# 原理介绍



了解红黑树之前需要了解二叉查找树（二叉排序数）

1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉排序树。

为什么要用红黑树？

红黑树放弃了追求完全平衡，追求大致平衡，任何不平衡都会在3次旋转之内解决,平衡二叉树不行

**红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树高度差不超过二者中较低那个的两倍**，具体来说红黑树是满足如下条件的二叉查找树

①每个节点要么是红色，要么是黑色

②根节点必须是黑色

③每个叶子节点都是黑色，指的是空节点

④红色节点不能连续（即，红色节点的孩子和父亲都不能是红色）

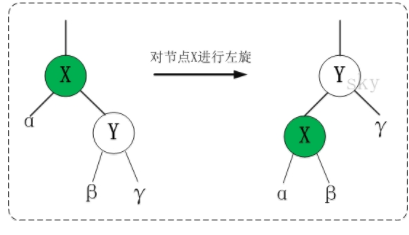
⑤对于每个节点，从该点到null（叶子节点之后）的任何路径，都含有相同个数的黑色节点

⑤性质确保没有一条路径会比其他路劲长出两倍，因为如果红色不能连续出现两次，所以红色之后一定是黑色，如果长过两倍，那么一定会多一个黑色

在树的结构发生变化时（插入删除操作），往往会破坏上述条件3或4，需要通过调整使得查找树重新满足红黑树的条件

# 红黑树的基本操作

## 左旋（相对于右孩子的右孩子后面插入节点会不平衡）



步骤：

①得到x的右子树，也就是y

②将x的右孩子指向y的左孩子

③y的左孩子的父亲指向x

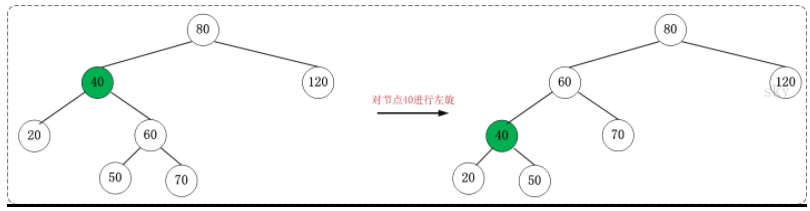
④y的父亲指向x的父亲（因为对父亲赋值没有方向，可以先设置）

---如果x的父亲为空，表明根节点，根节点直接指向y

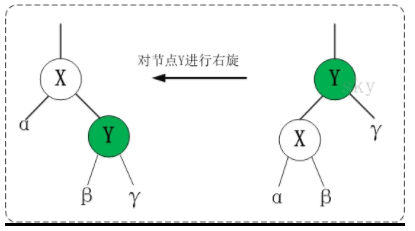
---如果x是父亲的左孩子，则x的父亲的左孩子指向y

---如果x是右孩子，则x的父亲的右孩子指向y

⑤y的左孩子指向x，x的父亲指向y



## 右旋（相对于左孩子的左孩子后面插入节点会不平衡）



①获取y的左孩子，赋值为x

②将y的左孩子指向x的右孩子

③x的右孩子的父亲指向y

④将x的父亲指向y的父亲

⑤如果y的父亲为空，就将根节点指向x

-----如果y是左孩子，就将y的父亲的左孩子指向x

-----如果y是右孩子，就将y的父亲的右孩子指向x

⑥x的右孩子指向y，y的父亲指向x

## 红黑树的插入操作

①将红黑树当成是二叉查找树，将节点插入

②将插入的节点当成--红色

