PROFESSOR: One more exacting lecture on hashing. And a couple reminders. I don't want to start out saying unpopular things, but we do have a quiz coming up next week on Tuesday. There will not be a lecture next Tuesday, but there will be a quiz. 7:30 to 9:30 Tuesday evening. I will send announcement. There's going to be a couple rooms. Some of you will be in this room. Some of you will have to go to a different room, since this room really can't hold 180 students taking a quiz. All right?

So hashing. I'm pretty excited about this lecture, because I think as I was talking with Victor just before this, if there's one thing you want to remember about hashing and you want to go implement a hash table, it's open addressing. It's the simplest way that you can possibly implement a hash table. You can implement a hash table using an array. We've obviously talked about link lists and chaining to implement hash tables in previous lectures, but we're going to actually get rid of pointers and link lists, and implement a hash table using a single array data structure, and that's the notion of open addressing.

Now in order to get open addressing to work, there's no free lunch, right? So you have a simple implementation. It turns out that in order to make open addressing efficient, you have to be a little more careful than if you're using the hash tables with chaining. And we're going to have to make an assumption about uniform hashing. I'll say a little bit more about that. But it's a different assumption from simple uniform hashing that Eric talked about. And we'll state this uniform hashing assumption.

And we look at what the performance is of open addressing under this assumption. And this is assumption is going to give us a sense of what good hash functions are for open addressing applications or for open addressing hash tables.

And finally we'll talk about cryptographic hashing. This is not really 6006 material, but it's kind of cool material. It has a lot of applications in computer security and cryptography. And so as we'll describe the notion of a cryptographic hash, and we'll talk about a couple of real simple and pervasive applications like password storage and file corruption detectors that you can implement using cryptographic hash functions, which are quite different from the regular hash functions that we're using in hash tables. Be it chaining hash tables or open addressing hash tables. All right?

So let's get started and talk about open addressing. This is another approach to dealing with collisions. If you didn't have collisions, obviously an array would work, right? If you could somehow guarantee that there were no collisions. When you have collisions, you have to worry about the chaining and ensuring that you can still find the keys even though you had two keys that collided into the same slot.

And we don't want to use chaining. The simplest data structure that we can possibly use are arrays. Back when I was a grad student, I went through and got a PhD writing programs in C, never using any other structure than arrays, because I didn't like pointers. And so open addressing is a way that you can implement hash tables doing exactly this.

And in particular, what we're going to do is assume an array structure with items. And we're going to assume that this one item-- at most one item per slot. So m has to be greater than or equal to n, right? So this is important because we don't have link lists. We can't arbitrarily increase the storage of a slot using a chain, and have n, which is the number of elements, be greater than m, right? Which you could in the link list table with chaining.

But here you only have these area locations, these indices that you can put items into. So it's pretty much guaranteed that if you want a working open addressing hash table that m, which is the number of slots in the table, should be greater than or equal to the number of elements, all right? That's important.

Now how does this work. Well, we're going to have this notion of probing. And the notion of probing is that we're going to try to see if we can insert something into this hash table, and if you fail we're actually going to recompute a slightly different hash for the key that we're trying to insert, the key value pair that we're trying to insert. All right? So this is an iterative process, and we're going to continually probe until we find an empty slot into which we can insert this key value pair. The key should index into it.

So you do have different hashes that are going to be computed based on this probing notion for a given key. All right? And so what we need now is a hash function that's different from the standard hash functions that we've talked about so far, which specifies the order of slots to probe, which is basically to try for a key. And this is going to be true for insert, search, and delete, which are three basic operations.

And they're a little bit different, all right? Just like they were different for the chaining hash table, they're different here, but they're kind of more different here. And you'll see what I mean when we go through this. And this is not just for one slot. It's going to specify an order of slots.

And so our hash function h is going to take the universe of keys and also take what we're going to call the trial count. So if you're lucky-- well, you get lucky in your first trial. And if you're not, you hope to get lucky in your second trial, and so on and so forth.

But the hash function is going to take two arguments. It's going to take the key as an argument, and it's going to take a trial, which is an integer between 0 to n minus 1, all right? And it's going to produce-- just like the chaining hash function it's going to produce a number between 0 and m minus 1, right? Where m is the number of slots in the table. All right. So that's the story.

