ERIK DEMAINE: All right. Today we start a new section in advanced data structures, and it's data structures for memory hierarchy. So the idea here is while almost all data structures think about memory as a flat thing. And in the RAM model, you've got a giant array. You can access the i-th element in your giant array in constant time.

The memory hierarchy, you admit the reality of almost all computers since, I don't know, '80s, which have caches. The idea is you have your CPU connected with a very high bandwidth channel to a relatively small cache, which is connected via a relatively narrow bandwidth channel to a really big memory. And in real computers, this keeps going. You've got level 1 cache, then level 2 cache, then level 3 cache maybe, then main memory, then disk, then network, whatever. Usually the disk has a cache.

So you have this very deep hierarchy, which is typically growing exponentially in size, and also growing exponentially in slowness, in latency. So you'd like to design data structures that most of the time are working at the cache level, because that's really fast. And as little as possible, you'd like to go to the deeper levels.

So we're going to define two models for working with memory hierarchies. The first model is called the external memory model. It's also called the I/O model. It's also called the Disk Access Model, I think just so that it could be called the DAM model.

I usually call it the external memory model. And this is sort of the simplest model. It's also the most well studied, I would say. And it's going to relate closely to the other models that we're going to look at.

And the idea with the external memory model is, yeah, OK, a real computer has many levels in this hierarchy. But let's just focus on the last one and the next to last one, because typically that last one is going to be way, way slower than all the others. And so you really want to minimize how many memory transfers happen here.

So I want to count the number of memory transfers between two levels. And so this model just thinks about two levels. I'll call them cache and disk. We think of disk as infinite and slow. We think of cache as not only fast, but infinitely fast. It takes zero time to access something in

cache.

All we're thinking about is how many transfers between the left and right have to happen. So computation here is free. Access over here is free. And we just pay for this. That's the external memory model.

You can also think about running time, make sure running time is still optimal and that will guarantee this part's good. But usually the hard part is to minimize this part, minimize number of memory transfers.

Now, to make this problem really interesting, we need to add one more assumption. And this is also the case in practice, that when you request-- when you transfer an item from disk to cache, you get not just that item, but entire block of items at the same speed. So in fact, disk we're going to think of as divided into blocks. Block size here is B, capital B.

Also over here, caches are divided into blocks of size B. And we have a limited number of them. We're going to suppose we have M over B cache lines they're usually called at the cache level. And so the total size of the cache is M. M stands for memory. Sorry. This notation is slightly confusing for historical reasons, but that's the way it is.

So B is block size. M is cache size, total cache size. There's M over B blocks of size B. Disk has infinitely many blocks of size B.

There's lots of motivations for this. But basically, the main cost you're paying here is latency. You have to send a request and then the answer has to come back. Think of that as the disk is far away from the CPU. And so it takes a long time for the request to get there and for the data to get back. So if you're sending data back, you might as well send-- it's just as fast to send a lot of data as a little bit of data. In fact, typically you set this block size that the amount of time it takes for data to arrive equals the latency to balance those two costs.

With disk it's especially the case, because you have this latency of moving the disk head. Once you move the disk head, you might as well read the entire track, the entire circle of that if you're dealing with hard drives. It's just as fast once the head gets there.

OK. So that's the model. And so now we want to minimize the number of block memory transfers this happens. And the blocks make things a lot more interesting, because that means when you read an item, you really want to use all the items right around it soon, before it gets kicked out of cache. Obviously, cache is limited size, so when you bring something, you've got

to kick something out. That's the block replacement strategy.

In the external memory model, you can do whatever you want. You have full control over this cache.

So let's see. Let me write some obvious bounds. And then I'll tell you some of the main results in this model.

So number of memory transfers is what we care about. And it's going to be at most the running time on a RAM, so this is a comparison between two models. At most, every operation you do on a RAM data structure incurs one memory transfer. That would be the worst case. So that's what we want to beat.

There's also a lower bound. So this is also at least the number of cell probes divided by B. I think we've talked briefly about the cell probe model, I think in retroactive data structures.

Cell probe model is this model where all you count is-- you just count how many words did I have to read in order to do this data structure operation? So computation is free. We just count number of word reads from memory. So it's a lower bound on RAM data structures.

And in this model, we're only counting memory accesses, but where cell probes you count word numbers, here we're counting blocks. And so at best, we can improve by a factor of B.

So this is a lower bound. This is an upper bound, so there's roughly a slop of a factor of B here. We want to do closer to this bound than this bound. This is the obvious thing. Want to get down to here whenever we can.

So let me give you some examples of that, so sort of a bunch of basic results in this model. Result 0 is scanning, so if I have an array of items and I just want to, say, add them all up or do a linear scan to search for something, anything that involves accessing them in order, this will cost I guess ceiling of N over B. If the array has size N, number of blocks in that array is ceiling of N over B. And so that's how many block accesses it costs.

So the annoying part here is the ceiling. If N is smaller than B, then this is not great. I mean, you have to pay one memory read, so it's optimal. But anyway, that's scanning. Pretty boring.

Slightly more interesting, although not that much more interesting, is search trees. If you want to be able to search for an item in this external memory model-- in this external memory

model, what do you do?

B-trees, conveniently named B-trees. We want the branching factor of a B-tree to be B. Anything that's theta B will do, but exactly B is great. If you can read all of these pointers and read all of these key values, if that fits in B, then we're happy.

