为了能以较快的时间O(logN)来搜索一棵树，需要保证树总是平衡的（或者至少大部分是平衡的），这就是说对树中的每个节点在它左边的后代数目和在它右边的后代数目应该大致相等。红-黑树的就是这样的一棵平衡树，对一个要插入的数据项，插入例程要检查会不会破坏树的特征，如果破坏了，程序就会进行纠正，根据需要改变树的结构，从而保持树的平衡。那么红-黑树都有哪些特征呢?

1.红-黑树的特征

它主要有两个特征：1.节点都有颜色；2.在插入和删除的过程中，要遵循保持这些颜色的不同排列的规则。首先第一个特征很好解决，在节点类中店家一个数据字段，例如boolean型变量，以此来表示节点的颜色信息。第二个特征比较复杂，红-黑树有它的几个规则，如果遵循这些规则，那么树就是平衡的。红-黑树的主要规则如下：

1.每个节点不是红色就是黑色的；

2.根节点总是黑色的；

3.如果节点是红色的，则它的子节点必须是黑色的（反之不一定）；

4.从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）。

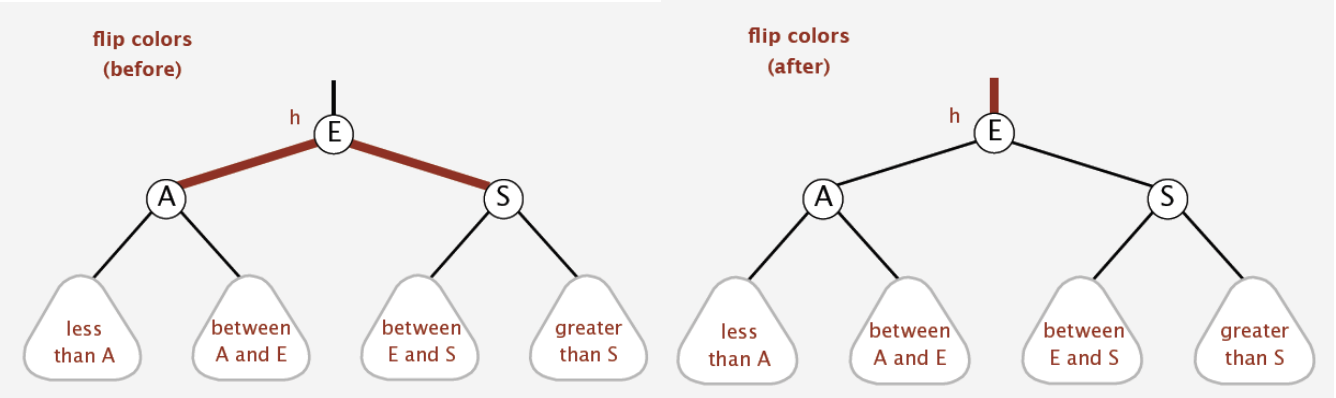
在红-黑树中插入的节点都是红色的，这不是偶然的，因为插入一个红色节点比插入一个黑色节点违背红-黑规则的可能性更小。原因是：插入黑色节点总会改变黑色高度（违背规则4），但是插入红色节点只有一半的机会会违背规则3。另外违背规则3比违背规则4要更容易修正。当插入一个新的节点时，可能会破坏这种平衡性，那么红-黑树是如何修正的呢?

2.平衡性的修正

红-黑树主要通过三种方式对平衡进行修正，改变节点颜色、左旋和右旋。这看起来有点抽象，我们分别来介绍它们。

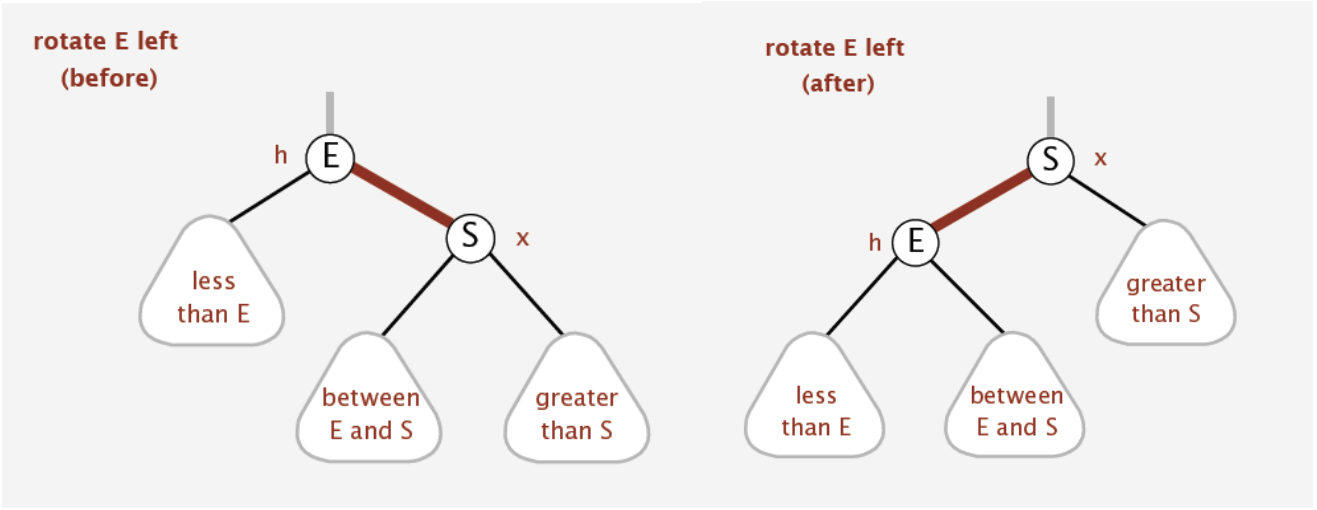
1.变色

改变节点颜色比较容易理解，因为它违背了规则3。假设现在有个节点E，然后插入节点A和节点S，节点A在左子节点，S在右子节点，目前是平衡的。如果此时再插一个节点，那么就出现了不平衡了，因为红色节点的子节点必须为黑色，但是新插的节点是红色的。所以这时候就必须改变节点颜色了。所以我们将根的两个子节点从红色变为黑色（至于为什么都要变，下面插入的时候会详细介绍），将父节点会从黑色变成红色。可以用如下示意图表示一下：

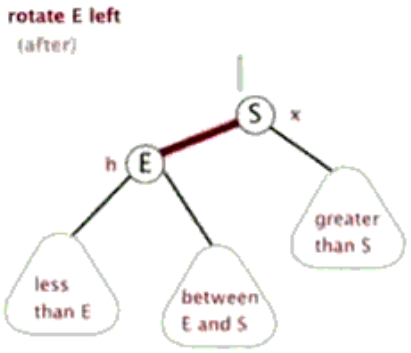


2.左旋

通常左旋操作用于将一个向右倾斜的红色链接旋转为向左链接。示意图如下：

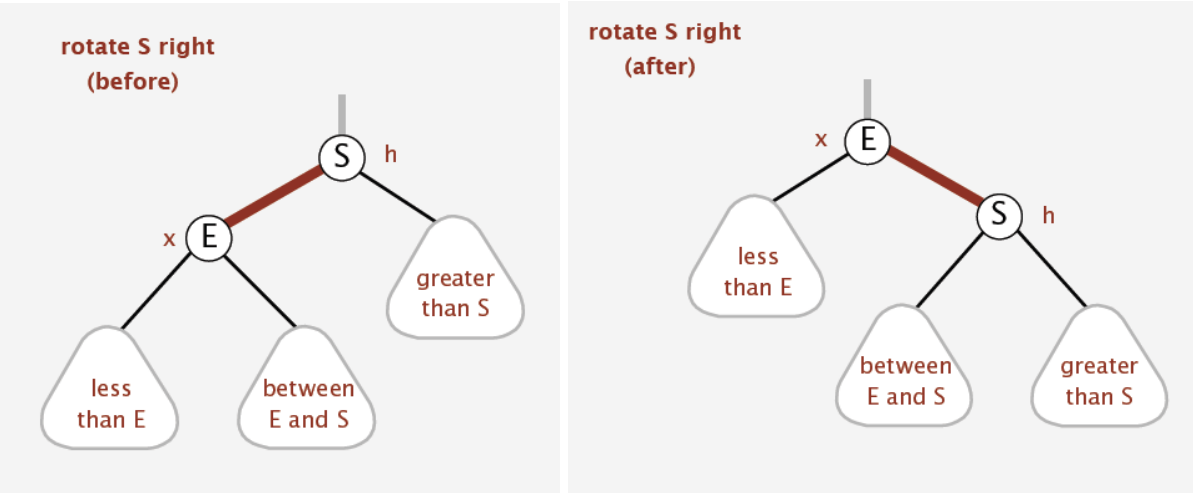


左旋有个很萌萌哒的动态示意图，可以方便理解：

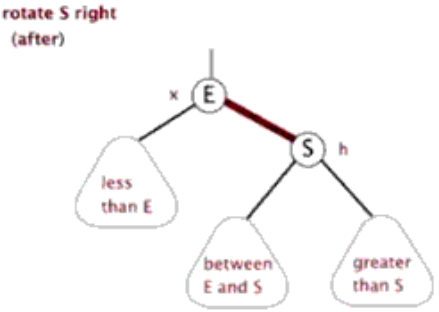


3.右旋

右旋可左旋刚好相反，这里不再赘述，直接看示意图：



当然咯，右旋也有个萌萌的动态图：



这里主要介绍了红-黑树对平衡的三种修正方式，大家有个感性的认识，那么什么时候该修正呢？什么时候该用哪种修正呢？这将是接下来我们要探讨的问题。

3.红-黑树的操作

红-黑树的基本操作是添加、删除和旋转。对红-黑树进行添加或删除后，可能会破坏其平衡性，会用到哪种旋转方式去修正呢？我们首先对红-黑树的节点做一介绍，然后分别对左旋和右旋的具体实现做一分析，最后我们探讨下红-黑树的具体操作。

1.红-黑树的节点

红-黑树是对二叉搜索树的改进，所以其节点与二叉搜索树是差不多的，只不过在它基础上增加了一个boolean型变量来表示节点的颜色，具体看RBNode<T>类：

```
public class RBNode<T extends Comparable<T>>{
    boolean color; //颜色
    T key; //关键字(键值)
    RBNode<T> left; //左子节点
    RBNode<T> right; //右子节点
    RBNode<T> parent; //父节点

    public RBNode(T key, boolean color, RBNode<T> parent, RBNode<T> left, RBNode<T> right) {
        this.key = key;
        this.color = color;
        this.parent = parent;
        this.left = left;
        this.right = right;
    }

    public T getKey() {
        return key;
```

```
    }

    public String toString() {
        return "" + key + (this.color == RED? "R" : "B");
    }
}
```

2.左旋具体实现

上面对左旋的概念已经有了感性的认识了，这里就不再赘述了，我们从下面的代码中结合上面的示意图，探讨一下左旋的具体实现：

```
/*************对红黑树节点x进行左旋操作 ******************/
/*
 * 左旋示意图：对节点x进行左旋
 *     p                       p
 *    /                       /
 *   x                       y
 *  / \                     / \
 * lx  y      ----->       x  ry
 *    / \                 / \
 *   ly ry               lx ly
 * 左旋做了三件事：
 * 1. 将y的左子节点赋给x的右子节点,并将x赋给y左子节点的父节点(y左子节点非空时)
 * 2. 将x的父节点p(非空时)赋给y的父节点，同时更新p的子节点为y(左或右)
 * 3. 将y的左子节点设为x，将x的父节点设为y
 */
private void leftRotate(RBNode<T> x) {
    //1. 将y的左子节点赋给x的右子节点，并将x赋给y左子节点的父节点(y左子节点非空时)
    RBNode<T> y = x.right;
    x.right = y.left;

    if(y.left != null)
        y.left.parent = x;

    //2. 将x的父节点p(非空时)赋给y的父节点，同时更新p的子节点为y(左或右)
    y.parent = x.parent;

    if(x.parent == null) {
        this.root = y; //如果x的父节点为空，则将y设为父节点
    } else {
        if(x == x.parent.left) //如果x是左子节点
            x.parent.left = y; //则也将y设为左子节点
        else
            x.parent.right = y;//否则将y设为右子节点
    }

    //3. 将y的左子节点设为x，将x的父节点设为y
    y.left = x;
    x.parent = y;
}
```

3.右旋具体实现

上面对右旋的概念已经有了感性的认识了，这里也不再赘述了，我们从下面的代码中结合上面的示意图，探讨一下右旋的具体实现：

```
/*************对红黑树节点y进行右旋操作 ******************/
/*
 * 左旋示意图：对节点y进行右旋
 *        p                   p
 *       /                   /
 *      y                   x
 *     / \                 / \
 *    x  ry   ------>     lx  y
 *   / \                     / \
 * lx  rx                   rx ry
 * 右旋做了三件事：
 * 1. 将x的右子节点赋给y的左子节点,并将y赋给x右子节点的父节点(x右子节点非空时)
 * 2. 将y的父节点p(非空时)赋给x的父节点，同时更新p的子节点为x(左或右)
 * 3. 将x的右子节点设为y，将y的父节点设为x
 */
private void rightRotate(RBNode<T> y) {
    //1. 将y的左子节点赋给x的右子节点，并将x赋给y左子节点的父节点(y左子节点非空时)
    RBNode<T> x = y.left;
    y.left = x.right;

    if(x.right != null)
        x.right.parent = y;

    //2. 将x的父节点p(非空时)赋给y的父节点，同时更新p的子节点为y(左或右)
    x.parent = y.parent;

    if(y.parent == null) {
        this.root = x; //如果x的父节点为空，则将y设为父节点
    } else {
        if(y == y.parent.right) //如果x是左子节点
            y.parent.right = x; //则也将y设为左子节点
        else
            y.parent.left = x;//否则将y设为右子节点
    }

    //3. 将y的左子节点设为x，将x的父节点设为y
    x.right = y;
    y.parent = x;
}
```

4.插入操作

分析完了红-黑树中主要的旋转操作，接下来我们开始分析常见的插入、删除等操作了。这里先分析插入操作。 由于红-黑树是二叉搜索树的改进，所以插入操作的前半工作时相同的，即先找到待插入的位置，再将节点插入，先来看看插入的前半

段代码：

```
/********************** 向红黑树中插入节点 **********************/
public void insert(T key) {
    RBNode<T> node = new RBNode<T>(key, RED, null, null, null);
    if(node != null)
        insert(node);
}

//将节点插入到红黑树中，这个过程与二叉搜索树是一样的
private void insert(RBNode<T> node) {
    RBNode<T> current = null; //表示最后node的父节点
    RBNode<T> x = this.root; //用来向下搜索用的

    //1. 找到插入的位置
    while(x != null) {
        current = x;
        int cmp = node.key.compareTo(x.key);
        if(cmp < 0)
            x = x.left;
        else
            x = x.right;
    }
    node.parent = current; //找到了位置，将当前current作为node的父节点

    //2. 接下来判断node是插在左子节点还是右子节点
    if(current != null) {
        int cmp = node.key.compareTo(current.key);
        if(cmp < 0)
            current.left = node;
        else
            current.right = node;
    } else {
        this.root = node;
    }

    //3. 将它重新修整为一颗红黑树
    insertFixUp(node);
}
```

这与二叉搜索树中实现的思路一模一样，这里不再赘述，主要看看方法里面最后一步insertFixUp操作。因为插入后可能会导致树的不平衡，insertFixUp方法里主要是分情况讨论，分析何时变色，何时左旋，何时右旋。我们先从理论上分析具体的情况，然后再看insertFixUp方法的具体实现。

如果是第一次插入，由于原树为空，所以只会违反红-黑树的规则2，所以只要把根节点涂黑即可；如果插入节点的父节点是黑色的，那不会违背红-黑树的规则，什么也不需要做；但是遇到如下三种情况时，我们就要开始变色和旋转了：

