Hash Tables

Also known as Hash Map.

Built-in data structures in pretty much every language. Because of their speed, hash tables are used pretty commonly.

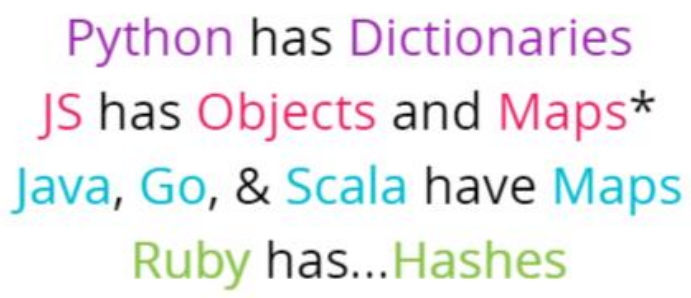
Hash Tables are used to store key-value pairs. Unlike arrays, where there only exists numeric ordered keys.

In Hash Tables, keys are not ordered and are not restricted to numeric.

Unlike arrays, hash tables are fast for all of the following operations:

Finding, Removing and Adding new values.

If you have sequential data, you could use an array but a lot of the time, our data may not fit that pattern. So, we could use a hash map for this type of data.



Python has Dictionaries
JS has Objects and Maps*
Java, Go, & Scala have Maps
Ruby has...Hashes

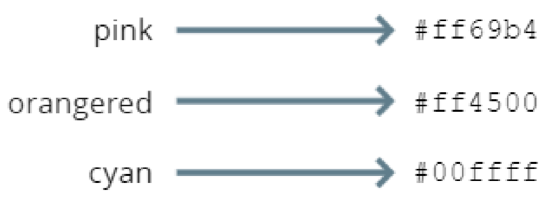
How would we implement our own version?

Imagine we want to store some colors:

We could store them like this: [“#ff69b4”, “#ff4500”, “00ffff”]

But it is not a great way of storing them. It would be if we have to just pick some out of them randomly. But if we need particular colors at a particular moment, it’s not great.

It would be nice if instead of using indicies to access the colors, we could use more human-readable keys.



colors[“cyan”] is way better than colors[2] in terms of readability.

How can we get human readability and computer redability.

Computers don’t know how to find an element at index cyan.

This is what hash tables are for!

To implement hash table, we’ll be using an array.

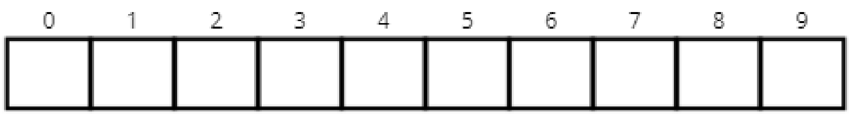
We need a way to take ‘cyan’ and turn it into a number to which index we can store that item in the array. So, in order to look up values by key, we need a way to convert keys into valid array indicies.

So if our array has 100 portions, we need a way to convert ‘cyan’ into a number between 0 an 99 and then we store the data at that number.

There are functions that help us do that and are called hash functions also known as hashing functions.

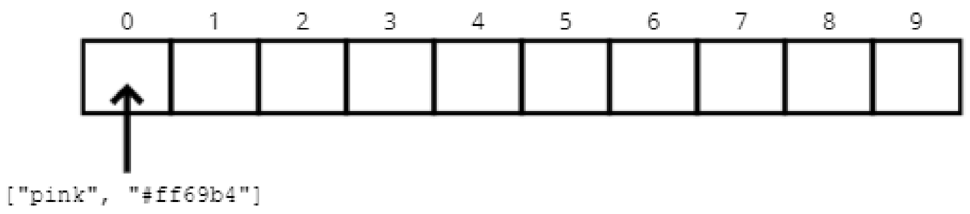
The idea is that we pass in a string, it gives us a number and every time we pass in the same string it should give us that same number.

