6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 7

Today starts a two-lecture sequence on the topic of hashing, which is a really great technique that shows up in a lot of places. So we're going to introduce it through a problem that comes up often in compilers called the symbol table problem. And the idea is that we have a table S holding n records where each record, just to be a little more explicit here. So each record typically has a bunch of, this is record x. x is usually a pointer to the actual data. So when we talk about the record x, what it usually means some pointer to the data. And in the data, in the record, so this is a record, there is a key called a key of x.

In some languages it's key, it's x dot key or x arrow key, OK, are other ways that that will be denoted in some languages. And there's usually some additional data called satellite data, which is carried around with the key. This is also true in sorting, but usually you're sorting records. You're not sorting individual keys. And so the idea is that we have a bunch of operations that we would like to do on this data on this table.

So we want to be able to insert an item x into the table, which just essentially means that we update the table by adding the element x. We want to be able to delete an item from the table -- -- so removing the item x from the set and we want to be able to search for a given key. So this returns the value x such that key of x is equal to k, where it returns nil if there's no such x. So be able to insert items in, delete them and also look to see if there's an item that has a particular key. So notice that delete doesn't take a key. Delete takes a record. OK, so if you want to delete something of a particular key and you don't happen to have a pointer to it, you have to say let me search for it and then delete it.

So these, whenever you have a set operations, where operations that change the set like in certain delete, we call it a dynamic set. So these two operations make the set dynamic. It changes over time. Sometimes you want to build a fixed data structure. It's going to be a static set. All you're going to do is do things like look it up and so forth. But most often, it turns out that in programming and so forth, we want to have the set be dynamic. Want to be able to add elements to it, delete elements to it and so forth. And there may be other operations that modify the set, modify membership in the set. So the simplest implementation for this is actually often overlooked.

I'm actually surprised how often people use more complicated data structures when this simple data structure will work. It's called a direct access table. Doesn't always work. I'll give the conditions where it does. So it works when the keys are drawn from our small distribution. So suppose the keys are drawn from a set U of m elements. OK, zero to m minus one. And we're going to assume the keys are distinct.

So the way a direct access table works is that you set up an array T -- -- from zero to m minus one to represent the dynamic set S -- -- such that T of k is going to be

equal to x if x is in the set and its key is k and nil otherwise. So you just simply have an array and if you have a record whose key is some value k, the key is 15 say, then slot 15 if the element is there has the element.

And if it's not in the set, it's nil. Very simple data structure. OK, insertion. Just go to that location and set the value to the inserted value. For deletion, just remove it from there. And to look it up, you just index it and see what's in that slot. OK, very simple data structure. All these operations, therefore, take constant time in the worst case. But as a practical matter, the places you can use this strategy are pretty limited. What's the issue of limitation here?

Yes. OK, so that's a limitation surely. But there's actually a more severe limitation. Yeah. What does that mean, it's hard to draw? No. Yeah. m minus one could be a huge number. Like for example, suppose that I want to have my set drawn over 64 bit values. OK, the things that I'm storing in my table is a set of 64-bit numbers. And so, maybe a small set. Maybe we only have a few thousand of these elements.

But they're drawn from a 64-bit value. Then this strategy requires me to have an array that goes from zero to 2 to the 64th minus one. How big is 2^64 minus one? Approximately? It's like big. It's like 18 quintillion or something. I mean, it's zillions literally because it's like it's beyond the illions we normally use. Not a billion or a trillion. It's 18 quintillion. OK, so that's a really big number. So, or even worse, suppose the keys were drawn from character strings, so people's names or something. This would be an awful way to have to represent it. Because most of the table would be empty for any reasonable set of values you would want to keep.

So the idea is we want to try to keep something that's going to keep the table small, while still preserving some of the properties. And that's where hashing comes in. So hashing is we use a hash function H which maps the keys randomly. And I'm putting that in quotes because it's not quite at random. Into slots table T. So we call each of the array indexes here a slot. So you can just sort of think of it as a big table and you've got slots in the table where you're storing your values.

