Hashing 3 : Internal Implementation & Problems

Problem Statement :- Given an array of size N & Q queries. In each query, an element is given. We have to check whether that element exists or not in the given array.

A[] = { 2, 4, 11, 15, 6, 8, 14, 9 }

Queries :-

K = 4 → True

K = 10 → False

K = 17 → False

K = 14 → True

Brute force

↳ For each query, Go & Search in array.

TC → O(N * Q)

SC → O(1)

⇒ OPTIMIZATION

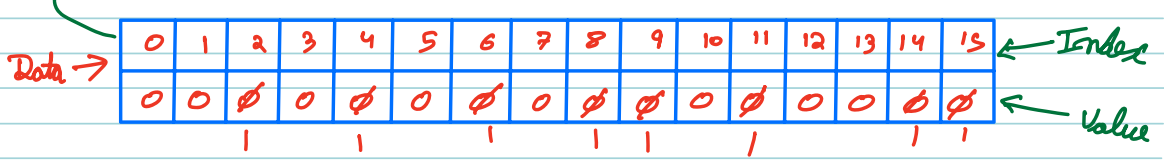
↳ OBSERVATION :- Create an array & mark the presence of an element against that particular Index.

A[] = { 2, 4, 11, 15, 6, 8, 14, 9 }

↓

So In the case of above example, we need to Create array of 16.

→ Data Access Table (DAT)



Code :-

```
for (i = 0; i < A.len; i++) {
    data[A[i]] = 1;
}
```

Advantages of DAT

↳ Time Complexity of Insertion :- O(1)

↳ Time Complexity of Searching :- O(1)

↳ Time Complexity of deletion :- O(1)

=> Issues with such a Representation

1.> Wastage of Space:-

Size of array A[] = {23, 60, 37, 90}, now just to store presence of 4 elements, we'll have to construct an array of size 91.

2.> Inability to Create Big Array:-

↳ If values in array are as big as 10^7, then we will not be able to create this big array. At max array size possible is around 10^6.

3.> Storing values other than positive Integers

↳ We'll have to make some adjustments to store negative numbers as characters. [We need some work-around if we want to store].

TASK :- How to overcome the issue provided we retain the advantage.

↳ Let say we have restriction to create array of size 10.

A[] = {21, 42, 37, 45, 99, 30}

Q How we can do so?

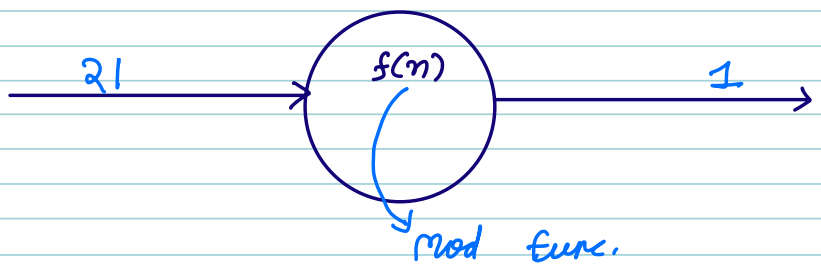
↳ By taking % with 10

A[]%10 = {1, 2, 7, 5, 9, 0}

0	1	2	3	4	5	6	7	8	9	← Index
30	21	42	0	0	45	0	37	0	99	← Value.

Q what we have Done ?

HASHING



→ We have basically done hashing. Hashing is a process where we pass our data through the Hash Function which gives us the hash value (index) to map our data to.

→ In this case, the hash function used is MOD. This is the simplest hash function. Usually, more complex hash function are used.

→ The DAT that we created is known as Hash Table in terms of Hashing.

Q what is Difference b/w hashing & Encryption ?

↳ Encrypted data can be decrypted.

