CSCI 2120:
Software Design & Development II

UNIT 2: Collections Framework & Generics

TreeSet

Overview

Introduction

A **TreeSet in Java** is another important implementation of the Set interface that is similar to the HashSet class, with one added improvement. It sorts elements in ascending order while HashSet does not maintain any order.

TreeSet implements SortedSet interface. It is a collection for storing a set of unique elements (objects) according to their natural ordering. It creates a sorted collection that uses a tree structure for the storage of elements or objects.

In simple words, elements are kept in sorted, ascending order in the tree set.

Introduction

For example, a set of books might be kept by height or alphabetically by title and author.

Fig: A set of books arranged by height

In TreeSet, access and retrieval of elements is fast because of the tree structure.

Therefore, TreeSet is an excellent choice for quick and fast access to large amounts of sorted data.

The only restriction with using tree set is that we cannot add duplicate elements in the tree set.

TreeSet Class Declaration

TreeSet is a generic class that is declared as:

```
class TreeSet<T> extends AbstractSet<T> implements NavigableSet
```

In this syntax, T defines the type of objects or elements that the set will hold.

Hierarchy of TreeSet class in Java

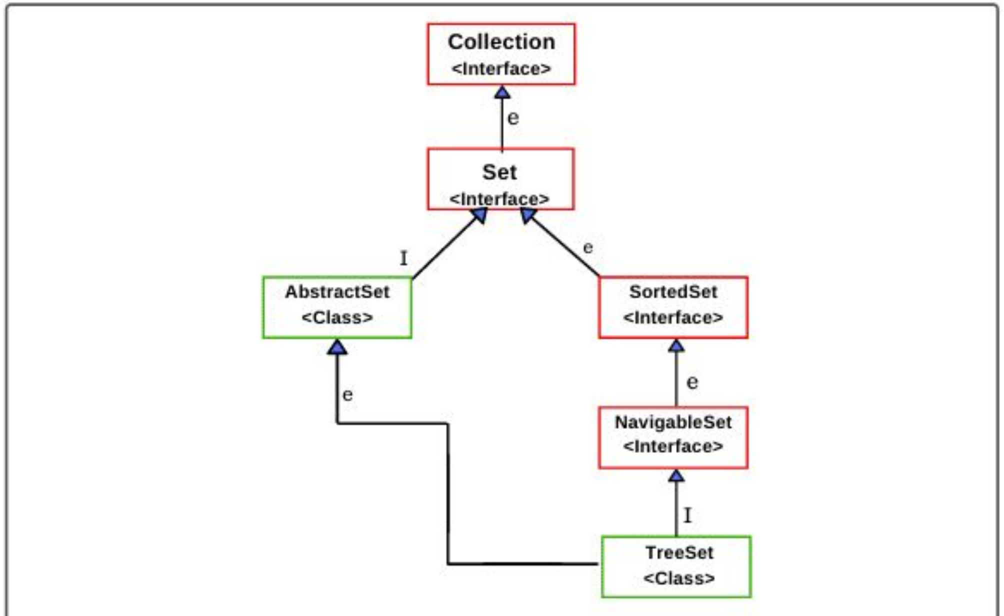


Fig: Java TreeSet Hierarchy Diagram

TreeSet class extends AbstractSet and implements NavigableSet interface.

NavigableSet extends SortedSet to provide navigation methods.

Features of TreeSet

The important features of TreeSet class:

1. TreeSet contains unique elements similar to the HashSet. It does not insert duplicate elements.

2. The access and retrieval times are quite fast.

3. TreeSet does not allow inserting null element.

4. TreeSet class is non-synchronized. That means it is not thread-safe.

5. TreeSet maintains the ascending order. When we add elements into the collection in any order, the values are automatically presented in sorted, ascending order.

6. Java TreeSet internally uses a TreeMap for storing elements.

Constructors of TreeSet

1. TreeSet( ): This default constructor creates an empty TreeSet that will be sorted in ascending order according to the natural order of its elements.

```
TreeSet<T> tset = new TreeSet<>();
```

Constructors of TreeSet

2. TreeSet(Collection c): It creates a tree set and adds the elements from a collection c according to the natural ordering of its elements. All the elements added into the new set must implement Comparable interface.

If collection c's elements do not implement Comparable interface or are not mutually comparable, this constructor throws ClassCastException.

If collection c contains null reference, this constructor throws NullPointerException

```
TreeSet<T> tset = new TreeSet<T>(Collection c);
```

Constructors of TreeSet

3. TreeSet(Comparator comp): This constructor creates an empty tree set that will be sorted according to the comparator specified by comp.

All elements in the tree set are compared mutually by the specified comparator. A comparator is an interface that is used to implement the ordering of data.

```
TreeSet<T> tset = new TreeSet<T>(Comparator comp);
```

Constructors of TreeSet

4. TreeSet(SortedSet s): This constructor creates a tree set that contains the elements of sorted set s.

```
TreeSet<T> tset = new TreeSet<T>(SortedSet s);
```

TreeSet Methods

Method	Description
boolean add(Object o)	This method is used to add the specified element to this set if it is not already present.
boolean addAll(Collection c)	This method is used to add all the elements of the specified collection to the set.
void clear()	It is used to remove all the elements from the set.
boolean isEmpty()	This method is used to check that the set has elements or not. It returns true if the set contains no elements otherwise returns false.
boolean remove(Object o)	This method is used to remove the specified element from the set if it is present.
boolean contains(Object o)	It returns true if the set has the specified element otherwise returns false.

TreeSet Methods

Method	Description
int size()	This method is used to get the total number of elements in the set.
Iterator iterator()	The iterator() method is used to iterate the elements in ascending order.
Spliterator spliterator()	The splititerator() method is used to create a late-binding and fail-fast spliterator over the elements in the tree set.
Object clone()	The clone() method is used to get a shallow copy of this TreeSet instance.
Iterator descendingIterator()	It is used to iterate the elements in descending order.

Methods from SortedSet interface

Since TreeSet implements SortedSet interface, all the methods defined by SortedSet interface can be used while using TreeSet class.

Methods from SortedSet interface

Method	Description
Object first()	It is used to get the first (lowest) element currently in the sorted set.
Object last()	This method returns the last (highest) element currently in the sorted set.
Comparator comparator()	It returns comparator that is used to order elements in the set. If the TreeSet uses the natural ordering, this method returns null.
SortedSet headSet(Object toObject)	This method returns the collection of elements that are less than the specified element.
SortedSet subSet(Object fromElement, Object toElement)	It returns elements from the set that lie between the given range in which fromElement is included and toElement is excluded.
SortedSet tailSet(Object fromElement)	It returns elements from the set that is greater than or equal to the specified element.

Methods from NavigableSet interface

Since Java TreeSet Class implements NavigableSet interface, therefore, all methods of this interface can be used while using tree set class.

Methods from NavigableSet interface

Method	Description
Object ceiling(Object o)	It returns the lowest element from the set equal to or greater than the specified element. If no such element is found, it returns null.
Object floor(Object o)	It returns the greatest element from the set equal to or less than the specified element. If no such element is found, it returns null element.
Object lower(Object o)	This method returns the largest element from the set strictly less than the specified element. If no such element is found, it will return null element.
Object higher(Object o)	This method returns the smallest element from the set strictly greater than the specified element. If no such element is found, it will return null element.
Object pollFirst()	It is used to remove and retrieve the first element in the tree set.

Methods from NavigableSet interface

Method	Description
Object pollLast()	It is used to remove and retrieve the last element in the tree set.
NavigableSet descendingSet()	This method returns elements in reverse order.
NavigableSet headSet(Object toObject, boolean inclusive)	This method returns the collection of elements that are less than or equal to (if, inclusive is true) the specified element.
NavigableSet subSet(Object fromElement, boolean fromInclusive, Object toElement, boolean toInclusive)	This method returns elements from the set that lie between the specified range.
NavigableSet tailSet(Object fromElement, boolean inclusive)	It returns elements from the set that is greater than or equal to (if, inclusive is true) the specified element.

TreeSet Examples

In this section, Let’s take different types of example programs based on the above methods defined by TreeSet, SortedSet, and NavigableSet.

Example 1: Adding Elements

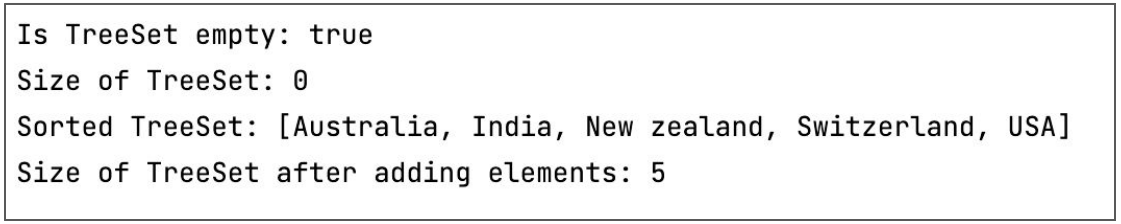
1. **Adding elements:** Let’s create a program where we will perform different operations such as adding, checking the size of TreeSet, and the set is empty or not. Look at the source code to understand better.

Example 1: Adding Elements

```java
import java.util.Set;
import java.util.TreeSet;
public class TreeSetTester1 {
   public static void main(String[] args) {
       // Create a tree set.
       Set<String> ts = new TreeSet<>();

       // Check Set is empty or not.
       boolean empty = ts.isEmpty();
       System.out.println("Is TreeSet empty: " +empty);

       // Checking the size of TreeSet before adding elements into it.
       int size = ts.size();
       System.out.println("Size of TreeSet: " +size);

       // Adding elements into TreeSet.
       ts.add("India"); // ts.size() is 1.
       ts.add("USA"); // ts.size() is 2.
       ts.add("Australia"); // ts.size() is 3.
       ts.add("New zealand"); // ts.size() is 4.
       ts.add("Switzerland"); // ts.size() is 5.

       System.out.println("Sorted TreeSet: " +ts);
       int size2 = ts.size();
       System.out.println("Size of TreeSet after adding elements: " +size2);
   }
}
```

Example 1: Adding Elements

Output:



```
Is TreeSet empty: true
Size of TreeSet: 0
Sorted TreeSet: [Australia, India, New zealand, Switzerland, USA]
Size of TreeSet after adding elements: 5
```

Example 2: Removing Elements

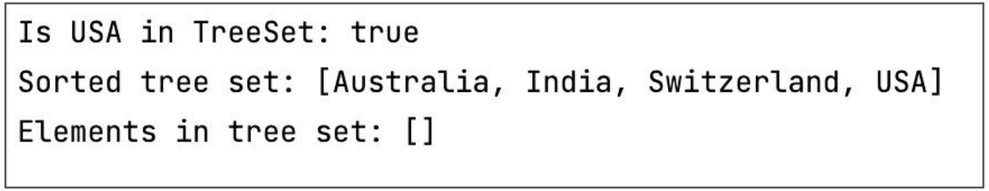
2. **Removing element:** Let’s make a program where we will perform operations like removing an element and checking of a specific element.

Example 2: Removing Elements

```java
import java.util.TreeSet;
public class TreeSetTester2 {
   public static void main(String[] args) {
       TreeSet<String> ts = new TreeSet<>();

       // Add Strings to tree set.
       ts.add("India");
       ts.add("USA");
       ts.add("Australia");
       ts.add("New zealand");
       ts.add("Switzerland");

       // Checking for a specific element in set.
       boolean element = ts.contains("USA");
       System.out.println("Is USA in TreeSet: " +element);

       // Removing element from the tree set.
       ts.remove("New zealand");
       System.out.println("Sorted tree set: " +ts);
       ts.clear();
       System.out.println("Elements in tree set: " +ts);
   }
}
```

Example 2: Removing Elements

Output:



```
Is USA in TreeSet: true
Sorted tree set: [Australia, India, Switzerland, USA]
Elements in tree set: []
```

Example 3: SortedSet methods

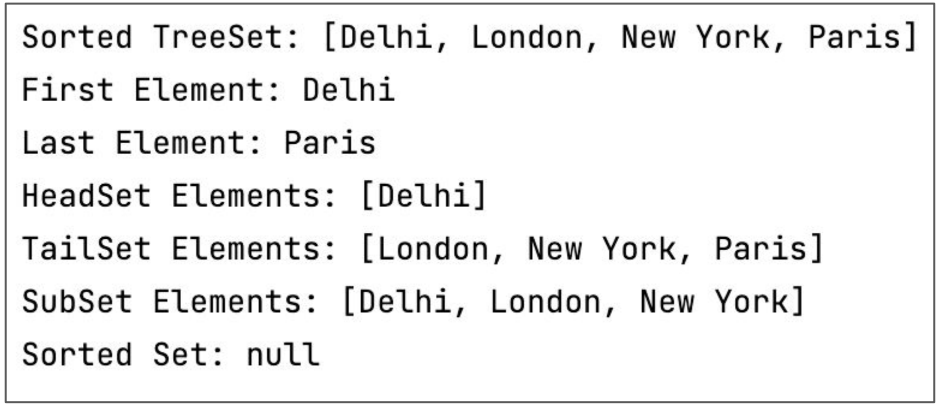
3. **SortedSet methods:** Now we will make a program to perform different operations based on methods defined by the SortedSet interface. Look at the source code

Example 3: SortedSet methods

```java
import java.util.HashSet;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;
public class TreeSetTester3 {
   public static void main(String[] args) {
       Set<String> s = new HashSet<>();
       s.add("Delhi");
       s.add("New York");
       s.add("Paris");
       s.add("London");
       s.add("Delhi"); // Adding duplicate elements.

       TreeSet<String> ts = new TreeSet<>(s);
       System.out.println("Sorted TreeSet: " +ts);

       // Using methods of SortedSet interface.
       System.out.println("First Element: " +ts.first());
       System.out.println("Last Element: " +ts.last());
       System.out.println("HeadSet Elements: " +ts.headSet("London"));
       System.out.println("TailSet Elements: " +ts.tailSet("London"));

       SortedSet<String> subSet = ts.subSet("Delhi", "Paris");
       System.out.println("SubSet Elements: " +subSet);
       System.out.println("Sorted Set: " +ts.comparator()); // It will return null because natural ordering is used.
   }
}
```

Example 3: SortedSet methods

Output:



```
Sorted TreeSet: [Delhi, London, New York, Paris]
First Element: Delhi
Last Element: Paris
HeadSet Elements: [Delhi]
TailSet Elements: [London, New York, Paris]
SubSet Elements: [Delhi, London, New York]
Sorted Set: null
```

Example 4: NavigableSet methods

4. **NavigableSet methods:** Let’s take an example program where we will perform operations based on NavigableSet interface methods.

Example 4: NavigableSet methods

```java
import java.util.TreeSet;
public class TreeSetTester4 {
   public static void main(String[] args) {
       TreeSet<Integer> ts = new TreeSet<>();
       ts.add(25);
       ts.add(80);
       ts.add(05);
       ts.add(100);
       ts.add(90);
       ts.add(200);
       ts.add(300);
       System.out.println("Sorted TreeSet: " +ts);

       // Using methods of NavigableSet interface.
       System.out.println("Largest element less than 100: " +ts.lower(100));
       System.out.println("Smallest element greater than 100: " +ts.higher(100));
       System.out.println("Floor: " +ts.floor(85));
       System.out.println("Ceiling: " +ts.ceiling(10));

       System.out.println(ts.pollFirst()); // Remove and retrieve the first element from the set.
       System.out.println(ts.pollLast()); // Remove and retrieve the last element from the set.
       System.out.println("New Treeset: " +ts);

       System.out.println("HeadSet: " +ts.headSet(90,true));
       System.out.println("SubSet: " +ts.subSet(90, true, 200, true));
   }
}
```

Example 4: NavigableSet methods

Output:

```
Sorted TreeSet: [5, 25, 80, 90, 100, 200, 300]
Largest element less than 100: 90
Smallest element greater than 100: 200
Floor: 80
Ceiling: 25
5
300
New Treeset: [25, 80, 90, 100, 200]
HeadSet: [25, 80, 90]
SubSet: [90, 100, 200]
```

Example 5: Iterate TreeSet

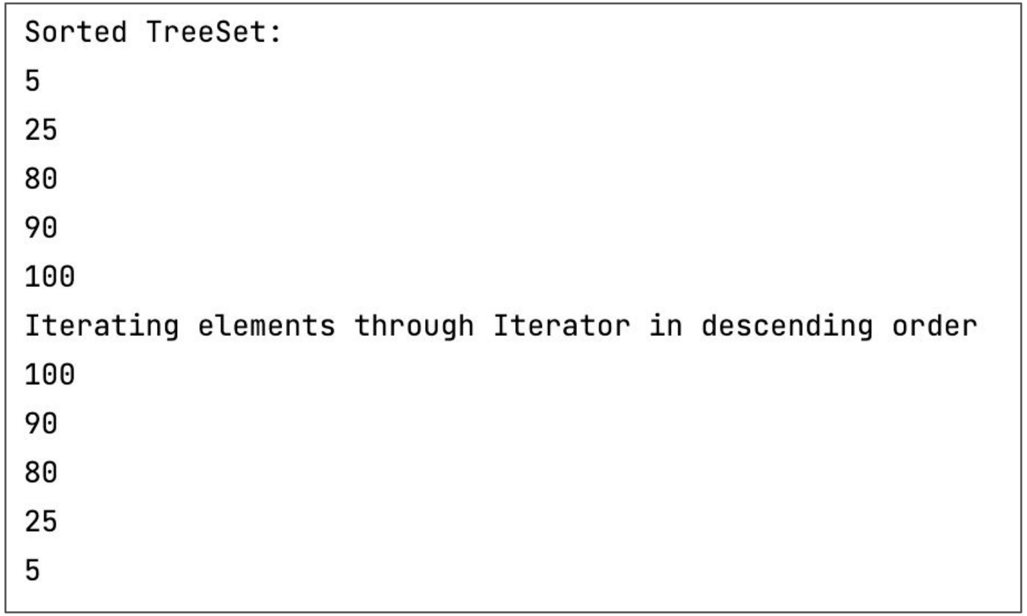
5. **Iterate TreeSet:** In this section, we will iterate elements of TreeSet using iterator() method in ascending and descending order. Look at the following source code

Example 5: Iterate TreeSet

```
import java.util.Iterator;
import java.util.TreeSet;
public class TreeSetTester5 {
   public static void main(String[] args) {
       TreeSet<Integer> ts = new TreeSet<>();
       ts.add(25);
       ts.add(80);
       ts.add(05);
       ts.add(100);
       ts.add(90);

       System.out.println("Sorted TreeSet:");
       // Traversing elements.
       Iterator<Integer> itr = ts.iterator();
       while(itr.hasNext()){
           System.out.println(itr.next());
       }
       System.out.println("Iterating elements through Iterator in descending order");
       Iterator<Integer> it = ts.descendingIterator();
       while(it.hasNext()) {
           System.out.println(it.next());
       }
   }
}
```

Example 5: Iterate TreeSet

Output:



```
Sorted TreeSet:
5
25
80
90
100
Iterating elements through Iterator in descending order
100
90
80
25
5
```

Sort TreeSet

A TreeSet in Java determines the order of elements in either of two ways.

Sort TreeSet

1. A TreeSet sorts the natural ordering of elements when it implements java.lang.Comparable interface. The ordering produced by comparable interface is called natural ordering. The syntax is given below:

```
public interface Comparable {
   public int compareTo(Object o); // Abstract method.

}
```

The compareTo() method of this interface is implemented by TreeSet class to compare the current element with element passed in as a parameter for the order.

If the element argument is less than the current element, the method returns +ve integer, zero if they are equal, or a -ve integer if element argument is greater.

Sort TreeSet

2. TreeSet in Java also determines the order of elements by implementing the Comparator interface. This technique is used when TreeSet class needs to impose a different sorting algorithm regardless of the natural ordering of elements.

The comparator interface provides two methods in which compare() method is more important. The syntax is as follows:

```
public interface Comparator {
   public int compare(Element e1, Element e2); // Abstract method.
   public boolean equals(Element e); // Abstract methods.

}
```

The compare() method of this interface accepts two objects (elements) arguments and returns an integer value that specifies their sort order.

This method returns a -ve value when the first element is less than second element, zero if they are equal, or +ve value if the first element is greater.

When to use TreeSet in Java?

TreeSet can be used when we want unique elements in sorted order.

Which is better to use: HashSet or TreeSet?

If you want to store unique elements in sorted order then use TreeSet, otherwise, use HashSet with no ordering of elements. This is because HashSet is much faster than TreeSet.

END