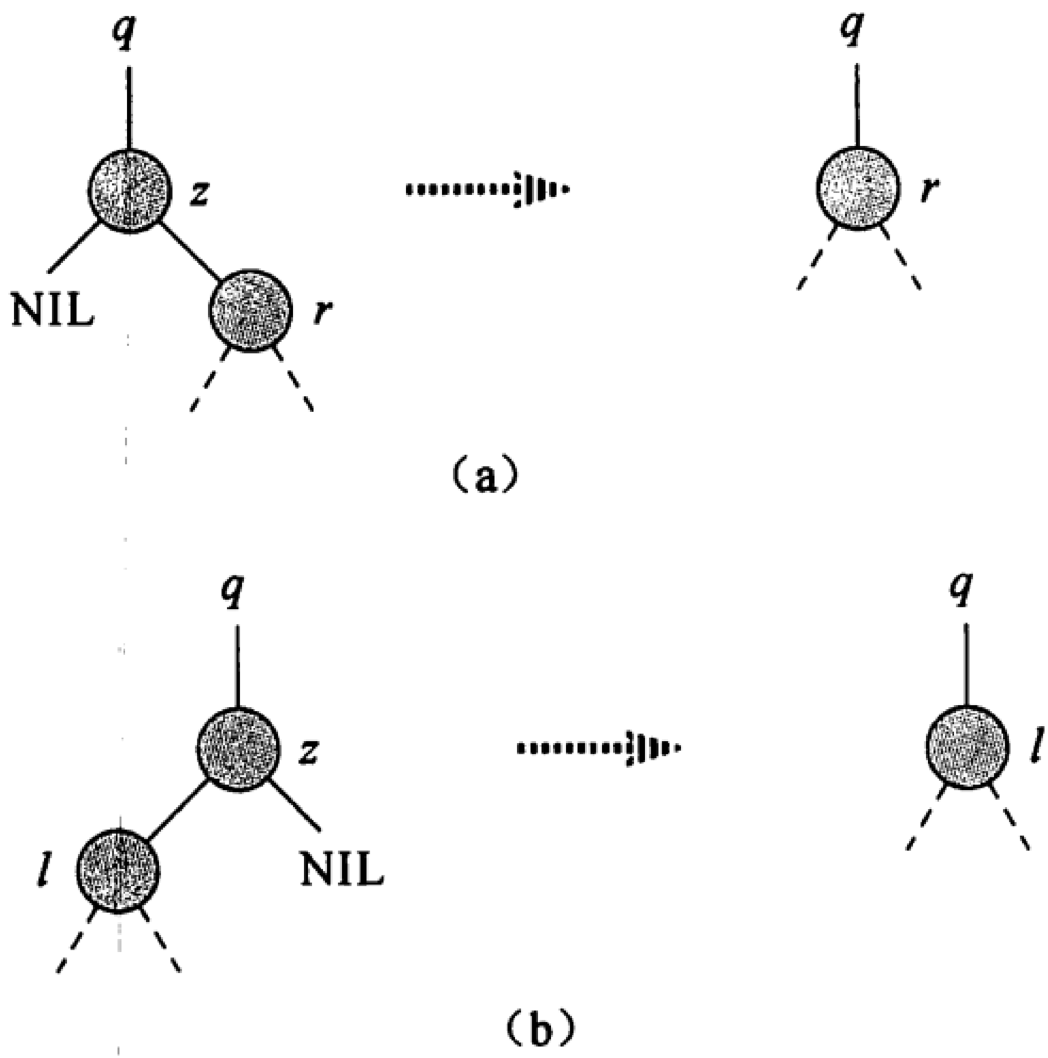
算法导论复习3——数据结构

Chapter 12 二叉搜索树

二叉搜索树的删除

- **只有左儿子或者只有右儿子：**

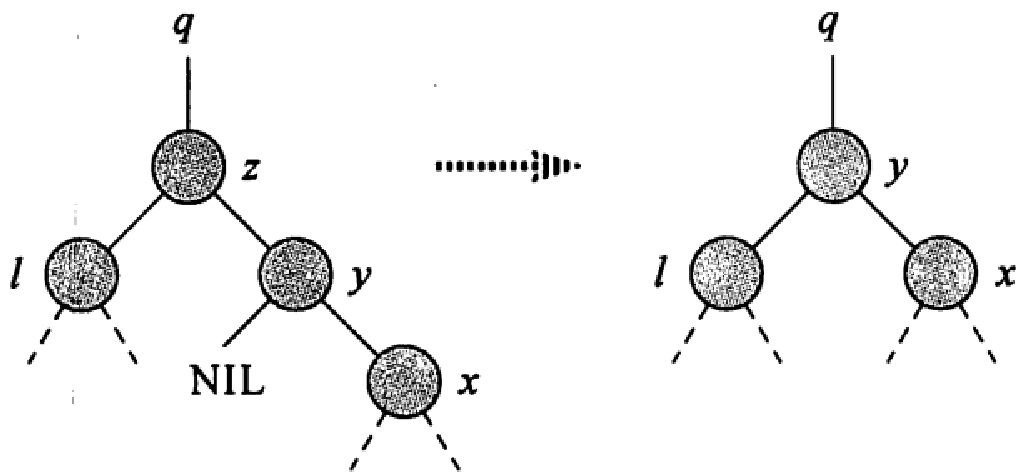
  直接将左儿子或右儿子上位替代删除节点



- **有左儿子和右儿子：**

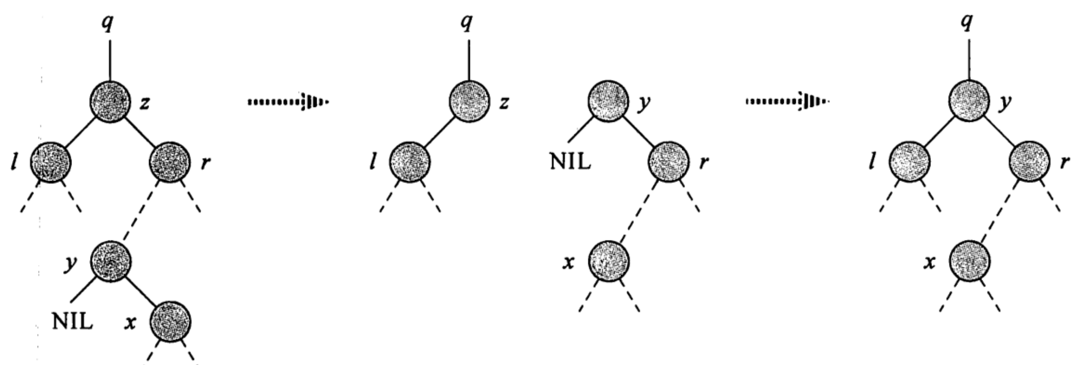
  找到直接后继（**右子树的最左节点**）

  - 如果直接后继是删除节点的右儿子，直接上位替代：



(c)

- 如果直接后继不是删除节点的右儿子：
  1. 将直接后继上提
  2. 并将其右子树接到其父亲左儿子上
  3. 上提后的直接后继上位替代删除节点



(d)

代码实现

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

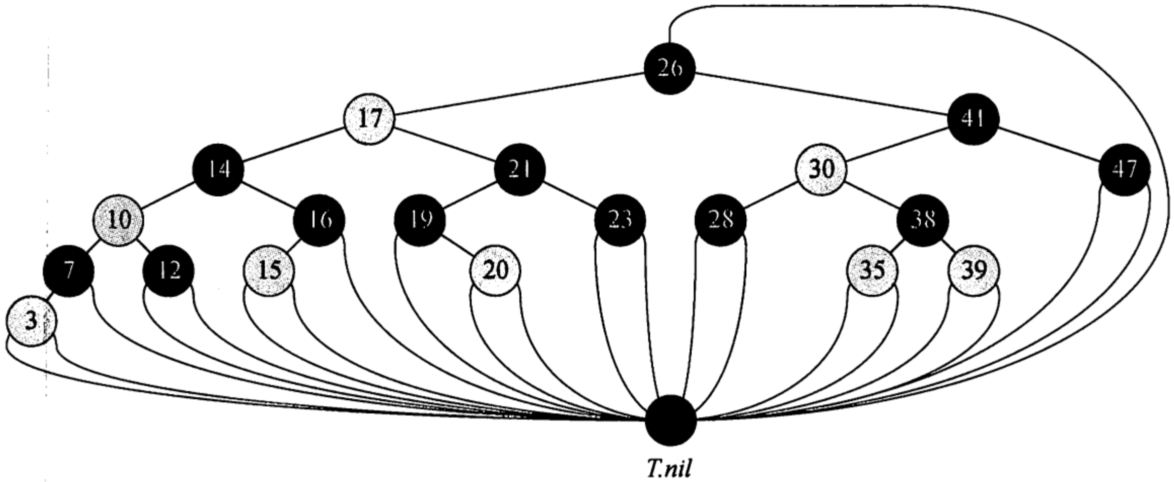
Chapter 13 红黑树

红黑树性质

红黑树是有如下性质的**二叉搜索树**：

1. 每个节点是红色或者黑色
2. 根节点是黑色的
3. 每个叶节点NIL是黑色的
4. 如果一个节点是红色的，那么它的两个儿子是黑色的
5. 每个节点到所有后代叶节点的路径上黑色节点数相同（黑高相同）

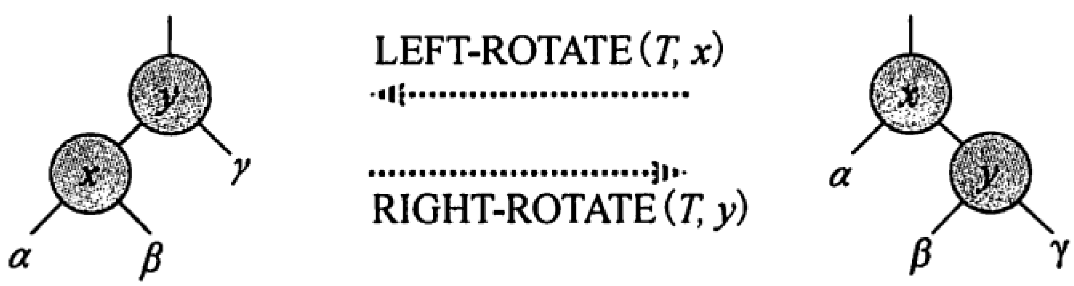
   推论：有黑色儿子的节点一定有两个儿子



- **一棵有n个内部节点的红黑树的高度至多为**$2log(n+1)$
- 所有从根到叶节点的路径中，没有长度相差两倍的路径——近似平衡
- root节点的父亲是NIL

红黑树的旋转

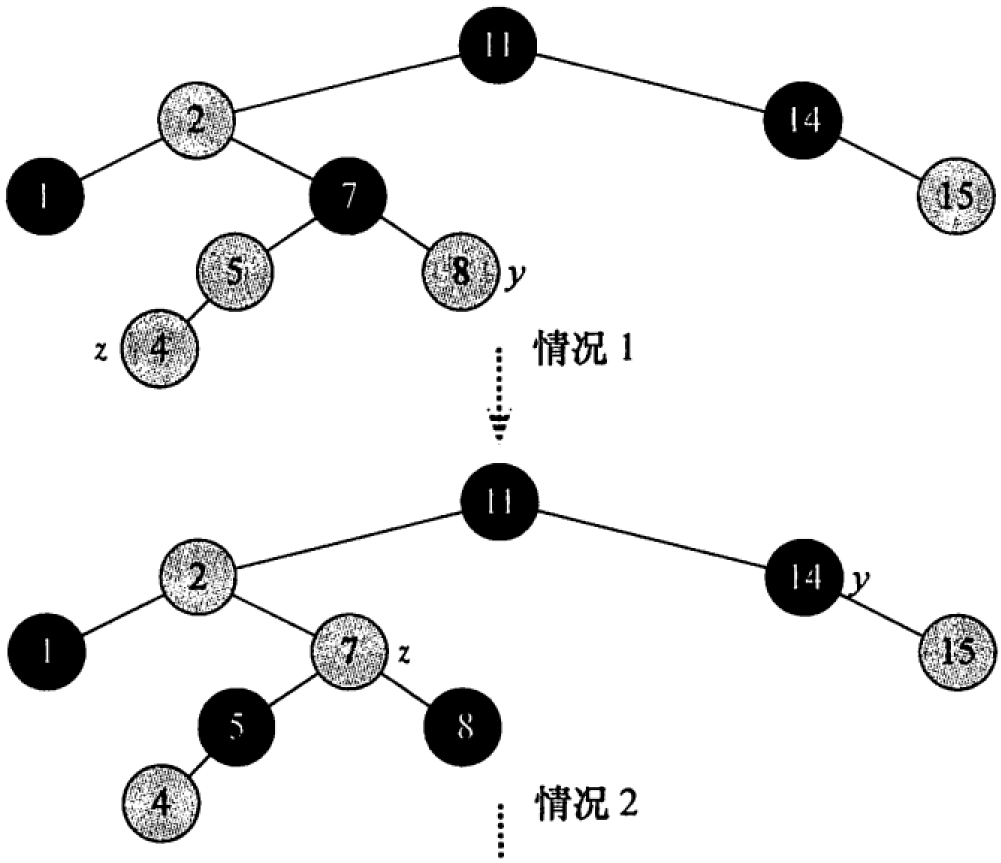
- 旋转操作可以保持二叉搜索树的性质
- 左旋：
  1. 左儿子成为父亲
  2. 左儿子的右儿子接到自己的左儿子上
- 右旋：
  1. 右儿子成为父亲
  2. 右儿子的左儿子接到自己的左儿子上



