Garbage Collection ⇒

⇒ In java destruction of object from memory is done automatically by the jvm.

⇒ No 'delete' keyword in java.

⇒ When there is no reference to an object, that object is not longer assumed to be no longer needed and memory occupied by the object are released.

This technique is called Garbage Collection.

⇒ This is accomplished by JVM.

jvm Threads ⇒ Whenever you run a java program, jvm creates three Threads.

(i) Main Threads
(ii) Thread Scheduler
(iii) Garbage Collector Thread.

⇒ In these three threads, main thred is a user thread, and remaining too are daemon threads which run in background.

⇒ The task of main thread is to execute the main method
⇒ The task of thread scheduler is to schedule the threads.
⇒ The task of Garbage Collect thread is to sweep out abondon objects from heap memory.

⇒ We can call garbage collector explicitly using System.gc() or RunTime.getRunTime().gc()

⇒ But it's just a request to garbage collector, not a command. It may run or not.

Finalize method ⇒ Before garbage collection starts to free any object, then before destructing that object it can call finalize method. It some how similar to distructor.

⇒ Finalize method (function) is protected, non-static function of java.lang.Object.

```
protected finalize() throws Throwable
{
    // Code here.
}
```

⇒ If we dont make finalize-method, default finalize will be call.

⇒ We can call explicitly finalize method, but it will not clear memory [destruct object].

Limitation with Array ⇒

(i) Fixed in size.

(ii) Can hold homogeneous type data only.

(iii) No underline data structure support.
So complexity of programme will increase

Collections ⇒

Note ⇒ We can store multiple object value in a single array, using object.array.

```
Object [] a = new Object [10000];
a[0] = new Student();
a[1] = new Customer();
```

⇒ Size can be vary at run time.

⇒ can hold homgeneous & Hetrogenous data.

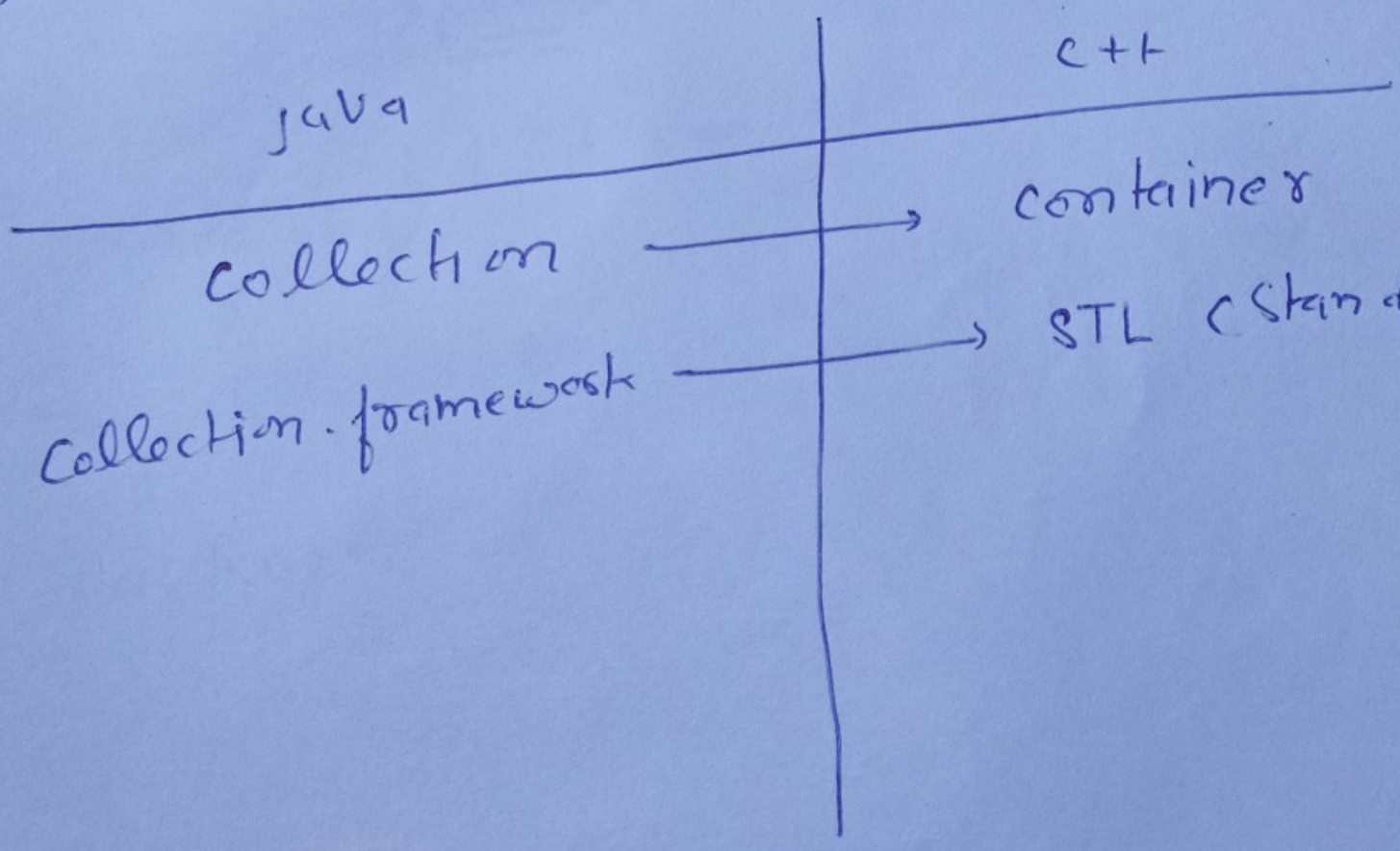
⇒ underline data structure support. readimade method support abt available.

