COLLECTIONS

Q. Why Collections ?

Suppose for an application less nubers of inputs are required, then we can assign them with variables.

```
int x = 10;
int y = 20;
int z = 30;
```

But, when the input count is 1000's or 10000's then taking that much of variables will not be a good practise and also it will increase the code readeability.

So, to reduce the effort "Arrays" concept is introduced.

* Array -

It is an indexed collection of fixed number of homogenous data elements.

Now with Array

```
int[] arr = new int[1000];
Student s = new Student[50000];
```

=> Advantages

1) We can represent multiple values using a single variable
2) Readeability can be improved.

=> Limitations

1) Fixed in size : Once Array is created, size cannot be increased or decreased.
2) Homogenous : Only singular data type is supported.

Solution for Homogenous

We can take Object as a datatype

```
Object obj = new Object[50];
obj[0] = new Student();
obj[1] = new Customer();
```

3) No Underlying data structure:

↳ ready method support is not there
↳ we need to write logic for everything like (search, add, sort, delete etc)
↳ Increases the complexity of code.

* Collections-

To overcome the problems of Array, we should go for collections.

=> Advantages-

1) Growable in Nature: Size is not fixed. It can be changed as per the requirement.

2) Homogenous & Hetrogenous: It can hold same as well as different objects or datatypes.

3) Underlying DataStructure: Every class is implemented based on some standard DataStructure.

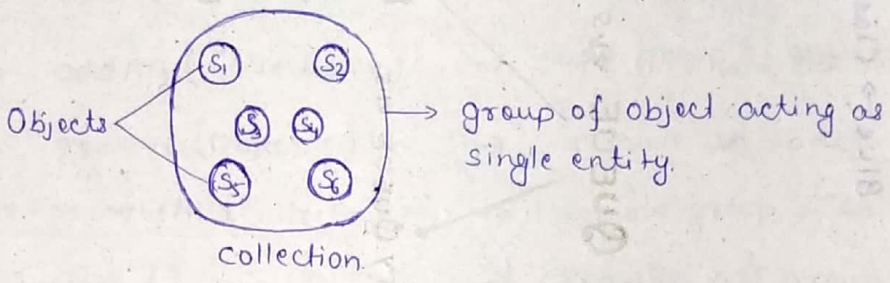
Arrays	Collections
1) Fixed in size.	1) Growable in size.
2) Recommendation: w.r.t performance => Yes w.r.t memory => No	2) Recommendation: w.r.t performance => No w.r.t memory => Yes
3) Homogenous datatype or Object	3) Homogenous and Hetrogenous datatype or object.
4) No underlying DataStructure	4) Based on some standard DataStructure
5) No predefined methods	5) Pre-defined methods for each classes
6) It can hold premitive and Object both.	6) It only holds objects.

Q- Which one is Recommended?

If we know the size then better to go with Arrays Concept because collection is growable in nature. i.e if size increases, data increases and hence cost in-efficient. and less performance.

* Collection-

It is a group of individual objects acting as a single entity.



NOTE-

-> It is an Interface

-> There is no concrete class which implements collection Interface directly.

* Collections-

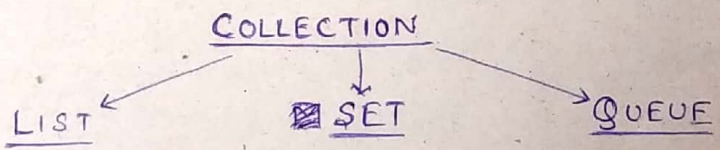
It is a class available in "java.util" package which defines several methods like sorting, searching etc. for collection objects

NOTE-

-> It is a class

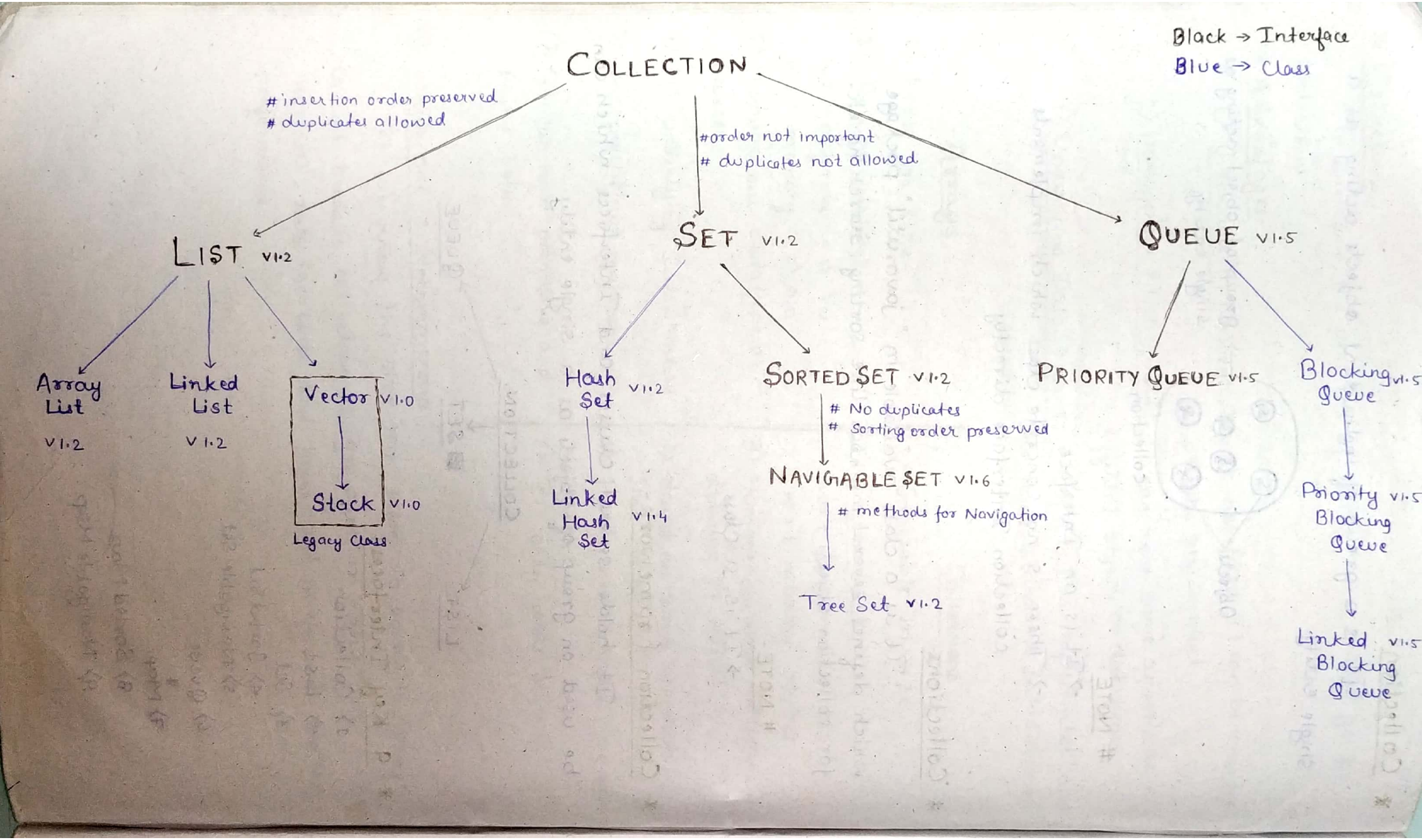
* Collection Framework-

It holds several classes and Interfaces which can be used on group of objects as a single entity.



* 9 Key Interfaces-

1) Collection
2) List
3) Set
4) Sorted Set
5) Navigable Set
6) Queue
7) Map
8) Sorted Map
9) Navigable Map.


Black → Interface
Blue → Class
COLLECTION
insertion order preserved
duplicates allowed
#order not important
duplicates not allowed
LIST V1.2
SET V1.2
QUEUE V1.5
Array List V1.2
Linked List V1.2
Vector V1.0
Stack V1.0
Legacy Class
Hash Set V1.2
Linked Hash Set V1.4
SORTED SET V1.2
No duplicates
Sorting order preserved
NAVIGABLE SET V1.6
methods for Navigation
Tree Set V1.2
PRIORITY QUEUE V1.5
Blocking Queue V1.5
Priority Blocking Queue V1.5
Linked Blocking Queue V1.5

* Methods under Collection Interface

=> boolean add (Object c) → add an object

=> boolean addAll (Collection c) → add group of object

=> boolean remove (Object c) → remove an object

=> boolean removeAll (Collection c) → remove group of object

=> void clear () → remove all group

=> boolean retainAll (Collection c) → remove all except c

=> boolean isEmpty () → Collection empty or not

=> int size () → if not empty then size

=> boolean contains (Object c) → particular object there or not

=> boolean containsAll (Collection c) → group of object there or not

NOTE -

=> In general collection Interface is considered as root Interface of Collection Framework.

=> Collection Framework does not contain any method to retrive objects.

* LIST -

↳ It is the child Interface of Collection.

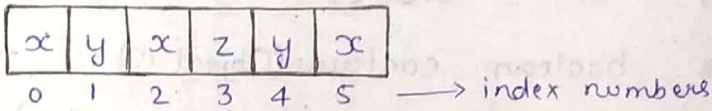
↳ If we want to represent group of individuals as a single entity where

a) Duplicates are allowed.

b) Insertion order is preserved.

↳ We can differenciate duplicates and insertion order with the help of index.

↳ Methods available in collection will be available in List.



=> arr[0] will have an object and the object will hold the element "x".

