ERIK DEMAINE: All right. Welcome back to 6046. Today we continue our theme of data structures but this time, instead of doing a fancy cool data structure, we're going to look at fancy cool analysis techniques for data structures. And these are useful for tons of different data structures, especially in the context when you're using a data structure to implement an algorithm.

For example, in Dijkstra, when you learn Dijkstra's algorithm, you had lots of different heap structures you could use for the priority queue in Dijkstra, and they gave different running times with Dijkstra. But the key thing in that context is that you cared about the total running time of the algorithm, less than you cared about the individual running time of each operation.

That's what amortization is about. It's, I guess, a technique from financial analysis, but we've appropriated it in computer science as an analysis technique to say, well, let's not worry about every single operation worst case cost, let's just worry about the total operation, the sum of all the operations cost. That's the whole idea of amortization, but there's a lot of different ways to do it. We're going to cover four different methods for doing it, and three-ish examples of doing it today.

You've seen some essentially in recitation last time, and you've seen a little bit in 6006, so let me first remind you of the example of table doubling from 6006. This came up in the context of hash tables. As you may recall, if you store n items in a hash table of size m-- there are m slots in the table, let's say, using chaining-- hashing with chaining-- then we got an expected cost constant plus the load factor, size of the table divided by the number of items.

So we wanted to get constant expected, and so we wanted this to always be, at most, a constant. I guess we could handle a larger table size, although then we are unhappy about our space, but we definitely want m to be at least around n so that this works out to order one. And the solution for doing that was table doubling. Whenever the table is too big, double it-- or sorry-- whenever the table is too small and we have too many items, double the size of the table.

If n-- n is the thing that we can't control. That's the number of items somebody is inserting into the table. If n grows to the value to match m, then double m. So m prime equals 2m, and to double the table size, you have to allocate a new array of double the size and copy over all the items, and that involves hashing. But overall this will take order, size of the table, work.

Doesn't matter whether I'm using m or m prime here, because they're within a constant factor of each other, and that's bad. Linear time to do an insertion is clearly bad. This is all during one insertion operation that this would happen, but overall it's not going to be bad, because you only double log n times.

And if you look at the total cost-- so maybe you think, oh, is it log n per operation, but it's not so bad because total cost for n insertions starting from an empty structure is something like 2 to the 0-- this is a big theta outside-- 2 to the 1, 2 to the 2. If we're only doing insertions, this is great. 2 to the log n. This is a geometric series and so this is order n. Theta head I guess.

So to do n insertions, cost theta n, so we'd like to say the amortized cost per operation is constant, because we did n operations. Total cost was n, so sort of on average per operation, that was the only constant. So this is the sense in which hash tables are constant, expected, amortized. And we'll get back to hashing in a future lecture, probably I think lecture 8, but for now we're just going to think about this as a general thing where you need table doubling, then this gives you a fast way to insert into a table. Later we'll think about deleting from a table and keeping the space not too big, but that's a starting point.

This is an example of a general technique called the aggregate method, which is probably the weakest method for doing amortization but maybe the most intuitive one. So the aggregate method says, well, we do some sequence of operations. Let's say, in general, there are k operations. Measure the total cost of those operations, divide by k, that's the amortized cost per operation.

You can think of this as a definition, but it's not actually going to be our definition of amortized cost. We're going to use a more flexible definition, but for simple examples like this, it's a fine definition, and it gives you what you want. When your sequence of operations is very clear, like here, there's only one thing you can do at each step, which is insert-- that's my definition of the problem-- then great, we get a very simple sum.

As soon as you mix inserts and deletes, the sum is not so clear. But in some situations, the sum is really clean, so you just compute the sum, divide by a number of operations, you get a

cost, and that could be the amortized cost. And that's the aggregate method, works great for simple sums.

Here's another example where it-- no, sorry. Let me now give you the general definition of amortized bounds, which becomes important once you're dealing with different types of operations. I want to say an insert costs one bound amortized and maybe a delete costs some other bound. So what you get to do is assign a cost for each operation. I should call it an amortized cost, such that you preserve the sum of those costs.

So what I mean is that if I look at the sum over all operations of the amortized cost of that operation, and I compare that with the sum of all the actual costs of the operations, the amortize should always be bigger, because I always want an upper bound on my actual cost. So if I can prove that the amortized costs are, at most, say, constant per operation, then I get that the sum of the actual cost is, at most, constant per operation. I don't learn anything about the individual costs, but I learn about the total cost.

And in the context of an algorithm like Dijkstra's algorithm, you only care about the total cost, because you don't care about the shortest paths at time t, you only care about the shortest paths when the algorithm is completely finished. So in a lot of situations, maybe not a real-time system, but almost everything else, you just care about the sum of the costs. As long as that's small, you can afford the occasional expensive operation. So this is a more flexible definition.