Collection V/s Array ⇒

Arrays	Collection
(i) Fixed in size	(i) growable in size.
(ii) memory utilisation not good	(ii) better memory utilisation
(iii) Performance wise, it is recommanded.	(iii) Performance wise, not recommanded
(iv) We can create primitive and object array	(v) collection can hold only object not primitive.

Collection ⇒ If we want to create a group of individual object as a single entity, we should go for collection.

Collection-Framework ⇒ It contains, classes and ~~interface~~ interfaces. which can be used to represent a group of individual object as a single entity i.e collection.



~~Interface~~

Collection ⇒

(i) It is an interface

(ii) If we want to represent a group of individual object as a single entity, then we should go for collection.

(iii) Collection interface defines most common methods which are applicable for any collection object, in general. Collection interface is considered as root-interface of collection-framework.

(iv) There ~~is~~ is no concrete class, ~~to use~~ which implements collection interface directly.

Difference b/w collection & collections ⇒

(i) collection is an interface. collections is class.

(ii) collections class defines several utility method for collection object.

example ⇒ ArrayList does not contain any method for shorting, but it can use collections utility method sort().

(iii) collections is an utility class present in java.util package.

Collection framework ⇒

⇒ The collections framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a de collection.

⇒ Collection ~~Interface~~ framework contains following ⇒

(i) Interface → These are abstract data types that repres collections.

(ii) Implementations → [Classes] → These are concrete implementation of collection interfaces.

(ii) Algorithms ⇒ These are the methods that perform useful computation, such as searching and shorting, on objects that implements collection- Interfaces.

⇒ Array has certain limitations.

(i) collection of similar data-types.

(ii) size must be fixed.

(iii) operations → searching, shorting etc. we need to make function.

→ whereas collection —

(i) can store dissimilar data-types

(ii) dynamically size can be vary.

(iii) predefined function for operation.

(iv) we can represent group of variables as a single entity.

(v) Classes, has abstract datatypes. → methods and structure.

Collection or Arrays ⇒

*. If size is known, arrays are better to implement in terms of performance.

*. Collection provides flexibility in memory which increases the time complexity.

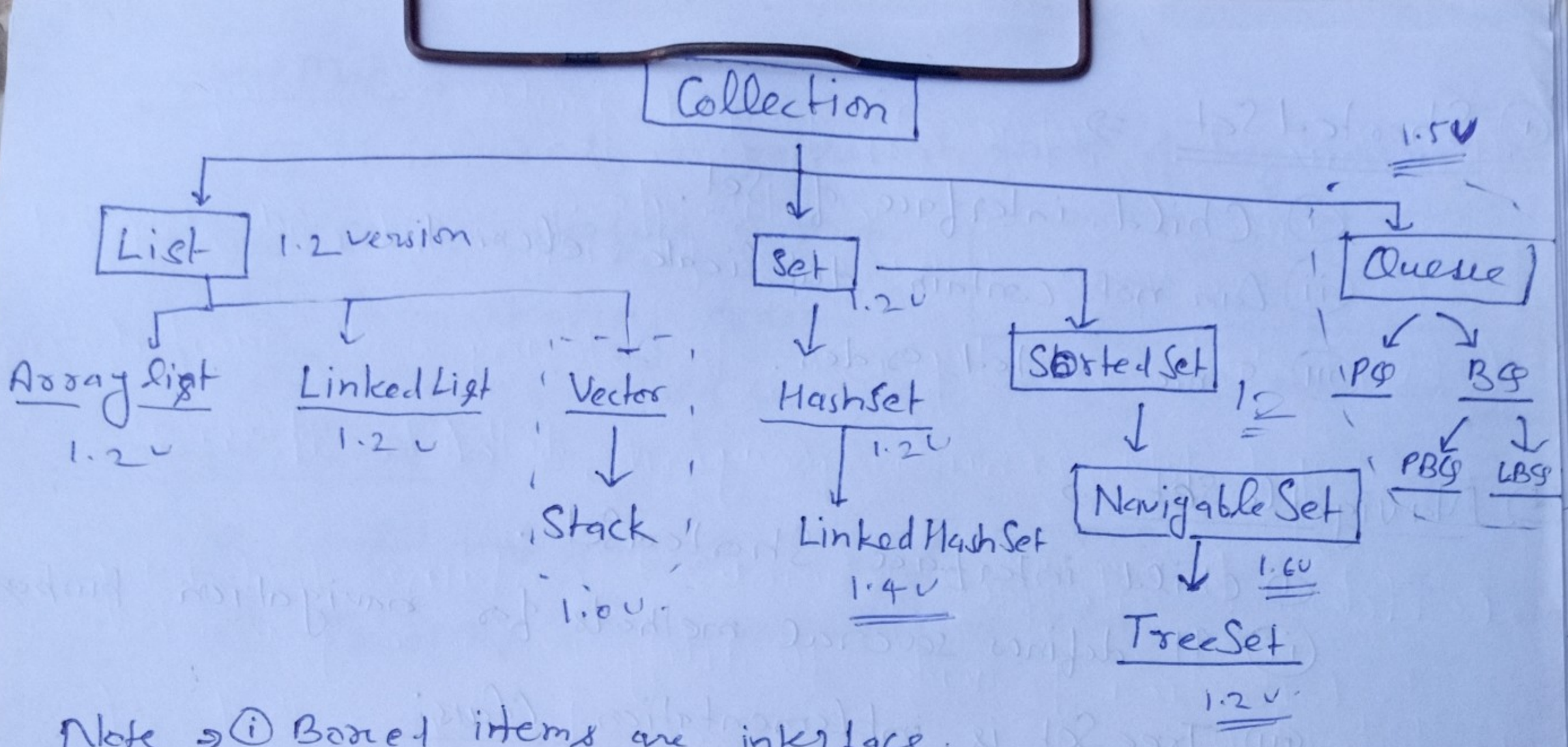
Interfaces in collection framework. ⇒

* Collection → Group of objects.

* List (i) Duplicates allowed.
(ii) Insertion order must be preserved.
(iii) Arraylist, Linkedlist, Vector, Stack are the implementation classes.

* Set → (i) A collection that can not contain duplicate elements
(ii) Insertion order is not preserved.
(iii) Hashset, LinkedHashset. are the classes implementing Set interfaces.

⇒ vector & Stack are Dagacy classes. [old version].
Here are reengineered to implement new version.



Note ⇒ (i) Boxed items are interface.

(ii) underlined items are classes.

(3) SortedSet ⇒

(i) Child interface of Set.

(ii) Can not contain duplicate element.

(iii) Some sorted order.

(4) NavigableSet →

(i) Child interface SortedSet.

(ii) It defines several methods for navigation purpose

(iii) TreeSet is implementation class.

(5) Queue.

(i) Child interface of Collection.

(ii) PriorityQueue, BlockingQueue, LinkedBlockingQueue

FIFO ← PriorityBlockingQueue.

(6) Map ⇒

(i) It is used to represent group of objects as key-value pair.

(ii) Map is not a child interface of collection.

(iii) key and values both are objects.

(iv) key must be unique but values can repeat

(v) Hashmap, LinkedHashmap, WeakHashmap, IdentityHashmap properties.

SortedMap

(i) It is used to represent group of key-value pairs
(ii) Child interface of Map.
(iii) Some sorting order. | - According to some order of keys
(iv) Sorting should be based on key but ~~or~~ not on value

NavigableTable →

(i) Child ~~imp~~ interface of sorted map.
(ii) TreeMap is implementation class.

Sorting

(i) Comparable → interface for default sorting.
(ii) Comparator → interface for customized sorting..

Collection and Collections ?

(i) Collection is an interface.
(ii) Collections is a class.

Collections class provides several utility methods like sort().

contains ADD collection) .

