Hashing

Brahm Capoor

CS 106B | Thursday, August 3rd

Based on lectures given by Chris Gregg and Anton Apostalatos

My laundry

My laundry on steroids

My laundry if I were a functional adult

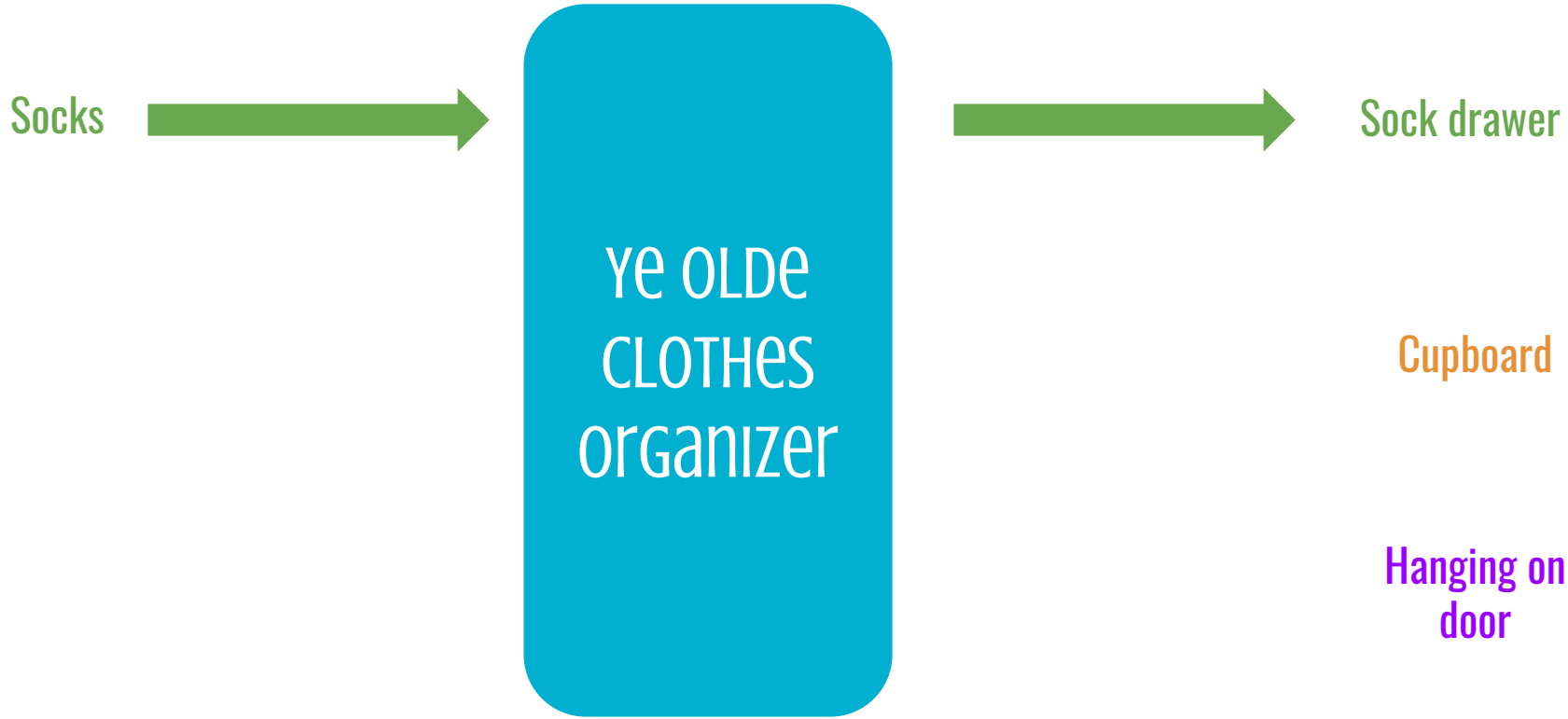
A mapping from clothes to storage locations

Sock drawer

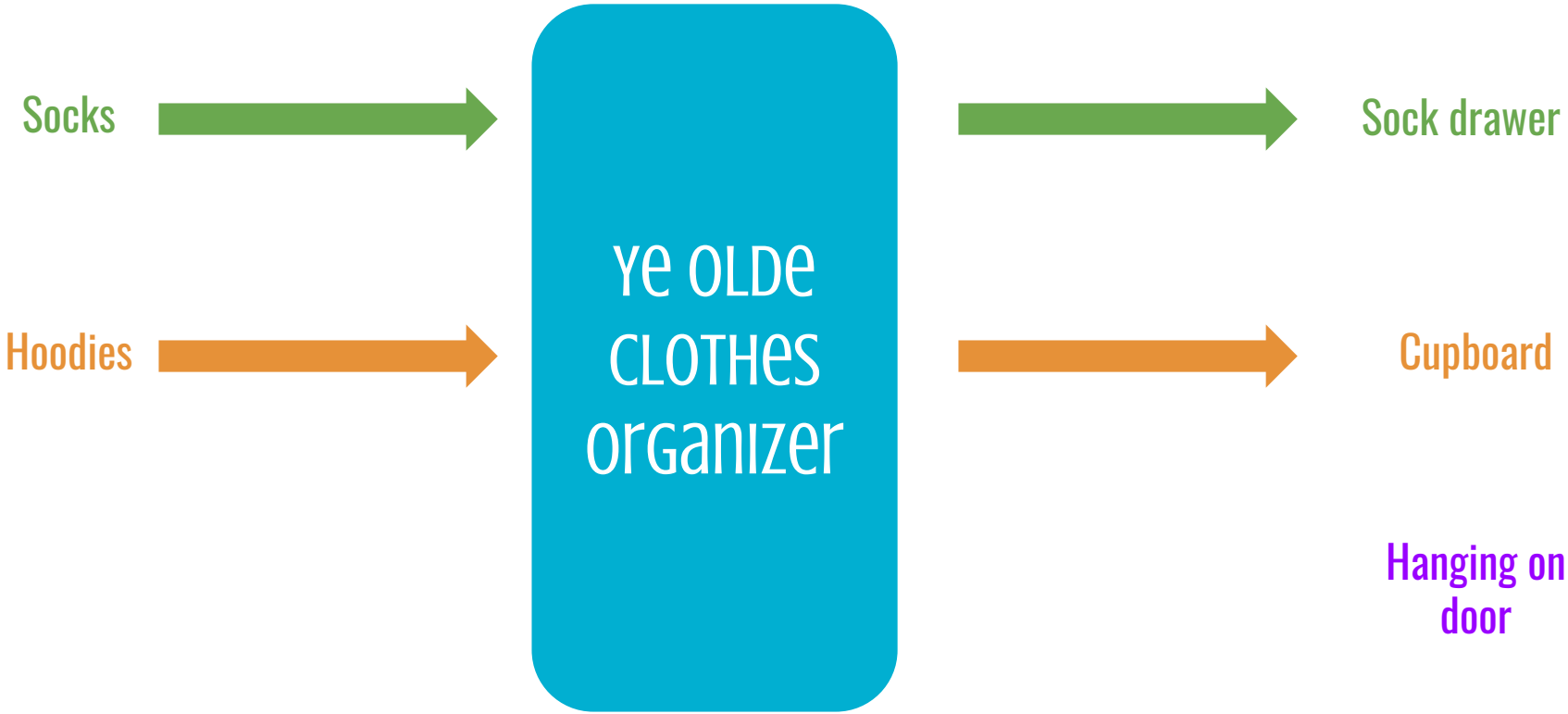
Cupboard

Hanging on door

A mapping from clothes to storage locations



A mapping from clothes to storage locations



A mapping from clothes to storage locations

How does this mapping help?

I can go directly to where the clothes would be

How does this mapping help?

I can go directly to where the clothes would be

Lookup is improved

How does this mapping help?

I can go directly to where the clothes would be

Lookup is improved

Insertion is improved

How does this mapping help?

I can go directly to where the clothes would be

Lookup is improved

Insertion is improved

Removal is improved

How does this mapping help?

I can go directly to where the clothes would be

Lookup is improved

Insertion is improved

Removal is improved

Assuming I have N clothes, operations go from O(N) to O(1)

How does this mapping help?

Lookup is O(1)

Insertion is O(1)

Removal is O(1)

} Could we use this in a data structure?

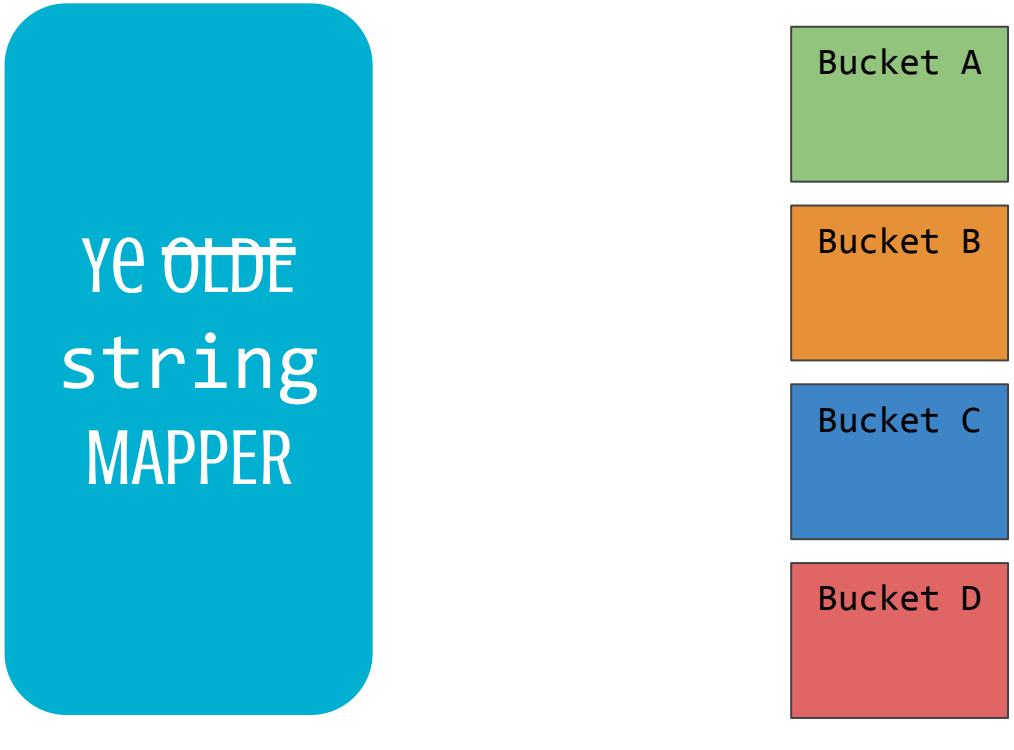
A mapping from clothes to storage locations

Sock drawer

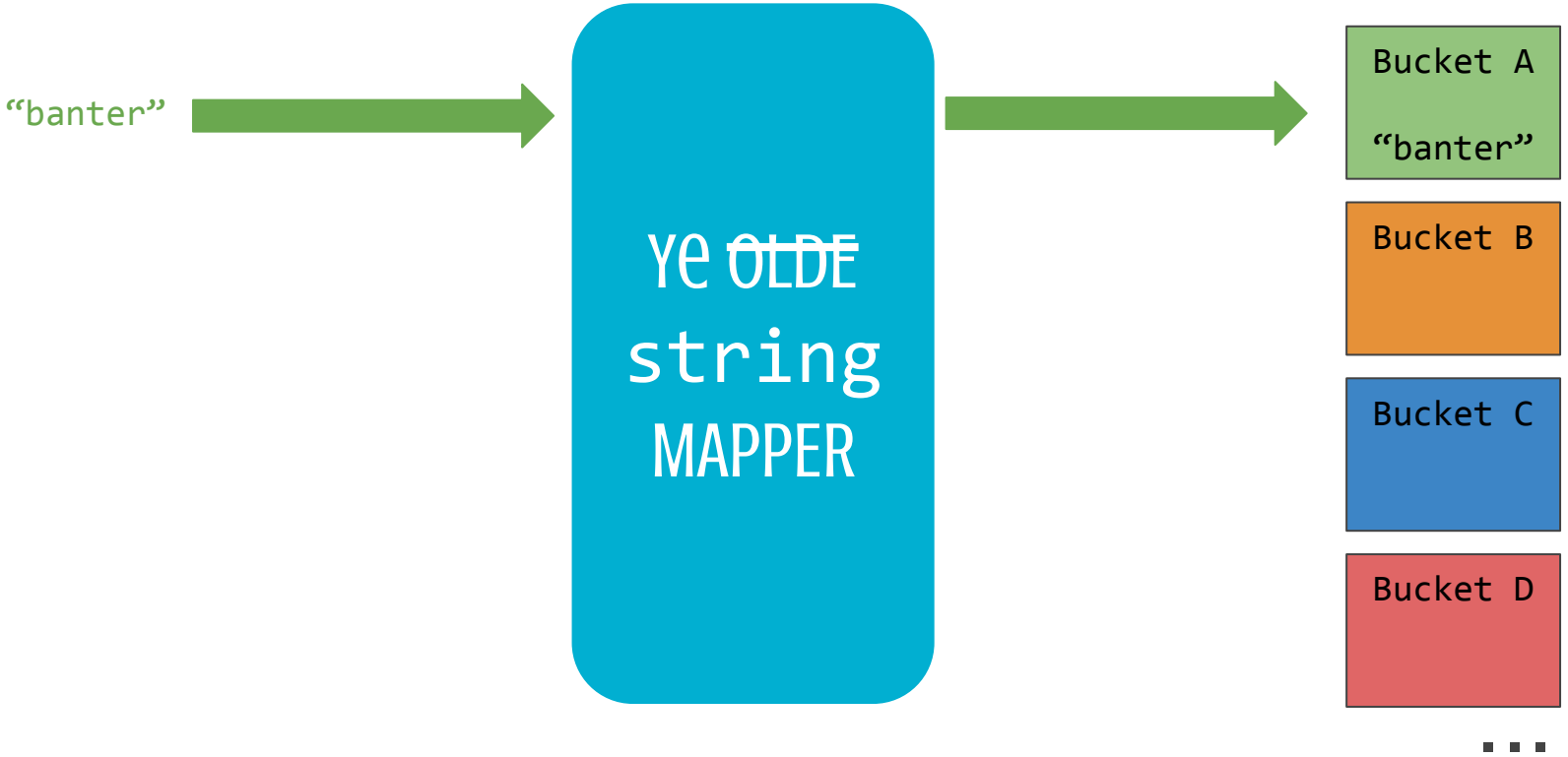
Cupboard

Hanging on door

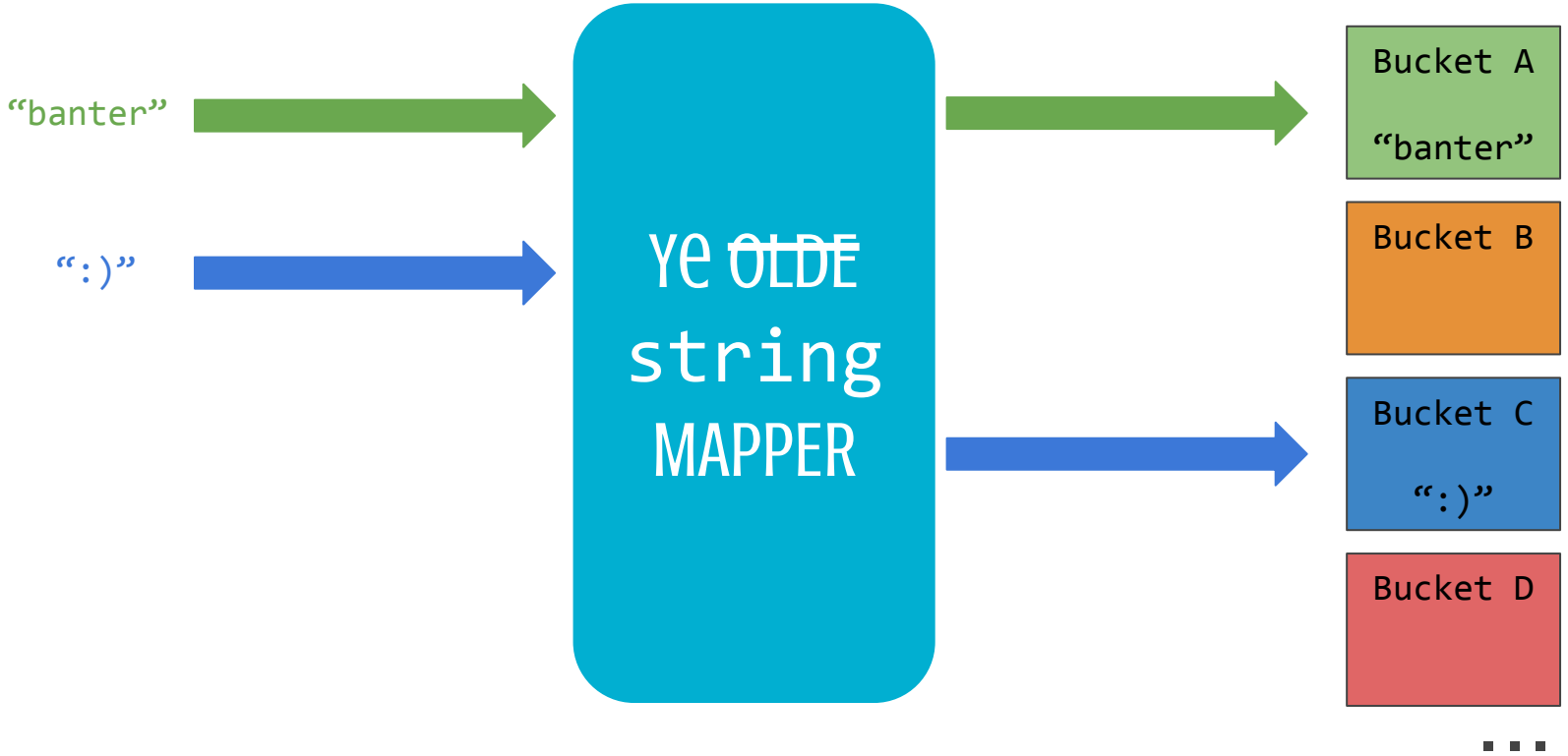
How a mapped data structure might look



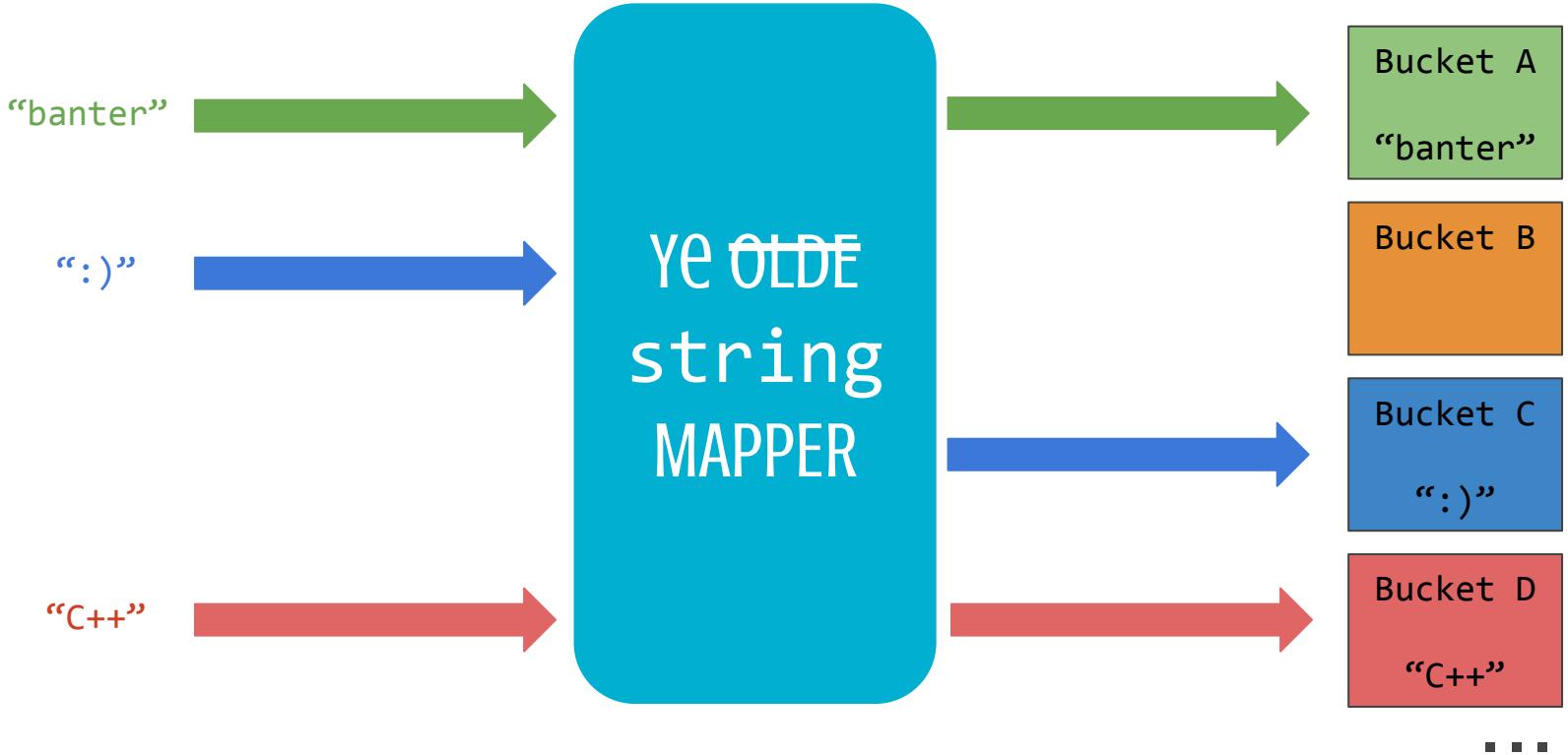
How a mapped data structure might look



How a mapped data structure might look

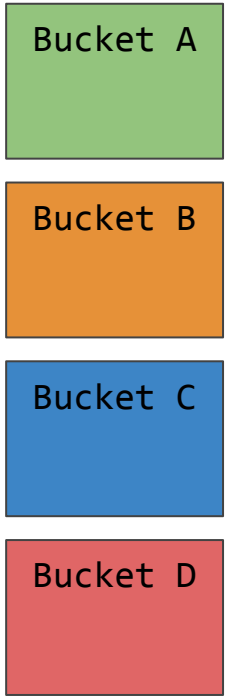


How a mapped data structure might look



The last piece of the puzzle

How do we formalize the mapping between `strings` and buckets?



