ProgrammingAbstractions-Lecture24

Instructor (Julie Zelenski):What are we doing today? We're gonna talk about hashing. Hashing's one of the coolest things you're ever gonna learn in 106b, so it's a good day to be here and learn something really neat, kinda clever, inventive idea for how you do search and retrieval in a very efficient way. That's covered in chapter 11 of the text and that actually is the last bit of the text, right, is chapter 11. So what we're gonna do next week is we're gonna try to pull some stuff together.

I'm gonna do one advanced lecture on some neat data structure I think that's fun to explore together and then try to kinda pull together a little bit of kind of what the big picture is now that you've seen all these things from the client side and the low level side kinda wrap it all up for you and get you ready to think about the final exam. Final exam is actually two weeks from today, so today is Friday, and our exam is at the Friday of exam week, so it's in the near future, but of course between now and then is pathfinders, your big task to get done before that. I'll be at the Terman café today after class, but this is your last chance. So if you haven't made it yet, this is the time to come because actually next Friday I am gonna be at a conference, so it is time for coffee and snacks, right today. Please come. Okay, so I'm gonna put us back to where we were a lecture ago, two lectures back to Monday, where we had talked about ways we might implement the map abstraction, right, the key value pairing to allow for quick search and retrieval and we had looked at vector and had just done a little thought experiment about what sorting the vector would buy us, right, and that would give us the ability to do searches faster because we could use binary search, right.

But in the case of add, right, we'd still be slow because the need be kinda shuffled or rearranged in the continuous memory and then that's what motivated our introduction to trees there, the binary search tree, which is designed exactly to enable that kind of searching, right, that logarithmic divide and conquer, divide in half and work your way down and then because of the flexibility of the pointer base structure there, right, we can also do the insert in the same logarithmic time because we're no longer constrained by that contiguous memory that the array fact data structures force us to. And so, at that point what we have seen this logarithmic and both get valued at and so on the PQ, right, which you were just finishing, you'll notice that logarithmic is pretty fast. In fact logarithmic for most values of n that you would normally run into in an ordinary use, right, it's un-measurable, right. So those of you who got your heap up and running and correctly working logarithmic and time was discovering that you can put 10,000, 20,000, 50,000, 100,000 things in it and just never be able to measure what it took to NQ or DQ from those structures because logarithmic is very, very fast.

So that said, that's actually a fine point to say, we have a great math notation, we don't have to look further, but in fact I'm gonna push it a little bit today and we're gonna actually getting it down to where we can do both those operations in constant time, which is to say that no matter how large the table gets, right, we expect we'll be able to insert and retrieve with the exact same time required kind of for a 10,000 table as a million entry table which is pretty neat. To note that as we start to get to these more complicated

data structures, one thing we also need to keep in mind, right, is that there are other things of trade offs, right, in adding that complexity of the code that starting with something that's simple and a little slower performer but that's easy to write might actually solve the problem we have at hand. So maybe that we don't really wanna go to one of these fancier things because we don't really need it to be that much faster because the BST, right, adds a bunch of memory, right, so left and right pointers for every cell, which is an 8 byte overhead on top of the entry itself.

Now we have to deal with pointers, dynamic memories, all this allocation, deallocation, right, we've got a lot of recursion going on in there and so just opportunities for error and stuff to creep into and if we take the steps to provide tree balancing, which we only kind of grazed the surface of. We just mentioned the need for it. It actually adds quite a lot of complication to the code to do the rotations or the rebalancing that from a simple binary search tree. So in fact the whole package, right, of building a really guaranteed log n performing binary search tree is actually a pretty big investment of human capital that you have to kinda balance against the future hit off of it running faster. I'm gonna show this strategy today that actually is kind of simple to write. It gets us solve one case and doesn't have the degenerate looming that the binary search tree did. So, let me kinda just give you a theoretical concept to think about how you do stuff. You have a big dictionary. You've got, you know, like a many thousand page dictionary, right, and you want to look up a word. You certainly don't do linear search. You're about to look up the word xylophone and you're actually not going to start from the A's and kind of page your way through, you know, 2,000 pages to get to the X's.

You don't even – like we do binary search. You can do binary searches pretty fast, right? It turns out like starting at the M's to get something that's in X, right? You tend to know about where X is and often dictionaries have little tabs along the pages that give you an idea of kinda where the breaks for certain words are. Maybe it says Xa is here and Xe is there or something, that lets you kinda more quickly just illuminate all the surrounding noise in the dictionary, kinda get straight to the right region where the X words are and then you can do a simple search from there, maybe a little bit of a linear or binary search to kind of zero in on the page where the word is you wanna find is. This is the idea behind this concept called a hash table or based on the algorithmic technique of hashing is that rather than try to figure out how to kinda keep this very large collection sorted or organized in a way that you can kind of jump around within it, it says, "Well how about we maintain a bunch of different collections." We break up the data set into little tiny pieces and if we can tell, given a key, like which piece of the data set it's supposed to be in and we'll call those pieces buckets. I'm gonna put these in different buckets.