Methods of Collection ⇒

(i) boolean add(Object o); add an object to the collection

return type — this object will be added

(ii) boolean addAll(Collection c);

will add all objects of collection c in the collection

(iii) boolean remove(object o) ⇒

Remove a single instance of the specified element from this collection

(iv) boolean removeAll(collection c)

Remove all of this collection's elements that are contained in collection c.

(v) boolean retainAll(collection c)

all objects that are not in collection c will be removed from the collection.

(vi) int size() ⇒ will return no of elements in collection.

(vii) boolean contains(Object o) ⇒ Returns true if object o contains the specified object.

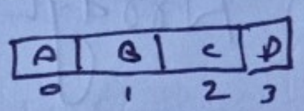
containsAll (collection c) ⇒

(viii) void clear() ⇒ all the instances (element) of the collection will be removed.

(ix) boolean isEmpty() ⇒ Returns true if this collection contains no element.

(x) Iterator<E> iterator() ⇒ Returning iterator over the elements in this collection.

Methods of List Interface ⇒ we can differentiate duplicate value by index no..



① void add(int index, E element) ⇒ inserts the specified element at the specified position in list (element may shift)

void addAll(int index, Collection c) ⇒ will add inorder objects of collection c, starting from index.

(ii) E set(int index, E element) ⇒ replaces the element at specified index, with specified element.

(iii) E get(int index) ⇒ returns element of specified index.

(iv) int indexOf(Object o) ⇒

(v) int LastIndexOf(Object o) ⇒

(VI) E remove (int index)

(VII) List<E> subList (int fromIndex, int toIndex) ⇒

Returns a view of the portion of the list b/w fromIndex (inclusive) & toIndex (exclusive).

(IX) ListIterator ListIterator();

ArrayList ⇒

(I) ArrayList is defined using dynamic array.

(II) So it is resizable.

(III) Duplicates are allowable.

(IV) order of elements are preserved.

(V) Null insertion is possible.

(VI) Heterogeneous objects are allowed.

Except TreeSet and TreeMap, Heterogeneous objects are allowed.

ArrayList: Constructor ⇒

(1) ArrayList a = new ArrayList() ⇒

creates an empty ArrayList object with default intial capacity "10" if arrayList reaches it's max capacity then a new ArrayList object will be created with

New capacity = (current capacity * 3/2) + 1
                                          ↓
                                         one

Program →

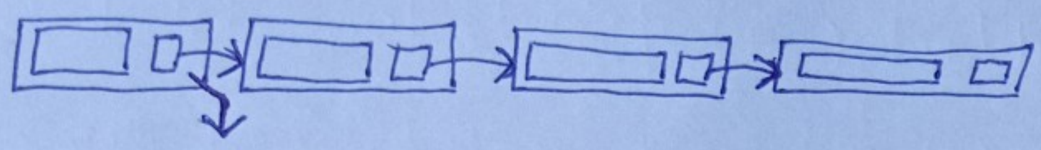
```
import java.util.*;
public class Example
{
  public static void main(String[] args)
  {
    ArrayList l1 = new ArrayList(4);
      l1.add("Ajay");
      l1.add("Rohan");
      l1.add(1, "Ram");
    Iterator i1 = l1.iterator();
    while(l1.hasNext())
    {
      System.out.println(i1.next());
    }
  }
}
```

(ii) ArrayList a = new ArrayList(int intialcapacity);→

Now intial size will not be 10, it will be equal to initial capacity.

(iii) ArrayList a = new ArrayList(Collection c);

LinkedList ⇒



ArrayList	LinkedList
① memory is taken in a row sequentially.	① Not important.
② cost of insertion an element is expensive.	

⇒ Usually we can say every collection interface's class by default implements -

① serialisable interface.

② Clonable interface.

⇒ ArrayList and Vector class implemts also RandomAccess interface.

Random-Access we can access any element with same speed. wit whether we need to get 1st or 1 Lacks th element.

⇒ It has no method so it is ~~marker~~ marker interface

⇒ When frequent element is retrieval, then ArrayList is best choice. because of Random Access.

⇒ When frequent operation is insertion or deletion any element in middle of ~~Arraylist~~, collection, ArrayList is worst choice.

⇒ When Frequent operation of insertion or deletion of any element in middle of collection is required, LinkedList is best choice.

ArrayList	Vector
(i) Every method present here are non-synchronised.	(i) Almost every method is synchronised.
(ii) multiple thread can operate at a time, so not thread safe.	(ii) Thread safe because one thread at a time allowed.
(iii) mutiple thread can operate so no need to wait any thread so performance is good	(iii) Peaformance is high.
(iv) Non-Lagacy class	(iv) Lagacy class

How to get synchronised version of ArrayList object

```
ArrayList l1 = new ArrayList ();
List l1 = Collections.synchronisedList (l1);
```

l1 ↓ Synchronised

l1 ↙ non-synchronised.

public static List synchronisedList (List l)

⇒ For map & ~~List~~ set also we can use →

⇒ Collections.synchronisedMap (map m)

⇒ Collections.synchronised Set (Set s);

⇒ **LinkedList** ⇒ the underlind data structure is doubly Linked List, ~~not~~

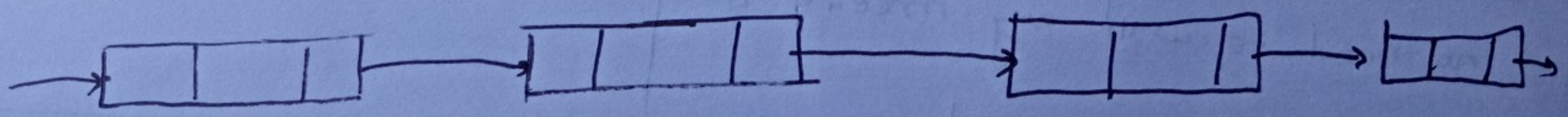
(i) insertion order is preserved.

(ii) duplicate objects are allowed

(iii) Hetrogeneous objects allowed.

(iv) Null insertion possible.

(v)



constructor in LinkedList ⇒

⇒ There is nothing like capacity in LinkedList.

⇒ (i) LinkedList l = new LinkedList (Collection c);

(ii) LinkedList l = new LinkedList ();

LinkedList specific Method ⇒

(i) Usually LinkedList can be used to develop stacks and queue. To provide support for this requirment LinkedList class defines following methods.

(i) void addFirst (object o)

(ii) void addLast (object o)

(iii) object getFirst ();

(iv) object getLast ();

(v) object removeFirst ();

(vi) object removeLast ();

This methods can only be used in LinkedList.

*Difference b/w ArrayList & LinkedList ⇒

ArrayList	Linked List.
when frequent retrival is required, it is best choice.	worst Choice
when frequent operation is insert or delete in middle, worst choice	best Choice.
elements are stored in consecutive memory location so retrival operation is easy	elements are not stored in consecutive memory location

Vector ⇒

(I) insertion order preserved.
(II) Duplicate ✓
(III) Hetrogeneous object ✓
(IV) Thread safe.
(V) Null insertion ✓
(VI) elements are stored in consective memory location.

Constructor ⇒.

(1) vector v = new vector(); ⟶ size 10.

new capacity = current capacity * 2.

(II) Vector V = new Vector (int initial_capacity)

(III) Vector V = new Vector (int intial_capacity, int increament)

once initial capacity reached, capacity will be increased by adding increament.

vector V = new vector (10, 5)

initial → 10
after → 15
20
:

(IV) Vector V = new Vector (collection C);

Methods in Vector :

(I) addElement (Object O);

(II) removeEement (object O);

(III) addElement At (int index);

(IV) remove All Elements ();

(V) ~~int size~~ int size ();

(VI) int capacity (); → returns capacity.

Stack ⇒ Last in First out.

(11) Child class of vector.

Constructor

① stack s = new stack();

Method

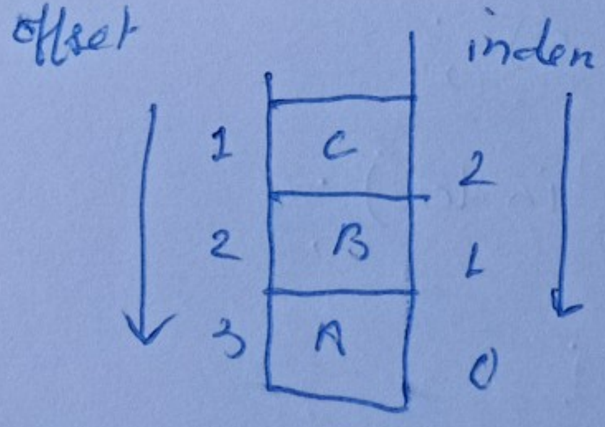
(i) obj.push (object o)

~~puff~~.

(ii) obj. pop ()~~object~~

(iii) bolean empty () → checks if empty.

(iv) int search (object) → returns offset



(v) object peak () → returns the element present at top of stack.

Cursor ⇒ If we want to get object one by one from collections, then we should go for cursor.