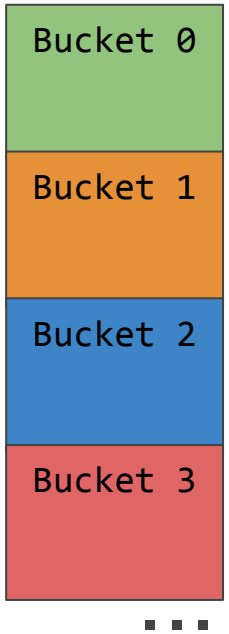
The last piece of the puzzle

How do we formalize the mapping between `strings` and buckets?

Step 1: Turn the buckets into an array

Ye ~~OLDE~~ `string` MAPPER

```
string *buckets = new string[nBuckets];
```

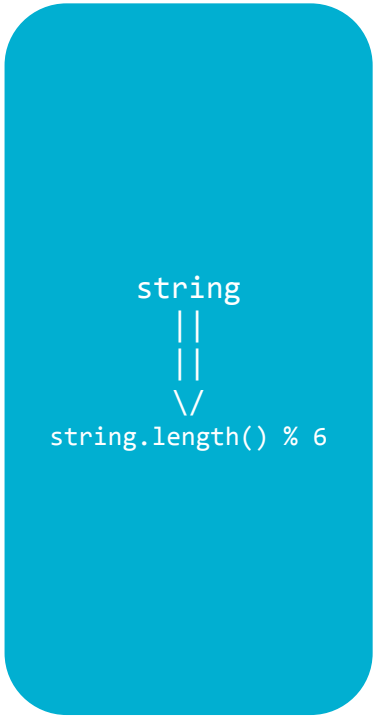


The last piece of the puzzle

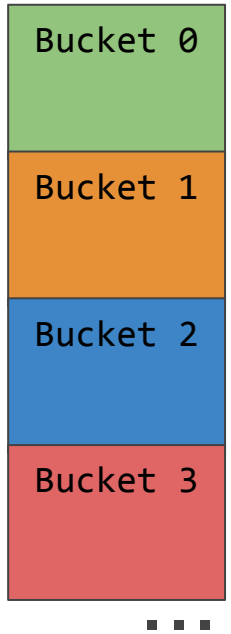
How do we formalize the mapping between `strings` and buckets?

Step 1: Turn the buckets into an array

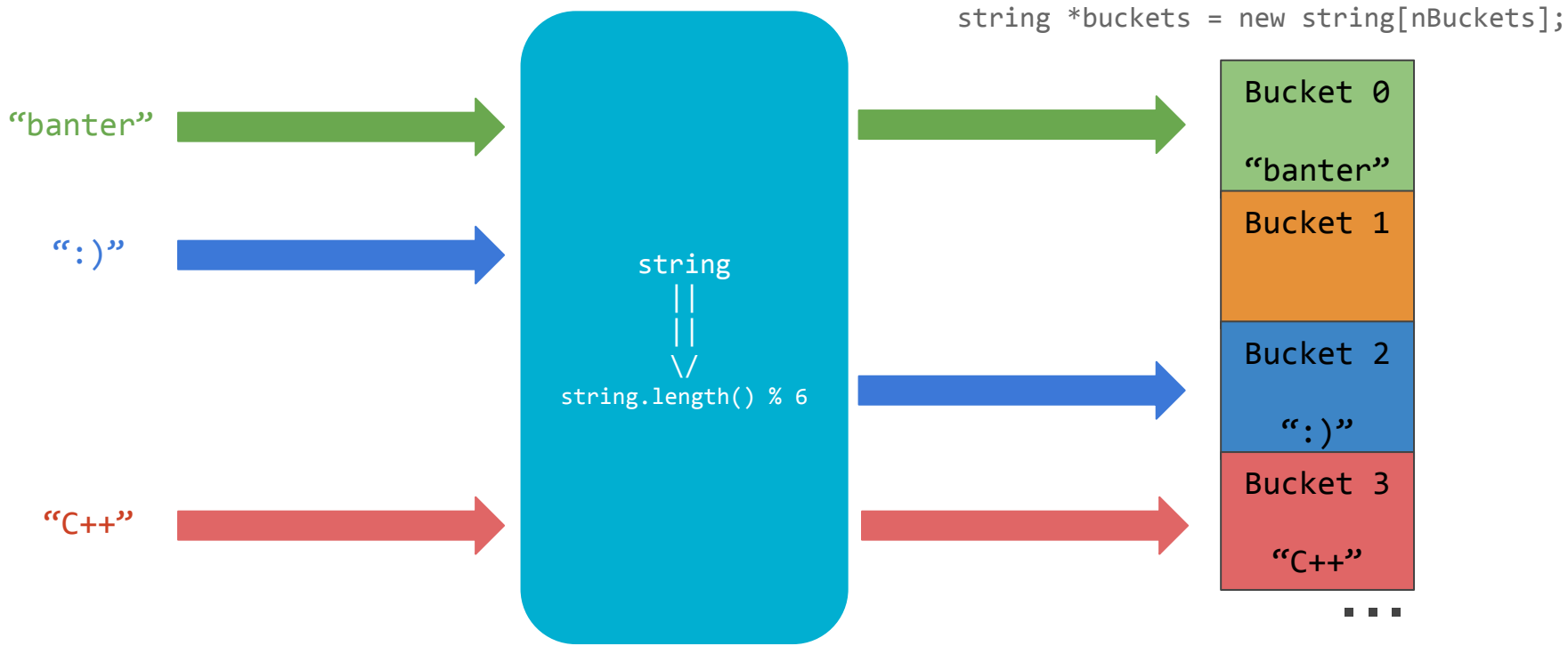
Step 2: Define a function from a `string` to the index of a bucket in the array



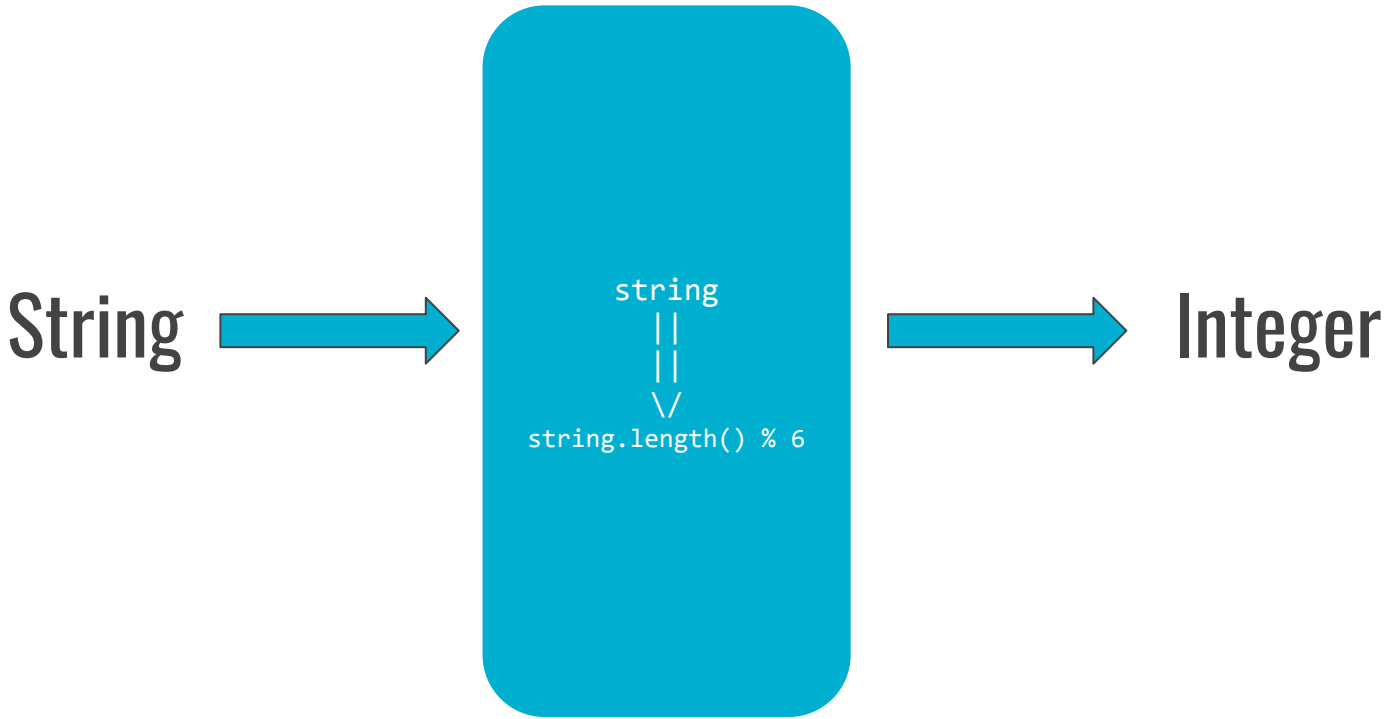

```
string *buckets = new string[nBuckets];
```



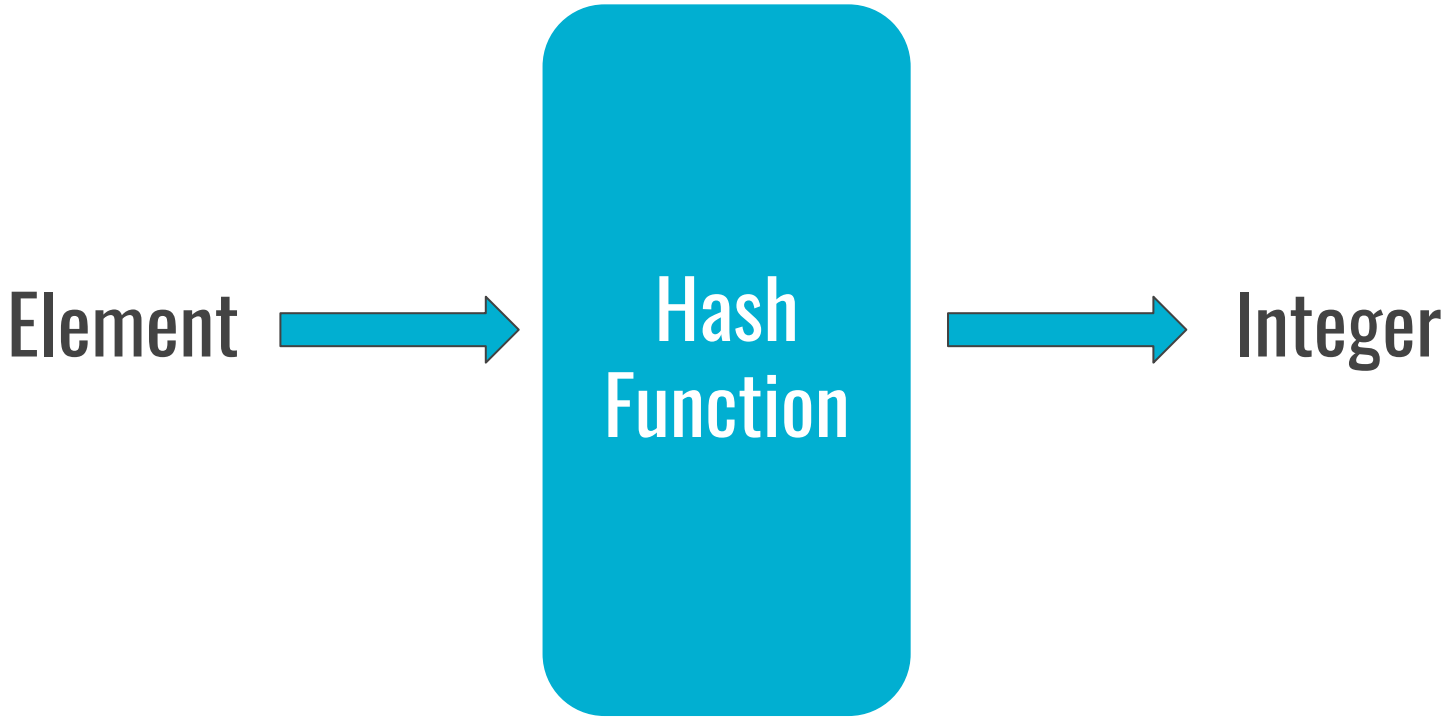
Putting it all together



Our Hash Function



The General Hash Function



Implementing a HashMap!

