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

PROFESSOR: A trilogy, if you will, on hashing. We did a lot of cool hashing stuff. In some sense, we already have what we want with hashing. Hashing with chaining, we can do constant expected time, I should say, constant as long as-- yeah. If we're doing insert, delete, and exact search. Is this key in there? If so, return the item. Otherwise, say no. And we do that with hashing with chaining. Under the analysis we did was with simple uniform hashing.

An alternative is to use universal hashing, which is not really in this class. But if you find this weird, then this is less weird. And hashing with chaining, the idea was we had this giant universe of all keys, could be actually all integers. So it's infinite. But then what we actually are storing in our structure is some finite set of n keys. Here, I'm labeling them k1 through k4, n is four. But in general, you don't know what they're going to be. We reduce that to a table of size m by this hash function h-- stuff drawn in red. And so here I have a three way collision. These three keys all map to one, and so I store a linked list of k1, k4 and k2. They're in no particular order. That's the point of that picture. Here k3 happens to map to its own slot. And the other slots are empty, so they just have a null saying there's an empty linked list there.

Total size of this structure is n plus m. There's m to store the table. There's n to store the sum of the lengths of all the lists is going to be n. And then we said the expected chain length, if everything's uniform, then the probability of a particular key going to a particular slot is 1/m. And if everything's nice and independent or if you use universal hashing, you can show that the total expected chain length is n/m. n independent trials, each probability 1/m of falling here. And we call that alpha, the load factor. And we concluded that the operation time to do an insert, delete, or search was order 1 plus alpha. So that's n expectation. So that was hashing with chaining.

This is good news. As long as alpha is a constant, we get constant time. And just for recollection, today we're not really going to be thinking too much about what the hash function is, but just remember two of them I talked about-- this one we actually will use today, where you just take the key and take it module m. That's one easy way of mapping all integers into the space zero through m minus 1. That's called the division method. Multiplication method is more fancy. You multiply by a random integer, and then you look at the middle of that multiplication. And that's where lots of copies of the key k get mixed up together and that's sort of the name of hashing. And that's a better hash function in the real world. So that's hashing with chaining. Cool?

Now, it seemed like a complete picture, but there's one crucial thing that we're missing here. Any suggestions? If I went to go to implement this data structure, what don't I know how to do? And one answer could be the hash function, but we're going to ignore that. I know you know the answer. Does anyone else know the answer? Yeah.

AUDIENCE: Grow the table.

PROFESSOR: Grow the table. Yeah. The question is, what should m be? OK, we have to create a table size m, and we put our keys into it. We know we'd like m to be about the same as n. But the trouble is we don't really know n because insertions come along, and then we might have to grow the table. If n gets really big relative to m, we're in trouble because this factor will go up and it will be no longer constant time.

The other hand, if we set m to be really big, we're also kind of wasteful. The whole point of this structure was to avoid having one slot for every possible key because that was giant. We want it to save space. So we want m to be big enough that our structure is fast, but small enough that it's not wasteful in space. And so that's the remaining question. We want m to be theta n. We want it to be omega n. So we want it to be at least some constant times n, in order to make alpha be a constant. And we want it to be big O of n in order to make the space linear. And the way we're going to do this, as we suggested, is to grow the table.

We're going to start with m equals some constant. Pick your favorite constant. That's 20. My favorite constant's 7. Probably want it to be a power of two, but what the hell? And then we're going to grow and shrink as necessary. This is a pretty obvious idea. The interesting part is to get it to work. And it's going to introduce a whole new concept, which is amortization. So it's going to be cool. Trust me. Not only are we going to solve this problem of how to choose m, we're also going to figure out how the Python data structure called list, also known as array, is implemented. So it's the exactly the same problem. I'll get to that in a moment.