So if I had a whole bunch of students I might put all the freshmen in this bucket and the sophomores in this bucket or something like that and then when I look for a student I would say, "Well which class are you?" You say, "Freshmen." I'd look in the freshmen bucket. Okay. You know, the idea in a simple case would be like, "Okay, well I – it has some criteria by which I can divide you into these buckets that don't overlap and everyone has a unique bucket to go in and if I can make up enough of those categorizations, have as many of those buckets around to divide you up into, then I can

hopefully keep the set that lands in that bucket pretty small, so that I don't have a smaller of students. So maybe I did it by the first letter of your last name. I could say, "Well everybody whose last name begins with W goes here and everybody who's B goes here and so on." Well, when somebody comes to me then I don't have to look through the B's. You know, and if you have a class of 200 students and you have 26 letters of the alphabet then you'd think that if it were evenly divided across those letters, it's not going to be to be, it turns out in practice, but as a thought experiment there'd be about 4 people in a bucket and it'd be pretty fast to just look through the bucket and find the one who I was looking for.

Okay, so let's take this idea. This idea of, like, there's going to be some strategy for how we divide into buckets and we call that strategy the hash function. That, given a key, and a buckets and we'll just label them with numbers. Zero to the number of buckets minus one. So let's say I have 99 buckets, I've got zero to 98 that I will take the key and the key in this case is the string key that we're using in the map, and I have a hash function which given a key is input produces a number as output in the range zero to B minus one. And that is what's called the hash value for that key and that tells me which bucket to look at, either find a match for that if I'm trying to do a get value or where to place a new entry if I'm trying to add into the table. So that's the basic idea. I'm gonna talk a little bit about hash functions. In terms of writing a really good hash function is actually pretty complicated and so we're gonna think about it in sort of a – kind of a big picture perspective and think about what qualities a hash function has to have and then I'm actually not gonna go through proving to you about what mathematically constitutes a good hash function.

I'm just gonna give you some intuitive sense of what kinda things matter when designing a hash function. So it's given a key that maps it to a number. Typically it's gonna say, you know, zero to the number of buckets minus one. Often that means that somehow it's gonna compute something and then use the mod operation to bring it back in range. So it's gonna take your number and then say, "Okay. Well, if you have five buckets to divide it in, I will give you a number zero to four." It's very important that it be stable, that you put the key in, you get a number out. If you put that same key in later you should get the same number out. So it has to be sort of reliable guaranteed mapping. It can't be random. That said, you want it to almost look as though it's random in terms of how it maps things to buckets, that you don't want it to have a real systematic plan where it puts a whole bunch of things in bucket one and very few things in bucket two or somehow always uses the even buckets and not the odds ones or something like that. So you wanna have a strategy that's gonna take those keys and use your whole bucket range in an equal manner. So that if you put 100 keys in and you have 100 buckets you're hoping that, on average, each bucket will have one thing. That you won't have these clusters where 25 things are in one bucket, ten here and then a whole bunch of buckets are empty.

So you want this kinda divided across the range. So given that we have string input, likely what we're gonna be using is the characters in the string as part of the – how can I take these characters and somehow kinda jungle them up in a way to sort of move them around in these buckets in a way that's reliable so that I get that same string back, I get

the same one. But that doesn't actually produce a lot of artifacts of similar strings, let's say mapping to the same place, hopefully spreading them around. So for example, some really simple things you could use, you could just take the first letter. So I talked about this sorta with last names. If I was trying to divide up my students, right, I could divide them by the first letter of their last name. That would use exactly 26 buckets, no more, no less, and if I happen to have 26 buckets then at least there's some chance it would get some coverage across the range. There is likely to be clustering, right, some names are a lot more common or letters are a lot more commonly used than others. So it would tend to produce a little bit of an artifact, but it would be reliable and it would use 26 buckets.

If I had 100 buckets, then this would be kind of a crummy function because in fact it would use the first 26 buckets and none of the other ones, right, if it was just taking that letter and deciding. So, you know, in some situations this would get me a little bit of the thing I wanted. Things like using the length of the word. A very, very bad clustering function here, right because there'd be a bunch of strings that are, like, six characters, seven characters, ten characters, but very few that are one or two, and very few that are more than ten, or something. You'd have this very tight little cluster in the middle, right, and would not use a large bucket range effectively at all. More likely what you're gonna try to use is kind of more of the information about the word, the key that you have. Instead of just a single letter or the count of letters, it's trying to use kinda all the data you have at hand, which is to say all the letters, and so one way to do it would be to take the ASCII values for the letters, so A is 97, you know, B is 98, is you would take a letter, a word like atoms and you'd add 96 plus 99 plus 96, you know, add them together and then you'd get some sum of them, 500, 600, whatever it is, depending on how many letters there are and then use mod to back it back into – let's say your range was, you had 100 buckets, right, so you'd want it to be zero to 99, so you'd take that thing and then mod it by 100, then take the last two digits of it. So it'd be, okay, the sum to 524 then it goes into bucket 24.

So if you did that, right, and so if act is the word that you have, right, you'd add these codes together and let's say this is 97 and then this is 99 and then that's, you know, a hundred and something and so we end up with – I'm making this number up, right – 283, then we might mod to the last two digit and we'll say this one goes into bucket 83. And if we have, you know, dog, maybe that adds to 267 and we would mod that down to 67. So this actually tries to use, kinda, all the information at hand, all the letters that we have and bring them down, and so if you figure that across the space of all the numbers that are being added here you are just as likely to get 210 as you are to get 299, you know that any of the subsequent numbers in between, then hopefully you'd get kind of a spread across your buckets. There is one thing this clusters on. Does anybody see what words cluster in this system? That you'd think of as maybe not being related, but would land in the same bucket?

