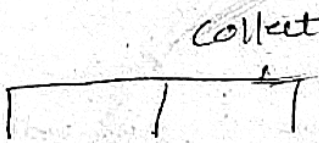
Collections: It is used for storing multiple objects, in a single entity.

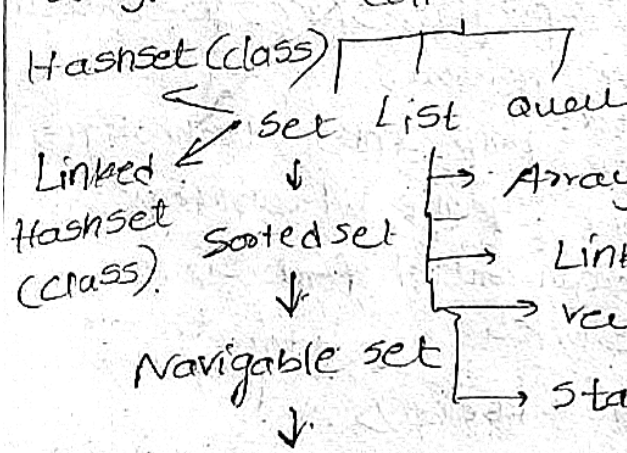
Properties of collection:-

→ Dynamically growable.
→ collection stores only objects.
→ It has underline datastructures for storing objects, it is temporary container.
→ It has predefined methods to access objects inside collections.
→ It stores homogenous and hetergenous.

Collection framework:- → library which contains predefined classes & interfaces.



→ All set, list, queue, map ~~is~~ all these are interfaces only.



→ collection framework is introduced in jdk 1.2 version
→ All collections are part of java.util package
→ In jdk version, collection.

D/F b/w collection & database

→ database is permanent storage container.
→ dynamically growable container.

set:-

set doesn't allow poistion based operations.
set doesn't allow duplicates so we may loose some data.

sorted set:- which doesn't allow duplicate

→ set interface stores and returns objects in random order.

(2)

→ data structure for hashset is hashmap
→ Datastructure for List is Hashmap+LinkedList
→ Datastruc for treeset Binary tree.
→ hashset is performing very fast, because it is not following any order.
→ compared to hashset linked hashset is slow.

List:
list is a collection / container of elements, which allows duplicate values.
list are ordered, persists insertion order.
It allows duplicate values, index/poistion operations are possible in lists.

Map: map is a container of <key, value> pairs when we want to store pair of elements then we use maps.

Collection framework:
It is a predefined library which comes along with jdk software, Collection framework written by sun micro system.

properties:
→ collection framework library contains various classes, interfaces, & logic related to different collection types
→ It was introduced in jdk 1.2 version
→ All collection framework classes & interfaces are part of java.util package.
→ In jdk 1.5 version collection framework underwent some major changes like,
  → type safety
  → support for - each with collection given
→ and they call it as generics. collections has lot of issues, in general we will be using only generics from now on.

Set Interface:

add() → to add new element to the set.

addAll() → addAll(Collection<> c)

it add elements of other collections to set.

clear() - to remove all elements of the set.

contains(Object obj) - to search whether an element exists or not.

containAll(Collection<> c) - to search if all elements available in the set or not.

isEmpty() - to check whether the set is empty or not.

Iterator<E> iterator() [It is an interface]

to read elements from set.

remove(Object obj)

to remove particular element from set.

removeAll(Collection<

retainAll(Collection<> c): - It find common elements between other collections and set.

size() - to return the total number of elements from the set.

Object[] toArray() - to convert the set into array

SortedSet Interface:-

first() - returns first element

last() - returns last element

headset(Object o) - returns all the elements which are less than object o.

tailset(Object o) - returns all the elements which are greater than or equal to the object o1 and less than object o2.

Comparator() :- returns Comparator object which is used to know the underlying sorting technique.

