PROFESSOR: Are people understanding AVLs? That's good, because if everyone raised their hands, we'd be done and out of here. So we're not covering new material in this recitation. We're talking about AVLs again, except this time you'll tell me how we'll do them, and we'll look at the code and see how the theory matches the code. And I need one of these. Thank you.

So let's start simple. What's a BST?

AUDIENCE: Binary search tree.

PROFESSOR: OK, binary search tree. It's binary because every node has at most two children. Why is it a search tree?

AUDIENCE: It's easy to search.

PROFESSOR: OK. Why is it easy to search?

AUDIENCE: Because you start [INAUDIBLE], is my number greater than this key or is it less than, and then you go from there.

PROFESSOR: OK. So if I would have to state this as a formal rep invariant thing, what would I say it is so that I can do that operation?

AUDIENCE: Node at right is greater than node at key, which is greater than node at left.

PROFESSOR: OK, excellent. So it turns out I can use this argument recursively to say that for a given node, everything that is to the right of that node is greater, and everything that is to the left is smaller. And this allows us to do search quickly because if you're looking for a key, say you have numbers in your tree and you're looking for 5. If you arrive at a node whose value is 4, you go right. If you arrive at a node whose value is 7, you go left. OK So what do we know how to do on BSTs? What are the operations that we know?

AUDIENCE: Minimum, maximum.

PROFESSOR: Min, max.

AUDIENCE: Left, right, parent.

PROFESSOR: So those are for the node. I want the queries and the updates for the BST type.

AUDIENCE: Insert and delete.

PROFESSOR: OK. Insert, Delete.

AUDIENCE: Next larger.

PROFESSOR: OK. And then in here, the S in BST. Someone give me the S so we can move on.

AUDIENCE: Find.

PROFESSOR: Find, search, depending on what code you're reading. What's the running time for these guys?

AUDIENCE: Order h.

PROFESSOR: Excellent. So everything has the same running time. Nice and easy to remember. Order h. What's h?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. How do we define this height? What's the height for this tree?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Good. What's the height for this tree? What's the height for this tree?

AUDIENCE: Two.

PROFESSOR: So now for a more general case, where this is the height, and the height of my left subtree is hl. The height of my right subtree is hr. What's h?

AUDIENCE: It's the maximum of hr times hl plus 1.

PROFESSOR: Nice save. I heard a "plus 1" somewhere there. Very good. This is r. So if we look at the first part of the code, lines one through eight, lines seven and eight implement the definition that we talked about here. So in our Python implementation, each node knows the height of the tree that he's the root of. And since we're storing that, we need to update it every once in awhile when we make changes to the tree, like when we insert nodes. And the way we do that is update height, which uses the formula that we came up with here. Now, there is a hack on lines two, three, four, five. Can anyone tell me what the hack is?

AUDIENCE: The negative 1.

PROFESSOR: OK. How does that work? Why do I need it?

AUDIENCE: That's so if you're at the root node, you can still calculate the height.

PROFESSOR: Depends on how the root node looks like.

AUDIENCE: If it has no children.

PROFESSOR: That's a leaf.

AUDIENCE: Yes.

PROFESSOR: OK. So if I'm in this case or in this case, what's hl, what's hr? In this case, I have this node of height zero, so I can make a small mistake here and it'll save me, but here, I have no children, so hl and hr have to be set in such a way that this formula evaluates to 0. If I set them to minus 1, I'll have minus 1, minus 1. The maximum is minus 1 plus 1 equals 0.

AUDIENCE: It's just to check the nodes for AVLs.

PROFESSOR: We use that to update the height. For AVLs, we need to know the height of a node instantaneously. We can't afford to go down the tree and compute the height every time we need it, so every node gets to store its height. There's a small field in each node that has the height.

So we need to update that every once in awhile when we do insertions and deletions. This is how we update it, and in order to update it for this case, where we're at a leaf, we have to say that the height of a non-existing tree is minus 1. Of course, in theory and in real life, there are no non-existing trees, so this is a clever hack to reduce code size.

So we said binary search trees would look something like this. Is this guy an AVL? Does anyone think it's an a AVL? Does anyone think it's not an AVL? Can one of you tell me why?