Student: To rearrange some numbers.

Student: Yes.

Instructor (Julie Zelenski):Any anagram, right, so you take cat which is just a ranagram – anagram of act, right, it's gonna add to the same thing, right, and so it would be a little bit of clustering in that respect. So anagrams come down to the same place. And so the words that you expect to, kinda like, act and cat to you seem like different words, they are gonna collide in this system and that might be a little bit of something we worry about.

So one way you can actually kind of avoid that in the way that the hash function we're going to eventually use will do, is it'll do this thing where not only does it add the letters together, but it also multiplies the letter by a very large prime number that helps to kinda say, "Well the large prime number times C plus the large prime number squared times A plus the large prime number cubed times C and then it just allows the, each position of the letter actually is also then encoded in the number that comes out and so something that's positionally different won't necessarily land in the same bucket without some other factors kinda coming into play. So a little bit of trying to just take the information and jumble it up a little bit. So let me show you a little bit about how this looks. Actually I'm gonna tell you about the collisions and then I'll show you the data structure. So certainly I said there's gonna be this problem where things collide, that 83 and 83 in this simple system of adding the codes will come out the same. So it can't be that each of these buckets holds exactly one thing.

The hashing, right, although we may try to get those buckets as small as possible we have to account for the possibility that two things will collide, that their hash function, even though they were different keys, will land in the same bucket based on how I've jumbled it up and so we're trying to avoid that, but when it does happen we also have to have a strategy for it. So we have to have a way of saying, "Well there can be more than one thing in a bucket." Or some other way of when there's a collision, deciding where to put the one that was placed in the table second. And in the strategy, we're gonna use the one called chaning, and the chaning basically says, "Well if once we have something in a bucket like ACT, if we hash something and it comes to the same bucket, we'll just tack it onto that bucket, in this case using a link list." So each of these buckets is actually gonna be a link list of the entries and so the entire table will be a vector or array of those. So let me show you, a little live action. So this is a hash table that has seven buckets. In this case number 0 through 6 are assigned to each one. They all start empty, and so these are all empty link lists illustrated by the null, and then the hash function is shown here like a black box.

You don't actually know how the workings of hash function is, but you know what it's job is to do, which is you give it a key and it gives you a number in the range 0 to 6, and that same key when passed, and again gives you the same number. So if I say 106B equals Julie, it pushes 106B through the hash function and says, "Okay, well 106B comes out as one." You add those numbers together, jumble them up, the letters, and you get one. And so it went into the table, and it said, "Okay, well that's bucket number one, make a new cell for 106B Julie and stick it in the table." So now there's a non empty chain in one place, the rest are empty. If I say 107 is being taught by Jerry, 107 happens to get the same hash function. 107, 106B, different letters but it just happen to come out that way, hash to one and so it went ahead in this case just tacked it on to the front of the

list. If I do 106A being taught by Patrick, 106A happens to sum to zero in the hash function. That's all we know about the black box that ends up loading it up into the zero's slot. I say, well wait, it's being taught by Nick, it also come out to be zero and so it ends up kinda pushing on the front of that.

So I go 103 is being taught by Maron and it's in bucket five and so as I kinda type things in I'm starting to see that things are just kinda going around in difference place in the table. A little bit of clustering in the top, a little bit of empty at the bottom, but as I continue to type in strings for other course numbers, I'd hopefully kinda fill out the table to where there was kind of an even, roughly even spread across that. When I want to look something up because I wanna see who's teaching 106B, then it will use the same process in reverse. It asks the hash function, "Well which bucket should I look in?" And it says, "Oh, you should look in bucket one because 106B's hash code is one." So all the other buckets are eliminated from consideration, so very quick kind of focused down to the right place and then it actually just does a link list reversal to find 106B in the bucket where it has to be, and so the key to realizing what makes this so magical, right, is that the choice of the number of buckets is up to me. I can pick the number of buckets to be whatever I want. I can pick it to be five, I can pick it to be 10, I can be to be 1,000, I can pick it to be a million.

If I choose the number of buckets to be kind of roughly in the same order of magnitude as the number of entries. So if I wanna put a thousand things in my table, if I make a thousand buckets to hold them in and my hash function can divide stuff pretty equally across those thousand buckets, then I will expect that each of those link lists is about length one. Some will be two, some will be zero, but if it's doing its job right then I have a bunch of little tiny buckets, each of which holds a tiny fraction of the data set, in this case one or two, and it's very easy to search through them. If I know I'm gonna put a million things in, then I make a million buckets, and then I divide one out of a million things, I run it through the hash function, it gives me this number that says, "Oh, it's 862,444." I go to that bucket and the item is either there in that bucket or I have to look a little bit to find it, but there's no other place in the table I have to look. So the fact that there's a million other entries kinda near by in these other buckets is completely not important to us, right, and so the searching and updating, adding, and replacing, and removing, and stuff all happens at the bucket level and by choosing the buckets to be the number, total number are gonna be roughly equal to the size of the things, then I have this constant time access to the tiny part of the subset of the set that matters. That's pretty cool.

Let's write some code. It's kinda good to see the whole thing doing what it needs to do. So I'm gonna go ahead and go back over here. All right, so I've got my map which has add and get value and not much else there, right, but kinda just set up and ready for us to go. What I'm gonna build here is a link list cell that's gonna hold a string key, a val type value, and a pointer to the next, and so I plan on having each one of these cells, right, for each entry that goes in the table and then I'm going to have an array of these, and I'm gonna make a constant for it, and then I'll talk a little bit about the consequences of that. So I'm gonna make a constant that is – so I make it 99. So then I have 99 buckets to start

with. That's not gonna solve all my problems, it's gonna show that it's kinda tuned for maps that expect to hold about a hundred things, but – and so I have an array in this case, so if you look at deciphering this declaration is a little bit complicated here. It says that buckets is the name of the variable, it is an array of num buckets length, so a fixed size, 99 entry array here. Each of those entries is a pointer to a cell, so ready to be the head pointer of a link list for us. I'm gonna add a – well, we'll get there in a second. I'm gonna go over here and work on the constructor first. So the first thing I'm gonna do is make sure that those pointers have good values. If I do not do this, right, they will be junk and there will be badness that will happen, so I can make sure that each of them starts off as an empty list, and then correspondently, right, I would need to be doing a delete all list cells here, but I've been lazy about doing that and I'm gonna continue to be lazy just knowing that I'd have to iterate through and then delete all the cells in each of those lists that are there.

And then for add and get value what I'm gonna – else we need to do is figure out in both cases, right, which is the appropriate list to operate on. So running the hash function on my key, seeing where it tells me to look, and then either searching it to find the match for get value to return the matching value or in the add case, to search for the matching entry to overwrite if it exists and if not, then to create a new cell and tack it on the front of the list. So let's write a helper function because they're both gonna do the same thing. I'm gonna need the same thing twice. So I'm gonna put a hash function in here that, given a key and a number of buckets, is gonna return to me the right bucket to look in, and then I'm gonna write a find cell function that, given a key and the pointer to a list, will find the cell that matches that key or return a null if it didn't find it. Okay, let me put my hash function in. I have a good hash function down there in a minute that I'm gonna pull out, but I just – so what I'm gonna do, I'm gonna make an interesting choice here and I'm gonna have my hash function just return zero, which seems pretty surprising.

But I'm gonna talk about why that is. It turns out, when I'm first building it that actually is an easy way to test that my code is working and to not be dependent on the behavior of the hash function. At this point it says, "We'll just put everything in the zero's bucket." Now that's not a good strategy for my long term efficiency, but it is a good way to test that my handling of the bucket, and the searching, and the finding, and inserting is correct, and the performance I should expect to be abysmal on this, right, it should be totally linear in terms of the number of elements to add or to get them because they'll all be in one link list with no easy access to them. And then eventually I can change this to divide across the buckets and get better spread, but the correctness of the hash table isn't affected by this choice. It's kind of an interesting way to think about designing the code. The other helper I have down here is gonna be my find cell that, given the head pointer to a list and the key, is gonna go find it, and it's basically just if this ker cells key equals the key, then we return it and then if we've gone through the whole list and haven't found it, we return a null.

So this guy is returning a cell T star, right, as the matching cell within the thing, and this one is gonna provoke that little C++ compilation snafu that we encountered on the binary search tree, as well, which is that cell T, right, is a type that's declared in the private

section within the scope of my map. So outside of the scope, which to say, before I've crossed the my map angle bracket val type colon, colon, it doesn't actually believe we're in my map scope yet. It hasn't seen that and it can't anticipate it, so I have to actually give the full name with the scope specifier on it, and then because of this use of the template name outside of its class is a special place for the C++ compiler, the type name keyword also needs to go there. Okay, so there's the heavy lifting of these two things, right, getting it to hash to the right place, which I kinda short changed, and then searching to find a cell. If I go back up here to get value, and I call the hash function, given my key and the number of buckets to find out which bucket to work on, I go looking for a match by calling find cell of the key in buckets of which bucket, and then if match does not equal null then I can return match as val and then otherwise we say, "No such key found." So that's the error case, right, for get value was that if it didn't find one, its behavior is to raise an error so that someone knows that they've asked something it can't deal with.

Okay, so the key thing here usually is that the hash is some sort of constant operation that just jumbles the key up, in this case it just returned zero, but could actually look at the elements of the key, and then it does its reversal down that link list, which given a correct working hash function should have divided them up in these little tiny chains, then we can just find the one quickly by looking through that link list. If we find it, right, we've got it, otherwise, right, we're in the error case. Add looks almost exactly the same. I start by getting the bucket, doing the find cell, if it's not – if it did find it, right, then we overwrite, and then in the second case, right, where we didn't find it, right, then we need to make ourselves a new cell, and copy in the key, copy in the value, set it – in this case, the easiest place, right, to add something to a link list is I should just do append it to the front, right, no need to make it hard on ourselves, right, we're not keeping them in any particular order, whatsoever, right, they're base on kind of order of entry.

