PROFESSOR: One of the cutest little data structures that was ever invented is called the heap. And we're going to use the heap as an example implementation of a priority queue. And we'll also use heaps to build a sorting algorithm, called heap sort, that is very, very different from either insertion sort or merge sort. And it has some nice properties that neither insertions sort nor merge sort have.

But what I want to do is get started with motivating the heap data structure, regardless of whether you're interested in sorting or not. So the notion of a priority queue, I think, makes intuitive sense to all of you. It's essentially a structure that implements a set S of elements. And each of these elements is associated with the key. And as you can imagine, a priority queue is something where you queue up for something, you want to buy something, you want to sell something. You have certain priorities assigned to you, and you want to pick the maximum priority or the min priority. You want to be able to delete it from the queue. You want to be able to insert things into this queue. You want to be able to change priorities in the queue.

So all of these operations are interesting operations that should run fast, and for some definition of fast. Obviously we are interested in the asymptotic complexity definition of fast. In that case, we'll be saying does this operation run an order n time, order log n time, et cetera.

So in general, I think for the next few lectures, you're going to see a specification of data structure in terms of the operations that the data structure should perform. And those of you who have taken six double O five, you'll see that it's basically an abstract data type that's associated with these operations. So it's a spec for the abstract data type.

In six double O five, you had really spent a lot of time on asymptotic complexity, or the efficiency of operations on the abstract data type. Here, in double O six, you'll specify this ADT, and specify the set of operations or methods in the ADT. And we'll talk about whether these are order end complexity log end complexity, and compare and contrast different ADTs.

So today's ADT is a heap. And what is the set of operations that we'd like to perform on a priority queue? So we can use that to motivate the development of the heap. And those are, insert s x. So you have a set of elements s, and you want to be able to insert element x into set s. You want to be able to do max of s, which is return the element of s with the largest key. And different from max of s is extract max of x, which not only returns the element with the largest key, but also removes it from s.

So you have a queue, and the person in the queue was serviced, or the element in the queue was serviced, and then removed from the queue. And finally you can imagine changing the priority of a particular element x in the set s. And this priority, there's an associated key as we have up there with each element. And that key is called a k. And increase key s x k would increase the value of x's key to the new value k.

And k could correspond to, it's just called increase. Most of the time, you're increasing the value in maybe a particular application. You could have suddenly a decrease key, and you would have to know what the previous value was. And is just a matter of exactly what operation you want to perform. You could call it update, or increment, whatever you like.

I'm going to spend most of the time here talking about how you maintain a rep invariant of this data structure called the heap, that allows you to do these operations in an efficient way. And we'll talk about what the efficiency is, and we'll try to analyze the efficiency of these algorithms that we put up.

So let's talk about a heap. A heap is an implementation of a priority queue. It's amazingly and array structure, except that you're visualizing this array as a nearly complete binary tree. And what does that mean exactly? Well, the best way to understand that is by looking at an example. We got 10 here, so. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

So here's my array of 10 elements. And the elements are 16, 14, 10, 8, 7. So some set of elements that are in random order, clearly not sorted, and I'm looking at the indices, and I'm looking at the elements. I'm going to visualize this as a nearly complete binary tree. Is not a full binary tree, because I only have 10 elements in it, and it would have to have 15 elements to be a complete binary tree. And we want to be able to do the general case of an arbitrary size array, and so that's why we have nearly complete here.

So what does it mean to visualize this as a tree? Well, index one is the root of the tree, and that item is the value is 16. And what I have are indices 2 and 3 are the children, and 4, 5, 6, and 7 are the children of 2 and 3. And 8, 9, and 10 are the children of 4 and 5, in this case. And so that's the picture you want to keep in your head for the rest of this lecture. Any time you see an array, and you say we're going to be looking at the heap representation of the array, the picture on the right tells you what the heap looks like.

And so that I'm not going to fill in all of these. You can, but I'll do a couple. So you have 10 here, and 8, 7, et cetera. So that's a heap structure.

So what's nice about this heap structure, is that you'll have tree representation of an array, and that lets you do a bunch of interesting things. What do you get out of this visualization? Well, the root of the tree is the first element corresponding to i equals 1. The parent of i is i over 2. The left child of i is 2i. And the right child of i is 2i plus 1. So that's essentially what this mapping corresponds to.

