

# Intro to Hashing

**Relevel**  
by Unacademy



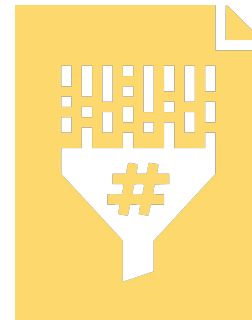
## Topics to be covered:

- What is Hashing
- Hash Function
- Collisions
- Hashtable
- HashMap
- Implementation of HashTable in JS



# What is Hashing

The process of transforming a key into any other value is known as Hashing. The transformation happens using a hash function that is an implementation of a mathematical hashing algorithm. For example when working with phone numbers we can take the first three or the last three numbers as the key to generate the hash for the number, this hash number will point to an integer where we can store the value in database. Now the advantage of doing so would be that next time when we have to search a particular number in the database we can narrow down our search by directly moving on to the position where the number is stored



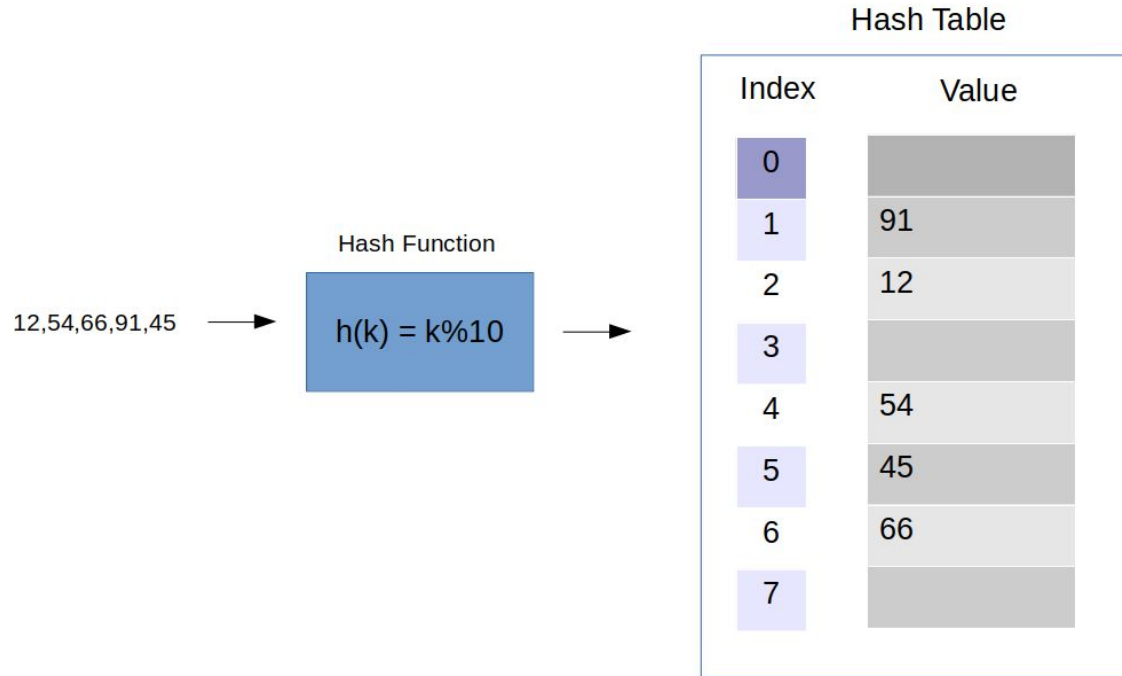
# Hash Tables

All these mapping of keys to their respective hash values are stored in a data structure called hash tables.