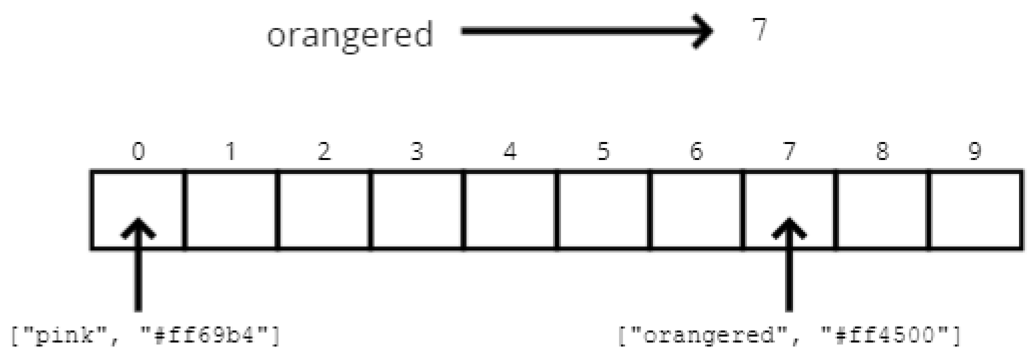
HASHING CONCEPTUALLY

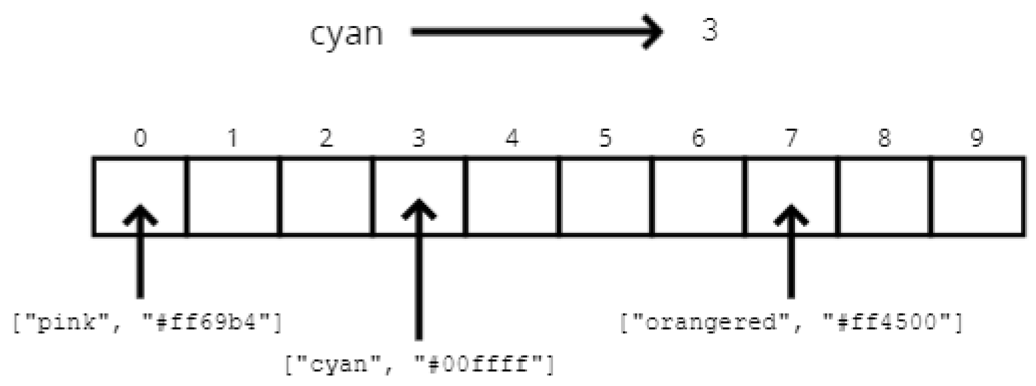


If I pass in the string ‘pink’ to the hash function, lets say it gives zero:

So we store pink as the key and #ff69b4 as the value at index 0







Now if we want to retrieve the value of index ‘cyan’, (our code still doesn’t know what index cyan is), so we pass ‘cyan’ to our hash function again, and it gives the same number it gave before.

Then I go to index 3, and get required value.

That is why it’s so important that we always get the same number back!

Otherwise, its of no use.

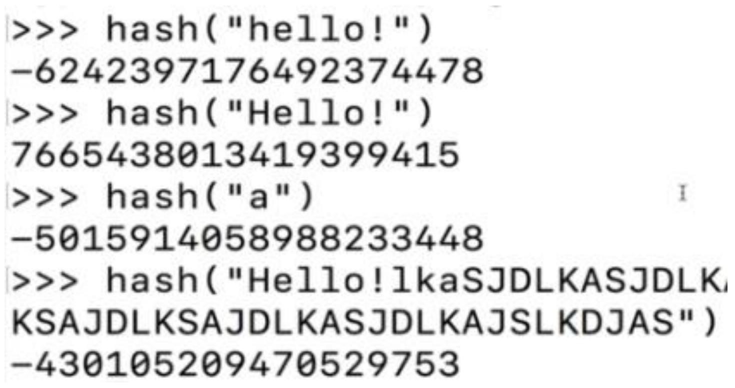
Writing a hash function that takes string, or any piece of data, image, pdf, video and it somehow spits out a number that is always consistent, it’s not easy!

Hash functions have uses all over the world, in the Internet, privacy, in hash tables, in cryptography and computing in general.

Basic hash function is a function that takes data of arbitrary size (thousands or millions of charactars) and it's going to spit out data of a fixed size.

It basically Map an input to an output of a fixed size.

Python has implementation of a hash function.



```
>>> hash("hello!")
-6242397176492374478
>>> hash("Hello!")
7665438013419399415
>>> hash("a")
-5015914058988233448
>>> hash("Hello!lkaSJDLKASJDLK
KSAJDLKSAJDLKASJDLKAJSLKDJAS")
-4301052094705297531
```

The number it gives back is of same size, no matter what the input is!

An important thing is that we can't work bacwards. If all we have is output number, we can't just figure out exactly what generated that number.

WHAT MAKES A GOOD HASH?

(not a cryptographically secure one)

➔ **Fast** (constant time)

When we insert something into our hash table it needs to be hashed. Also when we go and fetch, update, or remove it (everytime we try to access it afterwards) we still have to run that hash function again.