In order to ensure that you are using the hash table corresponding to open addressing properly, what you want is-- and this is an important property-- that h k 1, so that's a key that you're given. And this could be an arbitrary key, mind you. So arbitrary key k. And what you have in terms of the slots that are being computed is this, h k 1, h k 2, and so on and so forth to h k n minus 1.

And what you want is for this vector to be a permutation of 0 1 and so on to n minus 1. And the reason for this hopefully is clear. It's because you want to be able to use the entirety of your hash table. You don't want particular slots to go unused.

And when you get to the point where the number of elements n is pretty close to m, and maybe there's just one slot left, OK? And you want to fill up this last slot with this key k that you want to put in there, and what you want to be able to say is that for this arbitrary key k that you want to put in there that the one slot that's free-- and it could be that first slot. It could be the 17th slot. Whatever-- That eventually the sequence of probes is going to be able to allow you to insert into that slot. All right?

And we generalize this notion into the uniform hashing assumption in a few minutes, but hopefully this makes sense from a standpoint of really load balancing the table and ensuring that all slots in the table are sort of equal opportunity slots. That you're going to be able to put keys in them as long as you probe long enough that you're going to be able to get there.

Now of course the fact that you're using one particular slot for one particular key depends on the order of keys that you're inserting into this table. Again, you'll see that as we go through an example, all right?

So that's the set up. That's the open addressing notion. And that as you can see, we're just going to go through a sequence of probes and our hash function is going to tell us what the sequences is, and so we don't need any pointers or anything like that.

So let's take a look at how this might work in practice. So maybe the easiest thing to do is to run through an example, and then I'll show you some pseudocode. But let's say that I have a table here, and I'm going to concentrate on the insert operation. And I'm going to start inserting things into this table. And right here I have seven slots up there.

So let's say that I want to insert 586 into the table, and I compute h of 586 comma 1, and that gives me 1. OK? This is the first insert. So I'm going to go ahead and stick 586 in here, all right? And then I insert, for argument's sake, 133. I insert 204 out here. And these are all things because the hash table is empty. 481 out here and so on. And because the hash table is empty, my very first trial is successful, all right?

So h of 481-- I'm not going to write this all out, but h 481 1 happens to be 6 and so on. All right? Now I get to the point where I want to insert 496. And when I try to insert 496, I have h 496 1. It happens to be 4. OK?

So the first thing that happens is I go in here, and I say oops. This slot is occupied, because this-- I'm going to have a special flag associated with an empty slot, and we can say it's none. And if it's not none, then it's occupied. And 204 is not equal to none. So I look at this, and I say the first probe actually failed. OK? And so h 496 1 equals 4 fails, so I need to go do h 496 2.

And h 496 2 may also fail. You might be in a situation where h 496 2 gives you 586. So this was h 496 1 h 496 2 might give you 586. And finally it may be that h 496 3, which is your third attempt, equals 3. So you go in, and you say great. I can insert 496. And let me write that in bold here. Out there. All right? So pretty straightforward.

In this case, you've gone through three trials in order to find an empty slot. And so the big question really here is other than taking care of search and delete, how long is this process going to take? All right? And I'm talking about that in a few minutes, but let me explain, now that you've seen insert, how search would work, right? Or maybe I get one of you guys to explain to me once you have insert, how would search work? Someone? Someone from the back? No one. You guys are always answering questions. Yeah, all the way in the back.

AUDIENCE: Would you just do the same kind of probing [INAUDIBLE] where you find it or you don't find it?

PROFESSOR: Right. So you do exactly. It's very similar to insert. You have a situation where you're going to none would indicate an empty slot. And you can think of this as being a flag. And in the case of insert, what you did was you-- insert k v would say keep probing. I'm not going to write the pseudocode for it. Keep probing until an empty slot is found. And then when it's found, insert item.

And as long as you have the permutation property that we have up there, and given that m is greater than or equal to n, you're guaranteed that insert is going to find a slot. OK? That's the good news. Now it might take awhile, and so we have a talk about performance a bit later, but it'll work. OK?

Now search is a little bit different. You're searching for a key k, and you essentially say you're going to keep probing. And you say as long as the slots encountered are occupied by keys not equal to k. So every time you probe, you go in there and you say I got a key. I found a hash for it. I go to this particular slot. I look inside of it, and I check to see whether the key that's stored inside of it is the same as the key I'm searching for. If not, I go to the next trial. If it is, then I return it. Right? So that's pretty much it.

