**JASON KU:** Welcome to the fourth lecture of 6.006. Today we are going to be talking about hashing. Last lecture, on Tuesday, Professor Solomon was talking about set data structures, storing things so that you can query items by their key right, by what they intrinsically are-- versus what Professor Demaine was talking about last week, which was sequence data structures, where we impose an external order on these items and we want you to maintain those.

I'm not supporting operations where I'm looking stuff up based on what they are. That's what the set interface is for. So we're going to be talking a little bit more about the set interface today. On Tuesday, you saw two ways of implementing the set interface-- one using just a unsorted array-- just, I threw these things in an array and I could do a linear scan of my items to support basically any of these operations.

It's a little exercise you can go through. I think they show it to you in the recitation notes, but if you'd like to implement it for yourself, that's fine. And then we saw a slightly better data structure, at least for the find operations. Can I look something up, whether this key is in my set interface?

We can do that faster. We can do that in log n time with a build overhead that's about n log n, because we showed you three ways to sort. Two of them were n squared. One of them was n log n, which is as good as we showed you how to do yesterday.

So the question then becomes, can I build that data structure faster? That'll be a subject of next week's Thursday lecture. But this week we're going to concentrate on this static find. we got log n, which is an exponential improvement over linear right, but the question now becomes, can I do faster than log n time? And what we're going to do at the first part of this lecture is show you that, no, you--

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** What's up? No? OK-- that you can't do faster than log n time, in the caveat that we are in a slightly more restricted model of computation that we were-- than what we introduce to you a couple of weeks ago. And then so if we're not in that more constrained model of computation, we can actually do faster.

Log n's already pretty good. Log n is not going to be larger than like 30 for any problem that you're going to be talking about in the real world on real computers, but a factor of 30 is still bad. I would prefer to do faster with those constant factors, when I can. It's not a constant factor. It's a logarithmic factor, but you get what I'm saying.

OK, so what we're going to do is first prove that you can't do faster for-- does everyone understand-- remember what find key meant? I have a key, I have a bunch of items that have keys associated with them, and I want to see if one of the items that I'm storing contains a key that is the same as the one that I searched for.

The item might contain other things, but in particular, it has a search key that I'm maintaining the set on so that it supports find operations, search operations based on that key quickly. Does that make sense? So there's the find one that we want to improve, and we also want to improve this insert delete. We want to be-- make this data structural dynamic, because we might do those operations quite a bit. And so this lecture's about optimizing those three things.

OK, so first, I'm going to show you that we can't do faster than log n for find, which is a little weird. OK, the model of computation I'm going to be proving this lower bound on-- how I'm going to approach this is I'm going to say that any way that I store these-- the items that I'm storing in this data structure-- for anyway I saw these things, any algorithm of this certain type is going to require at least logarithmic time. That's what we're going to try to prove.

And the model of computation that's weaker than what we've been talking about previously is what I'm going to call the comparison model. And a comparison model means-- is that the items, the objects I'm storing-- I can kind of think of them as black boxes. I don't get to touch these things, except the only way that I can distinguish between them is to say, given a key and an item, or two items, I can do a comparison on those keys.

Are these keys the same? Is this key bigger than this one? Is it smaller than this one? Those are the only operations I get to do with them. Say, if the keys are numbers, I don't get to look at what number that is. I just get to take two keys and compare them. And actually, all of the search algorithms that we saw on Tuesday we're comparison sort algorithms.

What you did was stepped through the program. At some point, you came to a branch and you looked at two keys, and you branched based on whether one key was bigger than another. That was a comparison. And then you move some stuff around, but that was the general paradigm. Those three sorting operations lived in this comparison model.

You've got a comparison operations, like are they equal, less than, greater than, maybe greater than or equal, less than or equal? Generally, you have all these operations that you could do-- maybe not equal. But the key thing here is that there are only two possible outputs to each of these comparitors. There's only one thing that I can branch on. It's going to branch into two different lines.

It's either true and I do some other computation, or it's false and I'll do a different set of computation. That makes sense? So what I'm going to do is I'm going to give you a comparison-- an algorithm in the comparison model as what I like to call a decision tree.