`class HashMap<KeyType, ValueType>`

This class implements an efficient association between ***keys*** and ***values***. This class is identical to the `Map` class except for the fact that it uses a hash table as its underlying representation. Although the `HashMap` class operates in constant time, the iterator for `HashMap` returns the values in a seemingly random order.

Methods

`get(key)`	O(1)	Returns the value associated with `key` in this map.
`put(key, value)`	O(1)	Associates `key` with `value` in this map.
`remove(key)`	O(1)	Removes any entry for `key` from this map.

Lookup, insertion and removal are all O(1)!

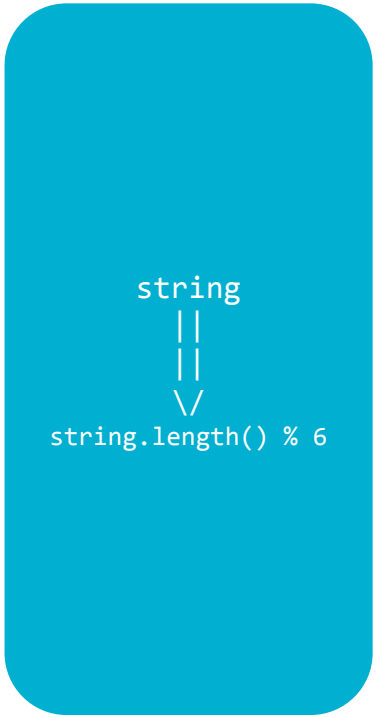
Implementing a HashMap!

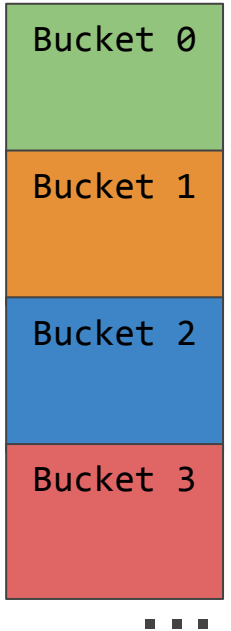
Let's make a HashMap with `string` keys and `int` values

Implementing a HashMap!

Let's make a HashMap with `string` keys and `int` values

How can we use our existing infrastructure?



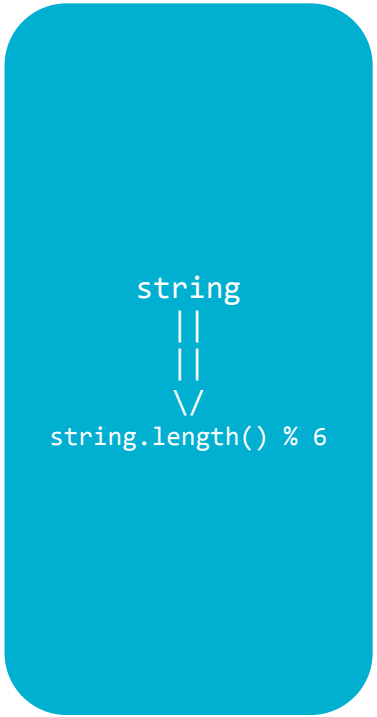


Implementing a HashMap!

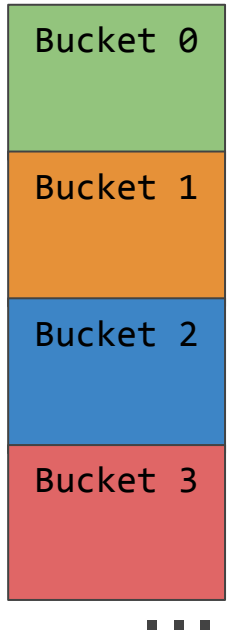
Let's make a HashMap with `string` keys and `int` values

How can we use our existing infrastructure?

What should we put in the buckets? Let's see what happens when we do a lookup




```
<type> *buckets = new <type>[nBuckets];
```



Implementing a HashMap: lookup

Let's imagine that the key-value pair `("banter", 1)` is already in our map.

How would get get that `1` out when we call `Map.get("banter")`?

Implementing a HashMap: lookup

Let's imagine that the key-value pair `("banter", 1)` is already in our map.

How would get get that `1` out when we call `Map.get("banter")`?

`"banter"`

Implementing a HashMap: lookup

Let's imagine that the key-value pair `("banter", 1)` is already in our map.

How would get get that `1` out when we call `Map.get("banter")`?

Implementing a HashMap: lookup

Let's imagine that the key-value pair `("banter", 1)` is already in our map.

How would get get that `1` out when we call `Map.get("banter")`?

Implementing a HashMap: lookup

Let's imagine that the key-value pair `("banter", 1)` is already in our map.

How would get get that `1` out when we call `Map.get("banter")`?

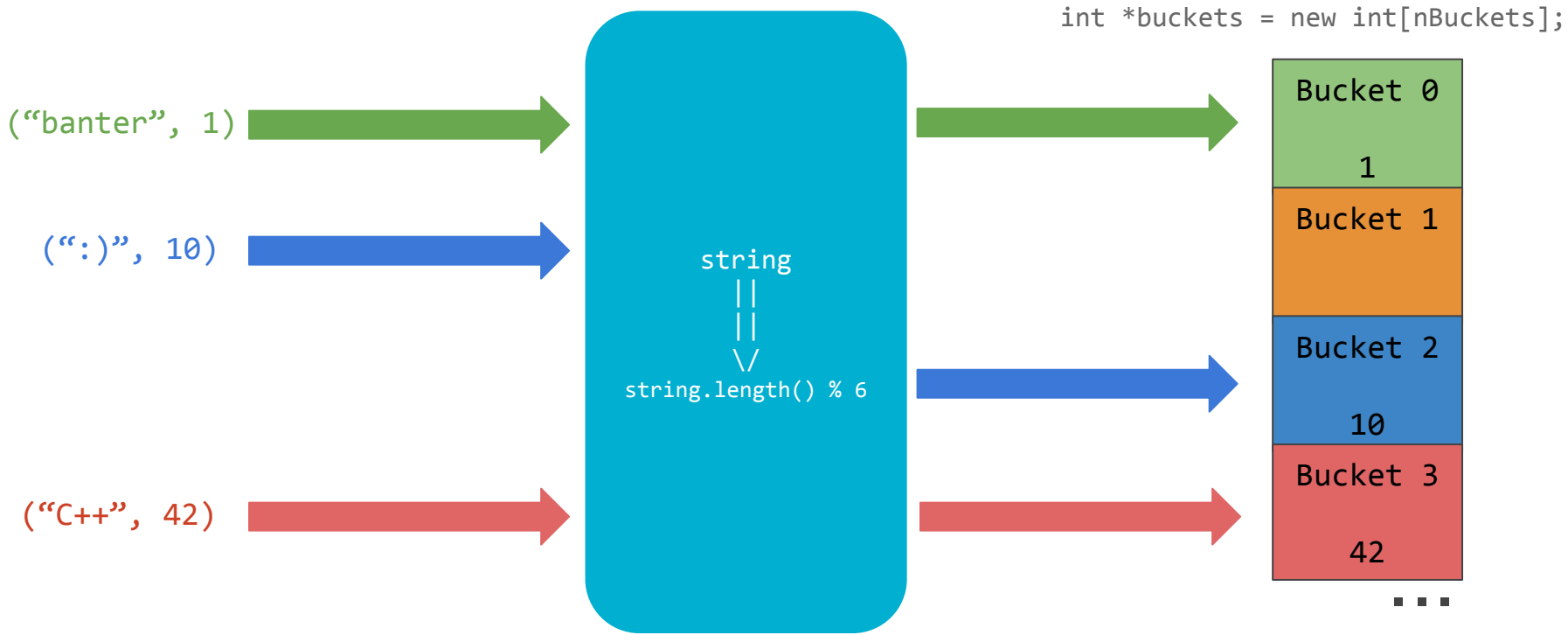
Implementing a HashMap: lookup

Let's imagine that the key-value pair `("banter", 1)` is already in our map.

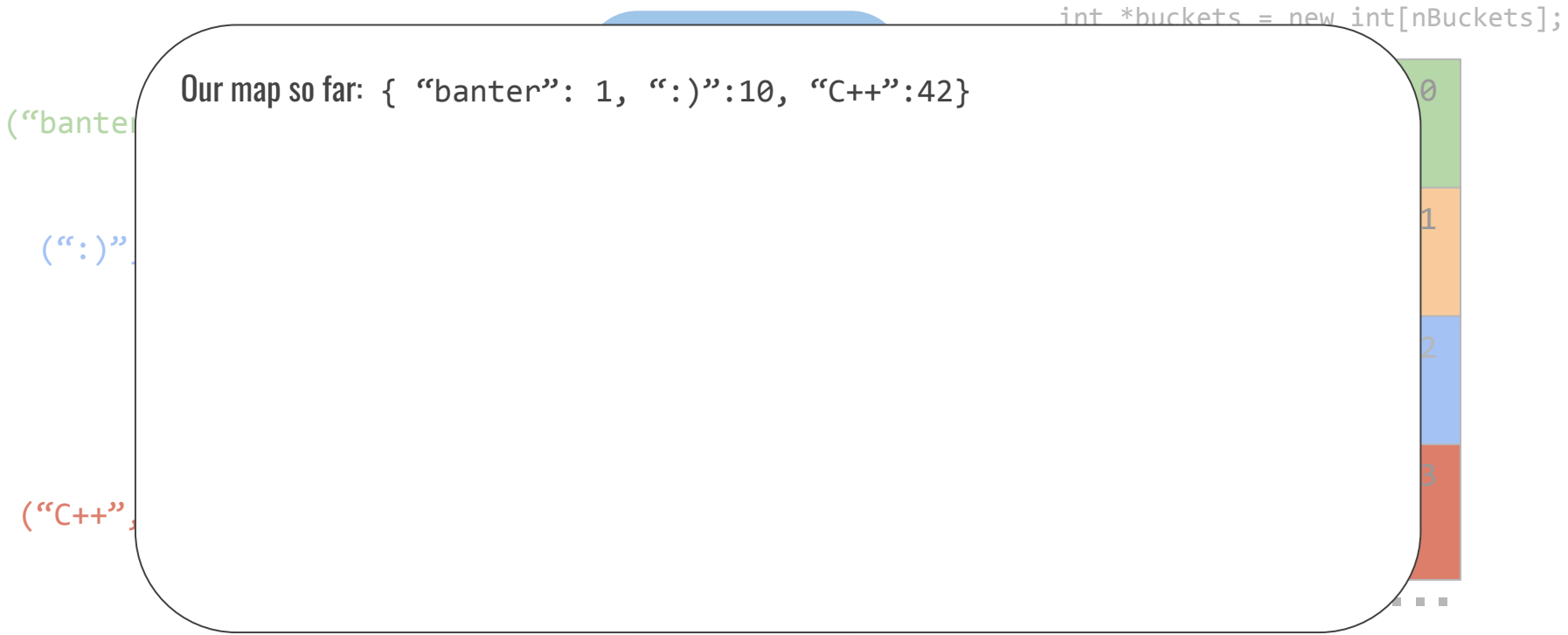
How would get get that `1` out when we call `Map.get("banter")`?