Hash Table

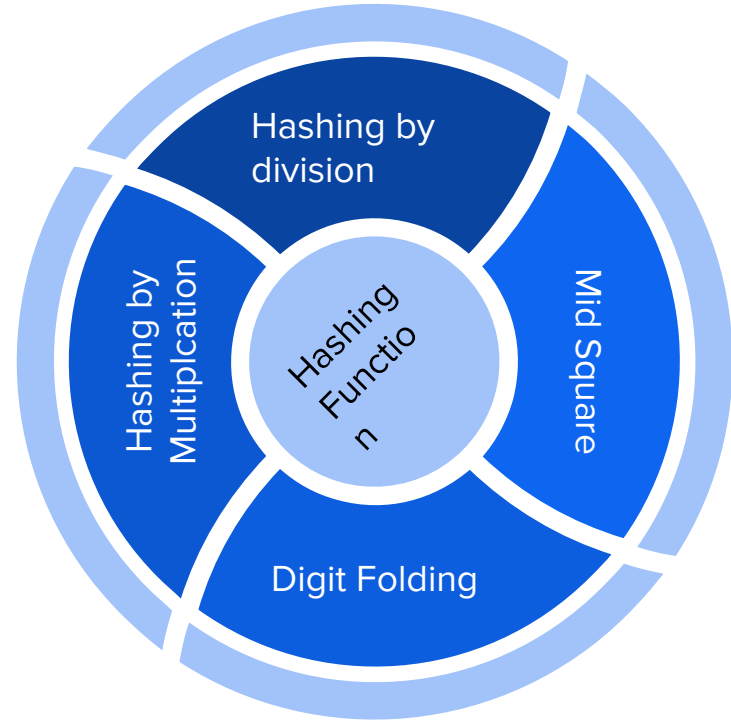
Index	Value
0	
1	
2	
3	
4	
5	
6	
7	

# Hash Tables

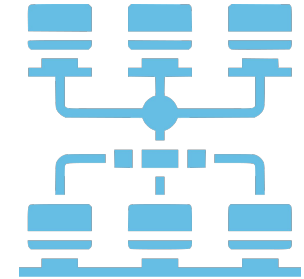


# Hash Tables

Few common heuristic methods to generate hash value



# Hash Tables



1. **Hash by multiplication:** In this approach we multiply the key `k` with a constant real number `c` where  $0 < c < 1$ . Then we will take a fractional part of their product and multiply it by the size of the hash table in which we will be storing it. Let's say the length of the hash table is  $m$  then the hash function  $h(k)$  will be given as :

$$h(k) = \text{floor}(m * \text{fraction}(k * c))$$

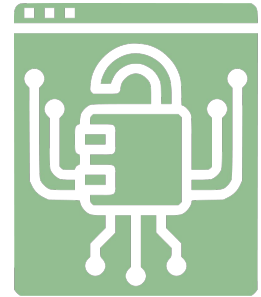
where:

$$\text{fraction}(k * c) = k * c - \text{floor}(k * c)$$

# Hash Tables

2. **Hash by Mod:** In this approach we take the mod of the key with the size of the hash table in which we have to store the value. Suppose  $k$  is the key and size of hash table is  $m$  then the mod function is given as

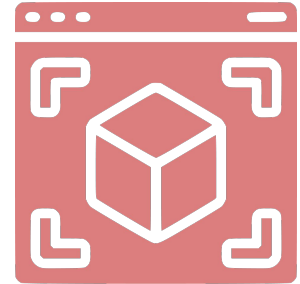
$$h(k) = k \bmod m$$





# Hash Tables

3. **Hashing by Mid Squaring:** In this hash function we square the key value and then extract the middle  $r$  digit which will be termed as the hash value. The value of  $r$  depends upon the size of the hash table that will be used to keep the hash values.



# Hash Tables

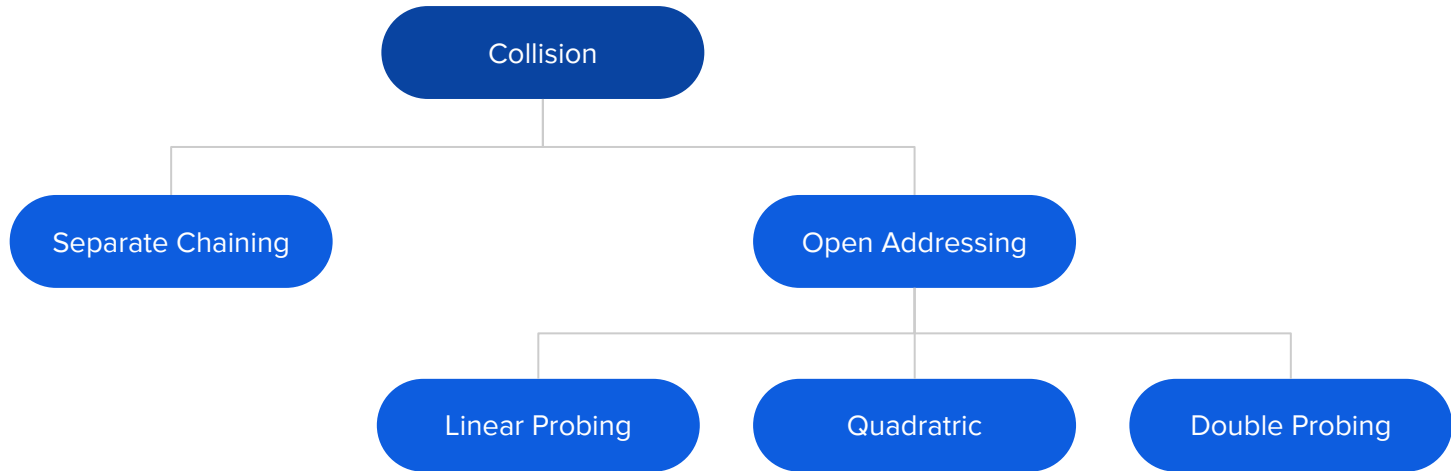
4. **Hashing by Digit Folding:** In this hash function we split the number in two k parts where each part have the same number of digits except the last part. Then we simply add those parts and get the hash value.



# Hash Tables

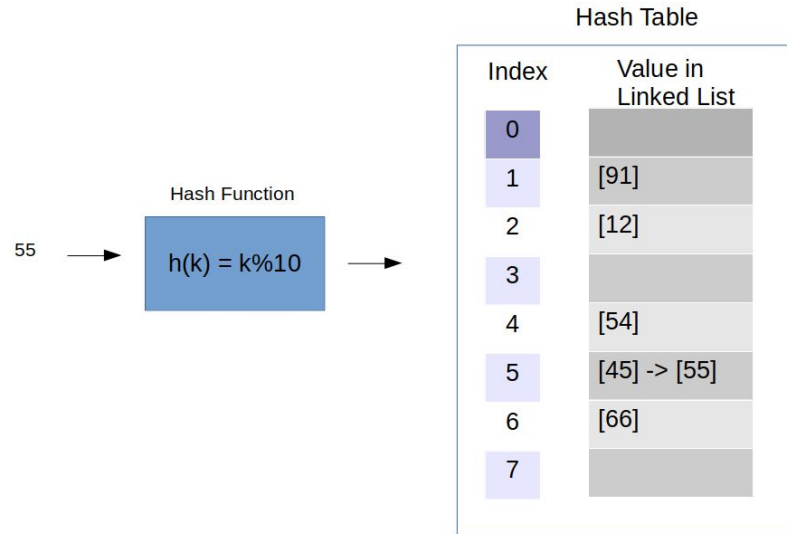
## Collision

Two key having the same hash value



# Hash Tables

**Separate Chaining:** In this collision handling method we use another data structure to store the data with similar hash values. This data structure is mostly linked list as we can perform dynamic memory allocation.



# Hash Tables

## **Advantage:**

This way the average cost of searching the element will be  $O(n)$  where  $n$  is the total number of keys.

## **Disadvantage:**

There is no bound on the size of the linked list therefore it might grow out of bound.

# Hash Tables

**Open Addressing:** In this collision handling method we don't use any new data structure, rather we look for next available spot in the table. Due to this technique this type of handling is mostly preferred in cases where we have limited storage space.

**Advantage:**

Consumes less storage

**Disadvantage:**

Slower than separate chaining.



# Hash Tables

## Linear Probing:

Hash Table


Index	Value
0	
1	91
2	12
3	
4	54
5	45
6	66
7	55

# Hash Tables

## Quadratic Probing:

Hash Table

Index	Value
0	54
1	91
2	12
3	
4	54
5	45
6	66
7	55





# Separate Chaining vs Open Addressing

Separate Chaining	Open Addressing
Less computation is needed	More computation is needed
No limit on the data that can be stored in table	Number of data that can be stored in table is limited
Used when the number of key and its operation frequency is unknown	Used when the number of key and its operation frequency is known
Needs links for extra spacing	No extra link is needed.
Waste of space	No space is wasted

# Separate Chaining vs Open Addressing

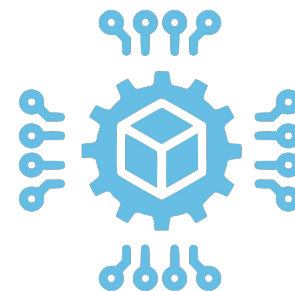
## Object:

In javascript a hash table is represented as object data where we perform the mapping of objects property key to its value.

For example, a student object will look like :

```
let student = {  
  'name':Atul,  
  'addmision_number':223012  
}
```

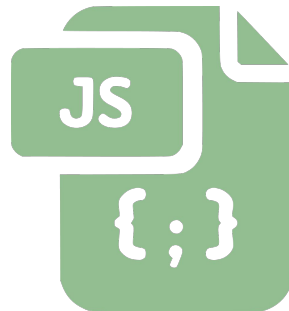
Where 'name' and 'addmission\_number' are the object keys and Atul, 223012 are the object property values.



