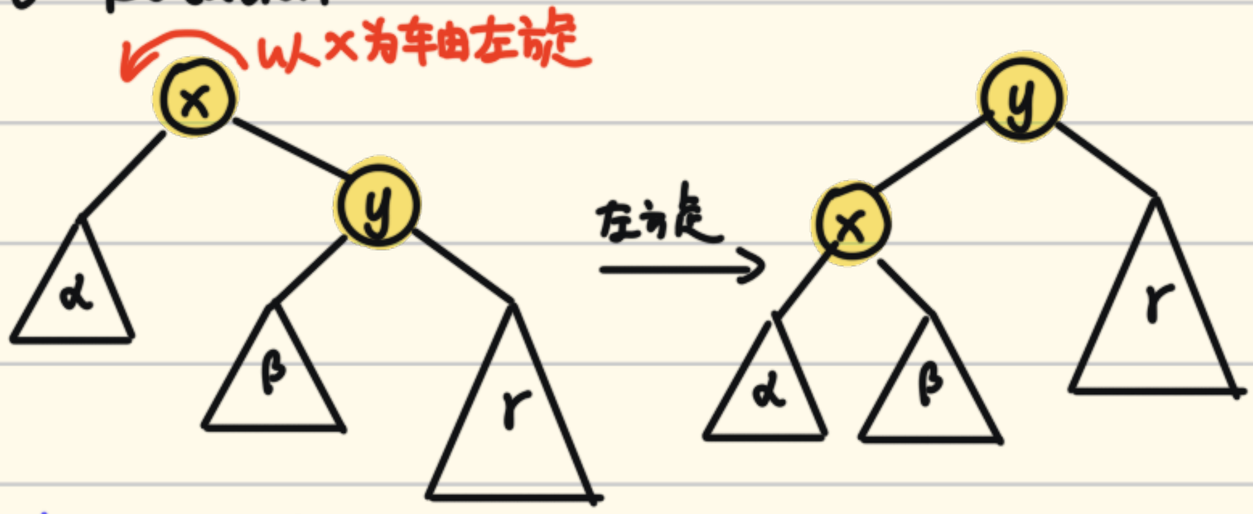
Data Structure Note 7, 8 : Binary Search Tree

