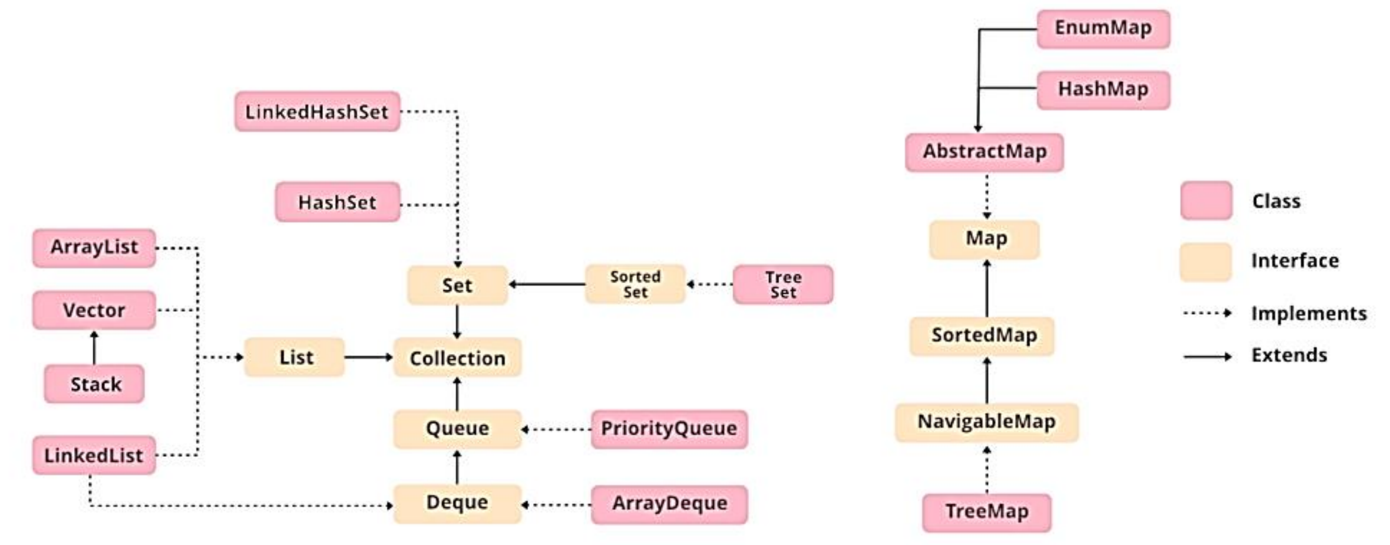
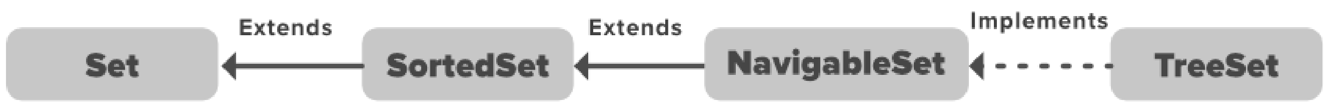
TreeSet in Java

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.



It can clearly be perceived from the above image that the navigable set extends the sorted set interface. Since a set doesn’t retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

Note:

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

How does TreeSet work Internally?
TreeSet is basically an implementation of a self-balancing binary search tree like a Red-Black Tree. Therefore operations like add, remove, and search takes O(log(N)) time. The reason is that in a self-balancing tree, it is made sure that the height of the tree is always O(log(N)) for all the operations. Therefore, this is considered as one of the most efficient data structures in order to store the huge sorted data and perform operations on it. However, operations like printing N elements in the sorted order take O(N) time.
Now let us discuss Synchronized TreeSet prior moving ahead. The implementation of a TreeSet is not synchronized. This means 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 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 the creation time, to prevent accidental unsynchronized access to the set. It can be achieved as shown below as follows:

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

Constructors of TreeSet Class are as follows:
In order to create a TreeSet, we need to create an object of the TreeSet class. The TreeSet class consists of various constructors which allow the possible creation of the TreeSet. The following are the constructors available in this class:

- **TreeSet():** This constructor is used to build an empty TreeSet object in which elements will get stored in default natural sorting order.

Syntax: If we wish to create an empty TreeSet with the name ts, then, it can be created as:

```
TreeSet ts = new TreeSet();
```

- **TreeSet(Comparator):** This constructor is used to build an empty TreeSet object in which elements will need an external specification of the sorting order.