So for example, let's say that we-- I said m should be theta n. Let's say we want m to be at least n at all times. So what happens, we start with m equals 8. And so, let's say we start with an empty hash table, an empty dictionary. And then I insert eight things. And then I go to insert the ninth thing. And I say, oh, now m is bigger than n. What should I do? So this would be like at the end of an insertion algorithm. After I insert something and say oh, if m is greater than n, then I'm getting worried that m is getting much bigger than n. So I'd like to grow the table. OK? Let's take a little diversion to what does grow a table mean. So maybe I have current size m and I'd like to go to a new size, m prime. This would actually work if you're growing or shrinking, but m could be bigger or smaller than m prime. What should I do-- what do I need to do in order to build a new table of this size? Easy warm up. Yeah?

AUDIENCE: Allocate the memory and then rehash [INAUDIBLE].

PROFESSOR: Yeah. Allocate the memory and rehash. So we have all these keys. They're stored with some hash function in here, in table size m. I need to build an entirely new table, size m prime, and then I need to rehash everything. One way to think of this is for each item in the old table, insert into the new table, T prime. I think that's worth a cushion. You got one? You don't want to get hit. It's fine. We're not burning through these questions fast enough, so answer more questions. OK. So how much time does this take? That's the main point of this exercise. Yeah?

AUDIENCE: Order n.

PROFESSOR: Order n. Yeah, I think as long as m and m prime are theta n, this is order n. In general, it's going to be n plus m plus m prime, but you're right. Most of the time that's-- I mean, in the situation we're going to construct, this will be theta n. But in general, there's this issue that, for example, to iterate over every item in the table you have to look at every slot. And so you have to pay order m just to visit every slot, order n to visit all those lists, m prime just to build the new table, size m prime. Initialize it all to nil. Good.

I guess another main point here is that we have to build a new hash function. Why do we need to build a new hash function? Because the hash function-- why did I call it f prime? Calling it h prime. The hash function is all about mapping the universe of keys to a table of size m. So if m changes, we definitely need a new hash function. If you use the old hash function, you would just use the beginning of the table. If you add more slots down here, you're not going to use them. For every key you've got to rehash it, figure out where it goes. I think I've drilled that home enough times. So the question becomes when we see that our table is too big, we need to make it bigger. But how much bigger? Suggestions? Yeah?

AUDIENCE: 2x.

PROFESSOR: 2x. Twice m. Good suggestion. Any other suggestions? 3x? OK. m prime equals 2 m is the correct answer. But for fun, or for pain I guess, let's think about the wrong answer, which would be, just make it one bigger. That'll make m equal to n again, so that seems-- it's at least safe. It will maintain my invariant that m is at least n. I get this wrong-- sorry, that's the wrong way. n is greater than m. I want m to be greater than or equal to n.

So if we just incremented our table size, then the question becomes, what is the cost of n insertions? So say we start with an empty table and it has size eight or whatever, some constant, and we insert n times. Then after eight insertions when we insert we have to rebuild our entire table. That takes linear time. After we insert one more, we have to rebuild. That takes linear time. And so the cost is going to be something like, after you get to 8, it's going to be 1 plus 2 plus 3 plus 4. So a triangular number. Every time we insert, we have to rebuild everything. So this is quadratic, this is bad.

Fortunately, if all we do is double m, we're golden. And this is sort of the point of why it's called table-- I call it table resizing there. Or to not give it away, but this is a technique called table doubling. And let's just think of the cost of n insertions. There's also deletions. But if we just, again, start with an empty table, and we repeatedly insert, then the cost we get-- if we double each time and we're inserting, after we get to 8, we insert, we double to 16. Then we insert eight more times, then we double to 32. Then we insert 16 times, then we double to 64. All these numbers are roughly the same. They're within a factor of two of each other. Every time we're rebuilding in linear time, but we're only doing it like log end times.

If we're going from one to n, their log end growths-- log end doublings that we're able to do. So you might think, oh, it's n log n. But we don't want n log n. That would be binary search trees. We want to do better than n log n. If you think about the costs here, the cost to rebuild the first time is concepts, like 8. And then the cost to rebuild the second time is 16, so twice that. The cost to build the next time is 64. So these go up geometrically. You've got to get from 1 to n with log end steps. The natural way to do it is by doubling, and you can prove that indeed this is the case. So this is a geometric series. Didn't mean to cross it out there. And so this is theta n.

