# Understanding SortedSet and TreeSet

You know, TreeSet does not only implement the Set interface, it also implements the SortedSet and NavigableSet interfaces. Therefore, besides inheriting behaviors of a typical Set, TreeSet also inherits behaviors of SortedSet and NavigableSet. The following picture illustrates the API hierarchy:

Understanding SortedSet:

The key characteristic of a SortedSet is that, it sorts elements according to their natural ordering or by a specified comparator. So considering using a TreeSet when you want a collection that satisfies the following conditions:

* Duplicate elements are not allowed.
* Elements are sorted by their natural ordering (default) or by a specified comparator.

Here’s an example illustrates this characteristic of a SortedSet: SortedSet<Integer> setNumbers = new TreeSet<>();

setNumbers.addAll(Arrays.asList(2, 1, 4, 3, 6, 5, 8, 7, 0, 9)); System.out.println("Sorted Set: " + setNumbers); Output:

Sorted Set: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Here, we add elements of an array list to a TreeSet, and as you can see, the duplicate elements are removed and they are sorted by alphanumeric order (natural ordering of numbers).

In addition to basic collection operations and normal set operations, the SortedSet provides the following types of operations:

* **Range view**: extracts a portion of the set, i.e. a range.
* **Endpoints**: returns the first and the last element in the sorted set.
* **Comparator** **access**: returns the comparator, if an, used to sort the set.

Hence the following interface abstracts a SortedSet:

public interface SortedSet<E> extends Set<E> {

// Range-view

SortedSet **subSet**(E fromElement, E toElement);

SortedSet **headSet**(E toElement);

SortedSet **tailSet**(E fromElement);

// Endpoints

E **first**();

E **last**();

// Comparator access

Comparator<? super E> **comparator**(); }

Let’s look at each type of operation in details.

Range view operations:

* + SortedSet **subSet(E fromElement, E toElement)**: returns a sorted set which is a portion of the set whose elements range from fromElement, inclusive, to toElement, exclusive.

* + SortedSet **headSet(E toElement)**: returns a sorted set which is a portion of the set whose elements are

strictly less than toElement.

* + SortedSet **tailSet(E fromElement)**: returns a sorted set which is a portion of the set whose elements are greater than or equal to fromElement.

Endpoint operations:

* + E **first():** returns the first (lowest) element currently in the set.
* + E **last()**: returns the last (highest) element currently in the set.

Comparator access:

+ **comparator()**: returns the comparator used to order the elements in the set, or null if this set uses the natural ordering of its elements.

Code Examples:

The following code example demonstrates how these operations work with a **TreeSet** implementation: SortedSet<Integer> setNumbers = new TreeSet<>();

setNumbers.addAll(Arrays.asList(2, 1, 4, 3, 6, 5, 8, 7, 0, 9));

System.out.println("Original Set: " + setNumbers);

Integer first = setNumbers.first();

System.out.println("First element: " + first);

Integer last = setNumbers.last();

System.out.println("Last element: " + last);

SortedSet<Integer> subSet = setNumbers.subSet(3, 7);

System.out.println("Sub Set: " + subSet);

SortedSet<Integer> headSet = setNumbers.headSet(5); System.out.println("Head Set: " + headSet);

SortedSet<Integer> tailSet = setNumbers.tailSet(5);

System.out.println("Tail Set: " + tailSet);

Comparator comparator = setNumbers.comparator();

System.out.println("Sorted by natural ordering? " + (comparator == null)); Output:

Original Set: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

First element: 0

Last element: 9

Sub Set: [3, 4, 5, 6]

Head Set: [0, 1, 2, 3, 4]

Tail Set: [5, 6, 7, 8, 9]

Sorted by natural ordering? true

The following code snippet shows how to use a comparator:

class ZtoAComparator implements Comparator<String> { public int compare(String s1, String s2) { return s2.compareTo(s1);

}

}

SortedSet<String> setStrings = new TreeSet<>(new ZtoAComparator()); setStrings.add("Banana"); setStrings.add("Apple"); setStrings.add("Grape"); setStrings.add("Lemon"); setStrings.add("Watermelon");

System.out.println(setStrings); Output:

[Watermelon, Lemon, Grape, Banana, Apple]

As you see, the specified comparator sorts the elements into descending order.

If you use Java 8, use Lambda expression to simplify the comparator class like this:

SortedSet<String> setStrings = new TreeSet<>((s1, s2) -> s2.compareTo(s1));

# Understanding NavigableSet and TreeSet

Understand the NavigabeSet interface in the Java Collections Framework with code examples using TreeSet. Besides Set and SortedSet, TreeSet also implements NavigableSet.

Understanding NavigableSet:

NavigableSet is a sub interface of the SortedSet interface, so it inherits all SortedSet’s behaviors like range view, endpoints and comparator access. In addition, the NavigableSet interface provides navigation methods and descending iterator that allows the elements in the set can be traversed in descending order.

Let’s look at each new method defined by this interface in details.

* lower(E): returns the greatest element which is less than the specified element E, or null if there is no such element.
* floor(E): returns the greatest element which is less than or equal to the given element E.
* ceiling(E): returns the least element which is greater than or equal to the given element E.
* higher(E): returns the least element which is strictly greater than the specified element E.
* descendingSet(): returns a reverse order view of the elements contained in the set.
* descendingIterator(): returns an iterator that allows traversing over elements in the set in descending order.
* pollFirst(): retrieves and removes the first (lowest) element, or returns null if the set is empty. ▪ pollLast(): retrieves and removes the last (highest) element, or returns null if the set is empty.

Furthermore, the NavigableSet interface overloads the methods headSet(), subSet() and tailSet() from the SortedSet interface, which accepts additional arguments describing whether lower or upper bounds are inclusive versus exclusive:

* headSet(E toElement, boolean inclusive)
* subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) ▪ tailSet(E fromElement, boolean inclusive) Now, let’s look at some code examples.

Code Examples:

The following example shows how to obtain a reverse order set from the original one: NavigableSet<Integer> setNumbers1 = new TreeSet<>();

setNumbers1.addAll(Arrays.asList(4, 8, 3, 9, 1, 6, 4, 5, 3, 2, 7, 8, 0));

NavigableSet<Integer> setNumbers2 = setNumbers1.descendingSet();

System.out.println("Set Numbers 1: " + setNumbers1);

System.out.println("Set Numbers 2: " + setNumbers2); Output:

Set Numbers 1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Set Numbers 2: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

The following example illustrates how to obtain the descending iterator from a navigable set: NavigableSet<String> setFruits = new TreeSet<>(); setFruits.add("Banana"); setFruits.add("Apple"); setFruits.add("Orange"); setFruits.add("Grape"); setFruits.add("Mango");

System.out.println("Set Fruits: " + setFruits);

Iterator<String> descIterator = setFruits.descendingIterator(); System.out.print("Fruits by descending order: "); while (descIterator.hasNext()) { System.out.print(descIterator.next() + ", "); } Output:

Set Fruits: [Apple, Banana, Grape, Mango, Orange]

Fruits by descending order: Orange, Mango, Grape, Banana, Apple,

The following example demonstrates the benefits of using navigation methods like lower(), higher(), ceiling() and floor(); and range view methods like headSet(), subSet() and tailSet().

