Agenda : ① Concurrent Data Structures
- ⓐ Atomic Data types
- ⓑ Concurrent Data types

② Deadlocks

---

③ Collections

④ Generics ⇒ Brief

⑤ Summary

① Concurrent Data Structures

⇒ Adder - Subtractor had a sync. problem

⇒ reason for sync problems ⇒ Shared Data

Ques Why it was causing a problem?

the statement ⇒ `count += i` ⇒

causing the issue

① X ← Read count

② X = X + 1

③ count ← X

counter += x ⟹ not atomic and that 's why it was causing an issue.

Atomic ops ⟹ that gets executed in one go & for such type of ops we have atomic data type.

Atomic Data type ⟹ Provide way to do atomic ops. on their non-atomic variant.

x += y ⟹ for such cases we have Atomic Integers (int, float)

(AtomicInteger a)

String s

Atomic Integers : (1) Provides Atomic method to perform some common ops on an integer

(2) we dont need to take locks when

dealing with Atomic ops.

When its a multi-threaded env ⇒ Atomic - Integer

Single threaded env ⇒ int

Atomic Data types have an overhead as well, so use it with caution.

(2) Concurrent Data types ⇒ helps us perform better in a concurrent env.

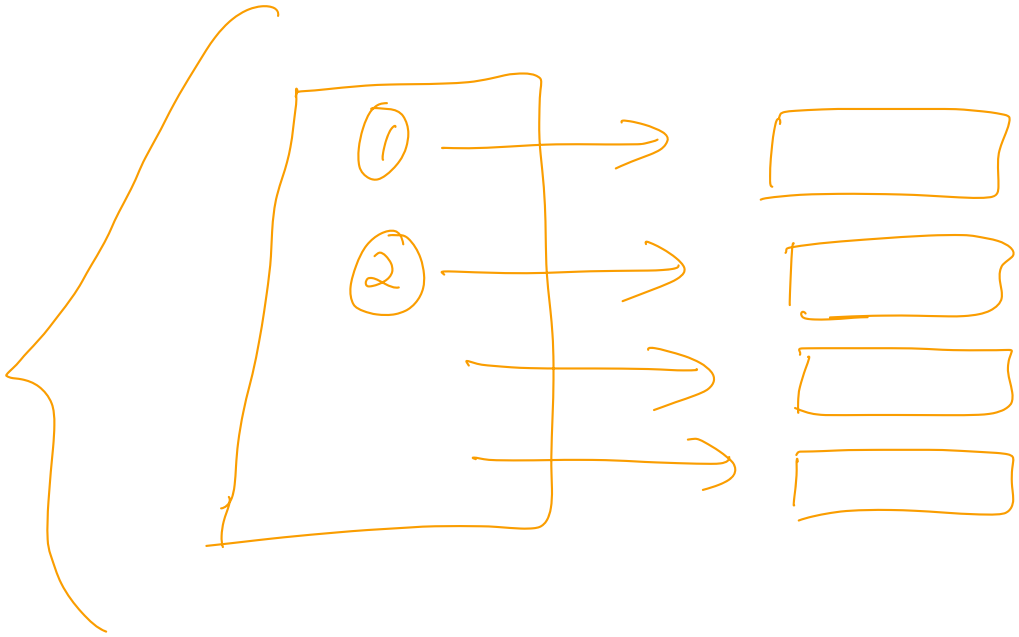
```
class Cache {
    Map<int, String> map;

    Void add(-,-) {
        map.add(—) ;
    }
}
```

Map ⇒ HashMap

HashMap is not thread safe, so you need to handle the sync. issues when

dealing with in a multi-threaded env.



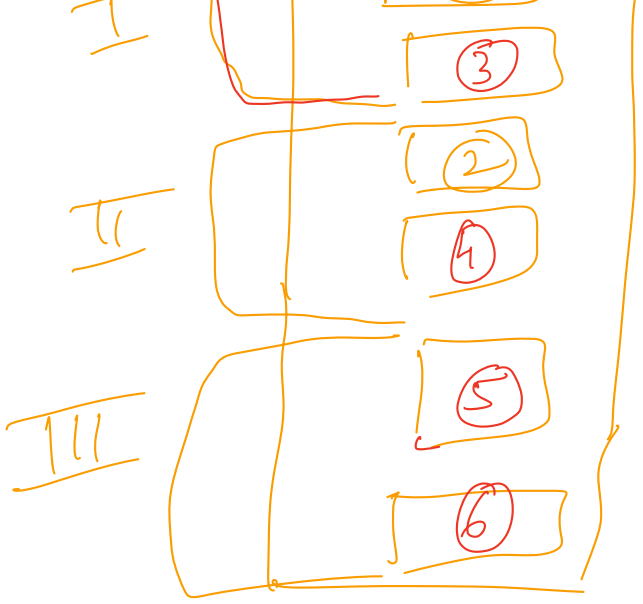
Is there a problem with taking a lock over the entire hashMap.?