So if I specify an algorithm to you, the first thing it's going to do-- if I don't compare items at all, I'm kind of screwed, because I'll never be able to tell if my keys in there or not. So I have to do some comparisons. So I'll do some computation. Maybe I find out the length of the array and I do some constant time stuff, but at some point, I'll do a comparison, and I'll branch.

I'll come to this node, and if the comparison-- maybe a less than-- if it's true, I'm going to go this way in my computation, and if it's false, I'm going to go this way in my computation. And I'm going to keep doing that with various comparisons-- sure-- until I get down here to some leaf in which I I'm not branching.

The internal nodes here are representing comparisons, but the leaves are representing-- I stopped my computation. I'm outputting something. Does that make sense, what I'm trying to do? I'm changing my algorithm to be put in this kind of graphical way, where I'm branching what my program could possibly do based on the comparisons that I do.

I'm not actually counting the rest of the work that the program does. I'm really only looking at the comparisons, because I know that I need to compare some things eventually to figure out what my items are. And if that's the only way I can distinguish items, then I have to do those comparisons to find out. Does that make sense?

All right, so what I have is a binary tree that's representing the comparisons done by the algorithm. OK. So it starts at one comparison and then it branches. How many leaves must I have in my tree? What does that question mean, in terms of the program?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** What's up?

**AUDIENCE:** The number of comparisons--

**JASON KU:** The number of comparisons-- no, that's the number of internal nodes that I have in the algorithm. And actually, the number of comparisons that I do in an execution of the algorithm is just along a path from here to the-- to a leaf. So what do the leaves actually represent? Those represent outputs. I'm going to output something here. Yep?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** The number of-- OK. So what is the output to my search algorithm? Maybe it's the-- an index of an item that contains this key. Or maybe I return the item is the output-- the item of the thing I'm storing. And I'm storing n things, so I need at least n outputs, because I need to be able to return any of the items that I'm storing based on a different search parameter, if it's going to be correct.

I actually need one more output. Why do I need one more output? If it's not in there-- so any correct comparison searching algorithm-- I'm doing some comparisons to find this thing-- needs to have at least n plus 1 leaves. Otherwise, it can't be correct, because I could look up the one that I'm not returning in that set and it would never be able to return that value. Does that make sense? Yeah?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** What's n? For a data structure, n is the number of things stored in that data structure at that time-- so the number of items in the data structure. That's what it means in all of these tables. Any other questions? OK, so now we get to the fun part. How many comparisons does this algorithm have to do? Yeah, up there--

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** What's up? All right, your colleague is jumping ahead for a second, but really, I have to do as many comparisons in the worst case as the longest root-to-leaf path in this tree-- because as I'm executing this algorithm, I'll go down this thing, always branching down, and at some point, I'll get to a leaf. And in the worst case, if I happen to need to return this particular output, then I'll have to walk down the longest thing, just the longest path.

So then the longest path is the same as the height of the tree, so the question then becomes, what is the minimum height of any binary tree that has at least n plus 1 leaves? Does everyone understand why we're asking that question? Yeah?

**AUDIENCE:** Could you over again why it needs n plus 1 leaves?

**JASON KU:** Why it needs n plus 1 leaves-- if it's a correct algorithm, it needs to return-- it needs to be able to return any of the n items that I'm storing or say that the key that I'm looking for is not there-- great question. OK, so what is the minimum height of any binary tree that has n plus 1-- at least n plus 1 leaves?

You can actually state a recurrence for that and solve that. You're going to do that in your recitation. But it's log n. The best you can do is if this is a balanced binary tree. So the min height is going to be at least log n height. Or the min height is logarithmic, so it's actually theta right here.

But if I just said height here, I would be lower bounding the height. I could have a linear height, if I just changed comparisons down one by one, if I was doing a linear search, for example. All right, so this is saying that, if I'm just restricting to comparisons, I have to spend at least logarithmic time to be able to find whether this key is in my set. But I don't want logarithmic time. I want faster. So how can I do that?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** I have one operation in my model of computation I presented a couple of weeks ago that allows me to do faster, which allows me to do something stronger than comparisons. Comparisons have a constant branching factor. In particular, I can-- if I do this operation-- this constant time operation-- I can branch to two different locations.

It's like an if kind of situation-- if, or else. And in fact, if I had constant branching factor for any constant here-- if I had three or four, if it was bounded by a constant, the height of this tree would still be bounded by a log base the constant of that number of leaves.

So I need, in some sense, to be able to branch a non-constant amount. So how can I branch a non-constant amount? This is a little tricky. We had this really neat operation in the random access machine that we could randomly go to any place in memory in constant time based on a number.

That was a super powerful thing, because within a single constant time operation, I could go to any space in memory. That's potentially much larger than linear branching factor, depending on the size of my model and the size of my machine. So that's a very powerful operation. Can we use that to find quicker? Anyone have any ideas? Sure.

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** We're going to get to hashing in a second, but this is a simpler concept than hashing-- something you probably are familiar with already. We've kind of been using it implicitly in some of our sequence data structure things. What we're going to do is, if I have an item that has key 10, I'm going to keep an array and store that item 10 spaces away from the front of the array, right at index 9, or the 10th index.

Does that make sense? If I store that item at that location in memory, I can use this random access to that location and see if there's something there. If there's something there, I return that item. Does that make sense? This is what I call a direct access array. It's really no different than the arrays that we've been talking about earlier in the class.

We got an array, and if I have an item here with key equals 10, I'll stick it here in the 10th place. Now, I can only now store one item with the key 10 in my thing, and that's one of the stipulations we had on our set data structures. If we tried to insert something with the same key as something already stored there, we're going to replace the item. That's what the semantics of our set interface was.

But that's OK. That's satisfying the conditions of our set interface. So if we store it there, that's fantastic. How long does it take to find, if we have an item with the key 10? It takes constant time, worst case-- great. How about inserting or deleting something?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** What's that?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** Again, constant time-- we've solved all our problems. This is amazing. OK. What's not amazing about this? Why don't we just do this all the time? Yeah?

**AUDIENCE:** You don't know how high the numbers go.

**JASON KU:** I don't know how high the numbers go. So let's say I'm storing, I don't know, a number associated with that the 300 or 400 of you that are in this classroom. But I'm storing your MIT IDs. How big are those numbers? Those are like nine-digit numbers-- pretty long numbers.

So what I would need to do-- and if I was storing your keys as MIT IDs, I would need an array that has indices that span the tire space of nine-digit numbers. That's like 10 to the-- 10 to the 9. Thank you. 10 to the 9 is the size of a direct access road off to build to be able to use this technique to create a direct access array to search on your MIT IDs, when there's only really 300 of you in here.

So 300 or 400 is an n that's much smaller than the size of the numbers that I'm trying to store. What I'm going to use as a variable to talk about the size of the numbers I'm storing-- I'm going to say u is the maximum size of any number that I'm storing. It's the size of the universe of space of keys that I'm storing. Does that make sense?

OK, so to instantiate a direct access array of that size, I have to allocate that amount of space. And so if that is much bigger than n, then I'm kind of screwed, because I'm using much more space. And these order operations are bad also, because essentially, if I am storing these things non-continuously, I kind of just have to scan down the thing to find the next element, for example. OK, what's your question?

**AUDIENCE:** Is a direct access array a sequence data structure?

**JASON KU:** A direct access array is a set data structure. That's why it's a set interface up there. Your colleague is asking whether you can use a direct accessory to implement a set-- I mean a sequence. And actually, I think you'll see in your recitation notes, you have code that can take a set data structure and implement sequence data structure, and take sequence data structure and implement a set data structure.

They just won't necessarily have very good run time. So this direct access array semantics is really just good for these specific set operations. Does that makes sense? Yeah?

**AUDIENCE:** What is u?

**JASON KU:** u is this the size of the largest key that I'm allowed to store. That makes sense? The direct access array is supporting up to u size keys. Does that make sense? OK, we're going to move on for a second. That's the problem, right? When u largest key-- we're assuming integers here-- integer keys-- so in the comparison model, we could store any arbitrary objects that supported a comparison.