So yeah, typically we assume the branching factor here is like B plus 1. So in particular, if B is 1, you still have a branching factor of 2, just to make it a reasonable tree. And so now in a constant number of block reads you can read all the data you need to decide which child pointer you follow. You follow that child pointer, and so it takes log base B plus 1 of N to do a search.

This is counting memory transfers. So this is not as big an improvement. Here, we've improved by a factor of B roughly. Here we're improving by a factor of basically log B. So not as good as you might hope to do, dividing by B, but it turns out this is optimal. You need log base B plus 1 of N to do a search in comparison model.

In the same way, you need log base 2 of N time to do a search in the comparison model, in this model you need log base B of N. Let me talk a little bit about the proof of that. It's a pretty simple information theoretic argument.

So we prove this lower bound. The idea is the following. You want to figure out among N items here where your item fits. Let's say it fits right here. So you want to know it's between this element and this element.

How many bits of information is that? Well, there's I guess N plus 1 different outcomes of where your item could fit. If you allow it to exactly match, it's maybe twice that. But there's theta N options for your answer. And so if you take logs, that is log N bits of information.

And this is how you normally show that you need log N comparison. You need log N comparisons because each comparison could at most give you one bit of information, yes or no. And there's log N bits to get, so that's the lower bound in binary search.

But now we're in this model where we're not counting comparisons, even though we're in the comparison model. We're counting number of block transfers. So when I read in some data, I get B items from my array, or from something.

So when I do a block read, all I can do in the comparison model is compare my item to everything that I read, so you might as well compare all of them. But the information you learn is only log B plus 1 bits, because what you learn is among these B items, where do I fit? And in the best case, that is log of B plus 1 bits of information. It could be worse, if these guys are not well distributed among your N items.

It's at most that. And so you take the ratio and you need log N over log B plus 1 block reads. And that's log base B plus 1 of N.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Yeah. Question?

AUDIENCE: Why is that you learn log B plus 1 bits when you [INAUDIBLE]?

ERIK DEMAINE: All right. So the claim is, in the comparison model, you compare your item to everyone in here. But if your array is presorted, then all you're learning is-- I mean, information theoretically, all your learning is where your item fits among these B items. So there's order B options of where you fit, and so only log B bits of information there.

You might consider this a vague argument. You can do a more formal version of it. But I like the information theoretic perspective.

Cool. So that's the best thing you can do for search in this model. Now, you might also wonder about insertions and deletions. B-trees support insertions and deletions in log base B plus 1 of N time, but that is not optimal. We can do better.

So for example, if you want to sort, the right answer for sorting is this crazy bound. N over B, which is really good, that's linear time in this model, times log base M over B of N over B. This is a slight-- this is roughly log base M of N. The over B's don't make too big a difference.

So this is an improvement. Normally sorting takes N log N with a base 2. We're improving the base of the log to M over B instead of 2. And we're improving the overall running time by a factor of B. So this is actually a really big improvement.

We're going faster than the RAM time divided by a B, because to sort in the cell probe model just takes linear time. You read all the data. You sort it. You put it back. So we can't quite get N over B, but we get this log factor.

So this is really good. And it's something that would not result from using B-trees. This is a little funny, because in regular RAM model, if you insert everything into a balanced binary search tree, then do an in order traversal, that's an optimal sorting algorithm. It gives you N log N.

Here, it's not optimal. You throw everything into a B-tree, you get N times log base B of N. So there's two things that are bad. One is that the base of the log is slightly off. It's B versus M over B. That's not so big a deal.

The big thing is it's N over B versus N. We don't want to pay N. We want to pay N over B. There's also a matching lower bound in the comparison model, which we will not cover here. By next lecture, we'll know how to do sorting optimally.

First, I want to tell you about a bunch of different ways or different problems and results. So a funny problem, which actually does arise in practice, but it's interesting just to think about, instead of sorting-- sorting is I give you an array. I don't know what order it's in. You have to find the right order and then put it into the right order.

There's an easier problem, which is called permuting. I give you an array. I give you a permutation that I'd like to do. I just want you to move the items around.

So you don't need to do any comparisons to get here. I tell you what the order is. You just have to physically move the items. This is harder than it sounds, because when you read, you read blocks of items in the original order. And so it's hard to just do an arbitrary permutation.

And there's two obvious algorithms. One is you could sort your items. So you could go to every item in order and just write down its desired number in the new permutation, write down the permutation. Then run a sorting algorithm. That will take N over B log base M over B of N over B.

Or the other obvious algorithm is to take the first item, put it where it belongs in the array, take the second item, put it where it belongs in the array, in a new copy of the array. And that will take at most N memory transfers, each one a completely separate memory transfer.

And the claim is the optimal thing to do is the min of those two. So you just check how does N and B and M compare, do the better of the two algorithms. Again, there's a lower bound.

In a model called the indivisible model, which is assuming that each word of your data doesn't get cut into subwords. So this is assuming-- the lower bound assuming your words do not get

cut up. I believe it's still unsolved whether you could do better otherwise.

The last result in this model I wanted to talk about is called buffer trees. Buffer trees are pretty cool. They're essentially a dynamic version of sorting, which is what we lack-- what B-trees do not give you.

