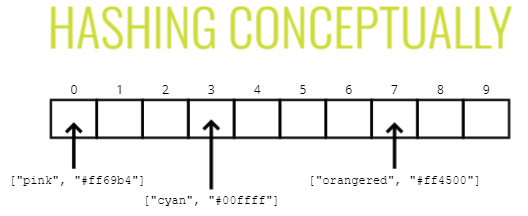
**Hash Table / Hash Map**

****

**WHAT IS A HASH TABLE?**

Hash tables are used to store key-value pairs.

They are like arrays, but the keys are not ordered.

Unlike arrays, hash tables are fast for all of the following operations: finding values, adding new values, and removing values!

Because of their speed, hash tables are very commonly used!

Nearly every programming language has some sort of hash table data structure

Python has Dictionaries

JS has Objects and Maps\*

Java, Go, & Scala have Maps

Ruby has...Hashes

\* Objects have some restrictions, but are basically hash tables

To implement a **hash table**, we'll be using an **array**.

In order to look up values by key, we need a way to convert keys into valid array indices.

A function that performs this task is called a **hash function**.

**Use of Hash Function** is almost everywhere. Places of using Hash Function are:

* Hash Tables
* To protect & store our data
* Authenticate you when you login in on a website
* They’re used in cryptocurrencies and cryptography in general.

There’s many, many types of hash functions, including subgroup called cryptographic hash functions.

The basic definition of a has function is just a function that takes data of arbitrary size, whether it’s 1000 characters or a million characters. And it’s going to spit out data of a fixed size.

The data return by a has function:

* Has fixed sized or similar sized data no matter what the input is.
* In most cases of a hash function, we can’t work backwards. This is usually a one way function.

So we’re not going to use such kind of hash function. We’re going to write our own simple one.

**WHAT MAKES A GOOD HASH function?**

* **Fast** (i.e. constant time)
* Doesn't **cluster outputs** at specific indices, but distributes uniformly (Having minimum number of collisions)
* **Deterministic** (same input yields same output), we need to get exact same output every time with the same input.

**Simple Hash Example** (hash that works on strings only):

function hash(key, arrayLen) {

let total = 0;

for (let char of key) {

**// map "a" to 1, "b" to 2, "c" to 3, etc.**

***//Gives position of Alphabets (A/a to Z/z)***let value = char.charCodeAt(0) – 96

total = (total + value) % arrayLen;

}

return total;

}

hash("pink", 10); ***//execute this hash***  
hash("orangered", 10);   
hash("cyan", 10);

**Output:**0 ***//Index of ‘pink’(key) in the array of length 10.***  
7 ***//Index of ‘orangered’(key) in the array of length 10.***  
3 ***//Index of ‘cyan’ (key) in the array of length 10.***

**Problems with our current hash**

1. Only hashes strings (we won't worry about this)
2. Not constant time - linear in key length
3. Could be a little more random

**Hashing Revisited**

function hash(key, arrayLen) {

let total = 0;  
let WEIRD\_PRIME = 31;

for (let i = 0; i< Math.min(key.length, 100); i++) {  
 let char = key[i];

***//Gives position of Alphabets (A/a to Z/z)***let value = char.charCodeAt(0) – 96

total = (total \* WEIRD\_PRIME + value) % arrayLen;

}

return total;

}

hash("pink", 10); ***//execute this hash***  
hash("orangered", 10);   
hash("cyan", 10);

**Output:**0 ***//Index of ‘pink’(key) in the array of length 10.***  
7 ***//Index of ‘orangered’(key) in the array of length 10.***  
3 ***//Index of ‘cyan’ (key) in the array of length 10.***

**Prime numbers? What.**

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.

You don't need to know why. (Math is complicated!) But here are some links if you're curious.

[Why do hash functions use prime numbers?](https://computinglife.wordpress.com/2008/11/20/why-do-hash-functions-use-prime-numbers/)

[Does making array size a prime number help in hash table implementation?](https://www.quora.com/Does-making-array-size-a-prime-number-help-in-hash-table-implementation-Why)

Basically, Prime Numbers helps in reducing number of collisions and make our data more distributed as compare to other numbers.

