The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high-quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

ERIK DEMAINE: All right. Today, we resume our theme of memory hierarchy efficient data structures. And last time, we saw cache-oblivious b-trees, which achieve log base B of N for all operations-- insert, delete, search.

And the cool part is that we could do that without knowing what B was. And it was basically a binary search tree stored in a funny order, this van Emde Boas order, with an ordered file on the bottom, which we left as a black box. And today, we're going to see how to actually do ordered files-- in log squared N, data moves per insert and delete.

And then as a little diversion, we'll see a closer related problem to this is called list labeling, which we needed in Lecture 1 and left it as a black box for full persistence. We had this version tree with full persistence, and we needed to linearize that version tree into a bunch of numbers so that we could then compare whether one version was an ancestor of another. And for that, we needed to be able to store a linked list, and insert and delete in the linked list, and be able to query, who is this node in the linked list, precede this node in the linked list in constant time per operation. So we'll also do that today because it's time.

And then we're going to do a completely different cache-oblivious data structure. That's interesting, mainly in the way that it adapts to M, not just B. So remember, B was the size of a memory block. When we fetch something from memory, we get the entire block of size B. M was the size of the cache. And so there were M over B blocks in the cache of size B. So that's what we'll do today.

I'm also going to need a claim-- which we won't prove here-- that you can sort cache-obliviously in N over B log base N over B of N over B. So I'm going to use this as a black box today. And we're not going to fill it in because it's not a data structure, and it's a data structures class.

To give you some feeling for why this is the right bound for sorting, if you know M and B, then the answer is M over B way mergesort. So you all know binary mergesort, where you split into two parts. If you split into M over B parts and then do an M over B way merge, that's exactly what a cache can handle. It can read one block from each of the lists that it's trying to merge. It has just enough cache blocks for that.

And then you do the merge block by block, load new blocks as necessary. That will give you a linear time merge. And so you'll get N over B times log base M over B. And it turns out the right thing in here is N over B or N is basically the same, because it's inside the log. It's not a big deal.

So external memory wise, that's how you do it. You can do this cache-obliviously in a similar way to-- roughly speaking, in a similar way to the way we do b-trees, where you're binary searching in the number of ways you should divide your array into. I'm not going to get into details on that. We'll focus on cache-oblivious priority queues, which do a similar kind of thing, but get it a dynamic. All right. But before we go there, let's do ordered file maintenance.

So let me first remind you of the problem. We want to store N items in a file, which think of as an array, size order N. This constant's bigger than 1, with constant-sized gaps. And then-- I should say in specified order, subject to inserting and deleting items in that order.

So this was the picture. We have an array. We get to store some objects in the array and have these blank cells in between. But each of these gaps has constant size. Maybe these data items are sorted, maybe not. And then we're able to say things like, OK, insert a new item 8 right after 7. And so then you'd like to do that.

Then you'd also, then, like to say OK, now insert new item 9, here. And then this guy will maybe get shifted over. So 12 is over here. This becomes blank, and then you can fit the 9, and so on. You want to be able to do insertions and deletions like that quickly.

And quickly, here, means whenever we do an insert or delete, we're going to rearrange items in an interval. And that interval is going to be small-- log squared N amortized. That's all I need to say here.

I guess we also want to say that when we're moving these items in the interval, we can do it efficiently cache-obliviously, because we really want log squared N divided by B. And we say that via constant number of interleaved scans. Scans, we know, as long as there's a number of them and your cache has at least a constant size number of blocks, then interleave scans are always going to be efficient. You always get to divide by B. But the focus will be on making sure the interval is small. The rearrangement will actually be very simple, so not too hard to do.

So this will give us log squared N over B amortized memory transfers. So that was the black box we needed to get cache-oblivious b-trees.

Remember, we got rid of the square in the log by using a level of indirection that removed one of the logs. So we got log N over B. So we were dominated by log base B of N, which is what we had for the search over here. So this is the step we need. And this is a general tool used in a bunch of different cache-oblivious data structures, sort of one of the first cache-oblivious data structure tools. It's pretty handy.

It's actually much older than cache-oblivious or external memory models. This results-- removing the last line and this part, which makes it efficient in this model. Just thinking about moving around intervals in a file goes back to Itai, Konheim, and Rodeh in 1981. So it's pretty old. And then it was brought to the cache-oblivious world in 2000, right when this model was getting started.

So that's the goal. Now let me tell you how this is going to work.

So a rough idea is very simple. You have your array. And when you insert an item, what we want to do is find an interval containing that item of some reasonable size that's not too full and not too sparse. If we can find-- so like right here, when we're inserting 9, it looks really bad right around there. And so there's, like, too many elements packed right around that element. And that feels bad to us. So we grow an interval around it until we've got enough gaps. And then we just evenly redistribute the items in that interval.

So that's basically what we're going to do. We just have to find the right size interval to rearrange items in. Then when we do the rearrangement, it's always going to be evenly redistributing within the interval. So that strategy is simple. And to think about intervals in a nice controlled way, we're going to build a binary tree-- our good friend. So let me just draw this binary tree.

Now I need to do something a little bit special at the leaves. I'm going to cluster together log N items. So down here is the array, and all of this stuff up here is conceptual. We don't really build it. In my lecture notes, I can just copy and paste and this is a lot easier. So we have these chunks of size theta log N at the bottom. I don't really care what the constant is. 1 is probably fine.

So this is the array down here, and we're splitting every log N items, or log N cells in the array. And then we say, OK, well conceptually build a binary structure tree, here. And then this node represents this interval. And this node represents this interval. Every node just represents the interval of all its descendant leaves. We've seen this trick over and over again.

