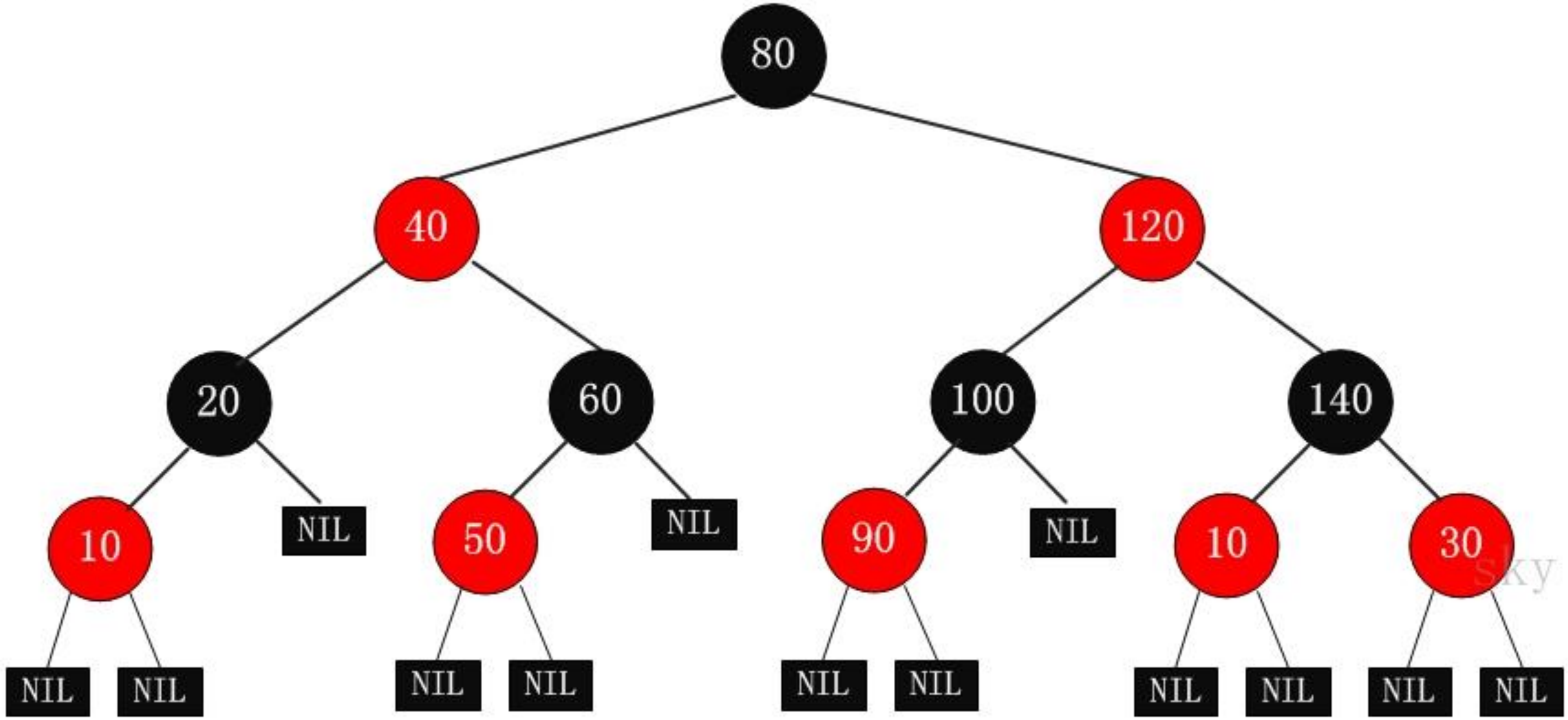
红黑树

吉林大学计算机学院
谷方明
fmgu2002@sina.com

红黑树简介

- 全称**Red-Black Tree**，缩写**RB Tree**，是一种特殊的二叉查找树，
  - ✓ 1972年由Rudolf Bayer发明，当时被称为symmetric binary B-trees。1978年，被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的“红黑树”
- 核心思想：每个结点都增加存储位表示结点的颜色：**Red** 或 **Black**。通过约束结点颜色，确保没有一条路径比其它路径长出**2**倍，因而近似平衡。

红黑树的例子



80
40
120
20
60
100
140
10
NIL
50
NIL
90
NIL
10
30
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL

红黑树

红黑树的定义

- 一棵红黑树是满足下述红黑性质的二叉查找树

1. 每个结点或是红色的，或是黑色的；
2. 根结点是黑色的；
3. 每个叶结点（NIL或NULL）是黑色的；
4. 若一个结点是红色的，则其两个孩子都是黑色的；
5. 每个结点到其后代叶结点的简单路径上，均包含相同数目的黑色结点。

红黑树的黑高

- 黑高(black-height)：结点x到达任意一个后代叶结点的一条简单路径上的黑色结点个数（不含x），记为bh(x)。
  - NIL的黑高为0
- 红黑树的黑高：根结点的黑高。
- 性质：以结点x为根的子树至少包含$2^{bh(x)}-1$个内结点；

- 证明：对x的高度进行归纳。
- **基础步骤**：h(x) = 0，x必为叶结点(nil)，以x为根的子树包含$2^0-1=0$个内结点。bh(x) =0，成立
- **归纳步骤**：h(x)>0时，x为内结点且有两个子结点，每个子结点有黑高bh(x)或bh(x)-1，取决于x的孩子的颜色。子结点的高度比x的高度低，由归纳假设，每个子结点为根的子树至少含有$2^{bh(x)-1}-1$个内结点。于是以x为的子树至少包含$2(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$个内结点

红黑树的高度

- **定理：一棵有n个内结点的红黑树的高度至多为2log(n+1).**

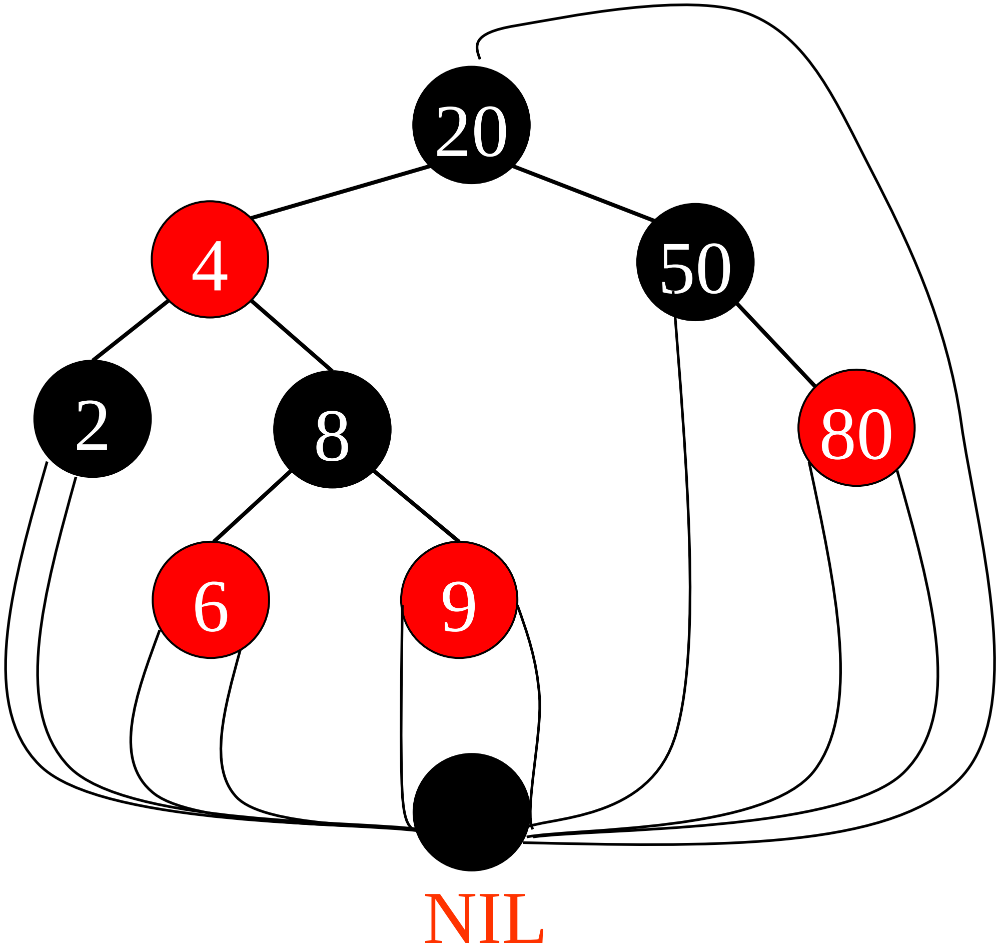
- 证明：

　设**h**为树的高度，根据性质**4**，从根到叶结点（不含根结点）的任意一条简单路径上都至少有一半的结点为黑色。因此，根的黑高至少为**h/2**; 于是，**n**$\geq 2^{h/2}-1$. 整理得： $h \leq 2\log(n+1)$

