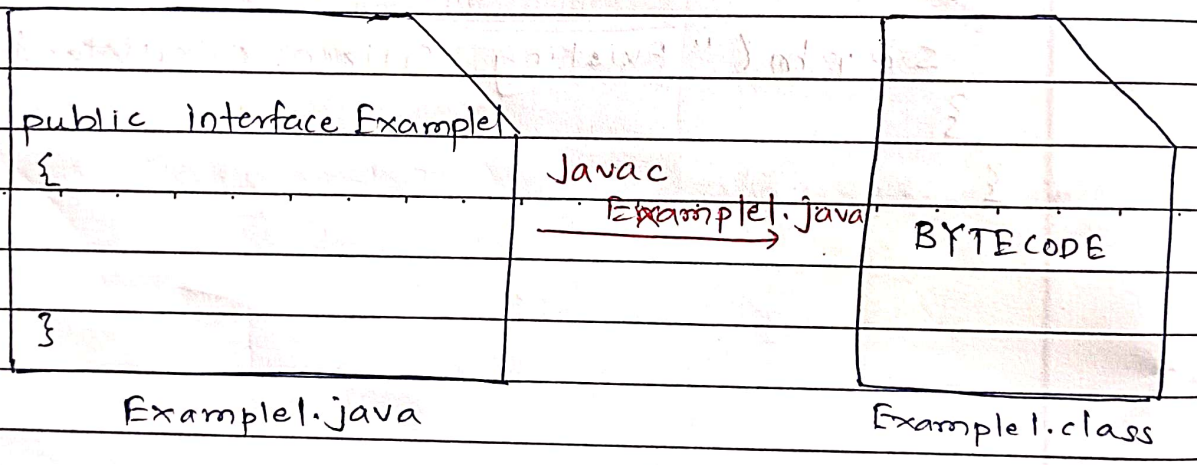
* INTERFACE

Syntax :-

```
[modifier] interface InterfaceName
{
    ========
}
```

-Interface is one of the component of java which helps in achieving 100% abstraction



Component allowed in interface.

1) Non-static abstract method is allowed.

public abstract void test();

2)

```
public interface Example3
{
    public void display()
    {
        S.o.pln ("display method");
    }
}
```

Error : interface abstract method cannot have body.
Non-static concrete method is not allowed.

4) public interface Example4
{
    void test();
}

Note:- If we declare a method with return type along with method signature and declaration ends with the ';' then the method is by default public and abstract in nature.

3) static concrete method are allowed. (From JDK-8)

public interface Example5
{
    public static void method()
    {
        S.O.pln ("static concrete method");
    }
}

Note: From JDK-8 onwards static methods are allowed in an interface.

4) From JDK 8 onward we can declare default methods which is allowed

```
public interface Example6
{
    default void test()
    {
      S.O.pln("Concrete method");
    }
}
```

5) Constructors are not allowed in interface

```
public interface Example7
{
    public Example7()
    {
      S.O.pln("Constructor");
    }
}
```

Error :- <Identifier> expected

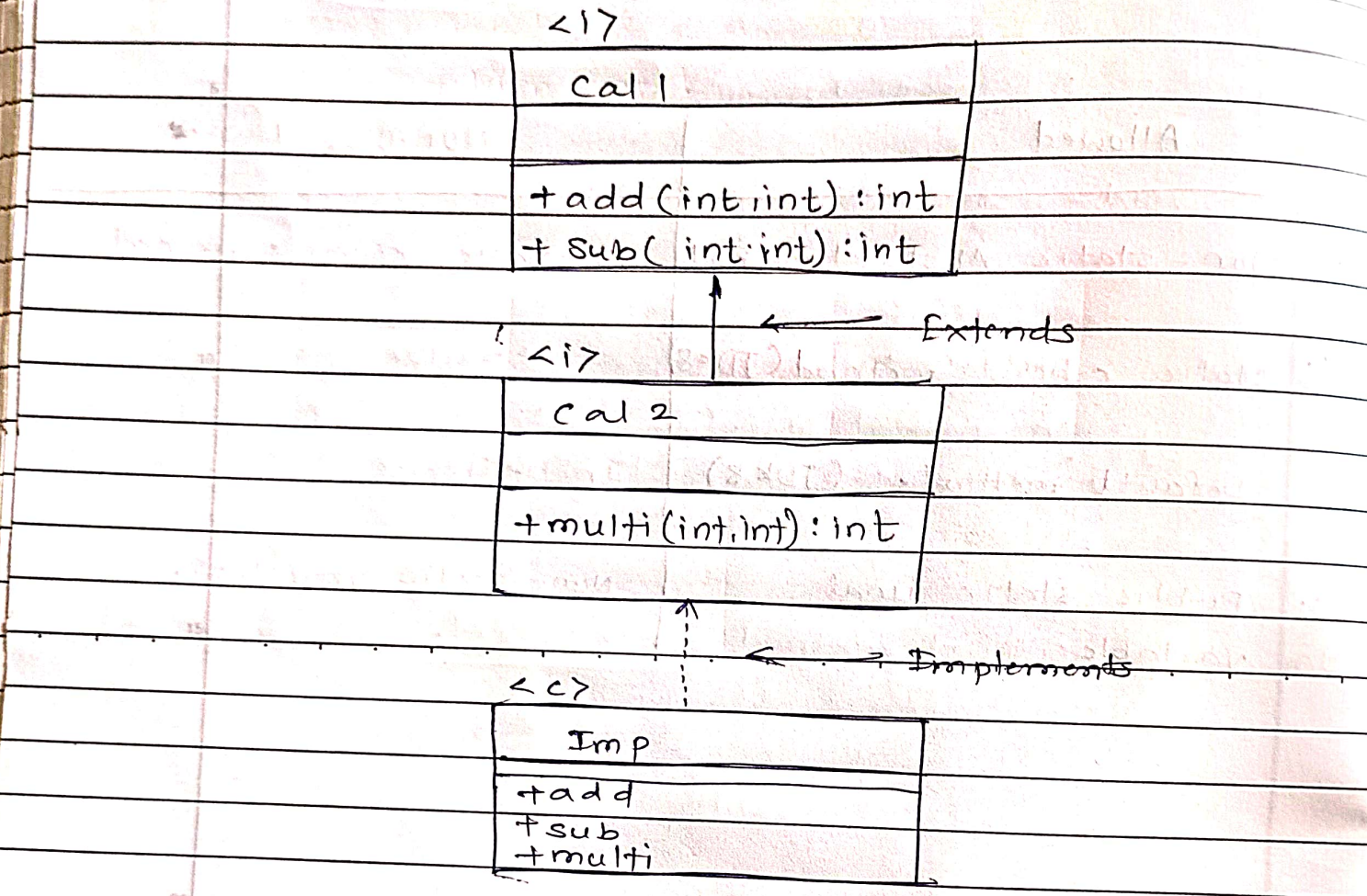
6) In an interface only one type of variable is allowed

```
public interface Example8
{
    int i = 10;
    static int j = 20;

    public static final int x = 1000;
}
```

77 Initializers are not allowed in interface (Non-static variables).

Allowed	Not Allowed
1. Non-static Abstract method	Non-static concrete method
2. static concrete method (JDK 8)	constructor
3. Default methods (JDK 8)	Initializers.
4. Public static final variable	Non-static variables.