* Methods under List Interface -

=> boolean add (Object c)

=> void add (int index, Object c)

=> boolean addAll (int index, Collection c)

=> Object get (int index)

=> Object remove (int index)

=> Object set (int index, Object new)

=> int indexOf (Object c)

=> int lastIndexOf (Object c)

=> ListIterator listIterator ()

1) Array List -

↳ It is resizable array

↳ Duplicates are allowed

↳ Insertion order maintained

↳ Hetrogenous in nature

↳ null insertion is possible

NOTE -

Most of the methods are inherited from the List Interface like add, remove, indexOf, etc.

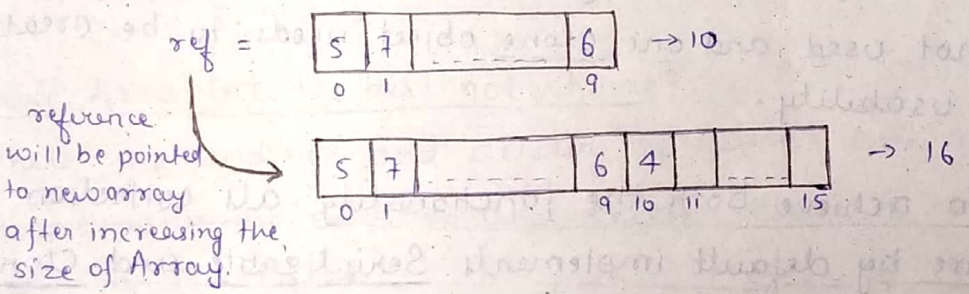
=> Constructing ArrayList -

ⓐ ArrayList al = new ArrayList();

↳ It will create an empty array list object with default initial capacity "10".

↳ When array size is full and one new object needs to be added then size of the array will increase by below formula.

$$newCapacity = \{currentCapacity \times \frac{3}{2}\} + 1$$



↳ Performance decreases

ⓑ ArrayList al = new ArrayList(int initialCapacity);

↳ This helps in performance improvement by defining the array size as per the requirement.

ⓒ ArrayList al = new ArrayList(Collection c);

↳ This constructor is meant for interconversion between Collection objects

LinkedList → ArrayList

Vector → ArrayList

ex:

```
ArrayList al = new ArrayList();
        al.add("A");
        al.add(10);
        al.add("A");
        al.add(null);
        SOP(al);          -> [A, 10, A, null]
        al.remove(2);     ->
        SOP(al);          -> [A, 10, null]
        al.add(2,"M");
        al.add("N");
        SOP(al);          -> [A, 10, M, null, N]
```

* NOTE-

To hold and transfer object from one place to another we can use collection but to transfer object from one place to another the object must be serialized.

After transfering the object, the received object is not used and one clone object needs to be created for usability.

=> To achieve both the functionality, all collection classes are by default implements Serializable and Cloneable

* Random Access -

It is a marker Interface which contains no method and is available in "java.util"

With the help of Random Access, one can access the elements at a very high speed. hence best for retrival.

ArrayList and Vector implements

RANDOM ACCESS

Array List	Vector
1) Non Synchronized	1) Synchronized
2) Not- Thread Safe	2) Thread Safe
3) Performance is high	3) Performance is low
4) Non- Legacy class (v1.2)	4) Legacy class (v1.0)

Q. How to get synchronized version of ArrayList Object?

```
ArrayList al = new ArrayList();          → not synchronized
List l = Collections.synchronizedList(al); → synchronized
```

NOTE-

With similar methods in Collections class, Set and Map can also be synchronized.

Q. Where ArrayList is best choice?

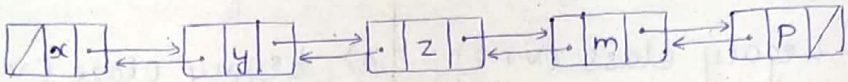
Where the frequent operation is retrival of object/elements from an object.

Q- Where Arraylist is best not choice?

Where the addition and deleting operations are required because with the addition and delete, elements has also needs to be shifted.

2) Linked List -

To overcome the drawbacks of ArrayList (i.e insertion and deletion), LinkedList is introduced. It works on node based structure.



When ever new insertion needs to be done, only pointer has to be changed, no shifting is required.

- ↳ Underlying data structure is double linked list
- ↳ Insertion order is preserved
- ↳ Duplicates are allowed
- ↳ Hetrogeneous in nature
- ↳ null insertion is possible
- ↳ Implements Serializable and Cloneable
- ↳ Do not implements Random Access

* Methods under LinkedList -

- ⇒ void addFirst (Object c);
- ⇒ void addLast (Object c);
- ⇒ Object getFirst ();
- ⇒ Object getLast ();
- ⇒ Object removeFirst ();
- ⇒ Object removeLast ();

these 6 methods can be used to do implementation with Queue & Stack

NOTE -

Most of the methods are inherited from the List Interface like, add, removee etc.

=> Constructor in LinkedList -

(a) LinkedList ll = new LinkedList();

It will create empty LinkedList Object

(b) LinkedList ll = new LinkedList(Collection c);

It will create equivalent LinkedList object for given collection.

ex-

```
LinkedList ll = new LinkedList();
ll.add("hello");              -> [hello]
ll.add(30);                   -> [hello, 30]
ll.add(null);                 -> [hello, 30, null]
ll.add("hello");              -> [hello, 30, null, hello]
ll.set(0,"software");         -> [software, 30, null, hello]
ll.add(0,"venky");            -> [venky, software, 30, null, hello]
ll.removeLast();              -> [venky, software, 30, null]
SOP(ll);                      -> [venky, software, 30, null]
```

Q. Where LinkedList is best choice?

Frequent operation like insertion, deletion in middle

Q. Where LinkedList is worse choice?

Frequent operation is retrival operation. and it always searches from the first node.

ArrayList	LinkedList
1) Best for retrival operation	1) Worse choice for retrival operation
2) Worse choice if frequent operation is insert/delete	2) Best choice if frequent operation is insert/delete
3) Underlying data structure is Growable Array	3) Underlying data structure is double linked list
4) It implements RandomAccess	4) It donot implements Random Access

3) Vector-

↳ It is resizable Array
↳ Duplicates are allowed
↳ Insertion order is preserved
↳ Hetrogenous in nature
↳ null insertion possible
↳ Implements Serializable and Cloneable
↳ Implements Random Access
↳ Most of the methods are Synchronized hence Thread Safe
↳ Best if frequent operation is retrival

* Methods under Vector-

=> For adding Object

-> add (Object c) -> From Collection
-> add (int index, Object c) -> From List
-> addElement (Object c) -> From Vector

=> For removing Object-

-> remove (Object c) -> From Collection
-> remove (int index) -> From List
-> removeElement (Object c) -> From Vector
-> removeElementAt (int index) -> From Vector
-> removeAllElement ( ) -> From Vector
-> clear() -> From Collection

=> For retriving elements

-> Object get (int index) -> From Collection
-> Object get (int index) -> From Collection
-> Object elementAt (int index) -> From Vector
-> Object firstElement() -> From Vector
-> Object lastElement() -> From Vector

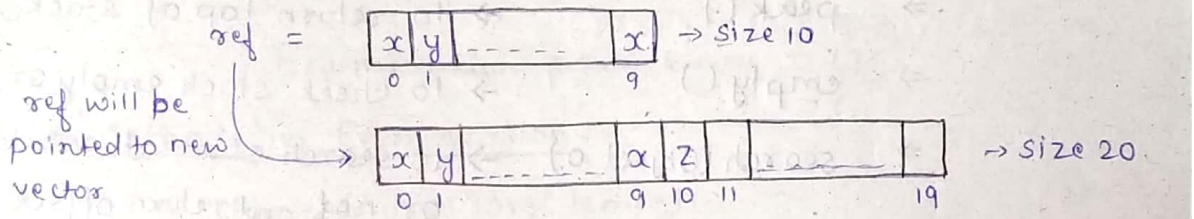
-> int size() -> No of elements available
-> int capacity() -> Size of Array
-> Enumeration elements() -> To get object one by one

=> Constructor in Vector-

ⓐ Vector v = new Vector();

↳ It will create an empty vector object with default capacity "10"

↳ When vector size is full and one new object needs to be added then a new array will be created of double the size of previous array and after adding previous array elements to new array, the 11th element will be added.



↳ performance is reduced.

ⓑ Vector v = new Vector(int initialcapacity);

↳ performance is improved but size is an issue because size will increase by double. so, memory wastage.

ⓒ Vector v = new Vector(int initcapacity, int incrementcapacity);

↳ In this performance and memory management both are taken care of. We can define the vector increment size value. once vector is full.

ⓓ Vector v = new Vector(Collection c);

It will create equivalent vector object for given collection.

ex-

	↓	↓	↓
	()	(25)	(10, 5)
Vector v = new Vector(); SOP(v.capacity);	→ 10	25	10
for(int i=0; i<10; i++){ v.addElement(i); } SOP(v.capacity);	→ 10	25	10
v.addElement("A"); SOP(v.capacity)	→ 20	25	15
	↓ {value double default value}	↓ {value set to 25}	↓ {increment by 5}

4) Stack -

↳ It is child class of Vector

↳ It is specially designed for LIFO

Last IN First OUT

* Methods in Stack

=> push(Object c) -> to add Object

=> pop() -> to remove last element

=> peak() -> to return top of stack

=> empty() -> to check stack empty or not

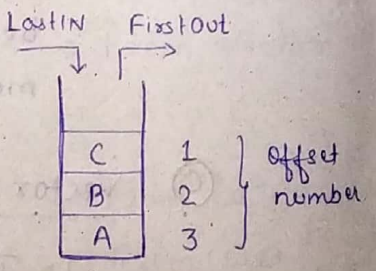
=> search(Object o) -> to search element available or not and return offset

* Constructor in Stack -

```
Stack s = new Stack();
```

ex-

```
Stack s = new Stack();
s.push("A");
s.push("B");
s.push("C");
SOP(s); -> [A,B,C]
SOP(s.search("A")); [3] -> returned offset number.
SOP(s.search("Z")); [-1] -> not available so -1
```



* CURSORS

There are 3 cursors available to retrive object one by one.

1) Ennumeration
2) Iterator
3) List Iterator

1) Enumeration-

It is used for legacy classes. We use enumeration for retaining object one by one.

```
Enumeration e = v.elements();
```

=> Methods in Enumeration-

- → boolean hasMoreElement()
- → Object nextElement()

=> Limitations of Enumerations-

1) Only for legacy classes hence not a universal cursor
2) Only "read" operation can be performed

#NOTE-
To overcome the drawbacks, Iterator is introduced.

ex-

```
Vector v = new Vector();
for(int i=0; i<10; i++){
    v.addElement(i);
}
SOP(v);
Enumeration e = v.elements();
while(e.hasMoreElements()){
    Integer I = (Integer) e.nextElement(); → return type is object so type casted.
    if(I%2==0){
        SOP(I); → 0,2,4,6,8,10
    }
}
SOP(v); → [0,1,2,3,.....9]
}
```

2) Iterator

↳ It can be used for any collection object for retaining one by one

↳ We can perform both read and remove operation

↳ It is a universal cursor

```
Iterator i = c.iterator();
```

=> Methods in Iterator-

```
boolean hasNext()
Object  next()
void    remove()
```

ex-

```
ArrayList al = new ArrayList();
for(int i=0; i<10; i++){
    al.add(i);
}
S.O.P (al);        → [0, 1, 2, ---- 9]

Iterator i = al.iterator();
while(i.hasNext()){
    Integer I = (Integer) i.next();
    if (I%2==0){
        SOP(I);
    }
    else
        i.remove();

    SOP(al);        → [0,2,4,6,8]
}
```

=> Limitations of Iterator-

1) Only Forward direction moment
2) Replace functionality is not available

 # NOTE-

To overcome these, List Iterator is introduced.

3) List Iterator-

↳ It applicable only for List implementation classes

↳ It is the most powerful cursor but is not universal

↳ It is bidirectional

↳ We can perform read, remove, replace, addition of new object.

```
ListIterator li = new ListIterator();
```

=> Methods in ListIterator-

-> Forward Operations-

```
boolean  hasNext()
Object   next()
int      nextIndex()
```

-> Backward Operation-

```
boolean  hasPrevious()
Object   previous()
int      previousIndex()
```

-> Other Operations-

```
void  remove()
void  set(Object c)    -> to set new Object
void  add(Object c)    -> to add new Object
```

ex-

```
LinkedList ll = new LinkedList();
ll.add("Ramu");
ll.add("Venky");
ll.add("Chiku");
ll.add("Nag");
SOP(ll)    -> [Ramu, Venky, Chiku, Nag]
ListIterator li = ll.listIterator();
while(li.hasNext()){
    String s = (String) li.next();
    if(s.equals("Venky"){
        ll.remove();
    }
    else if(s.equals("Nag"){
        ll.add("Chaitu");
    }
SOP(ll);
}
```

* Comparasion between Enumeration, Iterator & ListIterator

Property	Enumeration	Iterator	ListIterator
① Application	1) Only legacy class	1) All collection Object	1) Only List Class
② Movement	2) Only forward direction	2) Only forward direction	2) Both Forward/ Backward dir°
③ Accessability	3) Read only	3) Read & Remove	3) Read, Remove, add, set, replace
④ To get Object	4) .elements of vector class	4) .iterator of collection interface	4) .listIterator of ListIterator
⑤ Methods	5) hasMoreElement() nextElement()	5) hasNext() next() remove()	5) methods mentioned at back of page
⑥ Legacy	6) Yes (1.0v)	6) No (1.2v)	6) No (1.2v)

ex-

```
Vector v = new Vector();
→ Enumeration e = v.elements();
→ Iterator i = v.iterator();
→ ListIterator li = v.listIterator();

SOP(e.getClass().getName());   → Vector$1
SOP(i.getClass().getName());
SOP(li.getClass().getName());
```

Vector $ 1
↓ ClassName
| → Annonymous inner class
↓ InnerClass

NOTE-

We will not create the object for all 3 cursors.
We will use internal implementation of these classes.

* SET -

↳ It is child Interface of Collection.

↳ Duplicates are not allowed.

↳ Insertion order is not important.

↳ It donot defines any new method. Collection methods has to be used.

1) HashSet -

↳ Underlying data structure is HashTable

↳ Duplicates are not allowed.

↳ Insertion order is not preserved and object will be inserted based on HashCode

↳ Hetrogenous in Nature

↳ null insertion is allowed but only one

↳ Implements Serializable and Cloneable.

↳ Donot Implements Random Access.

↳ HashCode → Search operation is best

=> Constructor in HashSet -

ⓐ HashSet h = new HashSet();

Default initial value is "16"

Load Factor is "0.75"
ie- after filling 75% of the capacity new HashSet object will be created.

ⓑ HashSet h = new HashSet(int initialCapacity);

Load factor = 0.75

ⓒ HashSet h = new HashSet(int initCapacity, float loadfactor);

ⓓ HashSet h = new HashSet(Collection c);

ex-

```
HashSet h = new HashSet();
    h.add ("B");
    h.add ("C");
    h.add ("D");
    h.add ("Z");
    h.add (null);
    h.add (10);
    SOP (h.add ("Z")); → false (No duplicates)
    SOP (h); → [B, C, D, Z, null, 10]
```

NOTE-

Constructors are same as HashSet for

- → LinkedHashSet
- → HashMap
- → IdentityHashMap
- → WeakedHashMap
- → HashTable

2) Linked Hash Set -

↳ Child class of Hash Set

↳ Insertion order is preserved

↳ Underlying data Structure is Hash table & Linked List

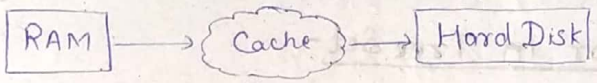
↳ No duplicates allowed

↳ Introduced in 1.4v

↳ null insertion allowed but only one

=> Where to use -

Cache based application



holds calculation
so no duplicate calcution
will be performed.

3) Sorted Set -

↳ Child Interface of Set

↳ Sorting Order is maintained

↳ Duplicates not allowed

=> Methods in Sorted Set -

first() → 100

last() → 115

headSet(104); → [100, 101, 103]

tailSet(104); → [104, 107, 110, 115]

SubSet(103, 110); → [103, 104, 107]

comparator(); → null

100
101
103
104
107
110
115

Sorted Set sample

NOTE -

Default Natural Sorting Order

For number	For Alphabets
Ascending	Alphabatically

4) Tree Set -

↳ Implementation class of Sorted Set.

↳ Underlying data structure is Balanced Tree

↳ Duplicates not allowed

↳ Insertion order not applicable

↳ Sorting order is preserved

↳ Hetrogeneous not Allowed

↳ null is accepted but only as 1st element

=> Constructor in Tree Set -

ⓐ TreeSet t = new TreeSet();

Elements will be inserted according to default Natural Sorting Order.

ⓑ TreeSet t = new TreeSet (Comparator c);

Elements will be inserted according to customized sorting order.

ⓒ TreeSet t = new TreeSet (Collection c);

ⓓ TreeSet t = new TreeSet (SortedSet s);

ex -

```
TreeSet t = new TreeSet();
t.add("A");
t.add("a");
t.add("B");
t.add("L");
t.add("Z");
// t.add("L"); -> error -> no duplicates allowed.
// t.add(null); -> error -> Null Pointer Exception
SOP(t); -> [A, B, L, Z, a]
```

↳ Default Natural Sorting Order

Rules for null acceptance-

for inserting "null" comparasion is required

=> Rule 1:

When elements are available in the Tree and if we try to add null then we get (NPE)

Internally comparable is comparing each element's getting inserted to maintain sorting order. Hence also Homogenous in nature.

```
t.add("A");
t.add(null); → NPE
```

=> Rule 2:

For empty TreeSet we can add "null" but after inserting "null", we add any element and it will throw (NPE) because we need to compare elements.

```
t.add(null);
t.add("A");→NPE
```

Class Cast Exception-

```
TreeSet t=new TreeSet();
t.add(new StringBuffer("A"));
t.add(new StringBuffer("B"));
SOP(t);    → Class Cast Exception
```

StringBuffer does not implements Comparable Interface hence gives Class Cast Exception.

#NOTE-

If we are depending on Default Natural Sorting Order

↳ It should be Homogenous and Comparable Interface

↳ String implements Comparable. So No CCE.

↳ Object is comparable only when it implements java.lang.Comparable Interface

↳ String and All Wrapper Class implements comparable interface but StringBuffer does not.

* Comparable-

↳ Available in java.lang package

↳ Has one method - compareTo()

public int compareTo(Object o)

Q. Why return type is int? and not boolean?

While comparing

obj1.compareTo(obj2)

Cases:

1) obj1 > obj2
2) obj1 < obj2
3) obj1 = obj2

So, we not have 3 values in boolean, hence int.