Navigable Interface:

floor(e)

lower(e)

ceiling(e)

higher(e)

pollFirst()

pollLast()

descendingset()

```
package exception;
import java.util.HashSet;
public class sample {
    public static void main(String[] args){
        HashSet<Integer> s1 = new HashSet<Integer>();
        s1.add(10);
        s1.add(25);
        s1.add(30);
        s1.add(4);
        s.o.p(s1); => [4, 25, 30, 10] [method to access set elements]
        for(int element:s1) { } -> it-method.
            s.o.p(element);
        }
HashSet<Integer> s2 = new HashSet<Integer>();
    s2.addAll(s1);
    s.o.p(s2);
    s2.add(100);
    s2.add(-2);
    s.o.p(s2);
s1.contains s.o.p(s1.contains(89)); -> false
            s.o.p(s1.contains(10)); -> True
            s.o.p(s1.contains(30)); -> False
            s.o.p(s1.containsAll(s2)); -> False
            s.o.p(s2.containsAll(s1)); -> True
        s1.remove(10);
        s.o.p(s1);
        s1.remove(100); -> If there is no element, then it doesn't give any error, just print what elements are there.
        s.o.p(s1);
    s2.removeAll(s1);
    s.o.p(s2);
    s.o.p(s2.size());
    Iterator<Integer> i = s2.iterator
```

-> hasNext is used to check next element is present or not

-> next() - method used to return the element.

remove() - It will remove the past element.

```
Iterator<Integer> i = s2.iterator();
while(i.hasNext()) {
```

s.o.p(

}

List: used to store group of individual objects in insertion order.
List allows duplicate elements.
→ Lists are implemented by Arraylist, linkedlist, vector.
→ Collection is an interface, which contains 12-methods apart from this some more methods are there in list.

1. add(int pos, EE) → object or element.
→ List interface performs position based operations.
2. addAll(int pos, collection<> c)
add all positions elements.
3. get(int pos) → returns element located at position pos.
4. indexOf(Object o) - returns the position of object o in in this list)
5 lastIndex(Object o) - searches from last
6. ListIterator<E> listIterator()
returns listiterator object to read/access elements by one by one.
7. remove(int pos) → removes elements at located at position
8. set(int pos EE)) → replaces element at position pos with new element e.
9. List<E> subList(int frompoistion int topposition)
returns a new list object which contains elements b/w frompoistion to topoistion

Arraylist:
→ which is same as array, which stores data in sequential order.
→ Arraylist size is dynamically growable.

5	7	9	10	12
0	1	2	3	4

insert(3,2)
→ dynamically

→ For insertion or deletion arraylist is not correctly suited i.e it takes so much of time (linked list).
→ whereas for searching process arraylist is useful.

→ package exceptions; Arraylist
import java.util.Hashset;
public class Sample {
public static void main(String[] args) {
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(10); → type safety
al.add(15);
al.add(20);
System.out.println(al)
al.add(1,63);
so.p(s1)

ArrayList<character> all = new ArrayList<character>(); → type safety
all.add('a');
all.add('i');
al.addAll(0, all); → it will get error, because it is having type safety. (i.e integer) here we want to add character, which is not possible.

→ Vector is synchronized for multiple threads, Array is not synchronized for multiple threads.

How to pass or store class in collection:

import java.util.Scanner;

public class sample {
public static void main(String[] args) {
arraylist<student> al = new arraylist<student>(); → here we are creating collection for student class
System.out.println("Do you want to store new object "yes"/"no");
String response = sc.next();
while (response.equals("yes")) {
s.o.p("Enter the student number");
int num = sc.nextInt();
s.o.p("Enter the student name");
String name = sc.nextInt();
student s1 = new student(num, name); → dynamic
al.add(s1)
student s2 = new student(12, "devi"); → we don't know how many elements we will add so use dynamic.

```
al.add(
S.O.P("Do you want to store new
        object? yes/no");
response=sc.next();
}
S.O.P(al);
```

SortedMap interface:-

firstKey() : returns the first (lowest) key currently in this map.

lastKey() :- returns the last (highest) key currently in this map.

headMap(K toKey) - Returns a view of the portion of this map whose keys are strictly less than toKey.

tailMap(K fromKey, K toKey) - returns a view of the portion of this map whose keys are greater than or equal to fromKey.