So if somebody gives us a new one, we might as well just stick it in the front, that'd make our job easy. So it will point to the current front pointer of the cell, and then we will update the bucket pointer to point to this guy. So I think we are in okay shape here. Let me take a look and make sure I like what I did, right, so hashing, finding the match, if it found a non empty, so – this is the overwrite case of finding an existing cell, we just overwrite with that. Otherwise, making a new cell and then attaching it to be the front of the bucket there. So for example, in the case where the bucket's totally empty, it's good to kinda trace that. That's the most common case when we start, right, if we hash to bucket zero and we do a search of a null link list, we should get a null out, so then we create a new cell, right, and we set the next field to be what the current bucket's head pointer was, which was null. So there should be a single to list and then we update the bucket to point to this guy. So then we should have a single list cell with null trailing it or inserting into the empty bucket case. I like it. Let's see if I have a little code over here that adds some things, car, car, cat, with some numbers. Okay, and that tries to retrieve the value of car, which it wrote a three and then it wrote a four on top of it, so hopefully the answer we should get out of this is four. Okay, if I – be interesting to do something, like ask for something that we know is not in the table, right, to see that our error case gets handled correctly, the switch key's found.

And if I were to continue to do this, I could do a little bit of stress testing to see how that zero is causing us some performance implications, but if I sat there and just put tons and tons of things in there, right, made up strings like crazy and stuffed them in that with numbers, right, I would just be creating one long chain off the zero bucket, ignoring all the other ones, right, but then it's kinda stress testing my link list handling about the adding and searching that list, but would also show us that we expect that as we got to be, have 100 entries, 200 entries, 300 entries that subsequent adds and get values, right, would show a linear increase in performance because of the cluster that we got going there.

So let me give you a real hash function. I'll talk you through what it's doing and then I'll leave you with the thought of, that writing one of these actually correctly and reliably is actually more of an advance exercise than we're gonna look at, but this is actually a hash function that's taken from Don Knuth's Art of Computer Science, some of the work that guides a lot of computer scientists still to this day, and it has the strategy I basically told you about of, for all the characters that are in the string, right, adding it into the thing, but having multiplied, right, the previous hash code that's being built up by this large multiplier which is in effect kinda taking advantage of the positional information. And so the multiplier happens to be a very large negative number that's a prime that says, okay, the first time through, right, it'll take the hash code of zero and multiply it by and add the first character. So the first character gets added in without any modification. The next character, though, will take the previous one and multiply it by one power of the multiplier and then add in the next character, and then that sum, right, on the sub [inaudible] gets multiplied again by the multiplier, so raising it to the squared, and the third, and to the fourth powers, it works down the string.

The effectiveness actually is that it's gonna do a lot of overflow. That's a very large number and it's adding and multiplying in this very large space, so in effect we're actually counting on the fact that the addition and multiplication that is built into C++ doesn't raise any errors when the numbers get so large that they can't be represented. It just kind of truncates back down to what fits, and so it kind of wraps around using a modular strategy here, like a ring. And so we'll actually end up kind of making a very big number out of this. The longer the string is, the more those multiplies, the more those powers raise. We've got this huge number, what we do is kinda truncating it back to what fits in to a long and then when we're done, we take whatever number fell out of this process and we mod it by the number of buckets to give us a number from zero to buckets minus one. And so whatever gets stuffed into here, right, it's reliable in the sense that it's, every time you put in the number you'll get the same thing back. We know it will be in the range end buckets because of that last mod call that worked it out for us and we then use that to say which bucket to go look in when we're ready to do our work.

So if I switch that in for my old hash function, I shouldn't actually see any change from the outside, other than the performance, right, should suddenly actually be a lot faster, which is hard to tell when I have three things in there, but – I'm gonna take out that error case. In this case car and cat now are probably not very likely to be going to the same bucket. In fact I can guarantee they don't go in the same bucket, whereas before they

were kind of overlapping. I'll leave that code there for a second, although really I would say that point of hashing really is not to get so worried about how it is that the hash code does it's work, but to understand the general idea of what a hashing function has to do, right, what its role is and how it relates to getting this constant time performance. So I'll give you an example of hashing in the real world, I think is really interesting to see it actually happen and people not even realizing how hashing is so useful.

So I ordered something from REI, which is this camping store and it – they didn't have it in stock, so they had to place an order for it. This is actually kinda awhile ago, but I – so then when I called to see if it's in, I call them on the phone, I say, "Oh, is my order in?" And they don't ask me my name. I thought this was very funny. Like, "Oh, it's – I'm Julie. I wanna know about my order." They're like, "Okay, what are the last two digits of your phone number?" And I'm like – you know, it's not one of those things you can answer right off the top of your head. Okay, 21. Turns out last two digits of my phone number are 21. And they say – and they go and they look in the 21 basket and then they say, "What's your name?" Now they want to know my name. I'm like, "Okay, my name's Julie." Okay, they look it up and they're like, "Okay, it's here." I'm like, "Okay." So then I go to get it, like a couple days later, and I go to the store and I'm standing in the line waiting for them, and I look up on the wall and they have 100 baskets, little wire baskets up on the wall and they're labeled 0 through 9 over here and 0 through 9 over there, and when I come in they ask me the same question. "What's the last two digits of your phone number?" Now I'm all prepared because I've already been primed on this question.

