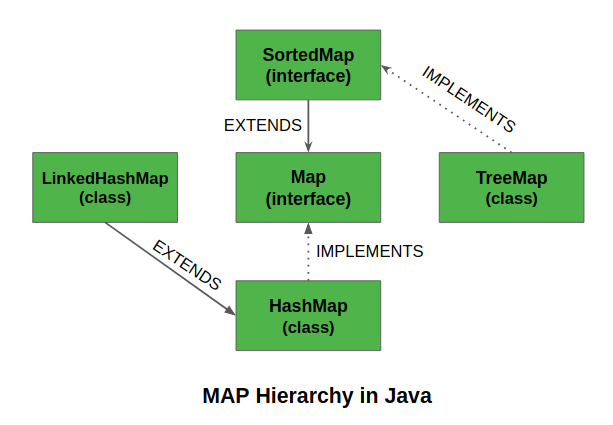
**TreeMap in Java**

The TreeMap in Java is used to implement [Map interface](https://www.geeksforgeeks.org/map-interface-java-examples/) and [NavigableMap](https://www.geeksforgeeks.org/navigablemap-interface-in-java-with-example/" \t "_blank) along with the AbstractMap Class. The map is sorted according to the natural ordering of its keys, or by a [Comparator](https://www.geeksforgeeks.org/comparator-interface-java/) 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.  
  
[](https://contribute.geeksforgeeks.org/wp-content/uploads/Selection_030.png)**Important Methods in TreeMap and there working.**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Time Complexity** |
| [**put(Object key, Object value)**](https://www.geeksforgeeks.org/treemap-put-method-in-java/) | **The method is used to insert a mapping into a map.** | **O(log n)** |
| [**remove(Object key)**](https://www.geeksforgeeks.org/treemap-remove-method-in-java/) | **Removes the mapping for this key from this TreeMap if present.** | **O(log n)** |
| [**containsKey(Object key)**](https://www.geeksforgeeks.org/treemap-containskey-method-in-java/) | **Returns true if this map contains a mapping for the specified key.** | **O(log n)** |

**Example 1:** Working of put(), containsKey() and Traversal of TreeMap.

java

*// Java program to demonstrate*

*// the working of TreeMap*

**import** **java.util.\***;

**class** **GFG** {

**public** **static** void main(String args[])

{

*// Initialization of a TreeMap*

*// using Generics*

TreeMap<Integer, String> t

= **new** TreeMap<Integer, String>();

*// Inserting the Elements*

t.put(10, "geeks");

t.put(15, "ide");

t.put(5, "courses");

*// Prints the TreeMap*

System.out.println(t);

*// Check for the key in the map*

System.out.println(t.containsKey(10));

*// Iterating over TreeMap*

**for**(Map.Entry<Integer, String>e : t.entrySet())

System.out.println(e.getKey() + " " + e.getValue());

}

}

**Output:**

{5=courses, 10=geeks, 15=ide}

true

5 courses

10 geeks

15 ide

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Time Complexity** |
| [**remove(Object key)**](https://www.geeksforgeeks.org/treemap-remove-method-in-java/) | **Removes the mapping for this key from this TreeMap if present.** | **O(log n)** |
| [**size()**](https://www.geeksforgeeks.org/treemap-size-method-in-java/) | **Returns the number of key-value mappings in this map.** | **O(1)** |

**Example 2:** Working of remove() and size().

java

*// Java program to demonstrate*

*// the working of TreeMap*

**import** **java.util.\***;

**class** **GFG** {

**public** **static** void main(String args[])

{

*// Initialization of a TreeMap*

*// using Generics*

TreeMap<Integer, String> t

= **new** TreeMap<Integer, String>();

*// Inserting the Elements*

t.put(10, "geeks");

t.put(15, "ide");

t.put(5, "courses");

*// Removing <10, "geeks">*

t.remove(10);

*// Displaying the size of map*

System.out.println(t.size());

}

}

**Output:**

2

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Time Complexity** |
| [**higherKey(K key)**](https://www.geeksforgeeks.org/treemap-higherkey-method-in-java-with-examples/) | **Used to return the least key strictly greater than the given key, or null if there is no such key.** | **O(log n)** |
| [**lowerKey(K key)**](https://www.geeksforgeeks.org/treemap-lowerkey-in-java-with-examples/) | **Used to return the greatest key strictly less than to given key, passed as the parameter** | **O(log n)** |
| [**floorKey(K key)**](https://www.geeksforgeeks.org/treemap-floorkey-in-java-with-examples/) | **Used to return the greatest key less than or equal to given key from the parameter.** | **O(log n)** |
| [**ceilingKey(K key)**](https://www.geeksforgeeks.org/treemap-ceilingkey-in-java-with-examples/) | **Used to return the least key greater than or equal to the given key or null if the such a key is absent.** | **O(log n)** |