So they achieve the sorting bound divided by N per operation, so this is optimal. I haven't told you what the operations are. The amount is amortized.

It has to be amortized, because this bound is typically little o of 1. Think of this as 1 over B. There's a log factor. But assuming that log factor is smaller than B, which is often the case, this is in order 1 over B. It's a little o of 1 cost, which is a little fun.

But of course, every real cost is an integer. It's an integer number of block transfers. But amortized, you can get a little o of 1 bound.

And what it lets you do is delayed queries and batch updates. So I can do an insert and I can a delete. I can do many of them.

And then I can do a query against the current data structure. I want to know, I don't know, the successor of an item. But I don't get the answer right then. It's delayed.

The query will sit there. Eventually the data structure will decide to give me an answer. Or it may just never give me an answer. But then there's a new operation at the end, which is flush, which is give me all the answers to all the questions that I asked. And the flush takes this much time.

So if you do N operations and then say flush, then it doesn't take any extra time. It's going to cost one sort of pass through the entire data structure, one sorting step. Yeah. The one thing you can get for free is a zero-cost find min.

All right. So buffer tree, you can do find min instantaneously. It maintains the min all the time. So this lets you do a priority queue. You can do insert and delete min in this much time online.

But if you want to do other kinds of queries, then they're slow to answer. You get the answer much later.

So this is a dynamic version of sorting. It lets you do sorting in particular. It let's you do priority queues, other great things. We will see not quite a buffer tree but another data structure

achieving similar bounds in the next lecture.

Today's lecture is actually going to be about B-trees, which seems kind of boring, because we already know how to do B-trees. But we're going to do them in another model, which is the cache-oblivious model. So let me tell you about that model.

All right. So the cache-oblivious model is much newer. It's from 1999. And it's from MIT originally, Charles Leiserson and his students invented it back then, in the context of 6406, I believe.

And it's almost the same as this external memory model. There's one change, which is that the algorithm doesn't know what B or M is. Now, this is both awkward and cool at the same time.

So the assumption is that the computer looks like this. The algorithm has no-- and the algorithm knows that fact, but it doesn't know what the parameters of the cache are. Doesn't know how the memory is blocked.

Now, as a consequence, it cannot manually say, read this block, write this block. It can't decide which block to kick out. This is going to be more like a typical algorithm. I mean, the algorithm will look like a regular RAM algorithm.

It's going to say read this word, write this word to memory, do an addition, whatever. So algorithm looks like a RAM algorithm. But the analysis is going to be different.

So this is convenient, because algorithms-- I mean, how you implement an algorithm becomes much clearer in this model. And this is-- now we're going to assume that the cache is maintained automatically. In particular, the way that blocks are replaced becomes automatic.

So the assumption is when you read a word from memory, the entire block comes over. You just don't know how much data that is. And when something gets kicked out, the entire block gets kicked out and written to memory. But you don't get to see how that's happening, because you don't know anything about B or M.

We need to assume something about this replacement strategy. And in the cache-oblivious model, we assume that it is optimal. This is an offline notion, saying what block we're going to kick out is the one that will be used farthest in the future, so this is very hard to implement in

practice, because you need a crystal ball.

But good news is, you could use something like LRU, Least Recently Used, kick that guy out, or even a very simple algorithm, first in, first out. The most recent-- the oldest block that you brought in is the one that you kick out. These are actually very close to optimal.

They're constant competitive against or given a cache of twice the size. So this is from the early days of competitive analysis. It says if I compare offline optimal with a cache size M versus-- or let's say offline optimal with a cache size M over 2, versus LRU or FIFO, which are online algorithms on a cache of size M, then this cost will be at most I think twice this cost.

Now, the good news is, you look at all these bounds, you change M by a factor of 2, nothing changes. Assuming everything is at least some constant, and M and B are not super super close, then changing M by a factor of 2 doesn't affect these bounds. These ones don't even have M in them, so it's not a big deal.

So we're going to assume this basically for analysis. It's not really that big a deal. You could also analyze with these models. The point is they're all the same within a constant factor for the algorithms we care about.

What other fun things? If you have a good, cache-oblivious algorithm, it's good simultaneously for all B and M. That's the consequence. So whereas B-trees for example, need to know B, they need to have the right branching factor for your value of B, the cache-oblivious model, you have to have-- your data structure has to be correct, has to be optimized for all values of B and all values of M simultaneously.

So this has some nice side effects. In particular, you essentially adapt to different values of B and M. So two nice side effects of that are that if you have a multi-level memory hierarchy, each level has its own B and its own M. If you use a cache-oblivious algorithm and all of those caches are maintained automatically using one of these strategies, then you optimize the number of-- maybe I should draw a picture.

Here's a multi-level memory hierarchy. You optimize the number of memory transfers between these two levels. You optimize the number of memory transfers between these two levels, between these two levels, and between these two levels. So you optimize everything.

Each of them has some-- if you just think of this as one big level versus this level or versus all the levels to the right, then that has some B and some M. And you're minimizing the number of

memory transfers here.

Also, if B and M are changing, lots of situations for that. If you have multiple programs running at the same time, then your cache size might effectively go down if they're really running in parallel on the same computer, or pseudo-parallel.

