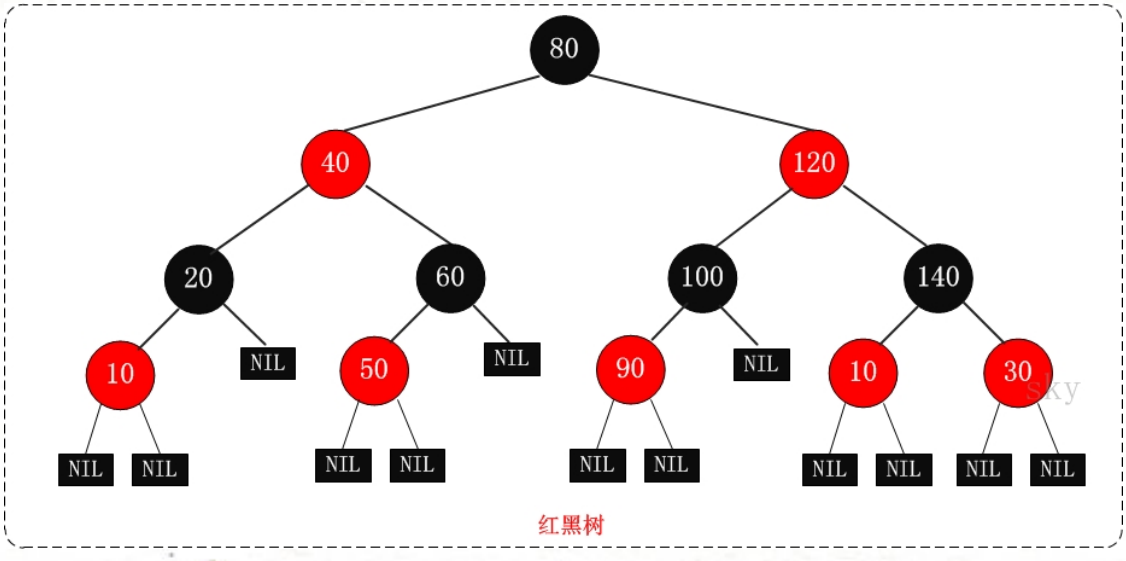
http://www.cnblogs.com/skywang12345/p/3624343.html

红黑树的介绍

红黑树(Red-Black Tree，简称R-B Tree)，它一种特殊的二叉查找树。
红黑树是特殊的二叉查找树，意味着它满足二叉查找树的特征：任意一个节点所包含的键值，大于等于左孩子的键值，小于等于右孩子的键值。
除了具备该特性之外，红黑树还包括许多额外的信息。
红黑树的每个节点上都有存储位表示节点的颜色，颜色是红(Red)或黑(Black)。
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) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]
(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(NIL)的所有路径上包含相同数目的黑节点。
关于它的特性，需要注意的是:
第一，特性(3)中的叶子节点，是只为空(NIL或null)的节点。
第二，特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。
红黑树示意图如下:



红黑树的Java实现(代码说明)

红黑树的基本操作是**添加**、**删除**和**旋转**。在对红黑树进行添加或删除后，会用到旋转

方法。为什么呢？道理很简单，添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一颗红黑树了，而是一颗普通的树。而通过旋转，可以使这颗树重新成为红黑树。简单点说，旋转的目的是让树保持红黑树的特性。

旋转包括两种：**左旋** 和 **右旋**。下面分别对红黑树的基本操作进行介绍。

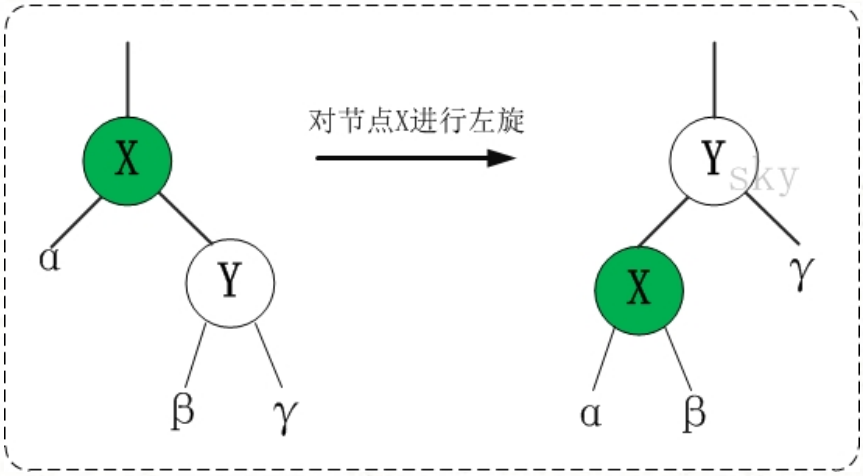
1. 基本定义

```
public class RBTree<T extends Comparable<T>> {

    private RBTNode<T> mRoot;    // 根结点

    private static final boolean RED   = false;
    private static final boolean BLACK = true;

    public class RBTNode<T extends Comparable<T>> {
        boolean color;         // 颜色
        T key;                 // 关键字(键值)
        RBTNode<T> left;    // 左孩子
        RBTNode<T> right;    // 右孩子
        RBTNode<T> parent;    // 父结点

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

    }

    ...
}
```

RBTree是红黑树对应的类，RBTNode是红黑树的节点类。在RBTree中包含了根节点mRoot和红黑树的相关API。

注意：在实现红黑树API的过程中，我重载了许多函数。重载的原因，一是因为有的API是内部接口，有的是外部接口；二是为了让结构更加清晰。

2. 左旋



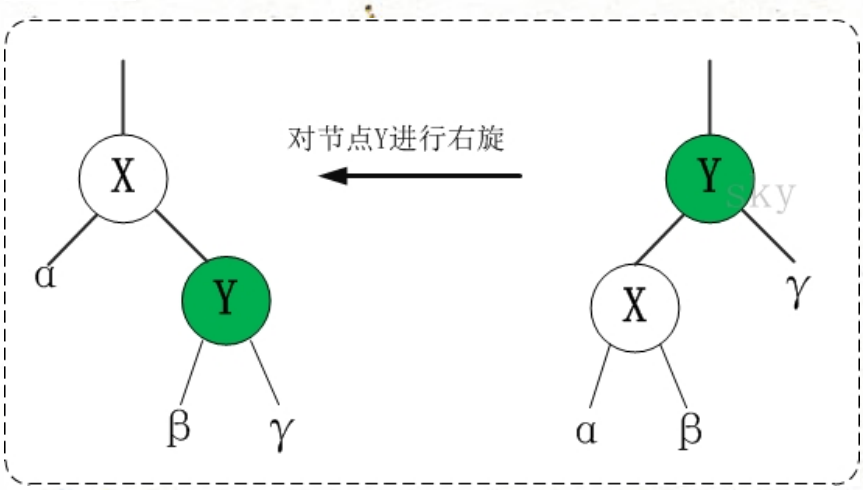
对x进行左旋，意味着"将x变成一个左节点"。

左旋的实现代码(Java语言)

```
/*
 * 对红黑树的节点(x)进行左旋转
 *
 * 左旋示意图(对节点x进行左旋):
 *      px                              px
 *     /                               /
 *    x                               y
 *   /  \      --(左旋)-.            / \                #
 *  lx   y                            x  ry
 *     /   \                         /  \
 *    ly   ry                       lx  ly
 *
 *
 */
private void leftRotate(RBTNode<T> x) {
    // 设置x的右孩子为y
    RBTNode<T> y = x.right;

    // 将 “y的左孩子” 设为 “x的右孩子”;
    // 如果y的左孩子非空，将 “x” 设为 “y的左孩子的父亲”
    x.right = y.left;
    if (y.left != null)
        y.left.parent = x;

    // 将 “x的父亲” 设为 “y的父亲”
    y.parent = x.parent;
```

```
    if (x.parent == null) {
        this.mRoot = y;             // 如果 "x的父亲" 是空节点，则将y设为根节点
    } else {
        if (x.parent.left == x)
            x.parent.left = y;      // 如果 x是它父节点的左孩子，则将y设为"x的父节点的左孩子"
        else
            x.parent.right = y;     // 如果 x是它父节点的左孩子，则将y设为"x的父节点的左孩子"
    }

    // 将 "x" 设为 "y的左孩子"
    y.left = x;
    // 将 "x的父节点" 设为 "y"
    x.parent = y;
}
```

