[SQUEAKING]

[RUSTLING]

[CLICKING]

**ERIK DEMAINE:** All right, let's get started. Welcome back to 006. Today we are doing some of the coolest data structures we will see in this class-- maybe some of the coolest data structures ever. Binary trees. You've certainly seen trees in many forms in the past, including in this class, we've talked-- used trees as a lower-bound tool for-- in the decision tree model.

But this lecture and the next lecture we're going to build one data structure that is almost superior to all data structures we have seen and can do almost anything really fast. First, recall all the data structures we've seen so far. arrays, linked lists, dynamic arrays, sorted arrays, hash tables.

And the two sets of operations we're interested in supporting, the two interfaces, one was sequences where we're maintaining items in a specified order. We want to be able to insert an item right after another item or delete an item in the middle of the list, and always be able to access the i-th item. We haven't seen any good data structures for that problem. We're really good at inserting and deleting at the beginning or the end of the sequence, but we haven't seen anything that's efficient at inserting in the middle of the list or deleting in the middle of the list.

Linked lists, you can't even get to the middle in less than linear time. Array, you can get to the middle, but if you make any changes, you have to do the shift, which is very expensive. So today-- or sorry, next lecture, for the first time we will see all of those operations efficient. I mentioned our goal where efficient means logarithmic.

So we're not quite as good as linked lists and dynamic arrays at inserting and deleting at the end. Those-- there that we achieve constant or constant amortized time. But up to this log factor, we're going to get the best of all worlds where we can solve all the things, all the operations that don't build or iterate through the entire structure, and that, of course, takes linear time. But we can do all the others in logarithmic time.

For sets, sets we're maintaining a bunch of items which have intrinsic keys and we want to search by key. So hash tables are great if you're only doing exact searches. If you want to find a key and get yes or no, is it in there? If it's in there, give me the item. That's what Python dictionaries do. And they're great at inserting and deleting, but they're really bad at find previous and find next. This is the unsuccessful case.

So if I search for a key and it's not in my structure, I would like to know more than just the answer no. I'd like to know what the previous and next items that are actually in the structure. So what are my nearest matches when I search by key? That's a natural query, and the only data structure we have that's good at it is a sorted array, because binary search gives this to us. If we search for a key bi binary search and we don't find it, the position that we end up at is right between the previous and next one.

But of course, sorted arrays are terrible for dynamic operations. We don't know how to maintain-- we can't maintain a sorted array without any gaps when we're doing insertions and deletions. In some sense, today and next class, binary trees let us represent a sorted order or a general and order of items dynamically and still allow us to do very fast things like get_at of i and find previous of the key.

So that's our goal. We're not going to quite get to this goal today. We're going to get an incomparable thing callec the height of the tree, and then on Thursday we'll be able to finish and achieve this goal. Today is just in service to that goal. So what is a binary tree? Let me draw an example and then define it more precisely. Mathematicians will call this a rooted binary tree, in case you've seen that in 042, say. Here is a picture.

So this is an example of a binary tree. It has a bunch of nodes which we're drawing in circles. It has items in the nodes which we're-- I'm writing as letters here. So this is item A, item B, item C. And it has these links between them. This is like linked lists. But in general, a node X is going to have a parent pointer, a left child, left pointer, and a right child, right pointer. And it also has an item inside of it.

So I'm going to talk about node.left is a pointer to the left-- the node down here. Node.right, node.parent, node.item gives me-- so if I look at the node A, its item is A. So let me draw for you some examples. OK, the parent of A is nothing, so we call A the root 0. There's going to be a unique node that has no parent. It's sad to have no parents, but here you go.

Then we have node B which-- whose parent is A. Node C whose parent-- its parent is A; node D, its parent is B; node E, its parent is B; and node F, its parent is D. The alphabetical order here happens to be ordered by parent. Then we have left pointer, so I'll just do a few of them. So the left pointer of A is B. The right pointer of A-- sorry, B the node-- these should all be nodes. I'm circling for nodes, I'm just reading the letter for the item to make it clear that those are different things. The right pointer for A is C; left pointer for B is D; right pointer for B is E; and so on.

