In log n time, I've got to be able to restore all these properties. OK, that will be the hardest part. The first thing we will do is prove that these properties imply that the tree has to have height order log n. Therefore, all searches and queries on a data structure will run fast. The hard part will be to make sure these properties stay true if they initially held true when we make changes to the tree. So, let's look at the height of a red black tree. And from this we will start to see where these properties come from, why we chose these properties. So, the claim is that the height of a red black tree with n keys, so, I'm not saying nodes here because I really only want to count the internal nodes, not these extra leaves that we've added, has height, at most, two times log of n plus one, so order log n. But, we have a pretty precise bound of a factor of two. There is a proof of this in the textbook by induction, and you should read that. What I'm going to give us more of a proof sketch. But you should read the proof by induction because all the practice you can get with proof by induction is good. The proof sketch on the other hand gives a lot more intuition with what's going on with red black trees and connects up with recitation on Friday. So, let me tell you that instead. I'm going to leave that board blank and go over here. So, the first thing I'm going to do, I'm going to manipulate this tree until it looks like something that I know. The main change I'm going to make is to merge each red node into its parent. And we know that the parent of a red node must be black. So, merge each red node into its black parent. So, let's look at that here. So, I'm going to take this red node, merge it into its parent, take this red node, merge it into its path, and so on. There's one up there which I can't reach. But I'm going to redraw this picture now. So, seven, so the top node now becomes, in some sense, seven and 18. They got merged together, but no one else joined them. Then, on the left, we have three. OK, nothing joined that, and there's some leaves as usual. Now, if you look at, maybe, I'm going to have to draw this. Uh-oh. I heard that sound before. So, I'm merging these nodes together, and I'm merging all of these nodes together because each of these red nodes merges into that black node. And, I'm merging these two nodes together. So, I'm putting this red node into that black node. So, now you can see from the root, which is now 7/18. There are three children hanging off. So, in that picture, I'd like to draw that fact assuming I can get this board back down. Good. So, between seven and 18, I have this conglomerate node, eight, ten, 11. And, there are four leaves hanging off of that node. And, off to the right, after 18, I have a conglomerate node, 22/26, and there are three leaves hanging off of there. OK, kind of a weird tree because we dealt mainly with binary trees so far, but this is a foreshadowing of what will come on Friday. This is something called a two-three-four tree. Any guesses why it's called a two-three-four tree? Every node can have two, three, or four kids, yeah, except the leaves. They have zero. There is another nice property of two-three-four trees maybe hinted at. So, there's really no control over whether you have two children or three children or four children. But, there is another nice property. Yeah? All of the leaves have the same depth, exactly. All of these guys have the same depth in the tree. Why is that? Because of property four. On Friday, you will see just how to maintain that property. But out of this transformation, we get that all the leaves have the same depth: because their depth, now, or let's say their height in the tree is their black height. And, the depth of these leaves will be the black height of the root. We are you raising all the red nodes, and we said if we look at a path, and we ignore all the red nodes, then the number of black nodes along a path is the same. Now we are basically just leaving all the black nodes. And so, along all these paths we'll have the same number of black nodes. And therefore, every leaf will have the same depth. Let me write down some of these properties. So, every internal node has between two and four children. And every leaf has the same depth, namely, the black height of the root. This is by property four. OK, so this is telling us a lot. So, essentially what this transformation is doing is ignoring the red nodes. Then,

leaf. OK, so this is telling us that... So, essentially what this transformation is doing is ignoring the red nodes. Then, if you just focus on the black nodes, height equals black height. And then, black height is telling us that all the root to leaf paths have the same length. Therefore, all these nodes are at the same level. Having leaves at the same level as a good thing because it means that your tree is pretty much balanced. If you have a tree where all the nodes are branching, so, they'll have at least two children, and all the leaves are at the same level, that's pretty balanced. OK, we will prove some form of that now. I'm going to call the height of this tree h prime. The height of the original tree is h. That's what we want to bound here. So, the first thing is to bound h prime. And then we want to relate h and h prime. OK, so the first question is how many leaves are there in this tree? And, it doesn't really matter which tree I'm looking at because I didn't really do anything to the leaves. All the leaves are black. So the leaves didn't change. How many leaves are there in this tree, and then therefore, industry? Sorry? Nine. Indeed, there are nine, but I meant in general, sorry. In this example there are nine. How many keys are there? Eight. So, in general, how do you write nine as a function of eight for large values of nine or eight? Sorry? Plus one, good, correct answer, by guessing. n plus one. OK, why is it n plus one? Let's look at the binary tree case where we sort of understand what's going on? Well, wherever you have a key, there are two branches. And, that's not a very good argument. OK, we have what is here called a branching binary tree. Every internal node has exactly two children. And, we are counting the number of leaves that you get from that process in terms of the number of internal nodes. The number of leaves in a tree, or a branching tree, as always one plus the number of internal nodes. You should know that. You can prove it by induction. OK, so the number of leaves is n plus one. It doesn't hold if you have a single child. It holds if every internal node has a branching factor of two. OK, this is a neither tree. And now, we want to pull out some relation between the number of leaves and the height of the tree. So, what's a good relation to use here? We know exactly how many leaves there are. That will somehow connect us to n. What we care about is the height. And let's look at the height of this tree. So, if I have a two-three-four tree of height h prime, how many leaves could it have? What's the minimum and maximum number of leaves it could have? 2^h to 4^h, or h prime. So, we also know in the two-three-four tree, the number of leaves has to be between four to the h prime, because at most I could branch four ways in each node. And, it's at least two to the h prime because I know that every node branches at least two ways. That's key. So, I only care about one of these, I think this one. So, I get that two to the h prime is, at most, n plus one. So the number of leaves is n plus one. We know that exactly. So, we rewrite, we take logs of both sides. It says h one is at most log of n plus one. So, we have a nice, balanced tree. This should be intuitive. If I had every node branching two ways, and all the leaves at the same level, that's a perfect tree. It should be exactly log base two of n plus one, and turns out not quite n. That should be the height of the tree. Here, I might have even more branching, which is making things even shallower in some sense. So, I get more leaves out of the same height. But that's only better for me. That will only decrease the height in terms of the number of leaves. n plus one here is the number of leaves. So: cool. That's an easy upper bound on the height of the tree. Now, what we really care about is the height of this tree. So, we want to relate h and h prime. Any suggestions on how we might do that? How do we know that the height of this reduced tree is not too much smaller than this one. We know that this one is, at most, log n. We want this to be, at most, two log n plus one. We know the answer. We've said the theorem. Sorry? Right. So, property three tells us that we can only have one red node for every black one. We can, at most, alternate red and black. So, if we look at one of these paths that goes from a root to a leaf, the number of red nodes can be, at most, half the length of the path. And we take the max overall paths, that's the height of the tree. So, we know that h is, at most, two times h prime, or maybe it's easier to think of h prime is at least a half, h. Assuming I got that right, because at most a half of the nodes on any root to leaf path -- -- are red. So, at least half of them have to be black. And, all-black nodes are captured in this picture so we have this relation, and therefore, h is, at most, two times log n plus one. OK: pretty easy. But you have to remember, this tree has to be balanced, and they are not too far away from each other. OK, so in Friday's recitation, you will see how to manipulate trees with this form. There is a cool way to do it. That's two-three-four trees. Today, we're going to see how to manipulate trees in this form as red black trees. And, you'll see today's lecture, and you'll see Friday's recitation, and they won't really seem to relate at all. But they're the same, just a bit hidden. OK, so this is good news. We now know that all red black trees are balanced. So as long as we can make sure that our tree stays a red black tree, we'll be OK. We'll be OK in the sense that the height is always log n. And therefore, queries in a red black tree, so queries are things like search, find a given key, find the minimum, find the maximum, find a successor, find a predecessor. These are all queries that we know how to support in a binary search tree. And we know how to do them in order height time. And the height here is log n so we know that all of these operations take order log n in a red black tree. OK -- So, queries are easy. We are done with queries, just from balance: not a surprise. We know that balances is good. The hard part for us will be to do updates. And in this context, updates means insert and delete. In general, and a data structure, we talk about queries which ask questions about the data in the structure, and updates which modify the data in the structure. And most of the time here, we are always thinking about dynamic sets. So, you can change the dynamics set by adding or deleting an element. You can ask all sorts of questions. In priority queues, there were other updates like delete Min. Here we have find Min, but we could then delete it. Typically these are the operations we care about. And we'll talk about updates to include those of these, and queries to include all of these, or whatever happens to be relevant. In problem sets especially, you'll see all sorts of different queries that you can support. OK, so how do we support updates? Well, we have binary search tree insert, which we call tree insert. We have binary search tree delete, tree delete. They will preserve the binary search tree property, but we know they don't necessarily preserve balance. We can insert a bunch of nodes. Just keep adding new minimum elements and you will get a really long path off the end. So, presumably, they do not preserve the red black properties because we know red black implies balance. In particular, they won't satisfy property one, which I've erased, which is every node is red or black. It'll add a node, and not assign it a color. So, we've got to assign it a color. And, as soon as we do that, we'll probably violate some other property. And then we have to fix that property, and so on. So, it's a bit tricky, but you play around with it and it's not too hard. OK, so updates must modify the tree. And to preserve the red black

