

wk8

2-wk4



wk8

# 0wk8-Overview

SUMMARY: This week wraps up our discussion of data structures. We'll finish with a bang by covering hash tables --- certainly one of the most important data structures --- in detail. First (Part XIV) we discuss hash table basics --- the supported operations, the types of problems they're useful for, and an overview of implementation approaches. Part XV has one required video about "pathological data sets" for hash functions, and three optional videos that cover great stuff --- how randomization dodges pathological data sets, a construction of simple hash functions that are guaranteed to (in expectation over the choice of a random hash function) spread every data set out evenly, and performance analyses of hashing with both chaining and open addressing. Part XVI contains two required videos on the implementation and performance of bloom filters, which are like hash tables except that they are insanely space-efficient and occasionally suffer from false positives.

THE HOMEWORK: Problem Set #4 should solidify your understanding of hash tables and bloom filters. In the programming assignment you'll implement another slick data structure application covered in the lectures (solving the 2-SUM problem using hash tables).

SUGGESTED READINGS FOR WEEK 4:

CLRS Chapter 11

KT Chapter 13 (Section 13.6)

SW Section 3.5

# 14-1-Hash Tables - Operations and Applications

In this video we'll begin our discussion of hash tables; we'll focus first on the support operations, and on some of the canonical applications. So hash tables are insanely useful. If you want to be a serious programmer or a computer scientist you really have no choice but to learn about hash tables. I'm sure many of you have used them in your own programs in the past in fact. Now on the one hand what's funny is they don't actually do that many things in terms of the number of supported operations, but what they do, do they do really, really well. So what is a hash table? Well conceptually, ignoring all of the aspects of the implementation, you may wanna think of a hash table as an array. So one thing that arrays do super well is support immediate random access. So if you're wondering what's the position number seventeen of some array, boom, with a couple of machine instructions you can find out, wanna change the contents of position number 23 in some array? Done, in constant time. So let's think about an application in which you want to remember your friends phone numbers. So if you're lucky your friends parents were all u nu, unusually unimaginative people and all of your friends names are integers let's say between one and 10,000. So if this is the case then you can just maintain an array of link 10,000. And to store the phone number of say, your best friend, 173, you can just use position 173 of this modest sized array. So this array based solution would work great, even if your friends change over time, you gain some here you lose some there, as long as all your friends names happen to be integers between 1-10,000. Now, of course, your friends have more interesting names: Alice, Bob, Carol, whatever. And last names as well. So in principal you could have an array with one position in the array for every conceivable name you might encounter, with at least 30 letters set. But of course this array would be way too big. It would be something like 26 raised to the thirtieth power and you could never implement it. So what you'd really want is you'd want an array of reasonable size, say, you know ballpark the number of friends that you'd ever have, so say in the thousands or something, where it's positions are indexed not by the numbers, not integers. [inaudible] Between one and 10,000, but rather by your friends Names And what you'd like to do is you'd like to have random access to this array based on your friend's name. So you just look up the quote unquote Alice position of this array and. Boom, there would be Alice's phone number in constant time. And this, on a conceptual level is basically what a hash table, can do for you. So there's a lot of magic happening under the hood of a hash table and that's something we'll discuss to some extent in other videos. So you have to have this mapping between the keys that you care about, like your friends' names, and, numerical positions of some array. That's done by what's called a hash function, but properly implemented, this is the kind of functionality that hash tables gives you, So like an array with its positions indexed by the keys that you're storing. So you can think of the purpose of the hash table as to maintain a possibly evolving set of stuff. Where of course the set of things that you're maintaining, you know, will vary with the application. It can be any number of things. So if you're running an e-commerce website, maybe you're keeping track of transactions. You know, again, maybe you're keeping track of people, like for example, your friends and various data about them. So maybe you're keeping track of I-P addresses, for example if you wanna know, who was, were there unique visitors to your websites. And so on. So a little bit more formally, you know, the basic operations, you need to be able to insert stuff into a hash table. In many, but not all applications, you need to be able to delete stuff as well. And typically the most important operation is look-up. And for all these three operation you do it in a key based way. Where as usual a key should just be a unique identifier for the record that you're concerned with. So, for example, for employees you might be using social security numbers. For transactions you might have a transaction ID number. And then IP addresses could act as their own key. And so sometimes all you're doing is keeping track of the keys themselves. So, for example, in IP addresses, maybe you just want to remember a list of IP addresses. You don't actually have any associated data but in many applications, you know, along with the key, is a bunch of other stuff. So along with the employee's social security number, you gotta remember a bunch of other data about that employee. But when you do the insert, when you do the delete, when you do the look up, you do it based. On this key, and then for example, on look up you feed the key into the hash table and the hash table will spit back out all of the data associated with that key. We sometimes hear people refer to data structures that support these operations as a dictionary. So the main thing the hash table is meant to support is look up in the spirit of a dictionary. I find that terminology a little misleading actually. You know, most dictionaries that you'll find are in alphabetical order. So they'll support something like binary search. And I want to emphasis something a hash table does not do is maintain an ordering on the elements that it supports. So if you're storing stuff and you do want to have order based operations, you wanna find the minimum or the maximum, or something like that, a hash table's probably not the right data structure. You want something more. You wanna look at a heap or you wanna look at a, a search tree. But for applications in which all you have to do is basically look stuff up you gotta, you gotta know what's there and what's not, then there should be a light bulb that goes off in your head. And you can say, let me consider a hash table, that's probably the perfect data structure for this application. Now, looking at this menu-supported operations, you may be left kinda unimpressed. Alright, so a hash table, in some sense, doesn't do that many things; but again, what it does, it does really, really well. So, to first order. What hash tables give you is the following amazing guarantee. All of these operations run in constant time. And again this is in the spirit of thinking of a hash table as just like an array. Where its positions are conveniently indexed by your keys, So just like an array supports random access in constant time, you can see if, you know, there's anything in the array position, and what it is. As similarly a hash table will let you look up based on the key in constant time. So what is the fine print? Well, there's basically two caveats. So the first thing is that hash tables are easy to implement badly. And if you implement them badly you will not get this guarantee. So this guarantee is for properly implemented hash tables. Now, of course if you're just using a hash table from a well known library, it's probably a pretty good assumption that it's properly implemented. You'd hope. But in the event that you're forced to come up with your own hash table and your own hash function and unlike many of the other data structures we'll talk about, some of you probably will have to do that at some point in your career. Then you'll get this guarantee only if you implement it well. And we'll talk about exactly what that means in other videos. So the second caveat is that, unlike most of the problems that we've solved in this course, hash tables don't enjoy worst case guarantees. You cannot say for a given hash table that for every possible data set you're gonna get cost and time. What's true is that for non-pathological data, you will get cost and time operations in a properly implemented hash table. So we'll talk about both of these issues a bit more in other videos, but for now just high order bits are, you know, hash tables, constant time performance, subject to a couple of caveats. So now that I've covered the operations that hash tables support and the recommend way to think about them, let's turn our attention to some applications. All of these applications are gonna be in some sense, you know, kinda trivial uses of hash tables, but they're also all really practical. These come up all the time. So the first application we'll discuss, which again is a conical one, is removing duplicates from a bunch of stuff, Also known as the deduplication problem. So in the De-duplication problem, the input is essentially a stream of objects. Where, when I say a stream I have kinda, you know two different things in mind as canonical examples. So first of all you can imagine you have a huge file. So you have, you know, a log of everything that happened on some website you're running. Or all of the transactions that were made in a store on some day, And you do a pass through this huge file. So you're just in the middle of some outer for loop going line by line through this massive file. The other example of a stream that I had in mind, is, where you're getting new data over time. So here, you might imagine that you're running software to be deployed on an internet router. And data packets are coming through this router at a constant extremely fast rate. And so you might be looking at, say, the IP addresses and the sender, and use your data packet which is going through your router. So it would be another example of a stream of objects. And now, what do you gotta do? What you gotta do is you gotta ignore the duplicates. So remember just the distinct objects that you see in this stream. And I hope you find it easy to imagine why you might want to do this task in various applications. So, for example, if you're running a website you might want to keep track of the distinct visitors that you ever saw in a given day or a given week. If you're doing something like a web crawl, you might want to identify duplicate documents and only remember them once. So, for example, it would be annoying if in search results both the top link and the second link both led to identical pages at different URLs, okay, so search engines obviously want to avoid that, so you want to detect duplicate web pages and only report unique ones. And the solution using a hash table is laughably simple. So every time a new object arrives in the stream, you look it up. If it?s there, then it?s a duplicate and you ignore it. If it?s not there, then this is a new object and you remember it. Qed, that's it. And so then after the string completes, so for example after you finish reading some huge file, if you just want to report all of the unique objects, hash tables generally support a linear scan through them and you can just report all of the distinct objects when this stream finishes. So let's move on to a second application slightly less trivial maybe but still quite easy, and this is the subject of Programming Projects number five. So this is a problem called the two sum problem. You're given as input an array of N number. These images are in no particular order. You're also given a target sum, which I'll call T. And what you want to know is are there two integers from amongst these N you are given that sum to T. Now the most obvious and naive way to solve this problem is just to go over all N, choose two pairs of integers in the input, and check each one separately. So that's clearly a quadratic time algorithm. But now, of course, we need to ask, can we do better? And, yes, we can. And first of all let's see what you'd do if you couldn't use any data structures. So if you were clever, but you didn't use any data structures like a hash table, here would be a reasonable improvement over the naive one. So the first step of a better solution is to sort A upfront, For example, using word sort or heap sort, something that runs in end log and time. So you may be asking about the motivation for sorting. Well, again, you know, one thing is just, you know whenever you're trying to do better than N squared; you might think that sorting your data somehow helps. Right and you can sort of do it almost for free in N log N time. Now, why would sorting the array up front help us? Well, then the clever insight is that for each entry of the array a, say the first entry, now we know what we're looking for to achieve this given target, right. If the target that we're trying to get to is summed to 100 and the first entry in the sorted array is 43, then we know we're looking for a 57 somewhere else in. This now sorted array. And we know that searching a sorted array is pretty easy, right. That just binary search. That just takes logarithmic time. So for each of the n array entries, we can look for a complementary. Entry, namely of reach X we can look for T - X using binary search. And to use binary search takes log N time. So the sorting upfront speeds up this entire batch of N searches. So that's why it's a win. So, in the second step, because we do a linear number of binary searches, again, this is just n, the number of searches, times log-n, the time per search. So, this is just another theta of N log N factor. Alright, so that's pretty cool. You, I don't think you could come up with this N log N solution without having some basic, facility with algorithms. This is already a really nice improvement over the naive N squared. But we can do even better. It is no reason we're stuck with an N log N lower bound for the [inaudible] problem. Obviously, because the array is unsorted, we have to look at all the integers. So we're not gonna do better than linear time. But we can do linear time via a hash table. So a good question you might ask at this point is what's the clue about this problem, about this task that suggests we want to use a hash table. Well, so hash tables are going to dramatically speed up any application where the bulk of the word is just repeated look-ups. And if we examine this n log n solution, once we have this idea of doing a search for T minus X for each value of X, we realize actually, you know, the only thing we needed the sorted array for was to support look-ups. That's all binary search here is doing, is just looking stuff up. So we say, ah-ha. All of the work here in step two is from repeated look-ups. We're paying an exorbitant relatively, logarithm per amount of time per look-up, whereas hash tables can do them in cost and time. So, repeated look-ups, ding, ding, ding, let's use a hash table; and indeed that's what gives us linear time in this problem. So from the amazing guarantee of hash tables, we get the following amazing solution for the true [inaudible] problem, although again this is subject to the same fine print about you better use it properly implemented hash table and you better not have pathological data. So rather than sorting, you just insert everything in the array into a hash table. So insertions cost time. So this is gonna be linear time rather than the end log [inaudible] we were paying before. Once all the stuff is in the hash table, we just do the same thing as in the n log-n solution. For each x in the array, we look for its matching elements, t-x in the hash table using the cost and time look-up operation exported by the hash table. And of course if for some X, you do find the matching element T minus X. Then you can just report X and T minus X. That proves that there is indeed a pair of integers of target sum T. If for every single element of the input array A, you fail to find this matching element T minus X in the hash table. Then, for sure there is no pair of integers in the input that sums to T. So this solves the problem correctly. Moreover, constant time insertion, so that means this first step is going to be O of end time. And constant time look-up. So that means that the second step is also gonna be linear time. That leaves subjects to the caveats that we discussed on the previous slide. So it's kind of amazing how many different applications of computer science boil down in their essence to repeated look up operations. Therefore, having a super fast look up operation, like that supported by a hash table, permits these applications to scale to fantastic sizes. It's really amazing, and it drives a lot of modern technology. So let me just mention a couple examples. Again, if you look around or do some research on the web, you'll quickly find many more. So originally what prompted researchers to think hard about data structures that support super fast look ups, was back when people were first building compilers. So this is a long time ago. This is in the fifties or so. And these repeated look ups to figure out, you know, what has and has not been defined before was, was emerging as a bottleneck in compilers. Back in the early days of programming languages. And that was one of the early applications of hash tables. Was to support super fast look ups to speed up compile time. To keep track of the function of variable names and things like that. Hash table technology is also super useful for software on routers in the Internet. So, for example, you might want to block network traffic from certain sources. So, for example, maybe you suspect that a certain IP address has been taken over by spammers and so any traffic coming from that IP address you just want to ignore. And you don't wanna even let it get to the end host, to the computer on someone's desktop, or to someone's mobile device but rather inside the internet. You wanna just drop packets that are coming certain, certain centers. So what is that problem boil down to? Well, you might have a blacklist of IP addresses that you're refusing traffic from and then the tasks faced by the router is really the look up problem. So if data packet comes in at some insanely fast data rate, and when you wanna. You immediately, just look up, is this in the blacklist or not, and if it is in the blacklist then you drop the packet, if it?s not, then you let it go through. So a very different application is for speeding up search algorithms. And when I say a search algorithm, what I'm thinking about here is something like a chess playing program. So something that does game tree exploration. So we've already talked a fair amount about graph search in this class, but in our discussion of breadth first and depth first search, we were thinking about graphs that you could basically write down. You could store them in the main memory of your machine or, in the worst case, on some big cluster. So maybe graphs, you know, about the size of the web graph or possibly smaller. But in a context of something like a chess playing program the graph that you're interested in is way, way, way bigger than the web graph. So what's the graph we care about for a chess playing program? Well, the nodes of the graph are going to correspond to all possible configurations of chess pieces On a chess board. So every chess board that you might ever encounter in a game of chess. So that's a. Massive, massive number of configurations. And you're never gonna be able to write down these vertices. The edges in this graph are going to take you from one configuration to another. And there gonna correspond to legal moves. So if you can move a bishop from. One place to another place, and you get from one configuration to another configuration, there's an edge in the graph corresponding to that move. Now you can't write down this graph. So you can't implement breadth versus depth versus search exactly as we discussed it before. But, you'd still like to do graph exploration, right? So you'd like to have your computer program, reason about the at least short term ramifications of your possible next move. So that will correspond to searching through this graph. Now, how are you gonna, it's remembering graphs search a really important property was you don't want to do redundant work, you don't want to re-explore things you've already explored. That would be silly and might lead into loops and so on. And you can't write down the graph just remembering where you've been, is suddenly a non-trivial problem; but what is remembering where you've been, fundamentally? Fundamentally that's a look up operation. So that is exactly what hash tables are for. So to be a little more concrete, you know, one where you use the hash table in, say, a chess playing program, is you'd stake, take the initial configuration. You would sort of imagine trying all possible moves from this configuration. And then you'd try, you'd sort of have all moves from your opponent and then you'd have all your moves in response. And you would always remember, as you were doing this reasoning, every chessboard configuration you'd ever looked at before and you'd stick in the hash table. And before you go exploring some configuration, you'd look it up in your hash table to see if you've already explored it in the past. And if you have, you don't bother. You've already seen it. All right. So chess playing programs operate by exploring systematically as many configurations as they'd have time for. You know, obviously, in a budget of three minutes or whatever you don't wanna waste any work exploring any given configuration more than once. How do you remember where you've been? Well everything you've explored you stick in a hash table Before you explore a configuration you look it up in a hash table and see if you've already done it. So these of course are just scratching the surface. I just wanted to highlight a couple, you know, fairly different looking applications, you know to convince you that hash tables come up all the time. And the reason they come up all the time is because you know the need for fast look-ups comes up all the time. It's kind of amazing how much technology is being driven just by you know repeated fast look-ups. So as homework I encourage you to just sort of think about you know your own life, or think about technology out there in the world, and come up with some. You know, guesses about where probably hash tables are making something out there running blazingly fast. I think it won't take you more than a few minutes to come up with some good examples.

