[SQUEAKING] [RUSTLING] [CLICKING]

ERIK DEMAINE: All right, welcome back to 006 Data Structures. Today, we're going to cover a different kind of tree-like data structure called a heap-- a binary heap. It's going to let us solve sorting problem in a new way. Let me first remind you of a portion-- the problem we're going to be solving today is called priority queue. This is the interface. We'll see several data structures, but one main data structure for today.

And this is a subset of the set interface. And subsets are interesting because, potentially, we can solve them better, faster, simpler, something. And so you'll recognize-- you should recognize all of these operations, except we didn't normally highlight the max operation. So here, we're interested in storing a bunch of items. They have keys which we think of as priorities. And we want to be able to identify the maximum priority item in our set and remove it.

And so there's lots of motivations for this. Maybe you have a router, packets going into the router. They have different priorities assigned to them. You want to route the highest priority first. Or you have processes on your computer trying to run on your single threaded-- single core, and you've got to choose which one to run next. And you usually run higher priority processes first. Or you're trying to simulate a system where events happen at different times, and you want to process the next event ordered by time.

All of these are examples of the priority queue interface. We'll even see applications within this class when we get to graph algorithms. But the main two things we want to be able to support are inserting an item, which includes a key, and leading the maximum item, and also returning it at the same time. We'll also talk some about being able to build the structure faster than just inserting it. But of course, we could implement build by starting empty and repeatedly inserting. And also the complexity of just finding the max without deleting it, this you could simulate with these two operations by deleting the max and then reinserting it, which works. But often, we can do faster.

But the two key, main operations are insert and delete max. And we're going to see a few data structures to do this. Any suggestions among the data structures we've seen in this class? What should we use to solve priority queue interface? Many possible answers.

AUDIENCE: Sequence AVL?

ERIK DEMAINE: Sequence AVL? Ooh, that's interesting! Sequence AVL is a good answer, but maybe the fancier version. Yeah?

AUDIENCE: Set AVL?

ERIK DEMAINE: Set AVL sounds good. Set AVL supports these operations and many more. All in log n time except for build, which takes n log n time because you have to sort first. So set AVL a good way to do this. We'll come back to your sequence AVL idea later. This gets log n for operation. Great. I mean, this is-- set AVL is our most powerful data structure. It does all the operations we care about on the set side. And the sequence AVL does all the operations on the sequence side.

But note that this is a set, not a sequence. We care about keys. There are hacks to get around that with sequence AVLs, but let's do that later.

So great, if we wanted to, for example, speed up find_max in a set AVL. We could add augmentation. We could-- remember subtree property augmentations? We can use that to get constant time find_max by storing in every node the maximum key item within the subtree. And that's a subtree property. It's one we mentioned last class. So we could even improve that to cut some time. Great.

So we're done. End of lecture. [CHUCKLES] In some sense, that's true. But what we're going to see today is another data structure called a binary heap, which is, in some sense, a simplification of set AVL. It achieves basically the same time bounds. Build will be faster by a log factor.

But that's not the main reason we care about them. The main advantage is that they're simpler and they give us an in-place sorting algorithm. So I have up here three of the operations I've been talking about-- build, insert, and delete_max. So we have set AVL trees there-- n log n build, log n insert, log n delete.

So along the way to our heap, I want to mention two other data structures. One is a dynamic but unsorted array. And the other is a dynamic sorted array. These are simpler data structures we've talked about many times before. And they're useful kind of motivations for getting started, because a heap is going to be built on top of arrays instead of-- well, it's sort of a fusion between arrays and trees.

So if I have an unsorted array, this is very easy to insert into, right? I just append to the end. This is what we called insert last. So insert is fast, constant amortized. We might have to resize the array, but so that's the amortized part.

But delete max is slow. In an unsorted array, I don't know where the maximum is. So I have to scan through the whole array. So I scan through the array, identify that the max is somewhere in the middle, and then, if I want to delete it-- I want to delete that maximum element, well, in a dynamic array, all I can really do is delete the last element efficiently.

So I could, for example, swap it with the last element. So I take this element and put it here, and then delete the last element in that array, which is pop in Python or delete_last in our world. So overall, this is linear time, which is bad. But I wanted to highlight exactly how it's done for a reason we'll get to in a moment.

A sorted array is sort of the reverse. It's very easy to find the max. Where is it? At the end. delete_max, the maximum element is always the last element in a increasing sorted array. I guess that's constant amortized, because then I have to delete it, which may incur resizing.

Insert, though, is going to be linear, because maybe I can binary search to find where the added item belongs. Let's say I just added this item here. I could binary search to find it, but then I'm going to have to do a big shift. So I might as well just swap repeatedly until I find the position where the added item x belongs. And now I've restored sorted order. That takes linear time, which is bad.

And what we want is somehow the best of these two worlds. Insert is fast for array. Delete is fast for a sorted array. We can't get constant time for both. But we can get log n time for both. We already know how with set AVL trees. But we're going to see a different way to do it today.

And the main motivation for a different way to do this is sorting. So I want to define a priority queue sort. So given any data structure that implements a priority queue interface, in particular insert and delete_max, I can make a sorting algorithm. What do I do? Insert all the items, delete all the items. But because when I delete them they come out largest first, I get them in reverse sorted order. Then I could reverse in linear time and I've sorted my items.

So we can insert (x) for x in A, or (build(A)), and then repeatedly delete_max. How much time does this algorithm take? I'm going to introduce some notation here. It takes however long it takes to build n items, call that T sub build (n) plus-- sorry-- plus n times the time to do a delete_max. Or we can write this as n times time to do an insert, plus time to do a delete_max.

So I'm using these T functions to just abstract what are the running times provided by my data structure that implements this interface. Interface says what's correct is, and these T functions give me my performance bounds. So if I plug in each of these data structures, I get a sorting algorithm. I get AVL sort, I get array sort, I get assorted array sort. What do those look like? It turns out many of these are familiar.

So set AVLs take log n per operation. So we get an n log n sorting algorithm out of them, which is insert all of the items into the AVL tree. I don't want to use AVL build because that uses sort, and not allowed to sort in order to implement sort. But we saw how to insert into an AVL tree and keep the thing balanced. So that takes log n each. And then we can find the max, delete it, rebalance, and so on. Total time will be n log n.

This is an algorithm we call AVL sort. It's a bit complicated, because AVL trees are complicated. But it gives us optimal comparison bound and log n.

Now, what about array sort? So suppose I use an unsorted array. I insert the item. So if I insert the items-- so I'm doing all the insertions here before all the deletions. So what's going to happen is I just insert the items in the original array order. In other words, I just take the array. And then what I do is repeatedly extract the maximum item by searching for it, moving it to the end of the array, and then repeating that process.

That sound familiar? That's selection sort from lecture three. So this-- arrays give us selection sort. This is a new way to think about what we were doing way back then.

With a sorted array, what are we doing? We insert all the items. That's actually where all the work happens, because we maintain the sorted array. So we start with an empty array. It's sorted. We add an item. OK, it's still sorted. We add a second item, and we swap if we need to in order to sort. In general, when we add an item, we swap it to the left until it's sorted again. That is insertion sort.

Kind of cool, this is a unifying framework for three sorting algorithms that we saw before. We didn't actually talk about AVL sort last time, but it was in the notes. And so that is the right part of this table.

So of course, these array data structures are not efficient. They take linear time for some of the operations. So the sorting algorithms are not efficient. But they're ones we've seen before, so it's neat to see how they fit in here.

They had the-- selection sort and insertion sort had the advantage that they were in place. You just needed a constant number of pointers or indices beyond the array itself. So they're very space efficient. So that was a plus for them. But they take n squared time, so you should never use them, except for n, at most, 100 or something.

AVL tree sort is great and then it gets n log n time, probably more complicated than merge sort and you could stick to merge sort. But neither merge sort nor set AVL tree sort are in place. And so the goal of today is to get the best of all those worlds in sorting to get n log n comparisons, which is optimal in the comparison model, but get it to be in place. And that's what we're going to get with binary heaps.

We're going to design a data structure that happens to build a little bit faster-- as I mentioned, linear time building. So it's not representing a sorted order in the same way that AVL trees are. But it will be kind of tree-based. It will also be array-based. We're going to get logarithmic time for insert and delete_max. It happens to be amortized, because we use arrays.

But the key thing is that it's an in-place data structure. It only consists of an array of the items. And so, when we plug it into our sorting algorithm-- priority queue sort or generic sorting algorithm-- not only do we get n log n performance, but we also get an in-place sorting algorithm. This will be our first and only-- to this class-- n log n in-place sorting algorithm. Cool.

That's the goal. Let's do it. So what we're going to do, because we're in place, basically we have to have an array storing our end items. That's sort of the definition of in-place, just using n slots of memory exactly the size of the number of items in our structure. But we're obviously not going to use a regular unsorted array or a regular sorted array.

We're going to use array just as sort of the underlying technology for how things are stored. But we'd really like logarithmic performance, which should make you think tree. Only way to get a log is the binary tree, more or less. So somehow, we want to embed a tree into an array. Let me grab an example.

Let me draw a tree. If I got to choose any old tree I want, I would choose a tree that's basically perfectly balanced. Perfectly balanced would be like this, where-- what's the property? That I have all of these levels-- all of these depths are completely filled with nodes. This is depth 0. Remember, this is depth 1, this is depth 2, this is depth 3. So what I'd really like is to have 2 to the i nodes at depth i. That would be a perfect binary tree.

But that only works when n is 1 less than a power of 2, right? I can't always achieve that for any n. And so the next best thing I could hope for is 2 to the i at nodes at depth i until the very last i-- the largest depth. And in that level, I'm still going to restrict things. I'm going to force all of the nodes to be as far left as possible. So I want to say, except at max depth where nodes are-- I'll call them left justified. And these two properties together is what I call a complete binary tree.

Why is this interesting? Because I claim I can represent a tree like this as an array. I've narrowed things down enough that I can draw an array down here. And what I'm going to do is write these nodes in depth order.

So I write A first, because that's step 0. Then B, C, that's step 1. Then, well, they're alphabetical. I made it that way. D, E, F, G is depth 2. And then H, I, J is step 3. This is very different from traversal order of a tree. Traversal order would have been H, D, I, B, J, E, A, F, C, G, OK? But this is what we might call depth order, do the lowest depth nodes first-- very different way to lay things out or to linearize our data. And this is what a heap is going to look like.

So the cool thing is, between complete binary trees and arrays is a bijection. For every array, there's a unique complete binary tree. And for every complete binary tree, there's a unique array. Why? Because the complete constraint forces everything-- forces my hand. There's only-- if I give you a number n, there is one tree shape of size n, right?

You just fill in the nodes top down until you get to the last level. And then you have to fill them in left to right. It's what you might call reading order for writing down nodes. And the array is telling you which keys go where. This is the first node you write down at the root, this is the next node you write down at the left child of the root, and so on. So here we have a binary tree represented as an array, or array representing a binary tree.

The very specific binary tree, it has a clear advantage, which is it is guaranteed balance. No rotations necessary in heaps, because complete binary trees are always balanced. In fact, they have the best height they possibly could, which is ceiling of log n. Balanced, remember, just meant you were big O of log n. This is 1 times log n. So it's the best level of balance you could hope for.

So somehow, I claim, we can maintain a complete binary tree for solving priority queues. This would not be possible if you were trying to solve the whole set interface. And that's kind of the cool thing about heaps, is that by just focusing on the subset of the set interface, we can do more. We can maintain this very strong property.

And because we have this very strong property, we don't even need to store this tree. We're not going to store left and right pointers and parent pointers, we're just going to store the array. This is what we call an implicit data structure, which basically means no pointers, just an array of the n items.

How are we going to get away without storing pointers? I'd still like to treat it like a tree. I'd still like to know the left child of B is D and the right child B is E. We'll see why in a moment. Well, we can do this with index arithmetic. So maybe I should add some labels before I get there.

So this array naturally has indices. This is index 0. This is index 1, index 2, index 3, index 4, index 5, index 6, 7, 8, 9, because there are 10 items, 0 through 9. And I can apply those labels up here, too. These are the same nodes, so 0, 1, 2. This is just a depth order.

But once I have this labeling, it's going to be a lot easier to figure things out. So if I wanted to know the left child of B is D, somehow, given the number 1, I want to compute the number 3. Add 2, there are all sorts-- multiply by 3, there are all sorts of operations that take 1 and turn it into 3. But there's only one that's going to work in all cases.