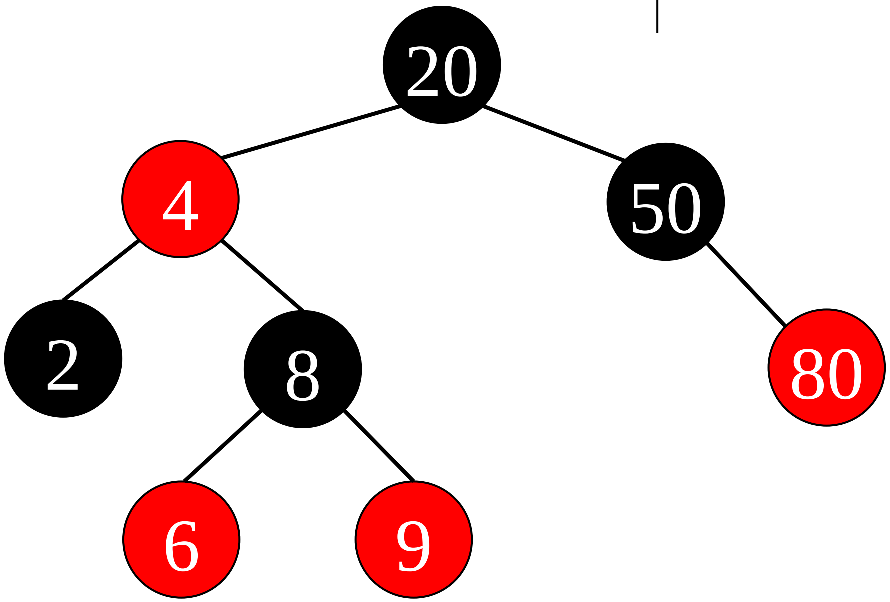
红黑树的存储结构

- **每个结点包含如下属性：color、key、left、right、p(双亲结点)等。**

- **哨兵结点NIL：为了便于处理红黑树。**
  - ✓ 所有结点的NIL孩子都指向NIL结点（节省空间）；
  - ✓ 根结点的父亲为NIL结点；
  - ✓ NIL结点的color为黑色；key、left、right、p等根据需要设置
  - ✓ 通常只关注内结点



