ERIK DEMAINE: All right, let's get started. Today we're going to continue the theme of randomization and data structures. Last time we saw skip lists. Skip lists solve the predecessor-successor problem. You can search for an item and if it's not there, you get the closest item on either side in log n with high probability.

But we already knew how to do that deterministically. Today we're going to solve a slightly different problem, the dictionary problem with hash tables. Something you already think you know. But we're going to show you how much you didn't know. But after today you will know.

And we're going to get constant time and not with high probability. That's hard. But we'll do constant expected time. So that's in some sense better. It's going to solve a weaker problem. But we're going to get tighter bound constant instead of logarithmic.

So for starters let me remind you what problem we're solving and the basics of hashing which you learned in 6006. I'm going to give this problem a name because it's important and we often forget to distinguish between two types of things. This is kind of an old term, but I would call this an abstract data type.

This is just the problem specification of what you're trying to do. You might call this an interface or something. This is the problem statement versus the data structure is how you actually solve it. The hash tables are the data structure. The dictionary is the problem or the abstract data type. So what we're trying to do today, as in most data structures, is maintain a dynamic set of items.

And here I'm going to distinguish between the items and their keys. Each item has a key. And normally you'd think of there also being a value like in Python. But we're just worrying about the keys and moving the items around. And we want to support three operations. We want to be able to insert an item, delete an item, and search for an item.

But search is going to be different from what we know from AVL trees or skip lists or even Venom [INAUDIBLE] That was a predecessor-successor search. Here we just want to know--

sorry, your not searching for an item. Usually you're searching for just a key-- here you just want to know is there any item with that key, and return it. This is often called an exact search because if the key is not in there, you learn absolutely nothing. You can't find the nearest key.

And for whatever reason this is called a dictionary problem though it's unlike a real dictionary. Usually when you search for a word you do find its neighbors. Here we're just going to either-- if the key's there we find that, otherwise not. And this is exactly what a Python dictionary implements. So I guess that's why Python dictionaries are called dicts.

So today I'm going to assume all items have distinct keys. So in the insertion I will assume key is not already in the table. With a little bit of work, you can allow inserting an item with an existing key, and you just overwrite that existing item. But I don't want to worry about that here.

So we could, of course, solve this using an AVL tree in log n time. But our goal is to do better because it's an easier problem. And I'm going to remind you of the simplest way you learn to do this which was hashing with chaining in 006. And the catch is you didn't really analyze this in 006. So we're going make a constant time per operation. It's going to be expected or something and linear space.

And remember the variables we care about, there's u, n, and m. So u is the size of the universe. This is the all possible keys. The space of all possible keys. n is the size of the set your currently storing. So that's the number of items or keys currently in the data structure. And then m is the size of your table. So say it's the number of slots in the table.

So you remember the picture. You have a table of slots. Let's say 0 to m minus 1. Each of them is a pointer to a linked list. And if you have, let's say over here is your universe of all possible keys, then we have a hash function which maps each universe item into one of these slots. And then the linked list here is storing all of the items that hash to that slot. So we have a hash function which maps the universe.

I'm going to assume the universe has already been mapped into integers 0 to u minus 1. And it maps to slots. And when we do hashing with chaining, I think I mentioned this last week, the bounds you get, we achieve a bound of 1 plus alpha where alpha is the load factor n/m. The average number of items you'd expect to hash to a slot is the number of items divided by the number of slots. OK. And you proved this in 6006 but you assumed something called simple uniform hashing.

Simple uniform hashing is an assumption, I think invented for CLRS. It makes the analysis very simple, but it's also basically cheating. So today our goal is to not cheat. It's nice as a warm up. But we don't like cheating. So you may recall the assumption is about the hash function. You want a good hash function.

And good means this. I want the probability of two distinct keys mapping to the same slot to be 1/m if there are m slots. If everything was completely random, if h was basically choosing a random number for every key, then that's what we would expect to happen. So this is like the idealized scenario.

Now, we can't have a hash function could choosing a random number for every key because it has to choose the same value if you give it the same key. So it has to be some kind of deterministic strategy or at least repeatable strategy where if you plug in the same key you get the same thing. So really what this assumption is saying is that the key's that you give are in some sense random. If I give you random keys and I have not-too-crazy hash function then this will be true.

But I don't like assuming anything about the keys maybe. I want my keys to be worst case maybe. There are lots of examples in the real world where you apply some hash function and it turns out your data has some very particular structure. And if you choose a bad hash function, then your hash table gets really, really slow. Maybe everything hashes to the same slot. Or say you take-- well yeah, there are lots of examples of that. We want to avoid that.

After today you will know how to achieve constant expected time no matter what your keys are, for worst case keys. But it's going to take some work to do that. So this assumption requires assuming that the keys are random. And this is what we would call an average case analysis.

You might think that average case analysis is necessary for randomized algorithms, but that's not true. And we saw that last week with quicksort. Quicksort, if you say I will always choose a of 1 to be my partition element, that's what the textbook calls basic quicksort, then for an average input that will do really well. If you have a uniform random permutation of items and you sort with the method of always choosing the first item as your partition, then that will be n log n on average if your data is average.

But we saw we could avoid that assumption by choosing a random pivot. If you choose a random pivot, then you don't need to assume anything about the input. You just need to

assume that the pivots are random. So it's a big difference between assuming your inputs are random versus assuming your coin flips are random. It's pretty reasonable to assume you can flip coins. If you've got enough dexterity in your thumb then you can do it.

But it's not so reasonable to assume that your input is random. So we'd like to avoid average case analysis whenever we can, and that's the goal of today. So what you saw in 006 was essentially assuming the inputs are random. We're going to get rid of that unreasonable assumption today. So that's, in some sense, review from 006.

I'm going to take a brief pause and tell you about the etymology of the word hash in case you're curious. Hash is an English word since the 1650's, so it's pretty old. It means literally cut into small pieces. It's usually used in a culinary sense, like these days you have corned beef hash or something.

I'll put the definition over here. It comes from French, hacher, which means to chop up. You know it in English from the word hatchet. So it's the same derivation. And it comes from old French-- I don't actually know whether that's "hash-ay" or "hash" but-- which means axe. So you can see the derivation. If you look this up in OED or pick your favorite dictionary or even Google, that's what you find.

But in fact there's a new prevailing theory that in fact hash comes from another language which is Vulcan, la'ash, I mean you can see the derivation right? Actually means axe. So maybe French got it from Vulcan or vice versa but I think that's pretty clear. Live long and prosper, and farewell to Spock. Sad news of last week.

So enough about hashing. We'll come back to that in a little bit. But hash functions essentially take up this idea of taking your key, chopping up into pieces, and mixing it like in a good dish. All right, so we're going to cover two ways to get strong constant time bounds. Probably the most useful one is called universal hashing. We'll spend most of our time on that.

But the theoretically cooler one is called perfect hashing. Universal hashing, we're going to guarantee there are very few conflicts in expectation. Perfect hashing , we're going to guarantee there are zero conflicts. The catch is, at least in its obvious form, it only works for static sets. If you forbid, insert, and delete and just want to do search, then perfect hashing is a good method.

So like if you're actually storing a dictionary, like the OED, English doesn't change that quickly.

So you can afford to recompute your data structure whenever you release a new edition. But let's start with universal hashing. This is a nice powerful technique. It works for dynamic data.

Insert, delete, and search will be constant expected time with no assumptions about the input. So it will not be average case. It's in some sense worse case but randomized. So the idea is we need to do something random. If you just say, well, I choose one hash function once and for all, and I use that for my table, OK maybe my table doubles in size and I change the hash function. But there's no randomness there.

We need to introduce randomness somehow into this data structure. And the way we're going to do that is in how we choose the hash function. We're going to choose our hash function randomly from some set of hash functions. Call it h. This is going to be a universal hash family. We're going to imagine there are many possible hash functions we could choose.

If we choose one of them uniformly at random, that's a random choice. And that randomness is going to be enough that we no longer need to assume anything about the keys. So for that to work, we need some assumption about h. Maybe it's just a set of one hash function. That wouldn't add much randomness. Two also would not add much randomness. We need a lot of them.

And so we're going to require H to have this property. And we're going to call it the property universality. Generally you would call it a universal hash family. Just a set of hash functions. What we want is that-- so we're choosing our hash function h from H. And among those choices we want the probability that two keys hash to the same value to be small. I'll say-- and this is very similar looking to simple uniform hashing. Looks almost the same here except I switched from k1 and k2 to k and k', but same thing.

But what we're taking the probability over, what we're assuming is random is different. Here we're assuming k1 and k2 a are because h was fixed. This was an assumption about the inputs. Over here we're thinking of k and k' as being fixed. This has to work for every pair of distinct keys. And the probability we're considering is the distribution of h.

So we're trying all the different h's Or we're trying little h uniformly at random. We want the probability that a random h makes k and k' collide to be at most 1/m. The other difference is we switch from equals to at most. I mean less would be better. And there are ways to make it less for a couple pairs but it doesn't really matter. But of course anything less than or equal to 1/m will be just as good.

So this is an assumption about H. We'll see how to achieve this assumption in a little bit. Let me first prove to you that this is enough. It's going to be basically the same as the 006 analysis. But it's worth repeating just so we are sure everything's OK. And so I can be more precise about what we're assuming.

The key difference between this theorem and the 006 theorem is we get to make no assumptions about the keys. They are arbitrary. You get to choose them however you want. But then I choose a random hash function. The hash function cannot depend on these keys. But it's going to be random. And I choose the hash function after you choose the keys. That's important. So we're going to choose a random h and H. And we're assuming H is universal.

Then the expected number of keys in a slot among those n keys is at most 1 plus alpha. Alpha is n/m. So this is exactly what we had over here. Here we're talking about time bound. But the time bound followed because the length of each chain was expected to be 1 plus alpha. And here the expectation is over the choice of h. Not assuming anything about the keys.

So let's prove this theorem. It's pretty easy. But I'm going to introduce some analysis techniques that we will use for more interesting things. So let's give the keys a name. I'll just call them-- I'll be lazy. Use k1 up to kn. And I just want to compute that expectation.

So I want to compute let's say the number of keys colliding with one of those keys, let's say ki. So this is of course the size of the slot that ki happens to go. This is going to work for all i. And so if I can say that this is at most 1/alpha for each i, then I have my theorem. Just another way to talk about it.

Now the number of keys colliding with ki, here's a general trick, whenever you want to count something in expectation, a very helpful tool is indicator random variables. Let's name all of the different events that we want to count. And then we're basically summing those variables. So I'm going to say-- I'm going to use I ij to be an indicator random variable. It's going to be 1 or 0. 1 if hash function of ki equals the hash function of kj.

So there's a collision between ki and kj j and 0 if they hash to different slots. Now this is, it's a random variable because it depends on h and h is a random thing. ki and kj are not random. They're given to you. And then I want to know when does h back those two keys to the same slot.

And so this number is really just the sum of Iij over all j. This is the same thing. The number in here is the sum for j not equal to i of Iij. Because we get a 1 every time they collide, zero otherwise. So that counts how many collide. Once we have it in this notation, we can use all the great dilemmas and theorems about in this case, E, expectation. What should I use here?

STUDENT: What?

ERIK DEMAINE: What's a good-- how can I simplify this formula?

STUDENT: The linearity of expectation.

ERIK DEMAINE: The linearity of expectation. Thank you. If you don't know all these things, read the probability appendix in the textbook. So we want to talk about expectation of the simplest thing possible. So linearity let's us put the E inside the sum without losing anything.

Now the expectation of an indicator random variable is pretty simple because the zeros don't contribute to the expectation. The 1's contribute 1. So this is the same thing as just the probability of this being 1. So we get sum of j9 equal to I of the probability that Iij equals 1. And the probability that Iij equals 1, well, that's the probability that this happens. And what's the probability that that happens? At most 1/m our universality.

So I'm going to-- I'll write it out. This is sum j not equal to I. Probability that h maps ki and kj to the same slot. So that's the definition of Iij. And this is at most sum j not equal to i of 1/m by universality. So here's where we're using it. And sum of j not equal to I, well that's basically n.

But I made a mistake here. Slightly off. From here-- yeah. So this line is wrong. Sorry. Let me fix it. Because this assumption only works when the keys are distinct. So in fact-- how did I get j-- yeah. , Yeah, sorry. This should have been this-- actually everything I said is true, but if you want to count the number of keys-- I really wanted to count the total number of keys that hash to the same place as ki.

So there's one more which is ki itself. Always hashes to wherever ki hashes. So I did a summation j not equal i but I should also have a plus Iii-- captain. So there's the case when I hashing to the same place which of course is always going to happen so you get basically plus 1 everywhere. So that makes me happier because then I actually get with the theorem said which is 1 plus alpha. There is always going to be the one guy hashing there when I assume that ki hashed to wherever it does. So this tells you that if we could find a universal hash family, then we're guaranteed insert, delete, and search cost order 1 plus alpha in expectation.

And the expectation is only over the choice of h, not over the inputs. I think I've stressed that enough times.

But the remaining question is can we actually design a universal hash family? Are there any universal hash families? Yes, as you might expect there are. Otherwise this wouldn't be very interesting.

Let me give you an example of a bad universal hash family. Sort of an oxymoron but it's possible. Bad. Here's a hash family that's universal. h is the set of all hash functions. h from 0,1 to u minus 1. This is what's normally called uniform hashing.

It makes analysis really easy because you get to assume-- I mean this says ahead of time for every universe item, I'm going to choose a random slot to put it. And then I'll just remember that. And so whenever you give me the key, I'll just map it by h. And I get a consistent slot and definitely it's universal. What's bad about this hash function? Many things but--

STUDENT: [INAUDIBLE] That's just as hard as the problem I'm solving.

ERIK DEMAINE: Sort of. I'm begging the question that it's just as hard as the problem I'm solving. And what, algorithmically, what goes wrong here? There are two things I guess. Yeah?

STUDENT: It's not deterministic?

ERIK DEMAINE: It's not deterministic. That's OK because we're allowing randomization in this algorithm. So I mean how I would compute this is I would do a four loop over all universe items. And I assume I have a way to generate a random number between 0 and m minus 1. That's legitimate. But there's something bad about that algorithm.

STUDENT: It's not consistent.

ERIK DEMAINE: Not consistent? It is consistent if I precompute for every universe item where to map it. That's good. So all these things are actually OK.

STUDENT: It takes too much time and space.

ERIK DEMAINE: It takes too much time and space. Yeah. That's the bad thing. It's hard to isolate in a bad thing what is so bad about it. But we need u time to compute all those random numbers. And we need u space to store that hash function. In order to get to the consistency we have to-- Oops. Good catch. In order to get consistency, we need to keep track of all those hash function

values. And that's not good.

You could try to not store them all, you know, use a hash table. But you can't use a hash table to store a hash function. That would be-- that would be infinite recursion. So but at least they're out there. So the challenge is to find an efficient hash family that doesn't take much space to store and doesn't take much time to compute.

OK, we're allowing randomness. But we don't want to much randomness. We can't afford u units of time of randomness. I mean u could be huge. We're only doing n operations probably on this hash table. u could be way bigger than n. We don't want to have to precompute this giant table and then use it for like five steps. It would be really, really slow even amortized.