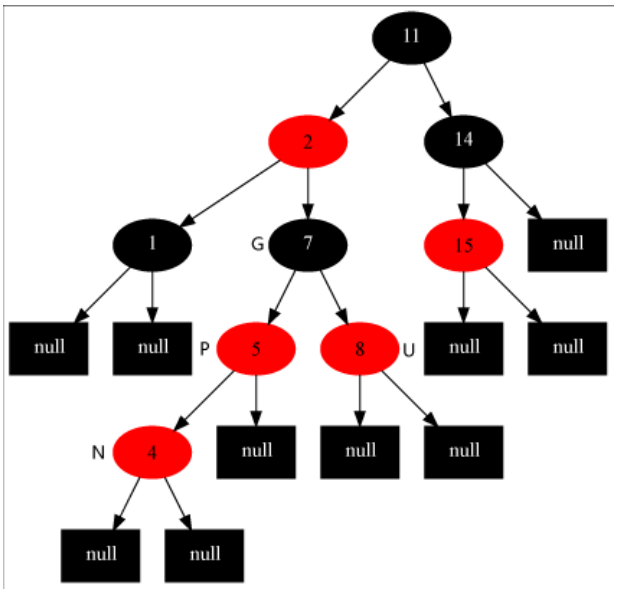
1. 插入节点的父节点和其叔叔节点（祖父节点的另一个子节点）均为红色的；

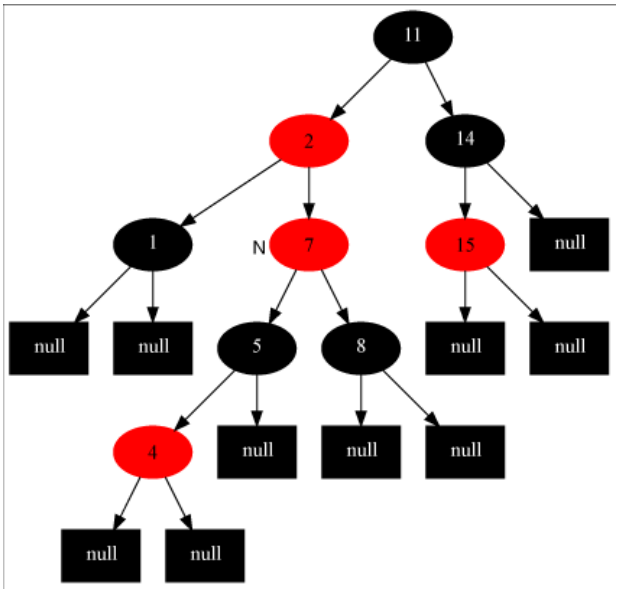
2. 插入节点的父节点是红色，叔叔节点是黑色，且插入节点是其父节点的右子节点；

3. 插入节点的父节点是红色，叔叔节点是黑色，且插入节点是其父节点的左子节点。

下面我们先挨个分析这三种情况都需要如何操作，然后再给出实现代码。

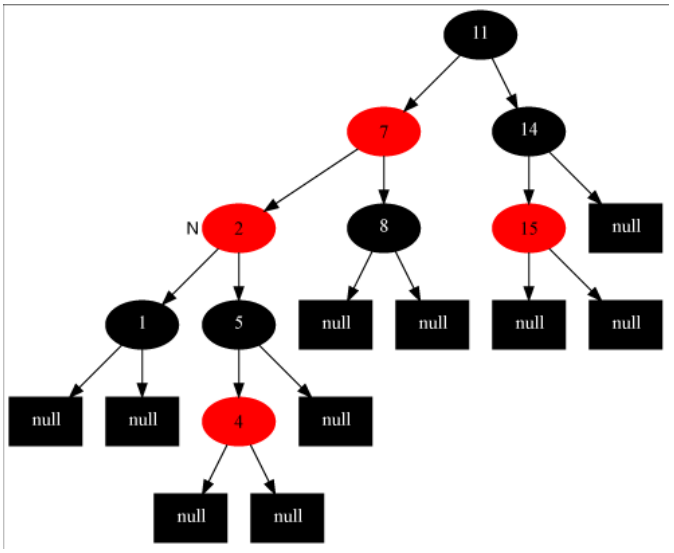
对于情况1：**插入节点的父节点和其叔叔节点（祖父节点的另一个子节点）均为红色的。**此时，肯定存在祖父节点，但是不知道父节点是其左子节点还是右子节点，但是由于对称性，我们只要讨论出一边的情况，另一种情况自然也与之对应。这里考虑父节点是祖父节点的左子节点的情况，如下左图所示：

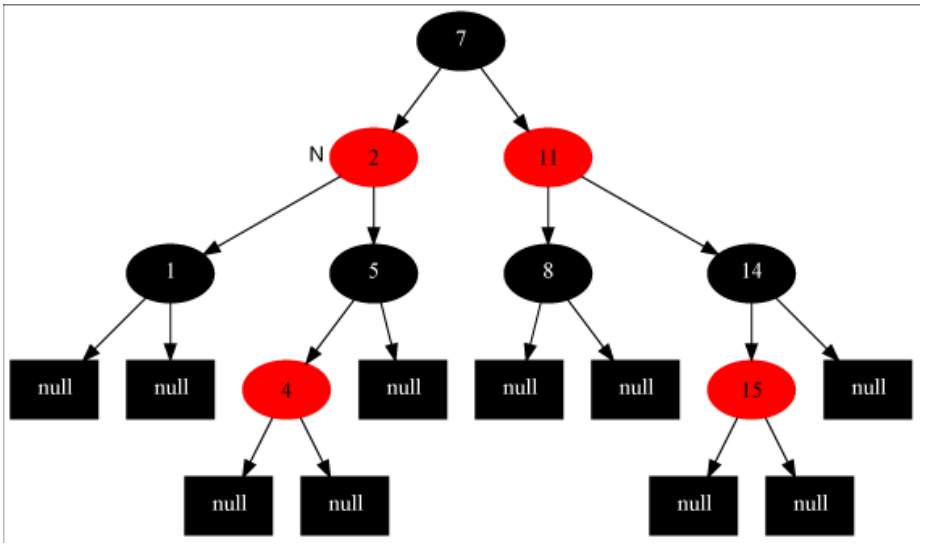




对于这种情况，我们要做的操作有：将当前节点(4)的父节点(5)和叔叔节点(8)涂黑，将祖父节点(7)涂红，变成上右图所示的情况。再将当前节点指向其祖父节点，再次从新的当前节点开始算法（具体等下看下面的程序）。这样上右图就变成了情况2了。

对于情况2：**插入节点的父节点是红色，叔叔节点是黑色，且插入节点是其父节点的右子节点。**我们要做的操作有：将当前节点(7)的父节点(2)作为新的节点，以新的当前节点为支点做左旋操作。完成后如左下图所示，这样左下图就变成情况3了。





对于情况3：**插入节点的父节点是红色，叔叔节点是黑色，且插入节点是其父节点的左子节点。**我们要做的操作有：将当前节点的父节点(7)涂黑，将祖父节点(11)涂红，在祖父节点为支点做右旋操作。最后把根节点涂黑，整个红-黑树重新恢复了平衡，如右上图所示。至此，插入操作完成！

我们可以看出，如果是从情况1开始发生的，必然会走完情况2和3，也就是说这是一整个流程，当然咯，实际中可能不一定会从情况1发生，如果从情况2开始发生，那再走个情况3即可完成调整，如果直接只要调整情况3，那么前两种情况均不需要调整了。故变色和旋转之间的先后关系可以表示为：变色->左旋->右旋。

至此，我们完成了全部的插入操作。下面我们看看insertFixUp方法中的具体实现（可以结合上面的分析图，更加利与理解）：

[java]

```java
private void insertFixUp(RBNode<T> node) {
    RBNode<T> parent, gparent; //定义父节点和祖父节点

    //需要修整的条件：父节点存在，且父节点的颜色是红色
    while(((parent = parentOf(node)) != null) && isRed(parent)) {
        gparent = parentOf(parent);//获得祖父节点

        //若父节点是祖父节点的左子节点，下面else与其相反
        if(parent == gparent.left) {
            RBNode<T> uncle = gparent.right; //获得叔叔节点

            //case1: 叔叔节点也是红色
            if(uncle != null && isRed(uncle)) {
                setBlack(parent); //把父节点和叔叔节点涂黑
                setBlack(uncle);
                setRed(gparent); //把祖父节点涂红
                node = gparent; //将位置放到祖父节点处
                continue; //继续while，重新判断
            }
```

```
20.
21.             //case2: 叔叔节点是黑色，且当前节点是右子节点
22.             if(node == parent.right) {
23.                 leftRotate(parent); //从父节点处左旋
24.                 RBNode<T> tmp = parent; //然后将父节点和自己调换一下，为下面右旋做准备
25.                 parent = node;
26.                 node = tmp;
27.             }
28.
29.             //case3: 叔叔节点是黑色，且当前节点是左子节点
30.             setBlack(parent);
31.             setRed(gparent);
32.             rightRotate(gparent);
33.         } else { //若父节点是祖父节点的右子节点,与上面的完全相反，本质一样的
34.             RBNode<T> uncle = gparent.left;
35.
36.             //case1: 叔叔节点也是红色
37.             if(uncle != null & isRed(uncle)) {
38.                 setBlack(parent);
39.                 setBlack(uncle);
40.                 setRed(gparent);
41.                 node = gparent;
42.                 continue;
43.             }
44.
45.             //case2: 叔叔节点是黑色的，且当前节点是左子节点
46.             if(node == parent.left) {
47.                 rightRotate(parent);
48.                 RBNode<T> tmp = parent;
49.                 parent = node;
50.                 node = tmp;
51.             }
52.
53.             //case3: 叔叔节点是黑色的，且当前节点是右子节点
54.             setBlack(parent);
55.             setRed(gparent);
56.             leftRotate(gparent);
57.         }
58.     }
59.
60.     //将根节点设置为黑色
61.     setBlack(this.root);
62. }
```

5.删除操作

上面探讨完了红-黑树的插入操作，接下来讨论删除，红-黑树的删除和二叉查找树的删除是一样的，只不过删除后多了个平衡的修复而已。我们先来回忆一下二叉搜索树的删除（也可以直接阅读这篇博客：二叉搜索树）：

1. 如果待删除节点没有子节点，那么直接删掉即可；

2. 如果待删除节点只有一个子节点，那么直接删掉，并用其子节点去顶替它；

3. 如果待删除节点有两个子节点，这种情况比较复杂：首选找出它的后继节点，然后处理“后继节点”和“被删除节点的父节点”之间的关系，最后处理“后继节点的子节点”和“被删除节点的子节点”之间的关系。每一步中也会有不同的情况，我们结合下面代码的分析就能弄清楚，当然了，如果已经弄懂了二叉搜索树，那自然自然都能明白，这里就不赘述了。

我们来看一下删除操作的代码及注释：

```java
/********************** 删除红黑树中的节点 **********************/
public void remove(T key) {
    RBNode<T> node;
    if((node = search(root, key)) != null)
        remove(node);
}

private void remove(RBNode<T> node) {
    RBNode<T> child, parent;
    boolean color;

    //1. 被删除的节点“左右子节点都不为空”的情况
    if((node.left != null) && (node.right != null)) {
        //先找到被删除节点的后继节点，用它来取代被删除节点的位置
        RBNode<T> replace = node;
        //  1). 获取后继节点
        replace = replace.right;
        while(replace.left != null)
            replace = replace.left;

        //  2). 处理“后继节点”和“被删除节点的父节点”之间的关系
        if(parentOf(node) != null) { //要删除的节点不是根节点
            if(node == parentOf(node).left)
                parentOf(node).left = replace;
            else
                parentOf(node).right = replace;
        } else { //否则
            this.root = replace;
        }
```

```
30.
31.         //  3). 处理“后继节点的子节点”和“被删除节点的子节点”之间的关系
32.         child = replace.right; //后继节点肯定不存在左子节点!
33.         parent = parentOf(replace);
34.         color = colorOf(replace);//保存后继节点的颜色
35.         if(parent == node) { //后继节点是被删除节点的子节点
36.             parent = replace;
37.         } else { //否则
38.             if(child != null)
39.                 setParent(child, parent);
40.             parent.left = child;
41.             replace.right = node.right;
42.             setParent(node.right, replace);
43.         }
44.         replace.parent = node.parent;
45.         replace.color = node.color; //保持原来位置的颜色
46.         replace.left = node.left;
47.         node.left.parent = replace;
48.
49.         if(color == BLACK) { //4. 如果移走的后继节点颜色是黑色，重新修整红黑树
50.             removeFixUp(child, parent);//将后继节点的child和parent传进去
51.         }
52.         node = null;
53.         return;
54.     }
55. }
```

下面我们主要看看方法里面最后的removeFixUp操作。因为remove后可能会导致树的不平衡，removeFixUp方法里主要是分情况讨论，分析何时变色，何时左旋，何时右旋。我们同样先从理论上分析具体的情况，然后再看removeFixUp方法的具体实现。

从上面的代码中可以看出，删除某个节点后，会用它的后继节点来填上，并且后继节点会设置为和删除节点同样的颜色，所以删除节点的那个位置是不会破坏平衡的。可能破坏平衡的是后继节点原来的位置，因为后继节点拿走了，原来的位置结构改变了，这就会导致不平衡的出现。所以removeFixUp方法中传入的参数也是后继节点的子节点和父节点。

为了方便下文的叙述，我们现在约定：后继节点的子节点称为“当前节点”。

删除操作后，如果当前节点是黑色的根节点，那么不用任何操作，因为并没有破坏树的平衡性，即没有违背红-黑树的规则，这很好理解。如果当前节点是红色的，说明刚刚移走的后继节点是黑色的，那么不管后继节点的父节点是啥颜色，我们只要将当前节点涂黑就可以了，红-黑树的平衡性就可以恢复。但是如果遇到以下四种情况，我们就需要通过变色或旋转来恢复红-黑树的平衡了。

1. 当前节点是黑色的，且兄弟节点是红色的（那么父节点和兄弟节点的子节点肯定是黑色的）；

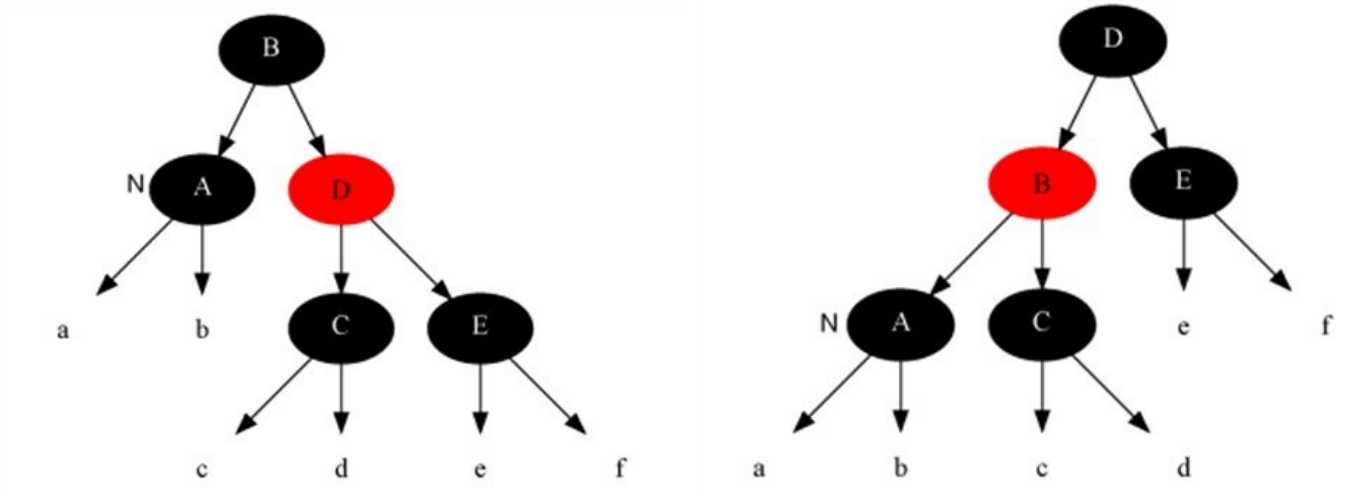
2. 当前节点是黑色的，且兄弟节点是黑色的，且兄弟节点的两个子节点均为黑色的；

3. 当前节点是黑色的，且兄弟节点是黑色的，且兄弟节点的左子节点是红色，右子节点时黑色的；

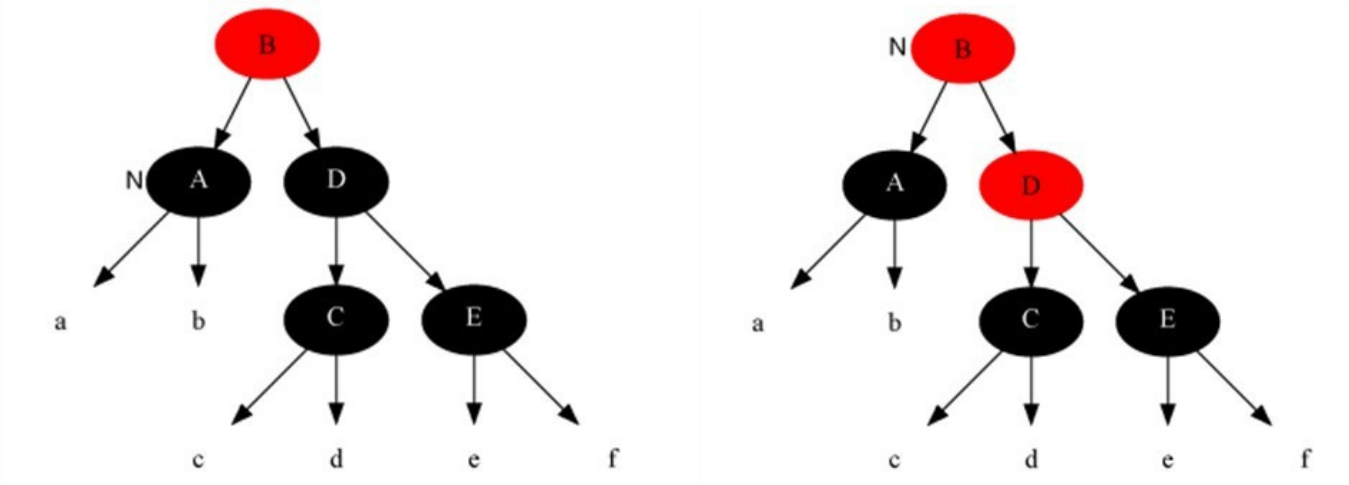
4. 当前节点是黑色的，且兄弟节点是黑色的，且兄弟节点的右子节点是红色，左子节点任意颜色。