But we're not going to build any data structure or have some augmentation for each of these nodes. This is how we're going to build the intervals. We're going to start at the leaf. Let's say we want to insert an item in here. So we insert it here. If there's not room for it, we're going to walk up the tree and say, OK, if this interval is too dense, I'll look at this node and its corresponding interval here to here. If that's still too dense, I'll walk up to the parent, and so look at this interval from here to here, and so on, until I find that in the end, at most, I redistribute the entire array. And when I do that, I'll just evenly redistribute.

Let me write down the algorithm for update. So for insert or delete, same algorithm, you update the leaf log N chunk. You can do that just by rewriting the entire chunk. We're trying to get a log squared N bound, so we can afford to rewrite it interval as size log N. So that's for free.

So whatever leaf contains the element you want to insert or delete. And then we're going to walk up the tree until we find a suitable interval. And we're going to call that node, or that interval, within threshold.

So let me define within threshold. We're going to look at the density of a node, or an interval. And that's just going to be the ratio of the number of elements that are actually down there versus the amount of slots in the array that are down there-- so just how much is occupied.

So look at that ratio. If it's 100%, then there are no blank cells down there. If it's 0%, then everybody is blank. So we don't want either of those extremes. We want something in between. And we're going to do that by specifying thresholds on this density and try to keep the density within those thresholds. Let me let me define those thresholds.

The fun part is that the density thresholds that you maintain depend on which level you are. Not, like, experience points, but in which height of the tree you are. So down here, we don't really care how well-distributed the leaves are. I mean, it can't be 0% because then that would be a really big gap. But it could be say between 50% and 100%. It could be totally full.

And then once it's overflowing, then we've got to go up. And the higher we go, the stricter we get-- hopefully, yes-- strictest at the top of the tree.

So in general, if we have a node of the depth, d, then we want the density to be at least 1/2 minus 1/4 d over h. And we want the density to be at most 3/4 plus 1/4 d over it h. So h, here, is the height of this tree, which is going to be something like log N minus log-log N. But it doesn't really matter. This is depth 0, depth 1, depth h.

We just are linearly interpolating between-- let's see, this is always between 1/4 and 1/2-- 1/2 when this is 0, 1/4 when this is h. So you get a 1/2 minus 1/4. And this one is always in the range 3/4 to 1. It's 3/4 when this is 0, and 1 when this is h.

So at the bottom-- I was a little bit off. At the bottom, the leaf level, when these are both h-- the density has to be at least a 1/4 and at most, 100%. And then at the root, it's going to have to be between 1/2 and 3/4. So it's a narrower range. And the higher you go up, the more narrow the range on the density gets. And we do it just sort of in the obvious linear interpolation way.

The not so obvious thing is that this is the right way to do it. There's a lot of choices for how to set these density thresholds. But we have to basically maintain constant density everywhere because we're trying to maintain gaps of constant size. So we don't have a lot of flexibility. But it turns out, this flexibility between two constants, like 1/4 and 1/2 is enough to give us the performance we need. So let's see why.

Let me finish this algorithm. We walk up the tree until reaching a node within threshold. Density is this. Density threshold is this. So now we know within threshold means. And then we evenly rebalance or redistribute all the descendant elements in that interval that is within threshold.

So what you need to check is that you can do this with a constant number of scans. It's not that hard. Just read the elements in order. Write them out to a temporary array, and then write them back. Or if you're fancy, you can do it in place. But you can just do it by a constant number of scans through the array. Just compute what should be the average gap between the elements. Leave that many gaps.

So the algorithm is pretty simple once you say, OK, I'm going to grow intervals. Then maybe, you think OK, I guess I'll grow intervals according to a binary tree. It's a little bit more controlled. Probably don't have to do it this way. You could just grow them by a factor of 2, just around your point. But it's easier to analyze in the setting of a binary tree.

And then once you're doing that, the tricky part is to set the density thresholds. But you fool around and this seems to be the best way to do it. Now the question is, why does this work.

How do we prove log squared amortized interval size when we follow these dense thresholds?

Notice that we're not keeping intervals within density at all times. I mean, the whole problem is that things are not within threshold right at the start. And we have to walk up the tree quite a ways, potentially-- claim is only about log-log N levels up the tree-- to find something that's within density. And then we can redistribute. And then we fix everything below us. All right. Well, let's get to that.

This is really the direction we want. The thresholds are getting tighter and tighter, more constrained as we go up. Because it means if we walk up a lot, we, essentially, can pay for it because we bring that interval even farther within threshold.

So we have some node, which is within threshold. So we bring it into the density thresholds of here. If we look at the children of that node, their density thresholds are smaller-- sorry, are more relaxed. So if we bring this node into threshold by rewriting all the leaves down here, these nodes will not only be within threshold, they'll be far within threshold.

If you look at their ratios, you know, their densities-- the number of elements in there, divided by the array slots. It's going to be exactly the same. The density is equal because we're uniformly distributing the items here. And there's some rounding errors, but other than rounding. And that's actually why we have these leaves as size theta log N, so the rounding doesn't bite us.

We're evenly redistributing so the density is equal everywhere. Left child had the same density as the parent. But if you look at the density thresholds of the child, they will be more relaxed compared to the parent. So if the parent is within threshold, the child will be far within threshold by at least a d over h additive amount. Sorry, 1 over h, because their depths differ by 1. If this is d, this would be d plus 1.

When we rebalance a node, we put the children far within threshold. Meaning, if we look at the absolute difference between the density and either the upper threshold or the lower threshold, that will be, I guess, at least 1 over 4h because we're increasing d by 1. And it's 1 over for 4h for each step we take. OK so the children are extra happy.

