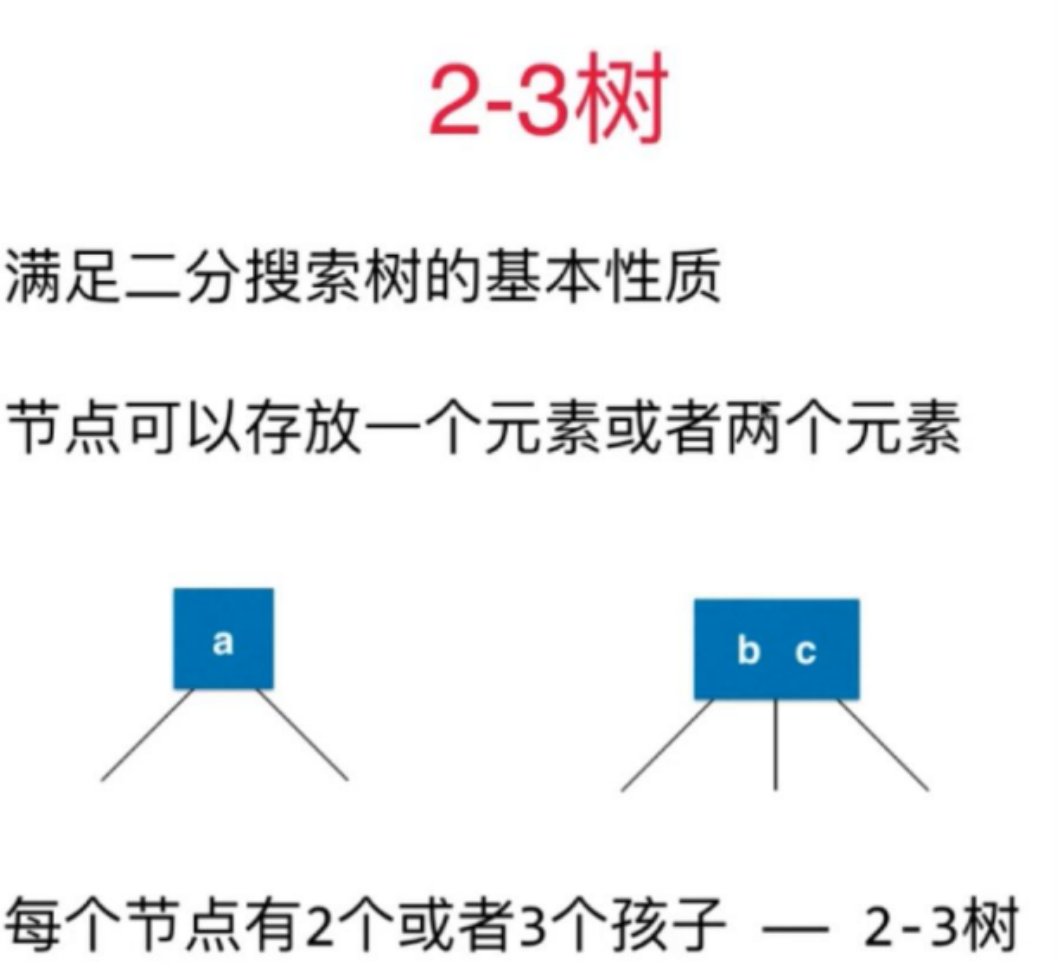
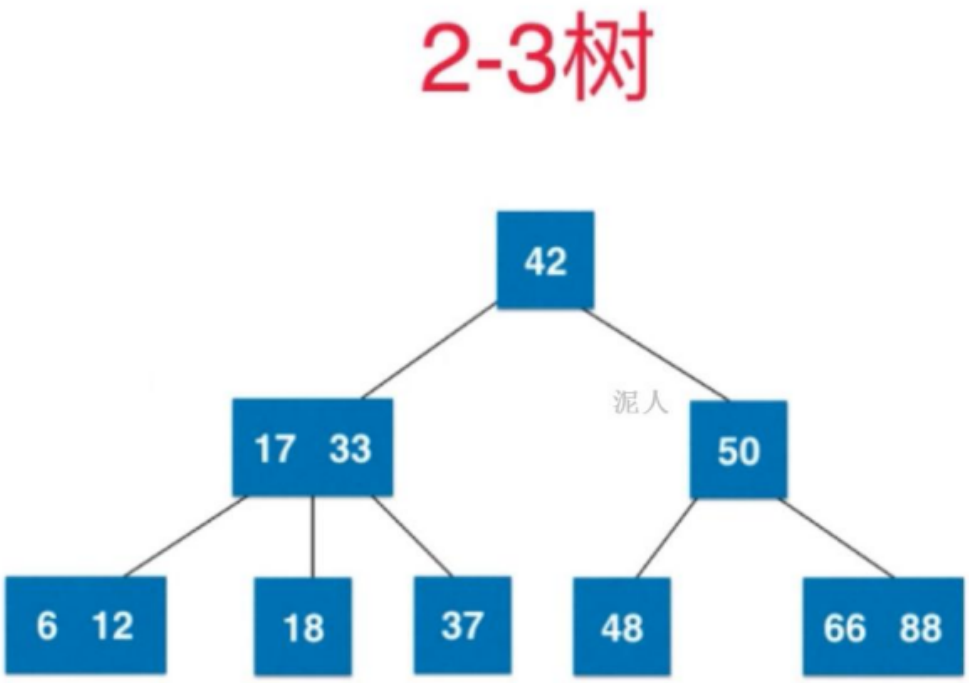
红黑树（高级数据结构，了解）