And so, we may have a big universe of keys. Let's call that U. And we have our table over here that we've set up that has -- -- m slots. And so we actually have then a set that we're actually going to try to represent S, which is presumably a very small piece of the universe. And what we'll do is we'll take an element from here and map it to let's say to there and take another one and we apply the hash function to the element. And what the hash function is going to give us is it's going to give us a particular slot.

Here's one that might go up here. Might have another one over here that goes down to there. And so, we get it to distribute the elements over the table. So what's the problem that's going to occur as we do this? So far, I've been a little bit lucky. What's the problem potentially going to be? Yeah, when two things are in S, more specifically, get assigned to the same value. So I may have a guy here and he gets mapped to the same slot that somebody else has already been mapped to. And when this happens, we call that a collision.

So we're trying to map these things down into a small set but we could get unlucky in our mapping, particularly if we map enough of these guys. They're not going to fit. So when a record -- -- to be inserted maps to an already occupied slot -- -- a collision occurs. OK. So looks like this method's no good. But no, there's a pretty

simple thing we can do. What should we do when two things map to the same slot? If we want to represent the whole set, but you can't lose any data, can't treat it like a cache. In a cache what you do is it uses a hashing scheme, but in a cache, you just kick it out because you don't care about representing a set precisely.

But in a hash table you're programming, you often want to make sure that the values you have are exactly the values in the sets so you can tell whether something belongs to the set or not. So what's a good strategy here? Yeah. Create a list for each slot and just put all the elements that hash to the same slot into the list. And that's called resolving collisions by chaining. And the idea is to link records in the same slot --

-- into a list. So for example, imagine this is my hash table and this for example is slot i. I may have several things that are, so I'm going to put the key value -- -- have several things that may have been inserted into this table that are elements of S. And what I'll do is just link them together. OK, so nil pointer here. And this is the satellite data and these are the keys. So if they're all linked together in slot i, then the hash function applied to 49 has got to be equal to the hash function of 86 is equal to the hash function of 52, which equals what?

There's only one thing I haven't. i. Good. Even if you don't understand it, your quizmanship should tell you. He didn't mention i. That's equal to i. So the point is when I hash 49, the hash of 49 produces me some index in the table, say i, and everything that hashes to that same location is linked together into a list OK. Every record. Any questions about what the mechanics of this. I hope that most of you have seen this, seen hashing, basic hashing in 6.001, right? They teach it in? They used to teach it 6.001. Yeah. OK. Some people are saying maybe. They used to teach it. Good. So let's analyze this strategy.

The analysis. We'll first do worst case. So what happens in the worst case? With hashing? Yeah, raise your hand so that I could call on you. Yeah. Yeah, all hash keys, well all, all the keys in S. I happen to pick a set S where my hash function happens to map them all to the same value. That would be bad. So every key hashes to the same slot. And so, therefore if that happens, then what I've essentially built is a fancy linked list for keeping this data structure. All this stuff with the tables, the hashing, etc., irrelevant. All that matters is that I have a long linked list. And then how long does an access take? How long does it take me to insert something or well, more importantly, to search for something. Find out whether something's in there. In the worst case.

Yeah, it takes order n time. Because they're all just a link, we just have a linked list. So access takes data n time if as we assume the size of S is equal to n. So from a worst case point of view, this doesn't look so attractive. And we will see data structures that in worst case do very well for this problem. But they don't do as good as the average case of hashing. So let's analyze the average case.

In order to analyze the average case, I have to, whenever you have averages, whenever you have probability, you have to state your assumptions. You have to say what is the assumption about the behavior of the system. And it's very hard to do that because you don't know necessarily what the hash function is. Well, let's imagine an ideal hash function. What should an ideal hash function do?

Yeah, map the keys essentially at random to a slot. Should really distribute them randomly. So we call this the assumption -- -- of simple uniform hashing. And what it means is that each key k in S is equally likely -- -- to be hashed to any slot in T and we're actually have to make an independence assumption. Independent of where other records, other keys are hashed. So we're going to make this assumption and includes n an independence assumption. That if I have two keys the odds that they're hashed to the same place is therefore what?

