

Hashes

Gabe Johnson

Applied Data Structures & Algorithms

University of Colorado-Boulder

Linked lists have an $O(n)$ complexity to find an item. Binary search trees do a lot better, with an $O(\log n)$ search. With those data structures, we had some concept of ordering, there's a first and last item, and the items are arranged in some way that is useful.

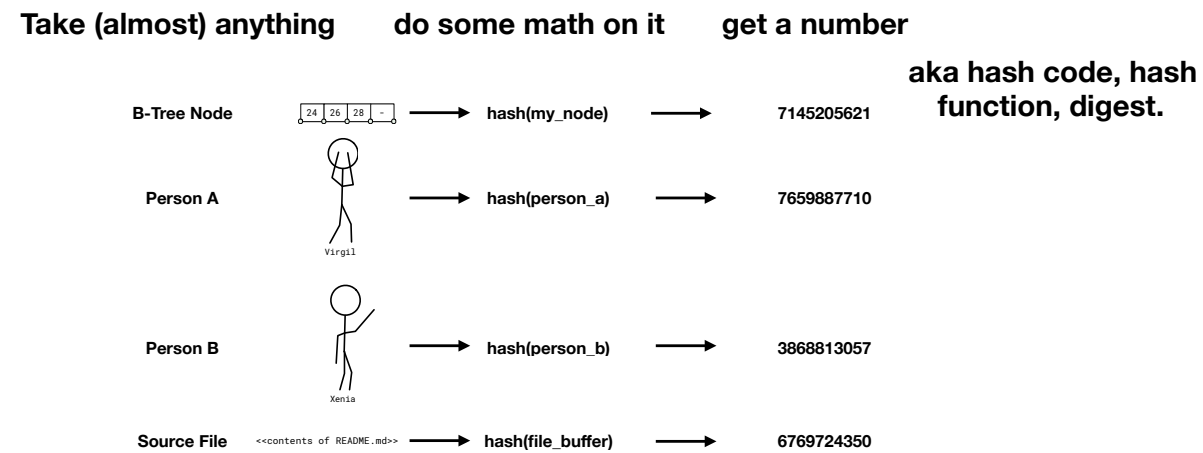
We've also talked about the Map ADT, which lets you associate keys and values, but there's no need to maintain any order between the elements.

And if we can get away with using a Map ADT, we can get really good time complexity with a data structure called a hash table. With hash tables, the time complexity for insert, remove, and search is all constant, $O(1)$. But before we cover hash tables, we need to understand the critical ingredient that makes them work, namely `__hash` functions.

Episode 1

Hash Functions: Overview

Hash Functions



Super high level, a hash function is a bit of math that is used to take some large input and compute a much smaller output. The input could be data like a pointer to a b-tree node, a user model in your website, or a text file. Whatever you want. The idea is to take any data, and output something much smaller and more manageable.

Sometimes this is called "reducing the dimensionality" of the data.

<next build>


The output is called alternately a hash, hash value, hash code, a digest, and several others also. I'm going to try calling them hash _codes_, since if I just say 'hash' you might think I'm talking about the hash _function_.

3 Parts to Hashing



Input: likely large

e.g. collected works of Shakespeare (5.5 M)

→ md5() → a810f89e9f8e213aebd06b9f8c5157d8

Hash Function

Using md5 here, but there are many others as well.

Output: small

e.g. 120 bits

There are three parts to the process of hashing. You first need an input, this is the works of Shakespeare. It could be anything, as long as there is a way to precisely encode it as data. Huge data values are OK, you could hash gigabytes of data. The Shakespeare I downloaded from Project Gutenberg is about five and a half megabytes.

Then there's the hash function, the math that churns your big thing into a little thing. There are lots of kinds of hash functions, and depending on what you want, there are different ways to write them. We'll cover this in detail later. One hashing algorithm is called the `MD5`, which is "message digest version 5". I use it here because I happen to have a command line utility for it on my Mac.

Then there's the output, the hash code. The hash function should be written to always produce the same hash code for the same input. The hash code might be tiny, like a single byte, or it might be maybe 512 bytes long. When I run the Shakespeare collection through `md5`, I get that code up there, a810 and so on.

Properties of Hash Functions

- **Same input —> same output**
- **Different inputs —> same output? (Collisions!)**
- **Similar inputs —> similar outputs?**
- **Given output —> reverse engineer input?**

Hash functions have a few general properties. I'll just read these off, and then we'll go into some detail on each.

- * They are supposed to consistently produce same output with same input
- * Does it produce the same output for different inputs? (They're called collisions)
- * Do similar inputs produce similar outputs?
- * Difficulty to reverse engineer the input if you have the output

Same Input, Same Output

hash(a) always yields x

Even on Thursdays

And full moons

Output only depends on input.

This is the main consistent property of hash functions. For any input, your hash function should always produce the same output. If you get different input, even slightly, you want the output to be different.

This means the hash function is a function in the mathematical sense. The output is only determined by the input. So you can't use a source of randomness, for example.

Collisions

hash(a) always yields x

hash(b) always yields z

usually: $x \neq z$

sometimes: $x = z$ Collision!

So we know that a hash function always produces the same output for the same input.

<next build>

And usually, the outputs aren't the same.

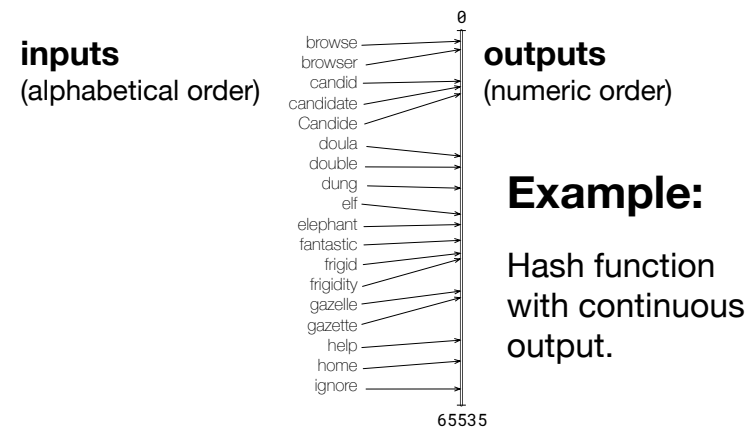
<next build>

But what if the hash function has two distinct inputs but produces the same output?

<next build>

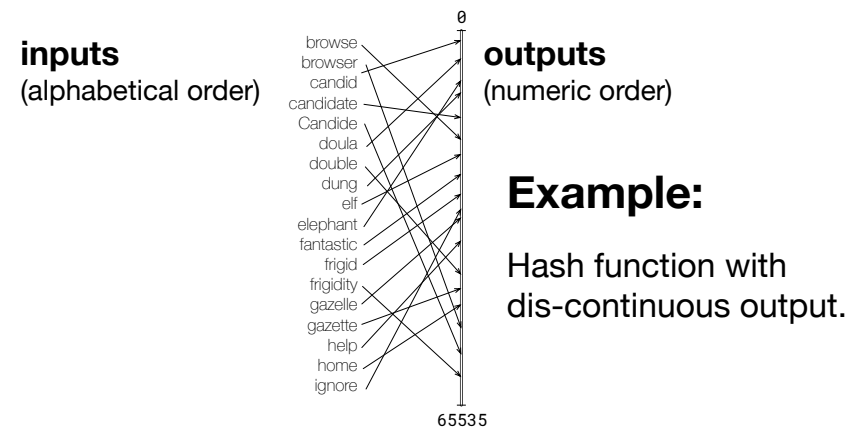
That's called a `_collision_`, and that's generally considered `_bad_`. But it is also inevitable that you'll have collisions, especially if your hash function's range is small. We'll see how data structures can handle collisions later on.

Continuity



In some applications it is a good thing for similar inputs to hash to similar outputs. For example if you're hashing student essays and your hash function outputs similar outputs if the inputs are similar, then you might be able to use that to detect plagiarism. Or if you're hashing sound files and the hash codes with similar sound profiles are similar to one another, you can use that as a way to 'sort sound'. These hash functions are called 'continuous' because similar inputs produce similar outputs.

(dis) Continuity



But usually we want the hash function to produce wildly different outputs even for nearly-identical inputs. This is useful for the hash table data structure that we'll get into great detail later. And if you're using any sort of cryptographic hash, it should be as dis-continuous as possible, otherwise you're inviting attacks.