➔ **Doesn't cluster** outputs at specific indices, but distributes uniformly

We don't want a hash function that always give us the middle of array of start of end. It's useless if every element is stored at the same spot. We might have collisions, that's okay we'll get to that later but we want it to be pretty evenly spread out.

➔ **Deterministic** (Same input yields same output)

We don't want there to be any uncertainty or multiple outputs. We want the same input to always give the same output. So our function shouldn't use something like Math.random() as it would make our number non-deterministic.

hash(“string”, lengthOfAarray)

    store our hash table in an array that is (lengthOfAarray) items long.

hash(“cyan”, 100)

    This will map ‘cyan’ to be a number between 0 and 99.

How would we just convert ‘cyan’ into a number?

There are many many ways of doing that.

We could just take length of ‘cyan’ and that’s it!
But the problem is that a lot of strings have the exact same length and we’re gonna have very clustured data.

Another way of ding this is to use the underlying UTF 16 character codes for each character. So every character has a numberic value associated with it and we can get that using .charCodeAt() method in JS.

“a”.charCodeAt(0) → 97

We could add these numbers together for a given string.

And if you subtract 96, that will give us alphabetic position.

“a”.charCodeAt(0) – 96	→	1
“b”.charCodeAt(0) – 96	→	2
“c”.charCodeAt(0) – 96	→	3
“z”.charCodeAt(0) – 96	→	26

We’re just subtracting 96 so that it would be more understandable for us. Otherwise there’s no need for that.

```
let total = 0;
total += "hello".charCodeAt(0) - 96;
total += "hello".charCodeAt(1) - 96;
total += "hello".charCodeAt(2) - 96;
total += "hello".charCodeAt(3) - 96;
total += "hello".charCodeAt(4) - 96;

console.log(total);      →     52
```

Now how do we consider lengthOfAarray in

hash("string", lengthOfAarray)

Simple way is to use modulus.

```
console.log(total % 10)   →   2
```

Here's a hash function that only work on strings:

```
function hash(string, arrayLength) {
   let total = 0;
   for (let c of string) {
       total += c.charCodeAt(0) - 96;
   }
   return total;
}

hash("hello" , 10)      →     2
hash("pink"  , 10)      →     0
hash("orange", 10)      →     0
hash("cyan"  , 10)      →     3
```

REFINING OUR HASH

Problems with our current hash

- ➔ **Only Hashes strings**

  It's not gonna work with other types because we use string method. But <u>we don't worry about this for now</u> because that is beyond our scope.

- ➔ **Not constant time**

  The time that it takes depends on the length of the string.

- ➔ **Could be a little more random**

  Our data can be clustered relatively easy. We do have ways of making it slightly more scattered. (using prime numbers)

In our prev hash function, as our string goes over thousand or million characters, it'll take a lot of time to give us our final number. So, we're not going to loop every single character.

We we could do is to add a max length

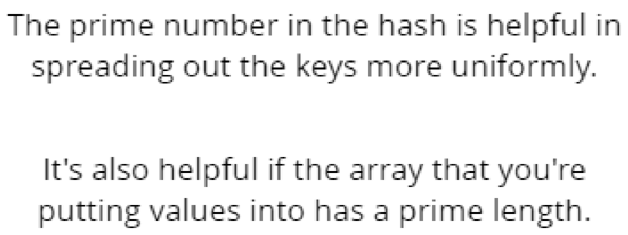
```
Math.min( string.length, 100 )
```

Now if our key is just 30 characcters we're looping 30 times but if our key is of thousand characters then we're only looping for 100 times.

Not a great solution but will work for us.

If you know that all of your data follows the pattern where first 100 numbers are same then maybe we would tweak our algorithm to start at the end of the string.

The second thing we’re going to do is to add in prime numbers. Hash functions always take advantage of prime numbers as it reduces collusion.



The prime number in the hash is helpful in spreading out the keys more uniformly.

It's also helpful if the array that you're putting values into has a prime length.

```
function hash(string, arrayLength) {
    let total = 0;
    const PRIME_NUMBER = 31;
    for (let i = 0; i < Math.min(string.length, 100); i++) {
        let char = string[i];
        let value = char.charCodeAt(0) - 96;
        total = (total * PRIME_NUMBER + value) % arrayLength
    }
    console.log(total);
    return total;
}

 hash("hello" , 13)      →    7
 hash("pink"  , 13)      →    5
 hash("orange", 13)      →    10
 hash("cyan"  , 13)      →    5
```

We’re not really gonna see any diffrence whether our distribution is better in such a small data set where the array length is 13. You’d want to use a larger prime number ideally.

This hash function is still not that great but it works fine in our kind of data set and for the sake of simplicity we’re going to use this for building our hash table.

As you can see there’s still collusion occuring with ‘pink’ and ‘cyan’. So do we handle it?

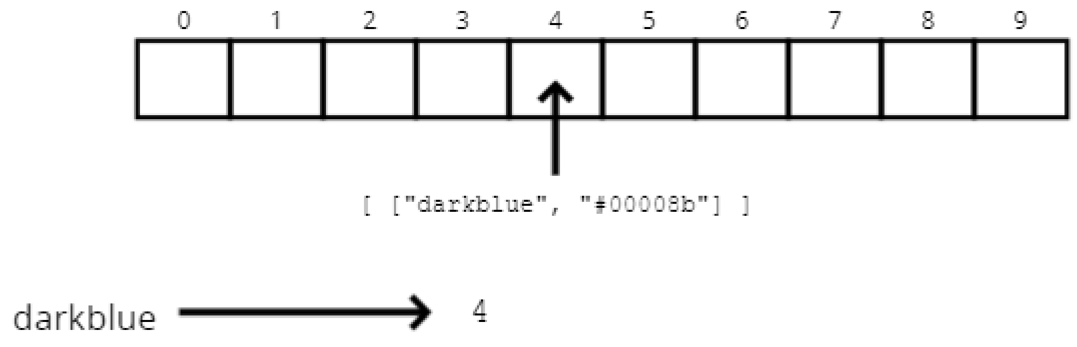
Even with very large array and a great hash function, collisions are inevitable.

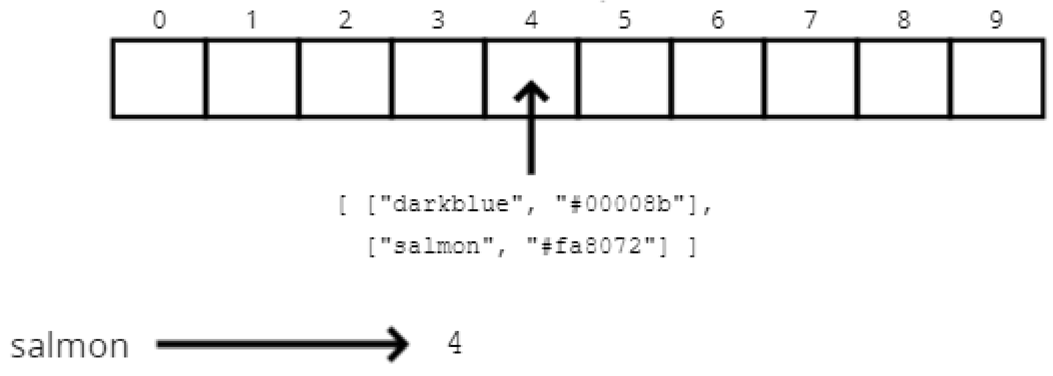
There are many strategies for dealing with collisons, two are:

- → Separate Chaining
- → Linear Probing

In Separate Chaining, we store the pieces of data at the same spot using another nested data structure.

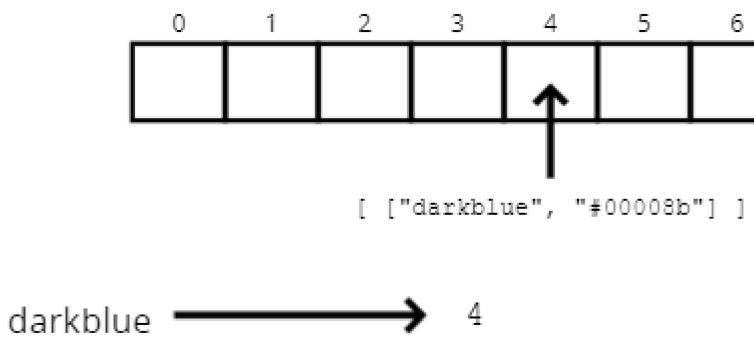
This allows us to store multiple key-value pairs at the same index.

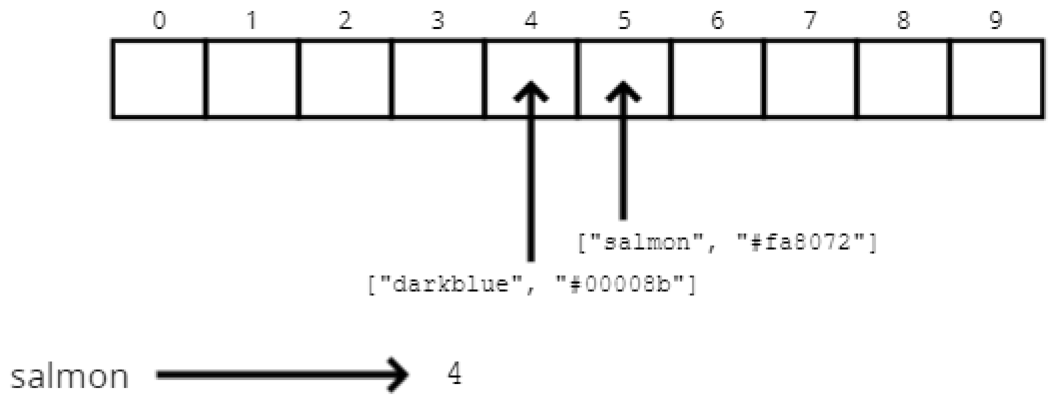




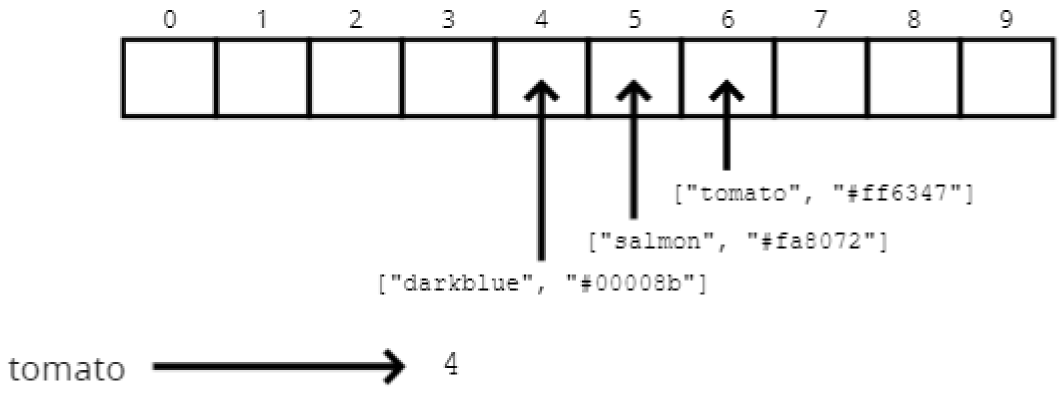
Now if we want to retrieve vlaue of salmon, we'd hash salmon, gets 4, go to index 4, loop through and look of salmon and once we find it we look for its value.

In Linear Probing, we only store one piece of data at each position and when there is a collision we look ahead for the next empty slot. This allows us to store a single key-value at each index.





As we get 4 again, we look at 4 and there's something there so we look at the next empty slot which to be 5.



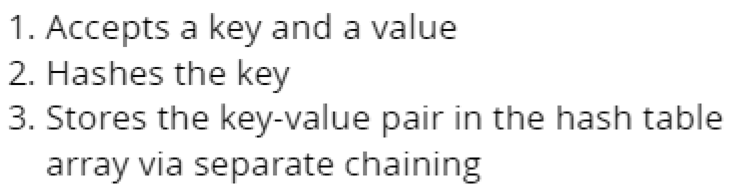
We're going to implement separate chaining in our hash table as it allows us to have more data than the length of our table because it does nesting.

But when we talk about linear probing, that only allows us to store data equal to the length of our table because we're only storing one thing at each position.

```javascript
class HashMap {
    constructor(size = 4) {
        this.keyMap = new Array(size);
    }

    hash(key) {
        let total = 0;
        const PRIME_NUMBER = 31;
        for (let i = 0; i < Math.min(key.length, 100); i++) {
            let char = key[i];
            let value = char.charCodeAt(0) - 96;
            total = (total * PRIME_NUMBER + value) % this.keyMap.length;
        }
        return total;
    }
}
```

SET

.set(key,value)



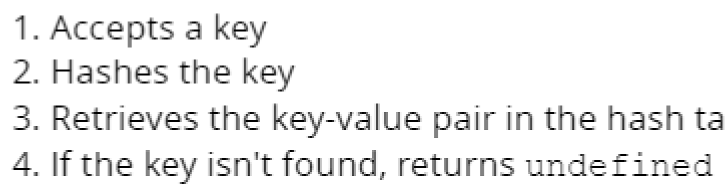
1. Accepts a key and a value
2. Hashes the key
3. Stores the key-value pair in the hash table array via separate chaining