# 14-2-Hash Tables - Implementation Details, Part I

So in this video we'll take a peek under the hood of hash functions. And I'll discuss some of the high level principles by which their implemented. So let's briefly review the raison d'etre of a hash table. So the purpose in life for a hash table is to support super-fast lookups. So maybe you're keeping track of the transactions that happened on your website yesterday. Maybe you're keeping track of your employees; maybe you're keeping track of IP addresses in an Internet router. Maybe you're keeping track of chess configurations in a, in a chess-playing program, Whatever, The point is, you want to be able to insert stuff into a hash table, and later remember whether something's there or whether something's not there. So the implementations we'll discuss will generally also support deletions. But that's pretty much it. It's a very restricted set of operations. But the hash, It was going to execute them at very, very well. So, basically in constant time, again subject to some fine print, which we'll discuss a little bit in this video but, then more deeply in a separate optional video. So, the two caveats are first of all the hash table had better be properly implemented. It's actually pretty easy to screw up a hash table to screw up hash functions. We'll talk a bit about that in a few minutes and, then, also, the data should, in some sense, be non-pathological, and that, we will discuss more deeply in a separate video. Alright, so let me give you an initial glimpse of some of the magic that's happening under the hood in hash functions. So, at first let me say exactly what the setup is. The first step is to identify all the things that you might want to be storing. So, in other words, the universe of your application, So this would be something like, all possible I-P addresses, of which there's 2^32 . All possible names you might encounter, perhaps with a maximum of, say, 30 characters. All possible configurations of a chessboard, and so on, And one thing I hope you can appreciate from these examples is that, in many cases, this universe is really big. Sothe number of ] IP address is, quote unquote, only two 2^32. The number of all names, you're probably talking more like 26 raised to the 30. All chessboard configurations I don't even wanna think about. And what you wanna accomplish is, you wanna maintain an evolving subset of this universe U. So maybe you wanna keep track of all the IP addresses you've seen on your website in the last 24 hours. You wanna keep track of the phone numbers of all of your friends. You wanna keep track of the chessboard configurations that you've explored in the past three seconds, whatever. And again I hope what is clear from the applications we've been discussing, is that the set S is usually of a reasonable size. It's, it's something you could store in main memory. You know, it maybe it's tens of thousands of IP addresses. Maybe it's, you know, a few hundred names of your various friends. You know, maybe it's in the, you know, millions of chessboard configurations, but still way, way, way smaller than the size of the universe. So without data structures, you'd have to resort to other unsatisfactory solutions to maintaining this set. So the first thing you could try, as we discussed in the previous video, would be just have an array with one position for every imaginable thing you might want to store in your set. So this is the solution that's going to work well if all of your friends happen to have names that are integers between 1 and 10,000, but doesn't scale when the universe size becomes really big, as in most of these applications. So, the good news is, is of course, is an array of and it's of course fast random access so you can access any position in constant time. So, if you have an array base solution index by all the elements of the universe, you can do constant time, insert, delete and look up. The bad news is, is the space requirement is proportional to the universe. And again, forget about being unsatisfactory. That's just literary impossible. Infeasible in many applications in which you'd use hash tables. Now of course to get the memory proportional to the size of the set stuff that you're storing, an easy solution would just be to use a list. You know, say a doubly-linked list. Something like that. Now with a list-based solution the good news is, is your memory is certainly proportional to the size of the set that you're storing, and independent of the size of the universe from which these elements are drawn. The bad news is that to figure out whether something is, or is not, in a list you generally have to traverse through most of that list. And that's gonna take up time proportional to the length of the list. So, really the question we're faced in implementing cache table is, can we get the best Of both worlds, of these two naive solutions. And the one hand, we want to have the constant time operations enjoyed by the array based solution. But on the other hand, we wanna have the, linear space in the size of the set that we're storing; that we get in the list based solution. So to get the best of both worlds, we are going to use an array based solution. But the array will not be big. It'll not be with size proportional to the universe. The array will only have size, you know, roughly the same as the set that we're storing, So somewhere in the ball park of the cardinality of S. So the first thing we do is we decide on how big we want our array to be. So that, that length is gonna be called n. We're gonna have an array of length n. And n is gonna be in the ballpark of the size of S. It's gonna depend on a few things. Exactly how n compares to S, but for now think of n as like double the size of S. We're gonna be calling each entry of the array a bucket, so there's n buckets, and then, the size of S is about 50 percent of the number of buckets, let's say. So one objection you might legitimately raise at this point is, you know I thought, I said the set was dynamic. The set S. Right? Stuff can be added, stuff can be deleted. So the size isn't always the same. It can fluctuate over time. So what does it mean to define an array which is the, roughly the same length as this changing set. So for simplicity, for the purposes of this video to focus on the key points I am going to assume that the set size S. While S itself can be changing, I'm going to assume that the size of S doesn't fluctuate too much. So there are additional bells and whistles you can add to a hash table implementation, and they're all quite natural. I think most of you could probably figure them out on your own, to deal with the fact that S might be changing sizes. So for example, you can just keep track of how many elements are in your hash table. And when it exceeds a big, a certain threshold, so when it's too big relative to the size of your array, you just double the array. And then you reinsert all of the elements into this new doubled array. Similarly, if you want to, if the set shrinks, you can have tricks for shrinking the array dynamically as well. So I'm not gonna discuss these bells and whistles for resizing your hash table dynamically. They are, of course Important for a real implementation, and they are part of the implementations in the standard programming libraries. But I view those as sort of a, a second order point in the implementation of a hash table. And I wanna focus on the first order points, in this video. So, summarizing, think of the set S. There are insertions and deletions we have to accommodate. But, you know, S is gonna be roughly the same size. And the number of buckets will be, you know, within a constant factor of the size of the set. All right so there we have our array with totally reasonable space, space proportional to the size of the set that we are storing. And now what we want is we want is some way of translating between the things that we care about, say our friends names or whatever the elements in the universe are to the positions in this array. So the object responsible for that translation from keys drawn from this universe to positions in this array is called a hash function. So formally, a hash function Takes as input a key. So this is gonna be an IP address or the name of somebody or a chessboard configuration or whatever. And it's going to spit out an position in this array. So I'm gonna label the array entries from 0 to n-1 for this lecture. Obviously at the moment this is super underspecified. There's a zillion functions you could choose. Which one you use, we'll talk about that, but for now there's just gonna be some hash function mapping from elements of the universe to buckets, to positions in this array. Now, as far as the semantics of this hash function, what the hash function is doing, it's telling us in which position we should store a given key from the universe. So, if we have some new friend named Alice. And we run Alice, we key Alice through the hash function and it gives us a 17. It says we should store Alice's phone number in position 17 of the array. If we have some crazy chessboard configuration, we feed it into a hash function and it spits out 172, it says we should remember this chessboard configuration in the 172nd bucket of this array. So again, given x, which is some key from this universe, we invoke a hash function to get a position in this array, to get a bucket. And then that is where we try to store this x and any associated data with it. So that's the high leveled idea of how you implement a hash table, but we're quite far from done, And in particular there is a serious issue, that we're going to have to deal with, that's fundamental to implementing hash tables, and that's the notion of a collision. So probably many of you may have already noticed that this problem might occur. Which is well what happens if we're storing our friend's phone numbers, and you know Alice shows up and we ask our hash function where to store Alice's phone number, and it says oh bucket number 17, And then our friend Bob shows up, and we ask our hash function where to store Bob's phone number, and what if the hash function also says bucket number 17 for Bob? What do we put in bucket at 17? Do we put Alice there, do we put Bob there, do we put them both there? How do we deal with these so-called collisions? So, the next quiz is meant to give, to get you thinking about collisions, and in some sense, how truly unavoidable they really are. [sound], [sound] All right. So the correct answer to this question is the first answer, believe it or not. All you need is 23 people in a room before you're equally likely to have two people with the same birthday as not. So if you're looking to, to skim a little money off of your non-mathematical friends, this is one way you can do it. Go to cocktail parties with about 40 people and place bets with people that there are two people in the room with the same birthday. So if you have 367 people, well there's only 366 distinct birthdays, I'm counting February 29th here as one of them. So by the pigeonhole principle, certainly the probability is 100%. By the time you get to 367. Now, by the time you're at 57. You're already at 99%. So you already have overwhelming probability to have a duplicate birthday with 57 people. So of course, with 184 you're gonna be almost at 100%, 99.99. Who knows? Some large number of 9's, And at 23, you're at 50%. So many people find this quite counter-intuitive that you only need 23 people to get a duplicate birthday on average. And so this is a, this is a quite famous example and it sometimes goes by the birthday paradox. Calling it a paradox is sort of a misnomer. A paradox, you know, often suggests some kind of logical inconsistency. There's no logical inconsistency here. It's just that people's brains are not really wired to have this intuition, for whatever reason. So, but it's really just math. You can work out the math, and, and, and you can just solve it. So, more generally, the principle behind the birthday paradox is the following. So suppose you have a calendar, perhaps on some different planet, which has K days. Where each, everybody's equally likely to have each of the K days as their birthday. Then it's about the square root of k people that you need in a room before you're equally likely to have a duplicate, or not have a duplicate. Okay, and the reason that you get the square root effect is because if you think about it. There's a quadratic number of pairs of people in the room, so that's a quadratic, and the number of people Opportunities to have a duplicate. Right? So, each pair of people could be a duplicate, there's a quadratic number of pairs. And so, that's why, once the number of pairs starts reaching about the number of different days, you're, you're about, you're likely to see a duplicate around that point. So you might be wondering why I'm telling you about the birthday paradox in the middle of a lecture about hashing, but really it's quite relevant. So imagine for example you defined a hash function in the following way. Now to be clear, this is not a practical hash function, but just for the purposes of discussion, imagine you have a hash function which randomly assigned every single key to a uniform bucket. 'Kay, so for each, each of the 1/n buckets equally likely. Then what the birthday paradox says is, even for a very small dataset, you are already gonna have a pair of things colliding. All right, So if you have an n buckets, so maybe your n is like, 10,000, all you need is roughly 100 elements in your data set, and despite the fact that the table is only going to be one percent full, you're already going to see a collision, okay? So 99 percent of them are empty, but you're going to have one bucket that has two, so that's sort of annoying. So the birthday paradox says, you start getting collisions with the hash function, even with the really tiny data sets. So in this sense, if you're going to have hash tables, you've got to deal with collisions. There's going to be a fair number of them, and you need some method for resolving them So, collisions are a fact of life when you're talking about hashing. Where again, by collision, what I mean is two different keys. So two different elements x and y from the universe that hash to the same bucket, Who have the same hash value, So in general we can think of a hash function as doing a compression of sorts. So we have a huge universe U and we have this very modest size array A with the only n buckets. Where n, we're thinking of as being much, much, much smaller than U. So, of course, this hash function has to map various elements of U to the same bucket. So what are we gonna do about it? How are we going to resolve these collisions? Well, there's two different solutions which are both quite prevalent in practice. So solution number one is called chaining, or sometimes you'll also see it called separate chaining. And this is a very natural solution; it's also the one that's relatively easy to analyze mathematically. What you do is just for elements that hash to the same bucket, you just revert to the list-based solution that we talked about in a previous slide. So, each of the n buckets will not necessarily contain just merely 0 or 1 element , it will contain a list within a principle unbounded number of elements. Okay, so when we use chaining, it's done quite straight-forward to figure out how to implement all of the hash table operations, namely, insert, delete and look-up, you just hash something to the appropriate bucket and then you just do insert, delete or look-up, as appropriate, in the list that's in that bucket. So just to make clear that everything is type checking, so here h(x), this is the bucket for x. That's what's specified by the hash function. And then, in the h(x) position of this array A, in the h (x), the bucket is where we find the linked list that is going to contain x. So just to give a cartoon example, if you had, say, four buckets, Maybe, you know, the first bucket has exactly one record. Corresponding to Alice, maybe the second bucket just has a null pointer. No one's been inserted in the second bucket. And then the third bucket we have, let's say, both Bob as well as Daniel. And then maybe in the fourth bucket we have Carol. Okay, so because we have a collision between Bob and Daniel, both map to the third bucket, and we resolve that just by having a linked list, with Bob and Daniel in some order. So the second solution which is trickier to talk about mathematically but still quite important practically is called open addressing. And the principal in open addressing is you're not going to use any space for pointers. You're not gonna have lists. So you're only gonna have one object per bucket of the array. So another question is what happens if, you know, you try and insert Daniel and you go, you invoke the hash function on Daniel and it takes you to a bucket that already contains Bob? That means there's no room for Daniel. So what you're going to do is you're gonna probe the hash table in some other position. So a hash function is, is now gonna be replaced by a hash sequence, where you try, the hash function tells you the first bucket to try to insert Daniel; failing that, a second bucket in which to try to insert Daniel; failing that, a third bucket to try to insert Daniel; and so on. And you just keep trying till you find an open position somewhere in the array. So there's various strategies for trying to figure out the probe sequence. One strategy is if you fail and save bucket 17, which is where the hash function tells you to go first. You just try bucket 18, then 19, then, 20, then 21 and so on, until you find your first open slot. So that's called linear probing. And another approach is double hashing. So this is a solution where you actually have two hash functions, hash function 1 and hash function 2. And the idea is, suppose you're trying to insert, say, Daniel, into a hash table with open addressing, and you evaluate both of the hash functions. And the first one comes up 17, and the fir-, the second one comes up 23. So, as usual, the has-, first hash function will specify where you look first. So if it evaluates on Daniel to 17, you look in the seventeenth position of the array, And if, if it's empty, that's where you insert Daniel. Now, if it's full, what you do is you use the second Hash value to be an additive shift, so. Unlike linear probing where after seventeen, you look at eighteen. With double hashing, if the second hash function gives you 23, that's gonna be your offset. So after seventeen, you look at bucket 40. If 40 is already full, you look at bucket 63. If bucket 63 is already full then you look at bucket 86. So you keep adding increments of 23 until you finally find a bucket where, that's empty and that's where you insert Daniel. Now of course, if you try to insert some other name, if you try to insert Elizabeth, you're gonna get two totally different numbers in general. So maybe you'll get 42 and 27, and so here the probed sequence will be 42, failing that 69, failing that 96 failing that 123 and so on, So a question you should have at this point, is, you know. I've told you two solutions to resolving collisions in a hash table. And you're probably asking, well, which ones should you use if you have to implement your own hash table? And, you know, as usual, if I present you with two different solutions for the same problem. You can probably rest assured that neither one dominates the other, right? Otherwise I wouldn't waste your time by presenting both of them to you. So, sometimes chaining's gonna perform better, and sometimes, open addressing's gonna perform better. And of course, it also depends on what kind of metric that you care about. So there are a couple of rules of thumb that I can tell you. So first of all if space is at a real premium you might want to consider open addressing instead of chaining, and that's cause with chaining you do have this excess, not huge but you do have this little bit of space overhead and dealing with all these pointers in this link list. So if you want to avoid that, you might want to think about open addressing. The second rule of thumb is deletion is trickier with open addressing than with chaining, but deletion is clearly not difficult at all, either to code or understand when you use chaining cause it just reduces chaining to a linked list which of course you all know how to do. Open Addressing is, it's not impossible to implement deletion but it's much trickier. So if deletion's a, a crucial operation for you, that might steer you towards thinking about chaining. But ultimately, if it's really kinda mission critical code, probably the best thing to do is implement both kinds of solutions and just see which one works better. It's a little hard to predict how they're gonna interact with memory hierarchies and that kind of thing. They're both useful in their own contexts. Alright so we've covered the two most prevalent ways of handling collisions. And we argued that collisions are inevitable no matter you design you hash function. You're stuck with collisions and you can do chaining or linked lists per bucket, or you can do addressing, where you actually have a probe sequence in order of which you look at buckets until you find an empty one. And the elephant in the room at this point is, you know, what is this hash function? I have told you nothing about hash functions. All I told you is there is some mapping from the set of the universe, so IP addresses, or names, or whatever to a bucket number. Well what kind of function should you use? Excellent question, Tons of research on that question, And to this day as much art as science. But let's start talking about it.

# 14-3-Hash Tables - Implementation Details, Part II

Let's begin by building up some intuition about what we would want from a hash function, now that we know how hash functions are usually implemented. So let's start with a hash function which is implemented by chaining. So what's going to be the running time of our lookup, insert, and delete operations in a hash table with chaining? Well, the happy operation in a hash table with chaining is insertion. Insertion, we can just say without any qualifications, is constant time. This requires the sort of obvious optimization that when you do insert, you insert something at the front of the list in its bucket. Like, there's no reason to insert at the end. That would be silly. So the plot thickens when we think about the other two operations, deletion and the lookup. So let's just think about lookup. Deletion's basically the same. So how do we implement lookup? Well, remember when we get some key x, we invoke the hash function. We call h(x). That tells us what bucket to look in. So if it tells us 17, we know that, you know, x may or may not be in the hash table. But at this point we know that if it's in the hash table, it's got to be in the linked list that's in the seventeenth bucket. So now we descend into this bucket. We find ourselves a linked list. And now, we have to resort to just an exhaustive search through this list in the seventeenth bucket, to see whether or not X is there. So we know how long it takes to search a regular list for some element. It's just linear and the list length. And now we're starting to see why the hash function might matter. Right, so suppose we insert 100 objects into a hash table with 100 buckets. If we have a super lucky hash function, then perhaps each bucket will get its own object. There'll be one object in each of the lists, in each of the buckets, so. Theta of the list length is just theta of one. We're doing great. Okay? So, a constant, constant link to lists, means constant time insert delete. A really stupid hash function would map every single object to bucket number zero. Okay, so then if you insert 100 objects, they're all in bucket number zero. The other 99 buckets are empty. And so every time you do insert or delete, it's just resorting to the naive linked list solution. And the running time is going to be linear and the number of objects in the hash table. So the largest list length could vary anywhere from m/n, where m is the number of objects, this is when you have equal linked lists, to if you use this ridiculous constant hash function, m, all the objects in the same list. And so the main point I'm trying to make here, is that you know, first of all, at least with chaining, where the running time is governed by the list length, the running time depends crucially on the hash function. How well the hash function distributes the data across the different buckets. And something analogous is true for hash tables that use open addressing. Alright so here there aren't any lists. So you don't, there's no linked lists to keep track of. So the running time is going to be governed by the length of the probe sequence. So the question is how many times do you have to look at different buckets in the hash table before you either find the thing you're looking for, or if you're doing insertion, before you find an empty bucket in which to insert this new object. So the performance is governed by the length of the probe sequence. And again, the probe sequence is going to depend on the hash function. For really good hash functions in some sense, stuff that spreads data out evenly, you expect probe sequences to be not too bad. At least intuitive, And for say the constant function you are going to expect these probe sequences to grow linearly with the numbers of object you insert into the table. So again this point remains true, the performance of a hash table in either implementation really depends on what hash function you use. So, having built up this intuition, we can now say what it is what we want from a hash function. So first we want it to lead to good performance. I'm using the chaining implementations as a guiding example. We see that if we have a size of a hash table, n, that's comparable to the number of objects, m, it would be really cool if all of the lists had a length that was basically constant; therefore we had our constant time operations. So equal length lists is way better than unequal length lists in a hash table with chaining. So we, we want the hash function to do is to spread the data out as equally as possible amongst the different buckets. And something similar is true with open addressing; in some sense you want hash functions to spread the data uniformly across the possible places you might probe, as much as possible. And in hashing, usually the gold standard for spreading data out is the performance of a completely random function. So you can imagine for each object that shows up you flip some coins. With each of the n buckets equally likely, you put this object in one of the n buckets. And you flip independent coins for every different object. So this, you would expect, you know, because you're just throwing darts at the buckets independently, you'd expect this to spread the data out quite well. But, of course, it's not good enough just to spread data out. It's also important that we don't have to work too hard to remember what our hash function is and to evaluate it. Remember, every time we do any of these operations, an insert or a delete or a lookup, we're going to be applying our hash function to some key x. So every operation includes a hash function evaluation. So if we want our operations to run a constant time, evaluating the hash function also better run in constant time. And this second property is why we can't actually implement completely random hashing. So there's no way we can actually adjust when say we wanted to insert Alice's phone number, flip a new batch of random coins. Suppose we did. Suppose we flipped some random coins and it tells us to put Alice's phone number into the 39th bucket, while. Later on, we might do a lookup for Alice's phone number, and we better remember the fact that we're supposed to look in the 39th bucket for Alice's phone number. But what does that mean? That means we have to explicitly remember what choice we made. We have to write down. You get a list of effects that Alice is in bucket number 39. In every single insertion, if they're all from in the point of coin flips, you have to remember all of the different random choices independently. And this really just devolves back to the naive list base solution that we discussed before. So, evaluating the hash function is gonna take us linear time and that defeats the purpose of a hash table. So we again we want the best of both worlds. We want a hash function which we can store in ideally constant space, evaluate in constant time, but it should spread the data out just as well as if we had this gold standard of completely random hashing. So I want to touch briefly on the topic of how you might design hash functions. And in particular good hash functions that have the two properties we identified on the previous slide. But I have to warn you, if you ask ten different, you know, serious hardcore programmers, you know, about their approach to designing hash functions, you're likely to get ten somewhat different answers. So the design of hash functions is a tricky topic, and, it's as much art as science at this point. Despite the fact that there's a ton of science, there's actually a very beautiful theory, about what makes good hash functions. We'll touch on a little bit of that in a, in a different optional video. And if you only remember one thing of, you know, from this video or from these next couple slides, the thing to remember is the following. Remember that it's really easy to inadvertently design bad hash functions and bad hash functions lead to poor hash table performance. Much poorer than you would expect given the other discussion we've had in this video. So if you have to design your own hash function, do your homework, get some examples, learn what other experts are doing and use your best judgment. If you do just something without thinking about it, it's quite possible to lead to quite poor performance, much poorer than you were expecting. So to drive home this point, suppose that you're thinking about keys being phone numbers. So let's say, you know, I'm gonna just be very kinda United States centric here. I'm just gonna focus on the, the ten digit phone numbers inside the US. So the universe size here is ten to the ten, which is quite big. That's probably not something you really want to implement explicitly and let's consider an application where, you know, you're only say, keeping track of at most, you know, 100 or 1,000 phone numbers or something like that. So we need to choose a number of buckets, let's say we choose 1,000 buckets. Let's say we're expecting no more than 500 phone numbers or so. So we double that, we get a number of buckets to be equal 1,000. And now I got to come up with a hash function. And remember, you know, a hash functions by definition. All it does is map anything in the universe to a bucket number. So that means it has to take as input a ten digit phone number and spit as output some number between zero and 999. And, beyond that we have flexibility of how to define this mapping. Now, when you are dealing with things that have all these digits it's very tempting to just project on to a subset of the digits. And, if you want a really terrible hash function, just use the most significant digits of a phone number to define a mapping from phone numbers to buckets. Alright, so I hope it's clear why this is a terrible choice of a hash function. Alright, so maybe you're a company based in the San Francisco Bay area. The area code for San Francisco is 415. So if you're storing phone numbers from customers in your area. You know maybe twenty, 30, 40 percent of them are gonna have area codes 415. All of those are going to hash to exactly the same bucket, bucket number 415 in this hash table. So you're gonna get an overwhelming percentage of the data mapping just to this one bucket. Meanwhile you know not all 1000 possibilities of, of these three digits are even legitimate area codes. Not all three digit numbers are area codes in the United States. So there'll be buckets of your hash table which are totally guaranteed to be empty. So you waste a ton of space in your hash table, you have a huge list in the bucket corresponding to 415, you have a huge list in the bucket corresponding to 650, the area code at Stanford. You're gonna have a very slow look up time for everything that hashes to those two buckets and there's gonna be a lot of stuff that hashes to those two buckets, So terrible idea. So a better but still mediocre hash function would be to do the same trick but using the last three digits instead of the first three digits. This is better than our terrible hash function because there aren't ridiculous clumps of phone numbers that have exactly the same last three digits. But still, this is sort of assuming you're using this hash function as tantamount to thinking that the last three digits of phone numbers are uniformly distributed among all of the 1,000 possibilities. And really there's no evidence if that's true. Okay? And so there's going to be patterns and phone numbers that are maybe a little subtle to see with the naked eye, but which will be exposed if you try to use a mediocre hash function like this. So let's look at another example. Perhaps you are keeping track of objects just based by where they are laid out in memory. So in other words the key for an object is just gonna be its memory location and if these things are in bytes, then you are guaranteed that every memory location will be a multiple of four. So for a second example let's think about a universe where the possible keys are the possible memory locations, So here you're just associating objects with where they're laid in memory, and a hash function is responsible for taking in as input some memory location of some object and spitting out some bucket number. Now generally, because of, you know the structure of bytes and so on, our memory locations are going to be multiples of some power of two. In particular, memory locations are going to be even, And so a bad choice of a hash function. Would be to take, remember, the hash function takes the input of the memory location, which is, you know, some possibly really big number, and we wanna compress it, we want to output a bucket number. Now let's think of a hash table where we choose N equals 10^3, or 1000 buckets. So then the question is, you know, how is this hash function going to take this big number, which is the memory location, and squeeze it down to a small number. Which is one of the buckets and so let's just use the same idea as in the mediocre hash function, which is we're gonna look at the least significant bits so we can express that using the mod operator. So let's just think about we pick the hash function h(x) where h is the memory location to be x mod 1000 There again, you know, the meaning of 1,000 is that's the number of buckets we've chosen to put in our hash table because, you know, we're gonna remember roughly at most 500 different objects. So don't forget what the mod operation means, this means you just, essentially subtract multiples of 1,000 until you get down to a number less than 1,000. So in this case, it means if you write out x base ten, then you just take the last three digits. So in that sense, this is the same hash function as our mediocre hash function when we were talking about phone numbers. So we discussed how the keys here are all going to be memory locations; in particular they'll be even numbers. And here we're taking their modulus with respect to an even number. And what does that mean? That means every single output of this hash function will itself be an even number. Right, you take an even number, you subtract a bunch of multiples of a 1000, you're still going to have an even number. So this hash function is incapable of outputting an odd number. So what does that mean? That means at least half of the locations in the hash table will be completely empty, guaranteed, no matter what the keys you're hashing is. And that's ridiculous. It's ridiculous to have this hash table 50 percent of which is guaranteed to be empty. And again, what I want you to remember, hopefully long after this class is completed is not so much these specific examples, but more the general point that I'm making. Which is, it's really easy to design bad hash functions. And bad hash functions lead to hash table performance much poorer than what you're probably counting on. Now that we're equipped with examples of bad hash functions. It's natural to ask about, you know, what are some good hash functions? Well it's actually quite tricky to answer that question. What are the good hash functions, and I'm not really going to answer that on this slide. I don't promise about hash functions that I'm going to tell you about right now, are good in a very strong sense of the word. I will say these are not obliviously bad hash functions, they're let's say, somewhat better hash functions. And in particular if you just need a hash function, and you need a quick and dirty one, you don't want to spend too much time on it. The method that I'm going to talk about on this slide is a common way of doing it. On the other hand, if you're designing a hash function for some really mission-critical code, you should learn more than what I'm gonna tell you about on this slide. So you, you should do more research about what are the best hash functions, what's the state of the art, if you have a super important hash function. But if you just need a quick one what's, what we say on this slide will do in many, in most situations. So the design of a hash function can be thought of as two separate parts. So remember by definition a hash function takes as input something from the universe. An IP address, a name, whatever and spits out a bucket number. But, it can be useful to factor that into two separate parts. So first you take an object which is not intrinsically numeric. So, something like s string or something more abstract. And you somehow turn an object into a number, possibly a very big number. And then you take a possibly big number and you map it to a much smaller number, namely the index of a bucket. So in some cases I've seen these two steps given the names like the first step is formulating the hash code For an object, and then the second step is applying a compression function. In some cases, you can skip the first step. So, for example, if your keys are social security numbers, they're already integers. If they're phone number, they're already integers. Of course, there are applications where the objects are not numeric. You know, for example, maybe they're strings, maybe you're remembering names. And so then, the production of this hash code basically boils down to writing a subroutine that takes, as input, a string, and outputs some possibly very big number. There are standard methods for doing that, it's easy to find resources to, to give you example code for converting strings to integers you know, I'll just say one or two sentences about it. So you know each character in a string it is easy to regard as a number in various ways. Either you know just say it is ASCII, well ASCII code then you just have to aggregate all of the different numbers, one number per character into some overall number and so one thing you can do is you can iterate over the characters one at a time. You can keep a running sum. And with each character, you can multiply the running sum by some constant, and then add the new letter to it, and then, if you need to, take a module list to prevent overflow. And the point of me giving you this one to two sentence of the subroutine is just to give you a flavor of what they're like, and to make sure th at you're just not scared of it at all. Okay? So it's very simple programs you can write for doing things like converting from strings to integers. But again, you know, I do encourage you to look it up on the Web or in a programming textbook, to actually look at those examples. Okay? But there are standard methods for doing it. So that leaves the quest, question of how to design this compression function. So you take as input this huge integer. Maybe your keys are already numeric, like Social Security numbers or IP addresses. Or maybe you've already some subroutine to convert a string, like your friend's name, into. Some big number, but the point is you have a number in the millions or the billions, and you need to somehow take that and output one of these buckets. And again think of there being maybe a thousand or so buckets. So the easiest way to do that is something we already saw in a previous slide, which is just to take the modulus, With respect to the number of buckets. Now certainly one positive thing you can say about this compression function is its super simple, Both in the sense that it's simple to code and in the sense that it's simple to evaluate. Now remember that was our second goal for a hash function. It should be simple to store, it is actually nothing to store. And it should be quick to evaluate. And this certainly fits the bill. Now the problem is, remember the first. Property of a hash function that we wanted is that it should spread the data out equally. And what we saw in the previous slide is that at least if you choose the number of buckets N poorly, then you can fail to have the first property. And in that respect you can fail to be a good hash function. So if for example if N is even and all of your objects are even, then it's a disaster, all of the odd buckets go completely empty. And honestly, you know, this is a pretty simplistic method. Like I said, this is a quick and dirty hash function. So, no matter how you choose the number of buckets N, it's not gonna be a perfect hash function in any sense of the word. That said, there are some rules of thumb for how to pick the number of buckets, how to pick this modulus, so that you don't get the problems that we saw on the previous slide. So, I'll conclude this video just with some standard rules of thumb. You know, if you just need a quick and dirty hash function, you're gonna use the, the modulus compression function, how do you choose N? Well, the first thing is we definitely don't want to have the problem we had in the last slide, where we're guaranteed to have these empty buckets no matter what the data is. So what went wrong in the previous slide? Well. The problem is that all of the data elements were divisible by two. And the hash function modulus, the number of buckets, was also divisible by two. So because they shared a common factor, namely two, that guaranteed that all of the odd buckets remained empty. And this is a problem, more generally, if the data shares any common factors with N, the number of buckets. So, in other words, if all of your data elements are multiples of three, and the number of buckets is also a multiple of three, you got a big problem. Then everything's gonna hash into bucket numbers which are multiples of three, too, that's if your hash table will go unfilled. So the upshot is, you really want the number of buckets to not share any factors With the data that you're hashing. So, how do you reduce the number of common factors? Well, you just make sure the number of buckets has very few factors, which means you should choose N to be a prime number, 'kay? A number that has no nontrivial factors, And let me remind you, the number of buckets should also be comparable to the size of the set that you're planning on storing. Again, at no point did we need "N" to be, you know, very closely connected to the number of elements that you're storing just within, say some small constant factor. And you can always find a prime within a small constant factor of a target number of elements to store. If the number of buckets in your hash table isn't too big, if it's just in say the thousands or maybe the tens of thousands, then, you know, you can just look up a list of all known primes up to some point, and you can just sort of pick out a prime which is about the magnitude that you're looking for. If you're gonna use a really huge number of buckets in the millions or more, then there are algorithms you can use for primarily testing which will help you find a prime in about the range that you're looking for. >> So that's the first order rule of thumb you should remember if you're using the modulus compression function, which is set the number of buckets equal to a prime. So you're guaranteed to not have non-trivial common factors of the modulus shared by all of your data. So there's also a couple of second order optimizations, which people often mention. And you also don't want the prime; you want the prime to be not too close to patterns in your data. So what does that mean, Patterns in your data? Well, in the phone number example we saw that patterns emerged in the data when we expressed it base ten. So for example, there is crazy amounts of Lumping in the first three digits when we expressed a phone number-based ten, Because that corresponded with the area code. And then, with. Memory locations when we express, express it base two, there are crazy correlations in the low orbits. And these are the two most common examples. Either there's some digit, to the base ten representation or digits in the base two representation where you have, you know, patterns that is non-uniformity. So that. Suggests that the prime, that, N that you choose, you know, all else being equal, shouldn't be too close to a power of two, and shouldn't be too close to a power of ten. The thinking being that, that will spread more evenly data sets that do have these patterns in terms of base two representation, or base ten representations. So in closing, this is a recipe I recommend for coding of hash functions if what you're looking to do is sort of minimize program ming, programmer time, subject to not coming up with a hash function, which is completely broken. But I want to reiterate, this is not the state of the art in hash function design. There are hash functions which are in some sense better than the ones that expand on this slide. If you're responsible for some really mission critical code that involves a hash function, you should really study more deeply than we've been able to do here. We'll touch on some issues in, of the different optional video, but really you should do additional homework. You should find out about the state-of-the-art about hash function design. You should also look into implementations of open addressing in those probing strategies. And above all you really should consider cold, coding up multiple prototypes and seeing which one works the best. There's no silver bullet, there's no panacea in the design of hash tables. I've given you some high-level guidance about the different approaches. But ultimately it's gonna be up to you to find the optimal implementation for your own application.