What are the odds that two keys under this assumption are hashed to the same slot, if I have, say, m slots? One over m. What are the odds that one key is hashed to slot 15? One over m. Because they're being distributed, but the odds in particular two keys are hashed to the same slot, one over m. So let's define. Is there a question? No. OK. The load factor of a hash table with n keys at m slots to be alpha which is equal to n over m, which is also if you think about it, just the average number of keys per slot.

So alpha is the average number of keys per, we call it the load factor of the table. OK. How many on average keys do I have? So the expected, we'll look first at unsuccessful search time. So by unsuccessful search, I mean I'm looking for something that's actually not in the table. It's going to return nil. I look for a key that's not in the table. It's going to be what? It's going to be order. Well, I have to do a certain amount of work just to calculate the hash function and so forth. It's going to be order at least one plus, then I have to search the list and on average how much of the list do I have to search?

What's the cost of searching that list? On average. If I'm searching at random. If I'm searching for a key that's not in the table. Whichever one it is, I got to search to the end of the list, right? So what's the average cost over all the slots in the table? Alpha. Right? Alpha. That's the average length of a list. So this is essentially the cost of doing the hash and then accessing the slot and that is just the cost of searching the list.

So the expected unsuccessful search time is proportional essentially to alpha and if alpha's bigger than one, it's order alpha. If alpha's less than one, it's constant. So when is the expected search time -- -- equal to order one? So when is this order one? Simple questions, by the way. I only ask simple questions. Some guys ask hard questions. Yeah. Or in terms first we'll get there in two steps, OK. In terms of alpha, it's when?

When alpha is constant. If alpha in particular is. Alpha doesn't have to be constant. It could be less than constant. It's O of one, right. OK, or equivalently, which is what you said, if n is O of m. OK, which is to say if the number of elements in the table is order, is upper bounded by a constant times n. Then the search cost is constant. So a lot of people will tell you oh, a hash table runs in constant search time. OK, that's actually wrong. It depends upon the load factor of the hash table. And people have made programming errors based on that misunderstanding of hash tables. Because they have a hash table that's too small for the number of elements they're putting in there.

Doesn't help. The number may in fact will grow with the, since this is one plus n over m, it actually grows with n. So unless you make sure that m keeps up with n, this doesn't stay constant. Now it turns out for a successful search, it's also one plus alpha. And for that you need to do a little bit more mathematics because you now

have to condition on searching for the items in the table. But it turns out it's also one plus alpha and that you can read about in the book. And also, there's a more rigorous proof of this. I sort of have glossed over the expectation stuff here, doing sort of a more intuitive proof. So both of those things you should look for in the book.

So this is one reason why hashing is such a popular method, is it basically lets you represent a dynamic set with order one cost per operation, constant cost per operation, inserting, deleting and so forth, as long as the table that you're keeping is not much smaller than the number of items that you're putting in there. And then all the operations end up being constant time. But it depends upon, strongly upon this assumption of simple uniform hashing. And so no matter what hash function you pick, I can always find a set of elements that are going to hash, that that hash function is going to hash badly. I just could generate a whole bunch of them and look to see where the hash function takes them and in the end pick a whole bunch that hash to the same place.

We're actually going to see a way of countering that, but in practice people understand that most programs that need to use things aren't really reverse engineering the hash function. And so, there's some very simple hash functions that seem to work fairly well in practice. So in choosing a hash function -- -- we would like it to distribute -- keys uniformly into slots and we also would like that regularity in the key distributions --

-- should not affect uniformity. For example, a regularity that you often see is that all the keys that are being inserted are even numbers. Somebody happens to have that property of his data, that they're only inserting even numbers. In fact, on many machines, since they use byte pointers, if they're sorting things that are for example, indexes to arrays or something like that, in fact, they're numbers that are typically divisible by four.