Here we really need to have integer keys, or else we're not going to be able to use those as addresses. So we're making an assumption on the inputs that I can only store integers now. I can't store arbitrary objects-- items with keys. And in particular, I also need to-- this is a subtlety that's in the word RAM model-- how can I be assured that these keys can be looked up in constant time? I have this little CPU. It's got some number of registers it can act upon. How big is those registers?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** What? Right now, they're 64 bits, but in general, they're w. They're the size of your word on your machine. 2 to the w is the number of dresses I can access. If I'm going to be able to use this direct accessory, I need to make sure that the u is less than 2 to the w, if I want these operations to run in constant time.

If I have kids that are much larger than this, I'm going to need to do something else, but this is kind of the assumption. In this class, when we give you an array of integers, or an array of strings, or something like that on your problem or on an exam, the assumption is, unless we give you bounds on the size of those things-- like the number of characters in your string or the size of the number in the-- you can assume that those things will fit in one word of memory.

w is the word size of your machine, the number of bits that your machine can do operations on in constant time. Any other questions? OK, so we have this problem. We're using way too much space, when we have a large universe of keys. So how do we get around that Problem any ideas? Sure.

**AUDIENCE:** Instead of [INAUDIBLE].

**JASON KU:** OK, so what your colleague is saying-- instead of just storing one value at each place, maybe store more than one value. If we're using this idea, where I am storing my key at the index of the key, that's getting around the us having to have unique keys in our data structure. It's not getting around this space usage problem. Does that make sense?

We will end up storing multiple things at indices, but there's another trick that I'm looking for right now. We have a lot of space that we would need to allocate for this data structure. What's an alternative? Instead of allocating a lot of space, we allocate-- less space. Let's allocate less space.

All right. This is our space of keys, u. But instead, I want to store those things in a direct access array of maybe size n, something like the order of the things that I'm going to be storing. I'm going to relax that and say we're going to make this a length m that's around the size of the things I'm storing.

And what I'm going to do is I'm going to try to map this space of keys-- this large space of keys, from 0 to u minus 1 or something like that-- down to arrange that 0 to m minus 1. I'm going to want a function-- this is what I'm going to call h-- which maps this range down to a smaller range. Does that make sense?

I'm going to have some function that takes that large base of keys-- sticks them down here. And instead of staring at an index of the key, I'm going to put the key through this function, the key space, into a compressed space and store it at that index location. Does that make sense? Sure.

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** Your colleague is-- comes up with the question I was going to ask right away, which was, what's the problem here? The problem is it's the potential that we might be-- have to store more than one thing at the same index location.

If I have a function that matches this big space down to this small space, I got to have multiple of these things going to the same places here, right? It can't be objective. But just based on pigeonhole principle, I have more of these things. At least two of them have to go to something over here. In fact, if I have, say, u is bigger than n squared, for example, there-- for any function I give you that maps this large space down to the small space, n of these things will map to the same place.

So if I choose a bad function here, then I'll have to store n things at the same index location. And if I go there, I have to check to see whether any of those are the things that I'm looking for. I haven't gained anything. I really want a hash function that will evenly distribute keys over this space.

Does that make sense? But we have a problem here. If we need to store multiple things at a given location in memory-- can't do that. I have one thing I can put there. So I have two options on how to deal-- what I call collisions. If I have two items here, like a and b, these are different keys in my universe of space.

But it's possible that they both map down to some hash that has the same value. If I first hash a, and a is-- I put a there, where do I put b? There are two options.

**AUDIENCE:** Is the second data structure [INAUDIBLE] so that it can store [INAUDIBLE]?

**JASON KU:** OK, so what your colleague is saying-- can I store this one is a linked list, and then I can just insert a guy right next to where it was? What's the problem there? Are linked lists good with direct accessing by an index? No, they're terrible with get_at and set_at They take linear time there.

So really, the whole point of direct this array is that there is an array underneath, and I can do this index arithmetic and go down to the next thing. So I really don't want to replace a linked list as this data structure. Yeah? What's up?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** We can make it really unlikely. Sure. I don't know what likely means, because I'm giving you a hash function-- one hash function. And I don't know what the inputs are. Yeah? Go ahead.