Put the values in the buckets?

What that would look like



A spanner in the works



A spanner in the works

```
int *buckets = new int[nBuckets];
```

(“banter

(“:)”

(“C++”

0

1

2

3

Our map so far: `{ “banter”: 1, “:)”:10, “C++”:42}`

What if I wanted to put `(“Razzmatazzes”, 13)` in the map?

A spanner in the works

```
int *buckets = new int[nBuckets];
```

(“banter

(“:)”

(“C++”

0

1

2

3

Our map so far: { “banter”: 1, “:)”:10, “C++”:42}

What if I wanted to put (“Razzmatazzes”, 13) in the map?

```
“Razzmatazzes”.length() % 6 == “banter”.length % 6
```

A spanner in the works

```
int *buckets = new int[nBuckets];
```

(“banter

(“:)”

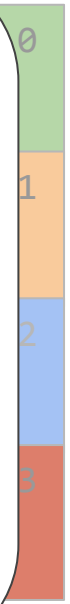
(“C++”

Our map so far: `{ “banter”: 1, “:)”:10, “C++”:42}`

What if I wanted to put `(“Razzmatazzes”, 13)` in the map?

`“Razzmatazzes”.length() % 6 == “banter”.length % 6`

How do I put it in the map without affecting the existing (“banter”, 1”) pair?



A spanner in the works

int *buckets = new int[nBuckets];

("banter

(":)"

("C++"

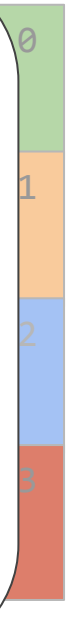
Our map so far: { "banter": 1, ":)":10, "C++":42}

What if I wanted to put ("Razzmatazzes", 13) in the map?

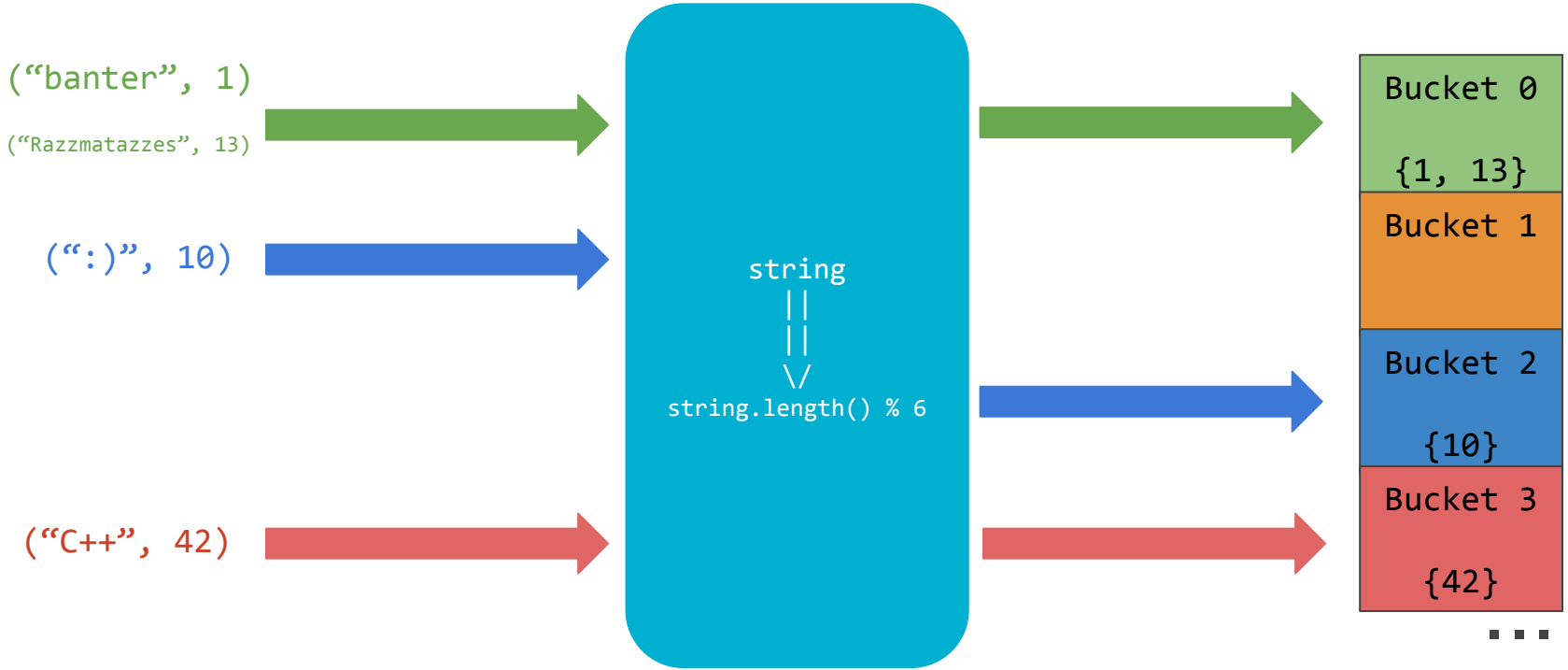
"Razzmatazzes".length() % 6 == "banter".length % 6

How do I put it in the map without affecting the existing ("banter", 1") pair?

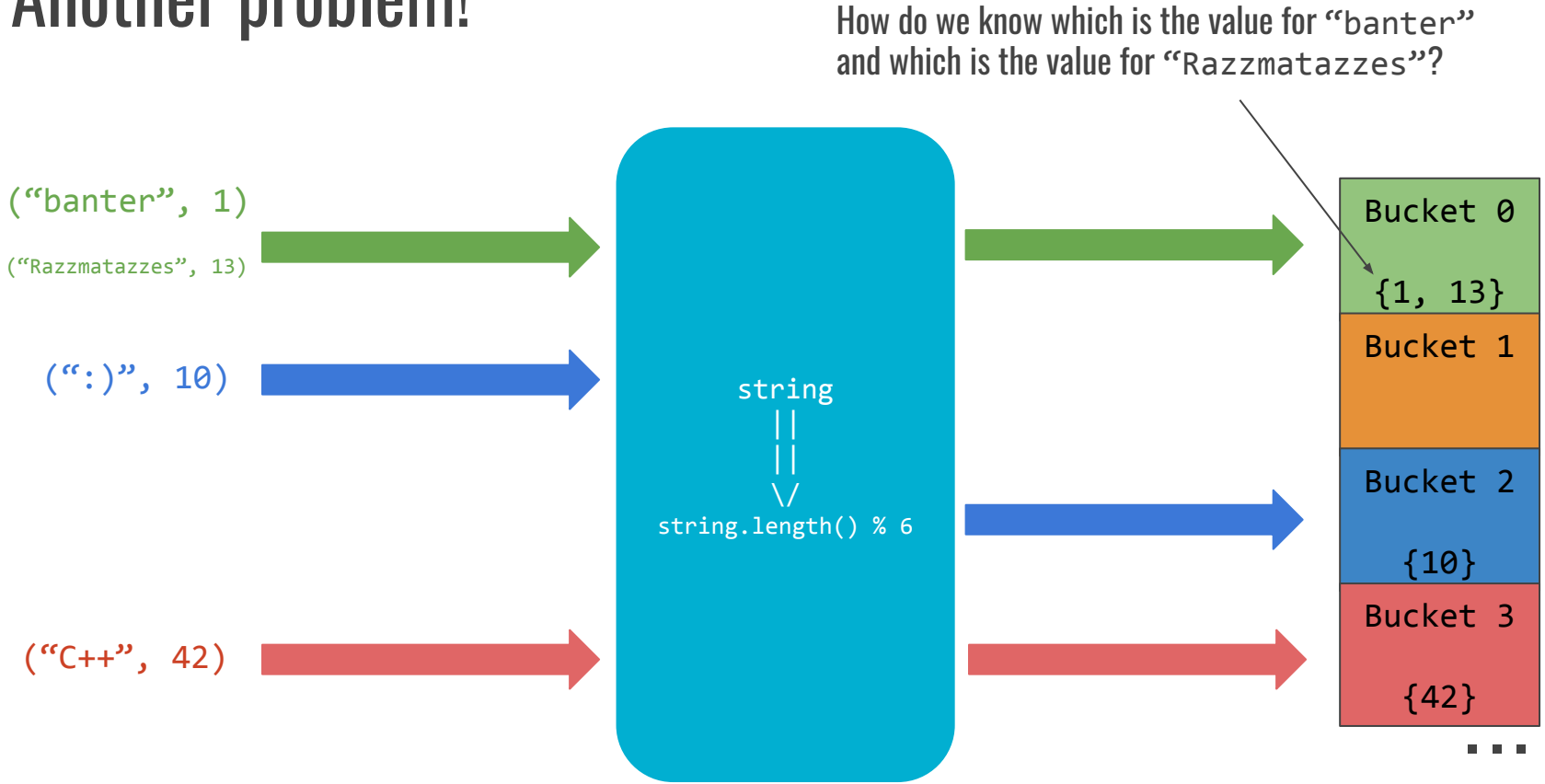
Possible solution: Make the buckets collections of values instead



What that would look like

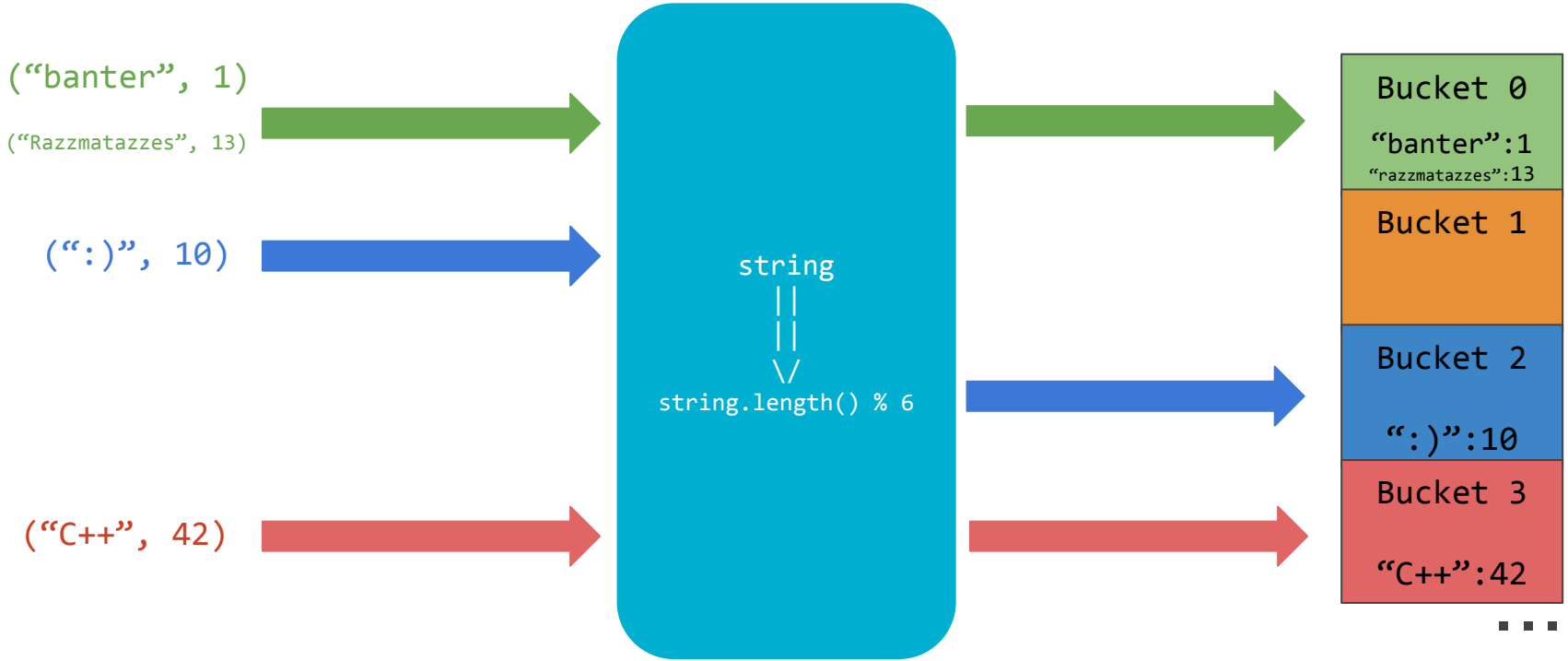


Another problem!



Another solution!

The solution: store key-value pairs as `structs` instead



Properties of hash functions

```
int hashFunction(const string &s) {
     return s.length() % 6;
}
```

Properties of hash functions

1. Deterministic: the same input always gives the same output

```
int hashFunction(const string &s) {
     return s.length() % 6;
}
// hashFunction("banter") is always 0
```

Properties of hash functions

1. Deterministic: the same input always gives the same output
2. Fast: Runs quickly

```
int hashFunction(const string &s) {
     return s.length() % 6;
}
// hashFunction(“banter”) is always 0
```

Properties of hash functions

1. Deterministic: the same input always gives the same output
2. Fast: Runs quickly
3. Well distributed output

```
int hashFunction(const string &s) {
     return s.length() % 6;
}
// hashFunction("banter") is always 0
```

Collisions 💥

Unless I have infinite buckets, I can't guarantee that everything will have its own bucket

Collisions 💥

Unless I have infinite buckets, I can't guarantee that everything will have its own bucket

If two things are hashed into the same bucket, a collision has occurred

Collisions 💥

Unless I have infinite buckets, I can't guarantee that everything will have its own bucket

If two things are hashed into the same bucket, a collision has occurred

This isn't necessarily a bad thing

Load factors

The load factor of a hashmap is `n/N`

`n` is the number of keys in the map
`N` is the number of buckets in the map

Load factors

The load factor of a hashmap is `n/N`

`n` is the number of keys in the map
`N` is the number of buckets in the map

If the load factor is low, and the hash function is well distributed, operations are O(1)

Load factors

The load factor of a hashmap is N/n

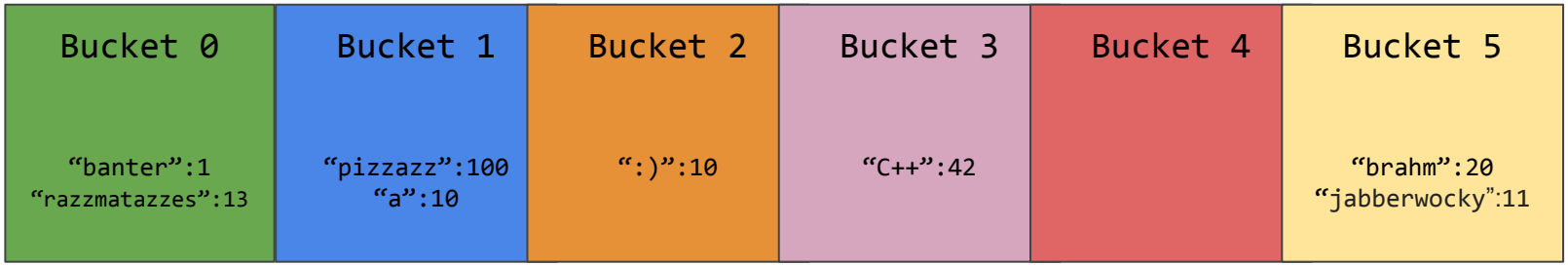
- N is the number of keys in the map
- n is the number of buckets in the map

If the load factor is low, and the hash function is well distributed, operations are O(1)

If the load factor is high (N >> n), or the hash function is badly distributed, operations are O(N)

Rehash if the load factor is too high

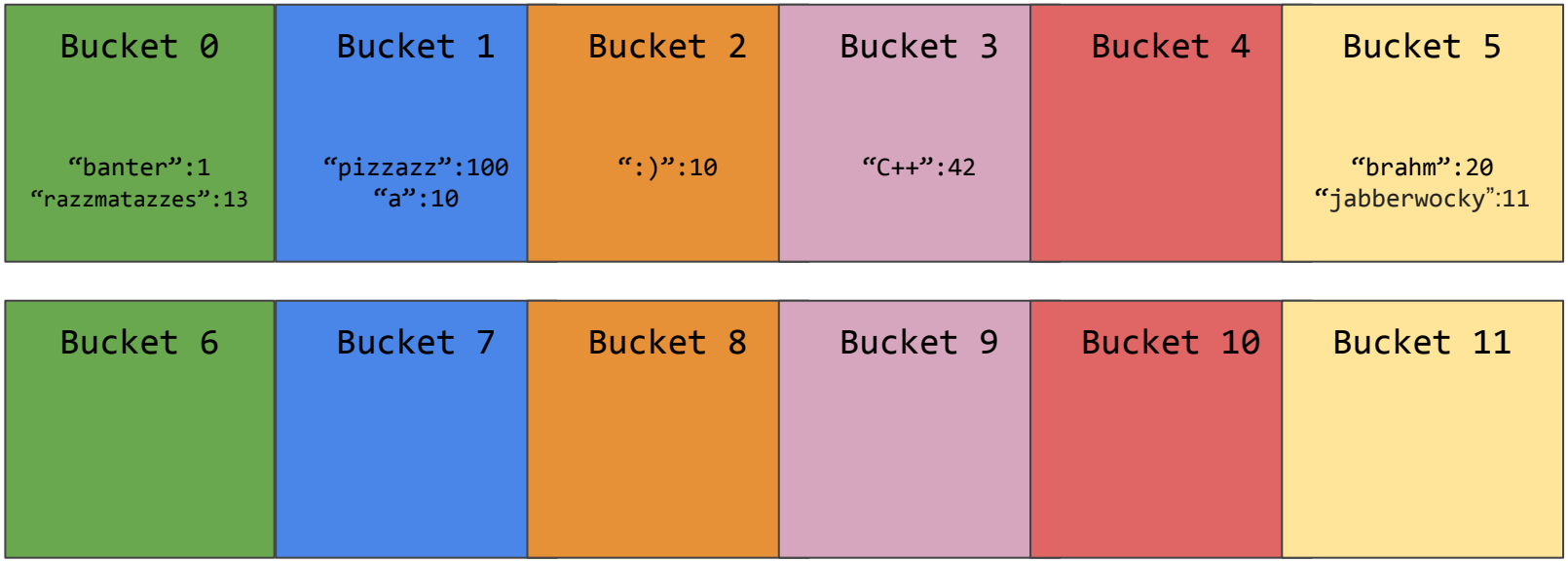
Hash function: `string.length() % 6`



Rehash if the load factor is too high

Problem: we only use 6/12 buckets!