And we keep probing until you either encounter k or find an empty slot. And this is the key. No pun intended. A notion which is that when you find an empty slot, it means that you have failed to discover this key. You fail to-- yeah, question back there?

AUDIENCE: What happens if you were to delete a key though?

PROFESSOR: I'll make you answer that question for a cushion. So we'll get to delete in a minute. But I want to make sure you're all on board with insert and search. OK? So these are actually fairly straightforward in comparison to delete. It's not like delete is much more complicated, but there is a subtlety there.

And so that's kind of neat, right? I mean this actually works. So if you had a situation where you were just accumulating keys, and you're looking for the number of distinct elements in the stream of data that was coming in, and that was pretty much it with respect to your program, you'd never have to delete keys, and this would be all that you'd have to implement. Right?

But let's talk about delete. Every once in awhile we'd want to delete a key? Yeah, you had a question?

AUDIENCE: I have a question about search. Why do you stop searching once you find an empty slot?

PROFESSOR: Because you're searching. So what that means is that you're looking to see if this key were already in the table. And if key were already in the table, you want to return the value associated with that key. If you find an empty slot, since you're using the same deterministic sequence of probes that you would have if you had inserted it, then-- that make sense? Good. All right. So so far so good? That's what works for insert and search.

Let's talk delete. So back there. How does delete work?

AUDIENCE: Well [INAUDIBLE] if you search until you find the none and assume that the key you're searching for was not put in there. But let's say you had one that was in that slot before and it got put back in, but then you delete the one that was in the slot before.

PROFESSOR: Great, great. You haven't told me how to fix it yet, but do you have the guts for this? No. OK, I think this veers to the right. I always wanted to do this to somebody in the back. All right. Whoa. All right, good catch. All right. OK. So you pointed out the problem, and I'm going to ask somebody else for a solution. All right?

But here's the problem. Here's the problem, and we can look at it from a standpoint of that example right there. Let's say for argument's sake that I'm searching-- now I've done all of the inserts that I have up there, OK? So I've inserted 496. All right? Then I delete 586 from the table, OK? I delete 586 from the table. So let's just say that what I end up doing-- I have 586, 133, 496, and then I have 204, and then a 481. And this is 0, 1, 2, et cetera.

So I'm deleting 586, and let's say I replace it with none. OK? Let's just say I replace it with none. Now what happens is that when I search for 496, according to this search algorithm what am I going to get?

AUDIENCE: None.

PROFESSOR: Well the first slot I'm going to look at is 1, and according to this search algorithm, I find an empty slot, right? And when I find an empty slot, I'm going to say I failed in the search. If you encounter k, you succeed and return the key value pair, right? Success means you return the value. And if you encounter an empty slot, it means that you've decided that this key is not in the table. And you say couldn't find it, right? That make sense?

So this is obviously wrong, right? Because I just inserted 496 into the table. So this would fail incorrectly. So failed to find the key, which is OK. I mean failure is OK if the key isn't there. But you don't want to fail incorrectly. Right? Everyone buy that? Everyone buy that? Good.

All right. So how do I fix it. Someone else? How do I fix this? Someone who doesn't have a cushion. All right, you.

AUDIENCE: [INAUDIBLE] you can mark that spot by a, and when search comes across a, you just [INAUDIBLE].

PROFESSOR: Right, great answer. We're now going to have to do a couple of different things for insert and search, OK? It's going to be subtly different, but the first thing we're going to do is we're going to have this flag, and I'll just call it delete me flag. OK? And we're going to say that when I delete something, replace deleted item with not the non flag, but a different flag that we'll call delete me. Is different from none. And that's going to be important, because now that you have a different flag, and you replace 586 with delete me, you can now do different things in insert versus search, right?

So in particular, what you would do is you'd have to modify this slightly, because the notion of an empty slot means that you're looking for none, right? And all it means is that-- well actually in some sense, the pseudo code doesn't really change because if you say you either encounter k or you would-- even if you encounter a delete me, you keep going. All right? That's the important thing. So I guess it does change, because I assume that you have only two cases here, but what you really have now are three cases.

The three cases are when you're doing the search is that you encounter the key, which is the easy case. You return it. You return the value. Or you can encounter a delete me flag, in which case you keep going. OK? And if you encounter an empty slot, which corresponds to none, at that point you know you failed and the key doesn't exist in the table. All right?

