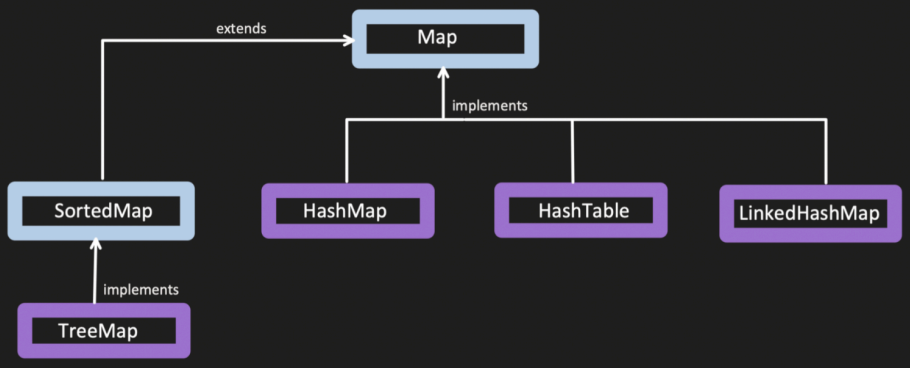
Java Collections Framework Part2

HashMap

Saturday, 16 September 2023 12:26 PM



Map Properties:

- its an interface and its implementatins are:
  - HashMap: do not maintains the order.
  - HashTable: Synchronized version of HashMap
  - LinkedHashMap: Maintains the insertion order.
  - TreeMap: sorts the data internally.
- Object that maps key to values.
- Can not contain duplicate key.

Methods available in Map interface:

Methods available in Map interface:

S.No.	Method Name	Usage
1.	size()	returns the number of key-value mapping present
2.	isEmpty()	returns true: if map contains no key-value mapping else false
3.	containsKey(Object key)	if given key is already present in the map returns true else false
4.	containsValue(Object value)	return true, if one ore more key mapped to the specified value.
5.	get(Object key)	returns the value to which this key is mapped
6.	put(K key, V value)	if map: - already has the same key present : it will overwrite the value now provided - do not have the key present: it will add new key-value mapping.
7.	remove(Object key)	removes the key-value mapping from the map for the specified key.
8.	putAll(Map<K,V> m)	insert the mappings from the specified map to this map.
9.	clear()	removes all the mappings from the map
10.	Set<K> keySet()	returns the Set view of the Key contained in the Map. Set is backed by the Map, so changes in the Map, will be reflected in the Set and vice-versa.
11.	Collection<V> values()	returns the collection view of all the values.
12.	Set<Map.Entry<K,V>> entrySet()	returns the set view of the mappings present in the Map.
13.	putIfAbsent(K key, V value)	if key already exists return the value already associated, else create a new mapping with this key and value.
14.	getOrDefault(key, defaultValue)	if key does not exist or value is null, it returns the default value.
13.	Entry sub-interface	Entry is the sub-interface of the Map. So, its accessed via Map.Entry

Method Name	Usage
getKey	return the key
getValue	returns the value
int hashCode()	obtains the hashCode value
boolean equals(Object o)	used to compare the 2 objects

HashMap:

- can store null key or value (HashTable do not contains null key or value)
- Hash Map do not maintains the insertion order
- Its not thread safe (instead use ConcurrentHashMap or HashTable for thread safe HashMap implemenetation)

```java
public class HashMapExample {

    public static void main (String args[]){

        Map<Integer, String> rollNumberVsNameMap = new HashMap<>();
        rollNumberVsNameMap.put(null, "TEST");
        rollNumberVsNameMap.put(0, null);
        rollNumberVsNameMap.put(1, "A");
        rollNumberVsNameMap.put(2, "B");

        //compute if present
        rollNumberVsNameMap.putIfAbsent(null, "test");
        rollNumberVsNameMap.putIfAbsent(0, "ZERO");
        rollNumberVsNameMap.putIfAbsent(3, "C");

        for(Map.Entry<Integer, String> entryMap : rollNumberVsNameMap.entrySet()){
            Integer key = entryMap.getKey();
            String value = entryMap.getValue();
            System.out.println("Key: " + key + " value: " + value);
        }

        //isEmpty
        System.out.println("isEmpty(): " + rollNumberVsNameMap.isEmpty());

        //size
        System.out.println("size: " + rollNumberVsNameMap.size());

        //containsKey
        System.out.println("containsKey(3): " + rollNumberVsNameMap.containsKey(3));

        //get(key)
        System.out.println("get(1): " + rollNumberVsNameMap.get(1));

        //getOrDefault(key)
        System.out.println("get(9): " + rollNumberVsNameMap.getOrDefault( key: 9,  defaultValue: "default value"));

        //remove(key)
        System.out.println("remove(nul): " + rollNumberVsNameMap.remove( key: null));

        for(Map.Entry<Integer, String> entryMap : rollNumberVsNameMap.entrySet()){
            Integer key = entryMap.getKey();
            String value = entryMap.getValue();
            System.out.println("Key: " + key + " value: " + value);
        }

        //keySet()
        for(Integer key: rollNumberVsNameMap.keySet()){
            System.out.println("Key: " + key);
        }

        //values()
        Collection<String> values = rollNumberVsNameMap.values();
        for(String value : values){
            System.out.println("value: " + value);
        }

    }
}
```