We walked up here because before-- let's say we walked up this path. So we walked from the right child. We didn't stop here, which means this node was was beyond threshold. But now we walked up and now we fixed this entire interval. And now it's far within threshold.

So before, you know, the density minus the threshold went the wrong way, had the wrong sign. Now we're good, and we're good by at least a 1 over 4h. Now h, here, was the height of the tree. It's log N minus log-log N. All we need is that this is theta log N-- sorry, theta 1 over log N. h is theta log N.

And this is a ratio-- 1 over log N-- of the number of items versus the number of slots. But we know the number of slots we're dealing with is theta log N. And so this is at least one item. This log N is designed to balance the h, here. OK, cool.

Let's go over here. So the idea is if we're far within threshold, we can charge to those items. That's our goal.

What we're interested in-- if we just rebalanced this node, say x. We want to know when is the next time x can be rebalanced? For x to have to be rebalanced, that means, again, one of its children will have to be out of threshold. And then we insert or delete within that child. And then that propagates up to x.

But right now, the children are far within threshold. So the question is, how long would it take for them to get out of threshold again? Well, you'd have to change the density by at least 1 over an additive-- 1 over log N. If you multiply by the size, it's the size of the interval divided by log N. You've got to have at least that many insertions or deletions.

Before this node rebalances again, one of its children must get out of balance. And so you must have done at least the size of the interval divided by theta log N updates for one of the children to become out of balance again. Boom. So when this rebalance happens again, we're going to charge to those updates, which is good because the time it takes us to do the rebalance is the size of the interval. We need to charge each of these items log N times.

So charge the rebalance cost, which is the size of the interval to these updates. And what we know is that the updates are within the interval.

So this looks like a log N bound, which is not right. It should be a log squared N bound. The idea is when we insert into one of these leaves, we're simultaneously making this node worse and this node worse and this node worse. Whenever we insert a node, it belongs to log N intervals that we care about.

So in fact, not only are we losing this log N, because there aren't quite enough items to charge to-- the log N factor less. We're also charging to each item another factor of log N times because it lives in all these different intervals.

Each update gets charged at most, h, which is order log N times.

We looked at what happens for node x, but we have to apply this argument simultaneously for all nodes x. Fortunately, this node versus this node, they don't share any descendants. And so there's no multiple charging. But node x and its parent and grandparent and all its ancestors, they're all talking about the same nodes. And so they will multiple charge, but only by a factor of log N. That's something we've seen a few times, charging log N times, for every node in the tree, like range trees having N log N space in two dimensions. Same deal.

So we've got size of the interval divided by log N guys to charge to, which we multiply charge log N times, so we get a log squared amortized bound.

So this log N is hard to avoid because we have a binary tree that's pretty natural. This log N, essentially, comes from this h. The fact that we can only go from one constant factor to another. And we've got log N different steps to make. We have to do 1 over log N increment every step. That's the best we could afford. That's why these are evenly spaced out in this linear way.

But if we had a little more space, we could do better. So that's going to lead us to this list labeling problem. But first, are there any questions about order file maintenance? At this point, we are done. Yeah.

AUDIENCE: Can you explain again how is it that you get from the size of the interval [INAUDIBLE] and that each--

ERIK DEMAINE: This amortization?

AUDIENCE: Yeah, how you got to the amortized.

ERIK DEMAINE: Yeah. So let me explain again. So when we do rebalance of an interval, the cost is the size of the interval. We're trying to analyze, what is the size the interval? Prove that is log squared N. So we have this cost of size of interval. We're charging it to the items which just got inserted or deleted into that interval. Before this node rebalances again, but in general, we're interested in the-- we can afford to rebalance every node at the beginning.

And then whenever a node rebalances-- before it rebalances, one of its children had to be out of whack. For one of its children to be out of whack, there had to have been an insertion of at least the size of the interval divided by log N, because log was h.

There's a slight discrepancy, here. We're talking about the size of the parent interval versus the size of the child interval, but that's just a factor of 2. So that's incorporated by this theta. So you could have a little theta here, too, if you like.

OK, so for the child to be out of whack, we had to have done updates of size interval divided by log N. So we charge this cost to them. And so we have to charge log N to each of those items, each of those updates.

And then there's a second problem, which is everybody gets charged by all of its ancestors. And it has log N ancestors. So in all, each update gets charged at most log squared N times. So you get amortized log squared per update.

Other questions?

Cool. So that's ordered files. Now we have b-trees. We can handle this log squared N. That was OK for B trees, using a layer of indirection. But it's natural to wonder whether you can do better. In general, the conjecture is that for ordered files, you cannot do better. If this is your problem setup, if you can only have constant sized gaps, you need a linear size array. You need log squared N interval updates, but no one's proved that lower bound. So that's one open question.

Another fun fact is we did amortized, but it's actually possible to do worst case. It's complicated. Willard did it in 1992.

But let's talk about relaxing the problem. So instead of saying, well, the array has to be linear size, what if we let it to be bigger? And then you can do better. And this is a problem called list labeling.

So I'm going to rephrase it, but it's essentially the same as order file maintenance. It's a little bit less restriction.

So it's a dynamic linked list problem, like we've seen before. And each node at all times stores a label such that labels are always monotone down the list.

So think of it this way. We have a linked list. And all we're interested is maintaining this linked list. So we want to be able to say, OK, delete this item and update these pointers. Or maybe, insert a new item over here. And so it's going to be linked like that. And at all times in this cell here, we're storing a number, 3, 7, 12, 14, 42, whatever. It could be any integer, let's say. And we need that the numbers are increasing order down the linked list.