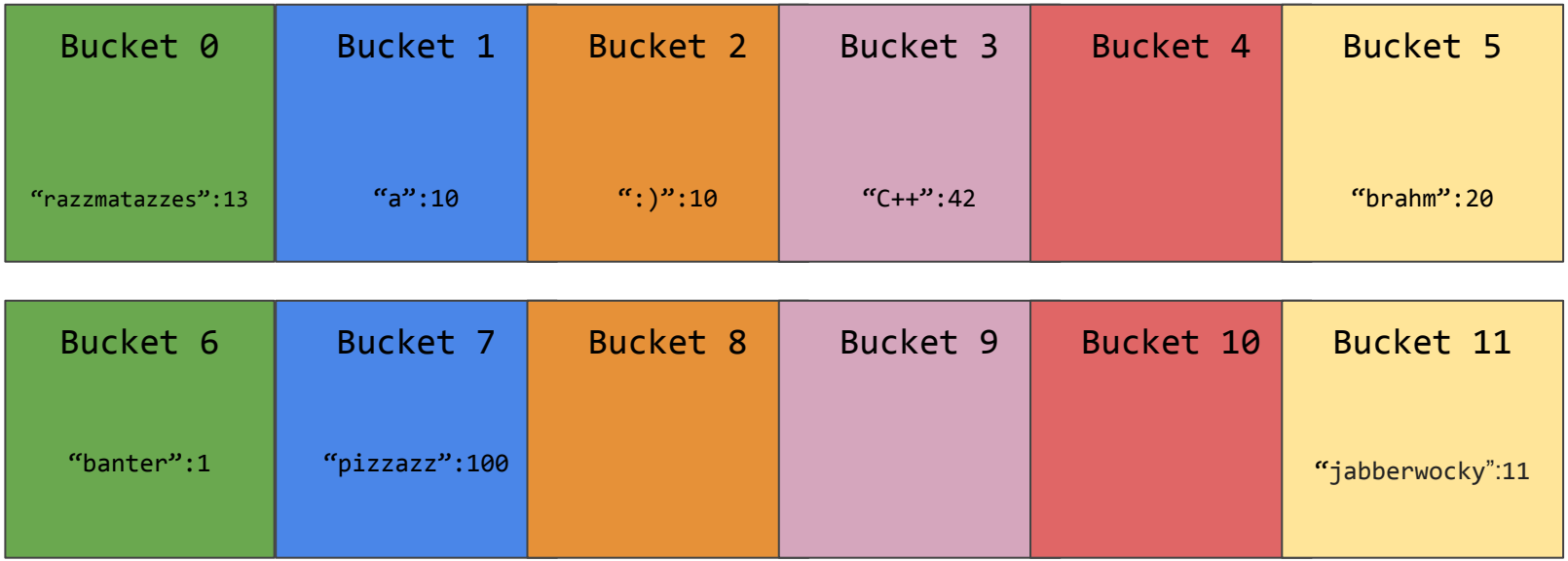
Hash function: `string.length() % 6`



Rehash if the load factor is too high

Solution: Change how much we compress the string's length

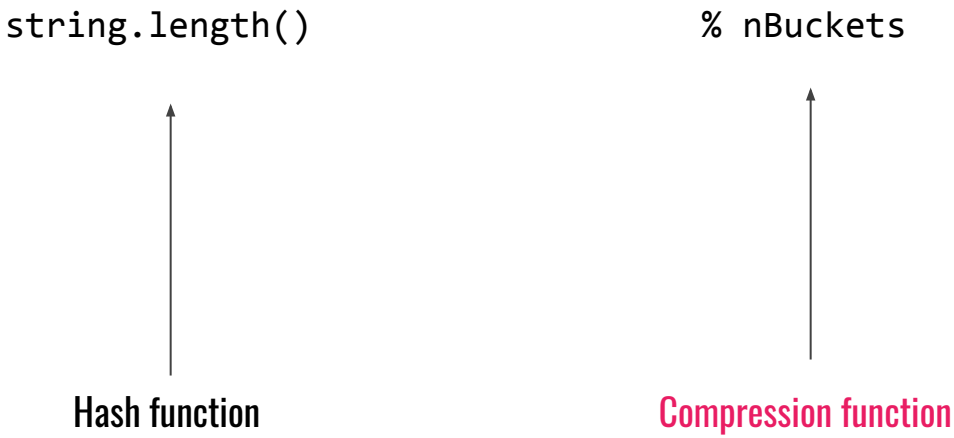
Hash function: `string.length() % 12`



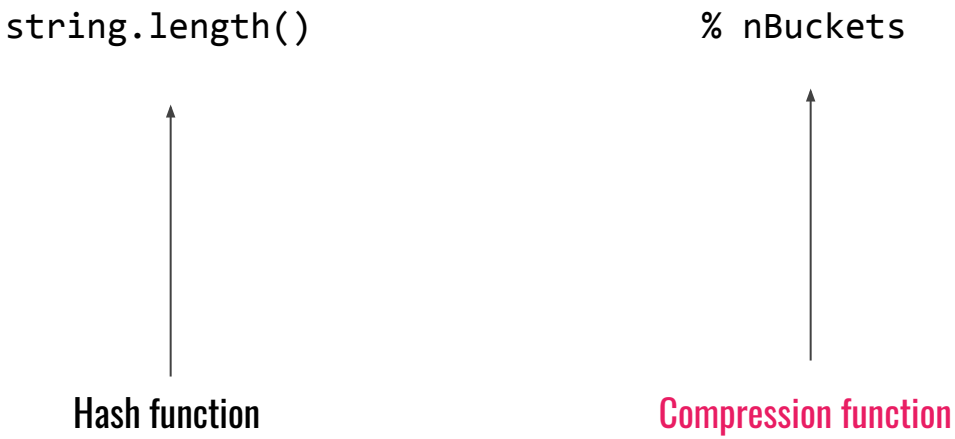
Compression functions

```
string.length() % nBuckets
```

Compression functions



Compression functions



When we rehash:

This stays the same!

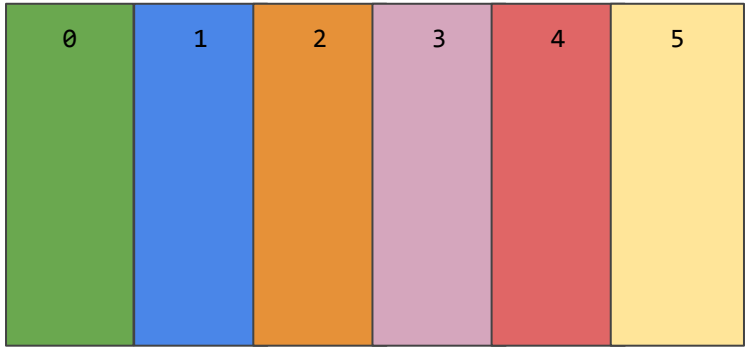
Only this changes!

Our `HashMap<int, int>`

```
nBuckets: 6
nElems: 0
```

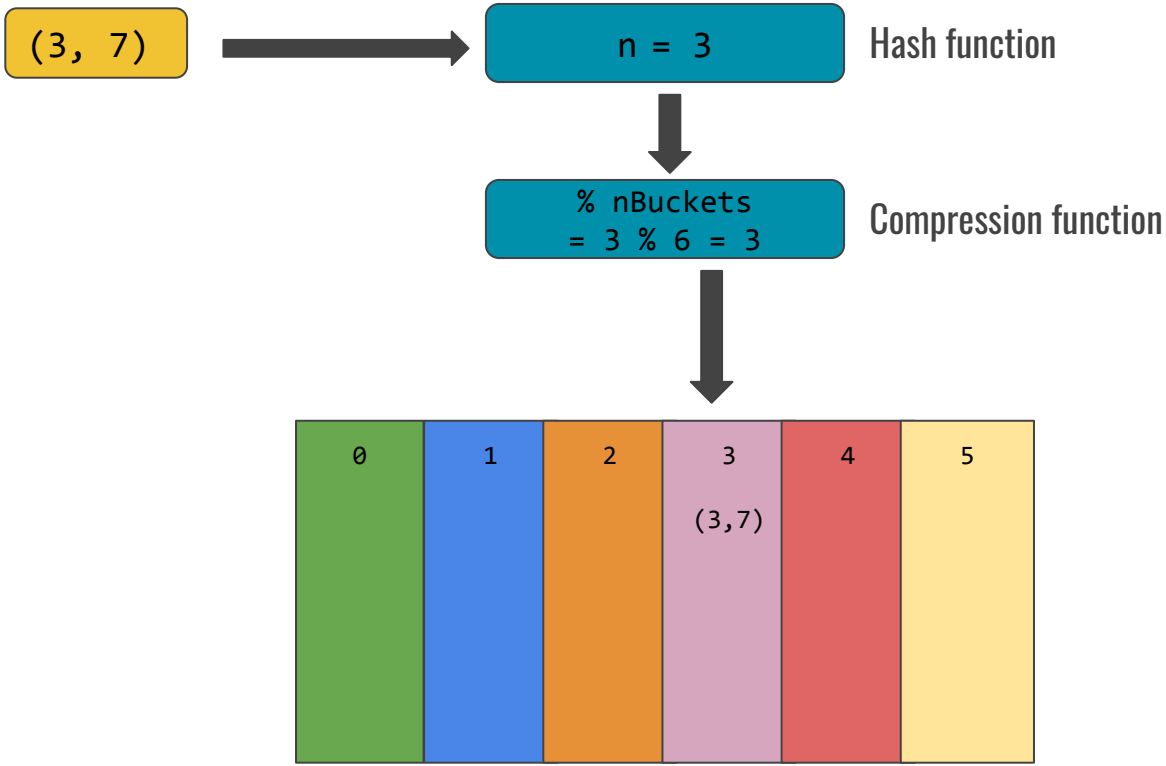
`n` Hash function

`% nBuckets` Compression function



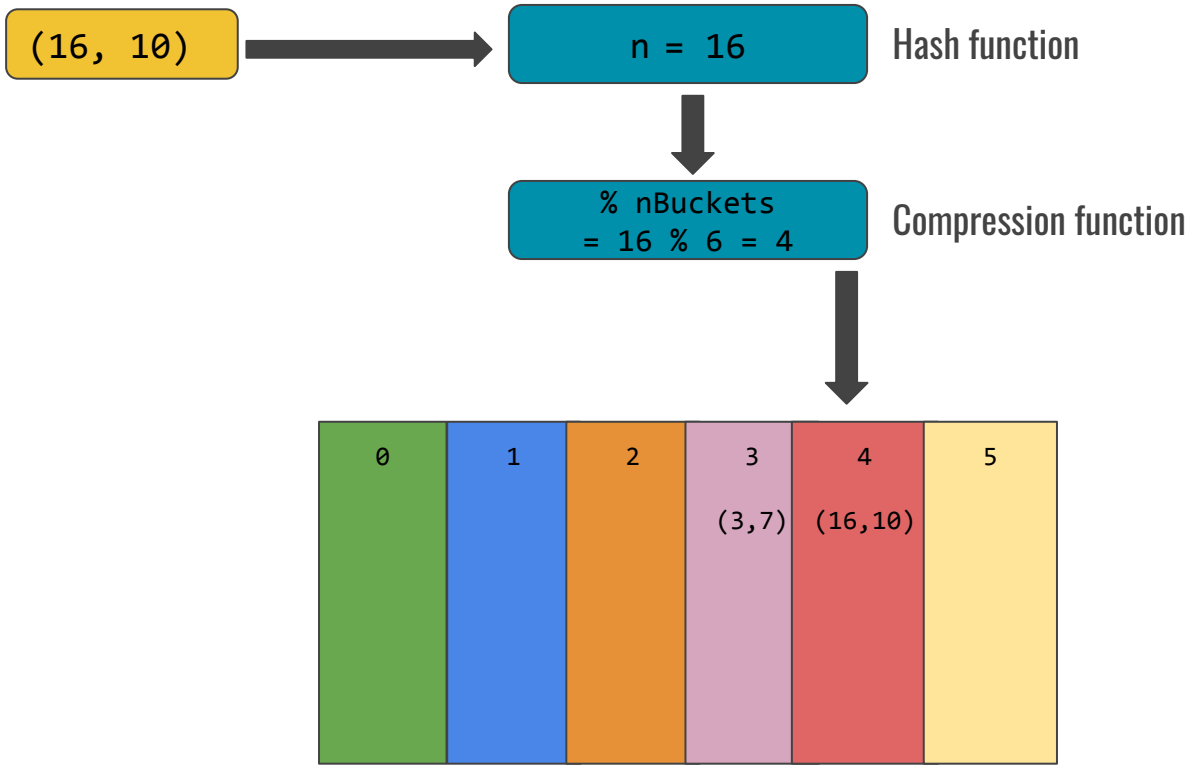
Map.put(3,7)

```
nBuckets: 6
nElems: 1
```



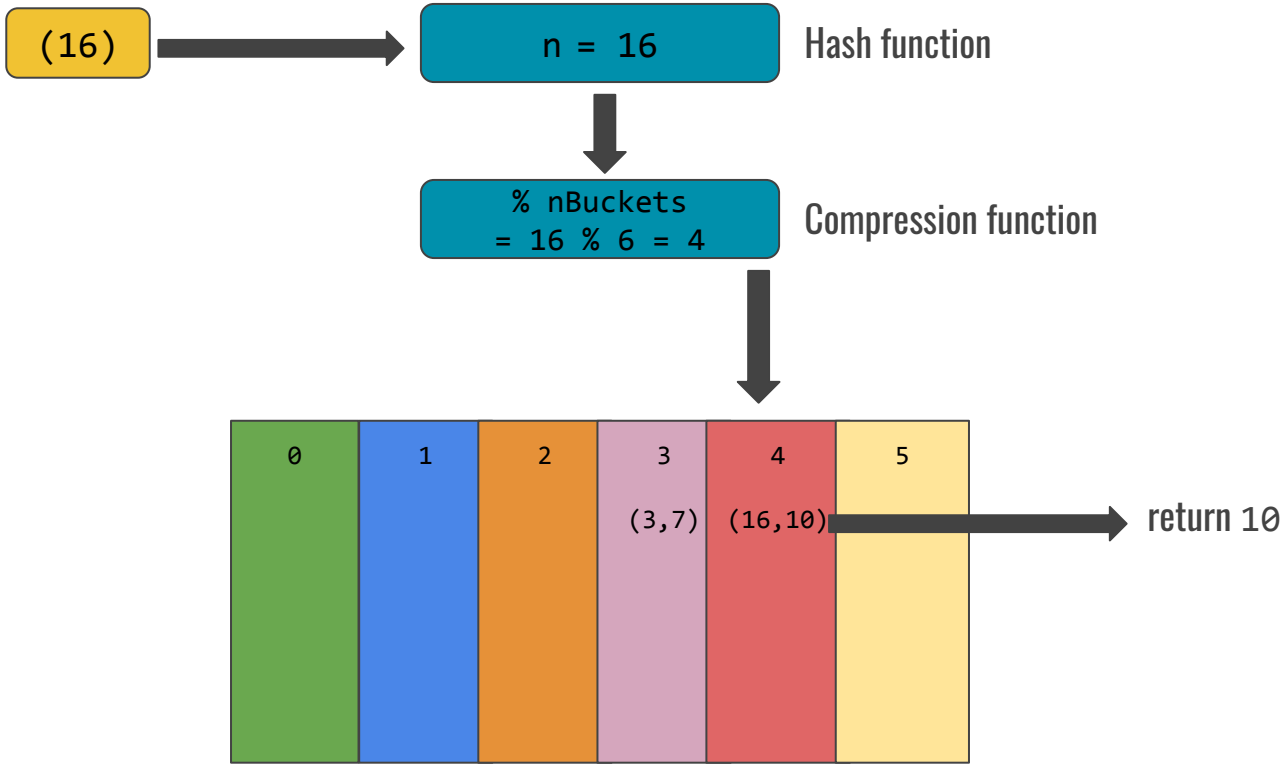
Map.put(16,10)

```
nBuckets: 6
nElems: 2
```



