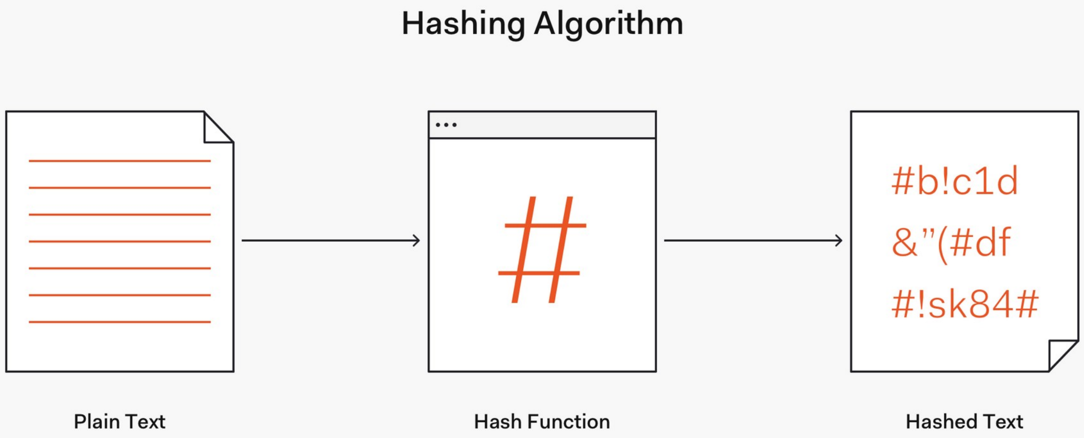
Hashing Algorithms

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.



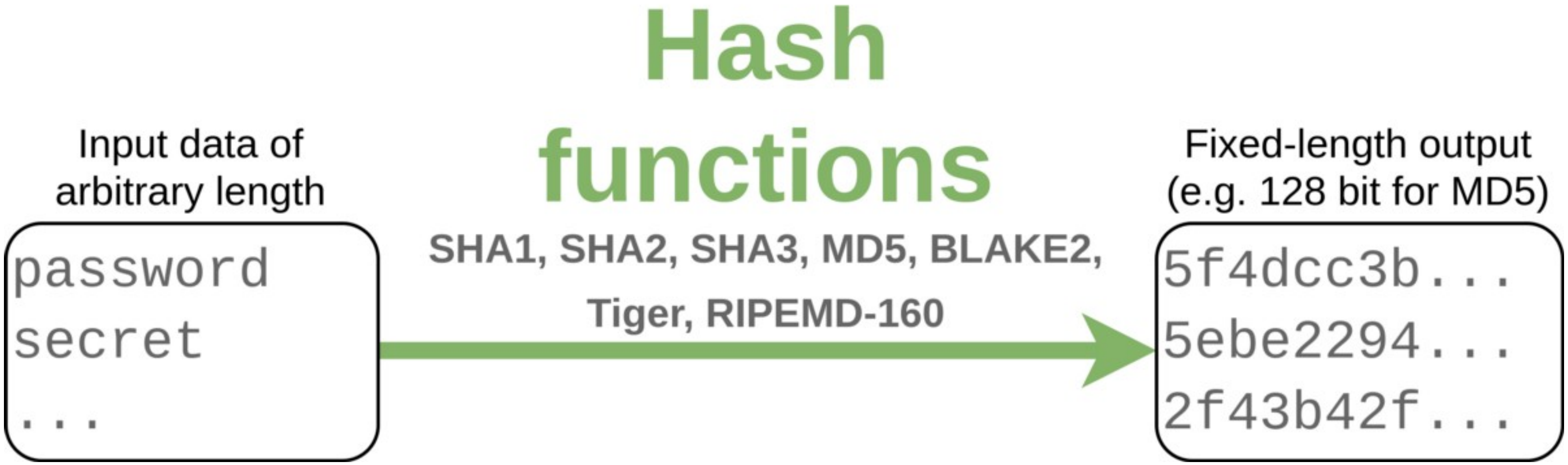
A good hashing algorithm:

- Should be a one-way algorithm
- Should be efficiently computable
- Should uniformly distribute the data