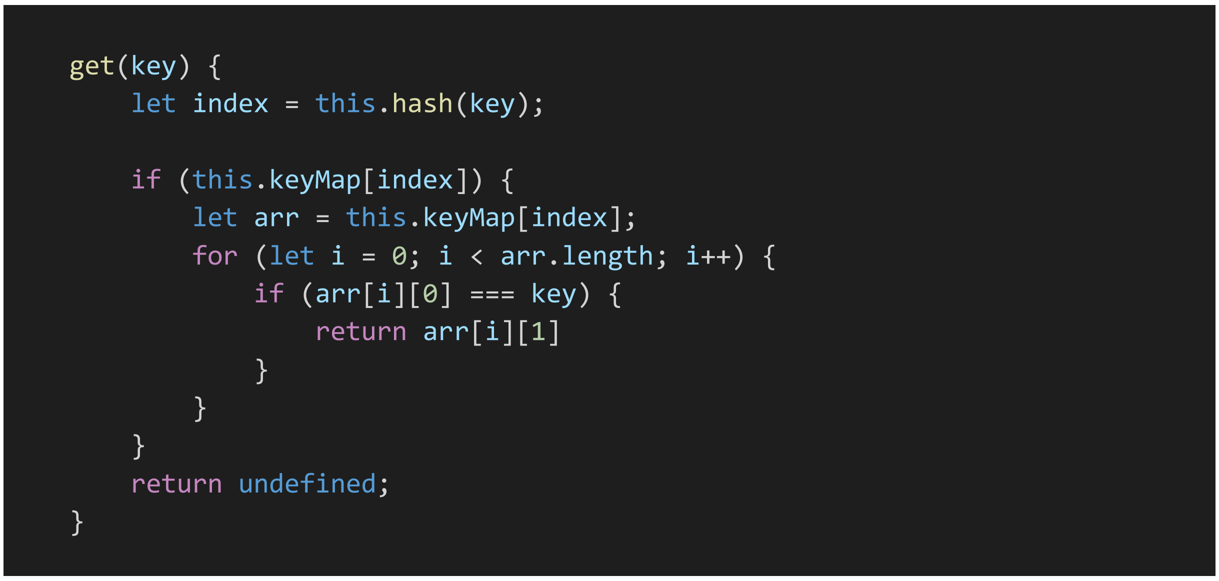
Plus we have to make sure that we don't let our hashmap duplicate keys. If user tries to enter duplicate key, then we would just update(overwrite) that key's value.

```javascript
    set(key, value) {
        let index = this.hash(key);
        if (!this.keyMap[index]) { // if there's nothing there
            this.keyMap[index] = [];
        } else {
            let arr = this.keyMap[index];
            for (let i = 0; i < arr.length; i++) {
                if (arr[i][0] === key) {
                    arr[i][1] = value;
                    return;
                }
            }
        }
        this.keyMap[index].push([key, value])
    }
```

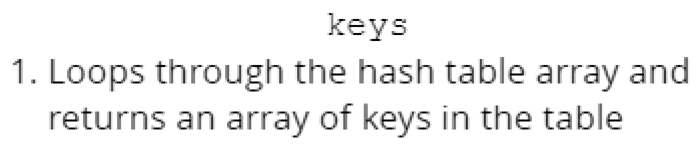
GET

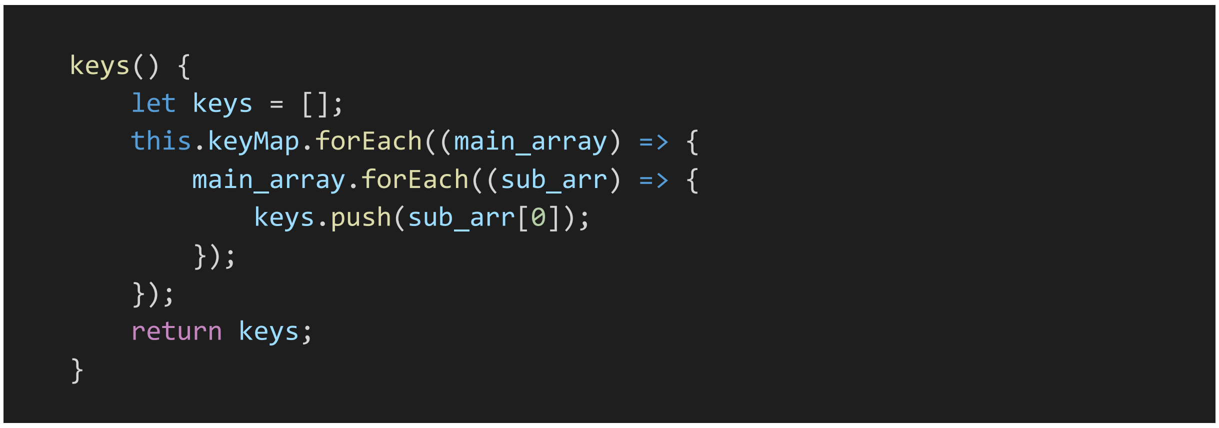
.get(key)