2-3树

- 2-3树与红黑树具有等价性，了解2-3树的性质可以帮助我们更好的掌握红黑树



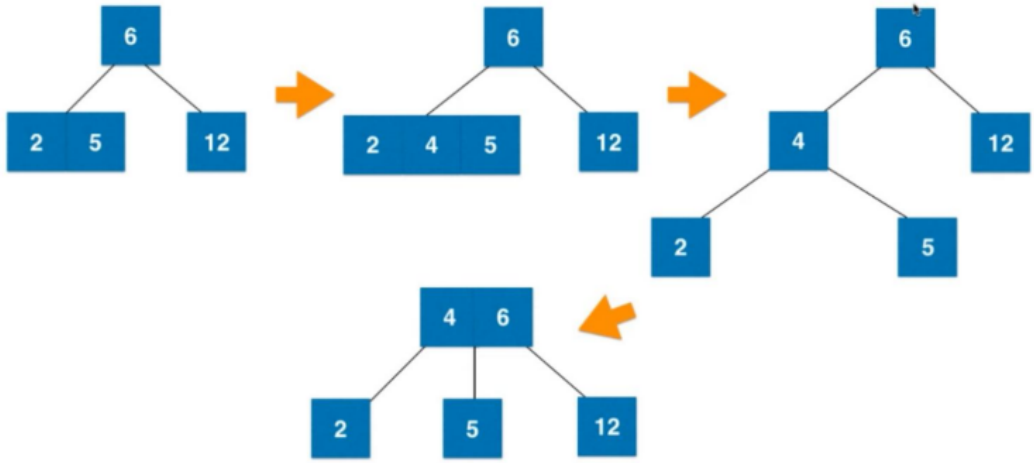
对于由两个孩子的节点我们称之为**2节点**，对于有三个孩子的节点我们则称之为**3节点**