Inheritance with interfaces



AbstractCalc1.java.

```
public interface AbstractCalc1
{
    int add (int a, int b);
    int sub ( int a, int b);
}
```

AbstractCalc2.java

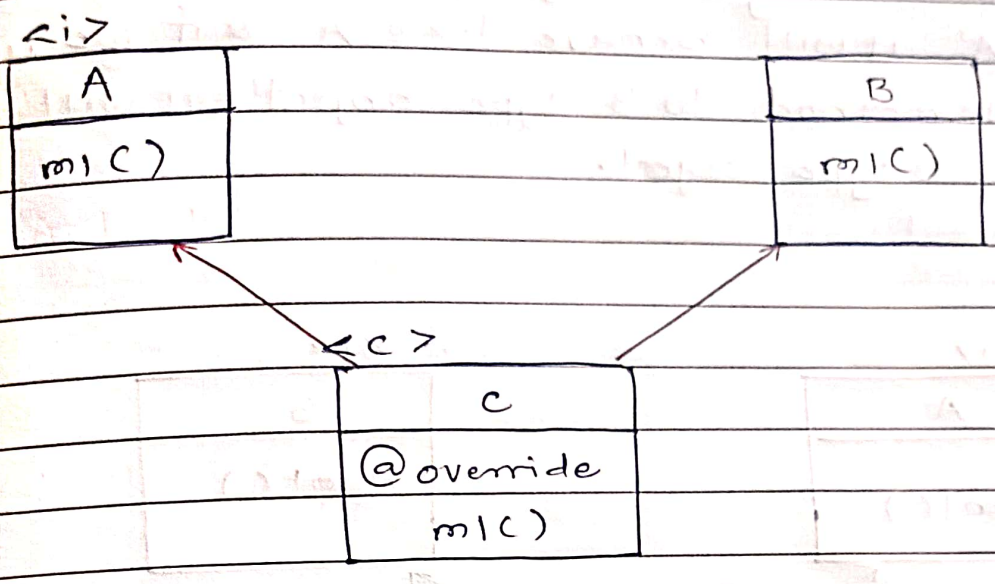
```
public interface AbstractCalc2 extends AbstractCalc1
{
    int multi( int a, int b);
}
```

ImplementationCalc.java

```
public class ImplementationCalc implements AbstractCalc2
{
  @override
  public int add (int a int b)
  {
      return a+b;
  }
  @override
  public int sub( int a, int b)
  {
      return a-b;
  }
```

```
@override
public int multi(int a, int b)
{
    return a*b;
}
}
```

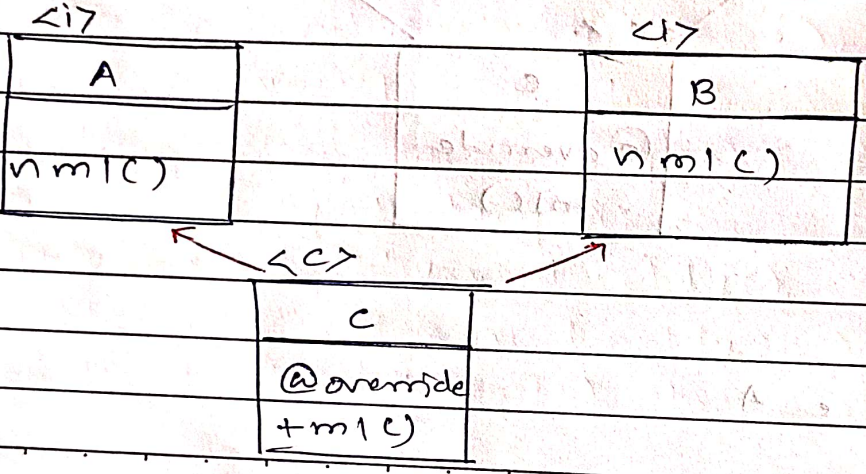
Case 1: Interface with multiple inheritance




```
interface A
{
   void m1();
}
interface B
 {
    void m2()
 {
class C implements A,B
 {
    @override
    public void m1()
    {
       s.o.pln ("from m1()");
    {
```

- After creating an object of A interface if we upcast m1() method to 'A' it will behave like 'A' & if we upcast m1() method to B type object it will behave like B type object.

CASE2:




```
interface A
{
  default void m1()
  {
    S.O.pln("from A");
  }
}
interface B
{
  default void m1()
  {
    S.O.pln("from B");
  }
```

```
class C implements A,B
{
    @override
    public void m1()
    {
        A. super.m1();
        B. super.m1();
    }
}
```

* COLLECTION :-

i> A group of objects called as collection

collection framework :-

i> collection framework is an architecture which provide improvised logic to manipulate collection.

ii> A collection frame work is a group of predefined interfaces abstract classes and concrete implementing classes.

iii> Collection framework provide algorithm to manipulate the collection.

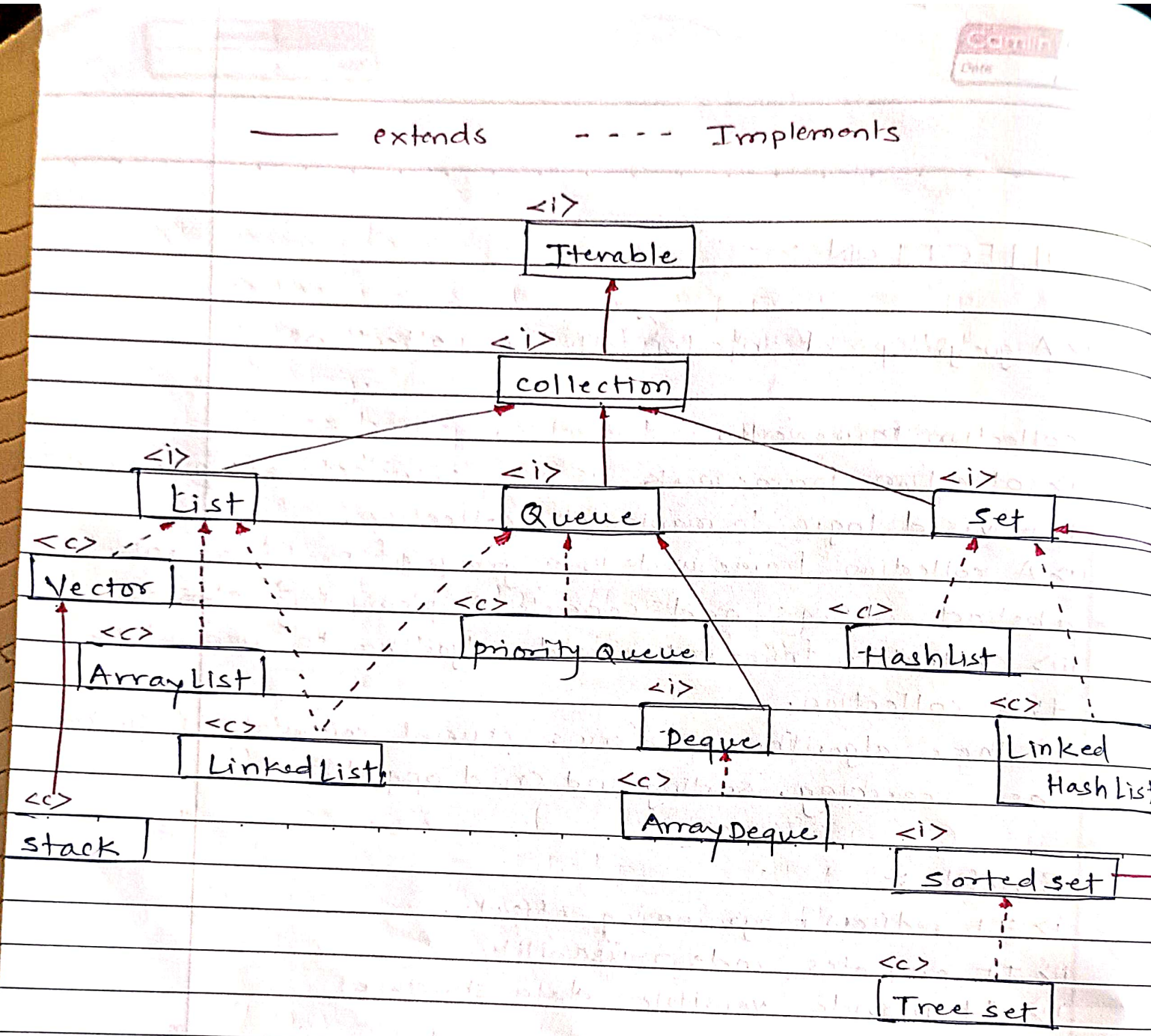
iv> These algorithms performs crucial computations such as searching, sorting, and crud operations.

Benefits :-

i> It reduces programming efforts.

ii> It promotes code reusability.

iii> It provide versitile data structure


—— extends - - - - Implements
<i>
Iterable
<i>
collection
<i>
List
<i>
Queue
<i>
Set
<c>
Vector
<c>
ArrayList
<c>
LinkedList
<c>
stack
<c>
priority Queue
<i>
Deque
<c>
ArrayDeque
<c>
HashList
<c>
Linked
Hash List
<i>
Sorted set
<c>
Tree set

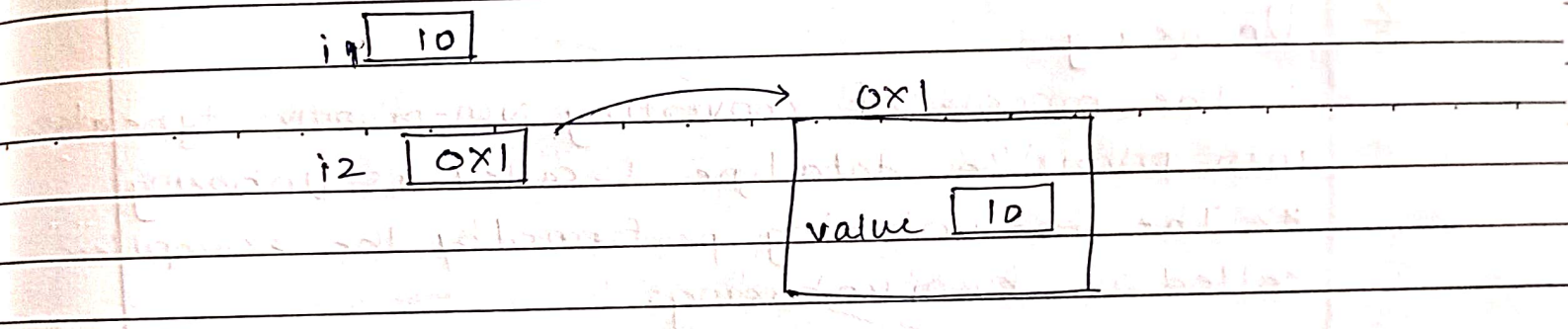
WRAPPER CLASSES.

i> In collection framework premitives are not allowed

ii> Wrapper classes provide a way through which we can convert premitive types into non-premitive types or vice-versa

Ex.

```
int i1 = 10;
Integer i2 = 10;          // autoboxing
Integer i3 = new Integer(20); // Not allowed.
```



Primitive Datatype		classes.
byte	→	Byte
short	→	Short
int	→	Integer
long	→	Long
float	→	Float
double	→	Double
char	→	Character
boolean	→	Boolean.

* Boxing / Autoboxing.

i) The process of converting primitive type data into non primitive type data is called as boxing.

ii) The boxing performed by the compiler is called as autoboxing.

```
// Boxing
Integer i2 = 10; // autoboxing
Integer i3 = new Integer(20) // boxing.
```

