实验三 二叉查找树、 红黑树的基本操作实现 报告

姓名：陈矛　　学号：SA22225116

一、实验原理

1. 算法流程图：

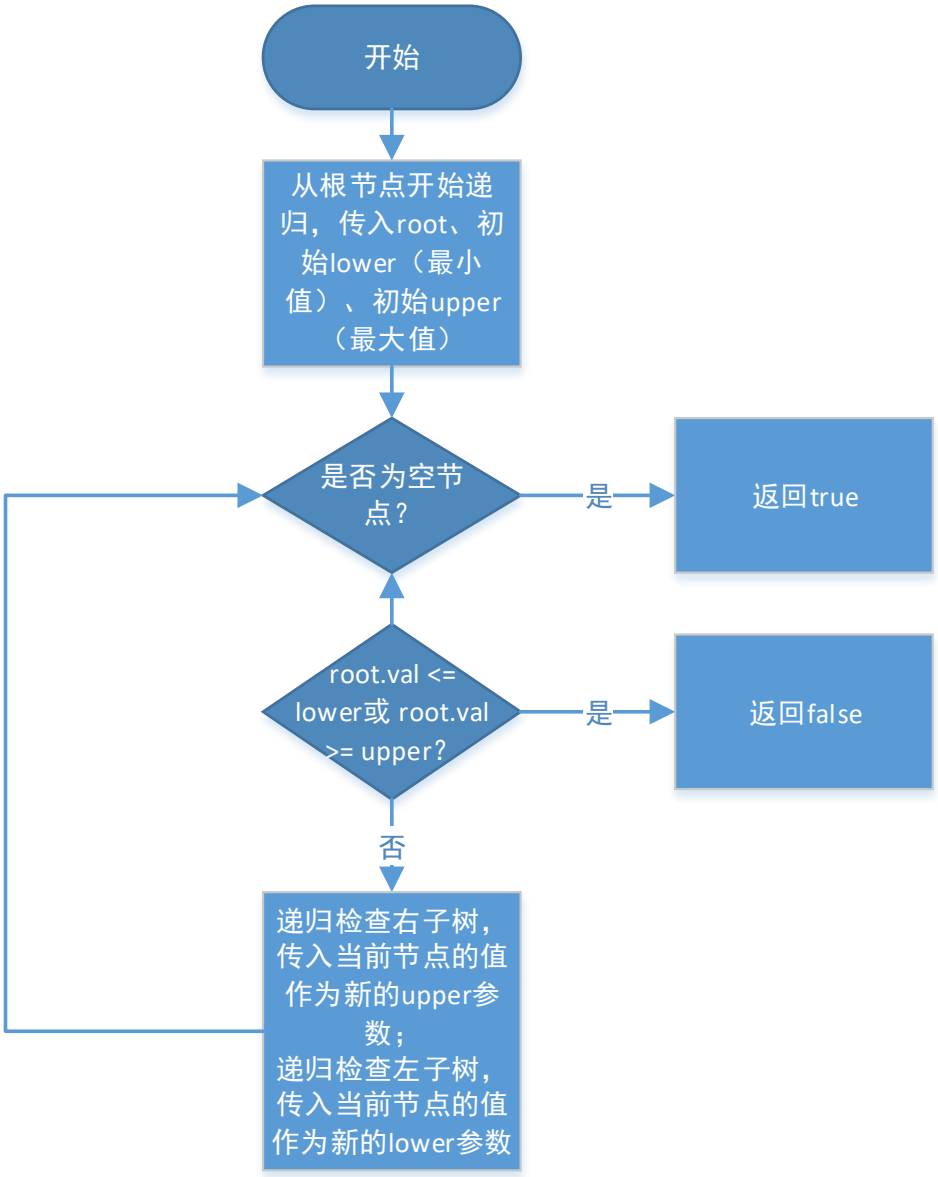

图 递归算法检查 BST 流程图

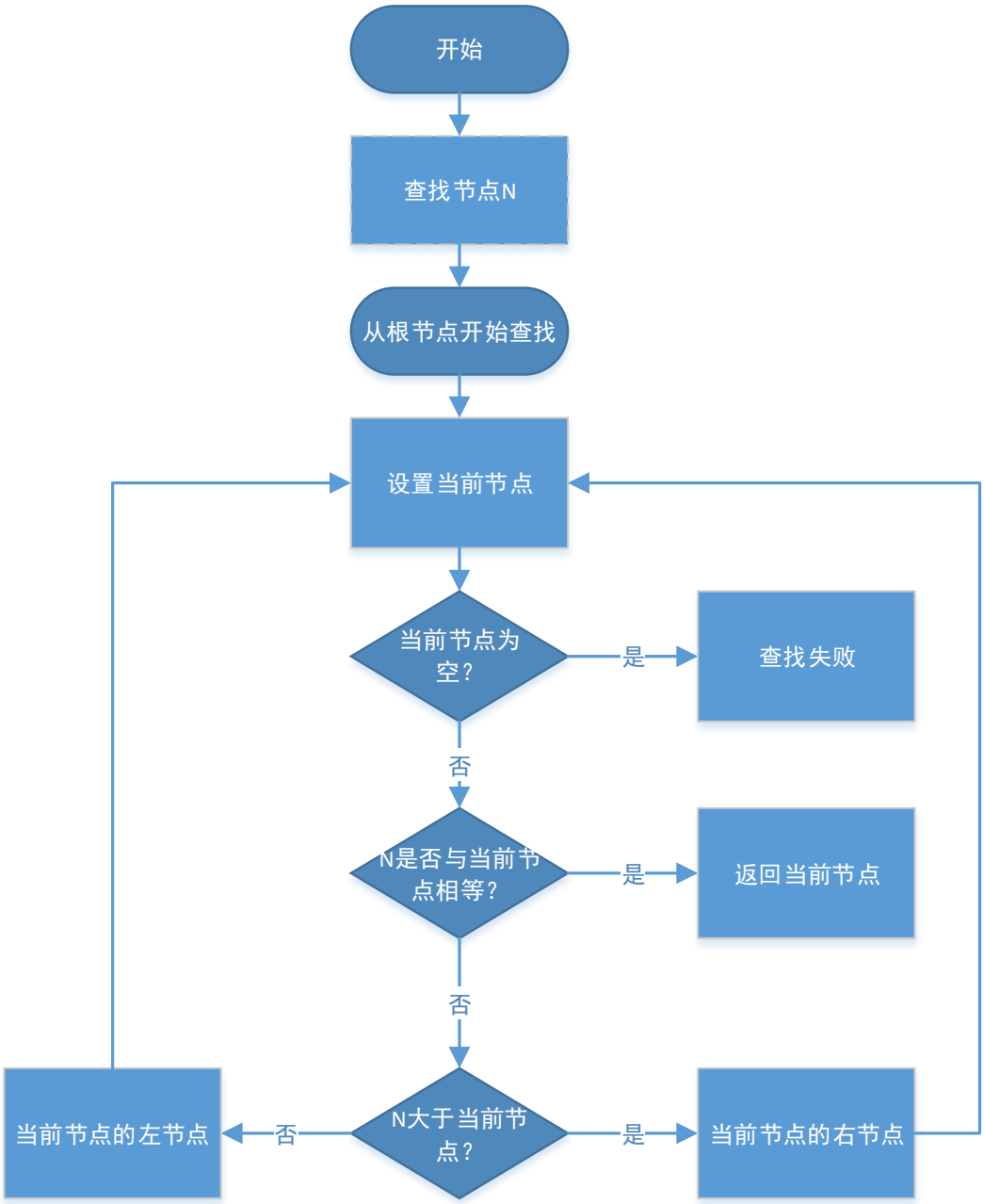

图 BST 的查找操作流程图

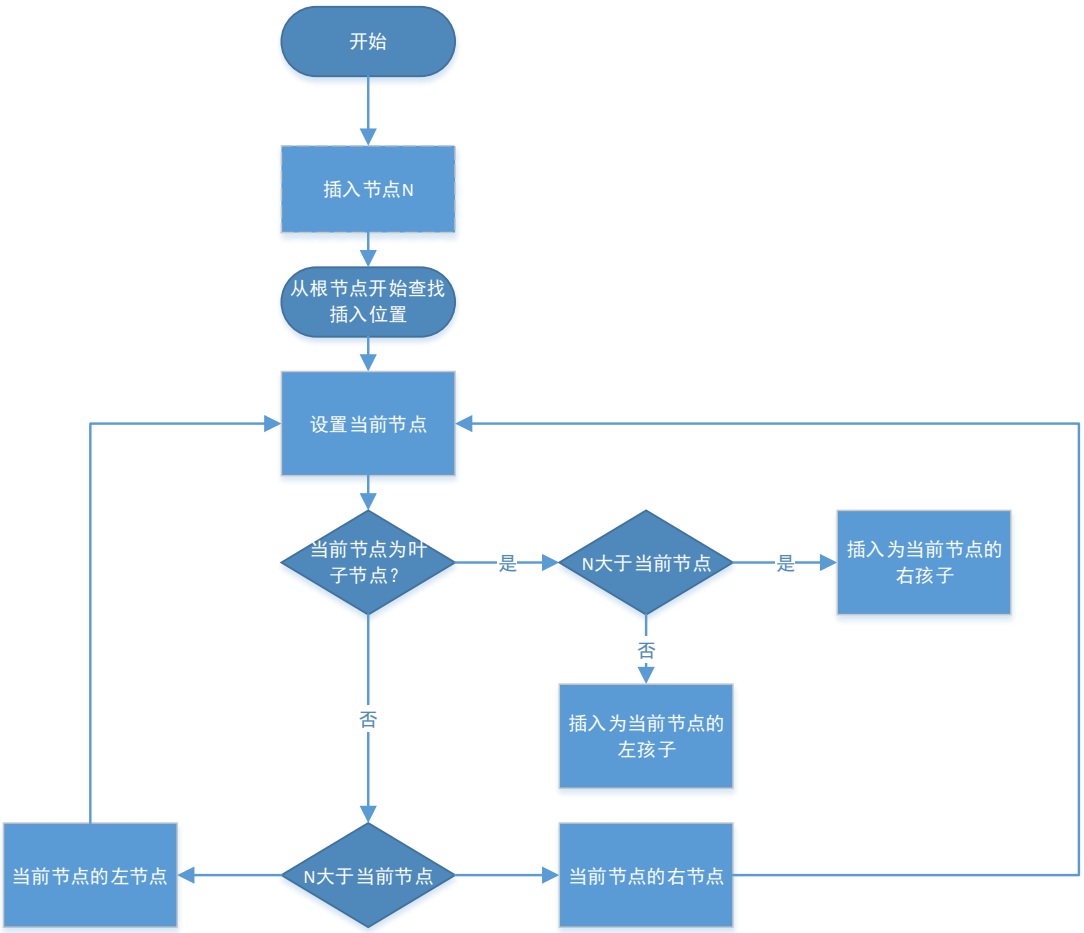

图 BST 的插入操作流程图

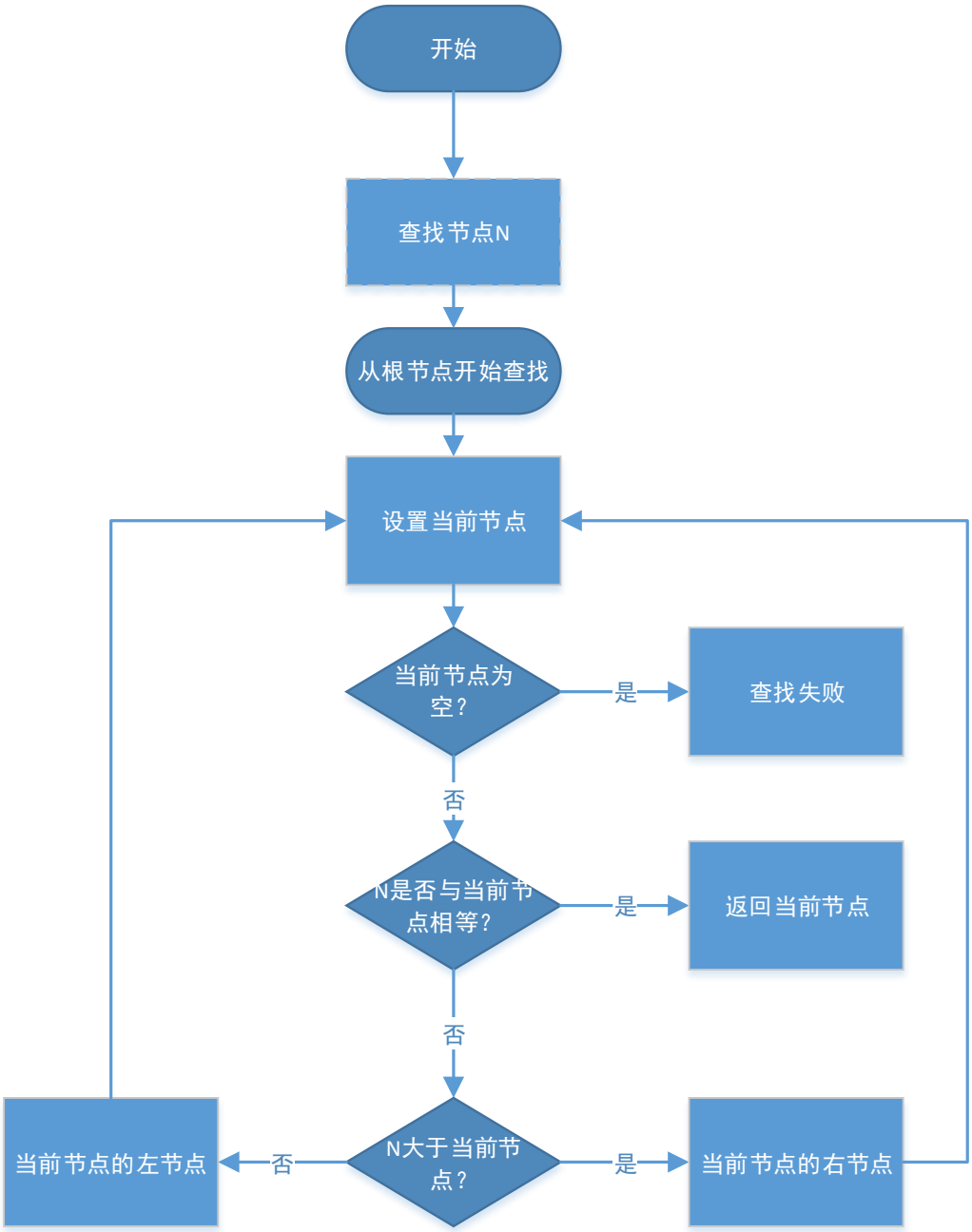

图 RBT 的查找操作流程图

2. 二叉查找树 **(Binary Search Tree)**

二叉查找树又称二叉搜索树， 二叉排序树， 特点如下:

1) 左子树上所有结点值均小于根结点
2) 右子树上所有结点值均大于根结点
3) 结点的左右子树本身又是一颗二叉查找树
4) 二叉查找树中序遍历得到结果是递增排序的结点序列。

典型操作： 插入、删除、查找

3. 红黑树

红黑树中每个结点包含五个域： color,key,left,right 和 p。 如果某结点没有一个子结点或父结点， 则该域指向 NIL。一棵二叉树如果满足下面的红黑性质， 则为一棵红黑树：

1) 每个结点或是红的， 或是黑的。
2) 根结点是黑的。
3) 每个叶结点 (NIL) 是黑的。
4) 如果一个结点是红的， 则它的两个儿子都是黑的。
5) 对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

从某个结点 x 出发 (不包括该结点) 到达一个叶结点的任意一条路径上， 黑色结点的个数称为该结点 x 的黑高度， 用 bh(x) 表示。
引理： 一颗有 **n** 个内结点的红黑树的高度至多为 **2lg(n+1)**。

4. 时间复杂度和空间复杂度

操作名(h树高)	二叉查找数	红黑树
查找	O(h)	O(lgn)
查最大/小元素	O(h)	O(lgn)
前驱/后继	O(h)	O(lgn)
插入	O(h)	O(lgn)
删除	O(h)	O(lgn)
旋转	无	O(1)
高度	下取整(lgn)+1<=h<=n	<=2lg(n+1)

二、运行结果及分析

1. 给你一个二叉树的根节点 root ， 判断其是否是一个有效的二叉搜索树。

算法流程图：

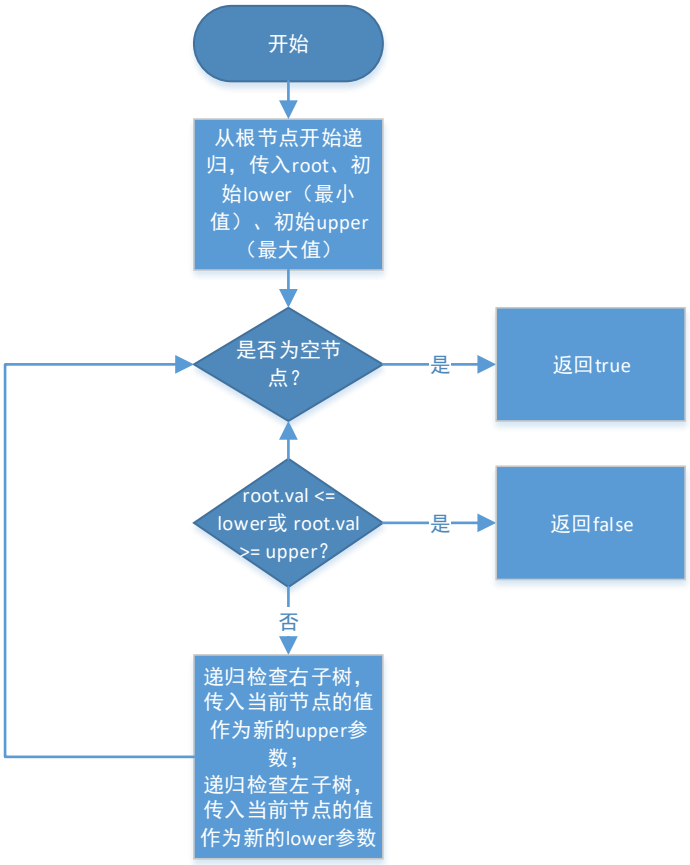

图 递归算法检查 BST 流程图

关键代码：

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root,Long.MIN_VALUE,Long.MAX_VALUE);
    }

    public boolean isValidBST(TreeNode root,long lower,long upper) {
        //空节点一定是 BST
        if (root == null) return true;

        //如果不满足条件，则不是 BST，返回 false
        if(root.val <= lower || root.val >= upper) return false;

        //如果满足条件，继续往下递归
        //更新 lower 和 upper，对于左子树，只有上界要求，不需要更新下界；对于右子树同理；
        //以左子树为例，本来应该传入上界 Math.min(upper,root.val)的，但是由于上一步已经检查过了 root.val 必然小于 upper，所以只需传入 root.val 即可
        return (isValidBST(root.left,lower,root.val)&&(isValidBST(root.right,root.val,upper)));
```

```
    }
}
```

算法实现流程：

1) 有效二叉搜索树定义如下：
   节点的左子树只包含**小于**当前节点的数。
   节点的右子树只包含**大于**当前节点的数。
   所有左子树和右子树自身必须也是二叉搜索树。
   根据这样的性质，可以看出，**当前节点的数**会在检查它的左子树和右子树的时候被做比较，所以在递归中，必须要把这个参数传递下去。
2) 发现当前节点为空节点时，向上返回 true，因为空树也是一个合法的 BST。
3) 发现当前节点不是空节点时，检查当前节点的值 root.val 是否在 lower 和 upper 之间，如果不在这个区间内，说明发生违规，返回 false。
4) 当第三步的检查通过后，则需要继续向下递归。对左子树和右子树都必须要进行检查，同时传递更新后的 lower 和 upper，对于左子树，只有可能更新了上界要求，不需要更新下界；对于右子树同理。以左子树为例，本来应该传入上界 Math.min(upper,root.val)的，但是由于上一步已经检查过了 root.val 必然小于 upper，所以只需传入 root.val 即可。
5) 一直进行递归，只要有一个节点的检查出现 false，那么递归的结果就会是 false；否则说明检验通过，返回 true。
6) 算法结束。

算法运行结果（使用 LeetCode98 题的测试集进行测试）：

得到这样结果的原因（以该示例为例）：

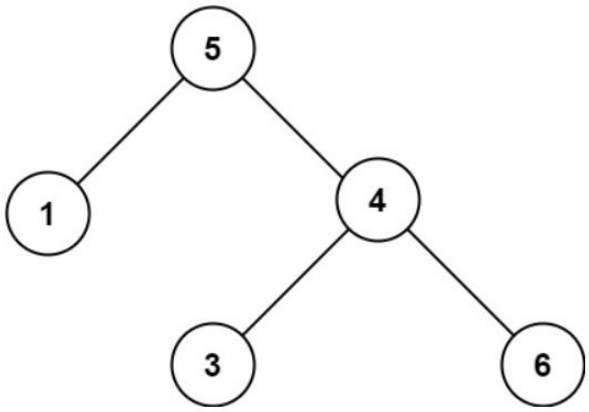

图 测试示例

1) 第一步传入根节点 root，以及初始的 lower 值负无穷大，和初始的 upper 值正无穷大。
2) 很显然 root.val = 5，介于正无穷大和负无穷大之间。于是继续向下递归并更新上下界。当前节点的值成为了右子树的上界，也成为了左子树的下界。
3) 对于值为 1 的节点，检查其值在负无穷大和它的上界 5 之间，检验通过，继续向下递归。
4) 发现其左右节点都为空，对于节点 1 的递归结束，向上返回 true。继续检查节点 4。
5) 检查节点 4 时，它的上界为正无穷大，下界为 5，发现其值不在上下界之间，检验不通过，返回 false。
6) 一直向上返回 false，递归结束，该树不是 BST。算法结束。

2. 用输入序列 {1,5,6,7,8,9,10,11,12,13,14,15}构建一个红黑树并展示结果。再用输入序列{14,9,5}执行删除操作并展示结果

关键代码：

红黑树节点的插入：

```
        /*
 * 将结点插入到红黑树中
 *
 * 参数说明：
 *     node 插入的结点
 */
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);
}
```

红黑树节点的删除：

```
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);
}
```

算法实现流程：

该部分流程可以分为两个部分：红黑树的插入节点和删除节点两个操作。

对于插入操作，需要遵循以下的流程：

1) 按照搜索树的特征进行插入

2) 插入时：插入的结点一定是红色的，如果为黑色，会破坏第五条规则
   a) 如果插入的结点是根节点，将颜色改为黑色
   b) 插入的结点的父结点是黑色的，则插入完成
   c) 插入的结点的父结点是红色的，则需要修复，且继续向上调整，直到为根或满足规则
   d) 如果根修改之后为红色，一定要改过来，改为黑色

通过这样的方式，依次插入所有的节点。

对于删除操作，又可以分为四种情况：
情况 1：删除节点为叶子节点。
情况 2：删除节点只有左孩子，没有右孩子。
情况 3：删除节点只有右孩子，没有左孩子。
情况 4：删除节点既有左孩子，又有右孩子。

对于每种情况，都有特定的规则可以解决它。按照对应的方法编写代码，即可实现删除算法。

算法运行结果：

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java ...
依次插入值为：1 5 6 7 8 9 10 11 12 13 14 15 的节点后，中序遍历红黑树各节点:
black   1
black   5
black   6
black   7
black   8
red     9
black   10
black   11
black   12
red     13
black   14
red     15
删除节点: 14
删除节点: 9
删除节点: 5

 中序遍历红黑树各节点:
red     1
black   6
black   7
red     8
black   10
black   11
black   12
black   13
black   15

Process finished with exit code 0
```

运行结果中分别以实验要求所需的方式，打印了插入节点后的红黑树以及删除了三个节点之后的红黑树的中序序列。

3. 请说明二叉查找树和红黑树的区别以及时间、 空间性能。

1) 二叉查找树的定义：
   a) 若任意节点的左子树不为空，则左子树上所有节点的值均小于它的根节点的值；
   b) 若任意节点的右子树不为空，则右子树上所有节点的值均大于或等于它的根节点的值；
   c) 任意节点的左、右子树分别为二叉查找树

2) 红黑树的定义：

红黑树（Red Black Tree）是一种自平衡二叉查找树。所谓的平衡树是指一种改进的二叉查找树，顾名思义平衡树就是将二叉查找树平衡均匀地分布，这样的好处就是可以减少二叉查找树的深度。红黑树除了具备二叉查找树的基本特性之外，还具备以下特性

   a) 节点是红色或黑色；
   b) 根节点是黑色；
   c) 所有叶子都是黑色的空节点（NIL 节点）；
   d) 每个红色节点必须有两个黑色的子节点，也就是说从每个叶子到根的所有路径上，不能有两个连续的红色节点；
   e) 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色节点。

总之，红黑树是更加平衡的二叉查找树，本身也具备二叉查找树的性质。

3) 二者的时间、空间性能

操作名(h树高)	二叉查找数	红黑树
查找	O(h)	O(lgn)
查最大/小元素	O(h)	O(lgn)
前驱/后继	O(h)	O(lgn)
插入	O(h)	O(lgn)
删除	O(h)	O(lgn)
旋转	无	O(1)
高度	下取整(lgn)+1<=h<=n	<=2lg(n+1)

三、实验总结

在本次实验中，通过实现红黑树和二叉查找树的具体算法，深入了解了两种数据结构在实现和解决问题中的作用，以及实现上的细节。红黑树本身就是一种特殊的二叉查找树。红黑树和二叉查找树都是被用来存储有序的数据的数据结构。二叉查找树是一种特殊的二叉树，其中每个节点的值都大于其左子树中的任意节点的值，而小于右子树中的任意节点的值。红黑树也是一种二叉查找树，但是它在二叉查找树的基础上增加了一些限制条件，以维护树的平衡。

