# Java TreeMap 源码解析

## 一、源码（signature）

源码分析如下：

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

可以看到，相比 HashMap 来说，TreeMap 多继承了一个接口 NavigableMap，也就是这个接口，决定了 TreeMap 与 HashMap 的不同：*HashMap 的 key 是无序的，TreeMap 的 key 是有序的*

## 二、接口 NavigableMap

NavigableMap 定义：

```
public interface NavigableMap<K,V> extends SortedMap<K,V>
```

发现 NavigableMap 继承了 SortedMap，再看 SortedMap 的定义：

```
public interface SortedMap<K,V> extends Map<K,V>
```

SortedMap 就像其名字那样，说明这个 Map 是有序的。这个顺序一般是指由 Comparable 接口提供的 keys 的自然序（natural ordering），或者也可以在创建 SortedMap 实例时，指定一个 Comparator 来决定。 当我们在用集合视角（collection views，与 HashMap 一样，也是由 entrySet、keySet 与 values 方法提供）来迭代（iterate）一个 SortedMap 实例时会体现出 key 的顺序。 这里引申下关于 Comparable 与 Comparator 的区别（参考这里）：

Comparable 一般表示类的自然序，比如定义一个 Student 类，学号为默认排序
Comparator 一般表示类在某种场合下的特殊分类，需要定制化排序。比如现在想按照 Student 类的 age 来排序
插入 SortedMap 中的 key 的类类都必须继承 Comparable 类（或指定一个 comparator），这样才能确定如何比较（通过 k1.compareTo(k2) 或 comparator.compare(k1, k2)）两个 key，否则，在插入时，会报 ClassCastException 的异常。 此为，SortedMap 中 key 的顺序性应该与 equals 方法保持一致。也就是说 k1.compareTo(k2)或 comparator.compare(k1, k2)为 true 时，k1.equals(k2)也应该为 true。 介绍完了 SortedMap，再来回到我们的 NavigableMap 上面来。NavigableMap 是 JDK1.6 新增的，在 SortedMap 的基础上，增加了一些“导航方法”（navigation methods）来返回与搜索目标最近的元素。例如下面这些方法：

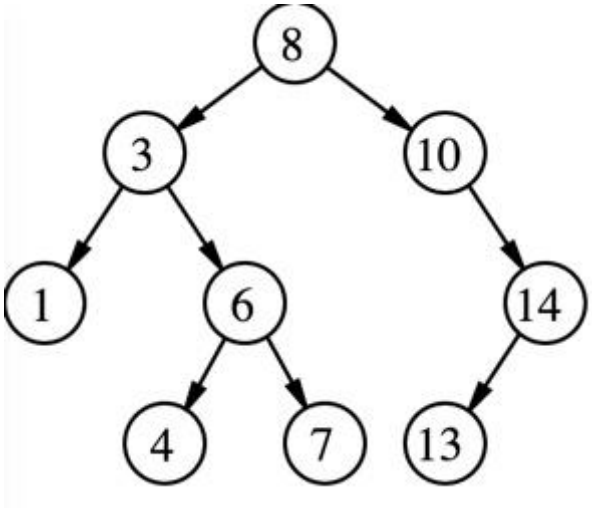
lowerEntry，返回所有比给定 Map.Entry 小的元素
floorEntry，返回所有比给定 Map.Entry 小或相等的元素

ceilingEntry，返回所有比给定 Map.Entry 大或相等的元素
higherEntry，返回所有比给定 Map.Entry 大的元素

## 三、设计理念（design concept）

TreeMap 是用红黑树作为基础实现的，红黑树是一种二叉搜索树，让我们在一起回忆下二叉搜索树的一些性质

二叉搜索树
先看看二叉搜索树（binary search tree，BST）长什么样呢？。




相信大家对这个图都不陌生，关键点是：
左子树的值小于根节点，右子树的值大于根节点。
二叉搜索树的优势在于每进行一次判断就是能将问题的规模减少一半，所以如果二叉搜索树是平衡的话，查找元素的时间复杂度为 log(n)，也就是树的高度。 我这里想到一个比较严肃的问题，如果说二叉搜索树将问题规模减少了一半，那么三叉搜索树不就将问题规模减少了三分之二，这不是更好嘛，以此类推，我们还可以有四叉搜索树，五叉搜索树……对于更一般的情况：

n 个元素，K 叉树搜索树的 K 为多少时效率是最好的？K＝2 时吗？

K 叉搜索树
如果大家按照我上面分析，很可能也陷入一个误区，就是

三叉搜索树在将问题规模减少三分之二时，所需比较操作的次数是两次（二叉搜索树再将问题规模减少一半时，只需要一次比较操作）

我们不能把这两次给忽略了，对于更一般的情况：

n 个元素，K 叉树搜索树需要的平均比较次数为 k*log(n/k)。

对于极端情况 k＝n 时，K 叉树就转化为了线性表了，复杂度也就是 O(n)了，如果用数学角度来解这个问题，相当于：

n 为固定值时，k 取何值时，k*log(n/k)的取值最小？

k*log(n/k)根据对数的运算规则可以转化为 ln(n)*k/ln(k)，ln(n)为常数，所以相当于取 k/ln(k)的极小值。这个问题对于大一刚学高数的人来说再简单不过了，我们这里直接看结果

当 k＝e 时，k/ln(k)取最小值。
自然数 e 的取值大约为 2.718 左右，可以看到二叉树基本上就是这样最优解了。
在 Nodejs 的 REPL 中进行下面的操作

```
function foo(k) {return k/Math.log(k);}
> foo(2)
2.8853900817779268
> foo(3)
2.730717679880512
> foo(4)
2.8853900817779268
> foo(5)
3.1066746727980594
```

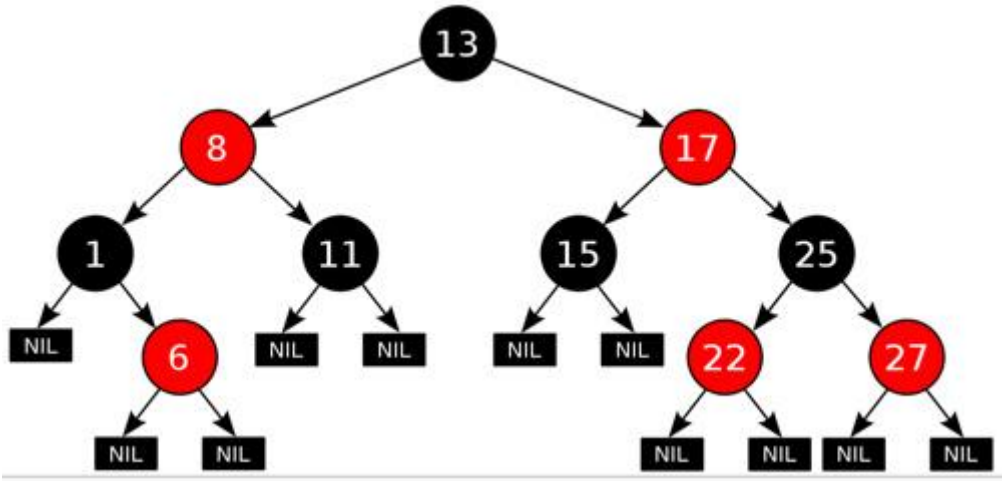
貌似 k＝3 时比 k＝2 时得到的结果还要小，那也就是说三叉搜索树应该比二叉搜索树更好些呀，但是为什么二叉树更流行呢？后来在万能的 stackoverflow 上找到了答案，主旨如下：

现在的 CPU 可以针对二重逻辑（binary logic）的代码做优化，三重逻辑会被分解为多个二重逻辑。

这样也就大概能理解为什么二叉树这么流行了，就是因为进行一次比较操作，我们最多可以将问题规模减少一半。 好了这里扯的有点远了，我们再回到红黑树上来。

## 四、红黑树性质

先看看红黑树的样子：



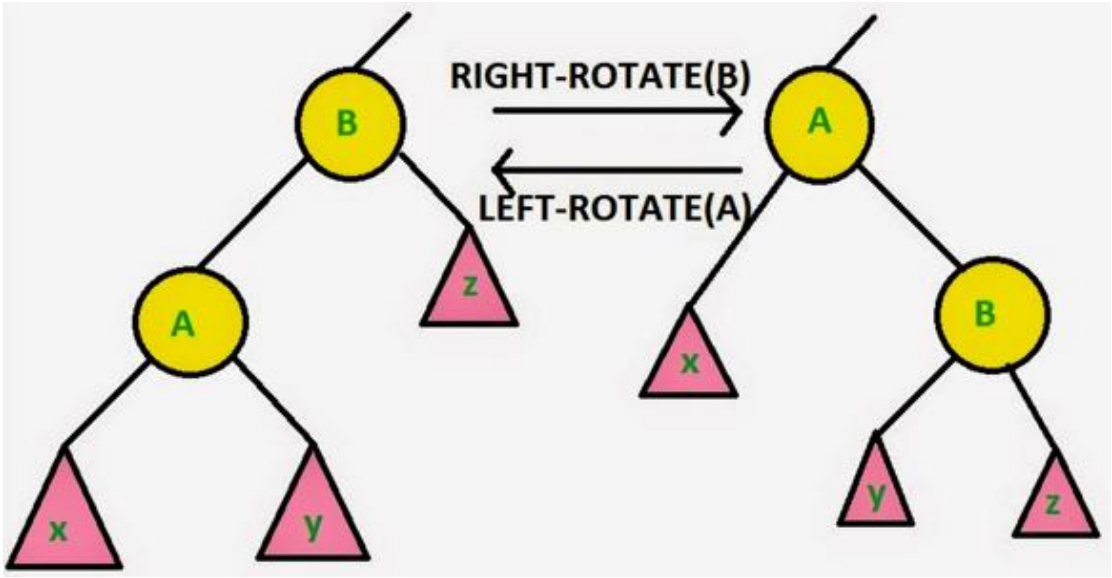

叶子节点为上图中的 NIL 节点，国内一些教材中没有这个 NIL 节点，我们在画图时有时也会省略这些 NIL 节点，但是我们需要明确，当我们说叶子节点时，指的就是这些 NIL 节点。

红黑树通过下面 5 条规则，保证了树是平衡的：

树的节点只有红与黑两种颜色
根节点为黑色的
叶子节点为黑色的
红色节点的字节点必定是黑色的
从任意一节点出发，到其后继的叶子节点的路径中，黑色节点的数目相同
满足了上面 5 个条件后，就能够保证：根节点到叶子节点的最长路径不会大于根节点到叶子最短路径的 2 倍。 其实这个很好理解，主要是用了性质 4 与 5，这里简单说下：

假设根节点到叶子节点最短的路径中，黑色节点数目为 B，那么根据性质 5，根节点到叶子节点的最长路径中，黑色节点数目也是 B，最长的情况就是每两个黑色节点中间有个红色节点（也就是红黑相间的情况），所以红色节点最多为 B－1 个。这样就能证明上面的结论了。

## 五、红黑树操作




关于红黑树的插入、删除、左旋、右旋这些操作，我觉得最好可以做到可视化，文字表达比较繁琐，我这里就不在献丑了，网上能找到的也比较多，像 v_July_v 的《教你透彻了解红黑树》。我这里推荐个 swf 教学视频（视频为英文，大家不要害怕，重点是看图??），7 分钟左右，大家可以参考。 这里还有个交互式红黑树的可视化网页，大家可以上去自己操作操作，插入几个节点，删除几个节点玩玩，看看左旋右旋是怎么玩的。

## 六、总结

TreeMap 的 key 是有序的，增删改查操作的时间复杂度为 O(log(n))，为了保证红黑树平衡，在必要时会进行旋转

HashMap 的 key 是无序的，增删改查操作的时间复杂度为 O(1)，为了做到动态扩容，在必要时会进行 resize。