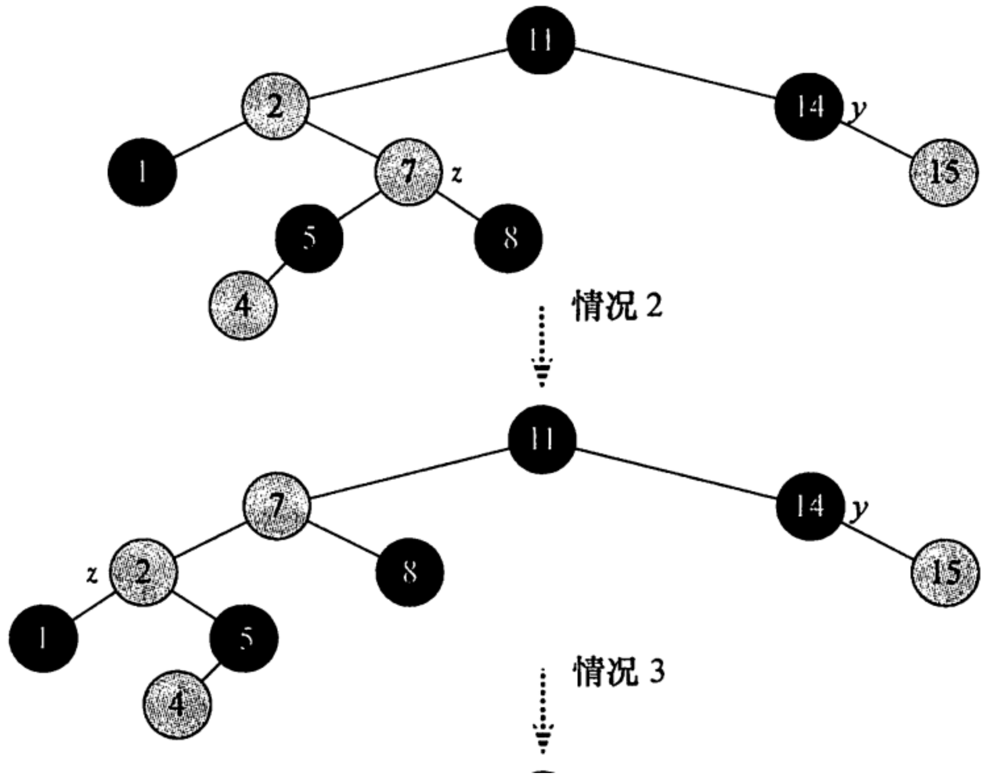
红黑树的插入

- 插入方法：
  1. **新节点一定是红色**
  2. 按照二叉搜索树插入法插入到一个叶子中
  3. RB_INSERT_FIXUP调整红黑树结构
- 插入后调整：
  - 任何时候都要调用调整，但只有当其父亲为红色节点时才实际进行调整（父亲红色可能使得性质4被违反），最后强制将根节点染黑即可保持性质2

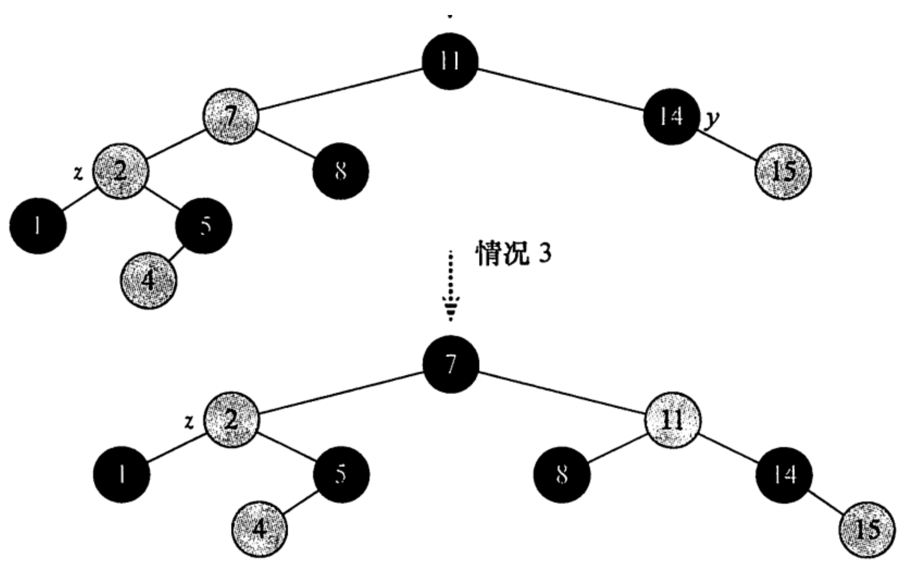
- 插入后调整步骤：
  - 若父亲为黑色，则将根节点染黑后退出
  - 否则，分为如下三种情况：（这时爷爷必定是黑色，且**假设父亲是爷爷左儿子**，右儿子对称即可）
    1. 若叔叔也为红色，则**将爷爷的黑色传递至叔叔和父亲，爷爷变成红色，将调整指针指向爷爷，再进行调整**



    2. 若叔叔为黑色，且自己为父亲右儿子，则**基于父亲左旋一次**转换为情况3



3. 若叔叔为黑色，且自己为父亲左儿子，则**将爷爷的黑色和父亲的红色对换，基于父亲做一次右旋**



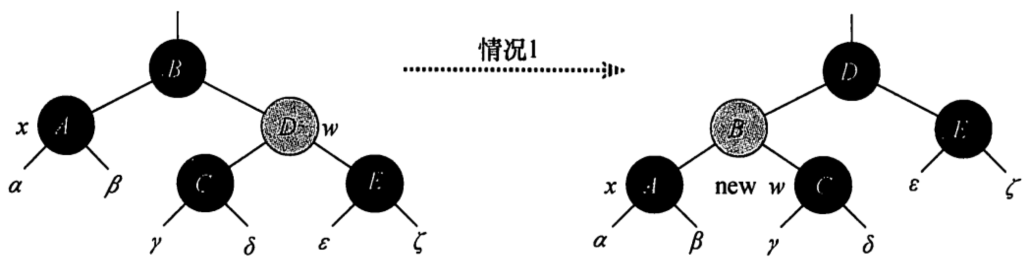
红黑树的删除

- 删除方法：
    1. 按照二叉搜索树删除方法删除节点
    2. RB_DELETE_FIXUP
- 删除后调整：
    - 实际删除的节点和删除的节点可能并不一样，下面讨论的都是**删除节点**
        - 对于仅有一个孩子的情况，实际删除的节点和删除的节点是一样的
        - 对于有两个孩子的情况，如果直接后继是右孩子，那么实际删除节点和删除的节点一样；**如果直接后继不是右孩子，那么删除的节点是其直接后继的那个位置对应节点**

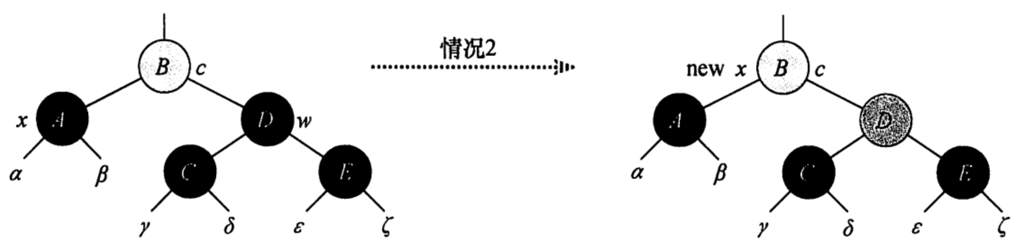
实际删除节点和删除节点不同是由于，对于需要“远程上位替代”的情况，可以通过渲染颜色保证上位的节点完美替代原来的颜色，而其原右子树上位节点离开的位置可能出现问题

- **只有当删除节点为黑色时，才有可能需要调整**，如果是红色，这个节点被删除一定不会影响黑高
- 调整时对于上位的节点，将其看为携带多一层黑色（即红黑色或二重黑色），红黑色直接染红即可，二重黑色需要进行旋转调整，向上传递黑色

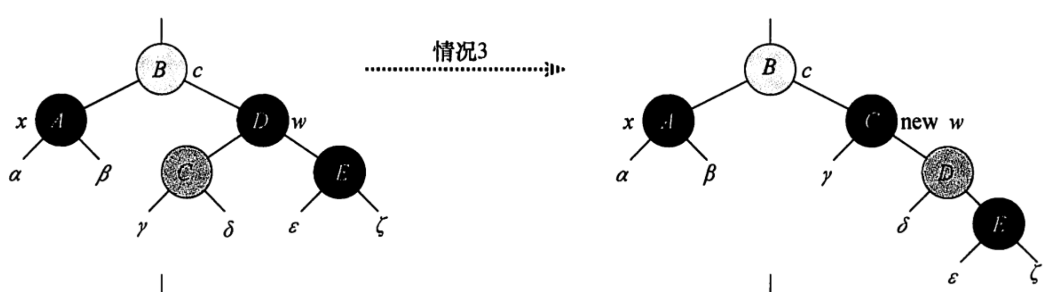
- 删除后调整步骤
  - 如果当前节点是红色或者当前节点是根节点，直接染黑后退出
  - 否则，假设当前节点是其父亲左孩子（右孩子直接相反即可）
    1. 如果其兄弟也为红色，**将其父亲的黑色与兄弟的红色对换**。由于兄弟为红色，故其两个孩子均为黑色。这时**基于父亲做一次左旋，使得其兄弟更改其侄子，也就是为黑色节点**，转化为情况2



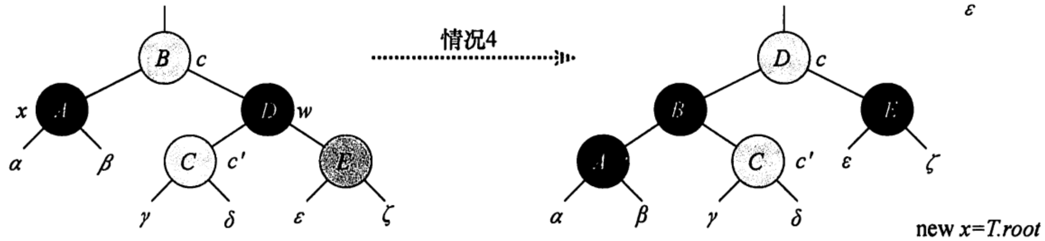
    2. 如果其兄弟为黑色，且其兄弟两个孩子均为黑色，那么**将其兄弟染为红色，将调整指针指向其父进一步调整**



    3. 如果其兄弟为黑色，且兄弟左孩子是红色，右孩子为黑色，那么**交换其兄弟的黑色和兄弟左孩子的红色，基于兄弟做一次右旋，重新定位其兄弟，转换为情况4**



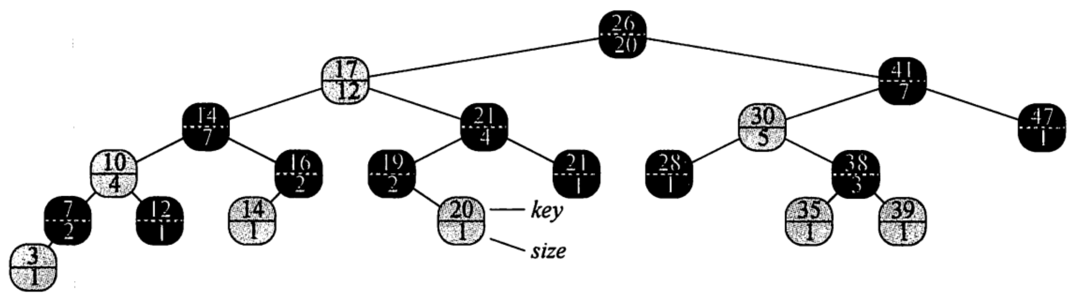
    4. 如果其兄弟为黑色，且兄弟右孩子为红色，那么**将其兄弟的黑色转移给其父亲和兄弟右孩子，基于其父亲做一次左旋**，解决了黑色重叠问题。将指针指向root以跳出循环



Chapter 14 数据结构的扩张

使用红黑树进行顺序统计

$$x.size = x.left.size + x.right.size + 1$$



- 每个节点维护一个size变量，这个变量记录了左右子树及其自身的节点个数总和，**也就是中序遍历其所在子树时最先遍历多少个节点**（这里体现了**顺序统计量**）
- 插入时，在插入路径上所有size加一，调整时只需要在旋转时重新按照公式统计被旋转节点的size即可
- 删除时，在删除节点到根节点所有size减一，调整时只需要在旋转时重新计算即可

查找第i小关键字

```
OS-SELECT(x, i)
    r = x.left.size + 1
    if i == r
        return x
    elseif i < r
        return OS-SELECT(x.left, i)
    else return OS-SELECT(x.right, i-r)
```

- i-r的原因：总排名第i的节点，**在其左子树和该节点处已经有r个较小的关键字了，所以应该是右子树的第i-r小的关键字**

确定元素的秩

```
OS-RANK(T, x)
r = x.left.size+1
y = x
while y ≠ T.root
    if y == y.p.right
        r = r + y.p.left.size+1
    y = y.p
return r
```

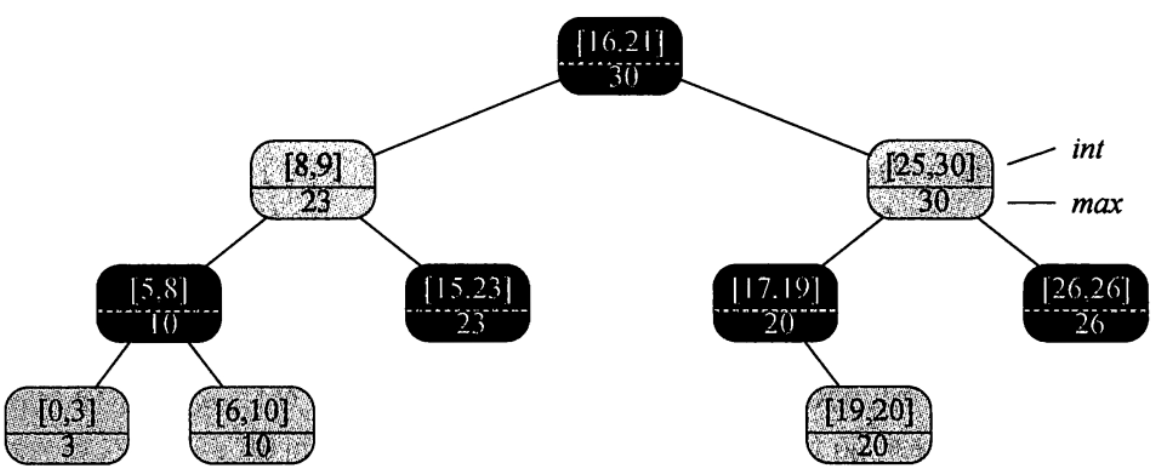
- 只有当节点是右孩子时，才需要加上其兄弟的size+1，如果是左孩子，那么中序遍历会优先遍历

红黑树的扩张

- 设f是 n 个结点的红黑树 T 扩张的属性，且假设**对任一结点 x， f的值仅依赖于结点 x 、x.left 和 x.right 的信息，还可能包括 x.left.f 和 x.right.f**。那么，我们可以在插入和删除操作期间对 T 的所有结点的f值进行维护，并且不影响这两个操作的O(lgn) 渐近时间性能。
- 黑高可以维护，但是深度不可维护

区间树

$$x.max = \max(x.int.high, x.left.max, x.right.max)$$



- 区间树由区间的low作为索引，并维护了max域，记录了**其左右孩子的max与自身区间high的最大值**
- max用以实现区间的二叉搜索：

```
INTERVAL-SEARCH(T, i)
x = T.root
while x ≠ T.nil and i does not overlap x.int
    if x.left ≠ T.nil and x.left.max ≥ i.low
        x = x.left
    else x = x.right
return x
```

- 对于旋转操作，只会影响两个被旋转节点的max域，重新计算即可
- 对于插入操作，只需要在向下查找叶节点时顺带更新这条路径上所有节点max域即可
- 对于删除操作，只需要删除后，从删除点向上到根节点维护一遍max域即可