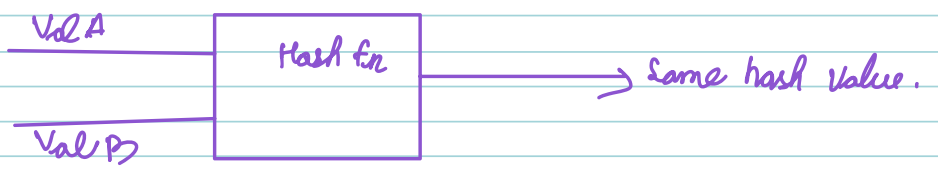
Ex :- 21 —encrypt→ 1 —decrypt→ 21

⇒ ISSUES WITH HASHING

A[] = {21, 42, 37, 45, 77, 99, 31} %10

21 → 1
31 → 1

→ 21 & 31 will map to same Index
→ 37 & 77 " " " " "



when two different values give the same hash value its called COLLISION

Q. Can we avoid the collision?

↳ No, no matter how good your hash function is.

WHY?

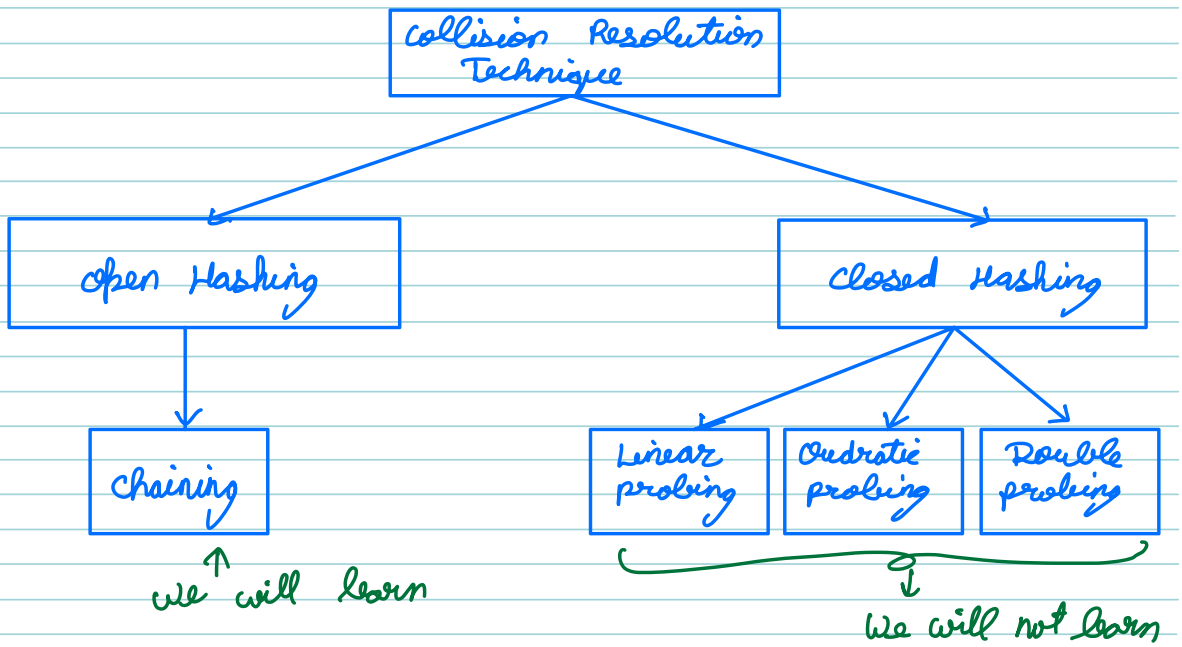
↳ We are trying to map larger range to smaller range. Then collision is bound to happen.

Ex:- Pigeon hole Principle

↳ Let say we have 11 pigeons & 8 holes to keep them.

Now since holes are less, so definitely atleast 2 pigeons will go into 1 hole.

TASK :- We need to find some Resolution to Collision



CHAINING

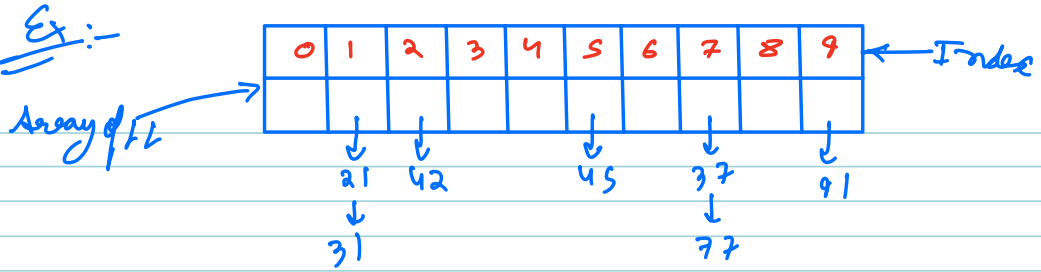
A[] = {21, 42, 37, 45, 77, 99, 31}

Here, 21 & 31 will map to index => 1
37 & 77 " " " " => 7

Q How can we Solve the Collision?

↳ We can have linked list at each index of DAT array.

Ex:-



⇒ This technique is known as CHAINING

CHAINING:- is a technique used in data structures, particularly hash tables, to resolve collision. when multiple item hash to same index, chaining stores them in a linked list or another data structure at that index.

⇒ Time Complexity of Insertion

Ex → 999 % 12 → some value

↳ Insertion at Tail → O(N)

↳ Insertion at Head → O(1)

Since order doesn't matter to us over here, so we can insert at Head

⇒ Time complexity of Deletion & Searching

↳ TC on average is always less than lambda(1).

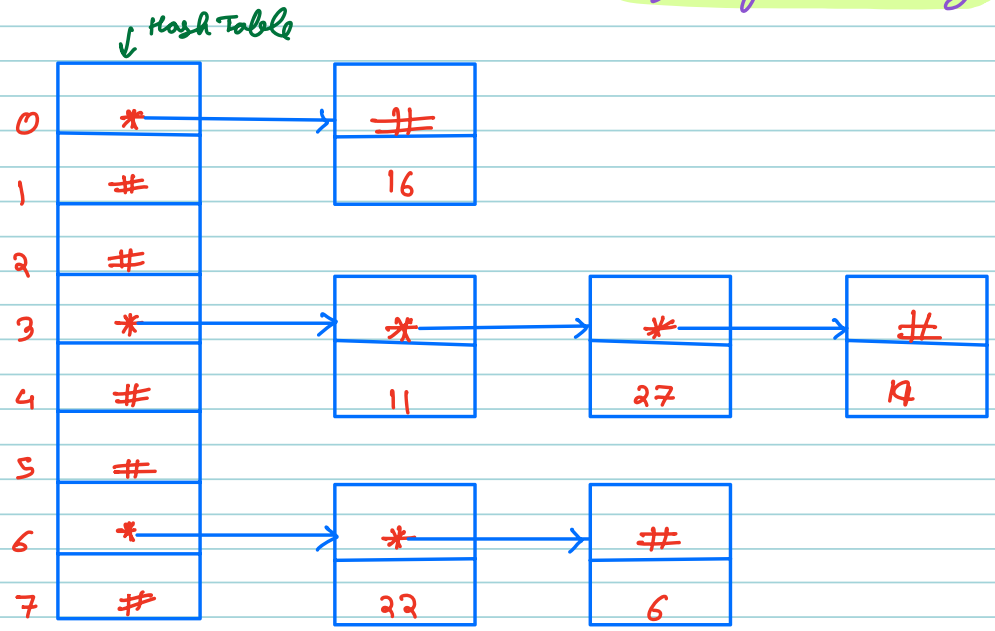
↳ TC in worst can case can be O(N).

TASK :- What is the Lambda (λ)

λambda (λ) function is nothing but a ratio of

= number of elements inserted / Size of the array

Ex



Total Array Size = 8

Inserted elements = 6

$\Rightarrow \lambda = \frac{6}{8} = 0.75$

Ex:-

Total Size of Array = 8

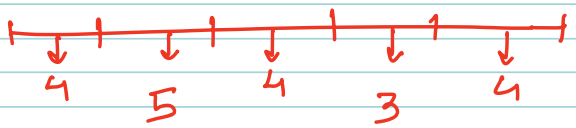
Inserted elements = 16

$\Rightarrow \lambda = \frac{16}{8} = 2$

Alternatively we can say

LAMBDA(λ) = Avg. no. of element in every index of array.