They are three types of cursor available in java.

(i) Enumaration.
(ii) Iterator.
(iii) ListIterator.

(i) Enumeration ⇒ we can use enumeration, to get objects one by one from legacy collection object.

we can create enumeration object by using elements method of vector class

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

methods -

(i) e.hasMoreElements() → returns true or false

(ii) e.nextElements() → returns next object.

~~e.next~~

e.nextElement().

Limitation of Enumeration ⇒

① Applicable only for lagecy class.

(11) we can get read Accers only, we can not remove elemove element.

(ii) **Iterator** ⇒ It is a global-curshor, so we can use it every object of collection.

(6) we can read and remove elements from collection.

```
Iterator I = c.Iterator ();
```
(c: Any object)

① methods

(i) public boolean hasNext ();

(ii) public Object next ();

(iii) pulic void remove ();

Limitation ⇒

① we can not get move in back-ward direction.

(11) we can not add element, replace element.

(11) List Iterator ⇒ we move forward and backward direction. So it is a bidirectional cursor.

⇒ we can read, write, replace and add and remove element.

```
ListIterator li = C.listIterator();
```

⇒ Methods child class of Iterator. So all methods of Iterator are applicable here. other than those method →

(i) pubic int nextIndex();

(ii) public boolean hasPrevious();

(iii) public object previous().

(iv) Public int previousIndex().

(v) public void remove()

(vi) public void add(object o)

(vii) public void set(object o);

Limitation ⇒ most power cursor. but applicable only in List object.

Set ⇒

(i) HashSet ⇒ underline data structure for HashSet is Hash table

(b) duplicate not allowed

(c) insertion order not preserved. objects are inserted based on

(d) Hash code.

(d) null - insertion ✓

(e) ~~the~~ where frequent operation required is search, then HashSet is best applicable.

* if we try to insert duplicate value, it will not be added and will return false, no compile time or ~~run~~ runtime error.

**.

Constructor ⇒

(1) HashSet h = new HashSet ();

initial size → 16

default Fill ratio = 0.75

it means after filling 75% of capacity, new ~~ex~~ extended capacity will increase. it will not wait for filling completely.

(II) HashSet h = new HashSet (int initial_capacity);

(III) HashSet h = new HashSet (int initial capacity, float fill ratio)

(IV) HashSet h = new HashSet (Collection c);

Linked Hash Set ⇒ the only difference b/w

HashSet and LinkedHashSet is that insertion order is preserved in LinkedHashSet. ∴

LinkedHashSet = LinkedList + HashSet

All constructor, methods are as same as HashSet.

HashSet	LinkedHashSet
(I) underline data structure Hashtable	(II) Linked + Hash set.
(II) Insertion order preserved not	(II) Preserved.
(III) introduced in 1.2 version	(IV) 1.4 version.

* Linked Hash set is used to develop chache memory type things.

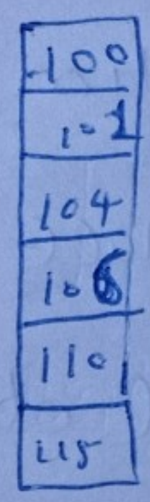
Shorted Set ⇒

(i) all the individual objects will be represented according to some sorting order, without duplicates, then we should go for shorted set.

Methods ⇒

(i) first() ⇒ will give smallest element after sorting.

will give → 100



(ii) last() → 110

(iii) headSet(106) ⟶ [100, 101, 104]

(iv) tailSet(106) ⟶ [106, 110, 115]

(v) subSet(101, 115) ⟶ [101, 104, 106, 110]

include exclusive

(vi) comparator(), returns comparator object that describes underlying sorting technique. If we are using default sorting technique, it will return null.

default sorting technique →

Number → assending order.

String → alphabatical order

TreeSet ⇒ Underlind data structure for TreeSet is **Balanced Tree**.

(i) duplicate → x

(ii) insertion order - x

(iii) Hetrogeneous object x.

⇒ If we try to insert Hetrogeneous object, class cast exception will be thrown.

(iv) Null insertion (only once allowed).

⇒ All objects will be inserted according to some sorting order.

Constructor ⇒

① TreeSet t = new TreeSet ( );

⇒ Here objects will be inserted according to default sorting order.

② TreeSet t = new TreeSet (comparator c);

⇒ Here objects will be inserted according to customized insertion order, specified by comparator c.

(iii) TreeSet t = new TreeSet (Collection c);

→ default natural sorting order.

