MITOCW | watch?v=Mf9Nn9PbGsE

ERIK DEMAINE: All right, today we're going to do an exciting topic, which is hashing. Do it all in one lecture, that's the plan. See if we make it.

You've probably heard about hashing. It's probably the most common data structure in computer science. It's covered in pretty much every algorithms class. But there's a lot to say about it.

And I want to quickly review things you might know and then quickly get to things you shouldn't know. And we're going to talk on the one hand about different kinds of hash functions, fancy stuff like k-wise independence, and a new technique that's been analyzed a lot lately, simple tabulation hashing, just in the last year. And then we'll look at different ways to use this hash function that actually build data structures, chaining is the obvious one; perfect hashing you may have seen; linear probing is another obvious one, but has only been analyzed recently; and cuckoo hashing is a new one that has its own fun feature. So that's where we're going to go today

Remember, the basic idea of hashing is you want to reduce a giant universe to a reasonably small table. So I'm going to call our hash function h. I'm going to call the universe integer 0 up to u minus 1. So this is the universe. And I'm going to denote the universe by a capital U.

And we have a table that we'd like to store. I'm not going to draw it yet because that's the second half of the lecture, what is the table actually. But we'll just think of it as indices 0 through m minus 1. So this is the table size.

And probably, we want m to be about n. n is the number of keys we're actually storing in the table. But that's not necessarily seen at this level. So that's a hash function. m is going to be much smaller than u. We're just hashing integers here, so if you don't have integers you map your whatever space of things you have to integers. That's pretty much always possible.

Now, best case scenario would be to use a totally random hash function. What does totally random mean? The probability, if you choose your hash function, that any key x maps to any particular slot-- these table things are called slots-- is 1 over m. And this is independent for all x.

So this would be ideal. You choose each h of x for every possible key randomly, independently. Then that gives you perfect hashing. Not perfect in this sense, sorry. It gives you ideal hashing.

Perfect means no collisions. This actually might have collisions, there's some chance that two keys hash to the same value. We call this totally random. This is sort of the ideal thing that we're trying to approximate with reasonable hash functions.

Why is this bad? Because it's big. The number of bits of information you'd need if you actually could flip all these coins, you'd need to write down, I guess U times log m bits of information.

Which is generally way too big. We can't afford U. The whole point is we want to store n items much smaller than U. Surprisingly, this concept will still be useful. So we'll get there.

Another system you've probably seen is universal hashing. This is a constraint on hash function. So this would be ideally you'd choose h uniformly at random from all hash functions. That would give you this probability. We're going to make a smaller set of hash functions whose size is much smaller. And so you can encode the hash function in many fewer bits.

And the property we want from that hash family is that if you look at the probability that two keys collide, you get roughly what you expect from totally random. You would hope for 1 over m. Once you pick one key, the probability the other key would hit it would be 1 over m.

But we'll allow constant factor. And also allow it to be smaller. It gives us some slop. You don't have to do this. If you don't do this you it's called strongly universal.

That's universal. And universal is enough for a lot of things that you've probably seen, but not enough for other things. So here are some examples of hash functions that are universal, which again, you may have seen.

You can take a random integer a, multiply it by x integer multiplication. You could also do this as a vector wise dot product. But here I'm doing it as multiplication. Modulo a prime. Prime has to be bigger than U, maybe bigger or equal is fine. Universe. And then you take the whole thing modulo em,

Now, this is universal but it loses a factor of 2 here, I believe, in general. Because you take things modulo prime and then you take things modulo whatever your table size is. If you set your table size to p that's great. I think you get a factor of 1.

If you don't, you're essentially losing possibly half the slots, depending on how m and p are related to each other. So it's OK, but not great. It's also considered expensive because you have do all this division, which people don't like to do.

So there's a fancier method which is a times x shifted right by log u minus log m. This is when m and u are powers of 2, which is the case we kind of care about. Usually your universe is of size 2 to the word size of your machine, 2 to the 32, 2 to the 64, however bigger integers are.

So it's usually a power of 2. It's fine to make your table a power of 2. We're probably going to use table doubling. So you just multiply and then take the high order bits, that's what this is saying.

So this is a method more recent, 1997. Whereas this one goes back to 1979. So '79, '97. And it's also universal.

There's a lot of universal hash functions. I'm not going to list them all. I'd rather get to stronger properties than universality.

So the next one is called k-wise independence. This is harder to obtain. And it implies universality. So we want a family of hash functions such that--

Maybe let's start with just pairwise independence, k equals 2. Then what this is saying is the probability of your choice of hash function, that the first key maps to this slot, t1, and the second key maps to some other slot, t2. For any two keys x1 and xk. If your function was random each of those happens with probability 1 over m, they're independent. So you get 1 over m to the k, or 1 over m squared for k equals 2.

Even in that situation that's different from saying the probability of 2 keys being equal is 1 over m. This would imply that. But here there could still be some co-dependence between x and y. Here there essentially can't be. I mean, other than this constant factor.

Pairwise independence means every two guys are independent, k-wise means every k guys are independent up to the constant factor. So this is for distinct xi's. Obviously if two of them are equal they're very likely to hash to the same slot. So you've got to forbid that.

OK, so an example of such a hash function. Here we just took a product. In general you can take a polynomial of degree k minus 1. Evaluate that mod p. And then if you want some, modulo that to your table size.

So in particular if k equals 2, we actually have to do some work. This function is not pairwise independent, it is universal. If you make it ax plus b for random a and b, then this becomes pairwise independent. In general, you want three wise independent, triple wise independent, you need ax squared plus bx plus c for random a, b's, and c's. So these are arbitrary numbers between 0 and p, I guess.

OK. This is also old, 1981. Wegman and Carter introduced these two notions in a couple of different papers. This is an old idea. This is, of course, expensive in that we pay order k time to evaluate it.

Also, there's a lot of multiplications and you have to do everything modulo p. So a lot of people have worked on more efficient ways to do k-wise independence. And there two main results on this. Both of them achieve m to the epsilon space, is not great.

One of them the query time depends on k, and it's uniform, and reasonably practical [? with ?] experiments. The other one is constant query for a logarithmic independence.

So this one is actually later. It's by Thorpe and [? Tsang. ?] And this is by Siegel. Both 2004, so fairly recent.

It takes a fair amount of space. This paper proves that to get constant query time for logarithmic independence you'd need quite a bit of space to store your hash function. Keep in mind, these hash functions only take-- well this is like k log in bits to store. So this is words of space.

So here we're only spending k words to store this hash function. It's very small. Here you need something depending on n, which is kind of annoying, especially if you want to be dynamic. But statically you can get constant query logarithmic wise independence, but you have to pay a lot in space.

There's more practical methods. This is especially practical for k equals 5, which is a case that we'll see is of interest. Cool. So this much space is necessary if you want. We'll see log wise independence is the most we'll ever require in this class. And as far as I know, in hashing in general. So you don't need to worry about more than log wise independence.

All right, one more hashing scheme. It's called simple tabulation hashing. This is a simple idea. It goes also back to '81 but it's just been analyzed last year. So there's a lot of results to report

on it.

The idea is just take your integer, split it up into some base so that there's exactly c characters. c is going to be a constant. Then build a totally random hash table. This is the thing that we couldn't afford. But we're just going to do it on each character.

So there's going to be c of these hash tables. And each of them is going to have size u to the 1 over c. So essentially we're getting u to the epsilon space, which is similar to these space bounds. So again, not great, but it's a really simple hash function. Hash function is just going to be you take your first table, apply it to the first character, x over that, with the second table applied to the second character, and so on through all the characters.

So the nice thing about this is it's super simple. You can imagine this being done probably in one instruction on a fancy CPU. if you convince people this is a cool enough instruction to have. It's very simple to implement circuit wide. But in our model you have to do all these operations separately. You're going to take orders c time to compute.

And one thing that's known about it is that it's three independent, three wise independent. So it does kind of fit in this model. But three wise independence is not very impressive. A lot of the results we'll see require log n independence. But the cool thing is, roughly speaking simple tabulation hashing is almost as good as log n wise independence in all the hashing schemes that we care about. And so we'll get there, exactly what that means.

So that was my overview of some hash functions, these two guys. Next we're going to look at basic chaining. Perfect hashing. How many people have seen perfect hashing just to get a sense? More than half. Maybe 2/3. All right, I should do this really fast.

Chaining, this is the first kind of hashing you usually see. You have your hash function, which is mapping keys into slots. If you have two keys that go to the same slot you store them as a linked list. OK, if you don't have anything in this slot, it's blank. This is very easy.

If you look at a particular slot t and call the length of the chain that you get there Ct. You can look at the expected length of that chain. In general, it's just going to be sum of the probability that the keys map to that slot. And then you sum over all keys. This is just writing this as a sum of indicator random variables, and then each linearity of expectation, expectation of each of the indicator variables is probability. So that is the expected number.

Here we just need to compute the probability each guy goes to each slot. As long as your hash function is uniform, meaning that each of these guys is equally likely to be hashed to. Well actually, we're looking at a particular slot. So we're essentially using universality here.

Once we fix one slot that we care about, so let t be some h of y that we care about, then this is universality. By universality we know this is 1 over m. And so this is 1 over n over m, usually called the load factor. And what we care about is this is constant for m equal theta n. And so you use table doubling to keep m theta n. Boom, you've got expected chain length constant.

But in the theory world expected is a very weak bound. What we want are high probability bounds. So let me tell you a little bit about high probability bounds. This, you may not have seen as much.

Let's start with if your hash function is totally random, then your chain lengths will be order log m over a log log n, with high probability. They are not constant. In fact, you expect the maximum chain to be at least log n over log log n. I won't prove that here. Instead, I'll prove the upper bound.

So the claim is that while in expectation each of them is constant, variance is essentially high. Actually let's talk about variance a little bit. Sorry, I'm getting distracted. So you might say, oh OK, expectation is nice.

Let's look at variance. Turns out variance is constant for these chains. There are various definitions of variance. But in particular the formula I want to use is this one. It writes it as some expectations.

Now, this expected chain length we know is constant. So you square it, it's still constant. So that's sort of irrelevant. The interesting part is what is the expected squared chain length.

Now this is going to depend exactly on your hash function. Let's analyze it for totally random. In general, we just need a certain kind of symmetry here. You can write I will look at the expected squared chain lengths.

And instead, what I'd like to do is just sum over all of them. This is going to be easier to analyze. So this is expected squared chain lengths. If I sum over all chains and then divide, take the average, then I'll probably get the expected chain length of any individual.

As long as your hash function is symmetric all the keys are sort of equally likely. This will be

true. And then you could just basically apply a random permutation to your keys to make this true if it isn't already.