# 15-1-Pathological Data Sets and Universal Hashing Motivation

This sequence of videos are going to take it to the next level with Hash tables and understand more deeply the conditions under which they perform well, amazingly well in fact As you know, we've constant time performance for all of their operations. The main point of this first video is to explain in sense in which every hash function has its own Kyptonite. A pathological data set for it which then motivates the need to tread carefully with mathematics in the subsequent videos. So, a quick review So remember that the whole purpose of a hash table is to enable extremely fast look ups, ideally constant time lookups. Now, of course, to have anything to look up, you have to also allow insertions. So all hash tables are going to export those two operations And then, sometimes, a hash table also allows you to delete elements from it. That depends a little bit on the underlying implementation. So certainly when you have it implemented using chaining, that's when you have one linked list per bucket, it's very easy to implement deletion. Sometimes, with open addressing, it's tricky enough, you're just gonna wind up punting on deletion. So when we first started talking about hash tables, I encouraged you to think about them logically. Much the way you do as an array, except instead of being indexed just by the positions of an array, it's indexed by the keys that you're storing. So just like an array via random access supports constant time look up, so does a hash table. There was some fine print however with hash tables. Remember there were these two caveats. So the first one is that the hash table better be properly implemented And this means a couple of things. So, one thing that it means is that the number of buckets better be commensurate with the number of things that you're storing in the Hash table, we'll talk more about that in a second. The second thing it means is you better be using a descent Hash function. So we discussed in a previous video the perils of bad Hash functions, and it will be even more stringent with our demands on Hash functions in the videos to come. The second caveat which I'll try to demystify in a few minutes is that you better have non-pathological data. So, in some sense, for ever Hash table there's Kyptonite , a pathological data set that will render its performance to be quite poor. So in the video on implementation detail we also discussed how hash tables inevitably have to deal with collisions. So you start seeing collisions way before your hash table's start filling up so you need to have some sort of method for addressing two different keys that map to exactly the same bucket. Which one do you put there? Do you put both there or what? So there's two popular approaches let me remind you what they are First called chaining. So this is a very natural idea, where you just keep all of the elements that hash to a common bucket in that bucket. How do you keep track of all of them? Well, you just use a linked list. So, in the seventeenth bucket, you will find all of the elements which ever hashed to bucket number 17. The second approach which also has plenty of applications in practice is open addressing. Here the constraint is that you are only going to store one item one key in each bucket. So if two things mapped the bucket number seventeen, you gotta find a separate place for one of them And so the way that you handle that is you demand from your hash function not merely one bucket but rather a whole probe sequence. So the sequence of buckets so that if you try to hash something into bucket number seventeen, 17's already occupied then you go on to the next. Bucket in the probe sequence. You try to insert it there And if it, you fail again you go to third bucket in the sequence. You try to insert it there, and so on. So we mentioned briefly the sorta two ways you can specify probe sequences. One is called linear probing. So this is where if you fail in bucket seventeen you move on to eighteen and then nineteen and then twenty and then 21. And you stop once you find an empty one And that's where you insert the new element And another one is double hashing And this is where you use a combination of two hash functions Where the first hash function specifies the initial bucket that you probe. The second hash function specifies the offset for each subsequent probe. So for example if you have a given elements say the name Alice and the two hash functions give you the number 17 and 23 then the corresponding finding probe sequence is going to be initially 17, failing that we'll try 40, still failing that we'll try 63, failing that we'll try 86 and so on. So in a course on the design and analysis of algorithms like this one you typically talk a little bit more about chaining than you do open addressing. That's not to imply that chaining is somehow the more important one both of these are important But chaining is a little easier to talk about mathematically. So we will talk about it a little bit more cuz I'll be able to give you complete proofs for chaining whereas complete proofs for open addressing would be outside the scope of this course. So I'll mention it in passing but the details will be more about chaining just for mathematical ease. So, there's one very important parameter which plays a big role in governing the performance of a hash table, and that's known as the load factor, or simply the load, of a hash table. And it's a very simple definition. It just talks about how populated, a typical bucket of the hash table is. So it's often denoted by alpha And in the numerator is the number of things that have been inserted, and not subsequently deleted in the hash table. Divided by the number of buckets in the hash table. So, as you would expect, as you insert more and more things into the hash table, the load grows, keeping the number of items in the hash table fixed as you scale up the number of buckets, the load drops. So just to make sure that the notion of the load is clear, and that also you're clear on the different strategies for resolving collisions, the next quiz will ask you about the range of relevant alphas for the chaining and open addressing implementations of the hash table. Alright, so the correct answer to this quiz question is the third answer. Load factors bigger than one do make sense they may not be optimal but they at least make sense for hash tables that implement with chaining but they don't make sense for hash tables with open addressing. And the reason is simple remember in open addressing you are required to store only one object per bucket so as soon as the number of objects exceeds the number of buckets there is no where to put the remaining objects. So the hash the hash table will simply crash if, if load factor is bigger than one. On the other hand a hash table with chaining there is no obvious problems with load factor bigger than one so you can imagine a load factor equal to two say, say you insert 2,000 objects into a hash table with 1,000 buckets. You know that means, hopefully At least in the best case. Each buckets just gonna have a length list with two objects in it. So there's no big deal. With having load factors bigger than one, and hash tables With chaining. Alright, so let's then make a, a quite easy but also very important observation about a necessary condition for hash tables to have good performance And this goes into the first caveat that you better properly implement the hash table if you expect to have good performance. So the first point is that you're only gonna have constant time look-ups if you keep the load to be constant. So for a hash table with open addressing, this is really obvious, because you need alpha not just O(1) but less than one, less than 100 percent full, otherwise the hash table is just gonna crash, cuz you don't have enough room for all of the items, but even for hash tables that you implement using chaining, where they at least make sense for load factors which are bigger than one, you'd better keep the load not too much bigger than one if you want to have constant-time operations. Right, so if you have, say, a hash table with. N buckets and you hash in NlogN objects, then the average number of objects in a given bucket is gonna be logarithmic And remember, when you do a lookup, after you hash to the bucket, you have to do an exhaustive search through the linked list in that bucket. So if you have NlogN objects and you hashed it with N buckets, you're expecting more like logarithmic lookup time - not constant lookup time And then, as we discussed with open addressing, of course, you need not just alpha = O(1), but alpha less than one. And in fact, alpha better be well below one. You don't want to let an open addressing table get to a 90 percent load or something like that. So I'm going to write need Alpha less than less than one. So that just means you don't want to let the load grow too close to 100%, you will see performance degrade. So again, I hope the point of this slide is clear. If you want good hash table performance, one of the things you're responsible for is keeping the load factor under control. Keep it at most a small constant with open address and keep it well below 100%. So you might wonder what I mean by controlling the load. After all, you know you writing this hash table when you have no idea what some client's gonna do with it. They can insert or delete whatever they want. So how do you, how do you control alpha? Well, what you can control under the hood of your hash table implementation is the number of buckets. You can control the denominator Of this alpha so if the numerator starts growing at some point the denominator is going to grow as well. So what actual implementations of hash tables do is they keep track of the population of the hash table. How many objects are being stored, and as this numerator grows, as more and more stuff gets inserted, the implementation ensures that the denominator grows at the same rate so that the number of buckets also increases. So if alpha exceeds some target, you know, that could be say 75%, .75, or maybe it's.5. Then what you can do is you can double the number of buckets, say in your hash table. So you define a new hash table, you have a new hash function with double the range, and now having doubled the denominator. The load has dropped by a factor two. So that's how you can keep it under control. Optionally, if space is at a real premium, you can also shrink the hash table if there's a bunch of deletions, say, in a chaining implementation. So that's the first take away point about what has to be happening correctly under the hood in order to get the desired guarantees for hash table performance you gotta control the load. So you have to have a hash table whose size is roughly the same as the as the number of objects that you are storing. So the second thing you've gotta get right and this is something we've touched on in the implementation videos is you better use a good enough hash function And so what's a good hash function? It's something that spreads the data out evenly amongst the buckets And what would really be awesome would be a hash function which works well independent of the data And that's really been the theme of this whole course so far, algorithmic solutions which work independent of any domain assumptions. No matter what the input is, the algorithm is guaranteed to, for example, run blazingly fast And I can appreciate that this is exactly the sort of thing you would want to learn from a course like this, right? Take a class in the design analysis of algorithms and you learn the secret hash function which always works well. Unfortunately I'm not going to tell you such a hash function And the reason is not cuz, I didn't prepare this lecture. The reason is not because people just haven't been clever enough to discover such a function. The problem is much more fundamental. The problem is that such a function cannot exist. That is for every hash function it has its own kryptonite. There is a pathological dataset under which the performance of this hash function will be as bad as the most miserable constant hash function you'd ever seen. And the reason is quite simple; it's really an inevitable consequence of the compressing that hash functions are effectively implementing from some massive universe to some relatively modest number of buckets. Let me elaborate. Fix any hash function as clever as you could possibly imagine. So this hash function maps some universe through the buckets indexed from 0 to n -1. Remember in all of the interesting situations of hash functions, the universe size is huge, so the cardinality of U should be much, much bigger than n. That's what I'm going to assume here. So for example, maybe you're remembering people's names, and then the universe is strings, which have, say at most, 30 characters, and N, I assure you in any application is going to be much, much, much smaller than say, 26 raised to the thirtieth power. So now let's use a variant on the pigeon hole principle, and acclaim that at least one of these n buckets has to have at least a 1/n fraction of the number of keys in this universe. That is, there exists a bucket I, somewhere between 0 and n -1 Such that, at least the cardinality of the universe over N Keys Hash to I get mapped to I under this hash function H. So the way to see this is just to remember the picture of a hash function mapping in principal any key from the universe, all keys from the universe to one of these buckets. So the hash function has to put each key somewhere in one of the n buckets So one of the buckets has to have at least a 1/n fraction above all of the possible keys. One more concrete way of thinking, thinking about it is that you might want to think about a hash table implemented with chaining. You might want to imagine, just in your mind, that you hash every single key into the hash table. So this hash table is going to be insanely over-populated. You'll never be able to store it on the computer because it will have the full cardinality of U objects in it, but it has U objects, it only has n buckets. One of those buckets has to have at least U /n fraction of the population. So the point here is that no matter what the hash function is no matter how clever you build it there's gonna be some buckets say bucket number 31 which gets its fair share of the universe maps to it. So having identified this bucket, bucket number 31 where it gets at least its fair share of the universe maps to it now to construct our pathological dataset all we do is picked from amongst these elements that get mapped to bucket number 31. So for such as a data set and we can make this data set basically as large as we want because the cardinality of U/n is unimaginably big, because U itself is unimaginably big Then in this data set, everything collides. The hash function maps every single one to bucket number 31 and that's gonna lead to Terrible hash table performance. Hast table performance which is really no better than the naive linked list solutions. So for example an hash table with collisions, in bucket 31, you'll just find a link listed every single thing that's ever been inserted into the hash table and for open addressing maybe it's a little harder to see, what's going on but again if everything collides, you're gonna basically wind up with linear time performance A far cry from constant time performance. Now, for those of you to whom this seems like just sort of pointless, abstract mathematics, I want to make two points. The first is that at the very least these pathological data sets. Tells us, indicates, that we will have to discuss hash functions in a way differently than how we've been discussing algorithms so far. So when we talked about merge sort, we just said it runs in n log n time. No matter what the input is. Whether we discuss the Dijkstra's algorithm It runs in n log n time, no matter what the input is. That first search, that first search many a time no matter what the input is. We're gonna have to say something different about hash functions. We'll not be able to say that a hash table has good performance, no matter what the input is. This slide shows that's false. The second point I want to make is that while this pathological data sets of course are not likely to arise just by random chance. Sometimes you're concerned about somebody constructing a pathological data set for your hash function, for example in a denial service attack. So there's a very clever illustration of exactly this point in a research paper, from 2003 By Crosby and Wallach. So the main point of Crosby and Wallach was that there's a number of real world systems And maybe there, most interesting application was a network intrusion detection system, for which you could bring them to their knees by exploiting badly designed hash functions. So these were all applications that made crucial use of hash tables and, the feasibility of these systems really, completed depended on getting cost and time performance from the hash tables. So if you could exhibit a pathological data set for these hash tables and make the performance devolve to linear, devolve to that of a simple link list solution, the systems would be broken, they would just crash of they'd fail to function. Now we saw in the last slide that every hash table does have its own Kryptonite. Has a pathological data set But the question is. How can you come up with such a pathological data set if you're trying to do a denial of service attack on one of these systems? And so the systems that Crosby and Wallach looked at generally shared two properties. So first of all, they were open source. You could inspect the code. You could see what hash function it was that they were using And second of all, the hash, function was often very simplistic. So it was engineered for speed more than anything else And as a result, it was easy to, just be inspecting the code, reverse engineer a data set. That really did break the hash table. That led it That devolved the performance to linear. So for example, in the network intrusion detection application, there was some hash table that was just remembering the IP addresses of packets that we re going through, because it was looking for patterns of pack, data packets that seemed to indicate some sort of intrusion. And Crosby and Wallach just showed how sending a bunch of data packets to this system with cleverly chosen sender IP's really did just crash the system because the hash table performance blew up to an unacceptable degree. So how should we address this fact of life that every hash function has a pathological data set? And that question is meaningful both from a practical perspective, so what hash functions should we use if we are concerned about someone constructing pathological data sets, for example, the implemented denial-of-service attack, and secondly, mathematically, if we can't give the kinds of guarantees we've given so far, data-independent guarantees, how can we mathematically say that hash functions have good performance. So let me mention two solutions, the first solution is, is meant more just on the practical point. You know what hash function should you implement if you are concerned with someone creating pathological data sets? So there are these things called cryptographic hash functions. There for example, one cryptographic hash function or really family of hash functions, for different numbers of buckets, is SHA-2 And these are really outside the scope of this course. I mean, you'll learn more about this in a In a course on cryptography And obviously using these keywords you can look it up on the web and read more about it. The one point I want to make is, you know these cryptographic hash functions like SHA2, they themselves, they do have pathological datasets. They have their own version of kryptonite. The reason that they work well in practice is because it's infeasible to figure out what this pathological dataset is So unlike the very simplistic hash functions which Crosby and Wallach found in the source code of their applications, where it was easy to reverse-engineer bad datasets, for something like SHA2 nobody knows how to reverse-engineer a bad dataset for it And when I say infeasible, I mean in the usual cryptographic sense, in a way similar to how one would say it's infeasible to break the RSA cryptosystem if you implement it properly, or it's infeasible to factor large numbers except in very special case, and so on. So that's all I'm going to say about cryptographic hash functions. I also want to mention a second solution, which would be reasonable both in practice, and also for which we can say a number of things mathematically about it Which is to use randomization. So more specifically we're not going to design, a single. Clever hash function Because again, a single hash function we know Must have a pathological data set But we're gonna design a very clever, family of hash functions And then, at run time. We're going to pick, one of these hash functions at random. Another kind of guarantee you are going to want, we are going to be able to prove on a family affairs function would be very much in the spirit of quick sort. So you would call that in the quick sort algorithm, pretty much. For any fixed pivot sequence there is a pathological input for which quick sort will devolve to quadratic running time. So our solution was randomize quick sort which is rather than committing up front to any particular method of choosing pivots at run time we are gonna pick pivots randomly. What did we prove about quick sort? We proved that for any possible input for any possible array the average running time with quick sort was O(nlogn) with the average was over the run time random choices of quick sort. Here we are gonna do the same thing we'll now be able to say for any dataset. On average, with respect to our run time choice of a hash function. The hash function will perform well, in the sense that it will spread out the data of the data set evenly. So we flip to the quantifiers from the previous slide. There, we said if we pre-commit to a single hash function. If we fix one hash function, Then there's a data set that breaks the hash function. Here we're flipping it, we're saying for each fixed data set a random choice of a hash function is going to do well on average on that data set. Just like in Quick Sort. This doesn't mean notice that we can't make our program open source. We can still publish code which says here is our family of hash functions and in the code we would be making a random choice from this set of hash functions But the point is by inspecting the code you'll have no idea what was the real time random choices made by the algorithm. So you'll know nothing about what the actual hash function is so you won't be able to reverse engineer pathological dataset for the real time choice of the hash function. So the next couple of videos are gonna elaborate on the second solution Of using a real time random choice of a hash function as a way of saying. You do well on every data set At least on average. So let me just give you a road map of where we're going to go from here. So I'm going to break the discussion of the details of a randomized solution into three parts, spread over two videos. So in the next video we're going to begin with the definition of what do I mean by a family of hash functions, so that if you pick one at random, you're likely to do pretty well. So that's a definition that's called a universal family of hash functions. Now a mathematical definition by itself is worth approximately nothing. For it to be valuable, it has to satisfy two properties. So first of all there have to be interesting and useful examples that meet the definition. So that is, there better be Useful looking hash functions that meet this definition of a universal family. So the second thing will be to show you that they do indeed exist And then the other thing a mathematical definition needs is applications. So that if you can meet, the definition. Then, good things happen. So that'll be part