So here's one that I will analyze. And there's another one in the textbook which I'll mention. This one's a little bit simpler to analyze. We're going to need a little bit of number theory, just prime numbers. And you've probably heard of the idea of your hash table size being prime. Here you'll see why that's useful, at least for this family.

You don't always need primality, but it's going to make this family work. So I'm going to assume that my table size is prime. Now really my table size is doubling, so that's a little awkward. But luckily there are algorithms given a number to find a nearby prime number. We're not going to cover that here, but that's an algorithmic number theory thing.

And in polylogarithmic time, I guess you can find a nearby prime number. So you want it to be a power of 2. And you'll just look around for nearby prime numbers. And then we have a prime that's about the same size so that will work just as well from a table doubling perspective.

Then furthermore, for convenience, I'm going to assume that u is an integer power of m. I want my universe to be a power of that prime. I mean, if it isn't, just make u a little bigger. It's OK if u gets bigger as long as it covers all of the same items. Now once I view my universe as a power of the table size, a natural thing to do is take my universe items, to take my input integers, and think of them in base m.

So that's what I'm going to do. I'm going to view a key k in base m. Whenever I have a key, I can think of it as a vector of subkeys, k1 up to kr minus 1. There are digits in base m because of this relation. And I don't even care which is the least significant and which is the most significant. That won't matter so whatever, whichever order you want to think of it. And each of the ki's here I guess is between 0 and m minus 1. So far so good.

So with this perspective, the base m perspective, I can define a dot product hash function as follows. It's going to be parametrized by another key, I'll call it a, which we can think of again as a vector. I want to define h sub a of k. So this is parametrized by a, but it's a function of a given key k as the dot product of those two vectors mod m. So remember dot products are just the sum from i equals 0 to r minus 1 of a1 times ki. I want to do all of that modulo m.

We'll worry about how long this takes to compute in a moment I guess. Maybe very soon. But the hash family h is just all of these ha's for all possible choices of a. a was a key so it comes from the universe u.

And so what that means is to do universal hashing, I want to choose one of these ha's uniformly at random. How do I do that? I just choose a uniformly at random. Pretty easy. It's one random value from one random key. So that should take constant time and constant space to store one number.

In general we're in a world called the Word RAM model. This is actually-- I guess m stands for model so I shouldn't write model. Random access machine which you may have heard. The word RAM assumes that in general we're manipulating integers. And the integers fit in a word. And the computational assumption is that manipulating a constant number of words and doing essentially any operation you want on constant number of words takes constant time.

And the other part of the word RAM model is to assume that the things you care about fit in a word. Say individual data values, here we're talking about keys, fit in a word. This is what you need to assume in [INAUDIBLE] that you can compute high of x in constant time or low of x in constant time. Here I'm going to use it to assume that we can compute h sub a of k in constant time.

In practice this would be done by implementing this computation, this dot product computation, in hardware. And the reason a 64-bit edition on a modern processor or a 32-bit on most phones takes constant time is because there's hardware that's designed to do that really fast. And in general we're assuming that the things we care about fit in a single word.

And we're assuming random access and that we can have a raise. That's what we need in order to store a table. And same thing in [INAUDIBLE], we needed to assume we had a raise. And I think this operation is actually pretty-- exists in Intel architectures in some form. But it's certainly not a normal operation. If you're going to do this explicitly, adding up and multiplying

things this would be r is the log base m of u, so it's kind of logish time.

Maybe I'll mention another hash family that's more obviously computable. But I won't analyze here. It's analyzed in the textbook. So if you're curious you can check it out there. Let's call this just another. It's a bit weird because it has two mods. You take mod p and then mod m. But the main computation is very simple. You choose a uniformly random value a. You multiply it by your key in usual binary multiplication instead of dot product. And then you add another uniformly random key. This is also universal. So H is hab for all a and b that are keys.

So if you're not happy with this assumption that you can compute this in constant time, you should be happy with this assumption. If you believe in addition and multiplication and division being constant time, then this will be constant time. So both of these families are universal. I'm going to prove that this one is universal because it's a little bit easier. Yeah?

STUDENT: Is this p a choice that you made?

ERIK DEMAINE: OK, right. What is p? P just has to be bigger than m, and it should be prime. It's not random. You can just choose one prime that's bigger than your table size, and this will work.

STUDENT: [INAUDIBLE]

ERIK DEMAINE: I forget whether you have to assume that m is prime. I'd have to check. I'm guessing not, but don't quote me on that. Check the section in the textbook. So good. Easy to compute. The analysis is simpler, but it's a little bit easier here. Essentially this is very much like products but there's no carries here from one.

When we do the dot product instead of just multiplying in base m we multiply them based on that would give the same thing as multiplying in base 2, but we get carries from one m-sized digit to the next one. And that's just more annoying to think about. So here we're essentially getting rid of carries. So it's in some sense even easier to compute. And in both cases, it's universal.

So we want to prove this property. That if we choose a random a then the probability of two keys, k and k' which are distinct mapping via h to the same value is at most 1/m So let's prove that.

So we're given two keys. We have no control over them because this has to work for all keys

that are distinct. The only thing we know is that they're distinct. Now if two keys are distinct, then their vectors must be distinct. If two vectors are distinct, that means at least one item must be different. Should sound familiar. So this was like in the matrix multiplication verification algorithm that [INAUDIBLE] taught.

So k and k' differ in some digit. Let's call that digit d. So k sub d is different from k sub d'. And I want to compute this probability. We'll rewrite it. The probability is over a. I'm choosing a uniformly at random. I want another probability that that maps k and k' to the same slot.

So let me just write out the definition. It's probability over a that the dot product of a and k is the same thing as when I do the dot product with k' mod m. These two, that sum should come out the same, mod m. So let me move this part over to this side because in both cases we have the same ai. So I can group terms and say this is the probability-- probability sum over i equals 0 to r minus 1 of ai times ki minus ki prime equals 0. Mod m. OK, no pun intended.

Now we care about this digit d. d is a place where we know that this is non-zero. So let me separate out the terms for d and everything but d. So this is the same as ability of, let's do the d term first, so we have ad times kd minus kd prime. That's one term. I'm going to write the summation of i not equal to d of ai ki minus ki prime.

These ones, some of them might be zero. Some are not. We're not going to worry about it. It's enough to just isolate one term that is non-zero. So this thing we know does not equal zero. Cool. Here's where I'm going to use a little bit of number theory. I haven't yet used that m is prime. I required m is prime because when you're working modulo m, you have multiplicative inverses. Because this is not zero, there is something I can multiply on both sides and get this to cancel out and become one.

For every value x there is a value y. So x times y equals 1 modulo m. And you can even compute it in constant time in a reasonable model. So then I can say I want the probability that ad is minus kd minus kd prime inverse. This is the multiplicative inverse I was talking about.

And then the sum i not equal to d whatever, I don't actually care what this is too much, I've already done the equals part. I still need to write mod m. The point is this is all about ad. Remember we're choosing a uniformly at random. That's the same thing as choosing each of the ai's independently uniformly at random. Yeah?

STUDENT: Is the second line over there isolating d [INAUDIBLE]? Second from the top.

ERIK DEMAINE: Which? This one?

STUDENT: No up.

ERIK DEMAINE: This?

STUDENT: Down. That one. No. The one below that.

ERIK DEMAINE: Yes.

STUDENT: Is that line isolating d or is that--

ERIK DEMAINE: No. I haven't isolated d yet. This is all the terms. And then going from this line to this one, I'm just pulling out the i equals d term. That's this term. And then separating out the i not equal to d.

STUDENT: I get it.

ERIK DEMAINE: Right? This sum is just the same as that sum. But I've done the d term explicitly.

STUDENT: Sure. I get it.

ERIK DEMAINE: So I've done all this rewriting because I know that ad is chosen uniformly at random. Here we have this thing, this monstrosity, but it does not depend on ad. In fact it is independent of ad. I'm going to write this as a function of k and k' because those are given to us and fixed. And then it's also a function of a0 and a1. Everything except d. So ad minus 1, ad plus 1, and so on up to ar minus 1.

This is awkward to write. But everything except ad appears here because we have i not equal to d. And these ai's are random variables. But we're assuming that they're all chosen independently from each other. So I don't really care what's going on in this function. It's something. And if I rewrite this probability, it's the probability over the choice of a. I can separate out the choice of all these things from the choice of ad. And this is just a useful formula.

I'm going to write a not equal to d. All the other-- maybe I'll write a sub i not equal to d. All the choices of those guys separately from the probability of choosing ad of ad equal to this function. If you just think about the definition of expectation, this is doing the same thing. We're thinking of first choosing the ai's where i is not equal to d. And then we choose ad. And this

computational will come out the same as that.

But this is the probability of a uniformly random number equaling something. So we just need to think about-- sorry. Important. That would be pretty unlikely that would be 1/u, but this is all working modulo m. So if I just take a uniformly random integer and the chance of it hitting any particular value mod m is 1/m. And that's universality. So in this case, you get exactly 1/m, no less than or equal to. Sorry, I should have written it's the expectation of 1/m, but that's 1/m because 1/m has no random parts in it. Yeah?

STUDENT: How do we know that the, that this expression doesn't have any biases in the sense that it doesn't give more, more, like if you give it the uniform distribution of numbers, it doesn't spit out more numbers than others and that could potentially--

ERIK DEMAINE: Oh, so you're asking how do we know that this hash family doesn't prefer some slots over others, I guess.

STUDENT: Of course like after the equals sign, like in this middle line in the middle. Middle board.

ERIK DEMAINE: This one? Oh, this one.

STUDENT: Middle board.

ERIK DEMAINE: Middle board. Here.

STUDENT: Yes. So how do we know that if you give it--

ERIK DEMAINE: This function.

STUDENT: --random variables, it won't prefer certain numbers over others?

ERIK DEMAINE: So this function may prefer some numbers over others. But it doesn't matter. All we need is that this function is independent of our choice of ad. So you can think of this function, you choose all of these random-- actually k and k' are not random-- but you choose all these random numbers. Then you evaluate your f. Maybe it always comes out to 5.

Who knows. It could be super biased. But then you choose ad uniformly at random. So the chance of ad equalling 5 is the same as the chance of ad equaling 3. So in all cases, you get the probability is 1/m. What we need is independence. We need that the ad is chosen independently from the other ai's.

But we don't need to know anything about f other than it doesn't depend on ad. So and we made it not depend on ad because I isolated ad by pulling it out of that summation. So we know there's no ad's over here. Good question. You get a bonus Frisbee for your question.

All right. That ends universal hashing. Any more questions? So at this point we have at least one universal hash family. So we're just choosing, in this case, a uniformly at random. In the other method, we choose a and b uniformly at random. And then we build our hash table. And the hash function depends on m.

So also every time we double our table size, we're going to have to choose a new hash function for the new value of m. And that's about it. So this will give us constant expected time-- or in general 1 plus alpha if you're not doing table doubling-- for insert, delete, and exact search. Just building on the hashing with chaining. And so this is a good method. Question?

STUDENT: Why do you say expected value of the probability? Isn't it sufficient to just say the probability of [INAUDIBLE]?

ERIK DEMAINE: Uh, yeah, I wanted to isolate-- it is the overall probability of this happening. I rewrote it this way because I wanted to think about first choosing the ai's where i does not equal d and then choosing ad. So this probability was supposed to be only over the choice of ad. And you have to do something with the other ai's because they're random.

You can't just say, what's the probability ad equaling a random variable? That's a little sketchy. I wanted to have no random variables over all. So I have to kind of bind those variables with something. And I just want to see what the-- This doesn't really affect very much, but to make this algebraically correct I need to say what the ai's, i not equal to d are doing. Other questions? Yeah.

STUDENT: Um, I'm a bit confused about your definition of the collision in the lower left board. Why are you adding i's [INAUDIBLE]?

ERIK DEMAINE: Yeah, sorry. This is a funny notion of colliding. I just mean I want to count the number of keys that hash to the same slot as ki.

STUDENT: So it's not necessarily like a collision [INAUDIBLE].

ERIK DEMAINE: You may not call it a collision when it collides with itself, yeah. Whatever you want to call it. But I just mean hashing to the same slot is ki. Yeah. Just because I want to count the total length

of the chain. I don't want to count the number of collisions in the chain. Sorry. Probably a poor choice of word.

We're hashing because we're taking our key, we're cutting it up into little bits, and then we're mixing them up just like a good corned beef hash or something.

All right let's move on to perfect hashing. This is more exciting I would say. Even cooler-- this was cool from a probability perspective, depending on your notion of cool. This method will be cool from a data structures perspective and a probability perspective. But so far data structures are what we know from 006. Now we're going to go up a level, literally. We're going to have two levels.

So here we're solving-- you can actually make this data structure dynamic. But we're going to solve the static dictionary problem which is when you have no inserts and deletes. You're given the keys up front. You're given n keys. You want to build a table that supports search. And that's it.

You want search to be constant time and perfect hashing, also known as FKS hashing because it was invented by Fredman, Komlos, and Szemeredi in 1984. What we will achieve is constant time worst case for search.

So that's a little better because here we're just doing constant expected time for search. But it's worse in that we have to know the keys up in advance. We're going to take the linear space in the worst case. And then the remaining question is how long does it take you to build this data structure? And for now I'll just say it's polynomial time. It's actually going to be nearly linear.

And this is also an expected bounds. Actually with high probability could be a little more strong here. So it's going to take us a little bit of time to build this structure, but once you have it, you have the perfect scenario. There's going to be in some sense no collisions in our hash table so it would be constant times first search and linear space. So that part's great. The only catch is it's static. But beggars can't be choosers I guess.

All right. I'm not sure who's begging in that analogy but. The keys who want to be stored. I don't know. All right, so the big idea for perfect hashing is to use two levels. So let me draw a picture. We have our universe, and we're mapping that via hash function h1 into a table. Look familiar? Exactly the diagram I drew before.

It's going to have some table size m. And we're going to set m to be within a constant factor of n. So right now it looks exactly like regular-- and it's going to be a universal, h1 is chosen from a universal hash family, so universal hashing applies.

The trouble is we're going to get some lists here. And we don't want to store the set of colliding elements, the set of elements that hash to that place, with a linked list because linked lists are slow. Instead we're going to store them using a hash table. It sounds crazy.

But we're going to have-- so this is position 1. This is going to be h2,1. There's going to be another hash function h2,0 that maps to some other hash table. These hash tables are going to be of varying sizes. Some of them will be of size 0 because nothing hashes there. But in general each of these slots is going to map instead of to a linked list to a hash table.

So this would be h2, m minus 1. I'm going to guarantee in the second level of hashing there are zero collisions. Let that sink in a little bit. Let me write down a little more carefully what I'm doing.

So h1 is picked from a universal hash family. Where m is theta n. I want to put a theta-- I mean I could m equals n, but sometimes we require m to be a prime. So I'm going to give you some slop in how you choose m. So it can be prime or whatever you want. And then at the first level we're basically doing hashing with chaining.

And now I want to look at each slot in that hash table. So between 0 and m-1. I'm going to let lj be the number of keys that hash, it's the length of the list that would go there. It's going to be the number of keys, among just the n keys, Number of, keys hashing to slot j.