Now this thing is just the number of pairs of keys that collide. So you can forget about slots, this is just the sum overall pairs of keys ij of the probability that xi hashes to the same spot as xj. And that's something we know by universality. This is 1 over m. Big O.

The number of pairs is m squared. So we get m squared times 1 over m, times 1 over m. This is constant.

So the variance is actually small. It's not a good indicator of how big our chains can get. Because still, with time probability one of the chains will be log n over log log n. It's just typical one won't be.

Let's prove the upper bound. This uses Chernoff bounds. This is a tail bound, essentially. I haven't probably defined with high probability.

It's probably good to remember, review this. This means probability at least 1 minus 1 over n to c where I get to choose any constant c. So high probability means polynomially small failure probability. This is good because if you do this polynomially many times this property remains true. You just up your constant by however many times you're going to use it.

So we prove these kinds of bounds using Chernov, which looks something like this. e to the c minus 1 mu over c mu c mu. So mu here is the mean. The mean we've already computed is constant. The expectation of the ct variable is constant.

So we want it to be not much larger than that. So say the probability that it's some factor larger-- c doesn't have to be constant here, sorry, maybe not great terminology. So the probability of ct is at least some c times the mean, is going to be at most this exponential. Which is a bit annoying, or a bit ugly.

But in particular, if we plug in c equals log n over log log n, use that as our factor, which is what we're concerned about here then. We get that this probability is essentially dominated by the bottom term here. And this becomes log n over log log n to the power log n over log log n. So essentially, get 1 over that.

And if you take this bottom part and put it into the exponent, you get essentially log log n. So this is something like 1 over 2 to the log n over log log n times log log n. And the log log n's

cancel. And so this is basically 1 over n.

And if you put a constant in here you can get a constant in the exponent here. So you can get failure probability 1 over n to the c. So get this with high probability bound as long as you go up to a chain length of log n over log log n. It's not true otherwise.

So this is kind of depressing. It's one reason we will turn to perfect hashing, some of the chains are long. But there is a sense in which this is not so bad. So let me go to that. I kind of want all these.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: What's that?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Since when is log n long. Well--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Right, so I mean, in some sense the name of the game here is we want to beat binary search trees. I didn't even mention what problem we're solving. We're solving the dictionary problem, which is sort of bunch of keys, insert delete, and search is now just exact search. I want to know is this key in there? If so, find some data associated with it.

Which is something binary search trees could do, n log n time. And we've seen various fancy ways to try to make that better. But in the worst case, you need log n time to do binary search trees.

We want to get to constant as much as possible. We want the hash function to be evaluatable in constant time. We want the queries to be done in constant time. If you have a long chain, you've got to search the whole chain and I don't want to spend log n over log log n.

Because I said so. Admittedly, log n over log log n is not that big. And furthermore, the following holds. This is a sense in which it's not really log n over log log n.

If we change the model briefly and say, well, suppose I have a cache of the last log n items that I searched for in the hash table. Then if you're totally random, which is something we assumed here in order to apply the Chernoff bound, we needed that everything was

completely random. Then you get a constant amortized bound per operation.

So this is kind of funny. In fact, all it's saying this is easy to prove. And it's not yet in any paper. It's on Mihai Petrescu's blog from 2011. All right, we're here.

We're looking at different chains. So you access some chain, then you access another chain, then you access another chain. If you're unlucky, you'll hit the big chain which cost log n over over log log n to touch, which is expensive. But you could then put all those guys into cache, and if you happen to keep probing there you know it should be faster.

In general, you do a bunch of searches. OK, first I search for x1, then I search for x2, x3, so on. Cluster those into groups theta log n. OK, let's look at the first log n searches, then the next log n searches, and analyze those separately. We're going to amortize over that period of log n.

So if we look at theta log n-- actually, this is written in a funny way. You've got the data, just log n. So I'm going to look at a batch of log n operations. And I claim that the number of keys that collide with them is theta log n, with high probability. If this is true, then it's constant each.

If I can do log n operations by visiting order log n total chain items with high probability, then I just charge one each. And so amortized over this little log n window of sort of smoothing the cost. With high probability now, not just expectation, I get constant amortized per operation. So I should have said, with high probability.

Why is this true? It's essentially the same argument. Here this is normally called a balls and bin argument. So you're throwing balls, which are your keys, randomly into your bins, which are your slots. And the expectation is constant probability to any one of them, I mean any one of them could go up to log n over log log n, high probability.

Over here, we're looking at log n different slots and taking the sum of balls that fall into each of the slots. And in expectation that's log n, because it's constant each. An expectation is linear if you're taking the sum over these log n bins. So the expectation is log n.

So you apply Chernoff again. Except now the mean is log n. And then it suffices to put c equals 2. We can run through this.

So here we get the mean is theta log n. We expect there to be log n items that fall into these log n bins. And so you just plug in c equals 2 to the Chernoff bound and you get e to the log n-

- which is kind of weird-- over 2 log n to the 2 log n. So this thing is like n to the log log n 2. So this is big, way bigger than this. So this essentially disappears.

And in particular, this is bigger than 1 over n to the c, for any c. So log log n is bigger than any constant. So you're done. So that's just saying the probability that you're more than twice the mean is very, very small. So with high probability there's only log n items that fall in these log n bins. So you just amortize, boom, constant.

This is kind of a weird notion. I've never actually seen amortized with high probability ever in a paper. This is the first time it seems like a useful concept. So if you think log n over log log n is bad, this is a sense in which it's OK. Don't worry about it.

All right, but if you did worry about it, next thing you do is perfect hashing. So, perfect hashing is really just an embellishment. This is also called FKS hashing. From the authors, Friedman, Komlosh, and [? Samaretti. ?] This is from 1984, so old idea.

You just take chaining, but instead of storing your chains in a linked list, store them in a hash table. Simple idea. There's one clever trick. You store it in a big hash table. Hash table of size theta ct squared.

Now this looks like a problem because that's going to be quadratic space, in the worst case, if everybody hashes to the same chain. But we know that chains are pretty small with high probability. So turns out this is OK. The space is sum of ct squared.

And that's something we actually computed already, except I erased it. How convenient of me. It was right here. I can still barely read it. When we computed the variance. We can do it again it's not really that hard.

This is the number of pairs of keys that collide. And so there's n squared pairs and each of them has a probability 1 over me of colliding if you have a universal hash function. So this is n squared over m, which if m is within a constant factor of n, is linear.

So linear space in expectation. Expected amount of space is linear. I won't try to do with high probability bound here. What else can I say?

You have to play a similar trick when you're actually building these hash tables. All right, so why do we use n squared? Because of the birthday paradox. So if you have a hash table of size n squared, essentially, or ct squared with high probability or constant probability you don't

get any collisions. Why?

Because then they expected number of collisions in ct. Well, there's ct pairs or ct squared pairs. Each of them, if you're using universal hashing, had a probability of 1 over ct squared of happening. Because that's the table size, 1 over m. So this is constant.

And if we set the constants right, I get to set this theta to be whatever I want. I get this to be less than 1/2. If the expected number of collisions is less than 1/2, then the probability that the number of collisions is 0 is at least a 1/2. This is Markov's inequality, in particular. The probability number of collisions is at least 1 is at most the expectation over 1, so which is 1/2.

So you try to build this table. If you have 0 collisions you're happy. You go on to the next one. If you don't have 0 collisions, just try again. So in an expected constant number of trials you're flipping a coin each time. Eventually you'll get heads.

Then you can build this table. And then you have 0 collisions. So we always want them to be collision free.

So in an expected linear time you can build this table and it will have expected linear space. In fact, if it doesn't have linear space you can just try the whole thing over again. So in expected linear time you'll build a guaranteed linear space structure.

The nice thing about perfect hashing is you're doing two hash de-references and that's it. So the query is constant deterministic. Queries are now deterministic, only updates are randomized. I didn't talk about updates. I talked about building. The construction here is randomized, queries are constant deterministic.

Now, you can make this dynamic in pretty much the obvious way you say, OK, I want to insert. So I compute which of the, it's essentially two level hashing. So first you figure out where it fits in the big hash table, then you find the corresponding chain, which is now hash table, and you insert into that hash table. So it's the obvious thing.

The trouble is you might get a collision in that hash table. If you get a collision, you rebuild that hash table. Probability of the collision happening is essentially small. It's going to remain small because of this argument. Because we know the expected number of collisions remains small unless that chain gets really big.

So if the chain grows by a factor of 2, then generally you have to rebuild the table. But if your

chain length grows by a factor of 2, then you rebuild your table to have factor 4 size larger because it's ct squared. And in general, you maintain that the table is sized for a chain of roughly the correct chain length within a constant factor. And you do doubling and halving in the usual way, like b-tree. Or, I guess it's table doubling really. And it will be constant amortized expected per operation.

And there's a fancy way to make this constant with high probability per insert and delete, which I have not read. But it's by [? Desfil ?] Binger and [? Meyer ?] [? Ofterheight, ?] 1990. So, easy to make this expected amortized. With more effort you could make it with high probability per operation. That is trickier. Cool.

I actually skipped one thing with chaining, which I wanted to talk about. So this analysis was fine, it just used universality. This cache analysis, I said totally random. This analysis, I said totally random. What about real hash functions? We can't use totally random.

What about universal k-wise independent simple tabulation hashing, just for chaining? OK, and similar things hold for perfect hashing, I think. I'm not sure if they're all known. Oh, sorry, perfect hashing.

In expectation everything's fine, just with universal. So we've already done that with universal.

What about chaining? How big can the chains get? I said log n over log log with a high probability, but our analysis used Chernoff bound. That's only true for Bernoulli trials. It was only true for totally random hash functions.

It turns out same is true if you have a log n over log log n wise independent hash function. So this is kind of annoying. If you want this to be true you need a lot of independence. And it's hard to get log n independence. There is a way to get constant, but it needed a lot of space.

That was this one, which is not so thrilling. It's also kind of complicated. So if you don't mind the space but you just want it to be simpler, you can use simple tabulation hashing. Both of these, the same chain analysis turns out to work.

So this is fairly old. This is from 1995. This is from last year. So if you just use this simple tabulation hashing, still has a lot of space, e to the epsilon. But very simple to implement.

Then still, the chain lengths are as you expect them to be. And I believe that carries over to this caching argument, but I haven't checked it. All right. Great, I think we're now happy. We've

talked about real hash functions for chaining and perfect hashing.

Next thing we're going to talk about is linear probing. I mean, in some sense we have good theoretical answers now. We can do constant expected amortized even with constant deterministic queries. But we're greedy, or people like to implement all sorts of different hashing schemes.