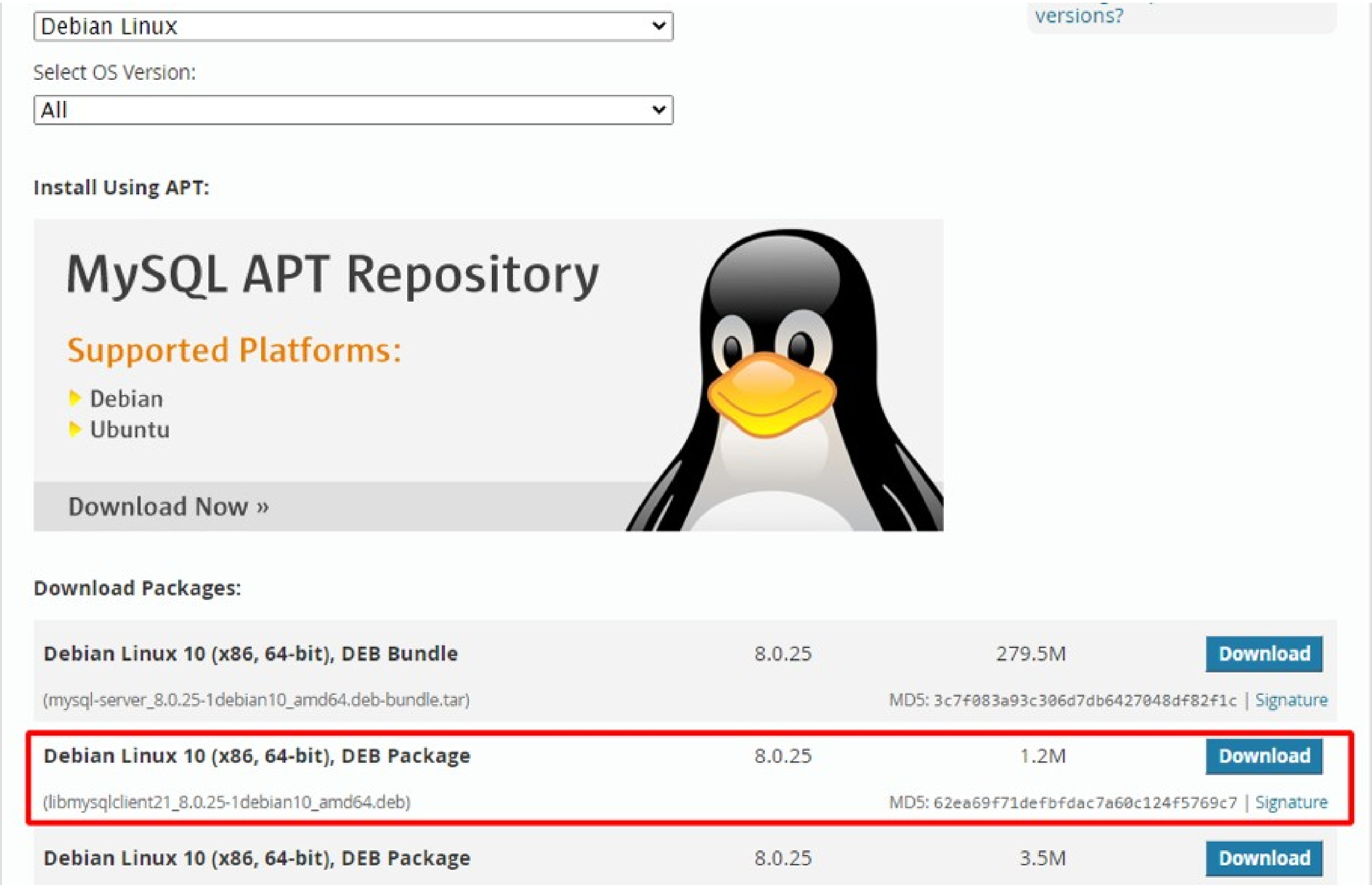
Some Popular Hashing Algorithms

- **MD5** (message digest version 5): Designed in 1991, this hashing algorithm produces a 128-bit hash value. It’s still one of the most commonly used despite being one of the most insecure algorithms. (It’s susceptible to brute force attacks. Stay away from it!)
- **SHA** (secure hashing algorithm) family:
  - **SHA**-1: This hashing algorithm generates a 160-bit hash value. Vulnerable to brute force attacks, it’s no longer considered a secure hashing algorithm. As a result, Microsoft, Google and Mozilla no longer accept SHA-1 SSL certificates (since 2017).
  - **SHA**-256: This hashing algorithm is a variant of the SHA2 hashing algorithm, recommended and approved by the National Institute of Standards and Technology (NIST). It generates a 256-bit hash value. Even if it’s 30% slower than the previous algorithms, it’s more complicated, thus, it’s more secure.
  - **SHA**-384: This hashing algorithm is the latest member of the SHA family, it’s much faster than the SHA-256 and it’s based on a totally different approach (sponge construction).
- **Whirlpool**: This hashing algorithm is based on the advanced encryption standard (AES) and produces a 512-bit hash digest.

Hashing Algorithms


Hash functions
Input data of arbitrary length
Fixed-length output (e.g. 128 bit for MD5)
SHA1, SHA2, SHA3, MD5, BLAKE2, Tiger, RIPEMD-160
password
secret
...
5f4dcc3b...
5ebe2294...
2f43b42f...

Hashing Algorithms



versions?

Debian Linux

Select OS Version:

All

Install Using APT:

MySQL APT Repository

Supported Platforms:

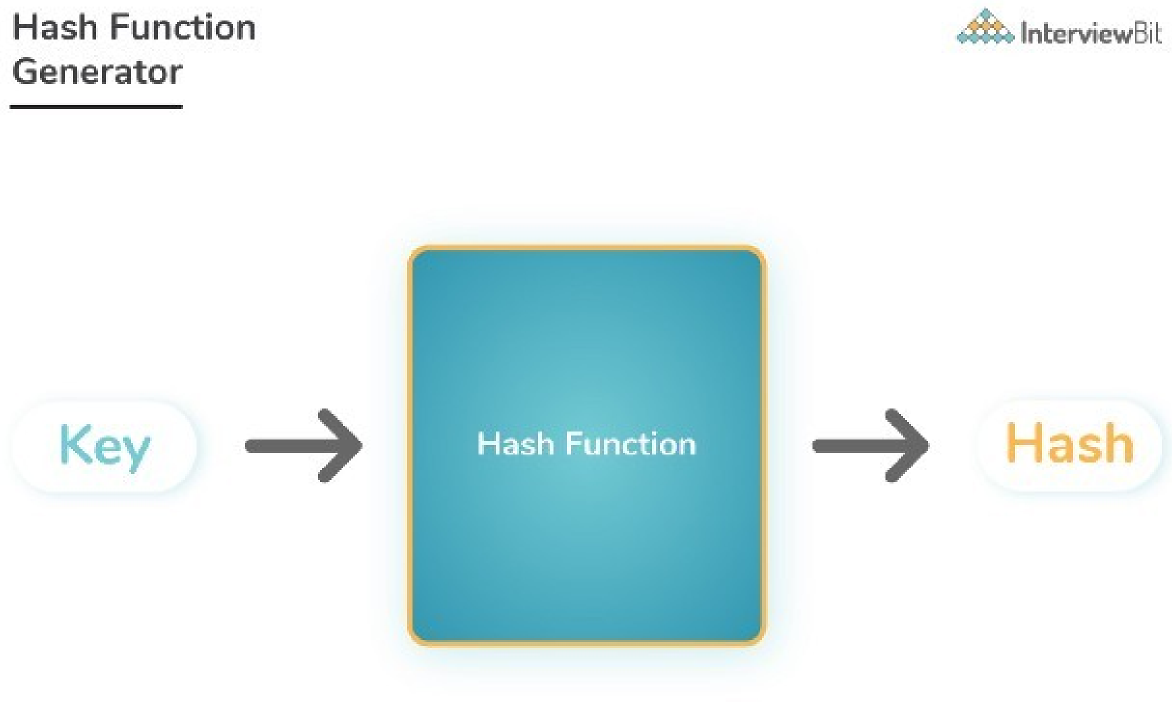
- Debian
- Ubuntu

Download Now »

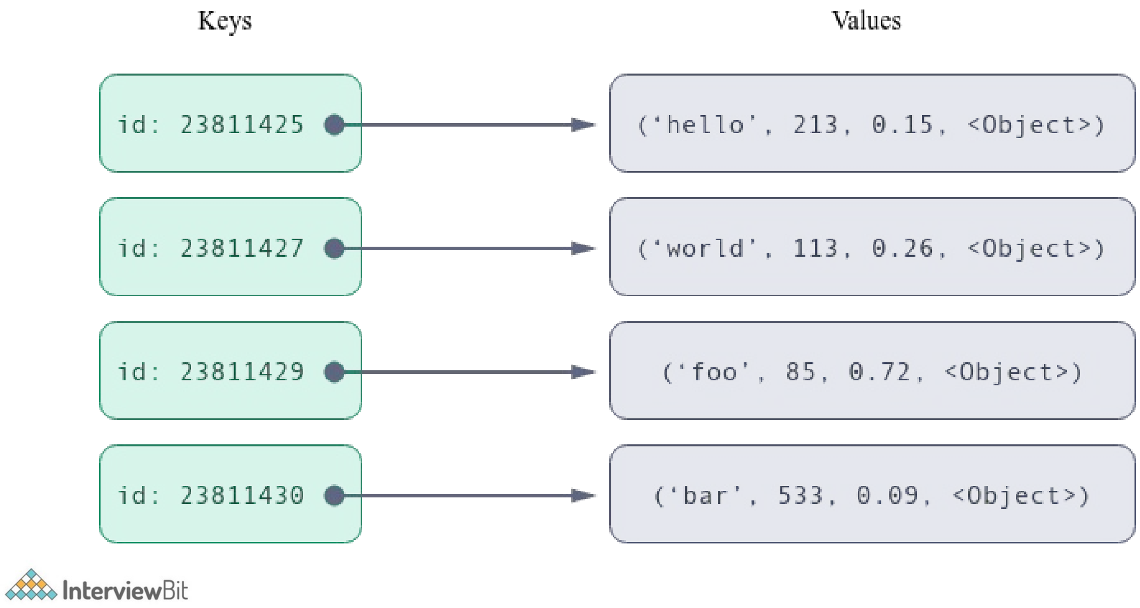
Download Packages:

Debian Linux 10 (x86, 64-bit), DEB Bundle	8.0.25	279.5M	Download
(mysql-server_8.0.25-1debian10_amd64.deb-bundle.tar)			MD5: 3c7f083a93c306d7db6427048df82f1c | Signature
Debian Linux 10 (x86, 64-bit), DEB Package	8.0.25	1.2M	Download
(libmysqlclient21_8.0.25-1debian10_amd64.deb)			MD5: 62ea69f71defbfdac7a60c124f5769c7 | Signature
Debian Linux 10 (x86, 64-bit), DEB Package	8.0.25	3.5M	Download

Key Hashing


Hash Function Generator
InterviewBit
Key
Hash Function
Hash

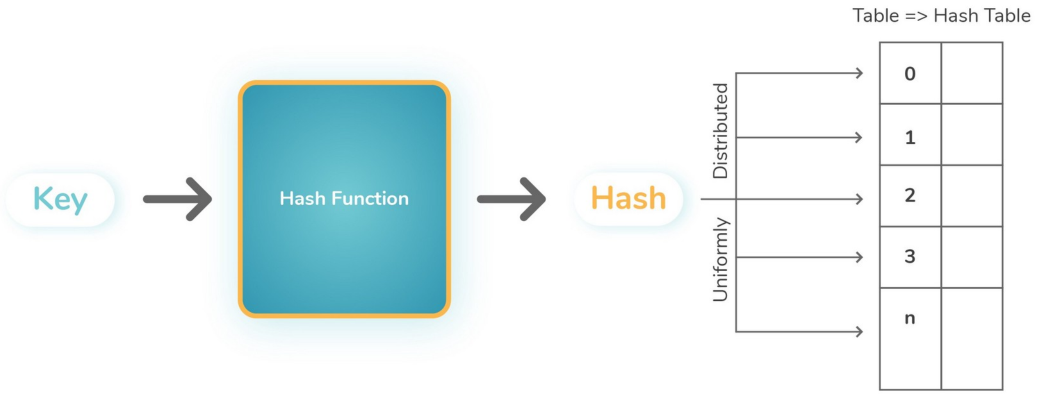
Hash Map/Table



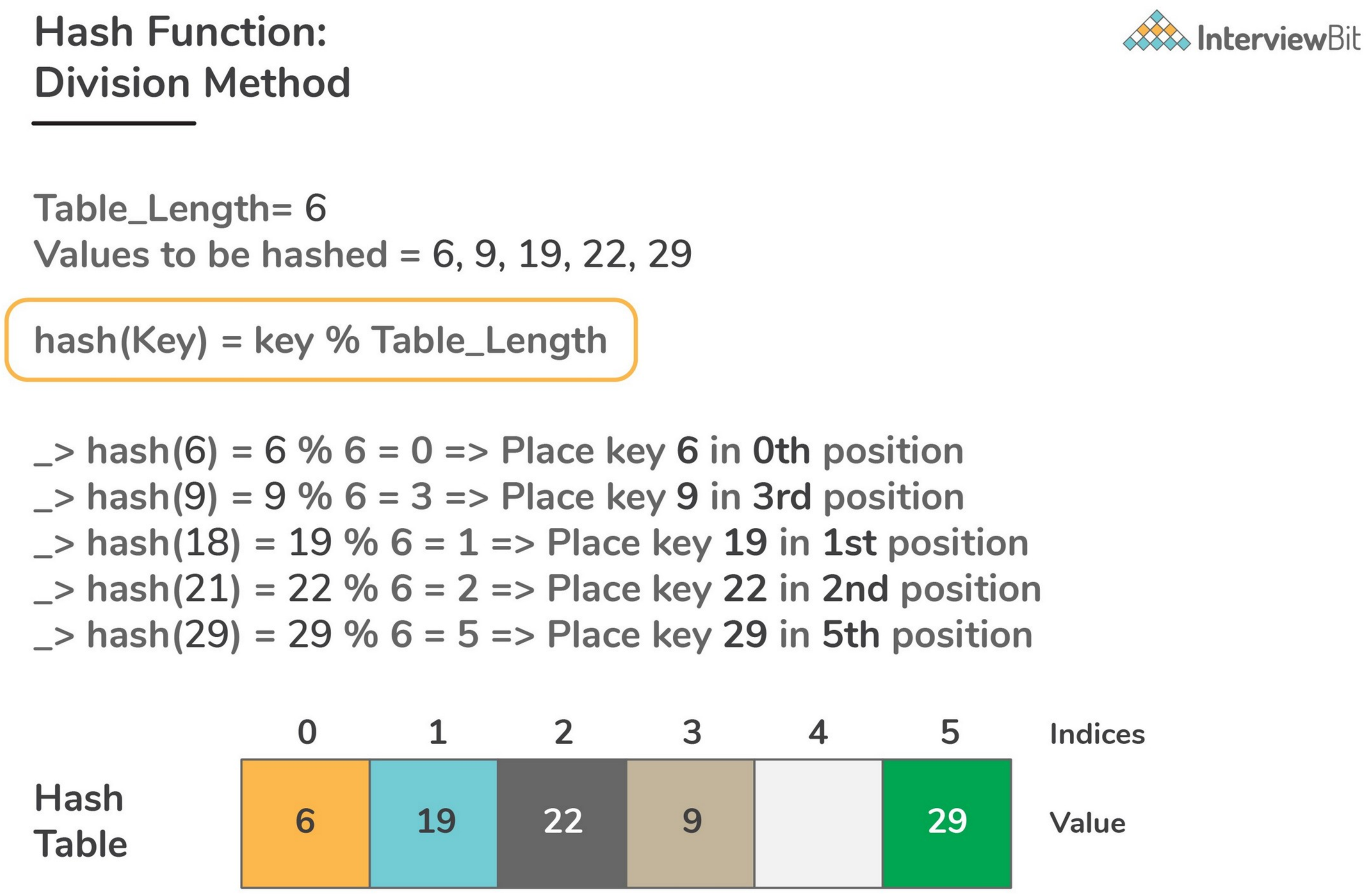
Keys	Values
id: 23811425	('hello', 213, 0.15, <Object>)
id: 23811427	('world', 113, 0.26, <Object>)
id: 23811429	('foo', 85, 0.72, <Object>)
id: 23811430	('bar', 533, 0.09, <Object>)