There's a phrase, a 'perfect hash function', which is a hash function that gives unique hashes for all the elements you're working with. This is sort of a unicorn-style ideal, and for it to be practically possible, your universe of inputs has to be pretty small.

Kinds of Hash Functions

Hash Functions:

- for keeping similar keys clustered together
- for keeping similar keys spread apart

Related concepts:

- checksums: for detecting transmission errors
- fingerprints: for giving an ID to some large data block

So we have some hash functions that strive to put similar inputs next to each other with their hash codes, and that might be good for classifying, sorting, or some other kind of arranging of data.

Then we have other hash functions, the kind that we'll do for the homework, that strive to keep keys spread apart over the full possible range.

There are a few concepts that are related to hash functions, including checksums and digital fingerprinting. The internet seems to disagree with me on this one, but they seem to fit the definition of hash functions. The difference in the terminology seems to be related to how they're used, and what their design goals are.

Checksums are used to detect errors in transmission from one location to another.

Digital fingerprints are used similarly. The idea there is that any two blocks of data should have the same fingerprint.

Cryptographic Hash Functions

Ideal Crypto Hashes:

- Use full range of output (e.g. all 160 bits)
- Never gives collisions
- Impossible to derive input based on output
- Slightest difference in input gives completely different output


Example Algorithms:

- SHA-3 (Secure Hash Algorithm Version 3)
- MD6 (Message Digest Version 6)

Then there's a class of hash functions that are used in the security and crypto world, called cryptographic hash functions. The ultimate goal of a crypto hash function is to be completely uniform in its output, never has collisions, gives no ability to reverse engineer the input given the output, and the slightest variation in inputs gives completely different outputs.

Hash Codes

Often fixed size with a set number of bits. E.g.

md5() → **a810f89e9f8e213aebd06b9f8c5157d8**

That's a hexadecimal (base 16) representation of md5's 128-bit output. In binary, that is:

10101000000100001111100010011110100111110001110001000010011101011101011110100000110101110011111000110001010001010101111011000

With 128 bits, there are 2^{128} possible hash codes with md5. That's a lot.

Vulnerabilities in md5 have been found so it is no longer recommended for crypto use.

Many languages use 32 or 64-bit integers as hash codes. Good enough for hash tables!

I haven't said much about the outputs, the hash codes.

The hash code is usually a fixed-size output, often expressed in bits instead of bytes. My Shakespeare hash from above was ``a810f89e9f8e213aebd06b9f8c5157d8``, which prints out as 32 characters at the terminal, but the actual hash function produced a 128-bit hash code.

<next build>

This means md5 hash codes have 2^{128} possible values, and the algorithm does a good job of hitting them with decent uniformity.

<next build>

But unfortunately md5 has been proven insecure because there are regularities. They're just subtle enough for the crypto math folks to identify, and exploit.

So MD5 isn't any good as a cryptographic hash, but it is still OK for fingerprinting!

<next build>

In many languages, the standard hash code is an integer, either 32 or 64 bits, and that's probably good enough to use as the starting point for a hash-based data structure like a hash table or a hash set.

Episode 2

Hash Functions: Implementing

Now that we have the basic anatomy of hash functions, we can write our own. We'll start simple, writing a continuous hashing function for the English letters a to z, and then move on to a discontinuous hashing function called ``djb2``. And for the hash codes to make any sense, I think I'll first introduce hash tables, just to give you some context. A later episode will cover that data structure in more detail.

Maps In Action

Keys and Values (rather than numeric indices)

```
// map key 'richard starkey' to value 'Liverpool'
town["richard starkey"] = "Liverpool"

// this prints "Ringo lives in Liverpool"
print "Ringo lives in " + town["richard starkey"]

// can change values using the key
town["richard starkey"] = "Los Angeles"

// this prints "Ringo lives in Los Angeles"
print "Ringo lives in " + town["richard starkey"]
```

A hash table implements the Map abstract data type. That's the one where we can associate some data called a key with some other data called the value. So we can then map the key to the data.

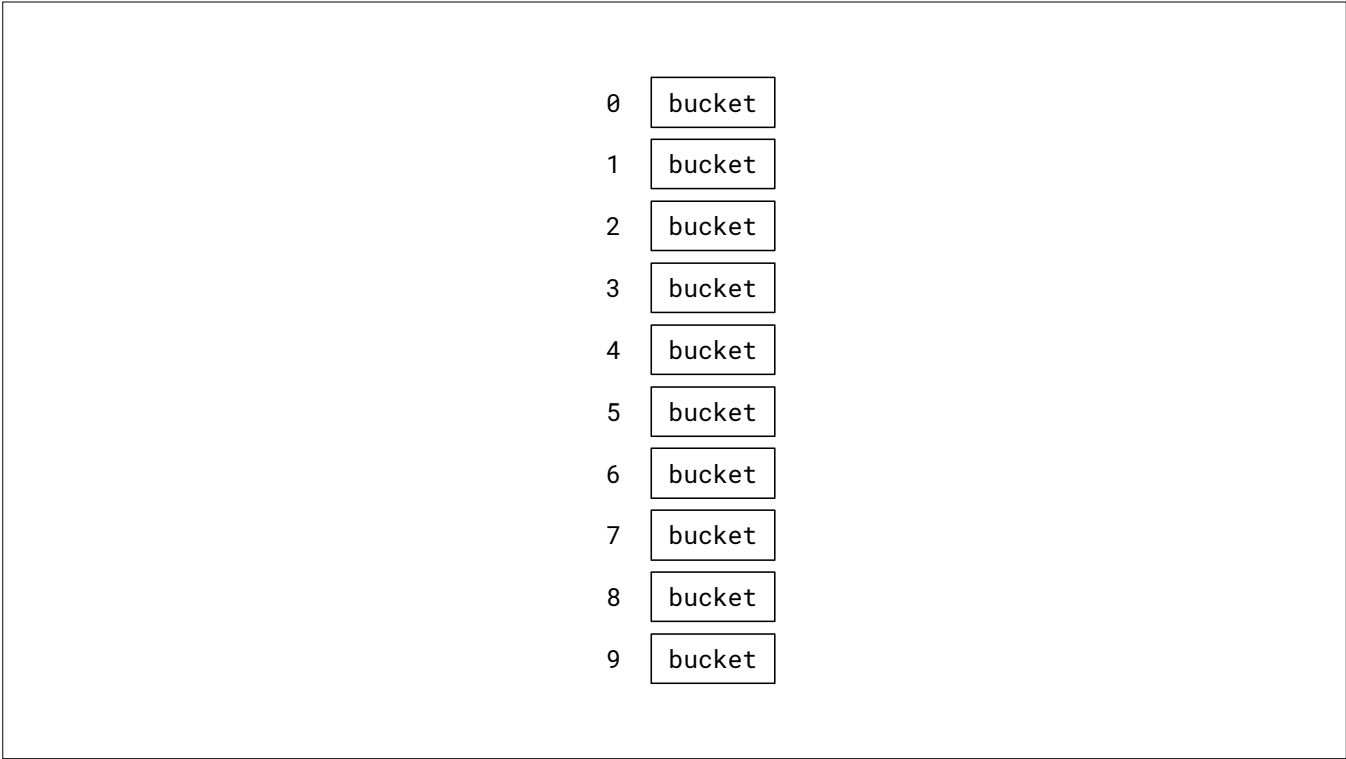
The behavior is sort of like lists, but instead of indexing with a number, we're indexing with arbitrary data, someone's name in this case.

And there's no concept of ordering in a basic Map.