* Unboxing

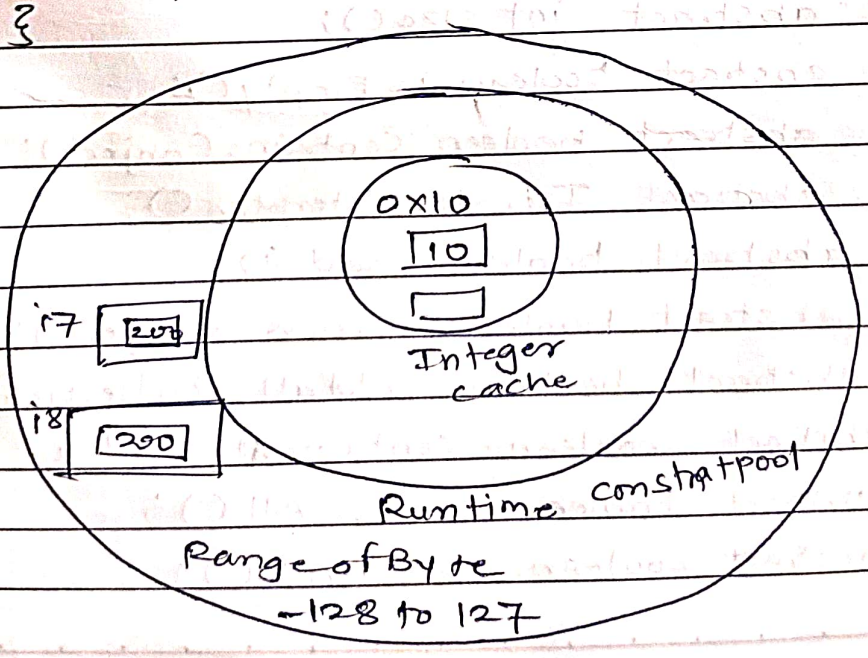
i) The process of converting Non-primitive type data into primitive data type is called as Unboxing

ii) The ~~box~~ unboxing performed by the compiler called as auto unboxing.

```
// Unboxing
int i4 = i2;
int i5 = Integer.valueOf(i3);
```

Note from jdk 9 onwards the constructor for wrapper classes. has been depricated.

```
public class Demo2
{
    public static void main (String [] args)
    {
        int i1 = 10;
        int i2 = 10;
        S.O.pln ("int --->" + (i1==i2)); //true

        Integer i3=10;
        Integer i4=10;
        S.O.pln ("integer>" + (i3=i4)); // true

        int i5 = 200;
        int i6 = 200;
        S.O.pln ( "int ---> + (i5=i6)); //true

        Integer i7 = 200;
        Integer i8 = 200;
        S.O.pln ("Integer ---> + (i7 = i8)); false
    }
}
```



* Iterable :-

i) Iterable is an interface.

ii) It is defined in java.lang package.

iii) Iterable provide the ability to a collection to get iterated.

~~iv) methods of itc~~

Methods of Iterable interface

i) public abstract Iterator iterator();

ii) public default void forEach(Consumer);

iii) public default spliterator spliterator();

* COLLECTION

i) Collection is an interface

ii) It is ~~defind~~ defined in java.util package

important method of collection interface

i) public abstract int size();

ii) public abstract boolean isEmpty();

iii) public abstract boolean contains(object);

Inherited. iv) public abstract Iterator iterator();

v) public abstract boolean add(E);

vi) public abstract boolean remove(object);

add collection into another collection exist in coll or not

vii) public abstract boolean addall(Collection);

remove col. from another col.

viii) public abstract boolean containsAll();

retain all comm elements & del other ele,

ix) public abstract boolean removeAll();

x) public abstract boolean retainAll();

x) public abstract void clear();

* List
i) List is an interface
ii) It is defined in java.util package
iii) It is a ordered collection. List maintain insertion order
iv) List allows duplicate elements

Important method of List interface
i) public ~~boolean~~ abstract boolean replaceAll(Unary operator);
ii) public abstract void sort();
iii) public abstract (object) E get(int);
iv) public abstract (object) E set(int, object);
v) ——— indexOf() → index of element
vi) ——— LastIndexOF()
vii) ——— ListIterator listIterator();
viii) ——— ListIterator listIterator(int);

1) * ArrayList :-
i) It is a concrete class.
ii) Underlying data structure for Arraylist is Growable Array type.
(object type array).

Object[] Arr2 = {10, 'a', 6.258, "hello", new Demo(), (byte) 18, true};

iii) Array List can store duplicate element
iv) Array list allows null.

v) Array list allow random access.

vi) Insertion order is maintain in array list

Ex.

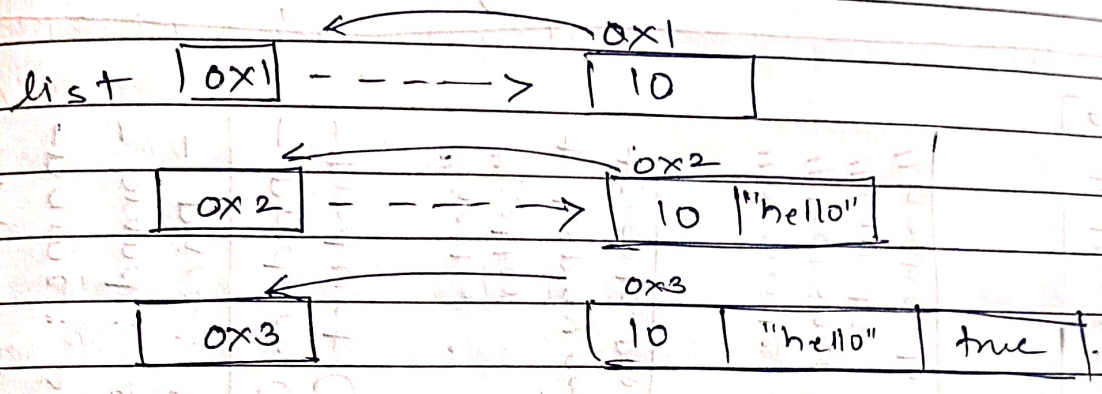
```
import java.util.ArrayList;

public class Demo1
{
    public static void main (String [] args)
    {
        // Creating Array list
        ArrayList list = new ArrayList();

        // add element to a list
        list.add (10);
        list.add ("hello");
        list.add (true);
        list.add (50.258);
        list.add ('a');
        list.add(new demo());

        // print list
        S.o.pln ("\n" + list + "\n");

        // remove element
        S.o.pln (list.remove (50.258));

        // after removal
        S.o.pln ("\n" + list + "\n");
    }
}
```

o/p :-

[10, ~~true~~, "hello", true, 50.258, 'a', new demo());

true

[10, "hello", true, 'a', new demo());



```
import java.util.ArrayList;
public class Demo01
{
    main()
    {
        ArrayList list = new ArrayList();
        // add elements to a list
        list.add(10);
        list.add("hello");
        list.add(true);
        list.add(50.258);
        list.add('a');
        list.add(new Demo01());

    // print list.
    S.O.pln("\n" +list +"\n");
    S.O.pln (list.remove(50.258));
    S.O.pln ("\n" + list + "\n");
    }
}
```

o/p :- [10, hello, true, 50.258, a, Demo@3725 78d]
true
[10, hellow, true, a, Demo@3725 78d]

list.size()
list.remove(true)
list.remove(1)
list.remove(list.indexof(10))

printlist
"
"

remove all duplicate

```
while (list.containce(10))
 {
    list.remove(10);
 }
```

load factor = 0.75

default capacity = 10.

```
public class Example2
{
  main()
  {
  Arraylist list = new ArrayList(50);
   List.add(10);
   List.add('e')
   List.add("hello");
   List.add(true);
   List.add(10.55f);
   ~~List~~S.o.pln(List);
```