AUDIENCE: The tree with 5 as its root is not balanced.

PROFESSOR: So this guy here is not balanced. Why is it not balanced?

AUDIENCE: Because it has two on its right and zero on its left, so the difference is greater than 1.

PROFESSOR: You're saying that there's something here that's two, and something here that's-- OK, so height. So then it's almost like that. It's 1 here and minus 1 here. So the reason I asked you to clarify is that first you said two and zero, and that's the node count, and AVL doesn't care about node count. AVL cares about height.

So for example, if I have something like this, this is a happy AVL. Three nodes, here one node here. The difference in node count is greater than 1, but the difference in height is 1. Therefore, this is a good AVL. So what's a rep invariant for an AVL?

AUDIENCE: The height of the left subtree for every node is within 1 of the height of the right subtree.

PROFESSOR: Excellent. AVL. The core property is that for every node, the left height and the right height differ by at most 1. What else? If we want to be completely rigorous, what else do we have to say? It's a BST. So an AVL is a special kind of BST, and that's why when we write the AVL code, we inherit from the BST and we use some of its methods heavily. Why do we like AVLs? What's so good about them?

AUDIENCE: It's faster because it ensures the minimal height of the entire tree because most of our complexities are o of h, so it would have to be smaller.

PROFESSOR: So we care about this. In a regular binary tree, the worst case that you have is this. Ignoring this part, this is a worst case binary search tree where it's basically a list, so height is order n. In an AVL, you're saying it's better.

So the reason why we care about AVLs is that height is order of log n. Now, did people understand from lecture why that's the case? Can anyone tell me why that's the case?

AUDIENCE: Well, it's just like every level you go down, if you split off that many times, of course it's log n. If you have n nodes and they're filled up to the edge, there's going to be log n of them.

AUDIENCE: It's close to a full binary tree, right?

PROFESSOR: For some definition of "close." So here's what I use to remember, and I think I can persuade you that the height is log n using the argument that I'll show you here. Let's start building a tree this way. Let's say I have a fixed height, and I want to have as few nodes as possible. If I have a tree with a big height and very few nodes, h is going to be bad when you write it as a function of n. So those trees are unbalanced. Big height, small number of nodes.

So say we're trying to build an AVL with the smallest number of nodes and a fixed height. What if the height is 0? What does that tree look like? It's not too complicated, right? This is an AVL of height 0. It's the only possible AVL of height 0.

Now, what if we're trying to build an AVL of height 1 that has as few nodes as possible? This is what it looks like, right? Height 0, height 1. I could add another node here, but I don't want to because I want as few nodes as possible. Now, what if I tried to build an AVL of height 2 that has as few nodes as possible? Can I do this?

AUDIENCE: So at the worst case, you have h minus 1 and then h minus 2 there.

PROFESSOR: OK. You're moving ahead. You're forcing me to move faster, but you're right. And the reason is this is unbalanced. The height at the left has to be the height of the right plus or minus 1. Can't be anything else. So at the very least, I have to build a tree of height 0 here, and I know that the best tree of height 0 that I have is this guy. Cool. So for height 3, I would have to do this. And then you're saying, what would I use at the left?

AUDIENCE: You'd use whatever height's on the right minus 2. The height on the right minus 1 per side.

PROFESSOR: So on the right, I have a height 2. On the left, I have a height 1. If I want to go up to height 4, I do the same thing. So if I want to build an AVL tree with as few nodes as possible and height h, I start with the root, then at the right, I build an AVL tree of height h minus 1, and at the left, an AVL tree of height h minus 2.

And if these have the minimum number of nodes, then it turns out that the whole thing has the minimum number of nodes. I don't want to build a tree where the heights are equal because that would mean more nodes here, so this is the best I can do. This is the best way I can build a tall tree with as few nodes as possible.

Suppose I want to write the number of nodes as a function of height. You're telling me what it is. When I was here, you were giving me the answer for this. So suppose I have a height, and I want to know how many nodes I have in my tree that has a minimum number of nodes. What is it?

AUDIENCE: It's N h minus 1, and then plus N h minus 2. There might be a constant.