So let me just write that out. Insert treats delete me the same as none. But search keeps going and treats it differently. And that's pretty much it.

So what would happen in our example? Well, going through exactly the same example, we started from here, and then we decided to delete 586. And so if we replaced 586 not with none, but with delete me, and the next time around when you search for 496, you're searching for 496. And what would happen is that you would go look at 586-- the slot that contained 586, and you see that there's a delete me flag in there. And so you go to the next trial.

And then in the next trial, you discover that, in this case, you have-- I'm sorry. I had 204 first as the first trial, and then in the second trial I had 586. And I would continue beyond the second trial and get to third trial, and in fact return 496 in this case. I would get to returning 496 in my third trial, which is exactly what I want.

The interesting thing here is that you can reuse storage. I mean the whole point of deleting is that you can take the storage and insert other keys in there. Once you've freed up the storage. And you can do that by making insert treat delete me the same as the none. So the next time you want to insert you could-- if you happen to index into the index corresponding to 586, you can override that. The delete me flag goes away, and some other key-- call it 999 or something-- would get in there. And you're all set with that. OK? Any questions? This all makes sense?

So you could imagine coding this up with an array structure is fairly straightforward.

What remains here to be discussed is how well does this work, right? You have this extra requirement on the hash function corresponding to creating an extra argument as an input to it, which is this trial count. And you'd like to have this nice property of corresponding to a permutation. Can we actually design hash functions like this? And we'll take a look at a bad hash function, and then at a better one.

So let's talk about probing strategies, which is essentially the same as taking a hash function and changing it so it is actually applicable to open addressing. So the notion of linear probing is that you do h k i equals h prime k, which is some hash function that you've chosen, plus i mod m, where this is an ordinary hash function. OK? So that looks pretty straightforward.

What happens here? Does this satisfy the permutation argument? Before I forget. Does it satisfy the permutation property that I want h k 1, h k 2, h k m minus 1 to be a permutation? That make sense? Yep, yep. Because I then I start adding. The mod is precisely kind of this round robin cycle, so it's going to satisfy the permutation. That's good.

What's wrong with this? What's wrong with this? Someone?

AUDIENCE: The fact that [INAUDIBLE] keys, which they're all filled, then if you hit anywhere in here [INAUDIBLE] list of consecutive keys.

AUDIENCE: Right. That's excellent. Excellent, excellent answer. So this notion of clustering is basically what's wrong with this probing strategy. And in fact, I'm not going to do this particular analysis, but I'll give you a sense of why the statement I'm going to make is true.

But the notion of clustering is that you start getting consecutive groups of occupied slots, OK? Which keep growing. And so these clusters get longer and longer. And if you have a big cluster, it's more likely to grow bigger, right? Which is bad. This is exactly the wrong thing for load balancing, right? And clustering is the reverse of load balancing, right? If you have a bunch of clumps and you have a bunch of empty space in your table, that's bad. Right?

The problem with linear probing is that once you start getting a cluster, given the, let's say, the randomness in the hash function, and h prime k is a pretty good hash function and can randomly go anywhere. Well, if you have 100 slots and you have a cluster of size 4, well there's a for 4/100 chance, which is obviously four times greater than 1/100, even I can do that, to go into those four slots. And if you going into those four slots you're going to keep going down to the bottom, and you're going to make that a cluster of size five, right?

So that's the problem the linear probing, and you can essentially argue through making some probabilistic assumptions that if, in fact, you use linear probing that you lose your average constant time look up in your hash table for most load factors. So what's happening out here pictorially really is that you have a table and let's say you have a cluster.

And this is your cluster. So if your h k 1-- it doesn't really matter what it is-- but h k i maps to this cluster, then you're going to-- linear probing says that the next thing you're going to try is if you map to 42 in the cluster, the next thing you're going to try is 43, 44, until you get maybe to this slot here, which is 57, for argument's sake. Right?

So you're going to keep going, and you're going to try 15 times in this relatively dumb fashion to go down to get to the open slot, which is 57. And oh, by the way, at the end of this you just increased your cluster length by one. All right? So it doesn't really work.

And in fact, under reasonable probabilistic assumptions in terms of what your hash functions are, you can say that when you have alpha, which is essentially your load factor, which is n over m less than 0.99, you see clusters of size log n, OK? Right.