T1	T2
① x = map.get(1)	② y = map.get(2)

for such type of situations, we have Concurrent Data Structures.

Concurrent HashMap

takes lock over the



bucket and not the entire map, thus giving us a faster performance.

T1 ⇒ map.get(1) ⇒ take a lock of I bucket

T2 ⇒ map.get(2) ⇒ take a lock of II bucket

so both T1 & T2 will run parallely.

(3) Deadlocks ⇒ Let us consider a scenario ⇒ Lock1, Lock2

T1	T2
(1) lock1.lock()	(2) lock2.lock()
(3) lock2.lock()	(4) lock1.lock()

Both the threads are waiting for a lock

which they want get, and they are going to wait indefinitely ⇒ Deadlock

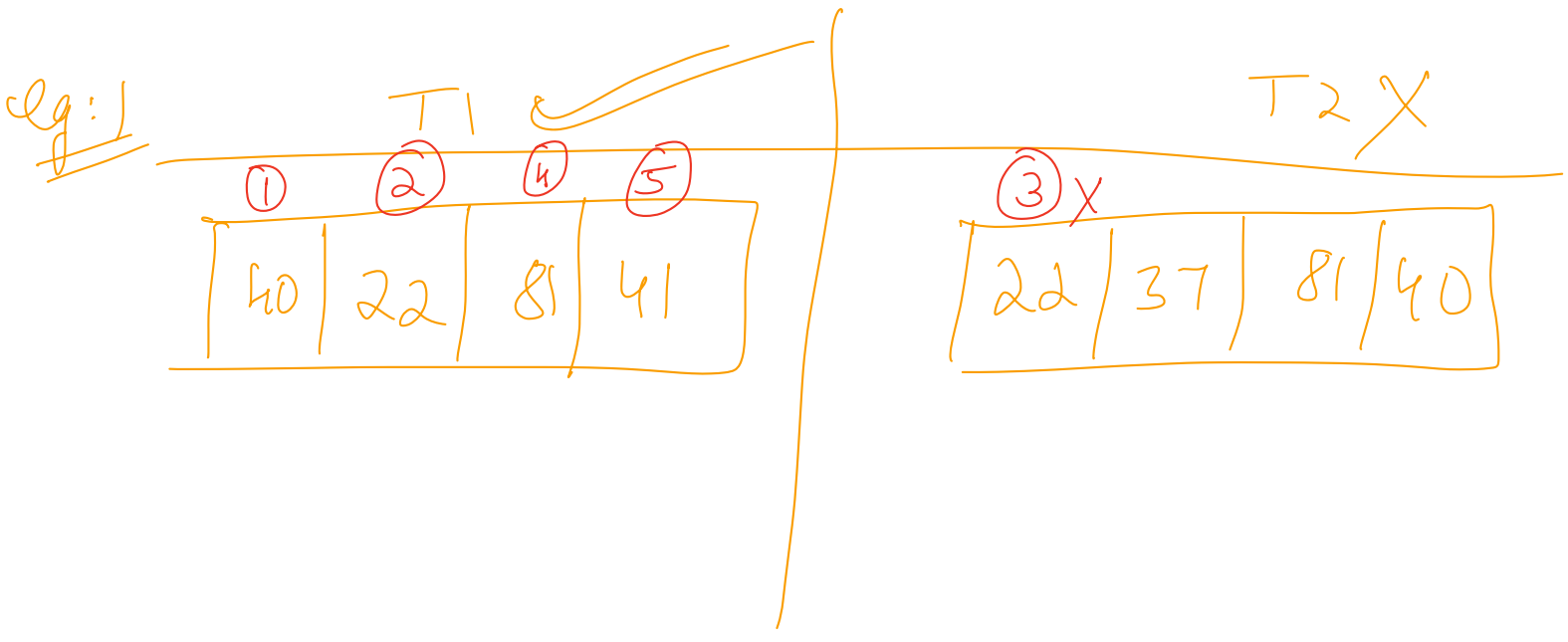
Solutions to a Deadlock

(1) Deadlock prevention ⇒ always try to avoid deadlocks.