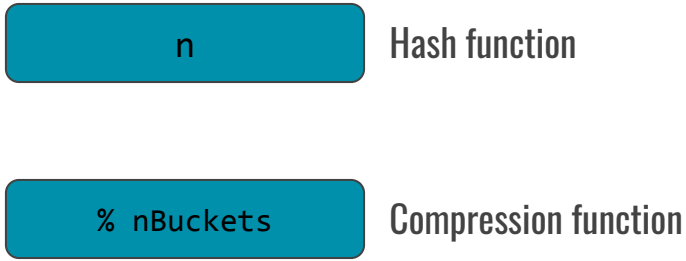
Map.get(16)

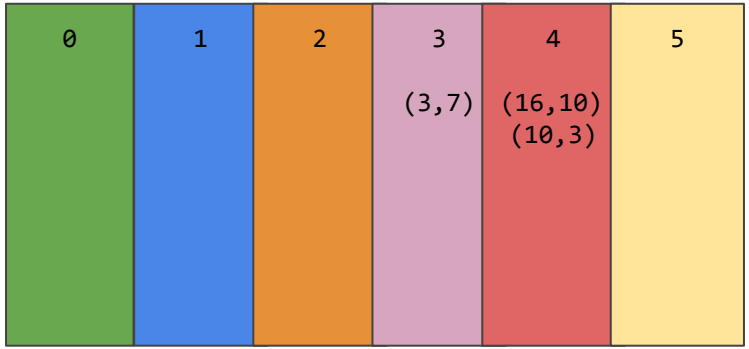
```
nBuckets: 6
nElems: 2
```



Rehashing

```
nBuckets: 6
nElems: 3
```





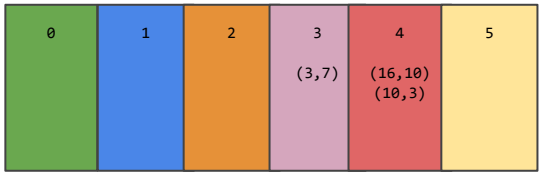
Rehashing

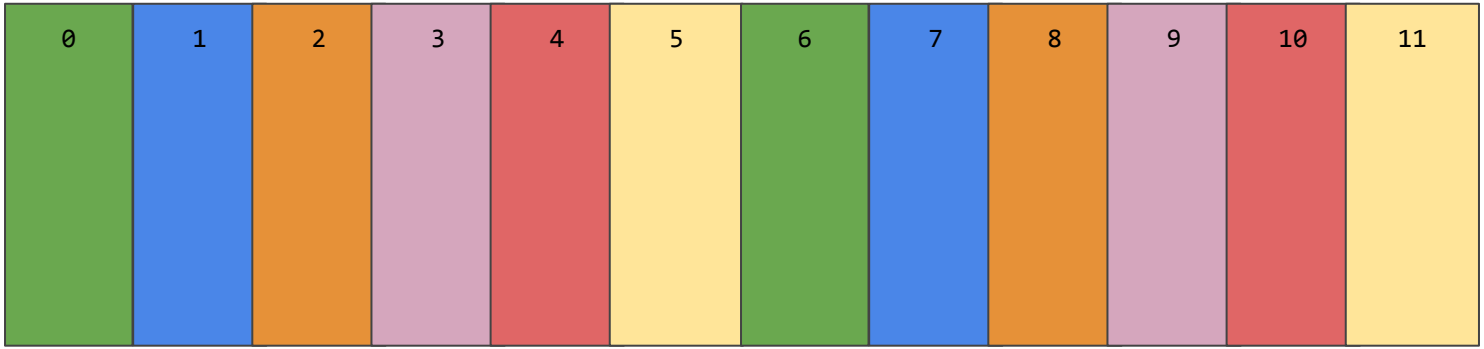
```
nBuckets: 12
nElems: 3
```

1. Make new buckets array

`n` Hash function

`% nBuckets` Compression function





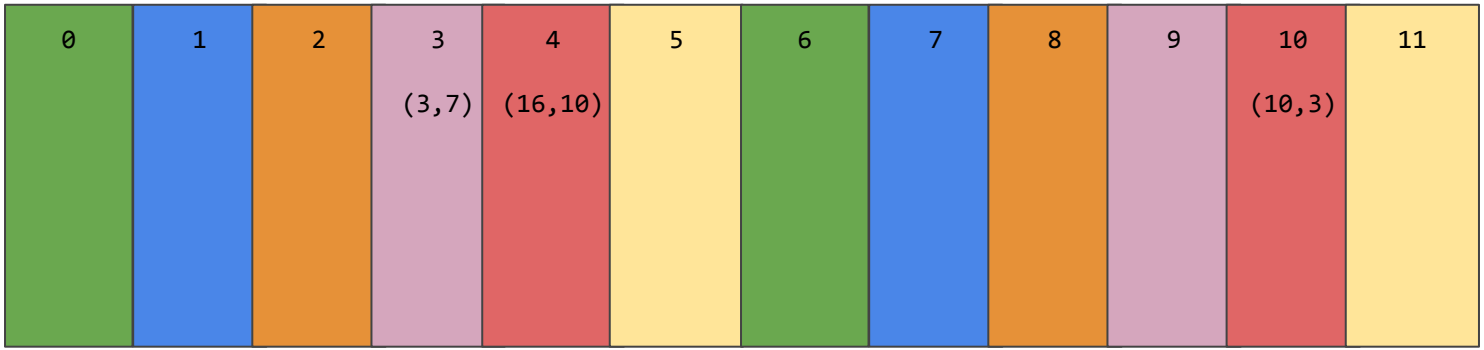
Rehashing

```
nBuckets: 12
nElems: 3
```

1. Make new buckets array
2. Put everything in new array

`n` Hash function

`% nBuckets` Compression function



How to not be hacked

Deep in the servers of facebook...

Email	Password
brahm@stanford.edu	banter
cgregg@stanford.edu	typewriters
cheson@stanford.edu	ILoveC++
elonmusk@tesla.com	electriccar
b.wayne@wayneenterprises.com	nanananabatman

Deep in the servers of facebook...

Email	Password
brahm@stanford.edu	banter
cgregg@stanford.edu	typewriters
cheson@stanford.edu	ILoveC++
elonmusk@tesla.com	electriccar
b.wayne@wayneenterprises.com	nanananabatman

A hacker could see these passwords!

Idea #1: store the length of the password instead

Email	Password length
brahm@stanford.edu	6
cgregg@stanford.edu	11
cheson@stanford.edu	8
elonmusk@tesla.com	11
b.wayne@wayneenterprises.com	14

Idea #1: store the length of the password instead

Email	Password length
brahm@stanford.edu	6
cgregg@stanford.edu	11
cheson@stanford.edu	8
elonmusk@tesla.com	11
b.wayne@wayneenterprises.com	14

Pros: Hackers can't see passwords in the database and can't reverse-engineer the passwords from their length

Cons: Any string of length 6 can log into Brahm's account!

Idea #1: store the length of the password instead

Email	Password length
brahm@stanford.edu	6
cgregg@stanford.edu	11
cheson@stanford.edu	8
elonmusk@tesla.com	11
b.wayne@wayneenterprises.com	14

Pros: one-way function

Cons: not well-distributed

We need a function that is...

- Deterministic

We need a function that is...

Deterministic

Fast

We need a function that is...

Deterministic

Fast

Well-distributed

We need a function that is...

Deterministic

Fast

Well-distributed

One-way

We need a function that is...

Deterministic

Fast

Well-distributed

}

One-way

#tbt

Properties of hash functions

1. Deterministic: the same input always gives the same output
2. Fast: Runs quickly
3. Well distributed output

Cryptographic hash functions

A hash function that is also one-way

Cryptographic hash functions

A hash function that is also one-way

One-way: extremely difficult to computationally reverse

Cryptographic hash functions

A hash function that is also one-way

One-way: extremely difficult to computationally reverse

Small changes in the password lead to large changes in the hash

"banter" hashes to `ef6571a62275adbb8b5cbd4ef9875a37`
"Banter" hashes to `c0027b4342d084ba1fb8a04d8e514ab2`

What that would look like

Email	Password hash
brahm@stanford.edu	ef6571a62275adbb8b5cbd4ef9875a37
cgregg@stanford.edu	5111a48e448f2a8912606c60d70a42e4
cheson@stanford.edu	fecb1ce853fb5ae06c41ec4ba06d115a
elonmusk@tesla.com	df0095aa19143a860e3ecb43ff533710
b.wayne@wayneenterprises.com	67988da12ad15278c841f0f06c69b209

What that would look like

Email	Password hash
brahm@stanford.edu	ef6571a62275adbb8b5cbd4ef9875a37
cgregg@stanford.edu	5111a48e448f2a8912606c60d70a42e4
cheson@stanford.edu	fecb1ce853fb5ae06c41ec4ba06d115a
elonmusk@tesla.com	df0095aa19143a860e3ecb43ff533710
b.wayne@wayneenterprises.com	67988da12ad15278c841f0f06c69b209

Passwords can't be re-engineered and every password has its own hash!

23rd February, 2017

Security

110

'First ever' SHA-1 hash collision calculated.

By John Leyden, Thomas Claburn and Chris Williams 23 Feb 2017 at 18:33

23rd February, 2017

Security

110

'First ever' SHA-1 hash collision calculated. All it took were five clever brains... and 6,610 years of processor time

Tired old algo underpinning online security must die now

By John Leyden, Thomas Claburn and Chris Williams 23 Feb 2017 at 18:33

One day later…

Security

49

Cloudbleed: Big web brands 'leaked crypto keys, personal secrets' thanks to Cloudflare bug

Heartbleed-style classic buffer overrun blunder

By Iain Thomson in San Francisco 24 Feb 2017 at 01:47

SHARE ▼

What you’re learning matters!

```
endl;
```