数据结构与算法设计

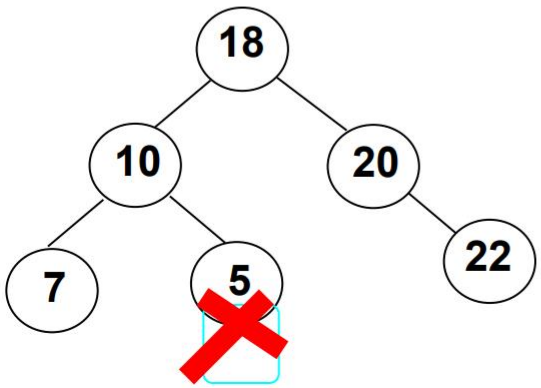
树和二叉树

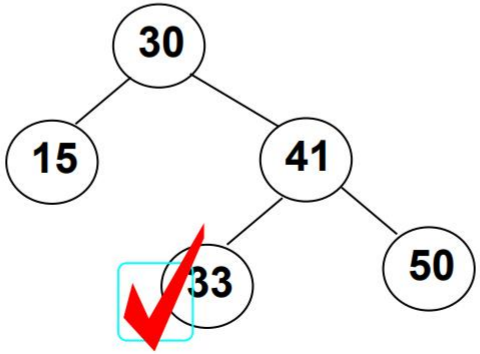
- 线性结构：线性表，栈，队列
- 非线性结构：树和二叉树，图，网

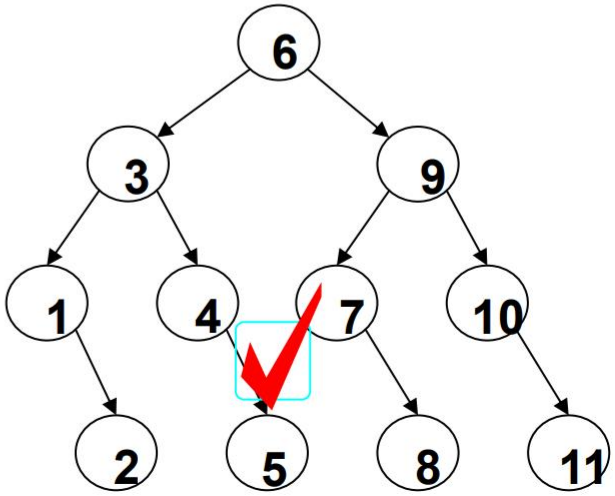
二叉搜索树（BST，Binary Search Tree），
也称二叉排序树或二叉查找树

二叉搜索树：一棵二叉树，可以为空；如果不为空，满足以下性质：

1. 非空左子树的所有键值小于其根结点的键值。
2. 非空右子树的所有键值大于其根结点的键值。
3. 左、右子树都是二叉搜索树。







二叉搜索树的基本操作：

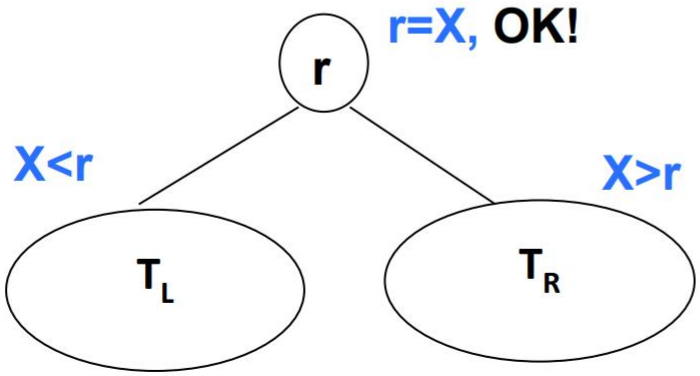
- Position Find(ElementType X, BinTree BST)：从二叉搜索树BST中查找元素X，返回其所在结点的地址；
- Position FindMin(BinTree BST)：从二叉搜索树BST中查找并返回最小元素所在结点的地址；
- Position FindMax(BinTree BST)：从二叉搜索树BST中查找并返回最大元素所在结点的地址。

- BinTree Insert(ElementType X, BinTree BST)
- BinTree Delete(ElementType X, BinTree BST)

二叉搜索树的查找操作：Find

- 查找从根结点开始，如果树为空，返回NULL
- 若搜索树非空，则根结点关键字和X进行比较，并进行不同处理：

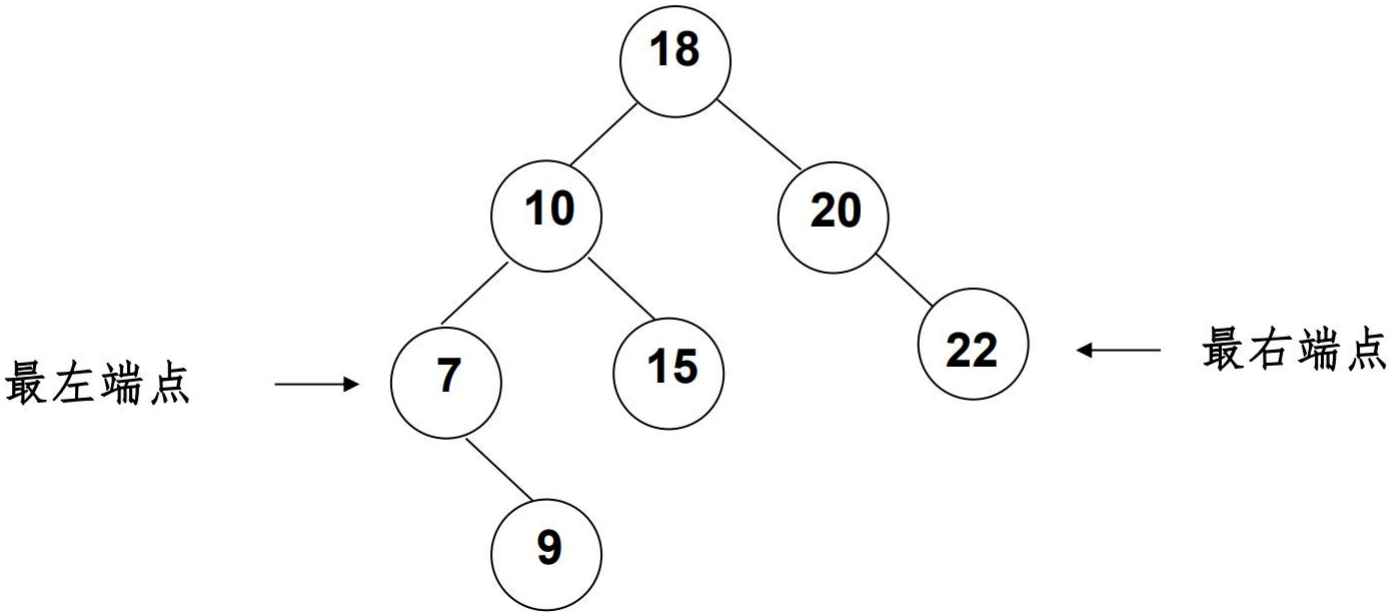
① 若X小于根结点键值，只需在左子树中继续搜索；
② 如果X大于根结点的键值，在右子树中进行继续搜索；
③ 若两者比较结果是相等，搜索完成，返回指向此结点的指针。



```
Position Find( ElementType X, BinTree BST )
{
    if( !BST ) return NULL; /*
    if( X > BST->Data )
        return Find( X, BST->Right ); /*在右子树中继续查找*/
    Else if( X < BST->Data )
        return Find( X, BST->Left ); /*在左子树中继续查找*/
    else /* X == BST->Data */
        return BST; /*查找成功，返回结点的找到结点的地址*/
}
```

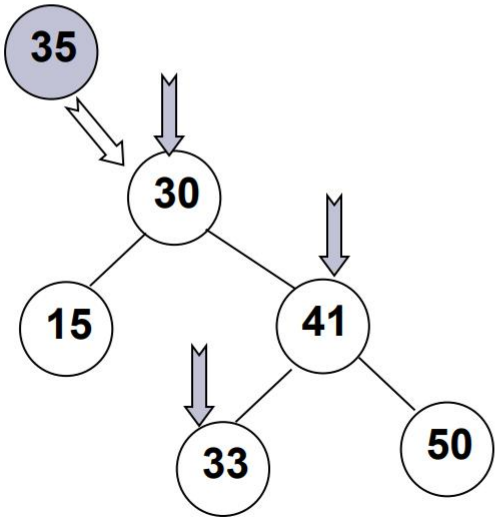
二叉搜索树的查找最大和最小元素操作：FindMax, FindMin

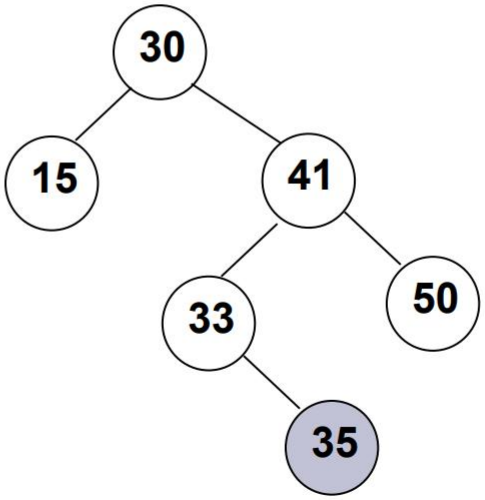
- 最大元素一定是在树的最右分枝的端结点上
- 最小元素一定是在树的最左分枝的端结点上