So in other words, each of these lines is a bidirectional pointer. In this direction, it's the parent direction; in this direction, it's left in this case. Because it's bidirectional, we don't draw the arrows, we just draw undirected lines. This is, in general, what a binary tree looks like. A key invariant is that if you take a node and say go to its left pointer-- left child and then go to that node's parent, this should be the same as node. So that's just saying these are-- parent is always the inverse of a left or right operation. This is also true for right. OK, and that's a binary tree.

Now, the intuition of what's going on here is, you could say we're inspired by a linked list. Linked lists had a very similar structure. Maybe an item-- or there's a node, it had an item in it, and it had a next pointer and it had a previous pointer. So in some sense, what we're-- if it's doubly linked, we had a previous pointer. If it was singly linked, we only had a next pointer.

And if you think about the limits of linked lists, especially singly-linked lists, if you just have one pointer per node, you can only build the list. And so the result is this node is going to have depth linear. Depth means how many pointers do I have to follow to get here from the root of the structure? Which for linked lists was the head. If it was doubly linked, OK, I can have a head and a tail and I can put bidirections on here. But then still, the middle item has depth linear. So there's no way to get there unless in linear time.

With binary trees, because we use two types of next pointers, left and right, we can build a tree. And we know trees in general have logarithmic-- can have logarithmic height. And so it's possible in a tree to get to any node starting from the root and only log n traversals. So that's the intuition on what's going on.

Now today, we're going to talk about the height of a tree. So let me define a couple of definitions here. Subtree height of a node. So a tree decomposes into subtrees. So for example, the subtree rooted at B or the subtree of B is this portion of the tree. So it's that node and all of the descendants of this node.

So because we have parents and children, we can generalize in the familial tree sense. We can talk about ancestors of a node. So the ancestors of F are its parent, its grandparent, great-grandparents and so on. Together, all of these are called ancestors. It doesn't quite correspond to familial trees because familial trees you have two parents. Here, you only have a unique parent. Or the poor root has no parent.

We also talk about-- it's like mixed metaphors-- leaves of the tree. These are people with no children. Parents will complain about this, but many-- like many of us in this room, they have no children yet, so we are called leaves. You can tell your parent, hey, I'm just the leaf, you know? Blowing in the wind.

So this-- ah, it's so many mixed metaphors, but we always draw trees downwards like the root structure of a tree. Yet we call the ends of the roots leaves, which is upside-down. Anyway, that's trees for you. Lots of entertaining analogies. OK, but ancestors are useful. Descendants are also useful. So the descendants of B are all of its children and all of its grandchildren and all the way down, but just within the subtree. So the subtree of X consists of X and its descendants. And we think of X being the root of that subtree. So we're kind of forgetting about everything outside of the subtree when we talk about the subtree of X.

Let's talk about the depth of a node. Depth of a node is-- I guess the number of its ancestors. That's right. I usually think of it as the number of edges in the path from X up to the root. So every node has a unique path that goes upwards until it can't go up anymore.

So the depth of E is 2 because there are two edges-- 1, 2-- in the path from the root A to E. So maybe I'll read some depths, depth of these is 2. Depth of these guys is 1, depth of the root is 0. 2, 3. So those are depth. I'm going to clean this up a little bit so we can focus on the image. All right.

Height-- so depth is measuring downwards because if you imagine depth within water, this is the surface of the water, and then we measure how deep you are from the surface. Height is in the reverse direction. We're going tc measure from the leaf level up, because leaves are the bottom of the tree.

So height of a node is going to be the number of edges and the longest downward path, which is the same thing as the maximum depth of a node in X's subtree. Let's do height in red. So, how long is the longest path from F to a leaf? Well, F is leaf, so all leaves have depth 0. Sorry, height 0. Don't get it backwards.

D here, it's-- so there are two ways to go down. This doesn't go to a leaf. This one goes to leaf and its height is 0. So this height is 1. There's one edge to a leaf here. B has two leaves it can get to. We take the longest one, so that's length 2. A similarly has height 3. OK. So height we measure upward, depth we measure downward. One thing we care about is just the height of the overall tree, which is the height of the root. And I call that h because we're going to use it a lot.