10, e, hello, true, 10.55f

```
// creating another list
 ArrayList List2 = new ArrayList();
    List2.add(100);
    List2.add("hii");
    List2.add("java")
```

List.addall(List2);
↓ s.o.pln(list);
10, e, hello, true, 10.55f, 100, hii, java
list.size() ⇒ 8
list.get(7) ⇒ java
List.retainall(List2);
↓↓ ⇒ 100, hi, java

// for accessing
```
for( int i=0; i<list.size(); i++)
 {
    s.o.pln(list.get(i));}
```

// enhanced for loop
```
for(object obj : list)
 {
    s.o.pln(list)
 }
```
syntax: (upcasted)
```
for(Datatype var : Collection)
 {
    s.o.pln(obj);
 }
```
(Datatype: object, var: obj)

// foreach
~~for~~ List.foreach(i → s.o.pln(i))
List.foreach(s.o.plh) // call by reference

Note
You cannot add arrylist<integer>
list.add(2.81);
because of generic datat.

Concurrent Modification Exception.

While we are ~~iteration~~ iterating your collection & if we try to modify collection then we get exception called as Concurrent Modification exception.

* Iterator (It is unidirection goes forward cannot backtrack

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

~~Li~~ List.add (10);
" (20)
" (30)
" (40)
" (50)

Iterator <Integer> i = List.iterator(); (factory method)

→ Iterator provide a way to access individual element of a collection. (It is an interface not a class).

Iterator has two method hasnext & next

hasnext → boolean next → object type

| 10 | 20 | 30 | 40 | 50 |

1 2 3

S.o.pln (i.hasnext()) // true
S.o.pln (i.next()) // 10

| 10 | 20 | 30 | 40 | 50 |

S.o.pln (i.hasnext) // true
S.o.pln (i.next) // 20

| 10 | 20 | 30 | 40 | 50 |

S.o.pln(i.hasnext); // true
S.o.pln(i.next); // 30

| 10 | 20 | 30 | 40 | 50 |

(i.hasnext); // true
(i.next); // 40

| 10 | 20 | 30 | 40 | 50 |

(i.hasnext) // true
(i.next) // 50

| 10 | 20 | 30 | 40 | 50 |

(i.hasnext) // false
(i.next) // exception

```
while (i.hasnext())
{
   s.o.p(i.next())
}
```

Note :- iterator are executive in nature i.e end so i want to use hasnext again so again create object of his iterator I = list.iterator();

* List iterator

→ is a child of iterator
it can move forward direction as well as backward direction.

```
// forward
List Iterator<Integer> f_itr = List.listIterator();
// backward
List Iterator <Integer> b.iter = List listIterator(
                                  list.size());
```

↳ method
hasnext();
next();
hasprevious();
previous();

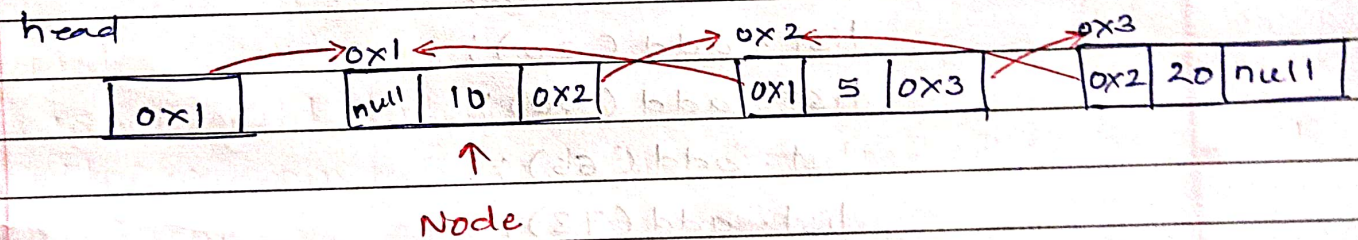
WAJP to print only the string elements from given non-generic arraylist

```
ArrayList List = new ArrayList();
List.add (10);
List.add ('a');
List.add ("hello");
List.add (true);
List.add (10.55f)
for(object obj : list)
  {
    if (obj instanceof (String))
      {
        S.o.pln (object);
      }
  }
for (int i =0 ; i< list.size(); i++)
  {
    object o = list.get(i);
      if (ob.instanceOf(String))
      {
        S.o.pln (list.get (i))
      }
  }
```

* Linked List:-

i) Linked List is a concrete implement class of list interface.
ii) It is defined in java.util package

characteristics of linked list.
i) Null is allowed.
ii) Insertion order is maintained in linked list.
iii) Duplicates are allowed in linked list
iv) Linked list can store Heterogeneous Data.
v) Linked list is growable in nature.
vi) Random access is allowed.
vii) Underlying data structure for linked list is doubly linked list




```
class Node
{
    Node prev;
    int data;
    node next;
}
```

In each linked list reference there is class node data.

Note:- i) Insertion & deletion operation are faster in linked list compared to arraylist

ii) Searching operation are faster in arraylist compare to linked list

```
import java.util.LinkedList;
import java.util.ArrayList;

public class LinkedList1;
{
    public static void main(String[] args)
    {
        ~~Linked list =~~
        LinkedList<Integer> list = new LinkedList<>();

        list.add(10);
        list.add(20);
        list.add(50);
        list.add(66);
        list.add(78);

    s.o.pln(list);

    ArrayList<Integer> list2 = new ArrayList<>(list);

    }
}
```

O/p:-

[10, 20, 50, 66, 78]

* Set :-

i) set is a interface present in java.util package

ii)

characteristics of set :-

i) Duplicates are not allowed in set.

ii) Homogeneous data is allowed only.

Types - a) Hashset
b) linked Hashset
c) Treeset

a) Hashset :-

i) Hash set is concrete implementing class of set interface.

ii) It is ~~is~~ defined in java.util package

iii) Underlying data structure - Hash table

Characteristics of Hashset :-

i) Heterogeneous data is allowed.

ii) Duplicates are not allowed in hash set.

iii) only one null is allowed.

iv) set is also growable (hash set).