Or by eight. So you don't want regularity in the key distribution to affect the fact that you're distributing slots. So probably the most popular method that's used just for a quick hash function is what's called the division method. And the idea here is that you simply let h of k for a key equal k modulo m, where m is the number of slots in your table. And this works reasonably well in practice, but you want to be careful about your choice of modulus.

In other words, it turns out it doesn't work well for every possible size of table you might want to pick. Fortunately when you're building hash tables, you don't usually care about the specific size of the table. If you pick it around some size, that's probably fine because it's not going to affect their performance. So there's no need to pick a specific value. In particular, you don't want to pick --

-- m with a small divisor -- -- and let me illustrate why that's a bad idea for this particular hash function. I should have said small divisor d. So for example -- -- if D is two, in other words m is an even number, and it turns out that we have the situation I just mentioned, all keys are even, what happens to my usage of the hash table? So I have an even slot, even number of slots, and all the keys that the user of the hash table chooses to pick happen to be even numbers, what's going to happen in terms of my use of the hash table? Well, in the worst case, they are always all going to point in the same slot no matter what hash function I pick.

But here, let's say that, in fact, my hash function does do a pretty good job of distributing, but I have this property. What's a property that's going to have no matter what set of keys I pick that satisfies this property? What's going to happen to the hash table? So, I have even number, mod an even number. What does that say about the hash function? It's even, right? I have an even number mod. It's even. So, what's going to happen to my use of the table? Yeah, you're never going to hash anything to an odd-numbered slot. You wasted half your slots. It doesn't matter what the key distribution is.

OK, as long as they're all even, OK, that means the odds slots are never used. OK, an extreme example, here's another example, imagine that m is equal to two to the r. In other words, all its factors are small divisors, OK? In that case, if I think about taking k mod n, OK, the hash doesn't even depend on all the bits of k, OK? So, for example, suppose I had one..., and r equals six, OK, so m is two to the sixth.

So, I take this binary number, mod two to the sixth, what's the hash value? If I take something mod a power of two, what does it do? So, I hash this function. This is k, OK, in binary. And I take it mod two to the sixth. Well, if I took it mod two, what's the answer? What's this number mod two? Zero, right. OK, what's this number mod four? One zero. What is it mod two to the sixth? Yeah, it's just these last six bits. This is H of k. OK, when you take something mod a power of two, all you're doing is taking its low order bits. OK, mod two to the r, you are taking its r low order bits. So, the hash function doesn't even depend on what's up here.

So, that's a pretty bad situation because generally you would like a very common regularity that you'll see in data is that all the low order bits are the same, and all the high order bits differ, or vice versa. So, this particular is not a very good one. So, good heuristics for this is to pick m to be a prime, not too close to a power of two or ten because those are the two common bases that you see regularity in the world. A prime is sometimes inconvenient, however. But generally, it's fairly easy to find primes. And there's a lot of nice theorems about primes. So, generally what you do, if you're just coding up something and you know what it is, you can pick a prime out of a textbook or look it up on the web or write a little program, or whatever, and pick a prime.

Not too close to a power of two or ten, and it will probably work pretty well. It will probably work pretty well. So, this is a very popular method, the division method. OK, but the next method we are going to see is actually usually superior. The reason people do this is because they can write in-line in their code. OK, but it's not usually the best method. And the reason is because division, one of the reasons is division tends to take a lot of cycles to compute on most computers compared with multiplication or addition. OK, in fact, it's usually done with taking several multiplications. So, the next method is actually generally better, but none of the hash function methods that we are talking about today are, in some sense, provably good hash functions.

OK, so for the multiplication method, the nice thing about it is just essentially requires multiplication to do. And, for that is, also, we are going to assume that the number of slots is a power of two which is also often very convenient. OK, and for this, we're going to assume that the computer has w bit words. So, it would be convenient on a computer with 32 bits, or 64 bits, for example. OK, this would be very convenient. So, the hash function is the following. h of k is equal to A times k mod, two to the w, right shifted by w minus r.