(I). The rotations of a Tree

1. Left - Rotation



过继y的左儿子β

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

- 过继：

y左儿子过继给 x 当右儿子

- 父结点：

处理x.y的父结点问题

- 父子互换：

x, y 父子关系发生变化

2. Right - Rotation

- 过继：

y右儿子过继给 x 当右儿子

- 父结点：

处理x.y的父结点问题

- 父子互换：

x, y 父子关系发生变化

(II). AVL Tree

一、定义：

1. 左子树与右子树的高度差的绝对值不超过1

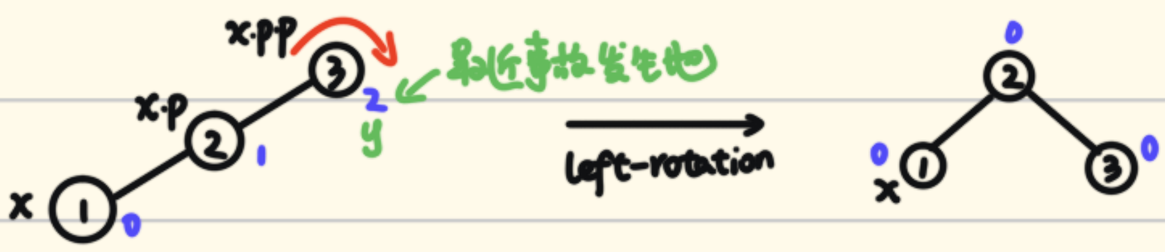
2. 左、右子树均为AVL树

显然，AVL树是平衡二叉树

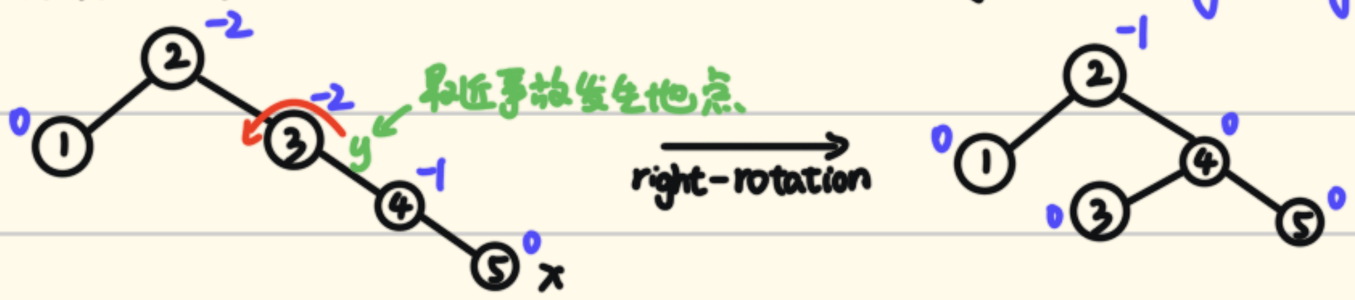
二、AVL树的插入

先完成 Naive BST 的插入后，再进行修复：

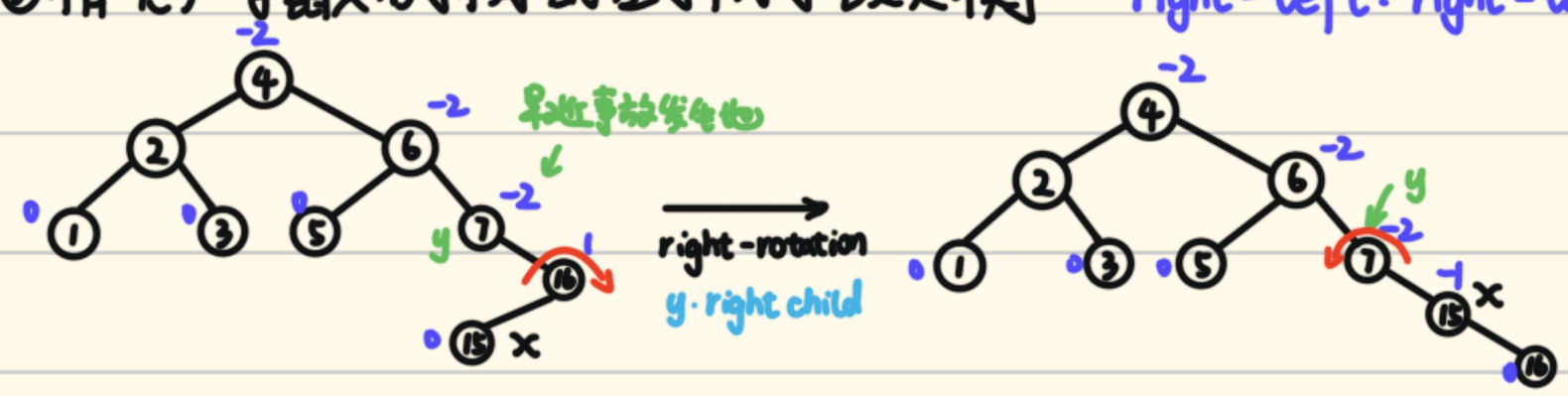
①情况1：插入左孩子的左子树导致失衡　left-left : right rotation

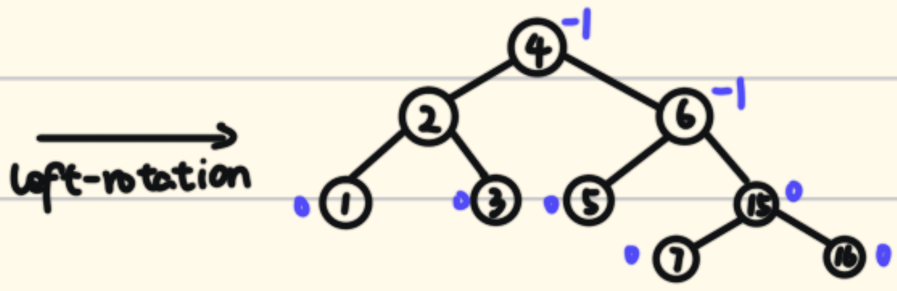


②情况2：插入右子孩子的右子树导致失衡　right-right : left rotation

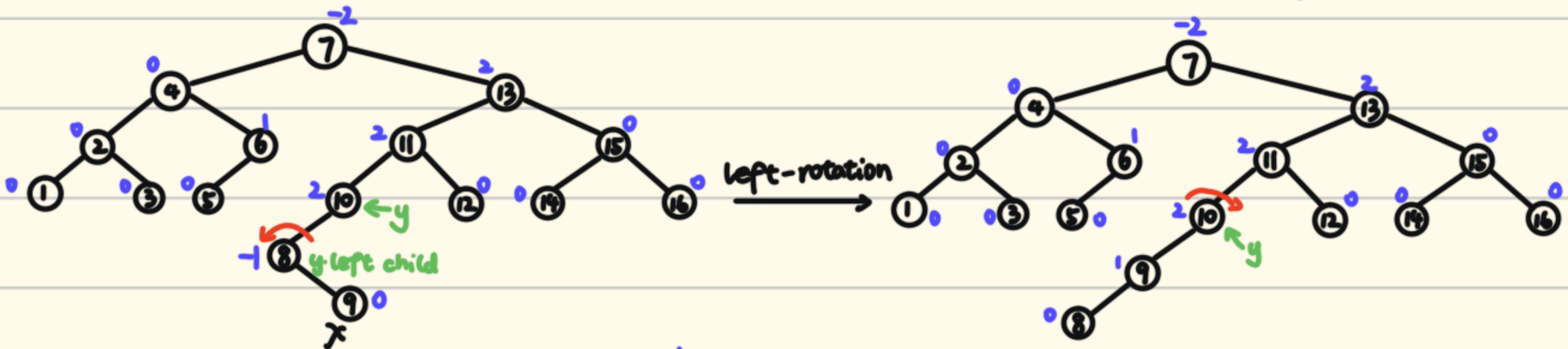


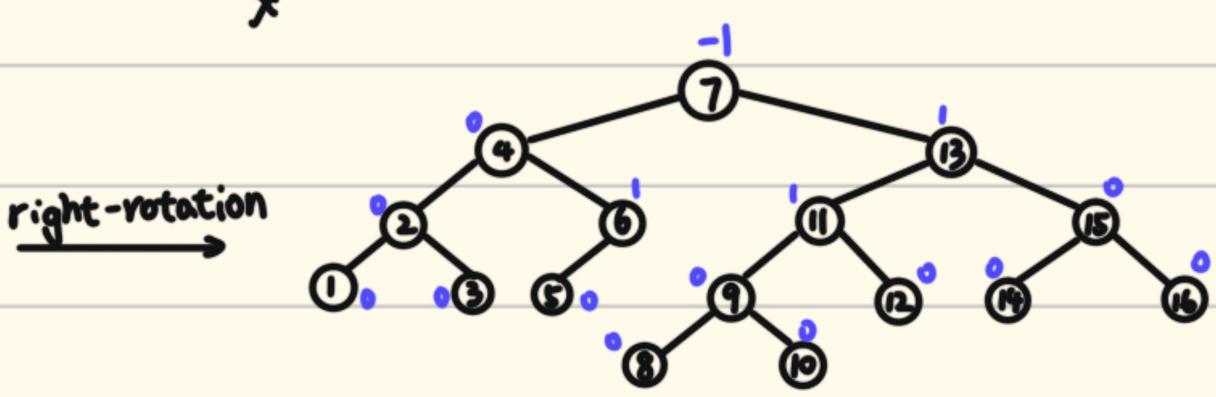
③情况3：插入右子树的左子树导致失衡　right-left : right-left rotation





④情况4：插入左子树的右子树导致失衡：　left-right : left-right rotation





7.3.4 AVL 树的删除

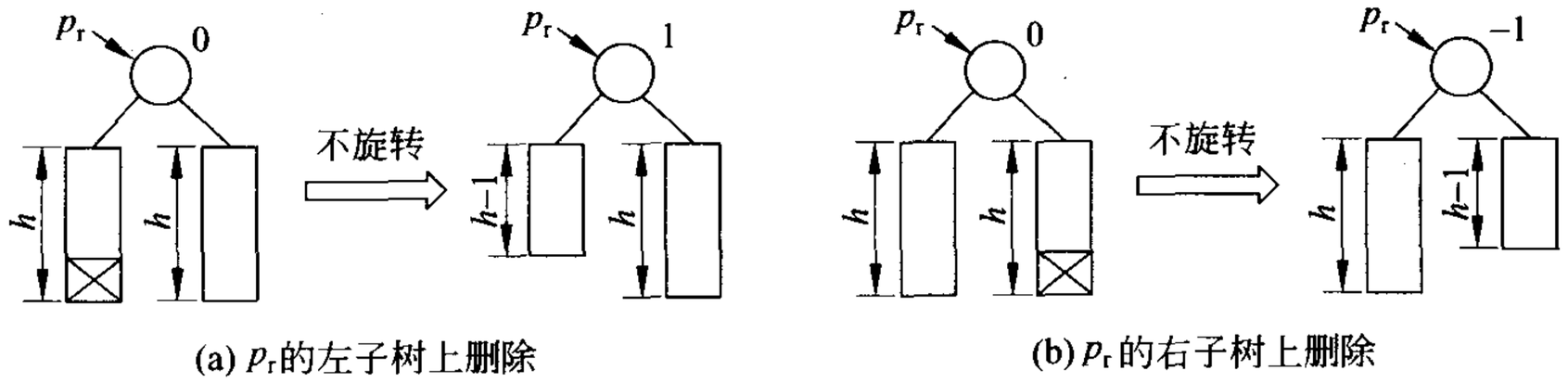
AVL 树的删除算法与二叉搜索树类似。不同之处在于：若删除后破坏了 AVL 树的高度平衡性质，还需要做平衡化旋转。

1. 如果被删结点 p 有两个子女。首先搜索 p 在中序次序下的直接前驱 q（同样可以找直接后继）。再把结点 q 的内容传送给结点 p，现在问题转移到删除结点 q。把结点 q 当作被删结点 p，它是只有一个子女的结点。

2. 如果被删结点 p 最多只有一个子女 q。可以简单地把 p 的父结点 p_r 中原来指向 p 的指针改指到 q；如果结点 p 没有子女，p 父结点 p_r 的相应指针置为 NULL。然后将原来以结点 p_r 为根的子树的高度减 1，并沿 p_r 通向根的路径反向追踪高度的这一变化对路径上各个结点的影响。

考查结点 q 的父结点 p_r。若 q 是 p_r 的左子女，则 p_r 的平衡因子应当增加 1，否则减少 1。根据修改后的 p_r 的平衡因子值，按三种情况分别进行处理：