(2) Deadlock cure / recovering ⇒ Kill the thread / Taking medicine

(3) Deadlock ignorance ⇒ most common soln. in OS deadlocks

(1) Deadlock prevention ⇒ Always try to code in such a way that you dont get any deadlocks.



eg2:

T1	T2
① ✓ ③ ✗	② ✓ ④ ✓ ⑤ ✓ ⑥ ✗
40 | 22 | 81 | 41	22 | 37 | 81 | 40

↳ would result in a deadlock.

⇒ Always try to take the lock in ascending order:

T1	T2
eg.1 ① ✓ ③ ✓ ④ ✓ ⑤ ✓ 22 | 40 | 41 | 81	② ✗ 22 | 37 | 40 | 81
eg2. ③ ✗ X X X 22 | 40 | 41 | 81	① ✓ ② ✓ ④ ✓ ⑤ ✓ 18 | 22 | 37 | 40 t2 ✓

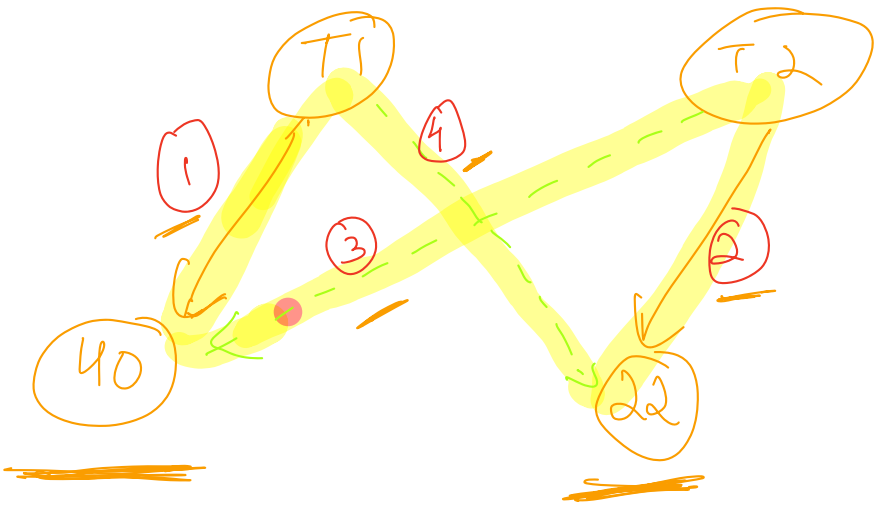
⇒ There is a mathematical proof, if you take

the locks in asc. order, there won't be a deadlock situation.

Deadlock Identification ⟹

(1) Wait for a timeout, ⟹ I am calling a function and then wait for 10s. If we don't get anything, it could be that there is a deadlock.

(2) Graph cyclic detection ⟹



If a graph forms a cycle, that means there is a deadlock.

(3) Ignorance ⟹ Ignore the deadlock, generally

used in OS. If there's a deadlock, system will hang, user will know that there is something wrong, user will restart the system, thus killing the threads.

③ Deadlock cure ⇒ {Kill one of the threads.}

Ques → How to take locks in production system?

① Identify the deadlock via a timeout
If the thread doesn't complete {progress in IOs (eg.)}, then kill the thread.

② Implement with try lock / caution.

try lock() { }

↳ It will immediately throw an error, if we are not able to get a lock

③ try lock (x)

↳ try for these

many records, then if we are not able to get, throw an error.

Summary ⇒
1. Semaphores ⇒ V. imp.
2. Atomic Data types
3. Deadlocks
4. Synchronization issues
   ↳ How to debug & resolve.

