Hashing

- It is a technique for storing and retrieving info as fast as possible.
- Worst case complexity of hashing is O(n) but on average gives O(1)
- It is very difficult to store huge array and difficult to sort.
- This means that we have a set of elements (keys) and limited locations in memory. We need to map all these possible keys to possible memory locations.
- The process of mapping the keys to locations is called hashing.

Components of Hashing

1. Hash Table
2. Hash Functions
3. Collisions
4. Collision Resolution Techniques.

Hash Table [or hash map]

- It is a generalization of array.
- It is a DS that :
  1. Stores the keys and their associated values
  2. It uses hash function to map keys to their associated values

Hash Function

- used to transform key into index.
- perfect hash func - if a hF map each item into a unique slot.
- ~~A hF should~~
- Goal is to create a hash function that
  → minimizes the number of collisions,
  → is easy to compute
  → evenly distributes the elements in the hash table.

Ways to find the index [convert key to index]

1. Key % table size.
2. Folding method -
   - divide elements into equal size pieces [last piece may not be eq]
   - Eg: 436-555-4601.
     ⇒ {43, 65, 55, 46, 01}
     43 + 65 + 55 + 46 + 01 = 210

If table size is 11 [11 slots],

then, 210 % 11 = 1

∴ 436-555-4601 hashes to slot 1.

Load Factor (LF)

- LF = $\frac{\text{no. of elements in hash table}}{\text{hash table size}}$
- If LF is high for a given set of keys then it means that it uses a good hash function.
- LF helps in determining the efficiency of the hF.

Collisions

- It is the condition where two records are stored in the same location.

Collision Resolution Techniques

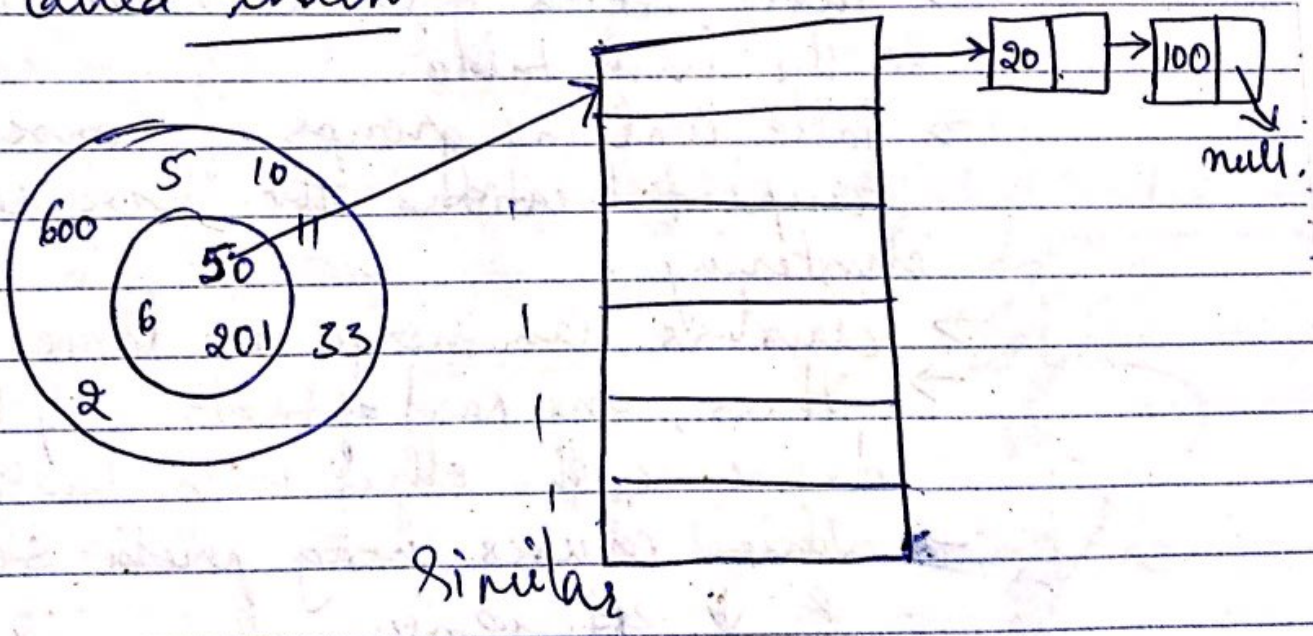
- The process of finding an alternate location is called collision resolution
- Types
  1) Direct chaining - An array of linked list application
     - Separate Chaining

2) Open addressing - Array based implementation

a) Linear probing (linear search)
b) Quadratic probing (nonlinear search)
c) Double hashing (use two hash funcs)

* Separate chaining.

- Combines linked representation with hash table.
- When 2 or more records are hashed to the same location, these records are constituted into a singly-linked list called chain



* Open Addressing [closed hashing]

- All keys are stored in the hash table itself. ∴ called closed hashing

- This is based on probing.

a) Linear probing.

- interval b/n probes is fixed at 1.
- HT is searched sequentially & starting from the original hash location.
- If a location is occupied, we check the next location.

Function for rehashing

rehash(key) = (n+1) % tablesize

- Problems
  - → table items gets clustered together in the hash table.
  - → Table contains groups of consecutively occupied locations that are called clustering.
  - → clusters can merge into larger clusters.
  - → Thus, one part of table may be very dense & the other with few items.
  - → This causes long probe searches & ↓ efficiency.

- The next location to be probed is determined by the step-size
- step-size should be relatively prime to the table size.

b) Quadratic Probing

- Interval b/n probes $\uparrow$'s proportionally to the hash value
- Clustering problem is eliminated in this
- We start from original hash location, i.e. if a location is occupied, we check locations $i+1^2$, $i+2^2$, ...

Function of Reshashing

$$\text{Rehash(key)} = (n+K^2)\ \%\ \text{tablesize}$$

Example

Keys {31, 19, 2, 13, 25, 24, 21, 9}.
Table size = 11.

hash fune $\Rightarrow$ h(Key) = Key mod 11.

31 mod 11 = 9.
19 mod 11 = 8.
2 mod 11 = 2.
13 mod 11 = 2 $\rightarrow$ $2+1^2 = 3$
25 mod 11 = 3 $\rightarrow$ $3+1^2 = 4$
24 mod 11 = 2 $\rightarrow$ $2+1^2$, $2+2^2 = 6$
21 mod 11 = 10.
9 mod 11 = 9 $\rightarrow$ $9+1^2 = 10$ $\rightarrow$ $9+2^2 = 13$
$\Rightarrow 13 \bmod 11 = 2$
$9+3^2 = 18 \bmod 11$
$= 7$.

0	1	2	3	4	5	6	7	8	9	10
		2	13	25		24	9	19	31	21

c) Double hashing:

- Interval b/n probes is computed by another hash function.
- The second hash function h2 should be
  $h2(key) \neq 0$ and $h2 \neq h1$
- we first probe the location h1(key).
  If occupied, then
  $h1(key) + h2(key), h1(key) + 2 * h2(key)$

Example:

Keys = {58, 14, 91, 25}.
Table Size = 11
hash func ⇒ h1(key) = key mod 11 & h2(key) = (7 - key mod 7)

58 mod 11 = 3

14 mod 11 = 3 → 3 + (7 - 14 mod 7)
3 + 7 = 10.

91 mod 11 = 3 → 3 + 7, 3 + 2 * 7
= 17 mod 11.
= 6.

25 mod 11 = 3 → 3 + 7, 3 + 2 * 7, 3 + 3
3 + (7 - 25 mod 7)
3 + 2(3)
3 + 6 = 9.

**

*How hashing gets O(1) Complexity?

- By using load factor (no. of ~~element~~ keys by table size) we make sure that each block on average stores the max no. of elements less than the LF.
- In practice, LF is a constant, usually 10 or 20.
  ∴ searching 20 or 10 elements becomes constant.
- Access time of the table depends on the load factor which in turn depends on the hash function.
  This is bcoz, hash func distributes the elements to the hash table.

∴ for this reason we say hash table gives O(1) Complexity on average.

Also generally we use hash tables in cases where searches are more than insertion & deletion operations.

When not to use HT

1) when data ordering is required
2) when multidimensional data is involved.
3) if keys are long & of variable lengths
4) when we have dynamic data
5) when data doesn't have unique keys

If the no. of collisions is very high, the worst case runtime is O(N), where N is the no. of keys.

We can also implement HT with a balanced binary search tree. This gives us O(log N)

→ Advantage of using this is that it gives us less space as we don't use large array it can also iterate through the keys in order.

~~When the number of collisions is very high the worst case RT is O(N),~~