Now, I claim this is basically the same as an ordered file problem. Just think of this number as being the index in the array that you store it. So this should be strictly monotone-- I guess, increasing down the list. That means none of these numbers are the same. So we can store the items-- whatever data is associated with this node-- we can store that in the array position 3, in the array position 7, in the array position 12. When we insert, we have to find a new label between 3 and 7.

Now we're allowed to change the labels dynamically-- that's what makes this possible-- which corresponds to moving items in the array. And the only difference is about how this is physically done. With ordered file, you have to physically move items in the array. Here, you just change a number and that's moving the item.

Where this gets interesting is if you allow the label space-- which is the storage space of the array-- to become super linear. With ordered files, it doesn't really make a lot of sense to go super linear, at least, not by more than a couple log factors because time is always at least space. If you have a giant array, you have to initialize it. And you can't afford to initialize an array of, say, N squared size, if you're trying to maintain N items.

But in list labeling, you can. I mean, if you say all these numbers are between 0 and N squared, that's no big deal because you can represent a number up to N squared, which is two log N bits. So squaring the space is no big deal for list labeling. It would be a big deal for order file maintenance, so that's why we rephrase. But they're basically the same problem-- just a little less useful for cache-oblivious stuff.

Let me tell you what's known about mislabelling. In terms of the amount of label space you're given, and how good a running time per update we can get, in terms of best known results.

So what we just saw is that if you do-- we just said linear space. But in fact, you can get 1 plus epsilon space. You can say, oh, I wouldn't want to waste 1% of my storage space in the array. And if you set the theta at the leaves, here, to the right value, then you can just waste 1% and still maintain an ordered file. I think this is cool from a file system perspective. But in particular, it gives us what we need.

And if you go up to N log N space, it doesn't seem to help you much. The best we know is log squared N. As I mentioned, it could be amortized or worst case.

If you bump up the space to N to the 1 plus epsilon-- so a little bit super linear-- and anything polynomial, then the best we know is log N. And there is actually a lower bound for this result in a particular model. So it seems pretty clear that-- at least for these style of data structures, the best you can do in this situation is log N. But hey, log N is better than log squared.

And the other obvious bound is if you have exponential space, you can do constant, essentially. Because with exponential space, you can just keep bisecting the interval between two items in constant time until you've inserted N items. And then you can rebuild the whole data structure. So that's sort of the trivial result.

If you don't care about how big these labels get-- and how big they would get is 2 to the N-- then you can do constant time. That's really the only way we know how to do constant. So the interesting new result, here, that I'm talking about is for polynomial space, we can get log N.

And rough idea is you just fixed these density intervals. Now you don't have to make it-- I mean, you're still going to be divided by h, here, but your spread can be much bigger. So your densities no longer have to be constants. Now we can afford a density-- near the root, we can get a density of like 1 over-- actually, anywhere, we can afford a density of 1 over N. Because if we have N squared slots and only N items to put in them, then a decent density is 1 over N, in fact.

It could also be constant. Constant would be all right. So we've got a big spread there, from 1 over N to constant. And so we can afford to take much bigger jumps here, of like 1 over N. And so that gets rid of this log N factor, essentially. That was the rough sketch.

I have written, here, that the densities we use are no longer uniformly spaced. The 1 over alpha to the d. Alpha, here, is some constant in the interval between 1 and 2. And d is the depth. So now we have exponentially increasing densities which give you a big gap-- no longer lose this log N factor.

So you do exactly the same data structure, these different densities. Now you've got room to fill in a whole bunch more densities when you have, say, N squared space. And so you only get the log N factor because of the number of ancestors of a node. And you lose the other log N factor.

Now let me tell you about another problem building on this. So this is the list order maintenance problem. And this is the problem we saw in a Lecture 1.

So here, same as before, maintain a linked list, subject to, insert a node at this location, delete a node at this location, and order queries. Is node x before node y in the list? This is what we needed to support full persistence. You have a big linked list. And then if I give you this node and this node, I want to know that this node is before that node in the list in constant time. I claim that we can solve this problem given our solutions to list labeling-- not so obvious, how.

List labeling is great because it lets you do order queries, right? You just compare the two labels and you instantly discover which is before which. And so if we could afford log N time over there, we could just use this solution. And this is reasonable. But we really want constant time per operation. Now if we do constant time per operation and we use exponential label space, this is not so good because it means you need N bits to write down a label. So it's going to take, like, linear time to modify or to compare two labels. This doesn't save you.

We can afford to do this. This is only log N bits per label. And we assume all of our integers can store at least log N bits-- something called a Word RAM Model. And so we can afford to store these labels, but we pay this log N time. So you've got to remove a log N factor. How do we do that? Indirection-- just like last class. Let's do that on this board.

On the top, we're going to store N over log N items using this list labeling with label space, say, N squared. Any polynomial will do. And so this takes log N per operation to do anything on these N over log N items. And then at the bottom, we have lots of little structures of size log N. And that's supposed to eat up a factor of log N in our update time if we do it right.

Actually, I can just do list labeling down here as well. So if I'm only storing log N items, then I can afford to use the exponential solution-- the trivial thing where I just bisect all the labels until all the log items have changed. Then, rewrite them. Because 2 to the log N is only N. So here, in each of these, I do a list labeling with space 2 to the log N, also known as N.

So these guys are constant time to do anything in to maintain the labels. And to maintain levels up here-- so to maintain, basically, each of these N over log N guys, is one representative element representing this entire group as N over log N of these groups. So this label structure is supposed to distinguish the different groups. And then the labels in here are distinguishing the items within the group.

Now this cost log N amortized to keep what we need up here. But now if I want the label of an item, I just take this label, comma, this label. So if I have an item here, for example, at first, I look at the label of this block as stored by this data structure. And then I look at the label within the block. And so my composite label is going to be an ordered pair of top label, comma, the bottom label.

This is kind of funny because it looks like we're solving the list labeling problem again, with now a space of N cubed. We've got N squared for the first coordinate, and then N for the second coordinate. So if you just concatenate those two labels, that lives in a bigger label space of size N cubed.

And yet, I claim this takes constant amortized and is not a solution to list labeling. It's just a matter of, again, how updates are performed. With order file maintenance, we had to physically move items. With list labeling problem, we had to modify the number stored with each node. That's expensive. In this world, if we change a label up here, we are basically instantly-- say, we changed the label corresponding to this group in here. We changed the label all of these items in one operation. Or actually, it takes log N time. And then we change the label of all log N of these items. So that's why we get constant amortized.

So how does this work? If we insert a new item, we stick it into one of these blocks. The block that it fits into. If this block gets too full, more than, say, 1 times log N, then we split it in half. If it gets too sparse by deletion, say, less than a 1/4 log N, then we'll merge it with one of its neighbors, and then possibly split. Each of those triggers a constant number of operations up here. And so we pay log N, but we only pay it when we've made theta log N changes to one of these blocks. So we can charge this log N cost to those log N updates. And so this turns into constant. This down here is always constant. And so it's constant amortized.

And if each of these blocks remembers what the corresponding-- and, basically, this is a linked list, here. And then we have linked list down here, and they have labels. And it's like a, what do you call, a skip list with just two levels. And every node here just remembers what its parent is up here.

So if you want to know your composite label, you just look at the bottom label and then walk up. Look at the top label. Those are just stored right there in the nodes. And so in constant time, you can find your top label, your bottom label. Therefore, you can compare two items in constant time. So this solves the problem we needed in Lecture 1. Question?

AUDIENCE: Sorry, why isn't it the same as the N cubed label space?

ERIK DEMAINE: OK. Why is it not same as N cubed label space, which I claim has a lower amount of log N? The difference is with list labeling, you have to explicitly change the label of each items. And here, we're basically computing the label of a node now becomes a constant time algorithm. We're allowed to look at the label, here, then walk up, then look at the label, here. And by changing the label up here, we change it simultaneously for log N guys down there. So that's the big difference between this list order maintenance problem from the list labeling problem.

These were actually all solved in the same paper by Dietz and Slater. But sort of successively-- slightly relaxing on a problem makes a huge difference in the running time you can achieve. Obviously, we can't get any better than constant. So we're done in-- of course, then there's external memory versions-- but in terms of regular data structures. And again, Willard made it worst case constant. That's a lot harder.

Other questions? Cool. So that does order file maintenance and list labeling. And so next, we're going to move to a very different data structure, which is cache-oblivious priority queue.

We haven't really done any cache-oblivious data structures, yet. So it's time to return to our original goal. We're not actually going to use ordered files. Sadly, that was last lecture. So All of this was a continuation of last lecture. Now we're going to do a different data structure. It's going to adapt to B, it's going to adapt to M. It's cache-oblivious. And it achieves priority queue.

Now remember, this sorting bound-- N over B log base M of N over B This is our sorting bound. Then the priority queue bound we want is this divided by N. So we want to be able to do 1 over B log base M over B, N over b. Insert and delete min, let's say. And this is an interesting bound because it's usually much less than 1. It's a sub constant bound. This is going to be a little o of 1, assuming this log is smaller than this B.

So it's a little o of 1 if, let's say, B is bigger than the log N, roughly. This would, in particular, be enough. No pun intended-- be enough. So if our block size is reasonably big-- cache line is bigger than log N, which is probably most likely in typical caches and architectures-- then this is more like 1 over B, never mind the log. And so we have to do B operations in one step. And then to really get the log right, we have to depend on M, not just B.

As I mentioned, that's the bound we're going to achieve. We do need to make an assumption about M and B and how they relate. So we're going to assume a tall cache, which is M is, let's say, B to the 1 plus epsilon. So it has to be substantially bigger than B, not by a huge amount. It could be B squared. It could be B to the 1.1 power. But M is definitely at least B. And I want it to be a little bit bigger. I want the number of blocks to be not so tiny. Here's how we do it.

I think I need a big picture.

So the kind of funny thing about cache-oblivious priority queues is we're not going to use trees. It's basically a bunch of arrays in a linear order. In some ways, it's an easier data structure. But it's definitely more complicated. But, hey, it's faster than b-trees. If you don't need to be able to do searches-- if you just need to be able to delete min, then priority queues are a lot faster than b-trees.

Let me draw this. I want to try to draw this pseudo-accurate size. The only part that's inaccurate about this picture is the dot dot dots. Lets me cheat a little.

So we have x to that 9/4, x to the 3/2 and x. This is sort of what the levels are going to look like. So we have a linear sequence of levels. They are increasing doubly exponentially in size. If you look at what's going on here-- double exponential--

AUDIENCE: Are those exponentials the same as those? Or are they inverted?

ERIK DEMAINE: It's supposed to be this-- so this is the top level, and this is the bottom level. So at the top, we're going to have something of size N. And at the bottom, we're going to have something of a size C, constant. And what I've drawn is the generic middle part. So we go from x, to x to 3/2, to x to the 9/4. Or you could say x to the 9/4, down to x to the 3/2, down to x, according to this exponential geometric progression, I guess. So that's how they go.

So I mean, if it was just exponential, it would be, like, N, N over 2, N over 4. But we're doing-- in logarithms, we're changing by constant factors. So it's doubly exponential. And then each of the levels is decomposed into top buffers and bottom-- or, sorry, up buffers and down buffers. And as you see, the size of the top buffer is equal to the sum of the sizes of the bottom buffers. So let me elaborate on how this works.

It's level x to the 3/2. We're going to, generically, be looking at x to the 3/2. So I can easily go down. That's of size x. And up is x to the 9/4-- easily.

There's always one up buffer of size x to the 3/2. So maybe some color. So this buffer, here, is x to the 3/2.

And then we also have up to x to the 1/2 down buffers, each of size theta x. Each of these guys has size theta x. And then the number of them is, at most, x to the 1/2. And so you take the product, that's x to 3/2, which is the same as the up buffer. So that's the way this cookie crumbles. That's how each of these guys decomposes into little pieces. And so, in particular, this up buffer, if you work it out, should be exactly the same size as this down buffer.

Maybe it's easier to do these two. So this down buffer has size x. And if you have a structure size x, the up buffer has size x. x of 3/2 has an up buffer size of x 3/2. So this will be size x. And so these match. These are equal size. That's why I drew it this way.

Now the dot dot dot hides the fact that they're x to the 1/2 of these. So there's a lot of down buffers. They're actually a lot smaller than up buffers. But these two match in size and these two match in size, and so on.

OK, there's a small exception. I'd put theta x here. The exception is that the very first down buffer might be mostly empty. So this one is not actually data x. Each of these will be theta x. And this one over on the left will be big O of x. Small typo in the notes, here. So if that's not sufficiently messy, let me redraw it-- make it a little cleaner.

I want to look at two consecutive levels and specify invariants. So after all, I'm trying to maintain a priority queue. What the heck is this thing? The idea is that towards the bottom, that's where the min is. That would seem good, because at the bottom, you're constant size. I can diddle with the thing of constant size if fits in a block or if it's in cache. It takes me zero time to touch things near the bottom. So as long as I always keep the min down there, I can do fine min in zero time. And delete min will also be pretty fast.

I'm also going to insert there, because I don't know where else to put it. I'd like the larger items to be at near the top. I'd like the smaller items to be near the bottom, but kind of hard to know, when I insert an item, where it belongs. So I'll just start by inserting at the bottom. And the idea is as I insert, insert, insert down here, things will start to trickle up. That's what the up arrows mean, that these items are moving up. And then down arrow items are items that are moving down. Somehow this is going to work. So let me tell you the invariants that will make this work.

If you look at the down buffers, they are sorted. Or I should say, all the items in here are less than all the items in here, and so on. But within each down buffer, it's disordered. Then we also know this. All the items in the up buffer in a given level-- this is x to the 3/2, let's say. And this is x. All of the up items are larger than all of the down items. I mean, it's just an inequality here. So these guys are basically in a chain. Again, the items in here are not sorted. But all the items here are bigger than all the items here are bigger than all of the items here are bigger than all the items here.

Now what about from level to level? This is a little more subtle. What we know is this. So again, we know that all the down buffers are sorted. And we know that this down buffer, all these items come before all this down buffer items. But these up buffer items, we don't know yet, because they're still moving up. Now we know this. They need to move up. But we still don't know how far up. We don't know how these items compare to these items. Or these items could have to go higher, could be they belong here, or are here, who knows.

So basically, the down buffers are more or less sorted. And so the mins are going to be down at the bottom. And the up buffer items, I mean, they're still moving up. We don't know where they belong, yet. But eventually they'll find their place and start trickling down. Roughly speaking, an item will go up for a while and then come back down. Although, that's not literally true. It's roughly true.

I should say something about how we actually store this in memory, because the whole name of the game is how you lay things out in memory. In cache-oblivious, that's all you get to choose, basically. The rest is algorithm, regular RAM algorithm. And all we do is store the items in order, say, from bottom to top. So store the entire C level, then the next level up, to store these items as consecutive arrays. That's what we need to. Leave enough space for x to the 1/2 down buffers, each at size theta x.

So how do we do inserts and deletes? Let's start with insert. As I mentioned, we want to start by inserting in the bottom, until we run out of room in the bottom. And then things have to trickle up. So here's the basic algorithm.

You look at the bottom level. You stick the item into the up buffer. This is not necessarily the right thing to do. So here, you stick it into the up buffer. But these things are supposed to be roughly sorted. So once you stick it there, you say, oh, well maybe I have to go down here. Go down here. The point is the up buffer is the only one I want to be growing in size.

So I insert into the up buffer. And say, oh, potentially I have to swap down here. Swap down here. I mean, this is constant size. I can afford to look at all the items here in zero time and find out which buffer it belongs to. As I move the item down here, I swap. So I take the max item here and move it up to here. Move the max item from the next down buffer and propagate it up, so that I keep these things in order.

Swap into bottom down buffers if necessary-- or maybe, as necessary. You might have to do a bunch of swaps. But it's all constant. And then the point is the only buffer that got larger by one item was the up buffer, because, here, we did swaps to preserve size. And if that overflows, then we do something interesting. And something interesting is called push. This is a separate team, which I'll define now.

So this is the bottom level push. But I wanted to find the generic level of push, which is we're going to be pushing x elements into level x to the 3/2. So why is that? Check it out.

If we're at level x and our up buffer overflows, and we're trying to push into the next level-- that's level x to 3/2-- then the up buffer that we're trying to push has size x. The up buffer has the same size as the level name. So if we're at level x, this has size x. We're trying to push x items up into the next thing. We're going to empty out the up buffer. Send all those items up there.

Cool. So what do we do? How do we do this push? First thing we do is sort the items-- those x items. This is where I need the black box, which is that we can sort N over B log base N over B, N over B cache-obliviously. So we're going to use that here. It's not hard, but I don't want to spend time on it.

