CSCI 2120:
Software Design & Development II

UNIT 2: Collections Framework & Generics

TreeMap

Overview

1. Introduction
2. Hierarchy of TreeMap
3. TreeMap class Declaration
4. Features of TreeMap
5. Constructors of TreeMap class
6. TreeMap methods
7. TreeMap Examples
8. When to use TreeMap
9. Key Points

Introduction

TreeMap in Java is a concrete class that is a red-black tree based implementation of the `Map` interface.

It provides an efficient way of storing key/value pairs in sorted order automatically and allows rapid retrieval.

A `TreeMap` implementation provides guaranteed log(n) time performance for checking, adding, retrieval, and removal operations.

The two main difference between `HashMap` and `TreeMap` is that:

- `HashMap` is an unordered collection of elements while `TreeMap` is sorted in the ascending order of its keys. The keys are sorted either using `Comparable` interface or `Comparator` interface.
- `HashMap` allows only one `null` key while `TreeMap` does not allow any `null` key.

Hierarchy of TreeMap class

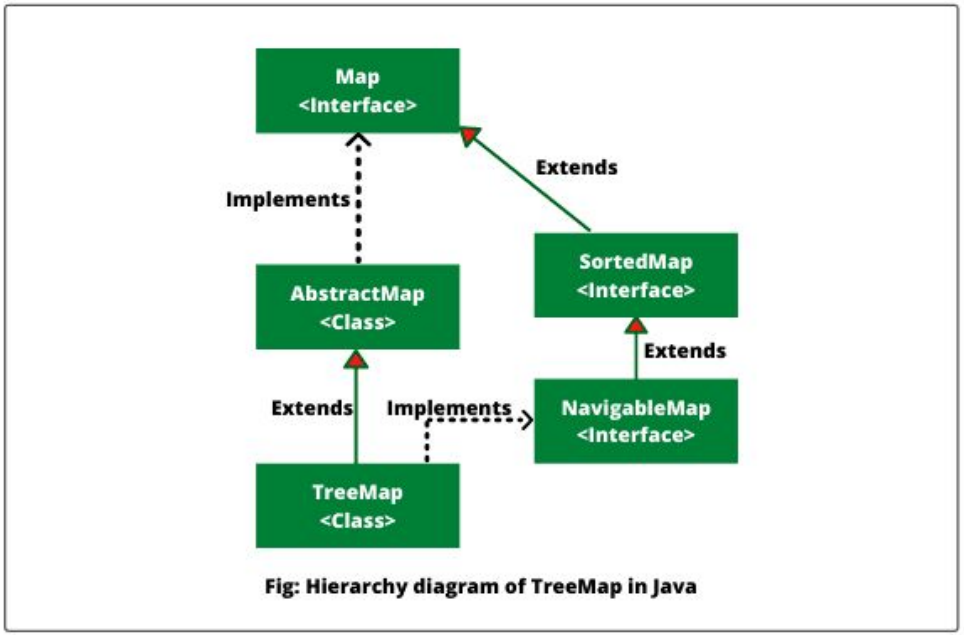

Fig: Hierarchy diagram of TreeMap in Java

`TreeMap` class is the subclass of `AbstractMap` and implements `Map`, `SortedMap` (a subinterface of Map interface), and `NavigableMap` interfaces.

`TreeMap` also implements `Cloneable` and `Serializable` interfaces.

The hierarchy diagram of `TreeMap` in Java is shown in the figure.

TreeMap class Declaration

TreeMap is a generic class that can be declared as follows:

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

Here, K defines the type of keys, and V defines the type of values.

Features of TreeMap

There are several features of `TreeMap` class that you need to keep in mind:

1. The underlying data structure of Java `TreeMap` is a red-black binary search tree.

2. `TreeMap` contains only unique elements.

3. `TreeMap` cannot have a null key but can have multiple null values.

4. `TreeMap` is non synchronized. That means it is not thread-safe. We can create a synchronized version of map by calling Collections.synchronizedMap() on the map.

5. `TreeMap` in Java maintains ascending order. The mappings are sorted in treemap either according to the natural ordering of keys or by a comparator that is provided during the object creation of `TreeMap` depending upon the constructor used.

6. `TreeMap` determines the order of entries by using either `Comparable` interface or `Comparator`.

7. The iterator returned by `TreeMap` is fail-fast. That means we cannot modify map during iteration.

Constructors of TreeMap class

Java TreeMap defines the following constructors. They are as follows:

Constructors of TreeMap class

1. TreeMap(): This constructor is used to create an empty `TreeMap` that will be sorted according to the natural order of its key. All keys added to tree map must implement `Comparable` interface.

The `compareTo()` method in the `Comparable` interface is used to compare keys in the map. If you put a string key into the map whose type is an integer, the put() method will throw an exception named `ClassCastException`.