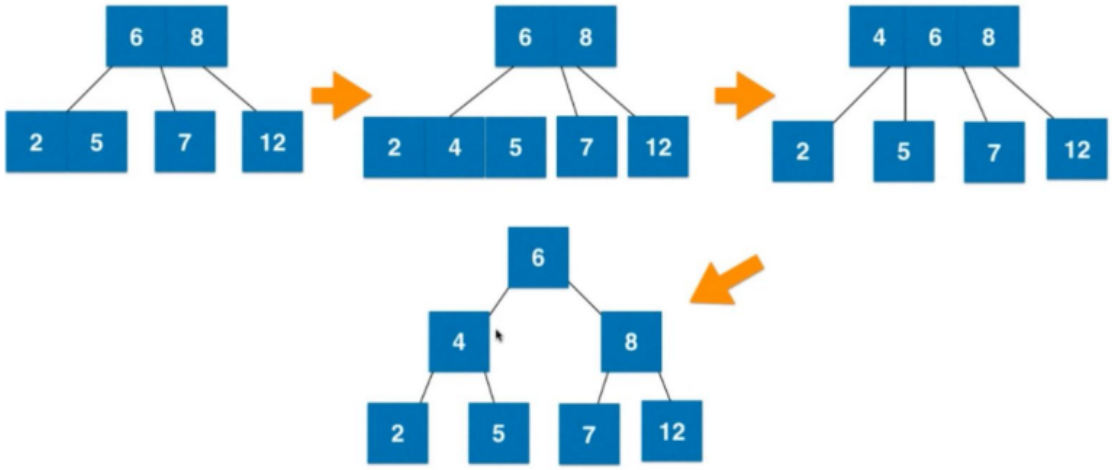
2-3树是一颗**绝对平衡的树**，即对于任何一个节点，左右子树的高度相同

- 2-3树的插入是一个比较麻烦的过程
  - 插入的节点是**2节点**则此时节点将转换成为一个**3节点**
  - 当插入的节点是**3节点**时我们需要将3节点向上分裂转换成2节点，如果分裂的3节点是一个子节点，则分裂后的根节点需要向上融合到父节点中，此时父节点就有可能会变成2节点或者3节点，因此我们需要重复这个操作，直到所有的节点都满足2-3树的要求。

如果插入3-节点，父亲为2-节点

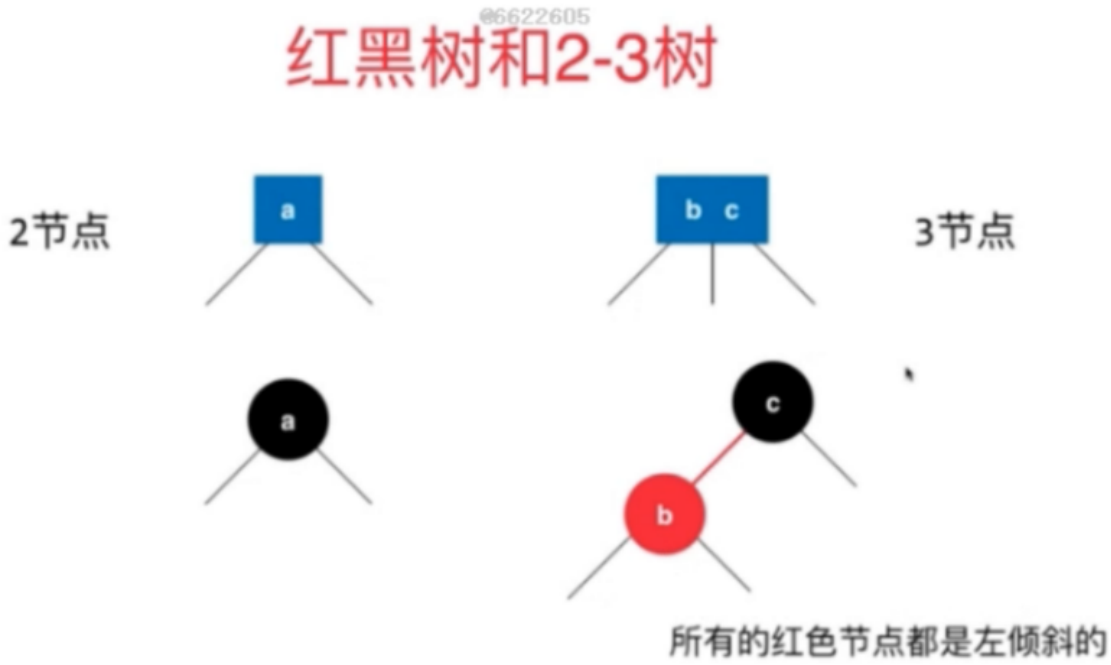


如果插入3-节点，父亲节点为3-节点



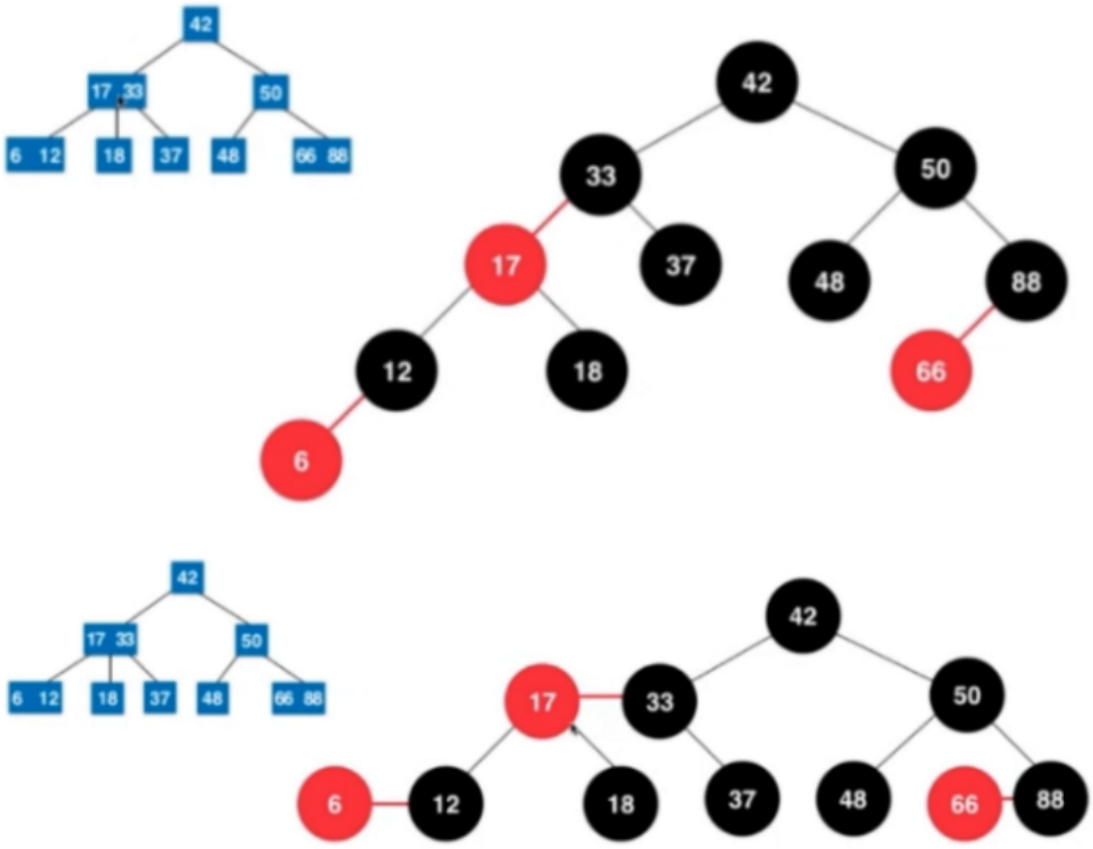
红黑树（左倾红黑树）

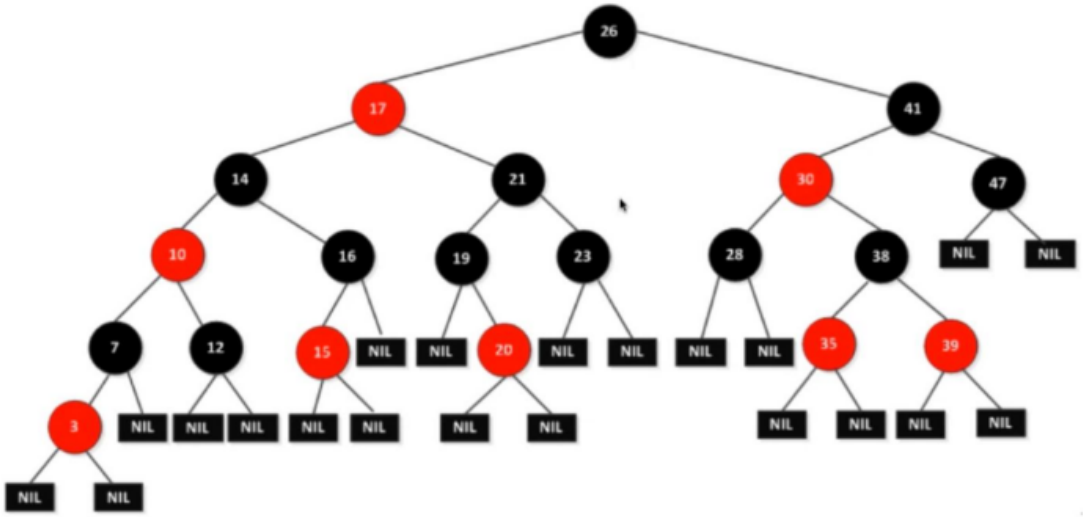
- 红黑树与2-3树具有等价性



红黑树中的节点我们都可以使用对应的2节点或者3节点表示