**Dealing with Collisions**

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

Here, **Collisions means** hashing multiple keys to the same index of the array.

Ex:

function hash(key, arrayLen) {

let total=0, primeNum = 31;

for (let i = 0; i < Math.min(key.length,100); i++) {

let char = key[i],

value = char.charCodeAt(0) - 96;

total = (total\*primeNum + value) % arrayLen;

}

return total;

}

hash("cyan", 13);

hash("pink", 13);

**Output:**  
5 ***//Index of cyan***  
5 ***//Index of pink***

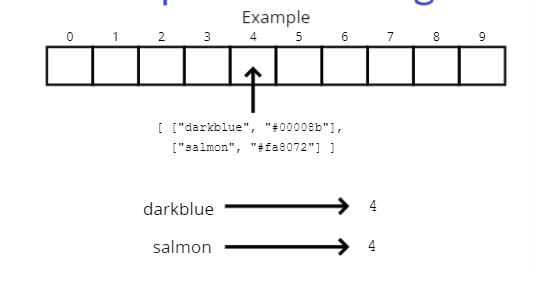
Here, both key value collide with each other. Means, both keys went into the same box/variable of the array. That’s why indices are same.

There are many strategies for dealing with collisions, but we'll focus on two:

1. Separate Chaining
2. Linear Probing

Separate chaining and linear probing are two strategies used to deal with two keys that hash to the same index.

1. **Separate Chaining**

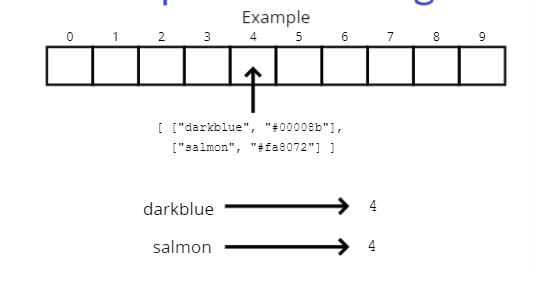


With separate chaining, at each index in our array we store values using a more sophisticated data structure (e.g. an array or a linked list).

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

It allow us to have more data than the length of our table.

1. **Linear Probing**



With linear probing, when we find a collision, we search through the array to find the next empty slot.

Unlike with separate chaining, this allows us to store a single key-value at each index.

It allow us to have data equal to the length of our table.

**Set / Get Method Implementation in hashMap**

**Set (To implement Set by Separate Chaining) Pseudo Code**

* Accepts a key and a value
* Hashes the key (Get Index)
* Stores the key-value pair in the hash table array via separate chaining

**Approach:**

-- We're going implement hashTable in an array and implement set() method to store keys & their corresponding values in hashMap at some index and for dealing with collision we’ll use Separate Chaining.

Due to Separate Chaining, data/element would be > length of array.

-- Create an array of Size(let say, 4) store it in 'keyMap'

-- Create a hash function called '\_hash' which are going to accept key and return the random index number inside array (keyMap) to store that key.

-- To push the previous key along with a value in 'keyMap' and deal with collision we're going to Create a 'set' Method.

Empty keyMap of size(4) --> ['', '', '', '']

call hash function '\_hash' to get the index of key and store that index to 'index' variable.

If(there is no key present at that particular index or say empty)

Just create an empty Array at that index

Empty array created at index[2] --> ['', '', '[]', '']

Otherwise (if there is something at that index of keyMap)

Just push the key and value in the form of an array.

Empty array created at index[2] --> ['', '', '[[key,value]]', '']

**Code Implementation for Set Method by Separate Chaining:**

class HashTable {

constructor(size = 4) {

this.keyMap = new Array(size);

}

\_hash(key){

let total = 0,

primeNum = 31;

for (let i = 0; i < Math.min(key.length, 100); i++) {

let char = key[i],

value = char.charCodeAt(0) - 96;

total = (total\* primeNum+value)%this.keyMap.length;

}

return total;

}

set(key, value){

let index = this.\_hash(key);

if(!this.keyMap[index]){

**// Set it to an empty array**

this.keyMap[index] = [];

}

this.keyMap[index].push([key, value]);

}

get(){

}

}

let hashtable = new HashTable();

hashtable.set('hello world', 'goodbye!!');

hashtable.set('dogs', 'are cool');

hashtable.set('cats', 'are fine');

hashtable.set('i love', 'pizza');

hashtable.set('hi', 'bye');

hashtable.set('french', 'fries');

**// ( key, value)**

**Output:-**

HashTable {keyMap: Array(4)}

keyMap: Array(4) **//Total size of the keyMap --> 4**

0: [Array(2)] **//At O index, there is 1 array of size(2).**

1: (3) [Array(2), Array(2), Array(2)] **//At 1 index, there is 3 array of size(2). This** is called collision

2: [Array(2)]

3: [Array(2)]

length: 4

Total length of keyMap --> 4

Total element/data present in keyMap --> 6

**Get (To get value of key) Pseudo Code**

* Accepts a key
* Hashes the key
* Retrieves the key-value pair in the hash table
* If the key isn't found, returns undefined

**Approach:**

-- We're going implement hashTable in an array and get the value of the key. Ex: Key ('yellow'): value ('#FFFF00')

Here, All keys along with their corresponding values has already been stored in the hashtable using 'set()' Method.

-- Create an array of some Size(let say, 17) store it in 'keyMap'

-- Now to get the value of a particular key, we're going to call 'get(key)' method by passing the required key.

Store the index of the required key in 'index' variable

Check, If(their is any element found in keyMap at that index? this.keyMap[index])

If Found,

So, their might be presence of multiple array(consisiting multiple element/data) due to collision

Here, Start looping over the array(might consisiting multiple array) at that index till its length. < this.keyMap[index].length

if(We got the key(that would be at the 0th index of any array)===key)

return the value of that particular key (that would be at the 1st index of that array.)

If Not Found,

return undefined.

**Code to implement Get Method:**

class HashTable {

constructor(size=17) {

this.keyMap = new Array(size);

}

\_hash(key){

let total = 0,

primeNum = 31;

for (let i = 0; i < Math.min(key.length, 100); i++) {

let char = key[i],

value = char.charCodeAt(0) - 96;

total = (total\* primeNum+value)%this.keyMap.length;

}

return total;

}

set(key, value){

let index = this.\_hash(key);

if(!this.keyMap[index]){

**// Set it to an empty array**

this.keyMap[index] = [];

}

this.keyMap[index].push([key, value]);

}

get(key){

**//hash the key (Getting the index of the key)**

let index = this.\_hash(key);

**// Check is there something at that index**

if(this.keyMap[index]){

**// return this.keyMap[index] the table containing key (Might be** **consisiting multiple arrays)**

for(let i=0; i<this.keyMap[index].length; i++){

if(this.keyMap[index][i][0]===key){

//return only the value of given key at particular index.

return this.keyMap[index][i][1];

}

}

}

return undefined;

}

}

let hashtable = new HashTable();

hashtable.set('maroon', '#800000');

hashtable.set('yellow', '#FFFF00');

hashtable.set('olive', '#808000');

hashtable.set('salmon', '#FA8072');

hashtable.set('lightcoral', '#FA8080');

hashtable.set('mediumvioletred', '#C71585');

hashtable.set('plum', '#DDa0DD');

**// ( key, value)**

**Output:-**

HashTable {keyMap: Array(17)}

keyMap: Array(17)

0: [Array(2)]

3: [Array(2)]

8: Array(2)

0: (2) ['maroon', '#800000']

1: (2) ['yellow', '#FFFF00']

10: [Array(2)]

13: [Array(2)]

16: [Array(2)]

length: 17

**//To execute get() method.**

hashtable.get('yellow'); ***//value of key ('yellow'):******'#FFFF00'***

'#FFFF00'

hashtable.get('plum'); ***//value of key ('plum'):*** ***'#DDa0DD'***

'#DDa0DD'

hashtable.get('red'); ***//value of key ('red'): undefined, there is no red.***

undefined

**Getting all Keys / Values implemented in hashMap**

**Keys Pseudo Code:**

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

**Values Pseudo Code:**

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

**Approach:**

-- We need to get all the keys and their values from the hashMap i.e. 'keyMap'

We need to avoid getting duplicates keys & values from the keyMap

-- values()

-- To get all values from 'keyMap', avoiding duplicates, we'll call values()

First, create an empty called 'valuesArr'

Start a loop till the length of keyMap (<this.keyMap.length)

IF(Their is something(element/data in the form of array) at any index then only, enter this block for that index)

Start second loop over the array found at that index, till the length of the array at that index. (<this.keyMap[i].length)

Check, If(the value we're getting from keyMap doesn't already exited in 'valuesArr', !valuesArr.includes(this.keyMap[i][j][1]))

Since, the value is unique, just push it to valuesArr.

-- return valuesArr;

-- **keys()**

-- To get all keys from 'keyMap', avoiding duplicates, we'll call keys()

First, create an empty called 'keysArr'

Start a loop till the length of keyMap (<this.keyMap.length)

IF(Their is something(element/data in the form of array) at any index then only, enter this block for that index)

Start second loop over the array found at that index, till the length of the array at that index. (<this.keyMap[i].length)

Check, If(the key we're getting from keyMap doesn't already exited in 'keysArr', !keysArr.includes(this.keyMap[i][j][0]))

Since, the key is unique, just push it to keysArr.

-- return keysArr;

\*/

**Code Implementation for getting all Keys & Values from hashMap:-**

class HashTable {

constructor(size=17) {

this.keyMap = new Array(size);

}

\_hash(key){

let total = 0,

primeNum = 31;

for (let i = 0; i < Math.min(key.length, 100); i++) {

let char = key[i],

value = char.charCodeAt(0) - 96;

total = (total\* primeNum+value)%this.keyMap.length;

}

return total;

}

set(key, value){

let index = this.\_hash(key);

if(!this.keyMap[index]){

**// Set it to an empty array**

this.keyMap[index] = [];

}

this.keyMap[index].push([key, value]);

}

get(key){

**//hash the key (Getting the index of the key)**

let index = this.\_hash(key);

**// Check is there something at that index**

if(this.keyMap[index]){

**// return this.keyMap[index] the table containing key (Might be consisiting multiple arrays)**

for(let i=0; i<this.keyMap[index].length; i++){

if(this.keyMap[index][i][0]===key){

**//return only the value of given key at particular index.**

return this.keyMap[index][i][1];

}

}

}

return undefined;

}

values(){

let valuesArr = [];

for (let i = 0; i < this.keyMap.length; i++) {

if(this.keyMap[i]){

for (let j = 0; j < this.keyMap[i].length; j++) {

**// if the values doesn't already includes in valuesArr. (To get unique values, if duplicates exist)**

if(!valuesArr.includes(this.keyMap[i][j][1]))

valuesArr.push(this.keyMap[i][j][1]);

}

}

}

return valuesArr;

}

keys(){

let keysArr = [];

for (let i = 0; i < this.keyMap.length; i++) {

if(this.keyMap[i]){

for (let j = 0; j < this.keyMap[i].length; j++) {

if(!keysArr.includes(this.keyMap[i][j][0]))

keysArr.push(this.keyMap[i][j][0]);

}

}

}

return keysArr;

}

}

let hashtable = new HashTable();

hashtable.set('maroon', '#800000');

hashtable.set('yellow', '#FFFF00');

hashtable.set('olive', '#808000');

hashtable.set('salmon', '#FA8072');

hashtable.set('lightcoral', '#FA8080');

hashtable.set('mediumvioletred', '#C71585');

hashtable.set('plum', '#DDa0DD');

hashtable.set('plum', 'duplicatePlum');

hashtable.set('purple', '#DDa0DD');

**// ( key, value)**

**Output:-**

hashtable.values();

(8) ['#DDa0DD', 'duplicatePlum', '#FA8072', '#800000', '#FFFF00', '#808000', '#FA8080', '#C71585']

hashtable.keys();

(8) ['plum', 'salmon', 'purple', 'maroon', 'yellow', 'olive', 'lightcoral', 'mediumvioletred']

**BIG O of HASH TABLES**

(Average Case)

Insert: **O(1)**

Deletion: **O(1)**

Access: **O(1)**

**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!