Class starts at 10:40 PM

⇒ Collections ⇒ framework

Collections ⟶
1. Set of classes
2. Set of Interfaces

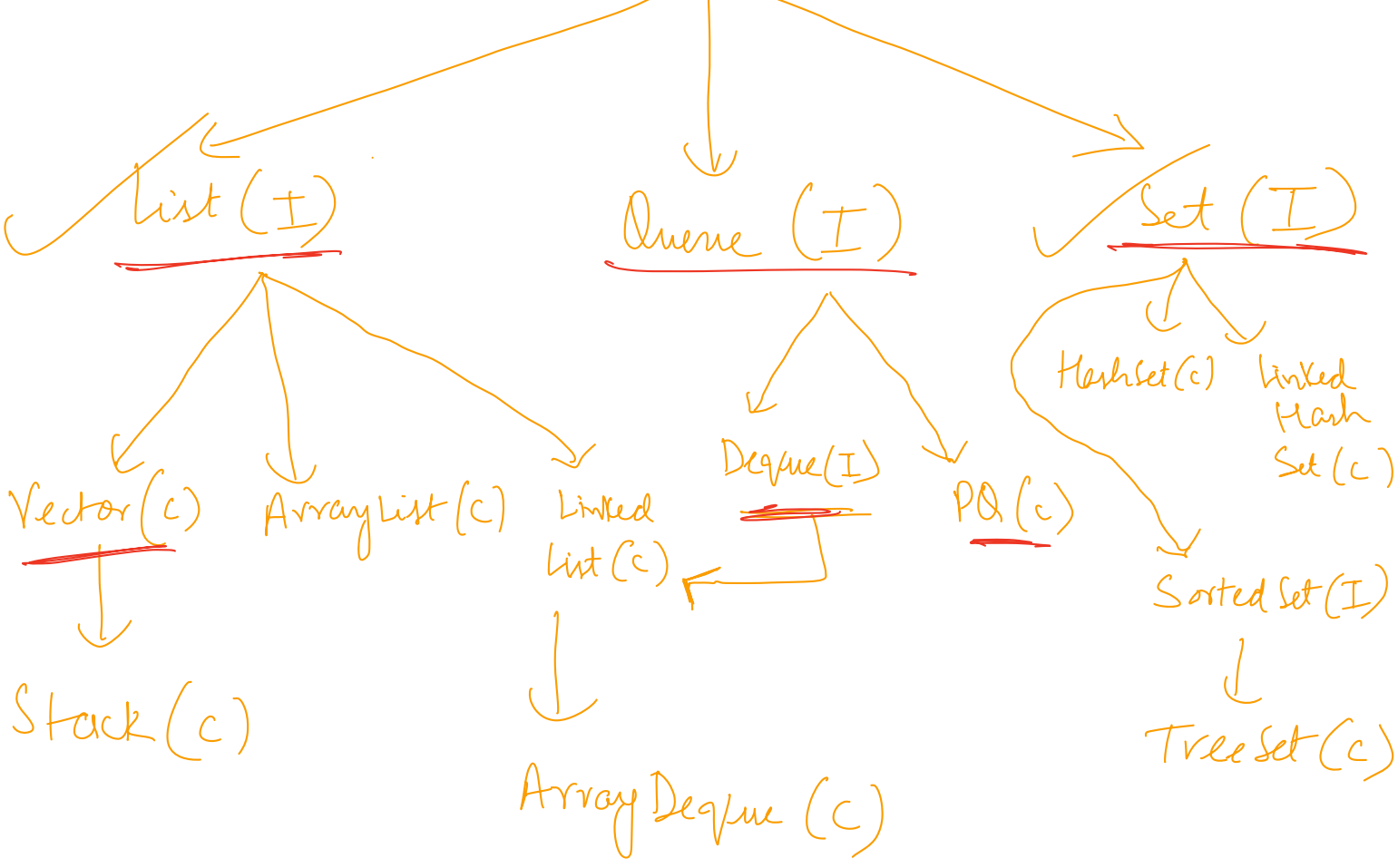
Data Structures ⇒ ArrayList, Tree, Stack, Map, Set

Hierarchy of Interfaces

↓

Collection (I)

