Container Classes

Concrete Collections: HashSet, TreeSet

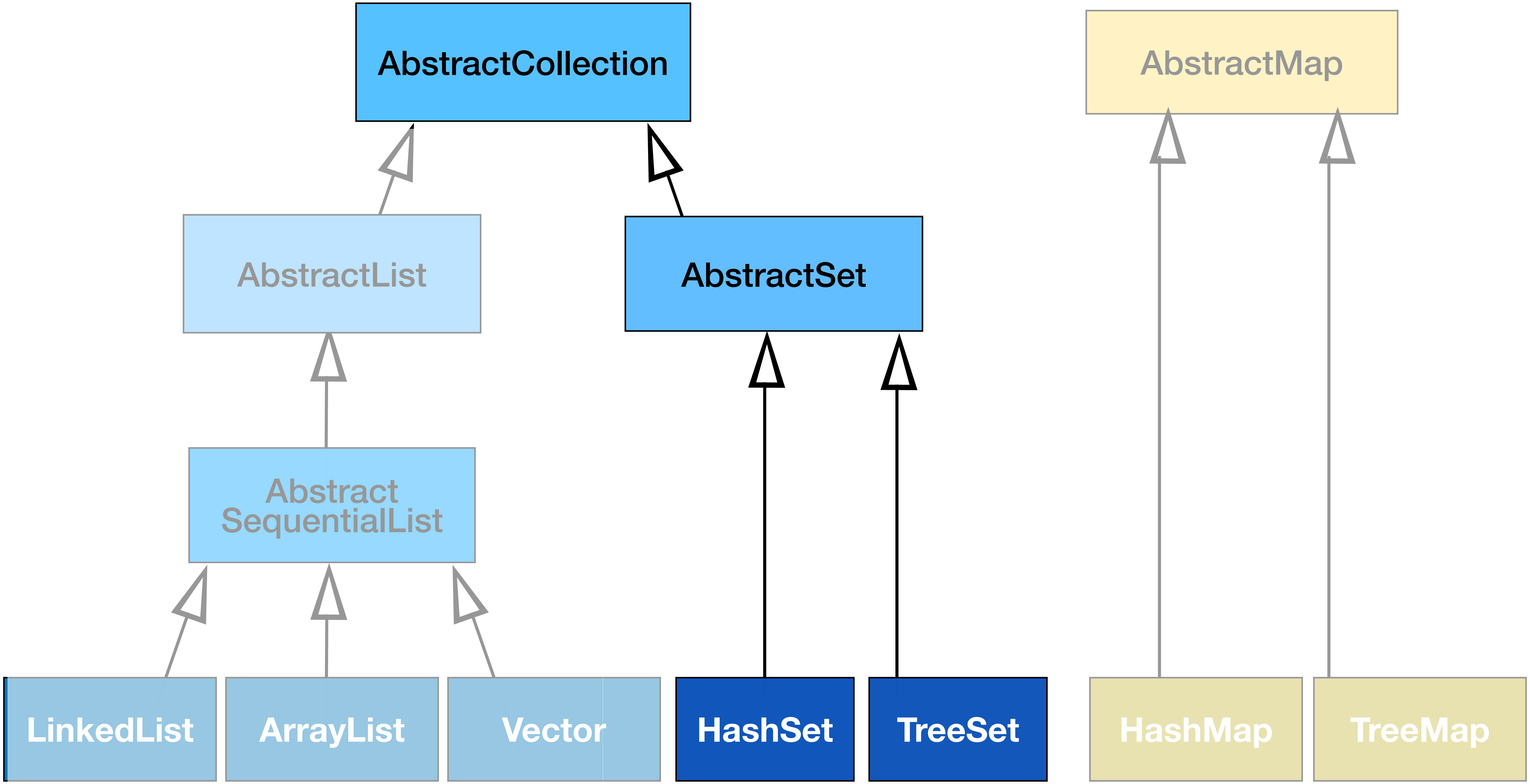
COMP2603
Object Oriented Programming 1

Week 10

Outline

- Concrete Collections
  - HashSet
  - TreeSet
- Comparator Interface

Classes in the Java Collections Framework



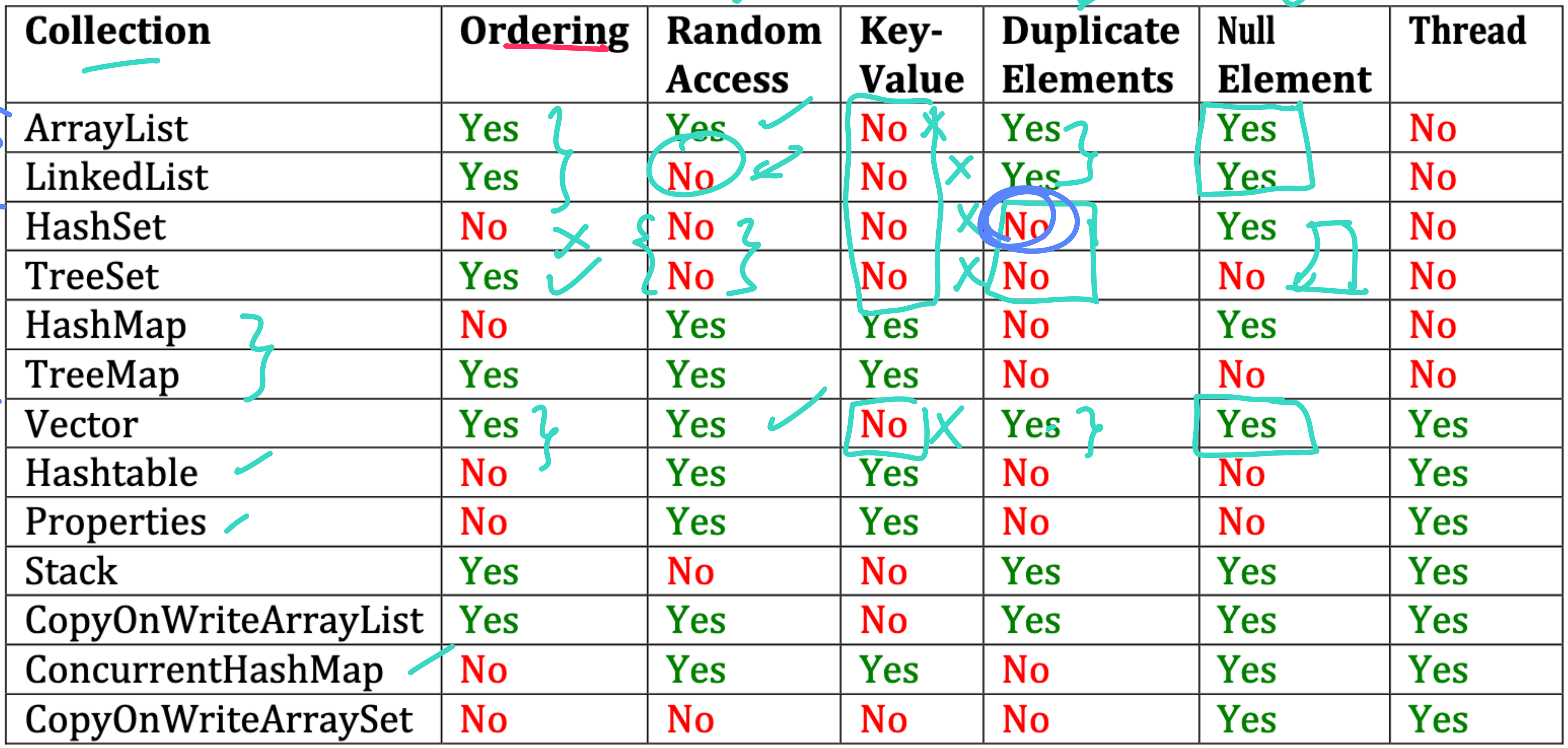
The Set Interface
(same as Collection)

