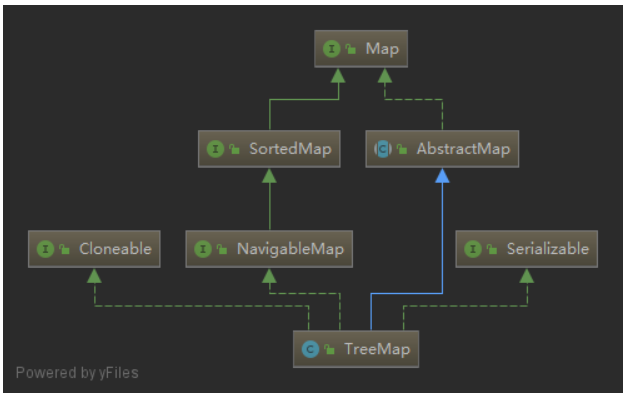
简介

TreeMap使用红黑树存储元素，可以保证元素按key值的大小进行遍历。

继承体系



TreeMap实现了Map、SortedMap、NavigableMap、Cloneable、Serializable等接口。

SortedMap规定了元素可以按key的大小来遍历，它定义了一些返回部分map的方法。

```
1.  public interface SortedMap<K,V> extends Map<K,V> {
2.      // key的比较器
3.      Comparator<? super K> comparator();
4.      // 返回fromKey（包含）到toKey（不包含）之间的元素组成的子map
5.      SortedMap<K,V> subMap(K fromKey, K toKey);
6.      // 返回小于toKey（不包含）的子map
7.      SortedMap<K,V> headMap(K toKey);
8.      // 返回大于等于fromKey（包含）的子map
9.      SortedMap<K,V> tailMap(K fromKey);
10.     // 返回最小的key
11.     K firstKey();
12.     // 返回最大的key
13.     K lastKey();
14.     // 返回key集合
15.     Set<K> keySet();
16.     // 返回value集合
17.     Collection<V> values();
18.     // 返回节点集合
19.     Set<Map.Entry<K, V>> entrySet();
20. }
21.
```

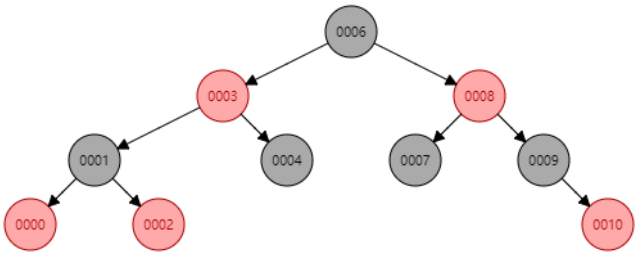
NavigableMap是对SortedMap的增强，定义了一些返回离目标key最近的元素的方法。

```
1.  public interface NavigableMap<K,V> extends SortedMap<K,V> {
2.      // 小于给定key的最大节点
3.      Map.Entry<K,V> lowerEntry(K key);
4.      // 小于给定key的最大key
```

```
        K lowerKey(K key);
        // 小于等于给定key的最大节点
        Map.Entry<K,V> floorEntry(K key);
        // 小于等于给定key的最大key
        K floorKey(K key);
        // 大于等于给定key的最小节点
        Map.Entry<K,V> ceilingEntry(K key);
        // 大于等于给定key的最小key
        K ceilingKey(K key);
        // 大于给定key的最小节点
        Map.Entry<K,V> higherEntry(K key);
        // 大于给定key的最小key
        K higherKey(K key);
        // 最小的节点
        Map.Entry<K,V> firstEntry();
        // 最大的节点
        Map.Entry<K,V> lastEntry();
        // 弹出最小的节点
        Map.Entry<K,V> pollFirstEntry();
        // 弹出最大的节点
        Map.Entry<K,V> pollLastEntry();
        // 返回倒序的map
        NavigableMap<K,V> descendingMap();
        // 返回有序的key集合
        NavigableSet<K> navigableKeySet();
        // 返回倒序的key集合
        NavigableSet<K> descendingKeySet();
        // 返回从fromKey到toKey的子map，是否包含起止元素可以自己决定
```

```
        NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                 K toKey,   boolean toInclusive);
        // 返回小于toKey的子map，是否包含toKey自己决定
        NavigableMap<K,V> headMap(K toKey, boolean inclusive);
        // 返回大于fromKey的子map，是否包含fromKey自己决定
        NavigableMap<K,V> tailMap(K fromKey, boolean inclusive);
        // 等价于subMap(fromKey, true, toKey, false)
        SortedMap<K,V> subMap(K fromKey, K toKey);
        // 等价于headMap(toKey, false)
        SortedMap<K,V> headMap(K toKey);
        // 等价于tailMap(fromKey, true)
        SortedMap<K,V> tailMap(K fromKey);
    }

```

存储结构



TreeMap只使用到了红黑树，所以它的时间复杂度为O(log n)，我们再来回顾一下红黑树的特性。

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。（注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！）

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

源码解析

属性

```
    /**
     * 比较器，如果没传则key要实现Comparable接口
     */
    private final Comparator<? super K> comparator;

    /**
     * 根节点
     */
    private transient Entry<K,V> root;

    /**
     * 元素个数
     */
    private transient int size = 0;

    /**
     * 修改次数
     */
    private transient int modCount = 0;

```

（1）comparator

按key的大小排序有两种方式，一种是key实现Comparable接口，一种方式通过构造方法传入比较器。

（2）root

根节点，TreeMap没有桶的概念，所有的元素都存储在一颗树中。

Entry内部类

存储节点，典型的红黑树结构。

```
    static final class Entry<K,V> implements Map.Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left;
        Entry<K,V> right;
        Entry<K,V> parent;
```

```
        boolean color = BLACK;
    }

```

构造方法

```
    /**
     * 默认构造方法，key必须实现Comparable接口
     */
    public TreeMap() {
        comparator = null;
    }

    /**
     * 使用传入的comparator比较两个key的大小
     */
    public TreeMap(Comparator<? super K> comparator) {
        this.comparator = comparator;
    }

    /**
     * key必须实现Comparable接口，把传入map中的所有元素保存到新的TreeMap中
     */
    public TreeMap(Map<? extends K, ? extends V> m) {
        comparator = null;
        putAll(m);
    }
```

```

    /**
     * 使用传入map的比较器，并把传入map中的所有元素保存到新的TreeMap中
     */
    public TreeMap(SortedMap<K, ? extends V> m) {
        comparator = m.comparator();
        try {
            buildFromSorted(m.size(), m.entrySet().iterator(), null, null);
        } catch (java.io.IOException cannotHappen) {
        } catch (ClassNotFoundException cannotHappen) {
        }
    }

```

构造方法主要分成两类，一类是使用comparator比较器，一类是key必须实现Comparable接口。

其实，笔者认为这两种比较方式可以合并成一种，当没有传comparator的时候，可以用以下方式来给comparator赋值，这样后续所有的比较操作都可以使用一样的逻辑处理了，而不用每次都检查comparator为空的时候又用Comparable来实现一遍逻辑。

```
    // 如果comparator为空，则key必须实现Comparable接口，所以这里肯定可以强转
    // 这样在构造方法中统一替换掉，后续的逻辑就都一致了
    comparator = (k1, k2) -> ((Comparable<? super K>)k1).compareTo(k2);