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** OK, right. So there are actually two solutions here. One is I-- maybe, if I choose m to be larger than n, there's going to be extra space in here. I'll just stick it somewhere else in the existing array. How I find an open space is a little complicated, but this is a technique called open addressing, which is much more common than the technique we're going to be talking about today in implementations.

Python uses an open addressing scheme, which is essentially, find another place in the array to put this collision. Open addressing is notoriously difficult to analyze, so we're not going to do that in this class. There's a much easier technique that-- we have an implementation for you in the recitation handouts.

It's what your colleague up here-- I can't find him-- over there was saying-- was, instead of storing it somewhere else in the existing direct access array down here, which we usually call the hash table-- instead of storing it somewhere else in that hash table, we'll instead, at that key, store a pointer to another data structure, some other data structure that can store a bunch of things-- just like any sequence data structure, like a dynamic array, or linked list, or anything right.

All I need to do is be able to stick a bunch of things on there when there are collisions, and then, when I go up to look for that thing, I'll just look through all of the things in that data structure and see if my key exists. Does that make sense? Now, we want to make sure that those additional data structures, which I'll call chains-- we want to make sure that those chains are short. I don't want them to be long.

So what I'm going to do is, when I have this collision here, instead I'll have a pointer to some-- I don't know-- maybe make it a dynamic array, or a linked list, or something like that. And I'll put a here and I'll b here. And then later, when I look up key K, or look up a or b-- let's look up b-- I'll go to this hash value here. I'll put it through the hash function.

I'll go to this index. I'll go to the data structure, the chain associated to that index, and I'll look at all of these items. I'm just going to do a linear find. I'm going to look. I could put any data structure here, but I'm going to look at this one, see if it's b. It's not b. Look at this one-- it is b. I return yes. Does that make sense?

So this is an idea called chaining. I can put anything I want there. Commonly, we talk about putting a linked list there, but you can put a dynamic array there. You can put a sorted array there to make it easier to check whether the key is there. You can put anything you want there.

The point of this lecture is going to try to show that there's a choice of hash function I can make that make sure that these chains are small so that it really doesn't matter how I saw them there, because I can just-- if there's a constant number of things stored there, I can just look at all of them and do whatever I want, and still get constant time. Yeah?

**AUDIENCE:** So does that means that, when you have [INAUDIBLE] let's just say, for some reason, the number of things [INAUDIBLE] is that most of them get multiple [INAUDIBLE]. Is it just a data structure that only holds one thing?

**JASON KU:** Yeah. So what your colleague is saying is, at initialization, what is stored here? Initially, it points to an empty data structure. I'm just going to initialize all of these things to have-- now, you get some overhead here. We're paying something for this-- some extra space and having pointer and another data structure at all of these things.

Or you could have the semantics where, if I only have one thing here, I'm going to store that thing at this location, but if I have multiple, it points to a data structure. These are kind of complicated implementation details, but you get the basic idea. If I just have a 0 size data structure at all of these things, I'm still going to have a constant factor overhead. It's still going to be a linear size data structure, as long as m is linear in n. Does that makes sense?

OK. So how do we pick a good hash function? I already told you that any fixed hash function I give you is going to experience collisions. And if u is large, then there's the possibility that I-- for some input, all of the things in my set go directly to the same hashed index value. So that ain't great.

Let's ignore that for a second. What's the easiest way to get down from this large space of keys down to a small one? What's the easiest thing you could do? Yeah?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** Modulus-- great. This is called the division method. And what its function is is essentially, it's going to take a key and it's going to say equal to be K mod m. I'm going to take something of a large space, and I'm going to mod it so that it just wraps around-- perfectly valid thing to do.

It satisfies what we're doing in a hash table. And if my kids are completely uniformly distributed-- if, when I use my hash function, all of the keys here are uniformly distributed over this larger space, then actually, this isn't such a bad thing. But that's imposing some kind of distribution requirements on the type of inputs I'm allowed to use with this hash function for it to have good performance.

