TreeMap in Java

Last Updated : 16 Jul, 2024

Let us start with a simple Java code snippet that demonstrates how to create and use a TreeMap in Java.

```java
import java.util.Map;
import java.util.TreeMap;

public class TreeMapCreation {
    public static void main(String args[]) {

        // Create a TreeMap of Strings (keys) and Integers (values)
        TreeMap<String, Integer> treeMap = new TreeMap<>();

        // Displaying the TreeMap
        System.out.println("TreeMap elements: " + treeMap);
    }
}
```

Output

```
TreeMap elements: {}
```

The **TreeMap** in Java is used to implement Map interface and NavigableMap along with the AbstractMap Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. The treemap implementation is not synchronized in the sense that if a map is accessed by multiple threads, concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally.

The TreeMap in Java is a concrete implementation of the java.util.SortedMap interface. It provides an ordered collection of key-value pairs, where the keys are ordered based on their natural order or a custom Comparator passed to the constructor.

A TreeMap is implemented using a Red-Black tree, which is a type of self-balancing binary search tree. This provides efficient performance for common operations such as adding, removing, and retrieving elements, with an average time complexity of O(log n).

Here's an example of how to perform different operations in Java TreeMap:

```
import java.util.Map;
import java.util.TreeMap;

public class Main {
    public static void main(String[] args)
    {
        Map<String, Integer> treeMap = new TreeMap<>();

        // Adding elements to the tree map
        treeMap.put("A", 1); // O(log n)
        treeMap.put("C", 3); // O(log n)
        treeMap.put("B", 2); // O(log n)

        // Getting values from the tree map
        int valueA = treeMap.get("A"); // O(log n)
        System.out.println("Value of A: " + valueA);

        // Removing elements from the tree map
        treeMap.remove("B"); // O(log n)

        // Iterating over the elements of the tree map
        for (String key : treeMap.keySet()) { // O(n)
            System.out.println(
                "Key: " + key + ", Value: "
                + treeMap.get(key)); // O(log n) for each
                                     // get operation
        }
    }
}
```

Output

```
Value of A: 1
Key: A, Value: 1
Key: C, Value: 3
```

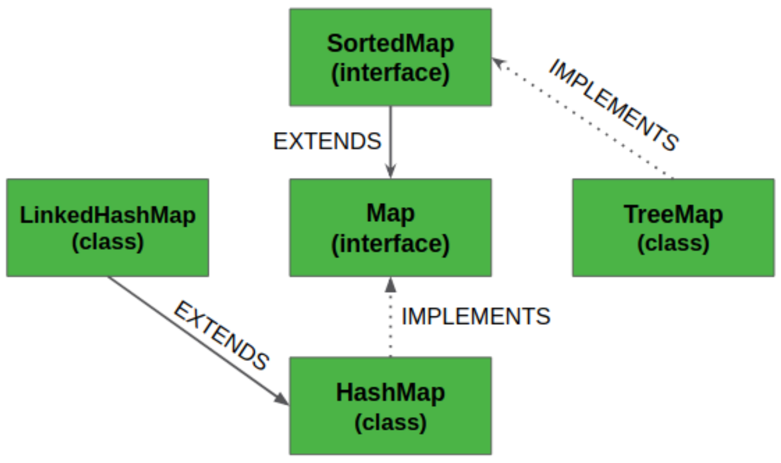
Time Complexity:

- Overall, the time complexity of the code is O(n log n), where n is the number of elements in the TreeMap.

Space Complexity:

- O(n) for storing the elements in the `TreeMap`.

TreeMap in Map Interface Hierarchy:



MAP Hierarchy in Java

Features of a TreeMap

Some important features of the TreeMap are as follows:

- This class is a member of the Java Collections Framework.
- The class implements Map interfaces including NavigableMap, SortedMap, and extends AbstractMap class.
- TreeMap in Java does not allow null keys (like Map) and thus a NullPointerException is thrown. However, multiple null values can be associated with different keys.
- Entry pairs returned by the methods in this class and its views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.

Now let us move forward and discuss Synchronized TreeMap. The implementation of a TreeMap is not synchronized. This means that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by using the Collections.synchronizedSortedMap method. This is best done at the creation time, to prevent accidental unsynchronized access to the set. This can be done as:

```
SortedMap m = Collections.synchronizedSortedMap(new
TreeMap(...));
```

How does the TreeMap Works Internally?

The methods in a TreeMap while getting keyset and values, return an Iterator that is fail-fast in nature. Thus, any concurrent modification will throw ConcurrentModificationException. A TreeMap is based upon a red-black tree data structure.

Each node in the tree has:

- 3 Variables (*K key=Key, V value=Value, boolean color=Color*)
- 3 References (*Entry left = Left, Entry right = Right, Entry parent = Parent*)

Constructors in TreeMap

In order to create a TreeMap, we need to create an object of the TreeMap class. The TreeMap class consists of various constructors that allow the possible creation of the TreeMap. The following are the constructors available in this class:

1. TreeMap()
2. TreeMap(Comparator comp)
3. TreeMap(Map M)
4. TreeMap(SortedMap sm)

Let us discuss them individually alongside implementing every constructor as follows:

Constructor 1: TreeMap()

This constructor is used to build an empty TreeMap that will be sorted by using the natural order of its keys.

Example:

```java
// Java Program to Demonstrate TreeMap
// using the Default Constructor

// Importing required classes
import java.util.*;

public class GFG {

    // Method 1
    // To show TreeMap constructor
    static void Example1stConstructor()
    {
        // Creating an empty TreeMap
        TreeMap<Integer, String> tree_map
            = new TreeMap<Integer, String>(); // O(1)

        // Mapping string values to int keys using put()
        // method
        tree_map.put(10, "Geeks"); // O(log n)
        tree_map.put(15, "4"); // O(log n)
        tree_map.put(20, "Geeks"); // O(log n)
        tree_map.put(25, "Welcomes"); // O(log n)
        tree_map.put(30, "You"); // O(log n)

        // Printing the elements of TreeMap
        System.out.println("TreeMap: " + tree_map); // O(n)
    }

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {
        System.out.println(
            "TreeMap using TreeMap() constructor:\n");

        // Calling constructor
        Example1stConstructor(); // O(n log n) for put and
                                 // O(n) for printing
    }
}
```

Output

```
TreeMap using TreeMap() constructor:


TreeMap: {10=Geeks, 15=4, 20=Geeks, 25=Welcomes, 30=You}
```

Time Complexity:

- Overall time complexity of the provided code snippet is: O(nlogn)
- O(n), where n is the number of elements in the TreeMap.

Constructor 2: TreeMap(Comparator comp)

This constructor is used to build an empty TreeMap object in which the elements will need an external specification of the sorting order.

Example:

```
// Java Program to Demonstrate TreeMap
// using Comparator Constructor

// Importing required classes
import java.util.*;
import java.util.concurrent.*;

// Class 1
// Helper class representing Student
class Student {
    int rollno;
    String name, address;

    public Student(int rollno, String name, String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    public String toString()
    {
        return this.rollno + " " + this.name + " "
            + this.address;
    }
}

// Class 2
// Helper class - Comparator implementation
class Sortbyroll implements Comparator<Student> {
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

// Class 3
// Main class
public class GFG {

    static void Example2ndConstructor()
    {
        TreeMap<Student, Integer> tree_map
            = new TreeMap<Student, Integer>(
                new Sortbyroll()); // O(1)

        tree_map.put(new Student(111, "bbbb", "london"),
                     2); // O(log n)
        tree_map.put(new Student(131, "aaaa", "nyc"),
                     3); // O(log n)
        tree_map.put(new Student(121, "cccc", "jaipur"),
                     1); // O(log n)

        System.out.println("TreeMap: " + tree_map); // O(n)
    }

    public static void main(String[] args)
    {
        System.out.println(
            "TreeMap using TreeMap(Comparator) constructor:\n");
        Example2ndConstructor(); // O(n log n) for put and
                                 // O(n) for printing
    }
}
```

Output

```
TreeMap using TreeMap(Comparator) constructor:


TreeMap: {111 bbbb london=2, 121 cccc jaipur=1, 131 aaaa nyc=3}
```

Time Complexity:

- **Overall Time Complexity:** O(nlogn)

Space Complexity:

- **Overall Space Complexity:** O(n)

Constructor 3: TreeMap(Map M)

This constructor is used to initialize a TreeMap with the entries from the given map M which will be sorted by using the natural order of the keys.

Example:

```
// Java Program to Demonstrate TreeMap
// using the Default Constructor

// Importing required classes
import java.util.*;
import java.util.concurrent.*;

// Main class
public class TreeMapImplementation {

    // Method 1
    // To illustrate constructor<Map>
    static void Example3rdConstructor()
    {
        // Creating an empty HashMap
        Map<Integer, String> hash_map
            = new HashMap<Integer, String>(); // O(1)

        // Mapping string values to int keys using put()
        // method
        hash_map.put(10, "Geeks"); // O(1)
        hash_map.put(15, "4"); // O(1)
        hash_map.put(20, "Geeks"); // O(1)
        hash_map.put(25, "Welcomes"); // O(1)
        hash_map.put(30, "You"); // O(1)

        // Creating the TreeMap using the Map
        TreeMap<Integer, String> tree_map
            = new TreeMap<Integer, String>(
                hash_map); // O(n log n)

        // Printing the elements of TreeMap
        System.out.println("TreeMap: " + tree_map); // O(n)
    }

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {
        System.out.println(
            "TreeMap using TreeMap(Map) constructor:\n");
        Example3rdConstructor();
    }
}
```

Output

```
TreeMap using TreeMap(Map) constructor:


TreeMap: {10=Geeks, 15=4, 20=Geeks, 25=Welcomes, 30=You}
```

Time Complexity:

- **Overall Time Complexity:** O(nlogn)

Constructor 4: TreeMap(SortedMap sm)

This constructor is used to initialize a TreeMap with the entries from the given sorted map which will be stored in the same order as the given sorted map.

Example:

```java
// Java Program to Demonstrate TreeMap
// using the SortedMap Constructor

// Importing required classes
import java.util.*;
import java.util.concurrent.*;

// Main class
public class GFG {

    // Method
    // To show TreeMap(SortedMap) constructor
    static void Example4thConstructor()
    {
        // Creating a SortedMap
        SortedMap<Integer, String> sorted_map
            = new ConcurrentSkipListMap<Integer,
                                        String>(); // O(1)

        // Mapping string values to int keys using put()
        // method
        sorted_map.put(10, "Geeks"); // O(log n)
        sorted_map.put(15, "4"); // O(log n)
        sorted_map.put(20, "Geeks"); // O(log n)
        sorted_map.put(25, "Welcomes"); // O(log n)
        sorted_map.put(30, "You"); // O(log n)

        // Creating the TreeMap using the SortedMap
        TreeMap<Integer, String> tree_map
            = new TreeMap<Integer, String>(
                sorted_map); // O(n log n)

        // Printing the elements of TreeMap
        System.out.println("TreeMap: " + tree_map); // O(n)
    }

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {
        System.out.println(
            "TreeMap using TreeMap(SortedMap) constructor:\n");
        Example4thConstructor(); // O(n log n) for put and
                                 // O(n) for printing
    }
}
```

Output

```
TreeMap using TreeMap(SortedMap) constructor:

TreeMap: {10=Geeks, 15=4, 20=Geeks, 25=Welcomes, 30=You}
```

Time Complexity:

- **Overall Time Complexity:** O(nlogn)
- **Overall Space Complexity:** O(n)

Methods in the TreeMap Class

Method	Action Performed
clear()	The method removes all mappings from this TreeMap and clears the map.
clone()	The method returns a shallow copy of this TreeMap.
containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
entrySet()	Returns a set view of the mappings contained in this map.
firstKey()	Returns the first (lowest) key currently in this sorted map.
get(Object key)	Returns the value to which this map maps the specified key.
headMap(Object key_value)	The method returns a view of the portion of the map strictly less than the parameter key_value.

Method	Action Performed
keySet()	The method returns a Set view of the keys contained in the treemap.
lastKey()	Returns the last (highest) key currently in this sorted map.
put(Object key, Object value)	The method is used to insert a mapping into a map.
putAll(Map map)	Copies all of the mappings from the specified map to this map.
remove(Object key)	Removes the mapping for this key from this TreeMap if present.
size()	Returns the number of key-value mappings in this map.
subMap((K startKey, K endKey)	The method returns the portion of this map whose keys range from startKey, inclusive, to endKey, exclusive.
values()	Returns a collection view of the values contained in this map.

Implementation: The following programs below will demonstrate better how to create, insert, and traverse through the TreeMap.