One option would be to assign the average cost to each operation, but we have a whole bunch more operations. We could say inserts cost more than deletes or things like that. In fact, let me do such an example. A couple weeks ago, you learned about 2-3 trees. This would work for any structure though.

So I claim I'm going to look at three operations on 2-3 trees. One is create an empty tree, so I need to think about how we're getting started in amortization. Let's say you always start with an empty tree. It takes constant time to make one. I pay log n time-- I'm going to tweak that a little bit-- for an insertion, and I pay 0 time per delete in an amortized sense. You can write big O of 0 if you like. Same thing.

So deletion you can think of as a free operation. Why? This is a bit counter-intuitive because, of course, in reality the actual cost of a deletion is going to be log n. Yeah.

AUDIENCE: You can never delete more elements than you've already inserted.

ERIK DEMAINE: You can never delete more elements than you've already inserted. Good.

AUDIENCE: Can you cap the cost of [INAUDIBLE]

ERIK DEMAINE: Yeah. So I can bound the deletion cost by the insertion cost, and in the context of just the aggregate method, you could look at the total cost of all the operations. I guess we're not exactly dividing here, but if we look at the total cost, let's say that we do c creations, i insertions and d deletions, then the total cost becomes c plus i times log n plus d times the log n.

And the point is d is less than or equal to i, because you can never delete an item that wasn't already inserted if you're starting from an empty structure. And so this is i plus d times log n, but that's just, at most, twice i times log n, so we get c plus i log n. And so we can think of that as having a d times 0, 0 cost per deletion.

So this is the sum of the actual costs over here. This is the sum of the amortized costs, where we say 0 for the deletion, and we just showed that this is an upper bound on that, so we're happy. Now, there's a slight catch here, and that's why I wrote star on every n, which is not every operation has the same cost, right? When you start from an empty structure, insertion cost constant time, because n is 0. When n is a constant, insertion is constant time. When n grows to n, it costs log n time. At different times, n is a different value, and n I'm going to use to mean the current size of the structure.

For this argument to work at the moment, I need that n is not the current value, because this is kind of charging work. Some insertions are for large structures, some are for small structures, some deletions are for small, some are for large. Gets confusing to think about. We will fix that in a moment but, for now, I'm just going to define n star to be the maximum size over all time. OK, if we just define it that way, then this is true. That will let me pay for any deletion, but we'll remove that star later on once we get better analysis methods, but so far so good.

Two very simple examples-- table doubling, 2-3 trees with free deletion. Of course, that would work for any structure with logarithmic insertion and deletion, but we're going to be using 2-3 trees in a more-- analyzing them in a more interesting way later on.

So let's go to the next method, which is the accounting method. It's like the bank teller's analysis, if you will. These are all just different ways to compute these sums or to think about the sums, and usually one method is a lot easier, either for you personally or for each

problem, more typically. Each problem usually one or more of these methods is going to be more intuitive than the others. They're all kind of equivalent, but it's good to have them all in your mind so you can just think about the problem in different ways.

So with the accounting method, what we're going to do is define a bank account and an operation can store credit in that bank account. Credits maybe not the best word, because you're not allowed for the bank account to go negative. The bank account must always be non-negative balance, because otherwise your summations won't work out.

So when you store credit in the bank account, you pay for it. It's as if you're consuming time now in order to pay for it in the future. And think of operations costing money, so whenever I do a deletion, I spend actual time, log n time, but if I had log n dollars in the bank, and I could pull those out of the bank, I can use those dollars to pay for the work, and then the deletion itself becomes free in an amortized sense.

So this is, on the one hand, operation-- and when I do an insertion, I'm going to physically take some coins out of myself. That will cost something in the sense that the amortized cost of insertion goes up in order to put those coins in the bank, but then I'll be able to use them for deletion. So this is what insertion is going to do.

I can store credit in the bank, and then separately we allow an operation to take coins out of the bank, and you can pay for time using the credit that's been stored in the bank. As long as the bank balance remains non-negative at all times, this will be good. The bank balance is a sort of unused time. We're paying for it to store things in there. If we don't use it, well, we just have an upper bound on time. As long as we go non-negative, then the summation will always be in the right direction. This inequality will hold.

Let's do an example. Well, maybe this is a first example. So when I do an insertion, I can put, let's say, one coin of value log n star into the bank, and so the total cost of that insertion, I pay log n star real cost in order to do the insertion, then I also pay log n star for those coins to put them in the bank.

When I do a deletion, the real cost is log n star, but I'm going to extract out of it log n star coins, and so the total cost is actually free-- the total amortized cost is free-- and the reason that works, the reason the balance is always non-negative, is because for every deletion there was an insertion before it. So that's maybe a less intuitive way to think about this problem, but

you could think about it that way.

More generally-- so what we'd like to say is that we only put log n without the star, the current value of n per insert and a 0 per delete amortized. So we'd like to say, OK, let me put one coin worth log n for each insertion, and when I delete, I consume the coin. And, in general, the formula here is that the amortized cost of an operation is the actual cost plus the deposits minus the withdrawals.

OK. So insertion, we just double the cost, because we pay log n to the real thing, we pay log n to store the coin. That's the plus deposit part, so insertion remains log n, and then deletion, we pay log n to do the deletion, but then we subtract off the coin of value log n, so that hopefully works out to zero 0. But, again, we have this issue that coins actually have different amounts, depending on what the current value of n was.

You can actually get this to work if you say, well, there are coins of varying values here, and I think the invariant is if you have a current structure of size n, you will have one coin of size log 1, log 2, log 3, log 4, up to log n. Each coin corresponds to the item that made n that value. And so when you delete an item at size n, you'll be removing the log nth coin, the coin of value log n. So you can actually get this to work if you're careful.

I guess the invariant is one coin of value log i for i equals 1 to n, and you can check that invariant holds. When I do a new insertion, I increase n by 1 and I make a new coin of log that value. When I do a deletion, I'm going to remove that last coin of log n. So this does work out. So we got rid of the end star.

OK, let's use this same method to analyze table doubling. We already know why table doubling works, but good to think of it from different perspectives. And it's particularly fun to think of the coins as being physical objects in the data structure. I always thought it would fun to put this in a programming language, but I don't think there is a programming language that has coins in it in this sense yet. Maybe you can fix that.

So let's go back to table doubling. Let's say when we insert an item into a table, and here I'm just going to do insertions. We'll worry about deletions in a moment. Whenever I do an insertion, I'm going to put a coin on that item, and the value of the coin is going to be a constant. I going to give the constant a name so we can be a little more precise in a moment-- c.

So here's kind of the typical-- well, here's an array. We start with an array of size 1, and we insert a single item here, and we put a coin on it. Maybe I'll draw the coin in a color, which I've lost. Here. So I insert some item x, and I put a coin on that item. When I do the next insertion, let's say I have to double the table to size 2. I'm going to use up that coin, so erase it, put a new coin on the item that I just put down. Call it y.

In general-- so the next time I double, which is immediately, I'm going to go to size 4. I erase this coin, then I put a coin here. When I insert item, of course, letter after z is w. Then I put another coin when I have to double again, so here I'm going to use these coins to charge for the doubling, and then in the next round, I'm going to be inserting here, here, and here, and I'll be putting a coin here, here, here, and here.

In general, you start to see the pattern-- so I used up these guys-- that by the time I have to double again, half of the items have coins, the other half don't, because I already used them. You have to be careful not to use a coin twice, because you only get to use it once.

You can't divide money into double money unless you're doing stocks, I guess. As soon as I get to a place where the array is completely full when n equals m, the last half of the items will have coins. I'm going to use them in order to pay for the doubling, so the number of coins here will be n over 2. So this is why I wanted to make this constant a little explicit, because it has to be bigger than 2 in some sense. However much work-- let's say it takes a times n work in order to do doubling, then this constant should be something like two times a, because I need to do the work to double, but I only have n over 2 coins to pay for it. I don't get coins over here.

So when we double, the last n over 2 items have coins, and so the amortized cost of the doubling operation is going to be the real cost, which is sum theta n minus the number of coins I can remove and their value. So it's going to be minus c times n over 2 and, the point is, this is 0 if we set c large. It only has to be a constant. It needs to be bigger than 2 times that constant.

And usually when you're working with coins, you want to make the constants explicit just to make sure there's no circular dependence on constants, make sure there is a valid choice of c that annihilates whatever cost you want to get rid of. So this is the accounting method view of table doubling.

Any questions so far? So far so good. Pretty simple example. Let's get to more interesting

examples. You also think about the amortized cost of an insert. It costs constant real time. Actual cost is constant. You have to also deposit one coin, which costs constant time so the amortized cost of the insert is still constant. So that's good.

Still we don't know how to deal with deletions, but let me give you a kind of reverse perspective on the accounting method. It's, again, equivalent in a certain sense, but in another sense may be more intuitive some of the time for some people. It's actually not in the textbook, but it's the one I use the most so I figure it's worth teaching.

It's called the charging method. It's also a little bit more time travel-y, if you will, so if you like time travel, this method is for you, or maybe a more pessimistic view is blaming the past for your mistakes. So what we're going to do is allow-- there's no bank balance anymore, although it's essentially there. We're going to allow operations to charge some of their cost retroactively to the past, not the future.

I actually have a data structures paper which proves that while time travel to the past is plausible, time travel to the future is not computationally. So you're not allowed to time travel to the future, only allowed to go to the past, and say, hey, give me $5. But you've got to be a little bit conservative in how you do it. You can't just keep charging the same operation a million times, because then the cost of that operation is going up. At the end of the day, every operation had to have paid for its total charge.

So there's the actual cost, which it starts with, and then there's whatever it's being charged by the future. So from an analysis perspective, you're thinking about the future. What could potentially charge me? Again, you can define the amortized cost of an operation is going to be the actual cost minus the total charge to the past.

