Hashmaps

Data Structures and Algorithms


HASHMAPS
CHANGED
MY LIFE

Hahaha what if we... had key-value pairs? In a list? ????!??!? hahaha jkjkjk... unless??

$$[\ (K_1,V_1),\ (K_2,V_2),\ (K_3,V_3),\ \ldots.\ ,\ (K_N,V_N)\ ]$$

For storing & looking up *important info*

[Jane: 781-239-5555;
Cassandra: 999-666-3333;
Anusha: 781-420-6969]

Or like IDK a dictionary..... IDK maybe Python has a data structure for this problem built in.....

~THE BIG PICTURE~

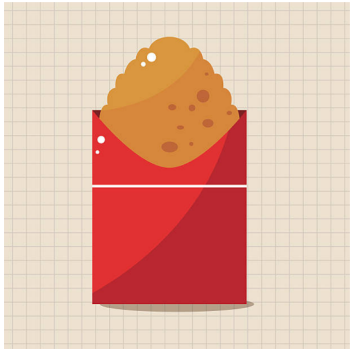
We Have: O(N) lookup time

We Want: O(1) lookup time

George Steelman after learning about hashmaps

To speed up lookup, we can use **hash functions** to access each key.

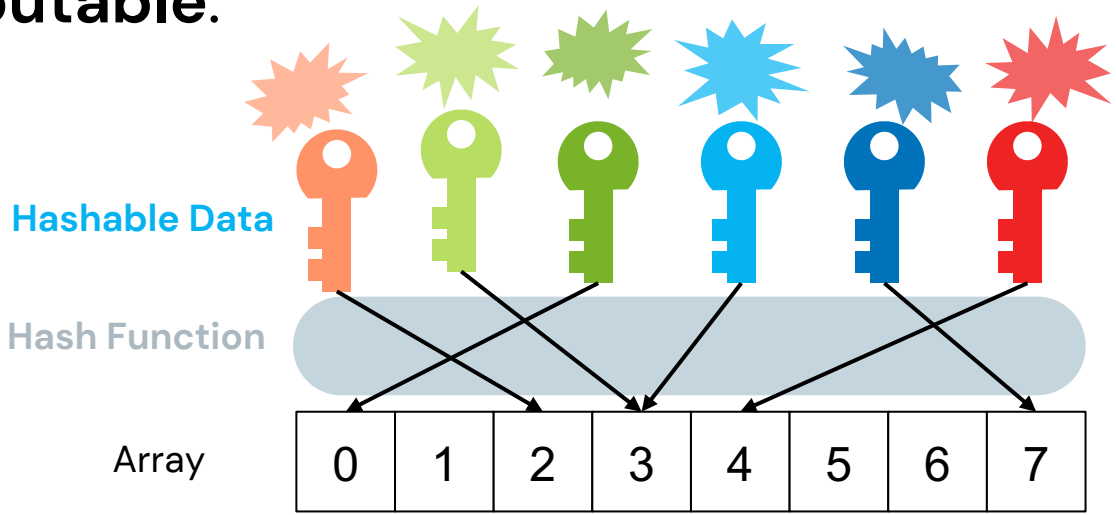
That way, we can quickly find the index where the key-value pair is stored by invoking the hash function.



What is Hashing?

Definition: Converting a piece of data to an array index in [0,M)

