# Hashing

## Hashing: Introduction: Hashing, Symbol Table, Hashing Functions

- Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function.
- It is a method for representing dictionaries for large datasets.
- It allows lookups, updating and retrieval operation to occur in a constant time

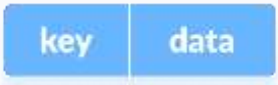
### Why Hashing is Needed?

- After storing a large amount of data, we need to perform various operations on these data.
- Lookups are inevitable for the datasets.
- Linear search and binary search perform lookups/search with maximum of n comparisons and log n comparisons respectively.
- As the size of the dataset increases, these comparisons also become significantly high which is not acceptable.
- We need a technique that does not depend on the size of data.
- Hashing allows lookups to occur in constant time that is small (constant number) of comparisons which does not depend on n. (You will study this in detail in algorithms, in time complexity)

**Hash Function:** A hash function is used for mapping each element of a dataset to indexes in the table.

**Hash Table:** The Hash table data structure stores elements in key-value pairs where

- **Key**- unique integer that is used for indexing the values
- **Value** - data that are associated with keys.

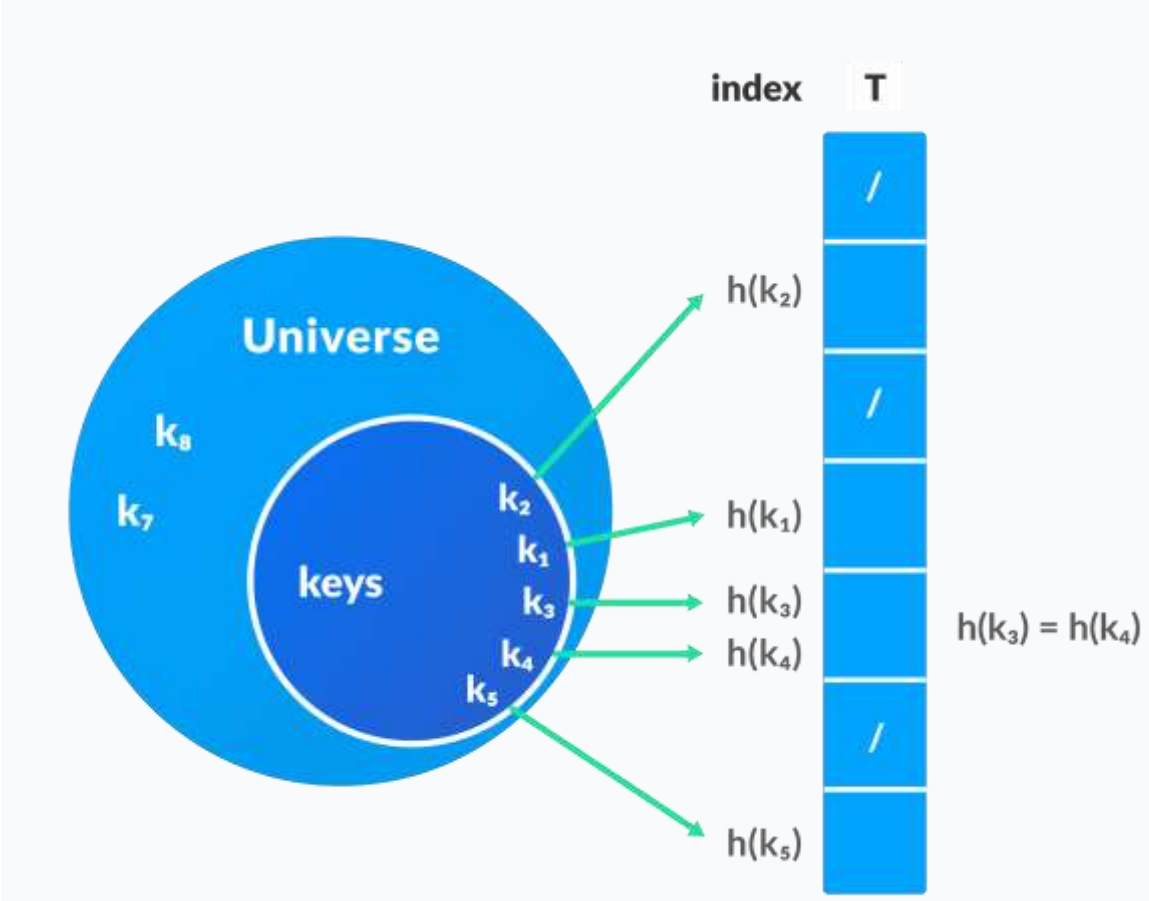



Key and Value in Hash table

### Hashing (Hash Function)

- In a hash table, a new index is processed using the keys.

- And, the element corresponding to that key is stored in the index. This process is called **hashing**.
- Let k be a key and h(x) be a hash function.
- Here, h(k) will give us a new index to store the element linked with k.



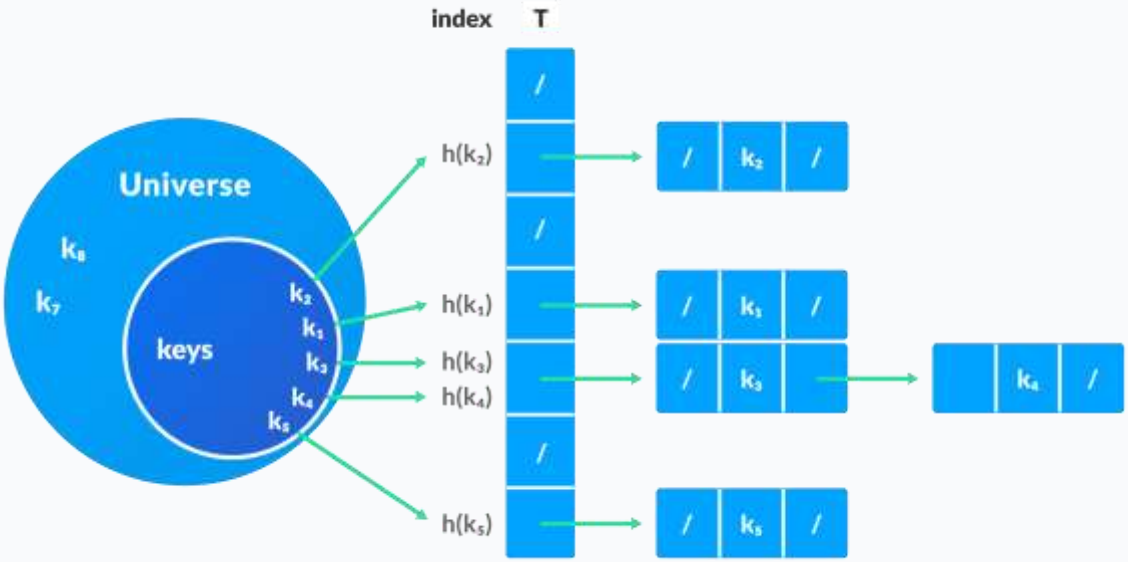

Hash table Representation

## Hash Collision

- When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision.**
- We can resolve the hash collision using one of the following techniques.
    1) Collision resolution by chaining (Open hashing)
    2) Open Addressing (Closed Hashing): Linear/Quadratic Probing and Double Hashing

## 1. Collision resolution by chaining

- In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.
- If j is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, j contains NIL.




Collision Resolution using chaining

## 2. Open Addressing

- Unlike chaining, open addressing doesn't store multiple elements into the same slot.
- Here, each slot is either filled with a single key or left NIL.

## Different techniques used in open addressing are:

### i. Linear Probing

In linear probing, collision is resolved by checking the next slot.

$h(k, i) = (h'(k) + i) \bmod m$

where

- i = {0, 1, ….}

- h'(k) is a new hash function

If a collision occurs at h(k, 0), then h(k, 1) is checked. In this way, the value of i is incremented linearly.

The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

### ii. Quadratic Probing

It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

where,

- $c_1$ and $c_2$ are positive auxiliary constants,
- i = {0, 1, ….}

### iii. Double hashing

If a collision occurs after applying a hash function h(k), then another hash function is calculated for finding the next slot.

$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

### Good Hash Functions

A good hash function may not prevent the collisions completely however it can reduce the number of collisions.

Here, we will look into different methods to find a good hash function

**1. Division Method**

If k is a key and m is the size of the hash table, the hash function h() is calculated as:

h(k) = k mod m

For example, If the size of a hash table is 10 and k = 112 then h(k) = 112 mod 10 = 2. The value of m must not be the powers of 2. This is because the powers of 2 in binary format are 10, 100, 1000, …. When we find k mod m, we will always get the lower order p-bits.

```
if m = 22, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 01

if m = 23, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 001

if m = 24, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 0001

if m = 2p, then h(k) = p lower bits of m
```

**2. Multiplication Method // This is not in your syllabus but provided here for knowledge purpose only.**

h(k) = ⌊m(kA mod 1)⌋

where,

- kA mod 1 gives the fractional part kA,
- ⌊ ⌋ gives the floor value
- A is any constant. The value of A lies between 0 and 1. But, an optimal choice will be ≈ (√5-1)/2 suggested by Knuth.

**3. Universal Hashing // This is not in your syllabus but provided here for knowledge purpose only.**

In Universal hashing, the hash function is chosen at random independent of keys.

## Where we can apply Hashing

Hash tables are implemented where

- constant time lookup and insertion is required
- cryptographic applications
- indexing data is required
- where data relationship does not matter
- where the order of data does not matter

## Application of Hashing

- File system
- Google page rank Algorithm
- Machine Learning
- Digital Signature