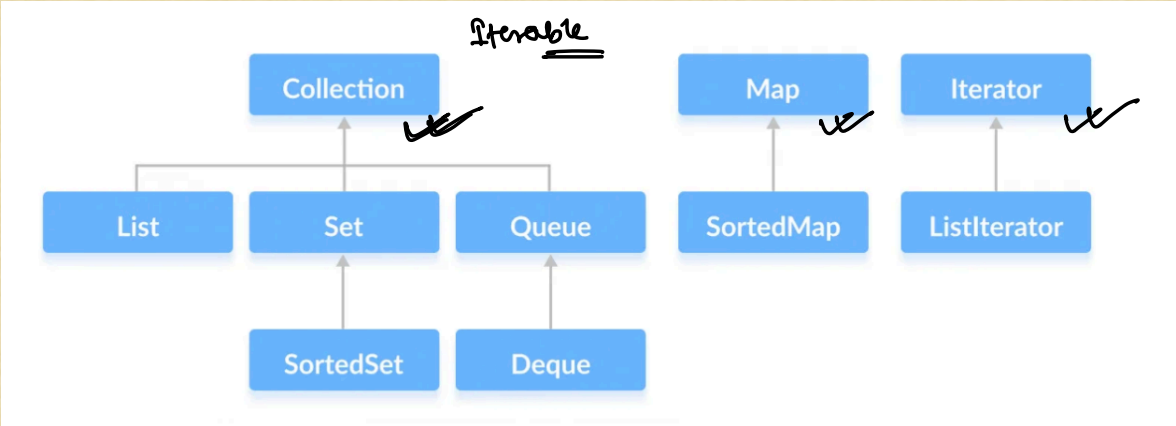
* Collections

↓

set of classes and interfaces that give us reusable commonly used data structures.

Java Collection framework (JCF)



⇒ methods [Collection]

→ add()

→ size()

→ remove()

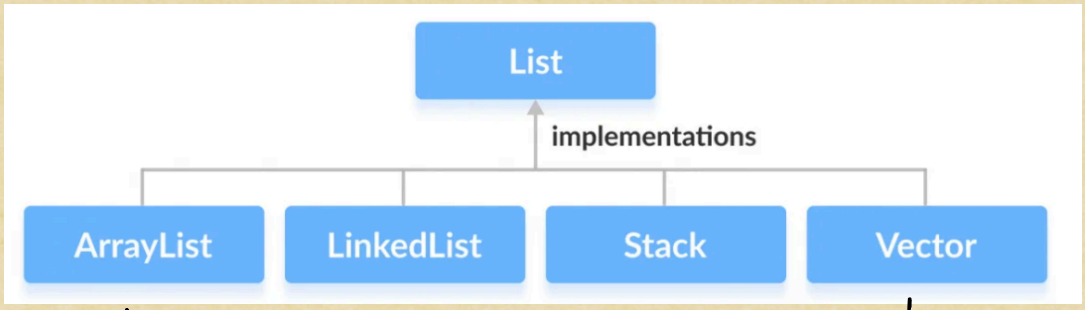
→ iterator() → gives an iterator Object that helps to iterate a DS.

→ addAll(<>)

→ removeAll(<>)

collection goes as an i/p parameter

→ clear() → remove everything



↓ dynamic array → CopyOnWriteArrayList [threadsafe arraylist]

↓ Synchronized List → [thread safe] * outdated

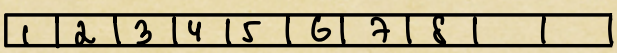
* How does an arraylist work ?

Ans. DS in Java ⇒ internally ⇒ array or LinkedList

Arraylist ⇒ initialsize, loadfactor

10 ⇒ default ; 0.75 ⇒ default

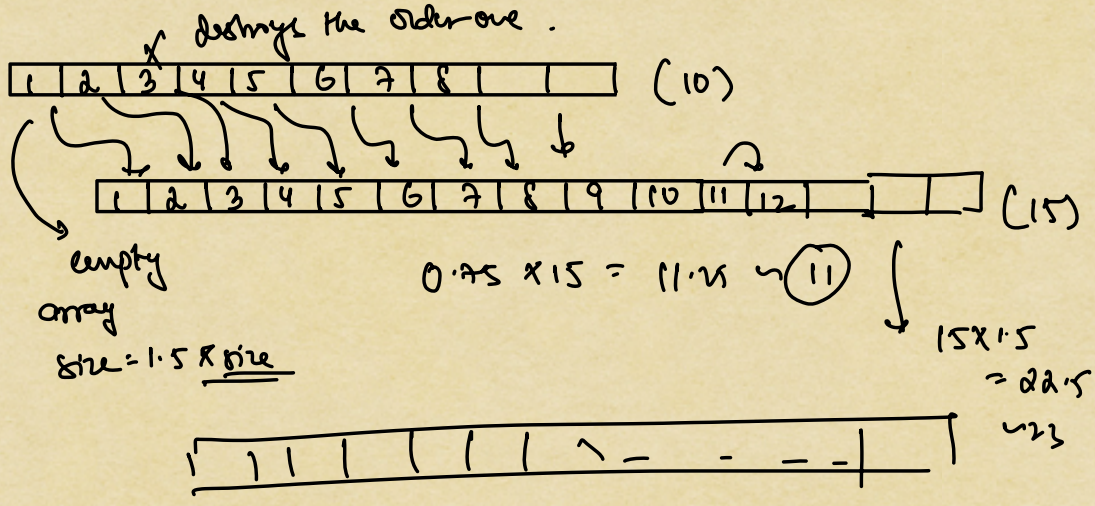
↓ array



LF ⇒ % till which we can insert items directly

0.75 X 10 ⇒ 7.5 ⇒ 8 items

size = size + $\frac{size}{2}$ ⇒ size ⇒ 10 15 (size)



Class Students ⇒ 50 ArrayList(10)

40 60-70 arraylist(50, 0.8).

TC ⇒ insertion ⇒ O(1)

get(i) ⇒ O(1)

search( ) ⇒ O(N)

index ⇒ iterator

ordered

duplicate elements (✓)

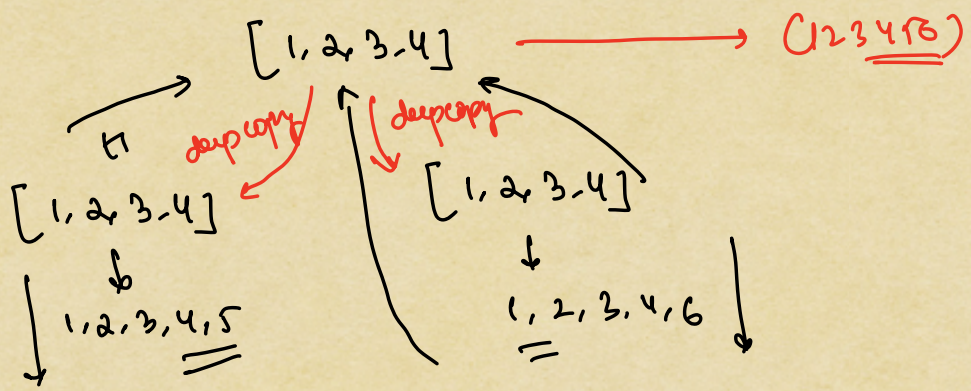
null (✓)

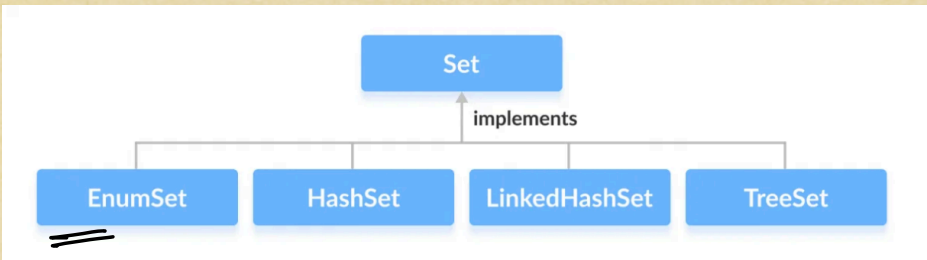
multiple nulls (✓)

⇒ CopyOnWriteArrayList :-

→ allows multiple threads to read the value from arraylist.

→ whenever any thread wants to write on it, it makes a copy for that thread.





(Set) ⇒ DS ⇒ * unordered data
* non-duplicate data
* null (✓)
* multiple nulls (X)
* indexed (X)
* add() | remove() → 1

* search = O(1)

Enum

↓

collection of constant value

```
enum Vehicle{
    TWO_WHEELER,
    THREE_WHEELER,
    FOUR_WHEELER
}
```

→ duplicate enum values are not possible

→ internally ⇒ SET

* HashSet ⇒ basic implement of set

index ✗

iterate ⇒ Iterator()

⇒ Maps

DS ⇒ <K,V> ⇒

<K,V>
<K,V>
<K,V>
<K,V>

⇒ insertion (k, v) get(k, v) removal → O(1)

⇒ search (k) → O(1)

⇒ duplicate keys → (X)

⇒ null key → (✓)

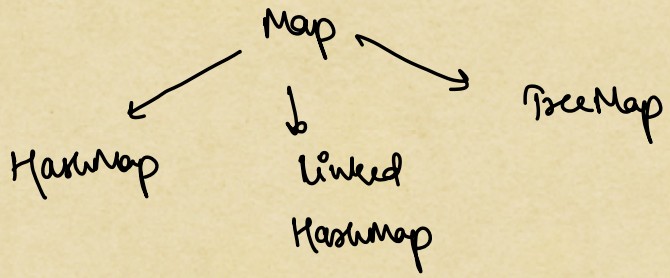
⇒ multiple keys → (X)

⇒ duplicate values (✓)

⇒ null values (✓)

⇒ multiple null values (✓)

≒ amortised



⇒ How hashmap works ?