Now on top of that, this is just what a heap corresponds to. We're going to have particular types of heaps that we'll call max-heaps and min-heaps. And as you can imagine, max-heaps and min-heaps have additional properties on top of the basic keep structures. So this is essentially a definition of a heap. Now I'm going to define what the max-heap property is. And the max-heap property says that the key of a node is greater than or equal to the keys of its children. OK, that's it.

It's obviously recursive, in the sense that you have to have this true for every node in the tree. And when you get down to the leaves of the tree, they're not children corresponding to the leaves, So that's a trivial property. But at higher levels, you're going to have children, and you have to check that.

So if you look at this example here, maybe I should fill this whole thing out. A have eight and seven here, and six would be nine. And I have three over here, and then two, four, one. So we can look at this and check whether it has the max-heap property or not. Does it have the max-heap property? This heap? Yeah. All you have to do is look at these nodes. one, two, three indices, index four, five, six, but you don't have to look at six and seven, because they don't have any children. But you could shop with five here, and you look at the children, and there you go. To the parent is greater than or equal to either of its children, or its only child, in the case of node five. And so you have the max-heap property. So fairly straightforward property.

And you can imagine defining the min-heap property in an equivalent way. Just replace the greater than or equal to, with less than or equal to. So right off the bat, what operation is going to be trivially performed on a max-heap? This is kind of trivial question. Yep. Just finding the biggest element. Exactly right. The max operation.

Now, what about extract max? Is that trivially performed on a max-heap? No. What do I mean by that? When you say, max is trivially performed, what it means is that you can return the max, you can find the maximum element, or a maximum element, and you obviously don't modify the heap. And the heap stays the same, so it stays a max-heap. In general, when we talk about data structures, and this goes back to rep invariance, which I've mentioned already, you typically want to maintain this rep invariant. And so the rep invariant of our data structure, in this case, is a max-heap property. OK. So we want to maintain the max-heap property as we modify the heat. So if you go from one heap to another, you start at the max-heap, you want to end with the max-heap.

It makes perfect sense, because in one of the simplest things that you want to do in a priority queue, is you want to be able to create a priority queue, and you want to be able to run extract max on the priority queue, over and over. And what that means, is that you take the max element, you delete it, take the next max element, delete it, and so on and so forth. And there you go.

It's a bit of a preview here, but you could imagine that if you did that, you would get a sorted list of elements in decreasing order. So you see the connection to sorting, because you could imagine that once we have this heap structure, and we can maintain the max-heap property, that we could continually run extract max on it. And if you could build extract max in an efficient way, you might have a fantastic sorting algorithm.

So, the big question that really remains, is how do we maintain the max-heap property as we modify the heap? And the other question, which I haven't answered is-- this array that turns out it was a max-heap, but it's quite possible that I have a trivial example of an array. In fact, let me make this one. That is not a max-heap. It's not a max-heap, it's not a min-heap, it's neither. Right? it's just a heap.

So if I just transform, or visualize I should say, this array as a heap, I don't have a max-heap, I don't have a min-heap. So if I'm very interested in sorting, and I am, there's this another thing that's sort of missing here that we have to work on, which is how are we going to build a max-heap out of an initially unsorted array. Which may or may not turn into a max-heap. This trivially happened to be exactly the right thing, because I picked it, and it turned into a max-heap just by visualizing it. But it's quite possible that you have arrays that are input to your sorting algorithm that look like that.

OK, so let's dive into heap operations. I'm going to have spend some time describing to you a bunch of different methods that you would call on a heap. And all of these methods are going to have to maintain our representation invariant of the max-heap property. So what are the heap operations that we have to implement and analyze the complexity for? Well, we're going to have build-max-heap which produces a max-heap from an arbitrary or unordered array. So somehow I got to turn this into, for example, four, two, one. Which is in effect, sorting this array. Or changing the order. Maybe not fully sorting it, but changing the order. So that's what I have to do, and build-max-heap is going to have to do that.

In order to do build-max-heap, the first procedure that I'm going to describe to you, is called max-heapify. Heapify. Sounds a little strange, but I guess you can -ify pretty much anything. So you correct a single violation of the heap property in a subtree, a subtree's root. And I'll explain what I mean by that in just a minute.

So max-heapify is the fundamental operation that we have to understand here. And we're going to use it over and over. What it does, is take something that is not a heap, not a max-heap. When I say not a heap from now on, pretend that I'm saying not a max-heap. We're only going to be talking about max-heaps for the rest of this lecture.

What max-heapify does, is take something that is not quite a max-heap. It can't take anything arbitrary. It's going to take something where there's a single violation of the max-heap property at some subtree of this heap that is given to you, and there's a single violation of that. And it's going to fix that. And we need to be able to do this recursively at different levels to go build a max-heap from an unordered array. Then once you have that, you can do all sorts of things like insert and extract max, and heap sort, and so on and so forth.

So let's take a look at max-heapify using an example. I'm not going to write pseudocode for max-heapify. I'll run through an example, and the pseudocode is in the notes. The big assumption, and you think of this as a precondition, for running max-heapify, is the trees rooted at left i and right i are max-heaps. So max-heapify is going to look like a comma i. a is simply the array, and i is the index. Max-heapify is willing to, you're allowed to crash and not do anything useful if this precondition is violated in max-heapify. But if the precondition is true, then what you have to do is, you have to return a max-heap correcting this violation. That's the contract.

So let's take a look at an example. I think what I want to do is start over here. I want you to see all of the steps here. So we'll take a simple example, and we'll run through max-heapify. And let's take a look at 16, four-- I'm just going to draw the indices for this first example, and then I won't bother. So there you go. Is this a max-heap? No. Because right here, I've got a problem. 4 is less than 14, therefore I have a violation. And so, if you look at the call max-heapify A comma 2, this is an index 2, and all you have to do is to look at this subtree. And what you need to be satisfied in order to run max-heapify, is that the subtrees of nodes index two, which is this four node, are max-heaps. And if you go look below, you see that this is a max-heap and that's a max-heap.

Most of the time, by the way, you will be sort of working bottom up, and that's why this is going to make sense. This will all work out, because leaves are by definition max-heaps. Because you don't have to check anything. When you put two leaves together, and you want to create a tree like that, or a heap like that, then you run max-heapify. And then when you have a couple different max-heaps, and you want to put them together to make it a bigger max-heap, you'd have run max-heapify. So that's the way it's going to work.

So you want to do a max-heapify A comma 2. One of the things that's going to be important, not in this example, but when we get to sorting, is that we want to know what the size of the heap is. And in this case, the heap size is 10.

So, what does max-heapify do? Well, all max-heapify does is exchanges elements. And so, if you looked at the code for max-heapify, and you walked through it, this is what it would do. You're going to look at 4 and 14, and it's going to say, OK, I'm going to look at both my children. And I'm going to go ahead and exchange with the bigger child. So I'm going to exchange A[2] with A[4]. And what that would do is, take this, make this 4, and make this 14. And that would be step one.

And then when you get to this point, recursively, you'd realize that the max-heap property at this level is violated. And so you would go ahead and call max-heapify A comma 4. And when that happens, that call happens, you're going to look at the two children corresponding to this little subtree there, and you're going to do the exchange. You're going to have 8 here and 4 here. So you would exchange A[4] with A[8]. And now you're done, so there's no more calls.

So, fairly straightforward. It's actually not any more complicated than this. There may be many steps. What might happen is that you'd have to go all the way down to the leaves. And in this case, you went a couple of steps, and then you got to stop. But obviously, you could have a large heap, and it could take a bunch of time. So, what is the complexity of max-heapify? Anybody? Yeah. Back there.

AUDIENCE: Ultimately, potentially, if the tree is totally upside down, you could potentially switch every node to make it order in.

PROFESSOR: Every node to make it order in. Everybody, anybody. Do you have a different answer?

AUDIENCE: Log n.

PROFESSOR: Why? Why is it log n.

AUDIENCE: Because I think the worst case scenario, all of your-- the worst case scenario you would have [INAUDIBLE] on the left-hand side, [INAUDIBLE] right-hand side. And it would be skewed. [INAUDIBLE].

PROFESSOR: So you're arguing that the solution to the recurrence gives you a logarithmic complexity. Alright. Not quite. There's an easier way of arguing this. this Yeah. Back there.

AUDIENCE: [INAUDIBLE].

PROFESSOR: That's right.

AUDIENCE: [INAUDIBLE].

PROFESSOR: That's right. So what is the complexity?

AUDIENCE: Log n.

PROFESSOR: Log n. Great. Excellent. Definitely worth a cushion. Missed you by that much.

AUDIENCE: Thank you.

PROFESSOR: It's pretty soft, by the way. Right. OK. So, if I hit somebody, they get a cushion. OK. That's exactly right. Thanks for that description.