Perfect hashing is pretty rare in practice. Why? I guess, because you have to hash twice instead of once and that's just more expensive.

So what about the simpler hashing schemes? Simple tabulation hashing is nice and simple, but what about linear probing? That's really simple.

Linear probing is either the first or the second hashing scheme you learned. You store things in a table. The hash function tells you where to go. If that's full, you just go to the next spot. If that's full, you go to the next spot till you find an empty slot and then you put x there. So if there's some y and z here, that that's where you end up putting it.

Everyone knows linear probing is bad because the rich get richer. It's like the parking lot problem. If you get big runs of elements they're more likely to get hit, so they're going to grow even faster and get worse. So you should never use linear probing. Has everyone learned that?

It's all false, however. Linear probing is actually really good. And first indication is it's really good in practice. There's this small experiment by Mihai Petrescu, who was an undergrad and PhD student here. He's working on AT&T now.

And he was doing some experiments and he found that in practice on a network router linear probing costs 10% more time than a memory access. So, basically free. Why? You just set m to be 2 times n, or 1 plus epsilon times n, whatever. It actually works really well. And I'd like to convince you that it works really well.

Now, first let me tell you some things. The idea that it works really well is old. For a totally random hash function you require constant time per operation. And Knuth actually showed this first in 1962 in a technical report. The answer ends up being 1 over epsilon squared.

Now you might say, 1 over epsilon squared, oh that's really bad. And there are other schemes that achieve 1 over epsilon, which is better. But what's a little bit of space, right? I mean, just

set epsilon to 1, you're done.

So I think linear probing was bad when we really were tight on space. But when you can afford a factor of 2, linear probing is great. That's the bottom line.

Now, this is totally random, not so useful. What about all these other hash functions? Like universal, turns out universal with the universal hash function, linear probing is really, really bad. And that's why it gets a bad rap.

But some good news. OK, first result was log n wise independence. This is extremely strong but it also implies constant expected per operation. Not very exciting.

The big breakthrough was in 2007 that five-wise independence is enough. And this is why this paper, Thorpe was focusing in particular on the case of k equals 4. Actually, they were doing k equals 4, but they solved 5 at the same time. And so this was a very highly optimized, practical, all that good stuff. Get five-wise independence, admittedly with some space. But it's pretty cool.

So this is enough to get constant expected. I shouldn't write order 1, because I'm not writing the dependence on epsilon here. I don't know exactly what it is. But it's some constant depending on epsilon.

And then this turns out to be tight. There are four-wise independent hash functions, including I think the polynomial ones that we did. These guys that are really bad. You can get really bad. They're as bad as binary search trees. You can get constant expected.

So you really need five-wise independence. It's kind of weird, but it's true.

And the other fun fact is that simple tabulation hashing also achieves constant. And here it's known that it's also 1 over epsilon squared. So it's just as good as totally random simple tabulation hashing. Which is nice because again, this is simple. Takes a bit of space but both of these have that property. And so these are good ways to use linear probing in particular.

So you really need a good hash function for linear probing to work out. If you use the universal hash function like a times x mod p mod m it will fail. But if you use a good hash function, which we're now getting to the point-- I mean, this is super simple to implement.

It should work fine. I think would be a neat project to take a Python or something that had hash

tables deep inside it, replace-- I think they use quadratic probing and universal hash functions. If you instead use linear probing and simple tabulation hashing, might do the same, might do better, I don't know. It's interesting. It would be a project to try out. Cool.

Well, I just quoted results. What I'd like to do is prove something like this to you. Totally random hash functions imply some constant expected. I won't try to work out the dependence on epsilon because it's actually a pretty clean proof, it looks nice. Very data structures-y.

I'm not going to cover Knut's proof. I'm essentially covering this proof. In this paper five-wise independence implies constant expected. They re-prove the totally random case and strengthen it, analyze the independence they need.

Let's just do totally random unbiased constant expected for linear probing. We obviously know how to do constant expected already with other fancy techniques. But linear probing seems really bad. Yet I claim, not so much.

And we're going to assume m is at least 3 times n. That will just make the analysis cleaner. But it does hold for 1 plus epsilon.

OK, so here's the idea. We're going to take our array, our hash table, it's an array. And build a binary tree on it because that's what we like to do. We do this every lecture pretty much. This is kind of like ordered file maintenance, I guess.

This is just a conceptual tree. I mean, you're not even defining an algorithm based on this because the algorithm is linear probing. You go into somewhere. You hop, hop, hop, hop until you find a blank space. You put your item there.

OK, but each of these nodes defines an interval in the array, as we know. So I'm going to call a node dangerous, essentially if its density is at least 2/3. But not in the literal sense because there's a little bit of a subtlety here.

There's the location where a key wants to live, which is h of that key. And there's the location that it ended up living. I care more about the first one because that's what I understand. h of x, that's going to be nice. It's totally random. So h of x is random independent of everything else. Great.

Where x ends up being, that depends on other keys and it depends on this linear thing which I'm trying to understand. So I just want to talk about the number of keys that hash via h to the

interval if that is at least 2/3 times the length of the interval. This is the number of slots that are actually there.

We expect the number of keys that hash via h to the interval to be 1/2. So the expectation would be 1/3 the length the interval. If it happens to be 2/3 it could happen because of high probability, whatever.

That's a dangerous node. That's the definition. Those ones we worry will be very expensive. And we worry that we're going to get super clustering and then get these giant runs, and so on.