Method	Description
boolean add (E o)	Inserts the object of the specified type into the collection; returns true if the object was added, false otherwise
boolean addAll (Collection c)	Inserts all the objects from the specified collection into the current collection
void clear( ) ✓	Removes all the elements from the collection
boolean contains (Object o)	Returns true id the specified object is present in the colleciton, and false otherwise
boolean isEmpty( ) ✓	Returns true if there are no elements in the collection, and false otherwise
boolean remove(Object o)	Deletes the specified object from the collection
int size( ) ✓	Returns the number of elements currently in the collection

https://docs.oracle.com/javase/7/docs/api/java/util/Set.html

Comparisons

MAP



Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element	Thread
ArrayList	Yes	Yes	No	Yes	Yes	No
LinkedList	Yes	No	No	Yes	Yes	No
HashSet	No	No	No	No	Yes	No
TreeSet	Yes	No	No	No	No	No
HashMap	No	Yes	Yes	No	Yes	No
TreeMap	Yes	Yes	Yes	No	No	No
Vector	Yes	Yes	No	Yes	Yes	Yes
Hashtable	No	Yes	Yes	No	No	Yes
Properties	No	Yes	Yes	No	No	Yes
Stack	Yes	No	No	Yes	Yes	Yes
CopyOnWriteArrayList	Yes	Yes	No	Yes	Yes	Yes
ConcurrentHashMap	No	Yes	Yes	No	Yes	Yes
CopyOnWriteArraySet	No	No	No	No	Yes	Yes

Source: http://www.journaldev.com/1260/java-collections-framework-tutorial

https://cdn.journaldev.com/wp-content/uploads/2013/01/java-collections-framework.pdf

HashSet

1.

This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance).

2

It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.

This class permits the `null` element.

3.

https://docs.oracle.com/javase/7/docs/api/java/util/HashSet

HashSet

Creating and instantiating:

```
1.HashSet<Plant> flowers = new HashSet<>();
```

ST DT

Creating and instantiating with existing Collection: ArrayList

```
2.LinkedList<Plant> morePlants = new LinkedList<>(plants);
3.HashSet<Plant> loadsOfPlants = new HashSet<>(morePlants);
```

https://docs.oracle.com/javase/7/docs/api/java/util/HashSet

HashSet

Adding objects:

```
flowers.add(new Plant(“Petunia”));
```

.size()

↓

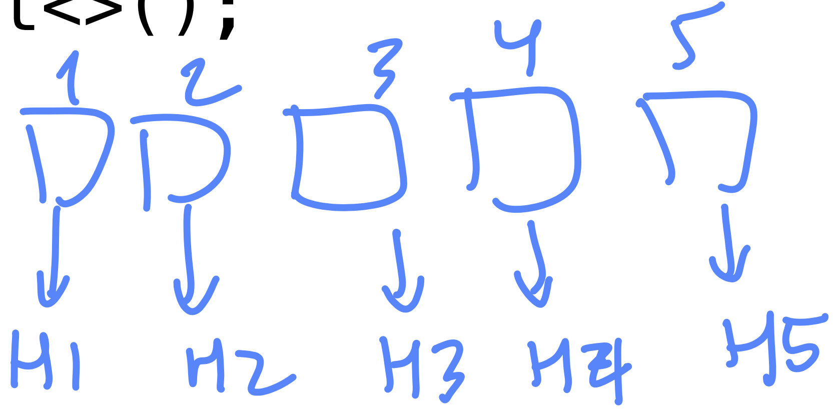
1

The object will be placed in a location based on the value returned by the hashCode( ) method.

https://docs.oracle.com/javase/7/docs/api/java/util/HashSet

Example 1

```
1. HashSet<Plant> flowers = new HashSet<>();
   flowers.add(new Plant("Petunia"));
   flowers.add(new Plant("Petunia"));
   flowers.add(new Plant("Petunia"));
   flowers.add(new Plant("Petunia"));
   flowers.add(new Plant("Petunia"));
   System.out.println(flowers);
```



Output:

```
Petunia
Petunia
Petunia
Petunia
Petunia
```

replace
hashCode()
in
Plant

Each object hashes to a separate location in the HashSet so they are deemed to be different:

```
0 804187522
1 1786349236
2 1958501231
3 374981513
4 1317252669
```

Example 1 - Plant Class

```
public class Plant{
    private String name;
    …
    public int hashCode(){
       return name.hashCode();
    }
    public boolean equals(Object obj){
       if(obj instanceof Plant){
          Plant p = (Plant) obj;
          if(this.name.equals(p.name))
             return true;
       }
       return false;
    }
}
```

Example 2

```
HashSet<Plant> flowers = new HashSet<>();
flowers.add(new Plant(“Petunia”));
flowers.add(new Plant(“Petunia”));
flowers.add(new Plant(“Petunia”));
flowers.add(new Plant(“Petunia”));
flowers.add(new Plant(“Petunia”));
System.out.println(flowers);
```

Output:
Petunia

Each object hashes to the same location in the HashSet so they are deemed to be the same:

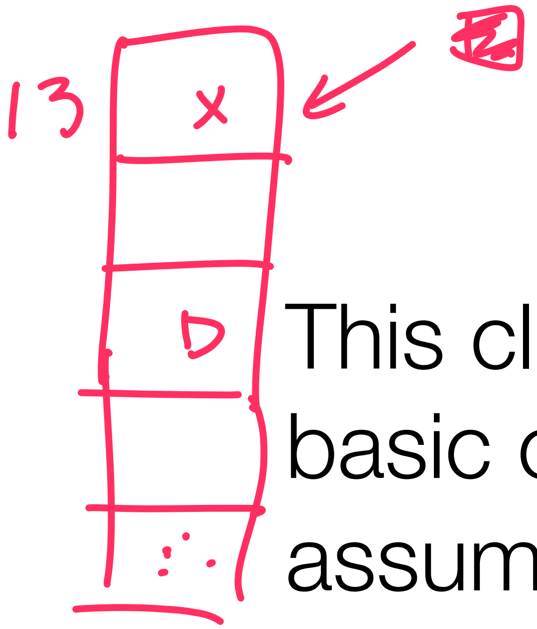
```
0 988117744
1 988117744
2 988117744
3 988117744
4 988117744
```

Example 3 - Plant Class

```
public class Plant{
    private String name;
    …
    public int hashCode(){
       return name.hashCode();
    }
    public boolean equals(Object obj){
       if(obj instanceof Plant){
          Plant p = (Plant) obj;
          return
          ~~if~~(this.name.equals(p.name))
             ~~return true;~~
       ~~} else~~
       ~~return false;~~
    }
    throw new IllegalArgumentException("Not a plant");
}
}
```

T | F | NP

HashSet



This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets.

Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets).

Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

https://docs.oracle.com/javase/7/docs/api/java/util/HashSet

Comparable → compareTo(Object o) → -1, 0, +1
interface

TreeSet

The implementation is based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).

https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

TreeSet

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the `Set` interface. (See `Comparable` or `Comparator` for a precise definition of *consistent with equals*.) compare(Object o1, Object o2) compareTo(Object)

This is so because the `Set` interface is defined in terms of the `equals` operation, but a `TreeSet` instance performs all element comparisons using its `compareTo` (or `compare`) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal.

The behavior of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Set` interface.

https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

Comparable Interface

This interface imposes a total ordering on the objects of each class that implements it.

This ordering is referred to as the class's natural ordering, and the class's **compareTo** method is referred to as its natural comparison method.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

int compareTo(Object obj)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Comparable Interface

The natural ordering for a class C is said to be consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C.

Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Example 4 - Plant Class

Tree Set

```
public class Plant implements Comparable{
    private String name;
    …
    …
    // Compare by Name – ascending A–Z
    public int compareTo(Object obj){
      if(obj instanceof Plant){
        Plant p = (Plant)obj;
         return name.compareTo(p.name);
      }
      throw new ClassCastException("Not a Plant");
    }
}
```

Apple
↓
Zaboca

?
Zaboca
↓
Apple

Example 5 - Plant Class

```
public class Plant implements Comparable{
    private int ID;
    …
    …
    // Compare by ID – ascending order
    public int compareTo(Object obj){
      if(obj instanceof Plant){
        Plant p = (Plant)obj;
         if(this.ID == p.ID) return 0;
         if(this.ID > p.ID) return 1;
         if(this.ID < p.ID) return –1;
      }
      throw new ClassCastException(“Not a Plant”);
    }
}
```

TreeSet

public class Flower {
} x

Creating and instantiating:

```
TreeSet<Plant> trees = new TreeSet<>();
```

Comparable

Creating and instantiating with existing Collection:

```
ArrayList<Plant> hardwoods = new ArrayList<>();
TreeSet<Plant> loadsOfTrees = new TreeSet<>(hardwoods);
```

TreeSet

```
TreeSet<Plant> trees = new TreeSet<>();
```

Adding elements:

```
trees.add(new Plant(“Poui”)); // Plant is Comparable; ok
trees.add(new Plant(“Mora”));
trees.add(new Plant(“Teak”));
trees.add(new Plant(“Pine”));
System.out.println(trees);
```

[Mora , , , ,]

Output:
Mora
Pine
Poui
Teak

The Plants are ordered based on the compareTo( ) method

System.out.println Math.random()

Comparable Interface

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort).

✓class instance

Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a **comparator**.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Comparator Interface

A comparison function, which imposes a *total ordering* on some collection of objects.

Comparators can be passed to a sort method (such as `Collections.sort` or `Arrays.sort`) to allow precise control over the sort order.

Comparators can also be used to control the order of certain data structures (such as `sorted sets` or `sorted maps`), or to provide an ordering for collections of objects that don't have a `natural ordering`.

https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html

Example 6 - Plant Class

ID

```
public class PlantComparator implements Comparator{
    //Compare IDs - descending order
    public int compare(Object obj1, Object obj2){
       if( (obj1 instanceof Plant) &&
           (obj2 instance Plant)){
       1. Plant p1 = (Plant)obj1;
       2. Plant p2 = (Plant)obj2;
       3. if(p1.ID == p2.ID) return 0;
       4. if(p1.ID < p2.ID) return 1;
       5. if(p1.ID > p2.ID) return -1;
       }
       throw new ClassCastException("Not a Plant");
    }
}
```

Example 7 - Plant Class

```
public class PlantComparatorZ–A implements
Comparator{
    //Compare names – Descending order
    public int compare(Object obj1, Object obj2){
      if(  (obj1 instanceof Plant) &&
           (obj2 instance Plant)){
        Plant p1 = (Plant)obj1;
        Plant p2 = (Plant)obj2;
        return p2.compareTo(p1);
      }
      throw new ClassCastException(“Not a Plant”);
    }
}
```

TreeSet & Comparator

```
1. PlantComparatorZ–A ZATrees = new PlantComparator();
2. TreeSet<Plant> trees = new TreeSet<>(ZATrees);
     trees.add(new Plant(“Poui”));
     trees.add(new Plant(“Mora”));
     trees.add(new Plant(“Teak”));
     trees.add(new Plant(“Pine”));
     System.out.println(trees);
```

Output:?

HW

The Plants are ordered based on the Comparator object

Collections Interface

This class consists exclusively of static methods that operate on or return collections.

It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

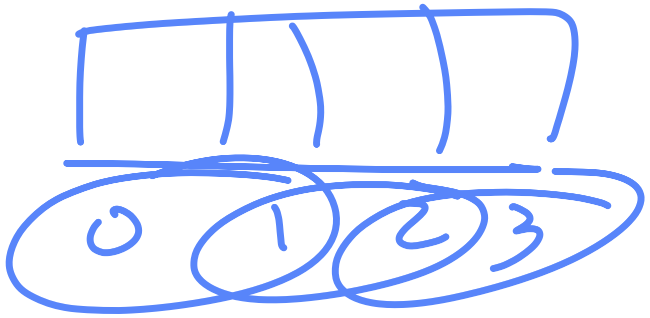
https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

Example - Collections Interface

```
1. ArrayList<Plant> trees2 = new ArrayList(trees);
2. PlantComparatorZ-A ZATrees = new PlantComparator();
3. Collections.sort(trees2, ZATrees);
```

D. P.C

```
1. System.out.println(trees);
   System.out.println(trees2);
```



Output:

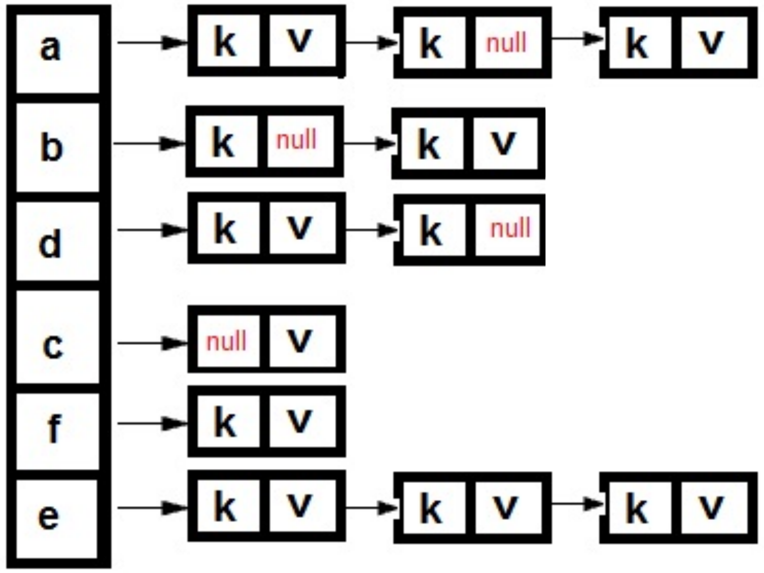
Pine , Poui, Mora, Teak
Teak, Mora, Poui, Pine

paste error!

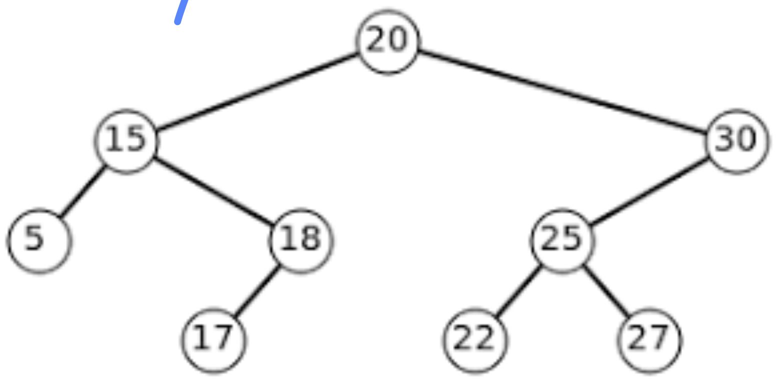
Iterating through a Collection

Different containers store objects in different ways:

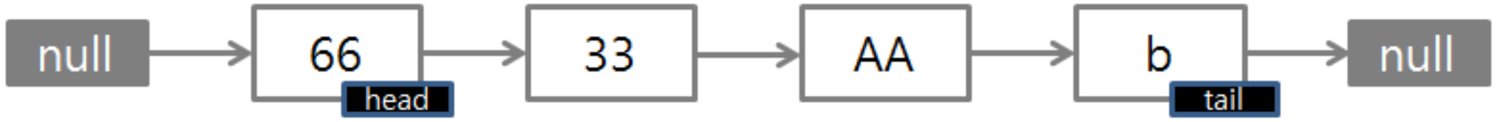
1 **HashMap** nest



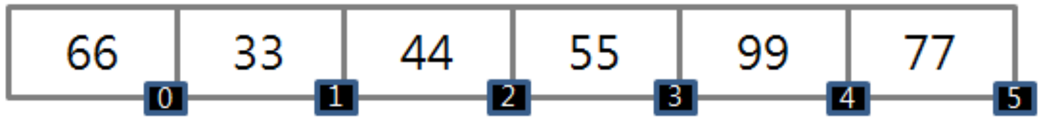
4 **TreeMap**



2 **Linked List**



3 **Array List**



Iterating through a Collection

It is highly desirable to enumerate the objects stored in any container without knowing the internal implementation details of the container.

It would possible therefore to access objects from different containers in a uniform manner.

Iterator Interface

The Iterator interface is typically implemented by a container class, often using an inner class.

The container class will provide a method iterator( ) that returns an instance of this inner class to a client object.

Many of the concrete collection classes in the Java Collections framework provide an iterator( ) method:

- ArrayList
- Vector
- TreeSet

The Iterator Interface

Method	Description
boolean hasNext( )	Returns true if the iteration has more elements.
<T> next( )	Returns the next element in the iteration.
void remove( )	Removes from the underlying collection the last element returned by this iterator

https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

Example - Iterator

```
ArrayList<String> words = new ArrayList<>();
words.add(“Hi”);
words.add(“Hey”);
words.add(“Hello”);

Iterator<String> iter = words.iterator();
while(iter.hasNext()){
    String s = iter.next( );
    System.out.println(s);
}
```

Output:

```
Hi
Hey
Hello
```