# 15-2-Universal Hashing - Definition and Example [Advanced - Optional]

So now that we understand why we can't have a single hash function which always does well at every single data set, that is every hash function is subject to a pathological data sets. We'll discuss the randomize solution of how we can have a family of hash functions and if you make a real time decision about which hash function to use, you're guaranteed to do well on average, no matter what the data is. So let me remind you of the three prong plan that I have for this part of the material. So in this video, we'll be covering the first two. So part one, which we'll accomplish in the next slide, will be to propose a mathematical definition of a good random hash function. So formerly, we're going to define a universal family of hash functions. Now, what makes this definition useful, well, two things. And so, part two, we'll show that there are examples of simple and easy to compute hash functions that meet this definition, that are universal in the sense described on the next slide. So that's important. And in the third part, which we'll do in the next video, will be the mathematical analysis of the performance of hashing, specifically with chaining when you use universal hashing. And we'll show that if you pick a random function from a universal family, then, the expected performance of all of the operations are constant. Assuming, of course, of the number of buckets is comparable to the number of objects in the hash table which we saw earlier is a necessary condition for good performance. So let's go ahead and get started and let's say what we mean by a good random hash function. So for this definition, we'll assume that the universe is fixed. So maybe it's IP addresses, maybe it's our friends names. Maybe it's configurations of a chessboard, whatever. But there's some fixed universe u and we'll also assume we've decided on the number of buckets n. And we call the set H universal if and only if it meets the following condition. In English, the condition says that for each pair of distinct elements, the probab ility that they collide should be no larger than with the gold standard of perfectly uniform random hashing. So for all distinct keys from the universe, call them x and y, what we want is that the probability if we choose a random hash function, h, from the set script h, the probability that x and y collide. And again, just to be clear, what that means is that, x and y hash to exactly the same bucket under this hash function h, this should be no more than 1/n, and don't forget n is the number of buckets. Again, to interpret this, you know, 1/n, where does this come from? So, we said earlier that an impractical but in some sense, gold standard hash function would be to just independently for each key, assign it bucket uniformly and random with different keys being assigned independently. Remember the reason this is not a practical hash function is because, you'd have to remember where everybody went. And then that would basically require maintaining a list which would devolve to the list solution, so you don't want that. You want hash functions where you have to store almost nothing and we can evaluate them in constant time. But, if we throw out those requirements of small space and small time then, random function should spread stuff out pretty evenly, right? I mean that's what they are doing. They're throwing darts completely at random at these n buckets. So what would be the collision probability of two given keys say, of Alice and of Bob if you are doing everything independently and uniformly at random. Well, you know, first Alice shows up and it goes to some totally random bucket, say, bucket number seventeen. Now, Bob shows up. So, what's the probability that it collides with Alice? Well, we have these n buckets that Bob could go to, each is equally likely, and there's a collision between Alice and Bob if and only if Bob goes to bucket seventeen. Since, each bucket's equally likely, that's only a one in n probability. So really what this condition is saying, is that, for each pair of elements, the collision probability should be as small, as good as with the holy grail of perfectly random hashing. So this is a pretty subtle definition, perhaps the most subtle one that we see in this entire course. So, to help you get some facility with this, and to force you to think about it a little deeply, the next quiz which is probably harder than a typical in class quiz, asks you to compare this definition of universal hashing with another definition and ask you to figure out to what extent they're the same definition. So the correct answer to this quiz question is the third one that there are hash function families h, that satisfy the condition on this slide that are not universal. On the other hand, there are hash function families H, which satisfy this property and are universal. So, I'm going to give you an example of each. I'd encourage you to think carefully about why this is an example and a non-example offline. So an easy example to show that sometimes the answer is yes, you have universal hash function families h, which also satisfy this property of the slide, would be to just take H to be the set of all functions from mapping the universe to the number of buckets. So that's an awful lot of functions, that's a huge set, but it's a set nonetheless. And, by symmetry of having all of the functions, it both satisfies the property of this slide. It is indeed true that exactly a one on one refraction of all functions, map arbitrary key k to arbitrary bucket i. And by the same reasoning, by the same symmetry properties, this is universal. So really, if you think about it choosing a function at random function from H, is now just choosing a completely random function. So it's exactly what we've been calling perfect, random hashing. And as we discussed in the last slide, that would indeed have a collision probability of exactly 1/n for each pair of distinct keys. So, this shows sometimes you can have both this property and be universal. An example where you have the property in this slide, but you're not universa l, would be to take h to be a quite small family, a family of exactly n functions, each of which is a constant function. So it's going to be one function which always maps everything to bucket zero, that's a totally stupid hash function. There's going to be another hash function which always maps everything to bucket number one, that's a different but also totally stupid hash function, and so on. And then the end function will be the constant function that always maps everything to bucket n-1. And if you think about it, this very silly set H does indeed satisfy this very reasonable looking property on this slide. Fix any key, fix any bucket, you know say bucket number 31 what's the probability that you pick a hash function that maps this key to bucket number 31? Well, independent of what the key is, it's going to be the probability that you pick the constant hash function whose output is always 31. Since there's n different constant functions, there's a one in n probability. So, that's an example showing that in some sense, this is not as useful a property as the property of universal hashing. So this is really not what you wanted. This is not strong enough. Universal hashing, that's what you want for strong guarantees. So now that we've spent some time trying to assimilate probably the subtlest definition we've seen so far in this class, let me let you in on a little secret about the role of definitions in mathematics. So on the one hand, I think mathematical definitions often get short shrift, especially in, you know, the popular discussion of mathematical research. That said, you know, it's easy to come up with one reason why that's true, which is that any schmo can come up and write down an mathematical definition. Nobody's stopping you. So, what you really need to do is you need to prove that a mathematical definition is useful. So how do you indicate usefulness of a definition? Well you gotta do two things. First of all, you have to show that the definition is satisfied by objects of interest. For us right now, objects of interest, are hash functions, we might imagine implementing. So they should be easy to store, easy to evaluate. So there better be such hash functions meaning, that complicated universal hash function definition. The second thing is, is something good better happen if you meet the definition. And in the context of hashing, what good thing do we want to have happen? We want to have good performance. So those are the two things that I owe you in these lectures. First of all, a construction of practical hash functions that meet that definition, that's what we'll start on right now. Second of all, why meeting that definition is a sufficient condition for good hash table performance. That will be the next video. So in this example, I'm going to focus on IP addresses although the hash function construction is general, as I hope will be reasonably clear. And as many of you know, an IP address is 32 bit integer consisting of four different eight bit parts. So let's just go ahead and think of an IP address as a four two fold, the way you often see it. And since each of the four parts is eight bits, it's going to be a number between zero and 255. And the hash function that we're going to construct, it's really not going to be so different than the quick and dirty functions as we talked about in the last video although in this case we'll be able to prove that the hash function family is in fact, universal. And we're again going to use the same compression function. We're going to take the modulas with respect to a prime number of buckets. The only difference is we're going to multiply these xi's by a random set of coefficients. We're going to take a, a random linear combination of x1, x2, x3 and x4. So I'm going to be a little more precise. So we're going to choose a number of buckets, n, and as we say over and over, the number of buckets should be chosen so it's in the same ball park of the number of objects you are storing. So you know, let's say that n should be roughly double the number of objects that you are storing as initial rule of thumb. So, for example, maybe we only want to maintain something in the ball park of 500 IP addresses and we can choose n to be a prime like 997. So here's the construction. Remember, we want to produce not just one hash function, but the definition is about a universal family of hash functions. So we need a whole set of hash functions that we're ultimately going to chose one member from, at random. So, how do we construct a whole bunch of hash functions in a simple way? Here's how we do it. So you define one hash function, which I'm going to note by h sub a, a here is a four tuple. The components of which I'm going to call a1, a2, a3 and a4. And, all of the components of a are integers between zero and n-1. So they're exactly in correspondence with the indices of the buckets. So if we have 997 buckets, then each of these ai's is an integer between zero and 996. So it's clear that this defines, you know, a whole bunch of functions. So in fact, for each of the four coefficients, that's four independent choices, you have n options. Okay so each of the integers between zero and n-1 for each of the four coefficients. So that's fine, not giving a name to end of the four different functions, but what is any given function? How do you actually evaluate one of these functions? Just remember what a hash function is supposed to do. Remember you know, how it type checks it takes as input something from the universe in this case an IP address, and outputs a bucket number. And the way we evaluate the hash function h sub a, and remember a here is a 4-tuple. And remember IP address is also a 4-tuple, okay, so each component of the IP address is between zero and 255. Each component of a is between zero and n-1, so for example, between zero and 996. And what we do is just take the dot products or the inner products of the vector a and the vector x, and then we take the modulus with respect the number of buckets. So that is we take a1 x1 + a2 x2 + a3 x3 +a4 x4. Now of course, remember the x's lie be tween zero and 255, the ai's lie between the zero and n-1, so say zero and 996, you know, so you do these form of multiplications now make it a pretty big number, you might well over shoot the number of buckets n. So to get back in the range of what the buckets are actually indexed that in the end we take the module, modulus the number of buckets. So in the end we do output, a number between zero and n-1 as desired. So that's a set of a whole bunch of hash functions, n to the fourth hash functions. And each one meets the criteria of being a good hash function from an implementation perspective, right? So remember, we don't want to have to store much to evaluate a function. And for a given hash function in this family, all we gotta remember are the coefficients, a1, a2, a3 and a4. So you just gotta remember these four numbers. And then to evaluate a hash function on an IP address, we clearly do a constant amount of work. We just do these four multiplications, the three additions, and then taking the modulus by the number of buckets n. So it's constant time to evaluate, constant space to store. And what's cool is, using just these very simple hash functions which are constant time to evaluate and constant space to store, this is already enough to meet the definition of a universal family of hash functions. So this fulfills the first promise that I owed you, after subjecting you to that definition of universal hashing. Remember the first promise was, there are simple, there are useful examples that meet the definition, and then of course, I still owe you why. Meaning this definition is useful, why does it leave the good performance. But I want to conclude, this video of actually proving this theorem to you, arguing that this is, in fact, a universal family of hash functions. Right. So this should be a mostly complete proof and certainly will have all of the conceptual ingredients of why the proof works There will be one spot where I'm a little hand-wavy because we need a little number theory, and I don't want to have a big detour into number theory. And if you think about it, you shouldn't be surprised that basic number theory plays at least some role. Like as I said, we should choose the number of buckets to be prime. So that means at some point in the proof, you should expect us to use the assumption that n is prime. And pretty much always you're going to use that assumption will involve at least elementary number theory, okay? But I'll be clear about where I'm being hand-wavy. So what do we have to prove? Let's just quickly review a definition of a universal hash function. So we have our set h that we, that we know exactly what it is. What does it mean that it's universal? It means for each pair of distinct keys, so in our context it's for each pair of IP addresses, the probability that a random hash function from our family script h causes a collision, maps these two IP addresses to the same bucket should be no worse than with perfectly random hashing. So no worse than 1/n where n is the number of buckets, say like 997. So, the definition we need to meet is a condition for every pair of distinct keys. So let's just start by fixing two distinct keys. So I'm going to assume for this proof that these two IP addresses differ in their fourth component. That is that I'm going to assume that x4 is different than y4. So I hope that it's intuitively clear that, you know, it shouldn't matter, you know, which, which set of 8-bits I'm looking at. So they're different IP addresses. They differ somewhere. If I really wanted, I could have four cases that were totally identical depending on whether they differ in the first eight bits, the next 8-bits, the next 8-bits, or the last 8-bits. I'm going to show you one case, because the other three are the same. So let's just think of the last 8-bits as being different. And now, remember what the definition asked us to prove. It asked us to prove that the probability that these two IP addresses are going to collide is at most, 1/n. So we need an upper bound on the collision probability w ith respect to a random hash function from our set of n to the fourth hash functions. So I want to be clear on the quantifiers. We're thinking about two fixed IP addresses. So for example, the IP address for the New York Times website and the IP address for the CNN website. We're asking for these two fixed IP addresses, what fraction of our hash functions cause them to collide, right? We'll have some hash functions which map the New York Times and CNN IP addresses to the same bucket, and we'll have other hash functions which do not map those two IP addresses to the same bucket. And we're trying to say, that the overwhelming majority, sends them to different buckets, only a 1/n fraction at most, sends them to the same bucket. So we're asking about the probability for the choice of a random hash function from our set h that the function maps the two IP addresses to the same place. So the next step is just algebra. I'm just going to take this equation which indicates when the two IP addresses collide over a hash function. I'm going to expand the definition of a hash function, remember it's just this inner product modulo the number of buckets n, and I am going to rewrite this condition in a more convenient way. Alright, so after the algebra, and the dust has settled. We're left with this equation being equivalent to the two IP addresses colliding. So again, we're interested in the fraction of choices of a1, a2, a3, and a4, such that this condition holds, right? Sometimes it'll hold for some choices of the ai's, sometimes it won't hold for other choices and we're going to show that it almost never holds. Okay, so it fails for all but a 1/n fraction of the choices of the ai's. So next we're going to do something a little sneaky. This trick is sometimes called the Principle of Deferred Decisions. And the idea is when you have a bunch of random coin flips, it's sometimes convenient to flip some but not all of them. So sometimes fixing parts of the randomness clarifies the role that the remaining randomness is going to play . That's what's going to happen here. So let's go ahead and flip the coins, which tell us the random choice of a1, a2, and a3. So again remember, in the definition of a universal hash function, you analyze collision probability under a random choice of a hash function. What does it mean to choose a random hash function for us? It means a random choice of a1, and a2, and a3, and a4. So we're making four random choices. And what I'm saying is, let's condition on the outcomes of the first three. Suppose we knew, that a1 turns up 173, a2 shows up 122 and a3 shows up 723, but we don't know what a4 is. A4 is still equally likely to be any of zero, one, two all the way up to n-1. So remember that what we want to prove is that at most 1/n fraction of the choices of a1, a2, a3, and a4, cause this underlined equation to be true, cause a collision. So what we're going to show is that for each fixed choice of a1, a2, and a3, at most a 1/n fraction of the choices of a4 cause this equation to hold. And if we can show that for every single choice of a1, a2, and a3, no matter how those random coin flips come out, almost a 1/n fraction of the remaining outcomes satisfy the equation, then we're done. That means that at most of 1/n fraction of the overall outcomes can cause the equation to be true. So if you haven't seen the principle of, for these decisions before, you might want to think about this a little bit offline, but it's easily justified by just say two lines of algebra. Okay, so we're done with the setup and we're ready for the meat of the argument. So we have done is, we've identified an equation which is now in green, which occurs if and only if we have a collision between the two IP addresses. And the question we need to ask is, for a fixed choices of a1, a2 and a3, how frequently will the choice of a4 cause this equation to be satisfied? Cause a collision? Now, here is why we did this trick of the Principle of Deferred Decisions. By fixing a1, a2, and a3, the right hand side of this equation is now just some fixed number b etween zero and n-1. So maybe this is 773, right? The xi's were fixed upfront, the yi's were fixed upfront. We fixed a1, a2, a3 at the beginning, at the end of the last slide, and those were the only ones involved in the right hand side. So this is 773 and over on the left hand side, x4 is fixed, y4 is fixed but a4 is still random. This is an integer equally likely to be any value between zero and n-1. Now here's the key claim, which is that the left-hand side of this green equation is equally likely to be any number between zero and n-1. And I'll tell you the reasons why this key claim is true. Although this is the point where we need a little bit of number theory, so I'll be kind of hand-wavy about it. So there's three things we have going for us, the first is that x4 and y4 are different. Remember our assumption at the beginning of the proof was that, you know, the IP addresses differ somewhere so why not just assume that they differ in the last 8-bits of the proof. Again this is not important if you really wanted to be pedantic you could have three other cases depending on the other possible bits in which the IP addresses might differ. But anyway, so, because x4 and y4 are different, what that means is that x4 - y4 is not zero. And in fact, now that I write this, it's jogging my memory of something that I should have told you earlier, and forgot, which is that the number of buckets n should be at least as large as the maximum coeffcient value. So for example, we definitely want the number of buckets n in this equation to be bigger than x4, and bigger than y4. And the reason is, otherwise you could have x4 and y4 being different from each other, but they still, the difference still winds up being zero mod-n. So for example, suppose n was four, and x4 was six and y4 was ten. Then x4-10 would be -four and that's actually zero modulo four. So that's getting now what you want. You want to make sure that if x4 and y4 are different, then they're difference is non-zero modulo n. And the way you ensure that is that you just make sure n is bigger than each. So you should choose a number of buckets bigger than the maximum value of the coefficient. So in our IP address example, remember that the coefficient don't get bigger than 255. And I was suggesting a number of buckets equal the same 997. Now, in general, this is never a big deal in practice, if you only wanted to use say, 100 buckets, you didn't want to use 1000, you wanted 100, well, then you could just use smaller coefficients, right, you could just break up the IP address, instead of into 8-bit chunks, you could break it into 6-bit chunks, or 4-bit chunks, and that would keep the coefficient size smaller than the number of buckets, okay? So you could choose the buckets first, and then you choose how many bits to chunk your data into, and that's how you make sure this is satisfied. So back to the three things we have going for us in trying to prove this key claim. So x4 and y4 are different, so their difference is non-zero modulo n. So second of all, n is prime, that was part of the definition, part of the construction. And then third, a4, this final coefficient is equally likely to take on any value between zero and n-1. So, just as a plausibility argument, let me give you a proof by example. Again, I don't want to detour into elementary number theory, although it's beautiful stuff, so you know, I encourage those of you who are interested to go learn some and figure out exactly how you prove it. You really only need the barest elementary number theory to give a formal proof of this. But just to show you that is true in some simple examples, so let's think about a very small prime. Let's just say there's seven buckets and let's suppose that the difference between x4 and y4 is two. Okay, so having chosen the parameters of set n = seven, I've set the difference equal to two. What I want to do is I want to step through the seven possible choices of a4, and look at what we get in this blue circle quantity, on the left hand side of the green equation. So, we want to say the left hand's equally likely to be any of the seven numbers between zero and six, so that means that as we try our seven different choices for a4, we better get the seven different possible numbers as output. So, for example, if we set a4 = zero, then the blue circled quantity is certainly itself zero. If we set it equal to one, then it's one two, so we hit two. For two, we get two two which is four. For three, we get three two which is six. Now, when we set a4 = four, we get four two which is eight, modulo seven is one. Five two - seven is three. Six two - seven is five. So as we cycle through a4, zero through six, we get the value zero, two, four, six, one, three, five. So indeed we cycle through the seven possible outcomes one by one. So if a4 is chosen uniformly and random, then indeed this blue circled quantity will also be uniformly random. So just to give another x4 and y4. Again, we have no idea what it is, other than that its non-zero. So, you know, maybe instead of three, maybe, maybe instead of two, it's three. So now again, let's stop through the seven choices of a4, and see what we get. So now we're going to get zero, then three, then six, then two, then five, and then one, and then four. So again, stepping through the seven choices of a4, we get all of the seven different possibilities of this left hand side. And it's not an accident that these choices are parameters. As long as n is prime, x4 and y4 are different, and y ranges over all possibilities, so will the value on the left-hand side. So by choosing a four uniformly random, indeed, the left-hand side is equally likely to be any of its possible values, zero, one, two up to n-1. And so, what does that mean? Well, basically it means that we're done with our proof cuz remember, the right-hand side that circled in pink is fixed. We fixed a1, a2, and the a3. The x's and y's have been fixed all along so this is just some number, like 773. And so, we know that there's exactly one choice of a4 that will cause the left-hand side to also be equal to 773. Now a4 has n different possible values and it's equally likely to take one on becaus e only a one-end chance that we're going to get the unlucky choice of a4 that causes the left-hand side to be equal to 773 and of course, there's nothing special about 773. Doesn't matter how the right-hand side comes out. We have only one-hand chance of being unlucky and having a collision and that is exactly the condition we are trying to prove and that establishes the universality of this function each of n^4, very simple, very easy to evaluate hash