So, one thing I want to compute was what's the probability of this happening. Probability of a node being dangerous. Well, we can again use Chernoff bounds here because we're in a totally random situation.

So this is the probability that the number of things that went there was bigger than twice the expectation. The expectation is 1/2, 2/3 is twice of 1/3. So this is the probability that you're at least twice the mean, which by Chernoff is small. It comes out to e to the mu over 2 to 2 mu.

So this is e over 4 to the mu. You can check e. It's 2.71828. So this is less than 1, kind of roughly a half-ish. So this is good. This is something like 1 over 2 to the mu. What's mu?

mu is 1/3 2 to the h for a height h node. It depends on how high you are. If you're at a leaf h is 0, so you expect 1/3 of an element there. As you go up you expect more elements to hash there, of course.

OK, so this gives us some measure in terms of this h of what's going on. But it's actually doubly exponential in h. So this is a very small probability. You go up a few levels.

Like, after log log n levels it's a polynomially small probability of happening. Because then 2 to the log log n is log n. And then e over 4 to the log n is about n. OK. But at small levels this may happen, near the leaves.

All right, so now I want to look at a run in the table. These are the things I have trouble thinking about because runs tend to get bigger, and we worry about them. This is now as items are actually stored at the table, when do I have a bunch of consecutive items in there that happen to end up in consecutive slots? So I'm worried about how long that run is.

So let's look at its logarithm and round to the nearest power of 2. So let's say it has length about 2 to the l. Sorry, plus 1. All right, between 2 to the l and 2 to the l plus 1. OK, look at that.

And it's spanned by some number of nodes of height h equals l minus 3. OK, so there's some interval that happens to be a run, meaning all of these slots are occupied. And that's 2 to the 2, I guess, since I got to level negative 1. A little hard to do in a small picture. But we're worried about when this is really big more than some constant.

OK, so let's suppose I was looking at this level. Then this interval is spanned, in particular, by these two nodes. Now it's a little sloppy because this node contains some non-interval, non-run stuff, and so does this one.

At the next level down it would be this way one, this one, and this one, which is a little more precise. But it's never going to be quite perfect. But just take all the nodes you need to completely cover the run.

Then this will be at least eight nodes because the length is 1 to the l. We went three levels down, 2 to the 3 is 8. So if it's perfectly aligned it will be exactly 8 nodes. In the worst case, it could be as much as 17. Because potentially, we're 2 to the l plus 1, which means we have 16 nodes if we're perfectly aligned.

But then if you shift if over it might be one more because of the slot. OK, but some constant number of nodes. It's important that it's at least eight. That's what we need.

Actually, we just need that's it at least five, but eight is the nearest power of two rounding up. Cool. So, there they are.

Now, I want to look at the first four nodes of these eight to 12 nodes. So first meaning leftmost. Earliest in the run.

So if you think about them, so there's some four nodes each of them spans some-- I should draw these properly. What we know is that these guys are entirely filled with items. The run occupies here. It's got to be at least one item into here, but the rest of this could be empty. And the interval keeps going to the right so we know that all of these are completely filled with items somehow.

So let's start with how many there are, I guess. They span more than three times 2 to the h slots of the run. So somehow 3 times 2 to the h-- because there's three of them that are

completely filled, otherwise it would be four. Somehow three times two to the h items ended here.

Now, how did they end up here? Notice there's a blank space right here. By definition this was the beginning of a run. Meaning the previous slot is empty. Which means all of the keys that wanted to live from here to the left got to.

So if we're just thinking about the keys that ended up in this interval, they had to initially hash to somewhere in here. h put them somewhere in this interval and then they may have moved to the right, but they never move to the left in linear hashing if you're not completely full. So because there was a blank spot here none of these keys could have fallen over to here, no deletions. So you're doing insertions. They may have just spread it out, and they made sconces have gone farther to the right, or they may filled in gaps, whatever, but h put them in this interval.

Now, I claim that in fact, at least one of these nodes must be dangerous. Now dangerous is tricky, because dangerous is talking about where h puts nodes. But we just said, got to be at least three times two to the h keys, where h put them within these four nodes, otherwise they wouldn't have filled in here.

Now, if none of those nodes were dangerous, then we'll get a contradiction. Because none of them were dangerous this means at most 4 times 2/3 times 2 to the h keys hash via h to them. Why? Because there's four of the nodes. Each of them, if it's not dangerous, has at most 2/3 of its size keys hashing there. 4 times 2/3 is 8/3, which is less than 9/3, which is 3.

OK, so this would be a contradiction because we just argued that at least 3 times 2 to the h nodes have to hash via h to somewhere in these nodes. They might hash here and then fallen over to here. So there is this kind of, things can move to the right, we've got to worry about it. But just look three levels up and it's OK. So one of these nodes, not necessarily all of them, are dangerous. And we can use that to finish our analysis.

This is good news because it says that if we have a run, which is something that's hard to think about because nodes are moving around to form a run, so keys are moving around to form a run, we can charge it to a dangerous node. Which is easy to think about because that's just talking about where keys hash via h, and h is totally random. There's a loss of a factor of 17, potentially. But it's a constant factor, no big deal.

If we look at the probability that the length of a run, say containing some key x, has length between 2 to the l and to the l plus 1, this is going to be at most 17 times the probability of a node at height l minus 3 is dangerous. Because we know one of them is, and so just to be sloppy it's at most the sum of the probabilities that any of them is. Then potentially there's a run of that length. And so union bound it's at most 17 times probability of this happening.