Block size could change. If you think of a disk and your block size is the size of your track, then closer to the center, the tracks are smaller. Closer to the rim of the disk then blocks are bigger. All these things basically in the cache-oblivious model, you don't care, because it will just work.

Now, there's no nice formalization of adapting to changing values of B and M. That's kind of an open problem to make explicit what that means. But we're going to assume in the cache-oblivious models you have one fixed B, one fixed M, analyze relative to that. But of course, that analysis applies simultaneously to all B's and all M's. And so it feels like you adapt optimally, whatever that means.

In some sense, the most surprising thing about the cache-oblivious world is that it's possible. You can work in this model and get basically all of these results. So let me go over these results.

So in the cache-oblivious model, of course scans are still good. Scans didn't assume anything about block size here, except in the analysis. Scanning was just a sequential read.

In general, we can do constant number of scans. Some of them can go to left. Some of them could go to the right. Some of them could be reading. Some of them could be writing.

That's the general form of scan. But all of that will take order N over B ceiling.

OK. Search. This is what we're going to be focused on today. You can achieve order log base B plus 1 of N search, insert, and delete. Just like B-trees, but without knowing what B is. We're going to do most of that today.

What's next? Sorting. Sorting, you can do the same bound. That we'll see next lecture.

OK. Next is permuting. Here there's actually a negative result, which is that you can't do the min. It doesn't say exactly what you can do.

In particular, we can certainly permute as quickly as we can sort. And we can permute-- we

can achieve this min in a weird sense, in that we can achieve N and we can achieve the sorting bound for permutation in the cache-oblivious model, either one, but not both with min around them, because to know which one to apply, you need to know how B and M relate to N, and you don't here.

And so there's a lower bound saying you really can't achieve this bound, even up to constant factors. But it doesn't explicitly say anything more than that. So a little awkward.

Fortunately, permuting isn't that big a deal. And most of the time the sorting bound is much faster than the end bound, so not so bad. But slightly awkward. So there's some things you can't do. This was the first negative result in cache-oblivious.

Another fun negative result is actually in search trees. You might ask, what is this constant factor here in front of the log base B plus 1 of N? And for B-trees, the constant factor is essentially 1. You get 1 times log base B plus 1 of N.

In cache-oblivious world, you can prove that 1 is unattainable and the best possible constant factor is log base 2 of e, which was like 1.44. So that is the right answer for the constant. Today we'll get a constant here of 4. This is for search. But 4 has since been improved to 1.44, so a tiny separation there between cache-oblivious and external memory.

Let's see. One more thing. Continuing over here is priority queue. Cache-oblivious we can do in the optimal time 1 over B log base M over B of N over B.

Only catch with that result is it assumes something called the tall cache assumption. Let's see what I want to say here. We'll say M is omega B to the 1 plus epsilon. That's an ugly-looking omega.

Certainly M is at least B. If you have one cache line, you have M equal B. But we'd like M to be a little bit bigger than B. We'd like the cache to be-- if M was at least B squared, then the height of the cache would be at least as big as the width of the cache, hence tall cache.

But we don't even need M equal to B squared. Just M to the-- M equaling B to the 1.001 would be enough. Then you can get this result.

And there's a theorem that shows for cache-oblivious you need to assume this tall cache assumption. You cannot achieve this bound without it. But this is usually true for real caches, so life is good.

All right. So that's a quick summary of cache-oblivious results. Any questions about those?

Then we proceed into achieving search trees log base B of N performance cache-obliviously. Make it interesting, because external memory, we already know how to do it with a B-tree.

So we did both models, cache-oblivious B-tree is what remain. This was the first result in cache-oblivious data structures. I'm going to start with the static problem.

Cache-oblivious static search trees. This was actually in a MEng thesis here by Harold Prokop. So the idea is very simple. We can't use B-trees because we don't know what B is, so we'll use binary search trees. Those have served us well so far.

So let's use a balanced binary search tree. And because it's static, might as well assume it's a perfect balanced binary search tree. Let me draw one over here. I'll draw my favorite one.

So how do you search in a binary search tree? Binary search. Our algorithm is fixed. So the only flexibility we have is how we map this tree into the sequential order in memory. Memory is an array.

So for each of these nodes, we get to assign some order, some position in the array of memory. And then each of these child pointers, that changes where they point to. And now cache-obliviously, you're going to follow some root to leaf path. You don't know which one. All of them are visiting nodes in some order, which looks pretty random. In the array of memory, it's going to be some sequence like this.

We would like the number of blocks that contain those nodes to be relatively small somehow. So that's the hard part. Now, you could try level order. You could try in order, pre order, post order. All of them fail.

The right order is something called van Emde Boas order. So let's say there are N nodes. What we're going to do is carve the tree at the middle level of edges.

So we have a tree like this. We divide in the middle. Leaves us with the top part and many bottom parts.

There's going to be about square root of N nodes in each of these little triangles, roughly speaking. The number of triangles down here is going to be roughly square root of N-- I think I have written here square root of N plus 1. It's a little overestimate, because I multiply those

two numbers, it's bigger than N.

But it's roughly that. If there were root N nodes up here, then there would be square root of N plus 1 children, and square root of N plus 1 things down there. So roughly, root N different things, each of size root N.

Then what we do is recursively lay out all those triangles of size roughly squared of N, and then concatenate. It's a very simple idea. Let's apply it to this tree.