1. Accepts a key
2. Hashes the key
3. Retrieves the key-value pair in the hash table
4. If the key isn't found, returns `undefined`

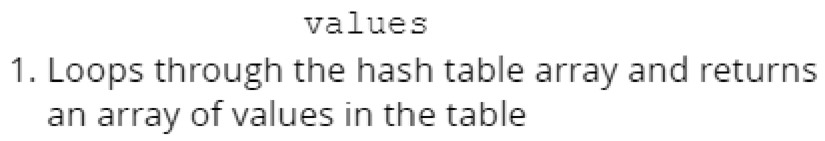

```
get(key) {
    let index = this.hash(key);

    if (this.keyMap[index]) {
        let arr = this.keyMap[index];
        for (let i = 0; i < arr.length; i++) {
            if (arr[i][0] === key) {
                return arr[i][1]
            }
        }
    }
    return undefined;
}
```

KEYS


`keys`
1. Loops through the hash table array and returns an array of keys in the table


```
keys() {
    let keys = [];
    this.keyMap.forEach((main_array) => {
        main_array.forEach((sub_arr) => {
            keys.push(sub_arr[0]);
        });
    });
    return keys;
}
```

VALUES

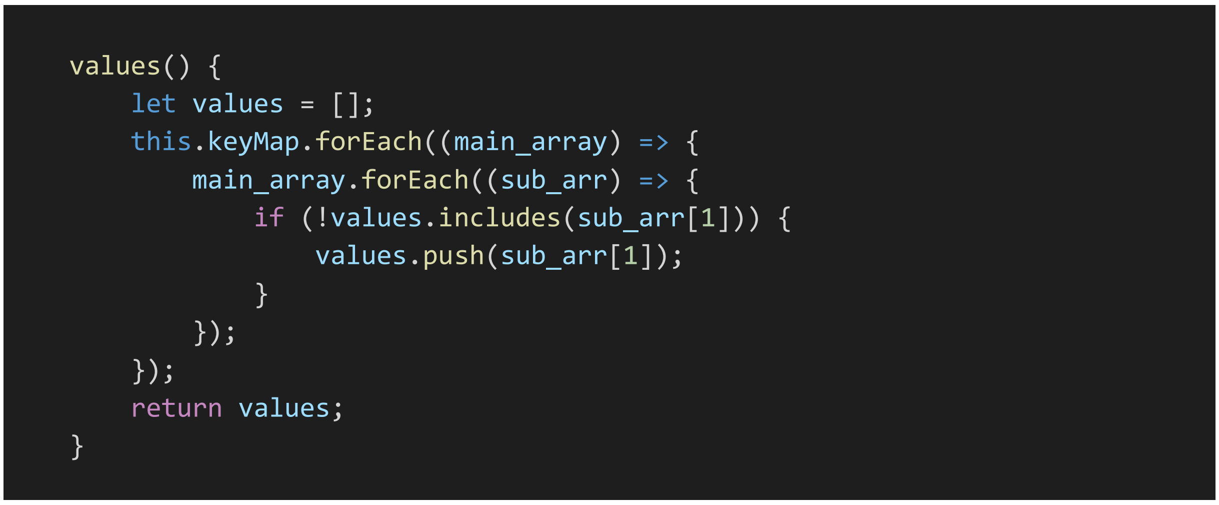


values

1. Loops through the hash table array and returns an array of values in the table

How will we handle duplicate data?
Well we didn’t had to take care of duplicate keys in our previous keys method as our keys were already unique.