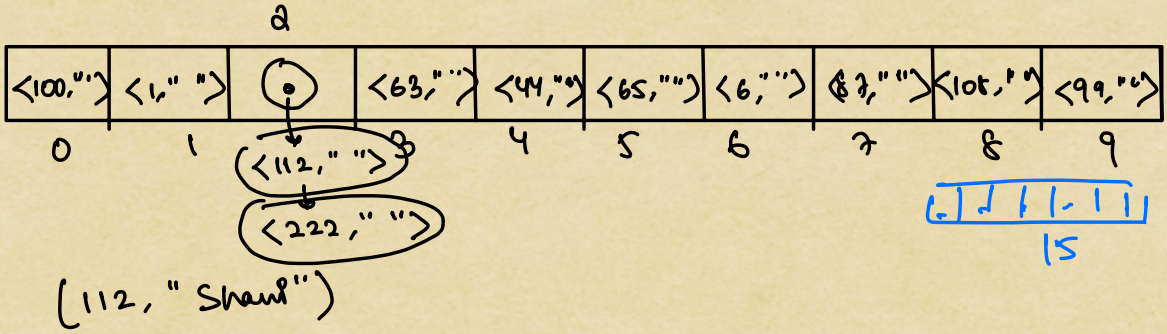
Ans Entry <K, V>

hashmap → array of entry<K,V> objects

(Entry<K,V> [ ]) ∈ Hashmap Storage

Hashmap < Id, name > ⇒ (10)

Integer String



(112, "Shani")

hash function ⇒ [default hashcode]

↓

digit % size ⇒ index

112 ⟶ hash(112) ⟶ 112 % 10 ⇒ 2

(63, "Chani")

63 ⟶ hash(63) ⟶ 63 % 10 ⇒ 3

↓ ↓ ↓

O(1) O(1) index ⇒ O(1)

insertion ⇒ O(1)

get(63) ⇒ hash(key)

↓ ↓

key digit % size ⇒ ✓

63 → hash(63) → 63 % 10 ⇒ 3 ← index O(1)

hash(63): O(1) 63 % 10: O(1)

get( ) ⇒ O(1)

(initial capacity, load factor) ⇒ both are updatable

initial capacity → 10

load factor → 0.75

entry ⇒ (<222, "Amit">)

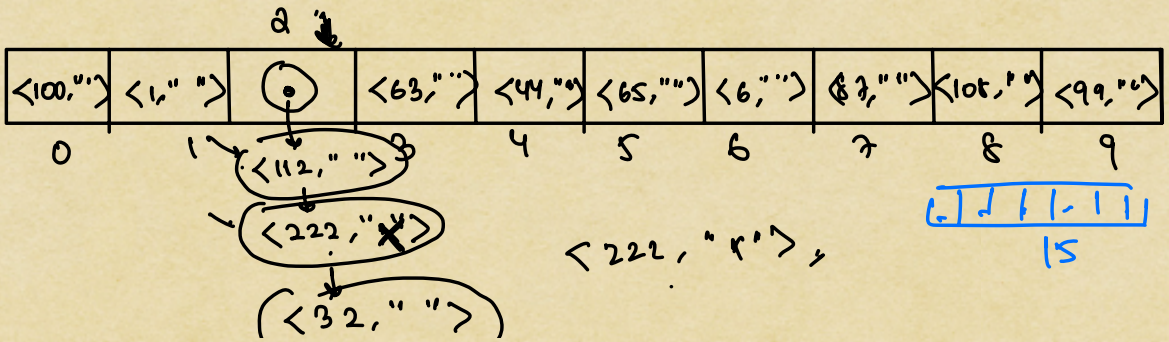
↓

hash(222) → 222 % 10 ⇒ 2

⇒ when we get an index for which already another value is present "collision"

* moment collision happens ⇒ LL at that index

<32, " "> ⇒ 2



get(32) ⇒ hash(32) ⇒ 32 % 10 ⇒ (2)

↓

LL ⇒ search in LL

↓

O(N)

⇒ fixed the size of LL

↓

max^m LL at an index can grow to a size of 8.