**Example 3:** Working of higherKey(), lowerKey(), floorKey(), ceilingKey().

java

*// Java program to demonstrate*

*// the working of TreeMap*

**import** **java.util.\***;

**class** **GFG** {

**public** **static** void main(String args[])

{

*// Initialization of a TreeMap*

*// using Generics*

TreeMap<Integer, String> t

= **new** TreeMap<Integer, String>();

*// Inserting the Elements*

t.put(10, "geeks");

t.put(15, "ide");

t.put(5, "courses");

*// returns the smallest key greater*

*// than the passed key i.e., 15*

System.out.println(t.higherKey(10));

*// returns the greatest smaller key*

*// than the passed key i.e., 5*

System.out.println(t.lowerKey(10));

*// greatest key <= passed key i.e., 10*

System.out.println(t.floorKey(10));

*// greatest key >= passed key i.e., 10*

System.out.println(t.ceilingKey(10));

}

}

**Output:**

15

5

10

10

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Time Complexity** |
| [**higherEntry(K key)**](https://www.geeksforgeeks.org/treemap-higherentry-method-in-java-with-examples/) | **Return a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.** | **O(log n)** |
| [**lowerEntry(K key)**](https://www.geeksforgeeks.org/treemap-lowerentry-method-in-java-with-examples/) | **return a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key** | **O(log n)** |
| [**floorEntry(K Key)**](https://www.geeksforgeeks.org/java-util-treemap-floorentry-floorkey-java/) | **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.** | **O(log n)** |
| [**ceilingEntry(K Key)**](https://www.geeksforgeeks.org/treemap-ceilingentry-and-ceilingkey-methods-in-java/) | **return 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.** | **O(log n)** |

**Example 4:** Working of higherEntry(), lowerEntry(), floorEntry(), ceilingEntry() and getValue().

java

*// Java program to demonstrate*

*// the working of TreeMap*

**import** **java.util.\***;

**class** **GFG** {

**public** **static** void main(String args[])

{

*// Initialization of a TreeMap*

*// using Generics*

TreeMap<Integer, String> t

= **new** TreeMap<Integer, String>();

*// Inserting the Elements*

t.put(10, "geeks");

t.put(15, "ide");

t.put(5, "courses");

*// returns the key-value pair corresponding*

*// to higher key and prints the associated value*

System.out.println(t.higherEntry(10).getValue());

*// returns the key-value pair corresponding*

*// to lower key prints the associated value*

System.out.println(t.lowerEntry(10).getValue());

*// returns a key-value mapping associated*

*// with the greatest key <= to the given key*

System.out.println(t.floorEntry(10).getValue());

*// returns a key-value mapping associated*

*// with the least key >= to the given key*

System.out.println(t.ceilingEntry(10).getValue());

}

}

**Output:**

ide

courses

geeks

geeks

**Few other Important Methods of TreeMap:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| [**containsValue(Object value)**](https://www.geeksforgeeks.org/treemap-containsvalue-method-in-java/) | **Returns true if this map maps one or more keys to the specified value.** |
| [**firstKey()**](https://www.geeksforgeeks.org/java-util-treemap-firstentry-firstkey-java/) | **Returns the first (lowest) key currently in this sorted map.** |
| [**lastKey()**](https://www.geeksforgeeks.org/treemap-lastkey-method-in-java/) | **Returns the last (highest) key currently in this sorted map.** |

**Sample Problem: Design a DS for item prices**

Given some items and their prices, the task is to design a data structure in order to perform the below-given operations efficiently.

* **add(price, item)**: Add a new item along with its price.
* **find(price)**: Fetch the item based on its price.
* **printSorted()**: Print all items in order of their price in increasing order.
* **printGreaterSorted(price)**: Print all items in increasing order of prices with price greater than price provided in argument.
* **printSmallerSorted(price)**: Print all items in increasing order of prices with price smaller than price provided in argument.

**Note**: For simplicity, we have assumed that all prices for different items are distinct.  
  
**Naive Solution:** A naive solution will be to use arrays to store the information of items along with their prices. We can use both sorted as well as unsorted arrays. Let's see the complexity analysis of operations in both cases.