# Separate Chaining vs Open Addressing

In sort we can say that javascript objects are special type of hash tables for two main reasons:

- Properties can be added into the object class, and each property will have a unique key. Incase if the key conflicts then the value of the corresponding key will be overridden.
- There is no track of the size of the object.



# Separate Chaining vs Open Addressing

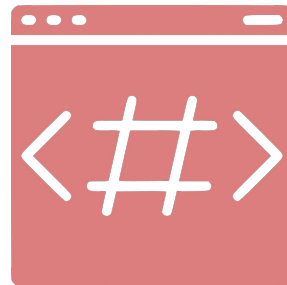
## HashMap in JS:

Map is a data structure that can store elements of any datatype and is also referred to as hashmap. To declare a map in javascript we can use new operator as :

```
`let map_data = new Map();` // to create an empty map
```

```
`let map_data = new Map([[1,"One"],[2,"two"]]);` // to create a map with values
```

And to access the elements from map, we can use the get method as  
`map_data.get(1);`

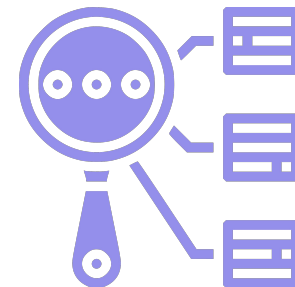


# Separate Chaining vs Open Addressing

It also has an `has()` function which can be used to check if there exists a property in the map or not.

Its important to not that hashmap allows the used of null values unlike the objects therefore it give a better flexibility.

Note: In javascript we also have a `map` function so this data structure should not be confused with that.



# Map vs Object

MAP	Object
A map key can be any value i.e. an object, integer, string , function, etc.	In object the key can either of a symbol,string, number, boolean, undefined
By default it does not contain any key	It contains default key
Map implements an iteration protocol therefore the number of elements inside the map can be retrieved using the size keyword	Object does not implement any such iteration protocol. Therefore we don't have any such parameter that can return the total number of elements.
Optimized and preferred in the scenario where the frequency of addition and insertion of elements is high	Not Optimized and hence not preferred in the scenario where the frequency of addition and insertion of elements is high
Does Not have any native support for performing the serialization or parsing	It does has native support for serialization from the Object to JSON using <code>Json.stringify()</code> and vice versa using <code>Json.parse()</code>

# Load Factor & Rehashing:

In the above example we resolved the problem of collision by implementing the Separate chaining method. And the overall time taken by the model was dependent on :

- **During hash generation:** if the hashmap size is extremely large we can say that the amount of time taken for hashing the key  $K$  would be negligible i.e. the time complexity for hash computation will be  $O(1)$
- **Searching of element:** To insert the element at a given index, first we had to iterate over all the elements at that index to check if the element is present or not and then insert the element.

Supposes in worst case the number of elements are  $n$  then the time complexity will become  $O(n)$

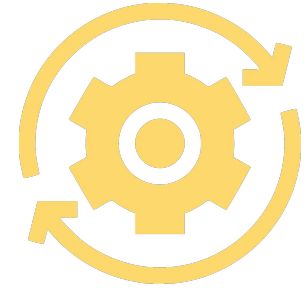
So we can say, If the size of the hashmap is  $L$  and the number of entries in it  $N$  then the average number of elements at each index would be  $N/L$ . This is the load factor of the hashmap.



# Load Factor & Rehashing:

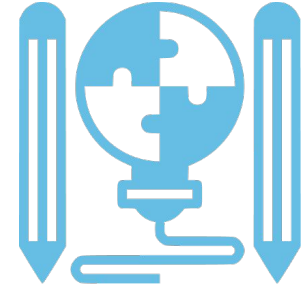
We will try to keep the load factor of the hashmap as low as possible so as to keep less number of entries at each index. Keeping less number of entries will make the complexity of the hashmap almost constant i.e.  $O(1)$ .

**Rehashing:** It is rehashing of the element, and is used when the load factor of the hashmap goes beyond the threshold. What we do in rehashing is :  
Check if the load factor has increased beyond the threshold for each entry. If yes then perform rehashing else continue  
In rehashing, double the size of the current array and create a new array.  
Copy the element from the current map, rehash it based upon the size of the new hashmap and place it at the required index.



# Problem Statement

1. Implement a hashtable with linked list data structure for collision avoidance and the hashing by multiplication function to generate hash value
2. Add load factor and rehashing to the created hash table



**Thank you!**