



# 福昕高级PDF编辑器

高效 · 安全 · 专业

立即下载

点击购买



OFFICE格式互转



OCR文字识别



文本图像编辑



加密和签署



交互式动态表单



互联PDF文档



# 福昕高级PDF编辑器

高效 · 安全 · 专业

立即下载

点击购买



OFFICE格式互转



OCR文字识别



文本图像编辑



加密和签署



交互式动态表单



互联PDF文档



中国科学技术大学 计算机科学与技术系  
University of Science and Technology of China  
DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

# 算法设计与分析

## Design and Analysis of Algorithms

主讲人 徐云

Fall 2018, USTC



Part 1 Foundation

Part 2 Sorting and Order Statistics

Part 3 Data Structure

chap 10 Elementary Data Structures

chap 11 Hash Tables

chap 12 Binary Search Trees

**chap 13 Red-Black Trees**

chap 14 Augmenting Data Structures

Part 4 Advanced Design and Analysis Techniques

Part 5 Advanced Data Structures

Part 6 Graph Algorithms

Part 7 Selected Topics

Part 8 Supplement



## 第13章 红黑树

### 13.1 红黑树的性质

### 13.2 旋转

### 13.3 插入

### 13.4 删除



# 13.1 红黑树的性质

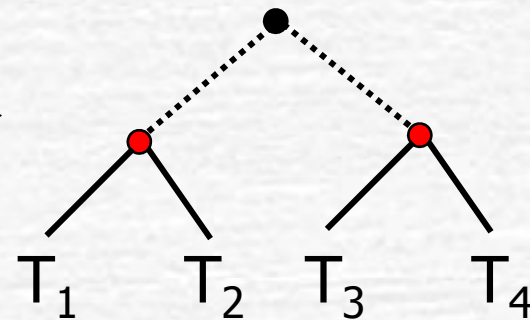
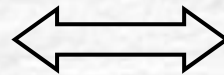
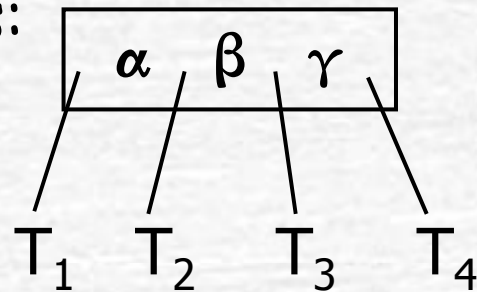
- 背景知识
- 红黑树的定义
- 一个图例
- 黑高的定义
- 关于高度的一个引理



# 背景知识 (1)

- 树的高度决定了树上操作的成本，一些搜索树的高度如下：
  - 平衡二叉搜索树:  $O(\log n)$
  - 1962年提出的AVL树:  $\leq 1.44 \log n$   
平衡二叉查找树
  - 1972年提出的红黑树:  $\leq 2 \log(n+1)$
- 4阶B树:  $\#key(1\sim 3), \#subtree(2\sim 4) \Leftrightarrow$  红黑树 //转化

3 keys:

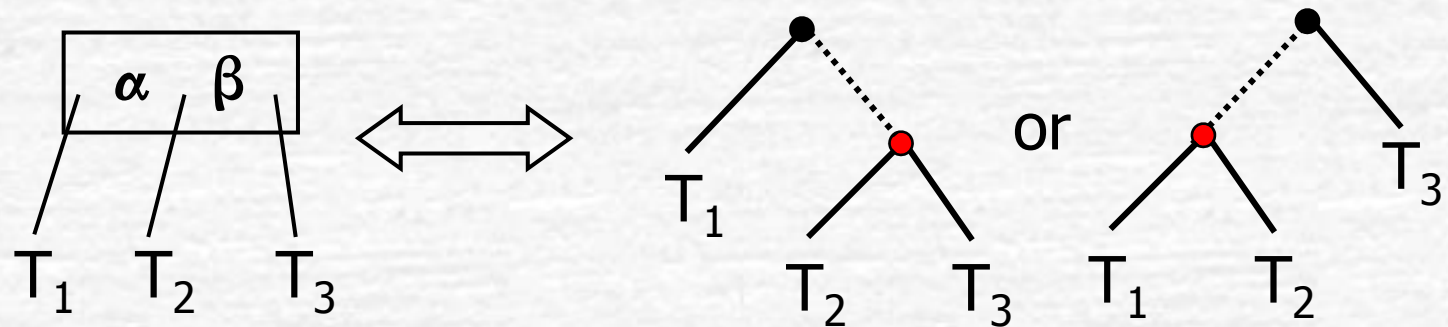


—— B树边

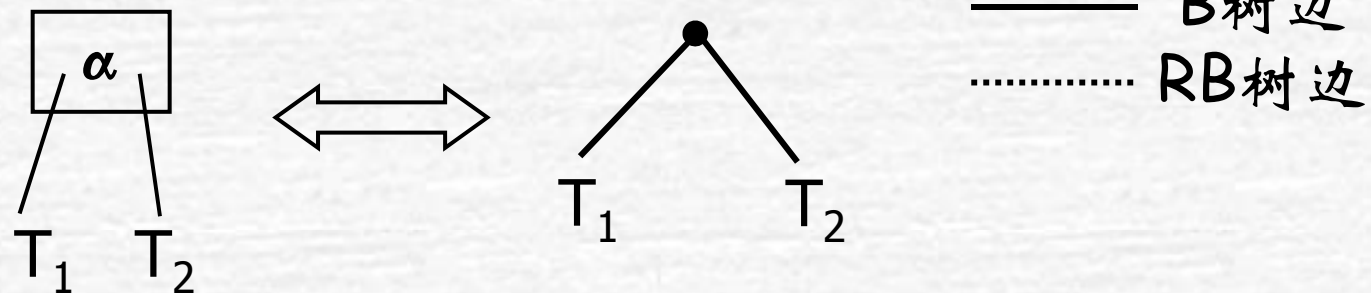
..... RB树边

## 背景知识 (2)

2 keys:



1 key:



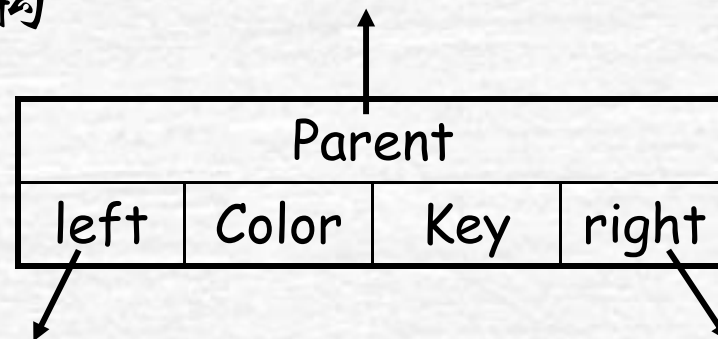


# 红黑树的定义

- Def. 1: **红黑树**是满足下述性质的二叉搜索树

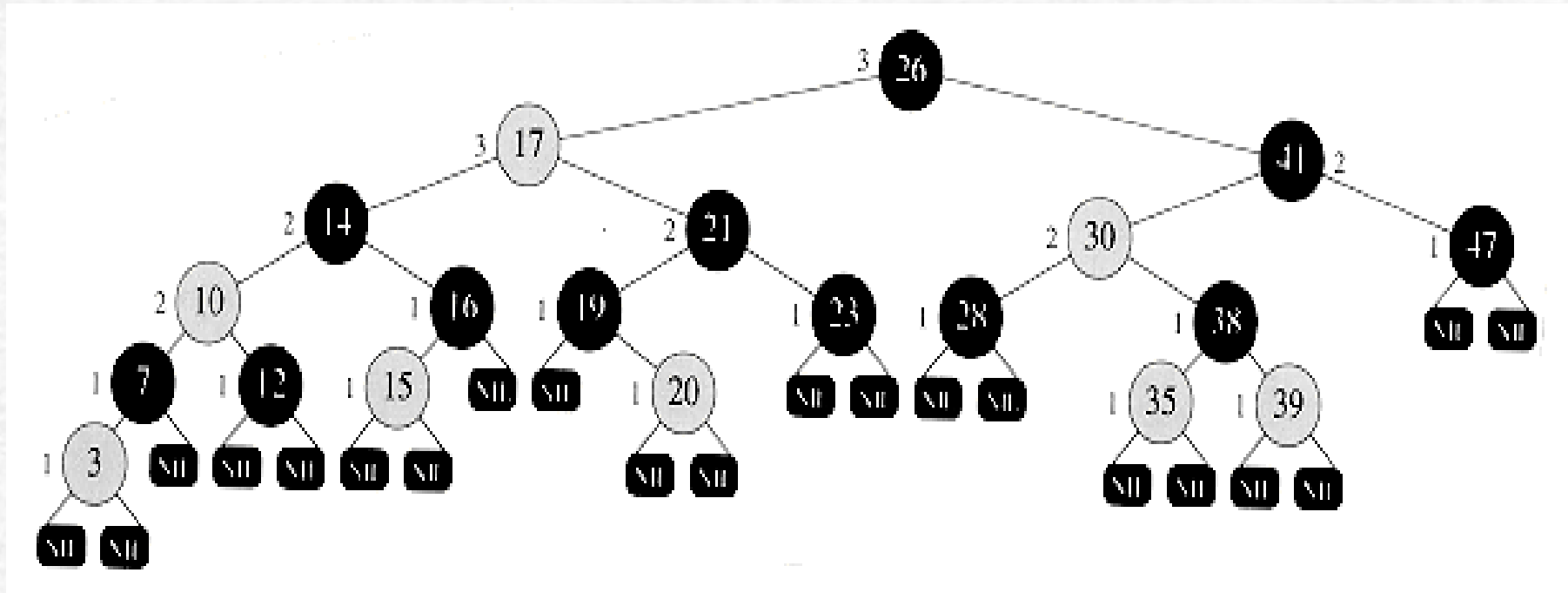
- ① 每个节点必须为红色或黑色; //性质1
- ② 根为黑色; //性质2
- ③ 树中的nil叶子为黑; //性质3
- ④ 若节点为红, 则其两个孩子必为黑; //性质4
- ⑤ 每节点到其后代叶子的所有路径含有同样多的黑节点; //性质5

- 节点的结构



# 一个图例

- Fig 13.1(a)



表达方式:

图(a)每个空指针域均连接到一个叶节点nil, 比较浪费存储空间;

图(b)所有空指针域共享一个哨兵nil[T], nil[T]为黑色;

图(c)省略nil[T];

# 黑高的定义

- Def. 2: 节点 $x$ 的**黑高** $bh(x)$ 是该节点到它的任何后代叶子路径上的**黑节点数**(不包括 $x$ 本身)  
注: Fig 13.1(a)中节点旁的数字
- Def. 3: **红黑树的黑高**是根的黑高, 记 **$bh(\text{root}[T])$**



# 关于高度的一个引理 (1)

- Lemma 13.1: 一棵 $n$ 个内点的红黑树的高度至多是 $2\log(n+1)$ 。

- Proof:

① 先证对任何以 $x$ 为根的子树其内节点数 $\geq 2^{bh(x)}-1$

归纳基础: 当 $bh(x)=0$ 时,  $x$ 就是 $nil[T]$

$\therefore 2^{bh(x)}-1 = 2^0-1=0$  即为0个内节点, 正确

归纳假设: 对 $x$ 的左右孩子命题正确

归纳证明:  $\because x$ 的左右孩子的黑高或为 $bh(x)$ 或为 $bh(x)-1$

$$\begin{aligned}\therefore x \text{的内点数} &= \text{左孩子内点数} + \text{右孩子内点数} + 1 \\ &\geq (2^{bh(x)-1}-1) + (2^{bh(x)-1}-1) + 1 \\ &= 2^{bh(x)}-1\end{aligned}$$