So when we charge to the past, we get free dollars in the present, but we have to pay for whatever the future is going to do. So we have to imagine how many times could I get charged in the future? I'm going to have to pay for that now in a consistent time line. You will have to have paid for things that come in the future.

So let's do an example. Actually it sounds crazy and weird, but I actually find this a lot more intuitive to think about even these very examples. Let's start with table doubling. So we have this kind of picture already. It's going to be pretty much the same. After I've doubled the table, my array is half full and, again, insertion only, although we'll insertion and deletion in the moment. In order to get from half full to completely full, I have to do n over 2 insertions.

It's looking very similar, but what I'm going to say is that when I double the array next time, I'm going to charge that doubling to those operations. In general, you can actually say this quite concisely-- whenever I do a doubling operation, I'm going to charge it to all the insertions since the last doubling. That's a very clear set of items. Doublings happen, and then they don't happen for a while, just all those insertions that happened since the last doubling charged to them.

And how many are there? Well, as we've argued, there are n over 2 of them, and the cost of-- in order to make this doubling free, I need to charge theta n. So this doubling cost theta n, but there's n over things to charge to. I'm going to uniformly distribute my charge to them, which means I'm charging a constant amount to each. And the key fact here is that I only charge an insert once. Because of this since clause, I never will charge an item twice as long as I'm only inserting for now.

If you look over all time, you will only charge an insert once. That's good, because the inserts have to pay for their total charge in the future. There's only one charge, and it's only a constant amount, then amortized cost of insert is still constant, amortized cost of doubling is 0, because we charged the entire cost to the past. So same example, but slightly different perspective.

Let's do a more interesting example-- inserts and deletes in a table. Let' say I want to maintain that the size of the table is always within a constant factor of the number of items currently in the table. If I just want an upper bound, then I only need to double, but if I want also a lower bound-- if I don't want the table to be too empty, then I need to add table halving. So what I'm going to do is when the table is 100% full, I double its size, when the table is 50% full, should I halve it in size? Would that work? No, because--

AUDIENCE: [INAUDIBLE] have to have it inserted in place of linear [INAUDIBLE].

ERIK DEMAINE: Right. I can basically do insert, delete, insert, delete, insert, delete, and every single operation costs linear time, because maybe I'm a little bit less than half full-- sorry, yeah, if I'm a little bit less than half full, then I'm going to shrink the array into half. Get rid of this part, then if I immediately insert, it becomes 100% full again. I have to double in size, and then if I delete, it becomes less than half full, and I have to halve in size.

Every operation would cost linear time, so amortized cost is linear time. That's not good. So

what I'll do is just separate those constants a little bit. When I'm 100% full, I will double. That seems pretty clear, but let's say when I'm a quarter full, then I will halve. Any value less than 50 would work here, but-- just halve, like that.

This will actually work. This will be constant amortized per operation, but it's-- especially the initial analysis we did of table doubling isn't going to work here, because it's complicated. The thing's going to shrink and grow over time. Just summing that is not easy. It depends on the sequence of operations, but with charging and also with coins, we could do it in a pretty clean way. I'm going to do it with charging.

So this particular choice of constants is nice, because when I double a full array, it's half full, and also when I have an array that's a quarter full, like this, and then I divide it-- and then I shrink it-- I get rid of this part, it's also half full. So whenever I do a double or a halve, the new array is half full, 50%. That's nice. That's nice, because 50% is far away from both 25% and 100%.

So our nice state is right after a doubling or a halve, then we know that our structure is 50%. In order to get to an under-flowing state where we have to halve, I have to delete at least a quarter of the items, a quarter of m. In order to get to overflowing where I have to double, I have to insert at least m over 2 items. Either way, a constant fraction times m, that's what I'm going to charge to.

Now, to be clear, when I'm 50% full, I might insert, delete, insert, delete, many different inserts and deletes. At some point, one of these two things is going to happen though. In order to get here, I have to do at least m over 4 deletions. I might also do more insertions and deletions, but I have to do at least that many, and those are the ones I'm going to charge to.

So I'm going to charge a halving operation to the at least m over 4 deletions since the last resize of either type, doubling or halving. And I'm going to charge the doubling to the at least m over 2 insertions since the last resize. OK, and that's it.

Because the halving costs theta m time, doubling costs theta m time, I have theta m operations to charge to, so I'm only charging constant for each of the operations. And because of this since last resize clause, it's clear that you're never charging an operation more than once, because you can divide time by when the resizes happen, grows or shrinks, halves or doubles. And each resize is only charging to the past a window of time.

So it's like you have epics of time, you separate them, you only charge within your epic. OK, so that's cool. So you only get a constant number of charges per item of a constant amount, therefore insertions and deletions are constant amortized. Halving and doubling is free amortized. Clear? This is where amortization starts to get interesting.

You can also think of this example in terms of coins, but with putting coins on the items, but then you have to think about the invariance of where the coins are, which I find to be more work. We actually had to do it up here. I was claiming the last half of the items had coins. You have to prove that really.

With this method, you don't. I mean, what you have to prove is that there are enough things to charge to. We had to prove here that there were n over 2 items to charge to. Kind of the same thing, but it was very clear that you weren't charging to the same thing more than once. You were never trying to use a coin that wasn't there because of the since clause. To each their own. I think either way would work.

I think I will skip this example, but I'll just mention it. So for 2-3 trees, we said deletions were free, and we did that with the coin invariant, that there was one coin of size log i for each i. You could instead say, when I delete an item, I'm going to charge it to the insert that made n this current value, because that insert paid log n the actual cost, so it can afford to pay another log n to pay for the deletion of some other item, the one we're currently deleting. And that works, that you don't double charge to an insert, because you're decreasing n right now. So for n to get up to that value again, you would have had to do another insert. So same thing, slightly different perspective.

Let's go to something even more interesting and in some sense more powerful, the last method on the list, which is potential method. This is a good exercise in how many ways can you skin a cat? So potential method, I like to call it defining karma in a formal way, is more like the counting strategy. We're going to think about there being a bank account with some balance, but we're going to define that balance as a function of the data structure state.

So that's called the potential function, but you can think of it as a bank balance. You can think of it as kinetic potential, I guess. Potential energy. Just like the bank account, we want this function to always be non-negative. We'll also make it an integer. That would be convenient.

The potential function is basically trying to measure how bad is the data structure right now? It's, again, like saving up for a rainy day. We want that whenever we have to do an expensive

operation, like a double or halve, that this potential has grown large enough that we can charge that cost to a decrease in the potential. So it's like this is storing up energy, and whenever we have some free time, we'll give some of that time to the potential function.

It's just like the accounting method, in a certain sense, but we're defining things differently. Over here, we explicitly said, hey look, I'm going to store some credit right now. So we were basically specifying the delta, and here we're saying I'm going to consume some credit right now. Over here, we're going to define this magical function of the current state. From that you can compute the deltas, but also from here you can integrate and compute the potential function.

So they're interchangeable, but usually it's easier to think about one perspective or the other. Really often, you can just look at what's going on with the data structure and say, hey, you know, this aspect of the data structure makes it bad, makes costly operations, and you can just define the potential function, then just check that it works. But it's a little bit of black magic to come up with these functions, so you depends how you like to think about things.

So, as before, we can define an amortized cost. It's going to be the actual cost plus the change in the potential. So change of potential is just the potential after the operation minus the potential before the operation. I highlight that, and it's kind of obvious from the way we set things up, but what I care about is the sum of the amortized costs. I care about that, because it's supposed to be an upper bound on the sum of the actual costs.

And if you just look at what that sum is, on the right-hand side I have amortized cost plus the fee after the operation minus the fee before the operation. If I add all those up, this part telescopes or you get cancellation from each term with the previous term. The sum of the amortized costs is equal to the sum of the actual costs plus phi at the end minus phi at the beginning.

So a slight catch with the potential method. When you define things this way, you also have to pay for phi at the beginning, because we want the actual cost to be, at most, amortized cost. So we need to take this apart and put it over here so it's, at most, some of amortized cost plus phi of the beginning. This part becomes negative, so we usually just ignore it. It can only help us.

So when you define a potential function, you'd really like it to be 0 at the beginning. It's funny, but you pay phi of the beginning state at the beginning of time, and when you've done 0

operations, you really like the cost to be 0, and you don't want to have to have stored stuff in the bank, so this should be a-- constant would probably be OK, or whatever the cost of your first operation is but should be constant or 0.

Usually we do this by saying, look, let's start with an empty structure and work from there. Usually phi of an empty structure is 0, and all is well. So when you're defining things with potential function, you have to be careful about your initial state. You have to make sure it's non-negative just like you did over here, but you didn't have to worry about this part over there.

All this infrastructure, what's it good for? Let's do some examples. These are going to be the most interesting examples. A kind of classic example of amortization is incrementing a binary counter. So when you have some binary value like this one and you increment it, many bits change, but only a constant number are going to change in an amortized sense. If I start with a 0 vector, 0 bit vector, and I increment-- well, the very first increment costs 1, the next increment costs 2, the next increment costs 1, next increment costs 3, then 1, then 2, then 1, then 4, then it's a fractal.

But instead of thinking about that fractal and working hard to prove that summation is linear for an operation, let's use the potential method. And the intuition here is actually pretty easy, because an increment has a very clear cost. It's just the number of trailing 1s plus 1. That's what it is in actual cost. We'd like that to be constant so, intuitively, what is making an increment bad? If you had to name one thing? If I just look at a configuration, is this bad? Is this bad? How bad is the configuration? Yeah.