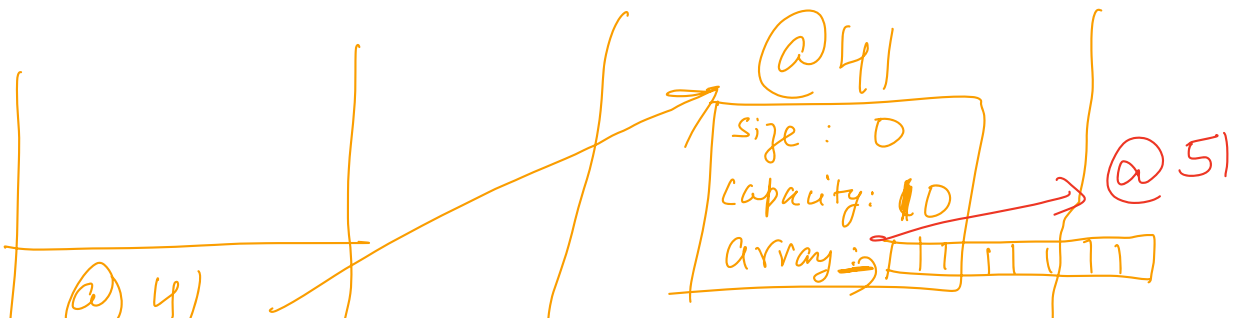
↑



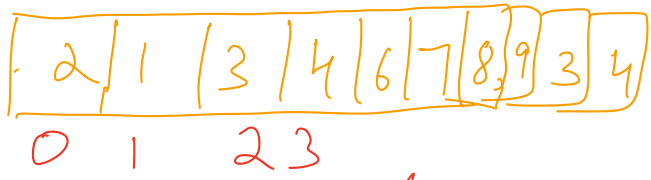
① List { Vector, AL, LL, Stack }

② ArrayList ⟹ Dynamic Array.
It can expand in size. Its a combination of ① Size ② Capacity.

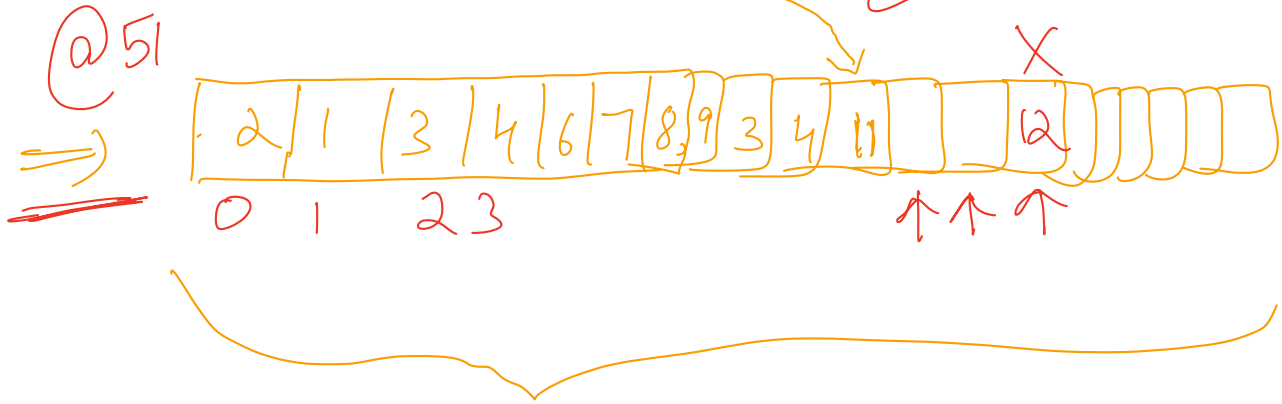
```
AL<> list = new AL<>();
```



(a) list.add( ) ⇒



list.add(11);



⇒ When size = capacity and you are trying to add new element, old array will get copied into a new array which will be of twice the initial capacity

⇒ we cannot skip a position while adding.

(b) list.set (↑ , p)

where what

(0 to curr.size - 1)

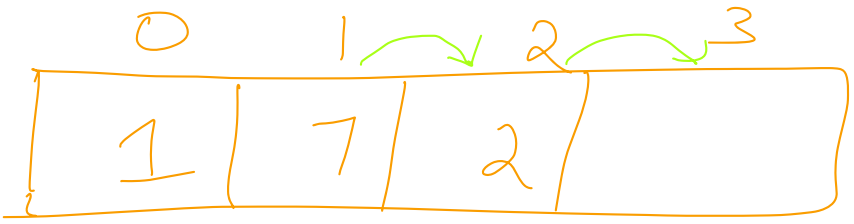
set is like updating.

(c) list.get(↑)
0 to size-1

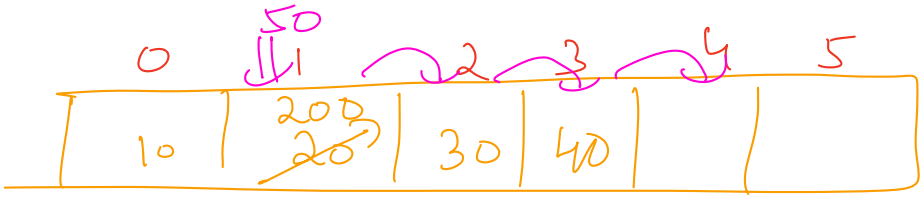
(d) list.add(↑ , ↑)
where what
↳ (0 to size)

size = 3
0 → 3

list.add(index 1, element to Add 6)



eg:



get()	set()	add()	add(,)
get(1)	set(1,200)	add(40)	add(1,50)
(20)			10 50 200 30 40