And what we're going to achieve today is all of these running times, instead of being log n, they're going to be order h. Today, our goal is to get all the operations we care about in order h time. And then next lecture we're going to guarantee that as always log n and then we'll get log n time.

So we need to do a bunch of work to achieve log n. Today we'll do the work that's all the tree manipulation. And as long as your tree is nice and shallow, it doesn't have high height, because logarithmic height, everything will be log n. Of course, there are trees that are really bad. We can have a tree like this, which is basically a linked list where we only use right pointers and all the left pointers are none. So there are trees that are very high, have high height. We want to avoid these, but we won't worry about that till next lecture. Question?

**AUDIENCE:** A couple of questions. Is the height of C-- of node C is 0 or--

**ERIK DEMAINE:** What is the height of node C? Height of node C is 0. Because the length of the longest path-- the number of edges in the longest downward path is 0.

**AUDIENCE:** Oh.

**ERIK DEMAINE:** Yeah. We're counting edges, not vertices. Yeah. So the height of the tree is, of course, just the depth-- the maximum depth. I think that's right. So the height here is 3, and the maximum depth is this terribly drawn 3. So these happen to correspond in the maximum case. But We use height to always mean maximum, and so that's why we talk about the height of the tree. Depth of the tree is not defined, just depth of nodes. OK.

How do we use these trees to represent either a sequence or a set? I claim that there is a natural order in trees called the traversal order of nodes or items in the tree. So I'm going to define a particular order, say, in this example. Let's do the example first. The traversal order is going to be F, D, B, E, A, C. I feel like I'm in music class. This is my guitar or something, but it's not, I hope. If it is, it's coincidence.

So what is this order? What I'd like to do is recursively define an order where the root of the tree is somewhere in the middle, and everything in the left subtree is left earlier in the order than the root, and everything in the right subtree is later. So you can see here C comes after A, and then all the other nodes come before A.

And recursively, if I look at a node B, this node B, which appears over here, E is to the right of it, but this is all to the left of A. So E is between B and A. With B on the left, and then F and D to the left of that. And again, F comes before D because F is in the left subtree of D. So we say for every node, the nodes in X.left are before X and the nodes in X.right come after X.

And this uniquely defines an order called a traversal order. That's also called in-order traversal order. It's called in-order because it's in the traversal order, so it's very circular, but you may have seen in-order traversal, this is the same thing. There's a very simple algorithm for computing this. If I want to iterate-- let's call this-- yeah. If I want to iterate all the nodes within a subtree X, rooted by X, I just iterate on all of the notes in the left subtree, then I output X itself, then I iterate on all the nodes in the right subtree.

OK, you may have seen that algorithm before. This is just another way to codify the same thing. The result is, all the nodes within a subtree appear continuously with no interruptions, and then the parent-- parents are going to come before or after depending on whether this is a left or a right child.

OK, so now it's just a matter of connecting the dots, because we're representing an order. And for a sequence, that is going to be the sequence order. If we want to store n items X0 through X1, we're going to build some kind of tree. We're going to put X0 here, X1 here, X2 here, X3 here, X4 here, X5 in here. You can see, I'm very used to dealing with traversal orders. It takes a little while. You could also see it here. We're going to put X1 on this node, X2-- sorry. X0 here, X1 here, X2 here, and so on. That's the same order that I gave.

OK, that's four sequences. For sets, that order is just going to be disordered order, and we're going to be effectively representing the sorted order of keys, say, increasing. But before we get to that, let's talk about different operations we can do just playing around with traversal order. And then we're going to use these to build the sequence and set operations that we care about.

So a first operation I'll call subtree first. Seems appropriate that it's called first. It's the first one. So, given a node, which I'll call node, this defines a subtree, which usually we draw subtrees as triangles hanging off of the node. So here, I would write X, and then there's some subtree of all the descendants of X.

So with subtree first, I would like to say among all the nodes in this subtree, which comes first in this traversal order? So just restricting to that subtree, subtree of node. So, where is it in this tree? Yeah?

**AUDIENCE:** A node can be only part of one subtree, right?

**ERIK DEMAINE:** A node is actually part of many subtrees, good question. F, F is in its own-- in the subtree of F. F is also in the subtree of D, F is in the subtree of B like I drew, F as in the subtree of A. It's in the subtree of exactly its ancestors. But in this operation, when we-- our node only defines one subtree. It is the root of only one subtree. And that's the subtree we're talking about.

And then I want to know, among all those nodes, which includes the node itself and other things, which one comes first in this traversal order? This is like practice with traversal orders. So where should I look for this node? Yeah. The leftmost leaf, yeah. And the picture, it's here, but pictures can be deceiving. We just want to go left as much as possible. When I say go left, I mean this iteration node equals node.left. If you just look at our definition, all the nodes in the left come before X and all the nodes in the right. So it's got to be on the left subtree if there is one.

Of course, we can't do this forever. So say until we would fall off the tree, which means node is none. OK, we stop before that would happen. So this is like the directions like, oh, you keep driving until you see the store, and it's the block right before that. It's like, well, that's not very helpful.

So you keep iterating node equals node.left until node becomes none, and then you undo one step. You all know how to program that, it's not hard. So that last non-non-node, which might actually be the root, it might be node, maybe it has no left children. But in that case I claim node is the very first in its subtree traversal order because if there are no nodes in the left that come before X, then x is actually first.

So that's it. Return node. So I'm modifying node in place here. And the very last one before I hit none, that's the minimum. That's the first item in the traversal order. Similarly, you can define subtree last. OK, let's do a more interesting one. Successor node. So in this case, I want to know what is the next after none in the overall tree's traversal order?

Here, I was restricting to a single subtree. Now I'm thinking about the entire tree and the entire traversal order. And given a node, I want to know which one comes next? Call this the successor. I feel like I should make some kind of royal family joke now, but I don't know how. So every node has a unique successor. Let's do-- let's do some examples. So we can start with F-- the successor of F. If we just index into this list, the successor is D. Successor D is B. The successor B is E. It's very easy to read successors off when I have the traversal order written down, but let's think about how to do it in the tree.

Let's see. There are going to be two cases. If we look at the successor of A, it has a right child. And in this case, the right child of A is the successor, but that's not always the case. I don't have an example, but if I had another node, here let's call it G, the successor of A is actually G. Because all of these items come after A in the order. But which one comes first? The leftmost leaf. That's the problem we just solved.

So if A has a right child, what we want is the leftmost leaf, the first thing in that subtree. The right subtree. Right child subtree. So this is case 1. If node.right-- so if we have a child, then what we want is subtree first of the right child. OK, great. We can reduce to this other operation.

But what if the node doesn't have a right child? For example, it could be a leaf. Say we're taking the successor-- I mean, it doesn't have to be leaf. It could be F, which has no children. It could be D, which has one child, but no right child. So what's the successor of F? Well, it's D, which in this case is the parent, but it's not always. For example, if we do a successor of E, its parent is actually earlier in the order because E was a right child. Here, F was a left child, and so its parent was after.

The successor of D happens to be B because it was the left child if its parent. OK, so that seems like the easy case. If we're the left child of our parent, then our successor is our parent basing on this small example, but we can argue generally in a moment. What's the successor of E? Well, it's not B because that comes earlier. In fact, all the things in B subtree come earlier or equal to E.

So we have to keep going up. And then it turns out the successor of E is A because this subtree was the left child of A, because B was a left child of A. So the general strategy is walk up the tree until we go up traversal whose reverse direction would be left. So walk up the tree. When I say walk up, I mean node equals node.parent iteration Until we go up a left branch. So this would mean that node-- before we do the change node equals node.parent.left.

So we can check that. And then after we do that traversal, that parent is exactly the node we're looking for. Why is this true in general? Let me draw a more generic picture. So we're starting at some node. And let's say its parent is to the right-- so it comes later in the order for a while. Sorry. I get this backwards, we're doing successor.

It goes to the left for a while. So these are-- all these nodes will come earlier in the order, because by the definition, everything in the right subtree comes after. And at some point we have a parent that's to the right, meaning this node was the left child of this parent. And that node, by definition, will come after all of the nodes in here. And could there be anything in between node and this parent, grandparent, ancestor? Only if there was something in this subtree.

And we're in the case here where there is no right subtree of our original node. So this is where all the nodes in between node and here would be. But there aren't any, and therefore, this is the successor. So that's sort of a general argument why this works. I see a question, yeah?

**AUDIENCE:** So when you say go up a left branch, does that mean that you mark the root node subtree that is left child into the traversal order children?

**ERIK DEMAINE:** Placed into the traversal order. So the traversal order is never explicitly computed. What we're-- it's always implicit. We can't afford to maintain this as, say, an array. This is just in our heads. Maybe I will draw it with a cloud around it. We're just thinking this. It's not in the computer explicitly. And the computer always stores this. And the reason is, this is expensive, we don't-- we can't maintain an array of things and be able to insert in the middle, whereas this is cheap. I can afford to maintain this structure and do all these things.

And so the reason we're talking about these operations is they're letting us manipulate the order-- or in this case, letting us iterate through the order. So this was an algorithm for iterating through the entire order, but that takes linear time. This was getting started in the order, finding the first thing in the order, and this was given one node, finding the next one. How long did these operations take? Yeah?

**AUDIENCE:** So for the first node, [INAUDIBLE] to get to that last node--

**ERIK DEMAINE:** Right. At most, the height of the entire tree-- in fact, it's going to be the depth of that first node, but in the worst case, that's the height of the entire tree. In general, all of these operations are going to be order h. We need to think about it in each case. Except for this one, which is order n, because we're iterating through the whole thing.

In this case, we're just calling subtree first, so that takes order h time. Here, we're walking up the tree instead of down, but that's going to cost exactly the height of the node. We happen to stop early, but worst case, order h. For all this all the operations we consider today, we just want to get an order h bound, and later we will bound h.

So the point is, these are fast. If h is small, like log n, these are almost instantaneous. Whereas if I had to update the explicit traversal order, say, as an array, I would have to spend linear time every time I make a change. And yes, it would be fast to do successor if I had this stored explicitly, but maintaining it would be impossible, maintaining efficiently would be impossible. Question?

**AUDIENCE:** [INAUDIBLE]

**ERIK DEMAINE:** Yes. OK. Cool. So these were queries where I want to follow-- see what's next in the traversal sequence. Now let's talk about actually changing the traversal sequence. So these are insert and delete operations. These will correspond roughly to insert at or delete at, but they're not quite-- we're not quite in the sequence world yet.

Instead, we're going to focus on inserting or leading in the middle of a subtree. So I'm going to have two nodes. So in the traversal order, so node already exists in the tree. New is a new node that does not yet exist in the tree, hence I call it new. And what I'd like to do is insert new right after node. There's a symmetric operation which is insert before. It will be implemented almost identically. So we'll just focus on after.

So I want to insert this new node in the reverse order, which, again, is in our heads. This is all in our thought bubble. That's what we want to achieve. And we have to do it by manipulating this tree, and however we change the tree, it defines a new traversal order. So maybe let's do an example first.

Actually, I probably want this in traversal order To. Keep track of that. So, let's say the first thing we want to do is insert G before E. I want to illustrate both of the operations. Insert H after E. So insert G before E. So conceptually what we want to do is insert G here. And the way-- so we're given the node E and we're given a sort of empty node-- I mean, a node that just contains G. It doesn't have any pointers yet. And we would like to put it before E. Where should I put it? Left child.

And so that's-- this is an easy case. If I'm trying to insert before and there's no left child, stick it there. If I'm trying to insert it after and there's no right child, stick it there. Easy. So let me write down case 1. So here, we're inserting after. So if there's no right child, put new there. I'm using informal language here, putting this new node there instead of writing, for example, node.right equals new. Because that's only one operation you need to do.