But this plus a little bit of extra mixing and bit manipulation is essentially what Python does. Essentially, all it does is jumbles up that key for some fixed amount of jumbling, and then mods it m, and sticks it there. It's hard coded in the Python library, what this hash function is, and so there exist some sequences of inserts into a hash table in Python which will be really bad in terms of performance, because these chain links are the amount number of collisions that I'll get at a single hash is going to be large.

But they do that for other reasons. They want a deterministic hash function. They want something that I do the program again-- it's going to do the same thing underneath. But sometimes Python gets it wrong. But if your data that you're storing is sufficiently uncorrelated to the hash function that they've chosen-- which, usually, it is-- this is a pretty good performance.

But this is not a practical class. Well, it is a practical class, but one of the things that we are-- that's the emphasis of this class is making sure we can prove that this is good in theory as well. I don't want to know that sometimes this will be good. I really want to know that, if I choose-- if I make this data structure and I put some inputs on it, I want a running time that is independent on what inputs I decided to use, independent of what keys I decided to store.

Does that makes sense? But it's impossible for me to pick a fixed hash function that will achieve this, because I just told you that, if u is large-- this is u-- if u is large, then there exists inputs that map everything to one place. I'm screwed, right? There's no way to solve this problem.

That's true if I want a deterministic hash function-- I want the thing to be repeatable, to do the same thing over and over again for any set of inputs. What can I do instead? Weaken my notion of what constant time is to do better-- OK, use a non-deterministic-- what does non-deterministic mean?

It means don't choose a hash function up front-- choose one randomly later. So have the user-- they pick whatever inputs they're going to do, and then I'm going to pick a hash function randomly. They don't know which hash function I'm going to pick, so it's hard for them to give me an input that's bad.

I'm going to choose a random hash function. Can I choose a hash function from the space of all hash functions? What is the space of all hash functions of this form? For every one of these values, I give a value in here. For each one of these independently random number between this range, how many such hash functions are there?

m to the this number-- that's a lot of things. So I can't do that. What I can do is fix a family of hash functions where, if I choose one from-- randomly, I get good performance. And so the hash function I'm going to use, and we're going to spend the rest of the time on, is what I call a universal hash function. It satisfies what we call a universal hash property-- so universal hash function.

And this is a little bit of a weird nomenclature, because I'm defining this to you as the universal hash function, but actually, universal is a descriptor. There exist many universal hash functions. This just happens to be an example of one of them. OK?

So here's the hash function-- doesn't look actually all that different. Goodness gracious-- how many parentheses are there-- mod p, mod m. OK. So it's kind of doing the same thing as what's happening up here, but before modding by m, I'm multiplying it by a number, I'm adding a number, I'm taking it mod another number, and then I'm getting by m.

This is a little weird. And not only that-- this is still a fixed hash function. I don't want that. I want to generalize this to be a family of hash functions, which are this habk for some random choice of a, b in this larger range. All right, this is a lot of notation here. Essentially what this is saying is, I have a has family. It's parameterized by the length of my hash function and some fixed large random prime that's bigger than u.

I'm going to pick some large prime number, and that's going to be fixed when I make the hash table. And then, when I instantiate the hash table, I'm going to choose randomly one of these things by choosing a random a and a random b from this range. Does that makes sense?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** This is a not equal to 0. If I had 0 here, I lose the key information, and that's no good. Does this make sense? So what this is doing is multiplying this key by some random number, adding some random number, modding by this prime, and then modding by the size of my thing.

So it's doing a bunch of jumbling, and there's some randomness involved here. I'm choosing the hash function by choosing an a, b randomly from this thing. So when I start up my program, I'm going to instantiate this thing with some random a and b, not deterministically. The user, when they're using this thing, doesn't know which a and b I picked, so it's really hard for them to give me a bad example.

And this universal hash function-- this universal hash family, shall we say-- really, this is a family of functions, and I'm choosing one randomly within that family-- is universal. And universality says that-- what is the property of universality? It means that the probability, by choosing a hash function from this hash family, that a certain key collides with another key is less than or equal to 1/m for all-- any different two keys in my universe. Does that make sense?