```

get(Object key)方法

获取元素，典型的二叉查找树的查找方法。

```
1.    public V get(Object key) {
2.        // 根据key查找元素
3.        Entry<K,V> p = getEntry(key);
4.        // 找到了返回value值，没找到返回null
5.        return (p==null ? null : p.value);
6.    }
7.
8.    final Entry<K,V> getEntry(Object key) {
9.        // 如果comparator不为空，使用comparator的版本获取元素
10.       if (comparator != null)
11.           return getEntryUsingComparator(key);
12.       // 如果key为空返回空指针异常
13.       if (key == null)
14.           throw new NullPointerException();
15.       // 将key强转为Comparable
16.       @SuppressWarnings("unchecked")
17.       Comparable<? super K> k = (Comparable<? super K>) key;
18.       // 从根元素开始遍历
19.       Entry<K,V> p = root;
20.       while (p != null) {
21.           int cmp = k.compareTo(p.key);
22.           if (cmp < 0)
23.               // 如果小于0从左子树查找
24.               p = p.left;
25.           else if (cmp > 0)
26.               // 如果大于0从右子树查找
27.               p = p.right;
28.           else
29.               // 如果相等说明找到了直接返回
30.               return p;
31.       }
32.       // 没找到返回null
33.       return null;
34.   }
35.
36.   final Entry<K,V> getEntryUsingComparator(Object key) {
37.       @SuppressWarnings("unchecked")
38.       K k = (K) key;
39.       Comparator<? super K> cpr = comparator;
40.       if (cpr != null) {
41.           // 从根元素开始遍历
42.           Entry<K,V> p = root;
43.           while (p != null) {
44.               int cmp = cpr.compare(k, p.key);
45.               if (cmp < 0)
46.                   // 如果小于0从左子树查找
47.                   p = p.left;
48.               else if (cmp > 0)
49.                   // 如果大于0从右子树查找
50.                   p = p.right;
51.               else
52.                   // 如果相等说明找到了直接返回
53.                   return p;
```

```
            }
        }
        // 没找到返回null
        return null;
    }

```

（1）从root遍历整个树；

（2）如果待查找的key比当前遍历的key小，则在其左子树中查找；

（3）如果待查找的key比当前遍历的key大，则在其右子树中查找；

（4）如果待查找的key与当前遍历的key相等，则找到了该元素，直接返回；

（5）从这里可以看出是否有comparator分化成了两个方法，但是内部逻辑一模一样，因此可见笔者 `comparator = (k1, k2) -> ((Comparable<? super K>)k1).compareTo(k2);` 这种改造的必要性。

我是一条美丽的分割线，前方高能，请做好准备。

特性再回顾

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

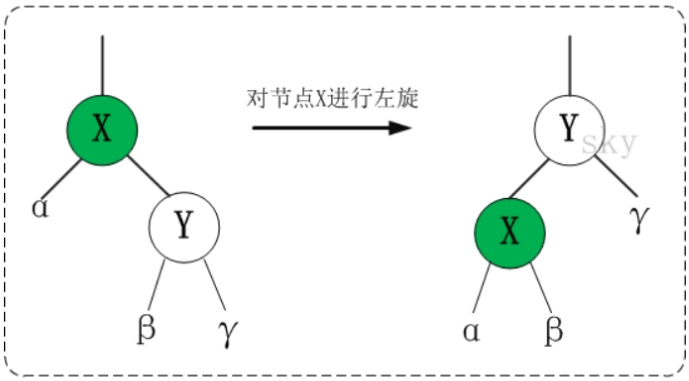
（3）每个叶子节点（NIL）是黑色。（注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！）

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

左旋

左旋，就是以某个节点为支点向左旋转。



整个左旋过程如下：

（1）将 y的左节点 设为 x的右节点，即将 β 设为 x的右节点；

（2）将 x 设为 y的左节点的父节点，即将 β的父节点 设为 x；

（3）将 x的父节点 设为 y的父节点；

（4）如果 x的父节点 为空节点，则将y设置为根节点；如果x是它父节点的左（右）节点，则将y设置为x父节点的左（右）节点；

（5）将 x 设为 y的左节点；

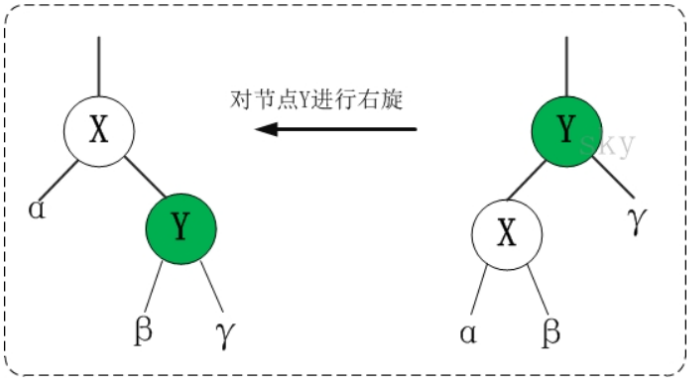
（6）将 x的父节点 设为 y；

让我们来看看TreeMap中的实现：

```
    /**
     * 以p为支点进行左旋
     * 假设p为图中的x
     */
    private void rotateLeft(Entry<K,V> p) {
        if (p != null) {
            // p的右节点，即y
            Entry<K,V> r = p.right;

            // （1）将 y的左节点 设为 x的右节点
            p.right = r.left;

            // （2）将 x 设为 y的左节点的父节点（如果y的左节点存在的话）
            if (r.left != null)
                r.left.parent = p;

            // （3）将 x的父节点 设为 y的父节点
            r.parent = p.parent;

            // （4）...
            if (p.parent == null)
                // 如果 x的父节点 为空，则将y设置为根节点
                root = r;
            else if (p.parent.left == p)
                // 如果x是它父节点的左节点，则将y设置为x父节点的左节点
                p.parent.left = r;
            else
                // 如果x是它父节点的右节点，则将y设置为x父节点的右节点
                p.parent.right = r;

            // （5）将 x 设为 y的左节点
            r.left = p;

            // （6）将 x的父节点 设为 y
            p.parent = r;
        }
    }

```

右旋

右旋，就是以某个节点为支点向右旋转。



整个右旋过程如下：

（1）将 x的右节点 设为 y的左节点，即 将 β 设为 y的左节点；

（2）将 y 设为 x的右节点的父节点，即 将 β的父节点 设为 y；

（3）将 y的父节点 设为 x的父节点；

（4）如果 y的父节点 是 空节点，则将x设为根节点；如果y是它父节点的左（右）节点，则将x设为y的父节点的左（右）节点；

（5）将 y 设为 x的右节点；

（6）将 y的父节点 设为 x；

让我们来看看TreeMap中的实现：

```
    /**
     * 以p为支点进行右旋
     * 假设p为图中的y
     */
    private void rotateRight(Entry<K,V> p) {
        if (p != null) {
            // p的左节点，即x
            Entry<K,V> l = p.left;

            // （1）将 x的右节点 设为 y的左节点
            p.left = l.right;

            // （2）将 y 设为 x的右节点的父节点（如果x有右节点的话）
            if (l.right != null) l.right.parent = p;

            // （3）将 y的父节点 设为 x的父节点
            l.parent = p.parent;

            // （4）...
            if (p.parent == null)
                // 如果 y的父节点 是 空节点，则将x设为根节点
                root = l;
            else if (p.parent.right == p)
                // 如果y是它父节点的右节点，则将x设为y的父节点的右节点
                p.parent.right = l;
            else
                // 如果y是它父节点的左节点，则将x设为y的父节点的左节点
                p.parent.left = l;

            // （5）将 y 设为 x的右节点
            l.right = p;

            // （6）将 y的父节点 设为 x
            p.parent = l;
        }
    }