So I split in the middle level and I recursively lay out this triangle up here-- draw it as a box. Then I recursively lay out this. Then I recursively lay out this and this and this.

So up here, I mean, to lay this out, I would again split in the middle level, lay out this piece, so this guy will go first, then this one, then this one. Now I'll go over and recursively do this, so it's 4, 5, 6. Then this one, 7, 8, 9, 10, 11, 12, 13, 14, 15.

That is the order in which I store the nodes. And I claim that's a good order. For example, if B equals 3, this looks really good, because then all of these nodes are in the first block. And so I get to choose which of these four children trees I should look at. I mean, I happen to go here and then maybe here. I look at those two nodes, look at their keys to decide, oh, I should go into this branch.

But I only pay one memory transfer to look at all those nodes. So this looks like a tree tuned for B equals 3. But it turns out it's tuned for all B's simultaneously, if we do this recursively. It's hard to draw a small example where you see all the B's at once, but that's the best I can do.

OK. Let's analyze this thing. So the order of the nodes is clear. If you know what the van Emde Boas data structure is, then you know why it's called this. Otherwise, see lecture 10 or so in the future. Forget exactly where we're covering it.

So this order was not invented by van Emde Boas, but it looks very similar to something else that was invented by van Emde Boas, so we call it that. But that's the name of a guy, Peter.

Analysis. Order's clear. The algorithm is clear. We follow our root to leaf path.

What we claim is any root to leaf path and any value of B, the number of blocks visited is order log base B of N. So to do that, in the analysis we get to know what B is. The algorithm didn't know. Algorithm here doesn't know B and doesn't need to know B.

But to analyze it, to prove an order log base B of N, we need to know what B is. So while the algorithm recurses all the way down, keeps dividing these blocks into smaller and smaller pieces, we're going to look at a particular level of that recursion. I'm going to call that a level of detail. Say I call it straddling B.

So my picture is going to be the following. These little triangles are going to be size less than B. But these big triangles are size greater than or equal to B.

OK. At some level of recursion that happens, by monotonicity, I guess. I could keep going. I mean, this tree is going to have substantial height. I don't really know how tall it is, or that's sort of the central question. Draw in some pointers here.

But just look at-- if you subdivide all the big triangles until you get to these small triangles that are size less than B, then you stop recursing. Now, the algorithm, of course, will recursively lay that out. But what we know, we recursively lay things out and then we concatenate, meaning the recursive layouts remain consecutive. We never like interleave two layouts.

So however this thing is laid out, we don't really care, because at that point it's going to be stored in memory, which is this giant array. It's going to be stored in interval of size less than B. So at worst, where are the block boundaries? Maybe draw that in red.

Block boundaries are size B. We don't know exactly how they're interspersed. Let's say these are the block boundaries.

So it could be that this little subarray falls into two blocks, but at most two blocks. So the little triangles of size less than B live in at most two memory blocks. So to visit each of these little triangles, we only spend two memory reads at most.

So all that's left is to count how many of those little triangles do we need to visit on a root to leaf path? I claim that's going to be log base B of N. So I guess the first question is, what is the height of one of these little triangles?

Well, we start with a tree of height log N. Then we divide it in half and we divide it in half and divide it in half. And we take floors and ceilings, whatever.

But we repeatedly divide the height in half. And we stop whenever the number of nodes in there is less than B. So that means the height is going to be theta log B. In fact, it's going to be in the range 1/2 log B to log B, probably with a paren that way. Doesn't really matter, because

you stop as soon as you hit B.

You're dividing the height in half repeatedly. You might overshoot by at most a half. That's it. So height is log B.

So if we look along a root to leaf path, each of these things we visit, we make vertical progress log B, theta log B. And the total vertical progress we need to make is log N, and so it's log N divided by log B. So I guess 4 times log N, 2 log N over log B.

Triangles, we reach a leaf. It's total progress we need to make divided by 1/2 log B here, progress per triangle. So we get 2 times log base B of N. And then for each triangle, we have to pay 2 memory reads, so this implies at most 4 log base B of N memory transfers.

And I implicitly assumed here that B is not 1. Otherwise, need a B plus 1 here to be interesting. So long as B isn't 1, then it doesn't matter.

OK. So that's the log base B of N analysis, actually really simple. Essentially what we're doing here is binary searching on B in a kind of weird way, using divide and conquer and recursion. So we say, well, maybe B is the whole thing, then B equals N. Then we don't have to do anything. Everything fits in a block.

Otherwise, let's imagine dividing here and supposing B is root N, and then we need two accesses. But B might be smaller than that, so we keep recursing, keep going down. And because we're dividing in half each time-- sorry. We're not binary searching on B. We're binary searching on log B, which is the height of a tree with B nodes in it.

And that's exactly what we can afford to do. And conveniently, it's all we need to do, because to get log base B of N correct, you only need to get log B correct up to a constant factor.

All right. Now, that was the easy part. Now the hard part is to make this dynamic. This is where things get exciting.

This works fine for a static tree. And this is how we do search. But we can't do inserts and deletes in this world easily. But I'll show you how to do it nonetheless.

So this is what's called a cache-oblivious B-tree. We're going to do it in five steps. And the first step, unfortunately, is delayed until next lecture, so I need a black box.

It's a very useful black box. It's actually one that we've briefly talked about before, I think in the time travel. If you remember, in time travel there was one black box, which we still haven't seen but we will do finally next lecture, which was you can insert into a linked list and then given two items in the linked list know which one comes before which other one in constant time per operation.

Closely related to that is a slightly different problem called order file maintenance. Here's what it lets you do. You can store N elements, N words in specified order, so just like it's a linked list. But we don't store it as a linked list. We want to store it in an array of linear size.

So there's going to be some blank spots in this array. We've got N items, an array of size, say 2N or 1.1 times N, anything like that will suffice. The rest, we call them gaps, gaps in the array.

And the updates-- so this is physically what you have to do. You can change this set of elements by either deleting an element or inserting an element between two given elements.

So this is just like the linked list updates. I can delete somebody from a linked list. I can insert a new item in between two given items in the linked list.

But now it has to be stored in a physical array of linear size. Now, we can't do this in constant time. But what we can do is move elements around.

OK. So let me draw a picture. So maybe we have some elements here. They could be in sorted order, maybe not.

There are some gaps in between. I didn't mention it, but all gaps will have constant size. So this will be constant-sized gaps.

And we're able to do things like insert a new item right after 42. Let's say we insert 12. So 12, we could put it right here, no problem.

Now maybe we say insert right after 42 the number 6. And we're like, oh, there's no room to put 6. So I've got to move some stuff. Maybe I move 12 over here and then I put 6 here.

And then I say, OK, now insert right after 42 the number 7. And you're like, uh-oh. I could move these guys over one and then put 7 here, and so on. Obvious algorithm to do this. Takes linear time, linear number of shifts in order to do one insertion.

Claim is you can do an insertion in log squared amortized. It's pretty cool. So worst case, you'll still going to need to shift everybody. But just leaving constant-sized gaps will be enough to reduce amortized costs to log squared. And this is conjectured optimal, though no one knows how to prove that.

And that's order file maintenance. Now, we're going to assume this is a black box. Next lecture, we'll actually do it.

But I want to first show you how it lets us get cache-oblivious dynamic B-trees. Log base B of N, insert, delete, search.

OK. We're going to use an ordered file maintenance data structure to store our keys, just stick them in there. Now, this is good-- well, it's really not good for anything, although it's a starting point.

It can do updates in log squared of N-- I didn't mention this part-- in a constant number of scans. So think of these as this kind of shift. I have to scan left to right, copy 23 to the left, 42 to the left, and so on. Scans are things we know how to do fast in N over B memory transfers.

So if you're going to do log squared-- if you're going to update an interval log squared guys using constant number of scans, this will take log squared N over B memory transfers. Not quite the bound we want. We want log base B of N.

This could be larger or smaller than log base B of N. It depends how big B is. But don't worry about it for now. This is still-- at least it's poly log. Updates are kind of vaguely decent. We'll improve them later.

On the other hand, search is really bad here. I mean, we're maintaining the items-- I didn't mention it, but we will actually maintain the items in sorted order. So yeah, you could do a binary search. But binary search is really bad cache-obliviously, because most of the time you'll be visiting a new block until the very end.

So binary search-- this is a fun exercise-- is log N minus log B. We want log N divided by log B. So we can't use binary search.

We need to use van Emde Boas layouts, this thing. So we just plug these together. We take this static search tree and stick it on top of an ordered file maintenance data structure. So here

we have a van Emde Boas layout static tree on top of this thing, which is an ordered file. And we have cross-linking.

Really here we have gaps. And we don't store the gaps up here. Or do we? Yeah, sure. We do, actually. Store them all. Put it all in there.

All right. I'm going to have an actual example here. Let's do it 3 slash 7, 9 slash 16, 21, 42.

OK. Cross-links between. I won't draw all of them. Then we store all the data up here as well.

And then the internal nodes here are going to store the max of the whole subtree. So here the max is 3. Here the max is 9, 16, 42, 42, and 9. If I store the max of every subtree, then at the root. If I want to decide should I go left or should I go right, I look at the max of the left subtree and that will let me know whether I should go left or right.

So now I can do binary search. I do a search up here. I want to do a search, I search the tree. This thing is laid out just like this. And so we pay log base B of N to search up here, plus 1, if you want.

And then we get to a key and we're done. That's all we wanted to know. So search is now good.

Updates, two problems. One is that updating just this bottom part is slow. Second problem is if I change the bottom thing, now I have to change the top thing. But let's look at that part first. It's not so bad to update-- I mean, this structure stays the same, unless N doubles. Then we rebuild the entire data structure.

But basically, this thing will stay the same. It's just data is moving around in this array. And we have to update which values are stored in this structure. The layout of the structure, the order of the nodes, that all stays fixed. It's just the numbers that are being written in here that change. That's the cool idea.

So that's step 2. Let's go to step 3. I actually kind of wanted that exact picture. That's OK.

I need to specify how updates are done. So let's say I want to do an insertion. If I want to insert a new key, I search for the key. Maybe it fits between 7 and 9. Then I do the insertion in the bottom structure in the ordered file. So we want to insert 8.

These guys move around. Then I have to propagate that information up here. And the key

thing I want to specify is an update does a search. Then it updates the ordered file. That's going to take log squared N over B.

And then we want to propagate changes into the tree in post-order, post-order traversal. That's the one thing I wanted to specify, because it actually matters in what order I update the item.

So for example, suppose I move 9 to the right and then I insert 8 here. So then this thing-- let's do it, what changes in that situation. 9 will change to an 8 here, which means this value has to change to an 8 and this value needs to change to an 8.

And this one I can't quite update yet, because I need to do the other tree. So here I change this value to a 9. This one doesn't change, so this is OK. This is OK. This is OK. Just taking the maxes as I propagate up.

So the red things are the nodes that I touch and compute the maxes of. And the order I did it was a post-order traversal. I need to do a post-order traversal in order to compute maxes. I need to know what the max of this subtree is and this subtree before I can re-compute the max of the root. So it's really the obvious algorithm.

Now, the interesting part is the analysis. So searches we've already analyzed. We don't need to do anything there. That's log base B of N.

But updates are the interesting thing to analyze. And again we're going to look at the level of detail that straddles B, just like before. So let me draw a good picture. I'm going to draw it slightly differently from how I drew it last time.

OK. What I'm drawing here is actually the bottom of the tree. So a real tree is like this. And I'm looking at the bottom, where the leaves are, because I care about where I connect to this ordered file. So there's the ordered file at the bottom.

And I'm refining each triangle recursively until they're size less than B. They're going to be somewhere between root B and B. These big triangles are bigger than B. OK. So that's where I want to look at things.

Now, when we update in the ordered file, we have this theorem. It says you move elements in an interval. So what changes is an interval. The interval has size log squared N amortized.

So we are updating some interval here, maybe something like this. And so all of these items change, let's say. Now, this is not going to be too big. It's going to be log square N amortized.

And all of the ancestors of those nodes have to be updated, potentially. So all these guys are going to get updated. Some of this tree, this path, all of this stuff is going to get updated, some of this stuff, the path up to here. The path up to the root gets updated, in general, and all these nodes get touched.

Now, I claim that's cheap, no more expensive than updating down here. Let's prove that.

OK. First issue is, in what order do I visit the nodes here? I want to think about this post-order. Maybe I'll use another color to draw that.

So let's say we start by updating this guy. We update this value. Then we go-- I'm doing a post-order traversal, this tree.

Once I've completely finished here, I can go up here, maybe go up here, visit over down this way, fix all the maxes here. Then I go up, fix all the maxes here, fix all the maxes, the maxes. When I come up, then I can fix the maxes up here, and so on.

OK. Think about that order. If you just look at the big picture, which is which triangles are we visiting, what we do is we visit this triangle, then this one, then this triangle, then this one, then this triangle, then this one. I guess this is the better picture. This triangle, left one, this triangle, next child, this parent, next child, this parent, next child.

And when we're done, we go over to the next big triangle. Within a big triangle, we alternate between two small triangles. As long as we have four blocks of cache, this will be good, because there's at most two blocks for this little triangle, at most two blocks for this little triangle. So jumping back and forth between these is basically free.

So we assume-- white chalk. We assume that we have at least four blocks of cache. Post-order traversal alternates between two little triangles. So this part is free.

The only issue is, how many little triangles do I need to load overall? Sorry. I should say I'm just looking at the little triangles in the bottom two levels. So this is in the bottom two levels. Sorry. I forgot to mention that.

As soon as you go outside the bottom big triangle, let's not think about it yet. Worry about that

later.

So just looking at these bottom two levels that I drew, we're just doing this alternation. And so the only issue is, how many little triangles do I have to load? So there could be these guys. These guys are good, because I'm touching all these nodes. The number of nodes here was bigger than B, so if I take this divided by B in the ceiling, the ceiling doesn't hurt me, because the overall size is bigger than B.

These guys I have to be a little bit careful. Here the ceiling might hurt me, because I only read some of these nodes. It may be much less than B. It could be as low as root B. I divide by B, take the ceiling, I get 1, so I have to pay 1 at the beginning, 1 at the end. But in between, I get to divide by B.

So the number of little triangles in the bottom two levels that I visit is going to be 1 plus log squared N over B amortized. The size of the interval is log squared N amortized, so this would be if I'm perfect. But I get a plus 1 because of various ceilings.

And also, the number of blocks is the same, because each of these little triangles fits inside at most two blocks. So it's going to be order 1 plus log squared N over B.

OK. So I can touch all those in this much time, because I assume that I can alternate between two of them quickly. And once I finish one of these leaf blocks, I never touch it again, so I visit each block essentially only once as long as my cache is this big.

Cool. So that deals with the bottom two levels. I claim the bottom two levels only pay this. And we were already paying that to update the ordered file, so it's no worse.

Now, there's the levels above this one that we can't see. I claim that's also OK. Sorry. Also need number of memory transfers is equal to that.

OK. One more part of the argument. Levels above the bottom two. Nice thing about the levels above the bottom two is there aren't too many triangles. All right. Well, there aren't too many nodes involved. Let's see.

So down here, there were like log squared N nodes. There was a lot of nodes. Once I go up two levels, how many of these nodes can there be, these roots?

Well, it was log squared N at the bottom. And then I basically shaved off a factor of B here, so

the number of these roots is at most log squared N over B. That's small.