InterviewBit

Hash Function Generator



Some basic hash functions



Hash Function:
Division Method

InterviewBit

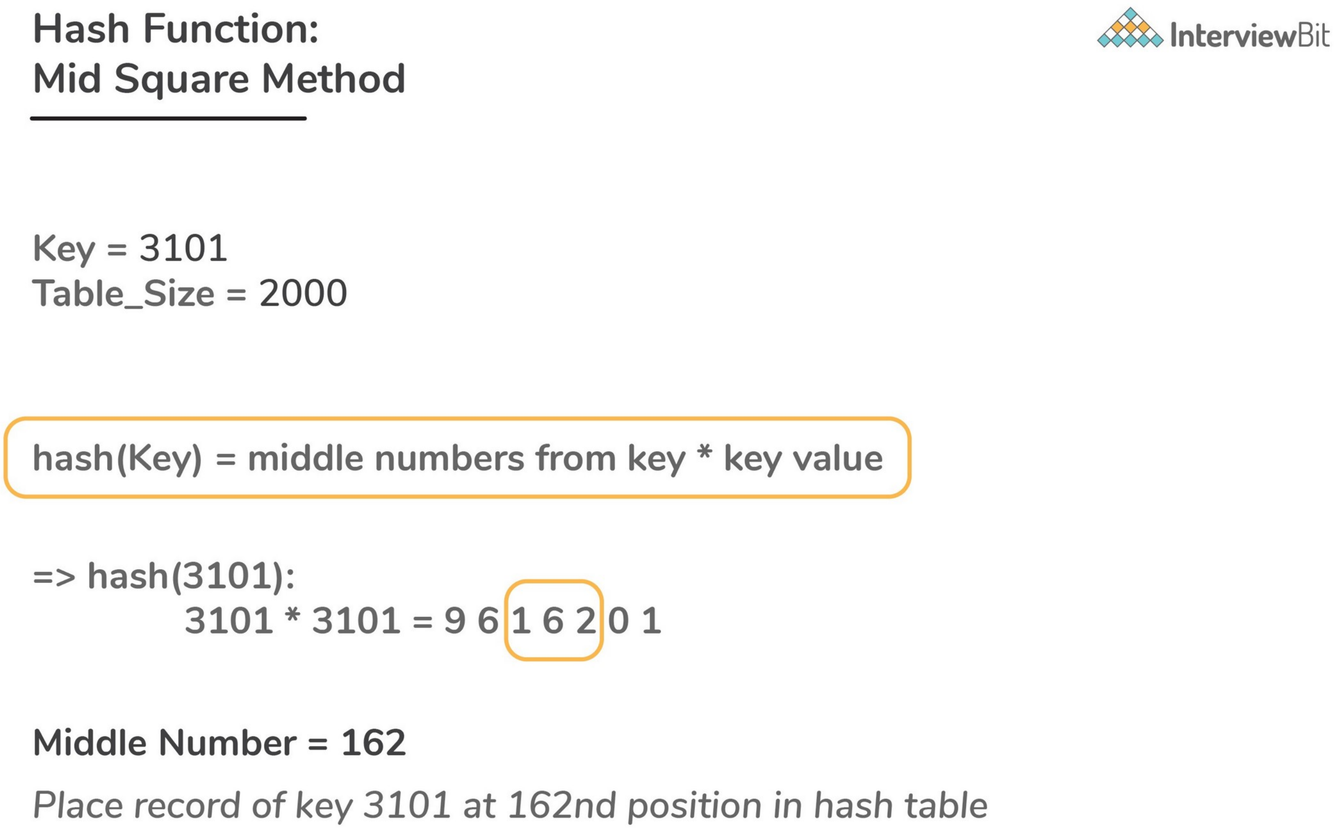
Table_Length= 6
Values to be hashed = 6, 9, 19, 22, 29

hash(Key) = key % Table_Length

_> hash(6) = 6 % 6 = 0 => Place key **6** in **0th** position
_> hash(9) = 9 % 6 = 3 => Place key **9** in **3rd** position
_> hash(18) = 19 % 6 = 1 => Place key **19** in **1st** position
_> hash(21) = 22 % 6 = 2 => Place key **22** in **2nd** position
_> hash(29) = 29 % 6 = 5 => Place key **29** in **5th** position

	0	1	2	3	4	5	Indices
Hash Table	6	19	22	9		29	Value

Some basic hash functions



Hash Function:
Mid Square Method

Key = 3101
Table_Size = 2000

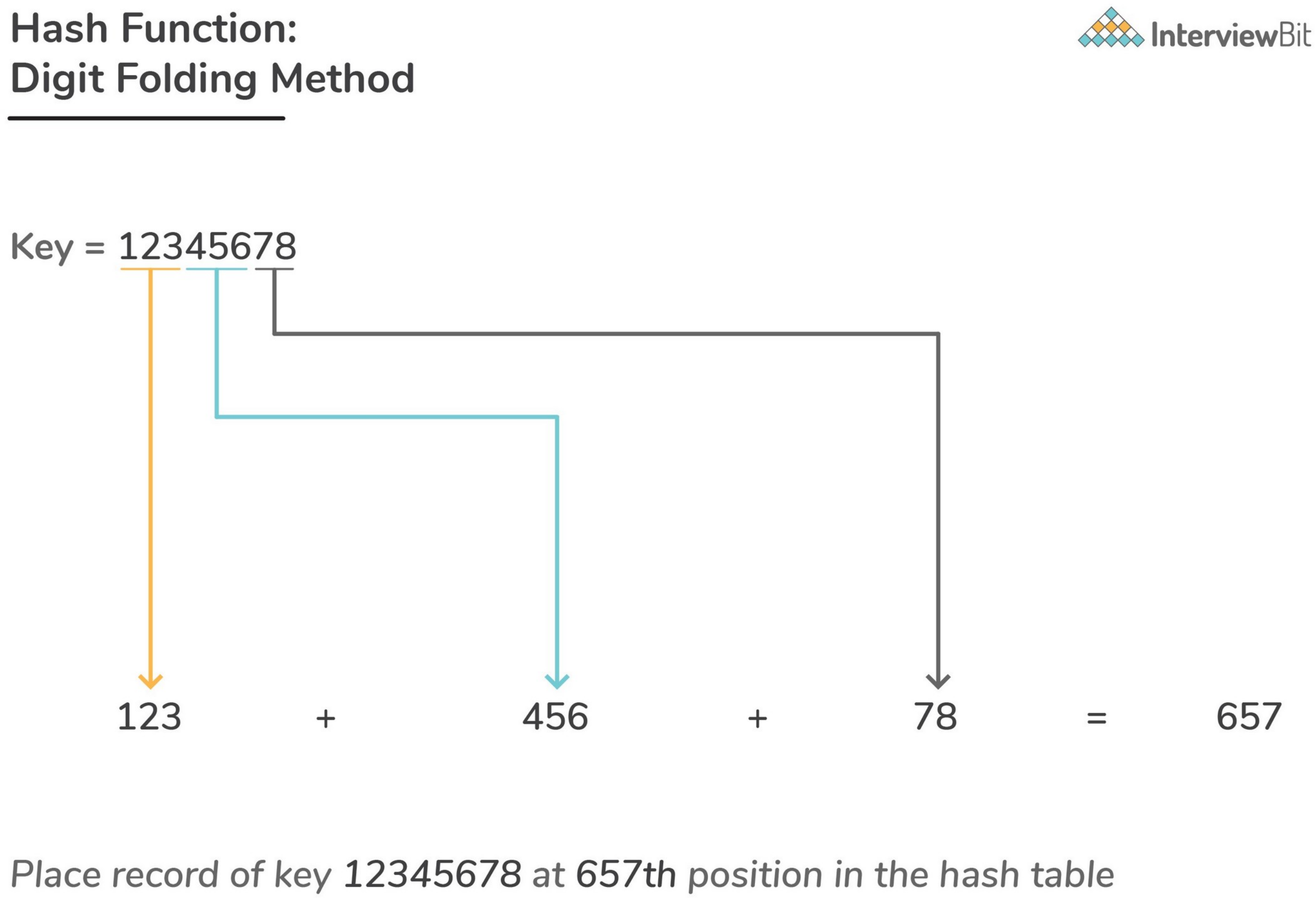
hash(Key) = middle numbers from key * key value

=> hash(3101):
3101 * 3101 = 9 6 1 6 2 0 1

Middle Number = 162

Place record of key 3101 at 162nd position in hash table

Some basic hash functions



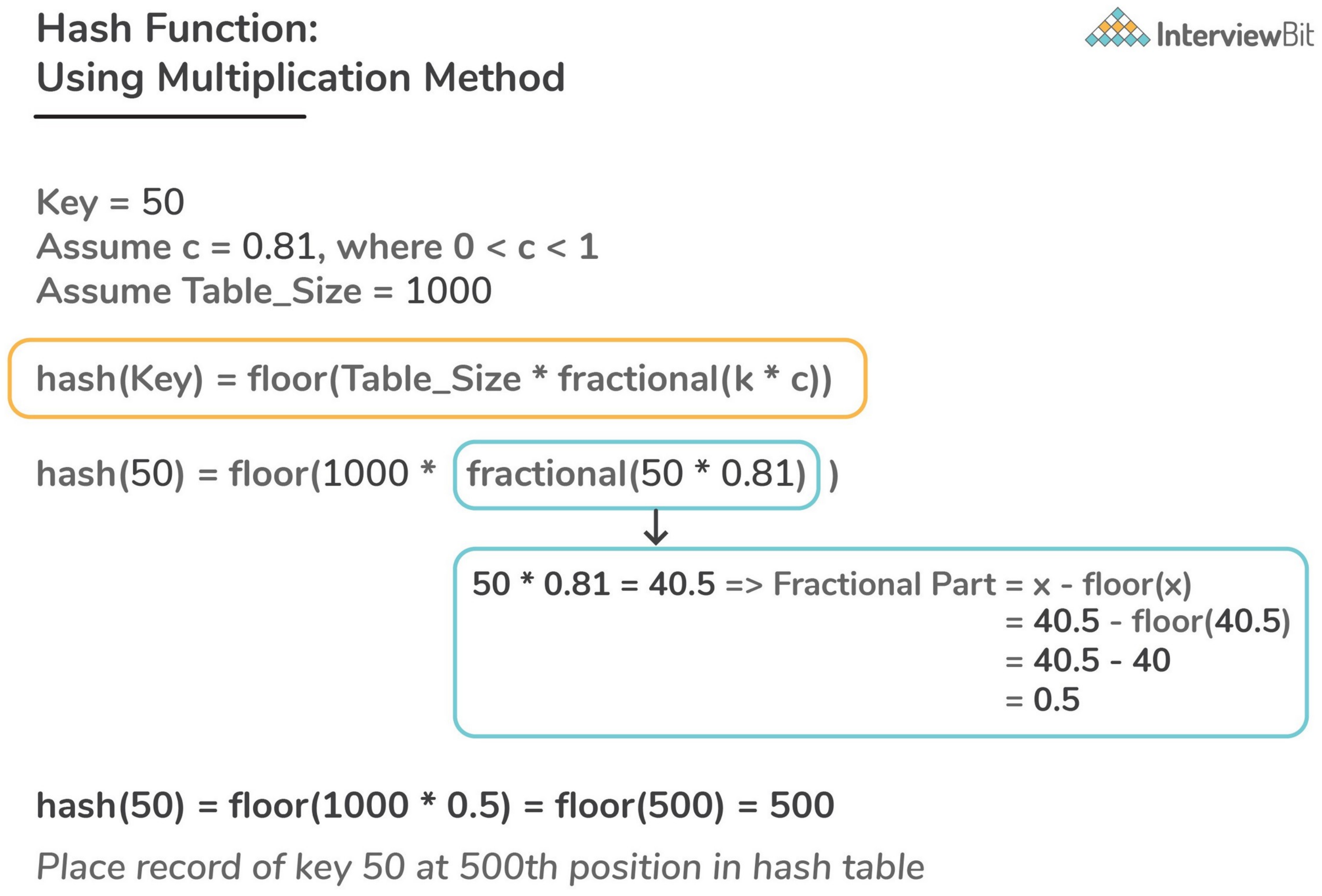
Hash Function:
Digit Folding Method

Key = 12345678

123 + 456 + 78 = 657

Place record of key 12345678 *at* 657th *position in the hash table*

Some basic hash functions



Hash Function:
Using Multiplication Method

Key = 50
Assume c = 0.81, where 0 < c < 1
Assume Table_Size = 1000

hash(Key) = floor(Table_Size * fractional(k * c))

hash(50) = floor(1000 * fractional(50 * 0.81))

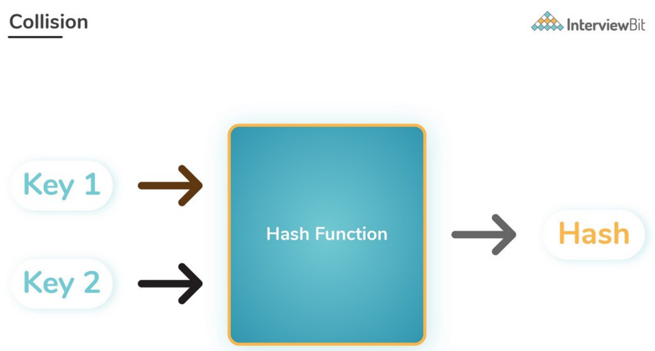
```
50 * 0.81 = 40.5 => Fractional Part = x - floor(x)
                                    = 40.5 - floor(40.5)
                                    = 40.5 - 40
                                    = 0.5
```

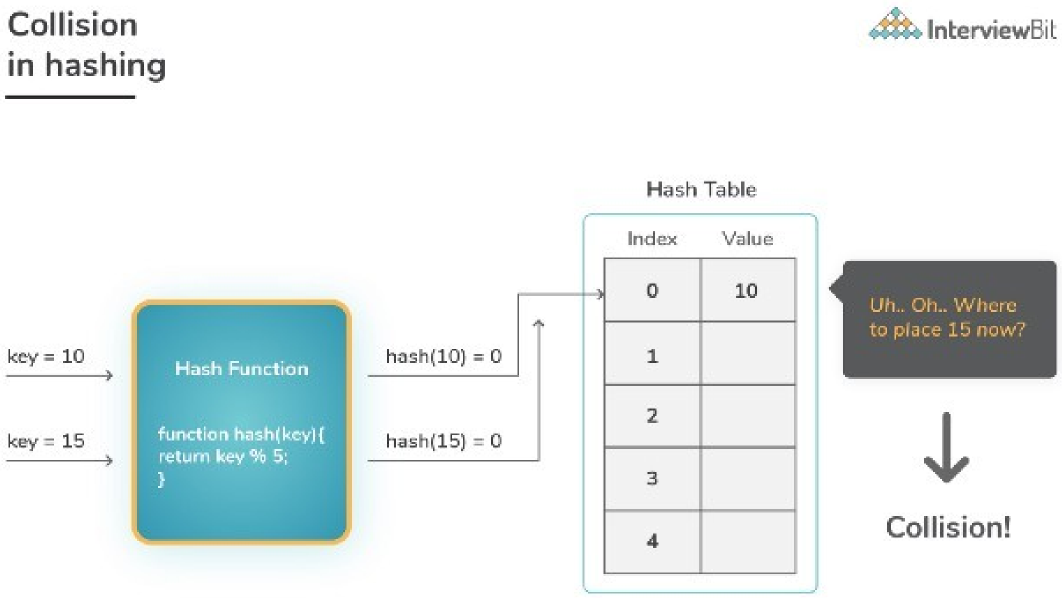
hash(50) = floor(1000 * 0.5) = floor(500) = 500

Place record of key 50 at 500th position in hash table

InterviewBit

Collision

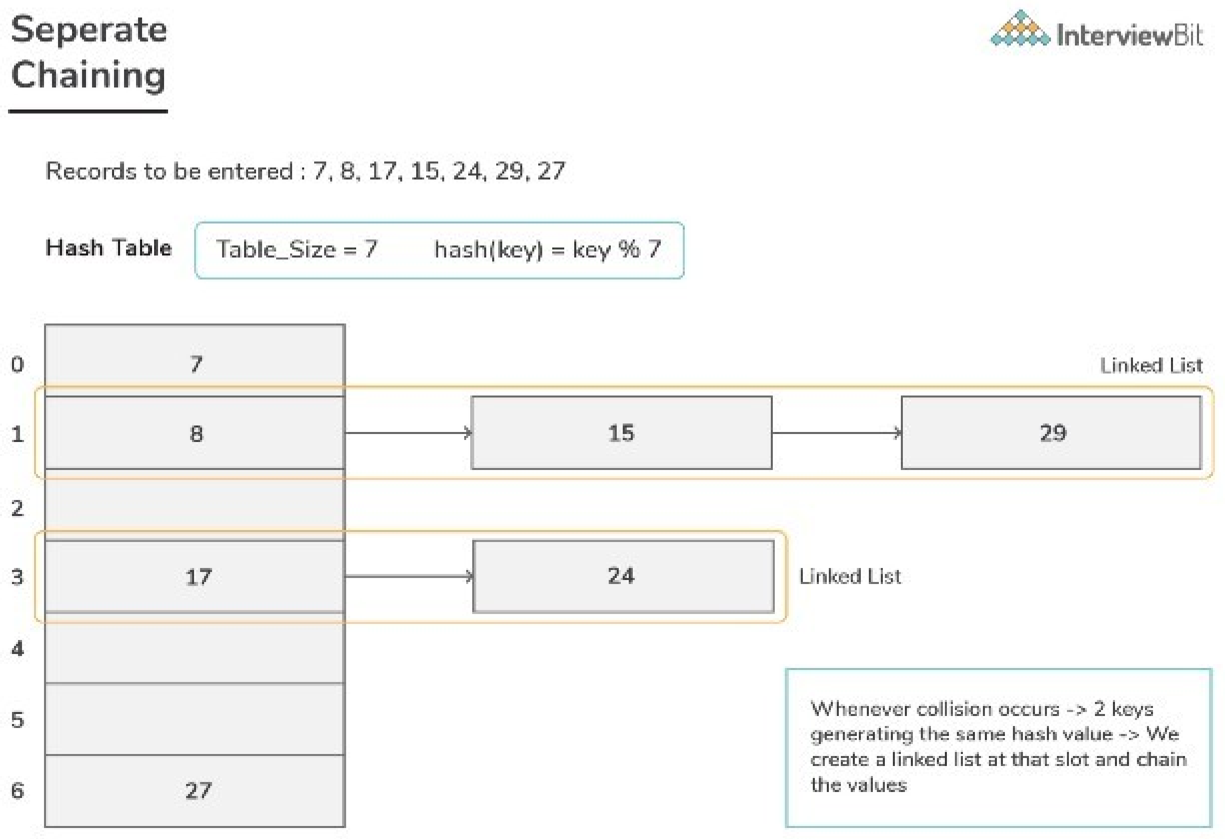




```
function hash(key){
return key % 5;
}
```

Index	Value
0	10
1	
2	
3	
4	

Collision



Seperate Chaining

InterviewBit

Records to be entered : 7, 8, 17, 15, 24, 29, 27

Hash Table

Table_Size = 7 hash(key) = key % 7

0	7
1	8 → 15 → 29
2	
3	17 → 24
4	
5	
6	27

Linked List

Linked List

Whenever collision occurs -> 2 keys generating the same hash value -> We create a linked list at that slot and chain the values

Collision

Open Addressing : Linear Probing

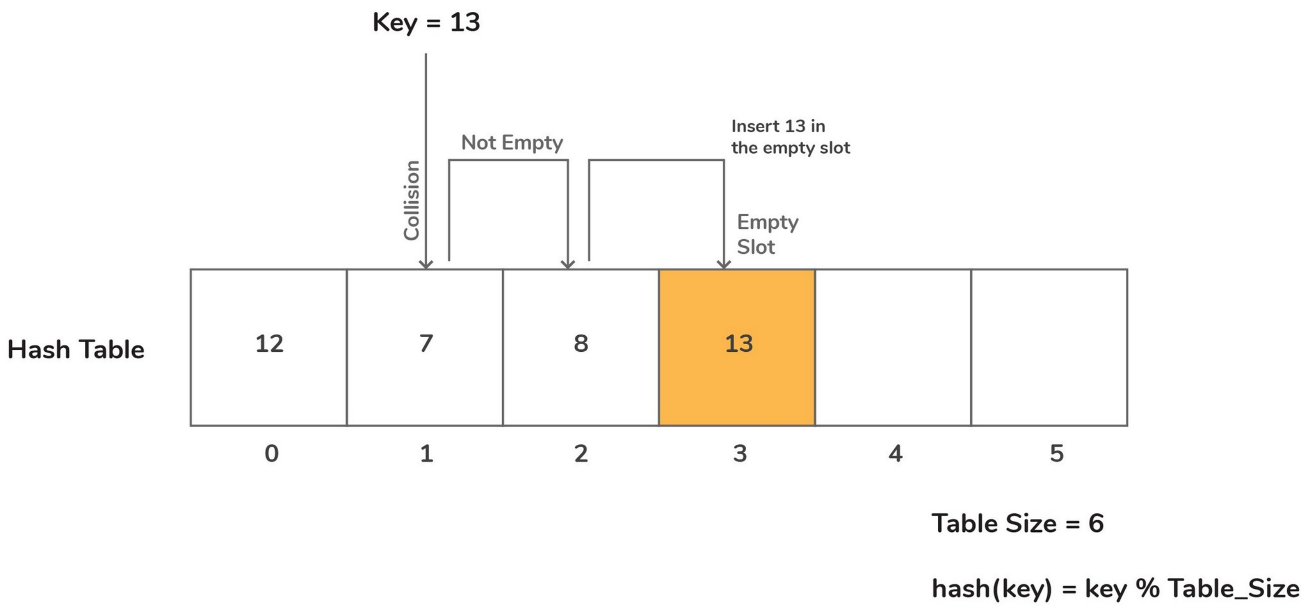
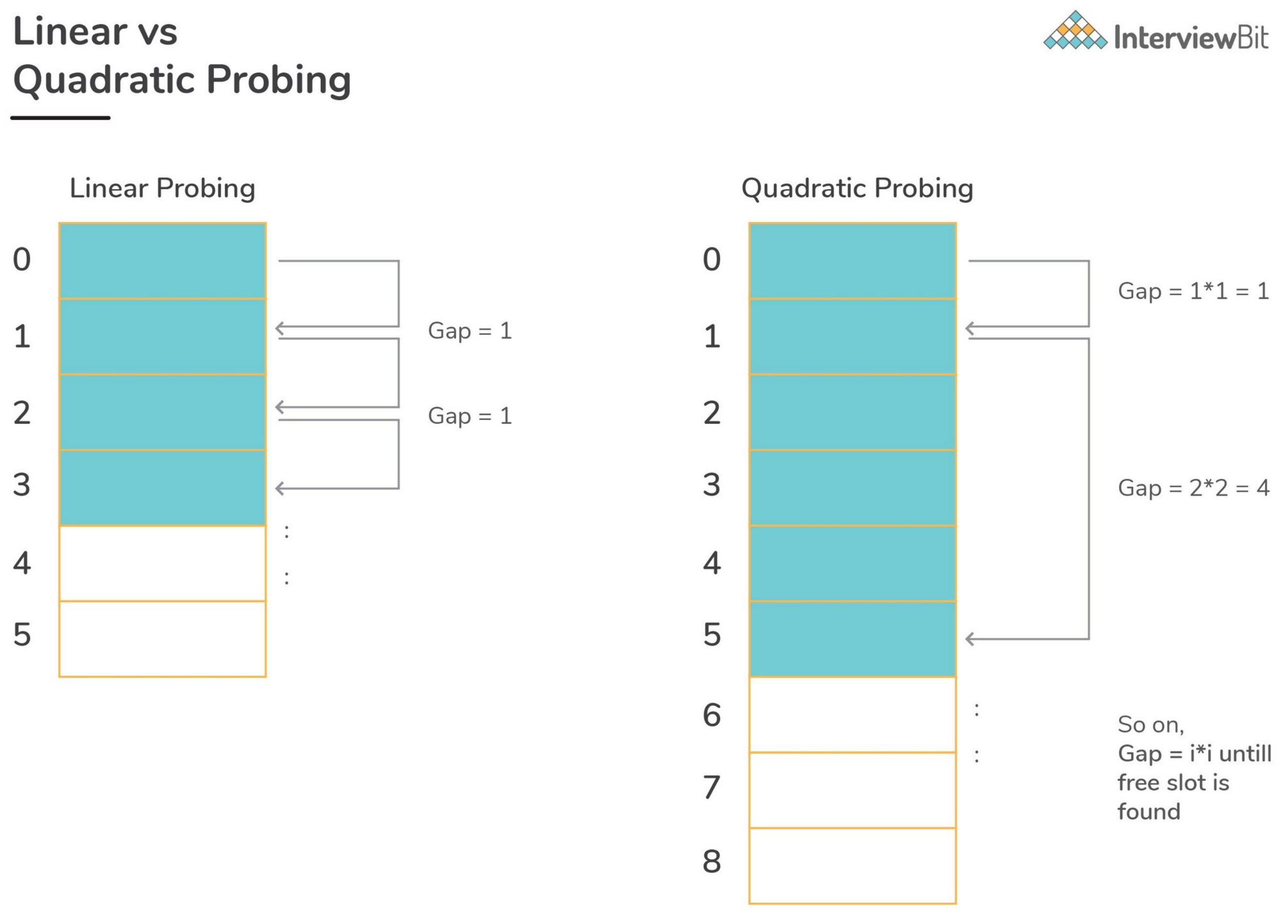