Now, it's a little strange to be talking about theta n. This is a data structure supposed to be constant time per operation. This data structure is not constant time per operation. Even ignoring all the hashing business, all you're trying to do is grow a table. It takes more than constant time for some operations. Near the end, you have to rebuild the last time, you're restructuring the entire table. That take linear time for one operation. You might say that's bad. But the comforting thing is that there are only a few operations, log end of them, that are really expensive. The rest are all constant time. You don't do anything. You just add into the table.

So this is an idea we call amortization. Maybe I should write here-- we call this table doubling. So the idea with amortization, let me give you a definition. Actually, I'm going to be a little bit vague here and just say-- T of n. Let me see what it says in my notes. Yeah. I say T of n.

So we're going to use a concept of-- usually we say running time is T of n. And we started saying the expected running time is some T of n plus alpha or whatever. Now, we're going to be able to say the amortized running time is T of n, or the running time is T of n amortized. That's what this is saying. And what that means is that it's not any statement about the individual running time of the operations. It's saying if you do a whole bunch of operations, k of them, then the total running time is, at most, k times T of n. This is a way to amortize, or to-- yeah, amortize-- this is in the economic sense of amortize, I guess.

You spread out the high costs so that's it's cheap on average all the time. It's kind of like-- normally, we pay rent every month. But you could think of it instead as you're only paying $50 a day or something for your monthly rent. It's maybe-- if you want to smooth things out, that would be a nice way to think about paying rent, or every second you're paying a penny or something. It's close, actually. Little bit off, factor or two.

Anyway, so that's the idea. So you can think of-- this is kind of like saying that the running time of an operation is T of n on average. But put that in quotes. We don't usually use that terminology. Maybe put a Tilda here. Where the average is taken over all the operations. So this is something that only makes sense for data structures. Data structures are things that have lots of operations on them over time. And if you just-- instead of counting individual operation times and then adding them up, if you add them up and then divide by the number of operations, that's your amortized running time.

So the point is, in table doubling, the amortized running time is beta 1. Because it's n in total-- at this point we've only analyzed insertions. We haven't talked about deletions. So k inserts. If we're just doing insertions, take beta k time in total. So this means constant amortized per insert.

OK, it's a simple idea, but a useful one because typically-- unless you're in like a real time system-- you typically only care about the overall running time of your algorithm, which might use a data structure as a sub routine. You don't care if individual operations are expensive as long as all the operations together are cheap. You're using hashing to solve some other problem, like counting duplicate words in doc dist. You just care about the running time of counting duplicate words. You don't care about how long each step of the for loop takes, just the aggregate. So this is good most of the time. And we've proved it for insertions.

It's also true when you have deletions. You have k inserts and deletes. They certainly take order k time. Actually, this is easy to prove at this point because we haven't changed delete. So, what delete does is it just deletes something from the table, leaves the table the same size. And so it actually makes life better for us because if it decreases m, in order to make m big again, you have to do more insertions than you had to before. And the only extra cost we're thinking about here is the growing, the rebuild cost from inserting too big. And so this is still true. Deletions only help us.

If you have k total inserts and deletes, then still be order k. So still get constant amortized. But this is not totally satisfying because of table might get big again. m might become much larger than n. For example, suppose I do n inserts and then I do n deletes. So now I have an empty table, n equals 0, but m is going to be around the original value of n, or the maximum value of n over time. So we can fix that. Suggestions on how to fix that? This is a little more subtle. There's two obvious answers. One is correct and the other is incorrect. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Good. So option one is if the table becomes half the size, then shrink-- to half the size? Sure. OK. That's on the right track. Anyone see a problem with that? Yeah?

AUDIENCE: [INAUDIBLE] when you're going from like 8 to 9, you can go from 8 to 9, 9 to 8, [INAUDIBLE].