(iv) TreeSet t = new TreeSet (SortedSet s);

→ sorting according to sorted Set.

null acceptence ⇒

(i) we can not add null in non-empty TreeSet.

if we do the same, we will get null pointer exception.

(ii) we can add null in empty TreeSet. but after inserting null, we can ~~inserting~~ not insert any other value. otherwise we will get null pointer exception

∴ untill 1.6 version null is allowed as first element in Treeset, by 1.7 version onwards, it is not allowed.

If we are depending upon default natural sorting order, we will ~~got~~ have → need

∴ Homogenous + Comparable object

(iii) TreeSet t = new TreeSet (Collection C);

→ default natural sorting order.

(iv) TreeSet t = new TreeSet (SortedSet s);

→ sorting according to sorted Set.

null acceptence ⇒

(i) we can not add null in non-empty TreeSet.

if we do the same, we will get null pointer exception.

(ii) we can add null in empty TreeSet. but after inserting null, we can ~~inserting~~ not insert any other value. otherwise we will get null pointer exception

* untill 1.6 version null is allowed as first element in Treeset, by 1.7 version onwards, it is not allowed.

⇒ If we are depending upon default natural sorting order, we will need have →

* Homogenous + Comparable object

⇒ String & wrapper class are comparable

⇒ But StringBuffer is not comparable.

⇒ Comparable has a methode i.e. compareTo (Object O);

public int compareTo (Object O)

O1.compareTo(O)
- → return (-ve) if O1 comes befor O
- → return (+ve) if O1 comes after O
- → returns 0 if O1 & O are same.

⇒ TreeSet uses, always compareTo method internally. to insert any value.

Comparator ⇒ comparator has two methods →

① compare ()⇒

Public int compare (Object O1, Object O2);

- → returns (-ve) O1 comes before O2
- → returns (+ve) O1 comes after O2
- → returns 0 O1 and O2 are same.

⑪ public boolean equals (Object O1);

O.equals (O1) → true or false

* Whenever we are implementing comparator interface compulsary we should provide implement for compare method. and we are not required to provide implementation for equals method ~~for~~ because it is already available to our class, from object class through inheritence.

⇒ Whenever we go for default sorting in TreeSet, we use comparable & for customised sorting we use comparator.

⇒ Write a code to put 0, 10, 15, 5, 20, in dessending order.

```
class TreeSetDemo
{ Public static void main(String [] args)
{ TreeSet t = new TreeSet(new MyComparator());
t.add(0);
t.add(10);
t.add(15);
t.add(5);
t.add(20);
t.add(20);
t.add(20);
}
```

```
Class MyComparator implements Comparator
{
    Public int compare ( Object O1, Object O2)
    {
        Integer I1 = (Integer) O1;
        Integer I2 = (Integer) O2;

        if ( I1 < I2)
            return (-1);
        else if ( I1 > I2)
            return (+1);
        else
            return (0);
    }
```

→ ①

If instead of ① we can also write →

① return - I1.compareTo (I2)

or

② return I2.compareTo (I1)

⇒ For String →

```
String S1 = (String) obj1
String S2 = obj2.toString();
return S2.compareTo(S1);
```

or `return - S1.compareTo(S2);`

⇒ If we are using default sorting, Homogeneous and comparable object must be.

⇒ But in customized sorting Heterogeneous object can be inserted.

Question Write a code to put Heterogeneous object in sorting order of increasing length, if two objects have same length, they should come in alphabetical order.

```
class Tree
{ public static void main (String [] args)
{   TreeSet t = new TreeSet (new MyComparator ());
    t.add ("A");
    t.add ("ABC");
    t.add (new StringBuffer ("ABCD"));
    t.add (new StringBuffer ("KL"));
    t.add (new "xx");
    Sop (t);
}
class MyComparator {
    pulic int compare (Object obj1, object obj2)
    String s1 = obj1.toString ();
    String s2 = obj2.toString ();
```

~~if(~~ ∈

```
int i1 = s1.length();
int i2 = s2.length();
if (i1 < i2)
    return(-1);
else if (i1 > i2)
    return(+1);
else
    return (s1.compareTo(s2));
```

∴ Sorting order is not present in HashSet and LinkedHashSet, it is only present in TreeSet.

-: Map :-

* Map is not child interface of Collection

* If we want to represent a group of object as key-value pairs, then we should use map.

* Each key-value pair is known as Entry.

* Both key and values are object only.

* Duplicate key is not allowed, but value can be duplicate.

So map can be said as ~~Entry~~ group of entry object.

* Collection method can not be used here, we Map has it's own specific method.

(1) Object put (Object key, Object value);

(a) To add one key value pair to the map.