20 elements :-



Q [Scenario] How we will make sure Avg. remains < λ.

⇒ Size of Array = 8

↳ Total element → 10 | λ = 10/8 = 1.2...

↳ Total element → 16 | λ = 16/8 = 2

↳ Total element → 17 | λ = 17/8 = 2.12...

Q How you will make sure avg TC < λ

↳ Grade Double length

∴ λ = Load Factor

Conclusion :- Let's assume the predefined threshhold be 0.7. The λ (Load factor) is exceeding this value. So we will need to rehash the table.

CODE IMPLEMENTATION

ArrayList<Integer> ——————

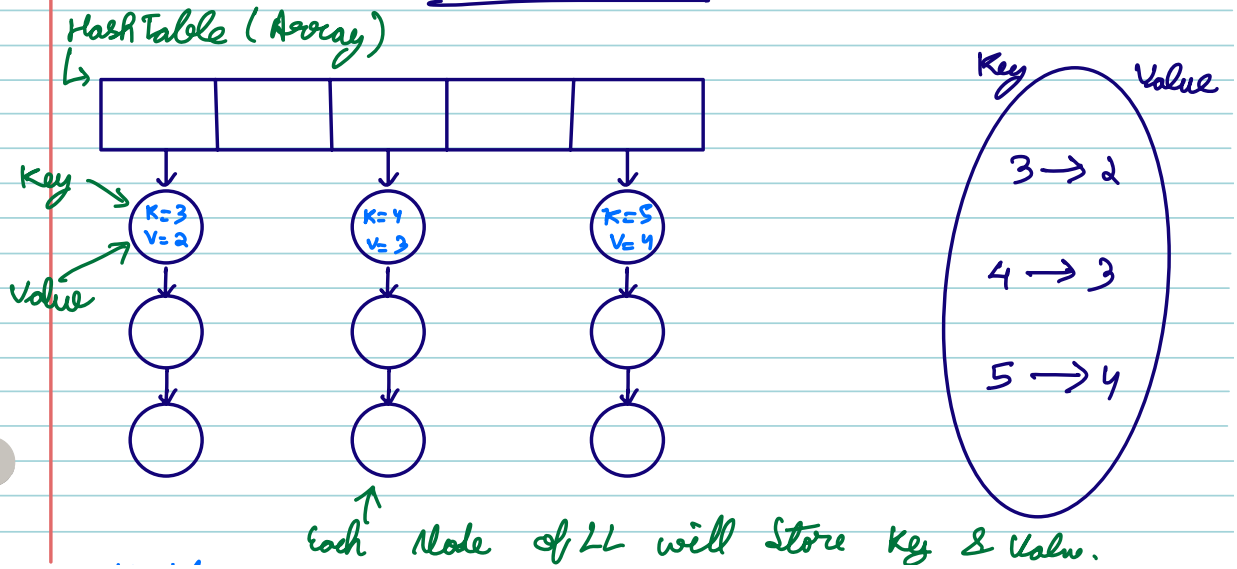
ArrayList<String> ——————

Generic

```
class ArrayList<T> {

    T get(int index) {

    }

}
```

→ (return T type of dd)

HASHMAP



(10, 50)
K V

K V
HashMap<Integer, String> hm = new HashMap<>();

General convention
class HashMap<K, V> {

Private class HMNode {

K Key

V Value

Public HMNode(K a, V b){

Key = a;

Value = b;

}

}

Public LinkedList<HMNode>[] buckets;

Count of Inserted elements → Private int size;

Public HashMap() {

buckets = new LinkedList[4];

for(i = 0; i < buckets.len; i++){

buckets[i] = new LinkedList<>();

}

}

```
Public void put ( K key, V value ) {

    int bno = hashFunction (key);
    HMNode fdat = foundAt ( buckets[bno], key);

    if (fdat == NULL) {
        HMNode hm = new HMNode (key, value);
        buckets[bno]. addLast ( hm);
        this. size ++;
    } else {
        fdat. value = value;
    }

    double lambda = this. size * 1.0
                    this. buckets. len

    if ( lambda > 2.0) {
        rehash ()
    }
}
```