Basically, this thing has the property that, if I randomly-- for any two keys that I pick in my universe space, if I randomly choose a hash function, the probability that these things collide is less than 1/m. Why is that good? This is, in some sense, a measure of how well distributed these things are.

I want these things to collide with 1/m probability so that these things don't collide very-- it's not very likely for these things to collide. Does that make sense? So we want proof that this hash family satisfies this universality property. You'll do that in 046. But we can use this result to show that, if we use a universal-- this universal hash family, that the length of our change-- chains is expected to be constant length. So we're going to use this property to prove that.

How do we prove that? We're going to do a little probability. So how are we going to prove that? I'm going to define a random variable, an indicator random variable. Does anyone remember what an indicator in a variable is? Yeah, it's a variable that, with some amount of probability, is 1, and 1 minus that probability is 0.

So I'm going to define this indicator random variable xij is a random variable over my choice-- over choice of a hash function in my has family. And what does this mean? It means xij equals 1, if hash Ki equals hKj-- these things collide-- and 0 otherwise.

So I'm choosing randomly over this hash family. If, for two keys-- key i and and j-- if these things collide, that's going to be 1. If they don't, then it's 0. OK? Then, how can we write a formula for the length of a chain in this model? So the size of a chain-- or let's put it here-- the size of the chain at i-- at i in my hash table-- is going to equal-- I'm going to call that the random variable xi-- that's going to equal the sum over j equals 0 to-- what is it-- over, I think, u minus 1 of summation-- or sorry-- of xij.

So basically, if I fix this location i, this is where this key goes. Sorry. This is the size of chain at h of Ki. Sorry. So I look at wherever Ki goes is hashed, and I see how many things collide with it. I'm just summing over all of these things, because this is 1 if there's a collision and 0 if there's not. Does that make sense? So this is the size of the chain at the index location mapped to by Ki.

So here's where your probability comes in. What's the expected value of this chain length over my random choice? Expected value of choosing a hash function from this universal hash family of this chain length-- I can put in my definition here. That's the expected value of the summation over j of xij. What do I know about expectations and summations? If these variables are independent from each other--

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** Say what?

**AUDIENCE:** [INAUDIBLE]

**JASON KU:** Linearity of expectation-- basically, the expectation sum of these independent random variables is the same as the summation of their expectations. So this is equal to the summation over j of the expectations of these individual ones.

One of these j's is the same as i. j loops over all of the things from 0 to u minus 1. One of them is i, so when xhi is hj, what is the expected value that they collide? 1-- so I'm going to refactor this as being this, where j does not equal i, plus 1. Are people OK with that? Because if i equals-- if j and i are equal, they definitely collide. They're the same key.

So I'm expected to have one guy there, which was the original key, xi. But otherwise, we can use this universal property that says, if they're not equal and they collide-- which is exactly this case-- the probability that that happens is 1/m. And since it's an indicator random variable, the expectation is there are outcomes times their probabilities-- so 1 times that probability plus 0 times 1 minus that probability, which is just 1/m.

So now we get the summation of 1/m for j not equal to i plus 1. Oh, and this-- sorry. I did this wrong. This isn't u. This is n. We're storing n keys. OK, so now I'm looping over j-- this over all of those things. How many things are there? n minus 1 things, right?

So this should equal 1 plus n minus 1 over m. So that's what universality gives us. So as long as we choose m to be larger than n, or at least linear in n, then we're expected to have our chain lengths be constant, because this thing becomes a constant if m is at least order n. Does that make sense?

OK. The last thing I'm going to leave you with is, how do we make this thing dynamic? If we're growing the number of things we're storing in this thing, it's possible that, as we grow n for a fixed m, this thing will stop being-- m will stop being linear in n, right? Well, then all we have to do is, if we get too far, we rebuild the entire thing-- the entire hash table with the new m, just like we did with a dynamic array.

And you can prove-- we're not going to do that here, but you can prove that you won't do that operation too often, if you're resizing in the right way. And so you just rebuild completely after a certain number of operations. OK, so that's hashing. Next week, we're going to be talking about doing a faster sort.