So now the big question is, if I have lj keys here, how big do I make that table? You might say, well I make a theta lj. That's what I always do. But that's not what I'm going to do. That wouldn't help. We get exactly, I think, the same number of collisions if we did that, more or less, in expectation. So we're going do something else.

We're going to pick a hash function from a universal family, h2,j. It again maps the same universe. The key thing is the size of the hash table I'm going to choose which is lj squared. So if there are 3 elements that happen to hash to this slot, this table will have size 9. So it's mostly empty. Only square root fraction-- if that's a word, if that's a phrase-- will be full. Most of it's empty. Why squared? Any ideas? I claim this will guarantee zero collisions with decent chance. Yeah.

STUDENT: With 1/2 probability you're going to end up with no collisions.

ERIK DEMAINE: With 1/2 probability I'm going to end up with no collisions. Why? What's it called?

STUDENT: Markov [INAUDIBLE]

ERIK DEMAINE: Markov's inequality would prove it. But it's more commonly known as the, whoa, as the birthday paradox. So the whole name of the game here is the birthday paradox. If I have, how's it go, if I have n squared people with n possible birthdays then-- is that the right way? No, less. If I have n people and n squared possible birthdays, the probability of getting a collision, a shared birthday, is 1/2. Normally we think of that as a funny thing. You know, if I choose a fair number of people, then I get immediately a collision.

I'm going to do it the opposite way. I'm going to guarantee that there's so many birthdays that no 2 of them will collide with probability of 1/2 No, 1/2 is not great. We're going to fix that. So actually I haven't given you the whole algorithm yet. There are two steps, 1 and 2. But there are also two other steps 1.5 and 2.5. But this is the right idea and this will make things work in expectation. But I'm going to tweak it a little bit.

So first let me tell you step 1.5. It fits in between the two. I want that the space of this data structure is linear. So I need to make sure it is. If the sum j equals 0 to m minus 1 of lj squared is bigger than some constant times n-- we'll figure out what the constant is later-- then redo step 1. So after I do step 1, I know how big all these tables are going to be. If the sum of those squares is bigger than linear, start over.

I need to prove that this will only have to take-- this will happen an expected constant number of times. log n times with high probability. In fact why don't we-- yeah, let's worry about that later. Let me first tell you step 2.5 which is I want there to be zero collisions in each of these tables. It's only going to happen with probability of 1/2 So if it doesn't happen, just try again.

So 2.5 is while there's some hash function h2,j that maps 2 keys that we're given to the same slot at the second level, this is for some j and let's say ki different from ki prime. But they map to the same place by the first hash function. So if two keys map to the same secondary table and there's a conflict, then I'm just going to redo that construction. So I'm going to repick h2,j. h2,j was a random choice.

So if I get a bad choice, I'll just try another one. Just keep randomly choosing the a or

randomly choosing this hash function until there are zero collisions in that secondary table. And I'm going to do this for each table. So we worry about how long these will take, but I claim expected constant number of trials.

So let's do the second one first. After we do this y loop there are no collisions with the proper notion of the word collisions, which is two different keys mapping to the same value. So at this point we have guaranteed that searches are constant time worst case after we do all these 4 steps because we apply h1, we figure out which slot we fit in.

Say it's slot j, then we apply h2j and if your item's in the overall table, it should be in that secondary table. Because there are no collisions you can see, is that one item the one I'm looking for? If so, return it. If not, it's not anywhere. If there are no collisions then I don't need chains coming out of here because it is just a single item.

The big question-- so constant worst case space because 1.5 guarantees that. Constant worst case time first search. The big question is, how long does it take to build? How many times do we have to redo steps 1 and 2 before we get a decent-- before we get a perfect hash table. So let me remind you of the birthday paradox, why it works here.

As mentioned earlier this is going to be a union bounds. We want to know the probability of collision at that second level. Well that's at most the sum of all possible collisions, probabilities of collisions. So I'm going to say the sum over all i not equal to ij of the probability. Now this is over our choice of the hash function h2,j. Of h2,j of ki equaling h2,j of ki prime.

So union bounds says, of course. The probability of any of them happening-- we don't know about interdependence or whatnot-- but certainly almost the sum of each of these possible events. There are a lot of possible events. If there 's li things, that there are going to be li choose 2 possible collisions we have to worry about. We know i is not equal to i prime. So the number of terms here is li choose 2. And what's this probability?

STUDENT: [INAUDIBLE]

ERIK DEMAINE: 1/li at most because we're assuming h2,j is a universal hash function so the probability of choosing-- sorry? li squared. Thank you. The size of the table. 1/m but m in this case, the size of our table is li squared. So the probability that we choose a good hash function and that these particular keys don't hit is at most 1/li squared. This is basically li squared/ 2. And so this is at most 1/2. It's a slightly less than li squared/2. So this is at most 1/2. And this is basically a

birthday paradox in this particular case.

That means there is a probability of at least a half that there is zero collisions in one of these tables. So that means I'm basically flipping a fair coin. If I ever get a heads I'm happy. Each time I get a tails I have to reflip. This should sound familiar from last time. So this is 2 expected trials or log n with high probability.

We've proved log n with high probability. That's the same as saying the number of levels in a skip list is log n with high probability. How many times do I have to flip a coin before I get a heads? Definitely at most log n. Now we have to do this for each secondary table. There are m equal theta and secondary tables.

There's a slight question of how big are the secondary tables. If one of these tables is like linear size, then I have to spend linear time for a trial. And then I multiply that by the number of trials and also the number of different things that would be like n squared log n n.

But you know a secondary table better not have linear sides. I mean a linear number of li equal n. That would be bad because then li squared is n squared and we guaranteed that we had linear space. So in fact you can prove with another Chernoff bound.

Let me put this over here. That all the li's are pretty small. Not constant but logarithmic. So li is order log n with high probability for each i and therefore for all i. So I can just change the alpha my minus 1 n to the alpha and get that for all i this happens. In fact, the right answer is log over log log, if you want to do some really messy analysis. But we just, logarithmic is fine for us.

So what this means is we're doing n different things for each of them with high probability li is of size log n. And then maybe we'll have to do like log n trials repeating until we get a good hash function there. And so the total build time for steps 1 and 2.5 is going to be at most n times log squared n. You can prove a tighter bound but it's polynomial. That's all I wanted to go for and it's almost linear.

So I'm left with one thing to analyze which is step 1.5. This to me is maybe the most surprising thing that it works out. I mean here we designed-- we did this li to li squared so the birthday paradox would happen. This is not surprising. I mean it's a cool idea, but once you have the idea, it's not surprising that it works. What's a little more surprising is that squaring is OK from a space perspective.

1.5 says we're going to have to rebuild that first table until the sum of these squared lengths is at most linear. I can guarantee that each of these is logarithmic so the sum of the squares is at most like n log squared n. But I claim I can get linear. Let's do that.

So for step 1.5 we're looking at what is the expectation of the sum of the lj squareds being more than linear. Sorry. Expectation. Let's first compute the expectation and then we'll talk about a tail bound which is the probability that we're much bigger than the expectation.

First thing is I claim the expectation is linear. So again whenever we're counting something-- I mean this is basically the total number of pairs of items that collide at the first level with double counting. So I mean if you think of lj and then I make a complete graph on those lj items, that's going to have like the squared number of edges, so, if I also multiply by 2.

So this is the same thing as counting how many pairs of items map to the same spot, the same slot. So this is going to-- and that I can write as an indicator random variable which lets me use linearity of expectation which makes me happy because then everything simple. So I'm going to write Ii,j. This is going to be 1 if each 1 of ki, I guess, equals h1 if kj and it's going to be 0 if h1 otherwise. This is the total number of pairwise colliding items including i versus i.

And so like if li equals 1, li squared is also 1. There's 1 item colliding with itself. So this actually works exactly. All right, with the wrong definition of colliding. If you bear with me. So now we can use linear of expectation and put the E in here. So this is sum i equals 1 to n sum j equals 1 to n of the expectation of Ii,j.

But we know the expectation of the Ii,j is the probability of it equaling 1 because it's an indicator random variable. The probability of this happening over our choice of h1 is at most 1/m by universality. Here it actually is m because we're at the first level. So this is at most 1/m which is theta n.

So when i does not equal j, so it's a little bit annoying. I do have to separate out the Ii terms from the i and different i not equal to j terms. But there's only-- I mean it's basically the diagonal of this matrix. There's n things that will always collide with themselves. So we're going to get like n plus the number of i not equal to pairs double counted. So it's like 2 times n choose 2. But we get to divide by m. So this is like n squared /n. So we get order n. So that's not-- well, that's cool. Expected space is linear. This is what makes everything work.

Last class was about getting with high probability bounds when we're working with logs. When

you want to get that something is log with high probability, you have to use, with respect to n, you have to use a turn off bound. But this is about-- now I want to show that the space is linear with high probability. Linear is actually really easy. You can use a much weaker bound called Markov inequality.

So I want to claim that the probability of h1 of this thing lj squareds being bigger than some constant times n is at most the expectation of that thing divided by cn. This is Markov's inequality. It holds for anything here. So I'm just repeating it over here. So this is nice because we know that this expectation is linear. So we're getting like a linear function divided by cn. Remember we get to choose c.

The step said if it's bigger than some constant times n then we're redoing the thing. So I can choose c to be 100, whatever. I'm going to choose it to be twice this constant. And then this is at most half. So the probability of my space being too big is at most a half. We're back to coin flipping. Every time I flip a coin, if I get heads I have the right amount of space at less than c times n space. If I get a tails I try again. So the expected number of trials is 2 at most not trails, trials. And it's also log n trials with high probability.

How much time do I spend for each trial? Linear time. I choose one hash function. I hash all the items. I count the number of collision squared or the sum of lj squared. That takes linear time to do. And so the total work I'm doing for these steps is n log n. So n log n to do step 1 and 1 prime and log squared n to do steps 2 and 2 prime. Overall n Polylog or polynomial time. And we get guaranteed no collisions for static data. Constant worst case search and linear worst case space. This is kind of surprising that this works out but everything's nice. Now you know hashing.