以上四种情况中，我们可以看出2,3,4其实是“当前节点是黑色的，且兄弟节点是黑色的”的三种子集，等会在程序中可以体现出来。现在我们假设当前节点是左子节点（当然也可能是右子节点，跟左子节点相反即可，我们讨论一边就可以了），分别解决上面四种情况：

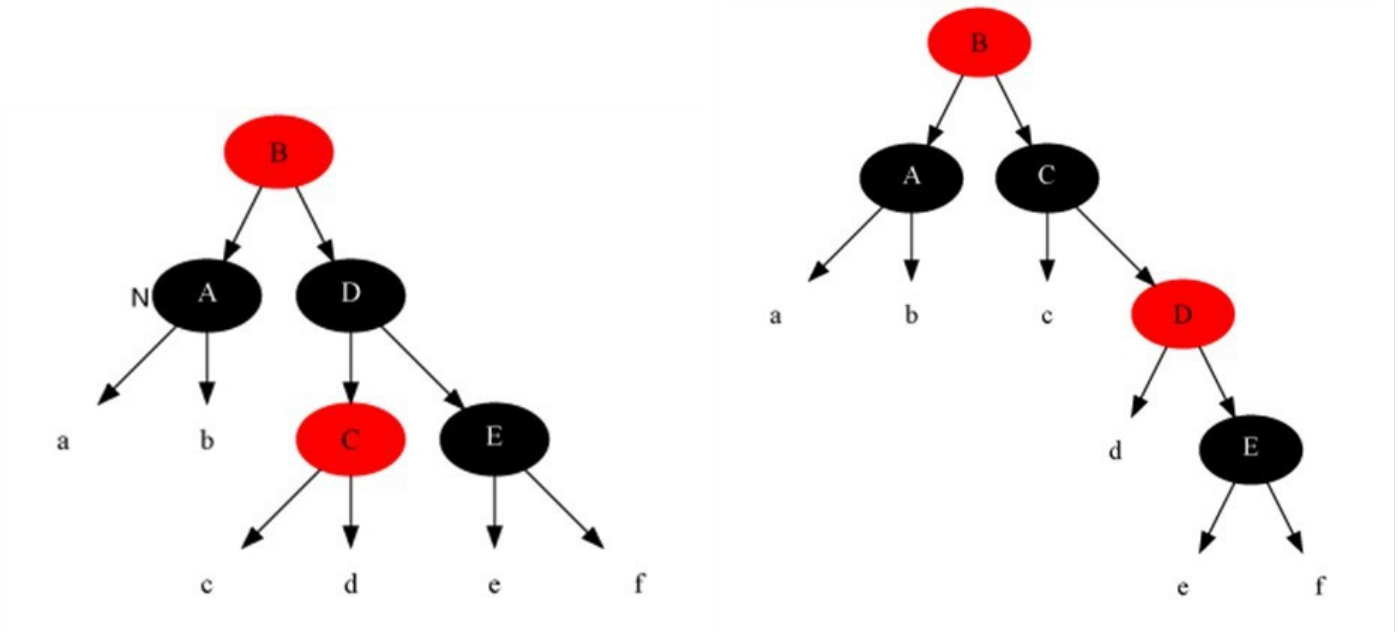
对于情况1：**当前节点是黑色的，且兄弟节点是红色的（那么父节点和兄弟节点的子节点肯定是黑色的）。**如左下图所示：A节点表示当前节点。针对这种情况，我们要做的操作有：将父节点（B）涂红，将兄弟节点（D）涂黑，然后将当前节点（A）的父节点（B）作为支点左旋，然后当前节点的兄弟节点就变成黑色的情况了（自然就转换成情况2，3,4的公有特征了），如右下图所示：



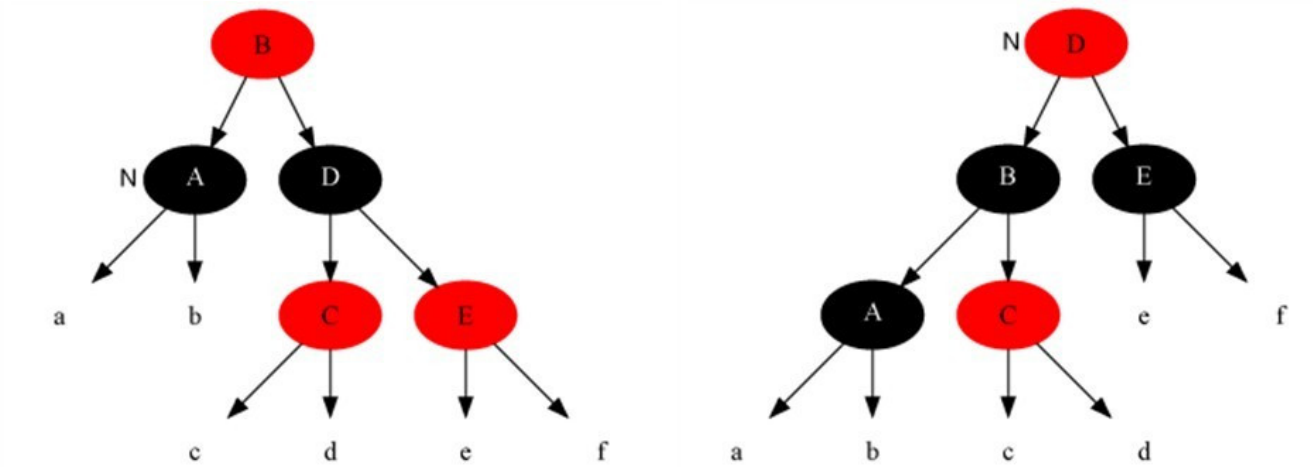
对于情况2：**当前节点是黑色的，且兄弟节点是黑色的，且兄弟节点的两个子节点均为黑色的。**如左下图所示，A表示当前节点。针对这种情况，我们要做的操作有：将兄弟节点（D）涂红，将当前节点指向其父节点（B），将其父节点指向当前节点的祖父节点，继续新的算法（具体见下面的程序），不需要旋转。这样变成了右下图所示的情况：



对于情况3：**当前节点是黑色的，且兄弟节点是黑色的，且兄弟节点的左子节点是红色，右子节点时黑色的。**如左下图所示，A是当前节点。针对这种情况，我们要做的操作有：把当前节点的兄弟节点（D）涂红，把兄弟节点的左子节点（C）涂黑，然后以兄弟节点作为支点做右旋操作。然后兄弟节点就变成黑色的，且兄弟节点的右子节点变成红色的情况（情况4）了。如右下图：



对于情况4：**当前节点是黑色的，且兄弟节点是黑色的，且兄弟节点的右子节点是红色，左子节点任意颜色。**如左下图所示：A为当前节点，针对这种情况，我们要做的操作有：把兄弟节点（D）涂成父节点的颜色，再把父节点（B）涂黑，把兄弟节点的右子节点（E）涂黑，然后以当前节点的父节点为支点做左旋操作。至此，删除修复算法就结束了，最后将根节点涂黑即可。



我们可以看出，如果是从情况1开始发生的，可能情况2，3，4中的一种：如果是情况2，就不可能再出现3和4；如果是情况3，必然会导致情况4的出现；如果2和3都不是，那必然是4。当然咯，实际中可能不一定会从情况1发生，这要看具体情况了。