Syntax: If we wish to create an empty TreeSet with the name ts with an external sorting phenomenon, then, it can be created as:

```
TreeSet ts = new TreeSet(Comparator comp);
```

- **TreeSet(Collection):** This constructor is used to build a TreeSet object containing all the elements from the given collection in which elements will get stored in default natural sorting order. In short, this constructor is used when any conversion is needed from any Collection object to TreeSet object.

Syntax: If we wish to create a TreeSet with the name ts, then, it can be created as follows:

```
TreeSet t = new TreeSet(Collection col);
```

- **TreeSet(SortedSet):** This constructor is used to build a TreeSet object containing all the elements from the given sortedset in which elements will get stored in default natural sorting order. In short, this constructor is used to convert the SortedSet object to the TreeSet object.

Syntax: If we wish to create a TreeSet with the name ts, then, it can be created as follows:

```
TreeSet t = new TreeSet(SortedSet s);
```

Methods in TreeSet Class are depicted below in tabular format which later on we will be implementing to showcase in the implementation part.
TreeSet implements SortedSet so it has the availability of all methods in Collection, Set, and SortedSet interfaces. Following are the methods in the Treeset interface. In the table below, the "?" signifies that the method works with any type of object including user-defined objects.

Method	Description
add(Object o)	This method will add the specified element according to the same sorting order mentioned during the creation of the TreeSet. Duplicate entries will not get added.

addAll(Collection c)	This method will add all elements of the specified Collection to the set. Elements in the Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet.
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.
clear()	This method will remove all the elements.
clone()	The method is used to return a shallow copy of the set, which is just a simple copied set.
Comparator comparator()	This method will return the Comparator used to sort elements in TreeSet or it will return null if the default natural sorting order is used.
contains(Object o)	This method will return true if a given element is present in TreeSet else it will return false.
descendingIterator?()	This method returns an iterator over the elements in this set in descending order.
descendingSet?()	This method returns a reverse order view of the elements contained in this set.
first()	This method will return the first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
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.
headSet(Object toElement)	This method will return elements of TreeSet which are less than the specified element.
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.
isEmpty()	This method is used to return true if this set contains no elements or is empty and false for the opposite case.
Iterator iterator()	Returns an iterator for iterating over the elements of the set.
last()	This method will return the last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
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.
pollFirst?()	This method retrieves and removes the first (lowest) element, or returns null if this set is empty.
pollLast?()	This method retrieves and removes the last (highest) element, or returns null if this set is empty.
remove(Object o)	This method is used to return a specific element from the set.
size()	This method is used to return the size of the set or the number of elements present in the set.
spliterator()	This method creates a late-binding and fail-fast Spliterator over the elements in this set.
subSet(Object fromElement, Object toElement)	This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.
tailSet(Object fromElement)	This method will return elements of TreeSet which are greater than or equal to the specified element.

Illustration: The following implementation demonstrates how to create and use a TreeSet.

- Java

```
// Java program to Illustrate Working of  TreeSet

// Importing required utility classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating a Set interface with reference to
        // TreeSet
        Set<String> ts1 = new TreeSet<>();

        // 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:

```
[A, B, C]
```

Implementation:
Here we will be performing various operations over the TreeSet object to get familiar with the methods and concepts of TreeSet in java. Let's see how to perform a few frequently used operations on the TreeSet. They are listed as follows:

- Adding elements
- Accessing elements
- Removing elements
- Iterating through elements

Now let us discuss each operation individually one by one later alongside grasping with the help of a clean java program.
Operation 1: Adding Elements
In order to add an element to the TreeSet, we can use the add() method. However, the insertion order is not retained in the TreeSet. Internally, for every element, the values are compared and sorted in ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are not accepted by the TreeSet.
Example

- Java

```
// Java code to Illustrate Addition of Elements to TreeSet

// Importing utility classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating a Set interface with
        // reference to TreeSet class
        // Declaring object of string type
        Set<String> ts = new TreeSet<>();

        // Elements are added using add() method
        ts.add("Geek");
        ts.add("For");
        ts.add("Geeks");

        // Print all elements inside object
        System.out.println(ts);
    }
}
```

Output:

```
[For, Geek, Geeks]
```

Operation 2: Accessing the Elements
After adding the elements, if we wish to access the elements, we can use inbuilt methods like contains(), first(), last(), etc.
Example

- Java