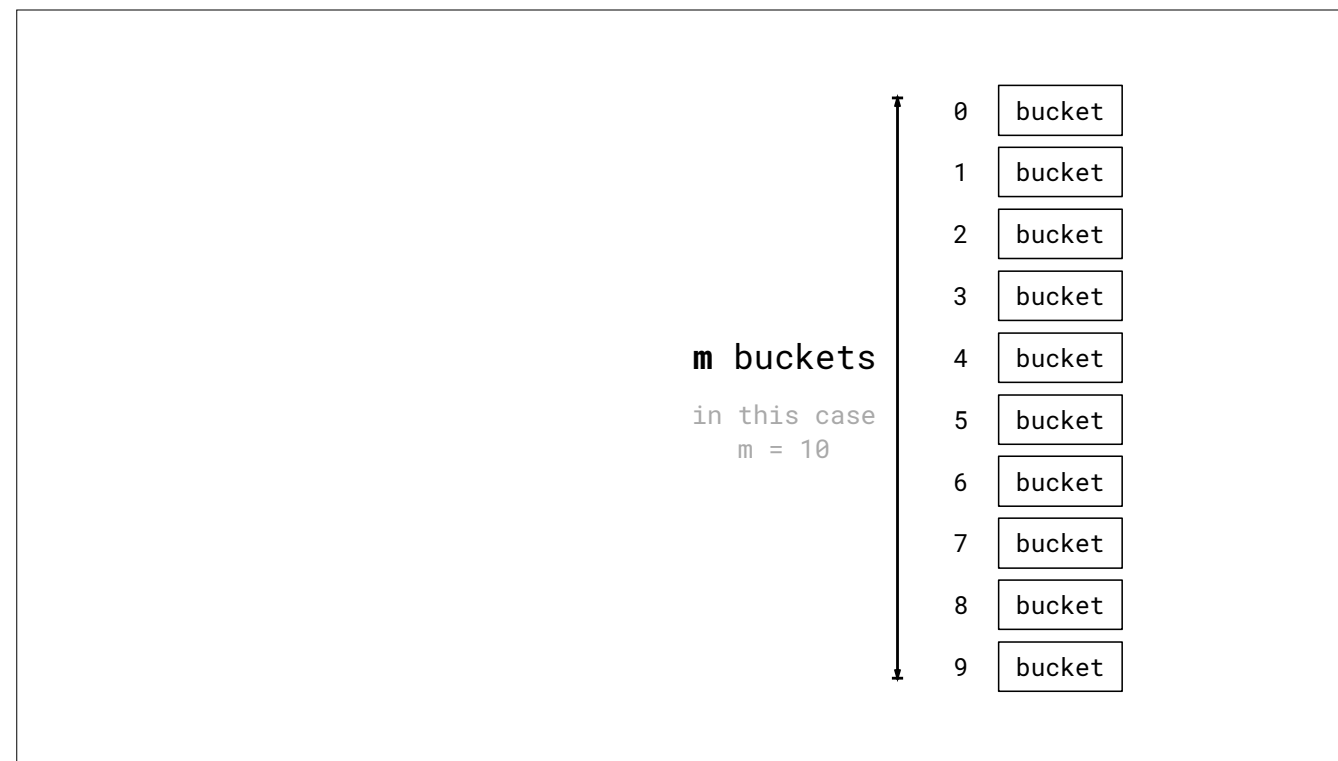
Operations are $O(1)$

(Unless the hash function is awful)

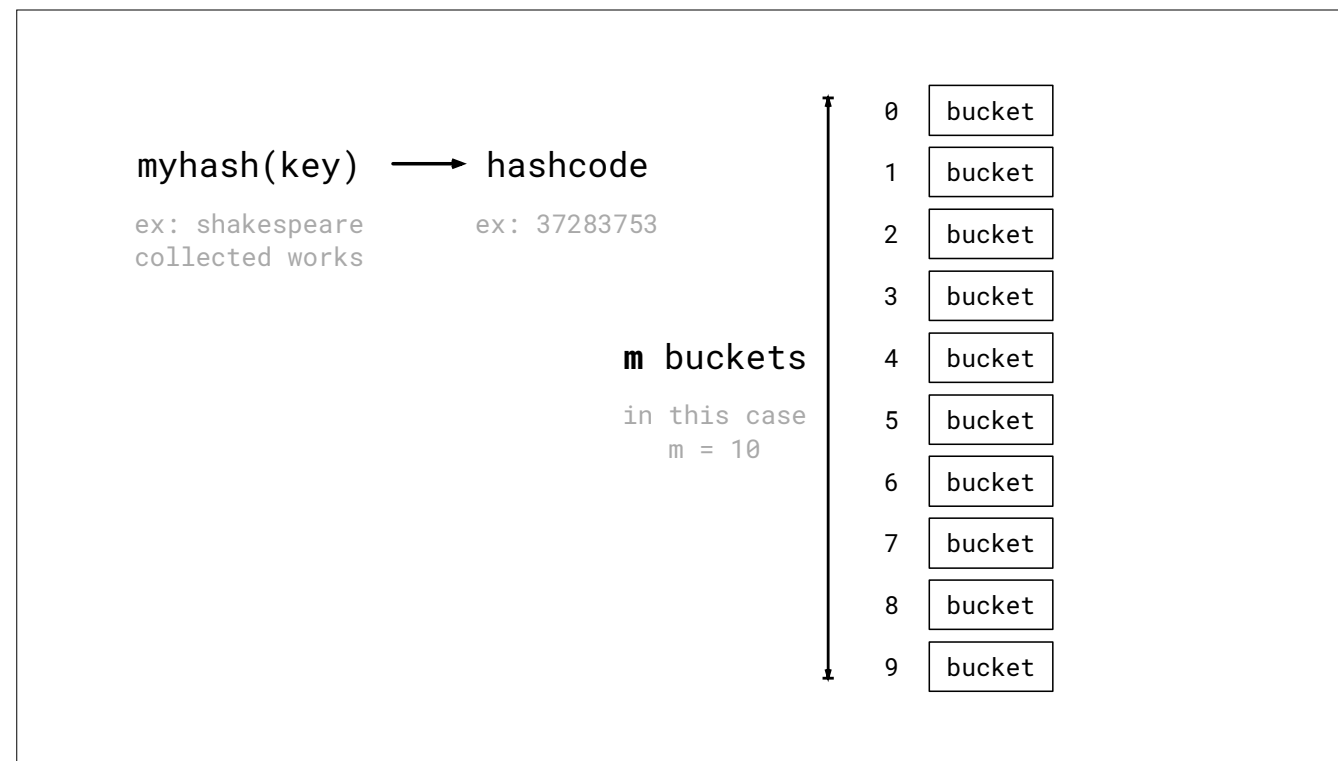
Assuming the hash function is reasonably OK, Hash Table operations are $O(1)$ - constant time complexity. That makes them much better than linked lists or binary trees, if the use case gives you the opportunity to use them.



A hash table has an array of buckets.

