# Understanding SortedMap and TreeMap

TreeMap doesn’t only implement the Map interface, it also implements the SortedMap and NavigableMap interfaces. Therefore, besides the behaviors inherited from the Map, TreeMap also inherits the behaviors defined by SortedMap and NavigableMap. The following picture depicts the API hierarchy of TreeMap:

A map with text and a tree map

Description automatically generated with medium confidence

Understanding SortedMap:

The main characteristic of a SortedMap is that, it orders the keys by their natural ordering, or by a specified comparator. So consider using a TreeMap when you want a map that satisfies the following criteria:

* null key or null value are not permitted.
* The keys are sorted either by natural ordering or by a specified comparator.

The following example realizes the concept of a SortedMap: SortedMap mapDomains = new TreeMap<>(); mapDomains.put(".com", "International"); mapDomains.put(".us", "United States"); mapDomains.put(".uk", "United Kingdom"); mapDomains.put(".jp", "Japan"); mapDomains.put(".au", "Australia");

System.out.println(mapDomains); Output:

{.au=Australia, .com=International, .jp=Japan, .uk=United Kingdom, .us=United States}

Here, this map contains mappings of domain=country, and as we see in the output, the domains (keys) are sorted by alphabetic order (natural ordering of Strings).

Besides the operations inherited from the Map interface, the SortedMap also defines the following operations:

* **Range view**: returns a sub sorted map whose keys fall within a range of keys in the original map.
* **Endpoints**: returns the first or last key in the sorted map.
* **Comparator access**: returns the comparator (implements the Comparator interface), if any, used to sort the map.

Hence the following code is the definition of a SortedMap:

public interface SortedMap extends Map{

Comparator comparator();

SortedMap subMap(K fromKey, K toKey);

SortedMap headMap(K toKey);

SortedMap tailMap(K fromKey);

K firstKey();

K lastKey();

}

Let’s look at each type of operation in details.

Range View Operations:

* **+ subMap(K fromKey, K toKey):** returns a sorted map whose keys range from fromKey, inclusive, to toKey, exclusive.
* **+ headMap(K toKey):** returns a sorted map whose keys are strictly less than toKey.
* **+ tailMap(K fromKey):** returns a sorted map whose keys are greater than or equal to fromKey.

Endpoint operations:

* **+ firstKey():** returns the first (lowest) key currently in the map.
* **+ lastKey():** returns the last (highest) key currently in the map.

Comparator access:

* **+ comparator():** returns the comparator used to order the keys in the map, or returns null if this map uses the natural ordering of its keys.

The following code example demonstrates how to work with these operations on a TreeMap: SortedMap mapHttpStatus = new TreeMap<>(); mapHttpStatus.put(100, "Continue"); mapHttpStatus.put(200, "OK");

mapHttpStatus.put(300, "Multiple Choices"); mapHttpStatus.put(400, "Bad Request"); mapHttpStatus.put(401, "Unauthorized"); mapHttpStatus.put(402, "Payment Required"); mapHttpStatus.put(403, "Forbidden"); mapHttpStatus.put(404, "Not Found"); mapHttpStatus.put(500, "Internal Server Error"); mapHttpStatus.put(501, "Not Implemented"); mapHttpStatus.put(502, "Bad Gateway");

System.out.println("All key-value pairs: "); for (Integer code : mapHttpStatus.keySet()) {

System.out.println(code + " -> " + mapHttpStatus.get(code));

}

System.out.println();

Integer firstKey = mapHttpStatus.firstKey();

String firstValue = mapHttpStatus.get(firstKey);

System.out.println("First status: " + firstKey + " -> " + firstValue); System.out.println();

Integer lastKey = mapHttpStatus.lastKey();

String lastValue = mapHttpStatus.get(lastKey);

System.out.println("Last status: " + lastKey + " -> " + lastValue);

System.out.println();

SortedMap map4xxStatus = mapHttpStatus.subMap(400, 500); System.out.println("4xx Statuses: "); for (Integer code : map4xxStatus.keySet()) {

System.out.println(code + " -> " + map4xxStatus.get(code));

}

System.out.println();

SortedMap mapUnder300Status = mapHttpStatus.headMap(300); System.out.println("Statuses < 300: "); for (Integer code : mapUnder300Status.keySet()) {

System.out.println(code + " -> " + mapUnder300Status.get(code));

}

System.out.println();

SortedMap mapAbove500Status = mapHttpStatus.tailMap(500); System.out.println("Statuses > 500: "); for (Integer code : mapAbove500Status.keySet()) {

System.out.println(code + " -> " + mapAbove500Status.get(code));

}

Comparator comparator = mapHttpStatus.comparator();

System.out.println("Sorted by natural ordering? " + (comparator == null)); Output:

All key-value pairs:

100 -> Continue

200 -> OK

300 -> Multiple Choices

1. -> Bad Request
2. -> Unauthorized
3. -> Payment Required
4. -> Forbidden
5. -> Not Found
6. -> Internal Server Error
7. -> Not Implemented
8. -> Bad Gateway

First status: 100 -> Continue

Last status: 502 -> Bad Gateway 4xx Statuses:

1. -> Bad Request
2. -> Unauthorized
3. -> Payment Required
4. -> Forbidden 404 -> Not Found

Statuses < 300:

100 -> Continue

200 -> OK

Statuses > 500:

1. -> Internal Server Error
2. -> Not Implemented
3. -> Bad Gateway

Sorted by natural ordering? true

And the following example shows how to use a comparator:

SortedMap mapHttpStatus = new TreeMap<>(new ReverseComparator()); mapHttpStatus.put(100, "Continue"); mapHttpStatus.put(200, "OK"); mapHttpStatus.put(300, "Multiple Choices"); mapHttpStatus.put(400, "Bad Request"); mapHttpStatus.put(401, "Unauthorized"); mapHttpStatus.put(402, "Payment Required"); mapHttpStatus.put(403, "Forbidden"); mapHttpStatus.put(404, "Not Found"); mapHttpStatus.put(500, "Internal Server Error"); mapHttpStatus.put(501, "Not Implemented"); mapHttpStatus.put(502, "Bad Gateway");

for (Integer code : mapHttpStatus.keySet()) {

System.out.println(code + " -> " + mapHttpStatus.get(code)); }

Here’s the code of the comparator class:

class ReverseComparator implements Comparator { public int compare(Integer num1, Integer num2) { return num2.compareTo(num1);

}

} Output:

502 -> Bad Gateway

501 -> Not Implemented

500 -> Internal Server Error

404 -> Not Found

403 -> Forbidden

402 -> Payment Required

401 -> Unauthorized

400 -> Bad Request

300 -> Multiple Choices

200 -> OK

100 -> Continue

As you can see, this comparator sorts the map by the descending order of its keys.

In case you are working on Java 8, use Lambda expressions to shorten the comparator code like this:

SortedMap mapHttpStatus = new TreeMap<>((i1, i2) -> i2.compareTo(i1));

# Understanding NavigableMap and TreeMap

Understanding NavigableMap:

NavigableMap is sub interface of SortedMap interface, so it inherits all behaviors of a sorted map like range view, endpoints and comparator access operations. In addition, the NavigableMap interface provides navigation methods and descending views like NavigableSet. And due to the nature of a map which stores key-value mappings, the additional APIs are designed for both keys and key-value entries in the map.

Let’s look at these methods in details.

Operations on key-value mappings (entries):

* **lowerEntry(K key):** returns a key-value mapping associated with the greatest key strictly less than the given key.
* **floorEntry(K key):** returns a key-value mapping entry which is associated with the greatest key less than or equal to the given key.
* **ceilingEntry(K key):** returns an entry associated with the lest key greater than or equal to the given key.
* **higherEntry(K key):** returns an entry associated with the least key strictly greater than the given key.
* Note that all these methods return null if there is no such key.
* **firstEntry():** returns a key-value mapping associated with the least key in the map, or null if the map is empty.
* **lastEntry():** returns a key-value mapping associated with the greatest key in the map, or null if the map is empty.
* **descendingMap():** returns a reverse order view of the mappings contained in the map.
* **pollFirstEntry():** removes and returns a key-value mapping associated with the least key in the map, or null if the map is empty.
* **pollLastEntry():** removes and returns a key-value mapping associated with the greatest key in the map, or

null if the map is empty.