```
// Java code to Illustrate Working of TreeSet by
// Accessing the Element of TreeSet

// Importing utility classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating a NavigableSet object  with
      // reference to TreeSet class
        NavigableSet<String> ts = new TreeSet<>();

        // Elements are added using add() method
        ts.add("Geek");
        ts.add("For");
        ts.add("Geeks");

         // Printing the elements inside the TreeSet object
        System.out.println("Tree Set is " + ts);

        String check = "Geeks";

        // Check if the above string exists in
        // the treeset or not
        System.out.println("Contains " + check + " "
                           + ts.contains(check));

        // Print the first element in
        // the TreeSet
        System.out.println("First Value " + ts.first());

        // Print the last element in
        // the TreeSet
        System.out.println("Last Value " + ts.last());

        String val = "Geek";

        // Find the values just greater
        // and smaller than the above string
        System.out.println("Higher " + ts.higher(val));
        System.out.println("Lower " + ts.lower(val));
    }
}
```

Output:

```
Tree Set is [For, Geek, Geeks]
Contains Geeks true
First Value For
Last Value Geeks
Higher Geeks
Lower For
```

Operation 3: Removing the Values
The values can be removed from the TreeSet using the remove() method. There are various other methods that are used to remove the first value or the last value.
Example

- Java

```
// Java Program to Illustrate Removal of Elements
// in a TreeSet

// Importing utility classes
import java.util.*;

// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of NavigableSet
        // with reference to TreeSet class
        // Declaring object of string type
        NavigableSet<String> ts = new TreeSet<>();

        // Elements are added
        // using add() method
        ts.add("Geek");
        ts.add("For");
        ts.add("Geeks");
        ts.add("A");
        ts.add("B");
        ts.add("Z");

        // Print and display initial elements of TreeSet
        System.out.println("Initial TreeSet " + ts);

        // Removing a specific existing element inserted
        // above
        ts.remove("B");

        // Printing the updated TreeSet
        System.out.println("After removing element " + ts);

        // Now removing the first element
        // using pollFirst() method
        ts.pollFirst();

        // Again printing the updated TreeSet
        System.out.println("After removing first " + ts);

        // Removing the last element
        // using pollLast() method
        ts.pollLast();

        // Lastly printing the elements of TreeSet remaining
        // to figure out pollLast() method
        System.out.println("After removing last " + ts);
    }
}
```

Output:

```
Initial TreeSet [A, B, For, Geek, Geeks, Z]
After removing element [A, For, Geek, Geeks, Z]
After removing first [For, Geek, Geeks, Z]
After removing last [For, Geek, Geeks]
```

Operation 4: Iterating through the TreeSet

There are various ways to iterate through the TreeSet. The most famous one is to use the enhanced for loop. and geeks mostly you would be iterating the elements with this approach while practicing questions over TreeSet as this is most frequently used when it comes to tree, maps, and graphs problems.

Example

- Java

```
// Java Program to Illustrate Working of TreeSet

// Importing utility classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of Set with reference to
        // TreeSet class

        // Note: You can refer above media if geek
        // is confused in programs why we are not
        // directly creating TreeSet object
        Set<String> ts = new TreeSet<>();

        // Adding elements in above object
        // using add() method
        ts.add("Geek");
        ts.add("For");
        ts.add("Geeks");
        ts.add("A");
        ts.add("B");
        ts.add("Z");

        // Now we will be using for each loop in order
        // to iterate through the TreeSet
        for (String value : ts)

            // Printing the values inside the object
            System.out.print(value + ", ");

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

Output:

```
A, B, For, Geek, Geeks, Z,
```

Features of a TreeSet:

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. If we are depending on the default natural sorting order, the objects that are being inserted into the tree should be homogeneous and comparable. TreeSet does not allow the insertion of heterogeneous objects. It will throw a classCastException at Runtime if we try to add heterogeneous objects.
5. The TreeSet can only accept generic types which are comparable.
   For example, the StringBuffer class implements the Comparable interface

- Java

```
// Java code to illustrate What if Heterogeneous
// Objects are Inserted

// Importing all utility classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Object creation
        Set<StringBuffer> ts = new TreeSet<>();

        // Adding elements to above object
        // 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"));
        ts.add(new StringBuffer(1));

        // Note: StringBuffer implements Comparable
        // interface

        // Printing the elements
        System.out.println(ts);
    }
}
```

Output

```
[, A, B, L, O, Z]
```