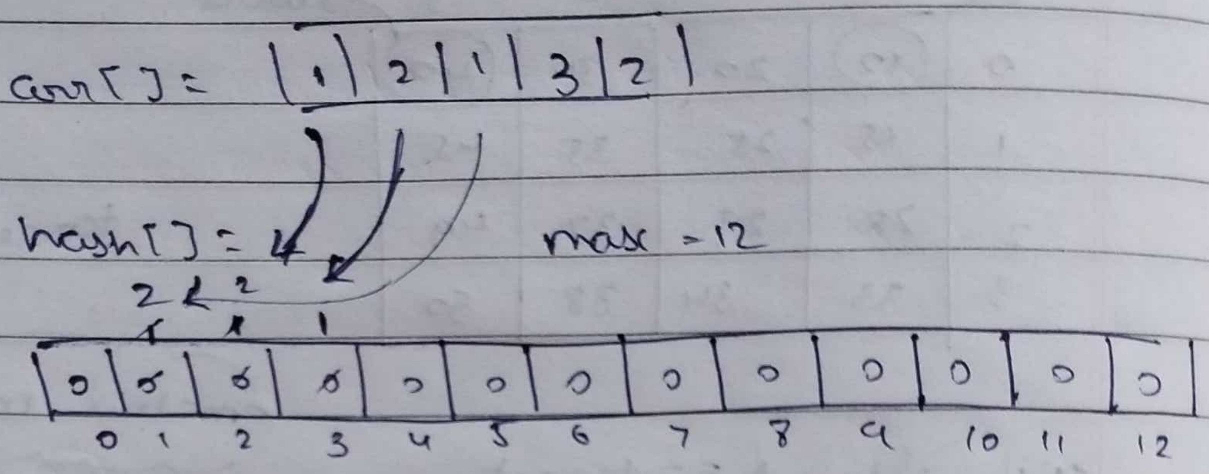
* HashMap:

→ provides functionality of hash table

→ Stores element in key/value pair



key	value
↓	↓
unique	may / may not be same

① Create a HashMap.

import java.util.HashMap package

syntax:

```
HashMap <K,V> numbers = new HashMap<>()
```

here K = key type

V = type of value

eg:

Hath

```
HashMap < String, Integer> numbers
    = new HashMap <>();
```

② Add Elements in HashMap.

map.put()

↙ ↘

exists → update value.

doesn't exist → new pair is inserted

eg:

```
numbers.put("One", 1)
numbers.put("Two", 2)
Sout("Hashmap", + numbers)
```

O/P: {One = 1, Two = 2}

③ Access Elements

key ?

map.get()

↙ ↘

key exist → value

does not exist → null

map.containsKey()

↙ ↘

✓ → true

× → false

eg:
```
if (map.containsKey("China")) { //true
}
```

→ this returns true or false

eg: map. get ("China")

-> returns the value if exists.

map. keySet() -> returns the keys

map. ~~Key~~value() -> returns values

map. entrySet() -> returns key / value

④ Replace value

map. put (2, "Python");

map. replace (2, "C++");

Here Python is replaced with C++

⑤ Remove element

map. remove (2)

=> Here, the key-value pair with key 2 will be removed.

⑥ Iterate through a HashMap.

→
```
// Iterate through keys
for (Integer key : map.keySet()) {
        sout (key)
 }
```

O/p: 1, 2, 3

→
```
// Iterate values only
for ( String value : map.values ()) {
        sout (value )
 }
```

O/p : Java, Python, C++

```
// Iterate through key/value entries
```

→
```
for (Entry < Integer, String> entry : map.entrySet()
        Sout (entry)
 }
```

O/p: 1=Java, 2= Python, 3= C++

if optional parameter is not used, then
default capacity → 16
load factor → 0.75

⑦ Create HashMap from other Maps.

```
TreeMap< String, Integer> evenNumbers =
      new TreeMap <> ();
```

```
// create HashMap from treemap
```

```
HashMap < String , Integer> numbers =
     new HashMap <> (evenNumbers);
```

NOTE

While creating a hashmap, we can includ
optional parameters : capacity and
load factor.

eg:

```
HashMap <K, V> numbers = new HashMap<> (8,0.6
```

Here : 8 (capacity) → means it can store
8 entries

: 0.6f (load factor 0.6) → means
whenever our hash table is
filled by 60%, the entries
are moved to a new hash
table double the size of
original hash table

* Character Hashing.

S= "abcdabefc"

0	0	0	0	0	....	0
0	1	2	3	4	.....	25

ASCII : 'a' → 97

'z' → 122

a = a - a = 0

b = b - a = 1 - 0 = 1

f = f - a = 102 - 97 = 5

* Time Complexity

→ unordered-map

Storing }
fetching } → O(1) {avg, best}
→ O(N) {worst}

↓ N= no of elements in map.

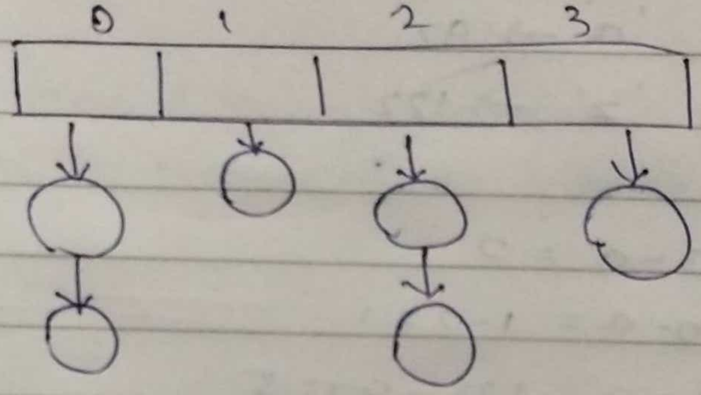
→ ordere

Storing }
fetching } log(N) → best
→ avg
→ worst

* Hashmap Implementation / working

Hashmap → array of Linked List (LL)

n(nodes) = 6 // number of nodes
N(size) = 4 // array size.
= no of ~~buckts~~ buckets



→ to make O(1) we need to maintain λ
O(λ)

$\boxed{\lambda = \frac{n}{N}}$ where $\lambda \le K$ (const value)
(threshold)

here n = 6 and N = 4
$\therefore \frac{n}{\cancel{4}N} = \frac{6}{4} = 1\ldots$

→ if one node increase then 7/4 = 1....
one more then $8/4 = \underline{\underline{2}}$

→ here time complexity is increased.

→ So to decrease the complexity, we have to increase the array size

∴ now N= 8

0	1	2	3	4	5	6	7

→ Now the nodes from the previous array with N=4 will be arranged in the new array with N=8 using any <u>hash function</u>.

now $\frac{n}{N} = \frac{6}{8} = 0....$ $\qquad \frac{8}{8} = 1$

NOTE:

If the array size changes, then the bucket index also changes.