collections

⇒ `List<Integer> list = new ArrayList<>();`

```
list.add(1); list.add(2);
for(int i = 0; i < list.size(); ++i) {
        sout(list.get(i));
        list.add(i);
}
```

`list.size()` → This will increase the size every time

`list.add(i);` → This will increase the size every time

Output:-

↳ Infinite loop

✱ Fail Fast iterators and Fail Safe iterators

```
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
        Integer num = it.next();
        sout(num);
}
```

list → [1, 2, 3, 4]

Out →
1
2
3
4

✱ Fail Fast iterators

```
while(it.hasNext()) ; → iterating
        int num = it.next();
        list.add(2); → modification to collection
}
```

⇒ Here in case of fail fast Concurrent Modification exception is thrown.

⇒ This happens because iterator is directly working on collection without creating a copy

⇒ ArrayList, LinkedList, HashMap, HashSet are fail-fast iterators

✱ Fail Safe

```
while (it.hasNext()); → iterating
    int num = it.next();
    list.add(2); → modification to collection
                   (Exception is not thrown)
}
```

i) Fail safe iterators acts on clone (snapshot) of that collection

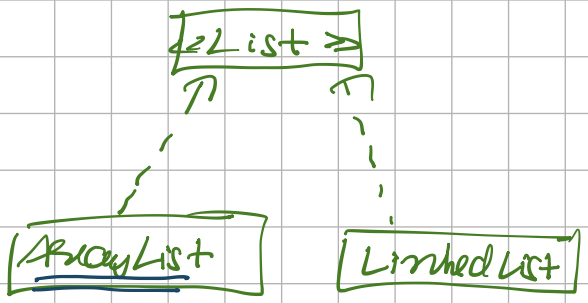
ii) Eg's → Concurrent HashMap, CopyOnWriteArrayList.

```
CopyOnWriteArrayList list; → {1,2,3}

while (it.hasNext())
{
    int val = it.next();
    sout(val);
    list.add(5);
}
```

→ 1, 2, 3

sout(list) → 1 2 3 5 5 5

✱

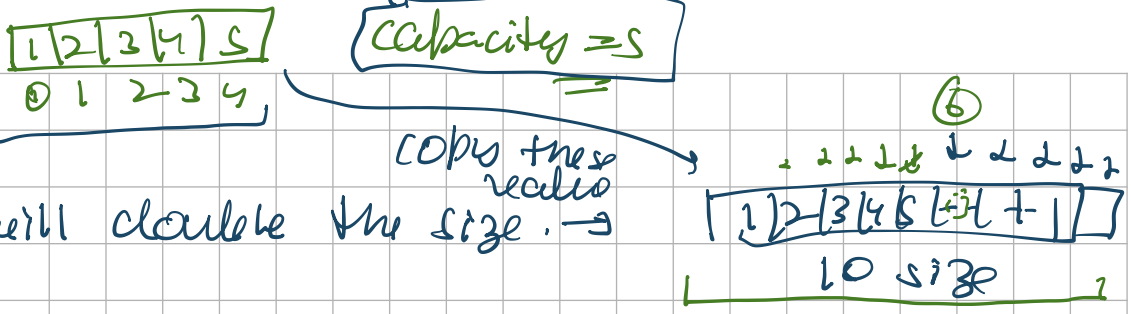



```
List<Integer> list = new ArrayList<>();
```

```
list.add(1);
list.add(2);
```

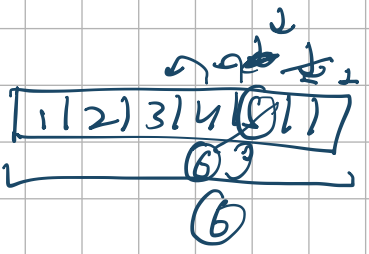
Time Complexity to add → O(1)

Amortized / Average



Modifying and adding

↳ TC: O(1)



Removes are costly

↳ TC: O(n)

Access → O(1)

★ Linked List (Doubly Linked List)

↳ Addition ↳ head / tail

→ Addition in start and end

TC: O(1)

⇒ Addition in middle

TC: O(n)

⇒ Accessing element

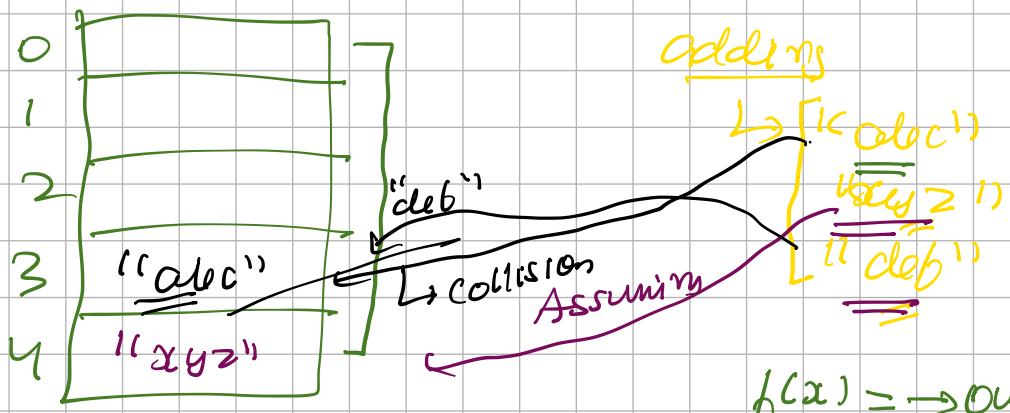
TC: O(n) h/T is exception

★ HashCode and equals:

```
Set<String> set = new HashSet<>();
set.add("abc");
set.add("abc");
sout(set.size()); → Size: 1
```

```
class Student {
    private int age;
    private int marks;
    private int name;
}
```

```
Set<Student> set = new HashSet();
set.add(new Student(--));
set.add(new Student(---));
```



capacity → 5
↳ Assuming string set

f(x) → output
↳ output ↑ capacity

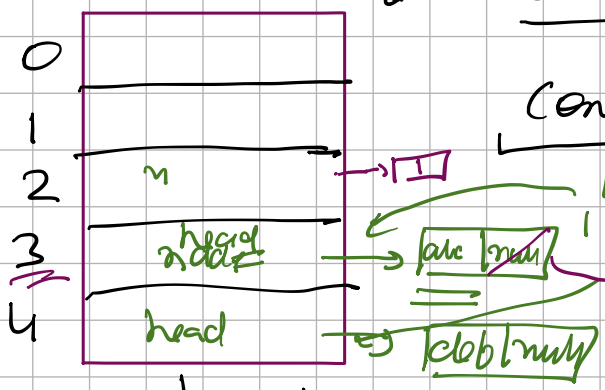
a b c d e ... z

3 2 1
abc →

$2 \times (26)^1 + 1 \times (26)^2 + 0 \times (26)^3 = 49 + 14$
$= 63 + 5$
$= 68$

Contains<String>

Contains<Node>



↳ capacity = 5

```
- public class Node {
-     String data;
-     Node next;
- }
```

i) First Part
↳ To find where our string will go (which index)

(ii) Second Part :-
↳ To verify the value.

set.find("abc")
↳ hashValue → iterate on LinkedList and verify the equal to the node value

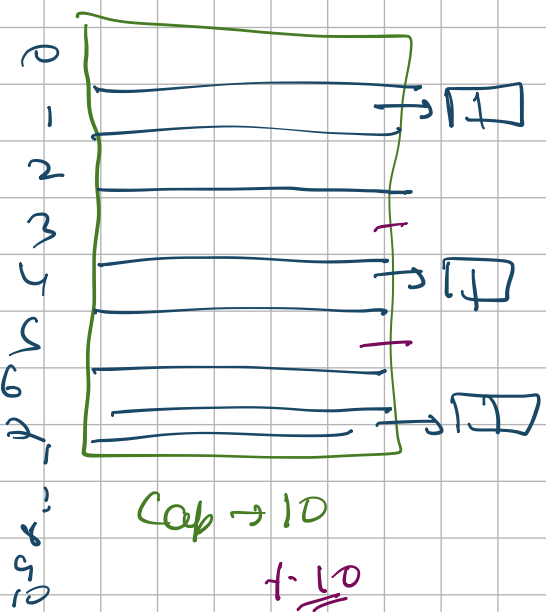
equals

* Object class provides us with two methods
  - → equals(Object obj);
  - → hashCode();

```
Student { name, age, rollNo;

int hashCode() {
    return Objects.hash(name);
}

bool equals(Object obj) {
    if(this == obj) return true;
    return Objects.equals(obj.name, this.name);
}
}
```