将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。

③通过一系列的旋转着色，使之重新成为红黑树

**先理解插入的过程，后理解调整的过程**

**插入过程**

①新建节点“y”，将y设为空节点

②将红黑树T的根节点指向x

③找到要插入的节点z的位置y

----循环x！=null

-------x复制给y，此时要的是满足条件的那个节点作为父节点，如果取x，那么循环退出的时候，x=null，因此要找到x之前访问的节点，即用y表示

-------如果z的值小于x的值

----------x=x的左孩子

-------否则 x = x 的右孩子

④设置z的父亲为y

⑤判断y是否是空节点

----如果是，将z设为根节点

----如果不是且z的值小于y的值，将z作为y的左孩子

----否则z作为y的右孩子

⑥此时z已经与y进行连接，只需设置左右孩子为空即可完成插入

⑦颜色设置为红色

⑧关键步骤：调整为红黑树

**调整过程：**

当红色节点被插入后，可分为三种情况：

①该红色节点插入后为根节点，将此节点涂为黑色

②该红色节点插入后的父节点是黑色，不需要做调整

③该红色节点插入后的父节点是红色，

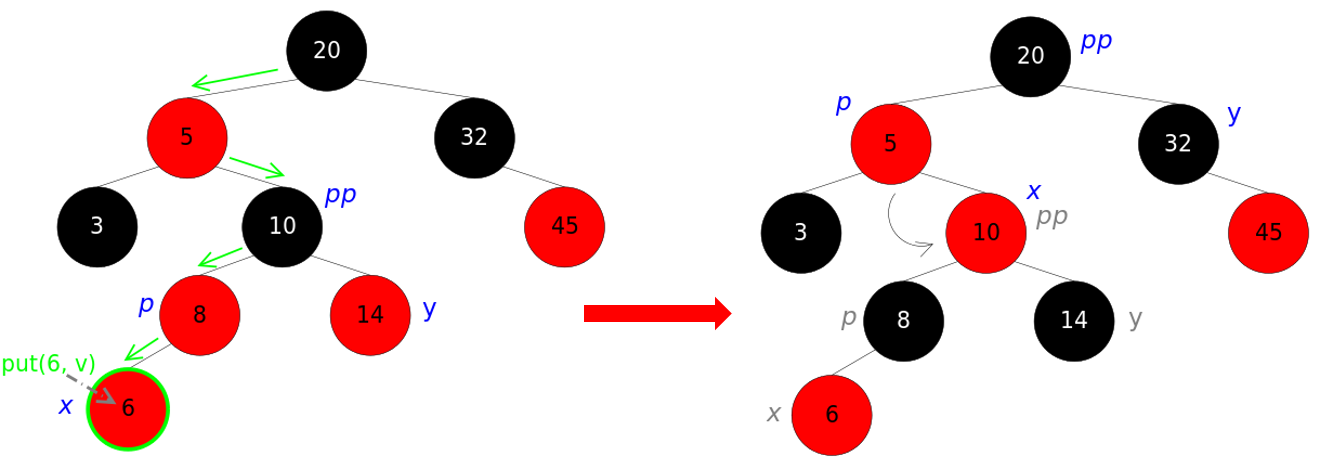
这里分析为什么父节点是红色，一定有祖父节点，因为父节点是红色，显然不会是根节点，因为根节点是黑色的，所以如果祖父不存在，那么不满足性质②



**上述3中情况处理问题的核心思想就是：将红色的节点移到根节点，然后将根节点设为黑色**

**Case1:叔叔是红色**

现象:当前节点(即，被插入节点)的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色



处理：

--将“父节点”设置为黑色

--将“叔叔节点”设置为黑色

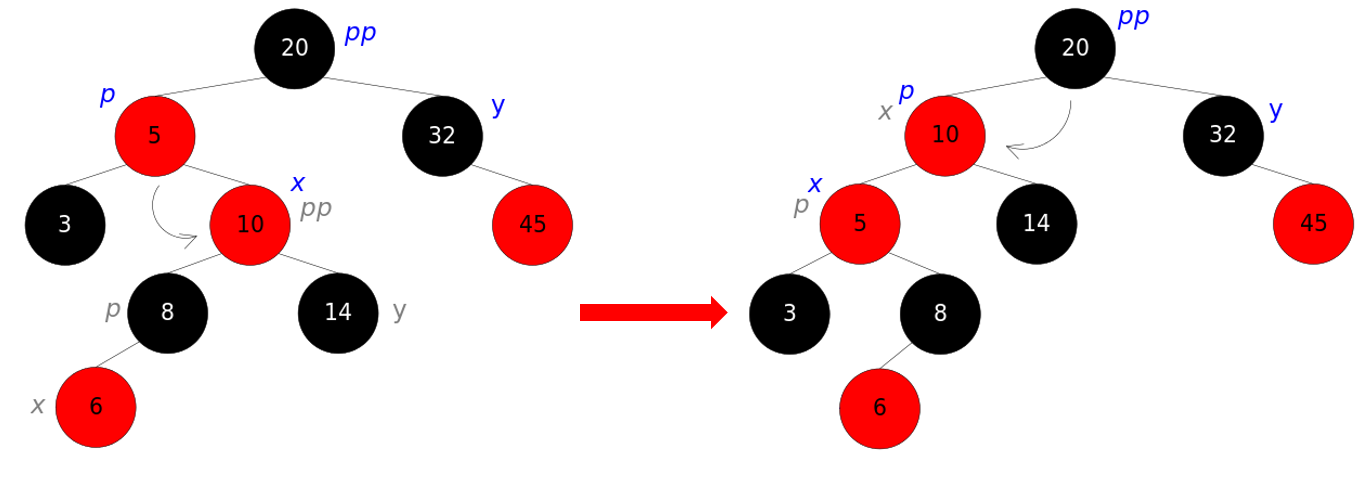
--将“祖父节点”设置为红色

--将祖父节点设置为当前节点（红色节点），即之后继续对当亲节点进行操作

原因：此时的情况时，插入节点红色，父亲和叔叔是红色，这显然不符合红色不连续的规则，然后将父亲变黑，但是父亲变黑后，包含父节点的分支的黑色节点的总数会加1，因此需要将祖父变红，但是祖父变红后又会引起，祖父和叔叔连续红色，并且包含叔叔的节点的黑色数会减1，因此需要同时变化。

**Case2叔叔是黑色，且当前节点是右孩子**

当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子



策略:

①将父节点作为“新的当前节点”

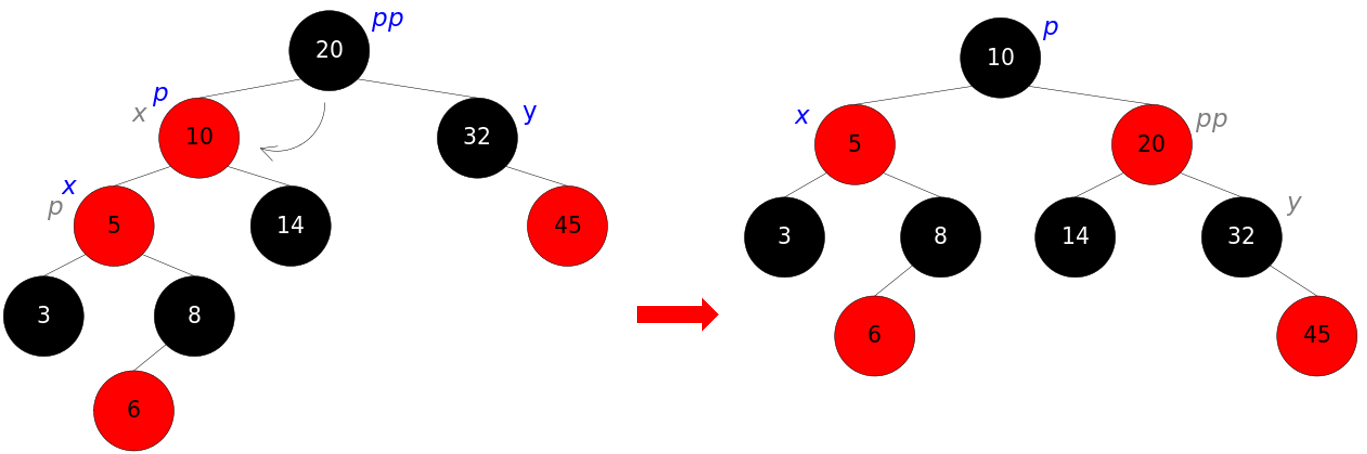
②以“新的当前节点”为支点进行左旋

为什么这么处理？

先讨论左旋，10是5的右孩子，处理红黑树的思想是，将红色节点移到根节点，然后根节点变为黑色，要使得10上移，10又是右孩子，因此左旋上移

此时5变为10的左孩子，如果此时10为根节点，那么直接将其设置黑色，结束；如果10不是根节点，需要执行步骤①，将5设为新的当前节点，为什么不继续以10为当前节点，我们处理问题要自下而上，即先解决孩子问题，在解决父亲问题。也可这么理解，由于需要先执行步骤①，然后在左旋，但是旋转传入的需要旋转的节点10的父亲，那么为了得到父亲，父亲作为新的当前节点

**Case3叔叔是黑色，当前节点是左孩子**



当前节点的父亲节点是红色，叔叔节点是黑色，且当前节点是父节点的左孩子

①将父节点设置为黑色

②将祖父几点设置为红色

③以祖父节点为支点右旋

## 删除

①将红黑树当成一棵二叉查找树，将节点删除

---被删除节点没有儿子，即叶子节点，那么直接将节点删除

---被删除节点只有一个儿子，那么，直接删除该节点，并用该节点的唯一孩子代替其位置

---被删除节点有两个儿子，先找出它的后继节点，然后将后继内容复制给被删除的节点，然后将后继节点删除。后继节点：左子树最大，或者右子树最小，因为这么做，如果左子树最大k覆盖删除节点，则剩下左子树都比k小，满足平衡；如果右子树最小k覆盖删除节点，则剩下右子树都比k大，也满足平衡，至于删除后继节点，如果没有儿子按照第一种；如果只有一个儿子，那么按照第二种