For those nodes, I could afford to do an entire memory transfer for every single node. So the number of nodes that need updating-- these are ancestors-- is order log squared N over B. That's the number of leaves down there.

At some point, you reach the common ancestor of those leaves. That tree-- so here, draw a big picture. Here's the bottom two layers. We're shaving that off.

Up here, the number of nodes here, these roots that we're dealing with, that's the log squared N over B. So if we build a tree on those nodes, total number of nodes in here is only log squared N over B times 2, because it's a binary tree. But then there's also these nodes up here. This is potentially log N nodes.

So number of nodes that need updating is at most this plus log N. So what's going on here is we first analyzed the bottom two layers. That was OK because of this ordering thing, that we're only dealing with two triangles at once and therefore we pay this size divided by B, log squared N over B.

Then there's the common ancestors of those nodes. That forms a little binary tree of size log squared N over B. For these nodes, this triangle up here, we can afford to pay one memory transfer for every node, so we don't care that we're efficient up here, which is-- I mean, well, let's not worry about whether we are. It doesn't matter.

So for these nodes, we pay at most one memory transfer each. And so we pay at most log squared N over B for them. That's fine, because we already paid that much.

And finally, we have to pay for these nodes up here, which are from the common ancestors of these guys up to the root. That has length at most log N. Could be smaller if this was a big interval.

So for these guys, we pay order log base B of N. Why? Because this is a root to node path. And this is the static analysis. We already know that following a root to node path we pay at most log base B of N.

So we pay log base B of N plus log squared N over B for this, plus log squared N over B for this, plus log squared N over B for the ordered file. So overall running time for an update is log base B of N plus log squared N over B.

We actually paid log base B of N twice, once to do the initial search. I said an update first has to search for the key. Then it updates the order file. That's where we've paid the first log squared N over B. And then to propagate all the changes up the tree, we pay log squared N over B for this part, and then another log base B of N for this part.

OK. That's updates with a bad bound. It turns out we're almost there. So at this point we have searches optimal, because searches are just searching through the van Emde Boas thing. That's log base B of N.

Updates, not quite optimal. I mean not quite matching a B-tree, I should say, because it's not an optimal bound we're aiming for. We're just aiming for B-trees. We'd want to get this bound, same as searching. But we have this extra term, which if B is small, for example if B is log N, this term will be bigger than this one. The breakpoint is when B equals log n times log log N.

So for small B, we're not happy. And we want to be happy for all B's simultaneously. So I want to fix this.

We're going to fix it using a general technique, great technique, called indirection. So this is step 5, indirection. The idea is simple.

We want to use the previous data structure on N over log N items. And, well, we need to store N items, so where do we put them? On the bottom. So I'm going to have theta log N nodes down here, theta log N nodes next to them-- or items, I should say. Theta log N.

OK. And then one of these items, let's say the min item, gets stored in this tree. The min item over here gets stored in this tree. The min item here gets stored up here.

OK. So this item only-- the data structure we just developed, I only store N over log N of the items. The rest that fit in between those items are stored down here.

Now I use standard B-tree tricks, actually. If I want to do an insertion, how do I do a search? I follow a root to leaf path. And then I do a linear scan of this entire structure.

How much does it cost to do a linear scan of log N? Only log N divided by B. This is smaller than log base B of N, so the hard part of search is the log base B of N.

OK. How do I do an update? Well, first, I search for the guy. I find the block that it fits into and just insert into that block. Done.

As long as this block doesn't get too big, how do I insert into a block? Rewrite the entire block. It's a scan. It only takes log N over B, so it's basically free to rewrite this entire block every time as long as it has size at most log N. If it gets too full, then I split it into two parts that are half-full.

So in general, each of these blocks will maintain the size to be between 1/4 log N and log N. So if it gets bigger than log N, I split into two guys of 1/2 size. If it gets smaller than 1/4 log N, then I'll merge it with one of its neighbors and then possibly have to re-split it. Or you can think of it as stealing neighbors from your siblings. That's what you do in B-trees.

But with a constant number of merges and splits, you can restore the property that these guys have sizes very close to 1/2 log N. And so if they end up getting full, that means you inserted 1/2 log N items. If they ended up getting empty, that means you deleted 1/4 log N items. And so in both cases, you have log N items to charge to.

Whenever you do an update up here-- so we update up here every time we split or merge down here, because this guy has to keep track of the blocks, not the items in the blocks. So we effectively slow down the updates in the top structure by a factor log N, which is great, because we take this update bound and we get to divide it by log N, which removes this square.

So we end up with-- so we used to have log base B of N plus log squared N over B. We get to divide that by log N, plus we have to do this part at the bottom. Plus we're doing log N over B at the bottom.

So basically, this log cancels with this square, so this is log N over B as well. This thing gets even smaller, so we end up being able to do updates in log N over B. Not really though, because at the beginning, we have to do a search to figure out where to do the update. If you knew where you had to update, you could update a little faster. You could update in log N divided by B. If you don't know where to update, you've got to spend log base B of N to search initially.

But that's the fun part. We can now update this data structure fast, just with this one layer of indirection with log N chunks at the bottom, speeding this data structure up by a factor of log N for updates. Searches don't take any longer. You just pay this extra term, which was already much smaller than what we were paying before.

Phew. That's cache-oblivious B-trees, other than ordered file maintenance, which we'll do next class.