Constructors of TreeMap class

2. TreeMap(Comparator c): This form of constructor is used to create an empty tree-based map. All keys inserted in the map will be sorted according to the given `Comparator c.`

The `compare()` method in the comparator is used to order the entries in the map based on keys.

Constructors of TreeMap class

3. TreeMap(Map m): This form of constructor is used to initialize a `TreeMap` with entries from `Map m` that will be sorted according to the natural order of the keys.

Constructors of TreeMap class

4. TreeMap(SortedMap sm): This constructor is used to initialize a treemap with the entries from the `SortedMap sm` which will be sorted according to the same ordering as `sm`.

TreeMap methods

Method	Description
void clear()	This method removes all objects (entries) from the tree map.
V put(K key, V value)	This method is used to insert a key/value pair in the tree map.
void putAll(Map m)	It is used to add key/value pairs from Map m to the current tree map.
V remove(Object key)	It is used to remove the key-value pair of the specified key from the tree map.
V get(Object key)	This method is used to retrieve the value associated with key. If key is null, it will throw NullPointerException.
K firstKey()	It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.
K lastKey()	It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.

TreeMap methods

Method	Description
boolean containsKey(Object key)	This method returns true if the tree map contains a particular key.
boolean containsValue(Object value)	This method returns true if the tree map contains a particular value.
int size()	This method returns the number of entries (objects) in the tree map.
Set keySet()	This method returns a set (collection) of all keys of the tree map.
Set entrySet()	This method returns a set view containing all key/value pairs in the tree map.
Collection values()	It returns a collection view of all values contained in the tree map.
Comparator comparator()	It is used to retrieve map's comparator that arranges the key in order, or null if the map uses the natural ordering of elements.
Object clone()	It is used to retrieve the copy of the tree map without cloning its elements.
Map.Entry<K, V> ceilingEntry(K key)	This method returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.

TreeMap methods

Method	Description
K ceilingKey(K key)	This method returns the least key, greater than or equal to the specified key, or null if there is no such key. Both ceilingEntry() and ceilingKey() methods throw ClassCastException if key cannot be compared with the current key in the map. They will throw NullPointerException when key is null because TreeMap does not permit null key.
NavigableSet<K> descendingKeySet()	It returns a reverse order navigable set-based view of the keys contained in the map. The iterator of set returns the keys in descending order.
NavigableMap<K,V> descendingMap()	It returns a reverse order view of this map.
Map.Entry firstEntry()	It returns the key-value pair having the least key in the map or null if the map is empty.

TreeMap methods

Method	Description
Map.Entry<K,V> floorEntry(K key)	It returns a key-value pair associated with the greatest key less than or equal to the specified key, or null when there is no such key.
K floorKey(K Key)	It returns the greatest key less than or equal to the specified key, or null when there is no such key.
SortedMap<K,V> headMap(K toKey)	This method returns the key-value pairs (a view) of that portion of this map whose keys are strictly less than toKey.
NavigableMap<K,V> headMap(K toKey, boolean inclusive)	This method returns key-value pairs (view) of that portion of this map whose keys are less than (or equal to if inclusive is true) toKey. Both headMap() methods will throw ClassCastException if toKey is not compatible with this map's comparator. If to Key is null, this method throws NullPointerException because TreeMap does not allow null key.

TreeMap methods

Method	Description
Map.Entry<K,V> higherEntry(K key)	This method returns a key-value pair or mapping corresponding to the least key strictly greater than the given key, or null if there is no such key.
K higherKey(K key)	It is used to retrieve the least key that is strictly greater than the specified key, or null when there is no such key. Both these methods throw ClassCastException when key cannot be compared with the current key in the map and NullPointerException if key is null.
Map.Entry<K,V> lastEntry()	It returns the key-value pair corresponding to the greatest key in the map, or null if the map is empty or there is no such key.
Map.Entry<K,V> lowerEntry(K key)	This method returns a key-value pair associated with the greatest key strictly less than the specified key, or null if there is no such key.

TreeMap methods

Method	Description
K lowerKey(K key)	It returns the greatest key strictly less than the specified key, or null if there is no such key. Both lowerEntry() and lowerKey() methods throw ClassCastException if key cannot be compared with the keys currently in the map, and NullPointerException when key is null.
NavigableSet<K> navigableKeySet()	It returns a navigable set view of the keys contained in this map.
Map.Entry<K,V> pollFirstEntry()	It removes and returns a key-value pair corresponding to the least key in this map, or null if the map is empty.
Map.Entry<K,V> pollLastEntry()	It removes and returns a key-value pair corresponding to the greatest key in this map, or null if the map is empty.

TreeMap methods

Method	Description
V replace(K key, V value)	It is used to replace the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It is used to replace the old value with the new value for a specified key.
NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	It returns key-value mapping from the map whose keys range from fromKey to toKey..
SortedMap<K,V> subMap(K fromKey, K toKey)	It returns key-value mapping or pairs whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V> tailMap(K fromKey)	This method returns key-value pairs whose keys are greater than or equal to fromKey.
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)	This method returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.

TreeMap Examples

Let’s take various example programs where we will perform some useful operations based on the methods of TreeMap.

Example 1: Creating ordered TreeMaps

1. Let’s take a program where we will create HashMap, LinkedHashMap, and TreeMap objects and see the order of entries of the map.

Example 1: Creating ordered TreeMaps

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.TreeMap;
public class TreeMapTester1 {
   public static void main(String[] args) {
       // Create a HashMap.
       HashMap<String, String> hmap = new HashMap<>();
       hmap.put("R", "Red");
       hmap.put("G", "Green");
       hmap.put("B", "Brown");
       hmap.put("O", "Orange");
       hmap.put("P", "Pink");
       System.out.println("Entries of HashMap: " +hmap);

       // Create a TreeMap from the previous HashMap.
       TreeMap<String, String> tmap = new TreeMap<>(hmap);

       System.out.println("Entries in ascending order of keys: " +tmap);

       // Create a LinkedHashMap.
       LinkedHashMap<String, String> lhmap = new LinkedHashMap<>();

       lhmap.put("R", "Red");
       lhmap.put("G", "Green");
       lhmap.put("B", "Brown");
       lhmap.put("O", "Orange");
       lhmap.put("P", "Pink");

       System.out.println("Entries in LinkedHashMap: " +lhmap);
   }
}
```

Example 1: Creating ordered TreeMaps

Output:

```
Entries of HashMap: {P=Pink, R=Red, B=Brown, G=Green, O=Orange}
Entries in ascending order of keys: {B=Brown, G=Green, O=Orange, P=Pink, R=Red}
Entries in LinkedHashMap: {R=Red, G=Green, B=Brown, O=Orange, P=Pink}
```

As you can see in the output of the program, HashMap does not maintain the order, LinkedHashMap maintains the order of entries as we inserted, while TreeMap sorted entries in ascending order of its keys.

Example 2: Adding, Removing, Replacing Entries

2. Let’s create a program and perform add, remove, and replace operations in the map.

Example 2: Adding, Removing, Replacing Entries

```
import java.util.TreeMap;
public class TreeMapTester2{
   public static void main(String[] args) {
       TreeMap<String, Integer> tmap = new TreeMap<>();
       int size = tmap.size();
       System.out.println("Size of TreeMap before adding entries: " +size);

       boolean isEmpty = tmap.isEmpty();
       System.out.println("Is TreeMap empty: " +isEmpty);

       // Adding entries in treemap.
       tmap.put("John", 25);
       tmap.put("Ricky", 22);
       tmap.put("Deep", 28);
       tmap.put("Mark", 20);
       tmap.put("Peter", 30);

       System.out.println("Entries in ascending order: " +tmap);
       tmap.remove("Mark");
       System.out.println("Entries of TreeMap after removing: " +tmap);

       tmap.replace("Peter", 18);
       System.out.println("Updated entries of TreeMap: " +tmap);
   }
}
```

Example 2: Adding, Removing, Replacing Entries

Output:

```
Size of TreeMap before adding entries: 0
Is TreeMap empty: true
Entries in ascending order: {Deep=28, John=25, Mark=20, Peter=30, Ricky=22}
Entries of TreeMap after removing: {Deep=28, John=25, Peter=30, Ricky=22}
Updated enrties of TreeMap: {Deep=28, John=25, Peter=18, Ricky=22}
```

Example 3: key/value setters/getters and contains

3. Let’s take an example program based on entrySet(), keySet(), values(), get(), containsKey(), and containsValue() methods.

Example 3: key/value setters/getters and contains

```
import java.util.TreeMap;
public class TreeMapTester3{
   public static void main(String[] args) {
       TreeMap<Character, String> tmap = new TreeMap<>();
       tmap.put('A', "Apple");
       tmap.put('P', "Parrot");
       tmap.put('C', "Cat");
       tmap.put('B', "Boy");
       tmap.put('D', "Dog");

       Object entrySet = tmap.entrySet();
       System.out.println("Entry set: " +entrySet);
       System.out.println("Key set: " +tmap.keySet());
       System.out.println("Value set: " +tmap.values());

       Object vGet = tmap.get('C');
       System.out.println("C: " +vGet);

       boolean containsKey = tmap.containsKey('B');
       System.out.println("Is key 'B' present in map: " +containsKey);

       boolean containsValue = tmap.containsValue("Apple");
       System.out.println("Is Apple present in map: " +containsValue);
   }
}
```

Example 3: key/value setters/getters and contains

Output:

```
Entry set: [A=Apple, B=Boy, C=Cat, D=Dog, P=Parrot]
Key set: [A, B, C, D, P]
Value set: [Apple, Boy, Cat, Dog, Parrot]
C: Cat
Is key 'B' present in map: true
Is Apple present in map: true
```

Example 4: Ceiling/Floor, First/Last Entries

4. Let’s take an example program based on ceilingEntry(), ceilingKey(), firstEntry(), lastEntry(), floorEntry() methods.

Example 4: Ceiling/Floor, First/Last Entries

```
import java.util.TreeMap;
public class TreeMapTester4{
   public static void main(String[] args) {
       TreeMap<Integer, String> tmap = new TreeMap<>();
       tmap.put(25, "John");
       tmap.put(22, "Shubh");
       tmap.put(30, "Ricky");
       tmap.put(35, "Peter");
       tmap.put(18, "Johnson");

       System.out.println("ceilingEntry: " +tmap.ceilingEntry(32));
       System.out.println("ceilingKey: " +tmap.ceilingKey(32));

       System.out.println("firstEntry: " +tmap.firstEntry());
       System.out.println("lastEntry: " +tmap.lastEntry());

       System.out.println("floorEntry: " +tmap.floorEntry(31));
       System.out.println("HigherEntry: " +tmap.higherEntry(30));
       System.out.println("LowerEntry: " +tmap.lowerEntry(30));

       System.out.println("pollFirstEntry: " +tmap.pollFirstEntry());
       System.out.println("pollLastEntry: " +tmap.pollLastEntry());
   }
}
```

Example 4: Ceiling/Floor, First/Last Entries

Output:

```
ceilingEntry: 35=Peter
ceilingKey: 35
firstEntry: 18=Johnson
lastEntry: 35=Peter
floorEntry: 30=Ricky
HigherEntry: 35=Peter
LowerEntry: 25=John
pollFirstEntry: 18=Johnson
pollLastEntry: 35=Peter
```

Example 5: headMap, tailMap, subMap

5. Let’s take one more example program based on headMap(), tailMap(), subMap() methods of SortedMap, and NavigableMap interfaces.

Example 5: headMap, tailMap, subMap

```
import java.util.TreeMap;
public class TreeMapTester5{
   public static void main(String[] args) {
       TreeMap<Integer, String> tmap = new TreeMap<>();
       tmap.put(25, "John");
       tmap.put(22, "Shubh");
       tmap.put(30, "Ricky");
       tmap.put(35, "Peter");
       tmap.put(18, "Johnson");
       System.out.println("Sorted tree map: " +tmap);
       // Use methods of NavigableMap interface.
       System.out.println("Descending order of tree map: " +tmap.descendingMap());
       // Returns entries whose keys are less than or equal to the specified key.
       System.out.println("headMap: "+tmap.headMap(22,true));
       // Returns entries whose keys are greater than or equal to the specified key.
       System.out.println("tailMap: "+tmap.tailMap(22,true));
       // Returns entries exists in between the specified key.
       System.out.println("subMap: "+tmap.subMap(18, false, 35, true));
       System.out.println("\n");
       // Use methods of SortedMap interface.
       // Returns entries whose keys are less than the specified key.
       System.out.println("headMap: "+tmap.headMap(40));
       // Returns entries whose keys are greater than or equal to the specified key.
       System.out.println("tailMap: "+tmap.tailMap(22));
       // Returns entries exists in between the specified key.
       System.out.println("subMap: "+tmap.subMap(19, 25));
   }
}
```

Example 5: headMap, tailMap, subMap

Output:

```
Sorted tree map: {18=Johnson, 22=Shubh, 25=John, 30=Ricky, 35=Peter}
Descending order of tree map: {35=Peter, 30=Ricky, 25=John, 22=Shubh, 18=Johnson}
headMap: {18=Johnson, 22=Shubh}
tailMap: {22=Shubh, 25=John, 30=Ricky, 35=Peter}
subMap: {22=Shubh, 25=John, 30=Ricky, 35=Peter}


headMap: {18=Johnson, 22=Shubh, 25=John, 30=Ricky, 35=Peter}
tailMap: {22=Shubh, 25=John, 30=Ricky, 35=Peter}
subMap: {22=Shubh}
```

When to use TreeMap in Java?

TreeMap is slower than HashMap but it is preferred:

- When we do not want null key in the map.
- When we want the order of entries in sorted ascending order.

Key points

1. HashTable is suitable when you are not working in a multithreading environment.

2. HashMap is slightly better than HashTable but it is not thread-safe. It is suitable if the order of elements is not an issue.

3. TreeMap is slower than HashMap but it is suitable when you need the map in sorted ascending order.

4. LinkedHashMap is also slower than HashMap, but it is preferred if more number of insertions and deletions happen on the map.

END