AUDIENCE: The more trailing ones you have, the worse the state is?

ERIK DEMAINE: The more trailing ones, the worse the state is. So that's one natural definition. Turns out, it won't work. Let's see why. I think here's an example. So near the end of our increment stage, we have a whole bunch of 1s but no trailing 1s, number of trailing 1s is 0. If I do a single increment, now the number of trailing 1s is n, so if you look at the amortized cost, it's the actual cost plus the change in phi and so I actually pay n for that operation in the amortized sense, and that's no good. We only want to pay constant, but it's on the right track.

So number of trailing 1, it is the natural thing to try, but it doesn't quite work for our definition of phi. Other ideas? Yeah.

AUDIENCE: The total number of [INAUDIBLE]

ERIK DEMAINE: The total number of 1s. Yeah. Let's define phi, could be the number of 1 bits. That will work, but you both get a Frisbee.

AUDIENCE: Oh, [INAUDIBLE].

ERIK DEMAINE: Sorry. Good thing I missed. Number 1 bits. Intuitively, 1s are bad, and this is a good definition, because when I increment I only create one 1, so I'm not going to have this issue that delta phi goes up by a lot-- sorry, that phi goes up by a lot, that delta phi is really large. Because even in this scenario, if I increment, I only add one 1.

In this scenario, I destroy three 1s and add one. In general, if there are, let's say, t trailing bits, then an increment destroys t 1 bits, and it creates one 1 bit. That's always what happens. T could be 0, and then I have a net positive of 1, but most of the time actually I destroy 1 bits-- well, more than half the time I destroy 1 bits, and I just create a single 1 bit, in terms of the total number of 1s.

So the amortized cost is the actual cost, which is this 1 plus t. I'm actually going to remove the-- well, yeah. I'd like to remove the big O if I could. I'm going to count-- I want to be a little bit precise about my counting, because I have to do a minus sign here. If I just wrote minus t, that doesn't quite work out, because there's a constant here that I have to annihilate.

If I count the number of bits that change, then that's exactly 1 plus t in an increment. And now the change of potential is that I decrease by t, I increase by 1, I get 0. That seems a little bit too small, 0 time per operation.

AUDIENCE: You're adding a 1, you're not subtracting [INAUDIBLE]. Sorry, you're not subtracting [INAUDIBLE]. Just subtracting something else.

ERIK DEMAINE: Oh, right, sorry. That's 2. Thank you. I just can't do the arithmetic. I wrote everything correct, but this is a plus 1 and a plus 1. T minus t is the key part that cancels. Now, if you were measuring running time instead of the number of changed bits, you'd have to have a big O here, and in that case you'd have to define phi to be some constant times the number of 1 bits. So you could still set that constant large enough so that this part, which is multiplied by c, would annihilate this part, which would have a big O. I guess I'll write it in just for kicks so you've seen both versions. This would be minus c see times t plus 1 times c. So that would still

work out. If you set c to the right value, you will still get 2. So binary counters, constant amortize operation. So I think this is very clean, much easier than analyzing the fractal of the costs.

Now, binary counter with increment and decrements, that doesn't work. There are other data structures to do it, but that's for another class.

Let's go back to 2-3 trees, because I have more interesting things to say about them. Any questions about binary counters? As you saw, it wasn't totally easy to define a potential function, but we're going to see-- if see enough examples, you get some intuition for them, but it is probably the hardest method to use but also kind of the most powerful. I would say all hard amortizations use a potential function. That's just life. Finding them is tough. That's reality.

I want to analyze insertions only in 2-3 trees, then we'll do insertions and deletions, and I want to count how many splits in a 2-3 tree when I do an insertion. So remember, when you insert into a 2-3 tree, so you started a leaf, you insert a key there. If it's too big, you split that node into two parts, which causes an insert of a key into the parent. Then that might be too big, and you split, and so on. So total number of splits per insert? Upper bounds?

AUDIENCE: Log n.

ERIK DEMAINE: Log n. OK. Definitely log n in the worst case. That's sort of the actual cost but, as you may be guessing, I claim the amortized number of splits is only constant, and first will prove this with insertion only. With insertion and deletion in a 2-3 tree, it's actually not true, but for insertion only this is true. So let's prove it. A 2-3 tree, we have two types of nodes, 2 nodes and 3 nodes. I'm counting the number of children, not the number of keys, is one smaller than the number of children. Sorry, no vertical line there. This is just sum key x, sum key and y.

So when I insert a key into a node, it momentarily becomes a 4 node, you might say, with has three keys, x, y, and z. So 4 node, it has four children, hence the 4, and we split it into x and z. There's the four children, same number, but now they're distributed between x and z. And then y gets promoted to the next level up, which allows us to have two pointers to x and z. And that's how 2-3 trees work. That's how split works.

Now, I want to say that splitting-- I want to charge the splitting to something, intuitively. Let's say y was the key that was inserted, so we started with x z, which was a 3 node. When we did an insert, it became a 4 node, and then we did a split, which left us with two 2 nodes and

something. So what can you say overall about this process?