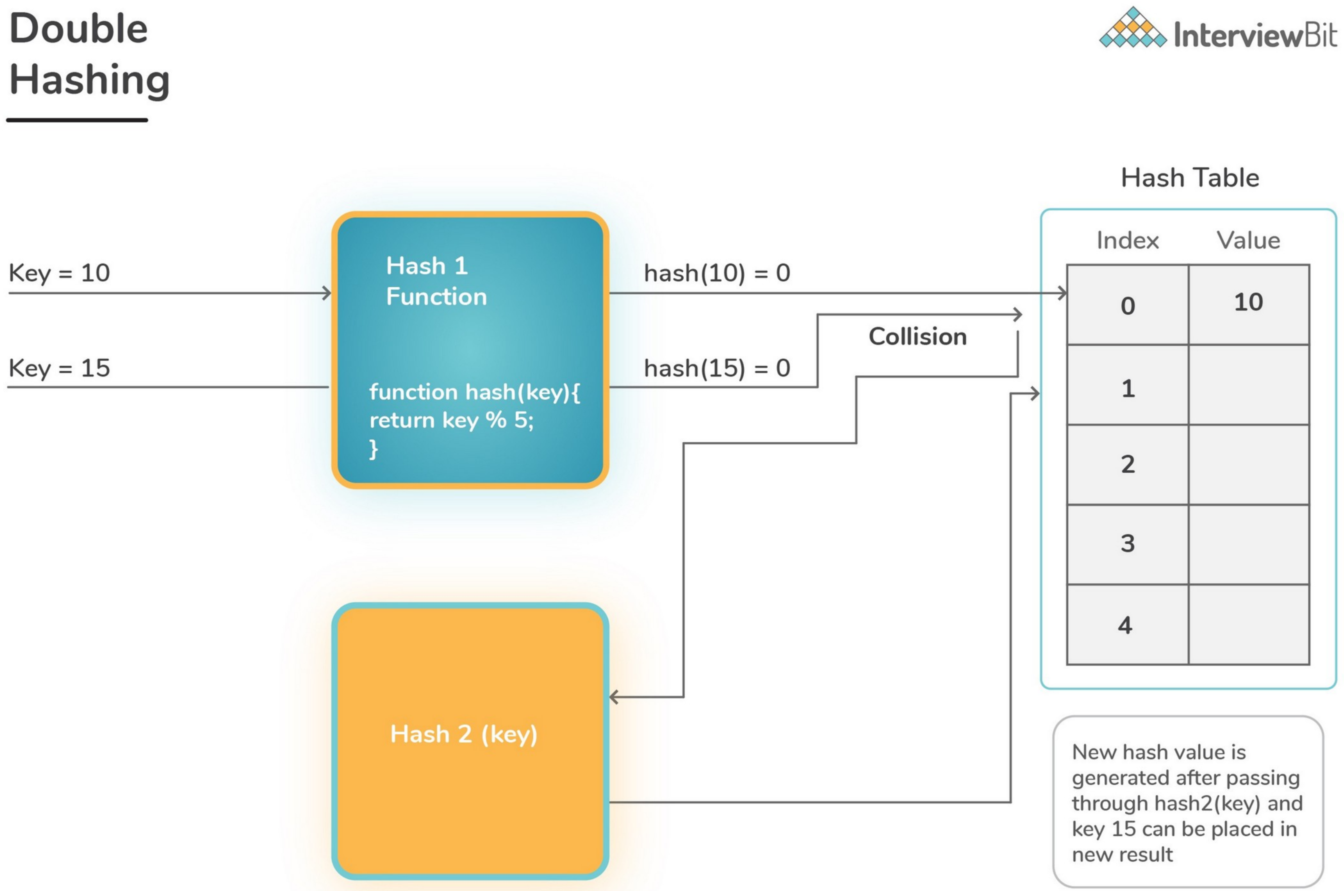
Table Size = 6

hash(key) = key % Table_Size

Collision


Linear vs
Quadratic Probing
InterviewBit
Linear Probing
0
1
2
3
4
5
Gap = 1
Gap = 1
Quadratic Probing
0
1
2
3
4
5
6
7
8
Gap = 1*1 = 1
Gap = 2*2 = 4
So on,
Gap = i*i untill
free slot is
found

Collision



```
function hash(key){
return key % 5;
}
```

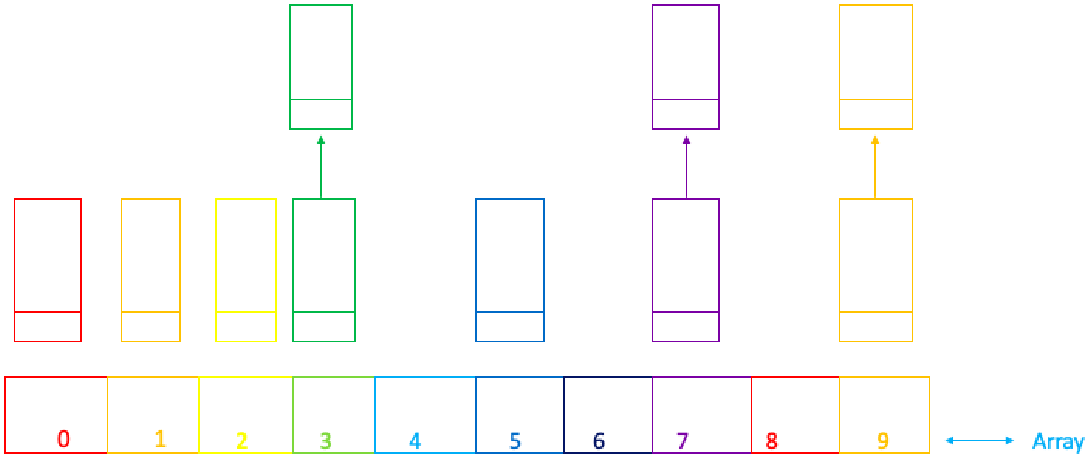
Index	Value
0	10
1	
2	
3	
4	

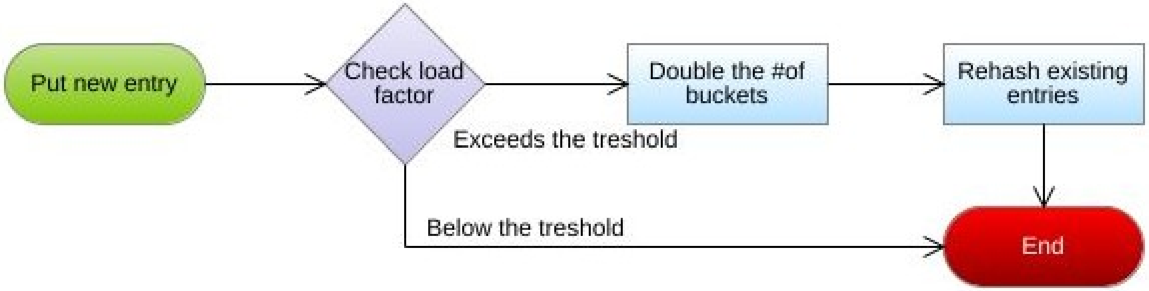
New hash value is generated after passing through hash2(key) and key 15 can be placed in new result

Collision

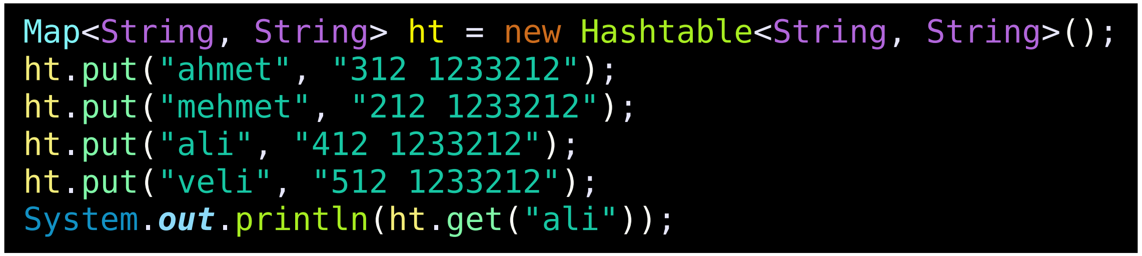
Hash table properties:

- ***#of buckets(indexes)***
- **Load factor = *#of entries* / *#of buckets***





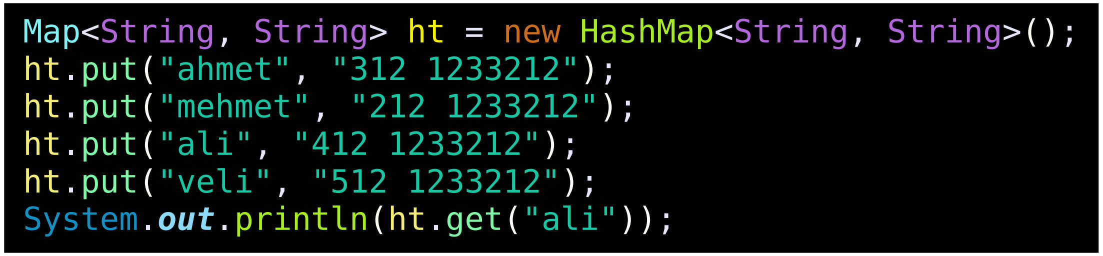
Java hash table implementations



```
Map<String, String> ht = new Hashtable<String, String>();
ht.put("ahmet", "312 1233212");
ht.put("mehmet", "212 1233212");
ht.put("ali", "412 1233212");
ht.put("veli", "512 1233212");
System.out.println(ht.get("ali"));
```

→ Thread safe

Allows nulls ←



```
Map<String, String> ht = new HashMap<String, String>();
ht.put("ahmet", "312 1233212");
ht.put("mehmet", "212 1233212");
ht.put("ali", "412 1233212");
ht.put("veli", "512 1233212");
System.out.println(ht.get("ali"));
```