And the intuition here is, well, I have to 2 the i nodes at level i. If I want to go to the child level, there's 2 to the i plus 1 nodes down there-- exactly double. So it's the very last one, but that won't really matter. If there is a left child, it will behave the same.

And so, intuitively, I have this space of size 2 to the i. I have to expand it to a space of size 2 to the i plus 1, So I should multiply by 2. And that's almost right, but then there's some constants. So I'd like to say 2 times i. But if we look at the examples here, 1 times 2 is 2, which is 1 less than 3. 2 times 2 is 4, which is 1 less than 5. Hey, we almost got it right. It's just off by 1. Off by 1 is-- index errors are the most common things in computer science.

What about the right child? If the left child is a 2i plus 1, where is the right child? I hear lots of mumbles. 2i plus 2-- one more. Because we're writing things left to right in depth order, the right child is the right sibling of the left child. So it's just one larger, OK?

Given those rules, we can also compute parent. It's just whatever is the inverse of both of these functions, which I want to divide by 2 at some point. I want to get back to i given 2i plus 1 or given 2i plus 2. And so if I subtract 1 from i, then I either get 2i or 2i plus 1. And then, if I take an integer division by 2, I get i-- the original i. Sorry, maybe I'll call this j to be clearer. So j is the left or right child. Then I can reconstruct i, which was the parent.

So this is constant number arithmetic operations. So I don't have to store left and right pointers. I can just compute them whenever I need them. Whenever I'm at some node like E, and I want to know what's its left child-- sorry, given the node index 4, which happens to contain the item E, and I want to know what's its left child, I just multiply by 2 and add 1. I get 9. And then, I can index into this array at position 9.

Because I don't-- this is just in my head, remember. We're just thinking that there's a tree here. But in reality, on the computer, there's just the array. So if we want to go from E to J, we can, from 4 to 9. If we go try to go to the right child, we multiply by 2. 8 add 2-- 10. And we see, oh, 10 is beyond the end of the array. But our array stores its size, so we realize, oh, E does not have a right child. This is something you can only do in a complete binary tree. In a general binary tree you don't have these nice properties.

Cool, so this is basically a heap. I just need to add one more property, naturally called the heap property. So there are multiple types of heaps. This type of heap is called a binary heap. We will talk about others in future lectures. I'm going to call it Q. Explicit thing-- this is an array representing a complete binary tree, called the array Q. And we want every node to satisfy the so-called max-heap property, which says Q[i] is greater than or equal to Q[j] for both children left of i and right of i.

So we have a node i. And it has two children-- 2i plus 1 and 2i plus 2. These are two values of j. What we want is a greater than or equal to relation here and here. So this node should be bigger than both this one and this one. Which of these is larger? We don't know, and we don't care-- very different from binary search trees or set binary trees, where we said these guys were less than or equal to this one, this one was less than or equal to all the nodes in the subtree here. We're just locally saying, this node is greater than or equal to this node and this node. So the biggest is at the top.

So one nice lemma about these heaps-- this is weird. Let me give you some more intuition. If you are a binary heap, if you satisfy this max-heap property everywhere, then in fact, you learn that every node i is greater than or equal to all nodes in its subtree. These are what we call descendants in subtree of i.

Let me look at this example. So I haven't written any numbers here. You can imagine. So A here is greater than or equal to both B and C, and B is greater than or equal to D and E, and C is greater than or equal to F and G, D is greater than or equal to H and I, and E is greater than or equal to J. That would make this structure a heap, not just a complete binary tree.

So what does that imply? It implies that A must be the maximum. So you look at any node here, like J, A is greater than or equal to B is greater than or equal to E is greater than or equal to J. And in general, what we're saying is that A is greater than or equal to all nodes in the tree. B is greater than or equal to all nodes in its subtree down here. C is greater than or equal to all nodes in its subtree.

That's what this lemma is saying. You can prove this lemma by induction. But it's really simple. If you have two nodes, i and j, and j is somewhere in the subtree, that means there's some downward path from i to j. And you know that, for every edge we traverse on a downward path, our key is going down non-strictly.

So every child is less than or equal to its parent. i is greater than or equal to this, is greater than or equal to this, is greater than or equal to this, is greater than or equal to j, OK? So by transitivity of less than or equal to, you know that i is, in fact, greater than or equal to j. Or sorry, the key in i is greater than or equal to the key in j. This is what we're calling i, the index. This is what we would call Q of i. This is Index j Q of j.