What's making this example bad? What's making the split happen, in some sense? I mean, the insert is one thing, but there's another thing we can charge to. Insert's not enough, because we're going to do log n splits, and we can only charge to the insert once if we want constant amortized bound. Yeah?

AUDIENCE: Number of 3 nodes?

ERIK DEMAINE: Number of 3 nodes, exactly. That's a good potential function, because on the left side of this picture, we had one 3 node. On the right side of the picture, we had two 2 nodes. Now, what's happening to the parent? We'll have to worry about that in a moment, but you've got the intuition. Number of 3 nodes.

I looked at just a single operation here, but if you look more generally about an expensive insert, in that it does many splits, the only way that can happen is if you had a chain of 3 nodes all connected to each other and you do an insert down here. This one splits, then this one splits, then this one splits. So there are all these 3 nodes just hanging around, and after you do the split, the parent of the very last node that splits, that might become a 3 node. So that will be up here somewhere. You might have made one new 3 node, but then this one is a couple of 2 nodes, this becomes a couple of 2 nodes, and this becomes a couple of 2 nodes. So if you had k 3 nodes before, afterwards you have one. Sound familiar?

This is actually exactly what's going on with the binary counter, so this may seem like a toy example, but over here we created, at most, one 1 bit. Down here we create, at most, one 3 node, which is when the split stops. When the split stops, that's the only time we actually insert a key into a node and it doesn't split, because otherwise you split. When you split, you're always making two nodes, and that's good.

At the very end when you stop splitting, you might have made one 3 node. So in an insert, let's say the number of splits equals k, then the change of potential for that operation is minus k plus 1, because for every split there was a 3 node to charge to-- or for every split there was a 3 node that became two nodes, two 2 nodes. So the potential went down by one, because you used to have one 3 node, then you had 0. At the very end, you might create one 3 node. That's the plus 1.

So the amortized cost is just the sum of these two things, and we get 1. That's k minus k plus

1 which is 1. Cool, huh? This is where a potential method becomes powerful, I would say. You can view this as a kind of charging argument, but it gets very confusing. Maybe with coins is the most plausible use. Essentially, the invariance you'd want is that you have a coin on every 3 node.

Same thing, of course, but it's I think easier to think about it this way. Say, well, 3 nodes seem to be the bad thing. Let's just count them, let's just see what happens. It's more like you say I want to have this invariant that there's a coin on every 3 node. How can I achieve that? And it just works magically, because A, it helps it was true and, B, we had to come up with the right potential function. And those are tricky and, in general with amortization, unless you're told on a p set prove order t amortize, you don't always know what the right running time is, and you just have to experiment.

Our final example, most impressive. Let's go over here. It's a surprise, I guess. It's not even on the list. I want to do-- this is great for inserts, but what about deletes? I want to do inserts and deletes. I'd like to do 2-3 trees, but 2-3 trees don't work. If I want to get a constant amortized bound for inserts and deletes, I've got to constant advertised here for inserts-- I should be clear. I'm ignoring the cost of searching. Let's just say searching is cheap for some reason. Maybe you already know where your key is, and you just want to insert there. Then insert only costs constant amortize in a 2-3 tree.

Insert and delete is not that good. It can be log n for every operation if I do inserts and deletes, essentially for the same reason that a binary counter can be n for every operation if I do increments and decrements. I could be here, increment a couple times, and then I change a huge number of bits. If I immediately decrement, then all the bits go back. In increment, all the bits go back. Decrement, all the bits go back. So I'm changing end bits in every operation.

In the same way, if you just think of one path of your tree, and you think of the 0 bits as 2 nodes and the 1 bits as 3 nodes, when I increment by inserting at the bottom, all those 3s turn to 1, except the top I make a 3. That's just like a binary counter. It went from all 1s to 1 0 0 0 0 0, and then if I decrement, if I delete from that very same leaf, then I'm going to have to do merges all the way back up and turn those all back into 3 nodes again. And so every operation is going to pay log n. Log n's, not so bad, but I really want constant.

So I'm going to introduce something new called 2-5 trees, and it's going to be exactly like b trees that you learned, except now the number of children of every node should be between 2

and 5. All the operations are defined the same. We've already talked about insert. Now insert-- when you have six children, then you're overflowing, and then you're going to split in half and so on.

So actually I should draw that picture, because we're going to need it. So if I started with a 5 node, which means it has four keys, and then I insert into it, I get a 6 node. That's too many. Six children. OK, that's too much, so I'm going to split it in half, which is going to leave a 3 node and a single item, which gets promoted to the parent, and another 3 node.

OK, so we started with a 5 node, and the result was two 3 nodes. OK, that split, and we also contaminate the parent a little bit, but that may lead to another split, which will look like this again. So if we're just doing insertions, fine, we just count the number of 5 nodes, no different, right? But I want to do simultaneously insert and delete.

