Creation of Hashmap :→

Hashmap → key, value

- void put
  - key present → update value
  - key Absent → Insert key-value pair
- value remove
  - key present → value return after removal
  - key Absent → null return
- value get
  - key present → value return
  - key Absent → null
- boolean contains key
  - key present → true
  - key Absent → false
- ArrayList<key> keySet → return all keys in a set
- int size → no. of keys
- void display → key-value pair printing.

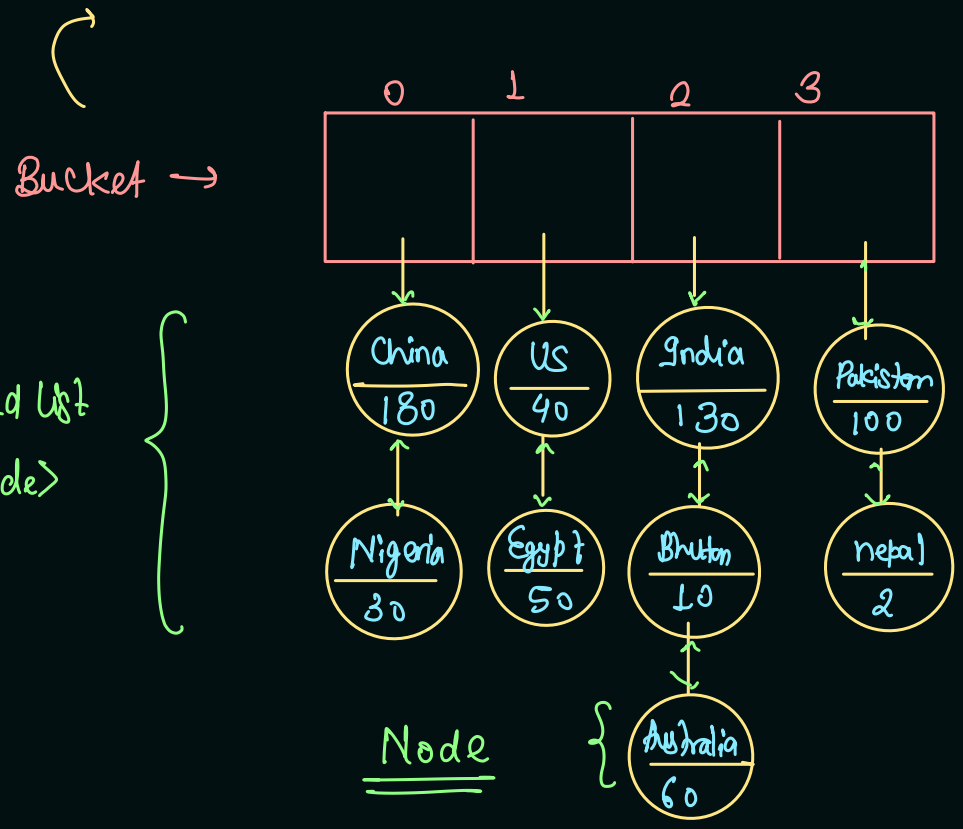
Assumption:

- key ∈ String
- value ∈ Integer

Behaviour of Hashmap:

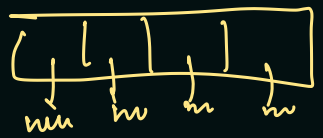
bucket → Initial Capacity ⇒ (4)

LinkedList<Node>[] bucket;



Node

key
value.



Key - Unique. population of countries

Hashing on Key.