Given the following entity class:

**class** Employee {

String name;

**int** salary;

**public** Employee(**int** salary) {

**his**.salary = salary;

}

**public** Employee(String name, **int** salary) {

**this**.name = name;

**this**.salary = salary;

}

**public** String toString() {

**return** **this**.name + "-" + salary;

}

**public** String getName() {

**return** **this**.name;

}

**public** Integer getSalary() {

**return** **new** ~~Integer~~(**this**.salary);

}

**public** **boolean** equals(Object obj) {

**if** (obj **instanceof** Employee) {

Employee another = (Employee) obj;

**if** (**this**.name.equals(another.name) && **this**.salary == another.salary) {

**return** **true**;

}

}

**return** **false**;

}

**public** **int** hashCode() {

**return** 31 \* name.hashCode() + salary;

}

}

Note that this class overrides the equals() and hashCode() methods based on employee’s name and salary. The following comparator class compares two employees based on their salary:

public class EmployeeComparator implements Comparator<Employee> { public int compare(Employee emp1, Employee emp2) { return emp1.getSalary().compareTo(emp2.getSalary()); }

}

We add 8 employees into a navigable set like this:

NavigableSet<Employee> setEmployees = new TreeSet<>(new EmployeeComparator()); setEmployees.add(new Employee("Tom", 80000)); setEmployees.add(new Employee("Jack", 35000)); setEmployees.add(new Employee("Jim", 62500)); setEmployees.add(new Employee("Peter", 58200)); setEmployees.add(new Employee("Mary", 77000)); setEmployees.add(new Employee("Jane", 69500)); setEmployees.add(new Employee("David", 54000)); setEmployees.add(new Employee("Sam", 82000));

System.out.println("Employees: " + setEmployees); Output:

Employees: [Jack-35000, David-54000, Peter-58200, Jim-62500, Jane-69500, Mary-77000, Tom80000, Sam-82000]

Here, an employee object is printed with name and salary.

* **Using the higher() method, we can know the employee whose salary is higher than the employee ‘Tom’:**

Employee Tom = new Employee("Tom", 80000);

Employee emp1 = setEmployees.higher(Tom); if (emp1 != null) {

System.out.println("The employee whose salary > Tom: " + emp1); } Output:

The employee whose salary > Tom: Sam-82000

**NOTE**: to allow this kind of search possible, the entity class must correctly override the equals() and hashCode() method, as shown in the Employee class above.

* **Using the lower() method, we can know the employee whose salary is less than the employee Tom:** Employee emp2 = setEmployees.lower(Tom); if (emp2 != null) {

System.out.println("The employee whose salary < Tom: " + emp2); } Output:

The employee whose salary < Tom: Mary-77000

* **Using the ceiling() method, we can know the employee whose salary is greater than 60,000 USD/year like this:** Employee emp3 = setEmployees.ceiling(new Employee(60000)); if (emp3 != null) {

System.out.println("The employee whose >= 60000: " + emp3); } Output:

The employee whose >= 60000: Jim-62500

* **Using the floor() method, we can know the employee whose salary is less than 40,000 USD like this:** Employee emp4 = setEmployees.floor(new Employee(40000)); if (emp4 != null) {

System.out.println("The employee whose <= 40000: " + emp4); } Output:

The employee whose <= 40000: Jack-35000

* **With the tailSet() method, we can know the employees who are high paid (salary > 70,000 USD):**

SortedSet<Employee> highPaidEmployees = setEmployees.tailSet(new Employee(70000));

System.out.println("High paid employees: " + highPaidEmployees);

Output:

High paid employees: [Mary-77000, Tom-80000, Sam-82000]

* **With the headSet() method, we can know the employees who are low paid (under 40,000USD/year):**

SortedSet<Employee> lowPaidEmployees = setEmployees.headSet(new Employee(60000));

System.out.println("Low paid employees: " + lowPaidEmployees);

Output:

Low paid employees: [Jack-35000, David-54000, Peter-58200]

* **With the subSet() method, we can know the employees who are good paid (salary is between 60,000 and 70,000):**

SortedSet<Employee> goodPaidEmployees = setEmployees.subSet(new Employee(60000), new

Employee(70000));

System.out.println("Good paid employees: " + goodPaidEmployees);

Output:

Good paid employees: [Jim-62500, Jane-69500]