Output:

```
Key: null value: TEST
Key: 0 value: ZERO
Key: 1 value: A
Key: 2 value: B
Key: 3 value: C
isEmpty(): false
size: 5
containsKey(3): true
get(1): A
get(9): default value
remove(nul): TEST
Key: 0 value: ZERO
Key: 1 value: A
Key: 2 value: B
Key: 3 value: C
Key: 0
Key: 1
Key: 2
Key: 3
value: ZERO
value: A
value: B
value: C
```

Time complexity:

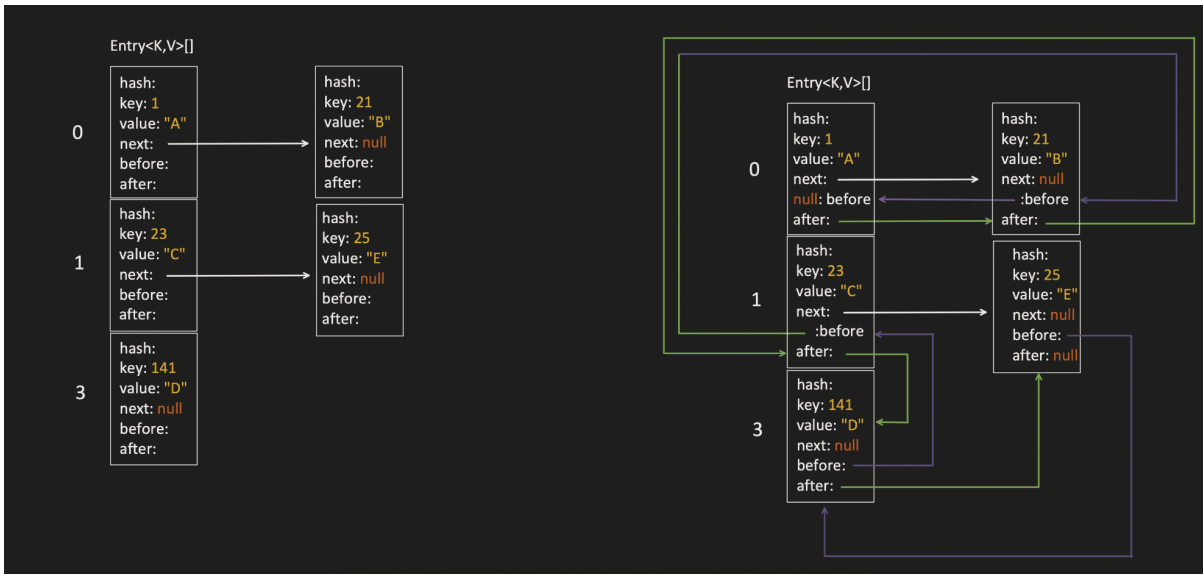
add : amortized O(1)
remove: Amortized O(1)
get: Amortized O(1)

LinkedHashMap:

1. Helps in Maintain Insertion order
   or
   Helps in Maintain Access order

2. Similar to HashMap, but also uses Double LinkedList.

```
map.put(1,"A");
map.put(21,"B");
map.put(23,"C");
map.put(141,"D");
map.put(25,"E");
```




```java
public class LinkedHashMapExample {

    public static void main(String args[]){

        System.out.println("----------below is LInkedHashMap output ------------");

        Map<Integer, String> map = new LinkedHashMap<>();
        map.put(1, "A");
        map.put(21, "B");
        map.put(23, "C");
        map.put(141, "D");
        map.put(25, "E");
        map.forEach((Integer key, String val) -> System.out.println(key + ":" + val));

        System.out.println("----------below is normal hash map output ------------");

        Map<Integer, String> map2 = new HashMap<>();
        map2.put(1, "A");
        map2.put(21, "B");
        map2.put(23, "C");
        map2.put(141, "D");
        map2.put(25, "E");
        for(Map.Entry<Integer, String> entry : map2.entrySet()){
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
}
```

Output:

```
----------below is LInkedHashMap output ------------
1:A
21:B
23:C
141:D
25:E
----------below is normal hash map output ------------
1:A
21:B
23:C
25:E
141:D
```

with ACCESS ORDER = TRUE

```java
public class LinkedHashMapExample {

    public static void main(String args[]){

        System.out.println("----------below is LInkedHashMap output ------------");

        Map<Integer, String> map = new LinkedHashMap<>( initialCapacity: 16, loadFactor: .75F, accessOrder: true);
        map.put(1, "A");
        map.put(21, "B");
        map.put(23, "C");
        map.put(141, "D");
        map.put(25, "E");

        //accessing some data
        map.get(23);
        map.forEach((Integer key, String val) -> System.out.println(key + ":" + val));
    }
}
```

Output:

```
----------below is LInkedHashMap output ------------
1:A
21:B
141:D
25:E
23:C
```

-Time complexity is same as of HashMap : Average O(1)

- Its not thread safe and there is no thread safe version available for this.
  so we have to explicitly make it this collection thread safe like this:

```
Map<Integer,String>map2=Collections.synchronizedMap(newLinkedHashMap<>());
```

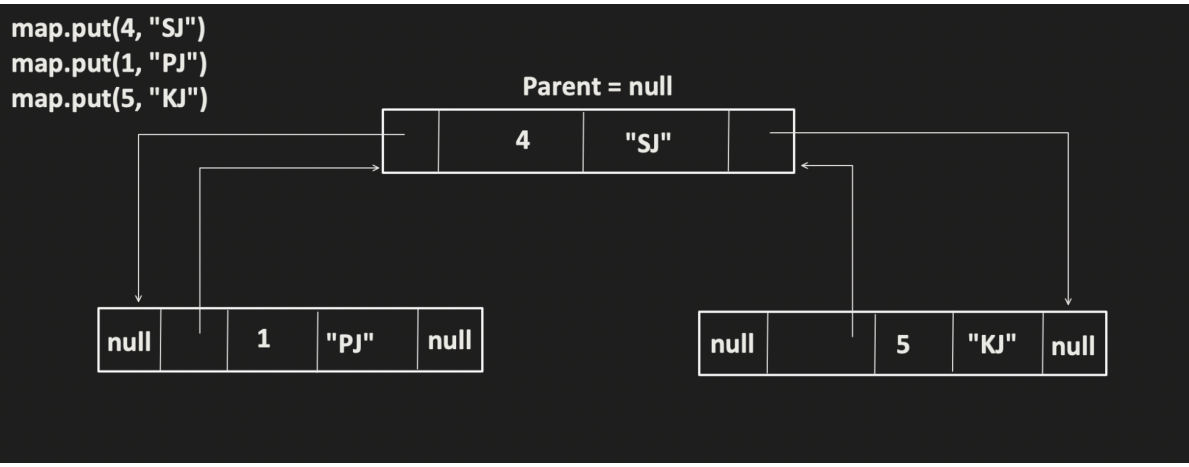
TreeMap:

- Map is Sorted according to its ***natural ordering*** of its Key or by ***Comparator*** provided during map creation.
- Its based on Red-Black tree (Self balancing Binary Search Tree)
- O(logn) time complexity of insert, remove, get operations.

```java
public class TreeMapExample {

    public static void main(String args[]){

        Map<Integer, String> map1 = new TreeMap<>((Integer key1, Integer key2) -> key2 - key1);
        map1.put(21, "SJ");
        map1.put(11, "PJ");
        map1.put(13, "KJ");
        map1.put(5, "TJ");
        //decreasing order
        map1.forEach((Integer key, String value) -> System.out.println(key + ":" + value));

        System.out.println("_____________________________");
        Map<Integer, String> map2 = new TreeMap<>();
        map2.put(21, "SJ");
        map2.put(11, "PJ");
        map2.put(13, "KJ");
        map2.put(5, "TJ");
        //increasing order
        map2.forEach((Integer key, String value) -> System.out.println(key + ":" + value));
    }
}
```

Output:

```
21:SJ
13:KJ
11:PJ
5:TJ
_____________________________
5:TJ
11:PJ
13:KJ
21:SJ
```



Methods Available in SORTEDMAP interface:

S.NO.	METHOD NAME
1.	SortedMap<K,V> headMap(K toKey);
2.	SortedMap<K,V> tailMap(K fromKey);
3.	K firstKey();
4.	K lastKey();

```
public class TreeMapExample {

    public static void main(String args[]){
        SortedMap<Integer, String> map2 = new TreeMap<>();
        map2.put(21, "SJ");
        map2.put(11, "PJ");
        map2.put(13, "KJ");
        map2.put(5, "TJ");

        System.out.println(map2.headMap( toKey: 13));

        System.out.println(map2.tailMap( fromKey: 13));

        System.out.println(map2.firstKey());

        System.out.println(map2.lastKey());
    }
```

Output:

```
{5=TJ, 11=PJ}
{13=KJ, 21=SJ}
5
21
```

Methods Available in NAVIGABLEMAP interface:

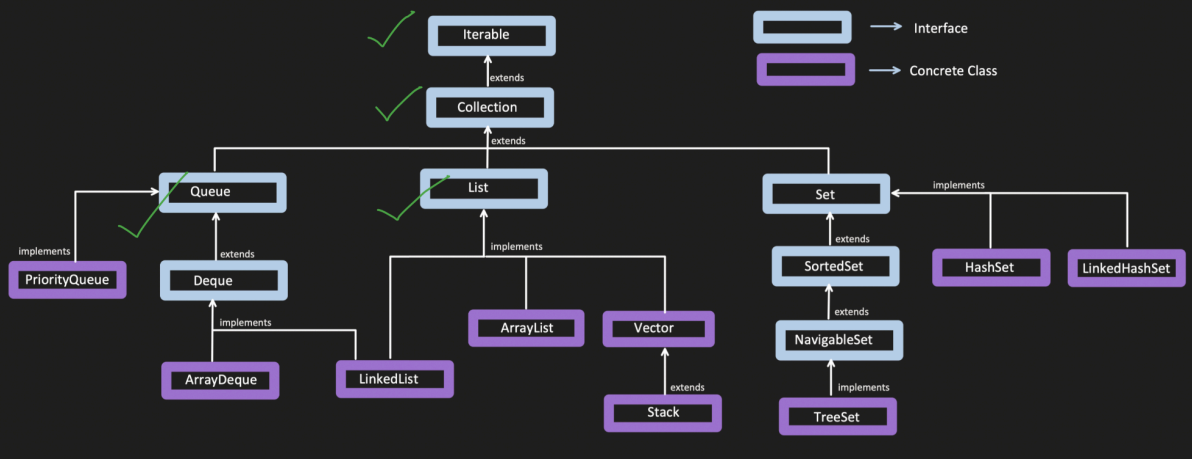
```
NavigableMap<Integer,String>  map = new  LinkedHashMap<>();
map.put(1,"A");
map.put(21,"B");
map.put(23,"C");
map.put(141,"D");
map.put(25,"E");
```

S.NO.	METHOD NAME	USAGE
1.	Map.Entry<K,V> lowerEntry (K key);	map.lowerEntry(23) output: 21,B returns the Entry(Key, value both) node less than 23. If there is no value returns null
2.	K lowerKey(K key);	map.lowerKey(23) output: 21 returns the Key only which is less than 23. If there is no value returns null
3.	Map.Entry<K,V> floorEntry(K key);	map.floorEntry(24) output: 23,C map.floorEntry(23) output: 23,C returns the Entry(Key, value both) which is less or equal than key. If there is no value returns null