properties, they're going to do it in three different kinds of modifications. The first thing we will indeed do is just use the BST operation, tree insert or tree delete. That's something we know how to do. Let's just do it. We are going to have to change the colors of some of the nodes. In particular, the one that we insert better be colored somehow. And in general, if we just rip out a node, we are going to have to recolor it, recolor some nearby nodes. There is one other kind of operation we're going to do. So, recoloring just means set to red or black. The other thing you might do is rearrange the tree, change the pointers, change the links from one node to another. And, we're going to do that at the very structured way. And, this is one of the main reasons that red black trees are interesting. The kinds of changes they make are very simple, and they also don't make very many of them. So, they're called rotations. So, here's a rotation. OK, this is a way of drawing a generic part of a tree. We have two nodes, A and B. There is some subtrees hanging off, which we draw as triangles. We don't know how big they are. We know they better all have the same black height if it's a red black tree. But in general, it just looks like this. There is some parent, and there's some rest of the tree out here which we don't draw. I'll give these subtrees names, Greek names, alpha, beta, gamma. And, I'll define the operation right rotate of B. So general, if I have a node, B, I look at it and I want to do it right rotation, I look at its left child enjoy this picture called the subtrees of those two nodes. And, I create this tree. So, all I've done is turn this edge 90âˆž. What was the parent of B is now the parent of A. A is now the new parent of B. The subtrees rearrange. Before, they were both subtrees of, these two were subtrees of A. And, gamma was a subtree of B. Gamma is still a subtree of B, and alpha still is a subtree of A. But, beta switched to being a subtree of B. OK, the main thing we want to check here is that this operation preserves the binary search tree property. Remember, the binary search tree property says that all the elements in the left subtree of a node are less than or equal to the node, and all the elements in the right subtree are greater than or equal to that value. So, in particular, if we take some node, little a in alpha, little b in beta, and little c in gamma, then a is less than or equal to capital A, is less than or equal to little b, is less than or equal to capital B, is less than or equal to little c. And, this is the condition both on the left side and on the right side because Alpha is left of everything. Beta is in between A and B, and gamma is after B. And the same thing is true over here. Beta is still, it's supposed to be all the nodes that come between capital A and capital B. So, this is good. We could definitely do this operation, still have the binary search tree, and we are going to use rotations in a particularly careful way to make sure that we maintain all these properties. That's the hard part. But, rotations will be our key. This was the right rotate operation. The reverse operation is left rotate. So, this is left rotate of A. In general, of the two nodes that are involved, we list the top one. So, its right rotate of B will give you this. Left rotate of A will give you this. So, these are reversible operations, which feels good. The other thing is that they only take constant time operations because we are only changing a constant number of pointers. As long as you know the node, B, that you are interested in, you set the left pointer of B to be, if you want it to be beta, so you set left of B to be right of A, and so on, and so on. You make constant number of those changes. You update the parents as well. It's only a constant number of links that are changing, so, a constant number of assignments you need to do. So, you've probably seen rotations before. But we are going to use them in a complicated way. So, let's look at how to do insertion. We'll see it three times in some sense. First, I'll tell you the basic idea, which is pretty simple. I mentioned some of it already. Then, we'll do it on an example, feel it in our bones, and then we'll give the pseudocode so that you could go home and implement it if you wanted. OK, this is, I should say, red black insert, which in the book is called RB insert, not for root beer, but for red black. OK, so the first thing we're going to do, as I said, is binary search tree, insert that node. So, x now becomes a new leaf. We searched for x wherever it's supposed to go. We create, I shouldn't call it a leaf now. It's now at node hanging off. It's an internal node hanging off one of the original nodes. Maybe we added it right here. It now gets two new leaves hanging off of it. It has no internal children. And, we get to pick a color for it. And, we will pick the color red. OK, why red? We definitely have to pick one of two colors. We could flip a coin. That might work, but it's going to make our job even messier. So, we are adding a new node. It's not a root or a leaf presumably, so we don't really need it to be black by property two. Property three, every red node has a black parent. That might be a problem. So, the problem is if its parent is red. Then we violate property two. The parent might be red, property three, sorry. OK, the good news is that property four is still true because property four is just counting numbers of black nodes down various paths. That's really the hard property to maintain. If we just add a new red node, none of the black heights change. None of the number of black nodes along the path changes. So, this still has to hold. The only thing we can violate is property three. That's reasonable. We know we've got to violate something at the beginning. We can't just do a binary search tree insert. OK, so, let's give it a try on this tree. I should say how we are going to fix this. How do we fix property three? We are going to move the violation of three up the tree. So, we're going to start at node x, and move up towards the root. This is via recoloring. The only thing, initially, we'll do is recoloring until we get to some point where we can fix the violation using a rotation -- -- and probably also recoloring. OK, so let's see this algorithm in action. I want to copy this tree, and you are going to have to copy it, too. So, I'll just redraw it instead of modifying that diagram. So, we have this nice red black tree. And, we'll try inserting a new value of 15. 