Bucket order using HashCode.

		key		value.
2	←	India	→	125
0	←	China	→	180
3	←	Pakistan	→	100
1	←	US	→	40
3	←	nepal	→	2
2	←	Bhutan	→	10
1	←	Egypt	→	50
0	←	Nigeria	→	30
2	←	Australia	→	60
2	←	India	→	130

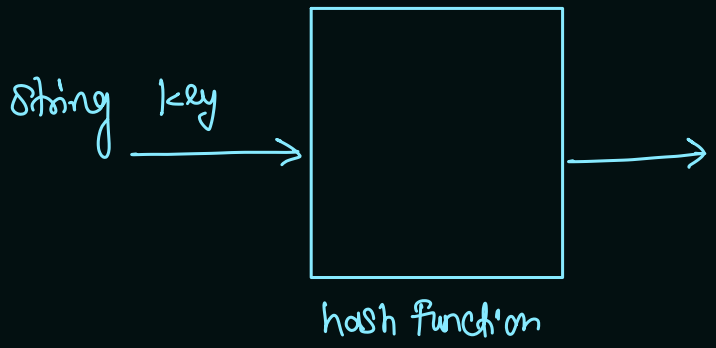
HashCode("India") = x

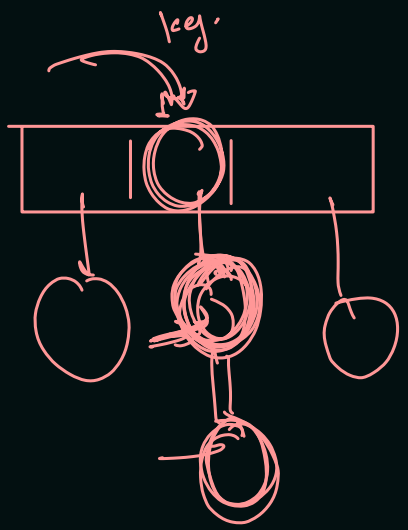
always same for India.

put function in hash map :

void put(String key, int value):

→ How do we know the bucket index of key.