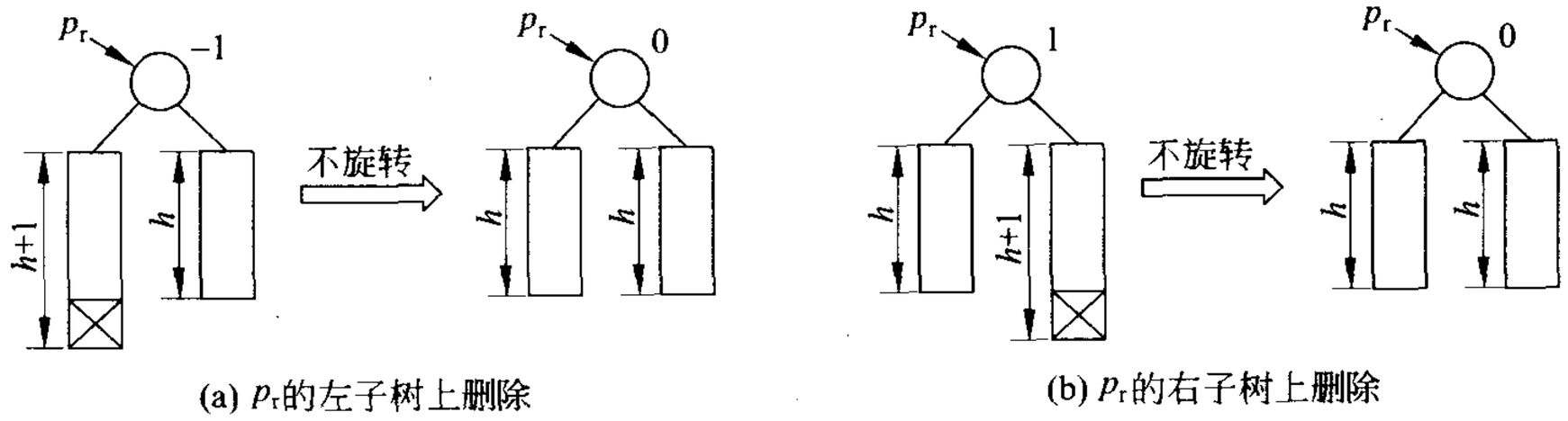
（1）p_r 的平衡因子原来为 0，在它的左子树或右子树被缩短后，则它的平衡因子改为 1 或 −1。由于以 p_r 为根的子树高度没有改变，从 p_r 到根结点的路径上所有结点都不需要调整。此时可结束本次删除的重新平衡过程。参看图 7.20。



(a) p_r的左子树上删除　　(b) p_r的右子树上删除

图 7.20　不旋转且无需平衡化旋转的情形

（2）结点 p_r 的平衡因子原不为 0，且较高的子树被缩短，则 p 的平衡因子改为 0。此时以 p_r 为根的子树平衡，但其高度减 1。为此需要继续考查结点 p_r 的父结点的平衡状态。参看图 7.21。



(a) p_r的左子树上删除　　(b) p_r的右子树上删除

图 7.21　不平衡但需继续向上做平衡化处理

（3）结点 p_r 的平衡因子原不为 0，且较矮的子树又被缩短，则在结点 p_r 发生不平衡。为此需进行平衡化旋转来恢复平衡。令 p_r 的较高的子树的根为 q（该子树未被缩短），根据 q 的平衡因子，有如下 3 种平衡化操作。

① 如果 q 的平衡因子为 0，执行一个单旋转来恢复结点 p_r 的平衡，如图 7.22 所示。图

中是左单旋转的例子，右单旋转的情形可以对称地处理。由于平衡旋转后以 q 为根的子树的高度没有发生改变，从 q 到根的路径上所有结点的平衡因子不变。此时可以结束重新平衡的过程。

