SortedMap

14 May 2025 10:09

It is the child interface of Map interface. If we want to represent a group of key - to some sorting order of keys. then we should go for SortedMap interface. Sorting is based on key, but not based on value.

Map(I)
|
SortedMap(I)

Object firstKey() → 101

Object lastKey() → 121

SortedMap headMap(112) → {101=A, 105=B, 109=C, 110=D}

SortedMap tailMap(112) → {112=E, 113=F, 115=G, 121=H}

SortedMap subMap(109, 115) → {109=C, 110=D, 112=E, 113=F}

Comparator comparator()

K	V
101	A
105	B
109	C
110	D
112	E
113	F
115	G
121	H

SortedMap (I)
|
NavigableMap(I)
|
TreeMap.

Red-Black Tree properties

1. Every node is either red or black.
2. The root is always black.
3. All leaves (null pointers) are considered black.
4. Red nodes cannot have red children.
5. Every path from a node to its decendent leaves contains the same number of black nodes. This is called black-height of the tree.

Why use Red-Black Trees?

⇒ To maintain balance while performing inserts and deletes

⇒ To guarantee logarithmic height

⇒ Used in many real-world aplication like Java's TreeMap, C++ STL's map and set. Linux process scheduler, etc.

public interface SortedMap<K,V>
extends SequencedMap<K,V>

A Map that further provides a total ordering on its keys. The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of SortedSet.)

All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such keys must be mutually comparable: k1.compareTo(k2) (or comparator.compare(k1, k2)) must not throw a ClassCastException for any keys k1 and k2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be consistent with equals if the sorted map is to correctly implement the Map interface. (See the Comparable interface or Comparator interface for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a tree map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

All general-purpose sorted map implementation classes should provide four "standard" constructors. It is not possible to enforce this recommendation though as required constructors cannot be specified by interfaces. The expected "standard" constructors for all sorted map implementations are:

1. A void (no arguments) constructor, which creates an empty sorted map sorted according to the natural ordering of its keys.
2. A constructor with a single argument of type Comparator, which creates an empty sorted map sorted according to the specified comparator.
3. A constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument, sorted according to the keys' natural ordering.
4. A constructor with a single argument of type SortedMap, which creates a new sorted map with the same key-value mappings and the same ordering as the input sorted map.

Note: several methods return submaps with restricted key ranges. Such ranges are half-open, that is, they include their low endpoint but not their high endpoint (where applicable). If you need a closed range (which includes both endpoints), and the key type allows for calculation of the successor of a given key, merely request the subrange from lowEndpoint to successor(highEndpoint). For example, suppose that m is a map whose keys are strings. The following idiom obtains a view containing all of the key-value mappings in m whose keys are between low and high, inclusive:

```
SortedMap<String, V> sub = m.subMap(low, high+"\0");
```

A similar technique can be used to generate an open range (which contains neither endpoint). The following idiom obtains a view containing all of the key-value mappings in m whose keys are between low and high, exclusive:

```
SortedMap<String, V> sub = m.subMap(low+"\0", high);
```

This interface is a member of the Java Collections Framework.

Since:
1.2

TreeMap

- **The underlying data structure is red-black tree.**

Concept of RED-BLACK Tree:

What is a Red-Black Tree?

A Red-Black Tree is a self-balancing binary search tree where each node has an additional attribute: a color, which can be either red or black. The primary objective of these trees is to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.
Properties of Red-Black Trees

A Red-Black Tree have the following properties:

Node Color: Each node is either red or black.
Root Property: The root of the tree is always black.
Red Property: Red nodes cannot have red children (no two consecutive red nodes on any path).
Black Property: Every path from a node to its descendant null nodes (leaves) has the same number of black nodes.
Leaf Property: All leaves (NIL nodes) are black.

These properties ensure that the longest path from the root to any leaf is no more than twice as long as the shortest path, maintaining the tree’s balance and efficient performance.

- Insertion order is not preserved and it is based on some sorting order of keys.
- Duplicate keys are not allowed, but values can be duplicated.
- If we are depending on default natural sorting order then. keys should be homogeneous and comparable. Otherwise we will get runtime exception, saying, ClassCastException.
  If we are defining our own sorting by comparator, then keys needed not to be homogeneous and Comparable. We can take heterogeneous, non-comparable objects also.
- Whether we are depending on default natural sorting order or customized sorting order, there are no restriction for values. We can store heterogeneous, non-comparable objects also.
- Null is not accepted at all in keys.

Constructors of TreeMap
1. TreeMap t = new TreeMap(); //DNSO of key
2. TreeMap t = new TreeMap(Comparator c); //for customized sorting order
3. TreeMap t = new TreeMap(Map m);
4. TreeMap t = new TreeMap(SortedMap m);

Official Documentation:

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

A Red-Black tree based NavigableMap implementation. 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 implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

Note that this implementation is not synchronized. If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with an existing key is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedSortedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

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

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly

and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

The methods ceilingEntry(K), firstEntry(), floorEntry(K), higherEntry(K), lastEntry(), lowerEntry(K), pollFirstEntry(), and pollLastEntry() return Map.Entry instances that represent snapshots of mappings as of the time of the call. They do not support mutation of the underlying map via the optional setValue method.

The putFirst and putLast methods of this class throw UnsupportedOperationException. The encounter order of mappings is determined by the comparison method; therefore, explicit positioning is not supported.

This class is a member of the Java Collections Framework.

Since:
1.2

Constructors

Constructor	Description
`TreeMap()`	Constructs a new, empty tree map, using the natural ordering of its keys.
`TreeMap(Comparator<? super K> comparator)`	Constructs a new, empty tree map, ordered according to the given comparator.
`TreeMap(Map<? extends K, ? extends V> m)`	Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
`TreeMap(SortedMap<K, ? extends V> m)`	Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`Map.Entry<K,V>`	`ceilingEntry(K key)`	Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
`K`	`ceilingKey(K key)`	Returns the least key greater than or equal to the given key, or `null` if there is no such key.
`void`	`clear()`	Removes all of the mappings from this map.
`Object`	`clone()`	Returns a shallow copy of this `TreeMap` instance.
`Comparator<? super K>`	`comparator()`	Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
`V`	`compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`	Attempts to compute a mapping for the specified key and its current mapped value, or `null` if there is no current mapping (optional operation).
`V`	`computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)`	If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null` (optional operation).
`V`	`computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`	If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value (optional operation).
`boolean`	`containsKey(Object key)`	Returns `true` if this map contains a mapping for the specified key.

boolean	containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
NavigableSet<K>	descendingKeySet()	Returns a reverse order NavigableSet view of the keys contained in this map.
NavigableMap<K,V>	descendingMap()	Returns a reverse order view of the mappings contained in this map.
Set<Map.Entry<K,V>>	entrySet()	Returns a Set view of the mappings contained in this map.
Map.Entry<K,V>	firstEntry()	Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
K	firstKey()	Returns the first (lowest) key currently in this map.
Map.Entry<K,V>	floorEntry(K key)	Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
K	floorKey(K key)	Returns the greatest key less than or equal to the given key, or null if there is no such key.
V	get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
SortedMap<K,V>	headMap(K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
NavigableMap<K,V>	headMap(K toKey, boolean inclusive)	Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.

Map.Entry<K,V>	higherEntry(K key)	Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.
K	higherKey(K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
Set<K>	keySet()	Returns a Set view of the keys contained in this map.
Map.Entry<K,V>	lastEntry()	Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
K	lastKey()	Returns the last (highest) key currently in this map.
Map.Entry<K,V>	lowerEntry(K key)	Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K	lowerKey(K key)	Returns the greatest key strictly less than the given key, or null if there is no such key.
V	merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value (optional operation).
NavigableSet<K>	navigableKeySet()	Returns a NavigableSet view of the keys contained in this map.
Map.Entry<K,V>	pollFirstEntry()	Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty (optional operation).

Map.Entry<K,V>	pollLastEntry()	Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty (optional operation).
V	put(K key, V value)	Associates the specified value with the specified key in this map.
void	putAll(Map<? extends K, ? extends V> map)	Copies all of the mappings from the specified map to this map.
V	putFirst(K k, V v)	Throws UnsupportedOperationException.
V	putLast(K k, V v)	Throws UnsupportedOperationException.
V	remove(Object key)	Removes the mapping for this key from this TreeMap if present.
int	size()	Returns the number of key-value mappings in this map.
NavigableMap<K,V>	subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	Returns a view of the portion of this map whose keys range from fromKey to toKey.
SortedMap<K,V>	subMap(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V>	tailMap(K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
NavigableMap<K,V>	tailMap(K fromKey, boolean inclusive)	Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.
Collection<V>	values()	Returns a Collection view of the values contained in this map.