即第①点得证。

# 关于高度的一个引理 (2)

- Proof(Cont.):

② 证明  $bh(\text{root}[T]) \geq h/2$ ,  $h$  为红黑树的树高

$\because$  红点的孩子必为黑 // 红黑树的性质4

$\therefore$  红点的层数  $< h/2$

因此  $\Rightarrow bh(\text{root}[T]) \geq h/2$

③ 证明最后结论

$\because$  红黑树有  $n$  个内点

由①  $\Rightarrow n \geq 2^{bh(\text{root}[T])} - 1 \geq 2^{h/2} - 1$

$\therefore \Rightarrow h \leq 2\log(n+1)$







## 第13章 红黑树

### 13.1 红黑树的性质

### 13.2 旋转

### 13.3 插入

### 13.4 删除

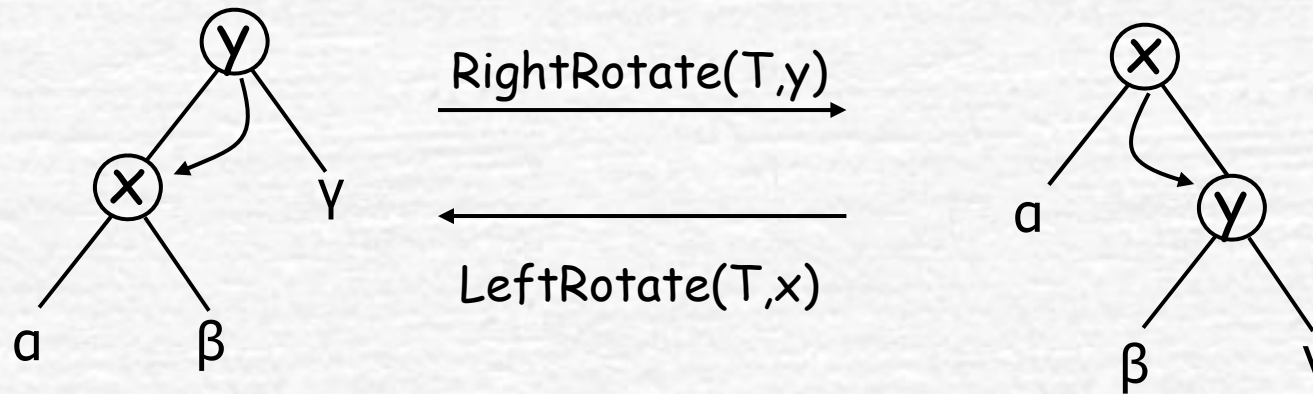


## 13.2 旋转

- 左、右旋转定义
- 左旋实现的步骤
- 左旋算法

# 左、右旋的定义

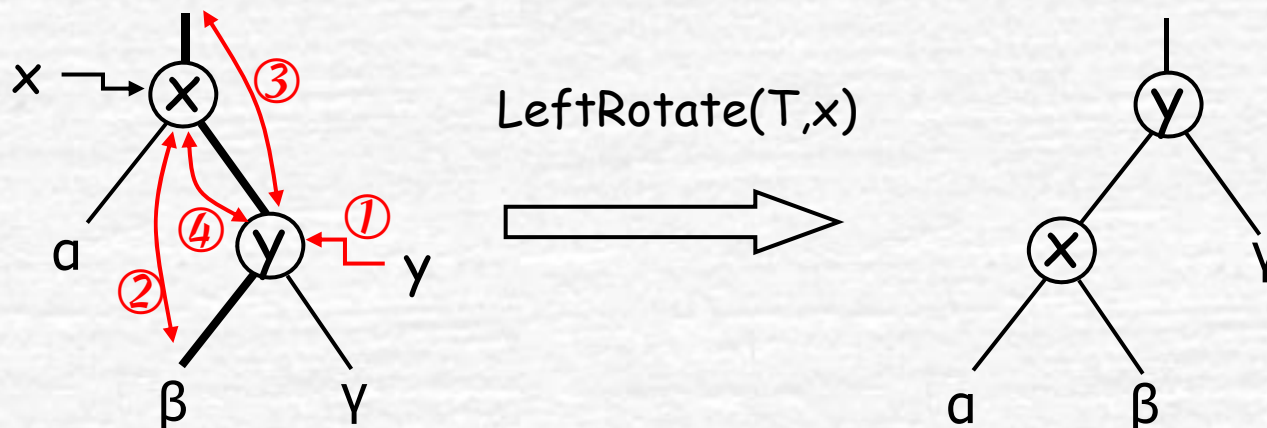
- 左、右旋转的图示



注：旋转过程中二叉搜索树(BST)性质不变： $a \leq x \leq \beta \leq y \leq \gamma$

# 左旋实现的步骤

- 左旋图示:



- 步骤解释: 需要变动的是3根粗链

临界情形

①  $y \leftarrow \text{right}[x]$  //记录指向y节点的指针

②  $\text{right}[x] \leftarrow \text{left}[y], p[\text{left}[y]] \leftarrow x$  //β连到x右

$\beta = \text{nil}[T]$

③  $p[y] \leftarrow p[x], p[x]$ 的左或右指针指向y //y连到p[x]  $P[x] = \text{nil}[T]$ , 即x为根

④  $\text{Left}[y] \leftarrow x, p[x] \leftarrow y$  //x连到y左

根

注: - 要注意先后顺序; - 每条边的修改涉及双向;

- 要考虑临界情形(特殊情形);