二叉搜索树的插入操作：Insert

〖分析〗关键是要找到元素应该插入的位置，
可以采用与Find类似的方法

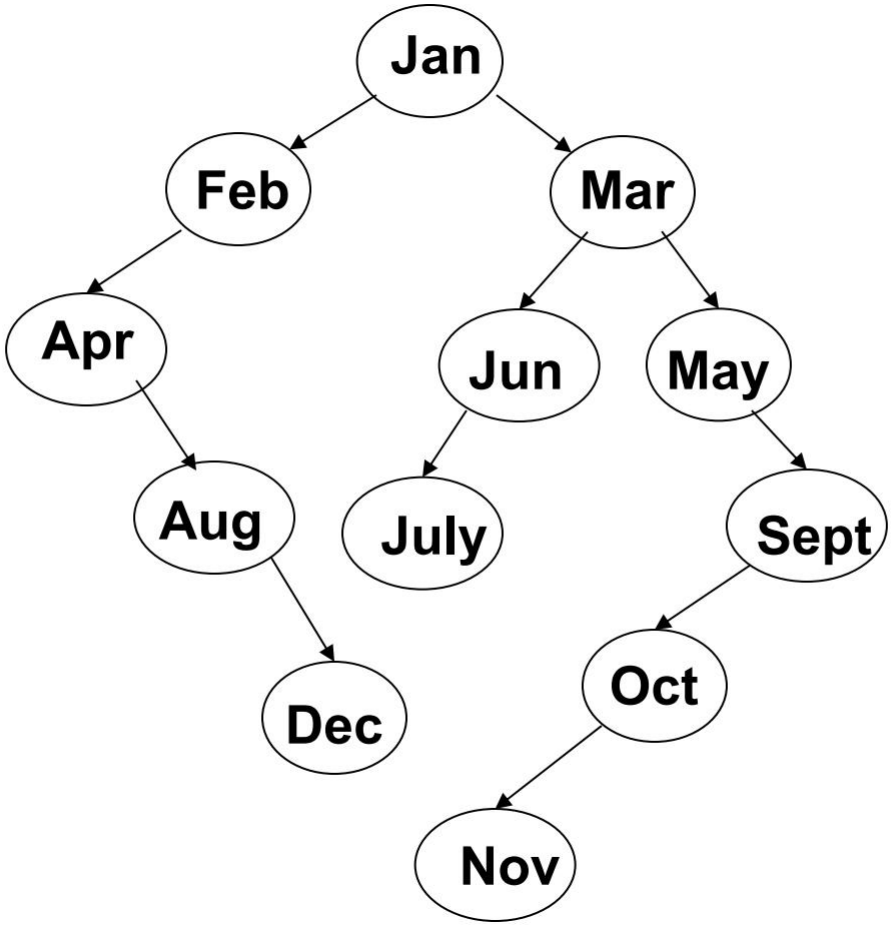




二叉搜索树的插入操作：是否一定插入成叶子节点?

```
BinTree Insert( ElementType X, BinTree BST )
{
    if( !BST ){
            /*若原树为空，生成并返回一个结点的二叉搜索树*/
        BST = malloc(sizeof(struct TreeNode));
        BST->Data = X;
        BST->Left = BST->Right = NULL;
    }else  /*开始找要插入元素的位置*/
        if( X < BST->Data )
            BST->Left = Insert( X, BST->Left);
                        /*递归插入左子树*/
        else  if( X > BST->Data )
            BST->Right = Insert( X, BST->Right);
                        /*递归插入右子树*/
        /* else X已经存在，什么都不做 */
    return BST;
}
```

【例】以一年十二个月的英文缩写为键值，按从一月到十二月顺序输入，即输入序列为（Jan, Feb, Mar, Apr, May, Jun, July, Aug, Sep, Oct, Nov, Dec）

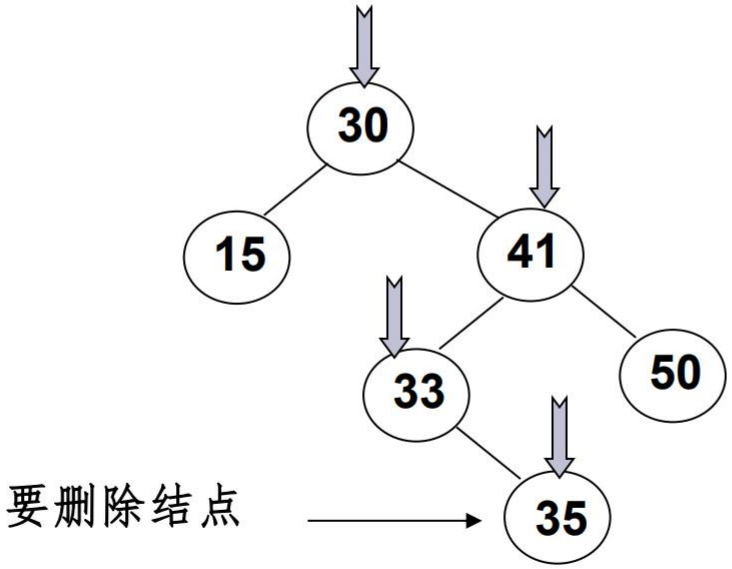


二叉搜索树的删除操作：Delete

考虑三种情况：

☞ 要删除的是叶结点：直接删除，并再修改其父结点指针---置为NULL

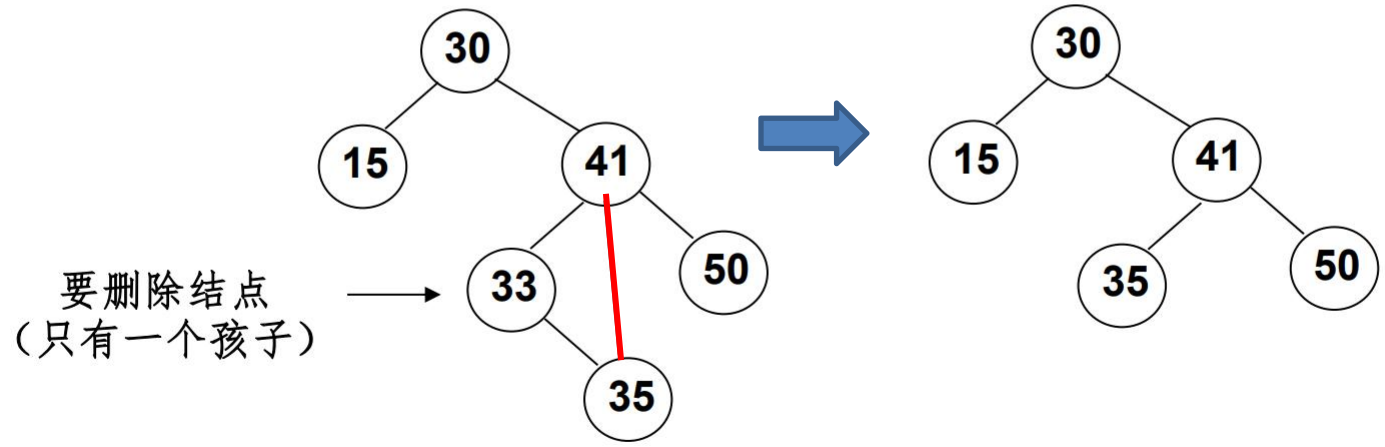
〖例〗：删除 35



☞ 要删除的结点只有一个孩子结点:

将其父结点的指针指向要删除结点的孩子结点

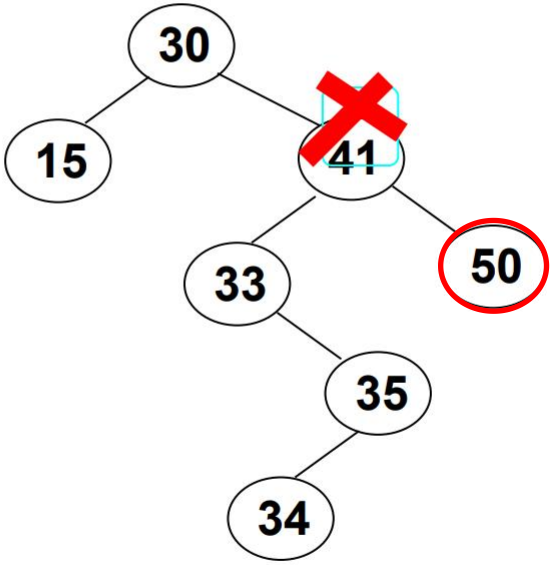
〖例〗：删除 33



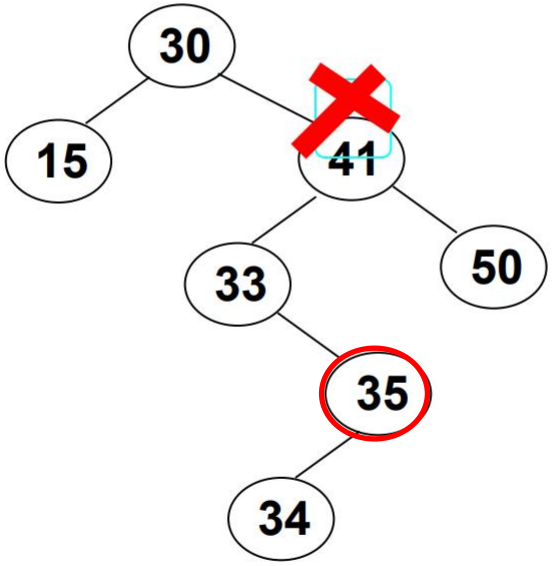
☞ 要删除的结点有左、右两棵子树：

用另一结点替代被删除结点：右子树的最小元素 或者 左子树的最大元素

〖例〗：删除 41



1、取右子树中的最小元素替代



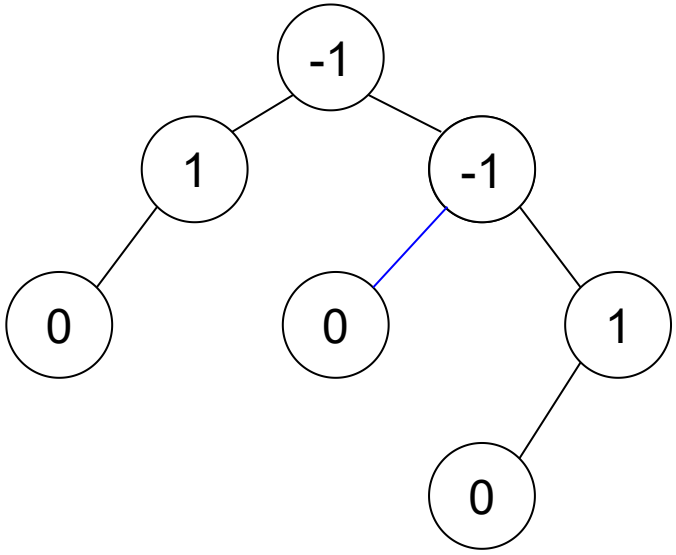
2、取左子树中的最大元素替代

```
BinTree Delete( ElementType X, BinTree BST )
{    Position Tmp;
     if( !BST ) printf("要删除的元素未找到");
     else if( X < BST->Data )
             BST->Left = Delete( X, BST->Left); /* 左子树递归删除 */
     else if( X > BST->Data )
             BST->Right = Delete( X, BST->Right); /* 右子树递归删除 */
     else /*找到要删除的结点 */
          if( BST->Left && BST->Right ) { /*被删除结点有左右两个子结点 */
              Tmp = FindMin( BST->Right );
                              /*在右子树中找最小的元素填充删除结点*/
              BST->Data = Tmp->Data;
              BST->Right = Delete( BST->Data, BST->Right);
                                        /*在删除结点的右子树中删除最小元素*/
          } else { /*被删除结点有一个或无子结点*/
              Tmp = BST;
              if( !BST->Left ) /* 有右孩子或无子结点*/
                   BST = BST->Right;
              else if( !BST->Right ) /*有左孩子或无子结点*/
                   BST = BST->Left;
              free( Tmp );
          }
     return BST;
```