Now all nodes look the same because we have a totally random hash function. So we just say any node at height l minus 3. We already computed that probability. That was this. Probability of being dangerous was e over 4 to the 1/3 2 to the h. So this is going to be at most 17 times e over 4 to the 2 to the l minus 3 power. Again, doubly exponential in l.

So if we want to compute the expected run length we can just expand out the definition. Well, let's round it to powers of 2. It could be the run length is about 2 to the l within a constant factor of 2 to the l. So it's going to be that times this probability.

But this thing is basically 1 over 2 to the 2 to the l. And so the whole thing is constant. This is l. I mean, l could go to infinity. I don't really care. I mean, this gets dwarfed by the double exponential.

This is super geometric. So a very low probability of getting long runs. As we said, after a log log n size-- yeah, it's very unlikely to run longer than log n. We proved that in particular. But in particular, you compute the expected run length, it's constant.

OK, now this of course assumed totally random. It's harder to prove-- where were we. Somewhere. Linear probing.

It's harder to prove five-wise independence is enough, but it's true. And it's much harder to prove simple tabulation hashing works, but it's true. So we can use them. This gives you some intuition for why it's really not that bad. And similar proof techniques are used for the five-wise independence.

Other fun facts. You can do similar caching trick that we did before. Again, the worst run is going to be log, or log over log log. I don't have it written here. But if you cache the last-- it's not quite enough to do the last log n.

But if you cache the last log to the 1 plus epsilon n queries. It's a little bit more. Then you can generalize this argument. And so at least for totally random hash functions you get constant amortize with high probability.

This weird thing that I've never seen before. But it's comforting because it's expected bounds are not so great, but you get it with high probability bound as long as you're willing to average over log to the 1 plus epsilon different queries. As long as you can remember them.

And the proof is basically the same. Except now instead of looking at the length of a run containing x, you're looking at the length of the run containing one of these log to the 1 plus epsilon n nodes. That's your batch. And you do the same thing. But now do it with high probability analysis.

But again, because the expectation is now bigger than log, expect there to be a lot of fairly long runs here. But that's OK, because on average is good. You expect to pay log to the 1 plus epsilon for log to the 1 plus epsilon queries. And so then you divide and amortize and you're done. It's a little bit more details in the notes about that if you want to read.

I want to do one more topic, unless there are questions about linear probing. So, yeah?

AUDIENCE: So, could you motivate why the [INAUDIBLE] value of mu is the mean for whatever quantity?

ERIK DEMAINE: So mu is defined to be the mean of whatever quantity we're analyzing. And the Chernoff bounds says, probability that you're at least something times the mean is the formula we wrote last time. Now here, we're measuring-- I didn't write what the left-hand side was. But here we're measuring what's the probability that the number of keys that hash via h to the interval is at least 2/3 the length of the interval.

Now, let's say m equals 3m then the expected number of keys that hash via h to interval is 1/3 times the length of the interval. Because we have a totally random thing, and we have a density of 1/3 overall. So you expect there to be 1/3 and so dangerous is when you're more than twice that. And so it's twice mu. Mu is, in this case, 1/3 the length the interval. And that's why I wrote that.

AUDIENCE: So this comes from the m squared. [INAUDIBLE].

ERIK DEMAINE: Yeah, it comes from m equals 3m and totally random.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Yeah, OK let's make this equal. Make this more formal. It's an assumption, anyway, to simplify the proof. Good. Change that in the notes too. Cool.

So then the expectation is exactly 1/3 instead of at most 1/3. So it's all a little cleaner. Of course, this all works when m is at least 1 plus epsilon times n, but then you get a dependence on epsilon. Other questions?

So bottom line is linear probing is actually good. Quadratic probing, double hashing, all those fancy things are also good. But they're really tuned for the case when your table is almost full. They get a better dependence on epsilon, which is how close to the bound you are.

And so if you're constant factor away from space bound, linear probing is just fine. As long as you have enough independence, admittedly. Double hashing, I believe, gets around that. It does not need so much independence.

OK. Instead of going to double hashing, I'm going to go to something kind of related double hashing, which is cuckoo hashing. Cuckoo hashing is a weird idea. It's kind of a more extreme form of perfect hashing. It says, look, perfect hashing did two hash queries. So I did one hash evaluation and another hash evaluation followed it, which is OK.

But again, I want my queries to only do two things, two probes. So it's going to take that concept of just two and actually use two hash tables. So you've got B over here, I've got A over here. And if you have a key x, you hash it to a particular spot in A via g, and you hash it to a particular spot in B via H. So you have two hash tables, two hash functions.

To do a query you look at A of g of x, and you look at B of h of x.

Oh sorry, I forgot to mention. The other great thing about linear probing is that it's cache performance is so great. This is why it runs so fast in practice. Why it's only 10% slower than a memory access. Because once you access a single slot, whole you get B slots in a cache with block size B.

So most of the time, because your runs are very short, you will find your answer immediately. So that's why we kind of prefer linear probing in practice over all the other schemes I'm going to talk about. Well, cuckoo hashing is all right because it's only going to look at two places and that's it. Doesn't go anywhere else.

I guess with perfect hashing the thing is you have more than two hash functions. You have the first hash function which sends you to the first table. Then you look up a second hash function. Using that hash function you rehash your value x.