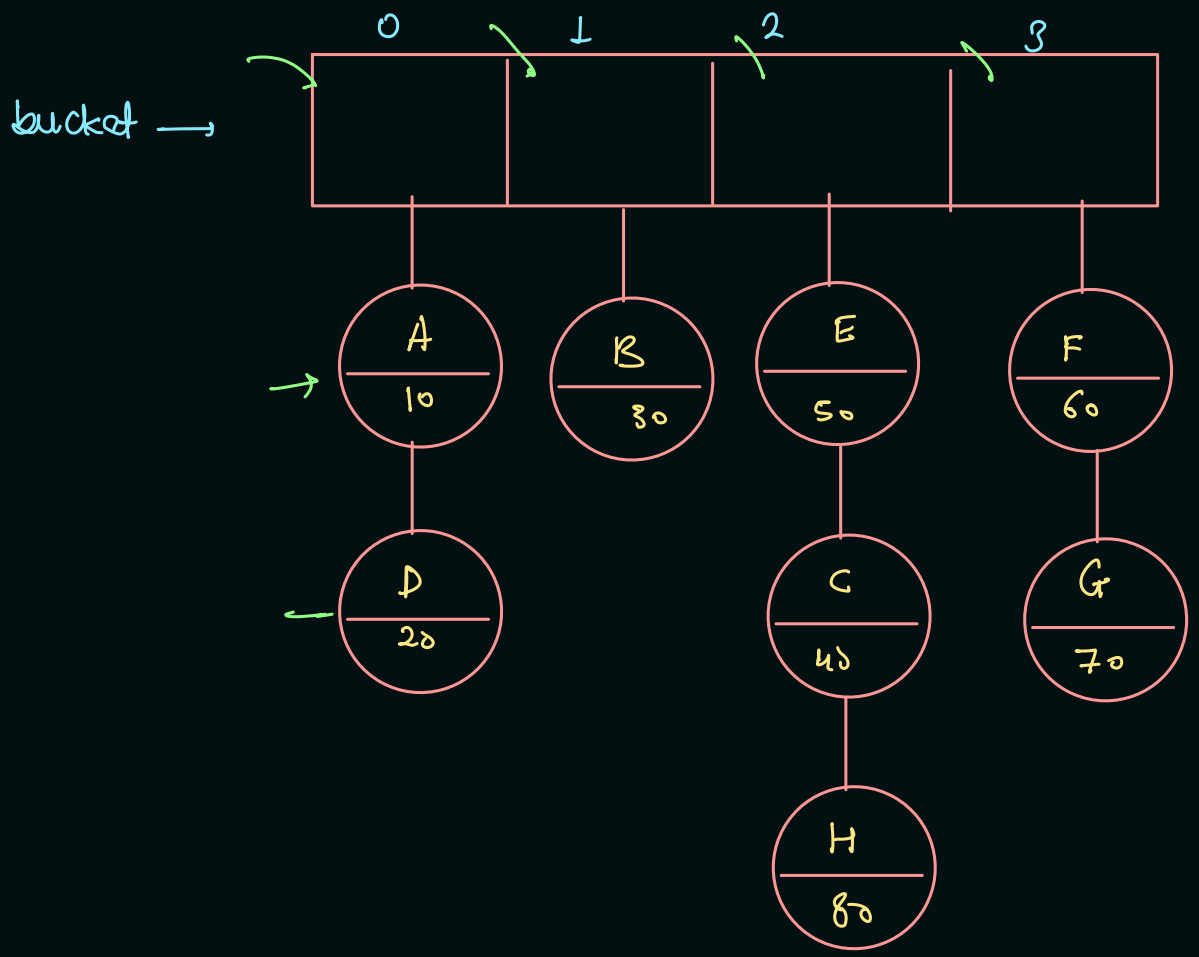
hash function

↳ Java (Default) for every inbuilt class.

find hashCode of key.

↳ convert hashCode into bucket index.

keySet function :