PROFESSOR: Good. So if you're sizing and say you have eight items in your table, you add a ninth item and so you double to 16. Then you delete that ninth item, you're back to eight. And then you say oh, now m equals n/2, so I'm going to shrink to half the size. And if I insert again-- delete, insert, delete, insert-- I spend linear time for every operation. So that's the problem. This is slow. If we go from 2 to the k to 2 to the k plus 1, we go this way via-- oh sorry, 2 to the k plus 1. Then, I said it right, insert to go to the right, delete to go to the left. Then we'll get linear time for operation. That is that. So, how do we fix this? Yeah.

AUDIENCE: Maybe m equal m/3 or something?

PROFESSOR: M equals n over 3. Yep.

AUDIENCE: And then still leave it [INAUDIBLE].

PROFESSOR: Good. I'm going to do 4, if you don't mind. I'll keep it powers of 2. Any number bigger than 3 will work-- or any number bigger than 2 will work here. But it's kind of nice to stick to powers of two. Just for fun. I mean, doesn't really matter because, as you say, we're still going to shrink to half the size, but we're only going to trigger it when we are 3/4 empty. We're only using a quarter of the space. Then, it turns out you can afford to shrink to half the size because in order to explode again, in order to need to grow again, you have to still insert n over m-- m over 2 items. Because it's half empty.

So when you're only a quarter full, you shrink to become a half full because then to grow again requires a lot of insertions. I haven't proved anything here, but it turns out if you do this, the amortized time becomes constant. For k insertions and deletions, arbitrary combination, you'll maintain linear size because of these two-- because you're maintaining the invariant that m is between n and 4n. You maintain that invariant. That's easy to check. So you always have linear size. And the amortized running time becomes constant. We don't really have time to prove that in the class. It's a little bit tricky. Read the textbook if you want to know it.

That's table doubling. Questions? All right. Boring. No. It's cool because not only can we solve the hashing problem of how do we set m in order to keep alpha a constant, we can also solve Python lists. Python lists are also known as resizable arrays. You may have wondered how they work. Because they offer random access, we can go to the ith item in constant time and modify it or get the value. We can add a new item at the end in constant time. That's append. list.append. And we can delete the last item in constant time. One version is list.pop. It's also delete list, square bracket minus 1. You should know that deleting the first item is not constant time. That takes linear time because what it does is it copies all the values over. Python lists are implemented by arrays.

But how do you support this dynamicness where you can increase the length and decrease the length, and still keep linear space? Well, you do table doubling. And I don't know whether Python uses two or some other constant, but any constant will do, as long as the deletion constant is smaller than the insertion constant. And that's how they work. So in fact, list.append and list.pop are constant amortized. Before, we just said for simplicity, they're constant time and for the most part you can just think of them as constant time. But in reality, they are constant amortized.

Now for fun, just in case you're curious, you can do all of this stuff in constant worst case time per operation. May be a fun exercise. Do you want to know how? Yeah? Rough idea is when you realize that you're getting kind of full, you start building on the side a new table of twice the size. And every time you insert into the actual table, you move like five of the items over to the new table, or some constant-- it needs to be a big enough constant. So that by the time you're full, you just switch over immediately to the other structure. It's kind of cool. It's very tricky to actually get that to work. But if you're in a real time system, you might care to know that. For the most part, people don't implement those things because they're complicated, but it is possible to get rid of all these amortized. Cool.

Let's move onto the next topic, which is more hashing related. This was sort of general data structures in order to implement hashing with chaining, but didn't really care about hashing per se. We assumed here that we can evaluate the hash function in constant time, that we can do insertion in constant time, but that's the name of the game here. But otherwise, we didn't really care-- as long as the rebuilding was linear time, this technique works.

Now we're going to look at a new problem that has lots of practical applications. I mentioned some of these problems in the last class, which is string matching. This is essentially the problem. How many people have used Grep in their life? OK, most of you. How many people have used Find in a text editor? OK, the rest of you. And so this are the same sorts of problems. You want to search for a pattern, which is just going to be a substring in some giant string which is your document, your file, if you will.

So state this formally-- given two strings, s and t, you want to know does s occur as a substring of t? So for example, maybe s is a string 6006 and t is your entire-- the mail that you've ever received in your life or your inbox, or something. So t is big, typically, and s is small. It's what you type usually. Maybe you're searching for all email from Piazza, so you put the Piazza from string or whatever. You're searching for that in this giant thing and you'd like to do that quickly. Another application, s is what you type in Google. t is the entire web. That's what Google does. It searches for the string in the entire web. I'm not joking. OK? Fine. So we'd like to do that. What's the obvious way to search for a substring in a giant string? Yeah?

AUDIENCE: Check each substring of that length.

PROFESSOR: Just check each substring of the right length. So it's got to be the length of s. And there's only a linear number of them, so check each one. Let's analyze that. So a simple algorithm-- actually, just for fun, I have pseudocode for it. I have Python code for it. Even more cool. OK. I don't know if you know all these Python features, but you should. They're super cool. This is string splicing. So we're looking in t-- let me draw the picture. Here we have s, here we have t. Think of it as a big string. We'd like to compare s like that, and then we'd like to compare s shifted over one to see whether all of the characters match there. And then shifted over one more, and so on.

And so we're looking at a substring of t from position i the position i plus the length of s, not including the last one. So that's of length exactly, length of s. This is s. This is t. And so each of these looks like that pattern. We compare s to t. What this comparison operation does in Python is it checks the first characters, see if they're equal. If they are, keep going until they find a mismatch. If there's no mismatch, then you return true. Otherwise, you return false. And then we do this roughly length of t times because that's how many shifts there are, except at the end we run out of room. We don't care if we shift beyond the right because that's clearly not going to match. And so it's actually length of t minus like of s. That's the number of iterations.

Hopefully I got all the index arithmetic right. And there's no plus ones or minus ones. I think this is correct. We want to know whether any of these match. If so, the answer is yes, s occurs as a substring of t. Of course, in reality you want to know not just do any match, but show them to me, things like that. But you can change that. Same amount of time. So what's the running time of this algorithm? So my relevant things are the length of s and the length of t. What's the running time?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Sorry?

AUDIENCE: [INAUDIBLE]

PROFESSOR: T by-- t multiplied by s, yeah. Exactly. Technically, it's length of s times length of t minus s. But typically, this is just s times t. And it's always at most s times t, and it's usually the same thing because s is usually smaller-- at least a constant factor than t. This is kind of slow. If you're searching for a big string, it's not so great. I mean, certainly you need s plus t. You've got to look at the strings. But s times t is kind of-- it could be quadratic, if you're searching for a really long string in another string. So what we'd like to do today is use hashing to get this down to linear time. So, ideas? How could we do that? Using hashing. Subtle hint. Yeah?

AUDIENCE: If we take something into account [INAUDIBLE].

PROFESSOR: OK, so you want to decompose your string into words and use the fact that there are fewer words than characters. You could probably get something out of that, and old search engines used to do that. But it's not necessary, turns out. And it will also depend on what your average word length is. We are, in the end, today, we're not going to analyze it fully, but we are going to get an algorithm that runs in this time guaranteed. In expectation because of a randomized-- yeah?

AUDIENCE: If we were to hash [INAUDIBLE] size s, that would [INAUDIBLE] and then we would check the hash [INAUDIBLE].

PROFESSOR: Good. So the idea is to-- what we're looking at is a rolling window of t always of size s. And at each time we want to know, is it the same as s? Now, if somehow-- it's expensive to check whether a string is equal to a string. There's no way getting around that. Well, there are ways, but there isn't a way for just given two strings. But if somehow instead of checking the strings we could check a hash function of the strings, because strings are big, potentially. We don't know how big s is.