Time complexities ⇒ get() ⇒ O(1)

set() ⇒ O(1)

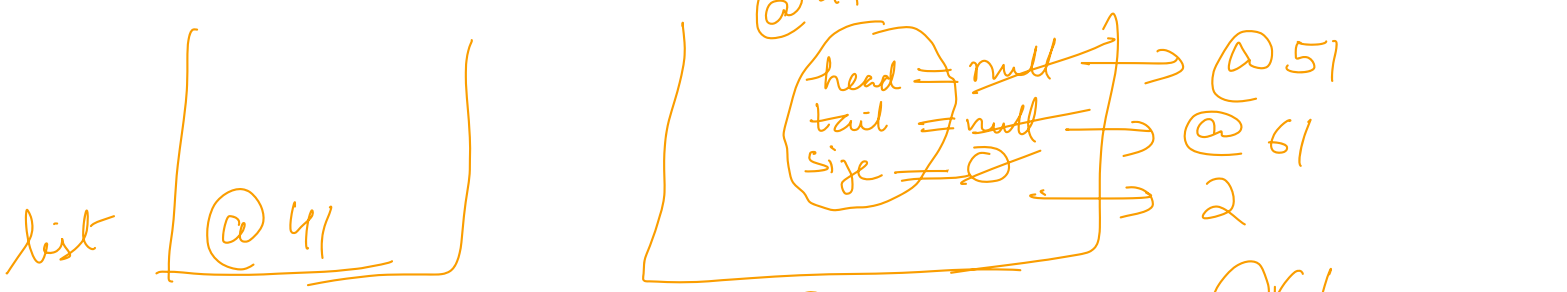
add() ⇒ D(1)

when size = capacity ⇒ O(n) ⇐

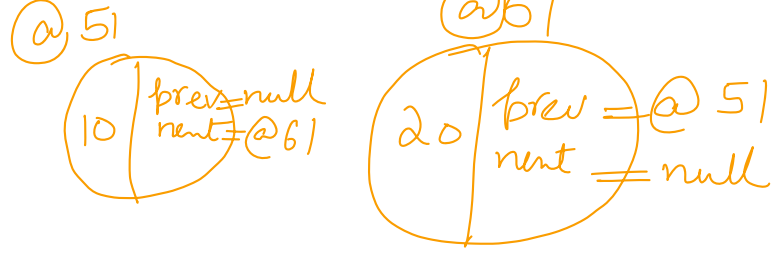
add(-,-) ⇒ O(n)

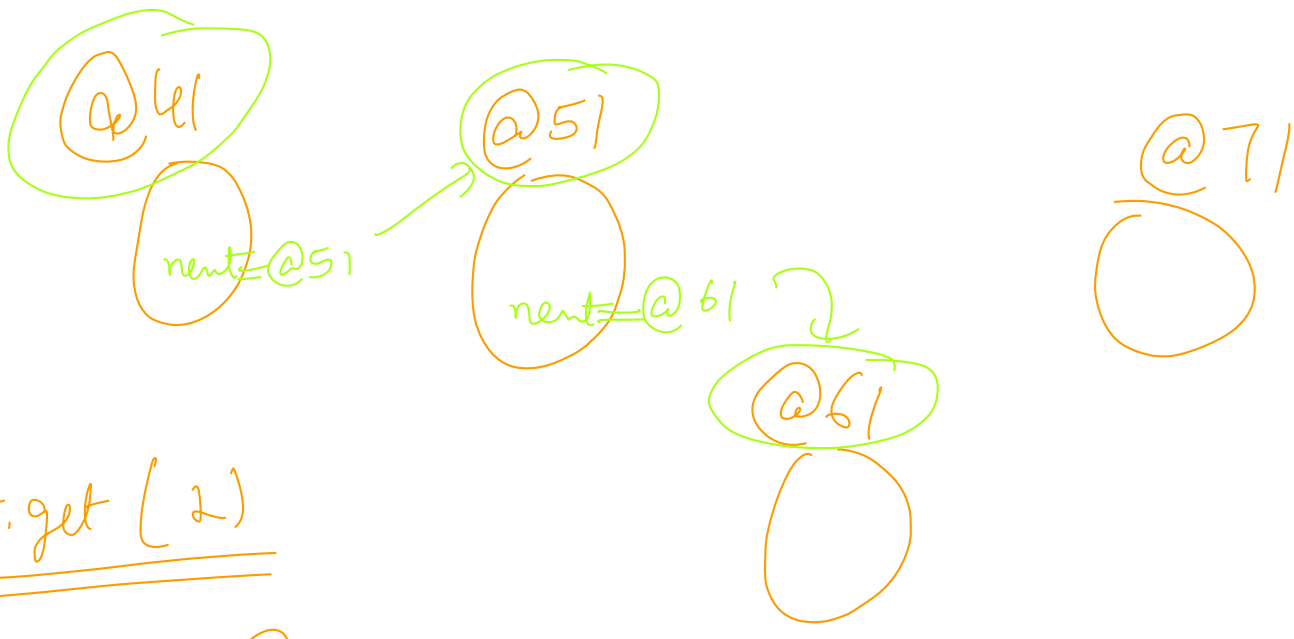
(b) Vector ⇒ Vector is very similar to ArrayList, only that it is thread safe.