=> Behaviour when int is returntype

↳ return -ve, if obj1 comes before obj2

↳ return +ve, if obj1 comes after obj2

↳ return 0, if obj1 equals obj2

```
SOP("A".compareTo("Z"));   → -ve value (-25)
SOP("Z".compareTo("B"));   → +ve value (24)
SOP("A".compareTo("A"));   → 0
SOP("A".compareTo(null));  → NPE
```

ex-

```
TreeSet t=new TreeSet();  → Default Natural Sorting Order
t.add("B");   → first element, no comparision
t.add("Z");   → "Z".compareTo("B"): → +ve (24)
t.add("A");   → "A".compareTo("B"); → -ve (-1)
SOP(t);       → [A, B, Z]
```

NOTE

Because of DNSO, it internally implements compareTo() for sorting elements.

* Comparator-

↳ Available in java.util package

↳ Has 2 methods - compare()
equals()

↳ It is used for customized sorting order.

=> Methods in Comparator.

public int compare (Object c1, Object c2)

public boolean equals()

=> Behaviour when int is return type-

↳ return -ve , obj1 comes before obj2

↳ return +ve , obj1 comes after obj2

↳ return 0 , obj1 equals obj2

NOTE-

↳ Whenever we are implementing comparator Interface we must provide implementation only for compare() and the remaining method implementation is optional.

↳ Implementing equals() is optional, because it is already available in java class from Object class through inheritance.

↳ If we donot pass comparator object then internally JVM will call compareTo() which is for DNSO (i.e - ascending order)

↳ If we pass comparator object then internally JVM will call compare() which is meant for customized sorting order. (i.e descending order).

ex- Integer Comparasion

```
class myComparator implements Comparator {
    public int compare (Object obj1, Object obj2){
        Integer I1= (Integer) obj1;
        Integer I2 = (Integer) obj2;
        if (I1 < I2)
            return +1;
        else if (I1 > I2)
            return -1;
        else
            return 0;
    }
}

Class Test {
    public static void main (String[] args){
        TreeSet t = new TreeSet (new myComparator());
        t.add (10);  → no comparasion
        t.add (0);   → compare (0,10) → +ve
        t.add (15);  → compare (15,10) → -ve
        t.add (20);  → {compare (20,10) → -ve
                       {compare (20,15) → -ve
        t.add (20);  → {compare (20,10) → -ve
                       {compare (20,15) → -ve
                       {compare (20,20) → 0 (i.e no insertion)
        SOP (t); → [20,15,10,0]
```

NOTE-

```
Integer I1= (Integer) obj1;
Integer I2= (Integer) obj2;
// return   I1.compareTo(I2) ;  → [0,10,15,20]
// return  -I1.compareTo(I2) ;  → [20,15,10,0]
// return   I2.compareTo(I1);   → [20,15,10,0]
// return  -I2.compareTo(I1);   → [0,10,15,20]
```

ex- String Comparasion-

```
class myComparator implements Comparator{
    public int compare (Object obj1, Object obj2) {
        String s1 = (String) obj1;
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    // or  return -s1.compareTo(s2);
    }
}
```

both can be used for type casting to string

```
class Test {
    public static void main (String[] args){
        TreeSet t = new TreeSet();
        t.add("Roja");
        t.add("Shobha Rani");
        t.add("Raja Kumari");
        t.add("Ganga Bhawani");
        t.add("Rahul Singh");
// without myComparator  SOP(t); -> [Ganga Bhawani, Rahul Singh, Raja Kumari,
                                     Roja, Shobha Rani]
// with myComparator     SOP(t); -> [Shobha Rani, Roja, Raja Kumari, Rahul Singh,
                                     Ganga Bhawani]
    }
}
```

new myComparator or else DNSO will be called for comparasion

NOTE-

For StringBuffer Default Natural Sorting order is not applicable because StringBuffer do not implements CompareTo() So, internally we need to convert StringBuffer to string using ".toString()"

```
public int compare (Object obj1, Object obj2){
    String s1 = obj1.toString();
    String s2 = obj2.toString();
    return s1.compareTo(s2);
}
```

ex- Increasing length Comparasion-

```
class myComparator implements Comparator{
    public int compare (Object obj1, Object obj2){
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        int l1 = s1.length();
        int l2 = s2.length();
        if (l1 < l2)
            return -1;
        else if (l1 > l2)
            return +1;
        else
            return s1.compareTo(s2);
    }
}

class Test{
    public static void main (String[] args){
        TreeSet t=new TreeSet (new myComparator);
        t.add ("A");
        t.add (new StringBuffer ("ABC"));
        t.add (new StringBuffer ("AA"));
        t.add ("XX");
        t.add ("ABCD");
        t.add ("A");
        SOP (t); -> [A, AA, XX, ABC, ABCD]
    }
}
```

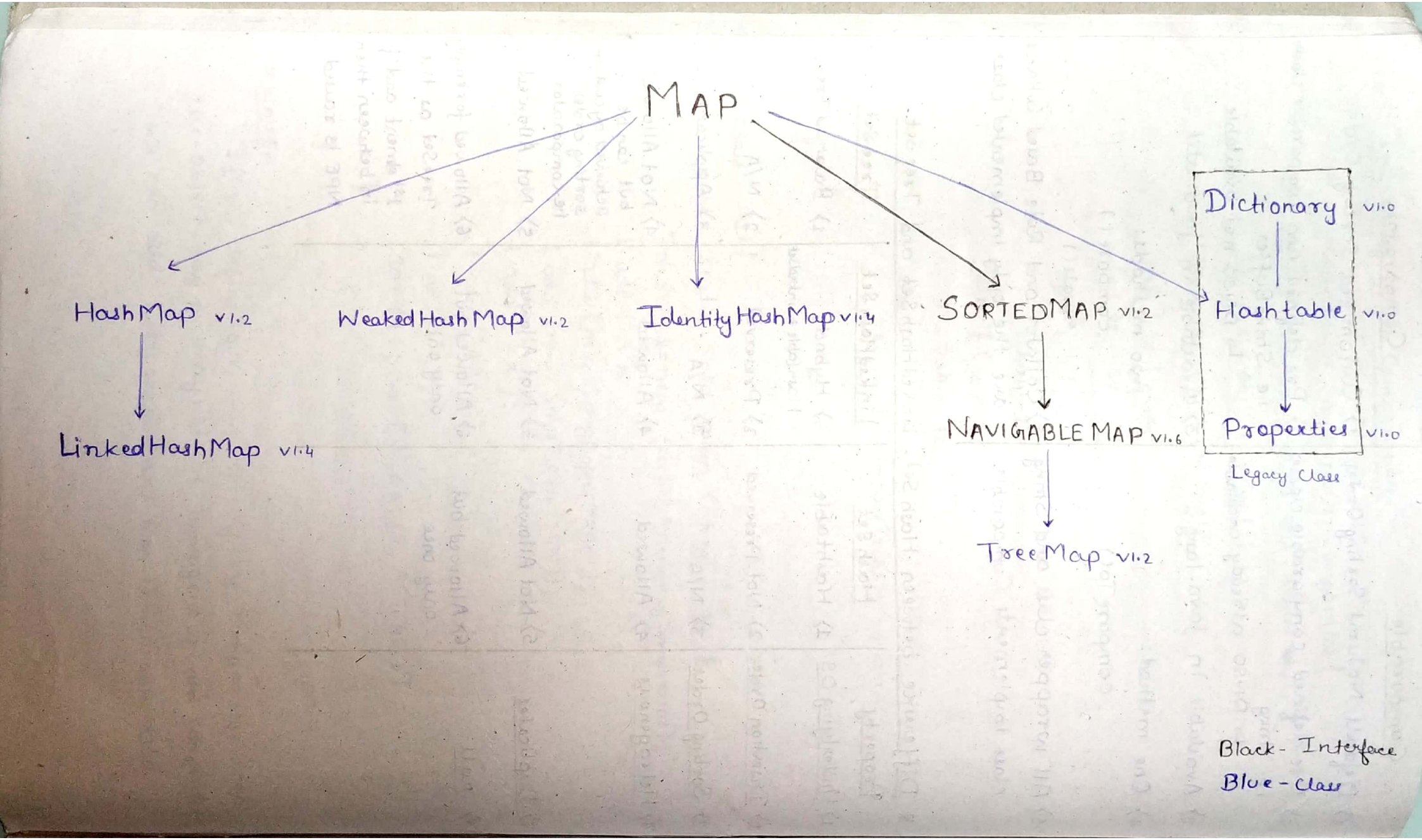
NOTE-

If we define our own sorting by comparator then the object may or may not be homogenous and comparable We can add Hetrogenous and Non-comparable also.

Comparable	Comparator
1) Default Natural Sorting Order	1) Customized Sorting Order
2) Pre-defined comparable classes i.e- String ↳ DNSO already available	2) Pre-defined non-comparable classes i.e- StringBuffer ↳ DNSO not available
3) Available in java.lang	3) Available in java.util
4) One method: compareTo()	4) Two methods: compare() equals()
5) All wrapper class and String class implements Comparable	5) Collator and Rule Based Collator are the only implemented class

* Difference between HashSet, LinkedHashSet and TreeSet.

