Internal Implementation of Hashmap

"Raja" : 6

↳ key ↳ value

"Jogesh" : 5

↳ key ↳ value

dist<pair>

```
class Pair{
    int key;
    int value;
}
```

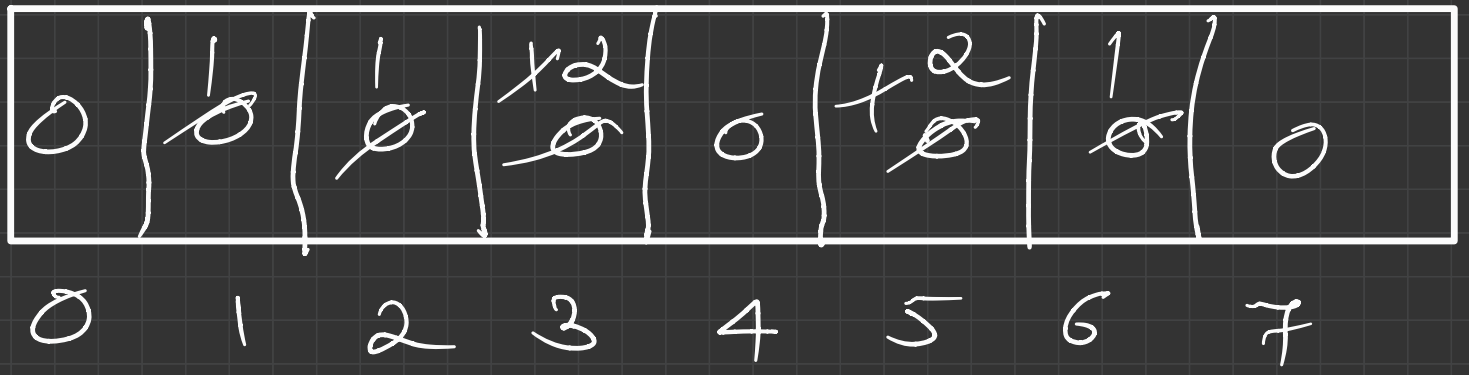
{(2,1,), (3,2), (5,2), (6,1)}

Find value for its corresponding key.

T.C : O(n)

Q: Calculate freq. of an array of integers.

[2, 3, 5, 5, 6, 1, 3]

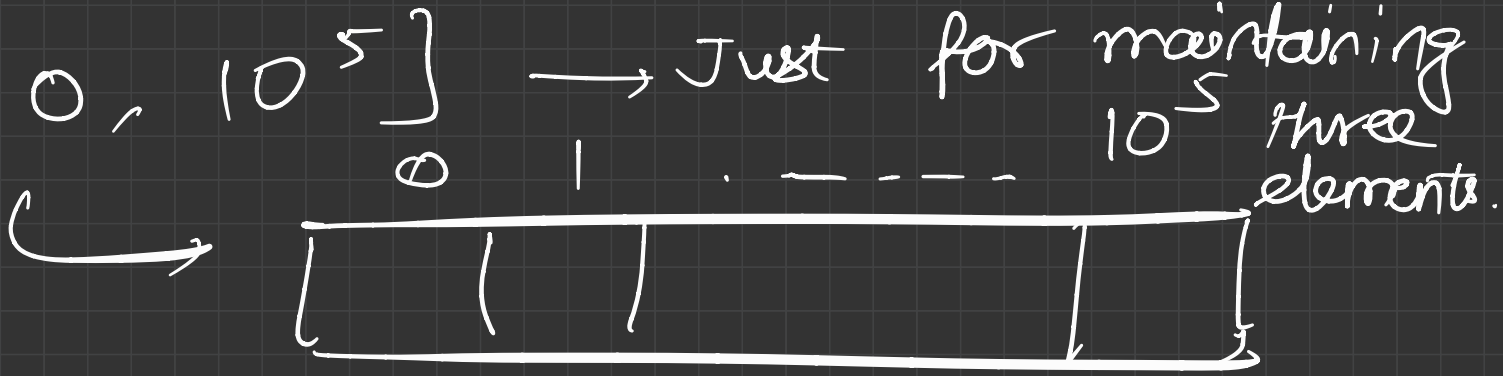


OCD to insert.

return arr[5];

Problems with this approach:

[1, 0, 10^5] → Just for maintaining 10^5 three elements.



Key : ⟶ index

Bucket array

Lets create the bucket array with some size initially (say 20).

Operations:

* Insert key-value pair.
* Remove a key-value pair.
* Find value for a key.

Eg:

"Jagesh" $\xrightarrow{H.F}$ 23

"abc" $\xrightarrow{H.F}$ 409

What if the value from hash func. is greater than max. index of bucket array?

① Hash Func.

② Hash Code.

③ Compression Func.

Hashcode % BucketArray.size()

object (key) $\xrightarrow{H.F}$ Hashcode $\xrightarrow{\%\ bucketsize}$ index of bucket array

Hash Function should be consistent.