Downside of that is you can't compare those two hash functions in parallel. So if you're like two cores, you could compute these two in parallel, look them both up simultaneously. So in that sense you save a factor of 2 with some parallelism.

Now, the weird thing is the way we do an insertion. You try to put it in the A slot, or the B slot. If either of them is empty you're golden. If neither of them are empty, you've got to kick out whoever's there.

So let's say if you kicked out y from it's A slot. So we ended up putting x in this one, so we end up kicking y from wherever it belonged. Then you move it to B of h of y. There's only one other place that that item can go, so you put it there instead.

In general, I think about A key it has two places it can go. There's some slot in A, some slot in B. You can think of this as an edge in a bipartite graph. So make vertices for the A slots, vertices for the B slots. Each edge is an item on a key. Key Can only live one spot in A, one spot in B for this query to work.

So what's happening is if both of these are full you take whoever is currently here and put them over in their corresponding slot over in B. Now, that one might be full, which means you've got to kick that guy to wherever he belongs in A, and so on. If eventually you find an empty slot, great, you're done. Just chain reaction of cuckoo steps where the bird's going from in and out, or from A to B, vice a versa.

If it terminates, you're happy. It doesn't terminate, you're in trouble because you might get a cycle, or a few failure situations. In that case you're screwed. There is no cuckoo hash table that works for your set of keys.

In that case, you pick another hash function, rebuild from scratch. So it's kind of a weird hashing scheme because it can fail catastrophic. Fortunately, it doesn't happen too often. It still rubs me a funny way. I don't know what to say about it.

OK, so you lose a factor of 2 in space. 2 deterministic probes for a query. That's good news.

All right, now we get to, what about updates? So if it's fully random or log n-wise independent, then you get a constant expected update, which is what we want. Even with the rebuilding cost. So you'll have to rebuild about every n squared insertions you do.

The way they say this is there's a 1 over n build failure probability. There's a 1 over n chance that your key set will be completely unsustainable. If you want to put all n keys into this table there's a 1 over n chance that it will be impossible and then you have to start over.

So amortize per insertion, that's about 1 over n squared. Insertions you can do before the whole thing falls apart and you have to rebuild. So it's definitely going to be this should be amortize expected, I guess. However you want to think about it. But it's another way to do constant amortized expected. Cool.

The other thing that's known is that six-wise independence is not enough. This was actually a project in this class, I believe the first time it was offered in 2003. Six-wise independence is not sufficient to get constant expected bound. It will actually fail with high probability if you only have six-wise independence.

What's not known is, do you need constant Independence? Or log n independence? With log n, very low failure probability. With six-wise, high probability you fail. Like, you fail with probability 1 minus 1 over n. Not so good.

Some good news is simple tabulation hashing. Means you will fail to build with probability not 1 over n, but 1 over n to the 1/3 power.

And this is theta. This is tight. It's almost as good as this. We really only need constant here. This is to build the entire table.

So in this case you can insert like n to the 4 those items before your table self-destructs. So simple tabulation hashing is, again, pretty good. That's I think the hardest result in this paper from last year.

So I do have a proof of this one. Something like that. Or part of a proof. So me give you a rough idea how this works. So if you're a fully random hash function.

The main concern is that what if this path is really long. I claim that if an insert follows a path of length k, or the probability of this happening, is actually at most 1 over 2 to the k. It's very small. Exponentially small in k. I just want to sketch how this works because it's a cool argument that's actually in this simple tabulation paper.

So the idea is the following. You have some really long path. What I'm going to give you is a way to encode the hash functions. There's hash functions g and h. Each of them has n values.

Each of those values is log m bits.

So if I just wrote them down the obvious way, it's 2 n log m bits to write down those hash functions. Now we're assuming these are totally random hash functions, which means you need this many bits. But I claim that if you follow a path of length k, I can find a new encoding scheme, a way to write down g and h that is basically minus k. This many bits minus k. I get to save k bits.

Now, it turns out that can happen but it happens only with probability 1 over 2 to the k. This is an information theoretic argument. You might get lucky. And the g's and h's you're trying to encode can be done with fewer bits, k fewer bits. But that will only happen with probability 1 over 2 to the k if g and h are totally random.

So how do you do it? Basically, I want to encode the things on the path slightly cheaper. I'm going to save one bit per node on the path. So what do I need to do?

Well, the idea is OK, I will start by writing down this hash value. This takes log m bits to write down that hash value. Then I'll write down this hash value. That takes a log m bit, log m bits. Generally there's going to be roughly k log n to write down all of the node hash values.

Then I need to say that it's actually x, this particular key that corresponds to this edge. So I've got to write that down. That's going to take a log n bits to say that x is the guy for the first edge, then y is the key that corresponds to the second edge of the path, then z, then w.

But nicely, things are ordered here. So it only takes me to log n, k log n to write down all these guys. So I get k times log m plus log n.

Now if m is 2 times n, this is k times 2 log m minus 1. So I get one bit of savings per k, per thing in the path. Essentially because it's easier for me to write down these labels to say, oh, it's the key x that's going here. Instead of having to write down slot names all the time, which cost log m bits, writing down key names only takes log n bits, which is a savings of 1 bit per thing on the path.

And so that was a quick sketch of how this proof goes. It's kind of neat, information theoretic argument why the paths can't get long. You then have to worry about cycles and things that look like this. That's kind of messy. But same kind of argument generalizes.

So that was your quick overview of lots of hashing stuff.