Java TreeMap vs HashMap

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. Introduction

In this article, we're going to compare two *Map* implementations: *TreeMap* and *HashMap*.

Both implementations form an integral part of the Java *Collections* Framework and store data as *key-value* pairs.

2. Differences

2.1. Implementation

We'll first talk about the *HashMap* which is a hashtable-based implementation. It extends the *AbstractMap* class and implements the *Map* interface. A *HashMap* works on the principle of *hashing (http://www.baeldung.com/java-hashcode)*.

This *Map* implementation usually acts as a bucketed *hash table*, but **when buckets get too large, they get transformed into nodes of *TreeNodes***, each structured similarly to those in *java.util.TreeMap.*

You can find more on the *HashMap's* internals in the article focused on it (http://www.baeldung.com/java-hashmap).

On the other hand, *TreeMap* extends *AbstractMap* class and implements *NavigableMap* interface. A *TreeMap* stores map elements in a *Red-Black* tree, which is a **Self-Balancing *Binary Search Tree***.

And, you can also find more on the *TreeMap's* internals in the article focused on it here (http://www.baeldung.com/java-treemap).

2.2. Order

HashMap* doesn't provide any guarantee over the way the elements are arranged in the *Map.

It means, **we can't assume any order while iterating over *keys* and *values* of a *HashMap*:**

```java
@Test
public void whenInsertObjectsHashMap_thenRandomOrder() {
    Map<Integer, String> hashmap = new HashMap<>();
    hashmap.put(3, "TreeMap");
    hashmap.put(2, "vs");
    hashmap.put(1, "HashMap");

    assertThat(hashmap.keySet(), containsInAnyOrder(1, 2, 3));
}
```

However, items in a *TreeMap* are **sorted according to their natural order**.

If *TreeMap* objects cannot be sorted according to natural order then we may make use of a *Comparator* or *Comparable* to define the order in which the elements are arranged within the *Map:*

```java
@Test
public void whenInsertObjectsTreeMap_thenNaturalOrder() {
    Map<Integer, String> treemap = new TreeMap<>();
    treemap.put(3, "TreeMap");
    treemap.put(2, "vs");
    treemap.put(1, "HashMap");

    assertThat(treemap.keySet(), contains(1, 2, 3));
}
```

2.3. *Null* Values

HashMap allows storing at most one *null key* and many *null* values.

Let's see an example:

```java
@Test
public void whenInsertNullInHashMap_thenInsertsNull() {
    Map<Integer, String> hashmap = new HashMap<>();
    hashmap.put(null, null);

    assertNull(hashmap.get(null));
}
```

However, *TreeMap* doesn't allow a *null key* but may contain many *null* values.

A *null* key isn't allowed because the *compareTo()* or the *compare()* method throws a *NullPointerException:*

```java
@Test(expected = NullPointerException.class)
public void whenInsertNullInTreeMap_thenException() {
    Map<Integer, String> treemap = new TreeMap<>();
    treemap.put(null, "NullPointerException");
}
```

If we're using a *TreeMap* with a user-defined *Comparator*, then it depends on the implementation of the compare*()* method how *null* values get handled.

3. Performance Analysis

Performance is the most critical metric that helps us understand the suitability of a data-structure given a use-case.

In this section, we'll provide a comprehensive analysis of performance for *HashMap* and *TreeMap.*

3.1. *HashMap*

***HashMap*, being a hashtable-based implementation, internally uses an array-based data structure to organize its elements according to the *hash function*.**

HashMap provides expected constant-time performance *O(1)* for most operations like *add()*, *remove()* and *contains().* Therefore, it's significantly faster than a *TreeMap*.

The average time to search for an element under the reasonable assumption, in a hash table is *O(1).* But, an improper implementation of the *hash function* may lead to a poor distribution of values in buckets which results in:

- Memory Overhead – many buckets remain unused
- Performance Degradation **–** the higher the number of collisions, the lower the performance

Before Java 8, *Separate Chaining* was the only preferred way to handle collisions. It's usually implemented using linked lists, *i.e.*, if there is any collision or two different elements have same hash value then store both the items in the same linked list.

Therefore, searching for an element in a *HashMap,* in the worst case could have taken as long as searching for an element in a linked list *i.e. O(n)* time.

However, with JEP 180 (http://openjdk.java.net/jeps/180) coming into the picture, there's been a subtle change in the implementation of the way the elements are arranged in a *HashMap.*

According to the specification, when buckets get too large and contain enough nodes they get transformed into modes of *TreeNodes*, each structured similarly to those in *TreeMap*.

Hence, in the event of high hash collisions, the worst-case performance will improve from *O(n)* to *O(log n).*

The code performing this transformation has been illustrated below:

```
if(binCount >= TREEIFY_THRESHOLD - 1) {
    treeifyBin(tab, hash);
}
```

The value for *TREEIFY_THRESHOLD* is eight which effectively denotes the threshold count for using a tree rather than a linked list for a bucket.

It is evident that:

- A *HashMap* requires way more memory than is needed to hold its data
- A *HashMap* shouldn't be more than 70% – 75% full. If it gets close, it gets resized and entries rehashed
- Rehashing requires *n* operations which is costly wherein our constant time insert becomes of order *O(n)*
- It's the hashing algorithm which determines the order of inserting the objects in the *HashMap*

The performance of a *HashMap* can be tuned by setting the custom *initial capacity* and the *load factor*, at the time of *HashMap* object creation itself.

However, we should choose a *HashMap* if:

- we know approximately how many items to maintain in our collection
- we don't want to extract items in a natural order

Under the above circumstances, *HashMap* is our best choice because it offers constant time insertion, search, and deletion.