1. Sorted Array:
   * add(price, item): This will take O(N) time because we need to insert element at somewhere in between the array to maintain sorted order.
   * find(price): This will take O(logN) time as we can use binary search in a sorted array.
   * printSorted(): This will take O(N) time for complete traversal of array.
   * printGreaterSorted(price): This will take O(N) in worst case, we can also write it as O(logN + count of greater elements) as we can first find the position of element using binary search and then simply traverse all elements after it.
   * printSmallerSorted(price): This will take O(N) in worst case, we can also write it as O(logN + count of smaller elements) as we can first find the position of element using binary search and then simply traverse all elements before it.
2. Unsorted Array:
   * add(price, item): This will take O(1) time because we can directly insert a new element at the end of an unsortedd array.
   * find(price): This will take O(N) time as we have to completely traverse the array to search the given element.
   * printSorted(): This will take O(N) time for complete traversal of array.
   * printGreaterSorted(price): This will take O(N \* logN) in worst case, as we need to first sort the greater elements according to price.
   * printSmallerSorted(price): This will take O(N\*logN) in worst case, as we need to sort the smaller elements according to price.

**Efficient Solution**: The idea is to use TreeMap in Java as it maintains keys in sorted order. So, we can use price as a key and their corresponding item names as value.

* We can add or extract an element from a TreeMap in O(logN) time, so the first two operations add() and find() takes O(logN) time in the worst case.
* The operation printSorted() will take O(N) time as TreeMap already maintains the keys in sorted order.
* TreeMap provides a built-in method tailMap() which returns a view of TreeMap containing all items greater than a given item. This method can be used to implement the printGreaterSorted() function efficiently in O(N) time.
* TreeMap provides a built-in method headMap() which returns a view of TreeMap containing all items smaller than a given item. This method can be used to implement the printSmallerSorted() function efficiently in O(N) time.

**Implementation**:

Java

**class** **MyDS**{

TreeMap<Integer, String> m;

MyDS()

{

m = **new** TreeMap<Integer, String>();

}

void add(int price, String item)

{

m.put(price, item);

}

String find(int price)

{

String res = m.get(price);

**if**(res == **null**)

**return** "";

**else**

**return** res;

}

void printSorted()

{

**for**(Map.Entry<Integer, String> e : m.entrySet())

{

System.out.println(e.getValue() + " " + e.getKey());

}

}

void printGreaterSorted(int price)

{

SortedMap<Integer, String> g = m.tailMap(price);

**for**(Map.Entry<Integer, String> e : g.entrySet())

{

System.out.println(e.getValue() + " " + e.getKey());

}

}

void printSmallerSorted(int price)

{

SortedMap<Integer, String> s = m.headMap(price);

**for**(Map.Entry<Integer, String> e : s.entrySet())

{

System.out.println(e.getValue() + " " + e.getKey());

}

}

}

**Sample Problem: Design a DS for prices - duplicates allowed**

**Problem**: Given some items and their prices, the task is to design a data structure in order to perform the below-given operations efficiently.

* **add(price, item)**: Add a new item along with its price.
* **find(price)**: Fetch the list of items with price as given price.
* **printSorted()**: Print all items in order of their price in increasing order.
* **printGreaterSorted(price)**: Print all items in increasing order of prices with price greater than price provided in argument.
* **printSmallerSorted(price)**: Print all items in increasing order of prices with price smaller than price provided in argument.

**Note**: Prices may be duplicate.  
  
**Naive Solution:** A naive solution will be to use arrays to store the information of items along with their prices. We can use both sorted as well as unsorted arrays. Let's see the complexity analysis of operations in both cases.

1. Sorted Array:
   * add(price, item): This will take O(N) time because we need to insert element at somewhere in between the array to maintain sorted order.
   * find(price): This will take O(logN + K) time as we can use binary search to find the price in a sorted array and print all elements matching this price.
   * printSorted(): This will take O(N) time for complete traversal of array.
   * printGreaterSorted(price): This will take O(N) in worst case, we can also write it as O(logN + count of greater elements) as we can first find the last occurrence of element with given price using binary search and then simply traverse all elements after it.
   * printSmallerSorted(price): This will take O(N) in worst case, we can also write it as O(logN + count of smaller elements) as we can first find the first occurrence of element with the given price using binary search and then simply traverse all elements before it.
2. Unsorted Array:
   * add(price, item): This will take O(1) time because we can directly insert a new element at the end of an unsortedd array.
   * find(price): This will take O(N) time as we have to completely traverse the array to search the given element.
   * printSorted(): This will take O(N) time for complete traversal of array.
   * printGreaterSorted(price): This will take O(N \* logN) in worst case, as we need to sort the greater elements according to price.
   * printSmallerSorted(price): This will take O(N\*logN) in worst case, as we need to sort the smaller elements according to price.

**Efficient Solution**: We discussed a TreeMap based approach in the previous articles for the same problem which had all distinct prices. Let us try to modify that solution to work for duplicate prices as well.  
  