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying DS	1) Hashtable	1) Hybrid, Hashtable + LinkedSet	1) Balanced Tree
2) Insertion Order	2) Not Preserved	2) Preserved	2) N/A
3) Sorting Order	3) N/A	3) N/A	3) Applicable
4) Hetrogenous	4) Allowed	4) Allowed	4) Not Allowed but can be achieved by custom sorting order i.e- comparator
5) Duplicates	5) Not Allowed	5) Not Allowed	5) Not Allowed
6) null	6) Allowed but only once	6) Allowed but only once	6) Allowed for empty TreeSet as the 1st element and if in between then NPE is raised.


MAP
HashMap v1.2
WeakedHashMap v1.2
IdentityHashMap v1.4
SORTEDMAP v1.2
Dictionary v1.0
Hashtable v1.0
Properties v1.0
Legacy Class
LinkedHashMap v1.4
NAVIGABLE MAP v1.6
TreeMap v1.2
Black - Interface
Blue - Class

* MAP-

- ↳ It is not a child Interface of Collection
- ↳ It represents group of Objects as key, value pairs
- ↳ Duplicate keys not allowed
- ↳ Duplicate values allowed
- ↳ Each key, value pair is called Entry
- ↳ Map contains its own specific methods
- ↳ No Collection methods are implemented

Map is called as group of Entry Objects.

Key	Value	
101	Ram	→ 1 entry
102	Sai	→ 2 entry
103	ABC	→ 3 entry
147	123	→ 4 entry

} group of entry

Key → Object
Value → Object

* Methods under Map Interface-

→ Object put (Object key, Object value)	→ add one key, value pair if key present - then old value replaced with new & returns old value.
→ void putAll (Map m):	→ Group of key, value pair
→ Object get (key)	→ get value, if no key - null
→ Object remove (key)	→ removes entry with key
→ boolean containsKey (key)	→ key availability check
→ boolean containsValue (value)	→ Value availability check
→ boolean isEmpty ()	→ Map is empty or not
→ int size()	→ Map size.
→ void clear()	→ Clears the Map entries

* Entry (I)-

Entry is an inner interface of Map Interface. Without the existance of Map Object, Entry Object cannot exists.

* Methods under Entry Interface-

→ Object getKey()

→ Object getValue()

→ Object setValue(Object newObject)

These methods are entry specified methods and can be applied only on entry Objects

* Collection views of Map -

Used for retrival operation

→ Set keySet() → key will be unique so return set

→ Collection values() → value can be duplicate so collection return type

→ Set entrySet() → entry will be unique so return set

ex-

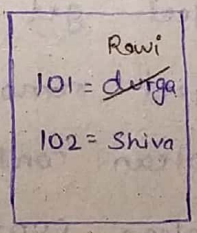
returns

null ← m.put(101, "durga");

null ← m.put(102, "Shiva");

durga ← m.put(101, "Rawi");

↓ old value



1) HashMap-

- ↳ Underlying DataStructure is Hashtable
- ↳ Insertion Order not preserved
- ↳ Based on hash code of keys.
- ↳ Duplicate keys not allowed
- ↳ Hetrogeneous allowed for keys and values
- ↳ Implements Serializable and Cloneable
- ↳ Do Not Implements Random Access
- ↳ Best choice if frequent operation is search.
- ↳ null acceptance → keys (only once)
  → values (Any number of times)

=> Constructors in HashSet-

ⓐ `HashMap m = new HashMap();`

Creates empty HashMap object with default initial capacity 16 and default Fill ratio - 0.75

ⓑ `HashMap m = new HashMap(int initialCapacity)`

Custom initial capacity with default Fill ratio - 0.75

ⓒ `HashMap m = new HashMap(int initCapacity, float fillratio);`

Custom initial capacity and fill ratio.

ⓓ `HashMap m = new HashMap(Map m);`

ex-

```
HashMap m = new HashMap();
m.put("Chiranjeevi",700);
m.put("Balaiah", 800);
m.put("Venkatesh", 200);
m.put("Nagarjuna", 500);
SOP(m);  → {K=V, K=V, K=V, K=V}
SOP(m.put("Chiranjeevi",1000));
```

→ value will be replaced by 1000 and will return 700.

ex- Retriving key, values

```
Set s = m.keySet();
SOP(s);        →{K, K, K, K}

Collection c = m.values();
SOP(s);        →{V, V, V, V}

Set s1 = m.entrySet();
SOP(s1);       →{K=V, K=V, K=V, K=V}
```

ex- Using Iterator

```
Iterator itr = s1.iterator();
while (itr.hasNext()) {
    Map.Entry me = (Map.Entry) itr.next();
    SOP(me.getKey() + ..... + me.getValue());
    if (me.getKey.equals("nagarjun.a")) {
        me.setValue(10000);
    }
}
SOP(m);
}
}
```

Q- How to get synchronized version of HashMap-?

```
HashMap m = new HashMap();

Map m1 = Collections.synchronizedMap(m);
```

Now, m1 is the Synchronized version for HashMap m.

HashMap	Hashtable
1) Every method in HashMap is Non-synchronized	1) Every method present in is Synchronized
2) Not Thread-Safe	2) Thread-Safe
3) Relatively high performance	3) Relatively Slow performance
4) null is allowed for key and values	4) null is not allowed for key and value, gives NPE
5) Not a Legacy Class V1.2	5) Legacy class V1.0.

2) LinkedHashMap-

- ↳ Child class of HashMap
- ↳ Underlying DataStructure is LinkedList + Hashtable
- ↳ Insertion order preserved
- ↳ Based on hashcode of keys
- ↳ Introduced in V1.4

All methods and constructors are same as HashMap.

NOTE-

LinkedHashSet and LinkedHashMap are commonly used for developing cached based applications.

ex-

```
LinkedHashMap lhm= new LinkedHashMap();
lhm.put ("Ravi", 101);
lhm.put ("Rahul", 102);
lhm.put ("Lokesh", 103);
SOP (lhm); → {K=V; K=V; K=V}
lhm.put ("Ravi", 107);
SOP (lhm); → value for Ravi will get replaced by
             107 and returns 101 (i.e old value);
```

3) IdentityHashMap-

eg:- Comparasion b/w == and .equals

== → compares reference of address

.equals → compares contents

```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
SOP (I1 == I2); → false   (address comparasion)
SOP (I1.equals(I2)) → true   (content comparasion)
```

↳ In case of HashMap, JVM uses .equals method to identify duplicate key.

↳ In case of IdentityHashMap, JVM uses == method to identify duplicate key.

#NOTE-

IdentityHashMap is same as HashMap including methods and constructors.

ex-

```
HashMap hm = new HashMap();
Integer I1 = new Integer(10);   → I1 = 10
Integer I2 = new Integer(10);   → I2 = 10
m.put (I1, "Pawan");            → 10 = pawan
m.put (I2, "Ravi");             → 10 = ravi (pawan)
SOP (hm); → {10 = Ravi}         → 10 = ravi
```

one value ←

```
IdentityHashMap ihm = new IdentityHashMap();
ihm.put (I1, "Pawan");
ihm.put (I2, "Ravi");
SOP(ihm); → {10 = pawan, 10 = ravi}
```

two value ←

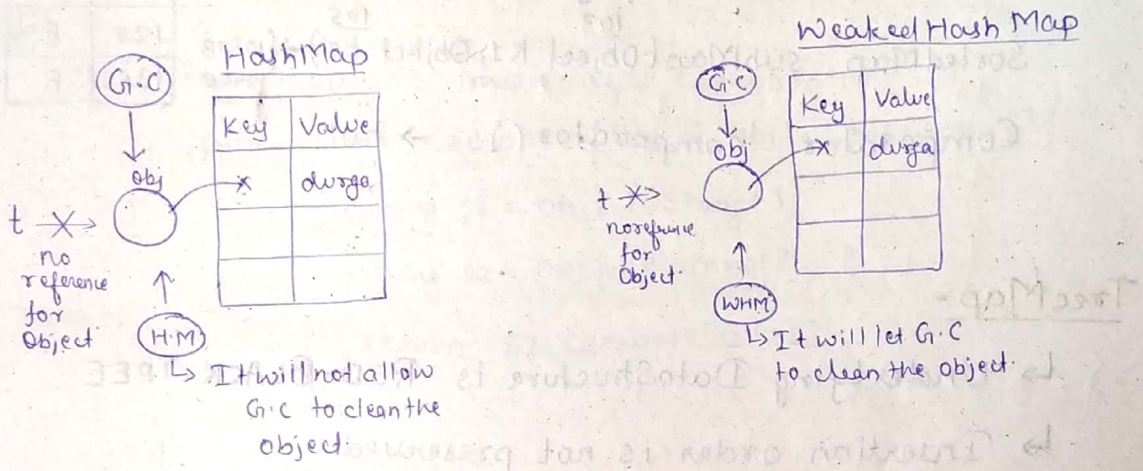
4) WeakedHashMap-

The HashMap which is weak is called WeakedHashMap.

It is same as HashMap except the below:

↳ In case of HashMap, even though object does not have any reference, it is not eligible for Garbage Collector i.e - HashMap dominates G.C and do not let GC to clean object.

↳ In case of WeakedHashMap, if object does not contains any reference, it will be eligable for Garbage Collector i.e - G.C dominates WeakedHashMap.



ex-

```
Class Temp{
    public String toString(){
        return "temp";
    }
    public void finalize(){
        SOP ("Finalized Called");
    }
}
class Main{
    HashMap m= new HashMap();
    Temp t=new Temp();
    m.put (t,"durga");
    SOP (m);  → {temp=durga}
    t= null;
    System.gc();
    Thread.sleep(5000);
    SOP(m);  → {temp=durga}
}
```

not eligible for G.C

=> for HashMap we get 2 O/p
{temp=durga}
{temp=durga}

=> for WeakedHashMap we get 1 O/p
{temp=durga}
{ }

```
Class Main{
    WeakedHashMap wm=new WeakedHashMap();
    Temp t = new Temp();
    wm.put (t,"durga");
    SOP (wm); → {temp=durga}
    t=null;
    System.gc();
    Thread.sleep(1000);
    SOP (wm); → { }
}
```

eligible for G.C hence entry removed

5) SortedMap-

↳ Child Interface of Map

↳ Represent group of entry in Sorting order of keys

↳ Sorting is based on key, not value

=> Methods in SortedMap Interface-

Object firstKey(): → 101

Object lastKey() → 136

SortedMap headMap(107 Key) → {101=A, 103=B, 104=C}

SortedMap tailMap(107 key) → {107=D, 125=E, 136=F}

SortedMap subMap(103 Object k1, 105 Object k2) → {103=B, 104=C}

Comparator comparator(): → null

Key	Value
101	A
103	B
104	C
107	D
125	E
136	F

6) TreeMap-

↳ Underlying DataStructure is RED-BLACK-TREE

↳ Insertion order is not preserved

↳ Duplicate Keys not allowed

↳ Duplicate values allowed

↳ If Default Natural Sorting Order
Keys should be Homogeneous and comparable

↳ If Custom Sorting Order
We can take Hetrogeneous and non-comparable

↳ No restriction for values, i.e Hetrogeneous and non-comparable also.

↳ null acceptance → Empty TreeMap as 1st element
→ Non-Empty, NPE.

NOTE-

=> After inserting null as first element as key, we will not be able to add another entry and it will give NPE as the keys are compared internally with .equals method. Until v1.6.

=> v1.7 onwards, null is not allowed for keys.

⇒ Constructors in TreeMap:-

ⓐ `TreeMap m = new TreeMap();`

Creates object of TreeMap with DNSO of keys

ⓑ `TreeMap m = new TreeMap(Comparator c);`

Creates object of TreeMap with C.S.O of keys

ⓒ `TreeMap m = new TreeMap(SortedMap m);`

ⓓ `TreeMap m = new TreeMap(Map m);`

eg:-

```
class myComparator implements Comparator {
    public int compare(Object obj1, Object obj2){
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

```
class Test1 {                                   DNSO
                                                 ↓
    TreeMap m = new TreeMap();
    m.put(100, "ZZZZ");
    m.put(103, "YYYY");
    m.put(101, "XXXX");
    m.put(104, 106);
    SOP(m);
        {100=ZZZZ, 101=XXXX,
         103=YYYY, 104=106}
    m.put("FFFF", "XXXX"); //CCE
    m.put(null, "XXXX"); //NPE
}
```

```
class Test2 {                                   C.S.O
                                                 ↓
    TreeMap m=new TreeMap(new myComparator());
    m.put("XXX", 10);
    m.put("AAA", 20);
    m.put("ZZZ", 30);
    m.put("LLL", 40);
    SOP(m); → {ZZZ=30, XXX=10,
               LLL=40, AAA=20}
```

7) Hashtable-

↳ Underlying DataStructure is Hashtable

↳ Insertion order is not preserved

↳ It is based on hash code of keys

↳ Duplicate keys not allowed

↳ Duplicate values are allowed

↳ Hetrogeneous objects allowed for both keys & values

↳ null is not allowed for both keys & values (NPE)

↳ Implements Serializable and Cloneable Interface

↳ Donot Implements Random Access

↳ Every method present is Synchronized

↳ Thread-Safe

↳ Best choice if frequent operation is Searching.

=> Constructor in Hashtable-

ⓐ Hashtable h = new Hashtable();

Creates Hashtable Object with initial capacity 11 and load factor 0.75

ⓑ Hashtable h = new Hashtable(int initialCapacity);

Custom initialCapacity with Default loadfactor 0.75

ⓒ Hashtable h = new Hashtable(int initCap, float loadfactor);

Custom initialCapacity and loadfactor

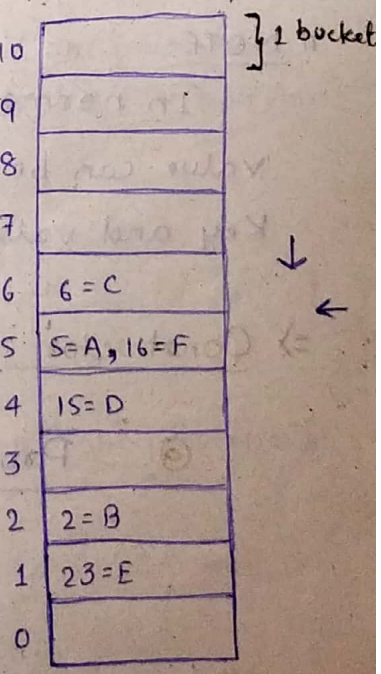
ⓓ Hashtable h = new Hashtable(Map m);

Q- Internal working of Hashtable.

It's hard to identify the hash code generated internally so we are overriding the hashCode() and toString methods so that we can provide custom hashCode.

```
class Temp{
    int i;
    Temp(int i){
        this.i=i;
    }
    public int hashCode(){
        return i;                    //> if changing as i%9;
    }                                   the complete value storage
    public String toString(){           will be changed in buckets
        return i+" ";
    }
}

Class Main{
    psvm (String arg[]){
    Hashtable h=new Hashtable();      -> default initial capacity
    h.put(new Temp(5),"A");             so buckets will be
    h.put(new Temp(2),"B");             created
    h.put(new Temp(6),"C");             11 buckets
    h.put(new Temp(15),"D");
    h.put(new Temp(23),"E");
    h.put(new Temp(16),"F");
    S.o.p(h); ->
        {6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
    //h.put("durga",null); //NPE
```

→ At 5
→ At 2
→ At 6
→ 15%11=4 At 4
→ 23%11=1 At 1
→ 16%11=5 At 5



Print order will be
Top to bottom
Right to Left

If we change default initial capacity as 25. then the output value will be changed as the values in the buckets gets changed.

8) Properties-

In our program if any thing which changes frequently (i.e Username, password, mail id, mobile No etc) are not recommended to hard code in java program because if there is any change, to reflect that change, Re compilation, Re-build, Re deploy application are required even sometimes server restart also required which creates a big bussiness impact to the client.

We can overcome these problems by using properties files. Such type of variable things we have to configure in properties file. From this properties file, we have to read into java program and we can use these properties.

The main advantage of this aproach is, if there is a change in properties file, to reflect that change just re-deployment is enough which won't create any bussiness impact to the client.

We can use java properties object to hold properties which are coming from properties file.

NOTE-

In normal Map (like HashMap, Hash Table) key and value can be any type but in the case of property, Key and value should be String type.

=> Constructor in Properties-

① Properties p = new Properties();

⇒ Methods in Properties-

→ String setProperty(String pname, String pvalue);

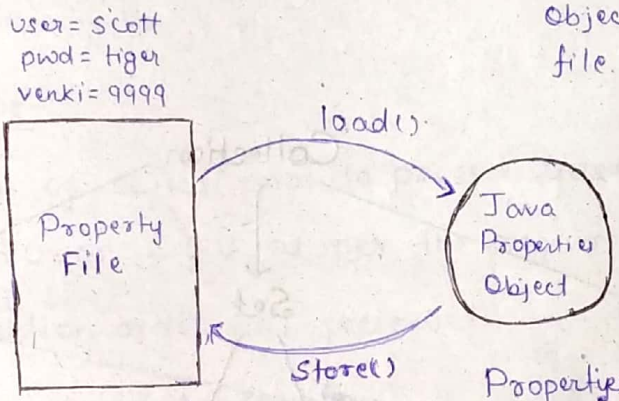
if property available then old will be replaced by new value → to set new property

→ String getProperty(String pname) → to get value of property if not available then null

→ Enumeration propertyNames() → to get property one by one

→ void load(InputStream is) → to load properties from properties file into java properties object

→ void store(OutputStream os, String comment) → to store properties from java properties object into properties file.



Properties p = new Properties()

ex-

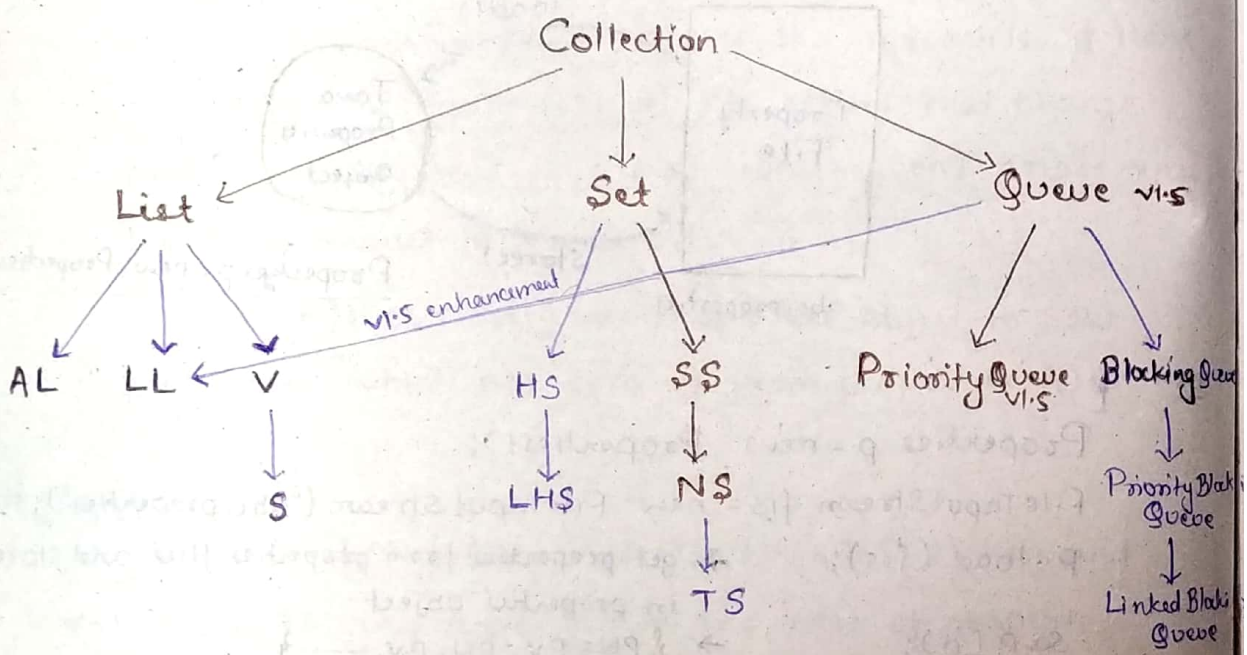
```
{
    Properties p = new Properties();
    FileInputStream fis = new FileInputStream("abc.properties");
    p.load(fis);          → get properties from properties files and stores in properties object
    So.P(p);              → {PN=PV, PN=PV, ----}

    String s = p.getProperty("venki");
    So.P(s);              → 9999
    p.setProperty("nag","88888");
    FileOutputStream fos = new FileOutputStream("abc.properties");
    p.store(fos,"Updated by Ankit");
}
}
```

Changes can be seen in properties file

ex- For Database Connection in properties file.

```
{
Properties p = new Properties();
FileInputStream fis = new FileInputStream("db.properties");
p.load(fis);
String url = p.getProperty("url");
String user = p.getProperty("user");
String pwd = p.getProperty("pwd");
Connection con = DriverManager.getConnection(url, user, pwd);
}
```

* Queue -



↳ Child Interface of Collection

↳ It represents group of individual prior to process.

↳ Generally follows FIFO order but can be prioritized based on our requirement (Priority queue)

↳ From 1.5v onwards LinkedList class also implements Queue Interface

↳ LinkedList based implementation always follows FIFO order

=> Methods in Queue Interface-

ⓐ boolean offer (Object obj) → add Object in queue

ⓑ Object poll () → remove & return head element.
if empty, return null.

ⓒ Object remove() → remove & return head element.
if empty, throw RuntimeExcpt.
(NoSuchElementException)

ⓓ Object peak() → return head element,
if empty, returns null

ⓔ Object element() → return head element.
if empty, throws R.E.
(NoSuchElementException).

1) Priority Queue-

↳ Group of object prior to process according to priority.

↳ D.N.S.O or C.S.O as per the requirement.

↳ Insertion order not preserved.

↳ Duplicates not allowed

↳ If DNSO → Only Homogenous and Comparable
C.S.O → Can be Hetrogenous and not comparable.

↳ null not accepted even as 1st element.

NOTE:-

Before sending SMS all mobile number needs to be stored in some DataStructure. In order we are adding number, same order is followed to deliver.

If we have some specific number which needs to be sent first then Priority Queue can be used.

For above First IN First OUT is best choice.

=> Constructor in Priority Queue-

(a) PriorityQueue q = new PriorityQueue();

Object of PriorityQueue will be created with default initial capacity "11" and DNSO.

(b) PriorityQueue q = new PriorityQueue(int capacity);

Custom initial capacity with D.N.S.O

(c) PriorityQueue q = new PriorityQueue(int cap, Comparator c);

Custom initial capacity and C.S.O.

(d) PriorityQueue q = new PriorityQueue(SortedSet s);

(e) PriorityQueue q = new PriorityQueue(Collection c);

ex-

```
PriorityQueue q = new PriorityQueue();   -> DNSO, will 11 cap
SOP(q.peak());  ->Null
// SOP(q.element()); -> R.E -> NoSuchElementException.
for(int i=0; i<=10; i++){
    q.offer(i);
}
SOP(q);  -> [0,1,2,3,4,----10]
SOP(q.poll()); -> [0],
SOP(q);  -> [1,2,3,4,5,----10]
```

NOTE-

Some time the output for q might gives shuffled o/p [1, 2, 5, 7, 4, 3, 8, 10, 9]. This happens because of Operating System. but not the application. Some O.S does not support PriorityQueue or Queue property.

This has to be fixed by patch file.

ex- Customized Priority-

```
Class myComparator implements Comparator{
    public int compare(Object obj1, Object obj2){
        String s1= obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}

Class Test{
    p.s.v.m.[String args[]){
        PriorityQueue q=new PriorityQueue(15, new myComparator);
        q.offer("A");
        q.offer("z");
        q.offer("L");
        q.offer("B");
        SOP(q);   → [z,L,B,A]
    }
}
```

* 1.6 v Enhancement in Collection Framework-

1) Navigable Set
2) Navigable Map

1) Navigable Set-

↳ Child Interface of Sorted Set

↳ Defines several purpose for Navigation purpose

=> Methods in Navigable Set-

→ floor(e) → returns highest element which is <= e

→ lower(e) → returns highest element which is < e

→ ceiling(e) → returns lowest element which is >= e

→ higher(e) → returns lowest element which is > e

→ pollFirst(e) → remove & return first element

→ pollLast(e) → remove & return last element

→ descendingSet() → returns NavigableSet in reverse order

ex-

```
TreeSet<Integer> t = new TreeSet<Integer>(); →DNSO
t.add(1000);
t.add(2000);
t.add(3000);
t.add(4000);
t.add(5000);
SOP(t); →                    → [1000,2000,3000,4000,5000]
SOP(t.ceiling(2000)); → 2000   {= or above 2000}
SOP(t.higher(2000)); → 3000    { above 2000}
SOP(t.floor(3000)); → 3000     {= or below 3000}
SOP(t.lower(3000)); → 2000     { below 3000}
SOP(t.pollFirst()); → 1000     {remove 1st & return}
SOP(t.pollLast()); → 5000      {remove last & return}
SOP(t.descendingSet()); → [4000,3000,2000]
SOP(t); → [2000,3000,4000]
```

2) Navigable Map-

↳ Child Interface of SortedMap

↳ Defines several methods for Navigation purpose

=> Methods in Navigable Map-

→ floorKey(e) → returns highest key which is <=e

→ lowerKey(e) → returns highest key which is <e

→ ceilingKey(e) → returns lowest key which is >=e

→ higherKey(e) → returns lowest key which is >e

→ pollFirstEntry() → remove & returns first Entry

→ pollLastEntry → remove & returns last Entry

→ descendingMap → returns NavigableMap in reverse order

ex-

```
TreeMap<String,String> t=new TreeMap<String,String>(); → DNSO
t.put("b","banana");
t.put("c","cat");
t.put("a","apple");
t.put("d","dog");
t.put("g","gun");
SOP(t);                          → {a=apple, b=banana, c=cat, d=dog, g=gun}
SOP(t.ceilingKey("c"));→ c   {= or above c}
SOP(t.higherKey("e"));→ g    { above e}
S.O.P(t.floorKey("e"); → d   {e not available} {= or below e}
SOP(t.lowerKey("e"); → d     { below e}
S.O.P(t.pollFirstEntry());→ a=apple
S.O.P(t.pollLastEntry()); → g=gun
SOP(t.descendingMap()); → {d=dog, c=cat, b=banana}
SOP(t);                          → {b=banana, c=cat, d=dog}
```

* Collections -

It defines several utility method for collection objects like sorting, searching, reversing etc.

=> Sorting Methods in Collections Class -

→ public static void sort(List l);

↳ According to D.N.S.O

↳ Must be comparable and Homogenous else CCE

↳ null not accepted → RE-NPE

→ public static void sort(List l, Comparator c);

↳ C.S.O

ex- D.N.S.O

```
ArrayList l = new ArrayList();
l.add("Z");
l.add("A");
l.add("K");
l.add("N");
// l.add(new Integer(10)); → CCE  {adding no problem while calling CCE}
// l.add(null);            → NPE
SOP("Before Sorting"+l); → [Z,A,K,N]
Collections.sort(l);
SOP("After Sorting"+l); → [A,K,N,Z]
```

ex- C.S.O