3.2. *TreeMap*

A *TreeMap* stores its data in a hierarchical tree with the ability to sort the elements with the help of a custom *Comparator.*

A summary of its performance:

- *TreeMap* provides a performance of *O(log(n))* for most operations like *add()*, *remove()* and *contains()*
- A *Treemap* can save memory (in comparison to *HashMap)* because it only uses the amount of memory needed to hold its items, unlike a *HashMap* which uses contiguous region of memory
- A tree should maintain its balance in order to keep its intended performance, this requires a considerable amount of effort, hence complicates the implementation

We should go for a *TreeMap* whenever:

- memory limitations have to be taken into consideration
- we don't know how many items have to be stored in memory
- we want to extract objects in a natural order
- if items will be consistently added and removed

- we're willing to accept *O(log n)* search time

4. Similarities

4.1. Unique Elements

Both *TreeMap* and *HashMap* don't support duplicate keys. If added, it overrides the previous element (without an error or an exception):

```java
@Test
public void givenHashMapAndTreeMap_whenputDuplicates_thenOnlyUnique() {
    Map<Integer, String> treeMap = new HashMap<>();
    treeMap.put(1, "Baeldung");
    treeMap.put(1, "Baeldung");

    assertTrue(treeMap.size() == 1);

    Map<Integer, String> treeMap2 = new TreeMap<>();
    treeMap2.put(1, "Baeldung");
    treeMap2.put(1, "Baeldung");

    assertTrue(treeMap2.size() == 1);
}
```

4.2. Concurrent Access

Both *Map* implementations aren't *synchronized* and we need to manage concurrent access on our own.

Both must be synchronized externally whenever multiple threads access them concurrently and at least one of the threads modifies them.

We have to explicitly use *Collections.synchronizedMap(mapName)* to obtain a synchronized view of a provided map.

4.3. Fail-Fast Iterators

The *Iterator* throws a *ConcurrentModificationException* if the *Map* gets modified in any way and at any time once the iterator has been created.

Additionally, we can use the iterator's remove method to alter the *Map* during iteration.

Let's see an example:

```
@Test
public void whenModifyMapDuringIteration_thenThrowExecption() {
    Map<Integer, String> hashmap = new HashMap<>();
    hashmap.put(1, "One");
    hashmap.put(2, "Two");

    Executable executable = () -> hashmap
      .forEach((key,value) -> hashmap.remove(1));

    assertThrows(ConcurrentModificationException.class, executable);
}
```

5. Which Implementation to Use?

In general, both implementations have their respective pros and cons, however, **it's about understanding the underlying expectation and requirement which must govern our choice regarding the same.**

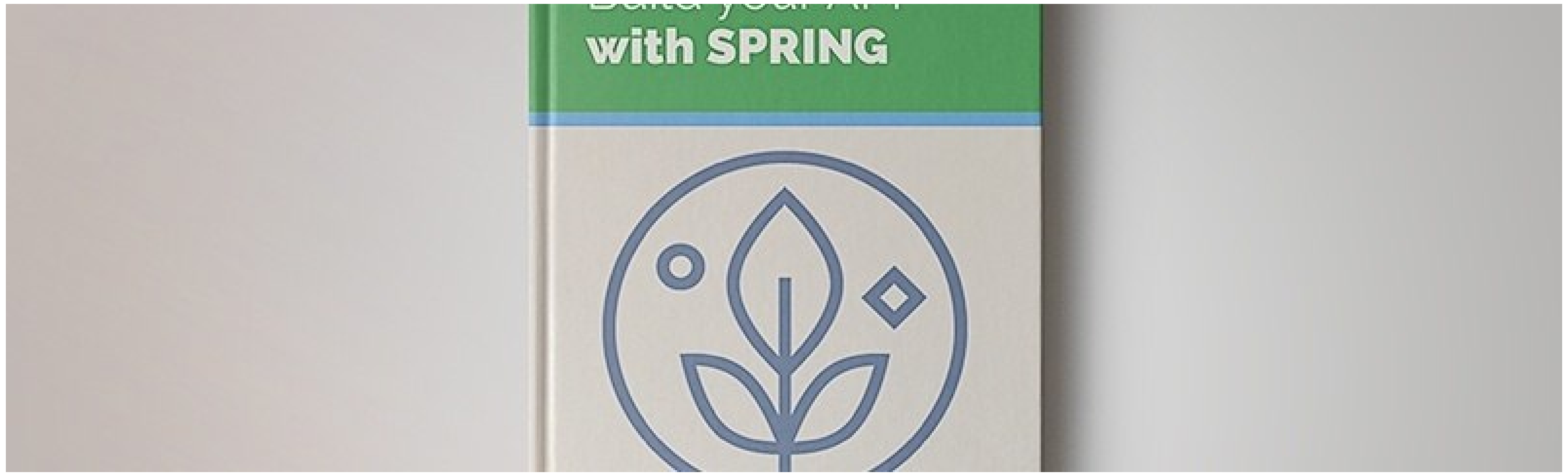
Summarizing:

- We should use a *TreeMap* if we want to keep our entries sorted
- We should use a *HashMap* if we prioritize performance over memory consumption
- Since a *TreeMap* has a more significant locality, we might consider it if we want to access objects that are relatively close to each other according to their natural ordering
- *HashMap* can be tuned using the *initialCapacity* and *loadFactor*, which isn't possible for the *TreeMap*
- We can use the *LinkedHashMap* if we want to preserve insertion order while benefiting from constant time access

6. Conclusion

In this article, we showed the differences and similarities between *TreeMap* and *HashMap*.

As always, the code examples for this article are available over on GitHub (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

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