至此，我们完成了全部的删除操作。下面我们看看removeFixUp方法中的具体实现（可以结合上面的分析图，更加利与理解）：

```java
//node表示待修正的节点，即后继节点的子节点（因为后继节点被挪到删除节点的位置去了）
private void removeFixUp(RBNode<T> node, RBNode<T> parent) {
    RBNode<T> other;

    while((node == null || isBlack(node)) && (node != this.root)) {
        if(parent.left == node) { //node是左子节点，下面else与这里的刚好相反
```

```
            other = parent.right; //node的兄弟节点
            if(isRed(other)) { //case1: node的兄弟节点other是红色的
                setBlack(other);
                setRed(parent);
                leftRotate(parent);
                other = parent.right;
            }

            //case2: node的兄弟节点other是黑色的，且other的两个子节点也都是黑色的
            if((other.left == null || isBlack(other.left)) &&
                    (other.right == null || isBlack(other.right))) {
                setRed(other);
                node = parent;
                parent = parentOf(node);
            } else {
                //case3: node的兄弟节点other是黑色的，且other的左子节点是红色，右子节点是黑色
                if(other.right == null || isBlack(other.right)) {
                    setBlack(other.left);
                    setRed(other);
                    rightRotate(other);
                    other = parent.right;
                }

                //case4: node的兄弟节点other是黑色的，且other的右子节点是红色，左子节点任意颜色
                setColor(other, colorOf(parent));
                setBlack(parent);
                setBlack(other.right);
                leftRotate(parent);
                node = this.root;
                break;
            }
        } else { //与上面的对称
            other = parent.left;

            if (isRed(other)) {
                // Case 1: node的兄弟other是红色的
                setBlack(other);
                setRed(parent);
                rightRotate(parent);
                other = parent.left;
            }

            if ((other.left==null || isBlack(other.left)) &&
```

```
                (other.right==null || isBlack(other.right))) {
                // Case 2: node的兄弟other是黑色，且other的俩个子节点都是黑色的
                setRed(other);
                node = parent;
                parent = parentOf(node);
            } else {

                if (other.left==null || isBlack(other.left)) {
                    // Case 3: node的兄弟other是黑色的，并且other的左子节点是红色，右子节点为黑色。
                    setBlack(other.right);
                    setRed(other);
                    leftRotate(other);
                    other = parent.left;
                }

                // Case 4: node的兄弟other是黑色的；并且other的左子节点是红色的，右子节点任意颜色
                setColor(other, colorOf(parent));
                setBlack(parent);
                setBlack(other.left);
                rightRotate(parent);
                node = this.root;
                break;
            }
        }
    }
    if (node!=null)
        setBlack(node);
}
```

4.完整源码

终于分析完了插入和删除操作的所有东西。另外，红-黑树还有一些其他操作，比如：查找特定值、遍历、返回最值、销毁树等操作我将放到源码中给大家呈现出来，详见下面红-黑树的完整代码。

[java]

```
package tree;
/**
 * @description implementation of Red-Black Tree by Java
 * @author eson_15
 * @param <T>
 * @date 2016-4-18 19:27:28
 */
public class RBTree<T extends Comparable<T>> {
```

```
9.      private RBNode<T> root; //根节点
10.     private static final boolean RED = false; //定义红黑树标志
11.     private static final boolean BLACK = true;
12.
13.     //内部类：节点类
14.     public class RBNode<T extends Comparable<T>>{
15.         boolean color; //颜色
16.         T key; //关键字(键值)
17.         RBNode<T> left; //左子节点
18.         RBNode<T> right; //右子节点
19.         RBNode<T> parent; //父节点
20.
21.         public RBNode(T key, boolean color, RBNode<T> parent, RBNode<T> left, RBNode<T> right) {
22.             this.key = key;
23.             this.color = color;
24.             this.parent = parent;
25.             this.left = left;
26.             this.right = right;
27.         }
28.
29.         public T getKey() {
30.             return key;
31.         }
32.
33.         public String toString() {
34.             return "" + key + (this.color == RED? "R" : "B");
35.         }
36.     }
37.
38.     public RBTree() {
39.         root = null;
40.     }
41.
42.     public RBNode<T> parentOf(RBNode<T> node) { //获得父节点
43.         return node != null? node.parent : null;
44.     }
45.
46.     public void setParent(RBNode<T> node, RBNode<T> parent) { //设置父节点
47.         if(node != null)
48.             node.parent = parent;
49.     }
50.
51.     public boolean colorOf(RBNode<T> node) { //获得节点的颜色
```

```
52.         return node != null? node.color : BLACK;
53.     }
54.
55.     public boolean isRed(RBNode<T> node) { //判断节点的颜色
56.         return (node != null)&&(node.color == RED)? true : false;
57.     }
58.
59.     public boolean isBlack(RBNode<T> node) {
60.         return !isRed(node);
61.     }
62.
63.     public void setRed(RBNode<T> node) { //设置节点的颜色
64.         if(node != null)
65.             node.color = RED;
66.     }
67.
68.     public void setBlack(RBNode<T> node) {
69.         if(node != null) {
70.             node.color = BLACK;
71.         }
72.     }
73.
74.     public void setColor(RBNode<T> node, boolean color) {
75.         if(node != null)
76.             node.color = color;
77.     }
78.
79.     /***************** 前序遍历红黑树 ********************/
80.     public void preOrder() {
81.         preOrder(root);
82.     }
83.
84.     private void preOrder(RBNode<T> tree) {
85.         if(tree != null) {
86.             System.out.print(tree.key + " ");
87.             preOrder(tree.left);
88.             preOrder(tree.right);
89.         }
90.     }
91.
92.     /***************** 中序遍历红黑树 ********************/
93.     public void inOrder() {
94.         inOrder(root);
```

```
95.     }
96.
97.     private void inOrder(RBNode<T> tree) {
98.         if(tree != null) {
99.              preOrder(tree.left);
100.             System.out.print(tree.key + " ");
101.             preOrder(tree.right);
102.         }
103.    }
104.
105.    /***************** 后序遍历红黑树 ********************/
106.    public void postOrder() {
107.        postOrder(root);
108.    }
109.
110.    private void postOrder(RBNode<T> tree) {
111.        if(tree != null) {
112.             preOrder(tree.left);
113.             preOrder(tree.right);
114.             System.out.print(tree.key + " ");
115.         }
116.    }
117.
118.    /**************** 查找红黑树中键值为key的节点 ***************/
119.    public RBNode<T> search(T key) {
120.        return search(root, key);
121. //     return search2(root, key); //使用递归的方法，本质一样的
122.    }
123.
124.    private RBNode<T> search(RBNode<T> x, T key) {
125.        while(x != null) {
126.            int cmp = key.compareTo(x.key);
127.            if(cmp < 0)
128.                x = x.left;
129.            else if(cmp > 0)
130.                x = x.right;
131.            else
132.                return x;
133.        }
134.        return x;
135.    }
136.    //使用递归
137.    private RBNode<T> search2(RBNode<T> x, T key) {
```

```
138.         if(x == null)
139.             return x;
140.         int cmp = key.compareTo(x.key);
141.         if(cmp < 0)
142.             return search2(x.left, key);
143.         else if(cmp > 0)
144.             return search2(x.right, key);
145.         else
146.             return x;
147.     }
148.
149.     /**************** 查找最小节点的值  *********************/
150.     public T minValue() {
151.         RBNode<T> node = minNode(root);
152.         if(node != null)
153.             return node.key;
154.         return null;
155.     }
156.
157.     private RBNode<T> minNode(RBNode<T> tree) {
158.         if(tree == null)
159.             return null;
160.         while(tree.left != null) {
161.             tree = tree.left;
162.         }
163.         return tree;
164.     }
165.
166.     /******************* 查找最大节点的值 ******************/
167.     public T maxValue() {
168.         RBNode<T> node = maxNode(root);
169.         if(node != null)
170.             return node.key;
171.         return null;
172.     }
173.
174.     private RBNode<T> maxNode(RBNode<T> tree) {
175.         if(tree == null)
176.             return null;
177.         while(tree.right != null)
178.             tree = tree.right;
179.         return tree;
180.     }
```

```
181.
182.     /********* 查找节点x的后继节点,即大于节点x的最小节点 **********/
183.     public RBNode<T> successor(RBNode<T> x) {
184.         //如果x有右子节点，那么后继节点为“以右子节点为根的子树的最小节点”
185.         if(x.right != null)
186.             return minNode(x.right);
187.         //如果x没有右子节点，会出现以下两种情况：
188.         //1. x是其父节点的左子节点，则x的后继节点为它的父节点
189.         //2. x是其父节点的右子节点，则先查找x的父节点p，然后对p再次进行这两个条件的判断
190.         RBNode<T> p = x.parent;
191.         while((p != null) && (x == p.right)) { //对应情况2
192.             x = p;
193.             p = x.parent;
194.         }
195.         return p; //对应情况1
196.
197.     }
198.
199.     /********* 查找节点x的前驱节点，即小于节点x的最大节点 ************/
200.     public RBNode<T> predecessor(RBNode<T> x) {
201.         //如果x有左子节点，那么前驱结点为“左子节点为根的子树的最大节点”
202.         if(x.left != null)
203.             return maxNode(x.left);
204.         //如果x没有左子节点，会出现以下两种情况：
205.         //1. x是其父节点的右子节点，则x的前驱节点是它的父节点
206.         //2. x是其父节点的左子节点，则先查找x的父节点p，然后对p再次进行这两个条件的判断
207.         RBNode<T> p = x.parent;
208.         while((p != null) && (x == p.left)) { //对应情况2
209.             x = p;
210.             p = x.parent;
211.         }
212.         return p; //对应情况1
213.     }
214.
215.     /*************对红黑树节点x进行左旋操作 *****************/
216.     /*
217.      * 左旋示意图：对节点x进行左旋
218.      *     p                       p
219.      *    /                       /
220.      *   x                       y
221.      *  / \                     / \
222.      * lx  y      ----->       x  ry
223.      *    / \                 / \
```

```
224.      *   ly ry              lx ly
225.      * 左旋做了三件事：
226.      * 1. 将y的左子节点赋给x的右子节点,并将x赋给y左子节点的父节点(y左子节点非空时)
227.      * 2. 将x的父节点p(非空时)赋给y的父节点，同时更新p的子节点为y(左或右)
228.      * 3. 将y的左子节点设为x，将x的父节点设为y
229.      */
230.     private void leftRotate(RBNode<T> x) {
231.         //1. 将y的左子节点赋给x的右子节点，并将x赋给y左子节点的父节点(y左子节点非空时)
232.         RBNode<T> y = x.right;
233.         x.right = y.left;
234.
235.         if(y.left != null)
236.             y.left.parent = x;
237.
238.         //2. 将x的父节点p(非空时)赋给y的父节点，同时更新p的子节点为y(左或右)
239.         y.parent = x.parent;
240.
241.         if(x.parent == null) {
242.             this.root = y; //如果x的父节点为空，则将y设为父节点
243.         } else {
244.             if(x == x.parent.left) //如果x是左子节点
245.                 x.parent.left = y; //则也将y设为左子节点
246.             else
247.                 x.parent.right = y;//否则将y设为右子节点
248.         }
249.
250.         //3. 将y的左子节点设为x，将x的父节点设为y
251.         y.left = x;
252.         x.parent = y;
253.     }
254.
255.     /*************对红黑树节点y进行右旋操作 ******************/
256.     /*
257.      * 左旋示意图：对节点y进行右旋
258.      *        p                   p
259.      *       /                   /
260.      *      y                   x
261.      *     / \                 / \
262.      *    x  ry   ----->      lx  y
263.      *   / \                     / \
264.      * lx  rx                   rx ry
265.      * 右旋做了三件事：
266.      * 1. 将x的右子节点赋给y的左子节点,并将y赋给x右子节点的父节点(x右子节点非空时)
```

```
267.      * 2. 将y的父节点p(非空时)赋给x的父节点，同时更新p的子节点为x(左或右)
268.      * 3. 将x的右子节点设为y，将y的父节点设为x
269.      */
270.     private void rightRotate(RBNode<T> y) {
271.         //1. 将y的左子节点赋给x的右子节点，并将x赋给y左子节点的父节点(y左子节点非空时)
272.         RBNode<T> x = y.left;
273.         y.left = x.right;
274.
275.         if(x.right != null)
276.             x.right.parent = y;
277.
278.         //2. 将x的父节点p(非空时)赋给y的父节点，同时更新p的子节点为y(左或右)
279.         x.parent = y.parent;
280.
281.         if(y.parent == null) {
282.             this.root = x; //如果x的父节点为空，则将y设为父节点
283.         } else {
284.             if(y == y.parent.right) //如果x是左子节点
285.                 y.parent.right = x; //则也将y设为左子节点
286.             else
287.                 y.parent.left = x;//否则将y设为右子节点
288.         }
289.
290.         //3. 将y的左子节点设为x，将x的父节点设为y
291.         x.right = y;
292.         y.parent = x;
293.     }
294.
295.     /********************** 向红黑树中插入节点 **********************/
296.     public void insert(T key) {
297.         RBNode<T> node = new RBNode<T>(key, RED, null, null, null);
298.         if(node != null)
299.             insert(node);
300.     }
301.
302.     //将节点插入到红黑树中，这个过程与二叉搜索树是一样的
303.     private void insert(RBNode<T> node) {
304.         RBNode<T> current = null; //表示最后node的父节点
305.         RBNode<T> x = this.root; //用来向下搜索用的
306.
307.         //1. 找到插入的位置
308.         while(x != null) {
309.             current = x;
```

```
            int cmp = node.key.compareTo(x.key);
            if(cmp < 0)
                x = x.left;
            else
                x = x.right;
        }
        node.parent = current; //找到了位置，将当前current作为node的父节点

        //2. 接下来判断node是插在左子节点还是右子节点
        if(current != null) {
            int cmp = node.key.compareTo(current.key);
            if(cmp < 0)
                current.left = node;
            else
                current.right = node;
        } else {
            this.root = node;
        }

        //3. 将它重新修整为一颗红黑树
        insertFixUp(node);
    }

    private void insertFixUp(RBNode<T> node) {
        RBNode<T> parent, gparent; //定义父节点和祖父节点

        //需要修整的条件：父节点存在，且父节点的颜色是红色
        while(((parent = parentOf(node)) != null) && isRed(parent)) {
            gparent = parentOf(parent);//获得祖父节点

            //若父节点是祖父节点的左子节点，下面else与其相反
            if(parent == gparent.left) {
                RBNode<T> uncle = gparent.right; //获得叔叔节点

                //case1: 叔叔节点也是红色
                if(uncle != null && isRed(uncle)) {
                    setBlack(parent); //把父节点和叔叔节点涂黑
                    setBlack(uncle);
                    setRed(gparent); //把祖父节点涂红
                    node = gparent; //将位置放到祖父节点处
                    continue; //继续while，重新判断
                }

```

```
353.                 //case2: 叔叔节点是黑色，且当前节点是右子节点
354.                 if(node == parent.right) {
355.                     leftRotate(parent); //从父节点处左旋
356.                     RBNode<T> tmp = parent; //然后将父节点和自己调换一下，为下面右旋做准备
357.                     parent = node;
358.                     node = tmp;
359.                 }
360.
361.                 //case3: 叔叔节点是黑色，且当前节点是左子节点
362.                 setBlack(parent);
363.                 setRed(gparent);
364.                 rightRotate(gparent);
365.             } else { //若父节点是祖父节点的右子节点,与上面的完全相反，本质一样的
366.                 RBNode<T> uncle = gparent.left;
367.
368.                 //case1: 叔叔节点也是红色
369.                 if(uncle != null & isRed(uncle)) {
370.                     setBlack(parent);
371.                     setBlack(uncle);
372.                     setRed(gparent);
373.                     node = gparent;
374.                     continue;
375.                 }
376.
377.                 //case2: 叔叔节点是黑色的，且当前节点是左子节点
378.                 if(node == parent.left) {
379.                     rightRotate(parent);
380.                     RBNode<T> tmp = parent;
381.                     parent = node;
382.                     node = tmp;
383.                 }
384.
385.                 //case3: 叔叔节点是黑色的，且当前节点是右子节点
386.                 setBlack(parent);
387.                 setRed(gparent);
388.                 leftRotate(gparent);
389.             }
390.         }
391.
392.         //将根节点设置为黑色
393.         setBlack(this.root);
394.     }
395.
```

```
396.     /********************** 删除红黑树中的节点 **********************/
397.     public void remove(T key) {
398.         RBNode<T> node;
399.         if((node = search(root, key)) != null)
400.             remove(node);
401.     }
402. 
403.     private void remove(RBNode<T> node) {
404.         RBNode<T> child, parent;
405.         boolean color;
406. 
407.         //1. 被删除的节点“左右子节点都不为空”的情况
408.         if((node.left != null) && (node.right != null)) {
409.             //先找到被删除节点的后继节点，用它来取代被删除节点的位置
410.             RBNode<T> replace = node;
411.             //  1). 获取后继节点
412.             replace = replace.right;
413.             while(replace.left != null)
414.                 replace = replace.left;
415. 
416.             //  2). 处理“后继节点”和“被删除节点的父节点”之间的关系
417.             if(parentOf(node) != null) { //要删除的节点不是根节点
418.                 if(node == parentOf(node).left)
419.                     parentOf(node).left = replace;
420.                 else
421.                     parentOf(node).right = replace;
422.             } else { //否则
423.                 this.root = replace;
424.             }
425. 
426.             //  3). 处理“后继节点的子节点”和“被删除节点的子节点”之间的关系
427.             child = replace.right; //后继节点肯定不存在左子节点!
428.             parent = parentOf(replace);
429.             color = colorOf(replace);//保存后继节点的颜色
430.             if(parent == node) { //后继节点是被删除节点的子节点
431.                 parent = replace;
432.             } else { //否则
433.                 if(child != null)
434.                     setParent(child, parent);
435.                 parent.left = child;
436.                 replace.right = node.right;
437.                 setParent(node.right, replace);
438.             }
```

```
439.             replace.parent = node.parent;
440.             replace.color = node.color; //保持原来位置的颜色
441.             replace.left = node.left;
442.             node.left.parent = replace;
443.
444.             if(color == BLACK) { //4. 如果移走的后继节点颜色是黑色，重新修整红黑树
445.                 removeFixUp(child, parent);//将后继节点的child和parent传进去
446.             }
447.             node = null;
448.             return;
449.         }
450.     }
451.     //node表示待修正的节点，即后继节点的子节点（因为后继节点被挪到删除节点的位置去了）
452.     private void removeFixUp(RBNode<T> node, RBNode<T> parent) {
453.         RBNode<T> other;
454.
455.         while((node == null || isBlack(node)) && (node != this.root)) {
456.             if(parent.left == node) { //node是左子节点，下面else与这里的刚好相反
457.                 other = parent.right; //node的兄弟节点
458.                 if(isRed(other)) { //case1: node的兄弟节点other是红色的
459.                     setBlack(other);
460.                     setRed(parent);
461.                     leftRotate(parent);
462.                     other = parent.right;
463.                 }
464.
465.                 //case2: node的兄弟节点other是黑色的，且other的两个子节点也都是黑色的
466.                 if((other.left == null || isBlack(other.left)) &&
467.                         (other.right == null || isBlack(other.right))) {
468.                     setRed(other);
469.                     node = parent;
470.                     parent = parentOf(node);
471.                 } else {
472.                     //case3: node的兄弟节点other是黑色的，且other的左子节点是红色，右子节点是黑色
473.                     if(other.right == null || isBlack(other.right)) {
474.                         setBlack(other.left);
475.                         setRed(other);
476.                         rightRotate(other);
477.                         other = parent.right;
478.                     }
479.
480.                     //case4: node的兄弟节点other是黑色的，且other的右子节点是红色，左子节点任意颜色
481.                     setColor(other, colorOf(parent));
```

```
482.                     setBlack(parent);
483.                     setBlack(other.right);
484.                     leftRotate(parent);
485.                     node = this.root;
486.                     break;
487.                 }
488.             } else { //与上面的对称
489.                 other = parent.left;
490. 
491.                 if (isRed(other)) {
492.                     // Case 1: node的兄弟other是红色的
493.                     setBlack(other);
494.                     setRed(parent);
495.                     rightRotate(parent);
496.                     other = parent.left;
497.                 }
498. 
499.                 if ((other.left==null || isBlack(other.left)) &&
500.                     (other.right==null || isBlack(other.right))) {
501.                     // Case 2: node的兄弟other是黑色，且other的俩个子节点都是黑色的
502.                     setRed(other);
503.                     node = parent;
504.                     parent = parentOf(node);
505.                 } else {
506. 
507.                     if (other.left==null || isBlack(other.left)) {
508.                         // Case 3: node的兄弟other是黑色的，并且other的左子节点是红色，右子节点为黑色。
509.                         setBlack(other.right);
510.                         setRed(other);
511.                         leftRotate(other);
512.                         other = parent.left;
513.                     }
514. 
515.                     // Case 4: node的兄弟other是黑色的；并且other的左子节点是红色的，右子节点任意颜色
516.                     setColor(other, colorOf(parent));
517.                     setBlack(parent);
518.                     setBlack(other.left);
519.                     rightRotate(parent);
520.                     node = this.root;
521.                     break;
522.                 }
523.             }
524.         }
```

```
525.         if (node!=null)
526.             setBlack(node);
527.     }
528. 
529.     /****************** 销毁红黑树 *********************/
530.     public void clear() {
531.         destroy(root);
532.         root = null;
533.     }
534. 
535.     private void destroy(RBNode<T> tree) {
536.         if(tree == null)
537.             return;
538.         if(tree.left != null)
539.             destroy(tree.left);
540.         if(tree.right != null)
541.             destroy(tree.right);
542.         tree = null;
543.     }
544. 
545.     /******************* 打印红黑树 *********************/
546.     public void print() {
547.         if(root != null) {
548.             print(root, root.key, 0);
549.         }
550.     }
551.     /*
552.      * key---节点的键值
553.      * direction--- 0:表示该节点是根节点
554.      *              1:表示该节点是它的父节点的左子节点
555.      *              2:表示该节点是它的父节点的右子节点
556.      */
557.     private void print(RBNode<T> tree, T key, int direction) {
558.         if(tree != null) {
559.             if(0 == direction)
560.                 System.out.printf("%2d(B) is root\n", tree.key);
561.             else
562.                 System.out.printf("%2d(%s) is %2d's %6s child\n",
563.                         tree.key, isRed(tree)?"R":"b", key, direction == 1?"right":"left");
564.             print(tree.left, tree.key, -1);
565.             print(tree.right, tree.key, 1);
566.         }
567.     }
```

```
568. }
```

下面附上测试程序吧：

[java]


```
1.  package test;
2.
3.  import tree.RBTree;
4.
5.  public class RBTreeTest {
6.
7.      private static final int a[] = {10, 40, 30, 60, 90, 70, 20, 50, 80};
8.      private static final boolean mDebugInsert = true;    // "插入"动作的检测开关(false，关闭；true，打开)
9.          private static final boolean mDebugDelete = true;    // "删除"动作的检测开关(false，关闭；true，打开)
10.
11.     public static void main(String[] args) {
12.         int i, ilen = a.length;
13.                 RBTree<Integer> tree = new RBTree<Integer>();
14.
15.                 System.out.printf("== 原始数据: ");
16.                 for(i=0; i<ilen; i++)
17.                     System.out.printf("%d ", a[i]);
18.                 System.out.printf("\n");
19.
20.                 for(i=0; i<ilen; i++) {
21.                    tree.insert(a[i]);
22.                     // 设置mDebugInsert=true,测试"添加函数"
23.                     if (mDebugInsert) {
24.                         System.out.printf("== 添加节点: %d\n", a[i]);
25.                         System.out.printf("== 树的详细信息: \n");
26.                         tree.print();
27.                         System.out.printf("\n");
28.                     }
29.                 }
30.
31.                 System.out.printf("== 前序遍历: ");
32.                 tree.preOrder();
33.
34.                 System.out.printf("\n== 中序遍历: ");
35.                 tree.inOrder();
36.
37.                 System.out.printf("\n== 后序遍历: ");
38.                 tree.postOrder();
```

```
39.                 System.out.printf("\n");
40.
41.                 System.out.printf("== 最小值: %s\n", tree.minValue());
42.                 System.out.printf("== 最大值: %s\n", tree.maxValue());
43.                 System.out.printf("== 树的详细信息: \n");
44.                 tree.print();
45.                 System.out.printf("\n");
46.
47.                 // 设置mDebugDelete=true,测试"删除函数"
48.                 if (mDebugDelete) {
49.                     for(i=0; i<ilen; i++)
50.                     {
51.                         tree.remove(a[i]);
52.
53.                         System.out.printf("== 删除节点: %d\n", a[i]);
54.                         System.out.printf("== 树的详细信息: \n");
55.                         tree.print();
56.                         System.out.printf("\n");
57.                     }
58.                 }
59.         }
60.
61. }
```

5.红-黑树的复杂度

前面也说了，当数据以升序或降序插入时，二叉搜索树的性能就会下降到最低，但是红-黑树的自我修复功能保证了即使在最坏的情况下，也能保证时间复杂度在O(logN)的级别上。

来源： https://blog.csdn.net/eson_15/article/details/51144079