②通过旋转着色调整红黑树，什么时候才会触发调整，只有在删除的是黑色的时候，因为黑色删除后破坏黑色节点的个数属性

**理解删除过程在进行调整**

**删除过程：删除节点为p**

①如果p有两个孩子

----找到其后继节点，然后对p进行代替操作

----首先将p指向的节点的值代替，然后让p指向后继节点，后面对后继节点进行操作

②要么p无孩子，要么只有左右其中一个孩子

---如果左孩子不为空，代替节点r指向p的左孩子

---如果左孩子为空，代替节点r指向p的右孩子

③如果r不为空，表明只有一个孩子

---r的parent指向p的parent

----如果p的parent为空，说明p是根节点，则root指向r；

----如果p是parent的左孩子，p的parent的左孩子执行r，否则指向右孩子

----删除p，左右父亲=null

----如果p的颜色为黑，调整

④如果r为空，且p的父亲为空，表明只有一个p节点，因此root =null

⑤如果r空，p的父亲不空，则此时p没有孩子，删除即可

----如果p的颜色为黑，调整

----如果p的父亲不为空

-------p是左孩子，则p的父亲的左孩子为空，否则p的父亲右孩子为空，

-------p的父亲为空。

**调整过程**

1. x指向一个红+黑节点，此时将x设为一个黑节点就行
2. x执行根，此时将x设为一个黑节点即可
3. 非前面两种状态

①情况说明：x是红黑节点

处理：直接将x设为黑色，结束

②x是黑黑节点，x是根

处理：什么都不做，结束

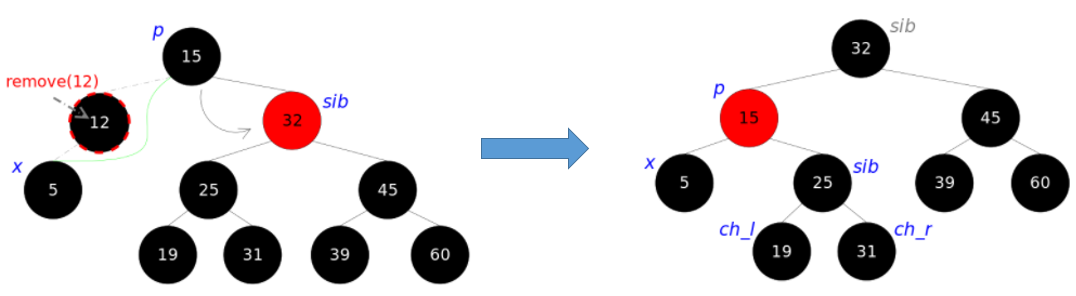
③x是黑黑节点，x不是根

处理：四种情况



**Case1：x是黑黑节点，x的兄弟是红色**

说明：x是"黑+黑"节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)



处理：

①x的兄弟设为黑色

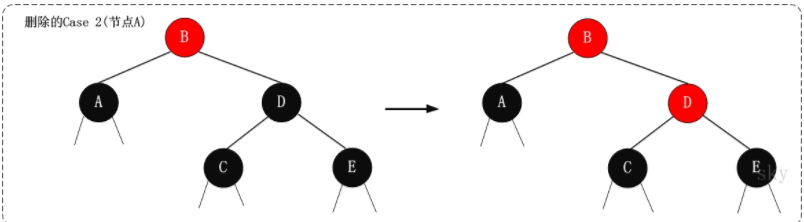
②x的父亲设为红色

③x的父亲左旋

④左旋后，重新设置x的兄弟节点

为什么这么处理：将case1转换为case2,3,4；从而进一步处理，对x的父节点进行左旋，左旋后，兄弟变为父亲，因此左旋前，需要将x的兄弟节点设为黑色，父节点设为红色，左旋后设置新的兄弟节点

**Case2：x是黑黑节点，x兄弟是黑色，x的兄弟节点的两个孩子都是黑色**

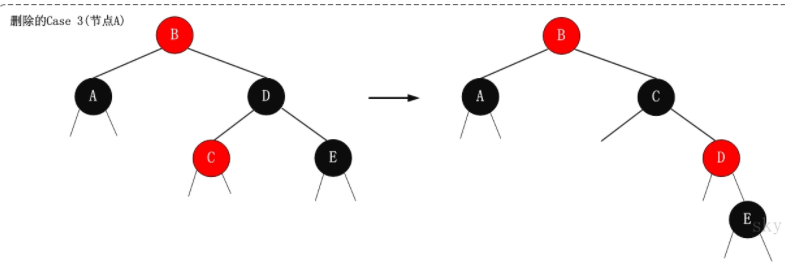


处理：

①x的兄弟节点设为红色

②x的父节点设为新的x节点

**Case3 x是黑黑节点，x的兄弟节点是黑色，x的兄弟节点左孩子红色，右孩子黑色**



处理：

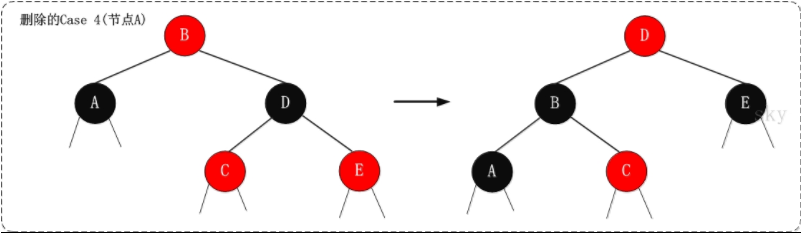
①x的兄弟节点的左孩子设为黑，

②x兄弟节点设置为红色

③对x的兄弟节点进行右旋，

④右旋后，重新设置x的兄弟节点

**Case4 x是黑黑节点，兄弟是黑的，兄弟的右孩子是红色，左孩子任意**



处理;