(c) LinkedList : LL dist = new LL(); (LinkedList) ⇒ doubly ended linked list.



① list.add(10)
② list.add(20)





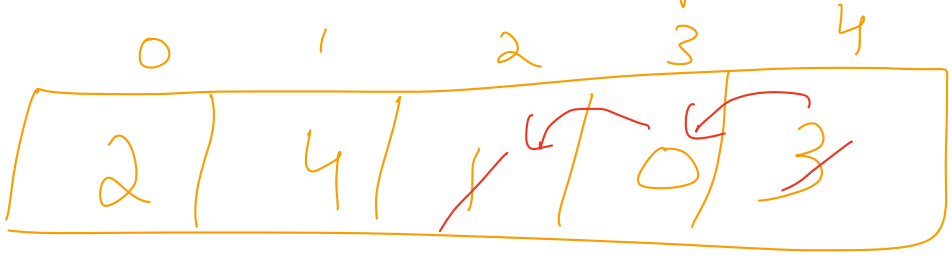
list.get(2)

head = @41

Time Complexity	AL	LL
get()	O(1)	O(n)
set()	O(1)	O(n)
add()	O(1)	O(n)
add(-,-)	O(n)	O(n)

remove (i) | O(n) | O(n)

AL.



↳ remove(2) ⇒ O(n)

(d) **Stack** : Stack derives from a Vector only and supports 3 additional behaviour | methods.

- ↳ push ⇒ Adding to top of stack
- ↳ pop ⇒ removing from top of stack
- ↳ peek ⇒ get from top of stack.

- ↳ push ⇒ list.add(size)
- ↳ pop ⇒ list.remove(size-1)
- ↳ peek ⇒ list.get(size-1)

(2) **Set** ⇒

HashSet (C)

LinkedHashSet (C)

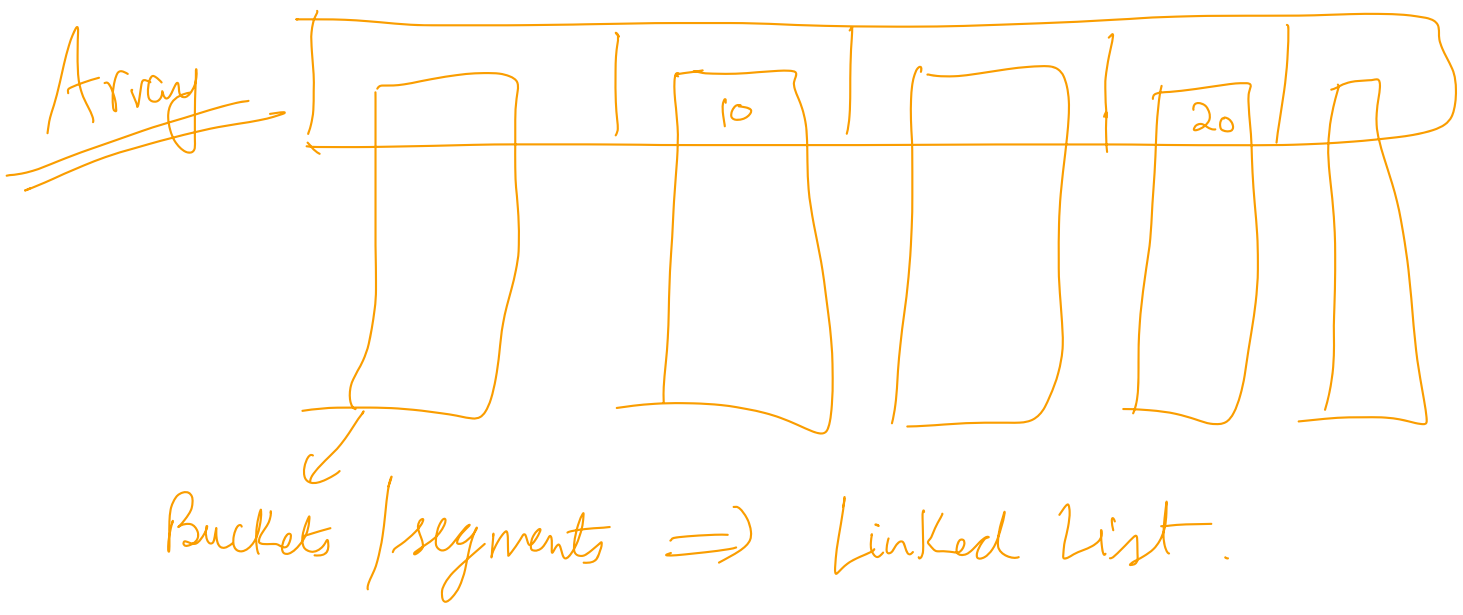
Sorted Set (I)
↓
TreeSet (C)