```

插入元素

插入元素，如果元素在树中存在，则替换value；如果元素不存在，则插入到对应的位置，再平衡树。

```
    public V put(K key, V value) {
        Entry<K,V> t = root;
        if (t == null) {
            // 如果没有根节点，直接插入到根节点
            compare(key, key); // type (and possibly null) check
            root = new Entry<>(key, value, null);
            size = 1;
            modCount++;
            return null;
        }
        // key比较的结果
        int cmp;
        // 用来寻找待插入节点的父节点
        Entry<K,V> parent;
        // 根据是否有comparator使用不同的分支
        Comparator<? super K> cpr = comparator;
        if (cpr != null) {
            // 如果使用的是comparator方式，key值可以为null，只要在comparator.compare()中允许即可
            // 从根节点开始遍历寻找
            do {
                parent = t;
                cmp = cpr.compare(key, t.key);
                if (cmp < 0)
```

```
                    // 如果小于0从左子树寻找
                    t = t.left;
                else if (cmp > 0)
                    // 如果大于0从右子树寻找
                    t = t.right;
                else
                    // 如果等于0，说明插入的节点已经存在了，直接更换其value值并返回旧值
                    return t.setValue(value);
            } while (t != null);
        }
        else {
            // 如果使用的是Comparable方式，key不能为null
            if (key == null)
                throw new NullPointerException();
            @SuppressWarnings("unchecked")
            Comparable<? super K> k = (Comparable<? super K>) key;
            // 从根节点开始遍历寻找
            do {
                parent = t;
                cmp = k.compareTo(t.key);
                if (cmp < 0)
                    // 如果小于0从左子树寻找
                    t = t.left;
                else if (cmp > 0)
                    // 如果大于0从右子树寻找
                    t = t.right;
                else
                    // 如果等于0，说明插入的节点已经存在了，直接更换其value值并返回旧值
```

```
                    return t.setValue(value);
                } while (t != null);
            }
            // 如果没找到，那么新建一个节点，并插入到树中
            Entry<K,V> e = new Entry<>(key, value, parent);
            if (cmp < 0)
                // 如果小于0插入到左子节点
                parent.left = e;
            else
                // 如果大于0插入到右子节点
                parent.right = e;

            // 插入之后的平衡
            fixAfterInsertion(e);
            // 元素个数加1（不需要扩容）
            size++;
            // 修改次数加1
            modCount++;
            // 如果插入了新节点返回空
            return null;
        }

```

插入再平衡

插入的元素默认都是红色，因为插入红色元素只违背了第4条特性，那么我们只要根据这个特性来平衡就容易多了。

根据不同的情况有以下几种处理方式：

- 插入的元素如果是根节点，则直接涂成黑色即可，不用平衡；
- 插入的元素的父节点如果为黑色，不需要平衡；
- 插入的元素的父节点如果为红色，则违背了特性4，需要平衡，平衡时又分成下面三种情况：

（如果父节点是祖父节点的左节点）

情况	策略
1）父节点为红色，叔叔节点也为红色	（1）将父节点设为黑色；（2）将叔叔节点设为黑色；（3）将祖父节点设为红色；（4）将祖父节点设为新的当前节点，进入下一次循环判断；
2）父节点为红色，叔叔节点为黑色，且当前节点是其父节点的右节点	（1）将父节点作为新的当前节点；（2）以新当节点为支点进行左旋，进入情况3）；
3）父节点为红色，叔叔节点为黑色，且当前节点是其父节点的左节点	（1）将父节点设为黑色；（2）将祖父节点设为红色；（3）以祖父节点为支点进行右旋，进入下一次循环判断；

（如果父节点是祖父节点的右节点，则正好与上面反过来）

情况	策略
1）父节点为红色，叔叔节点也为红色	（1）将父节点设为黑色；（2）将叔叔节点设为黑色；（3）将祖父节点设为红色；（4）将祖父节点设为新的当前节点，进入下一次循环判断；
2）父节点为红色，叔叔节点为黑色，且当前节点是其父节点的左节点	（1）将父节点作为新的当前节点；（2）以新当节点为支点进行右旋；

3）父节点为红色，叔叔节点为黑色，且当前节点是其父节点的右节点	（1）将父节点设为黑色；（2）将祖父节点设为红色；（3）以祖父节点为支点进行左旋，进入下一次循环判断；

让我们来看看TreeMap中的实现：

```
    /**
     * 插入再平衡
     *（1）每个节点或者是黑色，或者是红色。
     *（2）根节点是黑色。
     *（3）每个叶子节点（NIL）是黑色。（注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！）
     *（4）如果一个节点是红色的，则它的子节点必须是黑色的。
     *（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。
     */
    private void fixAfterInsertion(Entry<K,V> x) {
        // 插入的节点为红节点，x为当前节点
        x.color = RED;

        // 只有当插入节点不是根节点且其父节点为红色时才需要平衡（违背了特性4）
        while (x != null && x != root && x.parent.color == RED) {
            if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
                // a）如果父节点是祖父节点的左节点
                // y为叔叔节点
                Entry<K,V> y = rightOf(parentOf(parentOf(x)));
                if (colorOf(y) == RED) {
                    // 情况1）如果叔叔节点为红色
                    // （1）将父节点设为黑色
                    setColor(parentOf(x), BLACK);
                    // （2）将叔叔节点设为黑色
                    setColor(y, BLACK);
                    // （3）将祖父节点设为红色
                    setColor(parentOf(parentOf(x)), RED);
                    // （4）将祖父节点设为新的当前节点
                    x = parentOf(parentOf(x));
                } else {
                    // 如果叔叔节点为黑色
                    // 情况2）如果当前节点为其父节点的右节点
                    if (x == rightOf(parentOf(x))) {
                        // （1）将父节点设为当前节点
                        x = parentOf(x);
                        // （2）以新当前节点左旋
                        rotateLeft(x);
                    }
                    // 情况3）如果当前节点为其父节点的左节点（如果是情况2）则左旋之后新当前节点正好为其父节点的左节点了）
                    // （1）将父节点设为黑色
                    setColor(parentOf(x), BLACK);
                    // （2）将祖父节点设为红色
                    setColor(parentOf(parentOf(x)), RED);
                    // （3）以祖父节点为支点进行右旋
                    rotateRight(parentOf(parentOf(x)));
                }
            } else {
                // b）如果父节点是祖父节点的右节点
                // y是叔叔节点
                Entry<K,V> y = leftOf(parentOf(parentOf(x)));
```

```
                if (colorOf(y) == RED) {
                    // 情况1）如果叔叔节点为红色
                    // （1）将父节点设为黑色
                    setColor(parentOf(x), BLACK);
                    // （2）将叔叔节点设为黑色
                    setColor(y, BLACK);
                    // （3）将祖父节点设为红色
                    setColor(parentOf(parentOf(x)), RED);
                    // （4）将祖父节点设为新的当前节点
                    x = parentOf(parentOf(x));
                } else {
                    // 如果叔叔节点为黑色
                    // 情况2）如果当前节点为其父节点的左节点
                    if (x == leftOf(parentOf(x))) {
                        // （1）将父节点设为当前节点
                        x = parentOf(x);
                        // （2）以新当前节点右旋
                        rotateRight(x);
                    }
                    // 情况3）如果当前节点为其父节点的右节点（如果是情况2）则右旋之后新当前节点正好为其父节点的右节点了）
                    // （1）将父节点设为黑色
                    setColor(parentOf(x), BLACK);
                    // （2）将祖父节点设为红色
                    setColor(parentOf(parentOf(x)), RED);
                    // （3）以祖父节点为支点进行左旋
                    rotateLeft(parentOf(parentOf(x)));
                }
            }
        }
        // 平衡完成后将根节点设为黑色
        root.color = BLACK;
    }

```

插入元素举例

我们依次向红黑树中插入 4、2、3 三个元素，来一起看看整个红黑树平衡的过程。

三个元素都插入完成后，符合父节点是祖父节点的左节点，叔叔节点为黑色，且当前节点是其父节点的右节点，即情况2）。

情况2）需要做以下两步处理：

（1）将父节点作为新的当前节点；

（2）以新当节点为支点进行左旋，进入情况3）；

情况3）需要做以下三步处理：

（1）将父节点设为黑色；

（2）将祖父节点设为红色；

（3）以祖父节点为支点进行右旋，进入下一次循环判断；

下一次循环不符合父节点为红色了，退出循环，插入再平衡完成。

删除元素

删除元素本身比较简单，就是采用二叉树的删除规则。

（1）如果删除的位置有两个叶子节点，则从其右子树中取最小的元素放到删除的位置，然后把删除位置移到替代元素的位置，进入下一步。

（2）如果删除的位置只有一个叶子节点（有可能是经过第一步转换后的删除位置），则把那个叶子节点作为替代元素，放到删除的位置，然后把这个叶子节点删除。

（3）如果删除的位置没有叶子节点，则直接把这个删除位置的元素删除即可。

（4）针对红黑树，如果删除位置是黑色节点，还需要做再平衡。

（5）如果有替代元素，则以替代元素作为当前节点进入再平衡。

（6）如果没有替代元素，则以删除的位置的元素作为当前节点进入再平衡，平衡之后再删除这个节点。

```
1.     public V remove(Object key) {
2.         // 获取节点
3.         Entry<K,V> p = getEntry(key);
4.         if (p == null)
5.             return null;
6.
7.         V oldValue = p.value;
8.         // 删除节点
9.         deleteEntry(p);
10.        // 返回删除的value
11.        return oldValue;
```

```
    }

    private void deleteEntry(Entry<K,V> p) {
        // 修改次数加1
        modCount++;
        // 元素个数减1
        size--;

        if (p.left != null && p.right != null) {
            // 如果当前节点既有左子节点，又有右子节点
            // 取其右子树中最小的节点
            Entry<K,V> s = successor(p);
            // 用右子树中最小节点的值替换当前节点的值
            p.key = s.key;
            p.value = s.value;
            // 把右子树中最小节点设为当前节点
            p = s;
            // 这种情况实际上并没有删除p节点，而是把p节点的值改了，实际删除的是p的后继节点
        }

        // 如果原来的当前节点（p）有2个子节点，则当前节点已经变成原来p的右子树中的最小节点了，也就是说其没有左子节点了
        // 到这一步，p肯定只有一个子节点了
        // 如果当前节点有子节点，则用子节点替换当前节点
        Entry<K,V> replacement = (p.left != null ? p.left : p.right);

        if (replacement != null) {
            // 把替换节点直接放到当前节点的位置上（相当于删除了p，并把替换节点移动过来了）
            replacement.parent = p.parent;
            if (p.parent == null)
                root = replacement;
            else if (p == p.parent.left)
                p.parent.left  = replacement;
            else
                p.parent.right = replacement;

            // 将p的各项属性都设为空
            p.left = p.right = p.parent = null;

            // 如果p是黑节点，则需要再平衡
            if (p.color == BLACK)
                fixAfterDeletion(replacement);
        } else if (p.parent == null) {
            // 如果当前节点就是根节点，则直接将根节点设为空即可
            root = null;
        } else {
            // 如果当前节点没有子节点且其为黑节点，则把自己当作虚拟的替换节点进行再平衡
            if (p.color == BLACK)
                fixAfterDeletion(p);

            // 平衡完成后删除当前节点（与父节点断绝关系）
            if (p.parent != null) {
                if (p == p.parent.left)
                    p.parent.left = null;
                else if (p == p.parent.right)
                    p.parent.right = null;
```

```
                p.parent = null;
            }
        }
    }

```

删除再平衡

经过上面的处理，真正删除的肯定是黑色节点才会进入到再平衡阶段。

因为删除的是黑色节点，导致整颗树不平衡了，所以这里我们假设把删除的黑色赋予当前节点，这样当前节点除了它自己的颜色还多了一个黑色，那么：

（1）如果当前节点是根节点，则直接涂黑即可，不需要再平衡；

（2）如果当前节点是红+黑节点，则直接涂黑即可，不需要平衡；

（3）如果当前节点是黑+黑节点，则我们只要通过旋转把这个多出来的黑色不断的向上传递到一个红色节点即可，这又可能会出现以下四种情况：

（假设当前节点为父节点的左子节点）

情况	策略
1）x是黑+黑节点，x的兄弟是红节点	（1）将兄弟节点设为黑色；（2）将父节点设为红色；（3）以父节点为支点进行左旋；（4）重新设置x的兄弟节点，进入下一步；
2）x是黑+黑节点，x的兄弟是黑节点，且兄弟节点的两个子节点都是黑色	（1）将兄弟节点设置为红色；（2）将x的父节点作为新的当前节点，进入下一次循环；
3）x是黑+黑节点，x的兄弟是黑节点，且兄弟节点的右子节点为黑色，左子节点为红色	（1）将兄弟节点的左子节点设为黑色；（2）将兄弟节点设为红色；（3）以兄弟节点为支点进行右旋；（4）重新设置x的兄弟节点，进入下一步；
3）x是黑+黑节点，x的兄弟是黑节点，且兄弟节点的右子节点为红色，左子节点任意颜色	（1）将兄弟节点的颜色设为父节点的颜色；（2）将父节点设为黑色；（3）将兄弟节点的右子节点设为黑色；（4）以父节点为支点进行左旋；（5）将root作为新的当前节点（退出循环）；

（假设当前节点为父节点的右子节点，正好反过来）

情况	策略
1）x是黑+黑节点，x的兄弟是红节点	（1）将兄弟节点设为黑色；（2）将父节点设为红色；（3）以父节点为支点进行右旋；（4）重新设置x的兄弟节点，进入下一步；
2）x是黑+黑节点，x的兄弟是黑节点，且兄弟节点的两个子节点都是黑色	（1）将兄弟节点设置为红色；（2）将x的父节点作为新的当前节点，进入下一次循环；
3）x是黑+黑节点，x的兄弟是黑节点，且兄弟节点的左子节点为黑色，右子节点为红色	（1）将兄弟节点的右子节点设为黑色；（2）将兄弟节点设为红色；（3）以兄弟节点为支点进行左旋；（4）重新设置x的兄弟节点，进入下一步；
3）x是黑+黑节点，x的兄弟是黑节点，且兄弟节点的左子节点为红色，右子节点任意颜色	（1）将兄弟节点的颜色设为父节点的颜色；（2）将父节点设为黑色；（3）将兄弟节点的左子节点设为黑色；（4）以父节点为支点进行右旋；（5）将root作为新的当前节点（退出循环）；

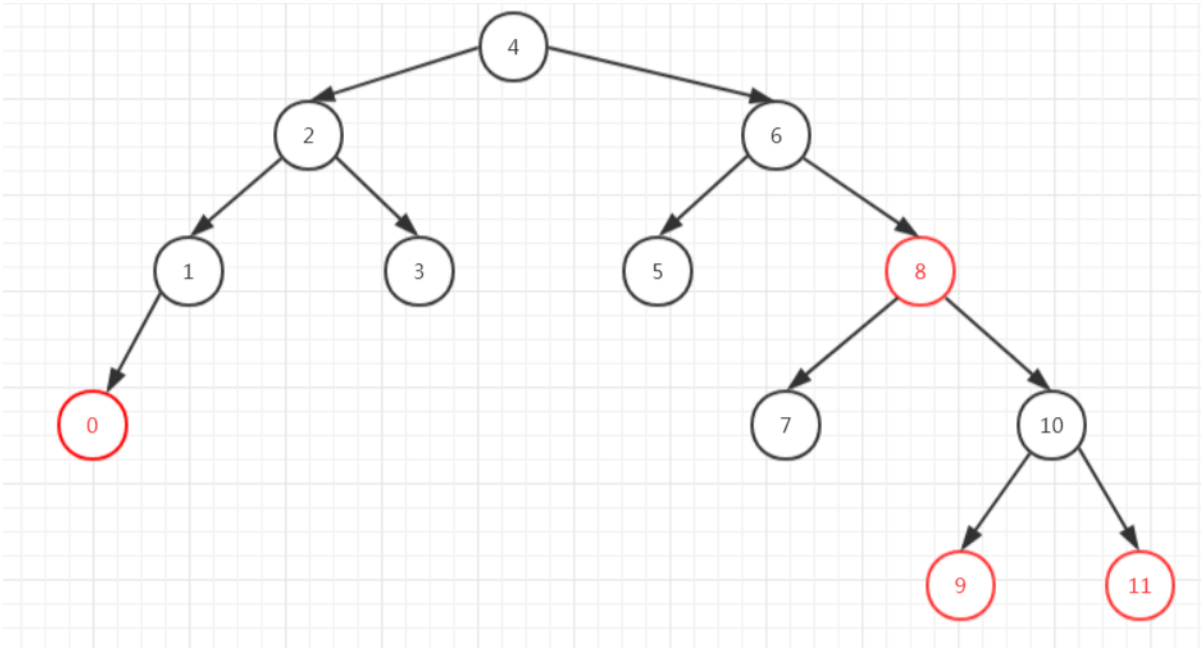
让我们来看看TreeMap中的实现：

```java
    /**
     * 删除再平衡
     *（1）每个节点或者是黑色，或者是红色。
     *（2）根节点是黑色。
     *（3）每个叶子节点（NIL）是黑色。（注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！）
     *（4）如果一个节点是红色的，则它的子节点必须是黑色的。
     *（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。
     */
    private void fixAfterDeletion(Entry<K,V> x) {
        // 只有当前节点不是根节点且当前节点是黑色时才进入循环
        while (x != root && colorOf(x) == BLACK) {
            if (x == leftOf(parentOf(x))) {
                // 如果当前节点是其父节点的左子节点
                // sib是当前节点的兄弟节点
                Entry<K,V> sib = rightOf(parentOf(x));

                // 情况1）如果兄弟节点是红色
                if (colorOf(sib) == RED) {
                    // （1）将兄弟节点设为黑色
                    setColor(sib, BLACK);
                    // （2）将父节点设为红色
                    setColor(parentOf(x), RED);
                    // （3）以父节点为支点进行左旋
                    rotateLeft(parentOf(x));
                    // （4）重新设置x的兄弟节点，进入下一步
                    sib = rightOf(parentOf(x));
                }

                if (colorOf(leftOf(sib))  == BLACK &&
                        colorOf(rightOf(sib)) == BLACK) {
                    // 情况2）如果兄弟节点的两个子节点都是黑色
                    // （1）将兄弟节点设置为红色
                    setColor(sib, RED);
                    // （2）将x的父节点作为新的当前节点，进入下一次循环
                    x = parentOf(x);
                } else {
                    if (colorOf(rightOf(sib)) == BLACK) {
                        // 情况3）如果兄弟节点的右子节点为黑色
                        // （1）将兄弟节点的左子节点设为黑色
                        setColor(leftOf(sib), BLACK);
                        // （2）将兄弟节点设为红色
                        setColor(sib, RED);
                        // （3）以兄弟节点为支点进行右旋
                        rotateRight(sib);
                        // （4）重新设置x的兄弟节点
                        sib = rightOf(parentOf(x));
                    }
                    // 情况4）
                    // （1）将兄弟节点的颜色设为父节点的颜色
                    setColor(sib, colorOf(parentOf(x)));
                    // （2）将父节点设为黑色
                    setColor(parentOf(x), BLACK);
                    // （3）将兄弟节点的右子节点设为黑色
                    setColor(rightOf(sib), BLACK);
                    // （4）以父节点为支点进行左旋
                    rotateLeft(parentOf(x));
```

```
                    // （5）将root作为新的当前节点（退出循环）
                    x = root;
                }
            } else { // symmetric
                // 如果当前节点是其父节点的右子节点
                // sib是当前节点的兄弟节点
                Entry<K,V> sib = leftOf(parentOf(x));

                // 情况1）如果兄弟节点是红色
                if (colorOf(sib) == RED) {
                    // （1）将兄弟节点设为黑色
                    setColor(sib, BLACK);
                    // （2）将父节点设为红色
                    setColor(parentOf(x), RED);
                    // （3）以父节点为支点进行右旋
                    rotateRight(parentOf(x));
                    // （4）重新设置x的兄弟节点
                    sib = leftOf(parentOf(x));
                }

                if (colorOf(rightOf(sib)) == BLACK &&
                        colorOf(leftOf(sib)) == BLACK) {
                    // 情况2）如果兄弟节点的两个子节点都是黑色
                    // （1）将兄弟节点设置为红色
                    setColor(sib, RED);
                    // （2）将x的父节点作为新的当前节点，进入下一次循环
                    x = parentOf(x);
                } else {
                    if (colorOf(leftOf(sib)) == BLACK) {
                        // 情况3）如果兄弟节点的左子节点为黑色
                        // （1）将兄弟节点的右子节点设为黑色
                        setColor(rightOf(sib), BLACK);
                        // （2）将兄弟节点设为红色
                        setColor(sib, RED);
                        // （3）以兄弟节点为支点进行左旋
                        rotateLeft(sib);
                        // （4）重新设置x的兄弟节点
                        sib = leftOf(parentOf(x));
                    }
                    // 情况4）
                    // （1）将兄弟节点的颜色设为父节点的颜色
                    setColor(sib, colorOf(parentOf(x)));
                    // （2）将父节点设为黑色
                    setColor(parentOf(x), BLACK);
                    // （3）将兄弟节点的左子节点设为黑色
                    setColor(leftOf(sib), BLACK);
                    // （4）以父节点为支点进行右旋
                    rotateRight(parentOf(x));
                    // （5）将root作为新的当前节点（退出循环）
                    x = root;
                }
            }
        }

        // 退出条件为多出来的黑色向上传递到了根节点或者红节点
        // 则将x设为黑色即可满足红黑树规则
        setColor(x, BLACK);
    }

```

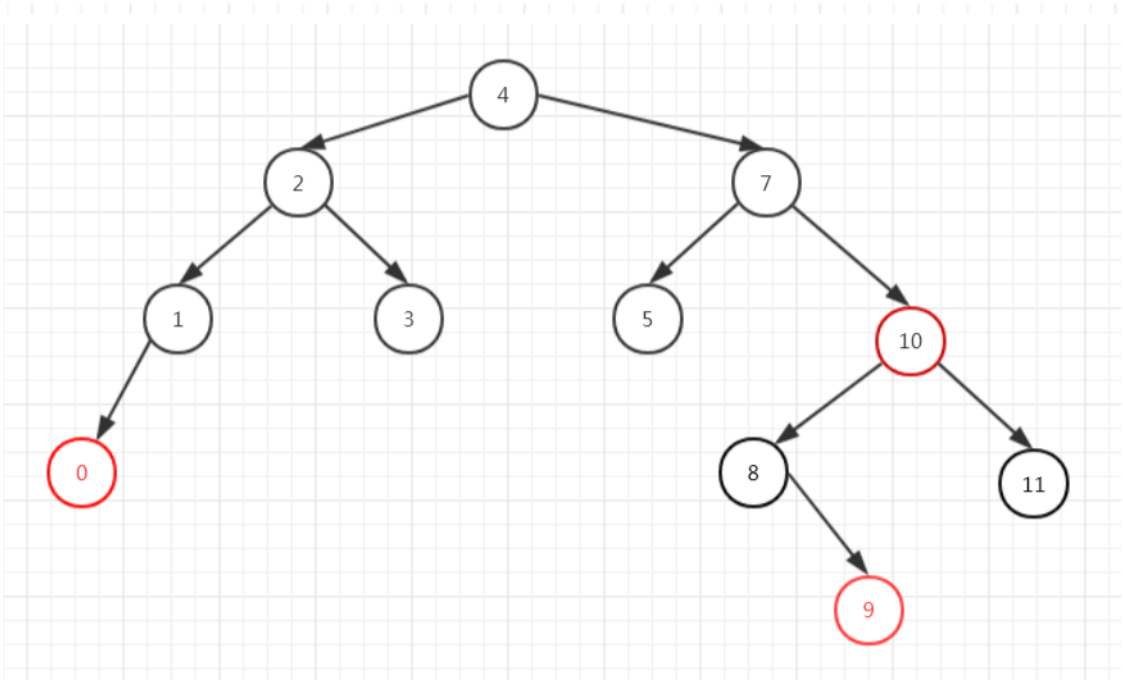
删除元素举例

假设我们有下面这样一颗红黑树。



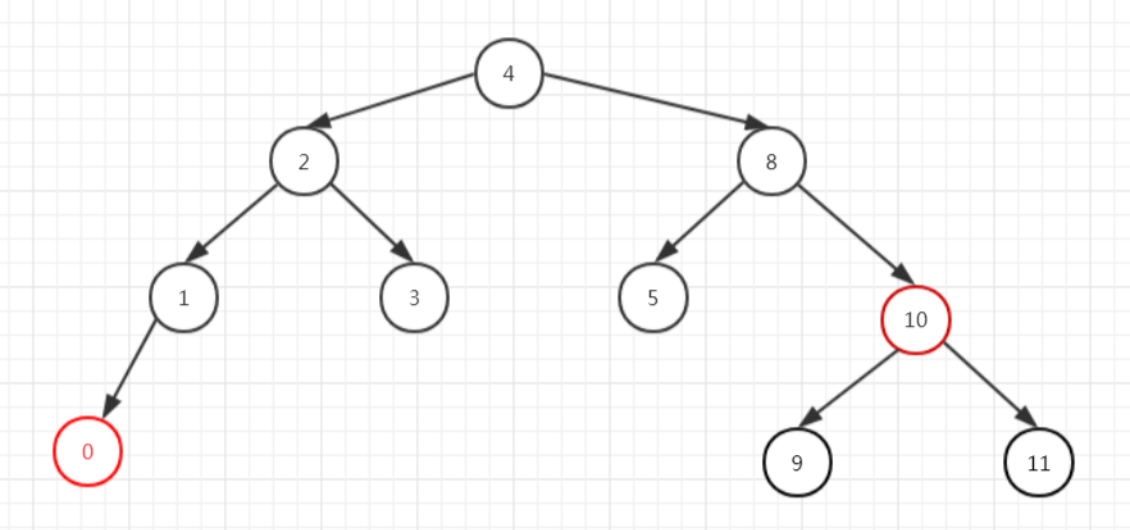
我们删除6号元素，则从右子树中找到了最小元素7，7又没有子节点了，所以把7作为当前节点进行再平衡。

我们看到7是黑节点，且其兄弟为黑节点，且其兄弟的两个子节点都是红色，满足情况4），平衡之后如下图所示。



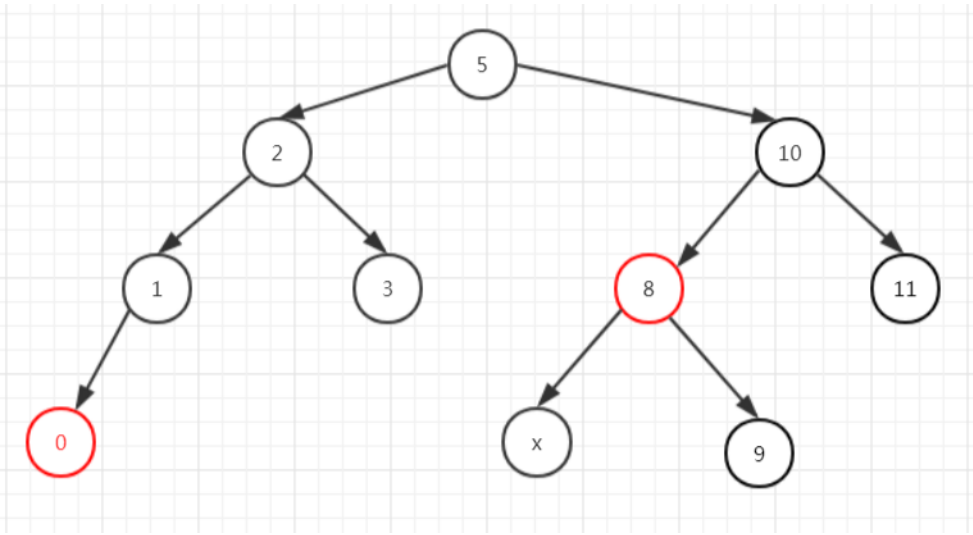
我们再删除7号元素，则从右子树中找到了最小元素8，8有子节点且为黑色，所以8的子节点9是替代节点，以9为当前节点进行再平衡。

我们发现9是红节点，则直接把它涂成黑色即满足了红黑树的特性，不需要再过多的平衡了。

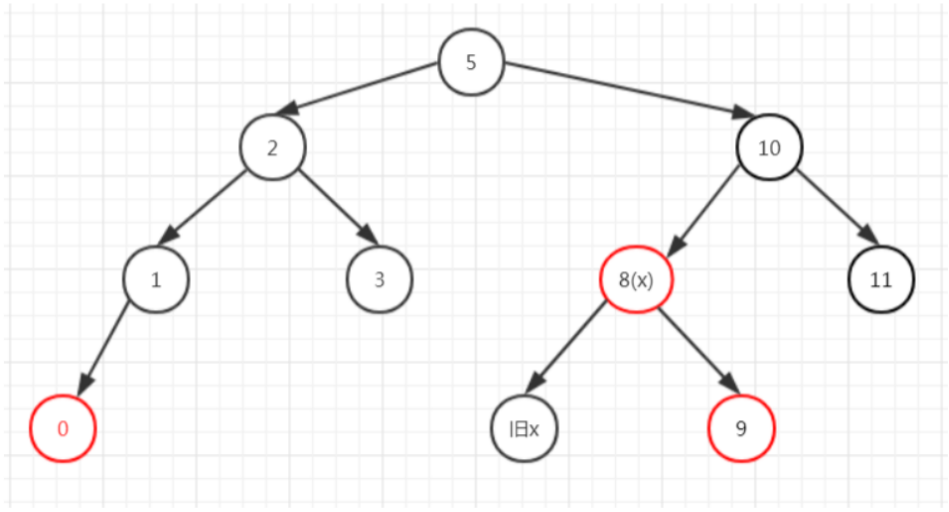


这次我们来个狠的，把根节点删除，从右子树中找到了最小的元素5，5没有子节点，所以把5作为当前节点进行再平衡。

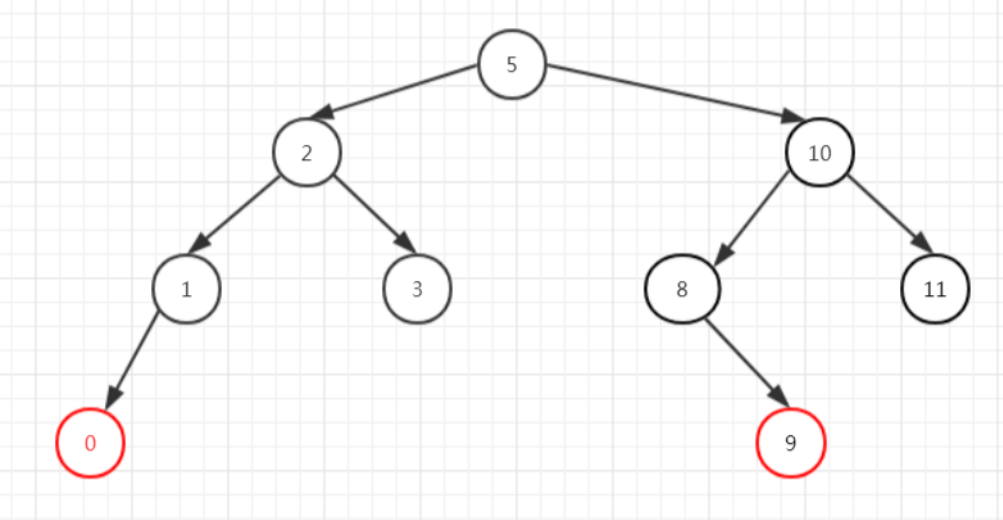
我们看到5是黑节点，且其兄弟为红色，符合情况1），平衡之后如下图所示，然后进入情况2）。



对情况2）进行再平衡后如下图所示。



然后进入下一次循环，发现不符合循环条件了，直接把x涂为黑色即可，退出这个方法之后会把旧x删除掉（见deleteEntry()方法），最后的结果就是下面这样。



二叉树的遍历

我们知道二叉查找树的遍历有前序遍历、中序遍历、后序遍历。

（1）前序遍历，先遍历我，再遍历我的左子节点，最后遍历我的右子节点；

（2）中序遍历，先遍历我的左子节点，再遍历我，最后遍历我的右子节点；

（3）后序遍历，先遍历我的左子节点，再遍历我的右子节点，最后遍历我；

这里的前中后都是以“我”的顺序为准的，我在前就是前序遍历，我在中就是中序遍历，我在后就是后序遍历。

下面让我们看看经典的中序遍历是怎么实现的：

```
1.  public class TreeMapTest {
2.
3.      public static void main(String[] args) {
4.          // 构建一颗10个元素的树
5.          TreeNode<Integer> node = new TreeNode<>(1, null).insert(2)
6.                  .insert(6).insert(3).insert(5).insert(9)
7.                  .insert(7).insert(8).insert(4).insert(10);
8.
9.          // 中序遍历，打印结果为1到10的顺序
10.         node.root().inOrderTraverse();
11.     }
12. }
13.
```

```java
/**
 * 树节点，假设不存在重复元素
 * @param <T>
 */
class TreeNode<T extends Comparable<T>> {
    T value;
    TreeNode<T> parent;
    TreeNode<T> left, right;

    public TreeNode(T value, TreeNode<T> parent) {
        this.value = value;
        this.parent = parent;
    }

    /**
     * 获取根节点
     */
    TreeNode<T> root() {
        TreeNode<T> cur = this;
        while (cur.parent != null) {
            cur = cur.parent;
        }
        return cur;
    }

    /**
     * 中序遍历
     */
    void inOrderTraverse() {
        if(this.left != null) this.left.inOrderTraverse();
        System.out.println(this.value);
        if(this.right != null) this.right.inOrderTraverse();
    }

    /**
     * 经典的二叉树插入元素的方法
     */
    TreeNode<T> insert(T value) {
        // 先找根元素
        TreeNode<T> cur = root();

        TreeNode<T> p;
        int dir;

        // 寻找元素应该插入的位置
        do {
            p = cur;
            if ((dir=value.compareTo(p.value)) < 0) {
                cur = cur.left;
            } else {
                cur = cur.right;
            }
        } while (cur != null);

        // 把元素放到找到的位置
        if (dir < 0) {
            p.left = new TreeNode<>(value, p);
```

```
                return p.left;
            } else {
                p.right = new TreeNode<>(value, p);
                return p.right;
            }
        }
    }

```

TreeMap的遍历

从上面二叉树的遍历我们很明显地看到，它是通过递归的方式实现的，但是递归会占用额外的空间，直接到线程栈整个释放掉才会把方法中申请的变量销毁掉，所以当元素特别多的时候是一件很危险的事。

（上面的例子中，没有申请额外的空间，如果有声明变量，则可以理解为直到方法完成才会销毁变量）

那么，有没有什么方法不用递归呢?