PROFESSOR: There might be a constant. N h minus 1 is this tree, N h minus 2 is this tree, so what's the constant?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yep. This guy.

AUDIENCE: Doesn't he have height 0, though?

AUDIENCE: We're talking number of nodes.

AUDIENCE: Oh. Number of nodes, yeah.

PROFESSOR: I'm not going to solve this now. We learned how to solve recurrences a long time ago, so I'll pretend I still remember how to do that, and I will tell you that the solution looks something like this. N of h is roughly 5, which is the magic number that we talked about in lecture-- roughly, which means there might be some things there that I forgot-- to the h.

What really matters is that it's an exponential in h. If you look at these guys, this is close to the Fibonacci number formula except there's a plus 1 here. So these guys are bigger than the Fibonacci numbers. This is definitely bigger than whatever the formula for Fibonacci numbers is, which is 5 to the h minus something over something.

What really matters is the minimum number of nodes in an AVL of height h is an exponential as a function of h. If you invert that, you get that the maximum height for a tree with N nodes is log N. This is a way to construct an AVL that shows you that the height has to be logarithmic as long as we can keep this rep invariant true.

AUDIENCE: I have a question. I didn't quite follow how you got h minus 2 and h minus 1 there.

PROFESSOR: Here, here, here?

AUDIENCE: There. The tree down. This one? This one?

AUDIENCE: This one, yeah.

PROFESSOR: OK. So suppose I'm at height 4 here. What's the best way to construct an AVL that has as few nodes as possible but height 4? If this guy has to be at height 4, then it has to have a child at height 3, at least one child. Otherwise, it's not going to be height 4.

Now, I need to build something here that has height 3. What should I build? The best AVL tree that I know that has height 3, right? I have to get to height 3 using as few nodes as possible, so I'm going to use this guy because it has as few nodes as possible and it has height 3. So this covers my right side.

Now, for my left side, what am I going to use here? Another AVL tree that has as few nodes as possible, right? So I'm definitely keeping that. But what's going to be the height of that?

AUDIENCE: So we want the difference to be less than or equal to 1?

PROFESSOR: Yeah. Otherwise, it's not an AVL.

AUDIENCE: So it would be of height 4.

PROFESSOR: So I know for sure that this guy has height 4. The difference has to be plus, minus 1.

AUDIENCE: So it would be 2.

PROFESSOR: 2, 3, 4. I know for sure I don't want it to be 4, and now I have to choose between 2 and 3. A tree of height 2 will have fewer nodes than a tree of height 3, so that's why we're doing it this way. So to build a tree of height 4, build a tree of height 3, build a tree of height 2, connect them together. That's how we get to this, and then this. Cool.

Do people remember how to do insertions and deletions in a regular binary tree? Yes? How do I insert 6.5 here in this one?

AUDIENCE: You take 6.5, you're like, oh, it's greater than 4, then you move to 5. Then you're like, oh, it's greater than 5. Go to 6. Oh, it's less than 6, and then it goes to the left of 6. I mean-- sorry. I meant left of 7.

PROFESSOR: OK. So it's bigger than--

AUDIENCE: Testing you all, guys. Oh yeah. 6.5 is bigger than 6, so then it goes to the left of 7 because it's less than 7.

PROFESSOR: I thought I had my example wrong for a second.

AUDIENCE: I'm just tired.

PROFESSOR: Me too, so don't scare me. All right. So suppose we have heights stored in the nodes here, because we want to do that for AVLs. We'll figure out why in a bit. The height of this guy used to be 0, 1, 2, 3, right? 0, 1, 2, 3. What happened when I added this node?

AUDIENCE: You added 1 to everything [INAUDIBLE].

PROFESSOR: Yeah. So I went down on my insertion path to find out where to insert a node, and then I added it. I just chose my case conveniently, but in some cases, all the heights of the nodes on that path have changed. So in an AVL, after we insert, we have to make sure that the height of every node on the path is updated. Does this make sense? So the heights will be 1, 2, 3, 4.

So the way we implemented the AVLs is that we do regular insertions and deletions, and then at the end, we say, well it used to be an AVL. Now we added or removed the node, so it might be a slightly unbalanced AVL, which means it's not an AVL. And we have the rebalance procedure.

So if you look at on the second page of the code, insert and delete are really tiny, lines 20 to 22 and 24 to 28. And they're really tiny because they call the old code and then they call Rebalance. So all the magic in an AVL is in Rebalance.

The first thing that Rebalance does, if you see, it has a while loop there, and the first thing that it does in the while loop is it updates the height, and this is why. The height might have changed after the insert, so any decision based on the old height is bad. That's why we have that there. Make sense so far?

So if you look at rebalance, don't try to understand it quite yet, but what it does is it calls Rotate Left and Rotate Right. Well, Left Rotate, Right Rotate. All the magic is in

Rebalance, and the tools that it uses are Left Rotate and Right Rotate. Now, I'm going to show you what a rotation is supposed to do in-- and the children of these nodes are a, b, c, d. Also, this node is hanging off of something here, e.

If I want to do a Right Rotate here, so if I want to rotate the tree like this, then after rotating, it's going to look like this. So notice that c got moved from B to A, but it got moved in such a way that the whole thing is still a BST. These guys show up in the same order as children, and these guys show up in the same order, so search will still work. This is how it's supposed to look like as a picture. Let's try to write the pseudocode for achieving this.

AUDIENCE: First, you identify the parent node. Step one.

PROFESSOR: Well, let's say we have it.

AUDIENCE: I mean from B. You say, A is my parent. That's now going to become my right child.

PROFESSOR: OK. So what do you want to change? By the way, this whole thing has to happen in constant time, so we're not allowed to go anywhere inside here, here, here, or here. We're allowed to change these links, but if you go inside and try to do more complicated restructuring, that's going to block the running time. We're only allowed to change the links that you see on the board.

AUDIENCE: I know in the BST, not BST node but BST, the delete, if you're deleting the root, they make up the pseudoroot.

PROFESSOR: Let's not worry about it. So Delete already did that magic for you, right? So the pseudoroot would be here, so this node has a root. We're happy.

AUDIENCE: That's not what I'm talking about. I'm saying that you could do the same with B because you're going to have to break a link with lowercase c onto B in order to flip it. So I'm saying you could have a placer, some sort of place to put it so you don't lose it.

PROFESSOR: OK. Let's see if we do that.

AUDIENCE: The new right child of B will be A, and the new left child of A will be C, and the new parent of c is A.

PROFESSOR: OK, let's move slower. I have to write them down. So the new right child of B is?

AUDIENCE: Is A. Also, you probably should do temp variables to store them.

AUDIENCE: I don't think you have to. You can just swap the one connection and swap the other.

PROFESSOR: Let's do it without worrying about temps, and then we can figure out temps later. So B's right becomes A. So it's going to be like this. Let me erase this confusing error.

AUDIENCE: And A's left child is c.

PROFESSOR: A's left child is c. OK.

AUDIENCE: C's parent is A.

PROFESSOR: C's parent is A. OK, very good. So I changed the child, then I changed the parent so that they would match up. So if B's right is A, then what should I do with--

AUDIENCE: A's parent is B.

PROFESSOR: Always do them in pairs so you don't lose track of them. And?

AUDIENCE: B's parent is e and e's subchild is B.

PROFESSOR: Sorry. B's parent is e, right?

AUDIENCE: Yeah.

PROFESSOR: So B's parent is e, and--

AUDIENCE: e's left.

PROFESSOR: Child is--

AUDIENCE: B.

PROFESSOR: Because I drew it like this, right? But if I drew it like this?

AUDIENCE: e's--

PROFESSOR: It would have to be the right child, so we have to look at both cases.

AUDIENCE: If B is greater than e, then it would be right child, and if B is less than e, it would be left child.

PROFESSOR: OK. So if B is greater than e, then e's right is B. Otherwise, e's left is B, right? Now, suppose comparisons are expensive and they don't want to do a comparison to find this out. I want to play with the tree instead. What do I do?

AUDIENCE: Can you see which one A was before that?

PROFESSOR: Yep.

AUDIENCE: If A used to be the right child, now B is the right child.

PROFESSOR: Yeah. I haven't changed the child yet, so I can do that.