①将父节点颜色给兄弟节点

②将x父节点设为黑色

③将x兄弟节点的右孩子设为黑色

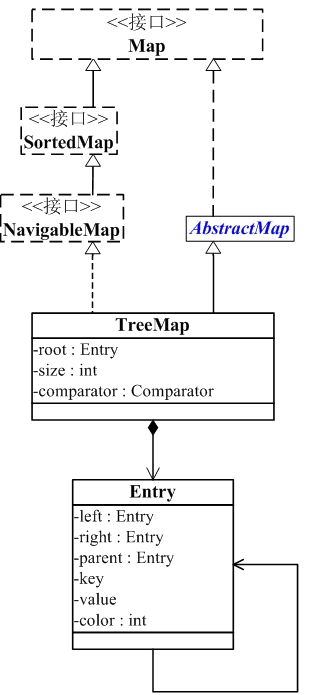
④对父节点进行左旋

⑤设置x为根节点

# TreeMap与TreeSet

**3.1TreeMap源码分析**

使用TreeMap存储进行排序，我们自定义的类，必须自己定义比较机制，一种是user类去实现**comparable接口**，并实现compareTo方法，另一种是mycomparator写一个类去实现**comparator接口**实现compare方法，将mycomparator作为参数传入到treemap构造方法



**3.1.1类名及其成员变量**

|  |
| --- |
| // 比较器对象  private final Comparator<? super K> comparator;  // 根节点  private transient Entry<K,V> root;  // 集合大小  private transient int size = 0;  // 树结构被修改的次数  private transient int modCount = 0;  // 静态内部类用来表示节点类型  static final class Entry<K,V> implements Map.Entry<K,V> {  K key; // 键  V value; // 值  Entry<K,V> left; // 指向左子树的引用（指针）  Entry<K,V> right; // 指向右子树的引用（指针）  Entry<K,V> parent; // 指向父节点的引用（指针）  boolean color = BLACK; //  } |

**3.1.2类的构造方法**

|  |
| --- |
| **默认comparator为null**  public TreeMap() { // 1,无参构造方法  comparator = null; // 默认比较机制  }  **2，自定义比较器的构造方法，实现comparator接口，并作为参数传入**  public TreeMap(Comparator<? super K> comparator) { // this.comparator = comparator;  }  3，构造已知Map对象为TreeMap，比较器仍然为空  public TreeMap(Map<? extends K, ? extends V> m) { //  comparator = null; // 默认比较机制  putAll(m);  }  4，构造已知的SortedMap对象为TreeMap  public TreeMap(SortedMap<K, ? extends V> m) { //  comparator = m.comparator(); // 使用已知对象的构造器  try {  此方法为排序集合中构造TreeMap，复杂度为线性  buildFromSorted(m.size(), m.entrySet().iterator(), null, null);  } catch (java.io.IOException cannotHappen) {  } catch (ClassNotFoundException cannotHappen) {  }  }  这里插入讲putAll方法  **public** **void** putAll(Map<? **extends** K, ? **extends** V> map) {  **int** mapSize = map.size();  **if** (size==0 && mapSize!=0 && map **instanceof** SortedMap) {  Comparator<?> c = ((SortedMap<?,?>)map).comparator();  **if** (c == comparator || (c != **null** && c.equals(comparator))) {  ++modCount;  **try** {  buildFromSorted(mapSize, map.entrySet().iterator(),  **null**, **null**);  } **catch** (java.io.IOException cannotHappen) {  } **catch** (ClassNotFoundException cannotHappen) {  }  **return**;  }  }  **super**.putAll(map);  }  ①获得sortedmap的size  ②**如果size为0，map的size不为0，且map是sortedmap类型**  **---2.1获得map的比较器，**  **------2.1.1如果比较器与当前的比较器一样，执行buildfromsorted，后面分析**  **------2.1.2如果不一样，就调用父类putAll方法，内部调用了Treemap的put方法**  ③如果②的条件不满足，就直接条用putall方法  ----------------------------------  底层用TreeMap自身的put方法，后面介绍  **public** **void** putAll(Map<? **extends** K, ? **extends** V> m) {  **for** (Map.Entry<? **extends** K, ? **extends** V> e : m.entrySet())  put(e.getKey(), e.getValue());  }  --------------------------------------  **private** **void** buildFromSorted(**int** size, Iterator<?> it,  java.io.ObjectInputStream str,  V defaultVal)  **throws** java.io.IOException, ClassNotFoundException {  **this**.size = size;  root = buildFromSorted(0, 0, size-1, *computeRedLevel*(size),  it, str, defaultVal);  }  -------------------------------------------------------------  **计算完全二叉树的层数**  **private** **static** **int** computeRedLevel(**int** sz) {  **int** level = 0;  **for** (**int** m = sz - 1; m >= 0; m = m / 2 - 1)  level++;  **return** level;  }  高度为2的树的最后一个节点的索引为2，类推高度为3的最后一个节点为6，满足m = (m + 1) \* 2。  -----------------------------------------------------------  分为三部分；  **private** **final** Entry<K,V> buildFromSorted(**int** level, **int** lo, **int** hi,  **int** redLevel,Iterator<?> it,java.io.ObjectInputStream str  V defaultVal)  **throws** java.io.IOException, ClassNotFoundException {    **if** (hi < lo) **return** **null**;  **int** mid = (lo + hi) >>> 1;  Entry<K,V> left = **null**;  递归构造左节点  **if** (lo < mid)  left = buildFromSorted(level+1, lo, mid - 1, redLevel,  it, str, defaultVal);  构造middle节点；  K key;  V value;  为middle的key和value设定值  如果迭代器为空  ----key从流中获取，如果defaultval为空，value就是流中的对象，如果defaultval不为空，value就是defaultval；  如果迭代器不为空  ----如果defaultval为空，将迭代器的next转为mapentry，设置key和value  ----如果defaultval不为空，key为迭代器的next转为key，value为defaultval  **if** (it != **null**) {  **if** (defaultVal==**null**) {  Map.Entry<?,?> entry = (Map.Entry<?,?>)it.next();  key = (K)entry.getKey();  value = (V)entry.getValue();  } **else** {  key = (K)it.next();  value = defaultVal;  }  } **else** { // use stream  key = (K) str.readObject();  value = (defaultVal != **null** ? defaultVal : (V) str.readObject());  }  将key和value存到middle节点中  Entry<K,V> middle = **new** Entry<>(key, value, **null**);  只要把前面3层的结点都设置为黑色，第四层的节点设置为红色，则构造完的树，就是红黑树。level从0开始的，所以上述9个节点，计算出来的是3，实际上就是代表的第4层  **if** (level == redLevel)  middle.color = ***RED***;  如果左节点不为空，就将左节点作为middle的左孩子  **if** (left != **null**) {  middle.left = left;  left.parent = middle;  }  构造右节点  **if** (mid < hi) {  Entry<K,V> right = buildFromSorted(level+1, mid+1, hi, redLevel, it, str, defaultVal);  middle.right = right;  right.parent = middle;  }  **return** middle;  }  大体思路：  ①如果hi < lo，return null，表示为空，如果hi>=lo表示当前子树构造完成  ②获取mid 的值，lo + hi的一半  ③定义left entry为null；如果lo < mid; 构造左子树（left = 递归，其中level+1， hi = mid - 1）  ④处理当前根节点的key和value；如果迭代器不为空，取map的key，value要么等于map的value，要么等于defaultval；如果迭代器为空，从str流中取key，value要么等于流的对象，要么等于defaultval；  ⑤创建middle当前根节点，设定key，value；如果level==redlevel，标红，表示叶子节点即最后一层；  ⑥如果left不为空，设定middle的left等于left，left的parent等于middle  ⑦如果mid < hi；构建右子树（right = 递归，level+ 1，lo =mid +1）；结束后将right作为mid的右孩子，mid作为right的父亲；  ⑧返回middle |

