

[Courses](#)[Login](#)[Write an Article](#)

## TreeSet in Java

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting AbstractSet class.

Few important features of TreeSet are as follows:

1. TreeSet implements the **SortedSet** interface so duplicate values are not allowed.
2. Objects in a TreeSet are stored in a sorted and ascending order.
3. TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
4. TreeSet does not allow to insert Heterogeneous objects. It will throw `ClassCastException` at Runtime if trying to add heterogeneous objects.
5. TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time.
6. TreeSet is basically implementation of a self-balancing binary search tree like **Red-Black Tree**. Therefore operations like add, remove and search take  $O(\log n)$  time. And operations like printing  $n$  elements in sorted order takes  $O(n)$  time.

### Constructors of TreeSet class:

1. **TreeSet t = new TreeSet();**  
This will create empty TreeSet object in which elements will get stored in default natural sorting order.
2. **TreeSet t = new TreeSet(Comparator comp);**  
This constructor is used when external specification of sorting order of elements



is needed.

### 3. **TreeSet t = new TreeSet(Collection col);**

This constructor is used when any conversion is needed from any Collection object to TreeSet object.

### 4. **TreeSet t = new TreeSet(SortedSet s);**

This constructor is used to convert SortedSet object to TreeSet Object.

## **Synchronized TreeSet:**


The implementation in a TreeSet is not synchronized in a sense that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSortedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
TreeSet ts = new TreeSet();  
Set syncSet = Collections.synchronizedSet(ts);
```

Below program illustrates the basic operation of a TreeSet:

```
// Java program to demonstrate insertions in TreeSet  
import java.util.*;  
  
class TreeSetDemo {  
    public static void main(String[] args)  
    {  
        TreeSet<String> ts1 = new TreeSet<String>();  
  
        // Elements are added using add() method  
        ts1.add("A");  
        ts1.add("B");  
        ts1.add("C");  
  
        // Duplicates will not get insert  
        ts1.add("C");  
  
        // Elements get stored in default natural  
        // Sorting Order(Ascending)  
        System.out.println(ts1);  
    }  
}
```

## **Output:**

 <b>Pixel 3a</b> "The best camera phone", only at ₹39,999* - NDTV Gadgets 360	<b>₹39,999   Flipkart</b>  <b>HDFC BANK</b>   Up to ₹40C We understand your world   HDFC bank	
---	--	---

[A, B, C]

Two things must be kept in mind while creating and adding elements into a TreeSet:

- Firstly, insertion of null into a TreeSet throws *NullPointerException* because while insertion of null, it gets compared to the existing elements and null cannot be compared to any value.
- Secondly, if we are depending on default natural sorting order, compulsory the object should be **homogeneous** and **comparable** otherwise we will get

**RuntimeException:ClassCastException**

```
// Java code to illustrate StringBuffer
// class does not implements
// Comparable interface.
import java.util.*;
class TreeSetDemo {

    public static void main(String[] args)
    {
        TreeSet<StringBuffer> ts = new TreeSet<StringBuffer>();

        // Elements are added using add() method
        ts.add(new StringBuffer("A"));
        ts.add(new StringBuffer("Z"));
        ts.add(new StringBuffer("L"));
        ts.add(new StringBuffer("B"));
        ts.add(new StringBuffer("O"));

        // We will get RunTimeException :ClassCastException
        // As StringBuffer does not implements Comparable interface
        System.out.println(ts);
    }
}
```

#### Runtime Errors:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable
    at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
    at TreeSetDemo.main(File.java:8)
```

#### Output:

No Output

**NOTE:**



1. An object is said to be comparable if and only if the corresponding class implements **Comparable interface**.
2. **String** class and all **Wrapper** classes already implements Comparable interface but StringBuffer class doesn't implements Comparable interface. Hence we got *ClassCastException* in the above example.
3. For an empty tree-set, when trying to insert null as first value, one will get NPE from JDK 7. From 1.7 onwards null is not at all accepted by TreeSet. However upto JDK 6, null will be accepted as first value, but any if insertion of any more values in the TreeSet, will also throw NullPointerException.  
Hence it was considered as bug and thus removed in JDK 7.

