The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: All right. Let's get started. Today we start a brand new section of 006, which is hashing. Hashing is cool. It is probably the most used and common and important data structure and all of computer science. It's in, basically, every system you've ever used, I think. And in particular, it's in Python as part of what makes Python fun to program in. And basically, every modern programming language has it.

So today is about how to make it actually happen. So what is it? It is usually called a dictionary. So this is an abstract data if you remember that term from a couple lectures ago. It's kind of an old term, not so common anymore, but it's useful to think about.

So a dictionary is a data structure, or it's a thing, that can store items, and it can insert items, delete items and search for items. So in general, it's going to be a set of items, each item has a key. And you can insert an item, you can delete an item from the set, and you can search for a key, not an item.

And the interesting part is the search. I think you know what insert and delete do. So there are two outcomes to this kind of search. This is what I call an exact search. Either you find an item with a given key, or there isn't one, and then you just say key error in Python.

OK. This is a little different from what we could do with binary search trees. Binary search trees, if we didn't find a key, we could find the next larger and the next smaller successor and predecessor. With dictionaries you're not allowed to do that, or you're not able to do that.

And you're just interested in the question does the key exist? And if so, give me the item with that key. So we're assuming here that the items have unique keys, no two

items have the same key.

And one way to enforce that is when you insert an item with an existing key, it overwrites whatever key was there. That's the Python behavior. So we'll assume that. Overwrite any existing key. And so, it's well defined what search does. Either there's one item with that key, or there's no item with that key, and it tells you what the situation is. OK.

So one way to solve dictionaries is to use a balanced binary search tree like AVL trees. And so you can do all of these operations on log n time. I mean, you can ignore the fact that AVL trees give you more information when you do a search, and still does exact search.

So that's one solution, but it turns out you can do better. And while last class was about, well, in the comparison model the best way to sort is n log n and the best way to search is log n. Then we saw in the RAM model, where if you assume your items are integers we can sort faster, sometimes we can sort in linear time.

Today's lecture is about how to search faster than log n time. And we're going to get down to constant time. No-- basically, no assumptions except, maybe, that your keys are integers. We'll be able to get down to constant time with high probability.

It's going to be a randomized data structure. It's one of the few instances of randomization in 006, but it'll be pretty simple to analyze, so don't worry. But we're going to use some probability today. Make it a little exciting.

I think you know how dictionaries work in Python. In Python it's the dict data type. We've used it all over the place. The key things you can do are lookup a key and-- so this is the analog of search-- you can set a key to a value. This is the analog of an insert. It overwrites whatever was there. And what else? Delete. So you can delete a particular key. OK.

We'll usually use this notation because it's more familiar and intuitive. But the big topic today is how do you actually implement these operations for a dictionary, D? The one specific thing about Python dictionaries is that an item is basically a pair of

two things, a key and a value. And so, in particular, when you call d.items you get a whole bunch of ordered pairs, a key and a value.

And so the key is always-- the key of an item is always this first part. So it's well defined. OK. So that's Python dictionaries.

So one obvious motivation for building dictionaries is you need them in Python. And in fact, people use them all the time. We used them in docdist. All of the fastest versions of the document distance problem used dictionaries for counting words, how many times each word occurs in a document, and for computing inner products, for finding common words between two documents. And it's just it's the best way to do things, it's the easiest way to do things , and the fastest.

As a result, dictionaries are built into basically every modern programming language, Python, Perl, Ruby, JavaScript, Java, C++, C#. In modern versions, all have some version of dictionaries. And they all run in, basically, constant time using the stuff that's in this lecture and the next two lectures.

Let's see. It's also, in, basically, every database. There are essentially two kinds of databases in the world, there are those that use hashing, and there are those that use search trees. Sometimes you need one. Sometimes you need the other. There are a lot of situations in databases where you just need hashing.

So if you've ever used Berkeley DB, there's a hash type of a database. So if things like, when you go to Merriam-Webster, and you look up a word, how do you find the definition of that word? You use a hash table, you use a dictionary, I should say.