**3.1.3查找方法**

|  |
| --- |
| **底层调用getEntry，返回value值**  **public** V get(Object key) {  Entry<K,V> p = getEntry(key);  **return** (p==**null** ? **null** : p.value);  }  **如果比较器不为null，调用**getEntryUsingComparator查找，如果key为null，抛出异常；**如果比较器为null，调用key实现的comparable接口的compareTo方法与红黑树的节点进行比较；小于0往左，大于0往右；**  **final** Entry<K,V> getEntry(Object key) {  // Offload comparator-based version for sake of performance  **if** (comparator != **null**)  **return** getEntryUsingComparator(key);  **if** (key == **null**)  **throw** **new** NullPointerException();  @SuppressWarnings("unchecked")  Comparable<? **super** K> k = (Comparable<? **super** K>) key;  Entry<K,V> p = root;  **while** (p != **null**) {  **int** cmp = k.compareTo(p.key);  **if** (cmp < 0)  p = p.left;  **else** **if** (cmp > 0)  p = p.right;  **else**  **return** p;  }  **return** **null**;  }  用比较器进行比较，如果比较器不为空，调用compare方法将key和红黑树的节点比较，同样小于0往左；大于0往右  **final** Entry<K,V> getEntryUsingComparator(Object key) {  @SuppressWarnings("unchecked")  K k = (K) key;  Comparator<? **super** K> cpr = comparator;  **if** (cpr != **null**) {  Entry<K,V> p = root;  **while** (p != **null**) {  **int** cmp = cpr.compare(k, p.key);  **if** (cmp < 0)  p = p.left;  **else** **if** (cmp > 0)  p = p.right;  **else**  **return** p;  }  }  **return** **null**;  }  **获得红黑树的根节点，一直往左，就能得到最小的**  **final** Entry<K,V> getFirstEntry() {  Entry<K,V> p = root;  **if** (p != **null**)  **while** (p.left != **null**)  p = p.left;  **return** p;  }  **获得红黑树的根节点，一直往右，就能得到最大的**  **final** Entry<K,V> getLastEntry() {  Entry<K,V> p = root;  **if** (p != **null**)  **while** (p.right != **null**)  p = p.right;  **return** p;  } |

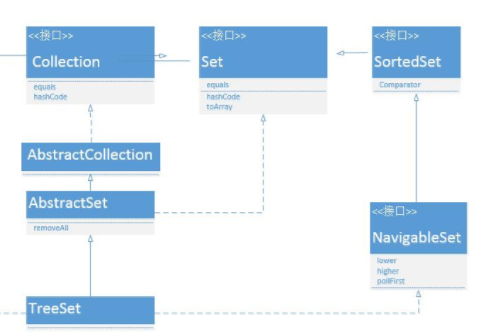
**3.1.4添加操作**

|  |
| --- |
| **比较方法：如果comparator为null，调用key的接口或者对象中带的compareto方法，否则就是用比较器的compare**  **final** **int** compare(Object k1, Object k2) {  **return** comparator==**null** ? ((Comparable<? **super** K>)k1).compareTo((K)k2)  : comparator.compare((K)k1, (K)k2);  }  **public** V put(K key, V value) {  Entry<K,V> t = root;  **if** (t == **null**) {  再次判断key不能为null  compare(key, key); // type (and possibly null) check  root = **new** Entry<>(key, value, **null**);  size = 1;  modCount++;  **return** **null**;  }  **int** cmp;  Entry<K,V> parent;  // split comparator and comparable paths  Comparator<? **super** K> cpr = comparator;  **if** (cpr != **null**) {  **do** {  parent = t;  cmp = cpr.compare(key, t.key);  **if** (cmp < 0)  t = t.left;  **else** **if** (cmp > 0)  t = t.right;  **else**  **return** t.setValue(value);  } **while** (t != **null**);  }  **else** {  **if** (key == **null**)  **throw** **new** NullPointerException();  @SuppressWarnings("unchecked")  Comparable<? **super** K> k = (Comparable<? **super** K>) key;  **do** {  parent = t;  cmp = k.compareTo(t.key);  **if** (cmp < 0)  t = t.left;  **else** **if** (cmp > 0)  t = t.right;  **else**  **return** t.setValue(value);  } **while** (t != **null**);  }  Entry<K,V> e = **new** Entry<>(key, value, parent);  **if** (cmp < 0)  parent.left = e;  **else**  parent.right = e;  fixAfterInsertion(e);  size++;  modCount++;  **return** **null**;  }  ①得到根节点，如果根节点为空，检查key不能为空，然后将key，value构造一个entry，赋值为根节点  ②定义父亲节点，即插入节点的父亲节点，也就是插入的位置  ③寻找带插入的位置，也就是找parent；如果比较器不为空，用比较器的compare方法，如果比较器为空，使用key实现的接口或者自带的compareTo方法，小于0往左找，大于0往右找；如果等于0，调用setValue，将value值替换，返回新的value；并将当前节点赋值给parent  ④根据比较的结果大于0还是小于0，将待插入节点设置为parent的左孩子或右孩子  ⑤调整红黑树 |