Very different way to organize keys in a tree, but as you might imagine, this is going to be good for priority queues. Because priority queues just need to find the maximum elements. Then they need to delete it. That's going to be harder, because leading the root is, like-- that's the hardest node to delete, intuitively. I'd really prefer to delete leaves.

But leading leaves and keeping a complete binary tree is actually kind of hard. If I want to delete H, that doesn't look like a binary tree, or it doesn't look like a complete binary tree anymore. It's not left justified. Similarly, if I want to delete F, that's bad. Because now, I don't have four nodes here.

The one node that's easy to delete is J, right? If I remove that node, I still have a complete tree. The last leaf, the last position in my array, is the one that's easy to delete. That's good, because arrays are good at leading the last item.

But what I've set up here is it's easy to find the max. It's going to be up here at the root. Deleting it is annoying. I'd like to somehow take that key and put it at position-- at the last position at the last leaf, because that's the one that's easy to delete. And that's indeed what we're going to do in a delete algorithm. Let me first do insert. I guess that's a little simpler, kind of symmetric to what we just said.

So if I want to insert a key or an item x which has some key, again, the only thing I really can do in an array-- if I want to add a new item, it has to go at the end. The only thing we know how to do is insert at the end of an array. This is what we called insert_last. this? Corresponds to adding a node containing x-- the item x-- in the very last level of the complete binary tree. Either it goes to the right of all the existing nodes, or starts a new level. But it's always going to be the last leaf. After we do the insertion, it will be at position size of Q minus 1.

This is probably not enough, though. We just inserted an arbitrary item in a leaf. And now, it may not satisfy the max-heap property anymore. So let's just check if it does, and if it doesn't, fix it. That's what we know how to do. But this time, we're not even going to need rotations, which is cool.

So I'm going to define an operation called max_heapify_up. This will make things more like a max-heap. We're going to start at size of Q minus 1 for our value i. But it's going to be recursive, so what we're going to do is look at a node i, in particular the one that just got inserted. And where could it violate things? Well, with its parent, because we have no idea what key we just put here. Maybe it's less than our parent. Then we're happy. But if it's greater than our parent, we're in trouble and we should fix it.

So if the item in the parent's key is less than i's key-- ah, I see I forgot to write key and all these spots. This should be dot key and dot key, because Q[i] is an item that gets its key. So this is the bad case. This is if the parent is smaller than the child. We wanted the parent to always be greater than or equal to its children. So in that case, what could we do? Swap them. Let's swap Q parent of i-- excellent, more chalk-- with Q[i]. Now they're in the right order.

Now, we need to think about what about the other child of that node? And what about its parent? So I have some numbers here. Let's say this was 5 and this was 10. What do I know about this picture before? Well, I know that 10 is this newly inserted item. It's the only one that could have caused violations when I first inserted it. So I know that before this-- before I moved 10 around, I knew all the things in this left subtree are less than or equal to 5, and everything up here are created equal to 5.

I also know that the nodes in here, in fact, were less than or equal to 5. Other than this node 10 that we just inserted, this was a correct heap. So 5 was a separator between-- things above it on the ancestor chain are greater than or equal to 5, and things in its subtree are less than or equal to it. So after I do this swap, which I'm just going to do-- after I swap the items 5 and 10, 10 is up here, 5 is here. And now, I realize, OK, great, this edge is happy, because now 10 is greater than or equal to 5. But also this edge is happy, because it used to be happy, and we only made its parent larger.

Now this edge maybe is bad. And so we need to recurse-- recurse on the parent. But that's it. So we fixed this one edge. Initially, this happens way down at the leaf. But in general, we're taking our item that we inserted, which is x, and it starts at the last leaf, and it may be bubbles up for a while. And maybe it gets all the way to the root if we inserted a new maximum item. But in each step, it goes up one.

And so the running time of all this stuff is the height of the tree, which is log n. And because there's only this one item that could potentially be wrong, if it ever stops moving, we've just checked that it satisfies the max-heap property. If it gets to the root, you can also check it satisfies the max-heap property. So there's a base case I didn't write here, which is if i equals 0, we're at the root, we're done. And then you can prove this correct by induction.

There's just one item that's in the wrong spot, initially. And we put it into a right spot. There are many places it could go, but we will move it to the, I guess, unique ancestor position that is correct-- that satisfies max-heap property, OK? So that's insert. Delete is going to be almost the same, delete_min, that is-- sorry, delete_max, thank you. You can of course define all of these things for min instead of max. Everything works the same. I just have a hard time remembering which one we're doing. Just don't switch you can't use a max-heap to do delete_min. You can't use a min-heap to do delete_max, but you can use a min-heap to do delete_min. That's fine.

So like I said, the only node we really know how to delete is the last leaf on the last level, which is the end of the array. Because that's what arrays can delete efficiently. And what we need to delete is the root item, because that's always the maximum one, which is at the first position in the array. So what do we do? Swap them, our usual trick.

I think the cool thing about heaps is we never have to do rotations. We're only going to do swaps, which is something we had to do with trees also-- binary trees. Q[0] with Q of the last item-- great, done. Now we have the last item is the one we want to delete. So we do delete_last, or pop in Python, and boom, we've got-- we've now deleted the maximum item.

Of course, we may have also messed up the max-heap property just like we did with insert. So with insert, we were adding a last leaf. Now, what we're doing is swapping the last leaf with the-- I'm pointing at the wrong picture. Let me go back to this tree. What we did is swap item J with A. So the problem is now-- and then we deleted this node.

The problem is now that that root node has maybe a very small key. Because the key that's here now is whatever was down here, which is very low in the tree. So intuitively, that's a small value. This is supposed to be the maximum value, and we just put a small value in the root. So what do we do? Heapify down. We're going to take that item and somehow push it down to the tree until the-- down in the tree until max-heap property is satisfied.

So this is going to be max_heapify_down. And we will start at position 0, which is the root. And max_heapify_down is going to be a recursive algorithm. So we'll start at some position i. And initially, that's the root. And what we're going to do is look at position i and its two children.

So let's say we put a very small value up here, like 0. And let's say we have our children, 5 and 10. We don't know-- maybe I'll swap their order just to be more generic, because that looks like not quite a binary search tree, but we don't know their relative order. But one of them is greater than or equal to the other in some order. And so what would I like to do to fix this local picture? Yeah, I want to swap.

And I could swap-- 0 is clearly in the wrong spot. It needs to go lower in the tree. I can swap 0 with 5 or 0 with 10. Which one? 10. I could draw the picture with 5, but it will not be happy. Why 10? We want to do it with the larger one, because then this edge will be happy, and also this edge will be happy. If I swapped 5 up there instead, the 5/10 edge would be unhappy. It wouldn't satisfy the max-heap property.

So I can do one swap and fix max-heap property. Except that, again, 0 may be unhappy with its children. 0 was this one item that was in the wrong spot. And so it made it have to go farther down. But 5 will be-- 5 didn't even move. So it's happy. Everything in this subtree is good.

What about the parent? Well, if you think about it, because everything was a correct heap before we added 0, or before we put 0 too high, all of these nodes will be greater than or equal to 10 on the ancestor path. And all of these nodes were less than or equal to 10 before, unless you're equal to 5. So that's still true. But you see, this tree is happy. This tree still may be unhappy. 0 still might need to push down farther. That's going to be the recursion.

So we check down here. There's a base case, which is if i is a leaf, we're done. Because there's nothing below them. So we satisfy the max-heap property at i because there's no children. Otherwise, let's look at the leaf among the left-- sorry, left not leaf-- among the two children left and right of i. Right if i might not exist. Then ignore it. But among the two children that exist, find the one that has maximum key value, Q[j].key. That was 10 in our example.

And then, if these items are out of order, if we do not satisfy-- so greater than would be satisfy. Less than Q[j] would be the opposite of the max-heap property here. If max-heap property is violated, then we fix it by swapping Q[i] with Q[j], and then we recurse on j-- call max_heapify_down of j. That's it. So pretty symmetric.

Insert was a little bit simpler, because we only have one parent. Delete_min, because we're pushing down, we have two children. We have to pick one. But there's a clear choice-- the bigger one. And again, this algorithm-- this whole thing-- will take order h time, the height of the tree, which is log n, because our node just sort of bubbles down. At some point, it stops. When it stops, we know the max-heap property was satisfied there. And if you check along the way, by induction, all the other max-heap properties will be satisfied, because they were before.