Now the interesting bit is how we do the push. The tricky part is we have all these items, we know that they're bigger than all the items below them. Maybe, here's a better picture. We know these guys need to go up. They're bigger than everything, all the down items below us. But we don't know, does it fit here? Here? Here? Or here?

But we do know that these things are ordered. And so if we sort these items, then we can say, well, let's start by looking here. Do any of these items fit in this block? Just look at the max, here. And if these guys are smaller than the max, here, then they belong here. So keep inserting there. Eventually, we'll get bigger than the max, then we go here. Look at the max item, here. As long as we have items here that are smaller than the max here, insert, insert,

insert, insert, and so on.

And then when we're beyond all of these-- maybe we're immediately beyond all of these. We have to check, oh, bigger than the max, bigger than the max, bigger than the max. Then we put all the remaining items into the up buffer. This is called distribution.

So we're looking at level x to the 3/2, and just scanning sequentially. Because we just sorted them, we're scanning them in order. We visit the down buffers in order. And insert into the appropriate down buffer as we go.

Now there's a little bit that can happen, here. Our down buffers have a limit in size. Down buffers are supposed to have size theta x at level x the 3/2. So as we're inserting into here, a down buffer might overflow. I've got theta slop, here. So when a down buffer overflows, I just split it in half. I then make two down buffers. Well, actually, it would be, like, here and right next to it.

I'm going to maintain a linked list of down buffers. And each of them will have space for say, 2x items. But once I do a split, they'll both be half full. So when a down buffer overflows, split in half and maintain a linked list of down buffers.

Another thing that can happen is when you increase the number down buffers, we have a limit on how many down buffers we can have-- up to x the 1/2 of them. So if we run out of down buffers by splitting, then we need to start using the up buffer. So maybe here, we split, maybe, this buffer-- is now too many down buffers total. Then we'll just take all the elements in here and stick them into the up buffer, because that's where they belong.

So when the number of down buffers is too big, when that number overflows, move the last down buffer into the up buffer. So there's basically two ways that elements are going to get into the up buffer. One way is that we run out of down buffers, and so the last down buffer starts getting promoted into the up buffer.

The other possibility is that we're inserting items that are just really big in value. And if the items that are getting promoted from here into the next level just happen to be larger than all these items, we will immediately start inserting into the up buffer. But in general, we have to look at this down buffer. Look at this one. Look at this one, then that one. That is insert and push.

I think before I talk about delete, I'd like to talk about the analysis of just insert and push. Keep it simple. And then I'll briefly tell you about deletion.

Oh, I didn't say. Sorry. There's one more step, which is the recursion. Running out a room, here. Maybe I'll go over here. This is nothing relevant.

So I need to recurse somehow. At some point, inserting things into the up offer, the up buffer might overflow. That's the one last thing that could overflow. When that happens, I just push it to the next level up.

So as we do insertions in here, we might start inserting a lot into here. Eventually, this will get too big. It's supposed to have size x the 3/2. If it gets bigger than that, take all these items and just recursively push them up to the next level. And conveniently, that's exactly the same size as we were doing before. Here, we did x into level x to the 3/2. Next will be x to the 3/2 into level x to the 9/4, and so on. Always the size of the up buffer.

So the claim is if we look at the push at level x to the 3/2-- which is what we just described-- and we ignore the recursion. Then the cost is x over B log base M over B of x over B-- so sorting bound on x items.

So we spend that right away in the very first step. We sort x items. So that costs x over B. It has log base M over B of x over B. And the whole point of the analysis is to show that this distribution step doesn't cost any more than the sorting step. So let's prove this claim.

And the first observation-- maybe, don't even need to write this down. So remember, with cache-oblivious b-trees, we looked at a level of detail that was sort of the relevant one that straddled B. Now we have this data structure and there's no longer recursive levels in this picture. It's just a list. But one of the things we said is that if you look at the very small structures, those are free because they just stay in cache.

I'm going to assume-- it's a little bit of a bastardization of notation-- assume that all the levels up to M fit in cache. It's really up to, like, size M over 2 or something, but let's just call it all. If x is less than M, then you know all this stuff has size order M. And so let's just say-- by redefining what M is by a constant factor-- that just fits in cache. Because we can assume whatever cache replacement strategy we want, assume that these things stay in cache forever.

So all of that bottom stuff, size up to M, is permanently in cache and costs zero to access. So those levels are for free. So it's all about when we touch the upper levels. And, really, the most important level would be the transition from the things that fit in cache to the next level up that does not fit in cache. That's going to be the key level.

But in general, let's look at push. Or let's just assume x to the 3/2 is bigger than M. Because we're looking at a push at level x to the 3/2. And just by definition, if it's less than M, it's free. Question?

AUDIENCE: So how can you assume that all of the things below that particular size always stay in cache empty? So you're making assumptions on the cache replacement?

ERIK DEMAINE: I'm making assumption on the cache replacement strategy, which I actually made last lecture. So I said, magically assume that we use optimal cache replacement. So whatever I choose, opt is going to be better than that. So you're right. The algorithm doesn't get to choose what stays in cache.

But for analysis purposes, I can say well, suppose all these things stay in cache. If I prove an upper bound in that world, then the optimal replacement will do better. And the LRU or FIFO replacement will do within a constant factor of that, again, by changing M by a constant factor. So I'm freely throwing away constant factors in my cache size, but then I will get that these effectively stay in cache. FIFO or LRU will do that just as well, or almost as well. Good question.

So now we can just look at the pushes above that level. And I really want to look at this transition level, but we're going to have to look at all of them. So let's do this. Now I also have tall cache assumption. M is at least B to the 1 plus epsilon.