So let's remember what happens with a delete. So if you just delete a key and a leaf, the issue is it may become too empty. So what's too empty? Well, the minimum number of children we're allowed to have is two, so too empty would be that I have one child. So maybe initially I have two children, and I have a single key x, then maybe I delete x, and so now I have 0 keys. This is a 1 node. It has a single child. OK. Weird.

In that case, there are sort of two situations. Maybe your sibling has enough keys that you can just steal one, then that was really cheap. But the other case is that you-- yeah. I'm also going to have to involve my parent, so maybe I'm going to take a key from x and merge all these things together. So that's y, then what I get is an x y. I had two children here, three children here.

OK. Also messed up my parent a little bit, but that's going to be the recursive case. This is a sort of merge operation. In general, I merge with my sibling and then potentially split again, or you can think of it as stealing from your sibling, as you may be experienced with doing. I don't have siblings, so I didn't get to do that, but I stole from my parents, so whatever. However you want to think about it, that is merging in a b tree. We started with a 2 node here. We ended up with a 3 node. Hmm, that's good. It's different at least. So the bad case here is a 5 node, bad case here is a 2 node. What should I use a potential function? Yeah.

AUDIENCE: Number of nodes with two children and number of nodes with five children?

ERIK DEMAINE: Number of nodes with two or five children, yeah. So that's it. Just combine with the sum. That's

going to be the number of nodes with two children plus the number of nodes with five children.

This is measuring karma. This is how bad is my tree going to be, because if I have 2 nodes, I'm really close to under flowing and that's potentially bad. If I happen to delete there, bad things are going to happen. If I have a bunch of 5 nodes, splits could happen there, and I don't know whether it's going to be an insert or delete next, so I'm just going to keep track of both of them. And luckily neither of them output 5s or 2s. If they did, like if we did 2-3 trees, this is a total nightmare, because you can't count the number of 2 nodes plus the number 3 nodes. That's all the nodes.

Potential only changes by 1 in each step. That would never help you. OK? But here we have enough of a gap between the lower bound and the upper bound and, in general, any constants here will work. These are usually called a-b trees, generalization of b trees, where you get to specify the lower bound and the upper bound, as long as a-- what's the way-- as long as a is strictly less than b over 2, then this argument will work.

As long as there's at least one gap between a and b over 2, then this argument will work, because in the small case, you start with the minimum number of children you can have. You'll get one more in the end, and in the other situation, you have too many things, you divide by 2, and you don't want dividing by 2 to end up with the bad case over here. That's what happened even with 2-4 trees-- 2-3-4 trees-- but with 2-5 trees, there's enough of a gap that when we split 5 in half, we get 3s only, no 2s, and when we merge 2s, we get 3s, no 5s.

So in either case, if we do the split-- if we do an insert with k splits, the change in potential is minus k plus 1. Again, we might make a single five-child node at the top when we stop splitting, but every time we split, we've taken a 5 node and destroyed it, left it with two 3 nodes, so that decreases by k, and so this k cost gets cancelled out by this negative k and change potential, so the amortized cost is 1 just like before.

But now, also with delete, with k merge operations where I'm treating all of this as one operation, again, the change of potential is minus k plus 1. Potentially when we stop merging, because we stole one key from our parent, it may now be a 2 node, whereas before it wasn't. If it was already a 2 node, then it would be another merge, and that's actually a good case for us, but when the merges stop, they stop because we hit a node that's at least a 3 node, then we delete a key from it, so potentially it's a 2 node. So potentially the potential goes up by 1.

We make one new bad node, but every time we do a merge, we destroy bad nodes, because

we started with a 2 node, we turned it into a 3 node. So, again, the amortized cost is the actual cost, which is k, plus the change in potential, which is minus k plus 1, and so the amortized cost is just 1. Constant number of splits or merges per insert or delete.

So this is actually really nice if you're in a model where changing your data structure is more expensive than searching your data structure. For example, you have a lot of threads in parallel accessing your thing. You're on a multi-core machine or something. You have a shared data structure, you really don't want to be changing things very often, because you have to take a lock and then that slows down all the other threads. If searches are really fast but splits and merges are expensive, then this is a reason why you should use 2-5 trees instead of 2-3 trees, because 2-3 trees, they'll be splitting emerging all the time, log n. It's not a huge difference, log n versus constant, but with data structures that's usually the gap.

Last class we were super excited, because we went from log to log log. Here we're excited we go from log to constant. It's a little better, but they're all small numbers, but still we like to go fast, as fast as possible. In a real system, actually it's even more important, because splitting the root is probably the worst, because everyone is always touching the root. In a 2-5 tree, you almost never touch the root, almost always splitting and merging at the leaves, whereas in a 2-3 tree, you could be going all the way to the root every single time. So that's my examples. Any questions?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: For free minutes. Cool. That's amortization.