# 15-3-Universal Hashing - Analysis of Chaining [Advanced - Optional]

So, in this video, we're going to start reasoning about the performance of hash tables. In particular, we'll make precise this idea that properly implemented they guarantee constant time lookup. So, let me just briefly remind you of the road map that we're in the middle of. So, we observed that every fixed hash function is subject to a pathological data set and so exploring the solution of making a real time decision of what hash function to use. And we've already gone over this really quite interesting definition of universal hash functions and that's the proposed definition of a good random hash function. More over, in the previous video I showed you that there are simple such families of hash functions. They don't take much space to store, they don't take much time to evaluate. And the plan for this video is to prove formally, that if you draw a hash function uniformly at random from a universal family of hash functions, like the one we saw in the previous video, then you're guaranteed expected constant time for all of the supported operations. So, here's the definition once more of a universal family of hash functions. We'll be using this definition, of course, when we prove that these hash functions give good performance. So, remember, we're talking now about a set of hash functions. These are all of the conceivable real time decisions you might make about which hash function to use. So, the universe is fixed, this is something like IP addresses, the number of buckets is fixed. You know that's going to be something like 10,000, say, and what it means for a family to be universal is that the probability that any given pair of items collides is as good, as small as with the gold standard of completely perfect uniform random hashing. That is for each pair xy of distinct elements of the universe, so for example, for each distinct pair of IP addresses, the probability over the choice of the random hash function little h from the family script h is no more than one over n, where n is the number of buckets. So, if you have 10,000 buckets, the probability that any given pair of IP addresses winds up getting mapped to the same bucket is almost one in 10,000. Let me now spell out the precise guarantee we're going to prove if you use a randomly chosen hash function from a universal family. So, for this video, we're going to only talk about hash tables implemented with chaining, with one length list per bucket. We'll be able to give a completely precise mathematical analysis with this collision resolution scheme. We're going to analyze the performance of this hash table in expectation over the choice of a hash function little h drawn uniformly from a universal family script h. So, for example, for the family that we constructed in the previous video, this just amounts to choosing each of the four coefficients uniformly at random. That's how you select a universal, that's how you select a hash function uniformly at random. So, this theorem and also the definition of universal hash functions dates back to a 1979 research paper by Carter and Wegman. The idea of hashing dates back quite a bit before that, certainly to the 50s. So, this just kind of shows us sometimes ideas have to be percolating for awhile before you find the right way to explain what's going on. So, Carter and Wegman provided this very clean and modular way of thinking about performance of hashing through this universal hashing definition. And the guarantee is exactly the one that I promised way back when we talked about what operations are supported by hash tables and what kind of performance should you expect, you should expect constant time performance. As always, with hashing, there is some fine print so let me just be precise about what the caveats of this guarantee are. So, first of all, necessarily this guarantee is an expectation. So, it's on average over the choice of the hash function, little h. But I want to reiterate that this guarantee does hold for an arbitrary data set. So, this guarantee is quite reminiscent of the one we had for the rand omized quick sort algorithm. In Quicksort, we made no assumptions about the data. It was a completely arbitrary input array and the guarantee said, on average over the real time randomized decisions of the algorithm, no matter what the input is, the expected running time was in log in. Here we're saying again, no assumptions about the data. It doesn't matter what you're storing in the hash table and expectation over the real time random decision of what hash function you use, you should expect constant time performance, no matter what that data set is. So, the second caveat is something we've talked about before. Remember, the key to having good hash table performance, not only do you need a good hash function which is what this universality key is providing us but you also need to control the load of the hash table. So, of course, to get constant time performance, as we've discussed, a necessary condition is that you have enough buckets to hold more or less the stuff that you're storing. That is the load, alpha, the number of objects in the table divided by the number of buckets should be constant for this theorem to hold. And finally, whenever you do a hash table operation, you have to in particular invoke the hash function on whatever key you're given. So, in order to have constant time performance, it better be the case that it only takes constant time to evaluate your hash function and that's, of course, something we also discussed in the previous video when we emphasized the importance of having simple universal hash functions like those random linear combinations we discussed in the previous video. In general, the mathematical analysis of hash table performance is a quite deep field, and there is some, quite mathematically interesting results that are well outside the scope of this course. But what's cool, in this theorem I will be able to provide you a full and rigorous proof. So, for hash tables with chaining and using randomly chosen universal hash functions, I'm going to now prove that you do get cons tant time performance. Right, so hash tables support various operations, Insert, Delete and Lookup. But really if we can just bound a running time of an unsuccessful lookup, that's going to be enough to bound the running time of all of these operations. So, remember in hash table with chaining, you first hash the appropriate bucket and then you do the appropriate Insert, Delete or Lookup in the link list in that bucket. So, the worst case as far as traversing though this length list is if you're looking for something but it's not there cuz you have to look at every single element in the list and followup into the list before you can conclude that the lookup has failed. Of course, insertion, as we discussed, is always constant time, deletion and successful searches, well, you might get lucky, and stop early before you hit the end of the list. So, all we're going to do is bother to analyze unsuccessful lookups that will carry over to all of the other operations. So, a little more precisely, let's let s be the data set. This is the objects that we are storing in our hash table. And remember that to get constant time lookup, it really needs to be the case that the load is constant. So, we are assuming that the size of s is bigger over the number of buckets n. And let's suppose we are searching for some other object which is not an s, call it x. And again, I want to emphasize we are making no assumptions about what this data set s is other than that the size is comparable to the number of buckets. So, conceptually doing a lookup in a hash table with chaining is a very simple thing. You just hash to the appropriate bucket and then you scan through the length list in that bucket. So, conceptually, it's very easy to write down the what the running time of this unsuccessful lookup is. It's got two parts. So, the first thing you do is you evaluate the hash function to figure out the right bucket. And again, remember we're assuming that we have a simple of a hash function and it takes constant time. Now, of course, the magic of hash functions is that given that this hash value, we can zip right to where the lenght list is to search for x using random access into our array of buckets. So, we go straight to the appropriate place in our array of buckets and we just search through the list ultimately failing to find what we're looking for s. Traversing a link list, as you all know, it takes time proportional to the length of the list. So, we find something that we discussed informally in the past which is that's the running time of hash table operations implemented with chaining is governed by the list lengths. So, that's really the key quantity we have to understand. So, lets call this, lets give this a name, Capital L, it's important enough to give a name. So, what I want you to appreciate at this point is that this key quantity, Capital L, the list of the length in x's bucket is a random variable. It is a function of which hash function little h, we wind up selecting in a real time. So, for some choices of our hash function, little h, this list length might be as small as zero but for other choices of this hash function h, this list, list length could be bigger. So, this is exactly analogous too in Quicksort where depending on the real time decision of which random pivot element you use, your going to get a different number of comparisons, a different running time. So, the list length and hence the time for the unsuccessful storage, depends on the hash function little h, which we're choosing at random. So, let's recall what it is we're trying to prove. We're trying to prove an upper bound on the running time of an unsuccessful lookup on average, where the on average is over the choice of the hash function little h. We've expressed that this lookup time in terms of the average list length in x's bucket where again the average is over the random choice of h. Summarizing, we've reduced what we care about, expected time for lookup to understanding the expected value of this random variable Capital L, the average list length in x's bucket. So, that's what we've got to do, we've got to compute the expected value of this random variable, Capital L. Now to do that, I want to jog your memory of a general technique for analyzing expectations which you haven't seen in awhile. The last time we saw it, I believe, was when we were doing the analysis of randomized Quicksort and counting its comparisons. So, here's a general decomposition principle which we're now going to use in exactly the same way as we did in Quicksort here to analyze the performance of hashing with chaining. So, this is where you want to understand the expect, expectation of random variable which is complicated but what you can express as the sum of much simpler random variables. Ideally, 0,1 or indicator random variables. So, the first step is to figure out the random variable, Capital Y, it's what I'm calling it here that you really care about. Now, we finished the last slide, completing step one. What we really care about is Capital L, the list length in x's bucket. So, that governs the running time a bit unsuccessful Look up, clearly that's what we really care about. Step two of the decomposition principle is well, you know, the random variable you care about is complicated, hard to analyze directly but decompose it as a sum of 0,1 indicator random variable. So, that's what we're going to do in the beginning of the next slide. Why is it useful to decompose a complicated random variable into the sum of 0,1random variables? Well, then you're in the wheelhouse of linear of expectations. You get that the expected value of the random variable that you care about is just the sum of the expected values of the different indicator random variables and those expectations are generally much easier to understand. And that will again be the case here in this theorem about the performance of hash tables with chaning. So, let's apply this three-step-decomposition principle to complete the proof of the Carter-Wegman theorem. So, for the record, let me just remind you about the random variable that we actually really care about, that's Capital L. The reason that's a random variable is that because it depends on the choice of the hash function, little h. This number could vary between zero and something much, much bigger than zero, depending on which each you choose. So, this is complicated, hard to analyze directly, so let's try to express it as the sum of 0,1 random variables. So, here are the0,1 random variables that are going to be the constituents of Capital L. We're going to have one such variable for each object y in the data set. Now, remember this is an unsuccessful search, x is not in the data set Capital S. So, if y is in the data set, x and y are necessarily different. And we will define each random variable z sub y, as follows. We'll define it as one if y collides with x under h and zero otherwise. So, for a given zy, we have fixed objects x and y So, x is some IP address, say, y is some distinct IP address, x is not in our hash table, y is in our hash table. And now, depending on which hash function we wind up using, these two distinct IP addresses may or may not get mapped to the same bucket of our hash table. So, this indicates a random variable just indicating whether or not they collide, whether or not we unluckily choose a hash function little h that sends these distinct IP addresses x and y to exactly the same bucket. Okay, so, that's zy, clearly by definition, it's a 0,1 random variable. Now, here's what's cool about these random variables is that Capital L, the list length that we care about decomposes precisely into the sum of the zy's. That is, we can write capital L as being equal to the sum over the objects in the hash table of zy. So, if you think about it, this equation is always true, no matter what the hash function h is. That is if we choose some hash functions that maps these IP address x to, say, bucket number seventeen, Capital L is just counting how many other objects in the hash table wind up getting mapped to bucket number seventeen. So, maybe ten different ob jects got mapped to bucket number seventeen. Those are exactly the ten different values of y that will have their zy equal to1, right? So, so l is just counting the number of objects in the data set s that's collide with x. A given zy is just counting whether or not a given object y in hash table is colliding with x. So, summing over all the possible things that could collide with x, summing over the zy's, we of course get the total number of things that collide with x which is exactly equal to the number, the population of x's bucket in the hash table. Alright, so we've got all of our ducks lined up in a row. Now, if we just remember all of the things we have going for us, we can just follow our nose and nail the proof of this theorem. So, what is it that we have going for us? Well, in addition to this decomposition of the list length in to indicate random variables, we've got linear expectation going for us, we've got the fact that our hash function is drawn from a universal family going for us. And we've got the fact that we've chosen the number of buckets and to be comparable to the data set size. So, we want to use all of those assumptions and finish the proof that the expected performance is constant. So, we're going to have a few inequalities and we're going to begin with the thing that we really care about. We care about the average list length in x's bucket. Remember, we saw in the previous slide, this is what governs the expected performance of the lookup. If we can prove that the expected value of capital L is constant, we're done, we've finished the theorem. So, the whole point of this decomposition principle is to apply linear of expectation which says that the expected value of a sum of random variables equals the sum of the expected values. So, because L can be expressed as the sum of these zy's, we can reverse the summation and the expectation and we can first sum over the y's, over the objects in the hash table and then take the expected value of zy. Now, something which came up in our Quicksort an alysis but which you might have forgotten is that 0,1 random variables have particularly simple expectations. So, the next quiz is just going to jog your memory about why 0,1 random variables are so nice in this context. Okay, so the answer to this quiz is the third one, the expected value of zy is simply the probability that x and y collide, that just follows from the definition of the random variable zy and the definition of expectation, namely recall how do we define zy. This is just this one, if this object y in the hash table happens to collide with the object x that we are looking for under the hash function x and zero otherwise, again, this will be, this will be one for some hash functions and zero for other hash functions. And then we just have to compute expectations. So, one way to compute the expected value of a 0,1 random variable is, you just say, well, you know, there are the cases where the random variable evaluates to zero and then there's the cases where the random variable evaluates to one, and of course we can cancel the zero. So, this just equals the probability that zy = one. And since zy being one is exactly the same thing as x and y colliding, that's what gives us the answer. Okay, so it's the probability that x collides with y. So, plenty of that into our derivation. Now, we again have the sum of all the objects y in our hash table and the set of the expected value of zy what's right that in the more interpretable form, the probability that this particular object in the hash table y collides with the thing we are looking for x. Now, we know something pretty cool about the probability that a given pair of distinct elements like x and y collide with each other. What is it that we know? Okay, so I hope you answered the second response to this quiz. This is really in some sense the key point of the analysis. This is the role, that being a universal family of hash functions plays in this performance guarantee. What does it mean to be universal? It means for any pair of objects distinct like x and y in that proof, if you make a random choice of a hash function, the probability of collision is as good as with perfectly random hashing, hashing. Namely at most, 1/ n where n is the number of buckets. So, now I can return to the derivation. What that quiz reminds you is that the definition of a universal family of hash functions guarantees that this probability for each y is at most 1/n, where n is the number of buckets in the hash table. So, let's just rewrite that. So, we've upper bounded the expected list length by a sum over the objects in the data set of 1/n. And this, of course, is just equal to the number of objects in the data set, the [inaudible] of s divided by n. And what is this? This is simply the load, this is the definition of the load alpha which we are assuming is constant. Remember, that was the third caveat in the theorem. So, that's why as long as you have a hash function which you can compute quickly in constant time. And as long as you keep the load under control so the number of buckets is commensurate with the size of the data set that you're storing. That's why, universal hash functions in a hash table with chaining guarantee expected constant time performance.