Code:

CollectionFramework > Map > TreeMapDemo.java > TreeMapDemo > main(String[])

```java
import java.util.TreeMap;

public class TreeMapDemo {
    Run | Debug
    public static void main(String[] args) {
        TreeMap m = new TreeMap();

        m.put(key:100, value:"Aman");
        m.put(key:87, value:"Naman");
        m.put(key:112, value:"Aditya");
        m.put(key:95, value:"Raghav");
        m.put(key:200, value:"Naveen");
        m.put(key:98, value:null);
        m.put(key:null, value:null);

        System.out.println(m);
    }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException
        at java.base/java.util.Objects.requireNonNull(Objects.java:233)
        at java.base/java.util.TreeMap.put(TreeMap.java:809)
        at java.base/java.util.TreeMap.put(TreeMap.java:534)
        at TreeMapDemo.main(TreeMapDemo.java:13)
```

```java
import java.util.TreeMap;
public class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap m = new TreeMap();
        m.put(100, "Aman");
        m.put(87, "Naman");
        m.put(112, "Aditya");
        m.put(95, "Raghav");
        m.put(200, "Naveen");
        m.put(98, null);
        // m.put(null, null);
        System.out.println(m);
    }
}
```

Output:

{87=Naman, 95=Raghav, 98=null, 100=Aman, 112=Aditya, 200=Naveen} – Sorted with keys

```java
import java.util.TreeMap;

public class TreeMapDemo {
    Run | Debug
    public static void main(String[] args) {
        TreeMap m = new TreeMap();

        m.put(key:100, value:"Aman");
        m.put(key:87, value:"Naman");
        m.put(key:112, value:"Aditya");
        m.put(key:95, value:"Raghav");
        m.put(key:200, value:"Naveen");
        m.put(key:98, value:null);
        // m.put(null, null);

        System.out.println(m);

        m.put(key:"Varun", value:"Anuj");

        System.out.println(m);
    }
}
```

Throwable

Exception — Error

RuntimeException — ClassCastException

Hetrogenous object in key is not allowed.

RuntimeException

Output:

```
    }
}
```

Heterogenous objects not allowed → RuntimeException

Output:

```
{87=Naman, 95=Raghav, 98=null, 100=Aman, 112=Aditya, 200=Naveen}
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class
java.lang.String (java.lang.Integer and java.lang.String are in module java.base of loader 'bootstrap')
        at java.base/java.lang.String.compareTo(String.java:141)
        at java.base/java.util.TreeMap.put(TreeMap.java:814)
        at java.base/java.util.TreeMap.put(TreeMap.java:534)
        at TreeMapDemo.main(TreeMapDemo.java:17)
```

```
import java.util.Comparator;
public class MyCompartor implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        return o2.toString().compareTo(o1.toString());
    }
}
```

```
import java.util.TreeMap;
public class TreeMapComparatorDemo {
    public static void main(String[] args) {
        TreeMap t = new TreeMap<>(new MyCompartor());
        t.put("Aditya", "Anchit");
        t.put("Aman", "Naman");
        t.put("Bhavvya", "Riddhi");
        t.put(125, "Surya");
        t.put("Shiv", "Students");
        System.out.println(t);
    }
}
```

Output:

```
{Shiv=Students, Bhavvya=Riddhi, Aman=Naman, Aditya=Anchit, 125=Surya}
```

← Sorted in descending order of 'keys'.

Hashtable

- The underlying data structure for hash table is hash table.
- Insertion order is not preserved and it is based on hash code of keys.
- Duplicate keys are not allowed and values can be duplicated.
- Heterogeneous objects are allowed for both keys and values.
- Null is not allowed for both key and values. Otherwise we will get runtime exception, saying, "NullPointerException".
- It implements Map<k><v>, Serializable and Cloneable interfaces, but not RandomAccess interface.
- Every method present in Hashtable is synchronized and hence Hashtable object is thread safe.
- Hash table is the best choice if our frequent operation is retrieval or search operation.