让我们来看看java中的实现：

```
    @Override
    public void forEach(BiConsumer<? super K, ? super V> action) {
        Objects.requireNonNull(action);
        // 遍历前的修改次数
        int expectedModCount = modCount;
        // 执行遍历，先获取第一个元素的位置，再循环遍历后继节点
        for (Entry<K, V> e = getFirstEntry(); e != null; e = successor(e)) {
            // 执行动作
            action.accept(e.key, e.value);

            // 如果发现修改次数变了，则抛出异常
            if (expectedModCount != modCount) {
                throw new ConcurrentModificationException();
            }
        }
    }

```

是不是很简单？！

（1）寻找第一个节点;

从根节点开始找最左边的节点，即最小的元素。

```
        final Entry<K,V> getFirstEntry() {
            Entry<K,V> p = root;
            // 从根节点开始找最左边的节点，即最小的元素
            if (p != null)
                while (p.left != null)
                    p = p.left;
            return p;
        }

```

（2）循环遍历后继节点；

寻找后继节点这个方法我们在删除元素的时候也用到过，当时的场景是有右子树，则从其右子树中寻找最小的节点。

```
    static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
        if (t == null)
            // 如果当前节点为空，返回空
            return null;
        else if (t.right != null) {
            // 如果当前节点有右子树，取右子树中最小的节点
            Entry<K,V> p = t.right;
            while (p.left != null)
                p = p.left;
            return p;
        } else {
            // 如果当前节点没有右子树
            // 如果当前节点是父节点的左子节点，直接返回父节点
            // 如果当前节点是父节点的右子节点，一直往上找，直到找到一个祖先节点是其父节点的左子节点为止，返回这个祖先节点的父节点
            Entry<K,V> p = t.parent;
            Entry<K,V> ch = t;
            while (p != null && ch == p.right) {
                ch = p;
                p = p.parent;
            }
            return p;
        }
    }

```

让我们一起来分析下这种方式的时间复杂度吧。

首先，寻找第一个元素，因为红黑树是接近平衡的二叉树，所以找最小的节点，相当于是从顶到底了，时间复杂度为O(log n)；

其次，寻找后继节点，因为红黑树插入元素的时候会自动平衡，最坏的情况就是寻找右子树中最小的节点，时间复杂度为O(log k)，k为右子树元素个数；

最后，需要遍历所有元素，时间复杂度为O(n)；

所以，总的时间复杂度为 O(log n) + O(n * log k) ≈ O(n)。

虽然遍历红黑树的时间复杂度是O(n)，但是它实际是要比跳表要慢一点的，啥？跳表是啥？安心，后面会讲到跳表的。

总结

到这里红黑树就整个讲完了，让我们再回顾下红黑树的特性：

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。（注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！）

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

除了上述这些标准的红黑树的特性，你还能讲出来哪些TreeMap的特性呢?

（1）TreeMap的存储结构只有一颗红黑树;

（2）TreeMap中的元素是有序的，按key的顺序排列;

（3）TreeMap比HashMap要慢一些，因为HashMap前面还做了一层桶，寻找元素要快很多;

（4）TreeMap没有扩容的概念;

（5）TreeMap的遍历不是采用传统的递归式遍历;

（6）TreeMap可以按范围查找元素，查找最近的元素;

（7）欢迎补充...

带详细注释的源码地址

微信用户请“阅读原文”，进入仓库查看，其它渠道直接点击此链接即可跳转。

彩蛋

上面我们说到的删除元素的时候，如果当前节点有右子树，则从右子树中寻找最小元素所在的位置，把这个位置的元素放到当前位置，再把删除的位置移到那个位置，再看有没有替代元素，balabala。

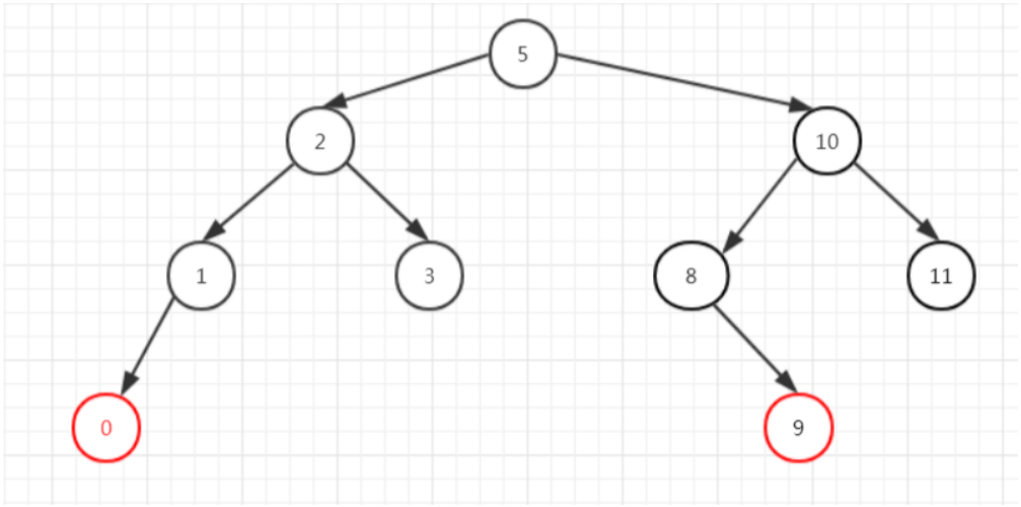
那么，除了这种方式，还有没有其它方式呢?

答案当然是肯定的。

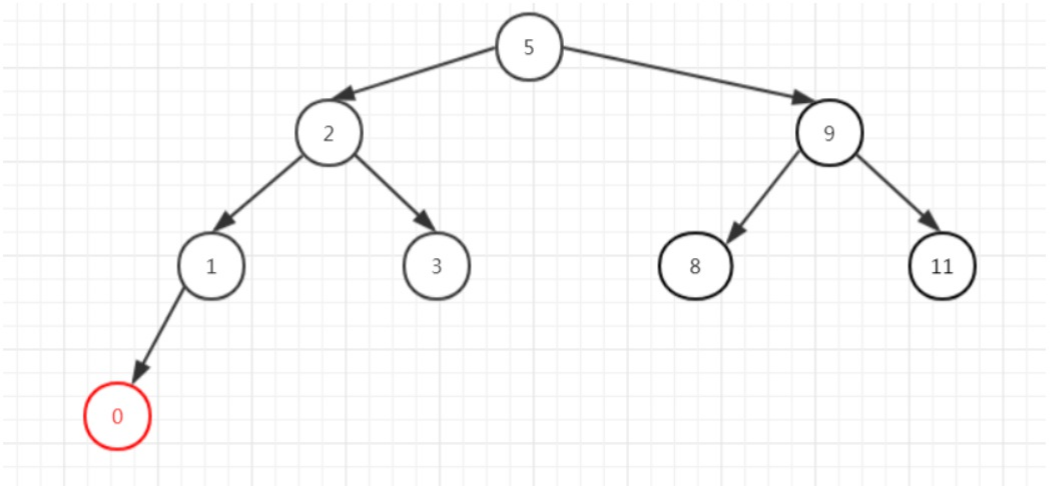
上面我们说的红黑树的插入元素、删除元素的过程都是标准的红黑树是那么干的，其实也不一定要完全那么做。

比如说，删除元素，如果当前节点有左子树，那么，我们可以找左子树中最大元素的位置，然后把这个位置的元素放到当前节点，再把删除的位置移到那个位置，再看有没有替代元素，balabala。

举例说明，比如下面这颗红黑树:



我们删除10这个元素，从左子树中找最大的，找到了9这个元素，那么把9放到10的位置，然后把删除的位置移到原来9的位置，发现不需要作平衡（红+黑节点），直接把这个位置删除就可以了。



同样是满足红黑树的特性的。

所以，死读书不如无书，学习的过程也是一个不断重塑知识的过程。