```
Class myComparator implements Comparator {
    public int compare (Object obj1, Object obj2){
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
Class Test {
    p s vm (String args[]) {
        ArrayList l = new ArrayList();
        l.add ("z");
        l.add ("A");
        l.add ("K");
        l.add ("L");
        S.O.P ("Before Sorting" + l); -> [z, A, K, L]
        Collections.sort (l, new myComparator());
        S.O.P ("After Sorting" + l); -> [z, L, K, A]
    }
}
```

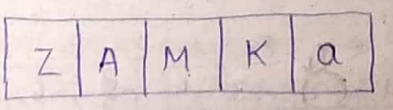
=> Searching methods in Collections Class-

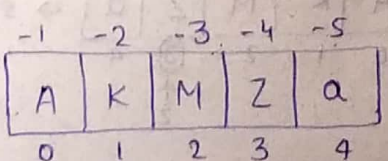
-> public static int binarySearch (List l, Object target);

↳ if element found then returns index value

↳ if element not found then returns insertion point

↳ for binarySearch ArrayList always should be sorted or else we can get unexpected output.





#NOTE-

This is used for D.N.S.O

→ public static int binarySearch(List l, Object target, Comparator c);

↳ Same as below method but here we can use this for Customized Sorting Order.

↳ List must be customized Sorted before appling binarySearch.

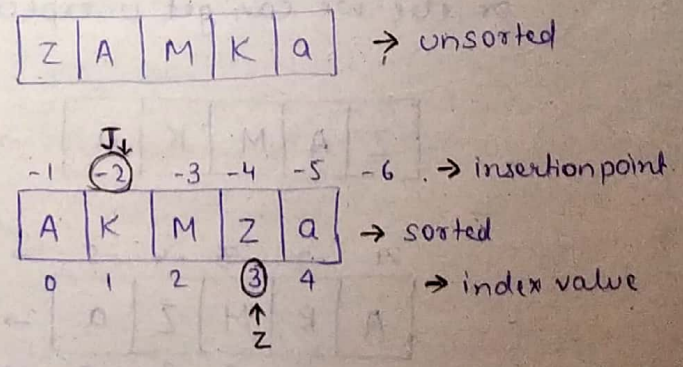
↳ At the time of search we need to define Comparator.

NOTE-

The above methods internally uses binarySearch Algorithems from DataStructures.

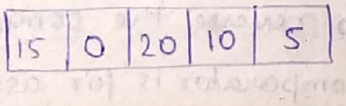
eg:- D.N.S.O.

```
ArrayList l = new ArrayList();
l.add("Z");
l.add("A");
l.add("M");
l.add("K");
l.add("a");
S.O.P(l);                                   → [Z, A, M, K, a]
Collections.sort(l); → sorting before binarySearch or else unpredictable output
S.O.P(l);                                   → [A, K, M, Z, a]
S.O.P(Collections.binarySearch(l,"Z")); → 3
S.O.P(Collections.binarySearch(l,"J")); → -2
```

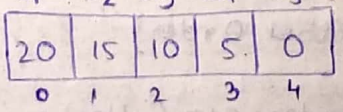


eg:- C.S.O

```
Class myComparator implements Comparator{
    public int compare(Object obj1, Object obj2){
        Integer i1 = (Integer)obj1;
        Integer i2 = (Integer) obj2;
        return i2.compareTo(i1);
    }
}

Class Test{
    p.s.v.m(String args[]){
        ArrayList l = new ArrayList();
        l.add(15);
        l.add(0);
        l.add(20);
        l.add(10);
        l.add(5);
        S.O.P(l);  -> [15,0,20,10,5]
        Collections.sort(l, new myComparator());
        S.O.P(l);  -> [20,15,10,5,0]
        S.O.P(Collections.binarySearch(l,10, new myComparator()));
                                                    -> 2
        S.O.P(Collections.binarySearch(l,13, new myComparator()));
                                                    -> -3
        S.O.P(Collections.binarySearch(l,17)); -> unpredictable o/p
```



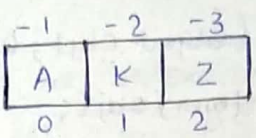
-> unsorted



-> insertion point
-> custom sorting order
-> index number

* Range-

⇒ Suppose for 3 elements



→ successful search result range : 0 to 2

→ unsuccessful search result range : -4 to -1

→ Total Result range : -4 to 2

⇒ For List or n elements

→ successful search result range : 0 to n-1

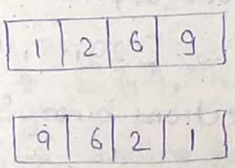
→ unsuccessful search result range : -(n+1) to -1

→ Total result range : -(n+1) to (n-1)

* Reversing Method in Collection Class-

→ public static void reverse (List d);

↳ reverse the set of elements in an Array List or List



↳ can be used on sorted and unsorted both.

→ public static ~~void~~ Comparator reverseOrder ()

↳ Used to reverse the comparator sorting order (i.e comparator is for ascending alphabet order then reverseOrder will make it descending alphabet order)

↳ Comparator c1 = Collections.reverseOrder (Comparator c);

↓ descending order

↓ Ascending order

* Arrays-

It is a utility class to define several utility methods for Arrays object.

⇒ To sort elements of Array-

→ public static void sort (primitive[] p) → DNSO

→ public static void sort (Object[] obj) → DNSO

→ public static void sort (Object[] obj, Comparator c) → CSO

ex-

```
→ int[] a = {10,5,20,11,6};
  SOP("Premitive before Sorting");
  for (int a1: a){
      SOP(a1);      → 10, 5, 20, 11, 6
  }
→ Arrays.sort(a);
  S.O.P("Primitive after Sorting");
  for (int a1:a){
      SOP(a1);      → 5, 6, 10, 11, 20
  }

→ String[] s = {"A", "Z", "B"};
  S.O.P("Object Array before Sorting");
  for (String s1 : s){
      SOP(s1);      → A, Z, B
  }
→ Arrays.sort(s);
  S.O.P("Object Array after Sorting");
  for (String s1: s){
      SOP(s1);      → A, B, Z
  }
→ Arrays.sort(s, new myComparator());
  S.O.P("Object Array after Customized Sorting");
  for (String s1:s){
      S.O.P(s1);    → Z, B, A
  }
```

#NOTE-

Take customized sorting order implementation class from back

NOTE-

We can sort premitive Arrays based on only D.N.S.O

but

We can sort Object Arrays based on D.N.S.O & C.S.O both

=> To search elements in Array

→ public static int binarySearch (primitive[] p, primitive target);
→ D.N.S.O

→ public static int binarySearch (Object[] obj, Object target);
→ D.N.S.O

→ public static int binarySearch (Object[] obj, Object target, Comparator c);
→ C.S.O

NOTE-

All roles of Arrays class binarySearch methods is same as Collections class binarySearch methods

ex-

① int[] a = {10, 5, 20, 11, 6}
Arrays.sort(a); → 5, 6, 10, 11, 20
Arrays.binarySearch(a, 6); → 1
Arrays.binarySearch(a, 14); → -5

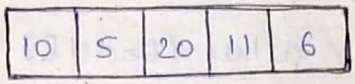
② String[] s = {"A", "Z", "B"};
Arrays.sort(s); → A, B, Z
Arrays.binarySearch(s, "Z"); → 2
Arrays.binarySearch(s, "S"); → -3

③ Arrays.sort(s, new myComparator()); → Z, B, A
S.O.P(binarySearch(s, "Z", new myComparator())); → 0
S.O.P(binarySearch(s, "S", new myComparator())); → -2
S.O.P(binarySearch(s, "N"); → unpredictable

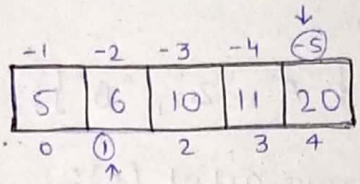
NOTE-

Take comparator implementation from back.

ex-1



→ unsorted.

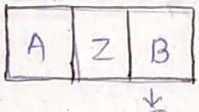


→ insertion point
→ Arrays.sort → D.N.S.O
→ index number

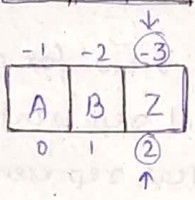
Arrays.binarySearch (a, 6); → 1

Arrays.binarySearch (a, 14); → -5

ex-2



→ unsorted



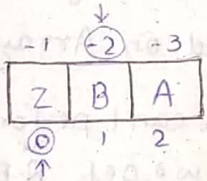
→ insertion point
→ Arrays.sort → D.N.S.O
→ index number

Arrays.binarySearch (s, "Z"); → 2

Arrays.binarySearch (s, "S"); → -3

ex-3

→ unsorted



→ insertion point
→ Arrays.sort → C.S.O
→ index number

Arrays.binarySearch (s, "Z", new myComparator());→ 0

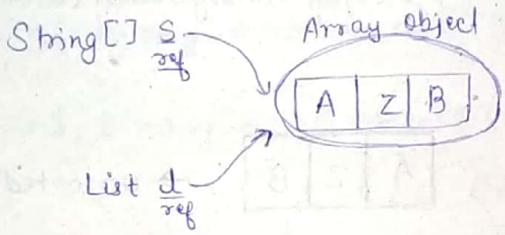
Arrays.binarySearch (s, "S", new myComparator());→ -2

Arrays.binarySearch (s, "N"); → unpredictable output

* Conversion of Array to List-

→ public static List asList (Object[] obj)

```
String[] s = {"A", "Z", "B"};
List l = Arrays.asList (s);
```



Its like creating a view for a table in database
view is - logical representation
table is - physical representation

This method donot creates indepent list object, for existing Array it creates a view form ~~for~~ of List from Array.

By using Array reference if we perform any change, the change will be reflected on List

Similarly if we perform any change in List, the change will be reflected on Array.

By using List referencewe can't perform any operation which varies the size or else we get R.E - UnsupportedOperationExc.

ex-

```
S[0] = "K"; →  [K, Z, B]
S.O.P(l);   →  [K, Z, B]

l.add("M");    →  } R.E - UnsupportedOperationException
l.remove(1);   →  }  (i.e it increases or decreases size of Array)

l.set(1,"N");→ [K, N, B]

l.set(1, new Integer(10)); → R.E. ArrayStoreException
                             (it dont allow Hetrogeneous
                              replacement and gives R.E)
```