# 15-4-Hash Table Performance with Open Addressing [Advanced - Optional]

In the last video, we discussed the performance of hash tables that are implemented using chaining, using one link list per bucket. In fact, we proved mathematically that if you use a hash function chosen uniformly at random from a universal family, and if you keep the buckets, number of buckets, comparable to the size of your data set, then in fact, you're guaranteed constant time expected performance But, recall that chaining is not the only implementation of hash tables. There's a second paradigm which is also very important called open addressing. This is where you're only allowed to store one object in each slot, and you keep searching for an empty slot When you need to insert a new object into your hash table. Now it turns out it's much harder to mathematically analyze hash tables implemented using open addressing But, I do want to say a few words about it to give you the gist of what kind of performance you might hope to expect from those sorts of hash tables. So recall how open addressing works. We're only permitted to store one object in each slot So this is unlike the case with chaining where we can have an arbitrarily long list in a given bucket of the hash table. With at most one object per slot, obviously open addressing only makes sense when the load factor alpha is less than one. When the number of objects you're storing in your table is less than the number of slots available Because of this requirement we have at most one object per slot we need to demand more of our hash function. Our hash function might ask us to put a given object, say with some IP address into say bucket number seventeen but bucket number seventeen might already be full, might already be populated. In that case, we go back to our hash function and ask it where to look for an empty slot next. So maybe it tells us to next look in bucket 41. If 41 is full it tells us to look in bucket number seven and so on Two specific strategies for producing a probe sequence that we mentioned earlier were double hashing and linear probing. D ouble hashing is where you use two different hash functions, h1 and h2. H1 tells you which slot in which to search first and then every time you find a full slot you add an increment which is specified by the second hash functions h2. Linear probing is even simpler you just have one hash function that tells you where to search first and then you just add one to the slot until you find an empty slot As I mentioned at the beginning, it is quite nontrivial to Mathematically analyze the performance of hash tables, using these various open addressing strategies. It's not impossible. There is some quite beautiful and quite informative theoretical work. That does tell us how hash tables perform But that's well outside the scope, of this course. So instead what I wanna do is I want to give you a quick and dirty calculation. That suggests, at least in an idealized world. What kind of performance we should expect from a hash table with open addressing If it's well implemented As a function of the load factor, alpha. Precisely, I'm going to introduce a heuristic assumption. It's certainly not true but we'll do it just for a quick and dirty calculation, that we're using a hash function in which each of the n-factorial possible probe sequences is equally likely. Now, no hash function you're ever going to use is actually going to satisfy this assumption, and if you think about it for a little bit, you realize that if you use double hashing or linear programming, you're certainly not going to be satisfying that assumption. So this will still give us a kind of best case scenario against to which you can compare the performance of your own hash table implementations. So if you [inaudible] hash table, and you're seeing performance as good, as what's suggested by this idealized [inaudible] analysis, then you're home free. You know your hash table is performing great. So what is the line in the sand that gets drawn, under this heuristic assumption? What is this idealized, idealized hash function performance? As a function of the lo ad alpha Well here it is. What I'm gonna argue next is that under this heuristic assumption, the expected amount of time to insert a new object into the hash table, is going to be essentially one over one minus alpha, where alpha is the load. Remember the load is the number of objects in the hash table divided by the number of available slots. So if the hash table is half full, then alpha's going to be.5. If it's 75 percent full then alpha's going to be three-fourths. So what this means is that, in this idealized scenario, if you keep the load pretty under control. So, say if the load is 50%, then the insertion time is gonna be great, right? If alpha's.5 And 1/ (1-alpha) =two, so you expect just two probes before the successful insert of the new object And of course, if you're thinking about lookup, that's going to be at least as good as insert, so if you're lucky a lookup might terminate early if you find what you are looking for. In the worst case you go all the way until an empty slot in an unsuccessful search, and that's gonna be the same as insertion. So if alpha is small bounded away from one, you're getting constant time performance. On the other hand, as the hash table gets full, as alpha gets close to one, this operation time is blowing up; it's such a going to infinity as alpha gets close to one. So if you need to have a nice. 90 percent full hash table with open addressing. You're gonna start seeing, ten probes. So, you really wanna keep hash tables with open addressing. You wanna keep the load under control Certainly no more than probably.7. Maybe even less than that To refresh your memory, with chaining, hash tables are perfectly well-defined even with loads factors bigger than one. What we derived is that under universal hashing, under a weaker assumption, we had an operation time of one plus alpha, for a load of alpha. So with chaining, you just gotta keep alpha, you know, at most, some reasonably small constant with open address, and you really got to keep it well bounded a way below one. So next let's understand why this observation is true. Why under the assumption that every probe sequence is equally likely do we expect a one over one minus alpha running time for hash tables with open addressing? So, the reason is pretty simple. And we can derive it by analogy with a simple coin flipping experiment. So, to motivate the experiment, think just about the very first probe that we do. Okay, so we get some new objects, some new IP address that we want to insert into our hash table. Let's say our hash table's currently 70 percent full. Say there's 100 slots, 70 are already taken by objects. Well, when we look at this first probe, by assumption it's equally likely to be any one of the 100 slots. 70 of which are full, 30 which are empty So, with probability of one minus alpha, or in the case, 30%, our first Probe will, luckily, find an empty slot and we'll be done. We'll just insert the new object into that slot If we get unlucky with a probability, 70%. We find a slot that's already occupied and then we have to try again. So we try a new slot, drawn at random And we again check is it full, or is it not full? And again, with 30 percent probability, essentially it's going to be empty and we can stop And if it's already full. Then we try, yet again. So Doing random probes, looking for an empty slot, is tantamount to flipping a coin with the probability of heads 1-alpha, or, in this example, 30 Percent And the number of probes you need until you successfully insert is just the number of times you flip this last coin until you see a heads. In fact this biased coin flipping experiment slightly overestimates the expected time for insertions and the heuristics assumptions and that's because in the insertion time whenever we're never going to try the same slot twice. We're going to try all end buckets in some order with each of the impact [inaudible] ordering equally likely So back to our example, where we have a hash table with 100 slots, 70 of which are full. The first probe indeed, we have a 30 in 100 chance of ge tting an empty slot. If that one fails then we're not going to try the same slot again. So there is only 99 residual possibilities. Again, 30 of which are empty. The one we checked last time was full. So we actually have a 30 over 99 percent chance of getting an empty slot on the second try. Like 30 over 98 on the third try, if the second one fails, and so on But, a valid upper bond is just to assume a 30 percent success probability with every single probe, and that's precisely, what this coin flipping experiment will get us. So the next quiz will ask you to actually compute the expected value of capital N, the number of coin flips, needed to get heads when you have a probability of heads of one minus alpha. As a hint, we actually analyzed this exact same coin flipping experiment when alpha equals a half, back when we discussed the expected running time of randomized linear time selection. Alright, so the correct answer is the first one. One over 1-alpha So to see why, let's return to our derivation, where we reduced analyzing the expected insertion time to this random variable. The expected number of coin flips until we see a heads. So, I'm gonna solve this exactly the same way that we did it back when we analyzed a randomized, selection algorithm. And it's quite a sneaky way, but very effective. What we're going to do is we're going to express the expected value of capital N, in terms of itself, and then solve. So how do we do that? Well on the left hand side let's write the expected number of coin flips, the expected value of capital N, and then let's just notice that there's two different cases, either the first coin flip is a heads or it's not. So in any case you're certainly going to have one coin flip so let's separate that out and count it separately. With probability alpha, the first coin flip is gonna be tails and then you start all over again And because it's a memory less process, the expected number of further coin flips one requires, given that the first coin flip was tails, is just the same as the expected number of coin flips in the first place. So now it's a simple matter to solve this one linear equation for the expected value of N, and we find that it is indeed one over one minus alpha, as claimed. Summarizing, under our idealized heuristic assumption, that every single probe sequence is equally likely, the expected insertion time is upper bounded by the expected number of coin flips, which by this argument is, at most, one over one minus alpha. So, as long as your load, alpha, is well bounded below one, you're good. At least in this idealized analysis, you're hash table will, will work extremely quickly. Now I hope you're regarding this idealized analysis with a bit of skepticism. Right, from a false hypothesis you can literally derive anything you want. And we started with this assumption which is not satisfied, by hash functions you're actually going to use in practice. This heuristic assumption, that all probe sequences are equally likely. So, should you expect this one over one minus alpha bound to hold in practice or not? Well, that depends to some extent. It depends on what open addressing strategy you're using. It depends on, how good a hash function you're using. It depends on whether the data is pathological or not. So, just to give course rules of thumb If you're Using double hashing and you have non-pathological data, I would go ahead and look for this 1/1-alpha bound in practice. So implement your hash table, check its performance as a function of the load factor alpha and shoot for the 1/1-alpha curve. That's really what you'd like to see. With linear probing, on the other hand, you should not expect to see this performance guarantee of 1/1-alpha even in a totally idealized scenario. Remember, linear probing is the strategy where your initial probe, the hash function, tells you where to look first, and then you just skim linearly through the hash table until you find what you're looking for, an empty slot, the. That you're looking up or whatever So a linear probing, even in a best case scenario, it's going to be subject to clumping. You're going to have contiguous Groups of slots which are all full, and that's because of the linear probing strategy. Now I encourage you to do some experiments with implementations to see this for yourself. So because of clumping with linear probing, even in the idealized scenario, you're not going to see one over one minus alpha. However, you're going to see something worse, but still in idealized situations. Quite reasonable so that's the last thing I want to tell you about In this video. Now needless to say, with linear probing the heuristic assumption is badly false. The heuristic assumption is pretty much always false to no matter what hashing strategy you're using, but with linear programming it's quote on quote really false. So to see that, the heuristic assumption, say that all in factorial probe sequences are equally likely. So your next probe is going to be uniform or random amongst everything you haven't probed so far but when you're probing, it's totally the opposite. Right once you know the first slot that you're looking into say bucket seventeen, slot a7 is gonna be the first slot, you know the rest of the sequence because it's a linear [inaudible] cancel the table. So it's kind of [inaudible] the opposite from each successive probe being independent from the previous ones except not exploring things twice. So to state a conjectured or idealized performance guarantee for hash tables with linear probing, we're going to place, replace the blatant false heuristic assumption by a still false, but more heuristic reasonable assumption. So what do we know? We know that the initial probe with linear probing determines the rest of the sequence. So let's assume that these initial probes are uniform at random, and independent for different keys. Of course, once you have the initial probe, you know everything else, but let's assume independence and uniformity amongst The initial probes. Now, this is a strong assumption. This is way stronger than assuming you ha ve a universal family of hash functions. This assumption is not satisfied Practice, but Performance guarantees we can derive under this assumption are typically satisfied in practice by well implemented hash tables that use linear probing. So, the assumption is still useful for deriving the correct, idealized performance of this type of hash table. So what is that performance? Well this is an utterly classic result from exactly 50 years ago From 1962 And this is a result by my colleague, the living legend, Don Canuth, author of Art of Computer Programming. At what can proved is, was that is that under this weaker [inaudible] assumptions, suitable for linear probing. The expected time to insert an object into a hash table with a load factor alpha, when you're using linear probing is worse than one over one minus alpha, but. It is still a function of the load alpha only and not a function of the number of objects in the hash table. That is with linear programming you will not get as good a performance guarantee, but it is still the case that if you keep the load factor bounded away from one. If you make sure the hash table doesn't get too full you will enjoy constant time operations on average so for example if with linear probing your hash table is 50 percent full then you are going to get an expected insertion time of roughly four probes. Note however this quantity does approach does blow up pretty rapidly as the hash table grows full. If it is 90 percent full this is already going to be something like a hundred probes on average. So you really don't wanna let hash tables get too full when you are using linear probing. You might well wonder if it's ever worth, implementing linear probing, given that it has the worst performance curve, one over one minus alpha squared. Then the performance curve you'd hope from something like double hashing, one over one, minus alpha. And it's a tricky cost benefit analysis between linear probing and more complicated but better performing strategies. That really depends on the ap plication. There are reasons that you do want to use linear probing sometimes, it is actually quite Common in practice For example, it's often interacts very well with memory hierarchies So again, as with all of this hash and discussion. You know the costs and benefits Are, are very subtle trade-offs between the different approaches. If you have mission critical code that's using a hash table and you really want to optimize it. Try a bunch of prototypes, and just test. Figure out which one is the best, for your particular type of application. Let me conclude the video with a quote from Canuck himself where he talks about the rapture of proving this one of our one man is half a square theorem and how it was life changing. He says I first formulated the following derivation, meaning, the proof of that last theorem in 1962. Ever since that day, the analysis of algorithms has, in fact, been one of the major themes in my life.

# 16-1-Bloom Filters - The Basics

So, in this video, we're going to discuss Bloom filters which is a data structure developed appropriately enough by Burton Bloom back in 1970. Bloom filters are variant on hash tables, you'll recognize a lot of the ideas from our hash table discussion. The win that you get in Bloom filters is that they are more space efficient than run of the mill hash tables and they're going to handle, they do allow for errors, there is a non zero false positive probability when you do look ups but that's still a win for some applications. So, it's a very cool idea, very cool data structure. You do see it used quite a bit in practice so let's start talking about it. So, we'll go through the usual topics that we do whenever we discuss a new data structure. So first, I want to tell you what operations they support and what kind of performance you're going to expect from those operations, in other words, what is the API corresponding to the data structure. Secondly, I'm going to talk a little bit about what it's good for. So, what are some potential application and then we'll take a peek under the hood. I'll tell you some of the implementation details with an emphasis on explaining why you get the kinds of performance trade offs that you do with Bloom filters. So, to first order, the raison d'锚tre of Bloom filters is exactly the same as a hash table. It supports super fast inserts, super fast look ups. You can put stuff in there and you can remember what you put in earlier. Now, of course, what you should be wondering is what we already know what data structure that supports super fast in certain look ups, a hash table. Why am I bothering to tell you about yet another data structure with exactly those same operations? So, let me tell you about the pros and cons of Bloom filters relative to run off the mill hash tables as we've already discussed. The big win is that Bloom filters are more space efficient than hash tables. No matter whether they are implemented with chaining or with open addressing, you can store much less space per objects. In fact, as we'll see, less space than that of an object itself using a Bloom filter. As far as the cons, well, first of all, this is really for applications where you just want to remember what kind of values you see. You are not trying to store pointers to the objects themselves and just trying to remember values. So, the first drawback of the Bloom filter is that because we want to be so space efficient, we don't even want to remember the object itself just whether or not we've seen it before. We're not going to be able to store the objects or even pointers to the objects in a Bloom filter. We're just going to remember what we've seen and what we haven't. So, some of you might know the terminology hash set for this kind of variant of a hash table as opposed to a full blown hash table or hash map. The second con is at least in the vanilla implementation of Bloom filters that I'm going to describe here, deletions are not allowed. You can only insert, you can't delete. The situation with deletions is very much similar to hash tables implemented with open addressing. It's not that you can't have a Bloom filter that accommodates deletion, you can, there are very instances of it but that requires significantly more work and we're not going to discuss it here. So, the first order at least for vanilla Bloom filters, you want to think of them as suitable for applications or deletions or not a first order of operation. Now, the third con and this is a drawback that we have not see previously using any data structures is Bloom filters can actually make mistakes. Now, what kind of mistake could this kind of data structure possibly make when all you're really doing is looking something up. Well, one of mistake would be a false negative and that means you have inserted something previously then you look it up and the hash table or the Bloom filter says, it's not there. So, Bloom filters will not have false negatives of this form. You've insert something, you look it up later, it's definitely going to confirm that you inserted it in the past. But Bloom filters will have false positives, that means that despite the fact you have never inserted say, a given IP address into the, into the Bloom filter, if you look it up later, it will say that you have. So, there will sometimes be in some sense phantom objects in Bloom filters, objects which it thinks have been inserted even though they haven't been. So, given that, I am now showing you two data structures with essentially the same functionality, hash tables and Bloom filters, at least, if we ignore the deletion issue. You might want to wonder which one is more appropriate, which one is more useful. And because there is these trade offs between the two, the answer as you expect is, it depends on the application, right? So, if it's an application where space is really at a premium, you might want to turn to Bloom filters especially if a small chance of a false positive is not deal breaker. If you have some kind of application where false positives are absolutely out of the question, of course, you should not use a Bloom filter and you want to think about a hash table. So, what are some situations where people actually do use Bloom filters where you either really care about space and/or you don't really care about this false positive probability. For one of the earliest applications of Bloom filters, this is not long time ago, this is something like 40 years ago, was the spell checkers. So, how would you implement a spell checker using a Bloom filter? Well, first you have this insert phase where you basically just go through the entire dictionary word-by-word and you insert every valid word into the Bloom filter. Then, afterwards, when you're presented with a new document that somebody has written, you're going to go through the document word-by-word for each word, you say, is this in the Bloom filter? That is, is this one of the legitimate word from the dictionary which is previously inserted? If the Bloom filters says yes, this word is in the dictionary as in we've stored and seen that before, then you treat is as a correctly spelled word and if it's not in the Bloom filters, then you treat it as incorrectly spelled word. Now, the false positive probability means this isn't a perfect spell checker. I mean sometimes, you're going to look up a misspelled word and the Bloom filter won't catch it and it willl actually say yes, with small probability, we'll say, this is a legitimate word. So, you know, it's not ideal but, you know, the, the English language is pretty big and space was definitely at a premium, 40 plus years ago. So, it was a win for that application at that time, to use Bloom filters to implement a spell checker. Another application which, you know, remains relevant today is to keep track of a list of forbidden passwords. Now, why would you have forbidden passwords? Well, maybe, you want to keep track of password which are too weak or too easy to guess or too common. You may, yourself, have used the piece of software or website at some point where it asked you for a password and if you typed in something which is too simple or too easy, rejected it and asked you to type in another one. So, one way to implement a list of forbidden passwords is just with the Bloom filter and the idea is similar to the spell checker. You first, insert into the Bloom filter all of the passwords that you don't want anybody to use for whatever reason. Then, when a client comes and tries to type in a new password, you look it up in the Bloom filter and if you get a positive look up, then you tell the user, no, that's no good, you can't use that password, choose another one. And this is an application where you really don't care about the errors, you really don't care about the fact that there's a false positive rate. Let's assume that the error rate is something like one percent or 0.1%. So, what would that means in context, that would just mean once in a while, one in a hundred clients or one in a thousand clients actually types in a perfectly strong password that gets rejected by the Bloom filter and they have to type in a second one. Okay, but big deal and if space is at the, the premium, this is definitely a win to use this super lightweight data structure to keep track of these blocked passwords. These days certainly one the killer applications of Bloom filters is in software deployed on network routers. So, the machinery out in the Internet which is responsible for transmitting packets from one place to another. So, what are the reasons why Bloom filters have found fertile application in network routers? Well, first of all, you do have a budget on space, typically on network routers. There's a lot of things that you got to do and you don't want to waste that much of it on some random data structure to do one's specific task. So, you do have a budget on space and also, you need super, super fast data structures, right? Since these packets are coming in at this torrential rate which you can't even imagine and you want to process these packets in real time, sending them off to the next top. Bloom filters are the work force behind a lot of different tasks that is done in the network router. You can imagine wanting to keep track of blocked IP addresses, you can imagine keeping track of the contents of some cache so you don't do spurious look ups. You can imagine maintaining statistics to check for denial of service attacks and so on and so forth. So, summarizing as a expert programmer, what is it that you should remember about Bloom filters, what purpose does this tool serve in your tool box? Well, as far as the operation supported which is the same as a hash table, the point is to have super fast inserts, super fast look ups. But Bloom filters are more lightweight version of a hash table. So, they are more space efficient but they do have this drawback of having a small error probability. So, those are the key features you should remember when deciding whether or not you are working on an application that could make good use of this data structure. So, having discussed one of th e operations and what these data structures are good for, let's take it to the next level, let's peer under the hood and see how they are actually implemented. Cuz this is really a quite simple, quite cool idea. So, like hash tables, Bloom filters have essentially two ingredients. First of all, there's an array and second of all, there's a hash function or in fact, several hash functions. So, we're going to have a random access array except, instead of having n buckets or n slots as we've been calling them, each entry in this array is just going to be a single bit. Each entry in this array can only take on two values, zero or one. And the way they think about the space occupied by Bloom filters is in terms of the number of bits per object that has been inserted into the Bloom filter. So, if you have inserted the data set capital S, then the total number of bits is n, the number of objects that have been inserted is cardinality of s. So, n / |s| is the number of bits in this data structure that you are using per entry in the data set. Now, you can tune a Bloom filter so this ratio is any number of different quantities but for now, I encourage you to think of this ratio as being eight, that is for each object stored in the Bloom Filter, you are using only eight bits of memory. That will help you appreciate just how amazing this data structures are, right, cuz maybe our data set is something like IP addresses which is 32 bits so what I'm saying here, if this is eight, I'm saying we are not, definitely not actually storing the IP address. So, we have this 32-bit object we are inserting and we are only using eight bits of memory. This is how we are going to remember whether its there or whether its not. And again, certainly, eight bits per object is way less than keeping a pointer to some associated memory somewhere. So, this is a really impressive minimal use of space to keep track of what we've seen and what we haven't. And secondly, we need mappings of given an object to say, given the IP address, what are the relevant bits for seeing if we've seen this IP address before or not? So, in a Bloom filter, its important to have not one hash function, but several hash functions. So, k is going to denote the number of hash functions in the Bloom filter which you think of k is some small constant somewhere, you know, three, four, five, or something like that. So, obviously it's a little bit more complicated to use multiple hash functions as supposed to just one hash function. But it's really not that big of deal. So, we'll call from our discussion of say, universal hashing, we have identified the entire families of hash functions which will work well on average. So, instead of choosing just using one hash function at random from universal family, you gave me k independent random choices from universal family. In fact, in practice, it seems to typically be enough to just use two different hash functions and then generate k different linear combinations of those two hash functions. But for the purposes of this video, let's just assume that we've done enough work to come up with k, different good hash functions and that's what we're going to be using in our Bloom filter. So, the code for both insert and delete is very elegant. So, let's start by insertion. So, suppose we have some new IP address and we want to stick into these Bloom filter, what we do? Well, we'll just evaluate each of our k hash functions on this new object. Each of those tells us an index into our array of bits and we'll just set those k bits equal to one. And when we do this insert, we don't even bother to look at what the previous values of these bits were.. So, zero or one, we don't care. We'll just blithely go in and set this k bits equal to one, whatever they were before. So, what about looking up? How are we going to implement that? Well, all you have to do is check for the footprint that was inevitably left by a prior insertion. So, if we're looking up an IP address and we know was inserted sometime in the past, what happened when we evaluated the k hash functions, we went t o appropriate positions in the array and we set all of those bits to one. So now, I'll just check that, that indeed happened, that is when we get a new IP address, we're looking it up. We evaluate the hash functions, all k of them. We look at the corresponding k positions and we verified that indeed those k bits have been set to one. So, what I hope is clear fairly quickly from inspecting this very elegant code is that we will not ever have false negatives, yet, we might have false positives. So, let's discuss those one other time. So, remember, a false negative would mean that the Bloom filter says, something isn't there when in fact, it is, that is we insert something and we'll look it up later and the Bloom filter rejects us. Well, that's not going to happen. Cuz when we insert something, we set the relevant k bits to one. Notice when a bit is one, it remains one forevermore. That bits are never reset back to zero. So, if anything was ever inserted in the subs when we look it up, definitely we well confirm that all those bits are one. So, we're never going to be rejected by something we inserted before. On the other hand, it is totally possible that we will have a false positive. It's totally possible that there will be a phantom object and we'll do a look up and the Bloom filter will turn yes when we never inserted that object. Suppose for example, the k = three. So, we're using three different hash functions. Consider some IP address, fixed IP address, maybe the three hash functions tell us the relevant bits are seventeen, 23, and 36. Maybe we never inserted this IP address, but we have inserted IP address number two and in its insertion, the seventeenth bit got set to one. We inserted some other IP address, IP address number three and the twenty-third bit got set to one. And then we inserted IP address number four and the 36th bit got set to one. So, three different IP addresses were responsible for setting these three different bits but whatever, its not like we are remembering that. And that once, once we look up this IP address we really care about, what do we do, we just inspect bit seventeen, its one. Inspect the 23, its one. We inspect the 36, its also one. For all we know, this thing really was inserted and the Bloom filter is going to say, yes, it's in the table. So, that's how we have false positives. All of the bits that are indicating whether or not a given object are in, are in the Bloom filter were previously set by insertions from other objects. So, there are two points that I hope are clear at this stage of the discussion. First of all, that this Bloom filter, the idea does suggests a possibility of a super space efficient variant of a hash table, right. So, we've been talking about setting the number of bits to be say roughly eight times the number of objects that you're storing so you're only using eight bits per object and for most objects, that's going to be radically smaller than just the simple array, storing the objects themselves. Again, if their IP addresses we're only have 25 percent much space as we actually stored those IP address in just an array with no extra bells and whistles. The second point is that we're inevitably going to have errors in a Bloom filter, we will have false positives or we look something up and it says, its there when in fact, its not. So, those two points I hope are clear. What's actually not clear is the bottom line. Is this actually a useful idea? For this to be useful, it'd better be the case that the error probability can be quite small even while the space per object is quite small. If we can't get those two things small simultaneously, this is a bad idea and we should always just use a hash table instead. So, to evaluate the quality of this idea, we're going to have to do little bit of mathematical analysis. That's what I'm going to show you in the next couple of slides.