3. 右旋



对y进行右旋，意味着"将y变成一个右节点"。

右旋的实现代码(Java语言)

```
/*
 * 对红黑树的节点(y)进行右旋转
 *
 * 右旋示意图(对节点y进行左旋)：
 *            py                               py
 *           /                                 /
```

```
 *           y                              x
 *          /  \       --(右旋)-.          /  \                          #
 *         x    ry                        lx   y
 *        / \                                / \                         #
 *       lx  rx                             rx  ry
 *
 */
private void rightRotate(RBTNode<T> y) {
    // 设置x是当前节点的左孩子。
    RBTNode<T> x = y.left;

    // 将 “x的右孩子” 设为 “y的左孩子”;
    // 如果"x的右孩子"不为空的话，将 “y” 设为 “x的右孩子的父亲”
    y.left = x.right;
    if (x.right != null)
        x.right.parent = y;

    // 将 “y的父亲” 设为 “x的父亲”
    x.parent = y.parent;

    if (y.parent == null) {
        this.mRoot = x;             // 如果 “y的父亲” 是空节点，则将x设为根节点
    } else {
        if (y == y.parent.right)
            y.parent.right = x;     // 如果 y是它父节点的右孩子，则将x设为“y的父节点的右孩子”
        else
            y.parent.left = x;      // (y是它父节点的左孩子) 将x设为“x的父节点的左孩子”
    }

    // 将 “y” 设为 “x的右孩子”
    x.right = y;

    // 将 “y的父节点” 设为 “x”
    y.parent = x;
}
```

4. 添加

将一个节点插入到红黑树中，需要执行哪些步骤呢？首先，将红黑树当作一颗二叉查找树，将节点插入；然后，将节点着色为红色；最后，通过"旋转和重新着色"等一系列操作来修正该树，使之重新成为一颗红黑树。详细描述如下：

第一步：将红黑树当作一颗二叉查找树，将节点插入。

红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。此外，无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。

好吧？那接下来，我们就来想方设法的旋转以及重新着色，使这颗树重新成为红黑树！

第二步：将插入的节点着色为"红色"。

为什么着色成红色，而不是黑色呢？为什么呢？在回答之前，我们需要重新温习一下红黑树的特性：

(1) 每个节点或者是黑色，或者是红色。

(2) 根节点是黑色。

(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]

(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。

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。接下来，就要努力的让这棵树满足其它性质即可；满足了的话，它就又是一颗红黑树了。o(∩∩)o...哈哈

第三步：通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。

第二步中，将插入节点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢？

对于"特性(1)"，显然不会违背了。因为我们已经将它涂成红色了。

对于"特性(2)"，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉查找数的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。

对于"特性(3)"，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。

对于"特性(4)"，是有可能违背的！

那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

添加操作的实现代码(Java语言)

```
/*
 * 将结点插入到红黑树中
 *
 * 参数说明：
 *     node 插入的结点        // 对应《算法导论》中的node
 */
private void insert(RBTNode<T> node) {
    int cmp;
    RBTNode<T> y = null;
```

```
    RBTNode<T> x = this.mRoot;

    // 1. 将红黑树当作一颗二叉查找树，将节点添加到二叉查找树中。
    while (x != null) {
        y = x;
        cmp = node.key.compareTo(x.key);
        if (cmp < 0)
            x = x.left;
        else
            x = x.right;
    }

    node.parent = y;
    if (y!=null) {
        cmp = node.key.compareTo(y.key);
        if (cmp < 0)
            y.left = node;
        else
            y.right = node;
    } else {
        this.mRoot = node;
    }

    // 2. 设置节点的颜色为红色
    node.color = RED;

    // 3. 将它重新修正为一颗二叉查找树
    insertFixUp(node);
}

/*
 * 新建结点(key)，并将其插入到红黑树中
 *
 * 参数说明：
 *     key 插入结点的键值
 */
public void insert(T key) {
    RBTNode<T> node=new RBTNode<T>(key,BLACK,null,null,null);

    // 如果新建结点失败，则返回。
    if (node != null)
        insert(node);
}
```

内部接口 -- insert(node)的作用是将"node"节点插入到红黑树中。

外部接口 -- insert(key)的作用是将"key"添加到红黑树中。

添加修正操作的实现代码(Java语言)

```
/*
 * 红黑树插入修正函数
 *
 * 在向红黑树中插入节点之后(失去平衡)，再调用该函数；
 * 目的是将它重新塑造成一颗红黑树。
 *
 * 参数说明：
 *     node 插入的结点        // 对应《算法导论》中的z
 */
private void insertFixUp(RBTNode<T> node) {
    RBTNode<T> parent, gparent;

    // 若"父节点存在，并且父节点的颜色是红色"
    while (((parent = parentOf(node))!=null) && isRed(parent)) {
        gparent = parentOf(parent);

        //若"父节点"是"祖父节点的左孩子"
        if (parent == gparent.left) {
            // Case 1条件：叔叔节点是红色
            RBTNode<T> uncle = gparent.right;
            if ((uncle!=null) && isRed(uncle)) {
                setBlack(uncle);
                setBlack(parent);
                setRed(gparent);
                node = gparent;
                continue;
            }

            // Case 2条件：叔叔是黑色，且当前节点是右孩子
            if (parent.right == node) {
                RBTNode<T> tmp;
                leftRotate(parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }

            // Case 3条件：叔叔是黑色，且当前节点是左孩子。
            setBlack(parent);
            setRed(gparent);
```

```
                rightRotate(gparent);
            } else {    //若“z的父节点”是“z的祖父节点的右孩子”
                // Case 1条件：叔叔节点是红色
                RBTNode<T> uncle = gparent.left;
                if ((uncle!=null) && isRed(uncle)) {
                    setBlack(uncle);
                    setBlack(parent);
                    setRed(gparent);
                    node = gparent;
                    continue;
                }

                // Case 2条件：叔叔是黑色，且当前节点是左孩子
                if (parent.left == node) {
                    RBTNode<T> tmp;
                    rightRotate(parent);
                    tmp = parent;
                    parent = node;
                    node = tmp;
                }

                // Case 3条件：叔叔是黑色，且当前节点是右孩子。
                setBlack(parent);
                setRed(gparent);
                leftRotate(gparent);
            }
        }

        // 将根节点设为黑色
        setBlack(this.mRoot);
    }
```

insertFixUp(node)的作用是对应"上面所讲的第三步"。它是一个内部接口。

5. 删除操作

将红黑树内的某一个节点删除。需要执行的操作依次是：首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。详细描述如下：

第一步：将红黑树当作一颗二叉查找树，将节点删除。

这和"删除常规二叉查找树中删除节点的方法是一样的"。分3种情况：

① 被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就OK了。

② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

③ 被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给“该节点的内容”；之后，删除“它的后继节点”。在这里，后继节点相当于替身，在将后继节点的内容复制给"被删除节点"之后，再将后继节点删除。这样就巧妙的将问题转换为"删除后继节点"的情况了，下面就考虑后继节点。 在"被删除节点"有两个非空子节点的情况下，它的后继节点不可能是双子非空。既然"的后继节点"不可能双子都非空，就意味着"该节点的后继节点"要么没有儿子，要么只有一个儿子。若没有儿子，则按"情况① "进行处理；若只有一个儿子，则按"情况② "进行处理。

第二步：通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

因为"第一步"中删除节点之后，可能会违背红黑树的特性。所以需要通过"旋转和重新着色"来修正该树，使之重新成为一棵红黑树。

删除操作的实现代码(Java语言)

```
/*
 * 删除结点(node)，并返回被删除的结点
 *
 * 参数说明：
 *     node 删除的结点
 */
private void remove(RBTNode<T> node) {
    RBTNode<T> child, parent;
    boolean color;

    // 被删除节点的"左右孩子都不为空"的情况。
    if ( (node.left!=null) && (node.right!=null) ) {
        // 被删节点的后继节点。(称为"取代节点")
        // 用它来取代"被删节点"的位置，然后再将"被删节点"去掉。
        RBTNode<T> replace = node;

        // 获取后继节点
        replace = replace.right;
        while (replace.left != null)
            replace = replace.left;

        // "node节点"不是根节点(只有根节点不存在父节点)
        if (parentOf(node)!=null) {
            if (parentOf(node).left == node)
                parentOf(node).left = replace;
            else
                parentOf(node).right = replace;
        } else {
            // "node节点"是根节点，更新根节点。
            this.mRoot = replace;
```

```
        }

        // child是"取代节点"的右孩子，也是需要"调整的节点"。
        // "取代节点"肯定不存在左孩子！因为它是一个后继节点。
        child = replace.right;
        parent = parentOf(replace);
        // 保存"取代节点"的颜色
        color = colorOf(replace);

        // "被删除节点"是"它的后继节点的父节点"
        if (parent == node) {
            parent = replace;
        } else {
            // child不为空
            if (child!=null)
                setParent(child, parent);
            parent.left = child;

            replace.right = node.right;
            setParent(node.right, replace);
        }

        replace.parent = node.parent;
        replace.color = node.color;
        replace.left = node.left;
        node.left.parent = replace;

        if (color == BLACK)
            removeFixUp(child, parent);

        node = null;
        return ;
    }

    if (node.left !=null) {
        child = node.left;
    } else {
        child = node.right;
    }

    parent = node.parent;
    // 保存"取代节点"的颜色
    color = node.color;

    if (child!=null)
```

```
        child.parent = parent;

    // "node节点"不是根节点
    if (parent!=null) {
        if (parent.left == node)
            parent.left = child;
        else
            parent.right = child;
    } else {
        this.mRoot = child;
    }

    if (color == BLACK)
        removeFixUp(child, parent);
    node = null;
}

/*
 * 删除结点(z)，并返回被删除的结点
 *
 * 参数说明：
 *     tree 红黑树的根结点
 *     z 删除的结点
 */
public void remove(T key) {
    RBTNode<T> node;

    if ((node = search(mRoot, key)) != null)
        remove(node);
}
```

内部接口 -- remove(node)的作用是将"node"节点插入到红黑树中。
外部接口 -- remove(key)删除红黑树中键值为key的节点。

删除修正操作的实现代码(Java语言)

```
/*
 * 红黑树删除修正函数
 *
 * 在从红黑树中删除插入节点之后(红黑树失去平衡)，再调用该函数；
 * 目的是将它重新塑造成一颗红黑树。
 *
 * 参数说明：
```

```
     *     node 待修正的节点
     */
    private void removeFixUp(RBTNode<T> node, RBTNode<T> parent) {
        RBTNode<T> other;

        while ((node==null || isBlack(node)) && (node != this.mRoot)) {
            if (parent.left == node) {
                other = parent.right;
                if (isRed(other)) {
                    // Case 1: x的兄弟w是红色的
                    setBlack(other);
                    setRed(parent);
                    leftRotate(parent);
                    other = parent.right;
                }

                if ((other.left==null || isBlack(other.left)) &&
                    (other.right==null || isBlack(other.right))) {
                    // Case 2: x的兄弟w是黑色，且w的俩个孩子也都是黑色的
                    setRed(other);
                    node = parent;
                    parent = parentOf(node);
                } else {

                    if (other.right==null || isBlack(other.right)) {
                        // Case 3: x的兄弟w是黑色的，并且w的左孩子是红色，右孩子为黑
色。
                        setBlack(other.left);
                        setRed(other);
                        rightRotate(other);
                        other = parent.right;
                    }
                    // Case 4: x的兄弟w是黑色的；并且w的右孩子是红色的，左孩子任意颜
色。
                    setColor(other, colorOf(parent));
                    setBlack(parent);
                    setBlack(other.right);
                    leftRotate(parent);
                    node = this.mRoot;
                    break;
                }
            } else {

                other = parent.left;
                if (isRed(other)) {
```

```
                // Case 1: x的兄弟w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: x的兄弟w是黑色，且w的俩个孩子也都是黑色的
                setRed(other);
                node = parent;
                parent = parentOf(node);
            } else {

                if (other.left==null || isBlack(other.left)) {
                    // Case 3: x的兄弟w是黑色的，并且w的左孩子是红色，右孩子为黑色。
                    setBlack(other.right);
                    setRed(other);
                    leftRotate(other);
                    other = parent.left;
                }

                // Case 4: x的兄弟w是黑色的；并且w的右孩子是红色的，左孩子任意颜色。
                setColor(other, colorOf(parent));
                setBlack(parent);
                setBlack(other.left);
                rightRotate(parent);
                node = this.mRoot;
                break;
            }
        }
    }

    if (node!=null)
        setBlack(node);
}
```

removeFixup(node, parent)是对应"上面所讲的第三步"。它是一个内部接口。