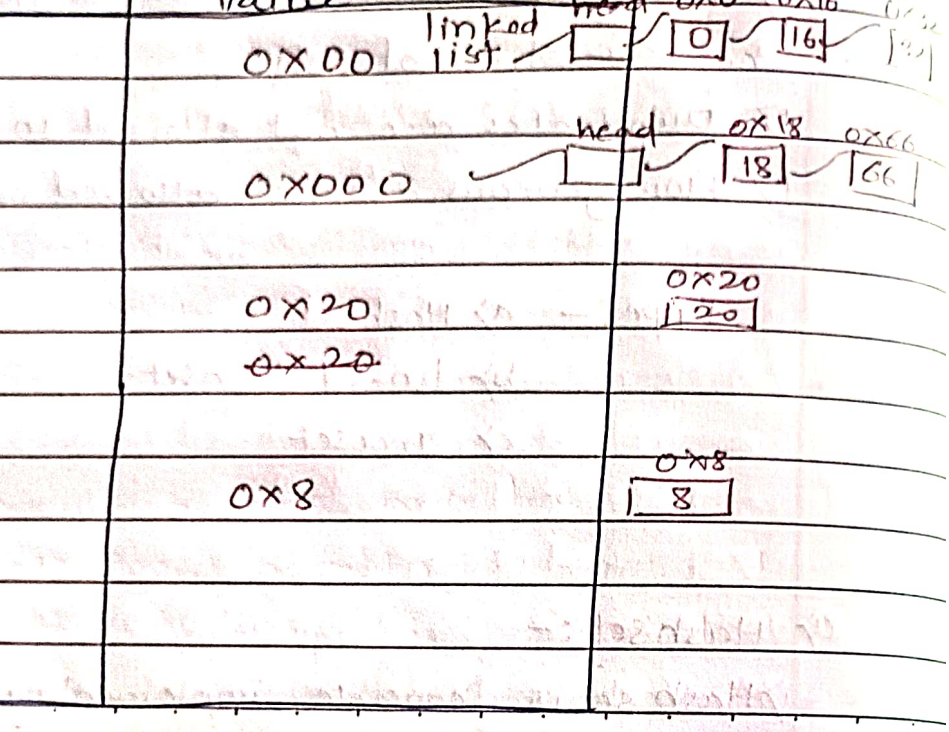
v) The underlying datastructure ~~is~~ for hash set is hash table.

vi) Insertion order is not maintain in hashset (Hashing order is maintain).

No duplicate ↓ Buckets ↓

0%16 →
8%16 →

Key	Value
0	0X00
1	
2	0X000
3	
4	0X20
5	~~0x20~~
:	
8	0X8
:	
14	
15	




```
public class Hashset2
{
    public static void main (string [] args)
    {
        Hashset < Integer> set = new Hashset <> ();

        set. add (0);
        Set. add (8);
        set. add (16);
        Set. add (20);
        set. add( 18);
        set. add (32);
        set. add(66);
```

```
set.add (0);
set.add (null);
set.add (null);

s.o.pln (set);
```

o/p :-
[0, 16, 32, null, 18, 66, 20, 8].

Note :- Before adding an element to hashset internally contains method is been called / invoked to check whether the element is already added to the hash set or not.

Note :-

i) Array :-

- Any collection ~~with~~ which start with array →
  underlying datastructure = Array.

ii) Hash :-

- Any collection which starts with hash →
  hashing order is maintained (0 to 1s)

iii) Linked :-

- Any collection which starts with linked →
  a) Insertion order is maintained
  b) underlying datastructure = Linked list

iv) Tree :-

- Any collection which starts with tree →
  a) natural sorting order is maintained.
  b) Underlying data structure = BST.

b) Linked Hash Set:-

i) It is a concrete implementing class of Set interface.

ii) Underlying data structure is Linked list & hashtable.

Characteristics of Linked Hashset:-

i) Insertion order is maintain in linked hashset.

ii) No duplicates are allowed in

iii) It can store heterogeneous data (Generics is applicable).

Note:-

- While inserting custumize obj, particularly in set type collection, we must override equals methods & hashcode.
- For checking duplicate elements.

Student.java

```
public class Student extends object
{
    private int rollno;
    private string name;

    public student()
    {}
    public student(int rollno, string name)
    {
        setRollno(rollno);
        setName(name);
    }
```

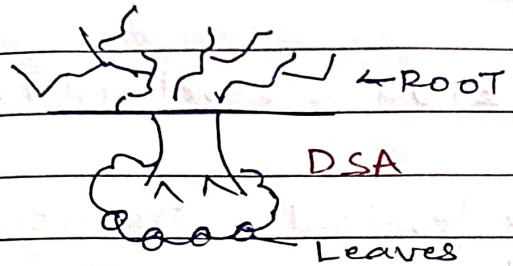
```
publiz int getRoll (~~int roll~~)
{
  ~~this.roll = roll;~~ return roll;
}
publiz ~~Nagetname~~ void int setRoll (~~)~~int roll)
  {
    this. roll = roll;
  }
public int getName()
  {
    return name;
  }
public void setName(string name)
  {{
    this.name = name;
  }

@ override
public string toString()
  {
  return ("+this.roll", "+"this.name")";
  {

@ override
  public int hashcode ()?
  {
    final int prime =31;
    int result=1;
    result =prime * result +roll;
```

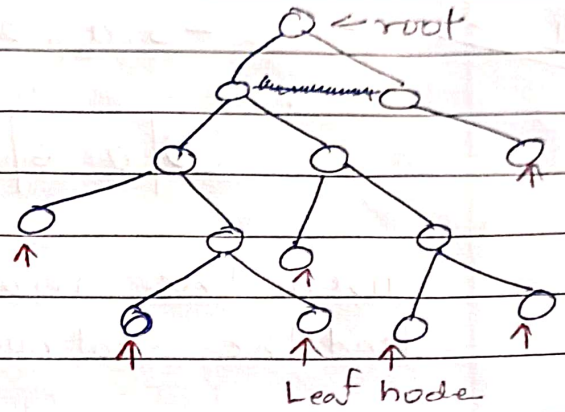
```
result = prime * result + (name == null) ? 0 :
                    name.hashcode());
return result;
}

@ override
public boolean equals(object obj)
    if (this == obj)
      return true;
    if (obj == null)
       return false)
    if (getclass £) != obj.getclass())
       return false;
    student other = (student) obj;
       if (roll != other.roll)
         return false;
       if (name == null){
         if (other.name != null)
           return false;
       }elseif (!name equals (other.name))
         return false;
       return true;
       }
}
```

LinkedHashset2.java

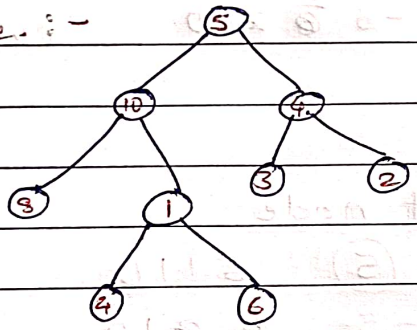
```
import java.util.LinkedHashset;

public class LinkedHashset2:
{
  public static void main (String[] args)
  {
    LinkedHashset <Student> set = new LinkedHashset<

    student s1 = new student (1, "John");
    student s2 = new student (2, "Smith");
    student s3 = new student (3, {"John");
    student s4 = new student (4, "Adams");
    student s5 = new student (1, "John");

    set.add (s1);
    set.add (s2);
    set.add (s3));
    set.add (s4);
    set.add (s5);

  s.o.pln (set);
  s.o.pln (set.contains (s1));

  }
}
```

c) Tree Set :-





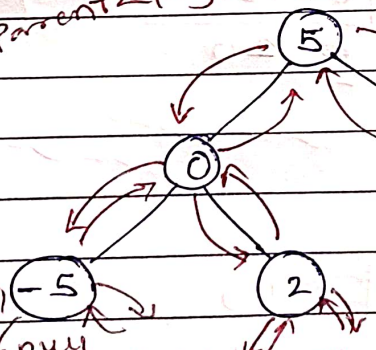
Binary Tree :-



i) In binary tree a parent node ~~only~~ can have atmax 2 child node

Binary search Tree :-

left < Parent < Right



Traversal :-

- BFS
  - ↳ Level order

In order → -5 0 2 5 6 10

Class Node
{
Node left
int data
Node Right
}

- DFS
  - ↳ a) PRE order PLR
  - b) IN order LPR
  - c) POST order LRP.

Q -5, 0, 2, (5), 6, 10 Inorder

(5) 0 6 -5 2 10 Level order.

Given the inorder & level order traversal of BST find the Rootnode and create the BST.

-5 0 2 5 6 10 Inorder

5 0 6 -5 2 10 level order

→ step 1 :-

find the root node

-5 0 2 (5) 6 10 In

(5) 0 6 -5 2 10 level

step 2 :-

find the left subtree & right subtree

-5 0 2 (5) 6 10 In

(5) 0 6 -5 2 10 level

step 2.1 :-

finding level

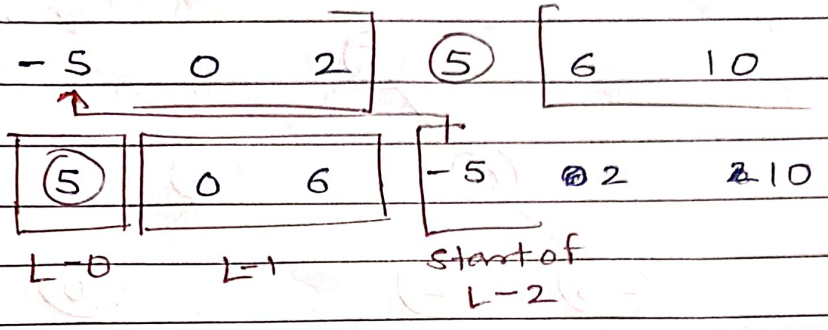
-5 0 2 (5) 6 10 In

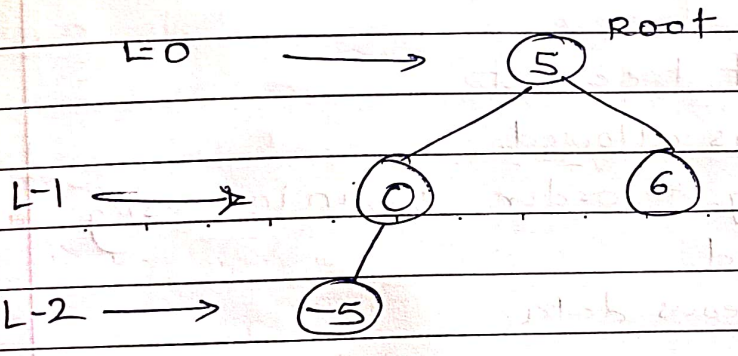
(5) 0 6 -5 2 10 level.

L-0 L-1

step 3 :-

Check starting of level 2 & locate the node in left sub tree or right subtree in Inorder

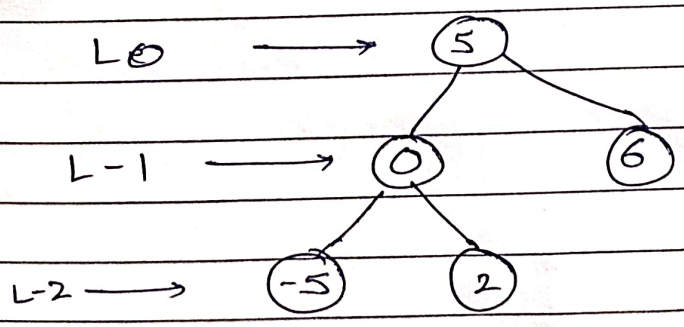




- BST follows one rule left < parent < right

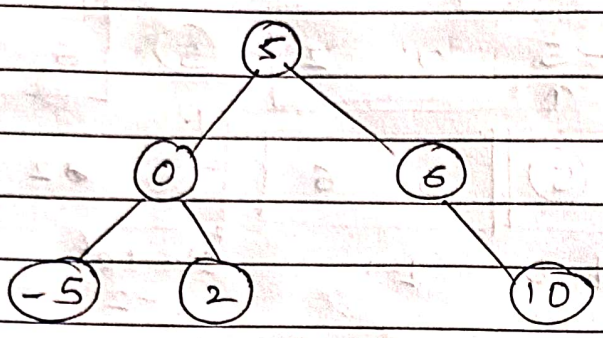
step 4 :-

For node 2. It reside in left sub tree and also it is at on level 2 of level order beti hence it will be placed at right side of node '0'.



step 5 :-

for node 10 It is on right subtree and on level 2 Hence it will be child of node 6



Characteristic of tree set

i) No duplicate is allowed

ii) Natural sorting order is in increasing

iii) No null allowed

iv) No heterogeneous data.