And so the universe of strings of length s is potentially very big. It's expensive to compare things. If we could just hash it down to some reasonable size, to something that fits in a word, then we can compare whether those two words are equal, whether those two hash values are equal, whether there's a collision in the table. That would somehow-- that would make things go faster. We could do that in constant time per operation. How could we do that? That's the tricky part, but that is exactly the right idea. So-- make some space.

I think I'm going to do things a little out of order from what I have in my notes, and tell you about something called rolling hashes. And then we'll see how they're used. So shelve that idea. We're going to come back to it. We need a data structure to help us do this. Because if we just compute the hash function of this thing, compare it to the hash function of this thing, and then compute the hash function of the shifted value of t and compare it, we don't have to recompute the hash of s. That's going to be free once you do it once. But computing the hash function of this and then the hash function of this and the hash function of this, usually to compute each of those hash function would take length of s time. And so we're not saving any time. Somehow, if we have the hash function of this, the first substring of length s, we'd like to very quickly compute the hash function of the next substring in constant time. Yeah?

AUDIENCE: You already have, like, s minus 1 of the characters of the--

PROFESSOR: Yeah. If you look at this portion of s and this portion of s, they share s minus 1 of the characters. Just one character different. First one gets deleted, last character gets added. So here's what we want. Given a hash value-- maybe I should call this r. It's not the hash function. Give it a rolling hash value. You might say, I'd like to be able to append a character. I should say, r maintains a string. There's some string, let's call it x. And what r.append of c does is add character c to the end of x.

And then we also want an operation which is-- you might call it pop left in Python. I'm going to call it skip. Shorter. Delete the first character of x. And assuming it's c. So we can do this because over here, what we want to do is add this character, which is like t of length of s. And we want to delete this character from the front, which is t of 0. Then we will get the next strength. And at all times, r-- what's the point of this r? You can say r-- let's say open paren, close paren-- this will give you a hash value of the current strength. So this is basically h of x for some hash function h, some reasonable hash function.

If we could do this and we could do each of these operations in constant time, then we can do string matching. Let me tell you how. This is called the Karp-Rabin string matching algorithm. And if it's not clear exactly what's allowed here, you'll see it as we use it. First thing I'd like to do is compute the hash function of s. I only need to do that once, so I'll do it. In this data structure, the only thing you're allowed to do is add characters. Initially you have an empty string. And so for each character in s I'll just append it, and now rs gives me a hash value of s. OK?

Now, I'd like to get started and compute the hash function of the first s characters of t. So this would be t up to length of s. And I'm going to call this thing rt, that's my rolling hash for t. And append those characters. So now rs is a rolling hash of s. rt is a rolling hash of the first s characters in t. So I should check whether they're equal. If they're not, shift over by one. Add one character at the end, delete character from the beginning. I'm going to have to do this many times.

So I guess technically, I need to check whether these are equal first. If they're equal, then we'll talk about it in a moment. The main thing I need to do is this for loop, which checks all of the other. And all I need to do is throw away the first letter, which I know is t of i minus length of s. And add the next letter, which is going to be t of i. And then after I do that, I don't change hs because that's fixed. That's just-- or, sorry, I switched from h to-- in my notes I have h. I've been switching to r, so all those h's are r's. Sorry about that.

So then if rs equals rt, then potentially that substring of t matches s. But it's potentially because we're hashing. Things are only true in expectation. There's some probability of failure. Just because the hash function of two strings comes out equal doesn't mean the strings themselves are equal, because there are collisions. Even distinct strings may map to the same slot in the table. So what we do in this situation is check whether s equals t-- I did it slightly less conveniently than before-- it's like i minus length of s plus 1 to i plus 1. Oh well. It wasn't very beautiful but it works.

So in this case, I'm going to check it character by character. OK? If they're equal, then we found a match. So it's kind of OK that I spent all this time to check them. In particular, if I'm just looking for the first match-- like you're searching through a text document, you just care about the first match-- then you're done. So yeah, I spent order s time to do this, but if they're equal it's sort of worth that effort. I found the match. If they're not equal, we basically hope or we will engineer it so that this happens with probability at most 1/s. If we can do that, then the expected time here is constant.