OK, so the key part of this is A, which has chosen to be an odd integer in the range between two to the w minus one and two to the w. OK, so it's an odd integer that the full width of the computer word. OK, and what you do is multiply it by whatever your key is, by this funny integer. And, then take it mod two to the w. And then, you take the result and right shift it by this fixed amount, w minus r. So, this is a bit wise right shift.

OK, so let's look at what this does. But first, let me just give you a couple of tips on how you pick, or what you don't pick for A. So, you don't pick A too close to a power of two. And, it's generally a pretty fast method because multiplication mod two to the w is faster than division. And the other thing is that a right shift is fast, especially because this is a known shift. OK, you know it before you are computing the hash function. Both w and r are known in advance.

So, the compiler can often do tricks there to make it go even faster. So, let's do an example to understand how this hash function works. So, we will have, in this case, a number of slots will be eight, which is two to the three. And, we'll have a bizarre word size of seven bits. Anybody know any seven bit computers out there? OK, well, here's one. So, A is our fixed value that's used for hashing all our keys. And, in this case, let's say it's 1011001. So, that's A. And, I take in some value for k that I'm going to multiply. So, k is going to be 1101011. So, that's my k.

And, I multiply them. What I multiply two, each of these is the full word width. You can view it as the full word width of the machine, in this case, seven bits. So, in general, this would be like a 32 bit number, and my key, I'd be multiplying two 32 bit numbers, for example. OK, and so, when I multiply that out, I get a 2w bit answer. So, when you multiply two w bit numbers, you get a 2w bit answer.

In this case, it happens to be that number, OK? So, that's the product part, OK? And then we take it mod two to the w. Well, what mod two to the w says is that I'm just taking, ignoring the high order bits of this product. So, all of these are ignored, because, remember that if I take something, mod, a power of two, that's just the low order bits. So, I just get these low order bits as being the mod. And then, the right shift operation, and that's good also, by the way, because a lot of machines, when I multiply two 32 bit numbers, they'll have an instruction that gives you just the 32 lower bits. And, it's usually an instruction that's faster than the instruction that gives you the full 64 bit answer. OK, so, that's very convenient. And, the second thing is, then, that I want just the, in this case, three bits that are the high order bits of this word.

So, this ends up being my H of k. And these end up getting removed by right shifting this word over. So, you just right shift that in, zeros come in, in a high order bit, and you end up getting that value of H of k. OK, so to understand what's going on here, why this is a pretty good method, or what's happening with it, you can imagine that one way to think about it is to think of A as being a binary fraction.

So, imagine that the decimal point is here, sorry, the binary point, OK, the radix point is here. Then when I multiply things, I'm just taking, the binary point ends up being there. OK, so if you just imagine that conceptually, we don't have to actually put this into the hardware because we just do what the hardware does. But, I can imagine that it's there, and that it's here. And so, what I'm really taking is the

fractional part of this product if I treat A as a fraction of a number. So, we can certainly look at that as sort of a modular wheel.

So, here I have a wheel where this is going to be, that I'm going to divide into eight parts, OK, where this point is zero. And then, I go around, and this point is then one. And, I go around, and this point is two, and so forth, so that all the integers, if I wrap it around this unit wheel, all the integers lined up at the zero point here, OK? And then, we can divide this into the fractional pieces. So, that's essentially the zero point. This is the one eighth, because we are dividing into eight, two, three, four, five, six, seven.

So, if I have one times A, in this case, I'm basically saying, well, one times A, if I multiply, is basically going around to about there, five and a half I think, right, because one times A is about five and a half, OK, or five halves of 5.5 eighths, essentially. So, it takes me about to there. That's A. And, if I do 2^A, that continues around, and takes me up to about, where? About, a little past three, about to there. So, that's 2^A. OK, and 3^A takes me, then, around to somewhere like about there. So, each time I add another A, it's taking me another A's distance around.

And, the idea is that if A is, for example, odd, and it's not too close to a power of two, then what's happening is sort of throwing it into another slot on a different thing. So, if I now go around, if I have k being very big, then k times A is going around k times. Where does it end up? It's like spinning a wheel of fortune or something. OK, it ends somewhere. OK, and so that's basically the notion. That's basically the notion, that it's going to end up in some place. So, you're basically looking at, where does ka end up? Well, it sort of whirls around, and ends up at some point. OK, and so that's why that tends to be a fairly good one. But, these are only heuristic methods for hashing, because for any hash function, you can always find a set of keys that's going to make it operate badly.

So, the question is, well, what do you use in practice? OK, the second topic that I want to tie it, so, we talked about resolving collisions by chaining. OK, there's another way of resolving collisions, which is often useful, which is resolving collisions by what's called open addressing. OK, and the idea is, in this method, is we have no storage for links. So, when I result by chaining, I'd need an extra linked field in each record in order to be able to do that. Now, that's not necessarily a big overhead, but for some applications, I don't want to have to touch those records at all. OK, and for those, open addressing is a useful way to resolve collisions.

So, the idea is, with open addressing, is if I hash to a given slot, and the slot is full, OK, what I do is I just hash again with a different hash function, with my second hash function. I check that slot. OK, if that slot is full, OK, then I hash again. And, I keep this probe sequence, which hopefully is a permutation so that I'm not going back and checking things that I've already checked until I find a place to put it. And, if I got a good probe sequence that I will hopefully, then, find a place to put it fairly quickly. OK, and then to search, I just follow the same probe sequence.

So, the idea, here, is we probe the table systematically until an empty slot is found, OK? And so, we can extend that by looking as if the sequence of hash functions were, in fact, a hash function that took two arguments: a key and a probe step. In other words, is it the zero of one our first one? It's the second one, etc. So, it takes two arguments. So, H is then going to map our universe of keys cross, our probe number into a slot.

So, this is the universe of keys. This is the probe number. And, this is going to be the slot. Now, as I mentioned, the probe sequence should be permutation. In other words, it should just be the numbers from zero to n minus one in some fairly random order. OK, it should just be rearranged. And the other thing about open addressing is that you don't have to worry about n chaining is that the table may actually fill up.

So, you have to have that the number of elements in the table is less than or equal to the table size, the number of slots because the table may fill up. And, if it's full, you're going to probe everywhere. You are never going to get a place to put it. And, the final thing is that in this type of scheme, deletion is difficult. It's not impossible. There are schemes for doing deletion. But, it's basically hard because the danger is that you remove a key out of the table, and now, somebody who's doing a probe sequence who would have hit that key and gone to find his element now finds that it's an empty slot. And he says, oh, the key I am looking for probably isn't there. OK, so you have that issue to deal with. So, you can delete things but keep them marked, and there's all kinds of schemes that people have for doing deletion.

But it's difficult. It's messy compared to chaining, where you can just remove the element out of the chain. So, let's do an example -- -- just so that we make sure we're on the same page. So, we'll insert a key. k is 496. OK, so here's my table. And, I've got some values in it, 586, 133, 204, 481, etc. So, the table looks like that; the other places are empty. So, on my zero step, I probe H of 496, zero. OK, and let's say that takes me to the slot where there's 204. And so, I say, oh, there's something there.

I have to probe again. So then, I probe H of 496, one. Maybe that maps me there, and I discover, oh, there's something there. So, now, I probe H of 496, two. Maybe that takes me to there. It's empty. So, if I'm doing a search, I report nil. If I'm doing in the insert, I put it there. And then, if I'm looking for that value, if I put it there, then when I'm looking, I go through exactly the same sequence. I'll find these things are busy, and then, eventually, I'll come up and discover the value.

OK, and there are various heuristics that people use, as well, like keeping track of the longest probe sequence because there's no point in probing beyond the largest number of probes that need to be done globally to do an insertion. OK, so if it took me 5, 5 is the maximum number of probes I ever did for an insertion. A search never has to look more than five, OK, and so sometimes hash tables will keep that auxiliary value so that it can quit rather than continuing to probe until it doesn't find something.