AUDIENCE: That's still a comparison though, right?

PROFESSOR: But now I'm doing a pointer comparison and not a key comparison.

AUDIENCE: If e.r is A, then--

PROFESSOR: If e.right is A, then it becomes B. Otherwise. OK, this looks good. So there's the issue of temp variables and assigning these in the right order so you don't have too many temp variables and too many lines of code, and the Python code in the handout takes care of that. But this is the logic that you want. So if you have to write it from scratch, you don't have to memorize that. Remember that you want to get from here to here, and do exactly the thought process that we have here. What if I want to do a left rotation instead of a right rotation?

AUDIENCE: You just have to change the r's to l's.

PROFESSOR: Very good. Copy, paste. Swap l's and r's and we're done. Why do we need rotations?

AUDIENCE: To rebalance stuff.

PROFESSOR: To rebalance stuff. OK. Why do we rebalance stuff?

AUDIENCE: Because you don't want your code to crash when you add nodes that are sequentially larger, and then you try to find something, and then it crashes.

PROFESSOR: OK. Why would it crash?

AUDIENCE: Because the recursion depth is exceeded because you're going down the line trying to find something, and you go down too far.

PROFESSOR: OK. So pretending we don't have a recursion depth issue, then it's going to be slow. So you start from a nice AVL, and if you don't rebalance, you get to a BST that's slow, slow, slow, and then you'll fail our tests and still get a 0. Yes?

AUDIENCE: But if you just had a carrot-like tree, or if you added, for instance, 4, and then you added in 5, and then you added in 3, and then you added in 6, and then you added in 2, you'd just get a carrot. So then I feel like AVL wouldn't cover that case.

PROFESSOR: Let's do them in sequence. So what are we inserting? So you said 4.

AUDIENCE: So you start with 4, and then you insert 5, then you insert 3.

PROFESSOR: Let's see. 4, 3.

AUDIENCE: Then you insert 6, then you insert 2, then you insert 7, and then 1.

AUDIENCE: You've got to rotate that.

AUDIENCE: Well, the thing is--

PROFESSOR: Well, is this an AVL?

PROFESSOR: Well, it's balanced.

PROFESSOR: Is it?

AUDIENCE: No, it's not. 5 is unhappy.

AUDIENCE: That height of the tree is only one greater than the other height.

PROFESSOR: So the height here is 1. What's the height here? There's nothing here, so the height is minus 1.

AUDIENCE: I mean, but the other side of the tree.

PROFESSOR: So if you're looking at this guy, things look balanced, but in an AVL, this has to hold for every node. If there's one node where the heights are unbalanced, the whole thing is unbalanced. Otherwise, the construction that we did before wouldn't make sense.

AUDIENCE: It's all [INAUDIBLE] within the [INAUDIBLE].

PROFESSOR: Yep. So now that we're going to look at rebalancing, which is the magic behind AVLs, we have to make sure that after we start with something that's slightly imbalanced, when we rotate things around, we have to make sure that everything gets balanced at the end and happy, so that's a good thing to keep in mind. Good. Any other questions?

AUDIENCE: Is it obvious that you can put any number of nodes in an AVL tree?

PROFESSOR: Point any number of nodes?

AUDIENCE: Yeah. It's obvious that you can do one node, or two nodes or three nodes, but is it true that for any number of nodes, you can arrange into an AVL tree? Does that make sense?

PROFESSOR: Yeah. So if you want to have any number of nodes, you call Insert, AVL Insert, and then you'll get an AVL.

AUDIENCE: Right. But what I'm trying to say is, is it possible to construct an AVL tree out of 13 nodes?

PROFESSOR: Sure.

AUDIENCE: Or 17 nodes?

AUDIENCE: Is there a limit for where that property will not fit?

AUDIENCE: Right. Is there some set of nodes--

PROFESSOR: I like this question. I like this question. So what would be the perfect binary search tree?

AUDIENCE: An element of log N height.

PROFESSOR: An element of log N height and the complete tree, so something that looks like this. Where did we see this thing before?

AUDIENCE: In the heap.

PROFESSOR: All right. So a heap looks exactly like this, except the values inside don't fulfill the BST requirement. Otherwise, we'd know how to build perfect BSTs. It so happens that we don't. But is this an AVL?

AUDIENCE: Yes.

AUDIENCE: That example or in general?

PROFESSOR: Let's start with that example. Is this an AVL?

AUDIENCE: No. The node just to the left of the root--

PROFESSOR: This guy?

AUDIENCE: No. That one's of height two, so it has height one.

PROFESSOR: This is beautiful. This is as good as it could get. This is an optimal binary search tree. This is perfect. It better match the definition of AVL, because otherwise, it would mean AVL doesn't like perfect trees.

So the good news is that any complete tree is going to be an AVL because everything is perfectly balanced or almost perfectly balanced. The only thing that's not complete is the last level. So all the paths from the roots to the last level are either height log N or height log N minus 1. So wherever you do the height comparison, you're going to get a difference of 1, and we can keep adding nodes to this. This is how I'd build a BST of as many nodes as we want.

AUDIENCE: I have a question. For the fixed heights, why we try to minimize the nodes? Wouldn't a better way to build a more efficient tree is to try to minimize the height? So instead of adding nodes up, just fill in the tree?

PROFESSOR: Very good. So you're thinking of how to build an efficient tree, how to build a good tree. Here, I'm trying to prove the property about the maximum height of a tree. So here I'm playing devil's advocate. I'm thinking, what is the worst case for AVLs? How do I make AVLs look as bad as possible? That's why I started this way.

So usually, we're the good guy coding, but after you're done coding and you know how your algorithm runs, if you want to have peace of mind and go to sleep well and get full points afterwards, it sometimes helps to think as devil's advocate. How would I break this algorithm? What's the worst thing that I could do to it? And if you can find something that breaks it, well, you probably know how to fix it. If not, you can sleep well at night and know that your algorithm is sound.

AUDIENCE: In that example on the board, where we said the height of the tree on the left side was h minus 1, and then the other side had the h minus 2. For that h minus 2 tree, every node in that has to be h minus 2 height, every leaf. Is that what we're saying? Otherwise, it falls apart because there would be one that's--

PROFESSOR: So we're saying that the root of this tree has to be h minus 2. And the way we do this is we copy this guy over, so this guy's lopsided, too. So it's not every path to the bottom, just one path.

AUDIENCE: OK. So it could be an incomplete.

PROFESSOR: Yep. In that case, is that balanced?

PROFESSOR: So this was an AVL of height 2, right? If I stick it here, it's still going to be an AVL of height 2.

AUDIENCE: Right.

PROFESSOR: This thing will have height 2. This thing will have height 3.

AUDIENCE: OK. So it's not everything on the same level, necessarily, that needs to be of the same height. It's only the children.

PROFESSOR: Yep. So the two children, so two nodes on the same level, might have their heights differ by one, but not more than one. If they could differ by more than one, you could have a link list. Because they differ by one, then it's sort of sane. It's almost balanced. Good. I like the questions. It means you guys are thinking. I really like it. Yeah?

AUDIENCE: If they differ by one. So you have an ideal tree up there, and then you have the worst case possible. That wouldn't affect performance at all, right, if you have the worst case?

PROFESSOR: So what we know what performance is we have this guarantee. It's at worst a constant times log n. This is constant times log n. The constant happens to be 1. This is a constant times log n. The constant happens to be something bigger than 1. I think it's somewhere between 2 and 3.

AUDIENCE: That just varies by the constant.

PROFESSOR: Yeah. And since we only care about asymptotics in this class, we're happy. But we don't know how to build this. This is just something pretty that we draw on a board, but we don't have an algorithm that efficiency builds this out of a random series of insertions and deletions. I hope we don't. Otherwise, I look bad. What do I want to delete?

AUDIENCE: Instead of a binary search tree as the base of the AVL, if you wanted to have four children per node, would this change that much except you'd just have twice as many variables?

PROFESSOR: That's a good exam question. So it turns out that there's this tree called a B tree, which has 1,000 or more nodes.

AUDIENCE: B tree? Doesn't that stand for binary?

PROFESSOR: No. It's B. Just B. That's used for databases on disk. So there, you want to make the height as small as possible, because every time you get a node, you do a disk access, and that's expensive. But once you do a disk access, the nodes can be as big as you want.

You can read a few bytes off the disk at roughly the same cost as you can read a megabyte, so you're better off reading a megabyte. I might be exaggerating a bit, but for a few kilobytes, that's true. So B trees have thousands of children, and they keep them sorted. It's sort of like that, but the fan out is huge. And it turns out everything is still log n as long as the number of children you have is constant.

AUDIENCE: But for this rotation thing, though.

PROFESSOR: Oh god, no. We don't have to rotate. It gets a lot more complicated. We haven't gotten to rebalancing yet, right? Wait until you see how rebalancing looks like with two children, and then imagine trying to figure out the cases for 1,000 children. That's not sane. In B trees, they use something else to make them balance the right way. And of course, they're harder than this.

So let me try to get through rebalancing somewhat quickly. This is where I forget what I need to do. Rebalancing. So suppose we call Rebalance on this guy, and we know that the nodes here have heights k minus 1, this is k minus 1 or k, and this is k, and I want to call Rebalance here. Let's first figure out if it's an AVL tree or if there's something wrong with it. What's the height here?

AUDIENCE: k plus 1.

PROFESSOR: So no matter what the height is here, the height here has to be k plus 1. What's the height here?

AUDIENCE: k plus 2.

PROFESSOR: OK. What else can you tell me about this node? If I call Check RI on it, it's going to crash because the rep invariant for AVLs does not hold here. This child at the top of this tree will have height k minus 1. This will have height k plus 1. The difference is more than 1. Not good. So this is unhappy. How do we fix the situation?

AUDIENCE: Rotate left around k plus 1.

PROFESSOR: All right. So it better be a rotation, because we spent so much figuring out how to rotate. Let's see what happens if we rotate this way, and the way to keep track of this is I'm going to label my nodes. B, A, left child of A has height k minus 1. This guy is k minus 1. So what's the height of A now?

AUDIENCE: k. Oh wait, that's not a node. Just k minus 1 or k.

PROFESSOR: So this is a tree of height k minus 1. This is a height of tree k minus 1 or k. The height of this guy is?

AUDIENCE: k.

AUDIENCE: k plus 1.

PROFESSOR: Or k plus 1. The height of this guy is definitely k. The height of this guys is k or k plus 1. Is that an AVL? Is it happy now? So this is the easy kind of rebalancing, one rotation and you're done.

AUDIENCE: [INAUDIBLE] the difference between k minus 1 and k minus 1 over k?

PROFESSOR: It's not k minus 1 over k. It's either k minus 1 or k. Bad notation. Thank you for the question. You saved everyone from confusion. No, there's no fractional heights. Anything else? Thank you. That was good.

All right. So this is easy, right? This is the easy rotation. Let's do the hard one now. What's the height here?

AUDIENCE: k plus 1.

PROFESSOR: What's the height here?

AUDIENCE: Plus 2.

PROFESSOR: AVL, not AVL?

AUDIENCE: No.

PROFESSOR: k minus 1 on the left child, k plus 1 on the right child. The difference is more than one. Not an AVL. How do we fix this?

AUDIENCE: Make a right rotation [INAUDIBLE].

PROFESSOR: Sorry?

AUDIENCE: Make a right rotation on k first.

PROFESSOR: So make a right rotation where?

AUDIENCE: On k [INAUDIBLE] plus 1.

PROFESSOR: Let me break this up so we could see how that would work out. So these two are k minus 1 and k minus 1, and these nodes are A, B, and C. So you're saying do a right rotation here, right? Let's see what that gets us to. So A is the same, and then here I'm going to have C instead of B, and B. The right child of B is the same as it was before. The left child of B is k minus 1. Then the left child of C is another k minus 1, and this guy is k minus 1. OK, what's the height at B now?

AUDIENCE: k.

PROFESSOR: OK. What's the height here?

AUDIENCE: k plus 1.

PROFESSOR: And the height here?

AUDIENCE: k plus 2.

PROFESSOR: Is this an AVL? So we're not done, but what's the good news about this?

AUDIENCE: Now you can rotate it left around C.

PROFESSOR: Yep. Exactly. That is this, and we know how to fix this in one step. You said exactly the right thing. This is exactly what we do. First, rotate here so that we get to that, and then we're in the happy case. So intuitively, the difference between the happy case and the harder case that's going to be happy eventually is where the heavy path is, so where you have the bigger height.

In this case, k plus 2, k plus 1, k. So the heavy path is this thing here. It's a straight line. Because of that, we can rotate here and then redistribute the weights uniformly. In this case, the heavy path is this. It's a zigzag, so one rotation won't fix it. Instead, we do one rotation here so that we make the weights be distributed in a straight line, and then one rotation here, which makes everything look good. Yes?

AUDIENCE: So that's just a general pattern. And you redistribute [INAUDIBLE]?

PROFESSOR: So the first thing we do is say you're at a node and you need to rebalance. First thing, you check the heights and see if it's already balanced. If it's not balanced, suppose your heavier node is on the right. Then you go and see if the right child is heavier or the left child is heavier. So you see if the height distribution is a straight line or is exact, and then you decide whether you're going to do one rotation or two rotations.

So this is all nice and good except for one thing. The heights keep changing here, and we're storing the heights in each node. So if we're doing all this rotating without updating the heights, then we're going to have something that looks like an AVL but doesn't keep track of the heights anymore, so eventually, we're going to make bad decisions. So we need to update the heights.

Where do we update the heights? You can look at the code and cheat or you can try to think and tell me the answer, either way. So this only does rotates, right? If rotates would be able to update the heights, everything would be good. I wouldn't have to worry about this.

Pretend I hadn't deleted the original case here, which I think was A, B, C, and then some children. What heights changed here? When I did this rotation, what heights changed?

AUDIENCE: B's height changed.

AUDIENCE: B, A, and C is greater than B.

PROFESSOR: B and potentially A. If I'm not sure, then I'm going to try to update the height anyway to make sure that I have the right height. After I do this, I have to update the height on B and update the heights on A. Is this correct?

AUDIENCE: And B.

PROFESSOR: And all the way up. Let's assume that happens. Does this look right?

AUDIENCE: Don't you also have to update for C?

PROFESSOR: For c. Did I change anything under c?

AUDIENCE: I think he's talking about capital C.

PROFESSOR: Oh, capital C. Yeah, OK. So the children are the same. They were a and b before, they're a and b afterwards, so I don't have to worry about this guy. The heights here haven't changed, so the height here hasn't changed. But you're thinking of a problem, and that is good because there is a problem here.

AUDIENCE: What if there's a subtree instead of small c?

PROFESSOR: A subtree instead of small c? OK, so yeah, this is a subtree. After rotating, this subtree moves over here.

AUDIENCE: Wait. You have to update A first, and then update B.

PROFESSOR: OK. So the height starts from the bottom. The bottom height zero, the next one up, the next one up, the next one up. Keep that in mind. Nothing changed below this guy, nothing changed below this guy, nothing changed here, so I don't have to update these. But when I compute the height, update height at the beginning of the listing assumes that the height of the children is correct.

So when I compute the height of my mode, the height of the children has to be updated. So if I call Update Height of B first, I already know that this might have changed, so whatever answer I get is bogus. So this doesn't work and I have to do this.

This is rotation. That's rebalancing. One more trick to rebalancing. So we talked about rebalancing the subtree at one node. What's missing from this picture?

AUDIENCE: So you check it on all levels?

PROFESSOR: Yeah. So I made some changes here. My tree might look happy starting from here on, but if I did an insertion or a deletion, heights changed all the way up, so I have to go all the way up and do more rebalancings potentially. And that's it. These are AVLs. Any other questions? Did you guys get them? Do they make more sense now?

AUDIENCE: Yes.

PROFESSOR: OK. Good. You'll have to play with them on the next Pset.

AUDIENCE: By "play," what do you mean?

PROFESSOR: Well, you're not writing it from scratch. You have to modify existing code.

AUDIENCE: What are we going to make it do?

PROFESSOR: So you'll have to keep track of something other than heights. You'll have to keep track of a new value.

AUDIENCE: Oh, like the minimum or something?

PROFESSOR: Or something, yeah.