22 black. 22 is the new black. OK, that should be the same tree. So now, I'm choosing the number 15 to insert, because that will show a fairly interesting insertion. Sometimes, the insertion doesn't take very much work. We just do the rotation and we're done. I just like to look at an interesting case. So, we insert 15. 15 is bigger than seven. It's less than 18. It's bigger than ten. It's bigger than 11. So, 15 goes here. So, we add a new red node of hanging off of it, replaced one black leaf. Now we have two. OK, now, we violate property three because we added a new red child of a red node. So, now we have two consecutive red nodes in a root to leaf path. We'd like to make this black, but that would screw up the black heights because now this node would have one black node over here, and two black nodes down this path. So, that's not good. What can we do? Well, let's try to re-color. Yes. This always takes a little while to remember. So, our fix is going to be to recolor. And, the first thing that struck me, which doesn't work, is we try to recolor around here. It doesn't look so good because we've got red stuff out here, but we've got a black node over here. So we can't make this one red, and this one black. It wouldn't quite work. If we look up a

a black node over here. So we can't make this one red, and this one black. It wouldn't quite work. If we look up a little higher at the grandparent of 15 up here, we have a black node here and two red children. That's actually pretty good news because we could, instead, make that two black children and a red parent. Locally, that's going to be fine. It's not going to change any black heights because any path that went through these nodes before will still go through the same number of black nodes. Instead of going through a black node always here, it will go through a black node either here or here because paths always go down to the leaves. So, that's what we're going to do, recolor these guys. And, we will get ten, which is red. We'll get eight, which is black, 11 which is black, and these things don't change. Everything else doesn't change. We are going to leave 15 red. It's no longer in violation. 15 is great because now its parent is black. We now have a new violation up here with 18 because 18 is also red. That's the only violation we have. In general, we'll have, at most, one violation at any time until we fix it. Then we'll have zero violations. OK, so, now we have a violation between ten and 18: somehow always counterintuitive to me. I had to look at the cheat sheet again. Really? No, OK, good. I was going to say, we can't recolor anymore. Good. I'm not that bad. So, what we'd like to do is, again, look at the grandparent of ten, which is now seven, the root of the tree. It is black, but one of its children is black. The other is red. So, we can't play the same game of taking the blackness of seven, and moving it down to the two children. Never mind that the root is supposed to stay black. We'll ignore that property for now. We can't make these two black and make this one red, because then we'd get an imbalance. This was already black. So now, paths going down here will have one fewer black node than paths going out here. So, we can't just recolor seven and its children. So, instead, we've got to do a rotation. We'd better be near the end. So, what I will do is rotate this edge. I'm going to rotate eight to the right. So that's the next operation: rotate right of 18. We'll delete one more operation after this. So, we rotate right 18. So, the root stays the same: seven, three, its children. Now, the right child of seven is no longer 18. It's now ten. 18 becomes the red child of ten. OK, we have eight over here with its two children. 11 and 15: that subtree fits in between ten and 18. So, it goes here: 11 and 15. And then, there's the right subtree. Everything to the right of 18, that goes over here: 22 and 26. And hopefully I'm not changing any colors during that operation. If I did, let me know. OK, it looks good. So, I still have this violation, still in trouble between ten and 18. But, I've made this straighter. OK, that's what we want to do, it turns out, is make the connection between 18, the violator, and its grandparent, a straight connection: two rights or two lefts. Here we had to zigzag right, left. We like to make it straight. OK, it doesn't look like a much more balanced tree that this one. In fact, it looks a little worse. What we can do is now rotate these guys, or rather, rotate this edge. I'm going to rotate seven to the left, make ten the root, and that things will start to look balanced. This is a rotate left of seven. And, I'm also going to do some recoloring at the same time just to save me drawing one more picture because the root has to be black. I'm going to make 10 black immediately. I'll make seven red. That's the change. And that the rest is just a rotation. So, we have 18 over here. I think I actually have to rotate to keep some red blackness here. Eight comes between seven and ten. So it goes here. 11 goes between ten and 18, so it goes here. 22 and 26 come after 18. Now, if I'm lucky, I should satisfy all of properties that I want. OK, now, if I'm lucky, I should satisfy all the properties that I want. Every node is red or black. Every black node has a child. This is the last place we change. Red nodes have black children, and all the black heights should be well defined. For every node, the number of black nodes along any node to leaf path is the same. And you check, that was true before, and I did a little bit of trickery with the recoloring here. But it's still true. I mean, you can check that just locally around this rotation. OK, we'll do that in a little bit. For now, it's just an example. It's probably not terribly clear where these re-colorings and rotations come from necessarily, but it worked, and it at least convinces you that it's possible. And now, we'll give a general algorithm for doing it. Any questions before we go on? So, it's not exactly, I mean, just writing of the algorithm is not terribly intuitive. Red black trees of the sort of thing where you play around a bit. You say, OK, I'm going to just think about recoloring and rotations. Let's restrict myself to those operations. What could I do? Well, I'll try to recolor. If that works great, it pushes the problem up higher. And, there's only log n levels, order log n levels, so that's going to take order log n time. At some point, I'll get stuck. I can't recolor anymore. Then it turns out, a couple of rotations will do it. Always, two rotations will suffice. And you just play with it, and that turns out to work. And here's how. OK, so let's suppose we have a red black tree. And value x, we want to insert. Here's the algorithm. First, we insert it into the BST. So that we know. Then, we color the node red. And here, I'm going to use a slightly more precise notation. Color is a field of x. And now, we are going to walk our way up the tree with a while loop until we get to the root, or until we reach a black node. So, in general, x initially is going to be the element that we inserted. But, we're going to move x up the tree. If ever we find that x is a black node, we're happy because maybe its parent is red. Maybe it isn't. I don't care. Black nodes can have arbitrarily colored parents. It's red nodes that we worry about. So, if x is red, we have to keep doing this loop. Of course, I just wrote the wrong one. While the color is red, we're going to keep doing this. So, there are three cases, or six, depending on how you count. That's what makes this a little bit tricky to memorize. OK, but there are some symmetric situations. Let me draw them. What we care about, I've argued, is between x and its grandparent. So, I'm using p of x here to denote parent of x just because it's shorter. So, p of x is x's grandparent. Left of p of x is the left child. So, what I'm interested in is I look at x. And, if I don't assign any directions, x is the child of some p of x, and p of x is the child of the grandparent, p of p of x. Now, these edges aren't vertical. They are either left or right. And, I care about which one. In particular, I'm looking at whether the parent is the left child of the grandparent. So, I want to know, does it look like this? OK, and I don't know whether x is to the left or to the right of the parent. But, is parent of x the left child of p of x, or is it the right child? And these two cases are totally symmetric. But I need to assume it's one way or the other. Otherwise, I can't draw the pictures. OK, so this will be, let's call it category A. And, this is category B. And, I'm going to tell you what to do in category A. And category B is symmetric. You just flip left and right. OK, so this is A. So, within category A, there are three cases. And within category B, there is the same three cases, just reversed. So, we're going to do is look at the other child of the grandparent. This is one reason why we sort of need to know which way we are looking. If the parent of x is the left child of the grandparent, we're going to look at the other child of the grandparent, which would be the right child of the grandparent, call that node y. This is also known as the

the grandparent, which would be the right child of the grandparent, call that node y. This is also known as the uncle or the aunt of x, depending on whether y is male or female. OK, so this is uncle or aunt. Unfortunately, in English, there is no gender-free version of this as far as I know. There's parent and child, but no uncle-aunt. I'm sure we could come up with one. I'm not going to try. It's going to sound bad. OK, so why do I care about y? Because, I want to see if I can do this recoloring step. The recoloring idea was, well, the grandparents, let's say it's black. If I can push the blackness of the grandparent down into the two children, then if both of these are red, in other words, then I'd be happy. Then I'd push the problem up. This guy is now red. This guy is black. So these two are all right. This one may violate the great grandparent. But we will just keep going up, and that will be fine. Today, if we're lucky, y is red. Then we can just do recoloring. So, if the color of y is red, then we will recolor. And, I'm going to defer this to a picture called case one. OK, let me first tell you how the cases breakup, and then we will see how they work. So, if we're not in case one, so this L should be aligned with that, then, then we are either in case two or three. So, here's the dichotomy. It turns out we've actually seen all of the cases, maybe not A versus B, but we've seen the case of the very beginning where we just recolor. That's case one. The next thing we saw is, well, it's kind of annoying that the grandparent and ten, so seven and ten were not straight. They were zigzagged. So, case two is when they are zigzagged. It turns out if x is the right child of its parent, and the parent is the left child of the grandparent, that's a we've assumed so far, that is case two. OK, the other case is that x is the left child of its parent. So, then we have a left chain, x, parent of x, grandparent of x. That is case three. OK, I did not write else here because what case two does is it reduces to case three. So, in case two, we are going to do the stuff that's here. And then, we're going to do the stuff here. For case three, we just do the stuff here. Or in case one, we just do the stuff here. And then, that finishes the three cases on the A side, then back to this if. We say else, this is case B, which is the same as A, but reversing the notions of left and right, OK, in the natural way. Every time we write left of something, we instead write right of something, and vice versa. So, this is really just flipping everything over. We'll just focus on category A. And, let's see what we do in each of the three cases. We've seen it in an example. But let's do it generically. Let's do it here. Sorry, there's one more line to the algorithm, I should say. It's not aligned with here. We color the root. There's a chance when you do all of this that the root becomes red. We always want the root to be black. If it's red, we set it to black at the very end of the algorithm. This does not change the black height property. Everything will still be fine because every path either goes to the root or it doesn't, every x to leaf path. So, changing the root from red to black is no problem. It will increase the black heights of everyone, but all the paths will still have the same value. It will be one larger. So, let's look at the three cases. And, I'm going to use some notation. Remember, we had triangles in order to denote arbitrary subtrees when we define a rotation. I'm going to use triangle with a dot on top to say that this subtree has a black root. So, when I fill something white, it means black because I'm on a black board. Sorry. OK, and I also have the property that each of these triangles have the same black height. So, this will let me make sure that the black height property, property four, is being observed. So, let me just show you case one. We always want to make sure property four is preserved because it's really hard to get that back. It's essentially the balance of the tree. So, let's suppose we have some node, C, left child, A, right child, B, and some subtrees hanging off of those guys. And, all of those subtrees have the same black height. So, in other words, these things are all at the same level. OK, this is not quite what I wanted, sorry. So, I'm considering, this is node x. x is red, and its parent is red. Therefore, we need to fix something. We look at the node, y, which is over here. And, I'll call it, the key is D. The node is called y. OK, it has subtrees hanging off as well, all with the same black height. So, that will be true. If all of these nodes are red, then all of these nodes have the same black height. And therefore, all of the child subtrees, which have black roots, all had to have the same black height as well. OK, so we're looking at a big chunk of red children subtree of a black node, looking at all the stuff that happens to be red. In case one, why is red so it participates? So, a way to think of this as if we converted into the two-three-four-tree, or tried to, we would merge all of this stuff into one node. That's essentially what we're doing here. This is not a two-three-four tree, though. We now have five children, which is bad. This is why we want to fix it. So, we're going to recolor in case one. And, we're going to take C. Instead of making C black, and A and D red, we are going to make A and D black, and C red. So, C is red. A is black. D is black. And, the subtrees are the same. B is the same. It's still red. OK, now we need to check that we preserve property four, that all of the paths have the same number of black nodes. That follows because we know we didn't touch these subtrees. They all have the same black height. And, if you look at any path, like, all the paths from A are going to have that black height. All the paths from C are going to have that black height plus one because there's a black node in all the left paths, and there is a black node in all the right paths. So, all the black links are the same. So, this preserves property four. And, it fixes property three locally because B used to violate A. Now B does not violate anything. C, now, might be violated. So, what we're going to do is set x, our new value of x, will be C. So, it used to be B. We move it up a couple levels. Or, in the original tree, yeah, we also move it up a couple levels. So, we're making progress up the tree. And then we continue this loop. That's case one: recolor, go up. C may violate its parent in which case we have to recurse. So, we are recursing, in some sense, or continuing on C. So now, let's look at case two. So, I'm still, in some sense, defining this algorithm by picture. This is some nice, graphical, programming language. So, let's draw case two. Yeah, I forgot to mention something about case one. So, I drew some things here. What do I actually know is true? So, let's look at the algorithm in which I've now reversed. But, we are assuming that we are in category A. In other words, the parent is the left child of the grandparent. So, A is the left child of C. That much I knew. Therefore, y is the right child. D is the right child of C. I didn't actually know whether B was the right child or the left child. It didn't matter. In case one, it doesn't matter. OK, so I should've said, the children of A may be reversed. But it just said the same picture. OK, I thought of this because in case two, we care. So, case one: we didn't really care. In case two, we say, well, case two is up there, is x the right child of the parent, or the left child? If it's the right child, we are in case two. So now, I can really know that x here, which is B, is the right child of A. Before, I didn't know and I didn't care. Now, I'm assuming that it's this way. OK, y is still over here. And now, now we know that y is black. So, y over here is a black node. So now, if I did the contraction trick, all of these nodes, A, B, and C, would conglomerate into one.

is a black node. So now, if I did the contraction then, all of these nodes, A, B, and C, would conglomerate into one. I only have four children. That actually looks pretty good. y would not be involved because it's black. So, in this case, we are going to do a left rotation on A. So, we take the edge, we turn at 90âˆž. What we get is A on the left, B on the right still. It should preserve the in order traversal, C up top still. We have the y subtree hanging off, as before. We have one of the other three subtrees hanging off B, and the other two now hang off A. So, this is just a generic rotation picture applied to this edge. OK, what that does, is before we had a zigzag between x and its grandparent. Now, we have a zigzig. We have a straight path between x. So, x is still down here. I'm not changing x in this case because after I do case two, I immediately do case three. So, this is what case three will look like. And now, I continue on to case three. So, finally, here's case three. And, this will finally complete the insertion algorithm. We have a black node, C. We have a red left child from C. We have a red, left, grandchild which is x. And then, we have these black subtrees all of the same black height hanging off, OK, which is exactly what we had at the end of case two. So, that definitely connects over. And remember, this is the only case left in category A. Category A, we assumed that B was the parent of x, was the left child of the grandparent, B or C. So, we know that. We already did the case one, y over here as red. That was case one. So, we are assuming y is black. Now, we look at whether x was the left child or the right child. If it was the right child, we made it into the left child. x actually did change here. Before, x was B. Now, x is A. OK, and then case three, finally, is when x is the left child of the parent who is the left child of the grandparent. This is the last case we have to worry about. And, what we do is another rotation just like the last rotation we did in the example. That was case three. So, we're going to do a right rotate in this case of C. And, we are going to recolor. OK, so, what do we get? Well, B now becomes the root. And, I'm going to make it black. OK, remember, this is the root of the subtree. There is other stuff hanging off here. I really should have drawn extra parents in all of these pictures. There was somewhere in the middle of the tree. I don't know where. It could be a rightward branch; it could be a leftward branch. We don't know. C becomes the child of B, and I'm going to make it a red child. A becomes a child of B, as it was before, keep it red. And, everything else just hangs off. So, there were four subtrees all at the same black height. And, in particular, this last one had y, but we don't particularly care about y anymore. Now, we are in really good shape because we should have no more violations. Before, we had a violation between x and its parent, A and B. Well, A and B still have a parent child relation. But B is now black. And, B is black, so we don't care what its parent looks like. It could be red or black. Both are fine. We are no longer violating property three. We should be done in this case. Property three is now true. If you want, you can say, well, x becomes this node. And then, the loop says, oh, x is no longer red. Therefore, I'm done. We also need to check that property four is preserved during this process. Again, it's not hard because of the two-three-four tree transformation. If I contract all the red things into their parents, everything else has a constant, I mean, every path in that tree has the same length because they have the same black length. And over here, that will still be true. It's a little bit trickier here, because we are recoloring at the same time. But, if you look at a path that comes through this tree, it used to go through a black node, C, and then maybe some red stuff; I don't care. And then, it went through these trees, which all have the same black height. So they were all the same. Now, you comment, and you go through a black node called B. And then, you go through some red nodes. It doesn't really matter. But all the trees that you go through down here have the same black height. So, every path through this tree will have the same black length, OK, if it starts from the same node. So, we preserve property four. We fix property three. That is the insertion algorithm. It's pretty long. This is something you'll probably just have to memorize. If you try a few examples, it's not so hard. We can see that all the things we did in this example were the three cases. The first step, which unfortunately I had to erase for space, all we did was recolor. We recolored ten, and eight, and 11. That was a case one. Ten was the grandparent of 15. Then, we looked at ten. Ten was the violator. It was a zigzag case relative to its grandparent. So, we did a right rotation to fix that, took this edge, and turned it so that ten became next to seven. That's the picture on the top. Then, 18, which is the new violator, with its grandparent, is a zigzig. They are both going in the same direction. And, now, we do one more rotation to fix that. That's really the only thing you have to remember. Recolor your grandparent if you can. Otherwise, make it zigzig. And then, do one last rotation. And recolor. And that will work. I mean, if you