So, first off, there's two important aspects to this argument. The first thing is, that we're visualizing this is a nearly complete binary tree. It is not an unbalanced tree. Alright? We'll talk about unbalanced trees and balanced trees in the next couple of lectures. But the visualization of a heap is a nearly complete binary tree. And, in fact, if you had 15 elements, it would be a perfect binary tree. So the good news is, that the height of this visualization tree is bounded by log n. That's the good news. And you want to exploit that good news by creating algorithms that go level by level. If you can do that, you're going to have logarithmic complexity algorithms. So that was one aspect of it.

The other aspect of it, is the key assumption that we're making, with respect to build-max-heap, that there was a single violation. It is true that the answer that was given that was order n, would be a problem. I could set it up so that's actually the right answer, if I did not have this assumption-- where do I have that here-- assume that the trees rooted at left i and right i are max-heaps. So maybe that's what you were thinking.

But this is a key assumption. This is going back and like making connections between classes. This is a precondition that makes the algorithm more efficient. Makes the implementation easier. And this precondition essentially says that you have to just go down and do a number of steps, that's the number of levels in the tree, which is logarithmic.

So that's the story here with the max-heapify. It's order log n, in terms of complexity. That's the number of steps that you have. And it's a basic building block for all of the other algorithms that we look at for the rest of this lecture, and in section tomorrow.

Let's talk about how you would take max-heapify and use it to do build-max-heap.

So the first step now, let's say that we want to go and get a nice sorting algorithm. We don't like insertion sort, we don't like merge sort. We'd like to get a heap-based sorting algorithm. One of the things that we need to do, as I said, is to take an unordered array, and turn it into a max-heap, which is a non-trivial thing to do. And once we do that, we can do this extract-max deal to sort the array.

So the first step is, we want to convert an array A 1 through n into a max-heap. And the key word here is max-heap, because every array can be visualized as a heap. And I am going write the pseudocode for build-max-heap, because it's just two lines of code. And that's about the limit of a size of a program I can really understand, or explain, I should say. And this is what it looks like. Alright. that's it.

Build-max-heap says go from i equals n, by 2, down to 1. Max-heapify A of i. So someone explain to me why I can start with n over 2, and why I'm going down to 1. Yep. I saw you first.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Leaves are good. Leaves are good. I'll let you go on in a second. Leaves are good, because if you look at elements A of n over 2, plus 1 through n, are all leaves. That's a good observation. And this is true for any array. It doesn't matter what n is. Doesn't have the power of 2, or 2 [INAUDIBLE] minus 1, or anything like that. And leaves a good, because they automatically satisfy the backseat property. Continue.

AUDIENCE: OK. [INAUDIBLE].

PROFESSOR: That's exactly right. Beautiful. I won't hit anybody here. So that's it. The reason this works, is because you're calling max-heapify multiple times, but every time you call it, you satisfy the precondition. And the leaves are automatically max-heaps. Then you start with n over 2. You are going to see two leaves as your children for the n over 2 node, right? I mean, just pick an example here. Our 2 is an A of 5, right? You're out here. In this case, depending on the value of n, you may have either two children, or just one child. And you have one child. But regardless of that, that's going to be a max-heap, because it's a leaf.

And so you'll have two leaves, and you need to put them together. And that's a fairly straightforward process of attaching the leaves together. You might have to do a swap, based on what the element is. One operation and you get a little small tree, that's a max-heap. And then you do a bunch of other things that all work on leaves, because n over 2 minus 1 is probably also going to have leaves as it's children, given the large value of n. There will be a bunch of things where you work on these level one nodes, if you will, that all have leaves as children. And then you work on the level two nodes, and so on and so forth. And as I said before, you're working your way up, and you're only working with max-heaps as your left child and your right child. That make sense?

If you do that, and this is a fairly straightforward question, if you do a straightforward analysis of this, what is the complexity of build-max-heap? Yep.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. So that's order. Order n log n. Now, this is through a simple analysis. Now I'm going to give you a chance to tell me if you can do better than that. Or not. In terms of analysis. It's a subtle question. It's a subtle question, that I'm asking. I'm saying, this is the algorithm, alright? I don't want you to change the algorithm, but I want you to change your analysis. The analysis that you just did was, you said, I got [INAUDIBLE] n steps here, because it's n by 2 steps. Looks like each of the steps is taking log n time. So that's n log n. And I was careful. I put big O here. OK? Because that's an upper bond. So that's a valid answer. Can you do better? Can you do a better analysis-- and I'll let you go first-- can you do a better analysis that somehow gives me better complexity?

