A Guide to TreeMap in Java

Last modified: May 5, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

Java Collections (http://www.baeldung.com/tag/collections/)

1. Overview

In this article, we are going to explore *TreeMap* implementation of *Map* interface from Java Collections Framework(JCF).

TreeMap is a map implementation that keeps its entries sorted according to the natural ordering of its keys or better still using a comparator if provided by the user at construction time.

Previously, we have covered *HashMap* (/java-hashmap) and *LinkedHashMap* (/java-linked-hashmap) implementations and we will realize that there is quite a bit of information about how these classes work that is similar.

The mentioned articles are highly recommended reading before going forth with this one.

2. Default Sorting in *TreeMap*

By default, *TreeMap* sorts all its entries according to their natural ordering. For an integer, this would mean ascending order and for strings, alphabetical order.

Let's see the natural ordering in a test:

```
@Test
public void givenTreeMap_whenOrdersEntriesNaturally_thenCorrect() {
    TreeMap<Integer, String> map = new TreeMap<>();
    map.put(3, "val");
    map.put(2, "val");
    map.put(1, "val");
    map.put(5, "val");
    map.put(4, "val");

    assertEquals("[1, 2, 3, 4, 5]", map.keySet().toString());
}
```

Notice that we placed the integer keys in a non-orderly manner but on retrieving the key set, we confirm that they are indeed maintained in ascending order. This is the natural ordering of integers.

Likewise, when we use strings, they will be sorted in their natural order, i.e. alphabetically:

```
@Test
public void givenTreeMap_whenOrdersEntriesNaturally_thenCorrect2() {
    TreeMap<String, String> map = new TreeMap<>();
    map.put("c", "val");
    map.put("b", "val");
    map.put("a", "val");
    map.put("e", "val");
    map.put("d", "val");

    assertEquals("[a, b, c, d, e]", map.keySet().toString());
}
```

TreeMap, unlike a hash map and linked hash map, does not employ the hashing principle anywhere since it does not use an array to store its entries.

3. Custom Sorting in *TreeMap*

If we're not satisfied with the natural ordering of *TreeMap*, we can also define our own rule for ordering by means of a comparator during construction of a tree map.

In the example below, we want the integer keys to be ordered in descending order:

```
@Test
public void givenTreeMap_whenOrdersEntriesByComparator_thenCorrect() {
    TreeMap<Integer, String> map =
      new TreeMap<>(Comparator.reverseOrder());
    map.put(3, "val");
    map.put(2, "val");
    map.put(1, "val");
    map.put(5, "val");
    map.put(4, "val");

    assertEquals("[5, 4, 3, 2, 1]", map.keySet().toString());
}
```

A hash map does not guarantee the order of keys stored and specifically does not guarantee that this order will remain the same over time, but a tree map guarantees that the keys will always be sorted according to the specified order.

4. Importance of *TreeMap* Sorting

We now know that *TreeMap* stores all its entries in sorted order. Because of this attribute of tree maps, we can perform queries like; find "largest", find "smallest", find all keys less than or greater than a certain value, etc.

The code below only covers a small percentage of these cases:

```
@Test
public void givenTreeMap_whenPerformsQueries_thenCorrect() {
    TreeMap<Integer, String> map = new TreeMap<>();
    map.put(3, "val");
    map.put(2, "val");
    map.put(1, "val");
    map.put(5, "val");
    map.put(4, "val");

    Integer highestKey = map.lastKey();
    Integer lowestKey = map.firstKey();
    Set<Integer> keysLessThan3 = map.headMap(3).keySet();
    Set<Integer> keysGreaterThanEqTo3 = map.tailMap(3).keySet();

    assertEquals(new Integer(5), highestKey);
    assertEquals(new Integer(1), lowestKey);
    assertEquals("[1, 2]", keysLessThan3.toString());
    assertEquals("[3, 4, 5]", keysGreaterThanEqTo3.toString());
}
```

5. Internal Implementation of *TreeMap*

TreeMap implements *NavigableMap* interface and bases it's internal working on the principles of red-black trees:

```
public class TreeMap<K,V> extends AbstractMap<K,V>
  implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

The principle of red-black trees is beyond the scope of this article, however, there are key things to remember in order to understand how they fit into *TreeMap*.

First of all, a red-black tree is a data structure that consists of nodes; picture an inverted mango tree with its root in the sky and the branches growing downward. The root will contain the first element added to the tree.

The rule is that starting from the root, any element in the left branch of any node is always less than the element in the node itself. Those on the right are always greater. What defines greater or less than is determined by the natural ordering of the elements or the defined comparator at construction as we saw earlier.

This rule guarantees that the entries of a treemap will always be in sorted and predictable order.

Secondly, a red-black tree is a self-balancing binary search tree. This attribute and the above guarantee that basic operations like search, get, put and remove take logarithmic time *O(log n)*.

Being self-balancing is key here. As we keep inserting and deleting entries, picture the tree growing longer on one edge or shorter on the other.

This would mean that an operation would take a shorter time on the shorter branch and longer time on the branch which is furthest from the root, something we would not want to happen.

Therefore, this is taken care of in the design of red-black trees. For every insertion and deletion, the maximum height of the tree on any edge is maintained at *O(log n)* i.e. the tree balances itself continuously.

Just like hash map and linked hash map, a tree map is not synchronized and therefore the rules for using it in a multi-threaded environment are similar to those in the other two map implementations.

6. Choosing the Right Map

Having looked at *HashMap* (/java-hashmap) and *LinkedHashMap* (/java-linked-hashmap) implementations previously and now *TreeMap*, it is important to make a brief comparison between the three to guide us on which one fits where.

A hash map is good as a general purpose map implementation that provides rapid storage and retrieval operations. However, it falls short because of its chaotic and unorderly arrangement of entries.

This causes it to perform poorly in scenarios where there is a lot of iteration as the entire capacity of the underlying array affects traversal other than just the number of entries.

A linked hash map possesses the good attributes of hash maps and adds order to the entries. It performs better where there is a lot of iteration because only the number of entries is taken into account regardless of capacity.

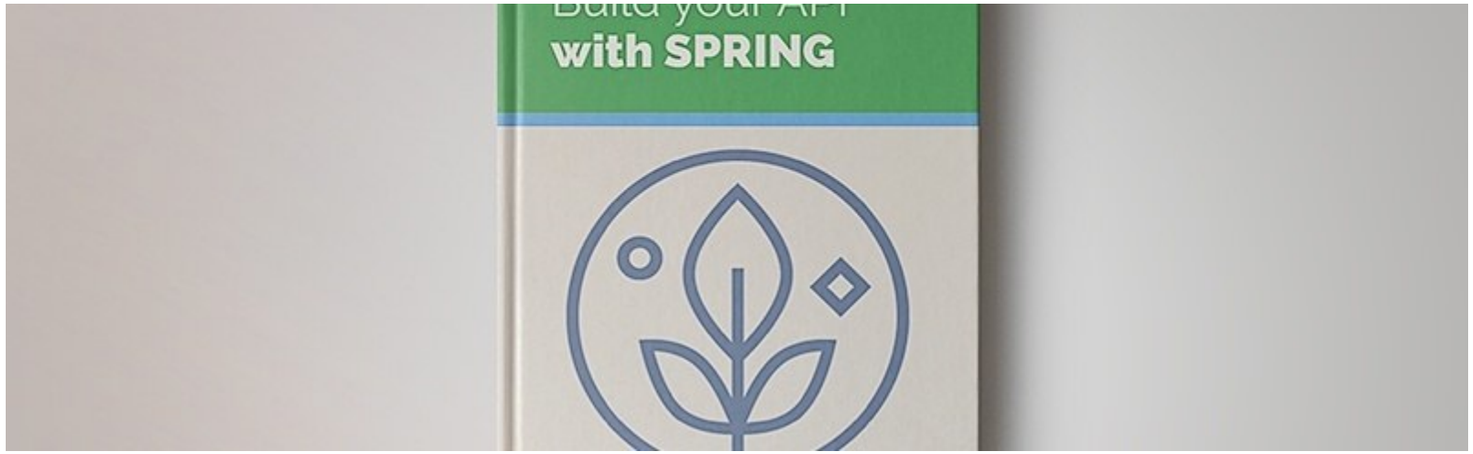
A tree map takes ordering to the next level by providing complete control over how the keys should be sorted. On the flip side, it offers worse general performance than the other two alternatives.

We could say a **linked hash map reduces the chaos in the ordering of a hash map without incurring the performance penalty of a tree map**.

7. Conclusion

In this article, we have explored Java *TreeMap* class and its internal implementation. Since it is the last in a series of common Map interface implementations, we also went ahead to briefly discuss where it fits best in relation to the other two.

The full source code for all the examples used in this article can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-collections).



(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)

THE COURSES (HTTP://COURSES.BAELDUNG.COM)

CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)

WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)

CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)

COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)

TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)

PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)

EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)

MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)