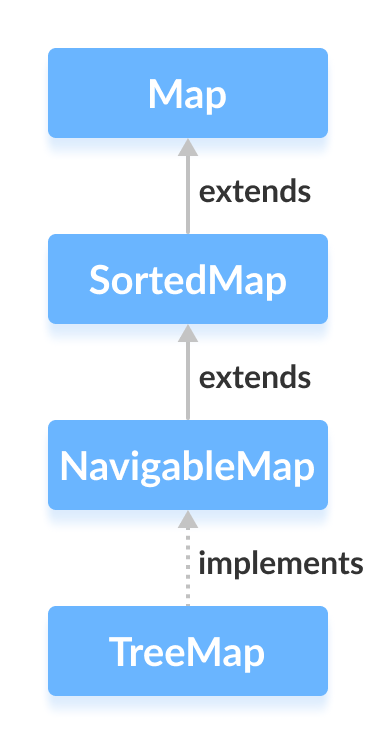
**Java TreeMap**

The TreeMap class of the Java collections framework provides the tree data structure implementation.

It implements the NavigableMap interface.



## Creating a TreeMap

In order to create a TreeMap, we must import the java.util.TreeMap package first. Once we import the package, here is how we can create a TreeMap in Java.

TreeMap<Key, Value> numbers = new TreeMap<>();

In the above code, we have created a TreeMap named numbers without any arguments. In this case, the elements in TreeMap are sorted naturally (ascending order).

However, we can customize the sorting of elements by using the Comparator interface. We will learn about it later in this tutorial.

Here,

* Key - a unique identifier used to associate each element (value) in a map
* Value - elements associated by keys in a map

## Methods of TreeMap

The TreeMap class provides various methods that allow us to perform operations on the map.

## Insert Elements to TreeMap

* put() - inserts the specified key/value mapping (entry) to the map
* putAll() - inserts all the entries from specified map to this map
* putIfAbsent() - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

// Creating TreeMap of even numbers

TreeMap<String, Integer> evenNumbers = new TreeMap<>();

// Using put()

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

// Using putIfAbsent()

evenNumbers.putIfAbsent("Six", 6);

System.out.println("TreeMap of even numbers: " + evenNumbers);

//Creating TreeMap of numbers

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("One", 1);

// Using putAll()

numbers.putAll(evenNumbers);

System.out.println("TreeMap of numbers: " + numbers);

}

}

**Output**

TreeMap of even numbers: {Four=4, Six=6, Two=2}

TreeMap of numbers: {Four=4, One=1, Six=6, Two=2}

## Access TreeMap Elements

**1. Using entrySet(), keySet() and values()**

* entrySet() - returns a set of all the key/values mapping (entry) of a treemap
* keySet() - returns a set of all the keys of a tree map
* values() - returns a set of all the maps of a tree map

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("TreeMap: " + numbers);

// Using entrySet()

System.out.println("Key/Value mappings: " + numbers.entrySet());

// Using keySet()

System.out.println("Keys: " + numbers.keySet());

// Using values()

System.out.println("Values: " + numbers.values());

}

}

**Output**

TreeMap: {One=1, Three=3, Two=2}

Key/Value mappings: [One=1, Three=3, Two=2]

Keys: [One, Three, Two]

Values: [1, 3, 2]

**2. Using get() and getOrDefault()**

* get() - Returns the value associated with the specified key. Returns null if the key is not found.
* getOrDefault() - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("TreeMap: " + numbers);

// Using get()

int value1 = numbers.get("Three");

System.out.println("Using get(): " + value1);

// Using getOrDefault()

int value2 = numbers.getOrDefault("Five", 5);

System.out.println("Using getOrDefault(): " + value2);

}

}

**Output**

TreeMap: {One=1, Three=3, Two=2}

Using get(): 3

Using getOrDefault(): 5

Here, the getOrDefault() method does not find the key Five. Hence it returns the specified default value 5.

## Remove TeeMap Elements

* remove(key) - returns and removes the entry associated with the specified key from a TreeMap
* remove(key, value) - removes the entry from the map only if the specified key is associated with the specified value and returns a boolean value

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("TreeMap: " + numbers);

// remove method with single parameter

int value = numbers.remove("Two");

System.out.println("Removed value: " + value);

// remove method with two parameters

boolean result = numbers.remove("Three", 3);

System.out.println("Is the entry {Three=3} removed? " + result);

System.out.println("Updated TreeMap: " + numbers);

}

}

**Output**

TreeMap: {One=1, Three=3, Two=2}

Removed value = 2

Is the entry {Three=3} removed? True

Updated TreeMap: {One=1}

## Replace TreeMap Elements

* replace(key, value) - replaces the value mapped by the specified key with the new value
* replace(key, old, new) - replaces the old value with the new value only if the old value is already associated with the specified key
* replaceAll(function) - replaces each value of the map with the result of the specified function

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

System.out.println("Original TreeMap: " + numbers);

// Using replace()

numbers.replace("Second", 22);

numbers.replace("Third", 3, 33);

System.out.println("TreeMap using replace: " + numbers);

// Using replaceAll()

numbers.replaceAll((key, oldValue) -> oldValue + 2);

System.out.println("TreeMap using replaceAll: " + numbers);

}

}

**Output**

Original TreeMap: {First=1, Second=2, Third=3}

TreeMap using replace(): {First=1, Second=22, Third=33}

TreeMap using replaceAll(): {First=2, Second=24, Third=35}

In the above program notice the statement

numbers.replaceAll((key, oldValue) -> oldValue + 2);

Here, we have passed a lambda expression as an argument.

The replaceAll() method accesses all the entries of the map. It then replaces all the elements with the new values (returned from the lambda expression).

## Methods for Navigation

Since the TreeMap class implements NavigableMap, it provides various methods to navigate over the elements of the treemap.

### 1. First and Last Methods

* firstKey() - returns the first key of the map
* firstEntry() - returns the key/value mapping of the first key of the map
* lastKey() - returns the last key of the map
* lastEntry() - returns the key/value mapping of the last key of the map

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

System.out.println("TreeMap: " + numbers);

// Using the firstKey() method

String firstKey = numbers.firstKey();

System.out.println("First Key: " + firstKey);

// Using the lastKey() method

String lastKey = numbers.lastKey();

System.out.println("Last Key: " + lastKey);

// Using firstEntry() method

System.out.println("First Entry: " + numbers.firstEntry());

// Using the lastEntry() method

System.out.println("Last Entry: " + numbers.lastEntry());

}

}

**Output**

TreeMap: {First=1, Second=2, Third=3}

First Key: First

Last Key: Third

First Entry: First=1

Last Entry: Third=3

### 2. Ceiling, Floor, Higher and Lower Methods

* **higherKey()** - Returns the lowest key among those keys that are greater than the specified key.
* **higherEntry()** - Returns an entry associated with a key that is lowest among all those keys greater than the specified key.
* **lowerKey()** - Returns the greatest key among all those keys that are less than the specified key.
* **lowerEntry()** - Returns an entry associated with a key that is greatest among all those keys that are less than the specified key.
* **ceilingKey()** - Returns the lowest key among those keys that are greater than the specified key. If the key passed as an argument is present in the map, it returns that key.
* **ceilingEntry()** - Returns an entry associated with a key that is lowest among those keys that are greater than the specified key. It an entry associated with the key passed an argument is present in the map, it returns the entry associated with that key.
* **floorKey()** - Returns the greatest key among those keys that are less than the specified key. If the key passed as an argument is present, it returns that key.
* **floorEntry()** - Returns an entry associated with a key that is greatest among those keys that are less than the specified key. If the key passed as argument is present, it returns that key.

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 5);

numbers.put("Third", 4);

numbers.put("Fourth", 6);

System.out.println("TreeMap: " + numbers);

// Using higher()

System.out.println("Using higherKey(): " + numbers.higherKey("Fourth"));

System.out.println("Using higherEntry(): " + numbers.higherEntry("Fourth"));

// Using lower()

System.out.println("\nUsing lowerKey(): " + numbers.lowerKey("Fourth"));

System.out.println("Using lowerEntry(): " + numbers.lowerEntry("Fourth"));

// Using ceiling()

System.out.println("\nUsing ceilingKey(): " + numbers.ceilingKey("Fourth"));

System.out.println("Using ceilingEntry(): " + numbers.ceilingEntry("Fourth"));

// Using floor()

System.out.println("\nUsing floorKey(): " + numbers.floorKey("Fourth"));

System.out.println("Using floorEntry(): " + numbers.floorEntry("Fourth"));

}

}

**Output**

TreeMap: {First=1, Fourth=6, Second=5, Third=4}

Using higherKey(): Second

Using higherEntry(): Second=5

Using lowerKey(): First

Using lowerEntry(): First=1

Using ceilingKey(): Fourth

Using ceilingEntry(): Fourth=6

Using floorkey(): Fourth

Using floorEntry(): Fourth=6

### 3. pollFirstEntry() and pollLastEntry() Methods

* pollFirstEntry() - returns and removes the entry associated with the first key of the map
* pollLastEntry() - returns and removes the entry associated with the last key of the map

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

System.out.println("TreeMap: " + numbers);

//Using the pollFirstEntry() method

System.out.println("Using pollFirstEntry(): " + numbers.pollFirstEntry());

// Using the pollLastEntry() method

System.out.println("Using pollLastEntry(): " + numbers.pollLastEntry());

System.out.println("Updated TreeMap: " + numbers);

}

}

**Output**

TreeMap: {First=1, Second=2, Third=3}

Using pollFirstEntry(): First=1

Using pollLastEntry(): Third=3

Updated TreeMap: {Second=2}

### 4. headMap(), tailMap() and subMap() Methods

**headMap(key, booleanValue)**

The headMap() method returns all the key/value pairs of a treemap before the specified key (which is passed as an argument).

The booleanValue parameter is optional. Its default value is false.

If true is passed as a booleanValue, the method also includes the key/value pair of the key which is passed as an argument.

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

numbers.put("Fourth", 4);

System.out.println("TreeMap: " + numbers);

System.out.println("\nUsing headMap() Method:");

// Using headMap() with default booleanValue

System.out.println("Without boolean value: " + numbers.headMap("Fourth"));

// Using headMap() with specified booleanValue

System.out.println("With boolean value: " + numbers.headMap("Fourth", true));

}

}

**Output**

TreeMap: {First=1, Fourth=4, Second=2, Third=3}

Using headMap() Method:

Without boolean value: {First=1}

With boolean value: {First=1, Fourth=4}

**tailMap(key, booleanValue)**

The tailMap() method returns all the key/value pairs of a treemap starting from the specified key (which is passed as an argument).

The booleanValue is an optional parameter. Its default value is true.

If false is passed as a booleanValue, the method doesn't include the key/value pair of the specified key.

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

numbers.put("Fourth", 4);

System.out.println("TreeMap: " + numbers);

System.out.println("\nUsing tailMap() Method:");

// Using tailMap() with default booleanValue

System.out.println("Without boolean value: " + numbers.tailMap("Second"));

// Using tailMap() with specified booleanValue

System.out.println("With boolean value: " + numbers.tailMap("Second", false));

}

}

**Output**

TreeMap: {First=1, Fourth=4, Second=2, Third=3}

Using tailMap() Method:

Without boolean value: {Third=3}

With boolean value: {Second=2, Third=3}

**subMap(k1, bV1, k2, bV2)**

The subMap() method returns all the entries associated with keys between k1 and k2 including the entry of k1.

The bV1 and bV2 are optional boolean parameters. The default value of bV1 is true and the default value of bV2 is false.

If false is passed as bV1, the method returns all the entries associated with keys between k1 and k2 without including the entry of k1.

If true is passed as bV2, the method returns all the entries associated with keys between k1 and k2 including the entry of k2.

For example,

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

numbers.put("Fourth", 4);

System.out.println("TreeMap: " + numbers);

System.out.println("\nUsing subMap() Method:");

// Using subMap() with default booleanValue

System.out.println("Without boolean value: " + numbers.subMap("Fourth", "Third"));

// Using subMap() with specified booleanValue

System.out.println("With boolean value: " + numbers.subMap("Fourth", false, "Third", true));

}

}

**Output**

TreeMap: {First=1, Fourth=2, Second=2, Third=3}

Using subMap() Method:

Without boolean value: {Fourth=4, Second=2}

With boolean value: {Second=2, Third=3}

## Other Methods of TreeMap

|  |  |
| --- | --- |
| Method | Description |
| clone() | Creates a copy of the TreeMap |
| containsKey() | Searches the TreeMap for the specified key and returns a boolean result |
| containsValue() | Searches the TreeMap for the specified value and returns a boolean result |
| size() | Returns the size of the TreeMap |
| clear() | Removes all the entries from the TreeMap |

## TreeMap Comparator

In all the examples above, treemap elements are sorted naturally (in ascending order). However, we can also customize the ordering of keys.

For this, we need to create our own comparator class based on which keys in a treemap are sorted. For example,

import java.util.TreeMap;

import java.util.Comparator;

class Main {

public static void main(String[] args) {

// Creating a treemap with a customized comparator

TreeMap<String, Integer> numbers = new TreeMap<>(new CustomComparator());

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

numbers.put("Fourth", 4);

System.out.println("TreeMap: " + numbers);

}

// Creating a comparator class

public static class CustomComparator implements Comparator<String> {

@Override

public int compare(String number1, String number2) {

int value = number1.compareTo(number2);

// elements are sorted in reverse order

if (value > 0) {

return -1;

}

else if (value < 0) {

return 1;

}

else {

return 0;

}

}

}

}

**Output**

TreeMap: {Third=3, Second=2, Fourth=4, First=1}

In the above example, we have created a treemap passing CustomComparator class as an argument.

The CustomComparator class implements the Comparator interface.

We then override the compare() method to sort elements in reverse order.

To learn more about the comparator, visit Java Comparator.