I say, "21." Then they walk over here, right, to the 21 basket here. It's like the top digit in this digit, and there's only one piece of paper in there, and they pull out my order thing, and then they get me my tent and everybody's happy. So while I'm there, I try to engage the cashier on how this is an example of a real world hashing system and I still got my tent, I'll have you know, but it was close, right. They're like, "Okay, yeah, you crazy crackpot. You can now leave now." I was like looking at it and them and I tried to talk to them about the number of digits because in some sets, right, this is a very good example of what the investment in the number of buckets is versus what tradeoff it gives you, right. Because there's this very physical sort of set up of this, it's like by choosing a larger number of buckets, right, you can more fine grain your access, but that investment in buckets means you're kind of permanently installing that story. So in this case, right, they didn't use just the last digit of my phone number. They could've done that and they would have only had ten buckets on the wall, but then they would expect, you know, ten times as many things in each bucket.

And apparently their estimate was the number of active orders in this backorder category was enough to warrant being more than ten, you know, significantly more than ten, but not so much more than 100, then in fact 100 buckets was the right investment to make in their bucket strategy. They didn't put three buckets on the wall because, you know, like what are they gonna do, have this big 3D sort of thing. They didn't enjoy this discussion, I went on for hours with them and they were, like, really not amused. But it had, you know, the three digits, then you'd get this even more likelihood that each bucket would

be empty, but – or have a very small number of things, but given their set up, that seemed to be the right strategy was to say 100 buckets and now – they didn't ask me the first two digits of my phone number. They asked me the last two. Why does that matter?

Student: Because you're using all [inaudible].

Instructor (Julie Zelenski):Yeah. It's hated a lot. If you ask, like Stanford students, right, like especially when, you know, like campus numbers used to all be 49's or 72's or something, so like, if you use the first two digits, right, you'd have everybody's in the 49 bucket or the 72 bucket or something, and a whole bunch of other ones not used. An example, even the first two digits are never 00, like there's a whole bunch of buckets that don't even get used as the first two digits of a phone number. Phone numbers never start with zeros. That they rarely actually have zeros or ones in them at all. They try to use those for the area code and stuff in the leading digits.

So I thought it was just a really interesting exercise and like, oh yeah, they exactly had kinda thought through hashing. Of course, they did not think it was hashing, and they thought I was a crackpot, and they didn't want to give me my stuff, but I paid my money and they [inaudible], but it kinda shows you, okay what you're trying to do here is try to make that number of buckets, right, roughly equal to the number of elements so that if your hash function's dividing up across your whole world, you've got constant access to what you have.

Student: Why is there an L at the end of one item?

Instructor (Julie Zelenski):You know, that's actually just a – C++ isn't for this. This is a long value that without it, it assumes it's an int and that number is to big to fit in an int and it is twice as big as a long in terms of space and so the L needs to be there, otherwise it will try to read it as an int and through away part of the information I was trying to give it. So it's the way of identifying it along constant. So let me talk about this just a little bit more in terms of actual performance, right, we've got N over B, right, so if the division is complete, right, the number of entries over the number of buckets, if we make them kind of on the same order of magnitude, kind of in the same range, so having to sum this could be to make a little bit of an estimated guess about where we're going, and then there are some choices here in how we store each bucket, right.

We could use a little array, we could use a link list, right, we could use a vector. We expect this to be very small, is the truth. We're hoping that it's gonna be one or two, and so there's not a lot of reason to buy any real expensive structure or even bother doing things like sorting. Like you could sort them, but like if you expect there to be two things, what's the point? You're gonna spend more time figuring out whether to put it in the front or the back then you would gain at all by being able to find it a little faster. So the link list is just gonna be easiest way to get kind of allocated without over capacity, excess capacity that's unused in the bucket. You know it's gonna be small, use the simple strategy. It's also, though, an important thing that the hash type is likely to do in kind of a industrial strength situation is it does have to deal with this idea, well what if that number

of buckets that you predicted was wrong? And so the map as we have given it to you, right, has to take this sponge because it doesn't know in advance how many things are you gonna put in. The only way to know that is to wait and see as things get added, and then realize your buckets are getting full.

So typically the industrial strength version of the hash table is gonna track what's called the load factor. And the load factor is just the number of total entries divided by the number of buckets. When that factor gets high, and high actually is actually quite small, in fact. If it's two, is often the trigger for causing a rehash situation, so not even letting it get to be three or four. Just saying as soon as you realize that you have exceeded by double the capacity you intended, you planned for 100 things, you got 200 things, go ahead and redo your whole bucket array. So in the case, for example, of our simple like 0 through 7 case, right, maybe it's 0 through 6. One, two, three, four, so I have one that had six buckets, right, and then I've gotten to where each of them has two or maybe three in some and one in another. The plan is to just take this whole bucket array and enlarge it by a factor of two, so grow it, and in this case, from 6 to 12, and then rehash to move everybody. The idea is that the earlier decision about where to place them was based on knowing that there was exactly six buckets and so where it was placed before is not likely to be the place it will place if you have, you know, 12 choices to lay them down into.

And ideal – in fact you don't even want it to be that case. Like, if they all land into the same bucket, right, you would have the same clustering and then a big empty clump, and so you would rehash everything, run it through the new hash function having changed the number of buckets, and say, "Well, now where does it go in the hashing scheme?" And hopefully you'd end up getting a clean table, right, where you had 12 items now with one in each bucket, ready to kinda give constant performance through that and then potentially, again if you overload your load factor, go back in and rearrange stuff again. So using this strategy a little bit like the one we talked about with the binary search tree of just wait and see. Rather than if you got, you know, just because you get two items in a bucket it doesn't immediately cause any real alarm or crisis, it waits until there's kinda sufficiently clear demand that exceeds the capacity plan for, to do a big rearrangement. So you'd end up saying that adds, adds, adds would be fast, fast, fast, fast, fast.

Starting to get just a hair slower, right, having to do a search through a few things as opposed to one, and then every now and then, right, you'd see an add which caused a big reshuffle of the table, which would be a totally linear operation in the number of elements and then they'd be fast again for another big clump of inserts until that same expansion might be complete or triggered by it growing even more. But dividing it sorta by the total number of inserts, so if you had 1,000 inserts before you overloaded and then you had to copy all 1,000 things to get ready for the next thousand, if you average that out, it still ends up being called a constant operation. So that's pretty neat. It's a – really one of the, sort of, more, I think, easily understood and beautiful algorithmic techniques, right, for solving, right, this hard problem of search and retrieval that the vector and the sorting and the BST are all trying to get at, but – and sometimes the sorter, vector and the BST are solving actually a much harder problem, right, which is keeping a total ordering of all the

data and being able to flip it, traverse it and sorted order is one of the advantages that the sorted vector and the BST have that hash table doesn't have at all.

In fact, it's jumbled it all up, and so if you think about how the iterator for the map works, our map is actually implemented using a hash table, it actually just iterates over the link list, and that explains why it almost appears completely random. If you put a bunch of things in the table and you go to iterate and visit them again, that the order you visit the keys seems to make no sense, and that's because it's based on their hash codes, right, which aren't designed to be any real sensible ordering to anybody not in the know of how the hash function works, whereas iterating over a vector that's sorted or iterating over a BST or a, in this case, our set, for example, is backed by a binary research tree will give you that sorted order. So it solves this harder problem, right, which cause there to be more kinda more investment in that problem, whereas hashing solves just exactly the one problem, which is I want to be able to find exactly this value again and update this value, and nothing more is needed than just identifying this, not finding the things that are less than this.

For example, if you needed to find all the values that were less than this key alphabetically, right, the hash table makes that no easier than an unsorted vector, whereas things like assorted vector and binary search tree actually enable that kinda search, you could find the place in the tree and kind of work your way down the left side to find the things that were less than that. That's not actually being solved by the hash at all. Its use of memory is actually comparable to a binary search tree in that it has a four byte pointer per entry, which is the next field on the link list chain, and then there's a four byte pointer for each bucket, the head of the link list cell. Given that we expect that the buckets to be roughly equal to entry, than we can kinda just summarize that as well. Each entry represented an eight byte overhead, which is the same eight byte overhead, right, that the left and right pointers of the binary search tree add.

Does it have any degenerate cases? That's a good question. So one of the things that made the binary search tree a little bit of a heavy investment was that to get that balancing behavior, even when we're getting data coming in in a way that's causing a lopsided construction, we have to go out of our way to do something special. What is the degenerate case for hash? Is there something that caused it to behave really, really badly? Anyone wanna help me out with that? Is it always good? Does it always give a good spread? Can we end up with everything in the same bucket somehow?

Student:[Inaudible] repeated it that way?

Instructor (Julie Zelenski):So if you have repeated elements the way that – both versions of the map work is they overwrite. So actually the nice thing is, yeah, if you did it again you just take that same cell and overwrite it, so in fact we wouldn't get a clustering based on the same entry going in and out. But how'd we end up with everything being in the same bucket? Mostly comes out, if you look at your hash function, when would a hash function collide? Right, and if you have a dumb hash function, right, you can definitely have some degenerate cases.

My dumb hash function of return zero, right, being the worst example of that, but any hash function, right, that wasn't being particularly clever about using all of its information might actually have some clustering in it. It is possible, for example, even with a particularly good hash function that there are strings that will hash to the same thing, and it's like, if you somehow got really, really unlucky that you – and had a hash function that wasn't doing the best job possible that you could get some clustering, but in general it doesn't require – the sole responsibility, right, for the generate case comes down to your hash function. So as long as your hash function and your inputs match your expectation, you don't have any surprises about how they got inserted the way it was in a binary search tree, which is sometimes hard to control.

Student: Just as you could underestimate the number of buckets you need –

Instructor (Julie Zelenski):Yeah.

Student:– could you over estimate the number of buckets you need by a large amount –

Instructor (Julie Zelenski):Um hm.

Student:– that's wasting a lot of it –

Instructor (Julie Zelenski):Exactly.

Student:– memory, and do you then go through and take that memory back?

Instructor (Julie Zelenski):Yeah. Yeah, so that's a great question. So a lot of data structures don't shrink, right, and for example, the vector often will only grow on demand and then even if you deleted a bunch of, or removed a bunch of elements, it doesn't necessarily consider shrinking a big deal, and so that's often a – maybe it's a little bit lazy, but it turns out, right, that having a bunch of memory around you're not using doesn't have nearly the same penalty that not having the memory when you needed it and having to do a lot of extra work to find things.