But value can have duplicates. So we have to make sure that in values function we don’t return any duplicates.



```
values() {
    let values = [];
    this.keyMap.forEach((main_array) => {
        main_array.forEach((sub_arr) => {
            if (!values.includes(sub_arr[1])) {
                values.push(sub_arr[1]);
            }
        });
    });
    return values;
}
```

Time Complexity

Insertion: O(1)

Deletion: O(1)

Access: O(1) using a given key, finding the corresponding value

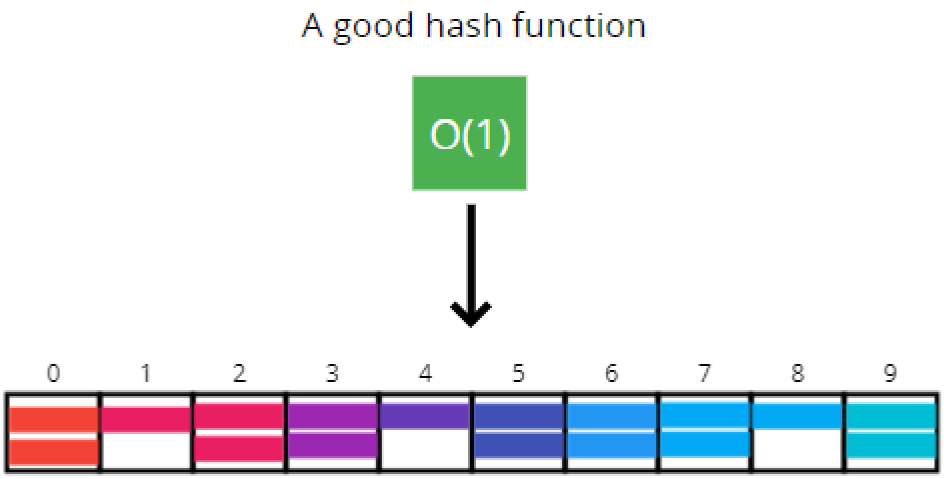
In case of searching,

If we’re searching for a key, it can be constant time. O(1)
But if we’re searching for a value, then we go through every single item and check. So that would be O(n)

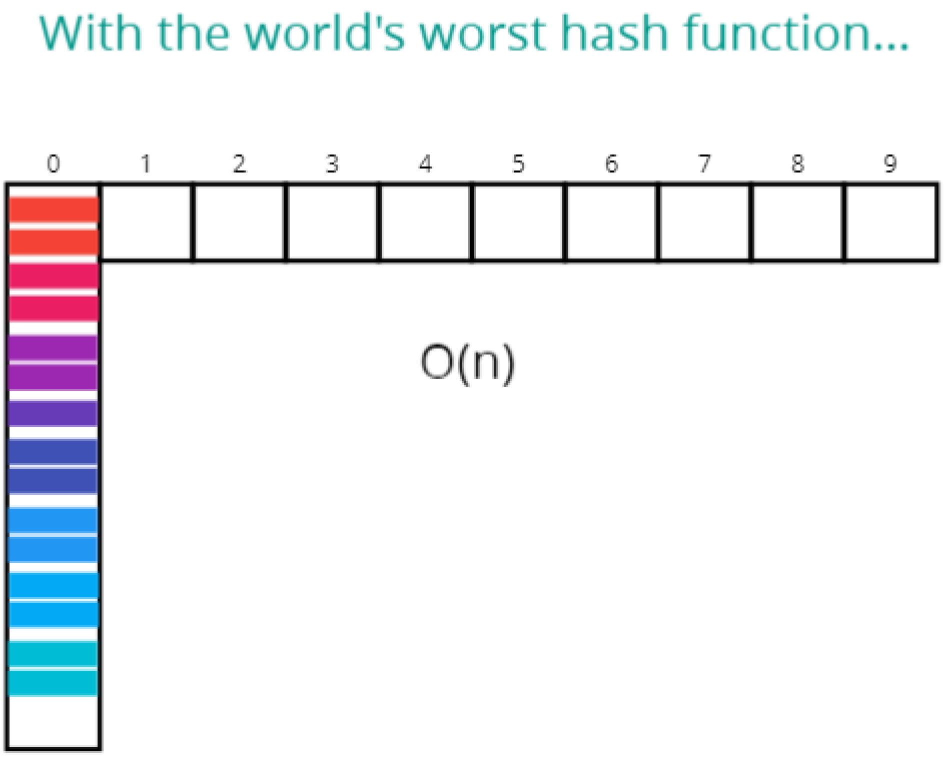
This really comes down to how good your hash funcion is,

how fast the hash function itself is!

how evenly it distributes things (minimize the no of collisions)



VS



O(n) in this case because even if hash function itself is constant time, if it’s distributing everything in one spot, then it’s basically just a list.

Recap

- Hash tables are collections of key-value pairs
- Hash tables can find values quickly given a key
- Hash tables can add new key-values quickly
- Hash tables store data in a large array, and work by *hashing* the keys
- A good hash should be fast, distribute keys uniformly, and be deterministic
- Separate chaining and linear probing are two strategies used to deal with two keys that hash to the same index
- When in doubt, use a hash table!when key-value pair is needed