(b) If key is already present, then old value will be replaced with new value and old value will be return. otherwise return will be NULL

(ii) putAll(map m);

(iii) m.get(key) → will return value.

(iv) m.remove(key) → to remove one key-value pair.

(v) m.containsKey(key) → is this key present or not.

(vi) m.containsValue(value) → is this value present or not

(vii) m.isEmpty() → map is empty or not

(viii) m.size() → size of map.

(ix) m.clear() → to clear map.

(x) Set keySet() → to get all the keys.

(xi) Collection values() → to get all the values.

(xii) Set entrySet() → to get all key-value pa

⇒ Last three methods are also known as, collection

-view pair.

* Entry ⇒ A map is a group of key-value pair.
and each key-value pair is Entry.

without existence of map, Entry-existence is meaningless, so Entry-interface is defined in

map-interface

```
interface Map{
  ...
  interface Entry{
  }
}
```

```
Interface Map
{
  ....
  Interface Entry
  {
    object getKey();
    object getValue();
    Object setValue(Object new_object);
  }
}
```

Entry specific method and we can apply only only on Entry object.

*. Hash Map ⇒

(i) Underline data structure is Hash Table.

(ii) Insertion order is not preserved, it is bassed on hash code of key.

(iii) Duplicate key not allowed, but duplicate value allowd.

(iv) Heterogeneous objects are allowed for both key & value.

(v) Null is allowed for key only once but for value it is allowed multiple times.

(vi) Hash map implimants serialisable and clonable but not Random Access.

(vii) Hash Map is best choice if frequent operation is searching.

Constructor

(i) HashMap h = new HashMap ( );
initial size = 16.
default fill ratio = 0.75

(ii) Hash Map h = new HashMap (int intialSize).

(iii) Hash Map h = new Hash map (int intialsize, float fillratio

(iv) HashMap h = new Hash Map (map m);

Hash Map	Hash Table
(i) Not synchronised	(i) Synchronised methods
(ii) multiple thread can operate	(ii) one thread at a time.
(iii) Not thread safe	(iii) Thread safe
(iv) Good performance	(iv) Low performance
(v) Null → key ✓, value ✓	(vi) Null → key ×, value ×
(vi) 1.2 version	(viii) 1.0 version

⇒ Linked Hash Set & Linked Hash map is commonly used for ~~cheat~~ cache. map is used.

⇒ '==' compares reference and .equals method compares value.

* .
```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);

I1 == I2 ⟶ false
I1.equals(I2) ⟶ True..
```

* Identity HashMap ⇒

```
HashMap m = new HashMap();
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);

m.put(I1, 'Pawan');
m.put(I2, 'kalyan');

syso(m);
```

⇒ HashMap uses .equals method to compare so I1 & I2 are same so only one value will be added o/p → {10='kalyan'}

⇒ How to get synchronised HashMap →

Using →

```
HashMap m = new HashMap();

Map m1 = new Collections.synchronisedMap(m);
```

synchronised. Not synchronised

LinkedHashMap – LinkedHashMap is exactly same as HashMap, the only difference is that →

(i)	HashMP	Linked Hash Map
(i)	underline data structure is Hash table.	(i) LinkList + Hash Table.
(ii)	insertion order not preserved and based on Hashcode of key	(ii) insertion order is preserved
(ii)	1.2 version	(iii) 1.4 version.

Linked Hash map is child class of Hash Map.

Sorted Map ⇒

If we want a group of key-value pair object according to some sorting of key.

Method ⇒

(i) firstKey() ⟶ 101

(ii) lastKey() ⟶ 136

(iii) headMap(107) ⟶ {101=A, 103=B, 104=C}

(iv) tailMap(107) ⟶ {125=E, 136=F}

(v) subMap(103, 125) → {103=B, 104=C, 107=D}

(vi) comparator(); → Null.

101 — A
103 — B
104 — C
107 — D
125 — E
136 ⟶ F.

TreeMap ⇒

(i) underline data structure → RED - Black Tree.

(ii) Insertion order X. sorting ↳

(iii) Duplicate X.

(iv) Hetrogeneous ⊗ — key X
value ✓

But in case of default sorting, we can take Hetrogeneous key and non-comparable key

⇒ HashMap & IdentityHashMap are exactly same, the only difference is →

In case of normal HashMap, jvm will use .equals method, to indentify duplicate key. which is meant for content comparsion.

But in IdentityHashMap, jvm will use '==' to identify duplicate ~~comp~~ key, i.e reference key.

WeakHashMap ⇒