# 左旋算法

```
LeftRotate(T, x)
{ //假定right[x] ≠ nil[T]
  //step ①
  y ← right[x];
  //step ②
  right[x] ← left[y]; p[left[y]] ← x;
  //step ③
  p[y] ← p[x];
  if p[x]=nil[T] then      //x是根
    root[T] ← y;          //修改树指针
  else if x=left[p[x]] then left[p[x]] ← y;
    else right[p[x]] ← y;
  //step ④
  left[y] ← x; p[x] ← y;
}
T(n)=O(1)
```





## 第13章 红黑树

13.1 红黑树的性质

13.2 旋转

**13.3 插入**

13.4 删除



## 13.3 插入

- 算法步骤
- RBInsert 算法
- RBInsertFixup 算法

# 算法步骤

- step 1: 将 $z$ 节点按BST树规则插入红黑树中,  
 $z$ 是叶子节点;
- step 2: 将 $z$ 涂红;
- step 3: 调整使其满足红黑树的性质;

# RBInsert算法 (1)

```
RBInsert(T, z)
{
  y ← nil[T];           //y用于记录：当前扫描节点的双亲节点
  x ← root[T];           //从根开始扫描
  while x ≠ nil[T] do    //查找插入位置
  {
    y ← x;
    if key[z] < key[x] then //z插入x的左边
      x ← left[x];
    else
      x ← right[x];        //z插入x的右边
  }
  p[z] ← y;               //y是z的双亲
  if y = nil[T] then      //z插入空树
    root[T] ← z;          //z是根
  else
    if key[z] < key[y] then
      left[y] ← z;        //z是y的左子插入
    else
      right[y] ← z;       //z是y的右子插入
  }
```



# RBInsert算法 (2)

```
left[z] ← right[z] ← nil[T];  
color[z] ← red;  
RBInsertFixup(T, z);  
}
```

时间:  $T(n) = O(\log n)$



# RBInsertFixup算法 (1)

- 调整分析

- idea: 通过旋转和改变颜色, 自下而上调整 (z进行上溯), 使树满足红黑树;

- z插入后违反情况:

- ∵ z作为红点, 其两个孩子为黑 ( $\text{nil}[T]$ )

- ∴ 满足性质1, 3, 5

- 可能违反性质2: z是根

- 可能违反性质4:  $p[z]$ 是红

- 调整步骤:

- (1) 若z为根, 将其涂黑;