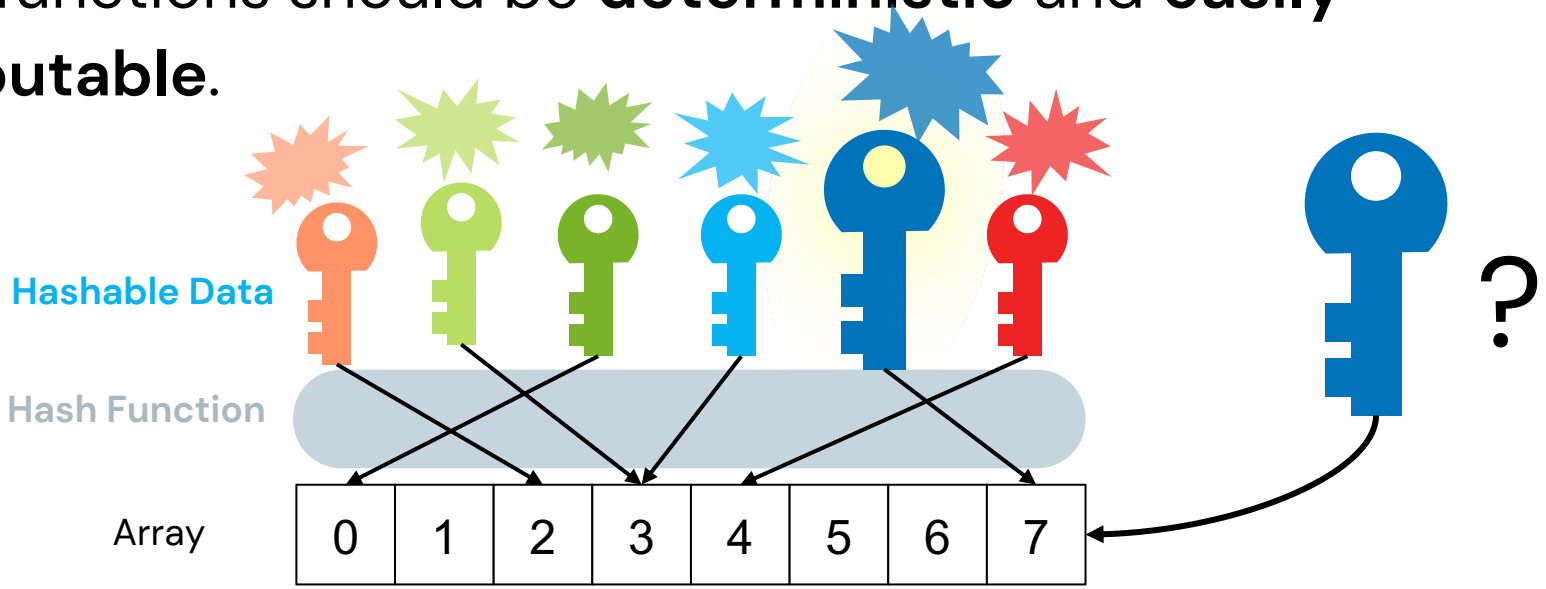
Hash functions should be **deterministic** and **easily computable**.



What is Hashing?

Definition: Converting a piece of data to an array index in [0,M)

Hash functions should be **deterministic** and **easily computable**.



Example

A classic: modular arithmetic. To hash any number ***i*** from ***[0, M)***, use the hash function ***i % M***.

Example: hashing this list of **key**-value pairs using **mod_3:**
[(**0**,3), (**5**,6), (**9**,21)].

To do this, apply our hash function to each **key**.
[0: (**9**,21) ➜ (**0**,3) ➜ null,
1: null,
2: (**5**,6) ➜ null]

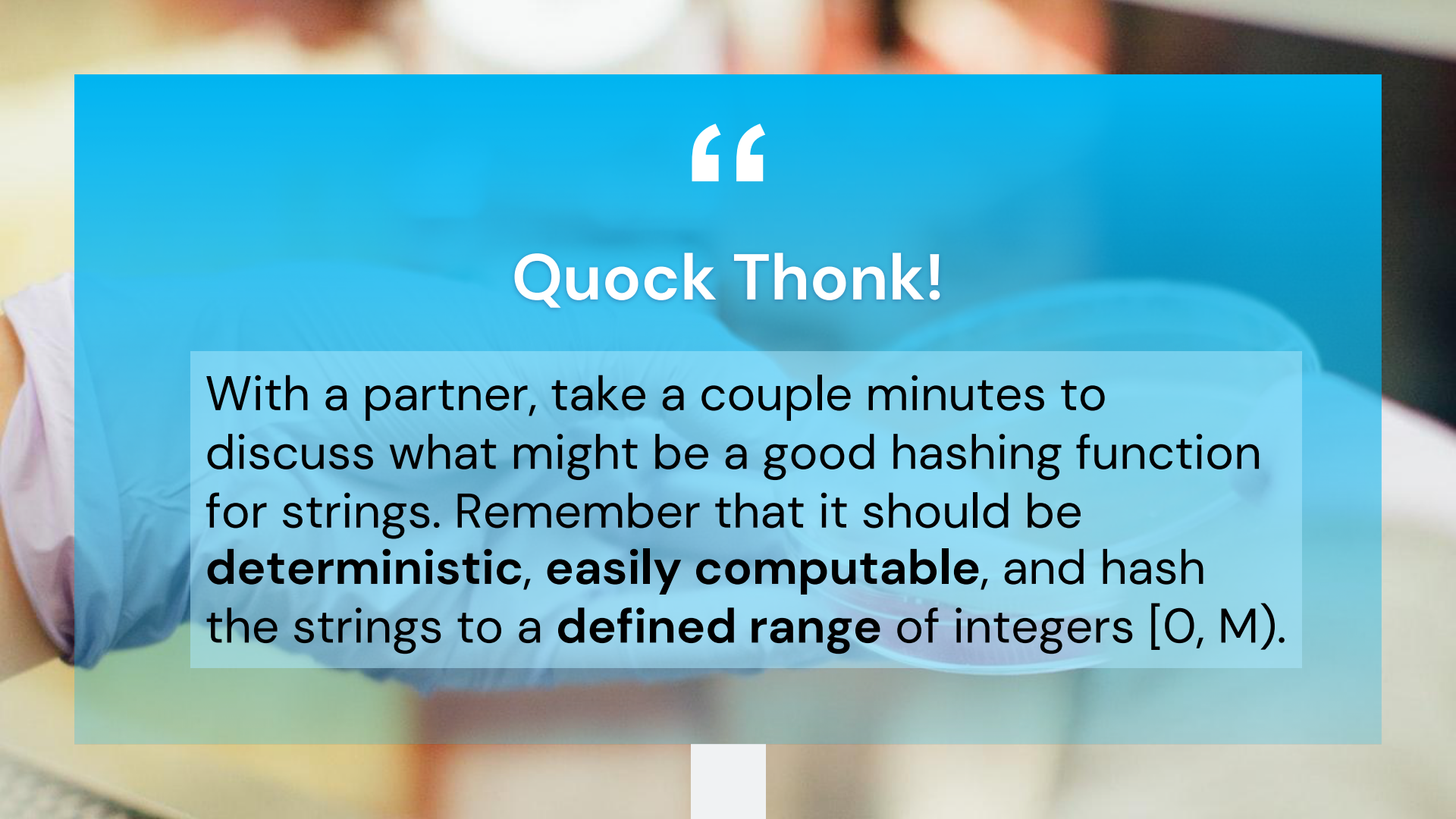
Here, both 0 and 9 hash to the same index: 0; this is called a **collision**. There are several **collision resolutions**, one of which is chaining (used here). Notice that at each index, we store a linked list, and we append to the head of the list if something hashes to that index.

Oh no a collision!

Generally, it should be **incredibly unlikely** two values hash to the same integer.

A good hash function will yield a **uniform distribution** of indices i.e. it is as likely to hash to 1 as it is to 5. But, it should not be random.

Python has a `hash()` method.



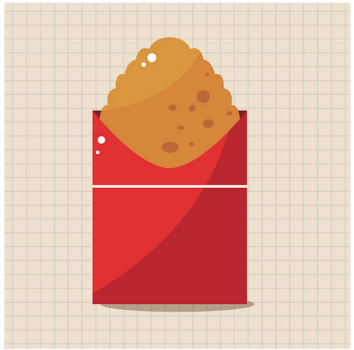
Quock Thonk!

With a partner, take a couple minutes to discuss what might be a good hashing function for strings. Remember that it should be **deterministic**, **easily computable**, and hash the strings to a **defined range** of integers [0, M).

Hashing puts (Key, Value) pairs in *BUCKETS*

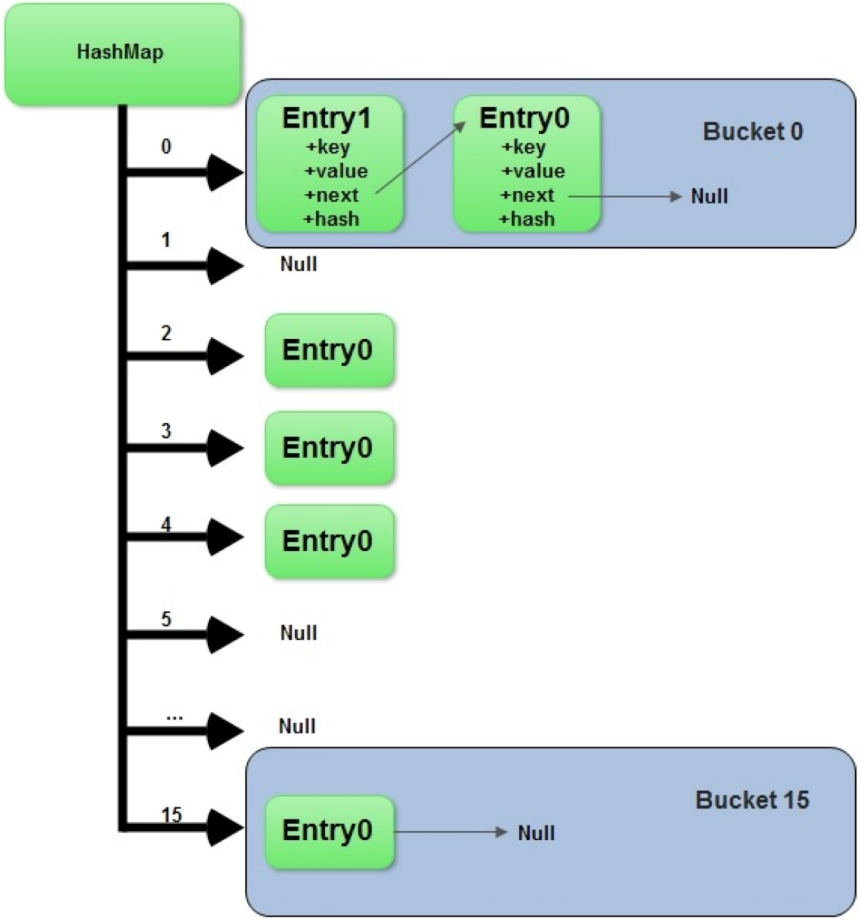
We split the original array of ***N*** (K,V) pairs into ***M*** buckets (subarrays) of (K,V) pairs. So each of the ***M*** buckets has depth ***N/M*** (let's call this $\boldsymbol{\alpha}$).

Now we operate in O($\boldsymbol{\alpha}$), which is constant (for now)!


Buckets?

Buckets are a way to refer to linked lists that store the key-value pairs.



$[(K_1,V_1), (K_2,V_2), (K_3,V_3), \ldots , (K_N,V_N)]$

Divide all N (K,V) pairs into **M** buckets

```
maps = [
        0: [ length α ],
        1: [ length α ],
        ...
        M-1: [ length α ]
       ]
```

Choose Your Bucket

```
def chooseMap(key):
        # hash the key to get the index of the correct bucket
        index = hash(key) # apply some hash function
        return maps.get(index) # return the correct bucket
```

This helper method picks out the bucket that the (K,V) pair will go in. All it does is hash the key and returns the correct bucket.

Summing up what we have so far

Given a **(K,V)**, we find **m = hash(K)**

Once we know which bucket **(K,V)** goes into, we put it into that bucket:
`chooseMap(K).append(K,V)`

The number of buckets, **M**, is constant. If **N** were to be constant, we would have O(**N/M**) = O(**α**) = O(1) lookup time!

Wowee!

Oh wait we're dumb...

But what if N GROWS (and it does)?

We need dynamic resizing.

When do we resize? When the average entries per bucket is higher than our designated α.

So if α = 2, and we have 7 entries over 3 buckets, then 7/3 > 2 → `resize();`

When resizing, every key-value pair is hashed to the new hashmap. This means when **N** grows, **M** also grows, thus **N/M** = α stays constant → operations are always O(α) or constant time.

HashMaps in the Wild

Python has some built-in data structures which use hashing.

Two common ones are dictionaries and sets.

d = dict() (or) d = {“Key”: Value, … }

s = set() (or) s = {“Item1”, “Item2” … }

Dictionaries store key-value pairs. Sets store unique items and

Thanks!!!

Questions? Email George Steelman at **gsteelman@olin.edu**