remember that, you will figure out the rest on any particular example. We rotate ten over. That better be black, because in this case it's becoming the root. But, we will make it black no matter what happens because there has to be one black node there. If we didn't recolor at the same time, we would violate property four. Why don't I draw that just for, OK, because I have a couple minutes. So, if we just did the rotation here, so let's say, not the following, we take B. B is red. This will give some intuition as to why the algorithm is this way, and not some other way. And, C is black. That's what we would have gotten if we just rotated this tree, rotated B, or rotated C to the right. So, these subtrees hang off in the same way. Subtrees look great because they all have the same black height. But, you see, there's a problem. If we look at all the paths starting from B and going down to a leaf, on the left, the number of black nodes is whatever the black height is over here. Label that: black height, whereas all the paths on the right will be that black height plus one because C is black. So now, we've violated property four. So, we don't do this in case three. After we do the rotation, we also do a recoloring. So, we get this. In other words, we are putting the black node at the top because then every path has to go through that node, whereas over here, some of the nodes went through the C. Some of them went through A. So, this is bad. Also, we would have violated property three. But, the really bad thing is that we are violating property four over here. OK, let me sum up a little bit. So, we've seen, if we insert into a red black tree, we can keep it a red black tree. So, RB insert adds x to the set to the dynamic set that we are trying to maintain, and preserves red blackness. So, it keeps the tree a red black tree, which is good because we know then it keeps logarithmic height. Therefore, all queries in red black trees will keep taking logarithmic time. How long does red black insert take? We know we are aiming for log n time preparation. We are not going to prove that formally, but it should be pretty intuitive. So, cases two and three, sorry, pointing at the wrong place, cases two and three are terminal. When we do case three, we are done. When we do case two, we are about to do case three, and then we are done. OK, so the only thing we really have to count is case one because each of these operations, they are recoloring, rotation, they all take constant time. So,

count is case one because each of these operations, they are recoloring, rotation, they all take constant time. So, it's a matter of, how many are there? Case one does some recoloring, doesn't change the tree at all, and moves x up by two levels. We know that the height of the tree is, at most, two log n plus one. So, the number of case ones is, at most, log n plus one. OK, so the number of case ones is, at most, log n. So, those take log n time. And then, the number of case twos and threes is, at most, one for one of these columns. Well, together, twos and threes is, at most, two. OK, so, log n time, cool. The other thing that is interesting about red black insertion is that it only makes order one rotations. So, most of the changes are recolorings. Case one just does recoloring, no rotations. Case two maybe does one rotation. Case three does one rotation if you happen to be in those cases. So, the number of rotations is, at most, two. It's either one or two in an insertion. It's kind of nice because rotating a tree is a bit more annoying than recoloring a tree. Why? Because if you had, say, a data structure, you have a search tree, presumably, people are using the search tree for something. They are, like, making queries. For example, the search tree represents all the documents matching the word computer in Google. You've got the Google T-shirt on here, so let's use a Google reference. You have the search tree. It stores all the things containing the word Google. You'd like to search may be for the ones that were modified after a certain date, or whatever it is you want to do. So, you're doing some queries on this tree. And, people are pummeling Google like crazy with queries. They get a zillion a second. Don't quote me on that. The number may not be accurate. It's a zillion. But, people are making searches all the time. If you recolor the tree, people can still make searches. It's just a little bit you are flipping. I don't care in a search whether a node is red or black because I know it will have logarithmic height. So, you can come along and make your occasional updates as your crawler surfs the Web and finds changes. And, recoloring is great. Rotation is a bit expensive because you have to lock those nodes, make sure no one touches them for the duration that you rotate them, and then unlock them. So, it's nice that the number of rotations is small, really small, just two, whereas the time has to be log n because we are inserting into a sorted list essentially. So, there is an n log n lower bound if we do n insertions. OK, deletion and I'm not going to cover here. You should read it in the book. It's a little bit more complicated, but the same ideas. It gets the same bounds: log n time order one rotations. So, check it out. That's red black trees. Now, you can maintain data in log n time preparation: cool. We'll now see three ways to do it.