- 一个栗子





此图中的2-3树和红黑树就是等价的

红黑树的定义

- 此处我们讲的定义是红黑树在《算法4》中的定义

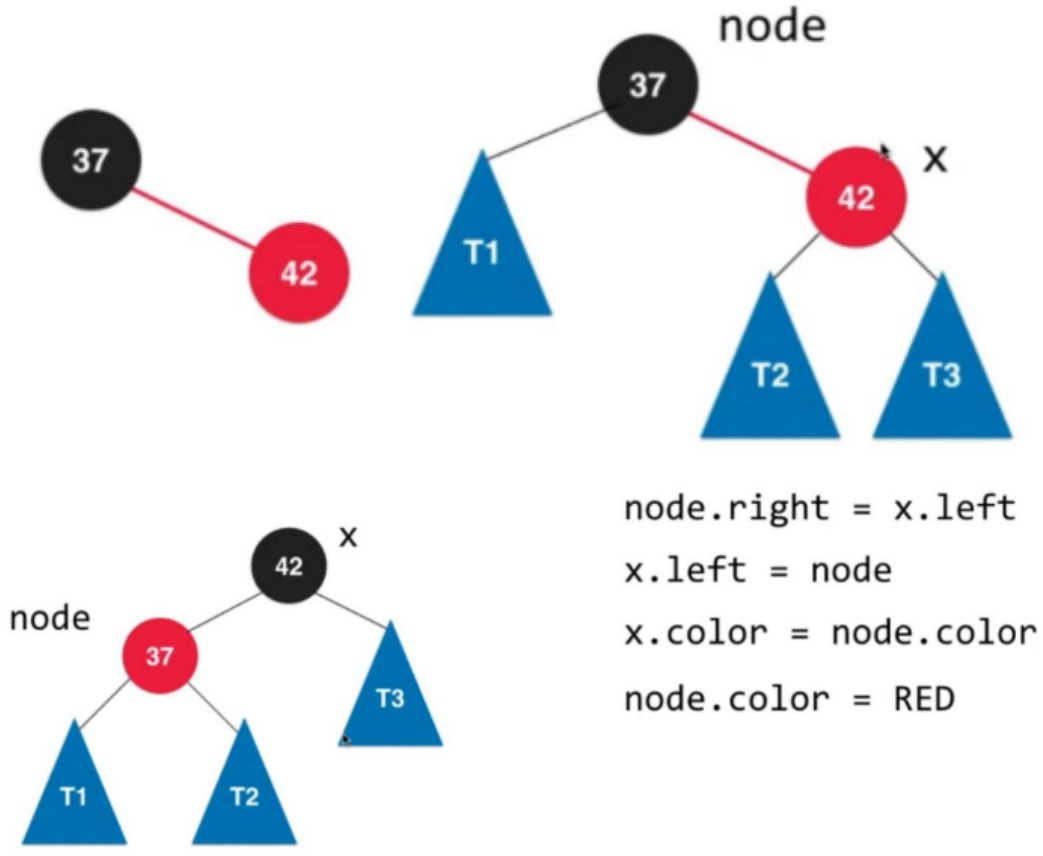
  1. 每个节点或者是红色的，或者是黑色的
  2. 根节点是黑色的
  3. 每一个叶子节点（最后的空节点）是黑色的
  4. 如果一个节点是红色的，那么他的孩子节点都是黑色的
  5. 从任意一个节点到叶子节点，经过的黑色节点是一样的