Constructors:

1. Hashtable h = new Hashtable(); ==> Creates an empty hash table object with default initial capacity 11 and default fill ratio, 0.75.

2. Hashtable h = new Hashtable(int initialCapacity, float fillRatio);

3. Hashtable h = new Hashtable(int initialCapacity);

4. Hashtable h = new Hashtable(Map m); ==>Interconversion between Map object.

Official Documentation:
public class Hashtable<K,V>
extends Dictionary<K,V>

implements Map<K,V>, Cloneable, Serializable
This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. Note that the hash table is open: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.

Generally, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put).

The initial capacity controls a tradeoff between wasted space and the need for rehash operations, which are time-consuming. No rehash operations will ever occur if the initial capacity is greater than the maximum number of entries the Hashtable will contain divided by its load factor. However, setting the initial capacity too high can waste space.

If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

This example creates a hashtable of numbers. It uses the names of the numbers as keys:

```
Hashtable<String, Integer> numbers
  = new Hashtable<String, Integer>();
numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
```

To retrieve a number, use the following code:

```
Integer n = numbers.get("two");
if (n != null) {
  System.out.println("two = " + n);
}
```

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the Hashtable is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The Enumerations returned by Hashtable's keys and elements methods are not fail-fast; if the Hashtable is structurally modified at any time after the enumeration is created then the results of enumerating are undefined.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

As of the Java 2 platform v1.2, this class was retrofitted to implement the Map interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Hashtable is synchronized. If a thread-safe implementation is not needed, it is recommended to use HashMap in place of Hashtable. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use ConcurrentHashMap in place of Hashtable.

Since:
    1.0

Code:

```java
import java.util.Hashtable;
public class HashTableDemo {
    public static void main(String[] args) {
        Hashtable<Key, String> h = new Hashtable<Key, String>(25);
        h.put(new Key(5), "A");
        h.put(new Key(2), "B");
        h.put(new Key(6), "C");
        h.put(new Key(15), "D");
        h.put(new Key(17), "E");
        h.put(new Key(23), "F");
        h.put(new Key(15), "G");
        h.put(new Key(17), "H");
        h.put(new Key(19), null);
        System.out.println(h);
    }
}
```

'null' concept is not allowed for 'key' or 'value'

Output:

```
In equals:
In equals:
Exception in thread "main" java.lang.NullPointerException
        at java.base/java.util.Hashtable.put(Hashtable.java:476)
        at HashTableDemo.main(HashTableDemo.java:14)
PS C:\Users\bluej\Documents\Students\RaghavISC\Java4Sem\Coding>
```

```java
import java.util.Hashtable;
public class HashTableDemo {
    public static void main(String[] args) {
        Hashtable<Key, String> h = new Hashtable<Key, String>(25);
        h.put(new Key(5), "A");
        h.put(new Key(2), "B");
        h.put(new Key(6), "C");
        h.put(new Key(15), "D");
        h.put(new Key(17), "E");
        h.put(new Key(23), "F");
        h.put(new Key(15), "G");
        h.put(new Key(17), "H");
        // h.put(new Key(19), null);
        System.out.println(h);
        Key k = new Key(15);
        System.out.println("Value of "+k+" is: "+h.get(k));
    }
}
```

```java
public class Key {
    private int i;
    public Key(int i) {
        this.i = i;
    }
    @Override
    public int hashCode() {
        return i;
    }
    @Override
    public boolean equals(Object obj) {
        Key k = null;
        if(obj instanceof Key){
            k = (Key)obj;
        }
        System.out.println("In equals ");
        if(this.i == k.i){
            return true;
        }
        else{
```

implement any hashing algorithm here.

Overriding of these methods are compulsory to smooth running of Hashtable operations.

```
            if(this.i == k.i){
                return true;
            }
            else{
                return false;
            }
        }
        @Override
        public String toString() {
            return "Key [i=" + i + "]";
        }

    }
```

Output:

```
In equals
In equals
{Key [i=23]=F, Key [i=17]=H, Key [i=15]=G, Key [i=6]=C, Key [i=5]=A, Key [i=2]=B}
In equals
Value of Key [i=15] is: G
```