How do you-- when you spell check your document, how do you tell whether a word is correctly spelled? You look it up in a dictionary. If it's not correctly spelled, how do you find the closest related, correct spelling? You try tweaking one the letters, and look it up in a dictionary and see if it's there. You do that for all possible letters, or maybe two letters.

That is a state of the art way to do spelling correction. Just keep looking up in a

dictionary. Because dictionaries are so fast you can afford to do things like trial perturbations of letters.

What else. In the old days, which means pre-Google, every search engine on the web would have a dictionary that says, for given word, give me all of the documents containing that word. Google doesn't do it that way, but that's another story. It's less fancy, actually.

Or when you log into a system, you type your username and password. You look in a dictionary that stores a username and, associated with that username, all the information of that user. Every time you log into a web system, or whatever, it is going through a dictionary.

So they're all over the place. One of the original applications is in writing programming languages. Some of the first computer programs were programming languages, so you could actually program them in a reasonable way.

Whenever you type a variable name the computer doesn't really think about that variable name, it wants to think about an address in memory. And so you've got to translate that variable name into a real, physical address in the machine, or a position on the stack, or whatever it is in real life. In the old days of Python, I guess this is pre-Python 2 or so, 2.1, I don't remember the exact transition it was. In the interpreter, there was the dictionary of all your global variables, there's a dictionary of all your local variables.

And that was-- it was right there. I mean you could modify the dictionary, you could do crazy things. And all the variables were there. And so they'd match the key to the actual value stored in the variable.

They don't do that anymore because it's a little slow, but-- and you could do better in practice. But at the very least, when you're compiling the thing, you need a dictionary. And then, later on, you can do more efficient lookups. Let's see.

On the internet there are hash tables all over, like in your router. Router needs to know all the machines that are connected to it. Each machine has an IP address, so

when you get a packet in, and it says, deliver to this IP address, you see, oh, is it in my dictionary of all the machines that are directly connected to me? If so, send it there.

If it's not then it has to find the right subnet. That's not quite a dictionary problem, a little more complicated. But for looking up local machines, it's a dictionary. Routers use dictionaries because they need to go really fast. They're getting a billion packets every second.

Also, in the network stack of a machine, when you come in you get it packet delivered to a particular port, you need to say, oh, which application, or which socket is connected to this port? All of these things are dictionaries. The point is they're in, basically, everything you've ever used, virtual memory, I mean, they're all over the place.

There are also some more subtle applications, where it is not obvious that's it a dictionary, but still, we use this idea of hashing we're going to talk about today. Like searching in a string. So when you hit-- I don't know-- in your favorite editor, you do Control-F, or Control-S, or slash, or whatever your way of searching for something is, and you type start typing. If your editor is clever, it will use hashing in order to search for that string. It's a faster way to do it.

If you use grep, for example, in Unix it does it in a fancy way. Every time you do a Google search it's essentially using this. It's solving this problem. I don't know what algorithm, but we could guess. Using the algorithms we'll cover in next lecture. It wouldn't surprise me.

Also, if you have a couple strings and you want to know what they have in common, how similar they are? Example, you have two DNA strings. You want to see how similar they are, you use hashing. And you're going to do that in the next problem set, PS4, which goes out on Thursday.

Also, for things like file and directory synchronization. So on Unix, if you rsync or unison, or, I guess, modern day-- these days, Dropbox, MIT startup-- Whenever

you're synchronizing files between two locations, you use hashing to tell whether a file has changed, or whether a directory has changed. That's a big idea. Fairly modern idea.

And also in cryptography-- this will be a topic of next Tuesday's lecture. If you're transferring a file and you want to check that you actually transferred that file, and there wasn't some person in the middle corrupting your file and making it look like it was what you wanted it to be, you use something called cryptographic hash functions, which [INAUDIBLE] will talk about on Tuesday.

So tons of motivation for dictionaries. Let's actually do it, see how they are done. We're going to start with sort of a very simple straw man, and then we're going to improve it until, by the end of today, we have a really good way to solve the dictionary problem in constant time for operation.

So the really simple approach is called a direct access table. So it's just a big table, an array. You have-- the index into the array is the key. So, store items in an array, indexed by key.

And in fact, Python kind makes you think about this because the Python notation for accessing dictionaries is identical to the notation for accessing arrays. But with arrays, the keys are restricted to be non-negative integers, 0 through n minus 1. So why not just implement it that way?

If your keys happen to be integers I could just store all my items in a giant array. So if I just want to store an item here with key 2, call that, maybe, item 2, I just put that there. If I want to store something with key 4 I'll just put it there.

Everything else is going to be null, or none, or whatever. So lots of blank entries. Whatever keys I don't use I'll just put a null value there. Every key that I want to put into the dictionary I'll just store it at the corresponding position. What's bad about this? Yeah.

AUDIENCE: It's hard to associate something with just an integer.

PROFESSOR: Hard to associate something with an integer. Good. That's one problem. There's actually two big problems with this structure. I want both of them. So bad-- badness number one is keys may not be integers. Good. Another problem. Yeah.

AUDIENCE: Possibility of collision.

PROFESSOR: Possibility of collision. So here there's no collisions. We'll get to collisions in a moment, but a collision is when two items go to the same slot in this table.

And we defined the problem so there weren't collisions. We said whenever we insert item with the same key you overwrite whatever is there. So collisions are OK. They will be a problem in a moment, so save your answer. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Running time?

AUDIENCE: [INAUDIBLE]

PROFESSOR: For deletion? Actually, running time is going to be great. If I want to insert-- I mean, I do these operations but on array instead of a dictionary. So if I want insert I just put something there. If I want to delete I just set it to null. If I want to search I just go there and see is it null? Yeah?

AUDIENCE: It's a gigantic memory hog

PROFESSOR: It's gigantic memory hog. I like that phrasing. Not always of course. If it happens that your keys are-- the set of possible keys is not too giant then life is good.

Let's see If I cannot kill somebody today. Oh yes. Very good.

But if you have a lot of keys, you need one slot in your array per key. That could be a lot. Maybe your keys are 64-bit integers. Then you need 264 slots just to store one measly dictionary. That's huge. I guess there's also the running time of initialize that.

But at the very least, you have huge space hog. This is bad. So we're going to fix

both of these problems one at a time.

First problem we're going to talk about is what if your keys aren't integers? Because if your keys aren't integers you can't use this at all. So lets at least get something that works.

And this is a notion called prehashing. I guess different people call it different things. Unfortunately Python calls it hash. It's not hashing, it's prehashing. Emphasized the "pre" here.

So prehash function maps whatever keys you have to non-negative integers. At this point we're not worrying about how big those integers are. They could be giant. We're not going to fix the second problem til later.

First problem is if I have some key, maybe it's a string, it's whatever, it's an object, how do I map it to some integer so I could, at least in principle, put it in a direct access table. There's a theoretical answer to how to do this, and then there's the practical answer. how to do this. I'll start with the mathematical.

In theory, I like this, keys are finite and discrete. OK. We know that anything on the computer could, ultimately, be written down as a string of bits. So a string of bits represents an integer. So we're done.

So in theory, this is easy. And we're going to assume in this class, because it's sort of a theory class, that this is what's happening. At least for analysis, we're always going to analyze things as if this is what's happening.

Now in reality, people don't always do this. In particular-- I'll go somewhere else. In Python it's not quite so simple, but at least you get to see what's going on.

There's a function called hash, which should be called prehash, and it, given an object, it produces a non-- I'm not sure, actually, if it's non-negative. It's not a big deal if it has a minus sign because then you could just use this and get rid of the sign.

But it maps every object to an integer, or every hashable object, technically. But

pretty much anything can be mapped to an integer, one way or another. And so for example, if you given it an integer it just returns the integer. So that's pretty easy.

If you give it a string it does something. I don't know exactly what it does, but there are some issues. For example, hash of this string, backslash 0B is equal to the hash of backslash 0 backslash 0C 64. It's a little tricky to find these examples, but they're out there. And I guess, this is probably the lowest one in a certain measure.