↓

8th collision (8th entry at the same index)

↓

LL automatically converts to a BBST (balanced BST)

↓

[Red black tree]

TC ⇒ worst complexity ⇒ BBST (element)

↓

O(log N)

* collision chances will decrease the more elements we add,

* 8 ⇒ search ⇒ O(8) ≈ O(1)

* BBST ⇒ O(log N) ↓ (extremely low)

$\log_2(2^{32}) \Rightarrow$ 11 iterations $\leq O(1)$

$\hookrightarrow 2K \Rightarrow O(1)$.

hashmap(1000) $\Rightarrow$ BBST $\Rightarrow$ < 10 iterations

↓ (Pudin, LL, BBST) ↓ $\hookrightarrow O(1)$

↓ average get / search $\Rightarrow O(1)$ [amortised]

10 $\Rightarrow$ (x % 10) $\Rightarrow$ (0-9) $\Rightarrow$ 10%

↓

15

↓

15 $\Rightarrow$ (x % 15) $\Rightarrow$ (0-14) $\Rightarrow \frac{100}{15} \Rightarrow 6.66\%$

$\Rightarrow$ HashSet

↓

hashmap with all values as static data

↓

<k, ->

<k, ->

<k, ->

<k, ->

=> LinkedHashMap => normal HashMap

↓

entry ordered

LinkedList — (maintains index count)

LinkedHashSet → LinkedHashMap

↓

entry ordered

LinkedList — (maintains index count)

=> TreeMap => sorted by keys

↓

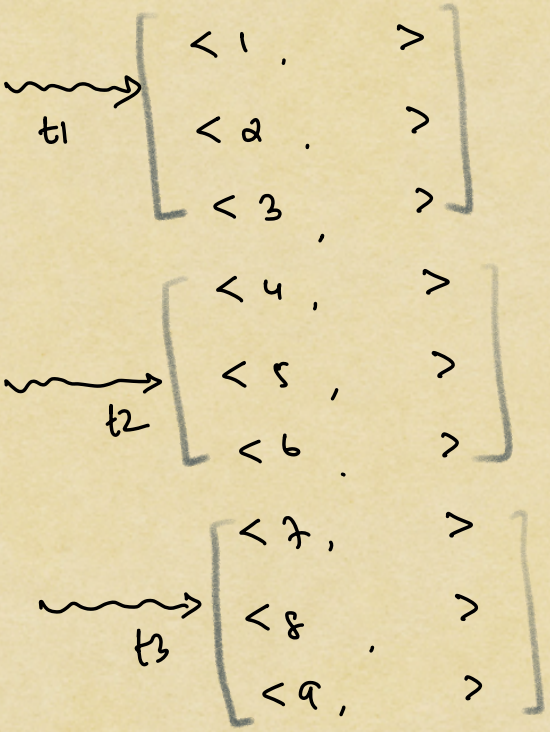
BBST => Red Black Tree

↓

search() | get() => O(logN)

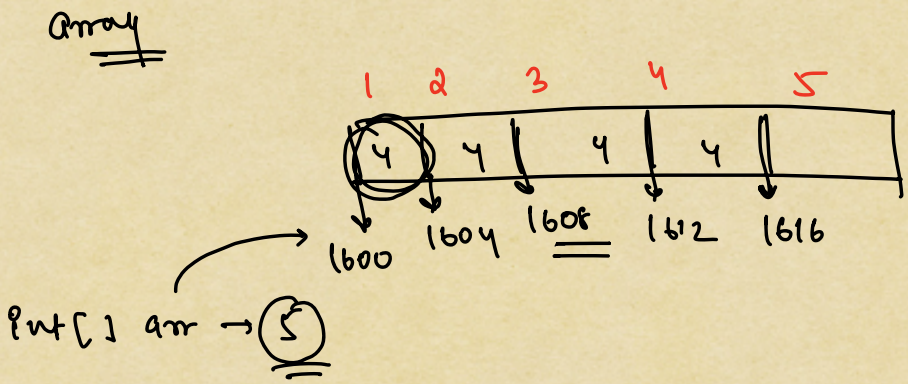
=> TreeSet → Internally => TreeMap

⇒ Concurrent Hashmap ⇒ thread safe hashmap



* Can multiple threads update Hashmap at the same time ⇒ Yes [NO

* Locks bucketwise



arr ⇒ 1600

arr ⇒ 3 ⇒ 1600 → 1608
4

Initial size + Size of datatype × (place − 1)

place → index

1st element → 1600 + 4 × (1 − 1)

⇒ 1600 + 4 × 0 ⇒ 1600

1600 + 4 × (0)

⇒ 1600

arr(i) ⇒ [arr(initial add.) + size of datatype × i]

arr(i) → 1600

size of datatype → 4

1600 + 4(i) ← address