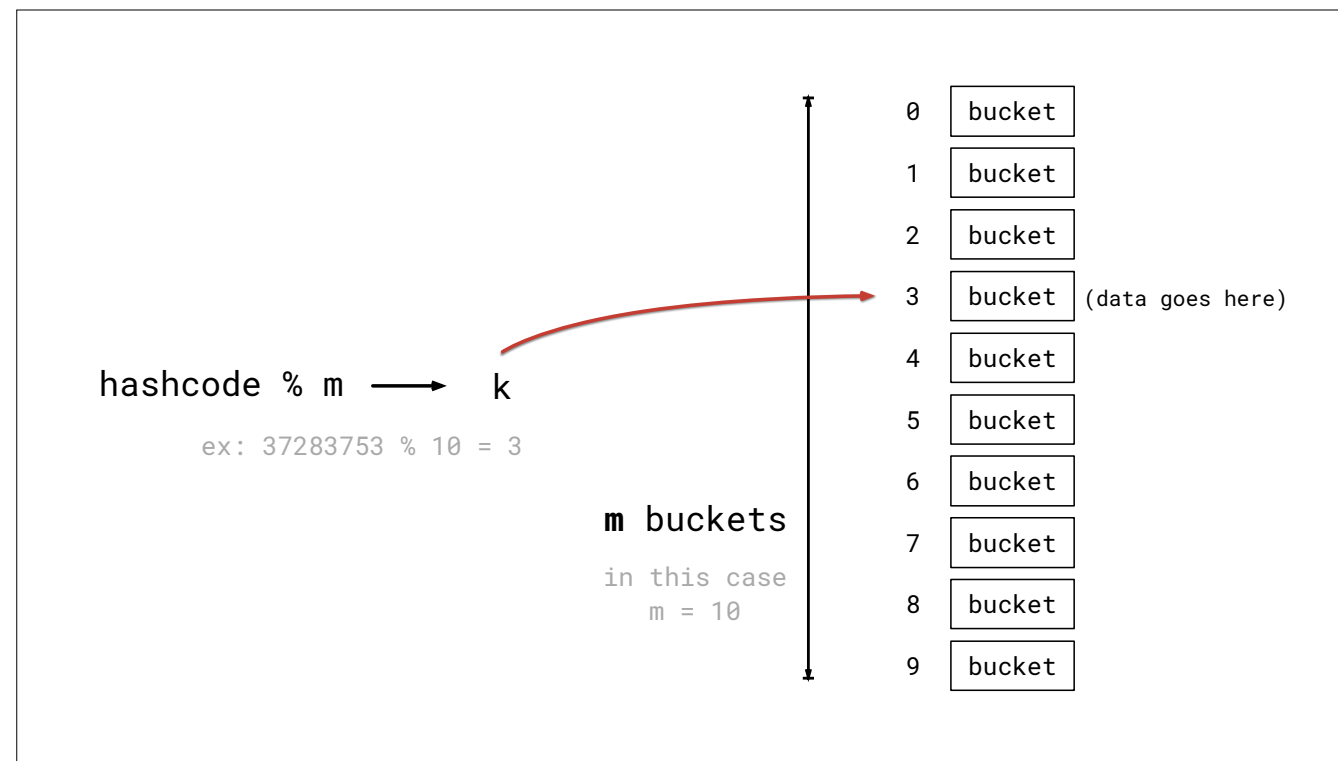


The number of buckets can be represented by a number like m . In this example there are 10 buckets.



When we want to stick a new key/value pair into our hash table, we use a hash function on the key. The key could be something short, like a person's name, or it could be something big, like the collected works of Shakespeare. Whatever that is, the hash function produces a hashcode, something typically smaller than the input, but still likely to be much larger than the number of buckets.

So, how do we use this hashcode to find which bucket?



There are a few ways to do this. A common way is to use the modulo operation. Take your hashcode, then use modulo m . This will give you a number from zero up to $m-1$. In the example here, we happen to get three, so we're going to put our data into the bucket array at index three.

You're probably already thinking, what happens when something else maps to that bucket? That's called a collision, and we'll spend an episode talking about collision mitigation strategies. But for now, the point is that our hash function gives us the hash code, which is then used by another sort-of-hash function, modulo in this case, to determine which bucket to look in.

Simple Continuous Hash

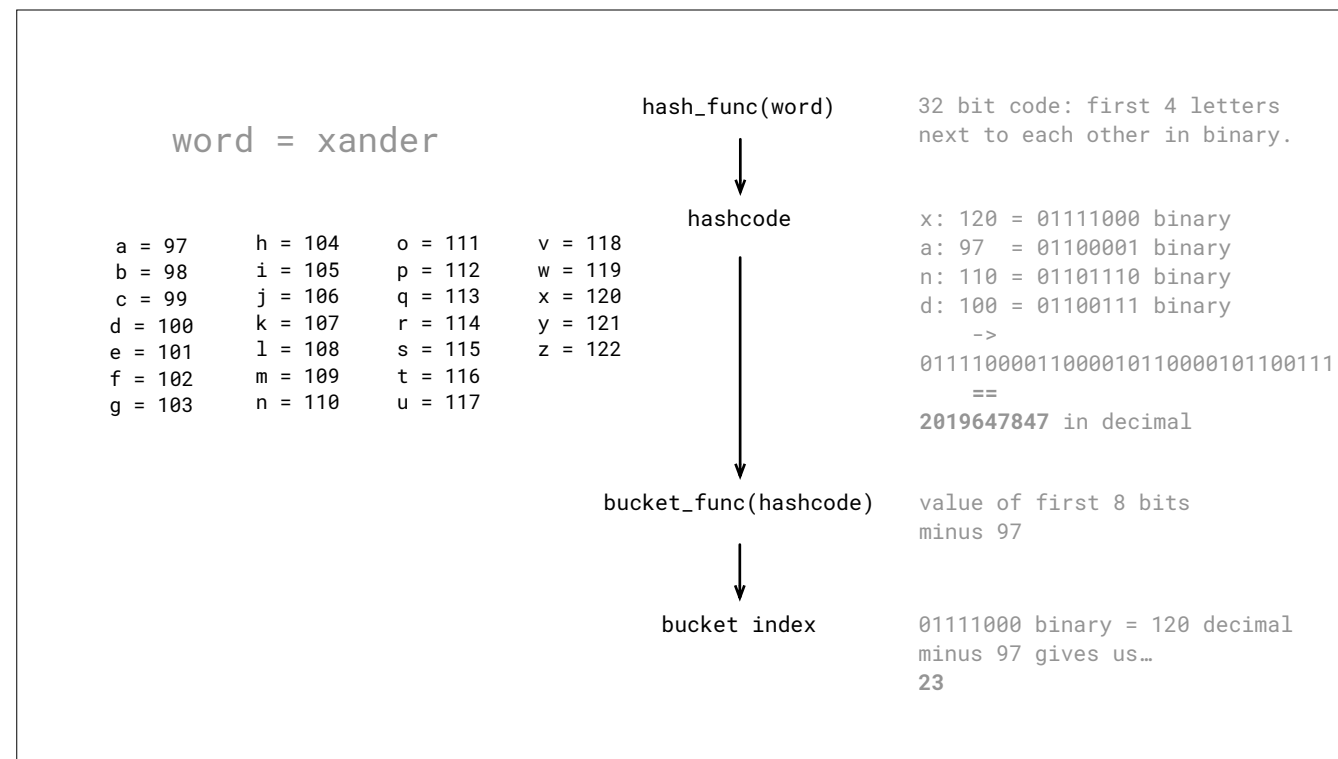
a = 97	h = 104	o = 111	v = 118
b = 98	i = 105	p = 112	w = 119
c = 99	j = 106	q = 113	x = 120
d = 100	k = 107	r = 114	y = 121
e = 101	l = 108	s = 115	z = 122
f = 102	m = 109	t = 116	
g = 103	n = 110	u = 117	

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p
16	q
17	r
18	s
19	t
20	u
21	v
22	w
23	x
24	y
25	z

Let's do a concrete example. Let's say we're going to write a hash function to store English words in a hash table, and we want the hash codes to let us keep words in alphabetical order. For example's sake, we'll make a 26-bucket hashtable, one for each letter of the alphabet.

<next>

Let's say our input consists entirely out of lower case letters a through z. In that case we can just use the ASCII encoding scheme, which just maps letters to a numbers. Turns out a to z is just 97 through 122.



OK, so let's write a little hash function that will produce hash codes that are in the same alphabetic order as the input. For this example, I'll hash the name 'xander', because it's a cool name. <next>

We want the hash code to be an integer with fixed size, let's say 32 bits, which is 4 bytes. Our hash function will take an arbitrary word and produce a 4-byte integer. To make things easy, we can just convert each letter into a byte. Since want a 4-byte hash code, let's use the first four letters from the word. And if the word is too short, maybe we just add zeros to the end. There are already problems with this scheme, and I hope you can start spotting them as I keep going with this example. So let's take compute the full 4-byte hash code. <next>

We're taking the first 4 letters of each word, so our input effectively gets truncated to 'x-a-n-d'. You can read the decimal versions of each letter on the left. Take each of those decimals into binary, and pad with zeros to the left so they are each one byte. To get the full hashcode, you just put each byte next to each other to give you that bit string. And if you interpret that in decimal, you have this number, two billion, nineteen million, and other digits as well. Ok, so now we have a hash code for this word. Now how do we use it? Remember earlier I mentioned that we can use another function to compute a bucket index based on a hashcode? <next>

We're going to make another function that does this, and for lack of a better term, I'm going to call it the bucket function, and it will give us a bucket index. We can map our hash codes to buckets for all 26 letters by looking at just the first byte. That will put all the words that start with the letter A into the first bucket, B-words into the second bucket, and so on. <next>

Looking at the first byte of our hash code, we see it is 120. Now, we want to map the numbers 97 through 122 into bucket indices 0 through 25, so if we just subtract 97, that should work. 120 minus 97 is 23. So we're going to put the Xander into bucket 23.

0	a	
1	b	
2	c	
3	d	
4	e	
5	f	
6	g	
7	h	
8	i	
9	j	
10	k	
11	l	
12	m	
13	n	
14	o	
15	p	
16	q	
17	r	
18	s	
19	t	
20	u	
21	v	
22	w	
23	x	X-Bucket
24	y	
25	z	

Bucket index collisions: Words that start with X but are different in the next 3 letters can all co-exist in the X-Bucket, because their hashes differ. Only the bucket index is the same.