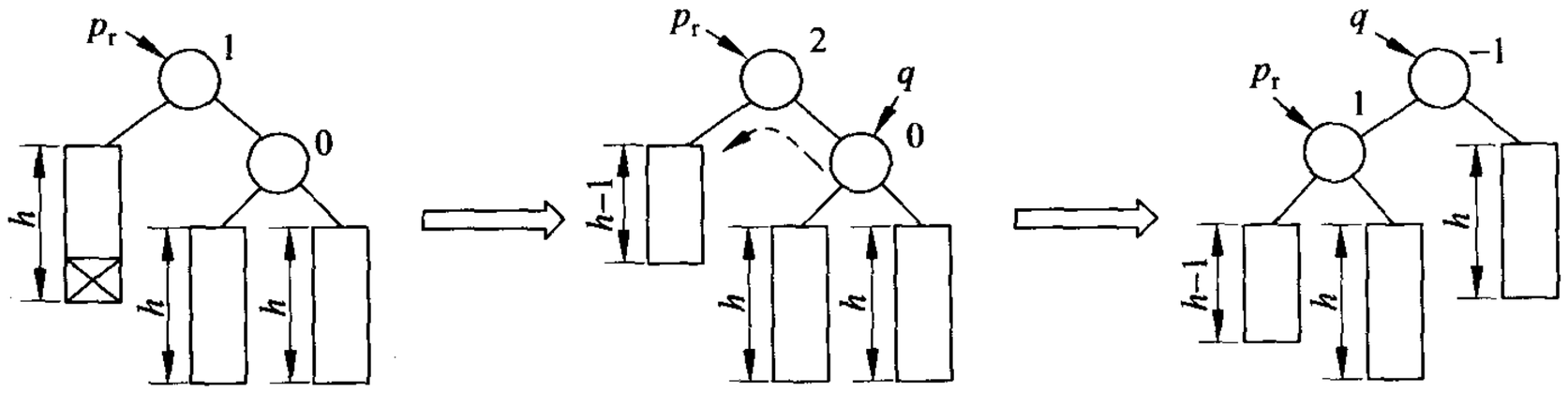


图 7.22　第一种单旋转平衡化处理

② 如果 q 的平衡因子与 p_r 的平衡因子（正负）号相同，则执行一个单旋转来恢复平衡，结点 p_r 和 q 的平衡因子均改为 0。如图 7.23 所示。图中是左单旋转的例子，右单旋转的情形可以对称地处理。由于经过平衡旋转后结点 q 的子树高度降低 1，故需要继续沿插入路径向上考查结点 q 的父结点的平衡状态，即将当前考查结点向根结点方向上移。

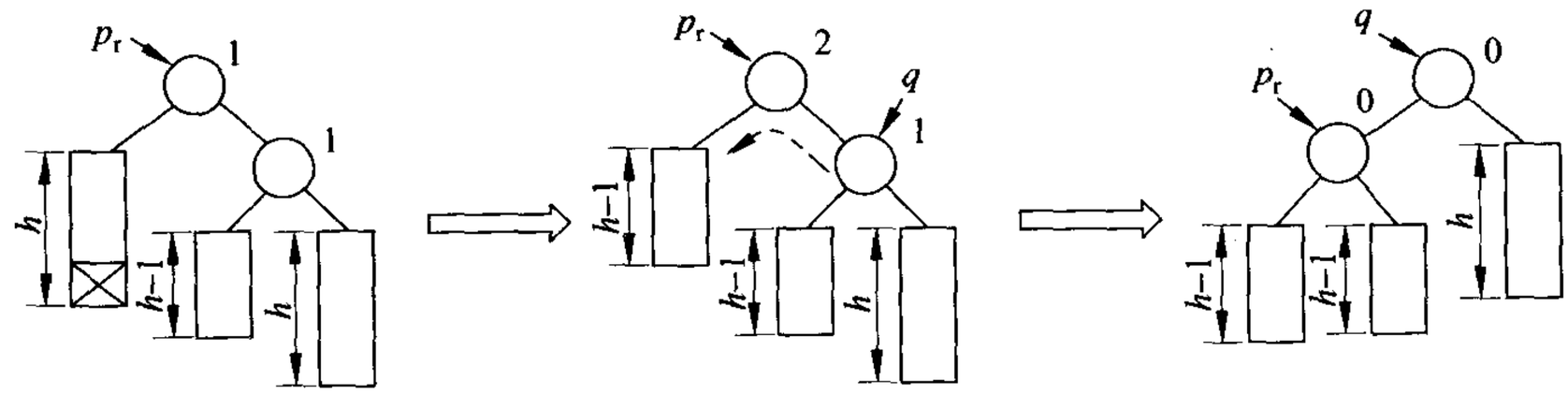


图 7.23　第二种单旋转平衡化处理

③ 如果 p_r 与 q 的平衡因子（正负）号相反，则执行一个双旋转来恢复平衡，先围绕 q 转再围绕 p_r 转。新的根结点的平衡因子置为 0，其他结点的平衡因子相应处理，同时由于经过平衡化处理后子树的高度降低 1，还需要考查它的父结点，继续向上层进行平衡化工作。参看图 7.24。