The idea is to use TreeMap in Java as it maintains keys in sorted order. So, we can use price as a key and a list of items matching this price as List of string for its value.

**Syntax**:

TreeMap > m;

The key is an Integer denoting the price.

The value is a list of string denoting the list of item names matching the given price.

* We can add or extract an element from a TreeMap in O(logN) time, so the first two operations add() and find() takes O(logN + K) time in the worst case if there are K duplicates for a given price.
* The operation printSorted() will take O(N) time as TreeMap already maintains the keys in sorted order.
* TreeMap provides a built-in method tailMap() which returns a view of TreeMap containing all items greater than a given item. This method can be used to implement the printGreaterSorted() function efficiently in O(N) time.
* TreeMap provides a built-in method headMap() which returns a view of TreeMap containing all items smaller than a given item. This method can be used to implement the printSmallerSorted() function efficiently in O(N) time.

**Implementation**:

Java

**class** **MyDS**{

TreeMap<Integer, List<String>> m;

MyDS()

{

m = **new** TreeMap<Integer, List<String>>();

}

void add(int price, String item)

{

**if**(m.get(price) == **null**)

{

m.put(price, **new** ArrayList<>());

}

m.get(price).add(item);

}

List<String> find(int price)

{

**return** m.get(price);

}

void printSorted()

{

**for**(Map.Entry<Integer, List<String>> e : m.entrySet())

{

int p = e.getKey();

**for**(String s: e.getValue())

System.out.println(s + " " + p);

}

}

void printGreaterSorted(int price)

{

SortedMap<Integer, List<String>> g = m.tailMap(price);

**for**(Map.Entry<Integer, List<String>> e : g.entrySet())

{

int p = e.getKey();

**for**(String s: e.getValue())

System.out.println(s + " " + p);

}

}

void printSmallerSorted(int price)

{

SortedMap<Integer, List<String>> s = m.headMap(price);

**for**(Map.Entry<Integer, List<String>> e : s.entrySet())

{

int p = e.getKey();

**for**(String s: e.getValue())

System.out.println(s + " " + p);

}

}

}

**Sample Problem: Count Greater Elements**

**Sample Problem:** Given an array **arr** of integers of size **N**, the task is to find, for every element, the number of elements that are greater than it.  
  
**Examples:**

**Input:** arr[] = {4, 6, 2, 1, 8, 7}

**Output:** {3, 2, 4, 5, 0, 1}

For 4, there are 3 greater elements: 6, 8, 7

For 6, there are 2 greater elements: 8, 7

For 2, there are 4 greater elements: 4, 6, 8, 7

For 1, there are 5 greater elements: 4, 6, 2, 8, 7

For 8, there are 0 greater elemenet.

For 7, there is 1 greater element: 8

**Input:** arr[] = {2, 3, 4, 5, 6, 7, 8}

**Output:** {6, 5, 4, 3, 2, 1, 0}

A **simple solution** is to use two nested loops, the outer loop is used to pick elements one by one and the inner loop will traverse the entire array for every picked element to count the number of elements greater than it.  
The time complexity of this solution will be θ(N\*N).  
  
An **efficient solution** to solve this problem is to use **TreeMap**in Java. A TreeMap in Java allows us to store an item, fetch an item in log(N) time and it also stores values in a defined order.  
  
So, we can store the frequencies of elements of the array in decreasing order in TreeMap and we can iterate the Map and store the sum of the frequency of all previously traversed elements (i.e. elements greater than it) for every element.  
  
Below is the implementation of the above approach:

Java

*// Java implementation of the above approach*

**import** **java.util.\***;

**class** **GfG** {

**public** **static** void printGreater(int arr[])

{

int n = arr.length;

TreeMap<Integer, Integer> mp =

**new** TreeMap<Integer, Integer>(Collections.reverseOrder());

*// Store the frequency of the*

*// array elements*

**for** (int i = 0; i < n; i++) {

mp.put(arr[i], mp.getOrDefault(arr[i], 0) + 1);

}

*// Store the sum of frequency of elements*

*// greater than the current eleement*

int cumFreq = 0;

**for** (Map.Entry<Integer, Integer> e : mp.entrySet()) {

Integer temp = e.getValue();

mp.put(e.getKey(), cumFreq);

cumFreq += temp;

}

**for** (int i = 0; i < n; i++)

System.out.print(mp.get(arr[i]) + " ");

}

**public** **static** void main(String args[])

{

int arr[] = { 2, 8, 10, 5, 8 };

printGreater(arr);

}

}

**Output:**

4 1 0 3 1  
**Time Complexity:** O(N\*logN)