Hash code collisions: Words that start with ‘x-a-n-d’ all have the same hash code, so can only coexist in this table if we have additional information available.

... and you can see that's the X bucket.

If I add any other X-words like Xeric or Xenon, they will have different hash codes, but they will all try to fit into the same bucket at index 23. That's a collision, and we'll talk about ways to handle that later on.

If I add any other X-words that start with X-A-N-D like Xandertastic, even if they're made up, they'll end up with exactly the same hash code. Our hash codes only depend on the first four letters. When distinct inputs lead to the same hash code, that's also called a collision, though it is a different kind of collision from the bucket-collision. This is a much worse kind of collision, because our hash table won't be able to resolve the problem without bringing in additional information.

Bernstein Hash Function

```
unsigned int djb2(string key) {  
    unsigned int hash = 5381;  
    for (size_t i=0; i < key.length(); i++) {  
        char c = key[i];  
        hash = ((hash << 5) + hash) + c;  
    }  
    return hash;  
}
```

This is a reasonably OK hash function. Not problem-free, but good enough for many purposes.

Your mileage may vary.

I'm going to show you a different function now, and this one isn't just a silly classroom example.

This is called the Bernstein function, often called the DJB2 function. It is a discontinuous hash function typically used for hashing strings, but I and other people have used it to hash other kinds of data, as long as you can get your data into a byte array.

I present it here not because it is the best hash function ever, but it is good go-to function that doesn't have a lot of collisions, though the spread isn't super duper good it is good enough for many purposes.

What this does is incrementally update the hash code based on the previous version of the hash code, and the next byte.

To explain superficially that math:

Bernstein Hash Function


```
unsigned int djb2(string key) {  
    unsigned int hash = 5381;  
    for (size_t i=0; i < key.length(); i++) {  
        char c = key[i];  
        hash = ((hash << 5) + hash) + c;  
    }  
    return hash;  
}
```


<<
bit-shift to the left

The two less than signs `<<` is a bit-shift to the left. Here we're shifting the bits of the current hash code value to the left by five bits. When you do this, the bits on the left fall off, and on the right, zeros appear.

Bernstein Hash Function

```
unsigned int djb2(string key) {  
    unsigned int hash = 5381;  
    for (size_t i=0; i < key.length(); i++) {  
        char c = key[i];  
        hash = ((hash << 5) + hash) + c;  
    }  
    return hash;  
}
```

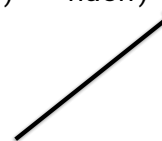


Add current value of hash to the left-shifted version.

Then it adds the current version of the hash to the left-shifted version, which is basically shuffling the bitstring.

Bernstein Hash Function

```
unsigned int djb2(string key) {  
    unsigned int hash = 5381;  
    for (size_t i=0; i < key.length(); i++) {  
        char c = key[i];  
        hash = ((hash << 5) + hash) + c;  
    }  
    return hash;  
}
```



Add current byte from the input string to the hash.

Last, add the current byte from the input string to the hash. This shuffles the last eight bits of the resulting hash, since bytes are only eight bits.

Bernstein Output

Hashing string "xander"

```
000000000000000001010100000101 5381
00000000000000101011011000011101 177693
00000000010110010111101000011110 5863966
00001011100010001011111001001100 193510988
01111100101000001000100000110000 2090895408
00010000101100011000111010010101 280071829
00100110111000110110000110100111 652435879
```

Hashed string "xander" to 652435879

Hashing string "xandertastic"

```
00000000000000000001010100000101 5381
00000000000000101011011000011101 177693
00000000010110010111101000011110 5863966
00001011100010001011111001001100 193510988
01111100101000001000100000110000 2090895408
00010000101100011000111010010101 280071829
00100110111000110110000110100111 652435879
00000011010011111001011011111011 55547643
01101101010000100111011010111100 1833072316
00010101100100010100111010101111 361844399
11000111101110110010010100000011 3350930691
10111111000111111100010111001100 3206530508
1010001100011000011111110101111 2736291759
```

Hashed string "xandertastic" to 2736291759

Here are a couple of words, xander and xandertastic, run through the DJB2 hash, printing out every version of the hash. Notice that xandertastic hashes

<next build>

exactly the same as xander, for the first six letters, as you'd expect.

Other Hash Functions

Codeproject discussion of hashing functions:

<https://goo.gl/XVggft>

Lots of examples. Check out MurmurHash2

There are many hash functions, built for many different purposes with different assumptions about the input data. There is a good article about it at this URL.

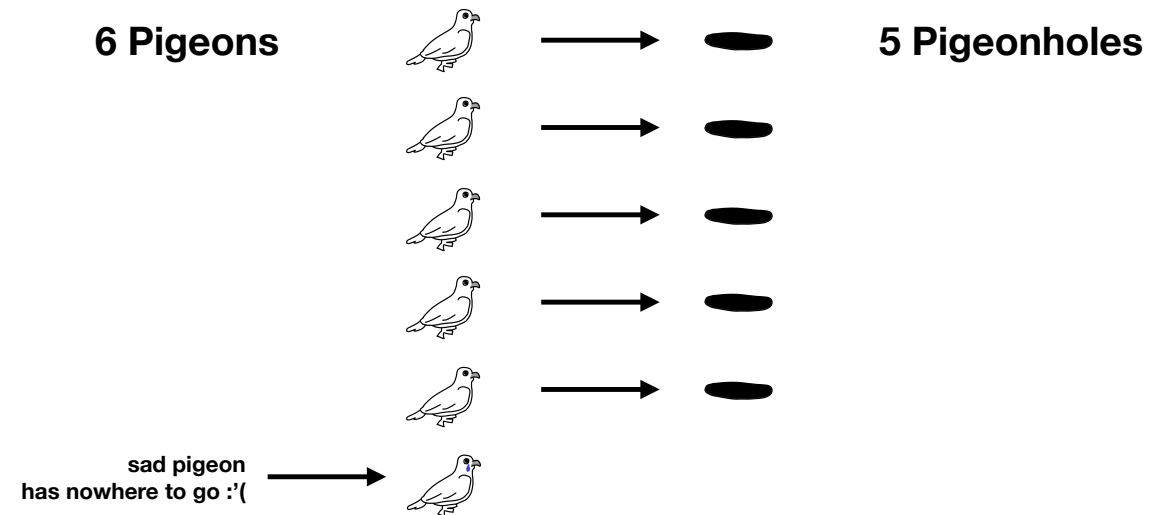
There is source code for some of the hash functions, including one called MurmurHash.