4.	K floorKey(K key);	map.floorKey(24) output: 23 map.floorKey(23) output: 23 returns the Key only which is less or equal than key. If there is no value returns null
5.	Map.Entry<K,V> ceilingEntry(K key);	map.ceilingEntry(23) output: 23,C map.ceilingEntry(24) output: 25,E returns the Entry(Key, value both) which is greater or equal than key. If there is no value returns null
6.	K ceilingKey(K key);	map.ceilingKey(23) output: 23 map.ceilingKey(24) output: 25 returns the Key only which is greater or equal than key. If there is no value returns null
7.	Map.Entry<K,V> higherEntry(K key);	map.higherEntry(23) output: 25,E map.higherEntry(25) output: 141, D returns the Entry(Key, value both) which is greater than key. If there is no value returns null

8.	K higherKey(K key);	map.higherKey(23) output: 25 map.higherKey(25) output: 141 returns the Entry(Key, value both) which is greater than key. If there is no value returns null
9.	Map.Entry<K,V> firstEntry();	map.firstEntry(23) output: 1,A retruns the least entry in the Map
10.	Map.Entry<K,V> lastEntry();	map.lastEntry(23) output: 141,D retruns the least entry in the Map
11.	Map.Entry<K,V> pollFirstEntry();	map.pollFirstEntry() output: 1,A remove and return the least element from the map Now 1 would be removed from the map
12.	Map.Entry<K,V> pollLastEntry();	map.pollLastEntry() output: 141,D remove and return the least element from the map. Now 141 would be removed from the map
13.	NavigableMap<K,V> descendingMap();	reverse the map and returns.
14.	NavigableSet<K> navigableKeySet();	[1, 21, 23, 25, 141] (if treeMap is sorted in ascending, keys will come in ascending, else in descending)
15.	NavigableSet<K> descendingKeySet();	[141, 25, 23, 21, 1] (if treeMap is sorted in ascending, keys will come in descending else in ascending)
16.	NavigableMap<K,V>headMap(K toKey, boolean inclusive);	map.headMap(23, true) {1=A, 21=B, 23=C}
17.	NavigableMap<K,V> tailMap(K fromKey, boolean inclusive);	map.tailMap(23, true) {23=C, 25=E, 141=D}

SET:



Few properties of SET:

---

1. Collections of Objects, but it does not contains duplicate value (any only one 'null' value you can insert).
2. Unlike List, Set is not an Ordered Collection, means objects inside set does not follow the insertion order.
3. Unline List, Set can not be accessed via index.

Few questions should have come to our mind:

---

1. What Data Structure is used in Stack internally (as it does not allow duplicate values) ?
2. As order is not gurantee, then what if we want to Sort the Set collection?

We will try to find the answer of both the above quesitons , as we go further.....

What all methods SET Interface contains:

All methods which are declared in Collections interface, generally that only is avaiable in SET interace. No new method specific to SET is added.

Collection Interface Methods as explained previously:

S.NO	METHODS	USAGE
1.	size()	
2.	isEmpty()	
3.	contains()	
4.	toArray()	
5.	add(E element)	returns true, after it insert element in the set only if element is not already present. else if same value is already present, then it returns false.
6.	remove(E element)	
7.	addAll(Collection c)	performs **UNION** of 2 Set Collection. set1 = [12, 11, 33, 4] set2 = [11, 9, 88, 10, 5, 12] set1 + set2 = [12, 11, 33, 4, 9, 88, 10, 5]
8.	removeAll(Collection c)	performs **DIFFERENCE** of 2 Set Collection. Delete the values from set which are present in another set. set1 = [12, 11, 33, 4] set2 = [11, 9, 88, 10, 5, 12] set1 - set2 = [33, 4]
9.	retainAll(Collection c)	performs **Intersection** of 2 Set Collection. Returns element which are present in both set1 = [12, 11, 33, 4] set2 = [11, 9, 88, 10, 5, 12] set1 intersect set2 = [12, 11]
10.	clear()	
11.	equals()	
12.	stream() and parallelStream()	
13.	iterator()	

HashSet:

- Data structure used : HashMap

HashMap<E, Object>map = new HashMap<>();

- During Add method invocation, it stored the element in the key part and in value it stores the dummy object :

map.put(element, new Object())

what if 2 values get the same hash value? how its handled? what is load factor

- No guarantee that the order will remain constant.

- HashSet is not threadSafe. ***newKeySet*** *method* present in ***ConcurrentHashMap*** class is used to create threadSafe Set.

```
public class SetExample {
    public static void main(String args[]){
        Set<Integer> set1 = new HashSet<>();
        set1.add(12);
        set1.add(11);
        set1.add(33);
        set1.add(4);

        Set<Integer> set2 = new HashSet<>();
        set2.add(11);
        set2.add(9);
        set2.add(88);
        set2.add(10);
        set2.add(5);
        set2.add(12);

        //UNION of 2 sets
        set1.addAll(set2);
        System.out.println("after union:");
        set1.forEach((Integer val) -> System.out.println(val));

        //INTERACTION of 2 sets
       set1 = new HashSet<>();
        set1.add(12);
        set1.add(11);
        set1.add(33);
        set1.add(4);

        set2 = new HashSet<>();
        set2.add(11);
        set2.add(9);
        set2.add(88);
        set2.add(10);
        set2.add(5);
        set2.add(12);

        set1.retainAll(set2);
        System.out.println("after Intersection:");
        set1.forEach((Integer val) -> System.out.println(val));

        //DIFFERENCE of 2 sets
        set1 = new HashSet<>();
        set1.add(12);
        set1.add(11);
        set1.add(33);
        set1.add(4);

        set2 = new HashSet<>();
        set2.add(11);
        set2.add(9);
        set2.add(88);
        set2.add(10);
        set2.add(5);
        set2.add(12);

        set1.removeAll(set2);
        System.out.println("after Difference:");
        set1.forEach((Integer val) -> System.out.println(val));
    }
```

Output:

```
after union:
33
4
5
88
9
10
11
12
after Intersection:
11
12
after Difference:
33
4
```

Collection	isThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion
HashSet	No	NO	Yes (only one)	NO	***newKeySet** method present in **ConcurrentHashMap** class*

```java
public class ThreadSafeSet {

    public static void main(String[] args){

        ConcurrentHashMap concurrentHashMap = new ConcurrentHashMap<>();
        Set<Integer> threadSafeSet = concurrentHashMap.newKeySet();
        threadSafeSet.add(1);
        threadSafeSet.add(2);

        Iterator<Integer> iterator = threadSafeSet.iterator();
        while (iterator.hasNext()){

            int val = iterator.next();
            if(val == 1){
                // iterator.remove(); //we can  remove
                threadSafeSet.add(9); //we should be able to add in the set as its thread safe
            }
        }
        threadSafeSet.forEach((Integer val) -> System.out.println(val));
    }
}
```

Without Thread safe, addition of element while iterating

```java
public class HashSetExample {

    public static void main(String args[]){
        Set<Integer> set1 = new HashSet<>();
        set1.add(1);
        set1.add(11);
        set1.add(33);
        set1.add(5);

        Iterator<Integer> iterator = set1.iterator();
        while (iterator.hasNext()){

            int val = iterator.next();
            if(val == 11){
              // iterator.remove(); //we can only remove
                set1.add(7); //we should not be able to add in the set as its not thread safe.
            }
        }
        set1.forEach((Integer val) -> System.out.println(val));
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1449)
    at java.util.HashMap$KeyIterator.next(HashMap.java:1493)
    at HashSetExample.main(HashSetExample.java:75)
```

Time complexity:

add : O(1)
remove: Amortized O(1)
contains: Amortized O(1)

LinkedHashSet:

- Internally it uses: LinkedHashMap
- Maintains the insertion Order of the element
- Its not thread safe:

  Set<Integer>set=Collections.*synchronizedMap*(new LinkedHashSet<>());

```java
public class LinkedHashSetExample {

    public static void main(String args[]){

        Set<Integer> intSet = new LinkedHashSet<>();
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
        intSet.add(5);

       Iterator<Integer> iterable =  intSet.iterator();
       while(iterable.hasNext()){
           int val = iterable.next();
           System.out.println(val);
       }
    }
}
```

Output:

```
2
77
82
63
5
```

TreeSet:

- Internally it uses: TreeMap
- It can not store null value

```java
public class SetExample {

    public static void main(String args[]){

       Set<Integer> treeSet = new TreeSet<>();
        treeSet.add(2);
        treeSet.add(77);
        treeSet.add(82);
        treeSet.add(63);
        treeSet.add(5);
      Iterator<Integer> iterable2 =  treeSet.iterator();

        while(iterable2.hasNext()){
            int val = iterable2.next();
            System.out.println(val);
        }
    }
}
```

Output:

```
2
5
63
77
82
```