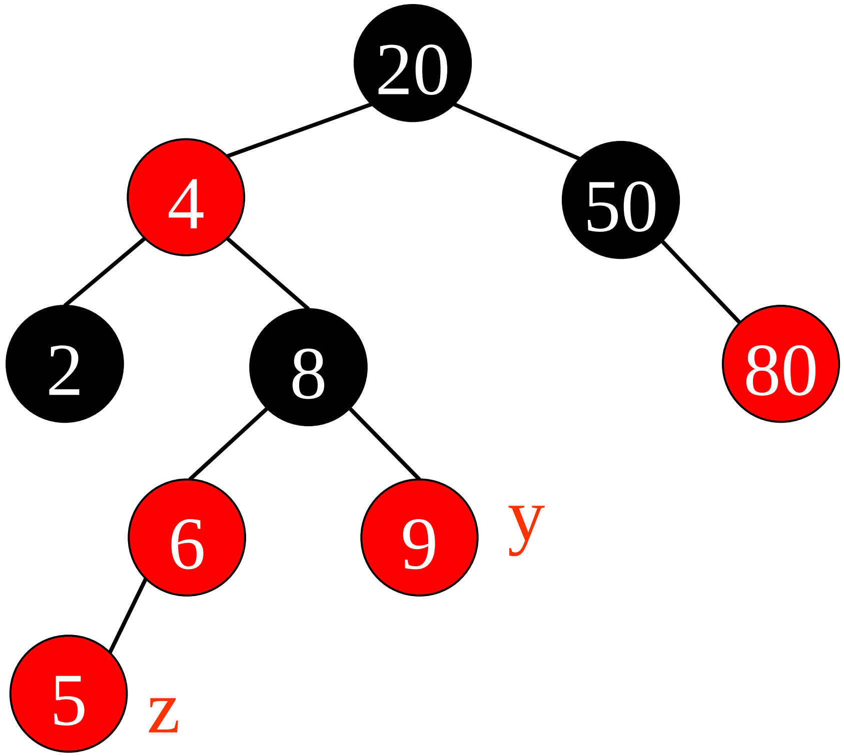
显示NIL结点



关注内结点

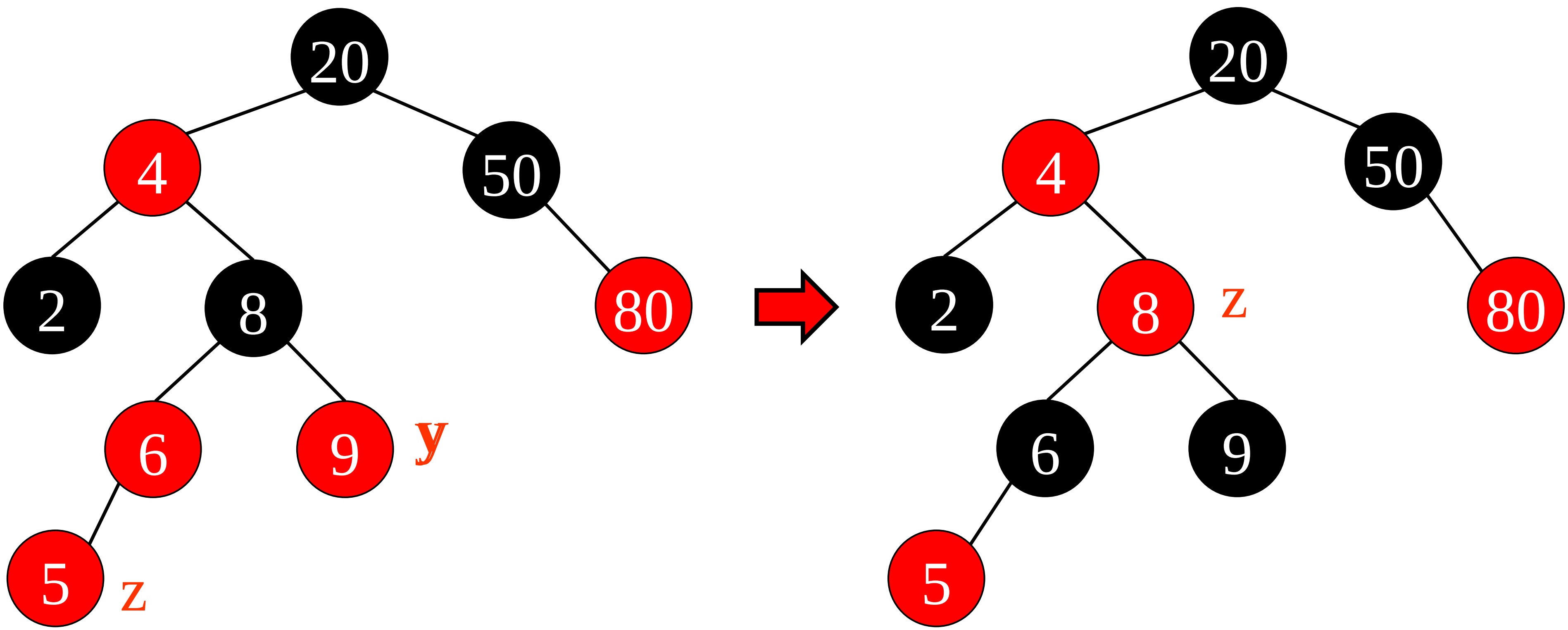
红黑树的插入操作

- 与普通BST插入类似
- 插入结点z着为红色
  - 思考：着黑色会怎样
- 若p(z)为红色，则要调整（着色和旋转），保证红黑性质。
  - 左右对称（左侧为例）
  - 设y为z的叔结点。