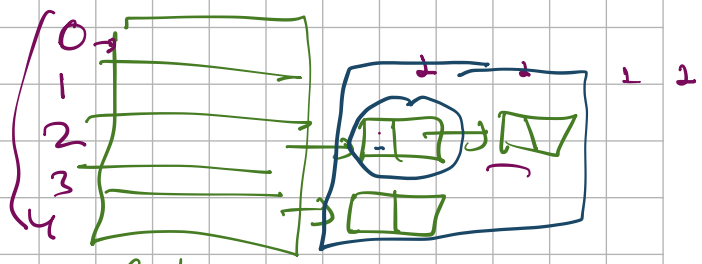
Load Factor → $\frac{\text{noOfElements}}{\text{capacity}}$ → $\frac{\text{totalNoOfElements}}{\text{capacity}}$

loadFactor ≥ 0.75 → Double the capacity of bucket
↳ Rehashing.



Cap → 10

1.10



Cap → 5

noOfElements → 6

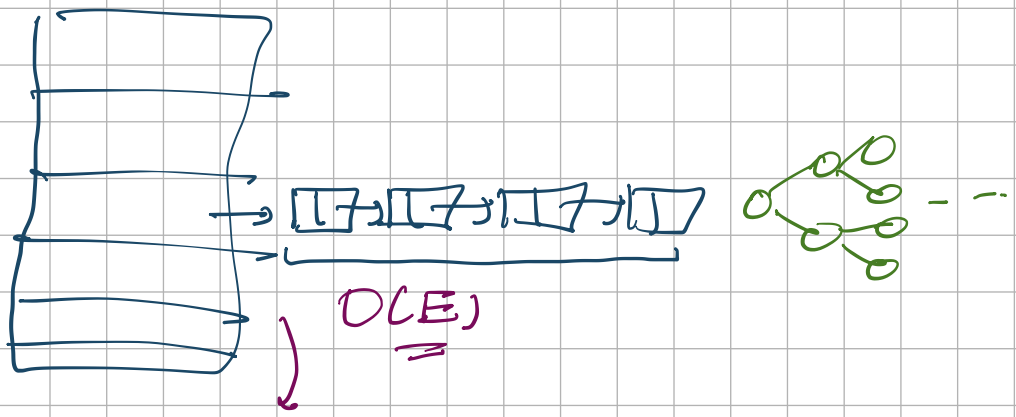
Cal

1.5

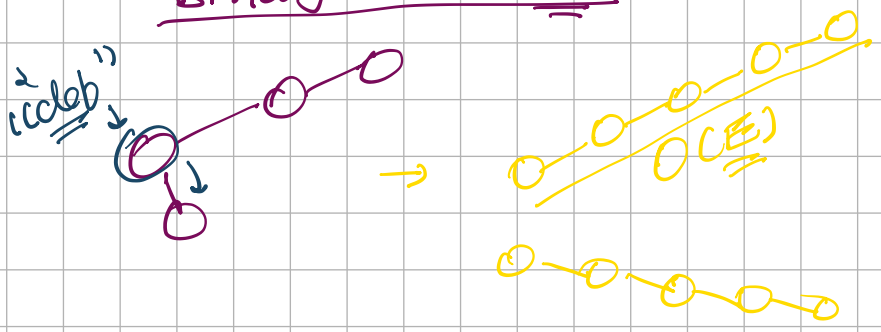
f(x) = Objects.hash(string)

TC: O(1) (Amortized)

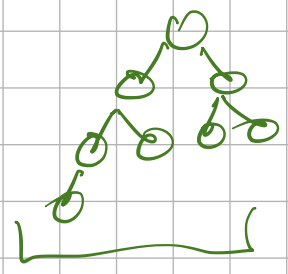
```
Class Node {
    String data;
    String value;
    Node next;
}
```



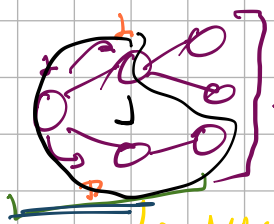
Binary Search Tree



BBST
Balanced Binary Search Tree
↳ AVL trees
↳ Red Black tree



$\log_2(E)$



E element
→ How it decides sort criterias
↳ Wrapper here
= Student
↳ Wealth, age, rollNo;

↳ Comparable and Comparator

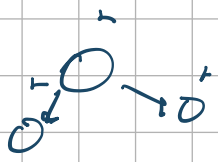
BBST

alec → generate some hash 7
def → 10
xy → 4
hashCode is same
def → 10

☆ TreeSet :

↳ RB tree

Comparators, Comparable



i) Comparable vs Comparator.

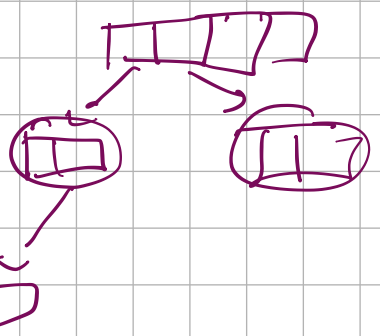
```
class Student implements Comparable<Student> {

    public int compareTo(Student student) {

        ret Integer.compare(this.rollNo,
                            student.rollNo);
    }
}

Collections.sort(student); → Ascending

ret Integer.compare(student.rollNo, this.rollNo);
```

↳ descending

[-1, 0, 1]

⇒ Collection.sort(List); → which sort algo?

↳ Timsort (Hybrid)



→ Insertion] Hybrid sort

this,

```
bool compare(Integer a, Integer b) {
    if (a <= b) ret true;
    ret false;
```

3

TreeSet<Integer> set = new TreeSet<>();

↳ Red Black tree impl.

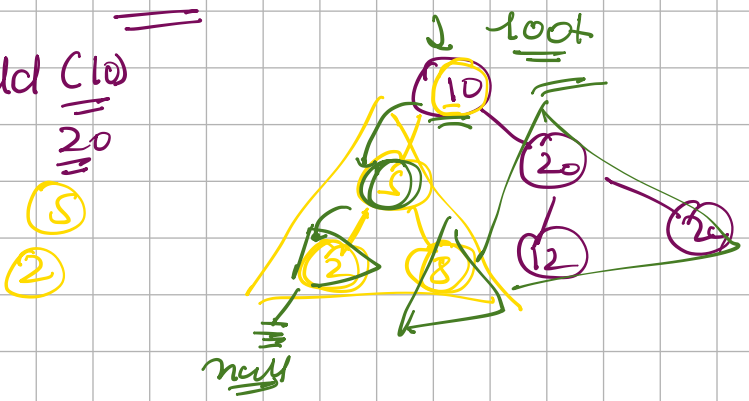
BBST

set.add(10)

20

5

2



bind(1);

TreSet<Student> set = new TreeSet<>();

Student with rollNo → 20

> Student with rollNo → 15

☆ Comparator : (Interface)

↳ compare (Student s1, Student s2);

class CustomComparator implement Comparator<Student>{

@Override

public int compare(Student s1, Student s2){

set Integer.compare(s1.grade, s2.grade);

}

}

Collections.sort(List); → Sorted based on roll No

Collection.sort(List, new CustomComparator<>());

↳ Sort based on grades

Student

TreeSet<> treeSet = new TreeSet<>();

TreeSet<Student> treeSet = new TreeSet<>

(new MyCustomComparator<>());

grades

new TreeSet<>((a, b) -> Integer.compare(a.grades, b.grades));

Functional Interface

<Collection>

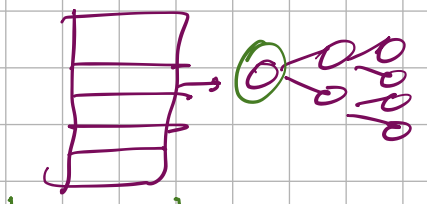
extends

<Set>

implements

TreeSet

HashSet



Threshold

HashSet

class Node {

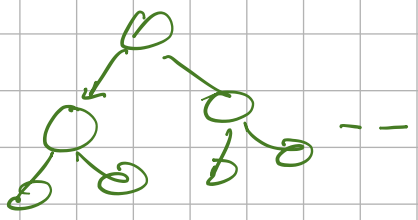
String data;

String value;

Node next;

}

TreeSet → Red Black Tree



★ C++

↳ MultiSet

↳ can have duplicates

Multiple elements with same value.

★ Question:

↳ We have to find whether that element exists in string or not when string contains lower case [a b c d - - z]]

"abccdee x"

Q queries → TC: O(n Q)

Store in map →] SC: O(26)
always →] TC: O(Q + n)

Bit manipulation

a = 0 → 31 bit

a → 0
z → 25

31 0 — — — — 0

1

1

a = a | (1 << (ch - 'a'));

a = 0 "a b c" 32 bits

a → 0 4 Bytes
z → 25

0 — — — 0

1

1

0 25

a b c — — — z

26 size 26 bytes

(abcd

↳ int a = 0; Query

↳ 4 Bytes

↳ 32 bits

1st → c

2nd →

31 — — — 0

a → 31 0 0 0 0 — — 0th

bit vector

★ EnumSet (DataStructures)

↳ ENUMS → Constants

↓

Fixed values

MONDAY, TUE, --- SUN

0, 1, 6

enum Map (MONDAY, SUN)
Set

↳ 1th

0000 100 1 0th

↓

Store

0 0 0 0 0 1 → (1)

- - - 1 1 → 3

1 0 → (2)

Linked HashMaps. =

Concurrent HashMaps. = → Thread Safety

HashTable, Vector ↳ Thread Safety

CopyOnWriteArrayList =

binarySearch() Eg.

Queue, Stack, Deque. → Find max in window of size k.

Collection

↑

Queue

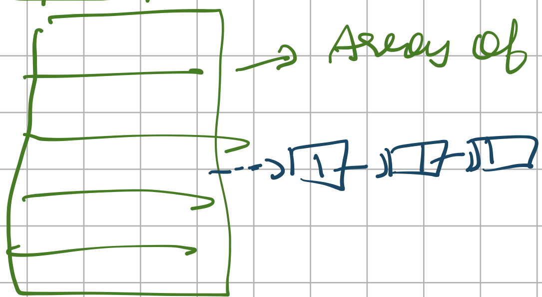
↑ ↑

Deque PQ

⋮

LinkedList
ArrayDeque.

★ Hashmap impl:

↳ 

→ Array of type what

class Node<T> {
→ private T data;
→ private Node<T> next;
}

→ Array

class Hashmap {

int[] arr = new int[10];

T[] arr = new T[10].

→ size of int

int → 4 bytes

how many bytes

Object[] arr = new Object[10];

size of an object → points

8 bytes

Create an array of Object

arr[0] = "abc";

Object

Generic type

key, value

↳ 123

// "hash"

// temp

→ hash function should

T is a generic

Objects.hash

Only String key, Generic values

★ class Hashmap<V> {

private Node[] arr;

private int capacity;

Hashmap() {

capacity = 5;

arr = new Node<V>[capacity];

}

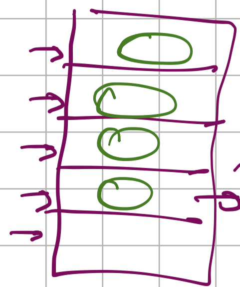
class Node<V> {

String key;

V data;

Node next;

}



node

void put(String key, V data) {

idx → hash(key) % capacity

if (arr[idx] == null) {

arr[idx] = new Node(key, data, null);

}

else {

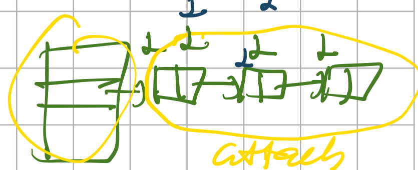
reach that key

↳ ↳ equals → cut / update node

temp → end of L.List

temp → next = new Node(key, data, null)

not found



// Else update data

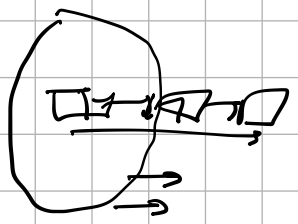
temp.setData(data);

}

loadFactor = noOfElements / capacity >= 0.75]

↳ Double the capacity

↳ Rehashing]

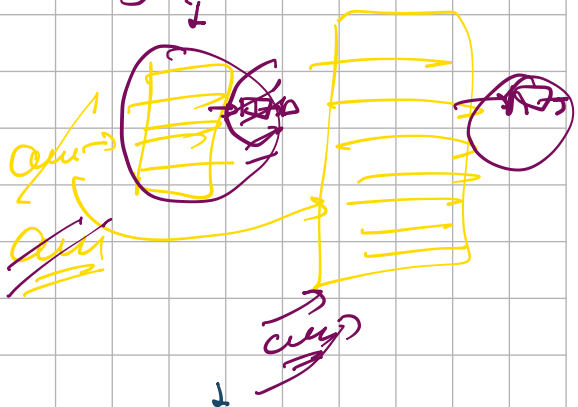


Node<String>[] tempArr = new Node<>[2*cap];

↳ Rehashing

↳ add to this array

this.arr = tempArr;



}

One operation will be bigger

void delete(String key) {

↳ hashcode

↳ iterate and equals

↳ remove the node

}

[noOfElement / capacity → 0.25

↓

we won't make it smaller

void putOfNotPresent(String key, T data) {

= Addition of key/data if not present

}