Treeset1.java

```
import java.util.Treeset;

public class Treeset1
{

    public static void main (string [] args)
    {
        Tree set <Integer> set = new Treeset<> ();

        set.add (5);
        set.add (0);
        set.add (6);
        set.add (10);
        set.add (2);
        set.add (-5);

        s.o.pln (set);
        //-5, 0, 2, 5, 6, 10
    }
    public static int [] sortArray (int[] arr)
    {
        Tree set <Integer> set = new Treeset<>();

        for ( int i : arr)
        {
            set.add (i);
        }
```

```
object[] obj = set.toArray();
for (int i = 0; i<arr.length; i++)
  {
    Integer integer = (Integer) obj[i];
        arr[i] = integer;
  }
 }
}
```

Inserting custumize object in Treeset.

i> At the time of insertion we need to check whether the object is smaller than ~~perv~~ previous object

ii> To compare the objects we need compare to method of comparable interface.

a> Comparable Interface

i> Comparable interface is defined in java.lang package.

ii> It has one abstract method

```
public abstract int compareTo(Object o);
```

a. i> How to override compareTo method.

```
public class student extends objects implements
                                    Comparable
```

⋮

```
@ override
public int compareTo (Object o)
{
    student s = (student)o~~obj~~;
    if(this.getRoll()< s.getRoll())
        return -1;
    else if (this.getRoll()==s.getRoll())
        return 0;
    else
        return 1;
}
```

Treeset2.java

```
import java.util.Treeset;

public class Treeset2
{
    public static void main (String [] args)
    Treeset < ~~Integer~~ student> student= new Treeset<>();

    student s4 = new student (4, "DEF");
    student s1= new student (5, "ABC");
    student s3= new student (7, "PQR");
    student s2 = new student (6, "XYZ");
    student s5 = new student (4, "DEF");

    student.add (s1);
```

```
        student.add(s2);
        student.add(s3);
        student.add(s4);
        student.add(s5);

        S.O.pln (students);

    }
}
```

o/p :-

[(4, DEF), (5, ABC), (6, XYZ), (7, PQR)]

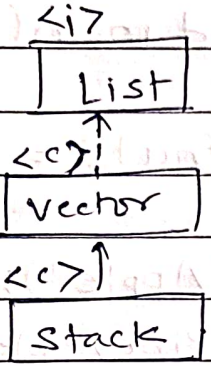
Note :-

For collection.sort(); it will accept only comparible type object.

* STACK :-

i) Stack is a concrete implementing class of list interface.

ii) It is a child class of vector (legacy class).



iii) It is defined in java.util package.

iv) Underlying data structure is linked list.

v) Stack follows LIFO (Last In First out).

Important method in stack

i) PUSH ():
To add element on the top;

ii) POP ();
To remove element from the top;
return type : object type.

iii) Peek ();
It will fetch the element from top but will not remove element

iv) IsEmpty ();
It will check whether stack is empty or not.

```
import java.util.Stack;

public ~~static~~ class Example1{
{
    public static void main(String[] args)
    {
        Stack <String> fruits = new Stack<>();

        fruits.push("Apple");
        fruits.push("PineApple");
        fruits.push("Mango");
        fruits.push("Durian");
        fruits.push("Banana");

        S.O.println(fruits);

        // removing
        S.O.println(fruits.pop());
        S.O.println(fruits);

        // view
        S.O.println(fruits.peek());

    }
}
```

~~Banana~~
Durian
Mango
PineApple
Apple

Pop()

peek ⇒ Durian.

* Queue.

i) Queue is a interface defined in java.util package.

ii) It works on FIFO principle (First in first out);

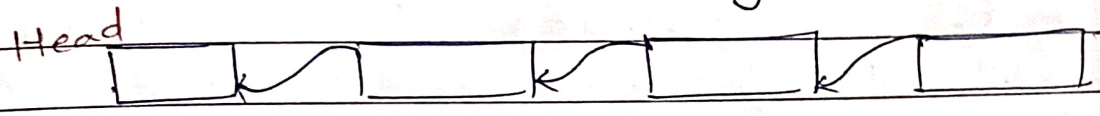
~~iii)~~

Important method of Queue interface

i) offer() / :- To add element at the tail.

ii) Poll () :- To remove the element from Head.

iii) peek () :- To retrieve 1st element of the queue without removing it.




```
import java.util.LinkedList;
import java.util.Queue;

public class Example1{
{
 public static void main (String [] args)
  {
    Queue <String> names = new LinkedList<> ();

    names.offer ("JAI");
    names.offer ("VIRU");
    names.offer ("BASANTI");
    names.offer ("KARAN");
    names.offer ("ARJUN");
```

```
        s.o.pln(names);

        // Dequeue operation.
        s.o.pln(names.poll());
        s.o.pln(names);

        //view
        s.o.pln(names.peek());
        s.o.pln(names);

    }

}
```

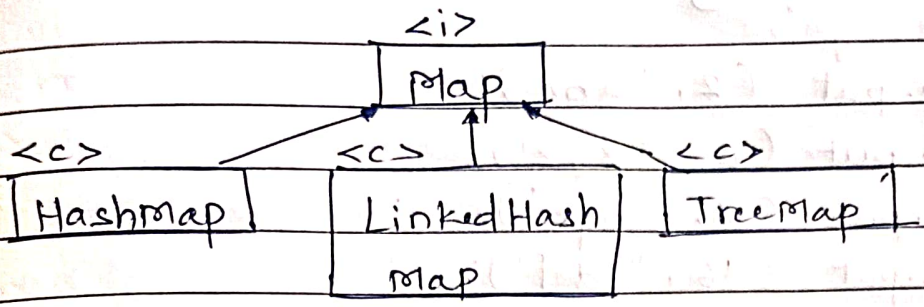
o/p :-

[JAI, VIRU, BASANTI, KARAN, ARJUN]

[VIRU, BASANTI, KARAN, ARJUN]

VIRU.

MAP :-



Entry ⇒ (key, value)

i) Map hierarchy is different hierarchy from collection hierarchy.
ii) In map we insert a key value pair reffered as entry of a map.

i) Hash map :-

i) Hash map defined in java. util package.
ii) Only one null key is allowed.
iii) multiple null values are allowed.
iv) Duplicate keys are not allowed.
v) Duplicate values are allowed.

```
import java.util.HashMap;
                set;

public class HashMap1;
{
   public static void main (string [] args)

Hashmap < Integer, string > map = new HashMap <> ();
```

anaylish Hashmap

```
// add Entry

map.put (2, "abc");
map.put (6, "xyz");
map.put (3, "pqr");
map.put (16, "def");

s.o.pln (map);

// To remove entry
// map.remove (6);

// To fetch value
s.o.pln (map.get (16));

// Iteration of map using key
Set < Integer> entryset = ~~new~~ map.keyset();
for (Integer ~~integer~~ : Entryset)
  {
    s.o.pln (map.get(i));
  }

for (Integer integer : ~~map.getmap~~.keyset())
  {
    s.o.pln (map.get (integer));
  }
```

```
// for fetching all values as collection

Collection<String> List = map.value();

// for fetching all entries as the map.Entry

Set< map.Entry < Integer, String > entry =
                    map.entrySet();

for (map.Entry<Integer, String> entry2 : entry)
    {
        entry2.getKey();
        entry2.getValue();
    }

// other imp method.
    map.IsEmpty();
    map.containsKey(2);
    map.containsValue("abc");
    map.clear();

  }
}
```