"5" (Integer) ⟶ return 5

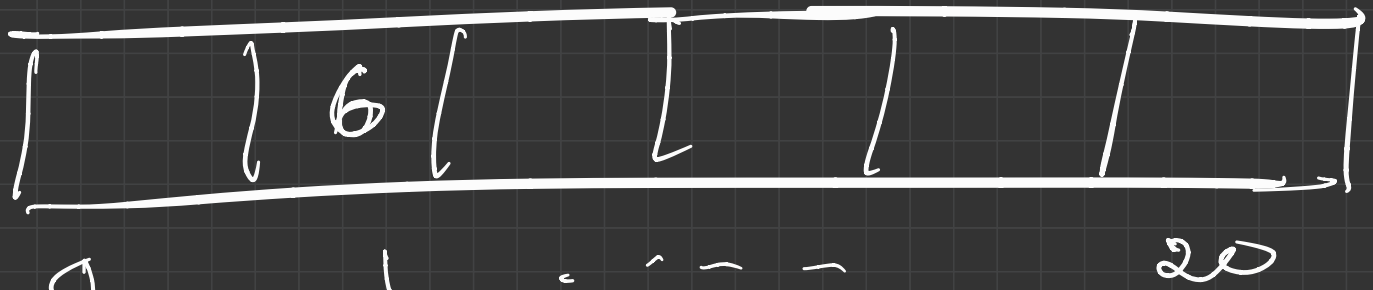
"6" (Integer) ⟶ return 6

```
int hashFunc(int a){
    return a;
}
```

"logi" → sum of ASCII characters

"iqol" → same hash code

HF("logi") = 6 HF("iqoL") = 6



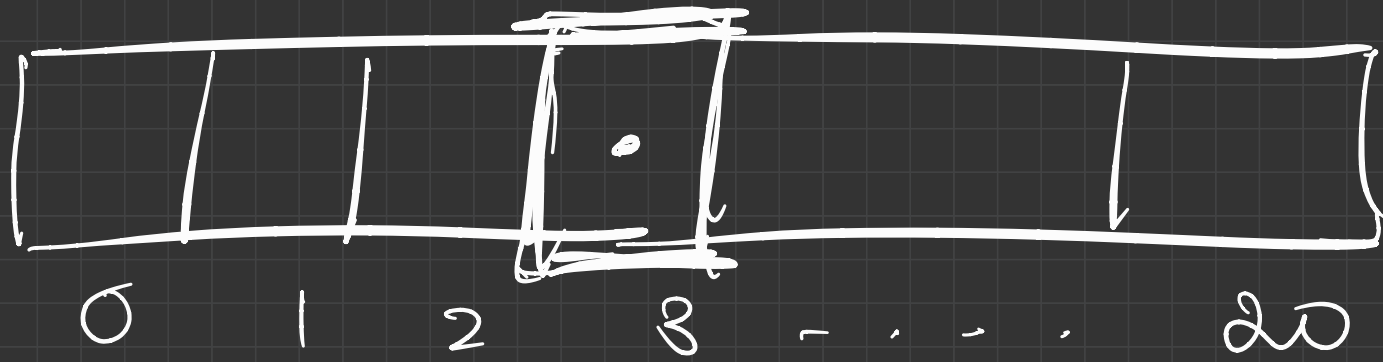
Hash Collision: In the same index, multiple keys are present.

Hence we need good Hash Func. to minimize Hash collisions.

Hash Collision:

* Closed Hashing / separate chaining.
* Open addressing

Deal with Hash Collision.



In a particular index, store a list of values.

Bucket Array is a list of head of Linked List.

Open Addressing:

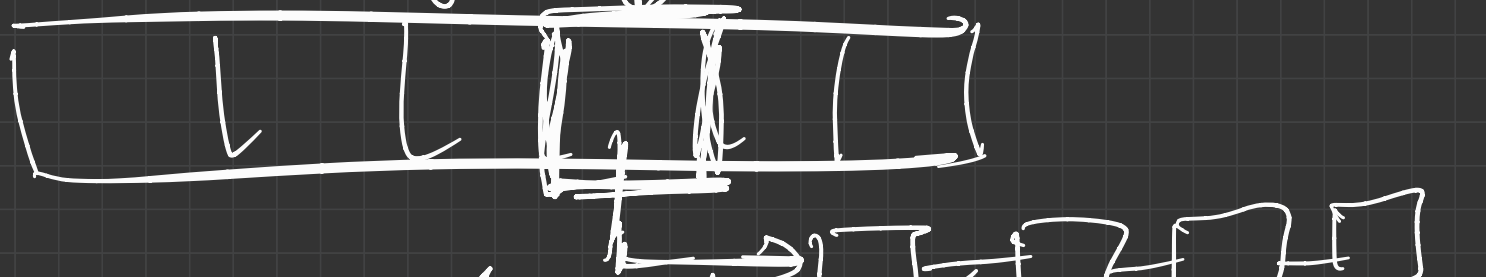
If there is a hash collision, find an alternative index.

- ↳ Linear Probing
- ↳ Quadratic Probing
- ↳ Double Hashing

In Java, the internal implementation is closed hashing / separate chaining.

Time Complexity:

Insert : key → H(F) % bucketsize → O(Constant)



n → no. of entries

b → no. of bucket

O(n) worst case.

$n \Rightarrow$ no. of entries

$b \rightarrow$ no. of buckets

At each index, the LL won't be greater than n/b

↳ load factor.

$\frac{n}{b} < 0.75$

lets take 0.7

$\frac{n}{b} < 0.7$

If it is greater than 0.7, we do something known as rehashing.

REHASHING:

$n/b \downarrow \Rightarrow b \uparrow\uparrow$

Double the bucket size.

Time Complexity for rehashing:

$O(n)$ → worst case.

However, this does not happen frequently.

Hence

Avg. T.C ⇒ $O(1)$.