# 16-2-Bloom Filters - Heuristic Analysis

So, before we embark in the analysis, what are we hoping to understand? Well, it seems intuitively clear is that there is going to be some trade off between the two resources of the bloom filter. One resource is space consumption, the other resource is essentially correctness so the more space we use, the larger number of bits, we'd hope that we'd make fewer and fewer errors. And then as we compress the table more and more, we use bits more and more for different objects then presumably the error rate is going to increase. So, the goal of the analysis that we're about to do is to understand this trade off precisely at qualitative level. Once we understand the trade off occur between these two resources, then we can ask is there is a sweet spot which gives us a useful data structure? Quite small space and quite manageable error probability. So the way we're going to proceed with the analysis, we'll be familiar to those of you who watched the open addressing video about hash tables so to make the mathematical analysis tractable, I'm going to make a heuristic assumption The strong assumption which is not really satisfied by hash functions you would use in the practice. We're going to use that assumption to derive a performance guarantee for bloom filters but as all as any implementation you should check that your implementation actually is getting performance comparable to what the idealizing analysis suggest. That said, if you use a good hash function and if you have a non-pathological data, the hopes and this is going out many empirical studies is that you will see performance comparable to what this heuristic analysis will suggest. So, what is the heuristic assumption? Well, it's going to be again familiar from my hashing discussions. We're just going to assume that all the hashing is totally random. So, for each choice of a hash function hi and for each possible object ax, the slots, the position of the array which the hash functions gives for that object is uniformly random and first of all and it's independen t from all other outputs of all hash functions on all objects. So the set up then is we have n bits. We have a data set, S which we have inserted into our bloom filter. Now our eventual goal is to understand the error rate or the false positive probability. That is the chance that an object which we haven't inserted into the bloom filter looks as if it has been inserted into the bloom filter but as a preliminary step, I want to ask about the population of 1s after we've inserted this data set S into the bloom filter. So, specifically let's focus on a particular position of the array and by symmetry it doesn't matter which one. And let's ask what is the probability that a given bit, a given position on this array has been set to one after we've inserted this entire data set S? Alright, so this, this is a somewhat difficult quiz question actually. The correct answer is the second answer. It's one - quantity one - 1/n raised to the number of hash functions k the number of objects cardinality of S, that's the probability let's say the first bit of the bloom filter has been set to one after the data set S has been inserted. So the, maybe the easiest way to see this is to first focus on the first answer. So, the first answer is going to be the probability I claim that the first bit is zero after the entire data set has been inserted. Then of course it's probably it's a one, is just the one - its quantity which is equal to the second answer. So we just seem to understand why the first choice is probably the first bit = zero. Well, it's initially zero, remember stuff is only set from zero to one. So we really need to analyze the probability that this first bit survives all of these darts that are getting thrown to the bloom filter over the course of this entire data set being inserted. So there, the cardinality of these objects each get inserted on an insertion k darts uniformly at random and independent from each other or effectively thrown at the array at the bloom filter. Any position of the dart hits, gets set to one. Maybe it was one already but if it was zero, it gets set to one. If it's one then it stays one. So, how is this first pick going to stay zero? We'll have to be missed by all of the darts. A given dart, a given bit flick is uniformly likely to be any of the n bits so the probability of the ones that being this bit is only 1/n but, if it even it's fortunately somebody else? Well, that's one - 1/n so you have a chance of surviving a single dart with probably one - 1/n There is the number of hash functions k the number of objects cardinality that's a dart being thrown. Right k per object that gets inserted so the overall probability of eluding all of the darts is one - one or n raised to the number of hash functions k the number of insertions cardinality of S. Again, the probability that is one which is the one - that quantity which is the second option in the quiz. So, let's go ahead and resume our analysis using the answer to that quiz. So, what do we discover, discover the probability that a given bit is one, is one - quantity one - 1/n or n is the number of position raised to the number of hash functions k the number of insertions cardinality of S. So, that's the kind of messy quantity so let's recall a simple estimation facts that we used once earlier. You saw this when we analyzed cardinals construction algorithm and the benefit of multiple repetitions or cardinals contraction algorithm. And the trick here is to estimate a quantity that's on the form of one + x or one - x by either the x or the - x as the case maybe. So you take the function one + x which goes through the points -ten and 01. And of course it's a straight line and then you also look at the function e to the x. Well, those two functions are going to kiss at the point 0,1 and everywhere else e to the x is going to be above one + x. So for any real value of x we can always upper bound the quantity one + x by either the - x. So let's apply this fact to this quantity here, one - 1/n raise to the k cardinality of S. We're going to take x to the - 1/n so that gives us an upper bound on this probability of one - e to th e - k the number of insertions over n, okay? So that's taking x to the - 1/n. Let's simplify and finalize a little bit further by introducing some notation. So, I'm going to let b denote the number of bits that were using per object. So this is the quantity I was telling you to think about as eighth previously. This is the ratio n, the total number of bits divided by the cardinality of S. So, this green expression becomes one - e^k where b is the number of bits per object. And now we're already seeing this type of trade off that we're expecting. Remember we're expecting that as we use more and more space, then the error rate we think should go down so if you can press the table a lot or use bits for lots of different objects that's when you start going to see a lot of false positives so in this light blue expression if you take the number of bits per objects with the number space, the amount of space, little b if you take that going very large expanding to infinity, this exponent to zero. So either the -zero is one. So overall, this probability of a given bit being one is turning to zero. So, that is, the more bits you have, the bigger space you have. The, well, the smaller of the fraction of 1s. The bigger the faction of 0s. That should translate to a smaller false positive probability unless we will make precise on the next and final slot. So let's, let's rewrite the upshot form the last slide but probability that a given bit is equal to one is that at above by one - e to the - k over b where k is the number of hash functions and b is the number of bits we're using per object. Now this is not the quantity that you care about. The quantity we care about is a false positive probability where something looks like it's in the bloom filter even though it's never been inserted so it's focused on some object like some IP address which is never ever been inserted into this bloom filter. So for a given object x which is not in the data set, that this has not been inserted into the bloom filter or what has to happen for us to have a success ful look up for false positive for this object? Well each one of its k bits has to be set to one. So, we already computed the probability that a given bit is set to one. So, what has to happen for all k of the bits that indicates x's membership in the bloom filter all k of them has to be set to one. So we just take the quantity we computed on the previous slide and we raise that to the kth power. Indicating that it has to happen k different times. So believe it or not we now have exactly what we wanted. What we set out to do which is derive a qualitative understanding of the intuitive trade off between the one hand space used and on the other hand on the error probability. The false positive of probability. So, we're going to call this green circle quantity and name it. We'll call it epsilon for the error rate and again all errors are false positives. And again as b goes to infinity, as we use more and more space, this exponent goes to zero so one - e to that quantity is going to zero as well. And of course, once we power it through the kth power, it gets even closer to zero. So if the bigger b gets the small of this error rate epsilon gets. So now let's get to the punch line. So remember the question is, is this data structure actually useful? Can we actually set all of the parameters in a way that we could both really usefully small space but a tolerable error epsilon? And, of course we wouldn't be giving this video if the answer wasn't yes. Now one thing I've been alluding all along is how do we set k? How do we choose the number of hash functions? I told you at the very beginning We think of k as a small constant like 2345. And now that we have this really nice qualitative version of how the error rate in the space trade off with each other. We can answer how to set k. Namely set k optimally so what do I mean? Well, fix the number of bits that you're using per object. Eight, sixteen, 24, whatever. For fixed b, you can just choose the k that minimize the screen quantity. That minimizes the error rate epsilon. So, how do you minimize t his quantity? Well, you do it just like you learn in calculus and I'll leave this as an exercise for you to do in the privacy of your own home. But for fixed b, the way to get this green quantity epsilon as small as possible is to set the number of hash functions k to be roughly the natural log of two. That's a number of < one notice that's like .693 b. So, in other words the number of hash functions for the optimal implementation of the bloom filter is scaling linearly than the number of bits that you're using per object. It's about .693 the bits per object. Of course this is generally not going to be an integer so you just pick k either this number rounded up or this number rounded down. But, continuing the heuristic analysis, now that we know how to set k optimally to minimize the error for a given amount of space we can plug that value of k back in and see well, how does the space and the error rate trade off against each other and we get a very nice answer. Specifically, we get that the error rate epsilon is just under an optimal trades to the number of hash functions k decreases exponentially in the number of bits that you use per object. So, it's roughly one half raised to the natural log of two or .693 roughly the number of bits per object b. But, again the key qualitative point here is notice that epsilon is going down really quickly as you scale b. If you double the number of bits that you're allocating per object, you're squaring the error rate and for small error rates, squaring it makes it much, much, much smaller. And of course this is just one equation in two variables. If you prefer, you can solve this equation to express b, the space requirement as a function of an error requirement. So if you know that the tolerance for false positives in your application is one percent you can just solve this for b and figure out how many bits per object you need to allocate. And so rewriting what you get is that the number of bits per object that you need is roughly 1.44 the log base two of one over epsilon. So, as expected as epsilon gets smaller and smaller, you want fewer and fewer errors, the space requirements will increase. So, the final question is, is it a useful data structure? Can you set all the parameters so that you get you know, really interesting space error trade off and the answer is totally. So, let me give you an example. Let's go back to having eight bits of storage per object so that corresponds to b = eight. Then, what this pick formula indicates is we should use five or six hash functions and already you have an error probability of something like two percent which for a lot of the motivating applications we talked about is already good enough. And again, if you double the number of bits to say sixteen per object, then this error probability would be really small. Pushing you know one in 5,000 or something like that. So, to conclude at least in this idealized analysis which again, you should check against at any real world implementation although empirically, it is definitely achievable with well implemented bloom filter in nonpathological data to get this kind of performance even with really a ridiculously minuscule amount of space per object much less generally than storing the object itself, you can get fast inserts, fast look ups, you do have to have false positives but with a very controllable amount of error rates and that what's make bloom filters a win in a number of applications.

# zwk8-meterial

Recall that a set H of hash functions (mapping the elements of a universe U to the buckets {0,1,2,…,n?1}) is universal if for every distinct x,y∈U, the probability Prob[h(x)=h(y)] that x and y collide, assuming that the hash function h is chosen uniformly at random from H, is at most 1/n. In this problem you will prove that a collision probability of 1/n is essentially the best possible. Precisely, suppose that H is a family of hash functions mapping U to {0,1,2,…,n?1}, as above. Show that there must be a pair x,y∈U of distinct elements such that, if h is chosen uniformly at random from H, then Prob[h(x)=h(y)]≥1/n?1/|U|.

# zwk8-prog

The goal of this problem is to implement a variant of the 2-SUM algorithm covered in this week's lectures.

The file contains 1 million integers, both positive and negative (there might be some repetitions!).This is your array of integers, with the ith row of the file specifying the ith entry of the array.

Your task is to compute the number of target values t in the interval [-10000,10000] (inclusive) such that there are distinct numbers x,y in the input file that satisfy x+y=t. (NOTE: ensuring distinctness requires a one-line addition to the algorithm from lecture.)

Write your numeric answer (an integer between 0 and 20001) in the space provided.

OPTIONAL CHALLENGE: If this problem is too easy for you, try implementing your own hash table for it. For example, you could compare performance under the chaining and open addressing approaches to resolving collisions.