So it's a concern. In practice you have to be careful about these things because what you'd like-- in an ideal world, and in the theoretical world-- this prehash function of x, if it equals the prehash function of y, this should only happen when x=y, when they're the same thing. And equals equal sense, I guess, would be the technical version.

Sadly, in Python this is not quite true. But mostly true. Let's see. If you define a custom object, you may know this, there is an __hash__ method you can implement, which tells Python what to do when you call hash of your object.

If you don't, it uses the default of id, which is the physical location of your object in memory. So as long as your object isn't moving around in memory this is a pretty good hash function because no two items occupy the same space in memory. So that's just implementation side of things.

Other implementation side of things is in Python, well, there's this distinction between objects and keys, I guess you would say. You really don't want this prehash function to change value. In, say, a direct access table, if you store-- you take an item, you compute the prehash function of the key in there, and you throw it in, and it says, oh, prehash value is four. Then you put it in position four. If that value change, then when you go to search for that key, and you call prehash of that thing, and if it give you five, you look in position five, and you say, oh, it's not there. So prehash really should not change.

If you ever implement this function don't mess with it. I mean, make sure it's defined in such a way that it doesn't change over time. Otherwise, you won't be able to find

your items in the table. Python can't protect you from that.

This is why, for example, if you have a list, which is a mutable object, you cannot put it into a hash table as a key value because it would change over time. Potentially, you'd append to the list, or whatever. All right.

So hopefully you're reasonably happy with this. You could also think of it is we're going to assume keys are non-negative integers. But in practice, anything you have you can map to an integer, one way or another.

The bigger problem in a certain sense, or the more interesting problem is reducing space. So how do we do that? This would be hashing. This is sort of the magic part of today's lecture.

In case you're wondering, hashing has nothing to do with hashish. Hashish is a Arabic root word unrelated to the Germanic, which is hachet, I believe. Yeah. Or hacheh-- I guess, something like that. I'm not very good at German. Which means hatchet. OK

It's like you take your key, and you cut it up into little pieces, and you mix them around and cut and dice, and it's like cooking. OK. What?

AUDIENCE: Hash browns.

PROFESSOR: Hash browns, for example. Yeah, same root. OK. It's like the only two English words with that kind of hash. OK.

In our case, it's a verb, to hash. It means to cut into pieces and mix around. OK. That won't really be clear until towards the end of today's lecture, but we will eventually get to the etymology of hashing. Or, we've got the etymology, but why it's, actually, why we use that term. All right.

So the big idea is we take all possible keys and we want to reduce them down to some small, small set of integers. Let me draw a picture of that. So we have this giant space of all possible keys. We'll call this key space. It's like outer space, basically. It's giant.

And if we stored a direct access table, this would also be giant. And we don't want to do that. We'd like to somehow map using a hash function h down to some smaller set. How do I want to draw this? Like an array.

So we're going to have possible values 0 up to m minus 1. m is a new thing. It's going to be the size of our hash table. Let's call the hash table. I think we'll call it t also. And we'd somehow like to map--

All right. So there's a giant space of all possible keys, but then there's a subset of keys that are actually stored in this set, in this dictionary. At any moment in time there's some set of keys that are present.

That set changes, but at any moment there's some keys that are actually there. k1, k2, k3, k4. I'd like to map them to positions in this table. So maybe I store k2-- or actually, item 2 would go here.

In particular, this is when h of k2, if it equals zero, then you'd put item 2 there. Item 3, let's say, it's at position-- wow, 3 would be a bit of a coincidence, but what the hell. Maybe h or k3 equals 3. Then you'd put item 3 here. OK. You get the idea.

So these four items each have a special position in their table. And the idea is we would like to be, m to be around n. n is the number of keys In the dictionary right now.

So if we could achieve that, the size of the table was proportional to the number of keys being stored in the dictionary, that would be good news because then the space is not gigantic and hoggish. It would just be linear, which is optimal. So if we want to store m things, maybe we'll use 2m space, a 3m space, but not much more.

How the heck are we going to define such a function h? Well, that's the rest of the lecture. But even before we define a function h, do you see any problems with this? Yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. This space over here, this is pigeonhole principle. The number of slots for your pigeons over here is way smaller than the number of possible pigeons. So there are going to be two keys that map to the same slot in the hash table. This is what we call a collision.

Let's call this, I don't know, ki, kj. h of ki equals h of kj, but the keys are different. So ki does not equal kj, yet their hash functions are the same, hash values are the same. We call that a collision. And that's guaranteed to happen a lot, yet somehow, we can still make this work. That's the magic.

And that is going to be chaining. We've done these guys. Next up is a technique for dealing with collisions. There are two techniques for dealing with collisions we're going to talk about in 006. One is called chaining, and next Tuesday, we'll see another method called open addressing.

But let's start with chaining. The idea with chaining a simple. If you have multiple items here all with the same-- that hash to the same position, just store them as a list. I'm going to draw it as a linked list. I think I need a big picture here.

So we have our nice universe, various keys that we actually have present. So these are the keys in the dictionary, and this is all of key space. These guys map to slots in the table. Some of them might map to the same value.

So let's say k1 and k2, suppose they collide. So they both go this slot. What we're going to store here is a linked list that stores item 1, and stores a pointer to the next item, which is item 2. And that's the end of the list. Or you could-- however you want to draw a null.

So however many items there are, we're going to have a linked list of that length in that slot. So in particular, if there's just one item, like say, this k3 here, maybe it just maps to this slot. And maybe that's all that maps to that slot.

In that case, we just say, follow this item 3, and there's no other items. Some slots are going to be completely empty. There nothing there so you just store a null pointer.

That is hashing with chaining. It's pretty simple, very simple really. The only question is why would you expect it to be any good?

Because, in the worst case, if you fix your hash function here, h, there's going to be a whole bunch of keys that all map to the same slot. And so in the worst case, those are the keys that you insert, and they all go here. And then you have this fancy data structure. And in the end, all you have is a linked list of all n items.

So the worst case is theta n. And this is going to be true for any hashing scheme, actually. In the worst case, hashing sucks.

Yet in practice, it works really, really well. And the reason is randomization, essentially, that this hash function, unless you're really unlucky, the hash function will nicely distribute your items, and most of these lists will have constant length.

We're going to prove that under an assumption. Well have to warm up a little bit. But I'm also going to cop out a little m as you'll see.

So in 006 we're going to make an assumption called Simple Uniform Hashing. OK. And this is an assumption, it's an unrealistic assumption. I would go so far as to say it's false, a false assumption. But it's really convenient for analysis, and it's going to make it obvious why chaining is a good idea.

Sadly, the assumption isn't quite true, but it gives you a flavor. If you want to see why hashing is actually good, I'm going to hint at it at the end of lecture but really should take 6.046 Yeah.

AUDIENCE: [INAUDIBLE] question. Is the hashing function [INAUDIBLE]? Like, how do we know the array is still [INAUDIBLE]?

PROFESSOR: OK. The hashing function-- I guess I didn't specify up here. The hashing function maps your universe to 0, 1, up to m minus 1, That's the definition. So it's guaranteed to reduce the space of keys to just m slots.

So your hashing function needs to know what m is. In reality there's not going to be

one hashing function, there's going to be 1 for each m, or at least one for each m. And so, depending on how big your table is, you use the corresponding hash function. Yeah, good question.

So the hash function is what does the work of reducing your key space down to small set of slots. So that's what's going to give us low space. OK. But now, how do we get low time? Let me just state this assumption and get to business.

Simply, uniform hashing is, essentially, two probabilistic assumptions. The first one is uniformity. If you take some key in your space that you want to store the hash function maps it to a uniform random choice. This is, of course, is what you want to happen. Each of these slots here is equally likely to be hashed to.

OK. That's a good start. But to do proper analysis, not only do we uniformity, we also need independence. So not only is this true for each key individually, but it's true for all the keys together. So if key one maps to a uniform random place, no matter where it goes, key two also matches to a uniform random place. And no matter where those two go, key three maps to a uniform random place.

This really can't be true. But if it's true, we can prove that this takes constant time. So let me do that. So under this assumption, we can analyze hashing-- hashing with chaining is what this method is called. So let's do it

I want to know-- I got to cheat, sorry. I got to remember the notation. I don't have any good notation here. All right.

What I'd like to know is the expected length of a chain. OK. Now this is if I have n keys that are stored in the table, and m slots in the table, then what is the expected length of a chain? Any suggestions. Yeah.

AUDIENCE: 1 over m to the n.

PROFESSOR: 1 over m to the n? That's going to be a probability of something. Not quite.

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's between 0 and 1. It's probably at least one, or something. Yeah.

AUDIENCE: m over n.

PROFESSOR: n over m, yeah. It's really easy. The chance of a key going to a particular slot is 1 over m. They're all independent, so it's 1 over m, plus 1 over m, plus 1 over m, n times. So it's n over m.

This is really easy when you have independence. Sadly, in the real world, you don't have independence. We're going to call this thing alpha, and it's also known as the load factor of the table. So if it's one, n equals m. And so the length of a chain is one.

If it's 10, then you have 10 times as many elements as you have slots. But still, the expected length of a chain is 10. That's a constant. It's OK.

If it's a 12, that's OK. It means that you have a bigger table than you have items. As long as it's a constant, as long as we have-- I erased it by now-- as long as m is theta n, this is going to be constant.

And so we need to maintain this property. But as long as you set your table size to the right value, to be roughly n, this will be constant. And so the running time of an operation, insert, delete, and search-- Well, search is really the hardest because when you want to search for a key, you map it into your table, then you walk the linked list and look for the key that you're searching for.

Now is this the key you're searching for? No, it's not the key you're searching for. Is this the key you're searching for? Those are not the keys you're searching for. You keep going. Either you find your key or you don't. But in the worst case, you have to walk the entire list.

Sorry for the bad *Star Trek* reference-- *Star Wars*. God. I'm not awake. All right.

In general, the running time, in the worst case, is 1 plus the length of your chain. OK. So it's going to be 1 plus alpha. Why do I write one?

Well, because alpha can be much smaller than 1, in general. And you always have to pay the cost of computing the hash function. We're going to assume that takes constant time. And then you have to follow the first pointer.

So you always pay constant time, but then you also pay alpha. That's your expected life. OK. That's the analysis. It's super simple.

If you assume Simple Uniform Hashing, it's clear, as long as your load factor is constant, m theta n, you get constant running time for all your operations. Life is good. This is the intuition of why hashing works.

It's not really why hashing works. But it's about as far as we're going to get in 006. I'm going to tell you a little bit more about why hashing is actually good to practice and in theory. What are we up to?

Last topic is hash functions. The one remaining thing is how do I construct h? How do I actually map from this giant universe of keys to this small set of slots in the table, there's m of them?

I'm going to give you three hash functions, two of which are, let's say, common practice, and the third of which is actually theoretically good. So the first two are not good theoretically. You can prove that they're bad, but at least they give you some flavor, and they're still common in practice because a lot of the time they're OK, but you can't really prove much about them. OK.

So first method, sort of the obvious one, called the division method. And if you have a key, this could be a giant key, huge universe of keys, you just take that key, modulo m, that gives you a number between zero and m minus 1. Done. It's so easy.

I'm not going to tell you in detail why this is a bad method. Maybe you can think about it. It's especially bad if m has some common factors with k.

Like, let's say k is even always, and m is even also because you say, oh, I'd like a table the size of power of two. That seems natural. Then that will be really bad

because you'll use only half the table. There are lots of situations where this is bad.

In practice, it's pretty good. If m is prime, you always choose a prime table size, so you don't have those common factors. And it's not very close to a power of 2 or power of 10 because real world powers of 2's and 10's are common. But it's very hackish, OK? It works a lot of the time but not always.

A cooler method-- I think it's cooler-- still, you can't prove much about it-- Division didn't seem to work so great, so how about multiplication? What does that mean? Multiply by m, that wouldn't be very good. Now, it's a bit different.

We're going to take the key, multiply it by an integer, a, and then we're going to do this crazy, crazy stuff. Take it mod 2 to the w and then shift it right, w minus r. OK. What is w?

We're assuming that we're in a w-bit machine. Remember way back in models of computation? Your machine has a word size, it's w bits. So let's suppose it's w bits. So we have our key, k. Here it is. It's w bits long.

We take some number, a-- think of a as being a random integer among all possible w bit integers. So it's got some zeros, it's got some ones. And I multiply these. What does multiplication mean in binary? Well, I take one of these copies of k for each one that's here.

So I'm going to take one copy here because there's a one there. I'm going to take one copy here because there's a one there. And I'm going to take one copy here because there's a one there. And on average, half of them will be ones.

So I have various copies of k, and then I just add them up. And you know, stuff happens. I get some gobbledygook here. OK.

How big is it? In general, it's two words long. When I multiply two words I get two words. It could be twice as long, in general. And what this business is doing is saying take the right word, this right half here-- let the right word in, I guess, if you see vampire movies-- and then shift right-- this is a shift right operation-- by w minus

r. I didn't even say what r is.

But basically, what I want is these bits. I want r bits here-- this is w bits. I want the leftmost r bits of the rightmost w bits because I shift right here and get rid of all these guys. r-- I should say, m, is two to the r. So I'm going to assume here I have a table of size a power of 2, and then this number will be a number between 0 and m minus 1. OK.

Why does this work? It's intuitive. In practice it works quite well because what you're doing is taking a whole bunch of sort of randomly shifted copies of k, adding them up-- you get carries, things get mixed up-- This is hashing. This is-- you're taking k, sort of cutting it up while you're shifting it around, adding things and they collide, and weird stuff happens.

You sort of randomize stuff. Out here, you don't get much randomization because most-- like the last bit could just be this one bit of k. But in the middle, everybody's kind of colliding together. And so intuitively, you're mixing lots of things in the center. You take those r bits, roughly, in the center. That will be nicely mixed up.

And most of the time this works well. In practice it works well-- I have some things written here. a better be odd, otherwise you're throwing away stuff. And it should not be very close to a power of 2. But it should be in between 2 to the r minus 1 and 2 to the r. Cool.

One more. Again, theoretically, this can be bad. And I leave it as an exercise to find situations, find key values where this does not do a good job.

The cool method is called universal hashing. This is something that's a bit beyond the scope of 006. If you want to understand it better you should take 046. But I'll give you the flavor and the method, one of the methods. There's actually many ways to do this.

We see a mod m on the outside. That's just division method just to make the number between 0 and a minus 1. Here's our key. And then there's these numbers a and b. These are going to be random numbers between 0 and p minus 1.

What's p? Prime number bigger than the size of the universe. So it's a big prime number. I think we know how to find prime numbers. We don't know in this class, but people know how to find the prime numbers.

So there's a subroutine here, find a big prime number bigger than your universe. It's not too hard to do that. We can do it in polynomial time. That's just set up.

You do that once for a given size table. And then you choose two random numbers, a and b. And then this is the hash function, a times k plus b, mod p mod m. OK.

What does this do? It turns out-- here's the interesting part. For worst case keys, k1 and k2, that are distinct, the probability of h of k1 equaling h of k2 is 1 over n. So probability of two keys that are different colliding is 1 over m, for the worst case keys. What the heck does that mean? What's the probability over? Any suggestions? What's random here?

AUDIENCE: a and b.

PROFESSOR: a and b. This is the probability over a and b. This is the probability over the choice of your hash function. So it's the worst case inputs, worst case insertions, but random hash function. As long as you choose your random hash function, the probability of collision is 1 over m. This is the ideal situation

And so you can prove, just like we analyzed here-- It's a little more work. It's in the notes. You use linearity of expectation. And you can prove, still, that the expected length of a chain-- the expected number of collisions that a key has with another key is the load factor, in the worst case, but in expectation for a given hash function. So still, the expected length of a chain, and therefore, the expected running time of hashing with chaining, using this hash function, or this collection of hash functions, or a randomly chosen one, is constant for constant load factor.

And that's why hashing really works in theory. We're not going to go into details of this again. Take 6.046 if you want to know. But this should make you feel more comfortable. And we'll see other ways do hashing next class.