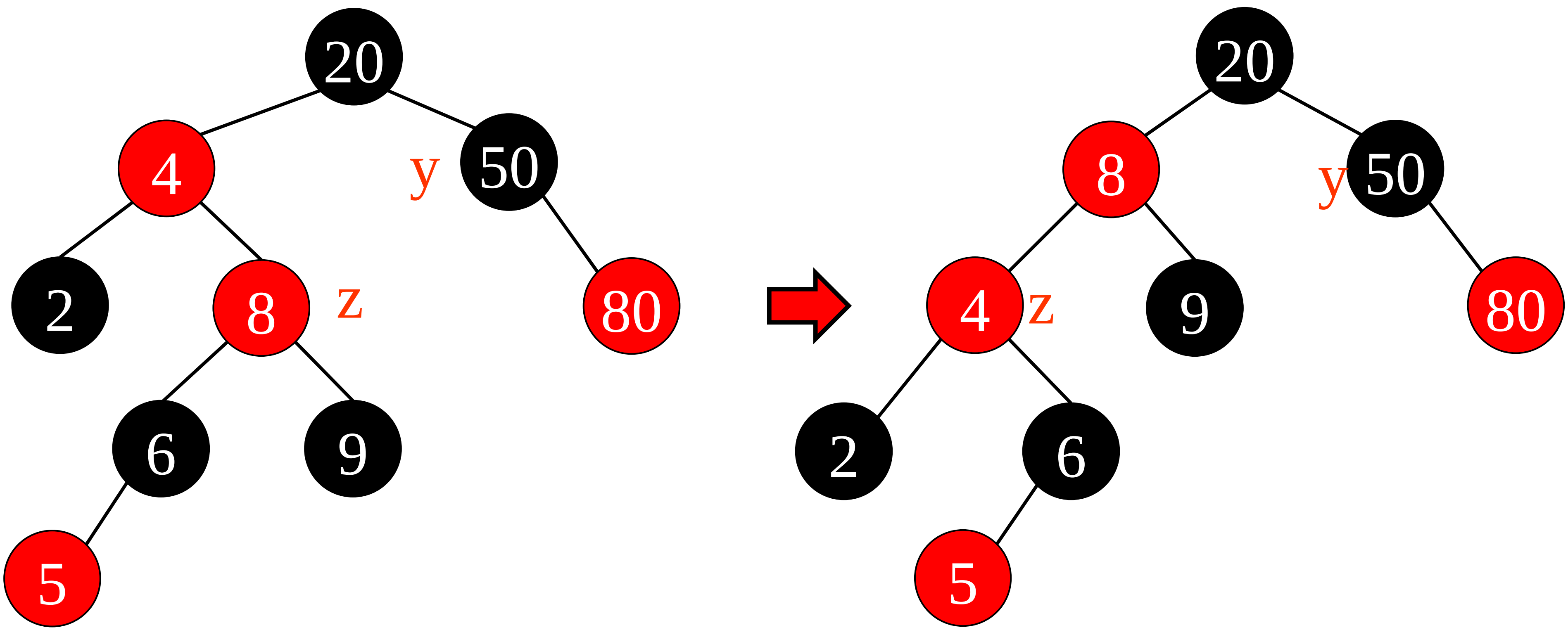
插入调整：Case 1

- 若y为红色
  - ✓ p(z)和y着黑色；p(p(z))着红色，作为新z



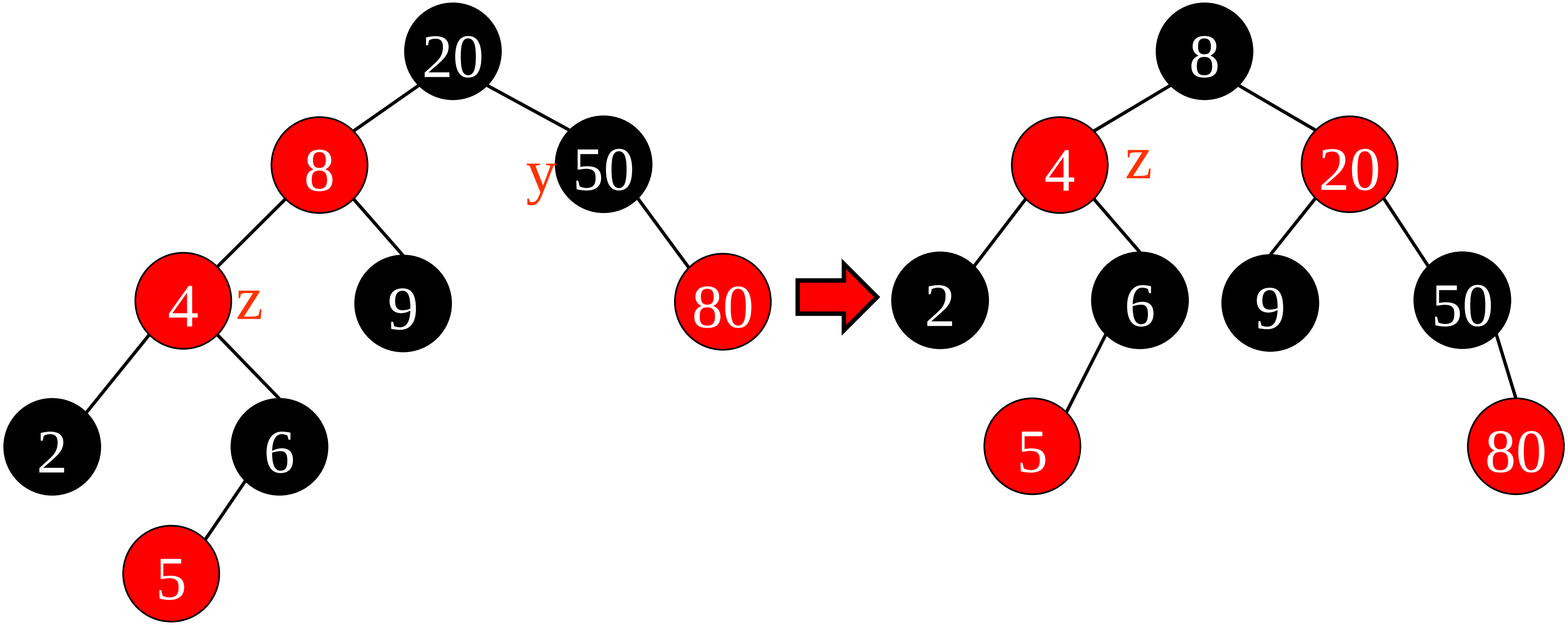
插入调整：Case 2

- 若y为黑色，且z是p(z)右孩子（之字型）
  - ✓ z=p(z)，左旋z（转化成一字型）



插入调整：Case 3

- 若y为黑色，且z是p(z)左孩子（一字型）
  - ✓ p(z)着黑色；p(p(z))着红色，右旋p(p(z))；调整结束



红黑树的插入调整参考实现

```
void rb_insert_fix(int z){ //assert(z!=0&&color[z]==RED)
	int y;
	while(color[pa[z]]==RED){
		if(pa[z]==lc[pa[pa[z]]]){
			y = rc[pa[pa[z]]];
			if(color[y]==RED){   //case 1
				color[pa[z]] = BLACK;
				color[y] = BLACK;
				color[pa[pa[z]]] = RED;
				z = pa[pa[z]];
			}else{                    //case 2
```

```
                    if(z==rc[pa[z]]){
                        z = pa[z];
                        left_rotate(z);
                    }
                    color[pa[z]] = BLACK; //case 3
                    color[pa[pa[z]]] = RED;
                    right_rotate(pa[pa[z]]);
                }
            }else{
                //pa[z]==rc[pa[pa[z]]]; swap: l and r
            }
        }
        color[root] = BLACK;
}
```

红黑树插入操作的分析

- 插入保证：红黑树性质1、3
- 调整保持：红黑树性质4、5
- 调整结束：红黑树性质2

- 红黑树插入操作的时间复杂度O(logn)
  - 调整时，情况1，z沿着红黑树上升两层，while才会循环执行，while执行的总次数为O(logn)；情况2和3都最多执行1次，因此调整总花费O(logn)
  - 插入时，时间花费O(logn)

红黑树的删除操作

- 与普通BST删除类似
- 设z为被删除结点，y为被移出的结点

  y = z

  if(lc[z]!=NIL && rc[z]!=NIL) y=minimum(rc[z]);
- 设x为替换y的结点

  x = (lc[y]!=NIL) ? lc[y] : rc[y];