**3.1.5删除操作**

|  |
| --- |
| **获取key对应的entry，如果是null，返回null，否则获取旧值，调用deleteEntry，返回旧值**  **public** V remove(Object key) {  Entry<K,V> p = getEntry(key);  **if** (p == **null**)  **return** **null**;  V oldValue = p.value;  deleteEntry(p);  **return** oldValue;  }  寻找t的后继节点；右子树不为空，就右子树最左，，否则向上找到第一个左父节点  目的在于找到最接近t的节点进行替换  **static** <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {  **if** (t == **null**)  **return** **null**;  **else** **if** (t.right != **null**) {  Entry<K,V> p = t.right;  **while** (p.left != **null**)  p = p.left;  **return** p;  } **else** {  Entry<K,V> p = t.parent;  Entry<K,V> ch = t;  **while** (p != **null** && ch == p.right) {  ch = p;  p = p.parent;  }  **return** p;  }  }  **private** **void** deleteEntry(Entry<K,V> p) {  modCount++;  size--;  **if** (p.left != **null** && p.right != **null**) {  Entry<K,V> s = *successor*(p);  p.key = s.key;  p.value = s.value;  p = s;  } // p has 2 children  // Start fixup at replacement node, if it exists.  Entry<K,V> replacement = (p.left != **null** ? p.left : p.right);  **if** (replacement != **null**) {  // Link replacement to parent  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;  // Null out links so they are OK to use by fixAfterDeletion.  p.left = p.right = p.parent = **null**;  // Fix replacement  **if** (p.color == ***BLACK***)  fixAfterDeletion(replacement);  } **else** **if** (p.parent == **null**) { // return if we are the only node.  root = **null**;  } **else** { // No children. Use self as phantom replacement and unlink.  **if** (p.color == ***BLACK***)  fixAfterDeletion(p);  **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**;  }  }  }  大体思路：  ①如果有两个孩子，找到后继节点s，将p的key，value用s替代；p指向s，接下来对通过p间接操作s，则s变为只有左孩子或右孩子之一，或者没有孩子  ②如果p只有一个孩子，将p的孩子赋值给replacement，如果rep不为空，就将rep的parent指向p的parent；如果p的parent为空，说明p为根节点，root指向rep；如果p是左孩子，就p的parent的left指向rep，否则right指向rep，然后将p的左右孩子和父亲设为null，如果p是黑色，要进行调整  如果rep为空，且p的parent也为空，那么说明只有一个p节点，则root =null  ③如果rep为空，p的parent不为空，说明p没有孩子；如果p是黑色，调整红黑树，如果p是左孩子，p的parent的left为空，否则right为空；然后p的parent为null |

**3.2TreeSet源码分析**



TreeSet是基于TreeMap实现的一个保证元素有序的集合，其底层的存储盒TreeMap一样也是一棵红黑树，执行和TreeMap的增删改查操作

**3.2.1内部属性**

|  |
| --- |
| **定义navigablemap接口类型m**  **private** **transient** NavigableMap<E,Object> m;  object类型的present，作为key-value中的value值，将需要保存的值放入key中  **private** **static** **final** Object ***PRESENT*** = **new** Object(); |

3.2.2**构造方法**

|  |
| --- |
| 多态，其实m是TreeMap的实现类  TreeSet(NavigableMap<E,Object> m) {  **this**.m = m;  }  无参构造器，传入一个TreeMap对象，key为E类型，value为Object类型  **public** TreeSet() {  **this**(**new** TreeMap<E,Object>());  }  传入一个带比较器的TreeMap  **public** TreeSet(Comparator<? **super** E> comparator) {  **this**(**new** TreeMap<>(comparator));  }  将指定集合放入treeset中  **public** TreeSet(Collection<? **extends** E> c) {  **this**();  addAll(c);  }  将有序集合s方法TreeSet中  **public** TreeSet(SortedSet<E> s) {  **this**(s.comparator());  addAll(s);  }  所有的构造方法都是调用TreeSet(NavigableMap<E,Object> m)因此m一定会被初始化为TreeMap类型  将集合加入TreeSet中，如果m是TreeMap类型，且c是有序的，使用map的方法进行添加，value就是Present；，如果不是有序的，就使用c集合实现类的addall方法  **public** **boolean** addAll(Collection<? **extends** E> c) {  // Use linear-time version if applicable  **if** (m.size()==0 && c.size() > 0 &&  c **instanceof** SortedSet &&  m **instanceof** TreeMap) {  SortedSet<? **extends** E> set = (SortedSet<? **extends** E>) c;  TreeMap<E,Object> map = (TreeMap<E, Object>) m;  Comparator<?> cc = set.comparator();  Comparator<? **super** E> mc = map.comparator();  **if** (cc==mc || (cc != **null** && cc.equals(mc))) {  map.addAllForTreeSet(set, ***PRESENT***);  **return** **true**;  }  }  **return** **super**.addAll(c);  }  默认调用有序的建立treemap的方法；  **void** addAllForTreeSet(SortedSet<? **extends** K> set, V defaultVal) {  **try** {  buildFromSorted(set.size(), set.iterator(), **null**, defaultVal);  } **catch** (java.io.IOException cannotHappen) {  } **catch** (ClassNotFoundException cannotHappen) {  }  } |