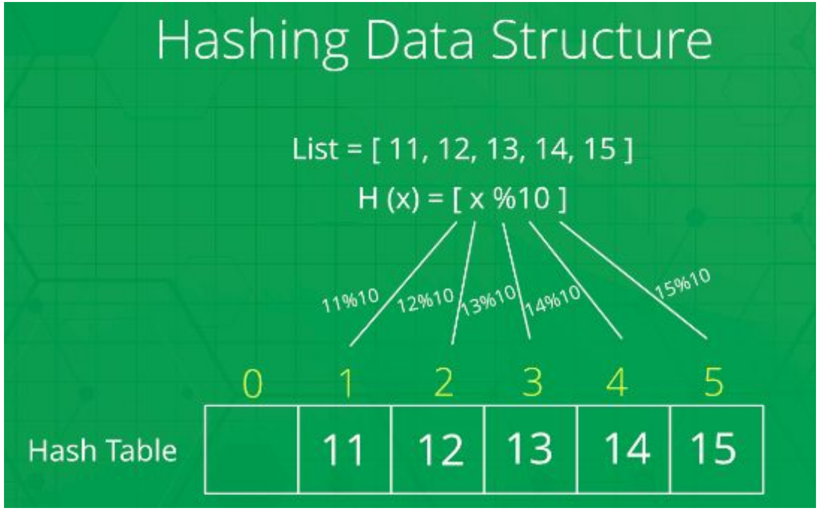
Hashing

Ar. Gör. Elif Ece Erdem

What is Hashing?

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for **faster access to elements**. The efficiency of mapping depends on the efficiency of the hash function used.

Let a **hash function H(x)** maps the value **x** at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.

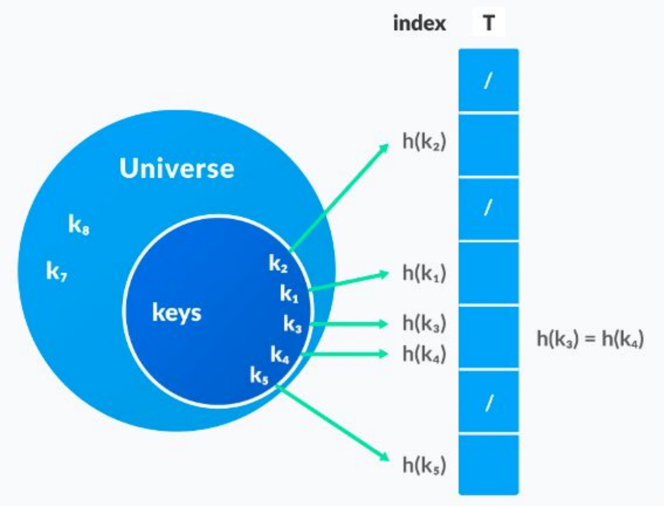


Hash Collision

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.

- Collision resolution by **chaining**
- **Open Addressing**: Linear/Quadratic Probing and Double Hashing



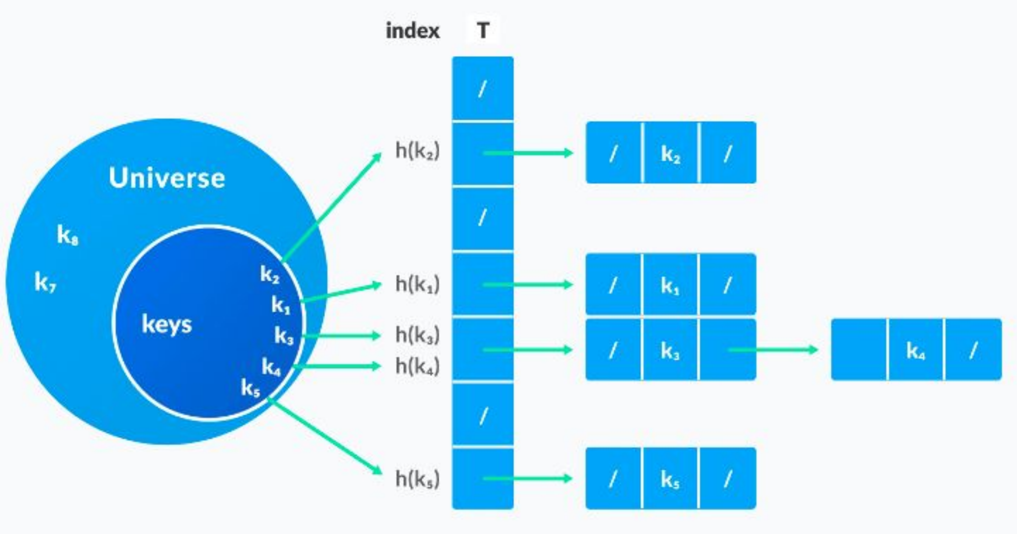
1. Collision resolution by chaining

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a 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`.

Pseudocode for operations

```
chainedHashSearch(T, k)
  return T[h(k)]
chainedHashInsert(T, x)
  T[h(x.key)] = x //insert at the head
chainedHashDelete(T, x)
  T[h(x.key)] = NIL
```



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:

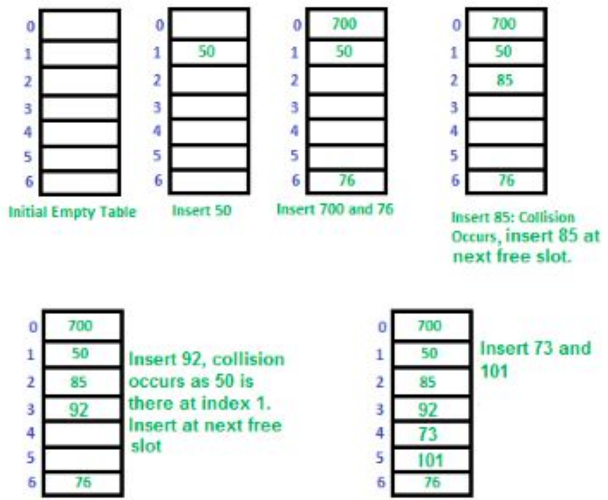
i. Linear Probing

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

`h(k, i) = (h′(k) + i) mod m`

- `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.



a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,

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.

2. Open Addressing

ii. Quadratic Probing

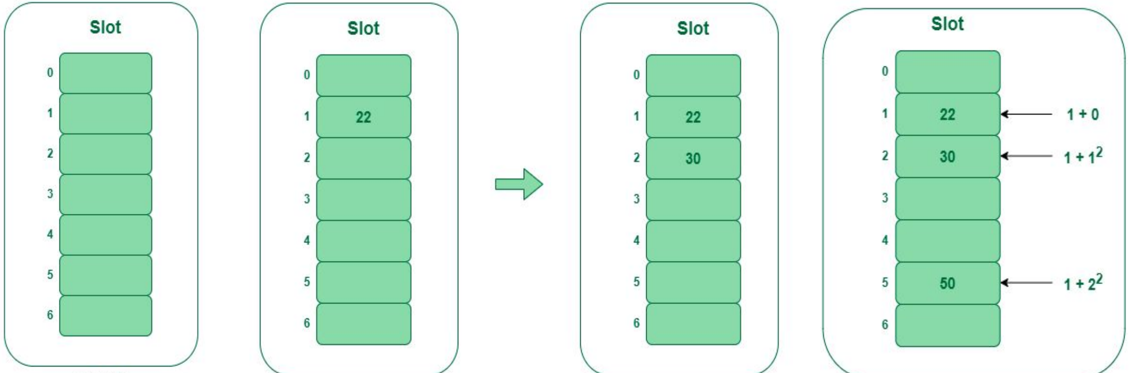
It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*

..

Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i2 . Insert = 22, 30, and 50.



Hash table

Insert keys 22 and 30 in the hash table

Insert key 50 in the hash table

Hash(50) = 50 % 7 = 1

In our hash table slot 1 is already occupied. So, we will search for slot 1+1 = 2,

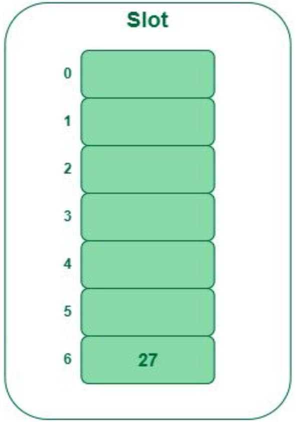
Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5,

Now, cell 5 is not occupied so we will place 50 in slot 5

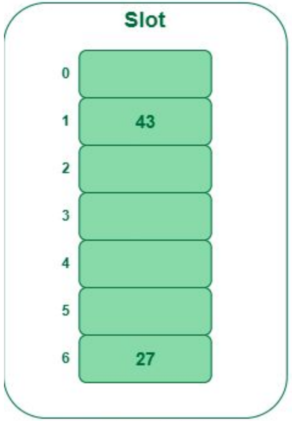
2. Open Addressing
Double hashing Example

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

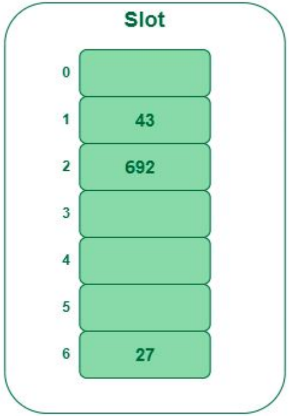
Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is h1(k) = k mod 7 and second hash-function is h2(k) = 1 + (k mod 5)



Insert key 27 in the hash table



Insert key 43 in the hash table



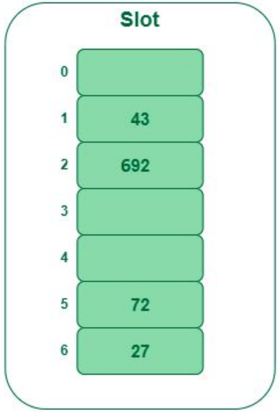
Insert key 692 in the hash table

$h_{new} = [h1(692) + i * (h2(692)] \% 7$

$= [6 + 1 * (1 + 692 \% 5)] \% 7$

$= 9\% 7$

$= 2$



Insert key 72 in the hash table

$h_{new} = [h1(72) + i * (h2(72)] \% 7$

$= [2 + 1 * (1 + 72 \% 5)] \% 7$

$= 5 \% 7$

$= 5,$