OK, so, search is the same probe sequence. And, if it's successful, it finds the record. And, if it's unsuccessful, you find a nil. OK, so it's pretty straightforward. So, once again, as with just hash functions to begin with, there are a lot of ideas about how you should form a probe sequence, ways of doing this effectively. OK, so the simplest one is called linear probing, and what you do there is you have H of k comma i. You just make that be some H prime of k, zero plus i mod m.

Sorry, no prime there. OK, so what happens is, so, the idea here is that all you are doing on the I'th probe is, on the zero'th probe, you look at H of k zero. On probe one, you just look at the slot after that. Probe two, you look at the slot after that. So, you're just simply, rather than sort of jumping around like this, you probe there and then just find the next one that will fit in. OK, so you just scan down mod m. So,

if you hit the bottom, you go to the top. OK, so the I'th one, so that's fairly easy to do because you don't have to recomputed a full hash function each time. All you have to do is add one each time you go because the difference between this and the previous one is just one. OK, so you just go down. Now, the problem with that is that you get a phenomenon of clustering. If you get a few things in a given area, then suddenly everything, everybody has to keep searching to the end of those things.

OK, so that turns out not to be one of the better schemes, although it's not bad if you just need to do something quick and dirty. So, it suffers from primary clustering, where regions of the hash table get very full. And then, anything that hashes into that region has to look through all the stuff that's there. OK, so: long runs of filled slots. OK, there's also things like quadratic clustering, where you basically make this be, instead of adding one each time, you add i each time. OK, but probably the most effective popular scheme is what's called double hashing. And, you can do statistical studies.

People have done statistical studies to show that this is a good scheme, OK, where you let H of k, i, let me do it below here because I have for them. So, H of k, i is equal to an H_1 of k plus i times H_2 of k. So, you have two hash functions on m. You have two hash functions, H_1 of k and H_2 of k. OK, so you compute the two hash functions, and what you do is you start by just using H_1 of k for the zero probe, because here, i, then, will be zero. OK. Then, for the probe number one, OK, you just add H_2 of k.

For probe number two, you just add that hash function amount again. You just keep adding H_2 of k for each successive probe you make. So, it's fairly easy; you compute two hash functions up front, OK, or you can delay the second one, in case. But basically, you compute two up front, and then you just keep adding the second one in. You start at the location of the first one, and keep adding the second one, mod m, to determine your probe sequences.

So, this is an excellent method. OK, it does a fine job, and you usually pick m to be a power of two here, OK, so that you're using, usually people use this with the multiplication method, for example, so that m is a power of two, and H_2 of k you force to be odd. OK, so we don't use and even value there, because otherwise for any particular key, you'd be skipping over. Once again, you would have the problem that everything could be even, or everything could be odd as you're going through. But, if you make H_2 of k odd, and m is a power of two, you are guaranteed to hit every slot. OK, so let's analyze this scheme. This turns out to be a pretty interesting scheme to analyze.

It's got some nice math in it. So, once again, in the worst case, hashing is lousy. So, we're going to analyze average case. OK, and for this, we need a little bit stronger assumption than for chaining. And, we call it the assumption of uniform hashing, which says that each key is equally likely, OK, to have any one of the m factorial permutations as its probe sequence, independent of other keys.

And, the theorem we're going to prove is that the expected number of probes is, at most, one over one minus alpha if alpha is less than one, OK, that is, if the number of keys in the table is less than number of slots. OK, so we're going to show that the number of probes is one over one minus alpha. So, alpha is the load factor, and of course, for open addressing, we want the load factor to be less than one because if we have more keys than slots, open addressing simply doesn't work, OK, because

you've got to find a place for every key in the table. So, the proof, we'll look at an unsuccessful search, OK?

So, the first thing is that one probe is always necessary. OK, so if I have n over m, sorry, if I have n items stored in m slots, what's the probability that when I do that probe I get a collision with something that's already in the table? What's the probability that I get a collision? Yeah? Yeah, n over m, right? So, with probability, n over m, we have a collision because my table has got n things in there.

I'm hashing, at random, to one of them. OK, so, what are the odds I hit something, n over m? And then, a second probe is necessary. OK, so then, I do a second probe. And, with what probability on the second probe do I get a collision? So, we're going to make the assumption of uniform hashing. Each key is equally likely to have any one of the m factorial permutations as its probe sequence. So, what is the probability that on the second probe, OK, I get a collision?

Yeah? If it's a permutation, you're not, right? Something like that. What is it exactly? So, that's the question. OK, so you are not going to hit the same slot because it's going to be a permutation. Yeah? That's exactly right. n minus one over m minus one because I'm now, I've essentially eliminated that slot that I hit the first time. And so, I have, now, and there was a key there. So, now I'm essentially looking, at random, into the remaining n minus one slots where there are aggregately n minus one keys in those slots.

OK, everybody got that? OK, so with that probability, I get a collision. That means that I need a third probe necessary, OK? And, we keep going on. OK, so what is it going to be the next time? Yeah, it's going to be n minus two over m minus two. So, let's note, OK, that n minus i over m minus i is less than n over m, which equals alpha, OK? So, n minus i over m minus i is less than n over m.

And, the way you can sort of reason that is that if n is less than m, I'm subtracting a larger fraction of n when I subtract i than I am subtracting a fraction of m. OK, so therefore, n minus i over m minus i is going to be less than n over m. OK, so, or you can do the algebra. I think it's always helpful when you do algebra to sort of think about it sort of quantitatively as well, you know, qualitatively what's going on.

So, the expected number of probes is, then, going to be equal to, it's going to be equal to because we're going to need some space, well, we have one which is forced because we've got to do one probe, plus with probability n over m, I have to do another probe plus with probability of n over m minus one I have to do another probe up until I do one plus one over m minus n. OK, so each one is cascading what's happened. In the book, there is a more rigorous proof of this using indicator random variables. I'm going to give you the short version. OK, so basically, this is my first probe. With probability n over m, I had to do a second one. And, the result of that is that with probability n minus one over m minus one, I have to do another. And, with probability n over two minus m over two, I have to do another, and so forth.

So, that's how many probes I'm going to end up doing. So, this is less than or equal to one plus alpha. There's one plus alpha times one plus alpha times one plus alpha, OK, just using the fact that I had here. OK, and that is less than or equal to one plus I just multiply through here. Alpha plus alpha squared plus alpha cubed plus k. I can just take that out to infinity. It's going to bound this.

OK, does everybody see the math there? OK, and that is just the sum, I, equals zero to infinity, alpha to the I, which is equal to one over one minus alpha using your familiar geometric series bound. OK, and there's also, in the textbook, an analysis of the successful search, which, once again, is a little bit more technical because you have to worry about what the distribution is that you happen to have in the table when you are searching for something that's already in the table. But, it turns out it's also bounded by one over one minus alpha.

So, let's just look to see what that means. So, if alpha is less than one is a constant, it implies that it takes order one probes. OK, so if alpha is a constant, it takes order one probes. OK, but it's helpful to understand what's happening with the constant. So, for example, if the table is 50% full, so alpha is a half, what's the expected number of probes by this analysis? Two, because one over one minus a half is two.

If I let the table fill up to 90%, how many probes do I need on average? Ten. So, you can see that as you fill up the table, the cost is going dramatically, OK? And so, typically, you don't let the table get too full. OK, you don't want to be pushing 99.9% utilization. Oh, I got this great hash table that's got full utilization. It's like, yeah, and it's slow. It's really, really slow, OK, because as alpha approaches one, the time is approaching and essentially m, or n.

Good. So, next time, we are going to address head-on in what was one of the most, I think, interesting ideas in algorithms. We are going to talk about how you solve this problem that no matter what hash function you pick, there's a bad set of keys. OK, so next time we're going to show that there are ways of confronting that problem, very clever ways. And we use a lot of math for it so will be a really fun lecture.