  从定义上我们可以发现，对于红黑树而言，其并不是严格意义上的平衡二叉树，但是其是保持“黑平衡”的二叉树，最大高度为 **2logn**.

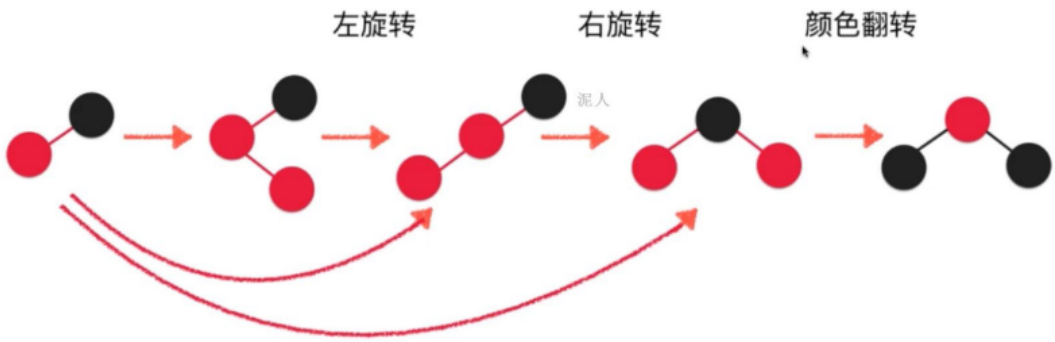
- 红黑树中节点的添加操作

  在红黑树中左旋转是对两个节点的操作，右旋转是对三个节点的操作

  - 节点的默认颜色为红色，当我们插入的节点比已存在的节点小时，插入过程就已经结束了，当我们插入的节点比当前节点大时（如图所示），则此时需要进行**左旋转**。当我们对旋转后的节点颜色进行改变时，可能会导致出现连续的两个节点都是红节点，不需要担心这一点，因为左旋转只是一个子过程，我们还需要进行其他的调整过程才能够满足红黑树的相应性质



- 插入过程如图，调整的时机与AVL树相同



- 如果插入的节点是2节点，且插入的位置是左孩子，则得到上图的起始状态
- 如果插入的节点是2节点，且插入的位置是右孩子，则需要进行左旋转得到起始状态
- 对于一个初始的3节点，如果插入的节点（**默认插入的节点都是红色节点**）是左孩子节点的右孩子则需要进行左旋转
- 如果插入的节点是左孩子节点的左孩子，则不跳过左旋转步骤
- 对调整过的节点进行右旋转，右旋转是对黑色的根节点进行旋转的，需要进行一次颜色维护，则可以得到根节点为黑色节点，左右节点为红色节点的子树
- 然后对子树进行颜色翻转即可
- 最后，我们需要调用函数保证根节点的颜色为黑色

实现

```
import java.util.ArrayList;

public class RBTree<K extends Comparable<K>, V> {

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

```
    private class Node{
        public K key;
        public V value;
        public Node left, right;
        public boolean color;

        public Node(K key, V value){
            this.key = key;
            this.value = value;
            left = null;
            right = null;
            color = RED;
        }
    }

    private Node root;
    private int size;

    public RBTree(){
        root = null;
        size = 0;
    }

    public int getSize(){
        return size;
    }

    public boolean isEmpty(){
        return size == 0;
    }

    // 判断节点node的颜色
    private boolean isRed(Node node){
        //需要注意的一个点，对于空节点，其颜色是黑色的
        if(node == null)
            return BLACK;
        return node.color;
    }