Operations on keys only:

* **lowerKey(K key):** returns the greatest key strictly less than the given key.
* **floorKey(K key):** returns the greatest key less than or equal to the given key.
* **ceilingKey(K key):** returns the least key greater than or equal to the given key.
* **higherKey(K key):** returns the least key strictly greater than the given key.
* **descendingKeySet():** returns a NavigableSet containing the keys in reverse order.

Note that all these methods return null if there is no such key.

Furthermore, the NavigableMap interface overloads the headMap(), subMap() and tailMap() methods of the SortedMap interface, which accept additional arguments describing whether lower or upper bounds are inclusive versus exclusive:

* + headMap(K toKey, boolean inclusive)
  + subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) - tailMap(K fromKey, boolean inclusive) Now, let’s look at some code examples.

NavigableMap Examples with TreeMap:

The following example shows us how to obtain the reverse order view of the keys in a map:

NavigableMap mapHttpStatus = new TreeMap<>();

mapHttpStatus.put(100, "Continue"); mapHttpStatus.put(200, "OK"); mapHttpStatus.put(400, "Bad Request"); mapHttpStatus.put(401, "Unauthorized"); mapHttpStatus.put(500, "Internal Server Error"); mapHttpStatus.put(501, "Not Implemented");

Set ascendingKeys = mapHttpStatus.keySet();

System.out.println("Ascending Keys: " + ascendingKeys);

Set descendingKeys = mapHttpStatus.descendingKeySet();

System.out.println("Descending Keys: " + descendingKeys); Output:

Ascending Keys: [100, 200, 400, 401, 500, 501]

Descending Keys: [501, 500, 401, 400, 200, 100]

Given the above map, the following code snippet gets a reverse order view of the map: NavigableMap descendingMap = mapHttpStatus.descendingMap(); for (Integer key : descendingMap.keySet()) {

System.out.println(key + " => " + descendingMap.get(key)); } Output:

501 => Not Implemented

500 => Internal Server Error

401 => Unauthorized

400 => Bad Request

200 => OK

100 => Continue

Using operations on keys is explained in the following example:

Integer lowerKey = mapHttpStatus.lowerKey(401); System.out.println("Lower key: " + lowerKey);

Integer floorKey = mapHttpStatus.floorKey(401);

System.out.println("Floor key: " + floorKey);

Integer higherKey = mapHttpStatus.higherKey(500);

System.out.println("Higher key: " + higherKey);

Integer ceilingKey = mapHttpStatus.ceilingKey(500);

System.out.println("Ceiling key: " + ceilingKey); Output:

Lower key: 400

Floor key: 401

Higher key: 501

Ceiling key: 500

And the following example demonstrates how to work with operations on key-value mapping entries:

Map.Entry firstEntry = mapHttpStatus.firstEntry();

System.out.println("First entry: " + firstEntry.getKey() + " => " + firstEntry.getValue());

Map.Entry lastEntry = mapHttpStatus.lastEntry();

System.out.println("Last entry: " + lastEntry.getKey() + " => " + lastEntry.getValue());

Map.Entry lowerEntry = mapHttpStatus.lowerEntry(401);

System.out.println("Lower entry: " + lowerEntry.getKey() + " => " + lowerEntry.getValue()); Map.Entry floorEntry = mapHttpStatus.floorEntry(401);

System.out.println("Floor entry: " + floorEntry.getKey() + " => " + floorEntry.getValue());

Map.Entry higherEntry = mapHttpStatus.higherEntry(500);

System.out.println("Higher entry: " + higherEntry.getKey() + " => " + higherEntry.getValue());

Map.Entry ceilingEntry = mapHttpStatus.ceilingEntry(500);

System.out.println("Ceiling entry: " + ceilingEntry.getKey() + " => " + ceilingEntry.getValue());

mapHttpStatus.pollFirstEntry(); mapHttpStatus.pollLastEntry();

System.out.println("\nMap after first and last entries were polled:"); for (Integer key : mapHttpStatus.keySet()) {

System.out.println(key + " => " + mapHttpStatus.get(key)); }

Output:

First entry: 100 => Continue

Last entry: 501 => Not Implemented

Lower entry: 400 => Bad Request

Floor entry: 401 => Unauthorized

Higher entry: 501 => Not Implemented Ceiling entry: 500 => Internal Server Error

Map after first and last entries were polled:

200 => OK

1. => Bad Request
2. => Unauthorized 500 => Internal Server Error