I actually want a somewhat bigger assumption, which is that M is greater than or equal to B squared. If it's not B squared, you have to change this 3/2 and 9/4 and stuff to be something a little bit bigger than 1. And it gets really messy to write that down. So this data structure with appropriate modification does work for other tall cache assumptions, but let's just assume this one. So this means M over B is at least B. That's sort of the clean statement. It's true for most caches also, but this will be my assumption.

OK. Cool. So if we just do some algebra. Claim is this is x is greater than B to the 4/3. So that's just taking this inequality, x the 3/2 is greater than or equal to B squared and raising to the exponent, 2/3. And so this turns into x and this turns into B to the 2 times 2/3, which would be the 4/3. So x is quite a bit bigger than B.

In particular, this means that x over B is bigger than 1, by a lot. But if we take ceiling's on this, no big deal. So we don't have to worry about the ceilings.

All right. Now the claim is that the distribution step costs-- this is really the interesting part-- x over B plus x to the 1/2 memory transfers. OK. Why?

Remember, up here we have x to 1/2 down buffers that we're looking at. And we're visiting them in order. We touch this down buffer. And really, we just care about the max, here. And then we start writing elements one by one. But if we write elements one by one, the very first element we write costs an entire memory transfer. We have to load in a block. But then we can fill that block and then write out that block. So we get to write out B items in one step.

So we pay x to the 1/2 because we have to touch the last block of this down buffer, this down buffer, this down buffer. And they're x to the 1/2 down buffers. So each one, we have to pay 1. That's this part. That's the expensive part.

But then, once we're actually writing items, if we stay here for a while, that's basically for free. Overall, we're writing x items. And so to write them all will only take x over B ceiling, sort of, over the entire summation. But we have to pay 1 to start out here, here, and here. So this is the real amount of time. And I want to amortize that, essentially.

So there's two cases now. If x is greater than or equal to B squared, then we're happy. x is greater than or equal to B squared, then this term dominates. And so this is tinier. All right. Say, x is B cubed, then this is x to the 3/2-- sorry, this is the B to the 3/2. This is B squared, so this is bigger.

And so then the cost is just x over B. And we're done, because we needed to prove it's x over B times log. We had to do this to sort, but this distribution will be free. Now that says all the high levels are free. All the low levels are free. If you're less than B squared, then your less than M and you're free. It's saying, if you're bigger than B squared, you're free.

But there's going to be one level right in between where you're not really bigger or smaller. So there's one level where it's going to B to the 4/3 less than or equal to x less than or equal to B squared, strictly less than. Because you're jumping in this doubly exponential way, you might miss slightly. And so you're in between these two levels.

Why is this? Because we only know that x to the 3/2 is less than M. We don't know that x is less than M. So we have a bit of slot there.

So then at this transition point, what we do is say, OK, we can't afford to store this whole-- x to 3/2 is not less than M. So we can't afford to store all this in the cache. But we can afford to store the last block of this guy and the last block of this guy, all the down buffers we can store the last block in cache.

Why? Because there's only x to the 1/2 of them. And we know that x is less than B squared. So if x is less than B squared, x to the 1/2 is less than B, which is less than M over B because M is least B squared, by tall cache assumption. So this is the number of blocks that we can afford in cache. This is number of blocks we want. I want 1 block for each of these.

So basically then, this x to the one half term disappears for the one transition level. So this is free. Because you can afford 1 block per down buffer in cache. And so, again, we get an x over B cost.

So the distribution is basically free. The hard part is the sorting. And then the idea is that, well, OK-- now this is one push. When we do an insert, that item might get pushed many times, but basically can only get pushed once per level. So you end up taking this cost and summing it over all x.

So if you look at an insertion, amortize what you pay-- or you look at the sum over all these things. You get the sorting bound on x-- summed over x where x is growing doubly exponential. And so this becomes a geometric series, or even super geometric.

And so you get-- I guess I should look at it per element. You look at the amortized cost per element, so I get to divide by x. Because when I do a push, I push x elements. So I get to divide by x, here. And then if an element gets pushed to all levels, I have to sum over all these x's. But you do this summation and you get order-- it's dominated by the last term. Sorry, this should be N.

This is the fun part. We're taking logs, here, but conveniently this was doubly exponential growing, the x over B. So when we sum these is singly exponential. So it's a geometric series. And so we are just dominated by the last term, which is where we get log base M over B of N over B. And this is our amortized cost per insertion.

Deletions are basically the same. You just two pulls instead of pushes. And you have, sort of, the reverse of a distribution step-- in some ways, even simpler. You don't have to do this clever analysis. Question?

AUDIENCE: Can you explain, once again, how is it that you got the distributed cost of x over B? So I understand that every time, you need to pay that x to the 1/2 because you need to load. But where did the x over B come?

ERIK DEMAINE: This is essentially amortized per element. We're just paying 1 over B. Once the block has been loaded-- it's only after we insert B items that we have to load another item. So that's why it's x over B, here. Good. Question?

AUDIENCE: Which buffers do we keep sorted at all times?

ERIK DEMAINE: Which buffer--

AUDIENCE: Which buffers do we keep sorted?

ERIK DEMAINE: We're only going to keep the very bottom buffer sorted at all times. So I mean, it doesn't really matter. You don't even have to keep those sorted, because you can afford to look at all of them.

AUDIENCE: [INAUDIBLE] I think the upper levels when we're trying to figure out where it goes?

ERIK DEMAINE: What we do need-- we don't need them in sorted order. But we need to know where the max is, yeah. So I guess, maybe, maintain a linked list of the items would be one way to do it. So the sort order is in there, but they're not physically stored in sorted order. That would be a little bit too much to hope for, I think. Yeah. We do need to keep track of the max, but that's easy to do as you're inserting. Cool.

So that's priority queues. You can look at the notes for deletions.