So this is a probabilistic argument, and you're assuming that you have a hash function that's a pretty good hash function. So h prime k can be this perfect hash function, all right? So there's a problem here beyond the choice of h prime k, which is this hash function that worked really well for chaining. All right? And the problem here is the linear probing aspect of it.

So what does that mean? If you have clusters of theta log n, then your search and your insert are not going to be constant time anymore. Right? Which is bad in a probabilistic sense. OK?

So how do we fix that? Well, one strategy that works reasonably well is called double hashing. And it literally means what it says. You have to run a couple of hashes. And so the notion of double hashing is that you have h k i equals h1 k plus i h2 k mod m. And h1 and h2 are just ordinary hash functions. OK?

Now the first thing that we need to do is figure out how we can guarantee a permutation, right? Because we still have that requirement, and it was OK for the linear probing part, but you still have this requirement that you need a permutation. And so those of you who are into number theory, can you tell me what property, what neat property of h2 and m can we ask for to guarantee a permutation? Do you have a question? You already do. Do you have a question?

AUDIENCE: [INAUDIBLE].

PROFESSOR: [INAUDIBLE] relatively prime. OK, good. So I figured some of you knew the answer, but I've seen you before. Right. Exactly right. Relatively prime. Just hand it to Victor. So h2 k and m being relatively prime, if that implies a permutation. It's similar to what we had before. You're multiplying this by i. i keeps increasing, and you're going to roll around. All right? I mean you could do a proof of it, but I'm not going to bother.

The important thing here is that you can now do something as simple as m equals 2 raised to r, and h2 k for all k is odd, and now you're in great shape. You can have your array to be 2 raised to something, which is what you really want. And you just use h2 k. You could even take a regular hash function and truncate it to make sure it's odd. You can do a bunch of things. There's hash functions that produce odd values, and you can use that. All right?

And so double hashing works fairly well in practice. It's a good way of getting open addressing to work. And in order to prove that open addressing actually works to the level at which chaining works, we have to make an assumption corresponding to uniform hashing. And I'm not going to actually do a proof, but it'll be in the notes.

But I do want to talk about the theorem and the result that the theorem implies, assuming you have the uniform hashing assumption. And let me first say that this is not the same as simple uniform happening, which talks about the independence of keys in terms of their mapping to slots. The uniform hashing assumption says that each key is equally likely to have any one of the m factorial permutations-- so we're talking about random permutations here-- as its probe sequence. All right?

This is very hard to get in practice. You can get pretty close using double hashing. But nobody's discovered a perfect hash function, deterministic hash function that satisfies this property. At least not that I know off.

So what does this imply? Assuming that you have this and double hatching gives you this property, to a large extent what this means is that if alpha is n over m, you can show that the cost of operations such as search, insert, delete, et cetera. And in particular we talk about insert is less than or equal to 1 divided by 1 minus alpha. OK?

So obviously this goes as alpha tends to 1. As alpha tends to 1, the load factor in the table gets large, and the number of expected probes that you need to do when you get an insert grows. And if alpha is 0.99, you're going, on average, require 100 probes. It's a constant number, but it's a pretty bad constant. Right?

So you really want alpha to be fairly small. And in practice it turns out that you have to re-size you're open addressing table when alpha gets beyond about 0.5, 0.6 or so, because by then you're really in trouble. Remember this is an average case we're talking about. All of this is using a probabilistic assumption.

But as you get to high alphas, suddenly by the time you get to 0.7, open addressing doesn't work well in relation to an equivalent table with the overall number of slots that correspond to a changing table, OK? So open addressing is easy to implement. It uses less memory because you don't need pointers. But you better be careful that your alpha stays around 0.5 and no more.

So all that means is you can still use it. You just have to re-size your table. You have slightly different strategies for resizing your table when you use open addressing as opposed to chaining hash tables. All right? So that's a summary of open addressing. I want to spend some time on cryptographic hashes in the time that I have left. I guess I have a few minutes left. But any questions about open addressing? Yep?

AUDIENCE: On this delete part, what's going to happen if, say, you fill the table up and then delete everything, and then you start searching. Isn't that going to be bad because it's going to search through everything?