    //   node                     x
    //  /   \     左旋转         /  \
    // T1   x   --------->   node   T3
    //     / \              /   \
    //    T2 T3            T1   T2
    private Node leftRotate(Node node){
        Node x = node.right;
        node.right = x.left;
        x.left = node;

        x.color = node.color;
        node.color = RED;

        return node;
    }

    //     node                   x
    //    /   \     右旋转       /  \
    //   x    T2   ------->   y   node
```

```
//  / \                        /  \
// y  T1                      T1  T2
private Node rightRotate(Node node){

    Node x = node.left;
    node.left = x.right;
    x.right = node;

    x.color = node.color;
    node.color = RED;
}

// 颜色翻转
//得到一个根节点为红色，左右子树为黑色的节点
private void flipColors(Node node){

    node.color = RED;
    node.left.color = BLACK;
    node.right.color = BLACK;
}

// 向红黑树中添加新的元素(key, value)
public void add(K key, V value){
    root = add(k,v);
    root.color = BLACK;
}

// 向以node为根的红黑树中插入元素(key, value)，递归算法
// 返回插入新节点后红黑树的根
private Node add(Node node, K key, V value){

    if(node == null){
        size ++;
        return new Node(key, value); // 默认插入红色节点
    }

    if(key.compareTo(node.key) < 0)
        node.left = add(node.left, key, value);
    else if(key.compareTo(node.key) > 0)
        node.right = add(node.right, key, value);
    else // key.compareTo(node.key) == 0
        node.value = value;
    //      root
    //      /
    //     newNode
    // 此时不需要处理

    //      root
    //        \
    //        newNode
    // 默认插入的节点都是红色，此时需要进行左旋转
    // 注意，此时node指向的是root，这是递归写法决定的
    // node.right = add(node.right, key, value)
    if(isRed(node.right) && !isRed(node.left)){
        //左孩子也是红结点的话直接颜色翻转就好了
        node = leftRotate(node);
    }
```

```
        //      root
        //      /
        //    node
        //     /
        //  newNode
        // 此时需要进行右旋转,node还是指向子树的根节点的
        // 递归定义，返回插入新节点后红黑树的根
        if(isRed(node.left) && isRed(node.left.left)){
            node = rightRotate(node);
        }

        // 经过上面的处理后
        //
        //      root
        //      /   \
        //    node  node
        //
        // 树的结构接变成这样了
        // 此时我们需要改变颜色
        if(isRed(node.left) && isRed(node.right))
            flipColors(node);

         //     root        root
        //      /             \
        //    node             node
        //       \               \
        //      newNode          newNode
        //
        // 不会有这两种情况，因为上面进行了左旋转

        return node;
    }

    // 返回以node为根节点的二分搜索树中，key所在的节点
    private Node getNode(Node node, K key){

        if(node == null)
            return null;

        if(key.equals(node.key))
            return node;
        else if(key.compareTo(node.key) < 0)
            return getNode(node.left, key);
        else // if(key.compareTo(node.key) > 0)
            return getNode(node.right, key);
    }

    public boolean contains(K key){
        return getNode(root, key) != null;
    }

    public V get(K key){

        Node node = getNode(root, key);
        return node == null ? null : node.value;
    }

    public void set(K key, V newValue){
```

```
        Node node = getNode(root, key);
        if(node == null)
            throw new IllegalArgumentException(key + " doesn't exist!");

        node.value = newValue;
    }
}
```

- 红黑树的删除逻辑比较复杂，可以自行了解，此处不再给出

红黑树的性能总结

- 对于完全随机的数据，普通的二分搜索树很好用！
- 缺点：极端情况退化成链表（或者高度不平衡）
- 对于查询较多的使用情况，AVL树很好用！
- 红黑树牺牲了平衡性（21ogn的高度）统计性能更优（综合增删改查所有的操作）

更多的树

- splay tree 伸展树：使用了局部性原理（刚被访问的内容下次高概率再次被访问）
- 右倾红黑树