Illustration:

```java
// Java Program to Illustrate Operations in TreeMap
// Such as Creation, insertion
// searching, and traversal

// Importing required classes
import java.util.*;
import java.util.concurrent.*;

// Main class
// Implementation of TreeMap
```

```java
public class GFG {

    // Declaring a TreeMap
    static TreeMap<Integer, String> tree_map;

    // Method 1
    // To create TreeMap
    static void create()
    {
        // Creating an empty TreeMap
        tree_map = new TreeMap<Integer, String>(); // O(1)
        System.out.println("TreeMap successfully created");
    }

    // Method 2
    // To Insert values in the TreeMap
    static void insert()
    {
        // Mapping string values to int keys using put()
        // method
        tree_map.put(10, "Geeks"); // O(log n)
        tree_map.put(15, "4"); // O(log n)
        tree_map.put(20, "Geeks"); // O(log n)
        tree_map.put(25, "Welcomes"); // O(log n)
        tree_map.put(30, "You"); // O(log n)
        System.out.println(
            "\nElements successfully inserted in the TreeMap");
    }

    // Method 3
    // To search a key in TreeMap
    static void search(int key)
    {
        System.out.println(
            "\nIs key \"" + key + "\" present? "
            + tree_map.containsKey(key)); // O(log n)
    }

    // Method 4
    // To search a value in TreeMap
    static void search(String value)
    {
        System.out.println(
            "\nIs value \"" + value + "\" present? "
            + tree_map.containsValue(value)); // O(n)
    }

    // Method 5
    // To display the elements in TreeMap
    static void display()
    {
        System.out.println("\nDisplaying the TreeMap:");
        System.out.println("TreeMap: " + tree_map); // O(n)
    }

    // Method 6
    // To traverse TreeMap
    static void traverse()
    {
        System.out.println("\nTraversing the TreeMap:");
        for (Map.Entry<Integer, String> e :
             tree_map.entrySet()) // O(n)
            System.out.println(e.getKey() + " "
                               + e.getValue());
    }

    // Method 6
    // Main driver method
    public static void main(String[] args)
    {
        // Calling above defined methods inside main()
        create();
        insert();
        search(50);
        search("Geeks");
        display();
        traverse();
    }
}
```

Output

```
TreeMap successfully created

Elements successfully inserted in the TreeMap

Is key "50" present? false

Is value "Geeks" present? true

Displaying the TreeMap:
TreeMap: {10=Geeks, 15=4, 20=Geeks, 25=...
```

Time Complexity:

- **Time Complexity:** Overall, the time complexity operations are inserting and searching, which are O(nlogn) for insertions and O(n) for value search.

Space Complexity:

- O(n) where n is the number of elements in the TreeMap.

Performing Various Operations on TreeMap

After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the TreeMap. Now, let's see how to perform a few frequently used operations on the TreeMap.

Operation 1: Adding Elements

In order to add an element to the TreeMap, we can use the put() method. However, the insertion order is not retained in the TreeMap. Internally, for every element, the keys are compared and sorted in ascending order.

Example:

```java
// Java Program to Illustrate Addition of Elements
// in TreeMap using put() Method

// Importing required classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {
        // Default Initialization of a TreeMap
        TreeMap tm1 = new TreeMap(); // O(1)

        // Inserting the elements in TreeMap using put()
        // method
        tm1.put(3, "Geeks"); // O(log n)
        tm1.put(2, "For"); // O(log n)
        tm1.put(1, "Geeks"); // O(log n)

        // Initialization of a TreeMap using Generics
        TreeMap<Integer, String> tm2
            = new TreeMap<Integer, String>(); // O(1)

        // Inserting the elements in TreeMap again using
        // put() method
        tm2.put(new Integer(3), "Geeks"); // O(log n)
        tm2.put(new Integer(2), "For"); // O(log n)
        tm2.put(new Integer(1), "Geeks"); // O(log n)

        // Printing the elements of both TreeMaps
        // Map 1
        System.out.println(tm1); // O(n)
        // Map 2
        System.out.println(tm2); // O(n)
    }
}
```

Output

```
{1=Geeks, 2=For, 3=Geeks}
{1=Geeks, 2=For, 3=Geeks}
```

Time Complexity:

- **Overall Time Complexity:** O(nlogn)

Space Complexity:

- **For storing elements in `TreeMap`:** O(n)

Operation 2: Changing Elements

After adding the elements if we wish to change the element, it can be done by again adding the element with the put() method. Since the elements in the treemap are indexed using the keys, the value of the key can be changed by simply inserting the updated value for the key for which we wish to change.

Example:

```java
// Java program to Illustrate Updation of Elements
// in TreeMap using put() Method

// Importing required classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {
        // Initialization of a TreeMap using Generics
        TreeMap<Integer, String> tm
            = new TreeMap<Integer, String>(); // O(1)

        // Inserting the elements in Map using put() method
        tm.put(3, "Geeks"); // O(log n)
        tm.put(2, "Geeks"); // O(log n)
        tm.put(1, "Geeks"); // O(log n)

        // Print all current elements in map
        System.out.println(tm); // O(n)

        // Inserting the element at specified corresponding
        // to specified key
        tm.put(2, "For"); // O(log n)

        // Printing the updated elements of Map
        System.out.println(tm); // O(n)
    }
}
```

Output

```
{1=Geeks, 2=Geeks, 3=Geeks}
{1=Geeks, 2=For, 3=Geeks}
```

Time Complexity:

- The dominant operations are the `put` operations and the printing, which makes the overall time complexity O(nlogn)

Space Complexity:

- **For storing elements in `TreeMap`:** O(n)

Operation 3: Removing Element

In order to remove an element from the TreeMap, we can use the remove() method. This method takes the key value and removes the mapping for the key from this treemap if it is present in the map.