So that would be good because then, if skip and append take constant time and this sort of double checking only takes constant expected time-- except when we find matches and then we're OK with it-- then this overall thing will take linear time. In fact, the proper thing would be this is you pay s plus t, then you also pay-- for each match that you want to report, you pay length of s. I'm not sure whether you can get rid of that term. But in particular, if you just care about one match, this is linear time. It's pretty cool.

There's one remaining question, which is how do you build this data structure? Is the algorithm clear though? I mean, I wrote it out in gory detail so you can really see what's happening, also because you need to do it in your problem set so I give you as much code to work from as possible. Question?

AUDIENCE: What is rs?

PROFESSOR: rs is going to represent a hash value of s. You could just say h of s. But what I like to show is that all you need are these operations. And so given a data structure that will compute a hash function, given the append operation, what I did up here was just append every letter of s into this thing, and then rs open paren, close paren gives me the hash function of s.

AUDIENCE: You said you can do r.append over here, but then you said rs--

PROFESSOR: Yeah. So there are two rolling hashes. One's called rs and one's called rt. This was an ADT and I didn't say it at the beginning-- line one I say rs equals a new rolling hash. rt equals a new rolling hash. Sorry, I should bind my variables. So I'm using two of them because I want to compare their values, like this. Other questions? It's actually a pretty big idea. This is an algorithm from the '90s, so it's fairly recent. And it's one of the first examples of really using randomization in a super cool way, other than just hashing as a data structure. All right.

So the remaining thing to do is figure out how to build this ADT. What's the data structure that implements this, spending constant time for each of these operations. Now, to tell you the truth, doing it depends on which hashing method you use, which hash function you want to use. I just erased the multiplication method because it's a pain to use the multiplication method. Though I'll bet you could use it, actually. That's an exercise for you think about.

I'm going to use the division method because it's the simplest hash function. And it turns out, in this setting it does work. We're not going to prove that this is true. This is going to be true in expectation. Expected time. But Karp and Rabin proved that this running time holds, even if you just use a simple hash function of the division method where m is chosen to be a random prime. Let's say about is big as-- let's say at least as big as length of s. The bigger you make it, the higher probability this is going to be true. But length of s will give you this on average.

So we're not going to talk about in this class how to find a random prime, but the algorithm is choose a random number of about the right size and check whether it's prime. If it's not, do it again. And by the prime number theorem, after log end trials you will find a prime. And we're not going to talk about how to check whether a number's prime, but it can be done. All right. So we're basically done.

The point is to look at-- if you look at an append operation and you think about how this hash function changes when you add a single character. Oh, I should tell you. We're going to treat the string x as a multi digit number. This is the sort of prehash function. And the base is the size of your alphabet. So if you're using Ascii, it's 256. If you're using some unique code, it might be larger. But whatever the size of your characters in your string, then when I add a character, this is like taking my number, shifting it over by one, and then adding a new value.

So how do I shift over by one? I multiply by a. So if I have some value, some current hash value u, it changes to u times a-- or sorry, this is the number represented by the string. I multiply by a and then I add on the character. Or, in Python you'd write ord of the character. That's the number associated with that character. That gives me the new string. Very easy. If I want to do is skip, it's slightly more annoying. But skip means just annihilate this value. And so it's like u goes to u minus the character times a to the power size of u minus 1. I have to shift this character over to that position and then annihilated it with a minus sign. You could also do x or.

And when I do this, I just think about how the hash function is changing. Everything is just modulo m. So if I have some hash value here, r, I take r times a plus ord of c and I just do that computation modulo m, and I'll get the new hash value. Do the same thing down here, I'll get the new hash value. So what r stores is the current hash value. And it stores a to the power length of u or length of x, whatever you want to call it. I guess that would be a little better. And then it can do these in constant a number of operations. Just compute everything modulo m, one multiplication, one addition. You can do append and skip, and then you have the hash value instantly. It's just stored. And then you can make all this work.