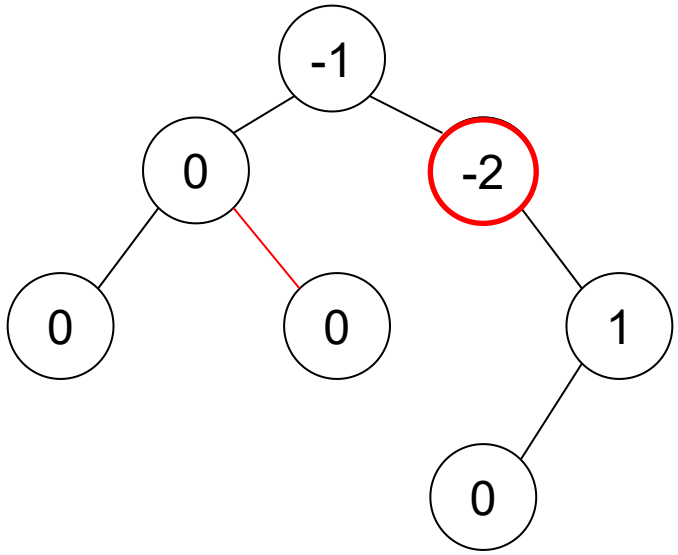
平衡二叉树

1）平衡因子（BF，Balance Factor）： 一个结点的平衡因子等于该结点左子树的深度减去它的右子树的深度。

2）平衡二叉树是一棵二叉树：或者为空，或者所有结点的平衡因子只能是-1、0、1。否则为不平衡状态。



平衡二叉树



不平衡的二叉树

实际应用中，平衡树还有多种类型。常用的有AVL树、红黑树、Treap、伸展树等。

AVL树：是最先发明的**自平衡二叉查找树**算法。

在AVL树中，任何结点的两个儿子的子树的高度最大差为1（平衡因子），所以它也被称为高度平衡树。

- **n个结点的AVL树最大深度约$1.44\log_2 n$**，查找、插入和删除在平均和最坏情况下都是O(logn)。
- AVL树是**严格的平衡二叉树**，平衡条件必须满足。因此每次插入或删除结点都可能引起一次或多次结点的旋转来重新平衡这个树，**维护代价很高**。
- AVL树适合用于插入与删除次数比较少，但查找多的情况，实际应用的不多。更多的是用**追求局部而不是严格整体平衡的红黑树**。

红黑树：也是一种**自平衡二叉查找树**，和AVL树类似，都是在进行插入和删除操作时通过特定操作保持树的平衡，从而获得较高的查找性能。

- 红黑树是在1972年由Rudolf Bayer发明的，当时被称为“平衡二叉B树”(symmetric binary B-trees)。后来，在1978年被Leo J. Guibas和Robert Sedgewick修改为如今的“红黑树”。
- 最坏情况运行时间非常良好，实践验证它是高效的： 它可以在O(logn)时间内做查找、插入和删除。
- 红黑树有着广泛的应用，JDK源码中的treeMap和JDK8的HashMap都用到了红黑树来存储；STL和linux都使用红黑树作为平衡树的实现。

Treap：是一棵二叉排序树，左子树和右子树分别是一个Treap。

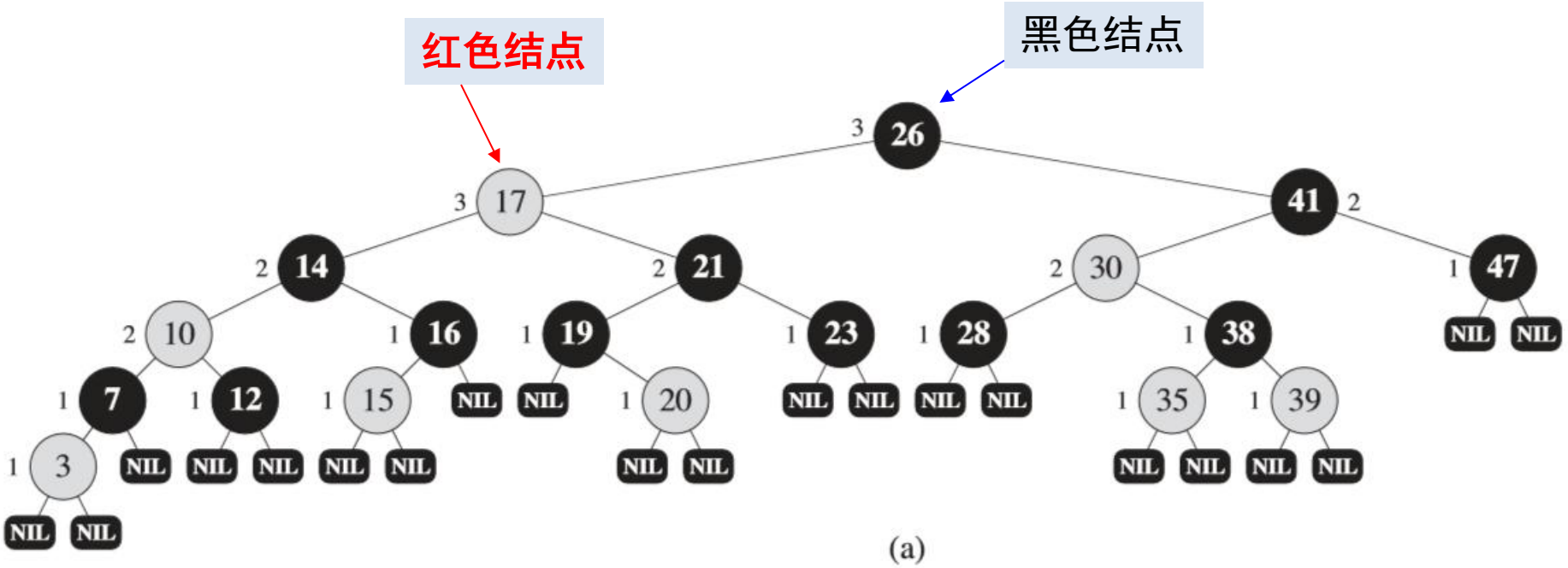
Treap和一般的二叉排序树不同的是，**Treap记录一个额外的数据，就是优先级**。Treap在以关键码构成二叉排序树的同时，还满足堆的性质(在这里我们假设节点的优先级大于该节点的孩子的优先级)。但是Treap和二叉堆有一点不同，就是二叉堆必须是完全二叉树，而Treap并不一定是。

伸展树（Splay Tree）：是一种二叉排序树，它能在O(log n)内完成插入、查找和删除操作。

它由Daniel Sleator和Robert Tarjan创造。它的优势在于不需要记录用于平衡树的冗余信息。在伸展树上的一般操作都基于伸展操作。

红黑树（Red-Black Tree）是一棵**二叉搜索树**，它在每个结点上增加了一个**存储位**来表示结点的颜色：**Red**或**Black**

—— 因此称为红黑树。如图所示：



为红黑树的每个结点定义5个基本属性：

color：**颜色**，红或者黑

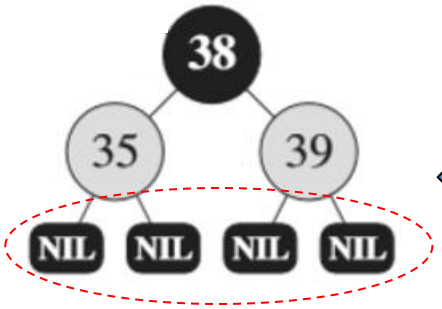
key：　可以比较大小的关键字

left：　指向左孩子的指针

right：指向右孩子的指针

p：　　**指向父结点的指针**

如果一个结点没有子结点或父结点，则相应指针的值为NIL



这里把“NIL”看做二叉搜索树的“**外部叶结点**”，代表“空结点”，所有为NIL的指针都指向这样一个结点。

红黑树的定义：

一棵红黑树是满足下面**红黑性质**的二叉搜索树：

1. 每个结点或者是红色的，或者是黑色的；

2. 根结点是黑色的；

3. 每个叶结点（NIL结点）是黑色的；

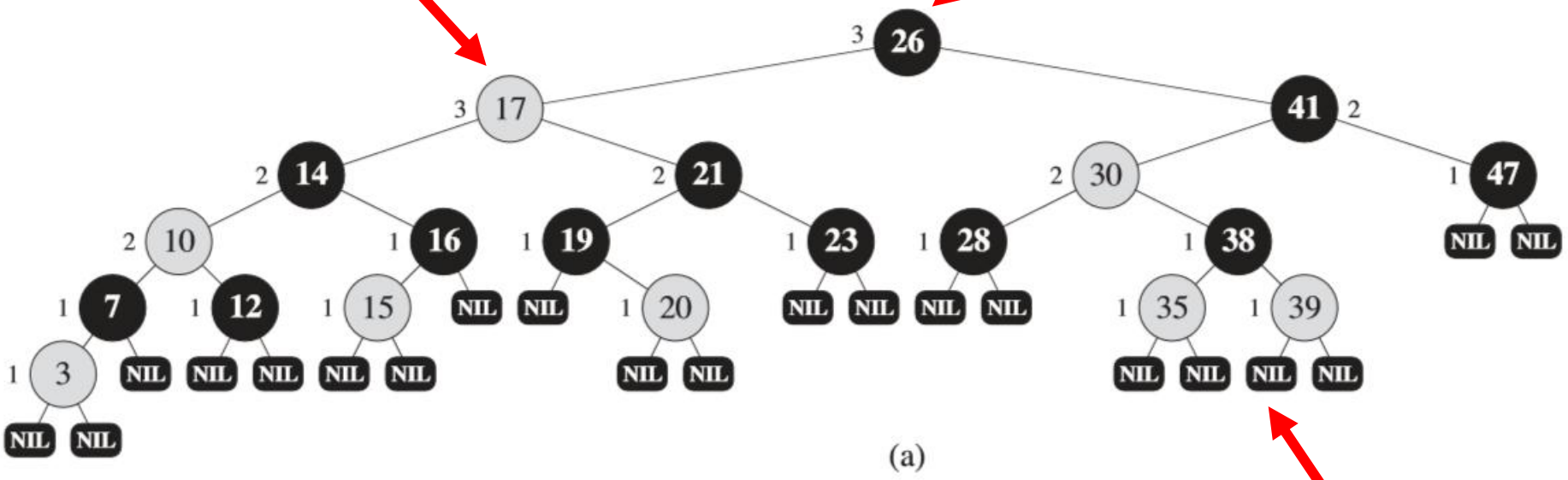
4. 如果一个结点是红色的，则它的两个子结点都是黑色的（也就是说不存在两个相邻的红色结点，但黑色结点可能相邻）；

5. 对每个结点，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。

一个红黑树的例子

如果一个结点是红色的，则
它的两个子结点都是黑色的

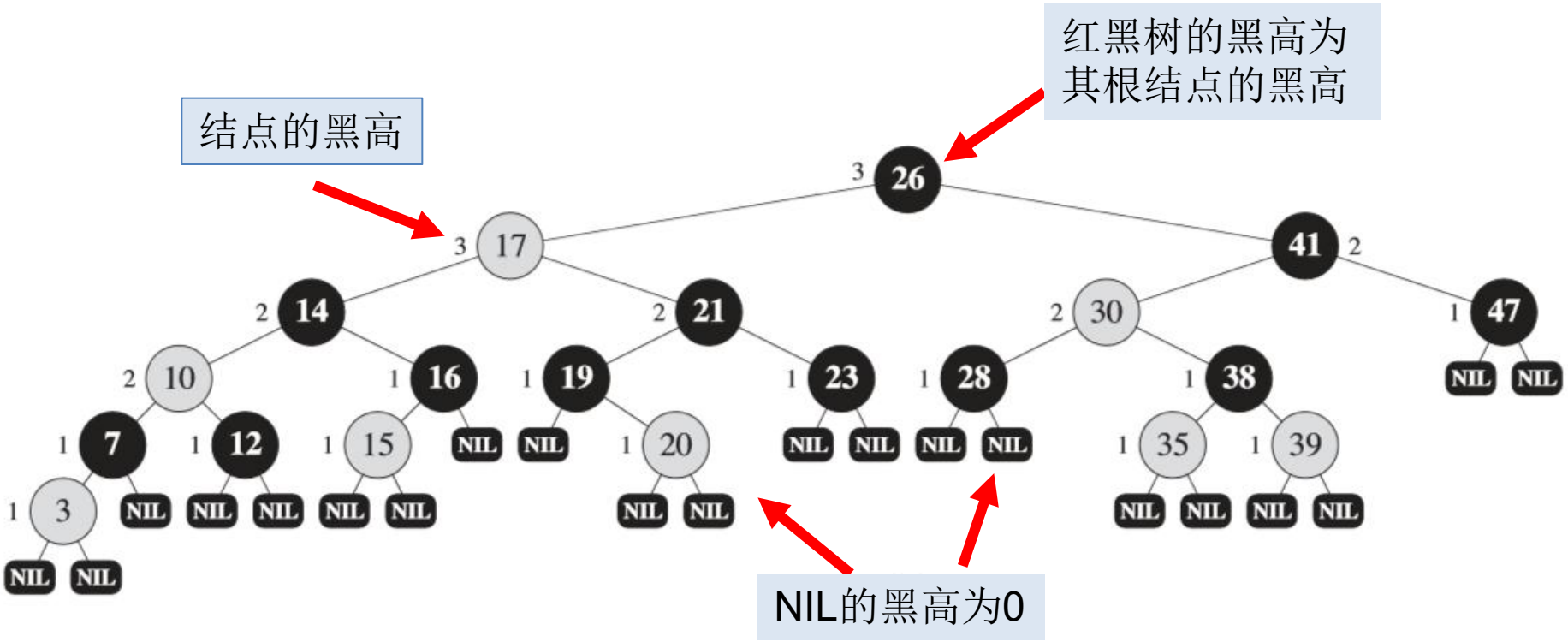
根结点是黑色的



NIL叶结点是黑色的

黑高：从结点x出发（不含该结点）到达一个NIL叶结点的任意一条简单路径上的**黑结点个数**（**包括NIL结点**）称为该结点的**黑高**（black-height），记为bh(x)。

- NIL叶结点的黑高为0。
- 红黑树的黑高等于根结点的黑高。



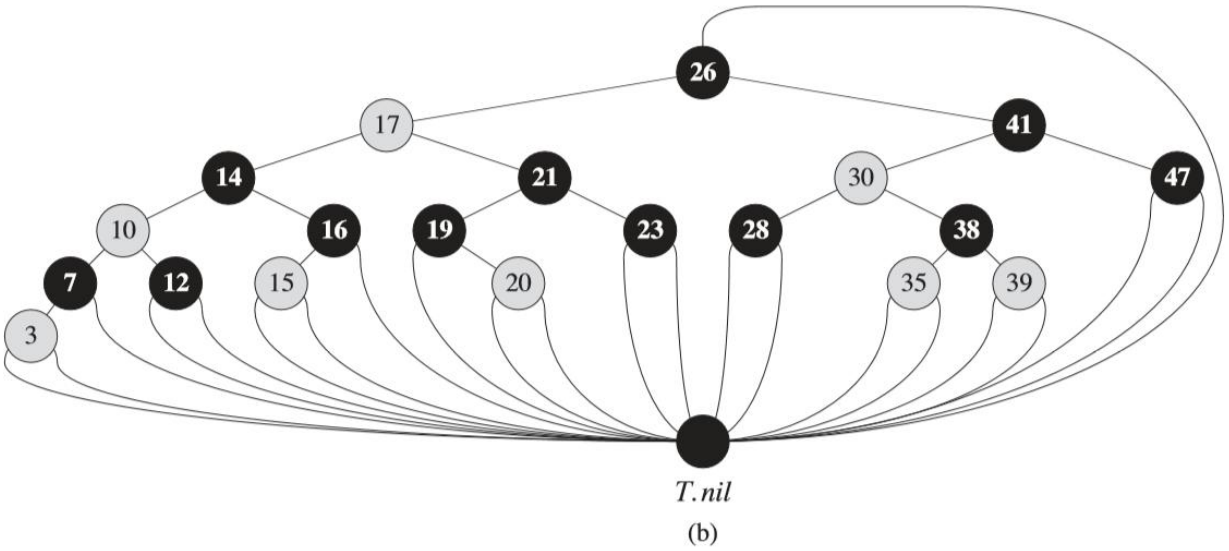
引入**哨兵结点**：

为了便于处理边界条件，这里使用一个**哨兵**来代表**NIL**：

对于一棵红黑树T，**哨兵T.nil**是一个与树中普通结点有相同属性的对象：它的color属性为BLACK，其它属性（ key、left、right、p ）可以设为任意值。

引入**哨兵结点T.nil**后，原来所有指向NIL的指针都用**指向T.nil的指针**替换。

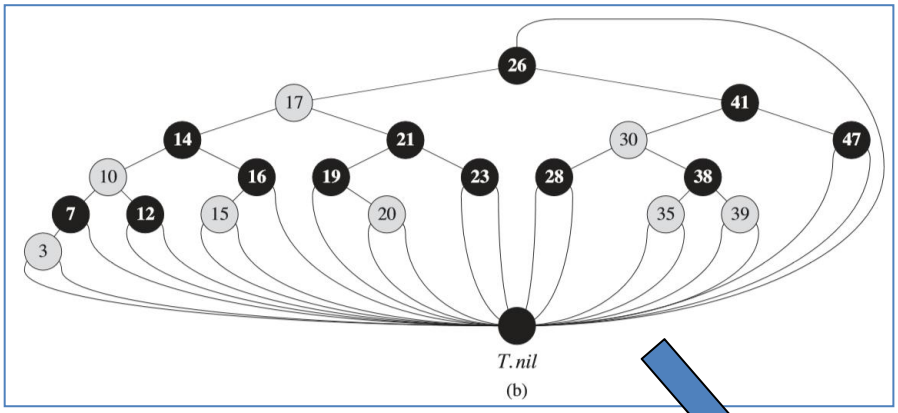
如图所示：

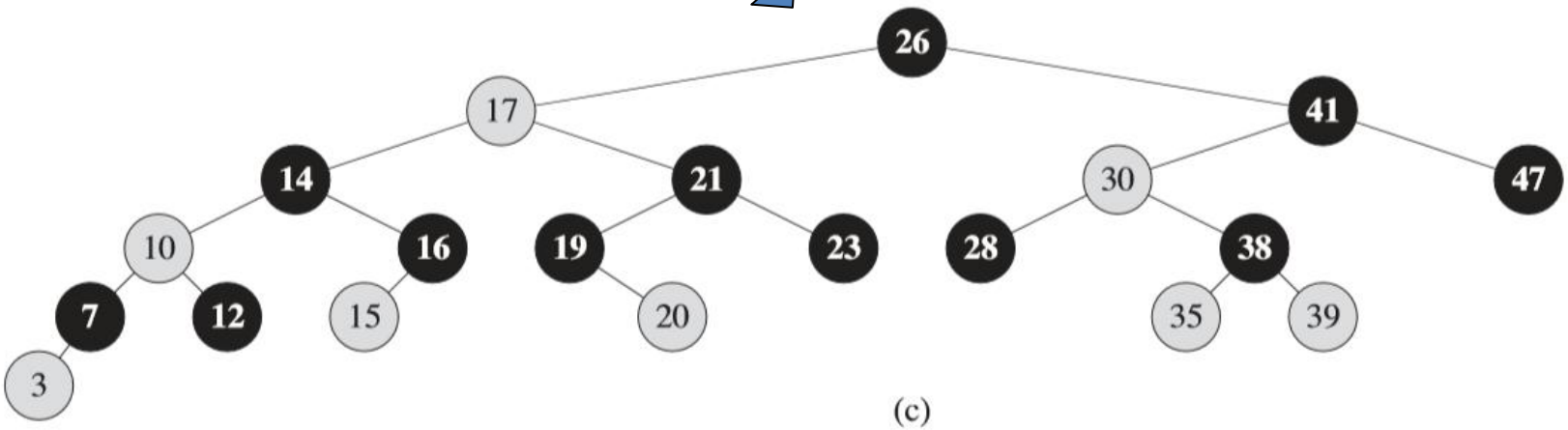


- 使用哨兵后，就可以将结点x的NIL孩子视为一个普通结点，其父结点是x。
  - 尽管可以为树内的每一个NIL新增一个不同的哨兵结点，使得每个NIL的父节点都有这样的良定义，但这种做法会浪费空间。
  - 取而代之的是，这里使用一个哨兵T.nil来代表所有的NIL，替代所有的NIL叶结点和根结点的父结点。
  - 哨兵的属性p、left、right和key的取值并不重要，其颜色被指定为BLACK。

- 引入T.nil后，算法的注意力主要在红黑树的内部结点上，故**在画图表示时，可以忽略叶结点。**

如：





省略了叶结点和根结点的父结点后的红黑树图形表示

红黑树性质：

红黑树通过对从根到叶子的简单路径上各个结点的颜色进行约束，确保没有一条路径会比其他路径长出2倍，达到近似于平衡的状态（而不是绝对平衡的）。

可以证明：一棵有n个内部结点的红黑树的高度至多为2log(n+1)。

这样可以保证在最坏情况下基本动态集合操作的时间复杂度为O(log*n*)。

引理： 一棵有n个内部结点的红黑树的高度至多为2log(n+1)。

证明：

1）首先采用归纳法证明：**以任一结点x为根的子树至少包含$2^{bh(x)}-1$个内结点 。**

(1)如果x的高度＝0，则x为叶结点（即T.nil），以x为根的子树不包含任何内结点，且黑高为0，所以结论成立（$2^{bh(x)}-1= 2^0-1=0$）。

(2)归纳：若x的高度＞0，则x有两个子结点，且**子结点的黑高或者为bh(x)或者为bh(x)-1**（取决于子结点为红色结点或者黑色结点）。

根据归纳假设，即有以x为根的子树至少包含

$$(2^{bh(x)-1}-1)+ (2^{bh(x)-1}-1)+1= 2^{bh(x)}-1$$

个内结点。得证。

2）引理证明

设树的高度为h，根据红黑树的性质4：如果一个结点是红色的，则它的两个子结点都是黑色的，从根到叶结点的任何一条简单路径上都至少有一半的结点为黑色(不包括根结点)。因此，**根的黑高至少为h/2**。

于是有：$n \geqslant 2^{h/2}-1$

所　以：$h \leqslant 2\lg(n+1)$

引理得证　■

由引理可知，基于红黑树的动态集合操作（SEARCH, MINIMUM, MAXIMUM，SUCCESSOR，PREDECESSOR）可以在O(lgn)时间内执行。

旋转（rotation）

和AVL树类似，红黑树上的结点插入和删除会造成树的结构违反红黑树性质的情况。这时需要改变树中某些结点的颜色以及指针结构，以维护红黑树的性质。

其中，指针结构的修改是通过旋转操作来完成的。

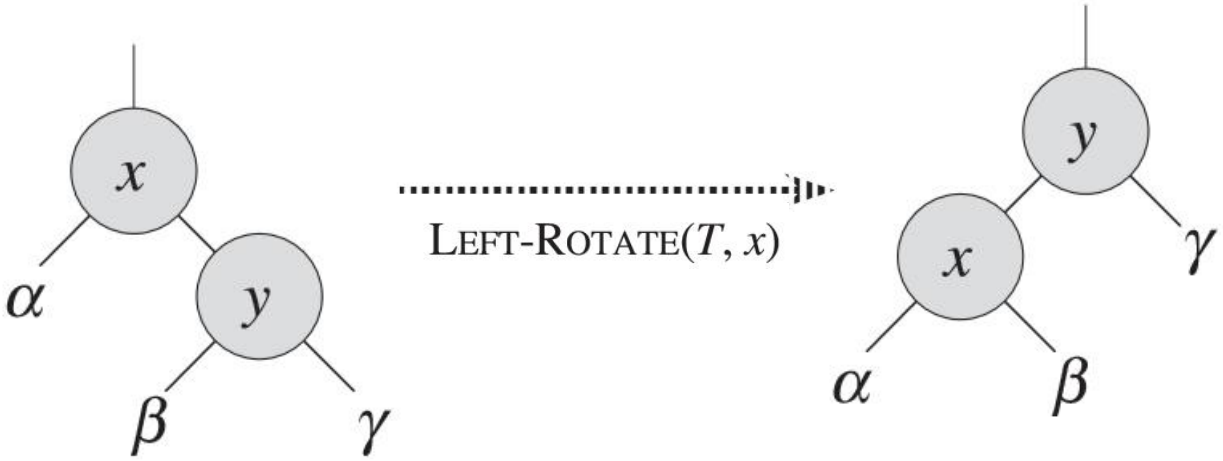
- 基本的旋转操作有**左旋**、**右旋**两种。

1. 左旋

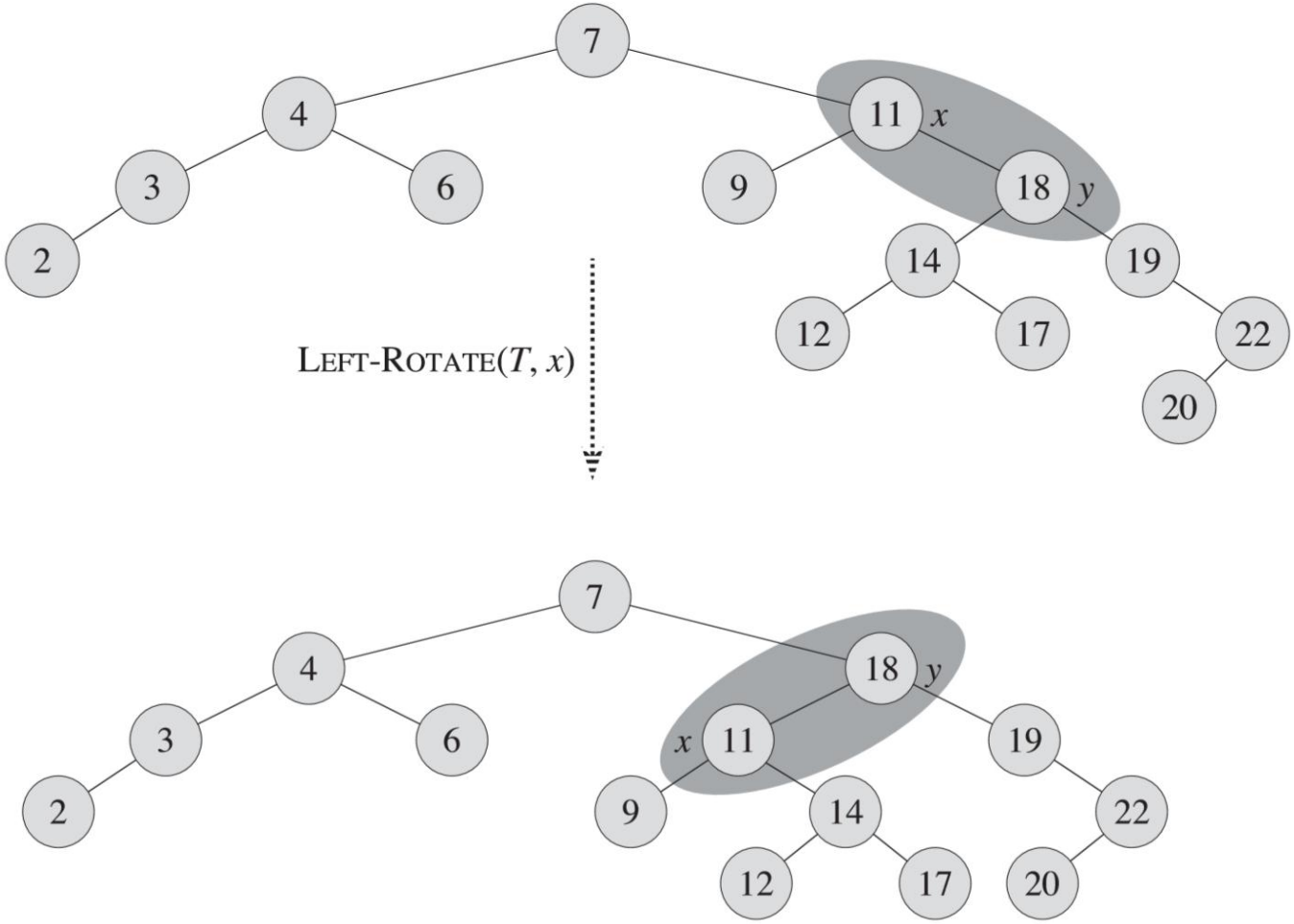
设x为树内的任意结点，y是x的右结点，**y ≠T.nil**。

左旋以x到y的链为支轴进行，它使y成为该子树新的根结点，x成为y的左孩子，y的左孩子成为x的右孩子。

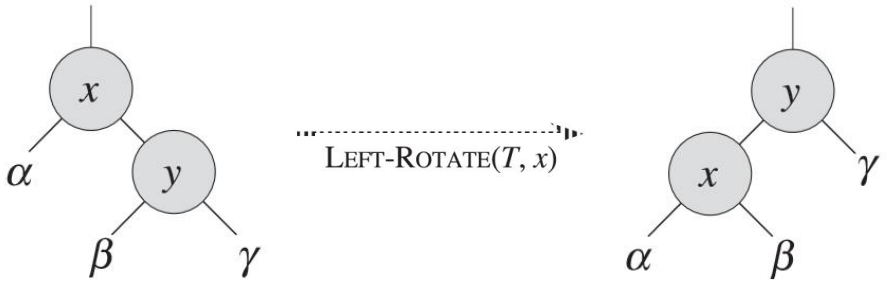
如图所示：



左旋的例子：


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

左旋的伪代码：




```
LEFT-ROTATE(T, x)
1   y = x.right
2   x.right = y.left
3   if y.left ≠ T.nil
4       y.left.p = x
5   y.p = x.p
6   if x.p == T.nil
7       T.root = y
8   elseif x == x.p.left
9       x.p.left = y
10  else x.p.right = y
11  y.left = x
12  x.p = y
```

//令y指向x的右孩子

//将y的左孩子置为x的右孩子

//如果y有左孩子，将y的左孩子的父结点置为x

//修改y的双亲结点为x的双亲结点

//若x是根结点，则重新置树的根结点为y

//否则（若不是根结点），若x是其双亲的左孩子，则将x双亲的左孩子置为y

//否则，则将x双亲的右孩子置为y

//旋转后，x为y的左孩子

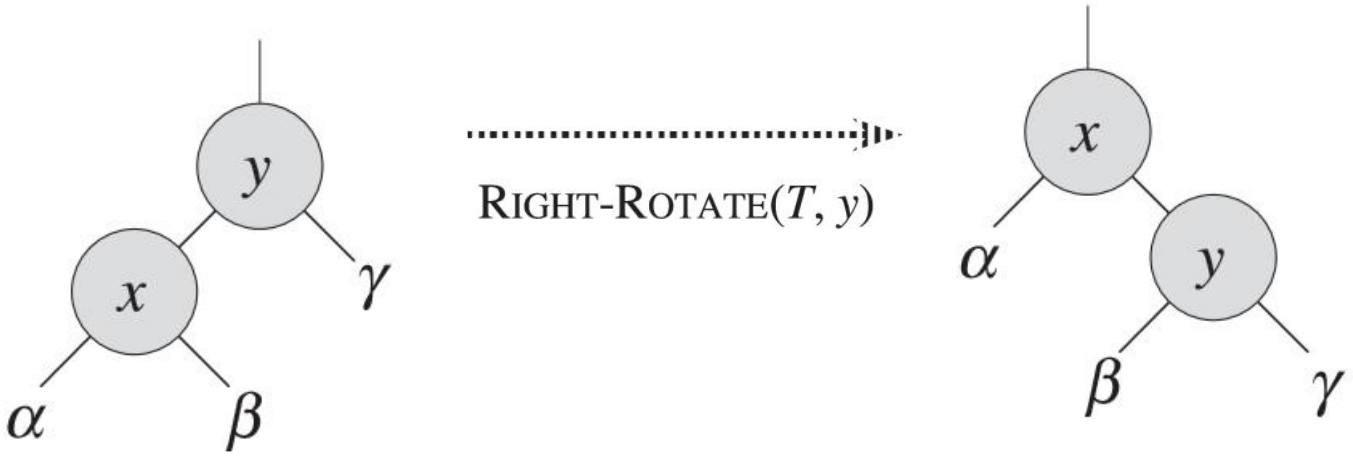
//置x的双亲为y

2. 右旋

设y为树内的任意结点，x是y的左结点，**x ≠T.nil**。

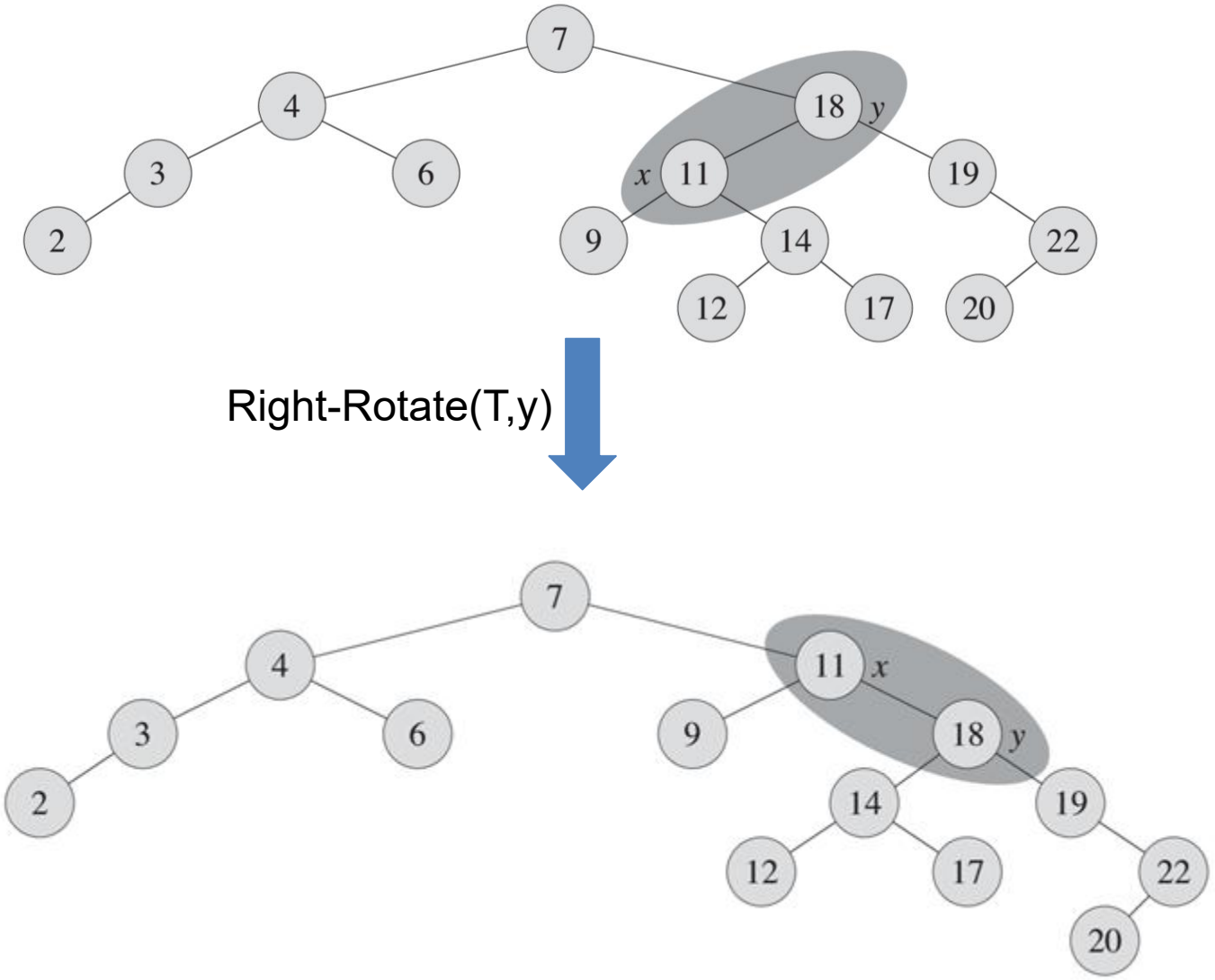
右旋以y到x的链为支轴进行，它使x成为该子树新的根结点，y成为x的右孩子，x的右孩子成为y的左孩子。

如图所示：



- 右旋的伪代码（略）。
- 旋转操作只改变部分指针的指向，结点的其它属性都保持不变。
- **左旋和右旋操作都可以在O(1)的时间内完成。**

右旋的例子：


7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20
Right-Rotate(T,y)
7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20

插入

设z是待插入的结点，分三步完成：

（1）按照和普通二叉搜索树一样的操作，将z插入到红黑树中的恰当位置；

（2）将z着为红色（思考：为什么不将z着为黑色？）；

（3）调用辅助过程RB-INSERT-FIXUP对必要的结点进行调整，包括**重新着色**和**旋转**，使得树的红黑性质得以保持。

上述操作可以在$O(\lg n)$时间内完成。

RB-INSERT的伪代码

```
RB-INSERT(T, z)
 1   y = T.nil
 2   x = T.root
 3   while x ≠ T.nil
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == T.nil
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
14   z.left = T.nil
15   z.right = T.nil
16   z.color = RED
17   RB-INSERT-FIXUP(T, z)
```

对比普通二叉搜索树的插入过程

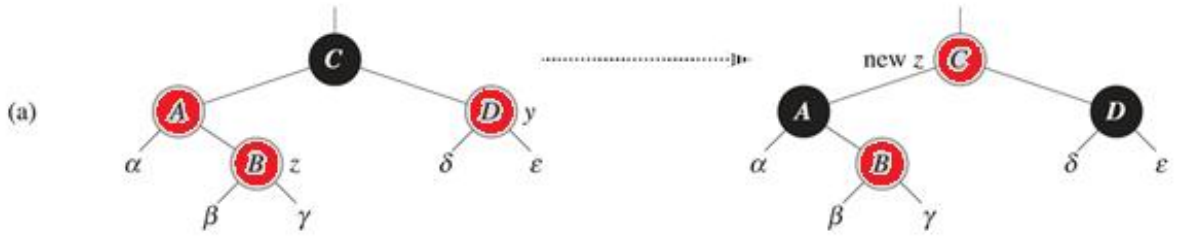
```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```

将z.left和z.right置为T.nil

将z着为红色，这时可能会使树违反其中的一条红黑性质，下面调用RB-INSERT-FIXUP过程来对树中的结点进行修正，以保持树的红黑性质。

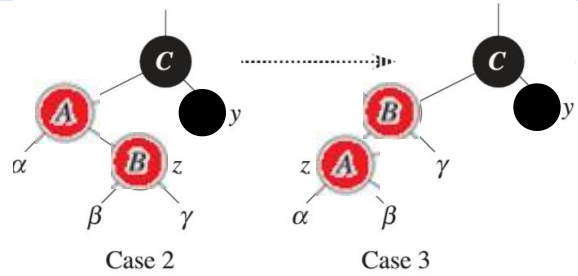
若z结点插在一个红色结点之下，这样出现“双红”的问题，需要进行修正。**注：此时z的“爷爷”一定是黑色。**

情况1： **z的双亲是其爷爷的左孩子，右“叔结点”y为红色**



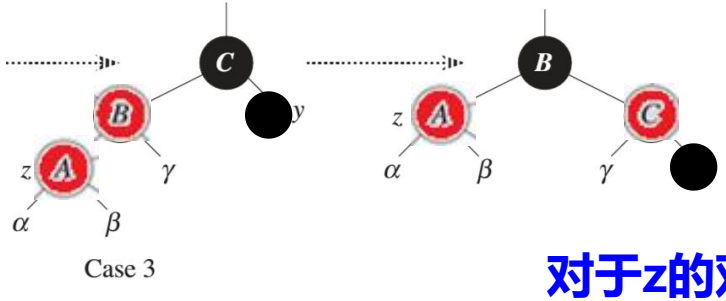
C由黑变红，A、D由红变黑，然后z指向C，**将冲突上传一层继续处理**。

情况2：**z的双亲是其爷爷的左孩子，右“叔结点”y为黑色。z是其双亲的右孩子**



基于A-B做一次左旋，然后z指向A，将情况2转换成情况3。

情况3： **z的双亲是其爷爷的左孩子，“右”叔结点y是黑色。z是其双亲的左孩子**



重置B为黑色，C为红色，基于B-C做一次右旋。

对于z的双亲是其爷爷的右孩子的情况做对称处理即可

RB-INSERT-FIXUP的伪代码

思考：为什么z的双亲是黑色时不需要做调整？

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                ▷ case 1
 6              y.color = BLACK                  ▷ case 1
 7              z.p.p.color = RED                ▷ case 1
 8              z = z.p.p                        ▷ case 1
 9          else if z == z.p.right
10              z = z.p                          ▷ case 2
11              LEFT-ROTATE(T, z)                ▷ case 2
12          z.p.color = BLACK                    ▷ case 3
13          z.p.p.color = RED                    ▷ case 3
14          RIGHT-ROTATE(T, z.p.p)               ▷ case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

只有z的双亲也是红色时才做下面的调整。

y是z的右“叔”结点

情况1：z的右“叔”结点y也是红色的。

此时仅将z的双亲结点和y改为黑色，将z的“爷爷”结点着为红色，将z指向它的爷爷结点。

情况2：y是黑色的，z是其双亲的右孩子。

将z指向其双亲结点。左旋，转变成情况3处理。

情况3：y是黑色的，z是其双亲的左孩子 。

将z的双亲置为黑色，z的爷爷结点置为红色，以z的爷爷结点为基点右旋一次。

分情况讨论如下：（z是插入的新结点，y是z的“叔”结点）

情况1：z的双亲是其爷爷的左孩子，右“叔结点”也为红色

RB-INSERT-FIXUP(T, z)

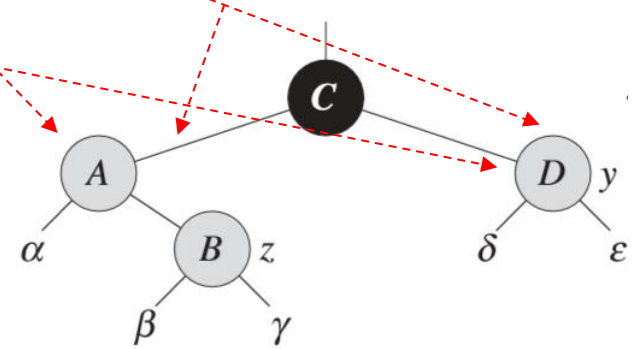
```
1   while z.p.color == RED
2       if z.p == z.p.p.left
3           y = z.p.p.right
4           if y.color == RED
5   // case 1      z.p.color = BLACK
6   // case 1      y.color = BLACK
7   // case 1      z.p.p.color = RED
8   // case 1      z = z.p.p
```

z的双亲是红结点

z的双亲是其“爷爷”结点的左孩子

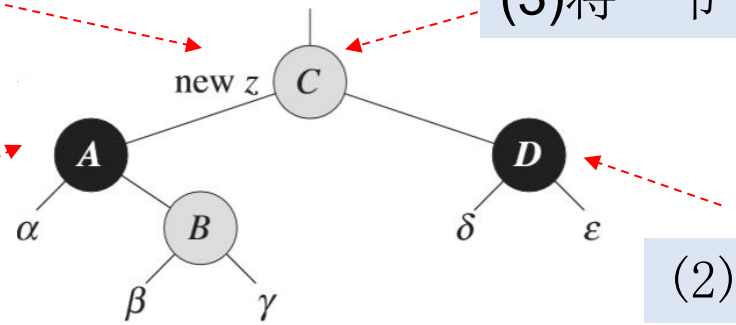
y是z的右“叔”结点

y也是红色的。



(3)将“爷爷”结点颜色着为红色

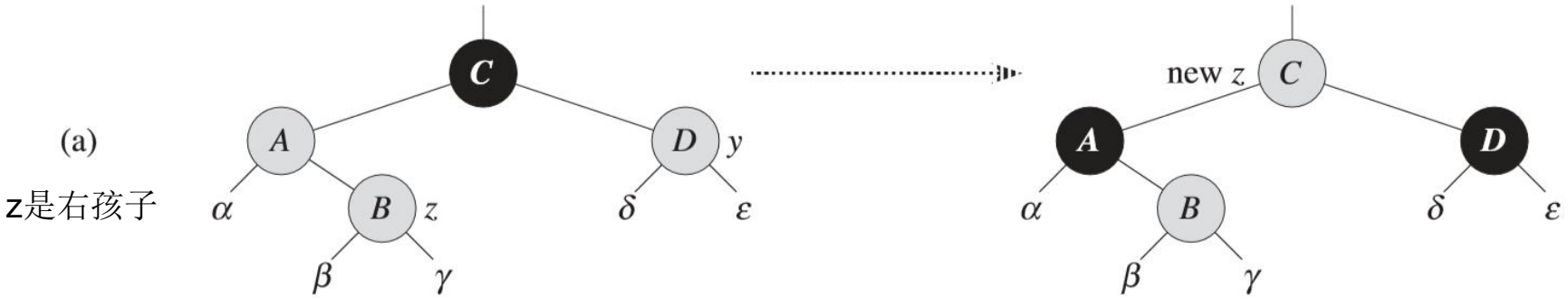
(4)将z重新指向它的爷爷结点



(1)将z的双亲结点的颜色改变为黑色

(2)将y的颜色改为黑色

修改前：由于z的双亲都是红色的，所以z的双亲不是树根结点，所以**C结点一定存在，且C一定是黑色的**。因为z是红色，所以z的插入对A、C，D的黑高没有带来影响。



(1)将z的双亲结点改变为黑色，(2)将y改为黑色，(3)将z的“爷爷”结点着为红色，(4)将z重新指向它的爷爷结点。

修改后：C由黑变红，A、D由红变黑, B没变, 所以A、C、D的黑高都没有变化。

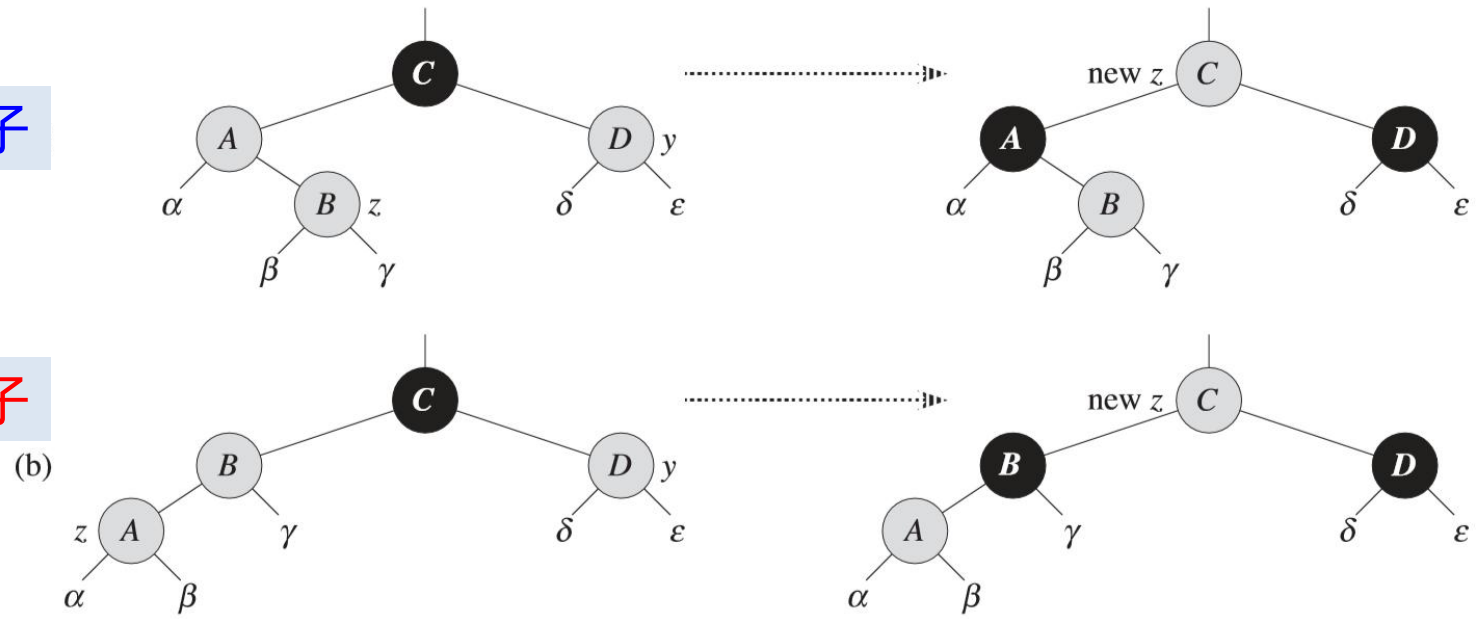
但由于C由黑变红，可能会引起C和C的双亲同时为红而破坏了红黑性质4。这时，通过将z重新指向C（z的“爷爷”结点），将冲突上传一层，在进入下一次循环时继续处理，这一过程可能一直向上“蔓延”到根。

但**最多经过O(lgn)次处理可以结束。**

上面是z是其双亲的右孩子的情况。如果z是其双亲的左孩子，处理方式相同。

z是右孩子

z是左孩子



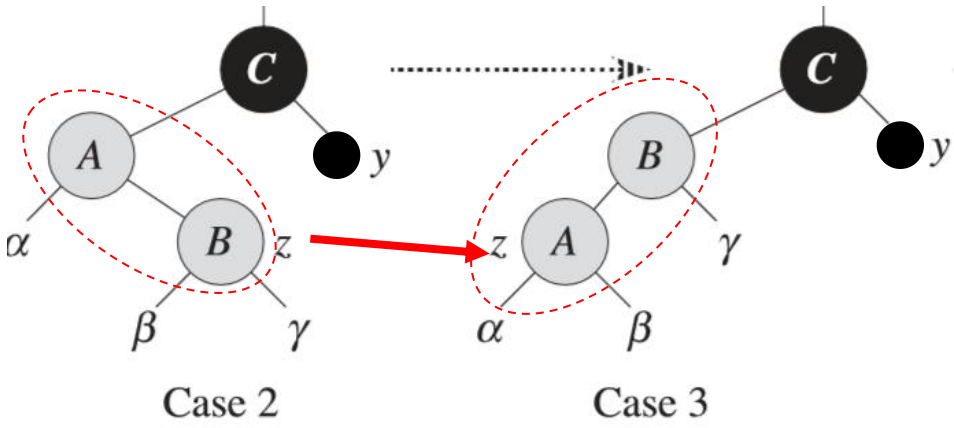
情况2：z的双亲是其爷爷的左孩子，但其右“叔结点”y为黑色。

同时，z是其双亲的右孩子。

修改前：如图，C是黑色（一定存在），y是黑色。z是其双亲的右孩子。

处理方式：令z=z.p，然后做一次左旋，将情况2转换成情况3。

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK
6              y.color = BLACK
7              z.p.p.color = RED
8              z = z.p.p
9          else if z == z.p.right
10             z = z.p                    // case 2
11             LEFT-ROTATE(T, z)          // case 2
```

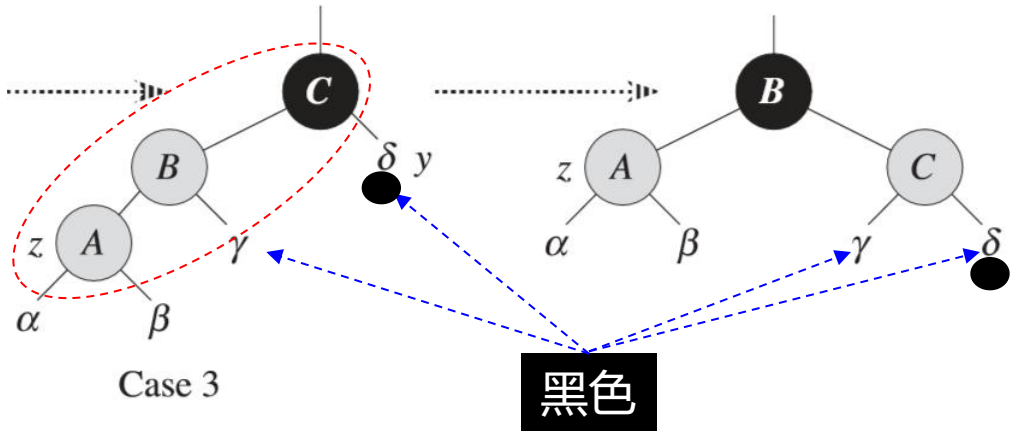


修改后：C不变，y不变。B成为C的左孩子，B的左孩子成为A的右孩子，A成为B的左孩子。从而将树的形态转变为情况3的情形。这一过程只做一次旋转，所以仅需O(1)的时间。

情况3：z的双亲是红色的，爷爷结点C结点存在且是黑色。“右”叔结点y是黑色。z是其双亲的左孩子。

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK
6              y.color = BLACK
7              z.p.p.color = RED
8              z = z.p.p
9          else if z == z.p.right
10                 z = z.p
11                 LEFT-ROTATE(T, z)
12             z.p.color = BLACK          // case 3
13             z.p.p.color = RED          // case 3
14             RIGHT-ROTATE(T, z.p.p)     // case 3
```

修改处理：令z的双亲B为黑色，”爷爷”结点C为红色，做一次右旋。



修改后：A、γ、δ的黑高没变。B成为子树的根，C成为B的右孩子。

由于C从黑变红，B从红变黑，所以旋转后子树A和C的黑高相等且等于原来C的黑高，从而使得树的红黑性质得以保持。

且右旋后，z（即A）的双亲B的颜色是黑色，while循环即可终止，所以这一过程只做一次旋转，仅需O(1)的时间。

进一步讨论：

（1）对于z的双亲是其“爷爷”结点右孩子的情形，只要将代码中的左、右互调即可写出对应的代码。

（2）只有情况1比较复杂：可能使得z一直上升至树的根结点，此时T.root.p= T.nil，是黑色，所以while循环依旧终止。

但退出循环时，T.root.color是红色（强制），最后算法强制执行**T.root.color=BLACK**，重置树根结点为黑色，这样就保持了树的红黑性质。

```
15    else (same as then clause
                with “right” and “left” exchanged)
16  T.root.color = BLACK
```

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK
6              y.color = BLACK
7              z.p.p.color = RED
8              z = z.p.p
```

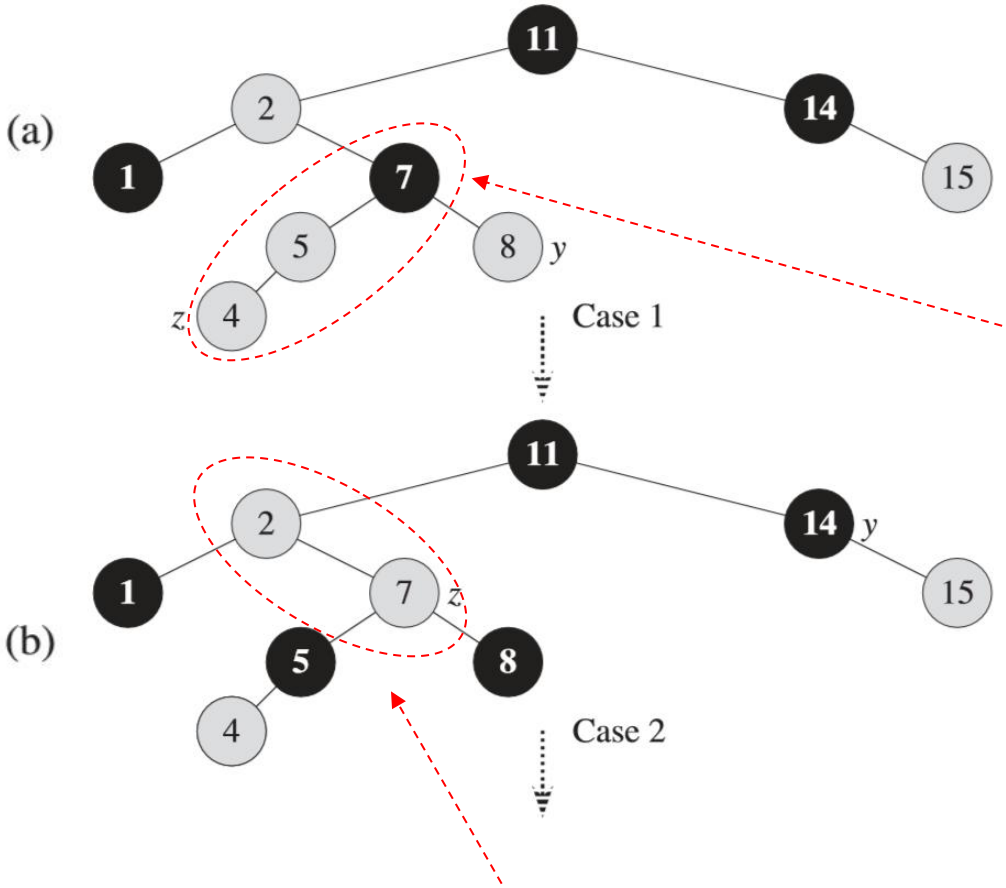
时间复杂度分析：

RB-INSERT 中，RB-INSERT-FIXUP算法之前的插入操作的时间复杂度是O(lgn) 。

RB-INSERT-FIXUP中，如果情况2发生，则从情况2转换为情况3再继续处理，总共至多两次旋转即可完成，所以情况2和情况3的时间复杂度均为O(1)。只有情况1最多需要O(lgn)的时间。所以，RB-INSERT-FIXUP算法的时间复杂度是O(lgn)。

这样，**RB-INSERT总的时间为O(lgn)**。

例：在下面的(a)图上执行RB-INSERT-FIXUP操作，**z是当前插入的新结点。**



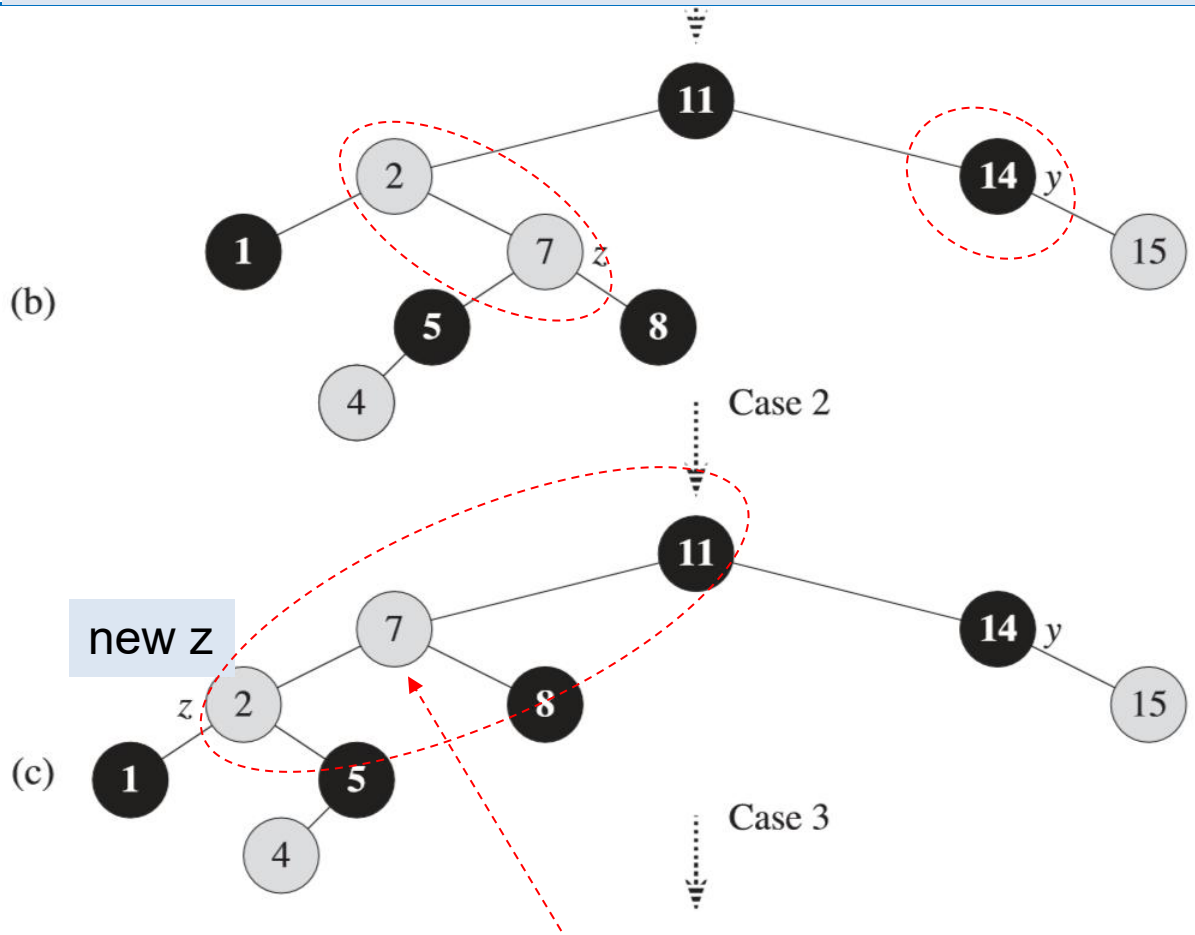
图(a)是情况1的情形：z的双亲是其爷爷的左孩子，红色，且其右“叔结点”也为红色。

图(a)修改：z.p和y置黑，**z.p.p置红**，z=z.p.p，**将冲突上传**。

然后，z指向新结点，红色，树结构变成情况2的情形（z是其双亲的右孩子）。

如图(b)所示。

图(b)： z为双亲的右结点，双亲是红色，且y是黑色

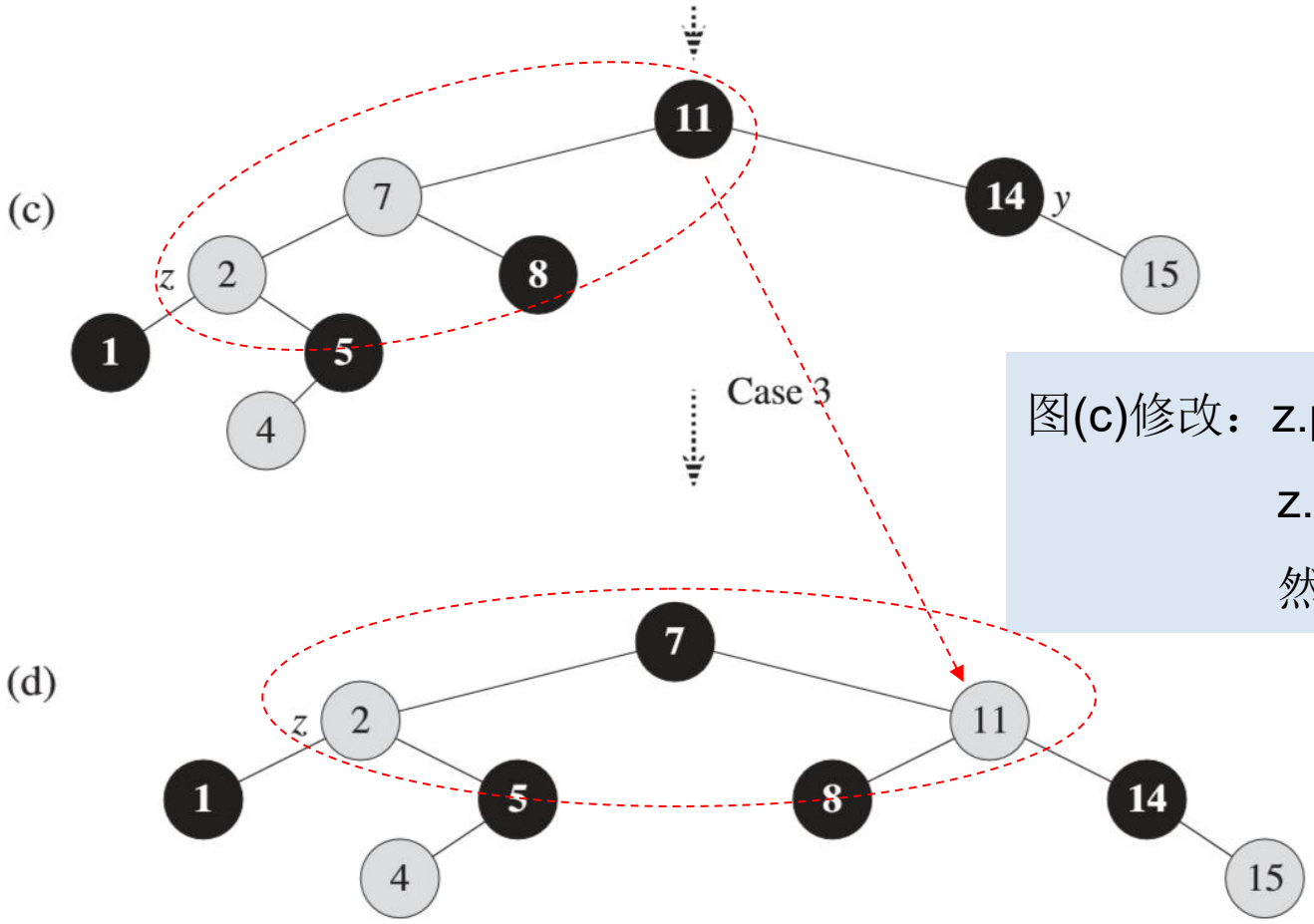


图(b)修改：z=z.p，然后**左旋**。

修改后变成情况3的情形：z是双亲的左孩子，双亲红色且其右“叔结点”是黑色。

如图(c)所示。

图(c):情况3的情形：z是双亲的左孩子，双亲红色，且右“叔结点”是黑色。



图(c)修改：z.p.color=BLACK，
z.p.p.color=RED，
然后**右旋**。

修改后的情形如图(d)所示，调整完毕。
此时的树是一棵合法的红黑树。

书上P179~182有关于**RB-INSERT-FIXUP**工作过程的进一步论述。并利用了循环不变式证明了过程的正确性，课下自学。

Thank You!

Q&A