Example:

```java
// Java program to Illustrate Removal of Elements
// in TreeMap using remove() Method

// Importing required classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {
        // Initialization of a TreeMap using Generics
        TreeMap<Integer, String> tm
            = new TreeMap<Integer, String>(); // O(1)

        // Inserting the elements using put() method
        tm.put(3, "Geeks"); // O(log n)
        tm.put(2, "Geeks"); // O(log n)
        tm.put(1, "Geeks"); // O(log n)
        tm.put(4, "For"); // O(log n)

        // Printing all elements of Map
        System.out.println(tm); // O(n)

        // Removing the element corresponding to key
        tm.remove(4); // O(log n)

        // Printing updated TreeMap
        System.out.println(tm); // O(n)
    }
}
```

Output

```
{1=Geeks, 2=Geeks, 3=Geeks, 4=For}
{1=Geeks, 2=Geeks, 3=Geeks}
```

Time Complexity:

- Overall time complexity O(nlogn)

Space Complexity

- For storing elements in `TreeMap`: O(n)

Operation 4: Iterating through the TreeMap

There are multiple ways to iterate through the Map. The most famous way is to use a for-each loop and get the keys. The value of the key is found by using the *getValue() method*.

Example:

```
// Java Program to Illustrate Iterating over TreeMap
// using

// Importing required classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String args[]) {
        // Initialization of a TreeMap using Generics
        TreeMap<Integer, String> tm = new TreeMap<Integer, String>(); //
O(1)

        // Inserting the elements using put() method
        tm.put(3, "Geeks"); // O(log n)
        tm.put(2, "For");   // O(log n)
        tm.put(1, "Geeks"); // O(log n)

        // For-each loop for traversal over Map via entrySet() Method
        for (Map.Entry mapElement : tm.entrySet()) {
            int key = (int)mapElement.getKey(); // O(1)
            String value = (String)mapElement.getValue(); // O(1)
            // Printing the key and value
            System.out.println(key + " : " + value); // O(1)
        }
    }
}
```

Output

```
1 : Geeks
2 : For
3 : Geeks
```

Time Complexity:

- Overall time complexity O(nlogn)

Space Complexity:

- For storing elements in `TreeMap`: O(n)

Advantages of TreeMap:

- **Sorted order:** The TreeMap provides a sorted order of its elements, based on the natural order of its keys or a custom Comparator passed to the constructor. This makes it useful in situations where you need to retrieve elements in a specific order.
- **Predictable iteration order:** Because the elements in a TreeMap are stored in a sorted order, you can predict the order in which they will be returned during iteration, making it easier to write algorithms that process the elements in a specific order.
- **Search performance**: The TreeMap provides an efficient implementation of the Map interface, allowing you to retrieve elements in logarithmic time, making it useful in search algorithms where you need to retrieve elements quickly.
- **Self-balancing:** The TreeMap is implemented using a Red-Black tree, which is a type of self-balancing binary search tree. This provides efficient performance for adding, removing, and retrieving elements, as well as maintaining the sorted order of the elements.

Disadvantages of TreeMap:

- Slow for inserting elements: Inserting elements into a TreeMap can be slower than inserting elements into a regular Map, as the TreeMap needs to maintain the sorted order of its elements.
- Key restriction: The keys in a TreeMap must implement the java.lang.Comparable interface, or a custom Comparator must be provided. This can be a restriction if you need to use custom keys that do not implement this interface.

Reference books:

"Java Collections" by Maurice Naftalin and Philip Wadler. This book provides a comprehensive overview of the Java Collections framework, including the TreeMap.

"Java in a Nutshell" by David Flanagan. This book provides a quick reference for the core features of Java, including the TreeMap.

"Java Generics and Collections" by Maurice Naftalin and Philip Wadler. This book provides a comprehensive guide to generics and collections in Java, including the TreeMap.

Comment More info

Placement Training Program

Next Article

TreeMap clear() Method in Java

Similar Reads

Java.util.TreeMap.pollFirstEntry() and pollLastEntry() in Java