So almost forced what we could do here. The amazing thing is that you can actually maintain a complete binary tree that satisfies the max-heap property. But once you're told that, the algorithm kind of falls out. Because we have an array. The only thing we can do is insert and delete the last item. And so we've got to swap things to there in order-- or out of there in order to make that work. And then, the rest is just checking locally that you can fix the property. Cool.

So that's almost it, not quite what we wanted. So we now have log n amortize insert and delete_max in our heap. We did not yet cover linear build. Right now, it's n log n if you insert n times. And we did not yet cover how to make this an in-place sorting algorithm. So let me sketch each of those. I think first is in-place. So how do we make this algorithm in-place? I guess I want that, but I don't need this.

So we want to follow priority queue sort. Maybe I do want that. But I don't want to have to grow and shrink my array. I would just like to start with the array itself. So this is in place. So what we're going to do is say, OK, here's my array that I want to sort. That's given to me. That's the input to priority queue sort. And what I'd like is to build a priority queue out of it. Initially, it's empty. And then I want to insert the items one at a time, let's say.

So in general, what I'm going to do is maintain that Q is some prefix of A. That's going to be my priority queue. It's going to live in this sub array-- this prefix. So how do I insert a new item? Well, I just increment. So to do an insert, the first step is increment size of Q. Then I will have taken the next item from A and injected into this Q.

And conveniently, if we look at our insert code, which is here, the first thing we wanted to do was add an item at the end of the array. So we just did it without any actual work, just conceptual work. We just said, oh, our Q is one bigger. Boom! Now this is at the end of the array. no more amortization, in fact, because we're not ever resizing our array, we're just saying, oh, now Q is a little bit bigger of a prefix. It just absorb the next item of A.

Similarly, delete_max is going to, at the end, decrement the size of Q. Why is that OK? Because at the end of our delete_max operation-- not quite at the end, but almost the end-- we deleted the last item from our array. So we just replaced that delete last with a decrement, and that's going to shrink the Q by 1. It has the exact same impact as leading the last item. But now, it's constant time, worst case not amortized.

And the result is we never actually build a dynamic array. We just use a portion of A to do it. So what's going to happen is we're going to absorb all the items into the priority queue, and then start kicking them out. As we kick them out, we kick out the largest key item first, and we put it here, then the next largest, then the next largest, and so on. The minimum item is going to be here. And, boom, it's sorted.

This is the whole reason I did max-heaps instead of min-heaps, is that in the end, this will be a upward sorted array with the max at the end. Because we always kick out items at the end. We delete the max first. So that is what's normally called heapsort. You can apply this same trick to insertion sort and selection sort, and you actually get the insertion sort and selection sort algorithms that we've seen which operate in prefixes of the array.

Cool, so now we have-- we've achieved the y up there, which is n log n sorting algorithm that is in-place. So that was our main goal-- heapsort. Let me very quickly mention you can build a heap in linear time with a clever trick. So if you insert the items one at a time, that would correspond to inserting down the array. And every time I insert an item, I have to walk up the tree. So this would be the sum of the depth of each node.

If you do that, you get n log n. This is the sum over i of log i. That turns out to be n log n. It's a log of n factorial. The cool trick is to, instead, imagine adding all the items at once and not heapifying anything, and then heapify up-- sorry, heapify *down* from the bottom up. So here we're heapifying up. Now, we're going to heapify down.

And surprisingly, that's better. Because this is the sum of the heights of the nodes. And that turns out to be linear. It's not obvious. But intuitively, for a depth, this is 0, this is log n, and we've got a whole ton of leaves. So right at the leaf level, you can see, we're paying n log n, right? Because there are n of them, and each one costs log n.

Down here, at the leaf level, we're paying constant. Because the height of the leaves are 1. Here, the height of the root is log n. And this is better. Now we're paying a small amount for the thing of which there are many. It's not quite a geometric series, but it turns out this is linear. So that's how you can do linear building heap.

To come back to your question about sequence AVL trees, turns out you can get all of the same bounds as heaps, except for the in-place part, by taking a sequence AVL tree, storing the items in an arbitrary order, and augmenting by max, which is a crazy idea. But it also gives you a linear build time. And yeah, there's other fun stuff in your notes. But I'll stop there.