相比于二叉查找树，红黑树具有以下几个优点：

1) 时间复杂度更优：在最坏情况下，红黑树的时间复杂度为 O(logn)，而二叉查找树的时间复杂度可以达到 O(n)。
2) 更加稳定：红黑树是一种自平衡的数据结构，因此它不容易出现长链的情况，从而使得树的高度更加稳定。
3) 操作更加方便：红黑树的插入和删除操作比二叉查找树更加方便，因为红黑树的平衡性使得在插入和删除节点时不需要考虑太多的情况。

但是，红黑树也有一些缺点：

1) 实现更加复杂：由于红黑树需要维护平衡性，因此它的实现要比二叉查找树更加复杂。
2) 空间复杂度略高：红黑树需要额外的一个存储空间来表示每个节点的颜色，因此它的空间复杂度略高于二叉查找树。

总的来说，红黑树是一种很优秀的数据结构，它在保证了二叉查找树的有序性的同时，还能够通过维护平衡性来提高操作效率。在需要频繁插入和删除节点的场景中，红黑树是一个很好的选择。

实验日期：2022 年 12 月 1 日

附录

实验源代码：

1. Solution.java

```java
package SolutionForLab3;

/**
 * @author ChenMao
 * @create 2022-12-01 20:39
 */
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root,Long.MIN_VALUE,Long.MAX_VALUE);
    }

    public boolean isValidBST(TreeNode root,long lower,long upper) {
        //空节点一定是BST
        if (root == null) return true;

        //如果不满足条件，则不是BST，返回false
        if(root.val <= lower || root.val >= upper) return false;
```

```
        //如果满足条件，继续往下递归
        //更新lower 和upper，对于左子树，只有上界要求，不需要更新下界；对于右子树同理；
        //以左子树为例，本来应该传入上界Math.min(upper,root.val)的，但是由于上一步已经检查过了root.val 必然小于upper，所以只需传入root.val即可
        return (isValidBST(root.left,lower,root.val)&&(isValidBST(root.right,root.val,upper)));

    }
}
```

2. BST.java

```
package SolutionForLab3;

/**
 * @author ChenMao
 * @create 2022-12-01 20:50
 */


public class BST {
    /*
        1. 实现下列关于二叉查找树、 红黑树的判断、 构建、 删除等操作。 并写出这些操作的流程图或伪代码。
        2. 请说明二叉查找树和红黑树的区别以及时间、 空间性能。

     */

    /*
        二叉搜索树的节点类 —— class TreeNode
        二叉搜索树的属性：要找到一颗二叉搜索树只需要知道这颗树的根节点。
     */



//    public TreeNode root;//BST 的根节点

    private int lastVisit = Integer.MIN_VALUE;

    public static void main(String[] args) {
```

```
        TreeNode root = new TreeNode(5);
        BST test = new BST();
        int[] nums_input = {1,4,3,6};

        for (int num : nums_input){
            test.insert(root,num);
        }

       TreeOperation operation_test = new TreeOperation();
        operation_test.show(root);

        System.out.println("是否为二叉搜索树？："+test.isBST(root));
    }

    /**
     * 查找是否存在节点
     * 思路：根据二叉排序树的特点：
     * ①如果要查找的值小于当前节点的值，那么，就往当前节点的左子树走
     * ②如果要查找的值大于当前节点的值，那么，就往当前节点的右子树走
     *
     * @param val 带查找的val
     * @return boolean 是否存在
     */
    public boolean find(TreeNode root,int val) {
        TreeNode cur = root;
        while (cur != null) {
            if (val < root.val) {
                cur = cur.left;
            } else if (val > root.val) {
                cur = cur.right;
            } else {
                return true;
            }
        }
        return false;
    }

    /**
     * 往二叉树中插入节点
     *
     * 思路如下：
     *
     * @param val 待插入的节点
     */
```

```java
    public void insert(TreeNode root,int val) {
        if (root == null) { //如果是空树，那么，直接插入
            root = new TreeNode(val);
            return;
        }

        TreeNode cur = root;
        TreeNode parent = null; //parent 为cur 的父节点
        while (true) {
            if (cur == null) { //在遍历过程中，找到了合适是位置，就指针
插入（没有重复节点）
                if (parent.val < val) {
                    parent.right = new TreeNode(val);
                } else {
                    parent.left = new TreeNode(val);
                }
                return;
            }

            if (val < cur.val) {
                parent = cur;
                cur = cur.left;
            } else if (val > cur.val) {
                parent = cur;
                cur = cur.right;
            } else {
                throw new RuntimeException("插入失败，已经存在 val");
            }
        }
    }

    /**
     * 删除树中节点
     *
     * 思路如下:
     *
     * @param val 待删除的节点
     */
    public void remove(TreeNode root, int val) {
        if (root == null) {
            throw new RuntimeException("为空树，删除错误！");
        }
        TreeNode cur = root;
        TreeNode parent = null;
```

```
        //查找是否val 节点的位置
        while (cur != null) {
            if (val < cur.val) {
                parent = cur;
                cur = cur.left;
            } else if (val > cur.val) {
                parent = cur;
                cur = cur.right;
            } else {
                break;
            }
        }
        if (cur == null) {
            throw new RuntimeException("找不到 val，输入 val 不合法");
        }

        //cur 为待删除的节点
        //parent 为待删除的节点的父节点
        /*
         * 情况 1：如果待删除的节点没有左孩子
         * 其中
         * ①待删除的节点有右孩子
         * ②待删除的节点没有右孩子
         * 两种情况可以合并
         */
        if (cur.left == null) {
            if (cur == root) { //①如果要删除的是根节点
                root = cur.right;
            } else if (cur == parent.left) { //②如果要删除的是其父节点的左孩子
                parent.left = cur.right;
            } else { //③如果要删除的节点为其父节点的右孩子
                parent.right = cur.right;
            }
        }
        /*
         * 情况 2:如果待删除的节点没有右孩子
         *
         * 其中:待删除的节点必定存在左孩子
         */
        else if (cur.right == null) { //①如果要删除的是根节点
            if (cur == root) {
                root = cur.left;
            } else if (cur == parent.left) { //②如果要删除的是其父节
```

```
点的左孩子
                parent.left = cur.left;
            } else { //③如果要删除的节点为其父节点的右孩子
                parent.right = cur.left;
            }
        }
        /*
         * 情况 3:如果待删除的节点既有左孩子又有右孩子
         *
         * 思路:
         * 因为是排序二叉树,要找到整颗二叉树第一个大于该节点的节点,只需要,先向右走一步,然后一路往最左走就可以找到了
         * 因此:
         * ①先向右走一步
         * ②不断向左走
         * ③找到第一个大于待删除的节点的节点,将该节点的值,替换到待删除的节点
         * ④删除找到的这个节点
         * ⑤完成删除
         *
         */
        else {
            TreeNode nextParent = cur; //定义父节点,初始化就是待删除的节点
            TreeNode next = cur.right; //定义next 为当前走到的节点,最终目的是找到第一个大于待删除的节点
            while (next.left != null) {
                nextParent = next;
                next = next.left;
            }
            cur.val = next.val; //找到之后,完成值的替换
            if (nextParent == cur) { //此时的父节点就是待删除的节点,那么说明找到的节点为父节点的右孩子(因为此时next 只走了一步)
                nextParent.right = next.right;
            } else { //此时父节点不是待删除的节点,即next 确实往左走了,且走到了头.
                nextParent.left = next.right;
            }
        }

    }

    public boolean isBST(TreeNode root){
        if(root==null)return true;              //空树也是BST
```

```
        boolean judgeleft = isBST(root.left);   //判断左子树是否是

        if(root.val >= lastVisit && judgeleft){ //当前节点比上次访问
的节点的数大
            lastVisit = root.val;
        }else {
            return false;
        }
        boolean judegright = isBST(root.right);//判断右子树是否是
        return judegright;
    }



}
```

3. TreeNode.java

```
package SolutionForLab3;

/**
 * @author ChenMao
 * @create 2022-12-01 21:54
 */
public class TreeNode {
    public int val;
    public TreeNode left;
    public TreeNode right;

    public TreeNode(){
        super();
    }

    public TreeNode(int val) {
        this.val = val;
    }
}
```

4. TreeOperation.java

```
package SolutionForLab3;



/**
 * @author ChenMao
 * @create 2022-12-01 21:34
```

```java
 */
// TreeOperation.java
public class TreeOperation {
    /*
    树的结构示例：
              1
            /   \
          2       3
         / \     / \
        4   5   6   7
    */

    // 用于获得树的层数
    public int getTreeDepth(TreeNode root) {
        return root == null ? 0 : (1 + Math.max(getTreeDepth(root.left),
getTreeDepth(root.right)));
    }


    public void writeArray(TreeNode currNode, int rowIndex, int
columnIndex, String[][] res, int treeDepth) {
        // 保证输入的树不为空
        if (currNode == null) return;
        // 先将当前节点保存到二维数组中
        res[rowIndex][columnIndex] = String.valueOf(currNode.val);

        // 计算当前位于树的第几层
        int currLevel = ((rowIndex + 1) / 2);
        // 若到了最后一层，则返回
        if (currLevel == treeDepth) return;
        // 计算当前行到下一行，每个元素之间的间隔（下一行的列索引与当前元
素的列索引之间的间隔）
        int gap = treeDepth - currLevel - 1;

        // 对左儿子进行判断，若有左儿子，则记录相应的"/"与左儿子的值
        if (currNode.left != null) {
            res[rowIndex + 1][columnIndex - gap] = "/";
            writeArray(currNode.left, rowIndex + 2, columnIndex - gap
* 2, res, treeDepth);
        }

        // 对右儿子进行判断，若有右儿子，则记录相应的"\"与右儿子的值
        if (currNode.right != null) {
            res[rowIndex + 1][columnIndex + gap] = "\\";
```

```
            writeArray(currNode.right, rowIndex + 2, columnIndex + gap
* 2, res, treeDepth);
        }
    }


    public void show(TreeNode root) {
        if (root == null) System.out.println("EMPTY!");
        // 得到树的深度
        int treeDepth = getTreeDepth(root);

        // 最后一行的宽度为2 的（n - 1）次方乘3，再加1
        // 作为整个二维数组的宽度
        int arrayHeight = treeDepth * 2 - 1;
        int arrayWidth = (2 << (treeDepth - 2)) * 3 + 1;
        // 用一个字符串数组来存储每个位置应显示的元素
        String[][] res = new String[arrayHeight][arrayWidth];
        // 对数组进行初始化，默认为一个空格
        for (int i = 0; i < arrayHeight; i ++) {
            for (int j = 0; j < arrayWidth; j ++) {
                res[i][j] = " ";
            }
        }

        // 从根节点开始，递归处理整个树
        // res[0][(arrayWidth + 1)/ 2] = (char)(root.val + '0');
        writeArray(root, 0, arrayWidth/ 2, res, treeDepth);

        // 此时，已经将所有需要显示的元素储存到了二维数组中，将其拼接并打
印即可
        for (String[] line: res) {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < line.length; i ++) {
                sb.append(line[i]);
                if (line[i].length() > 1 && i <= line.length - 1) {
                    i += line[i].length() > 4 ? 2: line[i].length() -
1;
                }
            }
            System.out.println(sb.toString());
        }
    }
}
```

5. RBTree.java

```java
package SolutionForLab3;


/**
 * @author ChenMao
 * @create 2022-12-01 21:34
 */
// TreeOperation.java
public class TreeOperation {
    /*
    树的结构示例:
              1
            /   \
          2       3
         / \     / \
        4   5   6   7
    */

    // 用于获得树的层数
    public int getTreeDepth(TreeNode root) {
        return root == null ? 0 : (1 + Math.max(getTreeDepth(root.left),
getTreeDepth(root.right)));
    }


    public void writeArray(TreeNode currNode, int rowIndex, int
columnIndex, String[][] res, int treeDepth) {
        // 保证输入的树不为空
        if (currNode == null) return;
        // 先将当前节点保存到二维数组中
        res[rowIndex][columnIndex] = String.valueOf(currNode.val);

        // 计算当前位于树的第几层
        int currLevel = ((rowIndex + 1) / 2);
        // 若到了最后一层，则返回
        if (currLevel == treeDepth) return;
        // 计算当前行到下一行，每个元素之间的间隔（下一行的列索引与当前元
素的列索引之间的间隔）
        int gap = treeDepth - currLevel - 1;

        // 对左儿子进行判断，若有左儿子，则记录相应的"/"与左儿子的值
        if (currNode.left != null) {
            res[rowIndex + 1][columnIndex - gap] = "/";
```

```
            writeArray(currNode.left, rowIndex + 2, columnIndex - gap
* 2, res, treeDepth);
        }

        // 对右儿子进行判断，若有右儿子，则记录相应的"\"与右儿子的值
        if (currNode.right != null) {
            res[rowIndex + 1][columnIndex + gap] = "\\";
            writeArray(currNode.right, rowIndex + 2, columnIndex + gap
* 2, res, treeDepth);
        }
    }


    public void show(TreeNode root) {
        if (root == null) System.out.println("EMPTY!");
        // 得到树的深度
        int treeDepth = getTreeDepth(root);

        // 最后一行的宽度为2 的（n - 1）次方乘3，再加1
        // 作为整个二维数组的宽度
        int arrayHeight = treeDepth * 2 - 1;
        int arrayWidth = (2 << (treeDepth - 2)) * 3 + 1;
        // 用一个字符串数组来存储每个位置应显示的元素
        String[][] res = new String[arrayHeight][arrayWidth];
        // 对数组进行初始化，默认为一个空格
        for (int i = 0; i < arrayHeight; i ++) {
            for (int j = 0; j < arrayWidth; j ++) {
                res[i][j] = " ";
            }
        }

        // 从根节点开始，递归处理整个树
        // res[0][(arrayWidth + 1)/ 2] = (char)(root.val + '0');
        writeArray(root, 0, arrayWidth/ 2, res, treeDepth);

        // 此时，已经将所有需要显示的元素储存到了二维数组中，将其拼接并打
印即可
        for (String[] line: res) {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < line.length; i ++) {
                sb.append(line[i]);
                if (line[i].length() > 1 && i <= line.length - 1) {
                    i += line[i].length() > 4 ? 2: line[i].length() -
1;
```

```
                }
            }
            System.out.println(sb.toString());
        }
    }
}
```

6. RBTreeTest.java

```
package SolutionForLab3;

/**
 * @author ChenMao
 * @create 2022-12-01 16:29
 */

public class RBTreeTest {

    private static final int a[] = {1,5,6,7,8,9,10,11,12,13,14,15};
    private static final boolean mDebugInsert = false;    // "插入"
动作的检测开关(false，关闭；true，打开)
    private static final boolean mDebugDelete = true;    // "删除"
动作的检测开关(false，关闭；true，打开)
    private static final int deleteList[] = {14,9,5};

    public static void main(String[] args) {
        int i, ilen = a.length;
        RBTree<Integer> tree=new RBTree<Integer>();

        System.out.printf("依次插入值为：");
        for(i=0; i<ilen; i++)
            System.out.printf("%d ", a[i]);
        System.out.printf("的节点后，");

        for(i=0; i<ilen; i++) {
            tree.insert(a[i]);
            // 设置mDebugInsert=true,测试"添加函数"
//            if (mDebugInsert) {
//                System.out.printf("== 添加节点：%d\n", a[i]);
//                System.out.printf("== 树的详细信息：\n");
//                tree.print();
//                System.out.printf("\n");
//            }
        }
```

```
        System.out.printf("中序遍历红黑树各节点: \n");
        tree.inOrder();


        // 设置mDebugDelete=true,测试"删除函数"
        int deleteLen = deleteList.length;
        if (mDebugDelete) {
            for(i=0; i<deleteLen; i++)
            {
                tree.remove(deleteList[i]);
                System.out.printf("删除节点: %d\n", deleteList[i]);
            }

            System.out.printf("\n 中序遍历红黑树各节点: \n");
            tree.inOrder();
        }

        // 销毁二叉树
        tree.clear();
    }
}
```