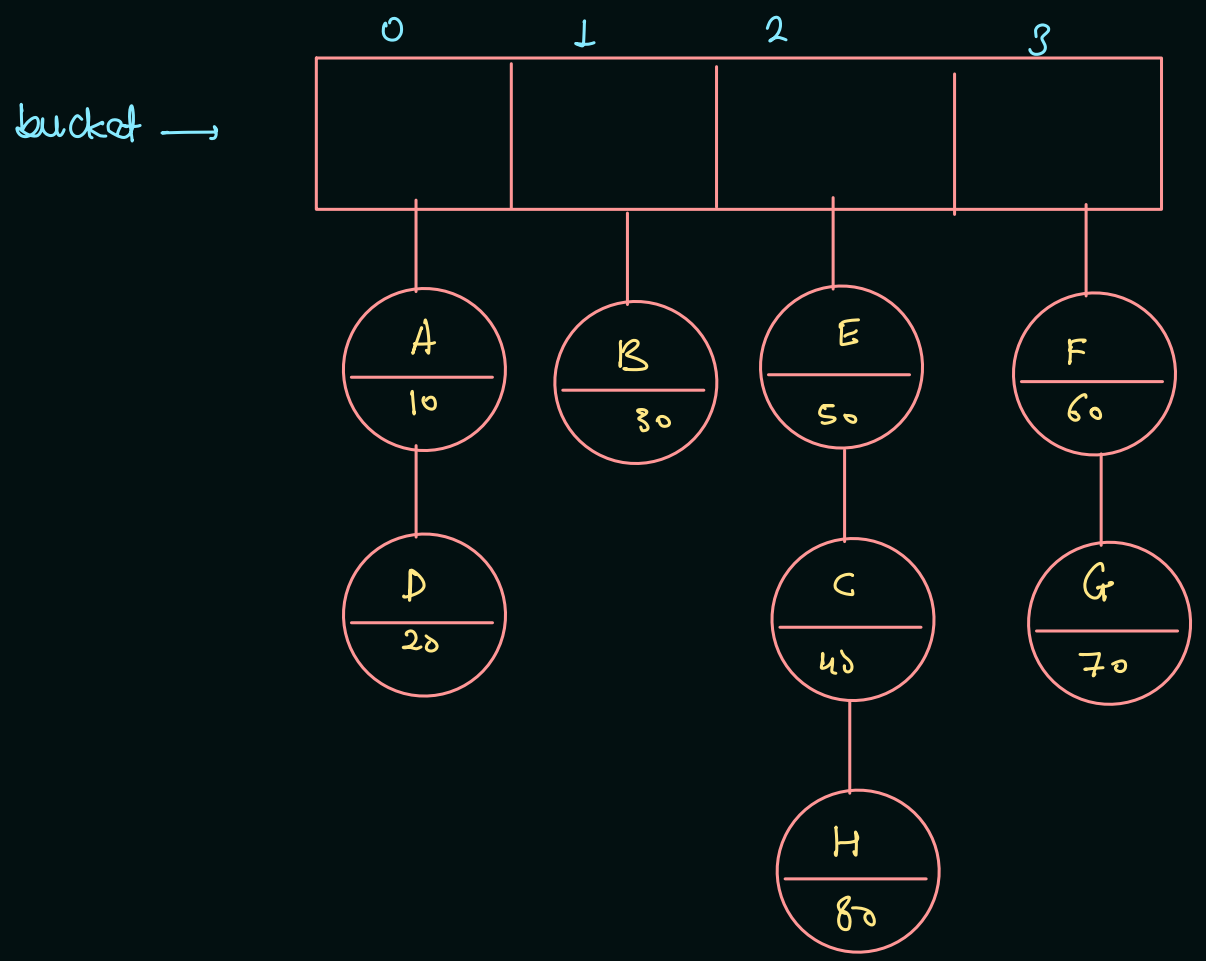


to get keys in an arraylist →

travel on every index of bucket & add key in an AL.

```
for(int bi=0; bi<bucket.length; bi++){
    for(Node node: bucket[bi]){
        list.add(node.key);
    }
}
```

Display :



Hash map View

bucket = 0 → [A → 10], [D, 20]

bucket = 1 → [B → 30]

bucket = 2 → [E - 50], [C - 45], [H - 80]

bucket = 3 → [F - 60], [G - 70]

Display:

[A → 10]

[D → 20]

[B → 30]

[E → 50]

[C → 45]

[H → 80]

[F → 60]

[G → 70]

Get function in Hashmap!

int get(string key)

1. check if key is present or not.
2. if key is present, return value associated with key
3. otherwise, return -1;

```
[Australia = 100]
[Nepal = 12]
[pak = 90]
[US = 70]
[Japan = 50]
[Bhutan = 55]
[India = 125]
[Egypt = 75]
```

Display

```
bucket : 0 -> [Australia = 100], [Nepal = 12],
bucket : 1 ->
bucket : 2 -> [pak = 90], [US = 70], [Japan = 50], [Bhutan = 55],
bucket : 3 -> [India = 125], [Egypt = 75],
```

```
[Australia = 100]
[Nepal = 12]
[England = 175]
[pak = 105]
[US = 70]
[Japan = 50]
[Bhutan = 55]
[newzeland = 156]
[India = 130]
[Egypt = 75]
```

update

fast

bi → O(1)

di → bucket.bi.length

get

```
bucket : 0 -> [Australia = 100], [Nepal = 12],
bucket : 1 -> [England = 175],
bucket : 2 -> [pak = 105], [US = 70], [Japan = 50], [Bhutan = 55], [newzeland = 156],
bucket : 3 -> [India = 130], [Egypt = 75],
```

```
hashmap map = new hashmap();
map.put(key: "India", value: 125);
map.put(key: "pak", value: 90);
map.put(key: "US", value: 70);
map.put(key: "Australia", value: 100);
map.put(key: "Japan", value: 50);
map.put(key: "Nepal", value: 12);
map.put(key: "Bhutan", value: 55);
map.put(key: "Egypt", value: 75);
map.display();
map.hashmapView();
map.put(key: "India", value: 130);
map.put(key: "pak", value: 105);
map.put(key: "newzeland", value: 156);
map.put(key: "England", value: 175);
map.display();
map.hashmapView();
```

Discussion for better Management of Time Complexity in Hmap!

put ⟶ O(bucket[bi].length)

get ⟶ ''

remove ⟶ ''

containsKey ⟶ ''

keySet ⟶ O(n)