(a) HashSet ⇒ If we try to add duplicate data, it wont allow.

```
HashSet<> set = new HashSet<>
set.add(10);
set.add(20);
set.add(10);
```

Common function of Set ⇒ add
remove
contains.

Underlying data structure of HashSet



set.add(10);
set.add(20);

⇒ How does a bucket gets decided?

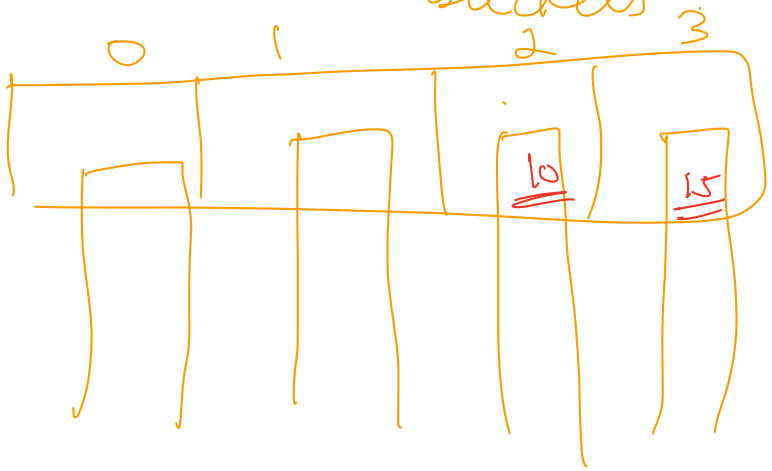
↪ HashCode ⟺ HashFunction.

⇒ for integers HashCode is equal to value itself.

⇒ H.C. + C.F. ⇒ compression function

↓

limits the value of hash function to the no. of buckets



set.add(10) ⇒

H.C.(10) ⇒ 10

C.F.(10) ⇒ abs(10) ⇒ 10

⇒ abs(10) % no. of buckets

$10 \% 4 \Rightarrow 2$

set.add(15)

H.C.(15) ⇒ 15

C.F.(15) = abs(15) ⇒ 15

= $15 \% 4 = 3$

```
private int hashfunction(Element el){
    (1) int hashCode = el.hashCode();
    (2) int abs = abs(hashCode);
    (3) return abs % no.Ofbuckets
}
```

add() ⇒ (1) It will find the bucket where it has to insert the value

(2) It will loop through the linked list and add it to the end.

⇒ equals() ⇒ this will ensure that no duplicate values get inserted

into a hashset. While looping through the linked list, it will check using equals().

Pseudo function of add.

```
boolean add ( K KeyToInsert ) {
        int bucketNo= hashfunction (Key)
    for ( Node node : buckets [bucketNo] ){
        if ( node.Key.equals (Key)) {
            found = true;
            break;
    }}

    if ( found == false) {
        Node node = New Node (Key);
        buckets [bucketNo].add (node);
    }
```

KeyToInsert

similar code for contains()

- TreeSet
- Exception Handling