So typically, right, you would try to pick a size that you are willing to commit to, and say, "Well, no matter what, I'll stay at this size. I won't get any smaller." But that, as you grow, you might not shrink back to that size. You might just let the capacity just kinda lay in waste. There are good reasons to actually be tidy about it, but again, some of this is hard to predict. It might be that your table kinda grows big and then a bunch of things get taken out, and then it grows big again. Like, maybe you have a new freshman class come in, and then you graduate a whole bunch of people. Then at the point where you graduate them all, you're about ready to take in a new freshman class and so it might be that in fact, right, the capacity you just got rid of, you're gonna plan on reusing in a minute anyway, and so maybe that clearing it out and totally releasing may actually be an exercise that was unimportant. You're planning on getting back to that size eventually, anyway. Most hash tables tend to grow is actually the truth. Right, you tend to be

collecting data in there and they tend to only enlarge. It's kind of unusual that you take out a bunch of entries you already put in.

And so I just put a little last note, which is like, okay, well I had said earlier that when we have the map, the key had to be string type and I was gonna at some point come back to this and talk about why that was the one restriction in all our template classes, right, you can put anything you want in statter or vector or stacker queue. That map lets you store any kind of value associated with that key, but that key was string type, and so given how our map is implemented as a hash table, that actually starts to make sense, that if you're gonna take the key and kind of jumble it up and map it to a bucket, you need to know something about how to extract information from the key, and so this case, knowing it's a string means you know it has characters, you know its length, and you can do those things.

If you wanted to make a map that could take any kinda key, maybe integers is key or student structures is key or, you know, doubles is key, any of these things, it's like, well, how can you write a generic form that could use a hashing operation on that kind of key and map it to bucket numbers. You would build a two type template to kinda get this working in C++, so you can actually have a template that has a type name for the key type, a type name for the value type, and then in the member functions you'd refer to both key type and value type as these two distinct place holders. And then when the client set it up, right, they'd be saying, "Okay, I want a map that has strings that map to integers." So maybe this is words and the page they appear in a document. I have a map of integers and a vector string. Maybe this is score on an exam and the names of the students who got that score, doing it that way, and to make this work, right, there has to be some kind of universal hashing that, given some generic type, can turn it into an integer in the range of buckets.

The – what it's gonna require here is that the client get involved. That you can not write this generic hash function that will work for all types, right, if it's a student structure, what are you gonna look at? The ID field, the names field, under like – there's just no sensible way that you can talk about a generic type and talk about how to hash it, so what you would have to do is have some sort of client call back, so probably you would create this by passing out a call back that's given a key type thing, and a number of buckets would do the necessary machinations, right, to hash it into a right number. And so that would be kinda one of those coordination things between the client and the implementer about the client knows what the data is what the data is it's trying to store, and kinda how to mash it up and then the map would know, okay, given that number, where to stick it.

So that's what it would take, and then rather sorta load that onto our novice heads, we went ahead just said, "Okay, it'll always be a string, so that we can do a hash internally without having to get involved." All right, any questions about hashing? I'm gonna give you like, a two second talk about set because it's the one ADT I never said, "Well how does it work? How does it work? What do we do?" We're talking about, it doesn't add any new things that we haven't already done, so in fact I'm just gonna tell you what it does, and then your picture will be complete, right. It wants to do fast searching, fast

updating, add and remove, and it also has all these high level operations, such as intersect, union and difference, that are kind of its primary set of things to do. A bunch of the things we've already seen, right, would actually be a reasonable strategy for upholding stat, right, using some kind of vector or array. Most likely you'd want to keep it in sorted order because that would buy you the fast lookup for contains, but the add and remove, right, would necessarily be slow in those cases because of the continuous memory.

The link list, not really probably too good of an option here because it contains then becomes linear and add and remove similarly, right, liner, so it doesn't actually get you anywhere, in fact it just adds the memory in pointers and gunk like that. The binary search tree, probably a pretty good call, right, that's the – kinda meshes the advantage of Adamic structure with the sorted property to be able to give it fast update adding and removing and searching all can be done in logarithmic time if you have balancing. And then it actually also enables the high level ops, so the idea of being able to do a union intersection difference, and actually being able to walk over both of the trees in sorted order, which is very easily done with a [inaudible] reversal, makes those operations actually more efficient to implement than actually kinda having to deal with data coming at you all different ways.

It turns out hashing really won't quite do it very easily for us because of the same need about the template situations, like, well you have to have a hash function that can hash these things and do the right thing, and so the tree happens to be a good of just – if the client can just give you ordering information, you can do all the hard work without pushing hashing onto them. So in fact the way our set really does work is it does use a binary search tree. It happens to use it through another layered abstraction that there actually is a BST class that we've never really encountered directly, but the BST class just takes the idea of a binary search tree, adds balancing, and all the properties about finding and inserting and adding, and packages it up, and then set actually just turns out to be one liners making calls into the binary search tree. This is where all the heavy lifting goes, is down there.

That kinda completes the big picture, so some of stuff, right, so our map uses our hash, right, our BST uses – or set uses the BST, which is the binary search tree, right, our vector, right, using an array, and then that stacks and queues having both viable strategies both for array vector based backing as well as link list both getting good time performance. And then you saw with the PQ, right, even yet another structure, the heap, right, which is kinda a variant of the tree, and so those tend to be the kind of really classic things in which a lot of things get built, and so having a chance to use them both as a client and kinda dig around as an implementer kinda gives you this big picture of some of the more common ways that cool data structures get built. So that's all for Friday. We're gonna do some more good stuff next week. I'm gonna go to the Terman café in a little bit, if you've got some time, please come, and have a good weekend. Work hard on pathfinder.

[End of Audio]

Duration: 52 minutes