After maintain of load factor

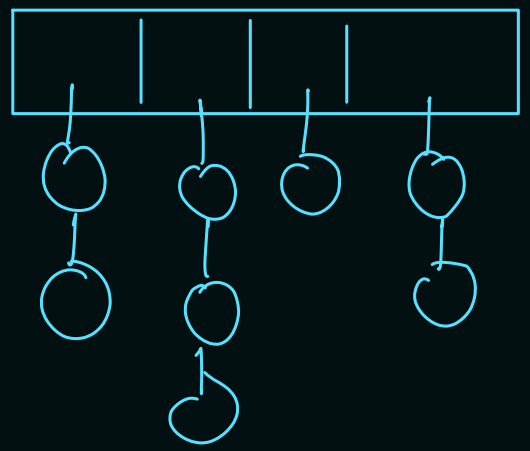
O(λ) ⟶ $0 \leq \lambda \leq 2$

O(1)

O(1) Lambda factor

O(1) ↳ ration of no. of nodes per bucket.

O(1)



N = 4 (bucket.length)

n = 6 [size]

$$\lambda \text{ (load factor)} = \frac{n}{N}$$

At every insertion load factor will Increase.

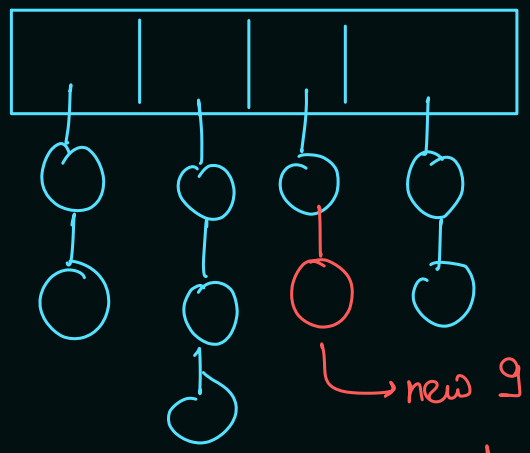
We have to maintain value of load factor.

$0 \leq \lambda \leq k$ ⟶ Range of load factor

$0 \leq \lambda \leq 2$ ⟶ no. of node per bucket is maximum = 2

upper value of k = 2

How to maintain load factor (lambda value) :



complexity is cross than O(2)

→ new insertion

$\lambda = \frac{n}{N} = \frac{8}{4} = 2$

$\lambda = \frac{n}{N} = \frac{9}{4} = 2.25$

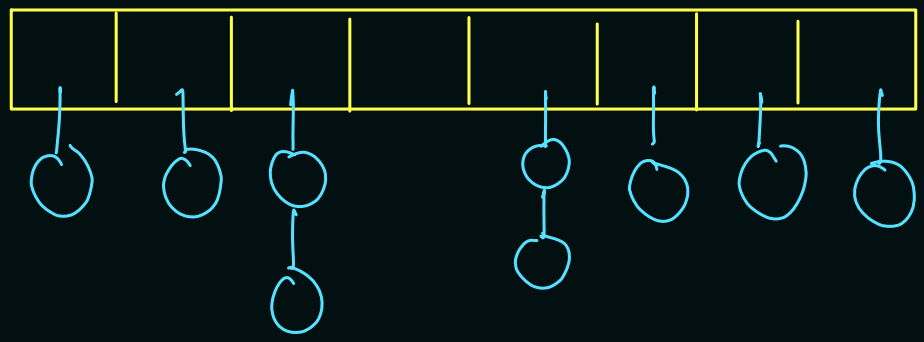
$\lambda > 2$]

Decrement in 'λ' is known as rehashing.

rehash() →

Rehashing.

capacity will become twice of oldcap.



After Rehash, $n = 9$ (no. of nods), $N = 8$ (size of bucket)

$\lambda = \frac{n}{N}$

$\lambda = \frac{9}{8} = 1.\ldots$

$0 \leq \lambda \leq 2$] lambda is in Range.

→ complexity is cover

How do we perform Rehash()

Impact on size ⟶ put function

In put function - if λ is greater than 2
then perform rehash.

Rehash() ⟶
1. Make new bucket twice of its older size.
2. Again put all key value pair in new bucket from old bucket.
3. Make new bucket as older one.

Because put, get have $O(\lambda)$ → time complexity.
they are not exactly → $O(1)$
but they are Amortised $O(1)$