subMap(K fromKey, K toKey) - Returns a view of the portion of this map whose keys range from fromKey inclusive to toKey exclusive

comparator() - Returns the comparator used to order the keys in this map or null if this map uses the natural ordering of its keys

Navigable interface:-

1. floorKey(e)
2. lowerKey(e)
3. ceilingKEY(e)
4. higherkey(e)
5. pollFirstEntry()
6. pollLastEntry()
7. descendingMap()

→ This is the child class of vector class.
→ datastructure [resizable array]
→ follows First in Last out.
→ pop, push, peek.
→ Every collection implement Colonable interface (taking copy of object) and Serializable interface → data transmission.

↓ For transmitting data from local [i.e your system] to server we use serizable interface.

Colonable and serizable interface are marker interface [marker interface means no or empty interface].

Map Interface:

store group of key value pairs.
map interface is not child interface of collection.
In map one key is associated with only one value.
keys should be unique.
→ key value pairs called as entry.
→ values can be duplicated.

- Map
  - Hashmap
  - Linkedhashmap
  - Treemap [Implemented from Navigable Map sorted map].

Hashmap

stores in random order.

Hashmap internally uses hashtable to store key, value pairs.

directly implemented map interface.

Fast

By default 11 key value pairs we

Linked Hashmap

stores in insertion order.

uses linked list along with hash table.

Linked hashmap is directly implementing map interface.

Slower than hashmap

Treemap

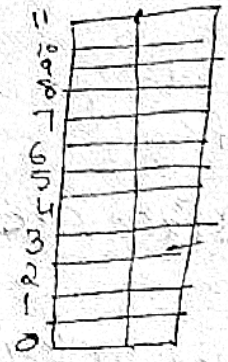
stores key, value pairs in sorted order By default ↓ Ascending

~~store~~ uses binary tree. (Red-Black tree or balanced tree].

Indirectly implementing map interface.

[directly implemented navigable map].

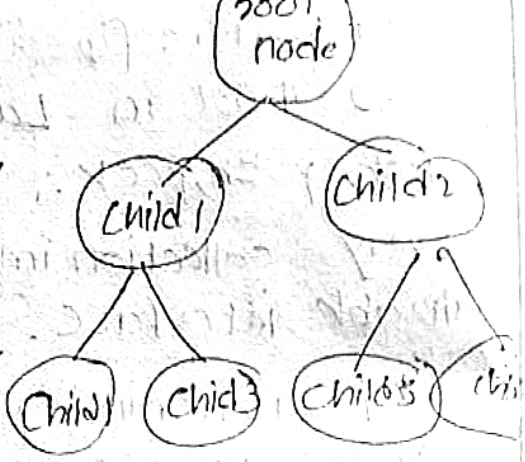
very slow.



And keep on dynamically growing.

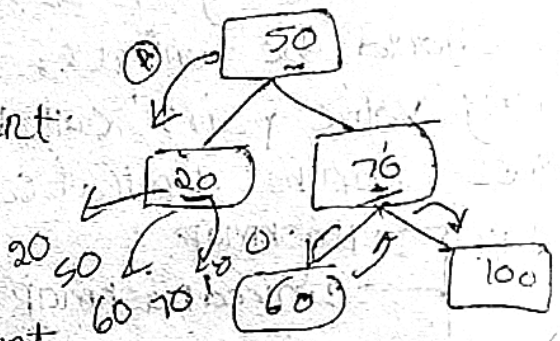
doubly linked list:

Actually data is stored in hash table only, but it will be given to Linked List.



Based on keys sorting will happen.

50 - Hi
20 - Hello
70 - welcome
100 → Pupa.



→ First it will search root node, and then first it search left side. → Here it 2 times.