- (2) 若z为非根, 则 $p[z]$ 存在

- ① 若 $p[z]$ 为黑, 无需调整

# RBInsertFixup算法 (2)

②若 $p[z]$ 为红，违反性质4，则需调整

$\because p[z]$ 为红，它不为根

$\therefore p[p[z]]$ 存在且为黑

➤ 分6种情况进行调整：

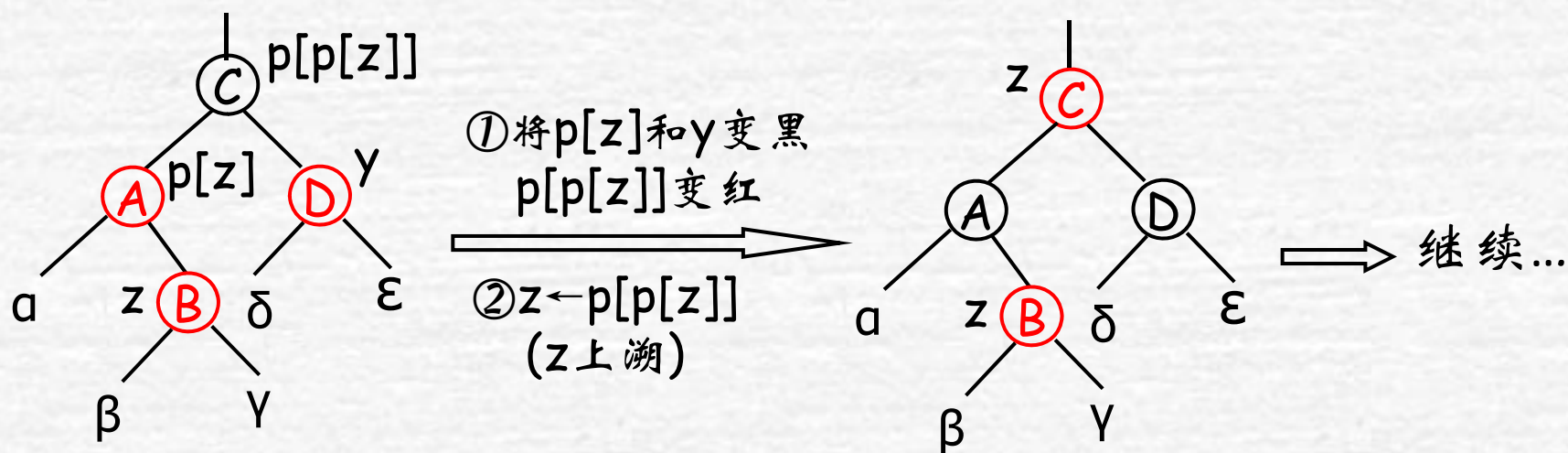
其中

case1~3为 $z$ 的双亲 $p[z]$ 是其祖父 $p[p[z]]$ 的左孩子，

case4~6为 $z$ 的双亲 $p[z]$ 是其祖父 $p[p[z]]$ 的右孩子。

# RBInsertFixup算法 (3)

Case 1:  $z$ 的叔叔 $y$ 是红色



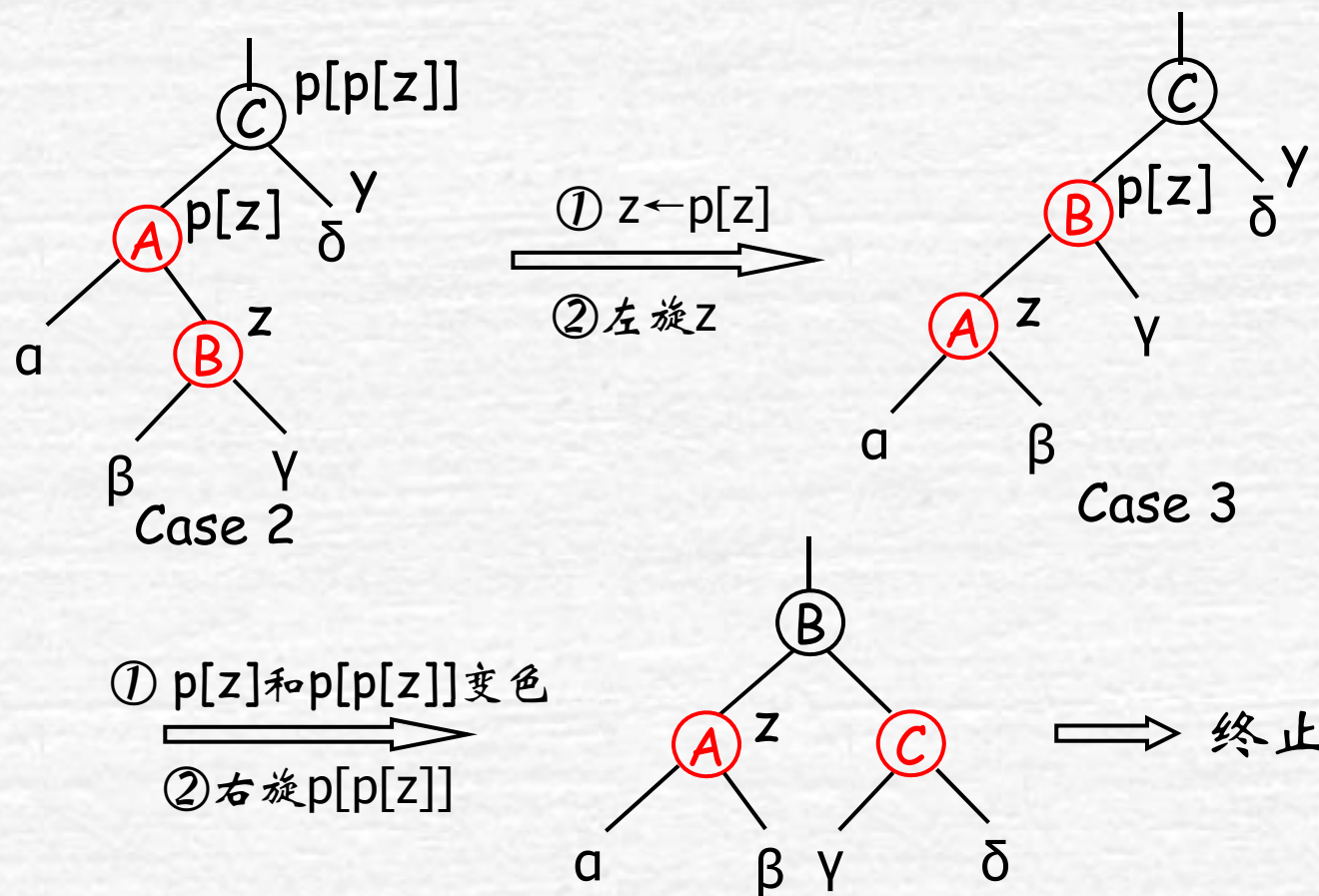
- 注: (1)变换后, 新的 $z$ (上溯后)可能违反性质4, 故调整最多至根;  
(2)若红色传播到根, 将根涂黑, 则树的黑高增1; //临界处理  
(3) $z$ 是 $p[z]$ 的左、右孩子均一样处理;



# RBInsertFixup算法 (4)

Case 2: 当 $z$ 的叔叔 $y$ 是黑色, 且 $z$ 是双亲 $p[z]$ 的右孩子

Case 3: 当 $z$ 的叔叔 $y$ 是黑色, 且 $z$ 是双亲 $p[z]$ 的左孩子



# RBInsertFixup算法 (5)

- RBInsertFixup算法

RBInsertFixup(T, z)

{ while ( color[p[z]]=red ) do

{ //若z为根, 则p[z]=nil[T], 其颜色为黑, 不进入此循环

//若p[z]为黑, 无需调整, 不进入此循环

if p[z]=left[p[p[z]]] then //case 1,2,3

{ y ← right[p[p[z]]]; //y是z的叔叔

if color[y]=red then //case 1

{ color[y]=black; color[p[z]]=black;

color[p[p[z]]]=red; z ← p[p[z]];

}

else //case 2 or case 3 y为黑



# RBInsertFixup算法 (6)

```
else    //case 2 or case 3  y为黑
{ if z=right[p[z]] then //case 2
  { z ← p[z];    //上溯至双亲
    leftRotate(T, z);
  } //以下为case 3
  color[p[z]]=black; color[p[p[z]]]=red;
  RightRotate(T, p[p[z]]); //p[z]为黑，退出循环
} //case 1's endif
} //case 2 or 3' s
else //case 4,5,6's 与上面对称
{ ... ... }
} //endwhile
color[root[t]] ← black;
}
```

# RBInsertFixup算法 (5)

- 算法的时间复杂性
  - 调整算法的时间:  $O(\log n)$
  - 整个插入算法的时间:  $O(\log n)$
  - 调整算法中至多使用2个旋转