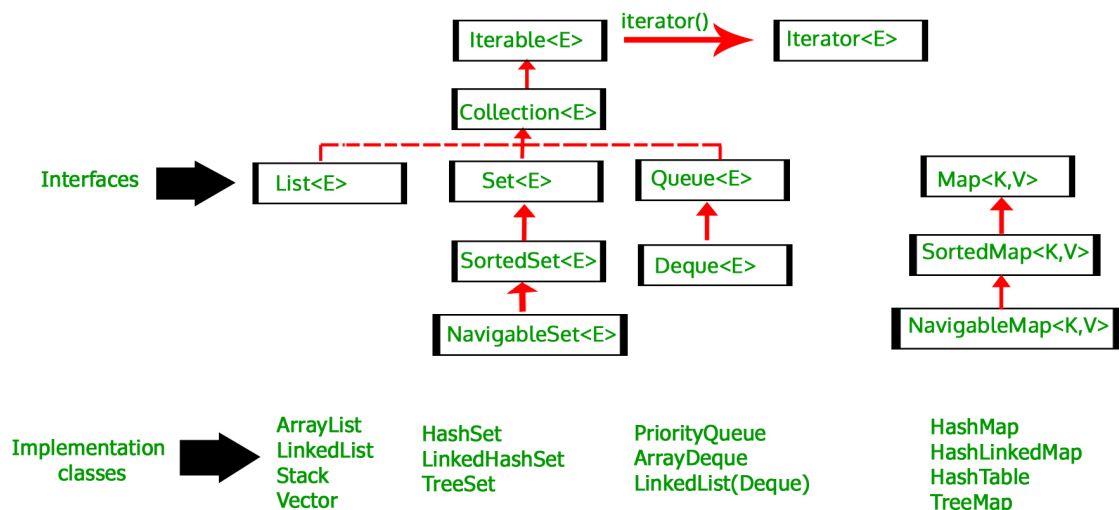
### Methods of TreeSet class:

TreeSet implements **SortedSet** so it has availability of all methods in Collection, **Set** and SortedSet interfaces. Following are the methods in TreeSet interface.

1. **void add(Object o)**: This method will add specified element according to some sorting order in TreeSet. Duplicate entries will not get added.
2. **boolean addAll(Collection c)**: This method will add all elements of specified Collection to the set. Elements in Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet.
3. **void clear()**: This method will remove all the elements.
4. **boolean contains(Object o)**: This method will return true if given element is present in TreeSet else it will return false.
5. **Object first()**: This method will return first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
6. **Object last()**: This method will return last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
7. **SortedSet headSet(Object toElement)**: This method will return elements of TreeSet which are less than the specified element.
8. **SortedSet tailSet(Object fromElement)**: This method will return elements of TreeSet which are greater than or equal to the specified element.
9. **SortedSet subSet(Object fromElement, Object toElement)**: This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.
10. **boolean isEmpty()**: This method is used to return true if this set contains no elements or is empty and false for the opposite case.
11. **Object clone()**: The method is used to return a shallow copy of the set, which is just a simple copied set.
12. **int size()**: This method is used to return the size of the set or the number of elements present in the set.
13. **boolean remove(Object o)**: This method is used to return a specific element from the set.



14. **Iterator iterator()**: Returns an iterator for iterating over the elements of the set.
15. **Comparator comparator()**: This method will return Comparator used to sort elements in TreeSet or it will return null if default natural sorting order is used.
16. **ceiling(E e)**: This method returns the least element in this set greater than or equal to the given element, or null if there is no such element.
17. **descendingIterator()**: This method returns an iterator over the elements in this set in descending order.
18. **descendingSet()**: This method returns a reverse order view of the elements contained in this set.
19. **floor(E e)**: This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
20. **higher(E e)**: This method returns the least element in this set strictly greater than the given element, or null if there is no such element.
21. **lower(E e)**: This method returns the greatest element in this set strictly less than the given element, or null if there is no such element.
22. **pollFirst()**: This method retrieves and removes the first (lowest) element, or returns null if this set is empty.
23. **pollLast()**: This method retrieves and removes the last (highest) element, or returns null if this set is empty.
24. **spliterator()**: This method creates a late-binding and fail-fast Spliterator over the elements in this set.



This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](https://www.geeksforgeeks.org/contribute) or mail your article to [contribute@geeksforgeeks.org](mailto:contribute@geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks.