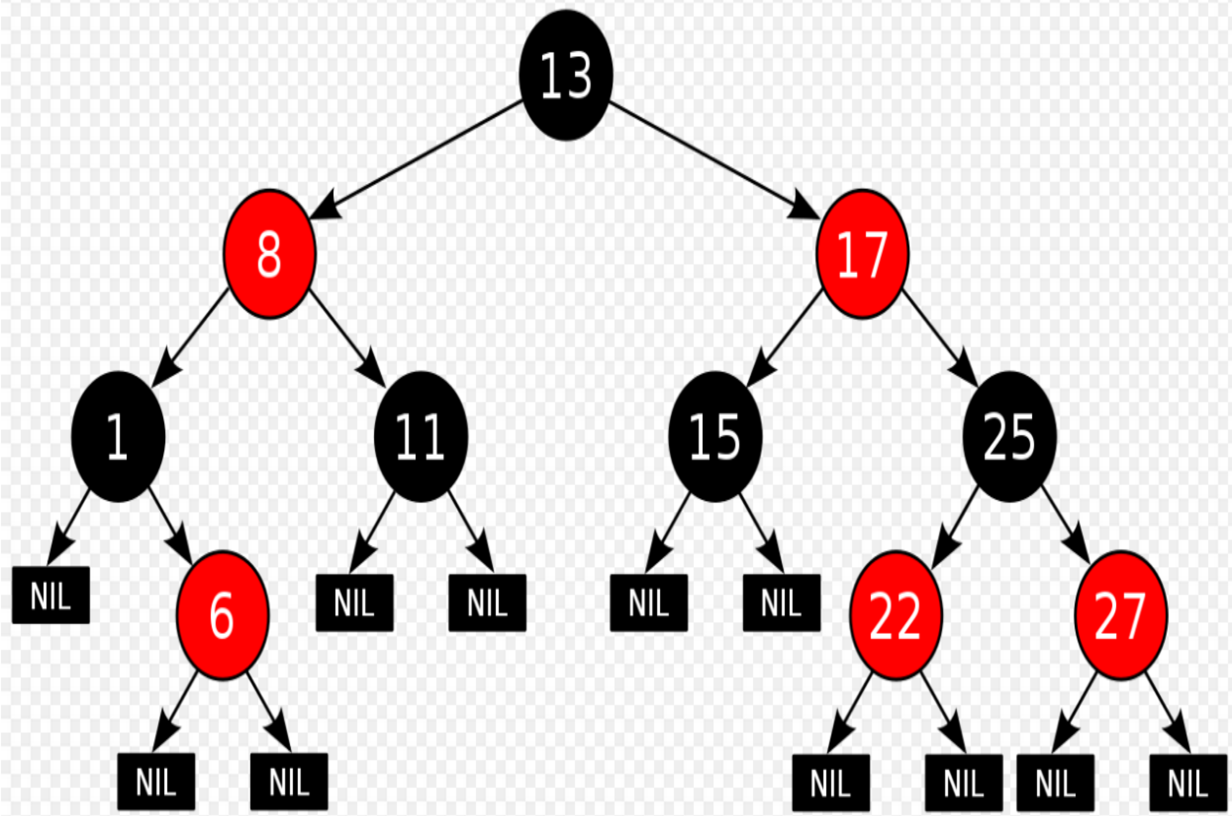
哈爾濱工業大學（威海）
HARBIN INSTITUTE OF TECHNOLOGY AT WEIHAI
规格严格 · 功夫到家



算法设计与分析

红黑树

主讲： 朱东杰 博士、硕导
地点： M楼305
电话/微信： 18953856806
Email： zhudongjie@hit.edu.cn

当在10亿数据中只需要进行10几次比较就能查找到目标时，不禁感叹编程之魅力！人类之伟大呀！ —— 学红黑树有感。

查找的基本概念

查找: 查询关键字是否在(数据元素集合）表中的过程。也称作检索。

主关键字: 能够惟一区分各个不同数据元素的关键字

次关键字: 通常不能惟一区分各个不同数据元素的关键字

查找成功: 在数据元素集合中找到了要查找的数据元素

查找不成功: 在数据元素集合中没有找到要查找的数据元素

静态查找: 只查找，不改变数据元素集合内的数据元素

动态查找: 既查找，又改变（增减）集合内的数据元素

静态查找表: 静态查找时构造的存储结构

动态查找表: 动态查找时构造的存储结构

平均查找长度: 查找过程所需进行的关键字比较次数的平均值，是衡量查找算法效率的最主要标准，

红黑树

1、静态搜索结构

静态搜索是指搜索结构在执行插入和删除操作的前后不发生变化。
静态搜索表：基于数组的数据表类。

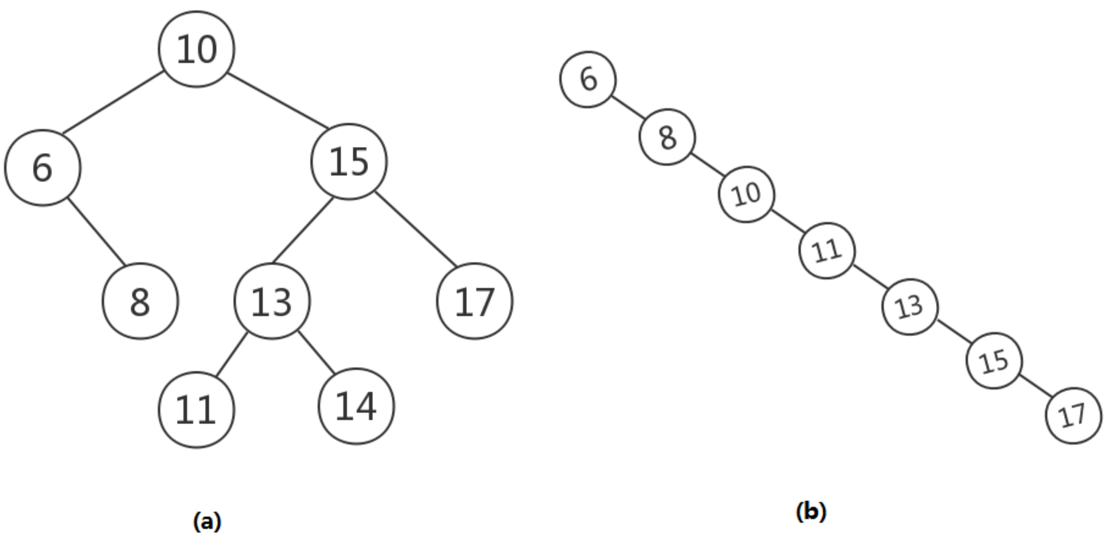
(1)顺序搜索
主要用于线性结构中的搜索。
优点：对表的特性没有特殊的饿要求。
缺点：搜索效率较低，特别当n比较大时效率低。
对于线性链表只能用顺序搜索。

(2)折半搜索
(3)基于有序顺序表的斐波那契搜索。
按某个斐波那契数确定“中间点”的位置

红黑树——二叉查找树

二叉查找树，也称二叉搜索树，或二叉排序树要么是一颗空树，要么就是具有如下性质的二叉树：

- 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 任意节点的左、右子树也分别为二叉查找树；
- 没有键值相等的节点。



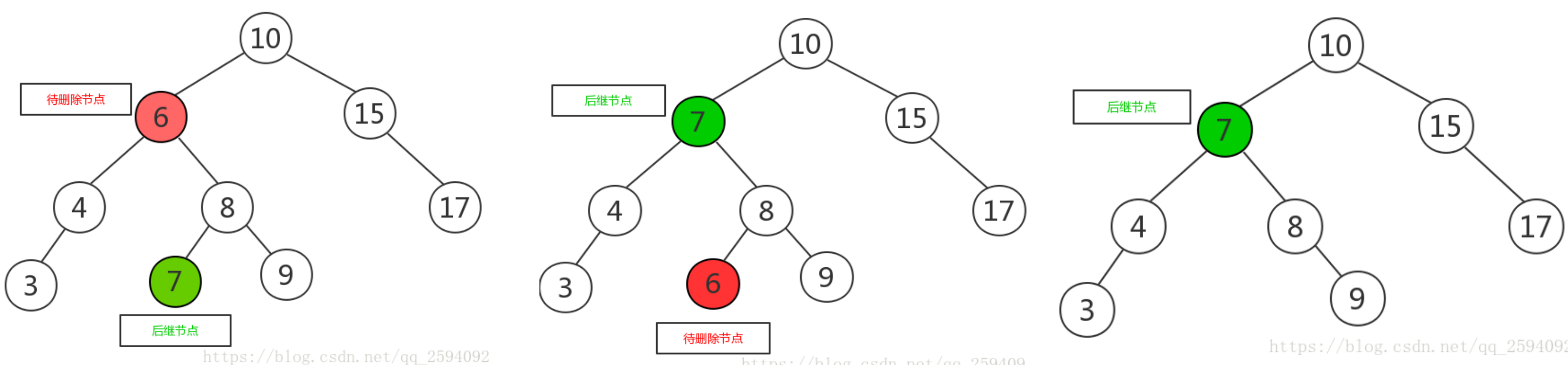
不同形态的二叉查找树

二叉查找树是对要查找的数据进行生成树，左支的值小于右支的值。在查找的时候也是一样的思路，从根节点开始，比节点大进入右支，比节点小进入左支，直到查找到目标值

红黑树——二叉查找树

二叉查找树的插入算法：空树，就首先生成根节点；不是空树就按照查找的算法，找到父节点，然后作为叶子节点插入，如果值已经存在就插入失败。

删除操作稍微复杂一点，有如下几种情况：
（1）如果删除的是叶节点，可以直接删除；
（2）如果被删除的元素有一个子节点，可以将子节点直接移到被删除元素的位置；
（3）如果有两个子节点，这时候就采用中序遍历，找到待删除的节点的后继节点，将其与待删除的节点互换，此时待删除节点的位置已经是叶子节点，可以直接删除

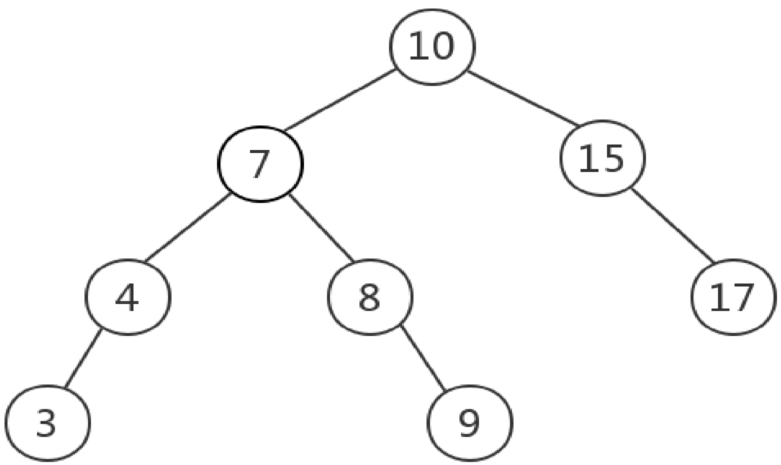


二叉查找树还有一个性质，即**对二叉查找树进行中序遍历，即可得到有序的数列**

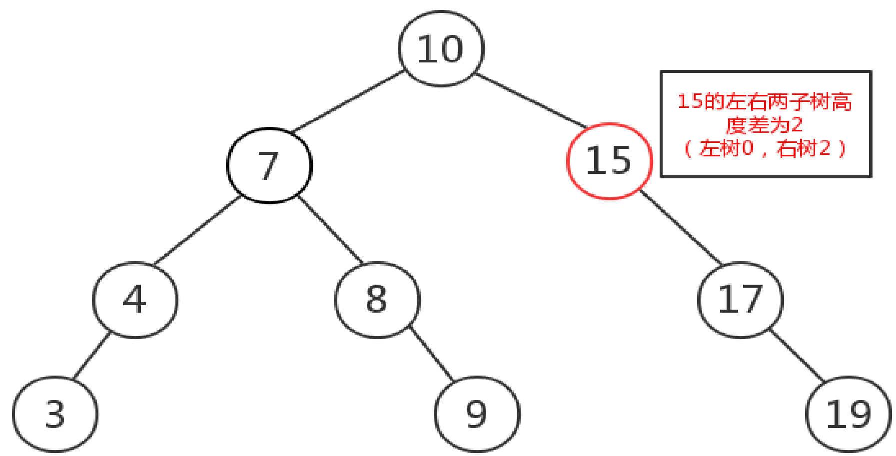
插入和查找的时间复杂度均为 O(logn)，但是在最坏的情况下仍然会有 O(n)

红黑树——平衡二叉树

由于普通的二叉查找树会容易失去“平衡”，极端情况下，二叉查找树会退化成线性的链表，导致插入和查找的复杂度下降到 O(n)，所以，这也是平衡二叉树设计的初衷。那么平衡二叉树如何保持”平衡“呢？根据定义，有两个重点，一是左右两子树的高度差的绝对值不能超过1，二是左右两子树也是一颗平衡二叉树。



平衡二叉树

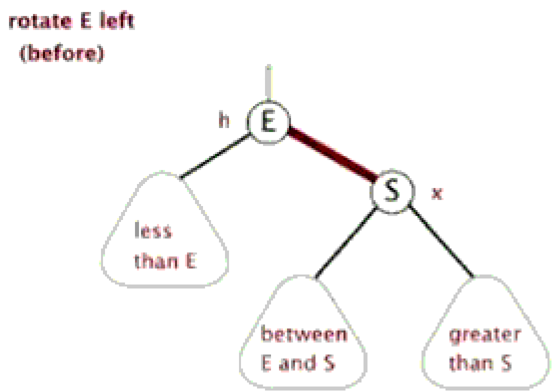


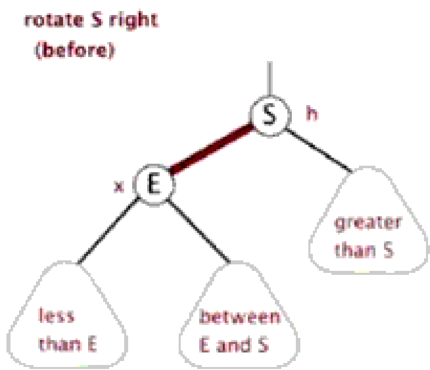
不是平衡二叉树

平衡二叉搜索树，又被称为AVL树，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树

红黑树——平衡二叉树

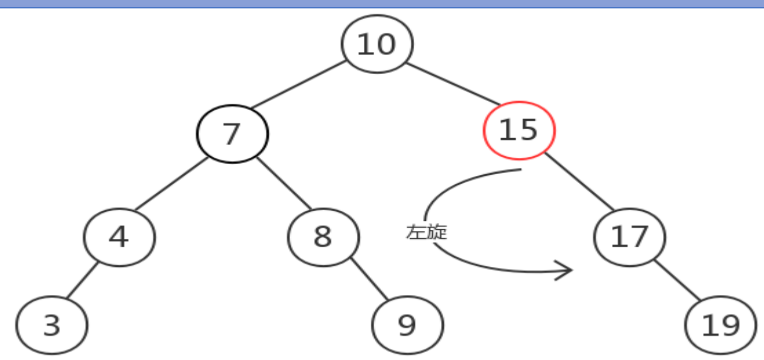
左旋



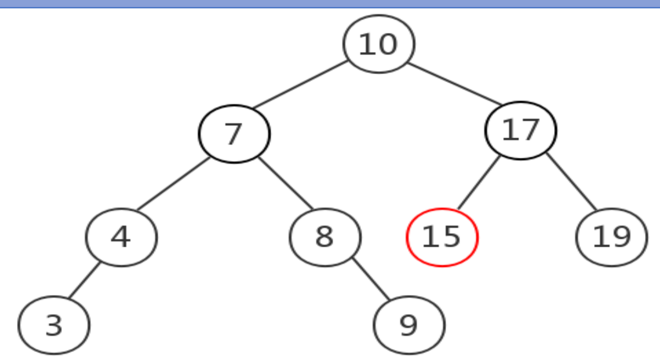


右旋

- 左旋就是将节点的右支往左拉，右子节点变成父节点，并把晋升之后多余的左子节点出让给降级节点的右子节点；
- 而右旋就是反过来，将节点的左支往右拉，左子节点变成了父节点，并把晋升之后多余的右子节点出让给降级节点的左子节点。
- 即左旋就是往左变换，右旋就是往右变换。不管是左旋还是右旋，旋转的目的都是将节点多的一支出让节点给另一个节点少的一支。



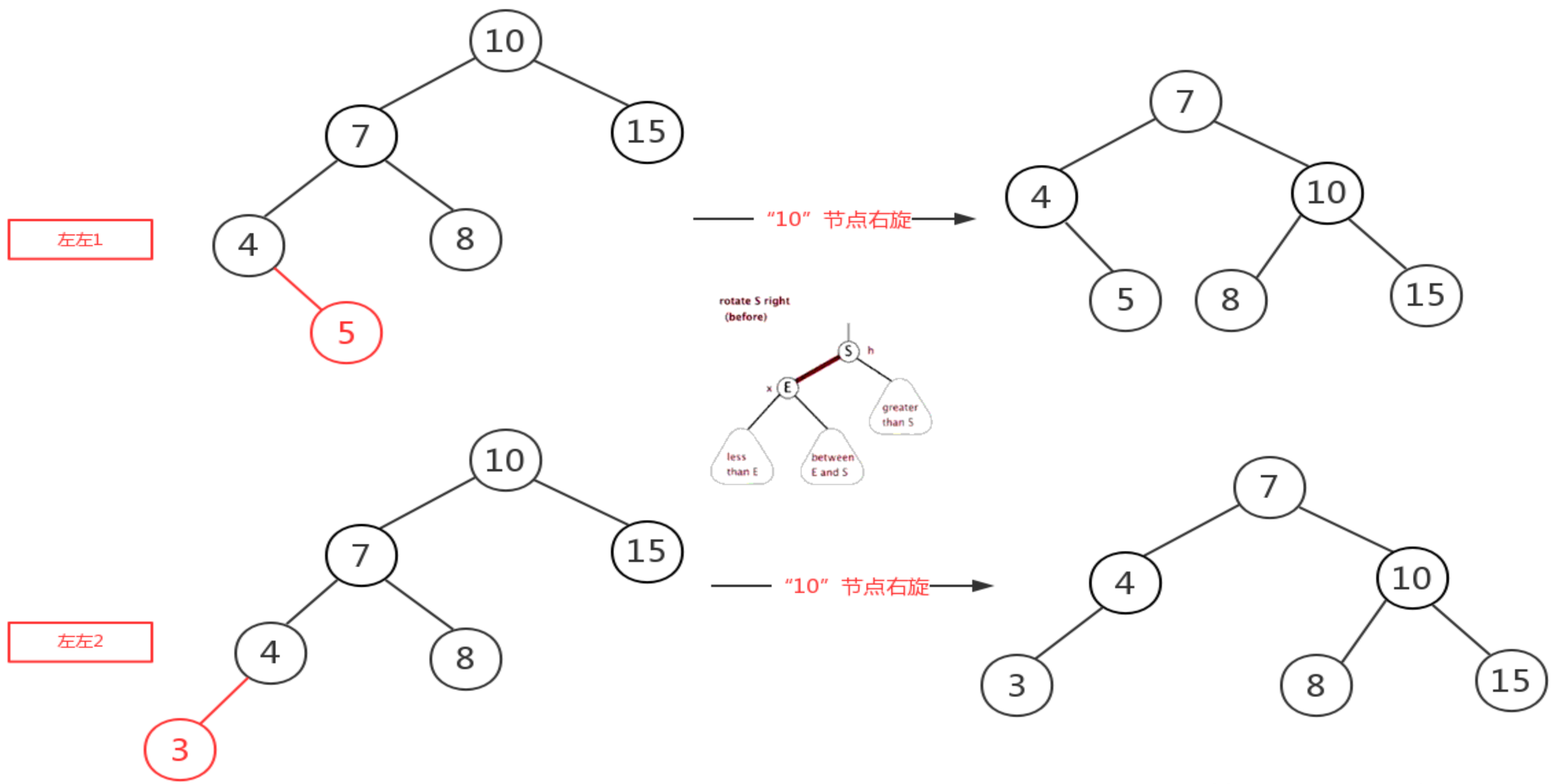
不是平衡二叉树



平衡二叉树

红黑树——平衡二叉树

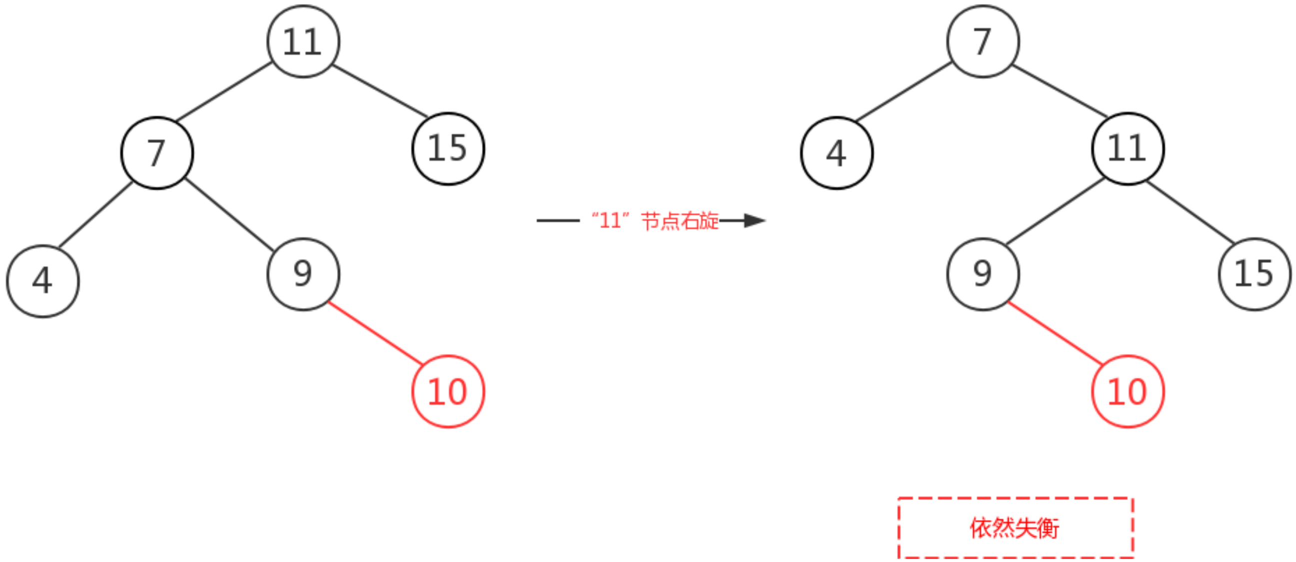
由于在构建平衡二叉树的时候，当有**新节点插入**时，都会判断插入后时候平衡，这说明了插入新节点前，都是平衡的，也即高度差绝对值不会超过1。当新节点插入后，有可能会有导致树不平衡，这时候就需要进行调整，而可能出现的情况就有4种，分别称作**左左，左右，右左，右右。**

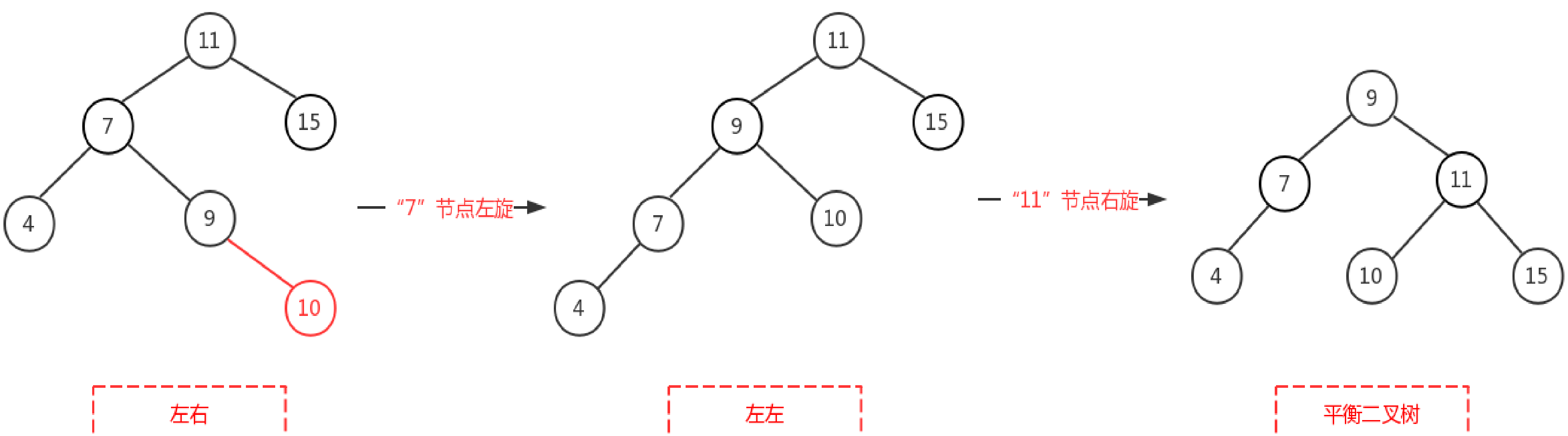


https://blog.csdn.net/qq_25940921

红黑树——平衡二叉树

左右：

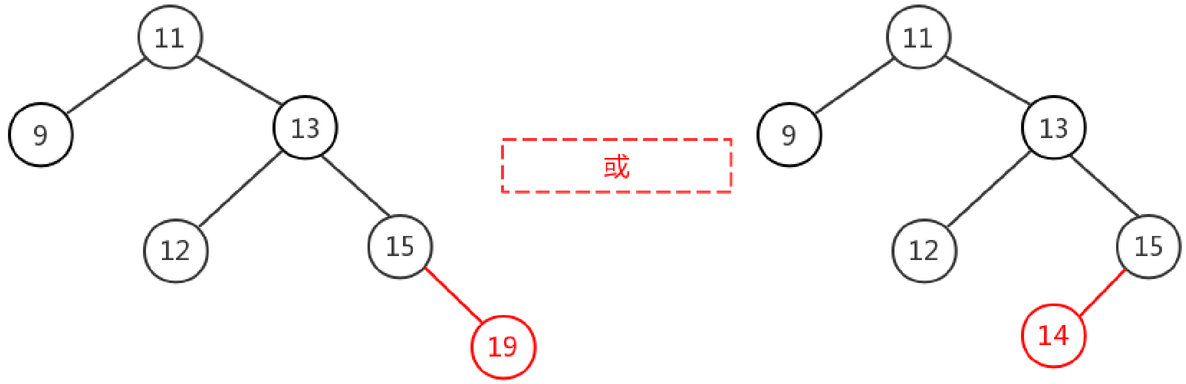


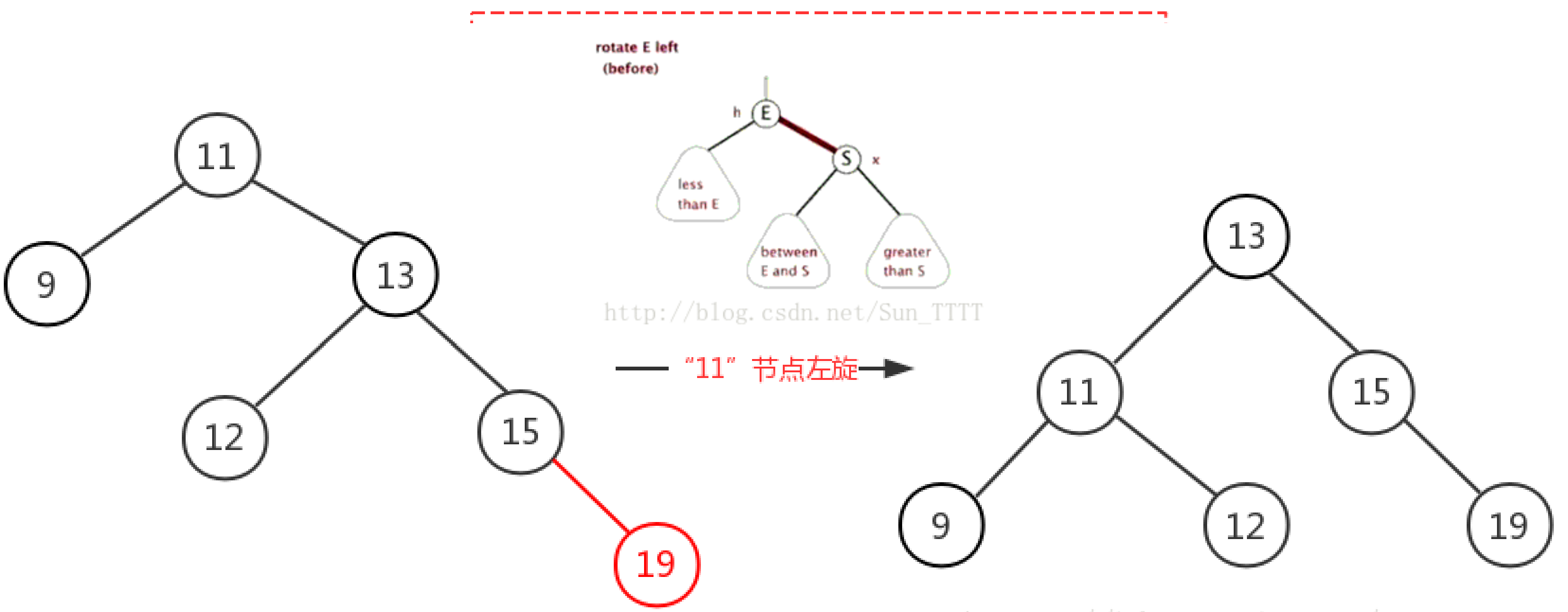


https://blog.csdn.net/qq_25940921

红黑树——平衡二叉树

右右：

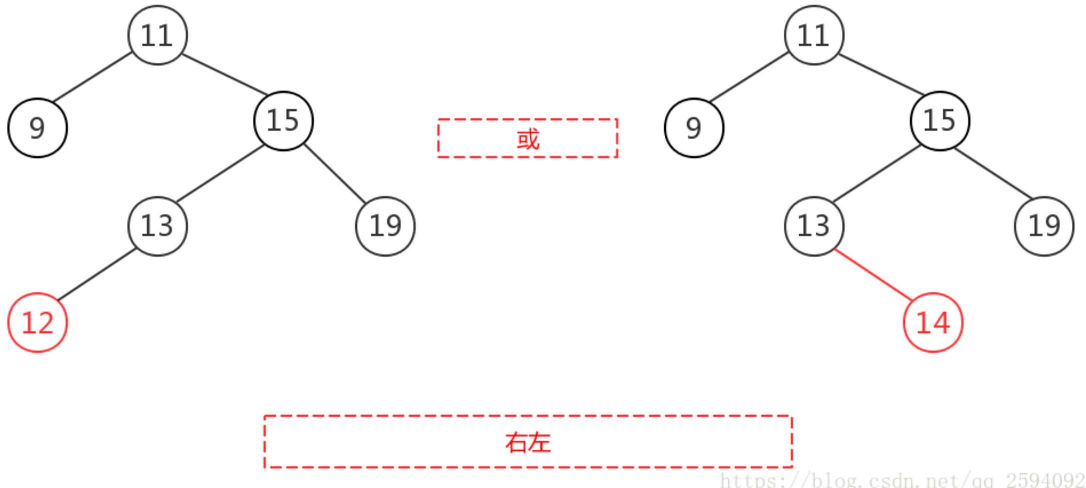


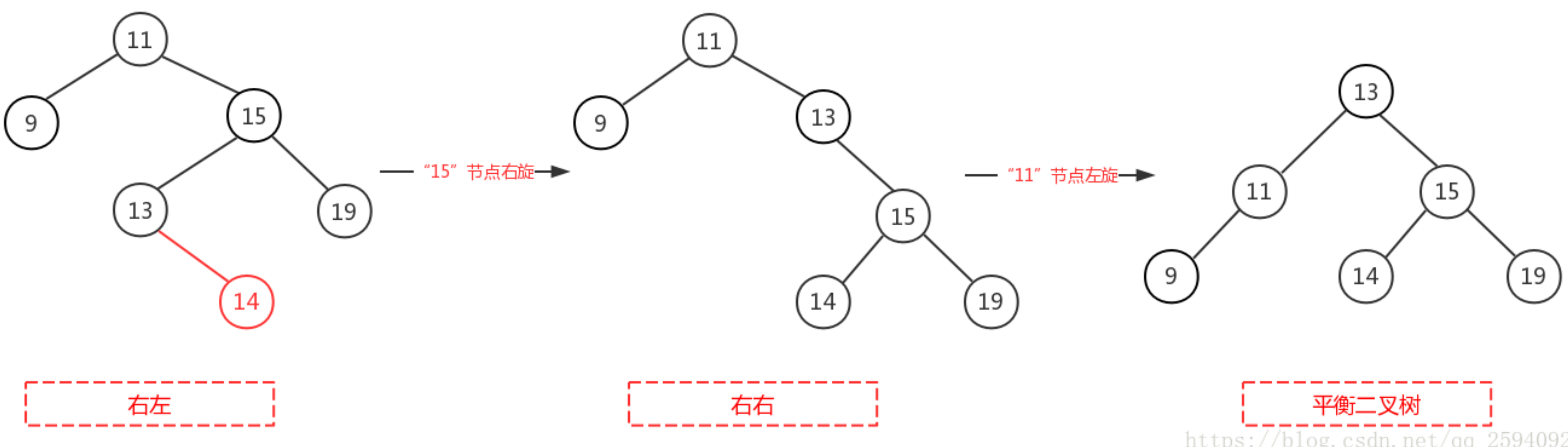


https://blog.csdn.net/qq_25940921

红黑树——平衡二叉树

右左：


11
9
15
13
19
12
或
11
9
15
13
19
14
右左
https://blog.csdn.net/qq_25940921


11
9
15
13
19
14
"15" 节点右旋
11
9
13
15
14
19
"11" 节点左旋
13
11
15
9
14
19
右左
右右
平衡二叉树
https://blog.csdn.net/qq_25940921

红黑树——平衡二叉树

平衡二叉树节点的删除

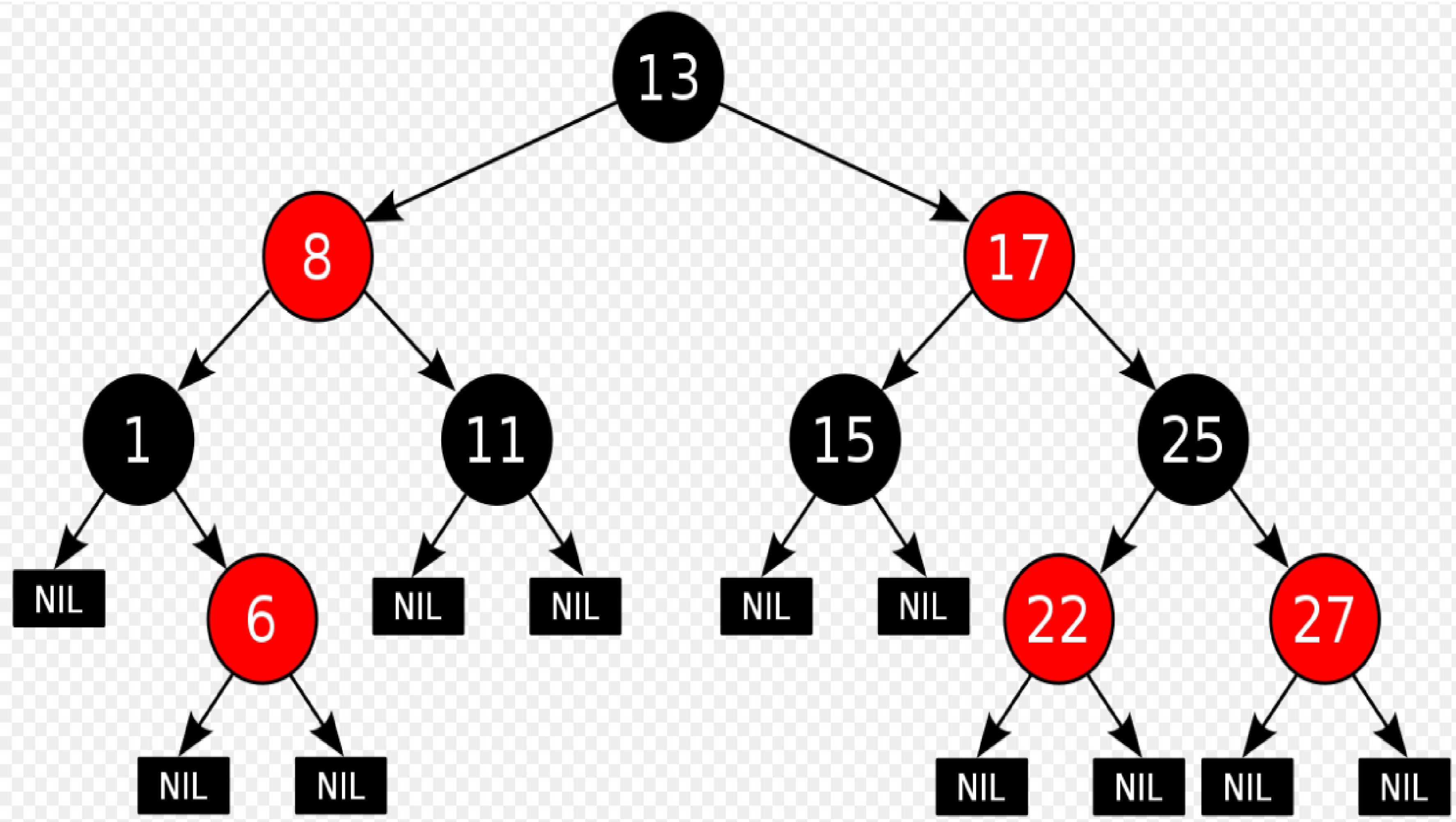
（1）当删除的节点是叶子节点，则将节点删除，然后从父节点开始，判断是否失衡，如果没有失衡，则再判断父节点的父节点是否失衡，直到根节点，此时到根节点还发现没有失衡，则说此时树是平衡的；如果中间过程发现失衡，则判断属于哪种类型的失衡（左左，左右，右左，右右），然后进行调整。
（2）删除的节点只有左子树或只有右子树，这种情况其实就比删除叶子节点的步骤多一步，就是将节点删除，然后把仅有一支的左子树或右子树替代原有结点的位置，后面的步骤就一样了，从父节点开始，判断是否失衡，如果没有失衡，则再判断父节点的父节点是否失衡，直到根节点，如果中间过程发现失衡，则根据失衡的类型进行调整。
（3）删除的节点既有左子树又有右子树，这种情况又比上面这种多一步，就是中序遍历，找到待删除节点的前驱或者后驱都行，然后与待删除节点互换位置，然后把待删除的节点删掉，后面的步骤也是一样，判断是否失衡，然后根据失衡类型进行调整。

总结：平衡二叉树是一棵高度平衡的二叉树，所以查询的时间复杂度是 O(logN) 。
插入失衡的情况有4种，左左，左右，右左，右右，即一旦插入新节点导致失衡需要调整，最多也只要旋转2次，所以，插入复杂度是 O(1)
但是平衡二叉树也不是完美的，从上面删除处理思路中也可以看到，就是删除节点时有可能因为失衡，导致需要从删除节点的父节点开始，不断的回溯到根节点，如果这棵平衡二叉树很高的话，那中间就要判断很多个节点。
所以后来也出现了综合性能比其更好的树—-红黑树

红黑树

- 红黑树（Red Black Tree），红黑树由Rudolf Bayer于1972年发明。
- 红黑树和之前所讲的平衡二叉树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。
- 红黑树和AVL树的区别在于它使用颜色来标识结点的高度，它所追求的是局部平衡而不是AVL树中的非常严格的平衡。

红黑树

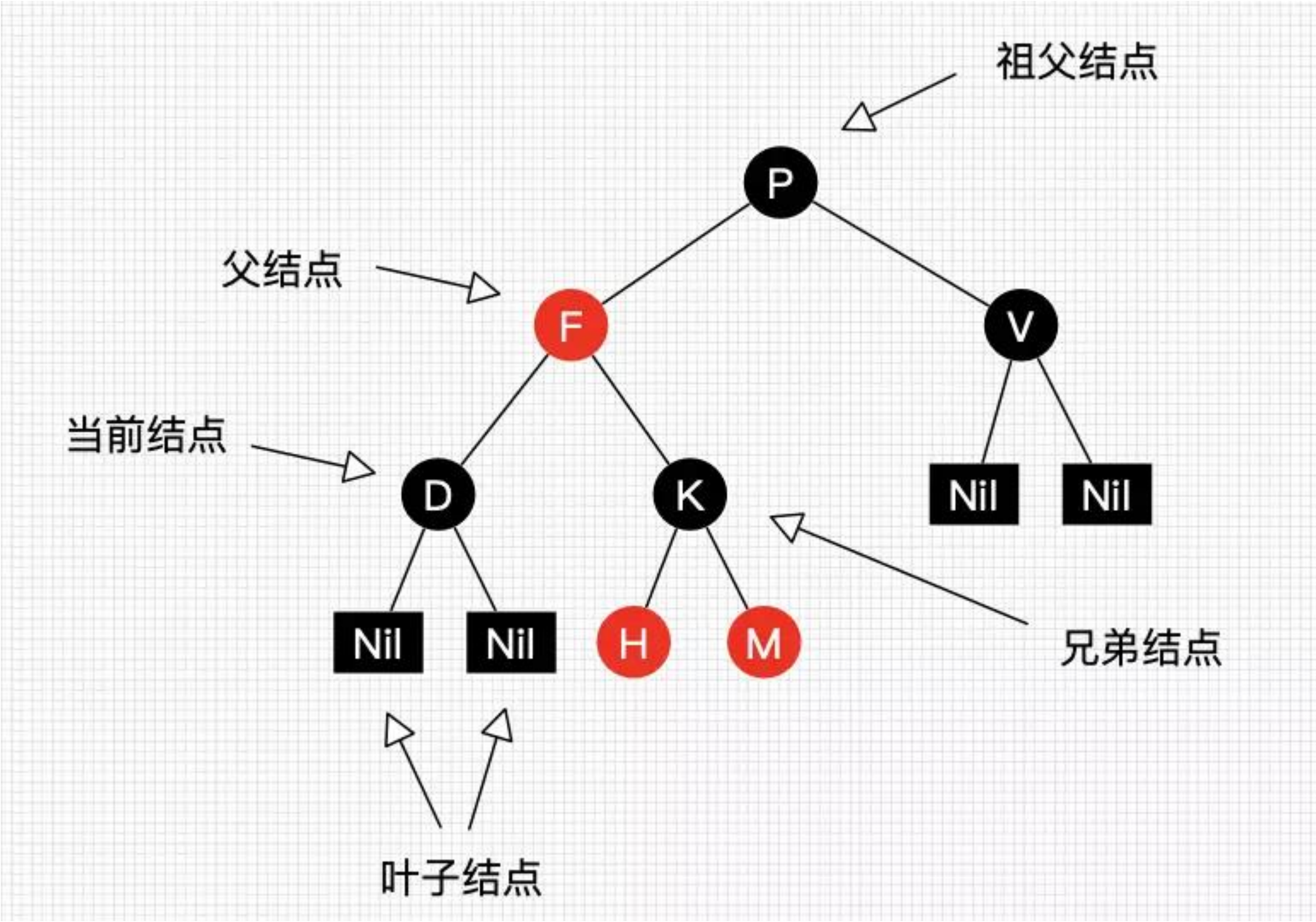


Nil 和 Null 的区别?

红黑树——5个性质

1. 每个结点的颜色只能是红色或黑色。
2. 根结点是黑色的。
3. 每个叶子结点都带有两个空的黑色结点（哨兵），如果一个结点n只有一个左孩子，那么n的右孩子是一个黑哨兵；如果结点n只有一个右孩子，那么n的左孩子是一个黑哨兵。
4. 如果一个结点是红的，则它的两个儿子都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。
5. 对于每个结点来说，从该结点到其子孙叶结点的所有路径上包含相同数目的黑结点。

红黑树



红黑树——操作

- 因为每一个红黑树也是一个二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。
- 在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。
- 恢复红黑树的性质需要少量(O(log *n*))的颜色变更和不超过三次树旋转(对于插入操作是两次)。
- 虽然插入和删除很复杂，但操作时间仍可以保持为 O(log *n*) 次。

红黑树——自平衡

- 左旋：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。
- 右旋：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。
- 变色：结点的颜色由红变黑或由黑变红。

红黑树——查找

- 从根结点开始查找，把根结点设置为当前结点；
- 若当前结点为空，返回null；
- 若当前结点不为空，用当前结点的key跟查找key作比较；
- 若当前结点key等于查找key，那么该key就是查找目标，返回当前结点；
- 若当前结点key大于查找key，把当前结点的左子结点设置为当前结点，重复步骤2；
- 若当前结点key小于查找key，把当前结点的右子结点设置为当前结点，重复步骤2；

非常简单，但简单不代表它效率不好。正由于红黑树总保持黑色完美平衡，所以它的查找最坏时间复杂度为O(2lgN)，也即整颗树刚好红黑相隔的时候。能有这么好的查找效率得益于红黑树自平衡的特性，而这背后的付出，红黑树的插入操作功不可没～

红黑树——插入

首先以二叉查找树的方法增加节点并标记它为红色。

二叉查找树B中插入一个节点s的算法：

1. 若B是空树，则将s点作为根节点；否则：
2. 若s等于B的根节点的数据值，则返回；否则：
3. 若s小于B的根节点的数据值，则把s点插入到左子树中（递归）；否则：
4. 把s所指节点插入到右子树中（递归）。

新插入节点总是叶子节点，是红色。

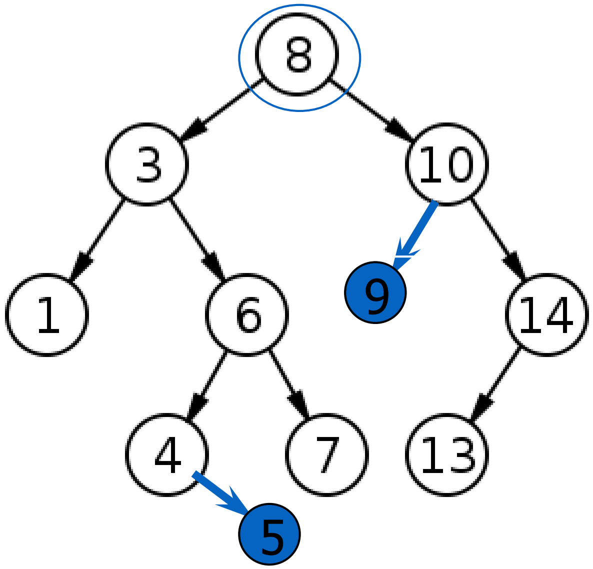
红黑树——插入

二叉查找树B中插入一个节点s的算法：

S.data = 8;

S.data = 5;

S.data =9;



新插入节点总是叶子节点

红黑树——插入

- ➢首先以二叉查找树的方法增加节点并标记它为红色。
- ➢设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换和树旋转来调整。
- ➢下面要进行什么操作取决于其他临近节点的颜色。

红黑树——插入

情形1：没有父节点，新节点N位于树的根上。

在这种情形下，把它重绘为黑色以满足性质2。

红黑树——插入

情形2： 新节点的父节点P是黑色，所以性质4没有失效（新节点是红色的）。

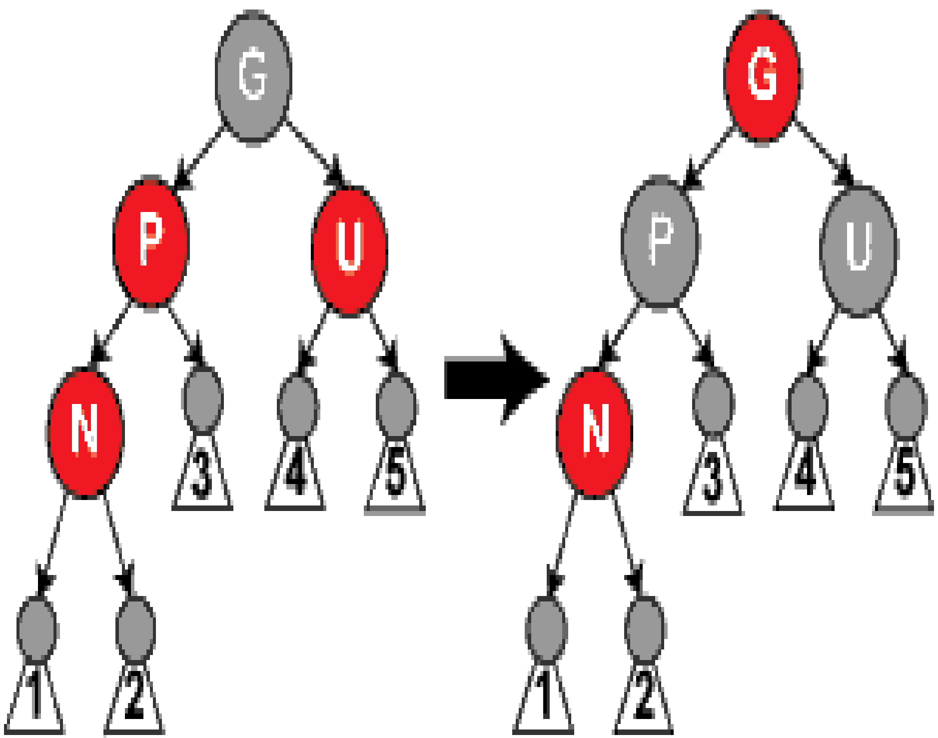
在这种情形下，树仍是有效的。

满足性质4和5。

注意：
在下列情形下我们假定新节点的父节点为红色。

红黑树——插入

情形3：N是新节点，如果父节点P和叔父节点U二者都是红色。



1. 则将它们两个重绘为黑色并重绘祖父节点G为红色(用来保持性质4)。
2. 新节点N有了一个黑色的父节点P。在这些路径上的黑节点数目没有改变。
3. 但是，红色的祖父节点G的父节点也有可能是红色的，这就违反了性质4。为此在祖父节点G上递归地进行情形1的整个过程。（把G当成是新加入的节点进行各种情形的检查 ）

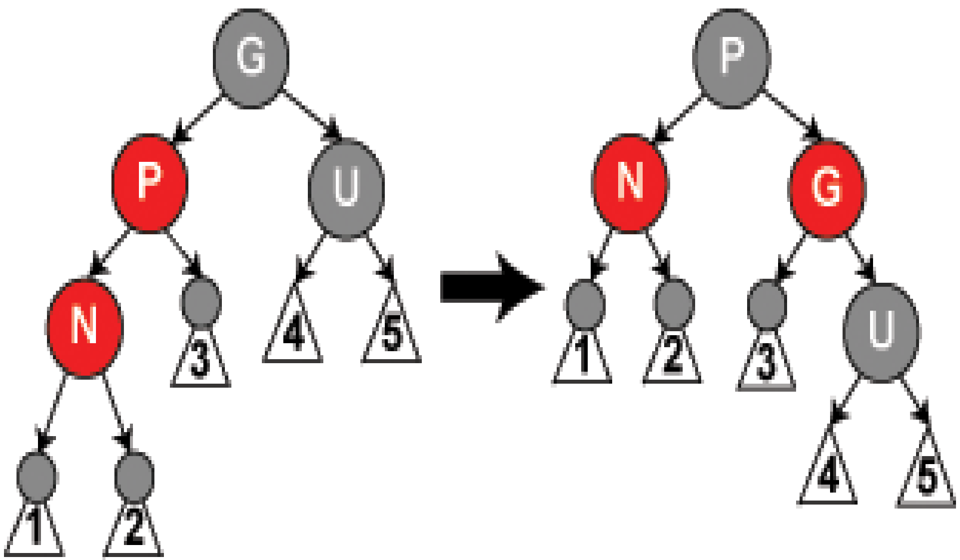
红黑树——插入

注意： 在余下的情形下，我们假定父节点P是其父亲G的左子节点。

如果它是右子节点，情形4和情形5中的左和右应当对调。

红黑树——插入

情形4：父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点，而父节点P又是其父节点G的左子节点。

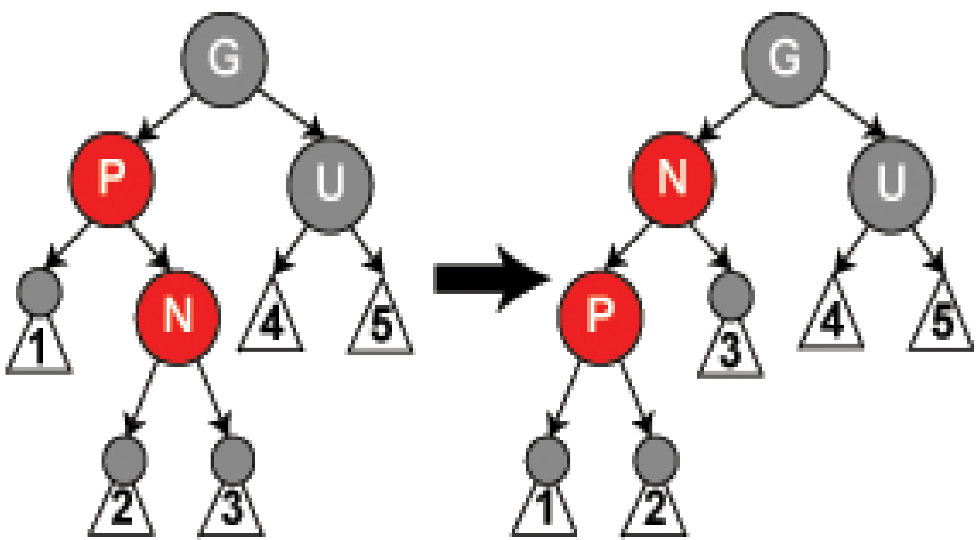


- 进行针对祖父节点G的一次**右旋转**; 以前的父节点P现在是新节点N和以前的祖父节点G的父节点。
- 以前的祖父节点G是黑色，以前的父节点P和祖父节点G的颜色，结果的树满足性质4。
- 性质5也仍然保持满足。

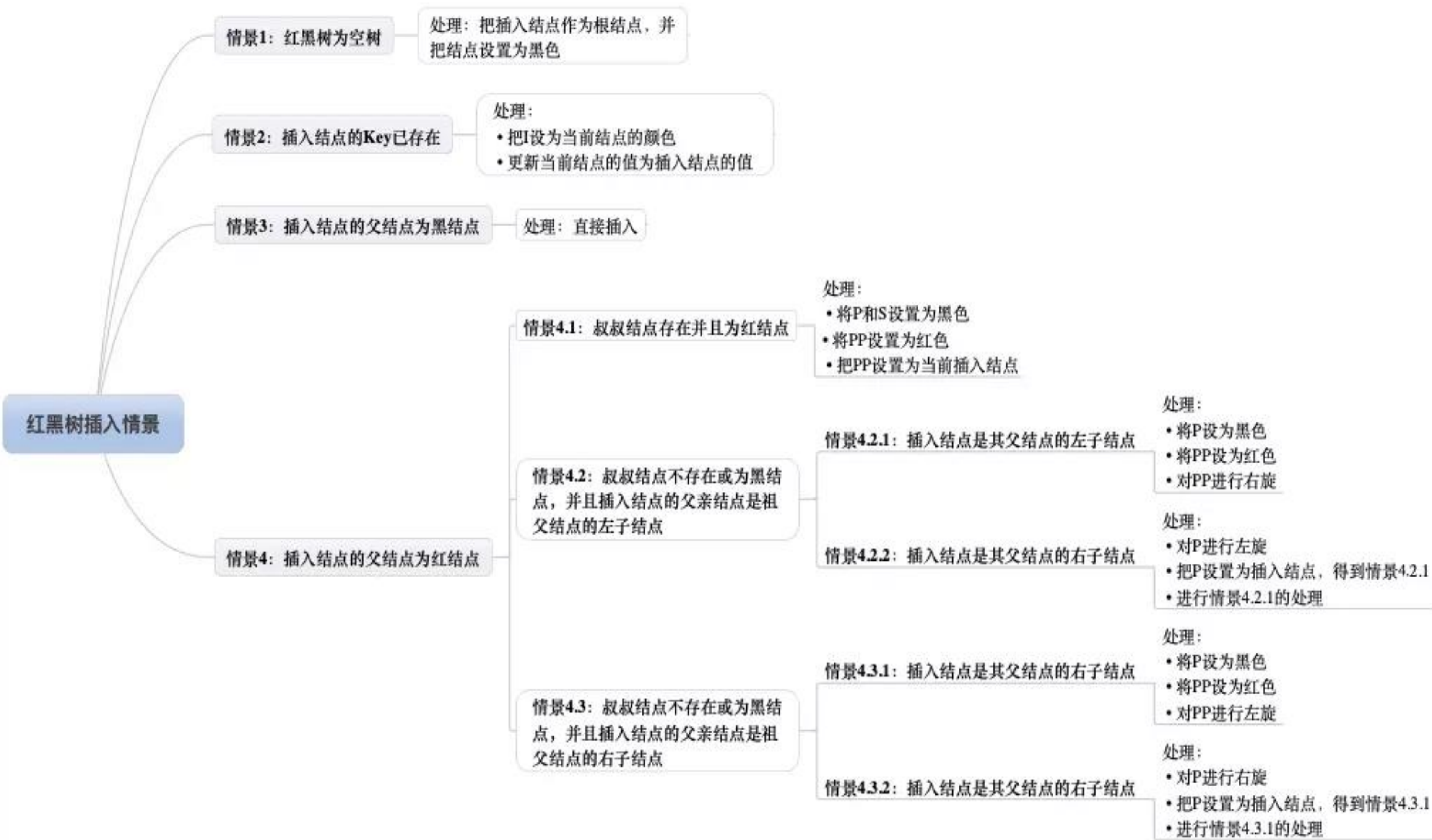
红黑树——插入

情形5： 父节点P是红色而叔父节点U是黑色或缺少，并且新节点N是其父节点P的右子节点而父节点P又是其父节点的左子节点。

进行一次**左旋转**调换新节点和其父节点的角色; 接着，按情形4处理。



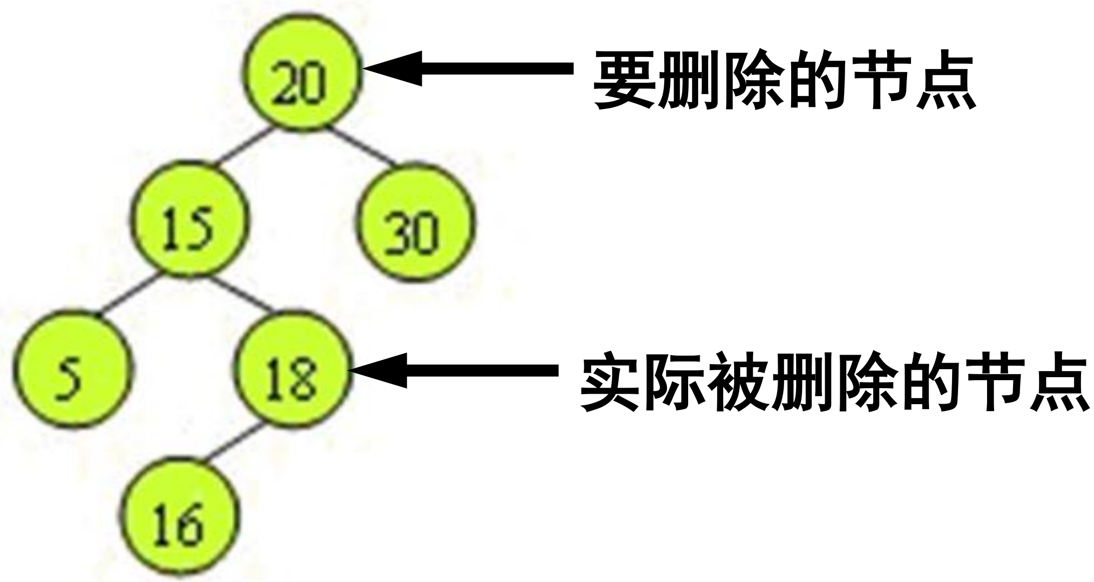
红黑树——插入总结



红黑树插入情景
情景1：红黑树为空树
处理：把插入结点作为根结点，并把结点设置为黑色
情景2：插入结点的Key已存在
处理：
• 把I设为当前结点的颜色
• 更新当前结点的值为插入结点的值
情景3：插入结点的父结点为黑结点
处理：直接插入
情景4：插入结点的父结点为红结点
情景4.1：叔叔结点存在并且为红结点
处理：
• 将P和S设置为黑色
• 将PP设置为红色
• 把PP设置为当前插入结点
情景4.2：叔叔结点不存在或为黑结点，并且插入结点的父亲结点是祖父结点的左子结点
情景4.2.1：插入结点是其父结点的左子结点
处理：
• 将P设为黑色
• 将PP设为红色
• 对PP进行右旋
情景4.2.2：插入结点是其父结点的右子结点
处理：
• 对P进行左旋
• 把P设置为插入结点，得到情景4.2.1
• 进行情景4.2.1的处理
情景4.3：叔叔结点不存在或为黑结点，并且插入结点的父亲结点是祖父结点的右子结点
情景4.3.1：插入结点是其父结点的右子结点
处理：
• 将P设为黑色
• 将PP设为红色
• 对PP进行左旋
情景4.3.2：插入结点是其父结点的右子结点
处理：
• 对P进行右旋
• 把P设置为插入结点，得到情景4.3.1
• 进行情景4.3.1的处理

红黑树——删除

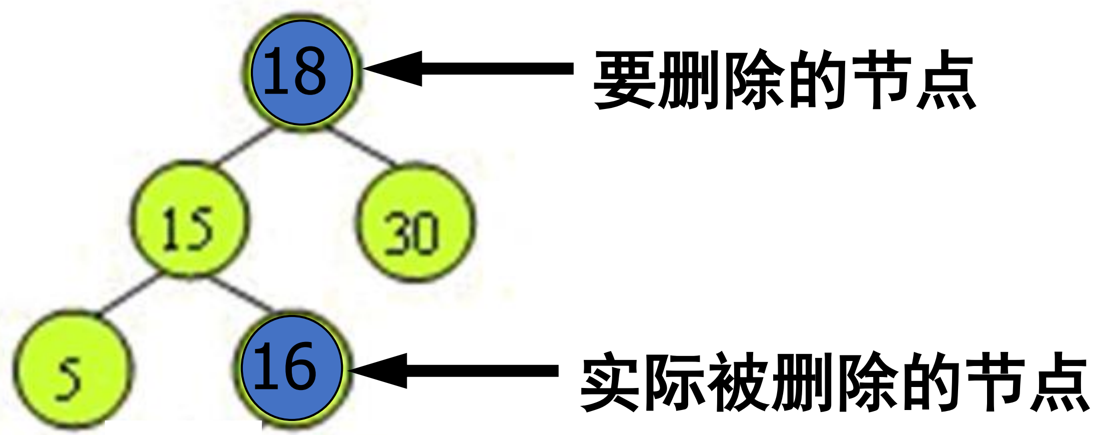
如图所示的二叉树中，删除数据值为20节点



如图所示，当删除节点20时，实际被删除的节点应该为18，节点20的数据变为18。

红黑树——删除

如图所示的二叉树中，删除数据值为20节点



如图所示，当删除节点20时，实际被删除的节点应该为18，节点20的数据变为18。

红黑树——删除

红黑树的基本操作(三) 删除

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：

红黑树——删除

删除常规二叉查找树中删除节点的方法是一样。分3种情况：

◆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。

◆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。

◆被删除节点有两个儿子。那么，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”。

红黑树——删除

- 我们只需要讨论删除只有一个儿子的节点
- 如果它两个儿子都为空，我们任意将其中一个看作它的儿子。
- 如果需要删除的节点有两个儿子，那么问题可以被转化成另一个删除只有一个儿子的节点的问题
- 难点在于要删除的节点和它的儿子二者都是黑色的时候：
- 如果儿子N和它原始的父亲是黑色，则删除父亲导致通过N的路径都比不通过它的路径少了一个黑色节点。 这违反了性质5，需要重新平衡。

红黑树——删除

情形1：删除后儿子N是新的根。

在这种情形下，我们就做完了。

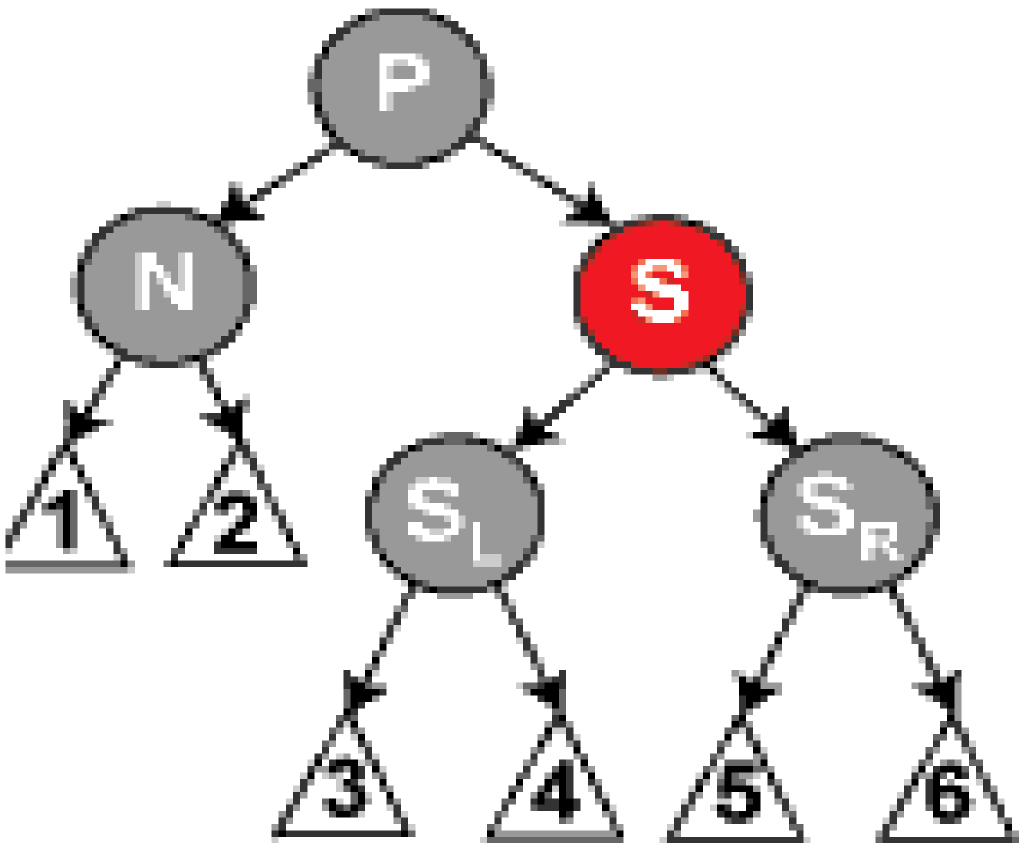
我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

注意：

在以下情形2、5和6下，我们假定N是它父亲的左儿子。如果它是右儿子，则在这些情形下的左和右应当对调。

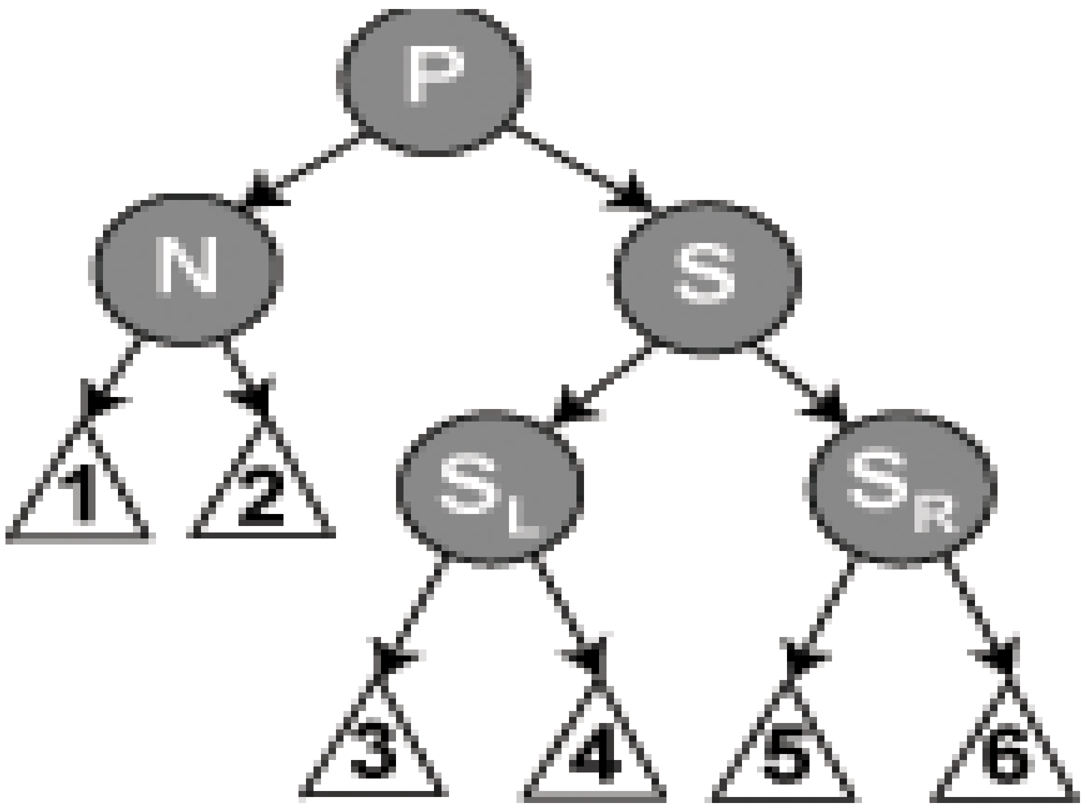
红黑树——删除

情形2：删除前S是红色，



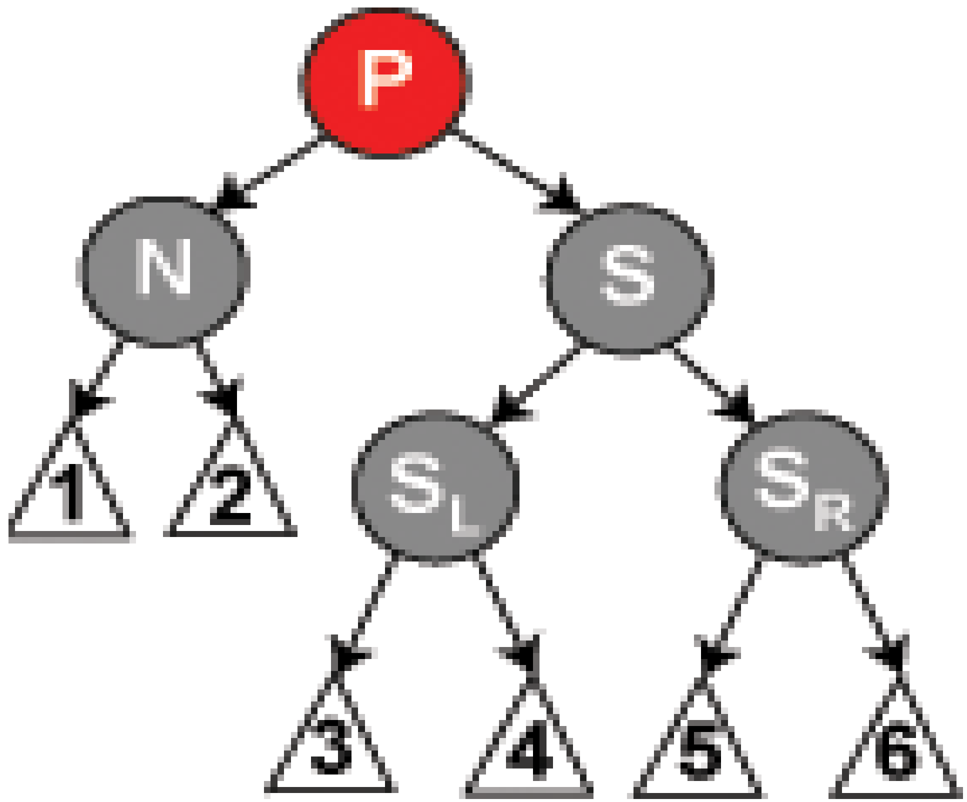
红黑树——删除

情形3：删除后新节点N的父亲、S和S的儿子都是黑色的。



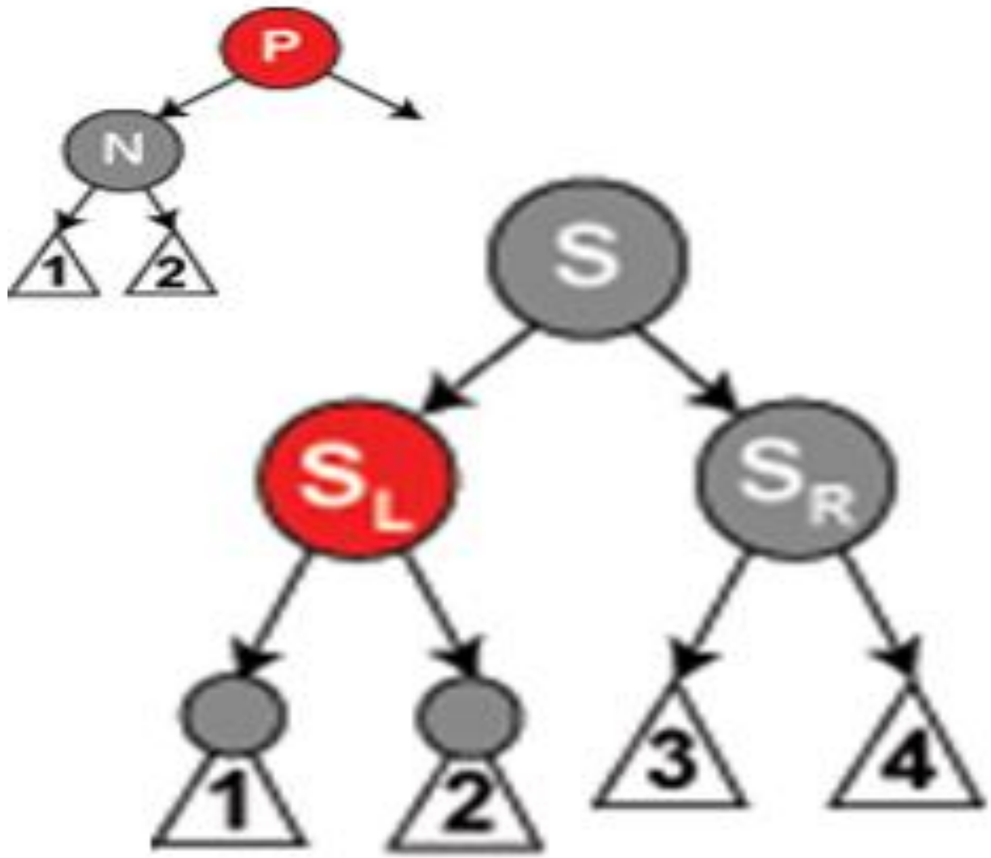
红黑树——删除

情形4：删除后新节点N、S和S的儿子都是黑色，但是N的父亲是红色。



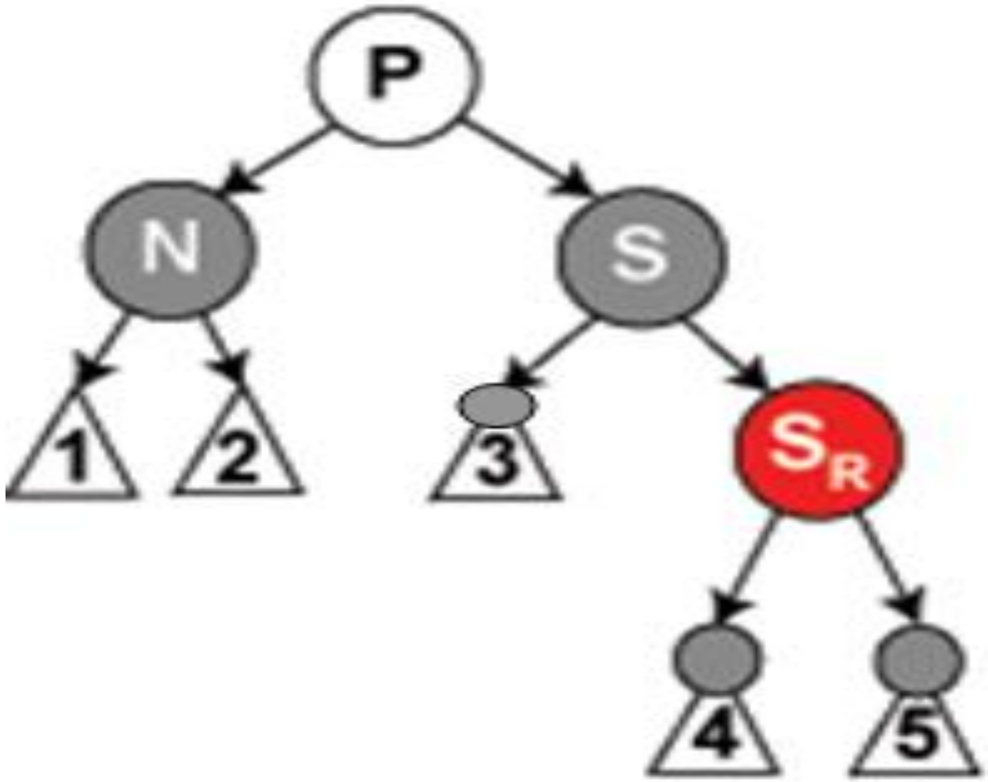
红黑树——删除

情形5：删除后新节点N的S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。



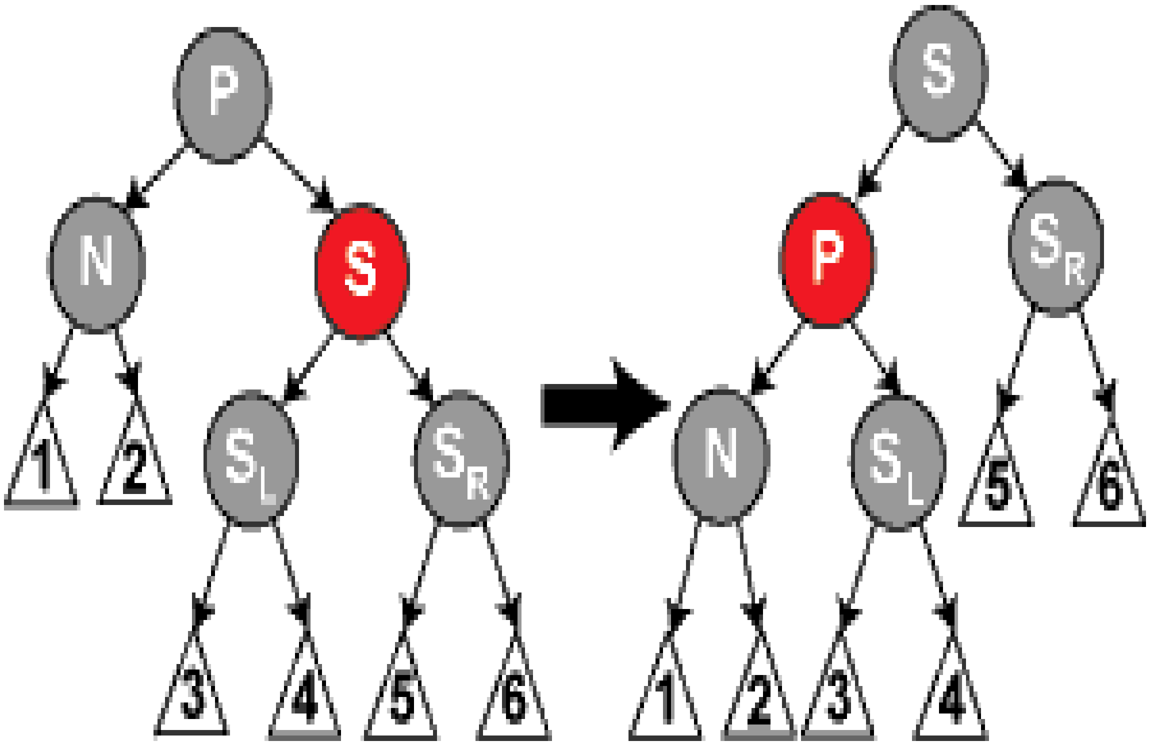
红黑树——删除

情形6: 删除后S是黑色，S的右儿子是红色， 而N是它父亲的左儿子。



红黑树——删除

情形2：删除前S是红色，
先变换后删除。



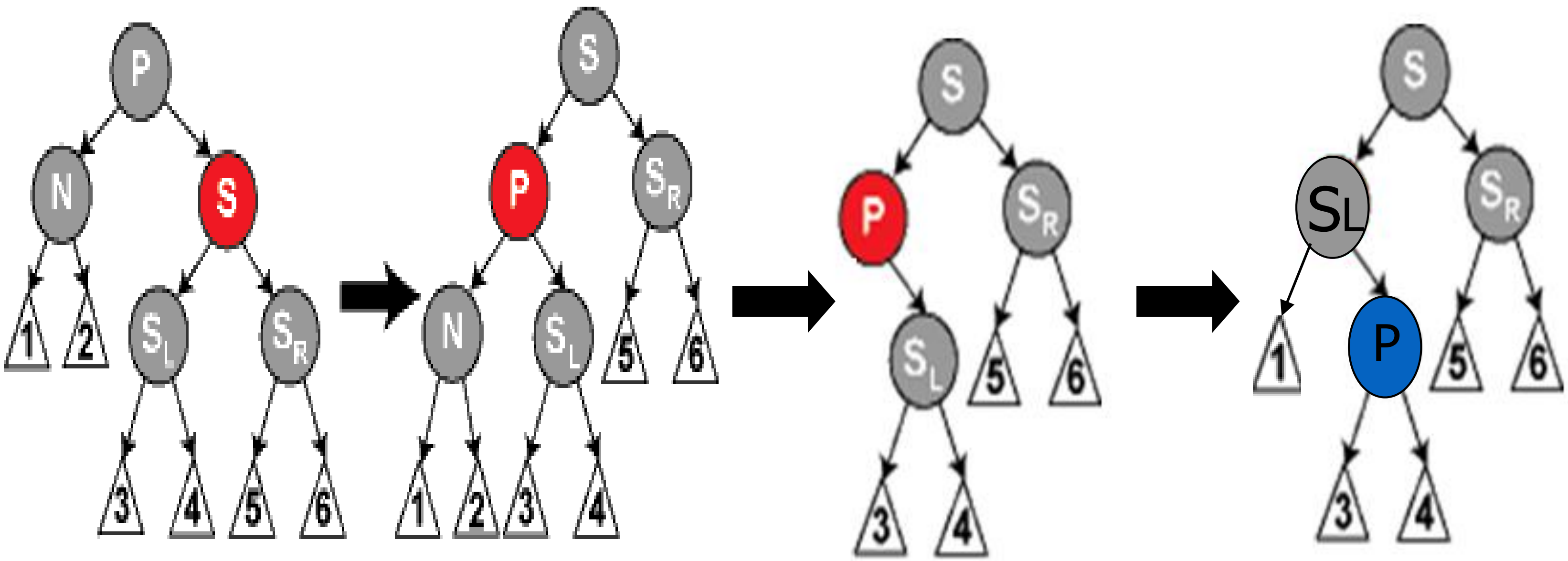
在N的父亲上做左旋转，把红色兄弟转换成N的祖父，对调N的父亲和祖父的颜色。

现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），

接下去按情形4、情形5或情形6来处理。

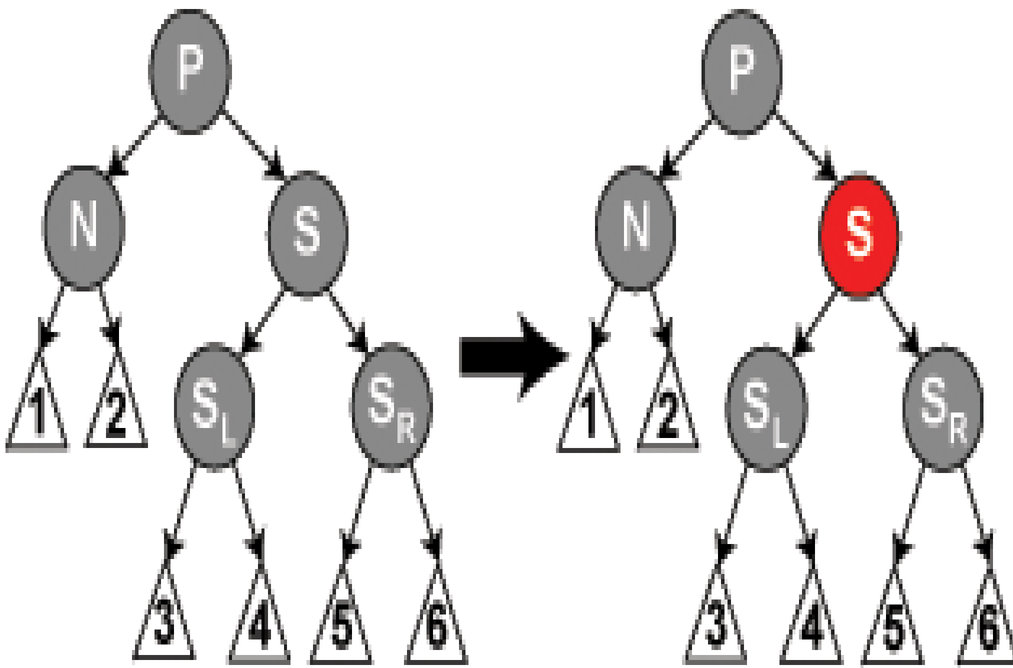
红黑树——删除

情形2：删除前S是红色，
先变换后删除。



红黑树——删除

情形3：删除后新节点N的父亲、S和S的儿子都是黑色的。



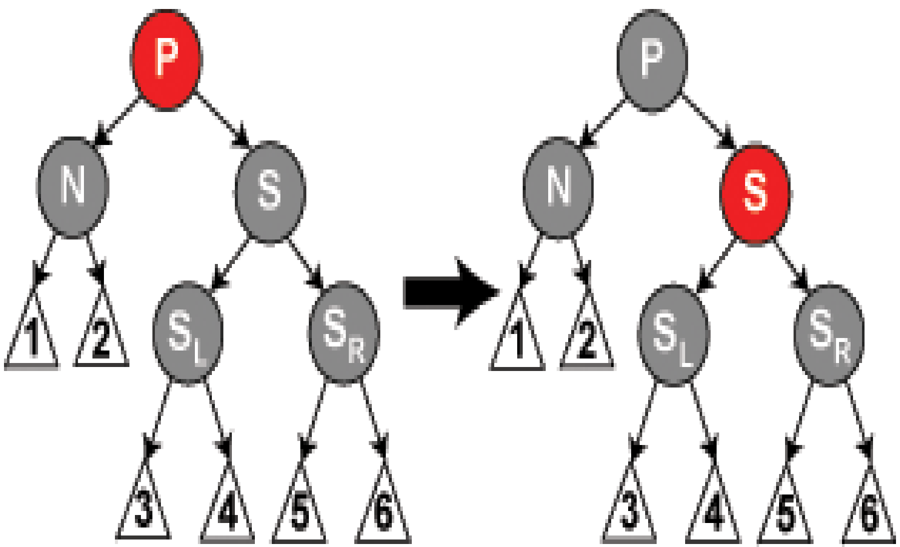
简单的重绘S为红色。

结果是通过S的所有路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。

但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从情形1开始，在P上做重新平衡处理。

红黑树——删除

情形4：删除后新节点N、S和S的儿子都是黑色，但是N的父亲是红色。

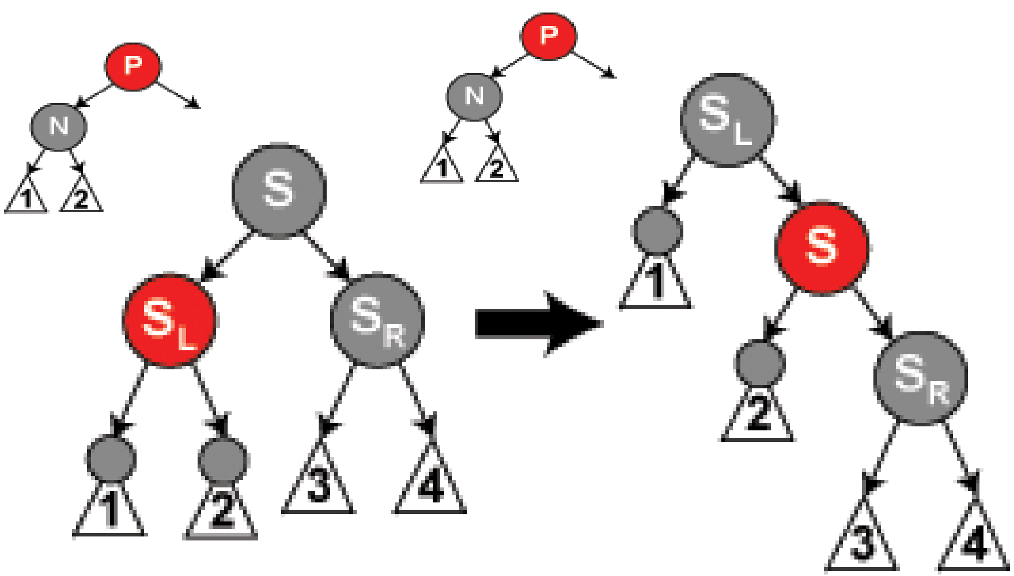


在这种情形下，交换N的兄弟和父亲的颜色。

这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

红黑树——删除

情形5：删除后新节点N的S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。



在S上做右旋转，这样S的左儿子成为S的父亲和N的新兄弟。

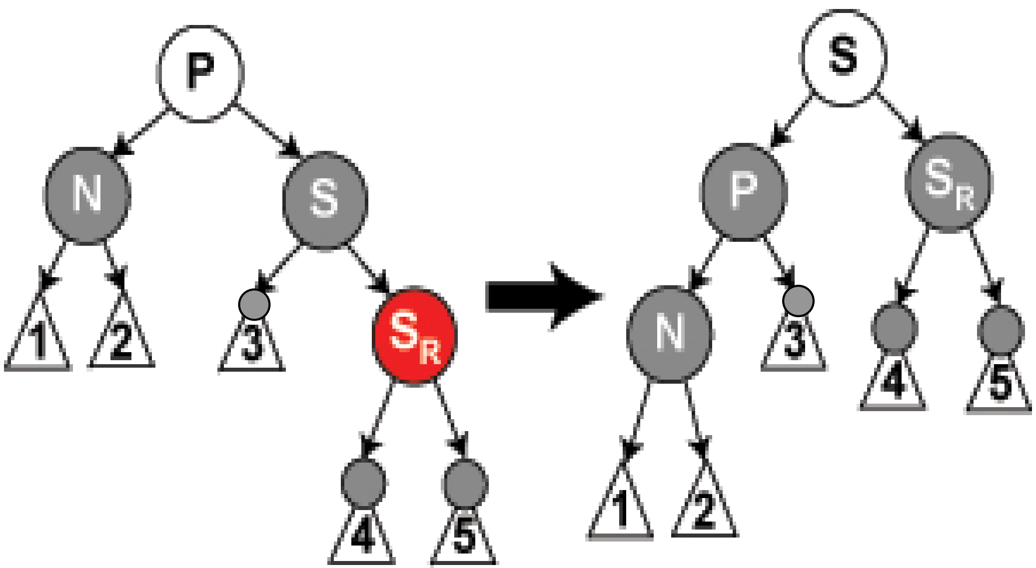
接着交换S和它的新父亲的颜色。

所有路径仍有同样数目的黑色节点，但是现在N有了一个右儿子是红色的黑色兄弟，所以我们进入了情形6。

N和它的父亲都不受这个变换的影响。

红黑树——删除

情形6：调整后N节点的S是黑色，S的右儿子是红色，而N是它父亲的左儿子。



在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。

接着交换N的父亲和S的颜色，并使S的右儿子为黑色。

子树在它的根上的仍是同样的颜色，所以性质3没有被违反。

红黑树——删除总结

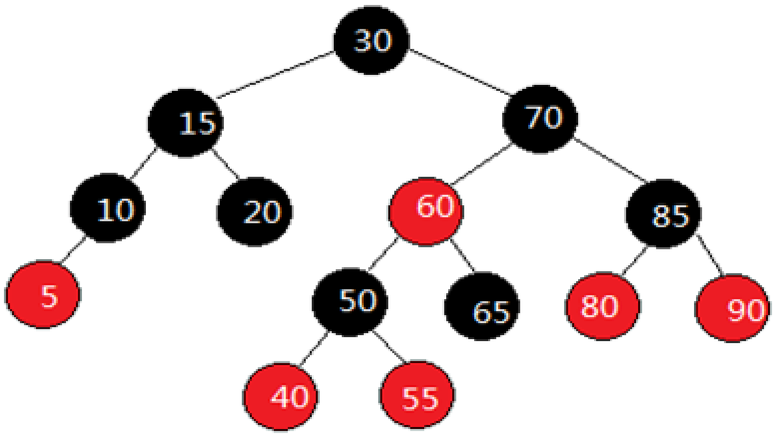
- 红黑树删除情景
  - 情景1：替换结点是红色结点
    - 处理：颜色变为黑色，也即父结点的颜色
  - 情景2：替换结点是黑结点
    - 情景2.1：替换结点是其父结点的左子结点
      - 情景2.1.1：替换结点的兄弟结点是红结点
        - 处理：
          - 将S设为黑色
          - 将P设为红色
          - 对P进行左旋，得到情景2.1.2.3
          - 进行情景2.1.2.3的处理
      - 情景2.1.2：替换结点的兄弟结点是黑结点
        - 情景2.1.2.1：替换结点的兄弟结点的右子结点是红结点，左子结点任意颜色
          - 处理：