PROFESSOR: So that's right. The bad thing about open addressing is that delete isn't instantaneous, right? In the sense that if you deleted something from the link list in your chaining table, then even if you went to that same thing, the chain got smaller, and that helps you, because your table now has lower load. But there's a delay associated with load when you have the delete me flag. OK?

So in some sense the alpha that you want to think about, you should be careful as to how you define alpha. And that's one of the reasons why when you get alpha being 0.5, 0.6 you get into trouble, because if you have all these delete me flags, they're still hurting you.

AUDIENCE: And when you resize do those delete me flags get deleted?

PROFESSOR: When you completely resize and you redo the whole thing, then you can clean up the delete me's and turn them into nones because you're rehashing it. All right. So yeah, back there. Question?

AUDIENCE: Yes, can you explain how you got the equation that the cost of operation insert is less than or equal to 1 over [INAUDIBLE].

PROFESSOR: That's a longish proof, but let me explain to you how that comes out. Basically the intuition behind the proof is that we're going to assume some probability p. And initially you're going to say something like if the table, your p-- I'll just write this out here-- is m minus n divided by m. So what is that? Right now I have n elements in the table, and I have m slots, OK?

So the probability that my very first trial is going to succeed is going to be m minus n divided by m, because these are the number of empty slots. And assuming my permutation argument, I could go into one of them. And so that's what I have here. And if you look at what this is, this is 1 minus alpha, OK?

And so then you run off and you remember 6041 or the high school probability course that you take, and you say generally speaking, you're going to be no worse than p for every trial. And so if you assume the worst and say every trial has a probability of success of p, the expected number of trials is 1/p, OK? And that's how you got the 1 over 1 minus alpha. So you'll see that written in gory detail in the notes. All right? OK.

Expected to have a little more time in terms of talking about cryptographic hashes, but cryptographic hashes are not going to be on the quiz. This is purely material FYI. For your interest only. And again I have some notes on it, but I want to give you a sense of the other kinds of hashes that exist in the world, I guess. And hashes that are used for many different applications.

So maybe the best way of motivating this is through an example. So let's talk about an example that is near and dear to every security person's heart and probably to people who aren't interested in security as well, which is password storage. So think about how, let's say, Unix systems work when you type in your password. You're typing in your password [INAUDIBLE], and this is true for other systems as well, but you have a password. And my password is a permutation of my first daughters first name.

[LAUGHTER]

Yeah, but haven't given it away, right? Haven't given it away. And so this password is something that I'm typing in every day, right? Now the sum check that needs to happen to ensure that I'm typing in the right password. So what is a dumb way of doing things. What's a dumb way of building systems?

AUDIENCE: Storing [INAUDIBLE].

PROFESSOR: This is kind of a freebie.

AUDIENCE: [INAUDIBLE].

PROFESSOR: In situ hashing. That's better. So you'd store it. I offered the dumb way. So there's a perfectly valid answer. So you could clearly store this in plain text in some file and you could call it slash etc slaw password. And you could make it read for the work, right? And that'd be great, and people do that, right?

But what you would rather do is you want to make sure that even the sysadmin doesn't know my password or your password, right? So how do you do that? Well you do that using a cryptographic hash that has this interesting property that is one way, OK? And what that means is that given h of x-- OK, this is the value of the hash-- it is very hard to find the x such that x basically hashes to this value.

So if h of x equals let's call it q, then you're only given h of x. And so what do you do now? Well, it's beautiful. Assuming you have this one way hash, this cryptographic hash, in your etc slash password file, you have something like login name, [INAUDIBLE], which happens to be the hash of my daughter's first name, or something.

But this is what's stored in there and the same thing for a bunch of different users, right? So when I log in and I type in the actual password, what does the system do? What does the system do? It hashes it. It takes x prime, which is the typed in password, which may or may not be equal to my password, because somebody else might be trying to break in, or I just mistyped, or forgot my daughter's first name, which would be bad.

And it will just check to see-- it doesn't need x, because it's stored h of x in the system, so it doesn't need x. So if we just compare against what I typed in, it would compute the hash again. And then would let me in assuming that these things matched and would not let me in if it didn't. So now we can talk about-- and I don't have time for this, but you can certainly read up on it on Wikipedia and a bunch in the notes. You can talk about what properties should this hash function have, namely one way collision resistance, in order to solve these problems and other problems. I'm happy to stick around and answer questions.