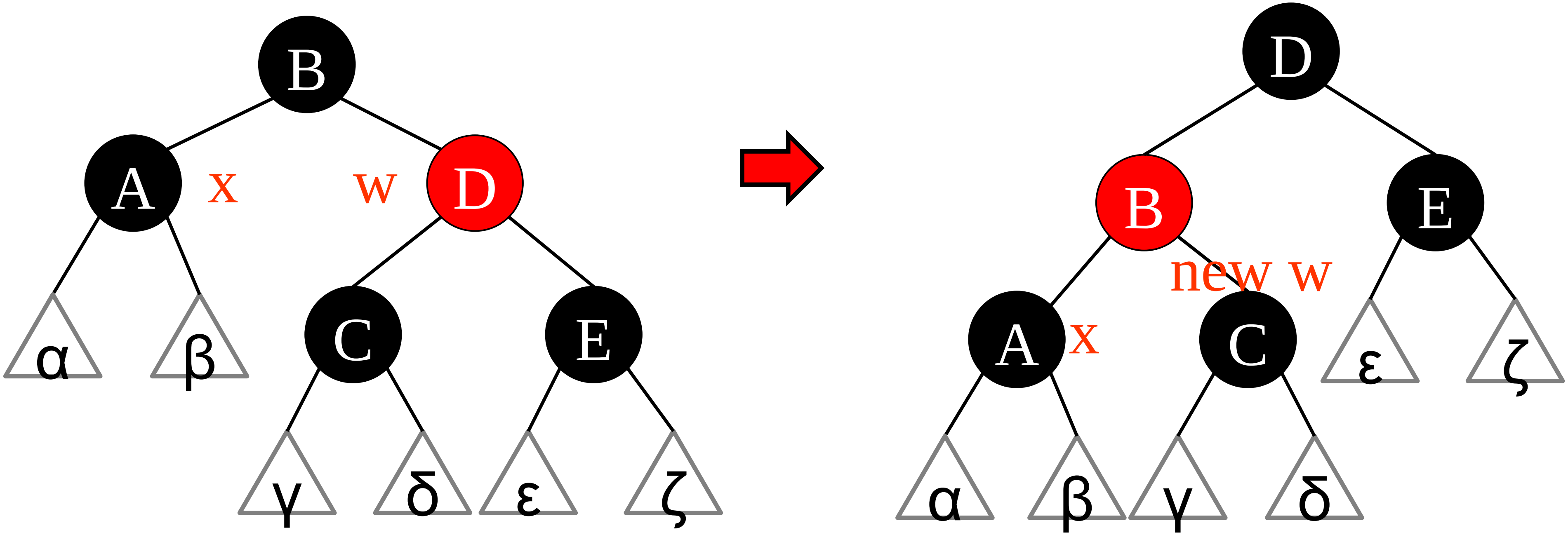
- 若color(y) 原为黑色，则从x开始调整；若color(y)原为红色，红黑性质仍保存，不必调整。

红黑树删除的调整规则

- 若color(y) 原为黑色，则
  1. 若y为根结点，x是红色成为新根，破坏性质(2)
  2. 若x和p(y)(现为p(x))都为红色，破坏性质(4)
  3. y的移出导致原来包含y结点(现为x)的路径都少了一个黑结点，破坏性质(5)。
- 调整方案
  - ✓ 若x是根结点 或 为红色，则将x着黑色，结束
  - ✓ 否则，分情况讨论。由对称性，以左侧为例(x为其父亲的左孩子)，w是x的兄弟

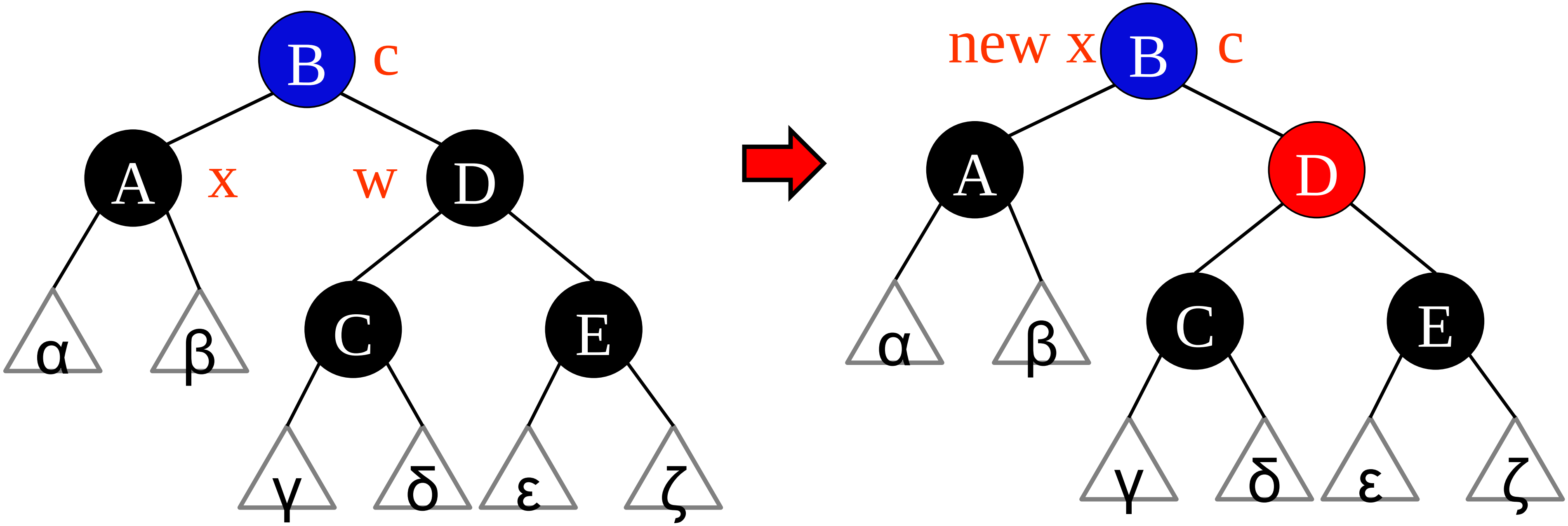
删除调整：case 1

- w为红色
  - w着黑色，pa(x)着红色，左旋pa(x)， w=rc(pa(x))
  - 转为case 2，3，4（转换为黑）



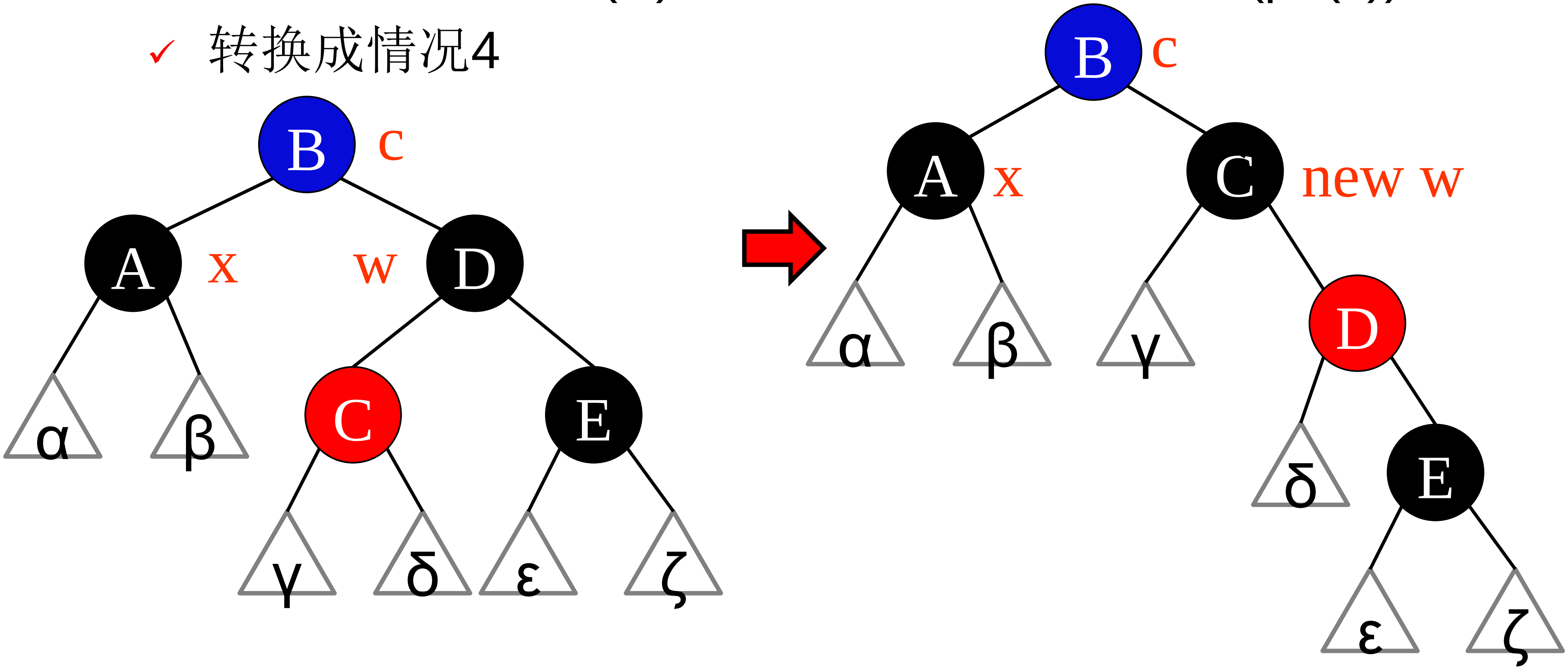
删除调整：case 2

- w为黑色，且w的两个孩子都为黑色
  - 将w着为红色；x = pa(x)，循环处理



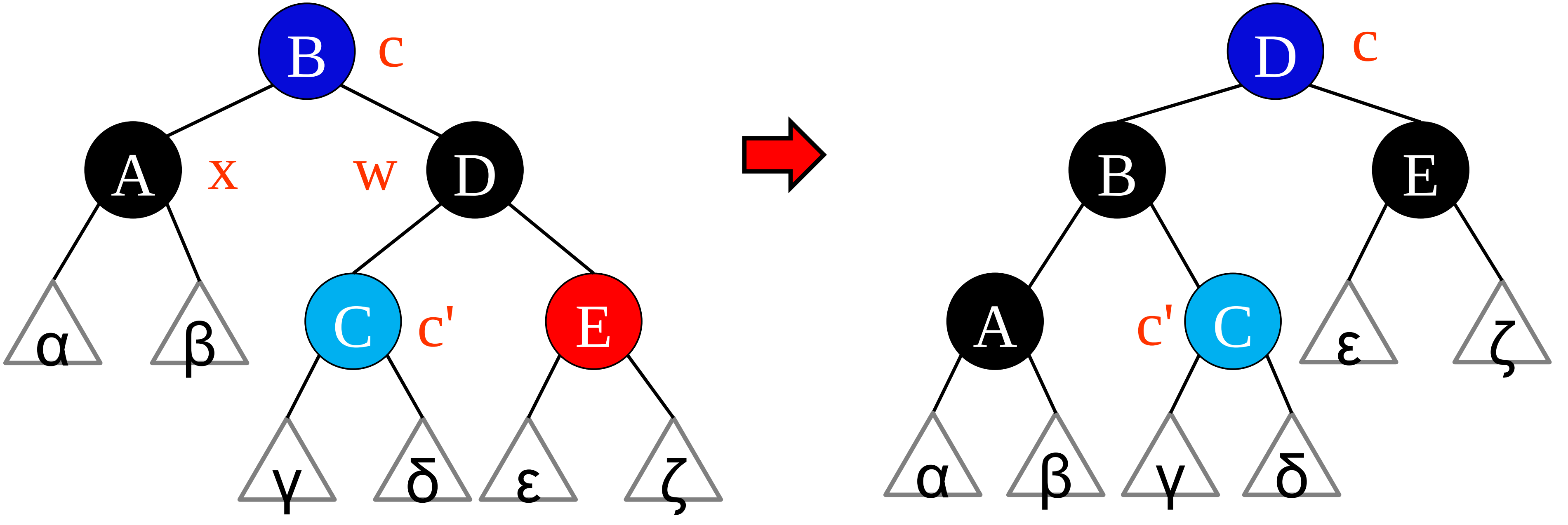
删除调整：case 3

- w为黑色，且w的左孩子红（必然）、右孩子黑
  - ✓ 将w着为红色；lc(w)着为黑色；右旋w；w=rc(pa(x))
  - ✓ 转换成情况4



删除调整：case 4

- w为黑色，且w的右孩子红
  - ✓ 将w着为pa(x)色；pa(x)、 rc(w)着黑色；左旋pa(x)；
  - ✓ 调整结束：x = root（或者break）



红黑树的删除调整参考实现

```
void rb_delete_fix(int x){//assert(x!=0)
	int w;
	while(x!=root&&color[x]==BLACK){
		if(x==lc[pa[x]]){
			w = rc[pa[x]];
			//case 1
			if(color[w]==RED){
				color[w] = BLACK;
				color[pa[x]] = RED;
				left_rotate(pa[x]);
				w = rc[pa[x]];
			}
```

```
if(color[lc[w]]==BLACK &&
color[rc[w]]==BLACK){     //case 2
        color[w] = RED;
        x = pa[x];
}else{
        //case 3
        if(color[rc[w]]==BLACK){
                color[lc[w]]=BLACK;
                color[w] = RED;
                right_rotate(w);
                w = rc[pa[x]];
        }
        //case 4
```

```
                    color[w] = color[pa[x]];
                    color[pa[x]] = BLACK;
                    color[rc[w]] = BLACK;
                    left_rotate(pa[x]);
                    x = root;
                }
            }else{
             // x==rc[pa[x]]); swap: l and r
            }
        }
        color[x]=BLACK;
}
```

红黑树删除操作的分析

- 删除调整方案中，**case 1**可以转为**case 2**、**3**和**4**。调整方案或者通过**case 3**、**4**结束，此时能恢复破坏的性质**(2)**、**(4)**和**(5)**；或者通过**case 2**循环结束，此时将将**x**着黑色，也能恢复破坏的性质**(2)**、**(4)**和**(5)**；
- 红黑树删除操作的时间复杂度**O(logn)**
  - 调整时，情况**1**可转为情况**2**、**3**、**4**；情况**3**和**4**最多各执行一次；情况**2**是**while**循环可以重复执行的唯一情况，指针**x**沿树上升至多**logn**次；因此调整总花费**O(logn)**
  - 删除时，时间花费**O(logn)**

红黑树小结

- 效率较高：由红黑树的高度定理得知，动态操作集合**SEARCH**、**MININUM**、**MAXIMUM**、**SUCCESSOR**和**PREDECESSOR**都可在红黑树上以**O(logn)**的时间执行。由前述分析可知，**INSERT**和**DELETE**也可在**O(logn)**的时间做到。
- 实现较难
- 典型应用：**map(STL)**