Method declarations in map interface:

1. int size() - returns total no. of key value pairs:
2. isEmpty() - returns true if map is empty.
3. containsValue(Object Value) - return true if the value is present in map.
4. containsKey(Object key) - returns true, if the key is present in map.
5. get(Object key) - returns value associated with this key, else null.
6. put(Object key, Object value); inserts a new key value pair in map.
7. remove(Object key) → removes key value pair associated with this key.
8. PutAll(map m); → inserts all key, value pairs of map m into current map.
9. Set<E> keySet(); returns all keys from this map.
10. Collection<E> values() → returns all values from the map.
11. Set<Entry> K, V >> entrySet(); returns all key value pairs from this map as set of object.

→ hashCode is nothing integer representation of your object address.

→ Created by using hashcode() present in object class.

```
Package exceptions;
import java.util.ArrayList;
public class sample {
    public static void main(String[] arg s) {
        Hashmap<String, String> hl = new Hashmap
For key &                    <String, String>();
value.  hl.put("name", "deepti");   Here we didn't use add because, map is not child of collection
        hl.put("emailId", "deepti@gmail.com");
        hl.put("address", "Banglore");
        S.O.println(hl);  → {address="Banglore, name=deepti,
        S.O.P(hl.entryset()); email=deepti@gmail.com}
        S.O.P(hl.get("name")) [address=Banglore, name=deepti,
                                email=deepti@gmail.com].
        S.O.P(hl.get("age")) → It will return null.
        S.O.P(hl.keyset()); → [address, name, emailid]
                              only keys → returns collection
        S.O.P(hl.values()); → [deepti] → collection
        S.O.P(hl.containsKey("emailid")); → True
        S.O.P(hl.containsValue("Banglore")); →
        Hashmap<String, String> h2 = new hashmap(String, String>();
        h2.put("course", "Java");
        h2.putall(hl);
        S.O.P(h2); → {} → everything will give.
        S.O.P(hl.size());
        S.O.P(h2.size());
```

Collection framework:-

Interface

→ Interface layer contains various interfaces used in this library.

→ each collection type is represented by one interface describing the behaviour of that collection type.

Implementation:-

→ this contains the actual logic of collection frameworks

→ all the collection frameworks interfaces are implemented by different classes by giving function body & logic.

algorithms:-

→ common functions which you can apply on multiple collections.

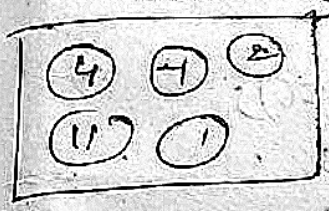
- search()
- swap()
- rotate()
- shuffle()

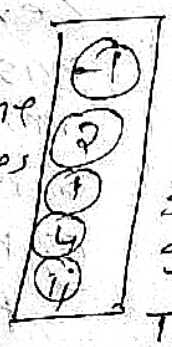
hashset	linkedhashset	treeset
→ doesn't allow duplicate values → un-ordered → hashset internally uses hashmap to store values	→ doesn't allow duplicate values → ordered set → it follows insertion ordered	→ doesn't allow duplicates → sorted set order → it reads elements either in ascending (or) descending ord.

hashset:-

1, 2, 4, 11, 1

stored randomly → get the values





→ it is very fast.

linked hashset:- ligntly slow

1, 2, 4, 11, 1

stored orderly → get the values

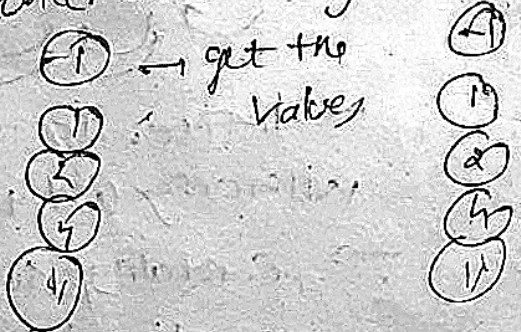
1
2
4
11
1

1
2
4
11

Treeset is a collection that implements the navigableset and sorted set interfaces providing a sorted set of elements. treeset is backed by a treemap, which means the element in the tree-set are stored automatically in a natural ascending order.

Treeset:- 1, 2, 4, 11, 1

→ sorted ascending (or) descending



Arraylist, linkedlist → slow
↓ why ?
stored in sequence order
faster

Iterators: - Are used to access elements of set or list, these are 2-types of iterators.

iterator

public interface iterator
<e> {
hasnext();
next();
remove();
}

parent interface,
supports only one way navigation.

next();

10 20 30 40 ⊙

iterator can be used to access elements of set/list.
Contains/supports few functalities.

listiterator

public interface listiterator
<e> extends iterator
<e> {
hasnext();
next();
remove();
hasprevious();
previous();
add(e e);
previousindex();
nextindex();
set(e e);
}

Child of iterator interface
supports only 2- way navigation.

next();
previous();

10 20 30 40

→ listiterator can be used with only list types.
→ Contains/supports 9 functionalities

→ Fixed size.
→ Array concept is not implemented on some standard data structure hence readymade method support is not available for every requirement we have to write the code explicitly. which is complexity of program.

Collections:
→ Growable in nature.
→ can hold both homogenous and heterogenous.
→ Every collection class can implement on standard data structure.

If ~~Array~~ we are ~~converting~~ taking one array with 10 size. If 11th element is created in Array it will give error. whereas in Arraylist, another bigger Arraylist object created interally all those are copied into new one and 11th element is added.
→ Performance arrays are good.
→ ↓ Are not good in collections
→ performance
→ Arrays can hold both primitive and object types.
→ Collections only take objects not primitives
Collection framework: It defines several classes and interfaces which can be used a group of objects as single entity.
→ In C++ we called collections as containers
→ In C++ we called collections framework as STL (Standard Temple library).
→ A class is thing which will implement interface
q-key interfaces of collection framework:
→ means which are implemented for all collection objects.
→ There is no concrete class which implements collection interface directly.

→ Collection vs Collections

→ Collection is an interface which can be used to represent a group of individual objects as a

Arraylist (1.2 version)
Linkedlist (1.2 version)
Vector (1.0 version)
Stack (1.0 version)
Collection (1.2 version)
Set (1.2 version)
Hashset (1.2 version)
Linkedhashset (1.4 version)

Collection (1.2) (1.2 version) → Set is child interface of Collection.

Difference

Sorted set: → child interface of set
→ If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects should be inserted according to some sorting order then we should go for sorted set.

Navigable set: (1.6) child interface of sorted set
↓ 1.6 version defines several methods for navigation purposes.

Navigableset
1.6 version →

Queue: child interface of collection.
If we want to represent a group of individual objects prior to processing then we should go for Queue.

Collection — Queue — PriorityQueue
(1.2) (1.5) (1.5)

→ If we want to represent a group of objects as key value pairs then we should go for map interface.

Map → Hashmap → Linkedhashmap

→ Duplicate keys are not allowed, but values can be duplicated.

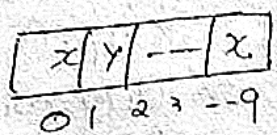
→ SortedMap (1.2) → Interface

should go for sortedmap.

Navigable map(I):-

→ child interface of map.
→ add() → when we want to add one element
→ addAll() → when we want to add group of elements
→ remove() → when we want to delete one element.
→ removeAll() → when we want to delete group of elements
→ clear()
→ retainAll(c) → remove all objects except those present in c.
→ isEmpty()
→ size()
→ contain(object o) → for single object.
→ containsAll(c) → for collection of elements)
→ object[] toArray()
→ Iterator iterator()

List(Interface):-



1) add("A");
2) add(int index, object o) → single object at specified index
3) addAll(int index, collection c)
group of objects at specified
4) remove(int index)
5) indexOf("A");
6) lastIndexOf("A"); → returns index.
7) get(int index) → object located in specified index.
8) ~~listIte~~ ListIterator listIterator()
9) set(int index, object o) → replace

Arraylist:- resizable or growable.
Duplicates.
insertion order
heterogenous objects. (except treeset and treemap heterogenous objects are allowed)
null insertion is possible.
10 → 16 → 25
→ Arraylist constructor

NewCapacity = (current capacity * 3/2) + 1
for arraylist.