            - 将S的颜色设为P的颜色
            - 将P设为黑色
            - 将SR设为黑色
            - 对P进行左旋
        - 情景2.1.2.2：替换结点的兄弟结点的右子结点为黑结点，左子结点为红结点
          - 处理：
            - 将S设为红色
            - 将SL设为黑色
            - 对S进行右旋，得到情景2.1.2.1
            - 进行情景2.1.2.1的处理
        - 情景2.1.2.3：替换结点的兄弟结点的子结点都为黑结点
          - 处理：
            - 将S设为红色
            - 把P作为新的替换结点
            - 重新进行删除结点情景处理
    - 情景2.2：替换结点是其父结点的右子结点
      - 情景2.2.1：替换结点的兄弟结点是红结点
        - 处理：
          - 将S设为黑色
          - 将P设为红色
          - 对P进行右旋，得到情景2.2.2.3
          - 进行情景2.2.2.3的处理
      - 情景2.2.2：替换结点的兄弟结点是黑结点
        - 情景2.2.2.1：替换结点的兄弟结点的左子结点是红结点，右子结点任意颜色
          - 处理：
            - 将S的颜色设为P的颜色
            - 将P设为黑色
            - 将SL设为黑色
            - 对P进行右旋
        - 情景2.2.2.2：替换结点的兄弟结点的左子结点为黑结点，右子结点为红结点
          - 处理：
            - 将S设为红色
            - 将SR设为黑色
            - 对S进行左旋，得到情景2.2.2.1
            - 进行情景2.2.2.1的处理
        - 情景2.2.2.3：替换结点的兄弟结点的子结点都为黑结点
          - 处理：
            - 将S设为红色
            - 把P作为新的替换结点
            - 重新进行删除结点情景处理

红黑树—应用

索引数据结构关联的科学问题：

- 如何设计并行数据库？提高数据处理性能；
- 如何设计内存数据库？提高数据处理性能；
- 如何设计可近似匹配的索引数据结构？提高数据处理功能，尤其是序列数据近似比对能力。

当在10亿数据中只需要进行10几次比较就能查找到目标时，不禁感叹编程之魅力！人类之伟大呀！ —— 学红黑树有感。

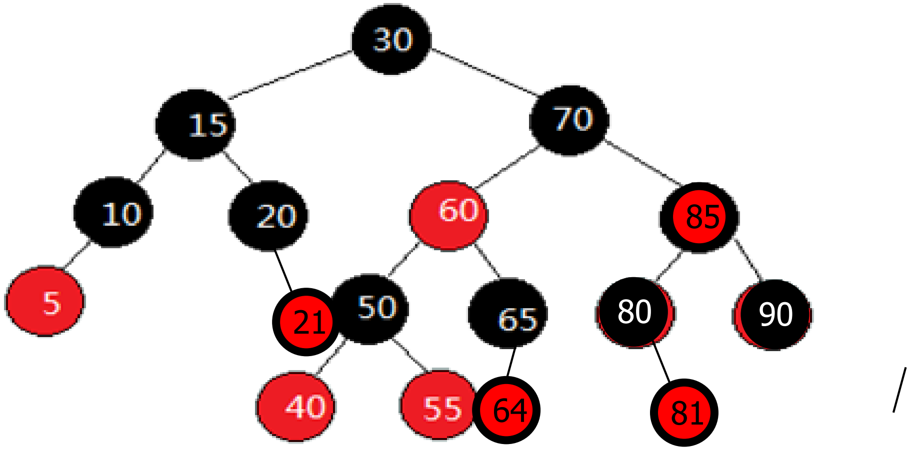


红黑树—练习题

右图1是一颗红黑树。

1. 插入21节点；
2. 插入64节点；
3. 插入81节点；

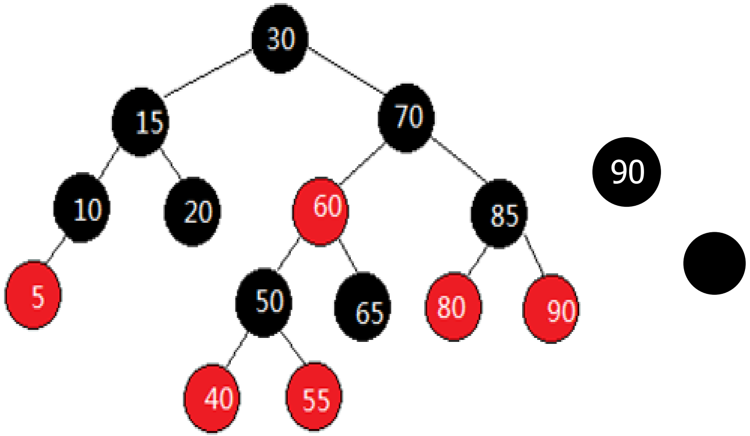
图1中描绘上述三种操作后的结果，标注节点的颜色(红/黑)。



右图2是一颗红黑树。

1. 删除10节点；
2. 删除50节点；

图2中描绘上述三种操作后的结果，标注节点的颜色(红/黑)。

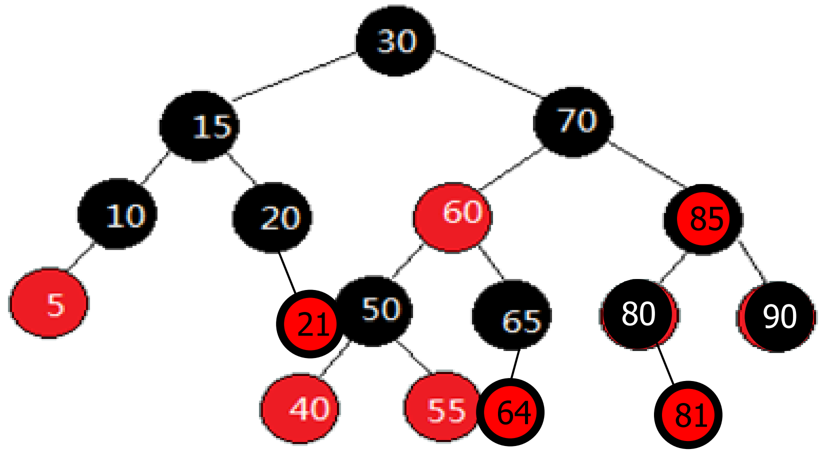


红黑树—练习题

右图1是一颗红黑树。

1. 插入21节点；
2. 插入64节点；
3. 插入81节点；

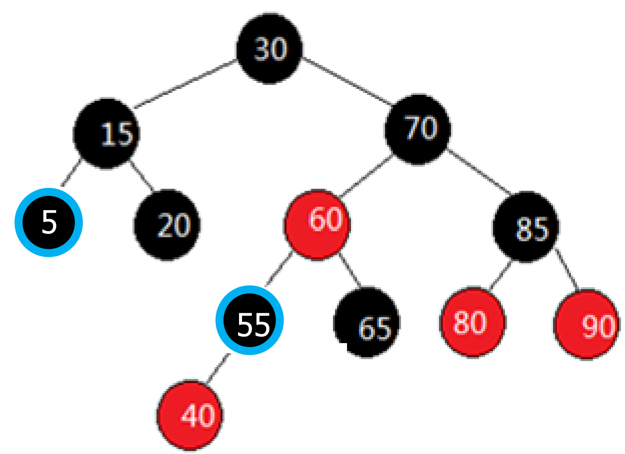
图1中描绘上述三种操作后的结果，标注节点的颜色(红/黑)。



右图2是一颗红黑树。

1. 删除10节点；
2. 删除50节点；

图2中描绘上述三种操作后的结果，标注节点的颜色(红/黑)。



红黑树——5个性质

这个不是用二三树来解释的

1. 每个结点的颜色只能是红色或黑色。
2. 根结点是黑色的。
3. 每个叶子结点都带有两个空的黑色结点（哨兵），如果一个结点n只有一个左孩子，那么n的右孩子是一个黑哨兵；如果结点n只有一个右孩子，那么n的左孩子是一个黑哨兵。
4. 如果一个结点是红的，则它的两个儿子都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。
5. 对于每个结点来说，从该结点到其子孙叶结点的所有路径上包含相同数目的黑结点。