Episode 3

Collision Mitigation

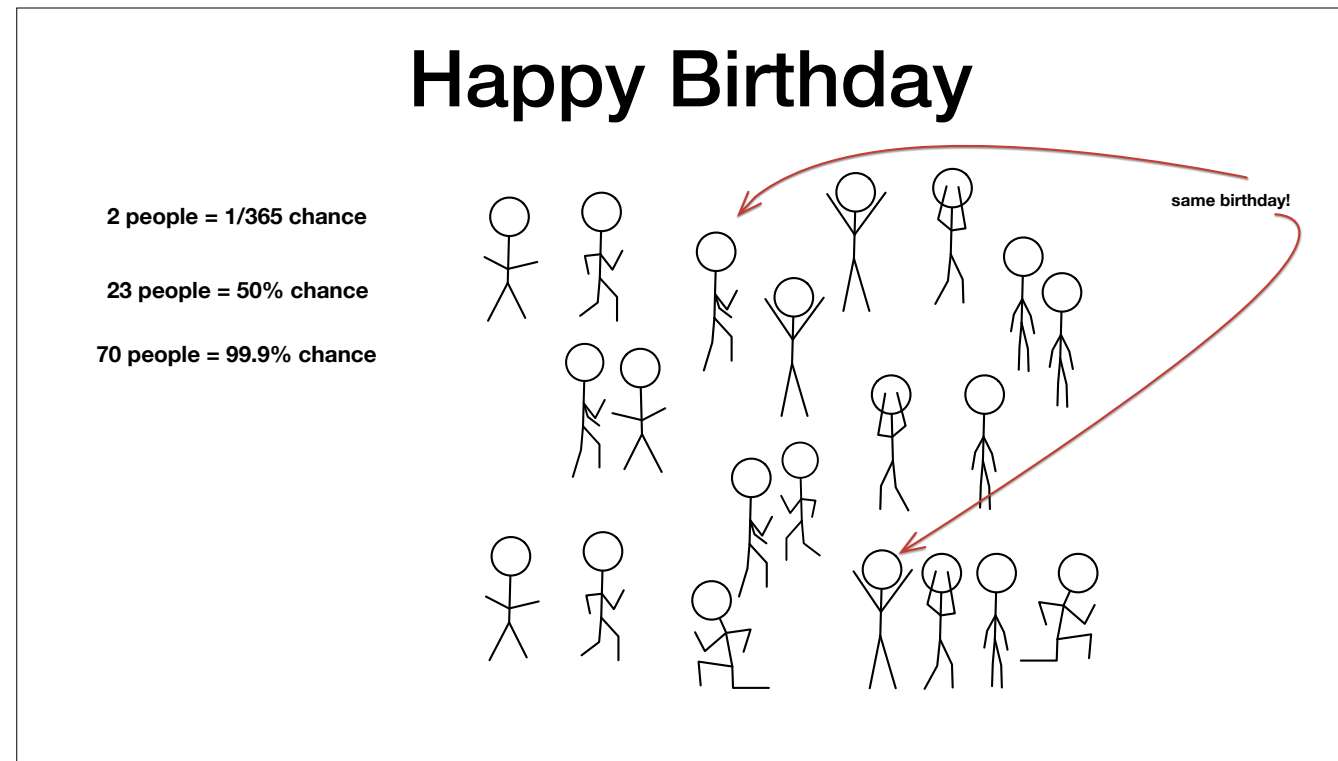
Since we've started this sequence, I've done this hand-waving thing when two different inputs might occasionally hash to the same value. It's called a collision, I've said, and I'll get to it later. Well, enough hand-waving! In this episode and the next we'll talk about ways to handle hash collisions.

Pigeonhole Principle



The Pigeonhole principle says that if you have 5 spots and 6 things to file away, you're going to be one spot short. Well, the basic idea with hash-based data structures is to try to fit some potentially gigantic number of things into a substantially smaller number of spots, and attempting to over-fill any spot is a collision. This is an overly fancy way of saying you're definitely going to have collisions if you have more pigeons than pigeonholes. Not to mention sad pigeons.

So if we have more pigeonholes than pigeons, what are the odds that we're going to have a collision?



To give you a sense of how common hash code collisions are, let's talk about a related thing, the Birthday Problem.

Say you've got a room full of people. What are the odds that there's at least one pair of people who share the same birthday?

<next build>

If it's just two people, the odds of that are one in 365, since there are 365 days in a year.

As you start filling the room with more and more people, the odds that some pair of people with the same birthday goes up surprisingly fast.

<next build>

With 23 people in the room, there's a 50% chance that there's a birthday pair,

<next build>

and with 70 people, the odds are 99.9%.

Happy Hashday?

Possible Birthdays = Hashtable Buckets

People = Hashtable Keys

Example:

1,000,000 buckets

2,450 keys

95% chance of collision

To relate this back to hash code collisions. Rather than days in the year, we have the number of possible hash codes, and instead of people in the room, we have the number of keys we're putting into our hash table.

<next>

I got this one off Wikipedia. If there's a million possible buckets and 2450 keys, there's a 95% chance that there will be at least one collision.

This is all about probabilities. But even if there was a 1% chance of a collision, and if we didn't have a collision mitigation strategy, our hash-based datastructure would be lossy, or its behavior undefined. And that would be catastrophic.

Collision Types

1. Hash code collisions
2. Bucket collision

There's one kind of collision where our hash function gives the same hash code for different inputs. I'll call that hash code collision.

Then there's another kind of collision where our bucketing strategy wants to put two different keys into the same bucket. I'll call that bucket code collision, and that's what we're going to handle now.

Resolving Collisions

1. Chaining (now)
2. Open Addressing (next)

homework = open addressing

There are two main ways to deal with bucket code collisions.

The first is called chaining, where each bucket in the hash table contains another data structure, probably a linked list, that chains together all the keys that were placed in that bucket.

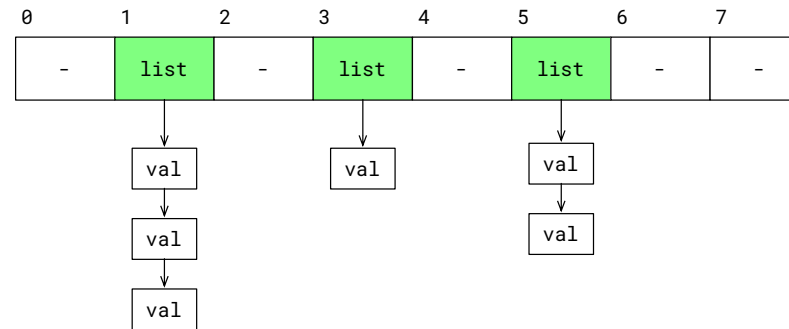
The second is called open addressing, where items are all stored in the hash table directly, without separate data structures. In this case, if a bucket is full, the data you're looking for (or inserting to) is nearby, according to some rules.

We're using open addressing for the homework assignment.

I'll go over both of these in turn---chaining in this episode, and then open addressing in the next.

Chaining

**Chaining
with Linked Lists**

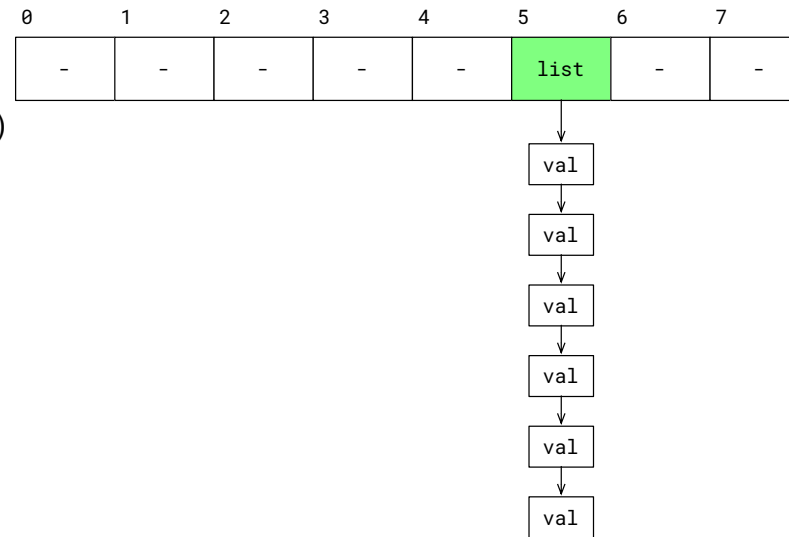


The idea with chaining is pretty straight forward. If our hash table only has, say, 8 slots, you'll get bucket collisions soon. In this 8-bucket hashtable, two (or more) keys that bucket-hash to the same bucket are put into some other data structure.

Linked lists are the most common, but you really could use any number of data structures there.

Chaining: Worst Case

**Chaining
with Linked Lists**
(nightmare scenario)



In the absolutely worst-case scenario, you'll have a hashtable with one lucky bucket that has all the data in it due to a bad hash function. In that case, interacting with the hashtable is no better than dealing with the backing data structure in the bucket.

But usually you don't have pathological situations like that, as long as your hash function is appropriate for your keys.

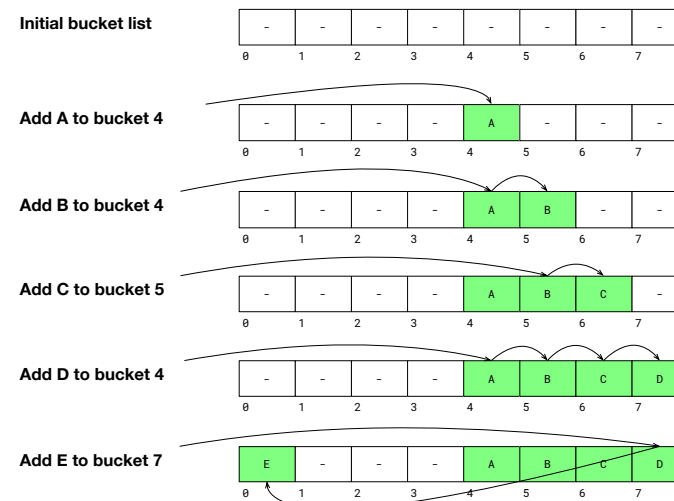
Episode 4

Open Addressing

We just saw that a hashtable can put a linked list in each bucket slot in order to resolve collisions. But if you have a bad hash function, you're not going to have wonderful performance. So there's another way of mitigating collisions, called Open Addressing.

With open addressing, the hash table stores everything in the hash table's bucket array itself. There are a bunch of esoteric methods for doing this. A straightforward method called linear probing. There are also quadratic probing, and double hashing. All these methods do is determine where to look when a bucket is occupied.

Linear Probing: Insert



When we want to put A into bucket four, and that bucket is available, we can store it without any hassle. **<next>**

But then we want to put B into bucket four, but it is full with A. With linear probing, we just go to the next bucket. **<next>**

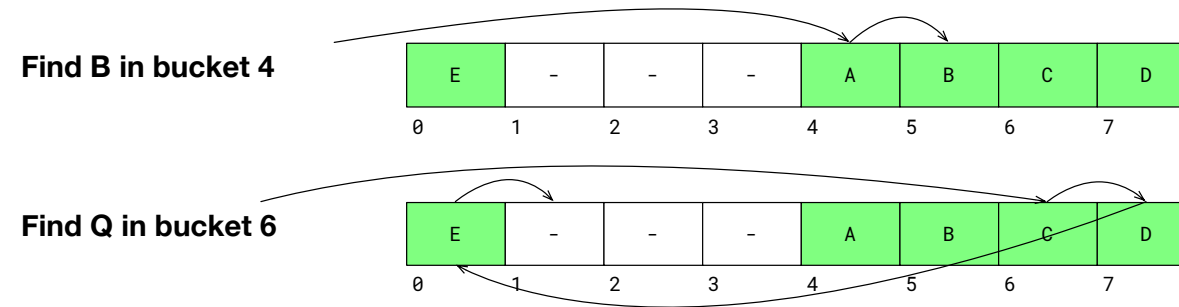
This continues, so if we want to be C into, say, bucket 5, and it is full, we end up putting C into bucket six. **<next>**

Try putting D into bucket four, we notice that four, five, and six are all full, and the next available is bucket seven. **<next>**

Happily the world is not flat - if E wants to go into bucket seven, we just wrap around to the beginning, so E goes into bucket zero.

The trick here is to be sure that your search process does not loop infinitely. Either keep track of how many buckets you've examined, or keep track of where you started, or something.

Linear Probing: Find



If we're looking for a value, we have to inspect what we find because it might be the wrong data.

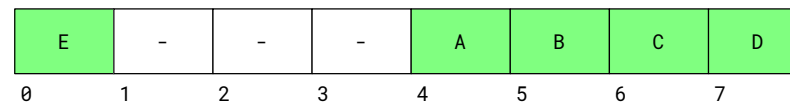
So if we look for B, which has a bucket code of 4, we look there, find A instead, so we examine the next spot. B is there, so we can return it.

What if something isn't there? Say we want Q, which has a bucket code of 6. **<next>**

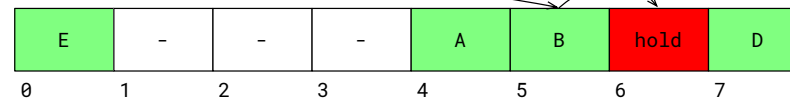
It's not in 6, not in 7, not in 0, and bucket 1 is empty. We can safely say that Q is not in the hash table.

Linear Probing: Remove

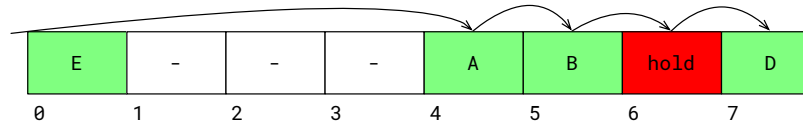
Current state



Remove C from bucket 5



Find D in bucket 4



Removing values is a bit trickier because we don't want to wreck the process for inserting and finding values. Those processes depended on having continuous ranges of filled cells, and stopping when we found an empty cell.

So rather than emptying a cell, we're going to mark it as removed, or held. **<next>**

Let's remove C from bucket 5. Works just like insert and find, we look at bucket five, it's not there, so with linear probing we look at the next one up. Hey, it's in six, so we mark it as removed. We're holding it in a special state, so it isn't confused with being empty.

This way when we do another operation, like finding D in bucket 4, **<next>**

it doesn't cause a problem that we've removed the element in bucket six. Bucket six isn't empty, it's been marked as held.

Quadratic Probing

Linear Probing:

when looking for key in bucket m , look at: m , $m+1$, $m+2$, $m+3$, $m+4$.

example: 7, 8, 9, 10, 11

Quadratic Probing:

when looking for key in bucket m , look at: m , $m+1$, $m+4$, $m+9$, $m+16$.

in other words: $m+0^2$, $m+1^2$, $m+2^2$, $m+3^2$, $m+4^2$

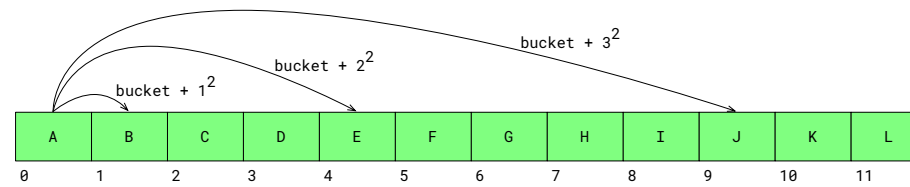
example: 7, 8, 11, 16, 23

With all of these indices, remember the table is circular, so use $(\text{index} \% \text{capacity})$

Quadratic probing is a similar strategy, but instead of examining buckets sequentially, 1, 2, 3..., you look at them with the pattern 1 squared, 2 squared, 3 squared, and so on.

Quadratic Probing

Find J in bucket 0



Quadratic probing is a similar strategy, but instead of examining buckets sequentially, 1, 2, 3..., you look at them with the pattern 1 squared, 2 squared, 3 squared, and so on. The reason you might do this is it is less prone to clustering, that is, having long sequences of used-up spaces that cause operations to take longer because they have to sift through more items.

In practice, linear probing can be great, so only use fancier strategies if you have an established need.

Episode 5

Hash Tables and Sets

By now we've talked about hash functions, how to smoosh arbitrary data down into manageable hash codes of known length. And we've talked about some of the uses and hazards with using these hash codes, collisions and how to deal with them. Now that you've been suitably warned about how hashing is not a silver bullet that solves all your problems, let's talk about data structures that use hashing, and some of the ins and outs thereof.

Hash Tables

Key	Value
“first name”	“Upton O. Goode”
“favorite color”	“chartreuse”
“birthday”	“Feb 29 1983”
“profession”	“magician”

We've seen the Map ADT, which lets you map some key to some other value. Maps don't have a built-in sense of ordering, so if you need to ask questions like "what's the first thing" , or "what's the thing after this one", or "how many things are in such-and-such range", a Map is probably not the way to go.

Maps are useful for storing "sparse" information, where we have some key, like "favorite color", to some value like the string "chartreuse" or **<next>** perhaps the hexadecimal number "7fff00". If you forget what your favorite color is, you can look it up with the key "favorite color", **<next>** and if you change your mind, you can set it to some other value.

Hash Table Complexity

- Set (K, V)
- Get (K)
- Remove (K)
- Contains (K)

All are $O(1)$

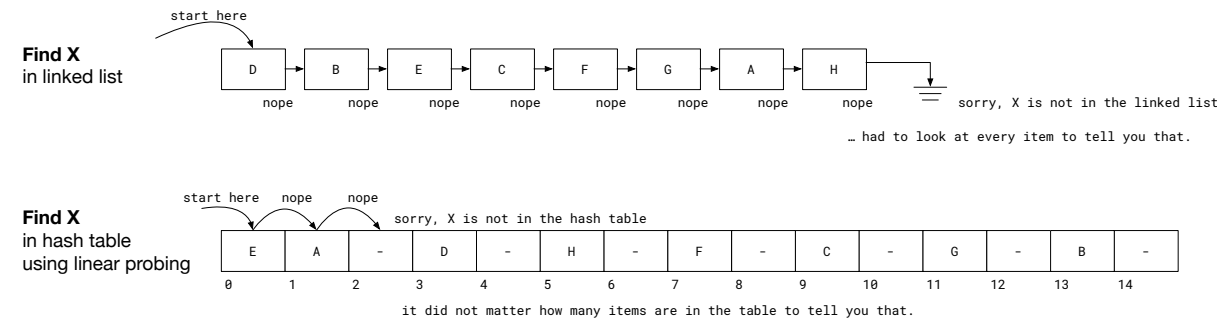
Unless your hash function is awful
Or your hash table is full

With a reasonable hash function, all of these operations

- Set (K, V)
- Get (K)
- Remove (K)
- Contains (K)

... are $O(1)$. And that property right there is why hashtables are so great, because they are super fast, and they conform to the Map ADT, which is a nice thing to work with.

Why Constant $O(1)$?



You might be wondering why the operations are constant time complexity, since with collisions we still might have to search a little bit, either through a chaining data structure, or by poking around the hashtable's buckets with some open addressing scheme.

Let's think way back to a few weeks ago when we talked about linked lists. If we query a linked list to see if it contains some value, we would have to potentially search every single cell in the linked list, which was an $O(n)$ operation. **<next>**

Searching through a hashtable, If it isn't completely full, the search ends quickly if the value isn't there because it'll encounter an empty cell. The probability that you'll have to look very far becomes really small if we keep lots of empty spots. If we have a completely full hashtable using open addressing, we _also_ have to search every single cell in the table. So why isn't that $O(n)$ also? Well, it _is_, but only in the _worst case scenario_.

So with a linked list, looking for an item that isn't there will require examining every single node, that's all N ; but with a reasonably empty hashtable, we will most likely only have to look at one spot. Maybe two or three, but not all N . Implementers of hash tables take special care to make sure their tables can resize to keep the number of steps small, perhaps even bounded to some number (like maybe 8 max) before the table's backing store is resized and all the data is rehashed to fit into different buckets.

Hash Sets

Operations on a single Set:

- Add (V) Put value V into the Set
- Remove (V) Remove value V from the Set
- Contains (V) Query if Set contains value V

(there are others)

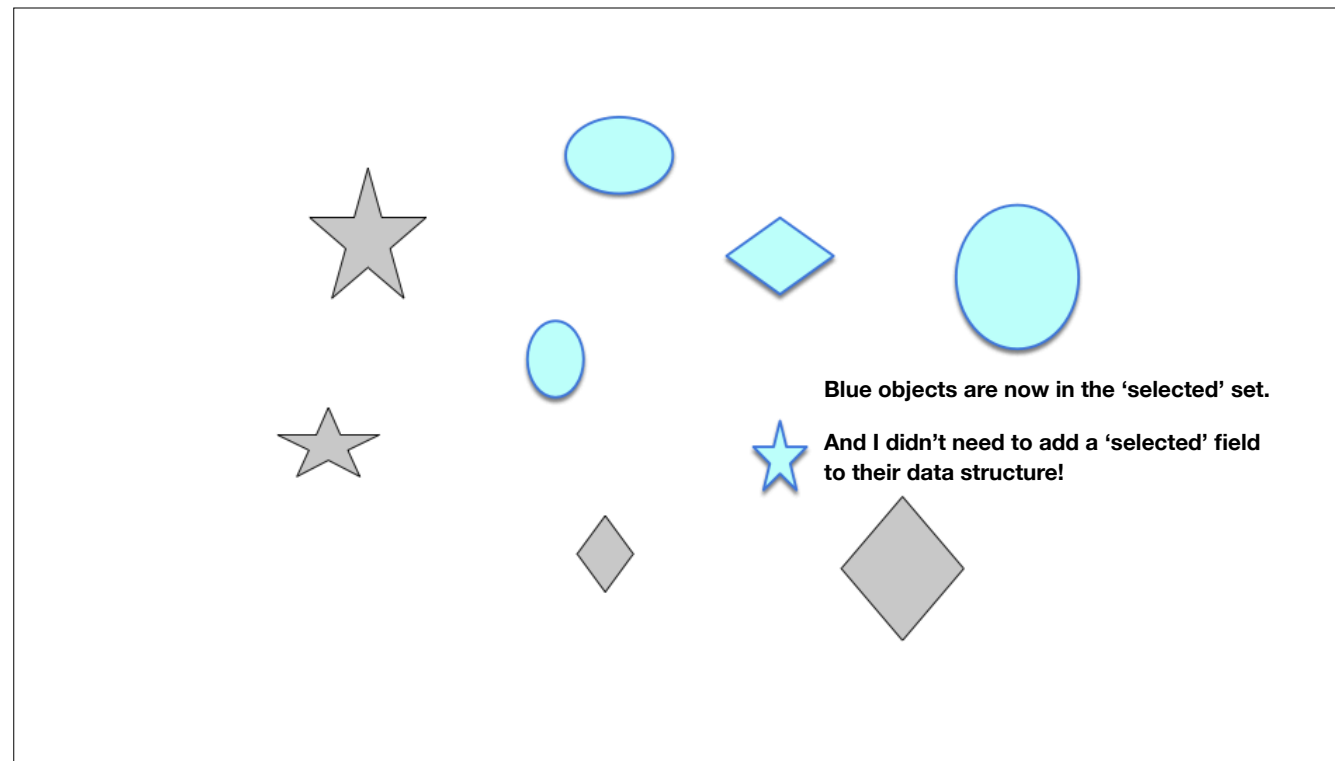
Set Algebra operations on two Sets:

- Union (A, B) Get set containing all of A and B
- Intersection (A, B) Get set containing items in both A and B
- Difference (A, B) Get set containing items in B but not in A

(there are others)

Hash tables are not the only hash-based data structure. Another common one is called a hash set, which implements the Set ADT.

Sets are similar to maps, but rather than associating a key with a value, sets model "present/not-present" on data. There are many mathematical uses to sets. And practically, sets help you out when you're writing programs because they let you keep track of data that has some property you care about.



Like if I'm writing a user interface where people can select a region with some objects in it. **<next>**

To record the fact that they're selected, I might want to create a set for all the objects inside that region, **<next>** so I can display them differently, and then if the user performs some action on them, like to upload them or delete them, I only need to get the values of the set.

<next> By doing this, I didn't need to go in to the object's definition to add a selected field or anything like that.

Homework

Key:	"favorite color"
Value:	"Gamboge"
Hashcode:	6183442980
Deleted:	false

The programming homework for this hash-stuff is to implement a hash table. The header file is documented pretty extensively. So I'll just give an overview of what's involved.

You'll have hash nodes, which contain the key and value, and also the already-computed hashcode, and a flag that says if the node has been marked as deleted.

Homework

Size:	4								
Capacity:	8								
Hash Function:	murmur2								
Table:	<table><tr><td><div><div></div><div></div><div></div><div></div></div></td><td><empty></td><td><div><div></div><div></div><div></div><div></div></div></td><td><empty></td><td><empty></td><td><div><div></div><div></div><div></div><div></div></div></td><td><div><div></div><div></div><div></div><div></div></div></td><td><empty></td></tr></table>	<div><div></div><div></div><div></div><div></div></div>	<empty>	<div><div></div><div></div><div></div><div></div></div>	<empty>	<empty>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<empty>
<div><div></div><div></div><div></div><div></div></div>	<empty>	<div><div></div><div></div><div></div><div></div></div>	<empty>	<empty>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<empty>		

Then you'll have the hash table, which contains metadata like the table's current size, and overall capacity. It has an array of hash nodes, and that's its backing store.

Homework Operations

init_node	initializes a new hash node
init_table	initializes a new hash table
set_kvp	establish or update a key/value pair mapping
load	report how full the table is
get_val	returns a value given some key, if it is present
contains	query if the table contains a key/value pair for some key
remove	removes a key/value pair if it is present
resize	grow or shrink the backing store, re-hash everything, update metadata.

These are the operations you'll code this week. Your implementation of the hashtable won't grow or shrink automatically as you add or remove things. But as extra credit, you can implement the resize function, which would then be called manually. A real hashtable would likely grow or shrink automatically as you add and remove things beyond some load threshold.