3. ArrayList al = new ArrayList (Collection c);

↓ ↓ linked list + vector, stack

→ Interconnection to arraylist.

```
l.add("A");
l.add("10");
S.O.P(l); [A, 10, A, null]
```

→ Cloning:- Every collection default

↓

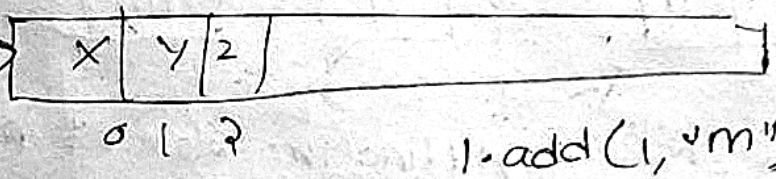
Serializable.

→ Usually we can use collections to hold and transfer objects from one place to another place, to provide support for this requirement every collection already implements serializable and cloneable interfaces.

→ Arraylist and random we can access randomly [random access interface].

→ If we are retrieving then arraylist is highly recommended.

```
S.O.P(ArrayList l1=new ArrayList();
      LinkedList l2= new LinkedList();
      S.O.P(l1 instanceof Serializable); //true
      S.O.P(l2 instanceof cloneable); //true
      S.O.P(l1 instanceof RandomAccess); //true
      S.O.P(l1 instanceof RandomAccess); //false
```



l.add(1, "m");

→ If we want to insert M at one place, then Y will move 2 and here 2 contains another element it will move to next.

→ So, internally inserting element then we add Array list is worst choice.

Difference b/w. ArrayList and vector:

ArrayList	Vector.
Every method present Arraylist is non-synchronize.	Every method present in linkedlist is synchronized.

allowed to operate on vector object and hence ArrayList is not thread safe.

operate on vector object is thread safe.

Threads are not required to wait to operate on ArrayList, hence relatively performance is high.

Threads are required to wait to operate on vector object and hence relatively performance is low.

→ Introduced in 1.2

→ 1.0 version.

How to get synchronized version of ArrayList object

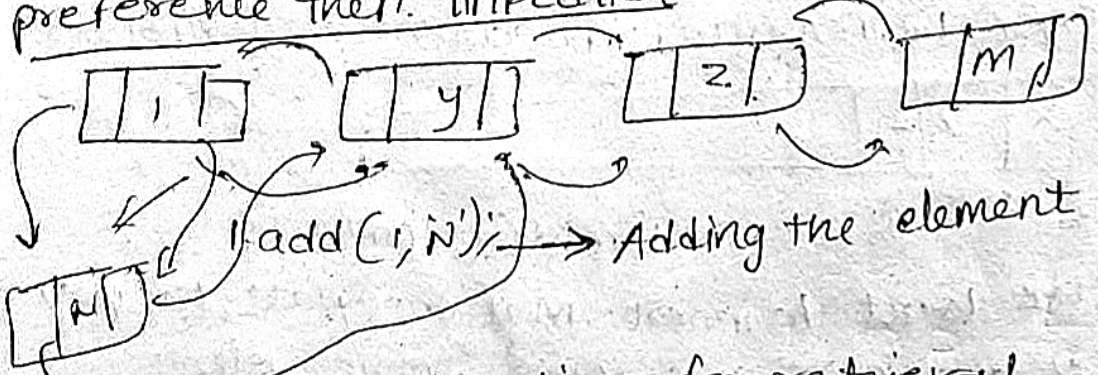
→ By default ArrayList is object is non-synchronized but we can get synchronized List() method

```
public static List synchronizedList (List l)
    ArrayList ll=new ArrayList(); → non synchronized
    List l = Collections.synchronized
                List(ll); → synchronized
```

similarly set and map we get synchronized.

```
public static set synchronized set (set s);
public static map synchronized map (map m);
```

LinkedList: Insertion and deletion of your first preference then linkedlist



→ Adding the element in middle.

→ It takes more time for retrieval, if we want 4th element, then it will check from 1st onwards.

→ underline datastructure is dably linked list.

→ insertion order.

→ Duplicates,

→ Hetereogenous elements.

→ null insertion possible