AUDIENCE: I think you bring it to [INAUDIBLE].

PROFESSOR: OK. How?

AUDIENCE: So each node get a maximum of two [INAUDIBLE]. So, for some n, there will be a constant number of comparisons to max-heapify that [INAUDIBLE].

PROFESSOR: Yeah. It's hard to explain. You're on the right track. Absolutely on the right track. So it turns out that, and I'll do this, it's going to take a few minutes here, because I write some things out. You have to sum up a bunch of arithmetic series, and so on. So it's a bit unfair to have to speak out the answer, but the correct answer, in fact, is that this is order n complexity. This algorithm that I put up here, if you do a careful analysis of it, you can get order n out of it. And we'll do this careful analysis.

And I'll tell you why it's order n, in terms of a hand wavy argument. A hand wavy argument is that you're doing basically, obviously no work for the leaves. But you're not even counting that, because you're starting with n over 2. But when you look at the n over 2 node, it's essentially one operation, or two operations, in whichever way you count, to build max-heap.

And so for that first level of nodes, it's exactly one operation. The first level that are above the leaves. For the next level, you may be doing two operations. And so there is an increase in operations as you get higher and higher up. But there are fewer and fewer nodes as you at higher and higher up, right? Because there's only one node that is the highest level node. The root node. That node has logarithmic number of operations, but it's only one node. The ones down on the bottom have a constant number of operations.

So I'll put all of this down, and hopefully you'll be convinced by the time we've done some math here, or some arithmetic here, but you can quantify what I just said fairly easily, as long as you're careful about the counting that we have to do. So this is really, truly counting. Analysis has a lot to do with counting. And we're just being more careful with the counting, as opposed to this straightforward argument that wasn't particularly careful with the counting.

So let's take a look at exactly this algorithm. And I want to make an observation. Which is what I just did, but I'd like to write it out. Where we say, max-heapify takes constant time for nodes that are one level above leaves. And, in general, order L time for nodes that are L levels above the leaves. That's observation number one.

Observation number two is that we have n over 4 nodes that, give or take one, depending on the value of n. I don't want to get hung up on floors and ceilings. And in any case, we're eventually going to get an asymptotic result, so we don't have to worry about that. But we have n over four nodes with level one, n over 8 with level two. And 1 node with log n, sort of the log n level, which is the root.

So this is decrease in terms of nodes as the work that you're doing increases. And that's the careful accounting that we have to do. And so all I have to do now to prove to you that this is actually an order and algorithm, is to write a little summation that sums up all of the work across these different levels.

And so the total amount of work in the 4 loop can be summed as n divided by 4, times 1, times c. So this sum, I have one level here, and I'm going to do some constant amount of work for that one level. So I'm just going to put c out there, because eventually I can take away the c, right? That's the beauty of asymptotics. So we don't need to argue about how much work is done at that one level, how many swaps, et cetera, et cetera. But the fact is that these n over four nodes are one level above the leaves. That's what's key.

And then I have n over 8 times 2c, plus n over 16 times 3c, plus 1 times log of n c. I've essentially written in an arithmetic expression exactly what I have observed on the board above. Stop me if you have questions.

Now I'm going to set-- just to try and make this a little easier to look at, and easy to reason about-- I'm going to set n over 4 to 2 raised to k, and I'm going to simplify. I'm just pulling out certain things, and this thing is going to translate to c times 2 raised to k, times 1, divided by 2 raised to 0, 2 divided by 2 raised to 1, 3 divided by 2 raised to 2, et cetera, k plus 1 divided by 2 raised to k. Now, if that was confusing, raise your hand, but it's essentially identical given the substitution and sort of just applying the distributive law.

And the reason I did this, is because I wanted you to see the arithmetic expression that's in here. Now we do know that 2 raised to k is n over four, of course. But if you look at this expression that's inside here, what is this expression? Anyone? Can you bound this expression? Someone? For the cushion. Remember your arithmetic series from wherever it was.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. You know better than I. I guess you took those courses more recently, but what happens with that? Those of you who have calculators, I mean, you could plug that in, and answer that. No one? Go ahead.

AUDIENCE: [INAUDIBLE]. You know that it's going to merge to two.