## 第13章 红黑树

13.1 红黑树的性质

13.2 旋转

13.4 插入

13.4 删除



## 13.4 删除

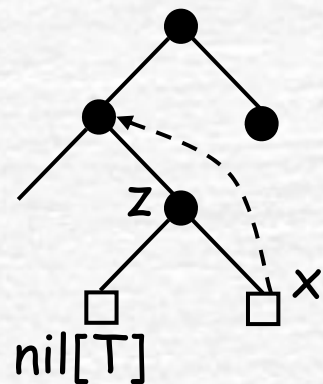
- 分析讨论
- 删除算法



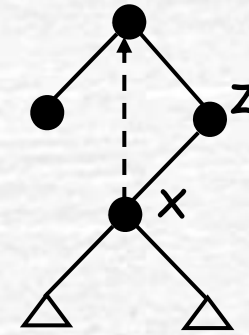
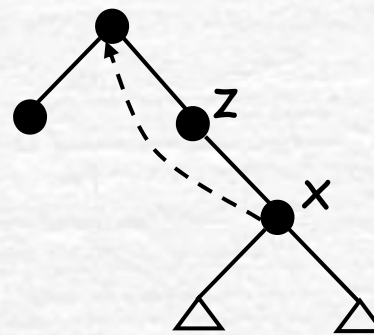
# 分析讨论 (1)

- z删除后BST的调整

➤ case 1: z为叶子;



case 2: z只有一个孩子(非空)



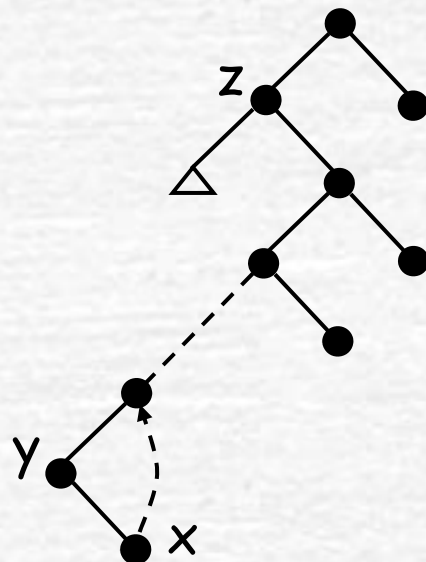
注: (1)删除z, 连接x。这里x是z的中序后继;

(2)case 1是case 2的特例, 因为处理模式是一样的。

(3)z是p[z]的左孩子, 类似讨论;

## 分析讨论 (2)

- case 3:  $z$  的两个孩子均非空;



注: (1)找 $z$ 的中序后继, 即找 $z$ 的右子树中最左下节点 $y$ ;

(2)删除 $y$ , 将 $y$ 的内容copy到 $z$ , 再将 $y$ 的右子连到 $p[y]$ 左下。

### ● RBT性质的影响

删红点不影响, 删黑点需要调整。 //这里是后面算法中 $y$ 的颜色

# 删除算法 (1)

- 删除算法

RBDelete(T, z)

{ if (left[z]=nil[T]) or (right[z]=nil[T]) then //case 1,2

    y ← z;         //后面进行物理删除y

else //z的两子树均非空, case 3

    y ← TreeSuccessor(z); //y是z的中序后继

    //此时, y统一地是x的双亲节点且是要删除节点

    //x是待连接到p[y]的节点, 以下要确定x

    if left[y] ≠ nil[T] then //本if语句综合了case1,2,3的x

        x ← left[y];

    else

        x ← right[y];

    //以下处理: 用x取代y与y的双亲连接

    p[x] ← p[y];



## 删除算法 (2)

```
if p[y]=nil[T] then //y是根
    root[T] ← x;      //根指针指向x
else //y非根
    if y=left[p[y]] then //y是双亲的左子
        left[p[y]] ← x;
    else
        right[p[y]] ← x;
if y≠z then //case 3
    y的内容copy到z;
if color[y]=black then
    RBDeleteFixup(T, x); //调整算法
return y; //实际是删除y节点
}
```



# 删除算法 (3)

- 调整算法:  $\text{RBDeleteFixup}(T, x)$

- 讨论

$x$ : 或是 $y$ 的唯一孩子; 或是哨兵 $\text{nil}[T]$

可以想象将 $y$ 的黑色涂到 $x$ 上, 于是

- 若 $x$ 为红, 只要将其涂黑, 调整即可终止;
- 若 $x$ 为黑, 将 $y$ 的黑色涂上之后,  $x$ 是一个双黑节点, 违反性质1。

处理步骤如下:

step 1: 若 $x$ 是根, 直接移去多余一层黑色(树黑高减1), 终止;

step 2: 若 $x$ 原为红, 将 $y$ 的黑色涂到 $x$ 上, 终止;

step 3: 若 $x$ 非根节点, 且为黑色, 则 $x$ 为双黑。通过变色、旋转使多余黑色向上传播, 直到某个红色节点或传到根;

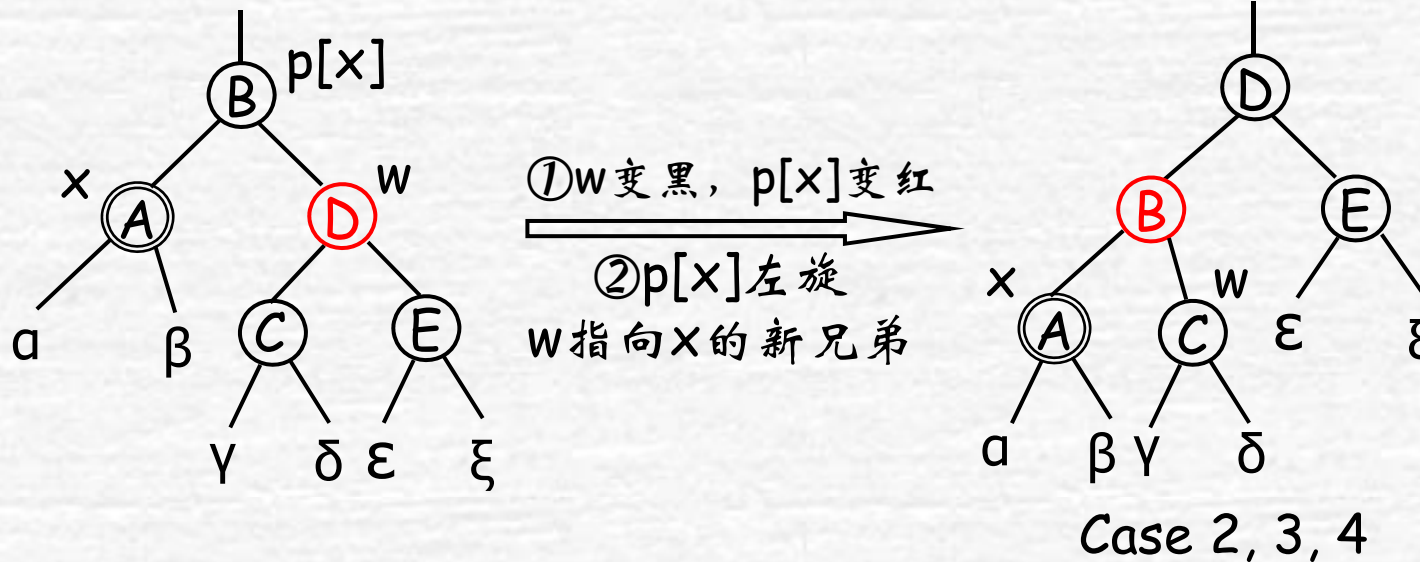
# 删除算法 (4)

## 调整分8种情况

case 1~4为 $x$ 是 $p[x]$ 的左子；case 5~8为 $x$ 是 $p[x]$ 的右子

**case 1:**  $x$ 的兄弟 $w$ 是红色

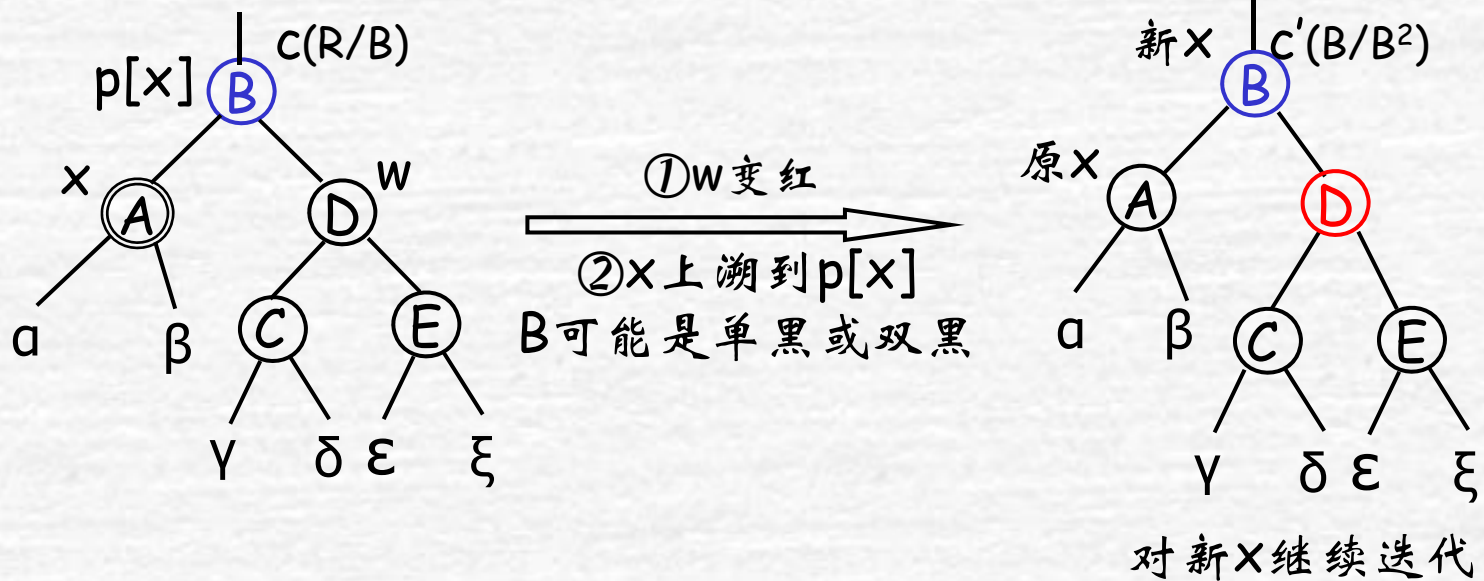
$\because w$ 是红,  $\therefore p[x]$ 必黑



**目标: 将case1转为case2,3,4处理**

# 删除算法 (5)

case 2:  $x$  的黑兄弟  $w$  的两个孩子均为黑

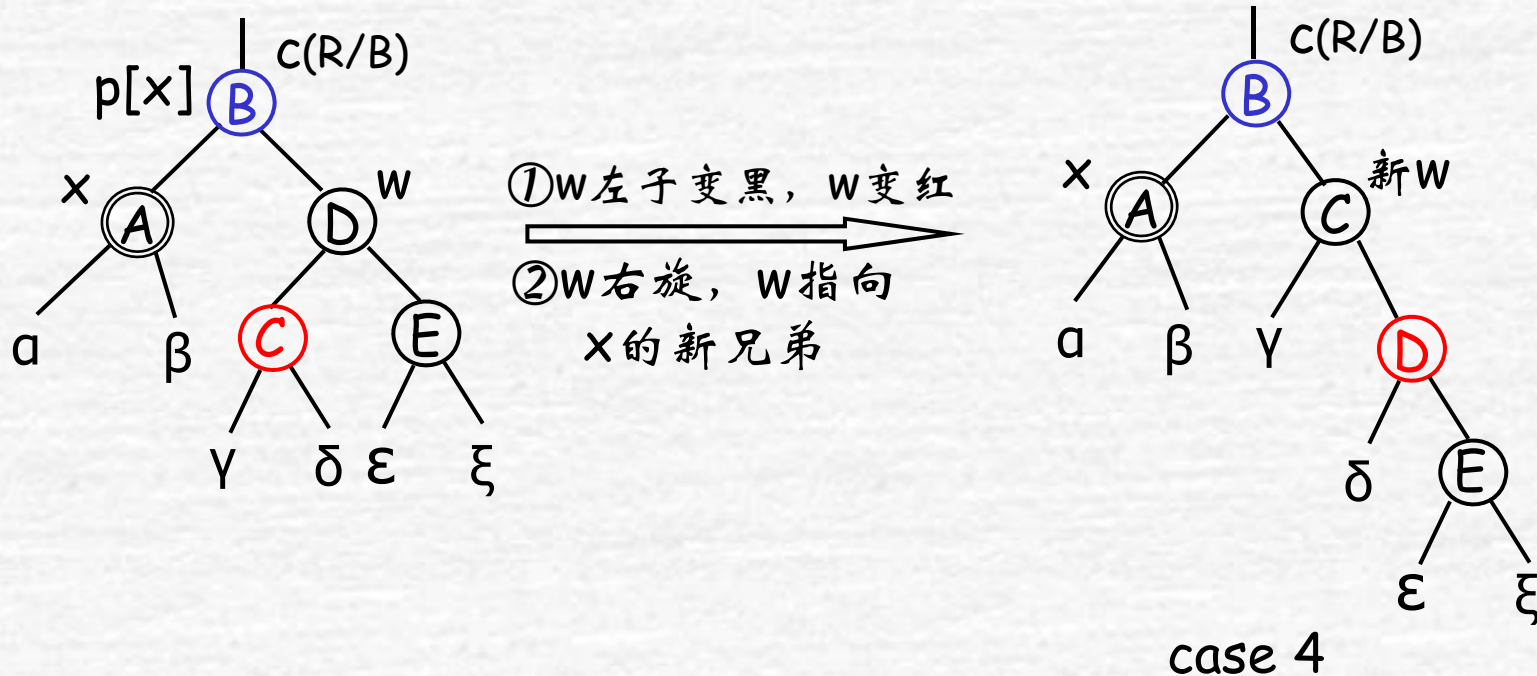


目标:  $x$  上移到  $B$ , 通过  $A$  和  $D$  的黑色上移



# 删除算法 (6)

case 3:  $x$  的黑兄弟  $w$  的右子为黑且左子为红

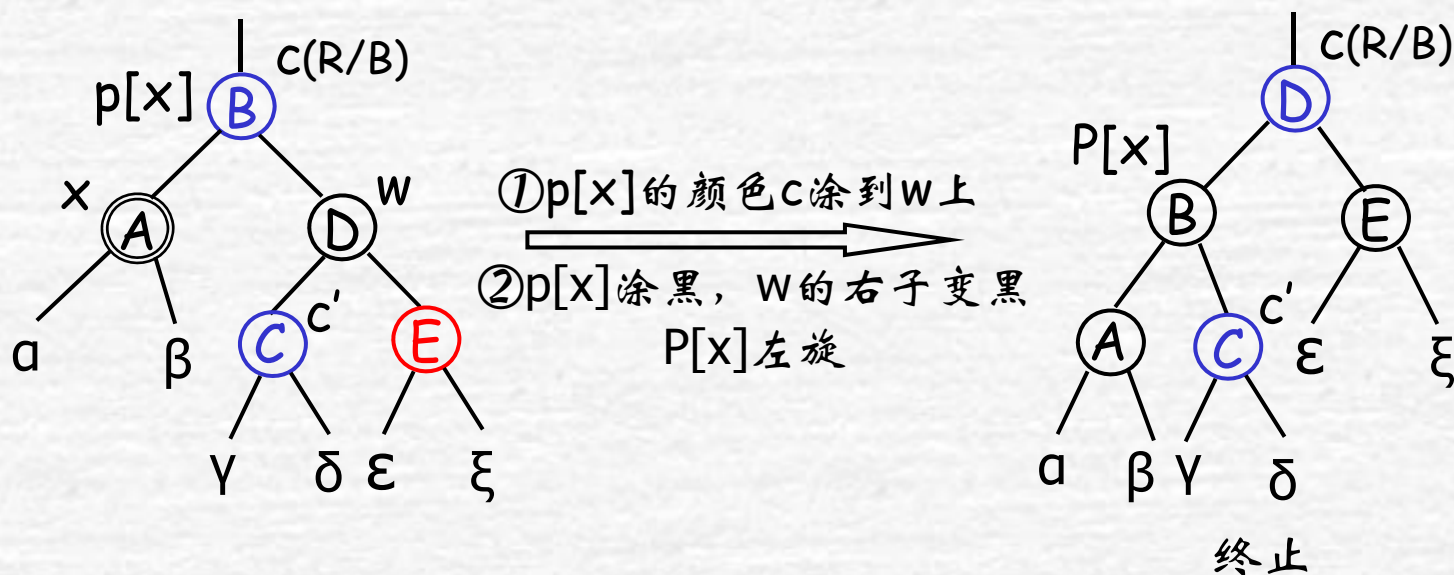


目标: 将case3转为case4



# 删除算法 (7)

case 4:  $x$  的黑兄弟  $w$  的右子为红(左子为黑或红)



目标: 终结处理。 $x$  的黑色上移给  $B$ ,  $B$  的原色下移给  $D$ ,  
 $D$  将黑色下移给  $C$  和  $E$ , 通过旋转解决矛盾点  $C$

- DeleteFixup( $T, x$ ) 算法: P185



# End of Chap13