Internal LL → buckets[bno]. addLast (hm);

And it maintains Tail.

```
Public V get(K key){
    int bno = hashFunction(key);
    HMNode fdat = foundAt(buckets[bno], key);

    if(fdat == NULL){
        return NULL;
    }else{
        return fdat.val;
    }
}
```

```
Public V remove(K key){
    int bno = hashFunction(key);
    HMNode fdat = foundAt(buckets[bno], key);

    if(fdat == NULL){
        return NULL;
    } else {
        buckets[bno].remove(fdat);
        this.size--;
    }
    return fdat.value;
}
```

internal LL's remove method.

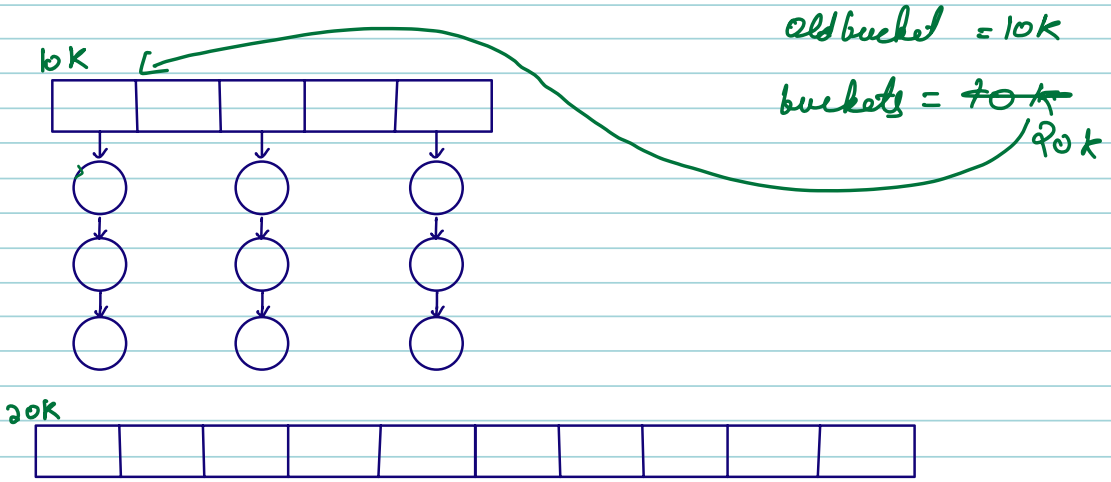
```
Public boolean containsKey(K key){
    int bno = hashFunction(key);
    HMNode fdat = foundAt(buckets[bno], key);
    if(fdat == NULL){
        return False;
    } else {
        return True;
    }
}
```

```
Private int hashFunction(K key){
    int idx = key.hashCode()
    idx = math.abs(idx);
    idx = idx % bucket.len
    return idx;
}
```

↳ Each Interval Class have this func. It returns a unique value for given data but two same data will have same hashcode.

```
Public HMNode foundAt(LinkedList<HMNode> list,
                      K key){
    for(HMNode Node : list){
        if(Node.key.equals(key)){
            return Node;
        }
    }
    return Null;
```

```
}

Public void Rehash() {
    LinkedList<HMNode>[] oldbucket = buckets;
    buckets = new LinkedList[2 * oldBucket.len];

    for(i=0; i < buckets.len; i++){
        buckets[i] = new LinkedList<>();
    }

    for( i=0; i<oldbucket.len; i++){
        for( HMNode curr : oldbucket[i]){
            this.put( curr.key, curr.Value);
        }
    }
}
```



↑ New bucket

Quiz 1 :- what is the TC of brute force approach for checking the existence of element in the array for Q queries?

↳ O(N*Q)

Quiz 2 :- what advantages does Direct Access Table (DAT) provide in terms of TC for insertion, deletion, & search operations.

↳ O(1)

Quiz 3 :- what is the purpose of Load factor (λ) in a hashmap.

↳ It determines when to trigger rehashing.

Quiz 4 :- what does the rehashing process involve in a hashmap?

↳ Creating a new hashtable with double the size & redistributing elements.