HowToDoInJava

Java TreeSet class

Last Updated: December 26, 2020 By: Lokesh Gupta Java Collections Java TreeSet

Java TreeSet class **extends AbstractSet** and `implements NavigableSet` interface. It is very similar to HashSet class, except it stores the element in **sorted order**.

The sort order is either natural order or by a Comparator provided at treeset creation time, depending on which constructor is used.

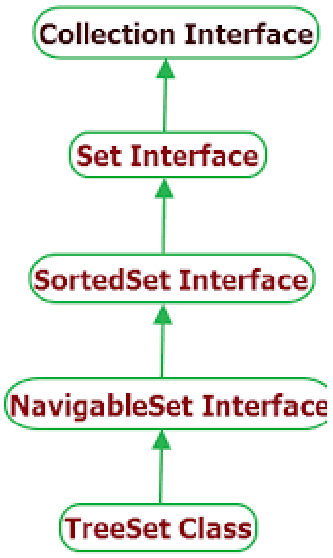
Table of Contents

1. TreeSet Hierarchy

The TreeSet class extends `AbstractSet` class and implements `NavigableSet` interface. The NavigableSet interface extends `SortedSet` in hierarchical order.

```
class TreeSet<E> extends AbstractSet<E>
```

```
        implements NavigableSet<E>, Cloneable, Serializable
{
  //implementation
}
```



TreeSet Hierarchy

2. TreeSet Features

- It extends `AbstractSet` class which extends `AbstractCollection` class.
- It implements `NavigableSet` interface which extends `SortedSet` interface.
- Duplicate values are not allowed in TreeSet.
- NULL is not allowed in TreeSet.
- It is an **ordered collection** which store the elements in sorted order.
- Like HashSet, this class offers constant time performance for the basic operations(add, remove, contains and size).
- TreeSet does not allow to insert heterogeneous objects because it must compare objects to determine sort order.
- TreeSet is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
- Use **Collections.synchronizedSortedSet(new TreeSet())** method to get the synchronized TreeSet.

- The iterators returned by this class's iterator method are **fail-fast** and may throw `ConcurrentModificationException` if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove()` method.
- TreeSet also implements Searlizable and Cloneable interfaces.

3. TreeSet Constructors

The TreeSet has four possible constructors:

1. **TreeSet():** creates a new, empty tree set, sorted according to the natural ordering of its elements.
2. **TreeSet(Comparator c):** creates a new, empty tree set, sorted according to the specified comparator.
3. **TreeSet(SortedSet s):** creates a new tree set containing the same elements and using the same ordering as the specified sorted set.
4. **TreeSet(Collection c):** creates a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.

4. TreeSet Methods

1. **boolean add(E e)** : adds the specified element to the Set if not already present.
2. **Comparator comparator()** : returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
3. **Object first()** : returns the first (lowest) element currently in this set.
4. **Object last()** : returns the last (greatest) element currently in this set.
5. **void clear()** : removes all the elements from the TreeSet.
6. **boolean contains(Object o)** : returns `true` if the TreeSet contains the specified element, othrwise `false`.

7. **boolean isEmpty()** : returns `true` if TreeSet contains no element, otherwise `false`.
8. **int size()** : returns the number of elements in the TreeSet.
9. **Iterator<E> iterator()** : returns an iterator over the elements in this set in **ascending order**.
10. **Iterator<E> descendingIterator()** : returns an iterator over the elements in this set in descending order.
11. **NavigableSet<E> descendingSet()** : returns a reverse order view of the elements contained in this set.
12. **boolean remove(Object o)** : removes the specified element from the TreeSet if it is present and return `true`, else returns `false`.
13. **Object clone()** : returns a shallow copy of the TreeSet.
14. **Spliterator<E> spliterator()** : creates a late-binding and fail-fast Spliterator over the elements in this TreeSet. It has same ordering as treeset provides.

5. TreeSet Example

5.1. TreeSet add, remove, iterator example

```
Java TreeSet Example

//1. Create TreeSet
TreeSet<String> TreeSet = new TreeSet<>();

//2. Add elements to TreeSet
TreeSet.add("A");
TreeSet.add("B");
TreeSet.add("C");
TreeSet.add("D");
TreeSet.add("E");

System.out.println(TreeSet);

//3. Check if element exists
boolean found = TreeSet.contains("A");        //true
System.out.println(found);
```

```
//4. Remove an element
TreeSet.remove("D");

//5. Iterate over values
Iterator<String> itr = TreeSet.iterator();

while(itr.hasNext())
{
    String value = itr.next();

    System.out.println("Value: " + value);
}
```

Program Output.

```
Console

[A, B, C, D, E]
true
Value: A
Value: B
Value: C
Value: E
```

5.2. Convert TreeSet to Array Example

Java example to convert a TreeSet to array using **toArrray()** method.

```
Java TreeSet Example

TreeSet<String> TreeSet = new TreeSet<>();

TreeSet.add("A");
TreeSet.add("B");
TreeSet.add("C");
TreeSet.add("D");
TreeSet.add("E");

String[] values = new String[TreeSet.size()];

TreeSet.toArray(values);

System.out.println(Arrays.toString(values));
```

Program Output.

```
Console

[A, B, C, D, E]
```

5.3. Convert TreeSet to ArrayList Example

Java example to convert a TreeSet to arraylist using Java 8 stream API.

```
Java TreeSet Example

TreeSet<String> TreeSet = new TreeSet<>();

TreeSet.add("A");
TreeSet.add("B");
TreeSet.add("C");
TreeSet.add("D");
TreeSet.add("E");

List<String> valuesList = TreeSet.stream().collect(Collectors.toList());

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

Program Output.

```
Console

[A, B, C, D, E]
```

6. TreeSet Usecases

TreeSet is very much like HashSet (unique elements) and provides predictable iteration order (sorted). Sorted order can overridden using custom comparator.

TreeSet uses **Red-Black tree** under the hood. So the set could be thought as a dynamic search tree. When you need a structure which is operated read/write frequently and also should keep order, the TreeSet is a good choice.

If you want to keep a collection sorted and you are mostly appending the elements, TreeSet with a Comparator is your best bet.

7. TreeSet Performance

- TreeSet provides guaranteed **log(n)** time cost for the basic operations (add, remove and contains).
- The operations like iterating the elements in sorted order takes **O(n)** time.

8. Conclusion

From above discussion, it is evident that TreeSet is very useful collection class in cases where we want to handle duplicate records in sorted manner. It also provide predictable performance for basic operations.

If sorted order of elements is not needed then it is recommended to use the lighter-weight HashSet and HashMap instead.

Drop me your questions related to **TreeSet in Java** in comments.

Happy Learning !!

Reference:

TreeSet Java Docs

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. [Solved]: javax.xml.bind.JAXBException: class java.util.ArrayList nor any of its super class is known to this context
2. Java TransferQueue – Java LinkedTransferQueue class
3. Java LinkedHashMap class
4. Java TreeMap class
5. Java Hashtable class
6. Java HashSet class
7. Java LinkedHashSet class
8. Java ArrayBlockingQueue class
9. Java CopyOnWriteArrayList class
0. Java CopyOnWriteArraySet class

Email Address

Subscribe

** We do not spam !!*

1 thought on "Java TreeSet class"

Piyush Jaiswal

January 24, 2019 at 11:46 pm

what is the initial capacity, load factor and grow factor for TreeSet?

Reply

Leave a Comment

Name *

Email *

Website

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...

VIDEOGAMER
NEW SOUTH PARK GAME WITHOUT UBISOFT COMING?!
» CLICK HERE «

HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial