1. 二叉平衡树

本节目标

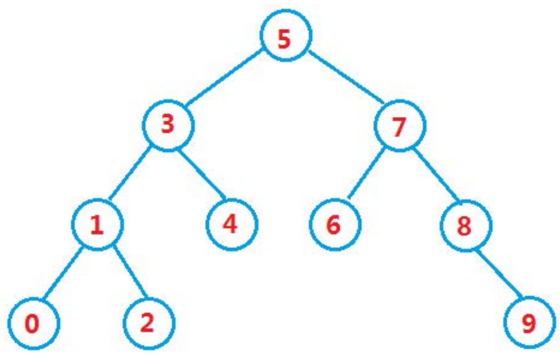
> - 二叉搜索树回顾以及性能分析
> - AVL树
> - 红黑树

1. 二叉搜索树回顾以及性能分析

1.1 二叉搜索树的概念

二叉搜索树又称二叉排序树，它或者是一棵空树，或者是具有以下性质的二叉树:

- **若它的左子树不为空，则左子树上所有节点的值都小于根节点的值**
- **若它的右子树不为空，则右子树上所有节点的值都大于根节点的值**
- **它的左右子树也分别为二叉搜索树**

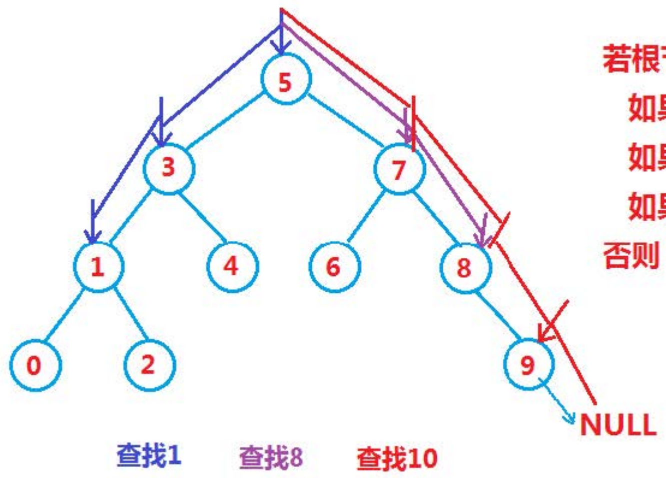


从上述概念以及图中可以看出，二叉搜索树具有以下特性：

1. 二叉搜索树中最左侧的节点是树中最小的节点，最右侧节点一定是树中最大的节点
2. 采用中序遍历遍历二叉搜索树，可以得到一个有序的序列

1.2 二叉搜索树的查找

既然将其称之为二叉搜索树，因此这棵树最主要的作用是进行查询，而且其查询原理特别简单，具体如下：



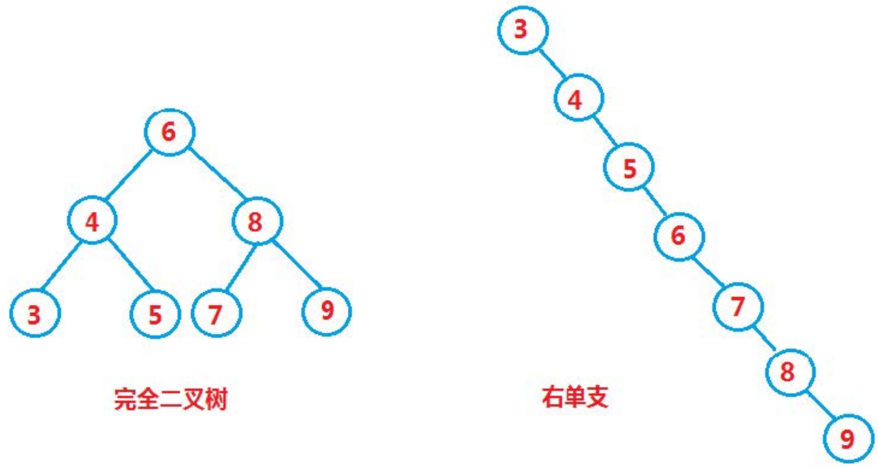
插入和删除操作，也都是建立在查找的基础之上的，那么请同学思考：**二叉搜索树的查找效率是多少呢?**

1.3 二叉树查询性能分析

插入和删除操作都必须先查找，查找效率代表了二叉搜索树中各个操作的性能。

对有n个结点的二叉搜索树，若每个元素查找的概率相等，则二叉搜索树平均查找长度是结点在二叉搜索树的深度的函数，即结点越深，则比较次数越多。

但对于同一个关键码集合，如果各关键码插入的次序不同，可能得到不同结构的二叉搜索树：



最优情况下，二叉搜索树为完全二叉树，其平均比较次数为：$log_2 N$

最差情况下，二叉搜索树退化为单支树，其平均比较次数为：$\frac{N}{2}$

问题：如果退化成单支树，二叉搜索树的性能就失去了。那能否进行改进，不论按照什么次序插入关键码，都可以是二叉搜索树的性能最佳?

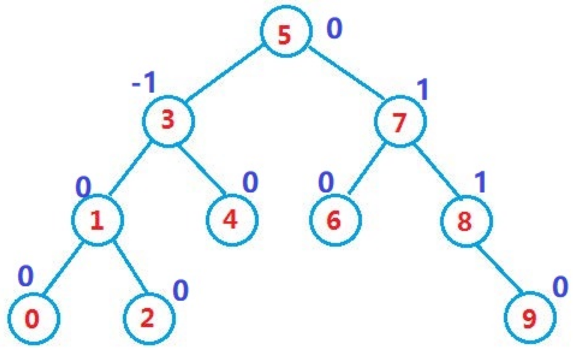
2. AVL树

2.1 AVL树的概念

二叉搜索树虽可以缩短查找的效率，但**如果数据有序或接近有序二叉搜索树将退化为单支树，查找元素相当于在顺序表中搜索元素，效率低下。**因此，两位俄罗斯的数学家G.M.Adelson-Velskii和E.M.Landis在1962年 发明了一种解决上述问题的方法：**当向二叉搜索树中插入新结点后，如果能保证每个结点的左右子树高度之差的绝对值不超过1(需要对树中的结点进行调整)**，即可降低树的高度，从而减少平均搜索长度。

一棵AVL树或者是空树，或者是具有以下性质的二叉搜索树：

- **它的左右子树都是AVL树**
- **左右子树高度之差(简称平衡因子)的绝对值不超过1(-1/0/1)**



如果一棵二叉搜索树是高度平衡的，它就是AVL树。如果它有n个结点，其高度可保持在$O(log_2 n)$，搜索时间复杂度O($log_2 n$)。

2.2 AVL树节点的定义

为了AVL树实现简单，AVL树节点在定义时维护一个平衡因子，具体节点定义如下：

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

    public AVLTreeNode left = null;    // 节点的左孩子
    public AVLTreeNode right = null;   // 节点的右孩子
    public AVLTreeNode parent = null;  // 节点的双亲
    public int val = 0;
    public int bf = 0;    // 当前节点的平衡因子
}
```

2.3 AVL树的插入

AVL树就是在二叉搜索树的基础上引入了平衡因子，因此AVL树也可以看成是二叉搜索树。那么AVL树的插入过程可以分为两步：

1. **按照二叉搜索树的方式插入新节点**
2. **调整节点的平衡因子**

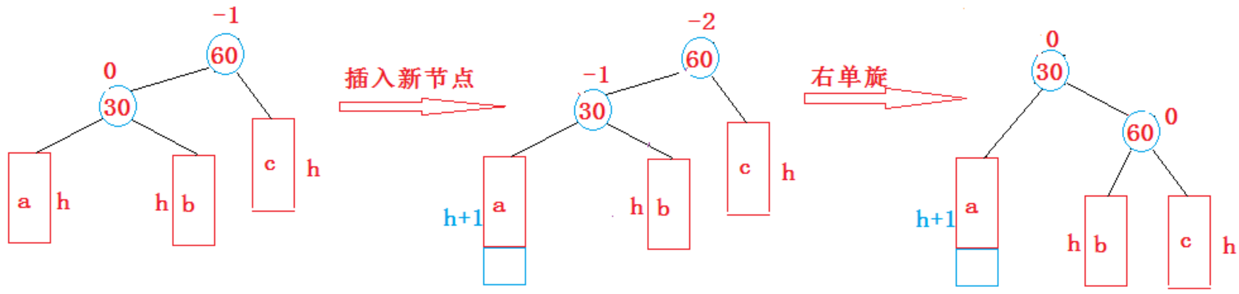
```
boolean insert(int val){
    /* 1. 先按照二叉搜索树的规则将节点插入到AVL树中
       2. 新节点插入后，AVL树的平衡性可能会遭到破坏，此时就需要更新平衡因子，并检测是否破坏了AVL树
```

```
          的平衡性

      pCur插入后，pParent的平衡因子一定需要调整，在插入之前，pParent
      的平衡因子分为三种情况：-1，0，1，分以下两种情况：
       1. 如果pCur插入到pParent的左侧，只需给pParent的平衡因子-1即可
       2. 如果pCur插入到pParent的右侧，只需给pParent的平衡因子+1即可

      此时：pParent的平衡因子可能有三种情况：0，正负1， 正负2
       1. 如果pParent的平衡因子为0，说明插入之前pParent的平衡因子为正负1，插入后被调整成0，此时满足
          AVL树的性质，插入成功
       2. 如果pParent的平衡因子为正负1，说明插入前pParent的平衡因子一定为0，插入后被更新成正负1，此
          时以pParent为根的树的高度增加，需要继续向上更新
       3. 如果pParent的平衡因子为正负2，则pParent的平衡因子违反平衡树的性质，需要对其进行旋转处理
     */

        // cur插入后，parent的平衡因子一定遭到破坏，必须对parent的平衡因子进行调整
        while(null != parent){

            // 更新双亲节点的平衡因子
            if(cur == parent.left)
                parent.bf--;
            else
                parent.bf++;

            if(parent.bf == 0)
                break;
            else if(parent.bf == -1 || parent.bf == 1) {
                cur = parent;
                parent = cur.parent;
            }
            else {
                // parent节点的平衡因子为2，违反了AVL树的性质
                // 此时需要对以parent为根的二叉树进行旋转处理
                if(2 == parent.bf) {
                    // parent的平衡因子为2，说明parent的右子树比较高，最终需要左旋
                    // ......
                }
                else{
                    // parent的平衡因子为2，说明parent的右子树比较高，最终需要左旋
                    // ......
                }

                // 旋转完成之后，以parent为根的树已经和插入之前的高度相同，不会再对上层树的平衡性造
成影响
                break;
            }
        }

        return true;
    }
```

###2.4 AVL树的旋转

如果在一棵原本是平衡的AVL树中插入一个新节点，可能造成不平衡，此时必须调整树的结构，使之平衡化。根据节点插入位置的不同，AVL树的旋转分为四种：

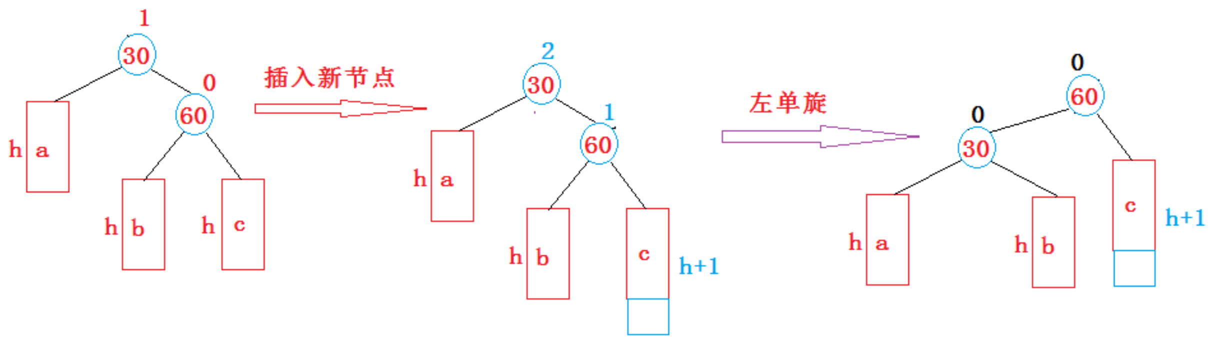
1. **新节点插入较高左子树的左侧---左左：右单旋**




```
/*
  上图在插入前，AVL树是平衡的，新节点插入到30的左子树(注意：此处不是左孩子)中，30左子树增加
  了一层，导致以60为根的二叉树不平衡，要让60平衡，只能将60左子树的高度减少一层，右子树增加一层，
  即将左子树往上提，这样60转下来，因为60比30大，只能将其放在30的右子树，而如果30有右子树，右子树
根的值一定大于30，小于60，只能将其放在60的左子树，旋转完成后，更新节点的平衡因子即可。在旋转过程
中，有以下几种情况需要考虑：
  1. 30节点的右孩子可能存在，也可能不存在
  2. 60可能是根节点，也可能是子树
     如果是根节点，旋转完成后，要更新根节点
     如果是子树，可能是某个节点的左子树，也可能是右子树

 同学们再此处可举一些详细的例子进行画图，考虑各种情况，加深旋转的理解
*/

 // 左单旋
 private void rotateLeft(AVLTreeNode parent){
    // 注意这几个特殊孩子节点的命名
    // subR为双亲的右孩子
    AVLTreeNode subR = parent.right;

    // 为subR的左孩子
    AVLTreeNode subRL = subR.left;

    // 节点的孩子域的指向只需要改变两个：结合图解
    // 1. 旋转完成后subRL成为parent的右孩子
    parent.right = subRL;
    if(null != subRL)
        subRL.parent = parent;

    // 2. 旋转完成之后，parent成为subR的左孩子
    subR.left = parent;

    // 更新parent和subR的双亲
    AVLTreeNode pparent = parent.parent;
    parent.parent = subR;
    subR.parent = pparent;

    // 更新原parent的上层
    // 1. 旋转前，parent可能是根节点
```

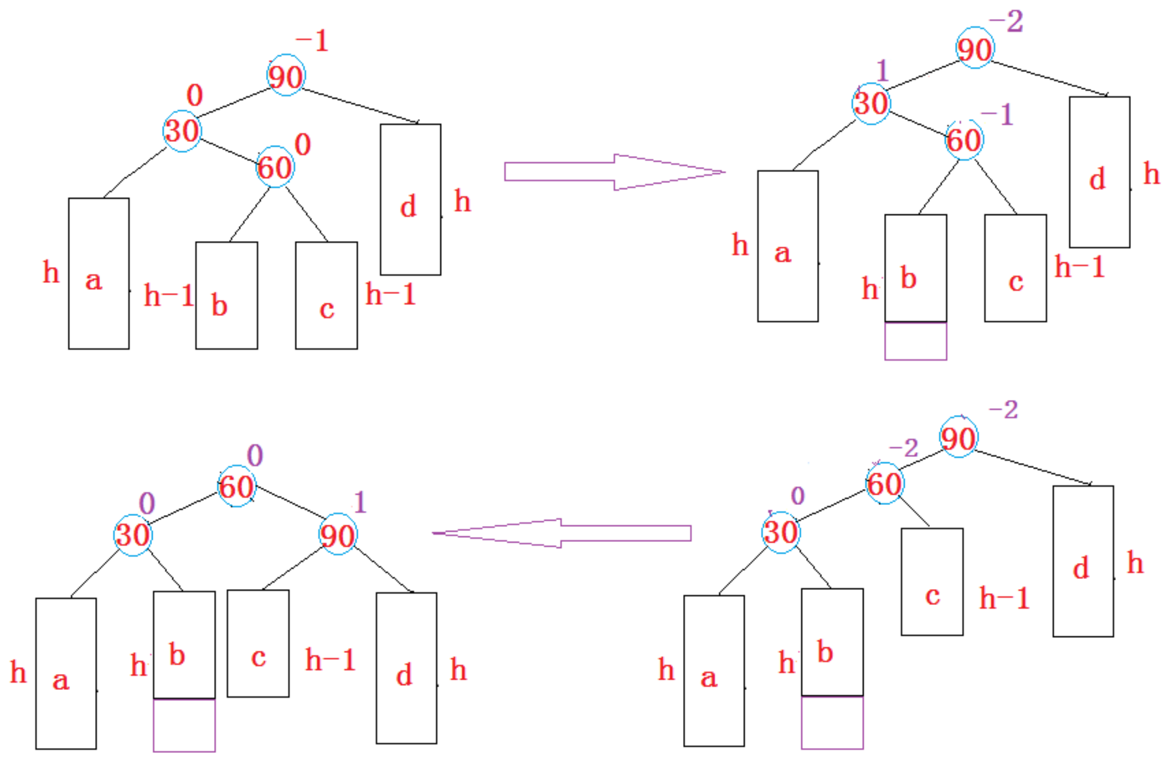
```
    // 2. 旋转前，parent可能是一棵子树，既然是子树，那parent可能是某个节点的左子树也可能是右子树
    if(null == pparent)
        root = subR;
    else{
        if(pparent.left == parent)
            pparent.left = subR;
        else
            pparent.right = subR;
    }

    // 旋转完成后，parent和subR节点的平衡因子已经是0
    parent.bf = subR.bf = 0;
}
```

2. **新节点插入较高右子树的右侧---右右：左单旋**



左单旋的实现，学生们可以参考右单旋的实现。

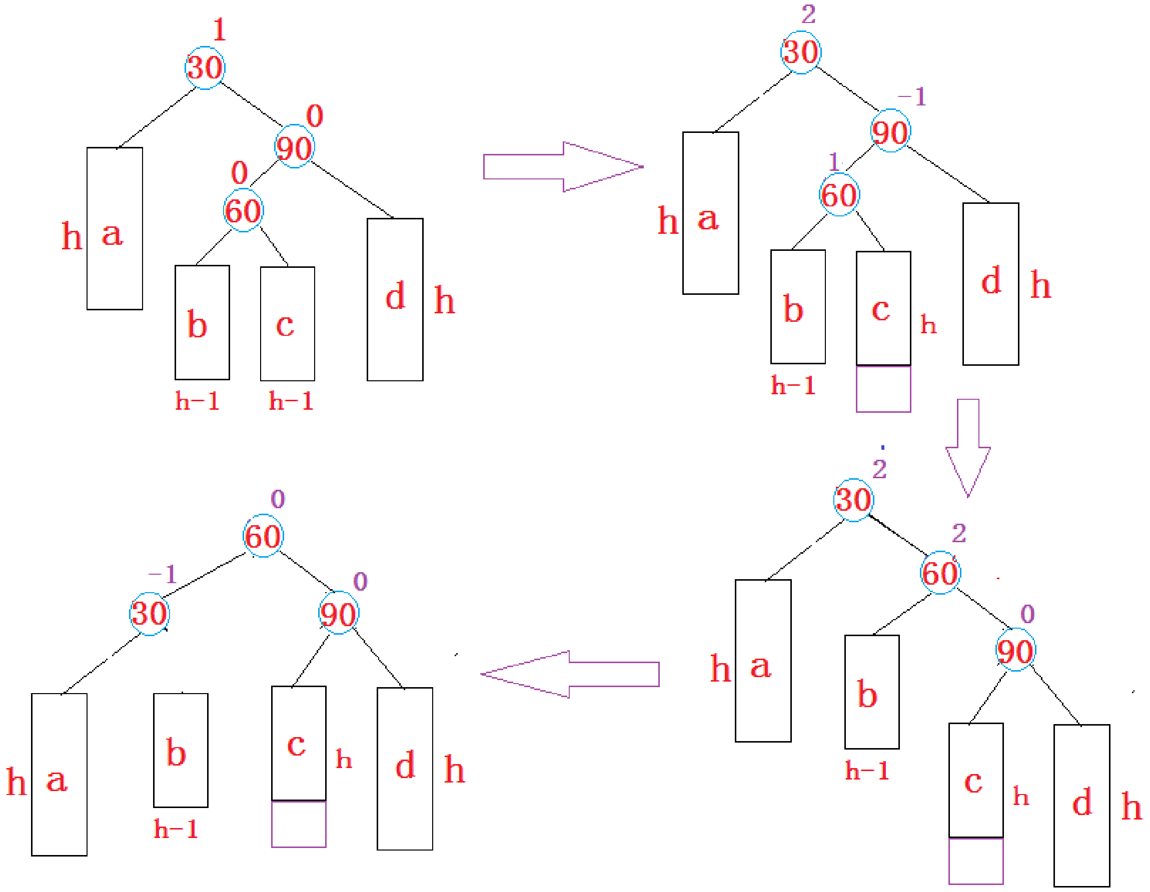
3. **新节点插入较高左子树的右侧---左右：先左单旋再右单旋**



将双旋变成单旋后再旋转，即：**先对30进行左单旋，然后再对90进行右单旋**，旋转完成后再考虑平衡因子的更新。

```
// 先左单旋再右单旋
// 旋转之前，60的平衡因子可能是-1/0/1，旋转完成之后，根据情况对其他节点的平衡因子进行调整
private void rotateLR(AVLTreeNode parent){
    AVLTreeNode subL = parent.left;
    AVLTreeNode subLR = subL.right;

    // 旋转之前，保存subLR的平衡因子，旋转完成之后，需要根据该平衡因子来调整其他节点的平衡因子
    int bf = subLR.bf;

    rotateLeft(parent.left);
    rotateRight(parent);

    if(1 == bf)
        subL.bf = -1;
    else if(-1 == bf)
        parent.bf = 1;
}
```

4. **新节点插入较高右子树的左侧---右左：先右单旋再左单旋**



右左双旋的实现，学生们可以参考左右双旋。

总结：

新节点插入后，假设以pParent为根的子树不平衡，即**pParent的平衡因子为2或者-2**，分以下情况考虑

1. pParent的平衡因子为2，说明pParent的右子树高，设pParent的右子树的根为pSubR
    - 当pSubR的平衡因子为1时，执行左单旋
    - 当pSubR的平衡因子为-1时，执行右左双旋
2. pParent的平衡因子为-2，说明pParent的左子树高，设pParent的左子树的根为pSubL
    - 当pSubL的平衡因子为-1是，执行右单旋
    - 当pSubL的平衡因子为1时，执行左右双旋

即：pParent与其较高子树节点的平衡因子时同号时单旋转，异号时双旋转。

旋转完成后，原pParent为根的子树个高度降低，已经平衡，不需要再向上更新。

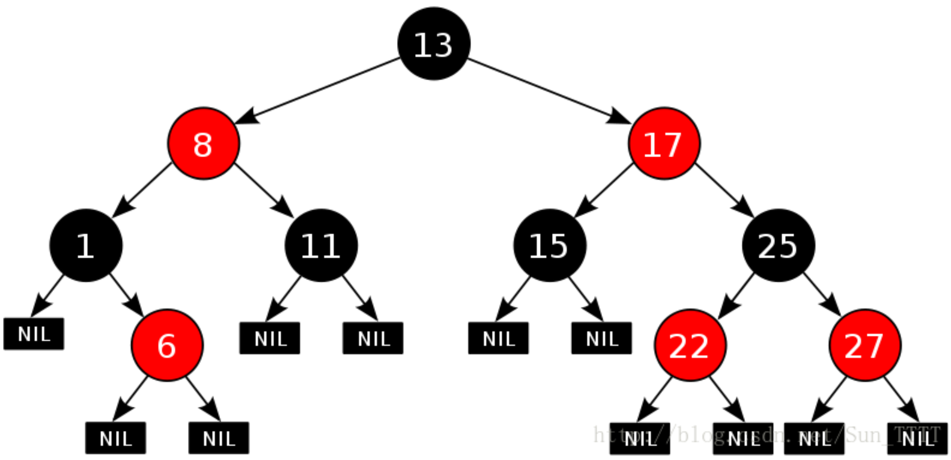
2.5 AVL树性能分析

AVL树是一棵绝对平衡的二叉搜索树，其要求每个节点的左右子树高度差的绝对值都不超过1，这样可以保证查询时高效的时间复杂度，即$log_2 (N)$。但是如果要对AVL树做一些结构修改的操作，性能非常低下，比如：插入时要维护其绝对平衡，旋转的次数比较多，更差的是在删除时，有可能一直要让旋转持续到根的位置。因此：如果需要一种查询高效且有序的数据结构，而且数据的个数为静态的(即不会改变)，可以考虑AVL树，但一个结构经常修改，就不太适合。

3. 红黑树

3.1 红黑树概念

红黑树，是一种**二叉搜索树**，但**在每个结点上增加一个存储位表示结点的颜色，可以是Red或Black**。 通过对**任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出俩倍**，因而是**接近平衡**的。



3.2 红黑树的性质

1. **每个结点不是红色就是黑色**
2. **根节点是黑色的**
3. **如果一个节点是红色的，则它的两个孩子结点是黑色的**
4. **对于每个结点，从该结点到其所有后代叶结点的简单路径上，均 包含相同数目的黑色结点**
5. **每个叶子结点都是黑色的(此处的叶子结点指的是空结点)**

思考：为什么满足上面的性质，红黑树就能保证：其最长路径中节点个数不会超过最短路径节点个数的两倍？

3.3 红黑树节点的定义

```
class RBTreeNode{
    RBTreeNode left = null;
    RBTreeNode right = null;
    RBTreeNode parent = null;
    COLOR color = RED;    // 节点的颜色
    int val;

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

思考：在节点的定义中，为什么要将节点的默认颜色给成红色的？

3.4 红黑树的插入

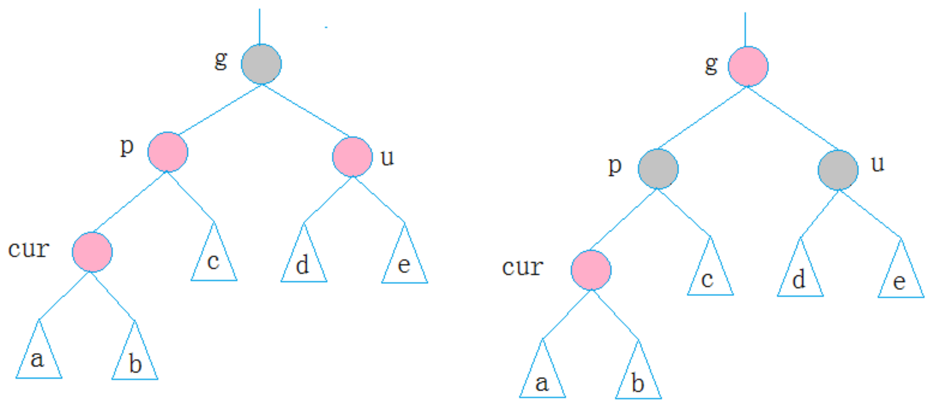
红黑树是在二叉搜索树的基础上加上其平衡限制条件，因此红黑树的插入可分为两步：

1. **按照二叉搜索的树规则插入新节点**
2. **检测新节点插入后，红黑树的性质是否造到破坏**

   因为**新节点的默认颜色是红色**，因此：如果**其双亲节点的颜色是黑色，没有违反红黑树任何性质**，则不需要调整；但**当新插入节点的双亲节点颜色为红色时，就违反了性质三不能有连在一起的红色节点**，此时需要对红黑树分情况来讨论：

   约定:cur为当前节点，p为父节点，g为祖父节点，u为叔叔节点

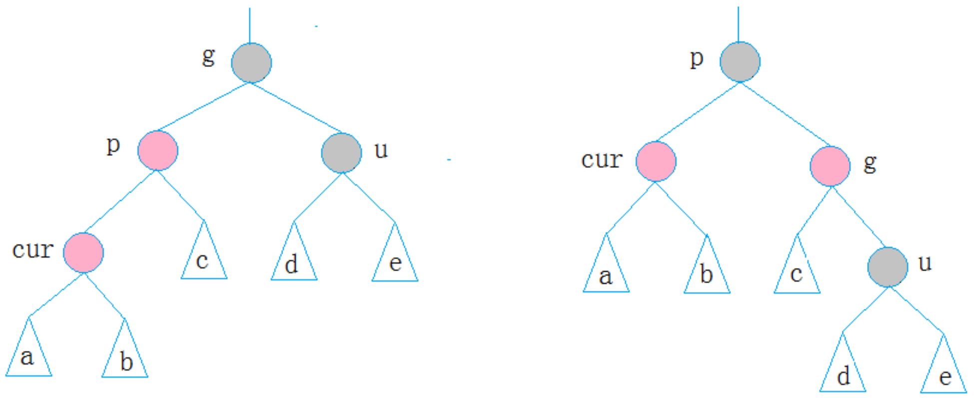
   - 情况一: **cur为红，p为红，g为黑，u存在且为红**

     cur和p均为红，违反了性质三，此处能否将p直接改为黑？

     解决方式：将p,u改为黑，g改为红，然后把g当成cur，继续向上调整。

   - 情况二: **cur为红，p为红，g为黑，u不存在/u为黑**

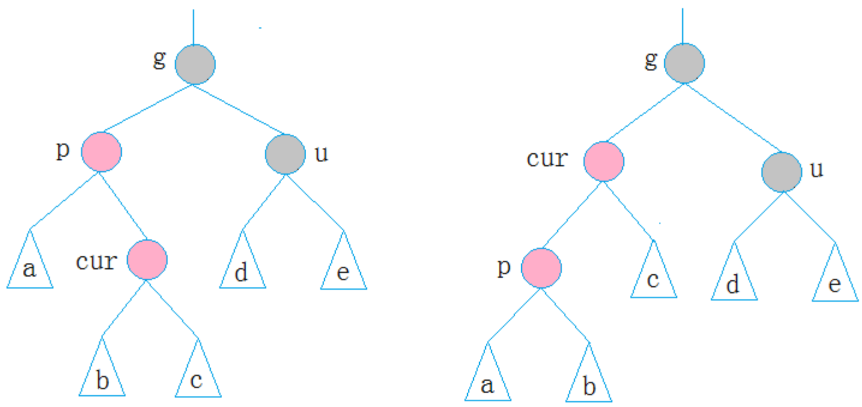


p为g的左孩子，cur为p的左孩子，则进行右单旋转；相反，

p为g的右孩子，cur为p的右孩子，则进行左单旋转

p、g变色--p变黑，g变红

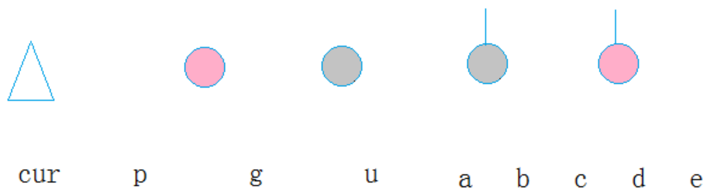
- 情况三: **cur为红，p为红，g为黑，u不存在/u为黑**



p为g的左孩子，cur为p的右孩子，则针对p做左单旋转；相反，

p为g的右孩子，cur为p的左孩子，则针对p做右单旋转

则转换成了情况2



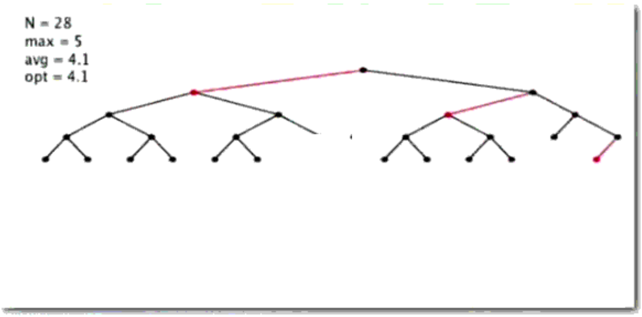
针对每种情况进行相应的处理即可。

```
public boolean insert(int val){
        // ...
        // 新节点插入后，如果parent节点的颜色是红色，一定违反性质三
        while(null != parent && COLOR.RED == parent.color){
            RBTreeNode grandFather = parent.parent;
            if(parent == grandFather.left){
                RBTreeNode uncle = grandFather.right;
                if(null != uncle && uncle.color == COLOR.RED){
                    // 情况一：叔叔节点存在且为红，
                    // 解决方式：将叔叔和双亲节点改为黑色，祖父节点改为红色
```

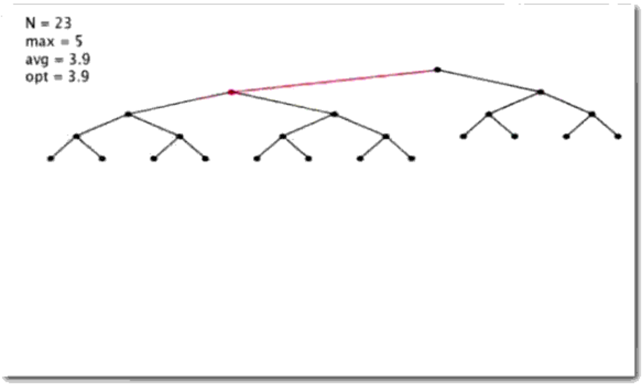
```
                // 如果祖父的双亲节点的颜色是红色，需要继续往上调整
                parent.color = COLOR.BLACK;
                uncle.color = COLOR.BLACK;
                grandFather.color = COLOR.RED;
                cur = grandFather;
                parent = cur.parent;
            }
            else
            {
                // 情况二和情况三
                // 叔叔节点不存在 || 叔叔节点存在，但是颜色是黑色
                if(cur == parent.right)
                {
                    // 情况三
                    rotateLeft(parent);
                    RBTreeNode temp = parent;
                    parent = cur;
                    cur = temp;
                }

                // 情况二
                parent.color = COLOR.BLACK;
                grandFather.color = COLOR.RED;
                rotateRight(grandFather);
            }
        }
        else{
            // 课件图解的反情况，即叔叔节点在左侧
            // 此处，请同学们自行处理
        }
    }

    // 在上述循环更新期间，可能会将根节点给成红色而违反性质1，因此此处必须将根节点改为黑色
    root.color = COLOR.BLACK;
    return true;
}
```

动态演示效果：

- **以升序(降序)插入构建红黑树**

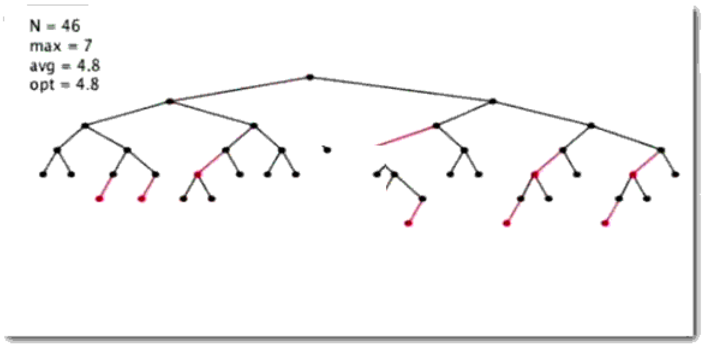


255 insertions in ascending order



255 insertions in descending order

- **随机插入构建红黑树**



255 random insertions

3.4 红黑树验证

红黑树的检测分为两步：

1. **检测其是否满足二叉搜索树(中序遍历是否为有序序列)**
2. **检测其是否满足红黑树的性质**

```
public boolean isValidRBTree()
    {
        // 空树也是红黑树
        if(null == root)
            return  true;

        if(root.color != COLOR.BLACK) {
            System.out.println("违反了性质1: 根节点不是黑色");
            return false;
        }
```

```java
        // 获取单条路径中节点的个数
        int blackCount = 0;
        RBTreeNode cur = root;
        while(null != cur){
            if(cur.color == COLOR.BLACK)
                blackCount++;

            cur = cur.left;
        }

        // 具体的检验方式
        return _isValidRBtree(root, 0, blackCount);
    }

    private boolean _isValidRBtree(RBTreeNode root, int pathCount, int blackCount){
        if(null == root)
            return true;

        // 遇到一个黑色节点，统计当前路径中黑色节点个数
        if(root.color == COLOR.BLACK)
            pathCount++;

        // 验证性质三
        RBTreeNode parent = root.parent;
        if(parent != null && parent.color == COLOR.RED && root.color == COLOR.RED){
            System.out.println("违反了性质3：有连在一起的红色节点");
            return true;
        }

        // 验证性质四
        // 如果是叶子节点，则一条路径已经走到底，检验该条路径中黑色节点总个数是否与先前统计的结果相同
        if(root.left == null && root.right == null){
            if(pathCount != blackCount){
                System.out.println("违反了性质4：路径中黑色节点格式不一致");
                return false;
            }
        }

        // 以递归的方式检测root的左右子树
        return _isValidRBtree(root.left, pathCount, blackCount) &&
                _isValidRBtree(root.right, pathCount, blackCount);
    }
```

3.5 红黑树的删除

红黑树的删除本节不做讲解，有兴趣的同学可参考：《算法导论》或者《STL源码剖析》

http://www.cnblogs.com/fornever/archive/2011/12/02/2270692.html

<http://blog.csdn.net/chenhuajie123/article/details/11951777

3.6 AVL树和红黑树的比较

红黑树和AVL树都是高效的平衡二叉树，增删改查的时间复杂度都是O($log_2 N$)，**红黑树不追求绝对平衡，其只需保证最长路径不超过最短路径的2倍，相对而言，降低了插入和旋转的次数，所以在经常进行增删的结构中性能比AVL树更优，而且红黑树实现比较简单，所以实际运用中红黑树更多。**

3.7 红黑树应用

1. java集合框架中的：**TreeMap、TreeSet底层使用的就是红黑树**
2. C++ STL库 -- map/set、mutil_map/mutil_set
3. linux内核：进程调度中使用红黑树管理进程控制块，epoll在内核中实现时使用红黑树管理事件块
4. 其他一些库：比如nginx中用红黑树管理timer等

http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html