One thing you would do is set node.right equals to new, but you also have to set new's parent to be node.right. So instead of worrying about those two pointer changes, because we always do bidirectional pointer changes, I'm just going to use pseudocode, and then a recitation you'll see actual Python code that does all this.

So then there's the other case. So that should be the second example. Insert h after A. Insert h after A. So we already have a node after A in the right child in this right subtree. So where do I want to put H relative to A? Well, it should be to the right of A, but it should be before C. It should be to the left of C.

So that would mean we want to put it here. In this case, it was pretty easy because this tree was small. Where do I want to put it in general? Well, wherever subtree first tells me to put it. Subtree first is going to give me the successor. These are all kind of parallel. We're in the case now where our node has a right child. And then successor tells us where the successor is. It is the first node, which is the leftmost descendant, in the right subtree of the node. A lot of pointers to follow in that sentence, but it's clear in the picture.

So in this case we had a node, and there was no right child, so we just added new to be its right child. In the other case, we had a right child. So here is node. There's this node here, node.right, which now we're supposing exists. And it defines a whole subtree. There's this one, which is the first node in the reverse order of the subtree, also known as the successor of node. So I'll call the successor of node in the current universal order.

But of course, we'd like to make new the new successor of the node. So where does it go? Here. We want to add it as a left child to the old successor. So put node as-- so take the successor. And if you look at the code for successor, where in this case, so we know it we'll just call subtree first of node.right.

And remember, subtree first went left as much as it possibly could. So what that means is this successor node is guaranteed to not have a left child. Because it was defined by going right once and then going left as much as you could. So there's no more left, which means we can make one more left. We just add new there and we're done.

Now if you look at the traversal order, it will be node, then new, then the old successor and the rest of that subtree. It's kind of cool. In all cases-- I mean, this was constant time. Here, we spent constant time after we called successor. Successor costs order h time, so this is order h.

**AUDIENCE:** It's new or node?

**ERIK DEMAINE:** Hmm?

**AUDIENCE:** New or node?

**ERIK DEMAINE:** New, new. Thank you. Put new there. Clear? OK, that was insertion. Let's do deletion. Get the spec right and the example. All of these are going to have two cases. So let me-- oh, I didn't update. So now H is after A. So it should be like this. You can check, the new traversal order of the tree is exactly that.

Next, I'm going to do a couple of deletions. Let's delete F first, and then we're going to-- wow, this is confusing. We're going to delete A. So whereas F, we're supposing we're given a pointer to F, this node, well, it's a leaf. So if I want to delete it, I just erase it. Easy. Leaves are easy to delete, there's no work to do. So what that means is I'm removing the pointer from D to F. We'll just erase that guy.

OK, now here's a trickier one. Suppose I want to delete the root of the tree. This is kind of the hardest case. But in general, it'd be somewhere in between leaf and root. So if you want to delete A, if I just erased it, then suddenly there are these pointers to nowhere and I disconnect the tree into two parts. I don't want to do that, I need to keep my tree connected.

So I'm going to play this trick, which is-- I forget if I use successor or predecessor. Predecessor. So I'm going to look at A. We already have to find successor and thereby predecessor. So I'm going to look at the predecessor of A, which is E. We can check that here, the one before A is E. This is in the left subtree. Find me the rightmost item. Keep going right until I can't. That's E.

So now these guys are adjacent in the order, and I'm about to remove A from the order. So I can momentarily cheat and swap their labels. I'm going to erase A and E here and put E after A. Why? Because it moves a down in the tree. And if I get to a leaf, I'm done. So I'm not quite done because this is not a leaf.

So again, I look at A's predecessor, it's now G. Predecessor, we hope, is always in the-- farther down in the tree. And then I swap A with G. I have preserved the traversal order, except where A falls, just by moving A earlier in the order here. And now A is a leaf and I can erase it.

So that's what we're going to follow. Now in actuality, it's a little tricky. Sometimes we need to use predecessors, sometimes we need to use successor. So the cases are, if node is a leaf, just detach it from the parent. Easy. That's sort of our base case in the recursion. Otherwise, there are two cases. If-- so if we're not a leaf, that means we have a left child or a right child or both. Both is going to be the easy case, but in general I have either there's the left child or there's a right child. In either case I'm going to be happy, so I don't need a both case.

OK. So what do I do? If I have a left child, that guarantees to me that the node's predecessor is inside that left subtree, which means it's lower in the tree. If I didn't have a left child, the predecessor would actually be higher in the tree, and I don't want to go higher. So if I have a left child, I know the predecessor is lower, and so I'm going to swap my item, the contents of my node with my predecessor's item. And then I'm going to recursively delete predecessor.

And that's the case that we looked at in this code-- in this example because we always had a left child. If we have a right child but no left child, we just do the reverse, we swap with our successor's item and then delete the successor. In either case we're going down. And so if we start at some node like the root, every time we do this operation, we're walking down, and then we're walking down, and in general we'll keep walking down, resuming where we left off, which means total amount of time we spend is proportional to the height of the tree, the worst case. Question?

**AUDIENCE:** Why do you do [INAUDIBLE] a right child to C, but it's like [INAUDIBLE] you never had a right child?

**ERIK DEMAINE:** Right. So E didn't used to have a right child. So we're changing identities of nodes when we do this, because we didn't actually move this circle. The circle stayed in place, and what we changed was the item that was stored in that circle. So whether you call this node E or A, it doesn't really matter. It is just the root node.

So we're going to play a lot of these tricks of moving the items around. So far we hadn't been doing that. We've just been creating nodes and placing them somewhere. But now in this delete operation is the first time where we're changing what's stored in the nodes. But we still can define the traversal order. The traversal order of this tree is D, B, G, E, H, C, which should be what we get here if I delete F and A.

**AUDIENCE:** But what if we created like a tree--

**ERIK DEMAINE:** And-- sorry, F.

**AUDIENCE:** [INAUDIBLE] like it really matters to you-- like it really matters to [INAUDIBLE]

**ERIK DEMAINE:** Trees will not preserve connections. That's just the name of the game. We are-- we have to allow this, otherwise we can't do anything. That's the short version. OK. In the last few minutes, let me talk about how we take these trees and implement a set or sequence. I've already alluded to this.

So for a sequence, we just make the traversal order equal to the order that we're trying to represent, the sequence order. And if we're trying to sort a set, items with keys, we're going to make the traversal order equal to ordered by increasing key. Increasing item key.

In some sense that's it, but then we need to think about how do we implement all of these operations. So maybe most enlightening is, for starters, is finding a key in a tree. So this is going to correspond to binary search. If I'm searching for a key-- let's say I'm searching for G's key. And I know-- this going to be hard in this example. Maybe I replace these all with numbers so I can think about key values.

So let's say 1, 7, 12, 17, 19, and 23. This is now in key order if you think of the traversal order. The property is that all the keys in the left subtree of the root are less than the root, and the root is less than all the keys in the right subtree, and recursively all the way down. This is something called the Binary Search Tree property, BST property. These here, we're calling them binary tree sets or set binary trees, but there are also none in the literature as binary search trees, a term you may have heard before.

So this is a special case of what we're doing where we're restoring the keys in order. And then if I want to search for a key like 13, I compare that key with the root. I see, oh, it's not equal, and it's to the left because it's less than 17. So 13 is left of here, 13 is right of 7, 13 is right of 12.

And so I know that this is where 13 would belong, but there's no right child there, and so I know-- and find that I just return nothing. If I was doing find previous, I would return this node, because I have tried to go to the right. The last time before I fell off the tree, I was trying to go to the right, and therefore, that last node I had was the previous item.

If I was trying to find next, what would I do? I would just take this node and computed successor, which we already know how to do, and that happens to be the root. So now I can do these inexact searches. When I do find previous and find next, when I fall off the tree, I find either the previous or the next. And then with predecessor or successor, I can find the other one. So that's how we can do find and find previous and find next. To do sequences, we need a little bit more work. We'll do that next time.