PROFESSOR: That's exactly what I was looking for. Essentially, well, it's not quite two, because you have a 1 here, and you have a 1 here, but you're exactly right. I mean, two is good. It's asymptotic, I mean, come on. I'm not going to complain about two versus three, right? So the point is it's bounded by a constant. It's bounded by a constant. This is a convergent series and it's bounded by a constant. And we can argue about what the constant is. It's less than three. And it doesn't matter of k goes to infinity. And you want k to go to infinity, but it doesn't matter if k is small or k is large, this is bounded by a constant. And that's the key observation.

What do we have left? What do we have left? We have a constant there. We have a c, which is a constant, and we have a 2 raised to k, which is really n. So there you go. There you have your theta n complexity. Now I can say theta n, because I know it's theta n. But big O of n, theta n, that's what it is.

So that's what I'd say is subtle analysis. Clearly a little more complicated than anything we've done so far, and let me see if there are questions. How many people got this? I did too. Someone who didn't get it, ask a question. What didn't you get? What step would you like me to repeat here? Any particular step?

AUDIENCE: [INAUDIBLE].

PROFESSOR: This thing here? Right here? OK, so you're not convinced that this expression got translated to this expression. So let me try and convince you of that, alright? So let's take a look at each of the terms. n by 4 is 2 raised to k. I'm just looking at this term and this term. n by 4 is 2 raised to k. c is c. And I just wrote 1 as 1 divided by 2 raised to 0, which is 1.

And the reason I want you to do this, is because I want to show you an expression where in some sense, this is the term that is the summation for your expression. If we just replace this, you can write this out as i equals 0 through k, I plus 1 divided by 2 raised to i. That is the symbolic form of this expression, which came from here.

And then the argument was made that this is a convergent series and is bounded by a constant. That make sense? Good. So that's pretty neat, right? I mean, you have the same algorithm and, whala, it suddenly got more efficient. Doesn't always happen, but that tells you that you have to have some care in doing your analysis, because what really happened here, was you did a rudimentary analysis. You said, this was order log n, big O log n, and you said this was theta n, and you ended up with this. But in reality, it's actually a faster algorithm.

So that's the good news. Build-max-heap can be done in order n time. Now in the time that I have left, it turns out, we are essentially all the way to heaps sort. Because all we have to do is use, once we have build-max-heap, I'll just write out the code for heap sort, and you can take a look at examples in the notes. The pseudocode, I should say, for heap sort. And it looks like this.

The first step that you do is you build max-heap from the unordered array. Then you find the maximum element A[1]. All of this I've said multiple times. Now the key step is, you could do extract max, but one nice way of handling this, is to swap the elements A[n] with A[1]. Let me write this out and explain exactly what that means.

Now the maximum element is at the end of the array. When you do the swap. That's the one step that I will have to spend another minute on. Now we discard node n from the heap, simply by decrementing heap size. So the heap becomes n minus 1 in size from n in the first iteration.

Now the new root after the swap may violate max-heap, we'll call it the max-heap property, but the children are max-heaps. So that's the one node that can possibly violate it. So what that means, is we can run max-heapify to fix this. And that's it .

Once you do that, you go back to that step.

So what's happened here exactly? Well this part we spent a bunch of time on. element is the maximum element, so you grab that. And you know that's a maximum element. One way of doing it is to use extract max, but rather than doing extract max, which I haven't explained to you, you could imagine that you go off and you swap the top element with the bottom element, and then you discard it.

So here's a trivial example, where let's say I had 4, 2, and 1, which is a max-heap. What would happen is you'd say, I'm going to take 4 and I'm going swap it with 1. And so you have, 1, 2, and 4. Now four used to be A[1], and that's the maximum element, and I'm just going to delete it from the heap, which means I'm going to end up with a heap that looks like-- a heap, not a max-heap-- that looks like this. And I write down 4 here. 4 is the first element in my sorted array.

Now I look at 1 and 2, and 1 and 2 there's obviously not a max-heap. But I can run max-- I know the child is a max-heap, so I can run max-heapify on this. And what this turns into is 2 and 1. And at this point, I know that the max is the root, because I've run max-heapify and I take 2 out, and after this, it becomes trivial. But that's the general algorithm.

So this whole thing takes order n log n time, because even though build-max-heap is order n and max element is constant time, swapping the elements is constant time. But running max-heapify is order log n time, and you have n steps. So you have an order n log n algorithm. But the first step was order n, which is what we spent a bunch of time on.

So I'll show you examples in the notes, and that will get covered again in section. I'll stick around for questions. See you next time.