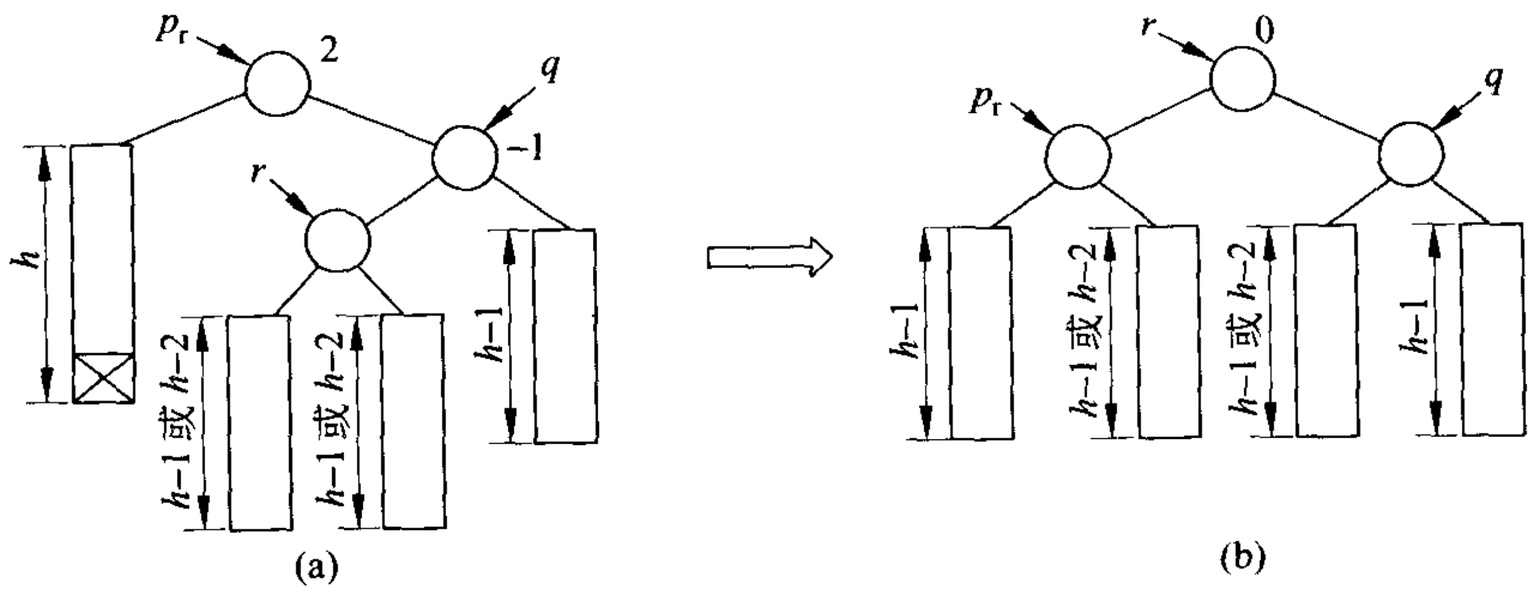


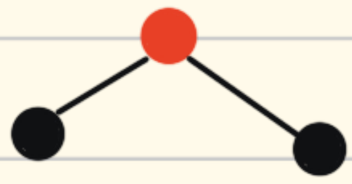
图 7.24　双旋转平衡化处理

程序 7.22 是删除算法，其中用到一个栈，在搜索被删除结点时记忆从根到被删除结点的路径，以便从下向上进行平衡化。

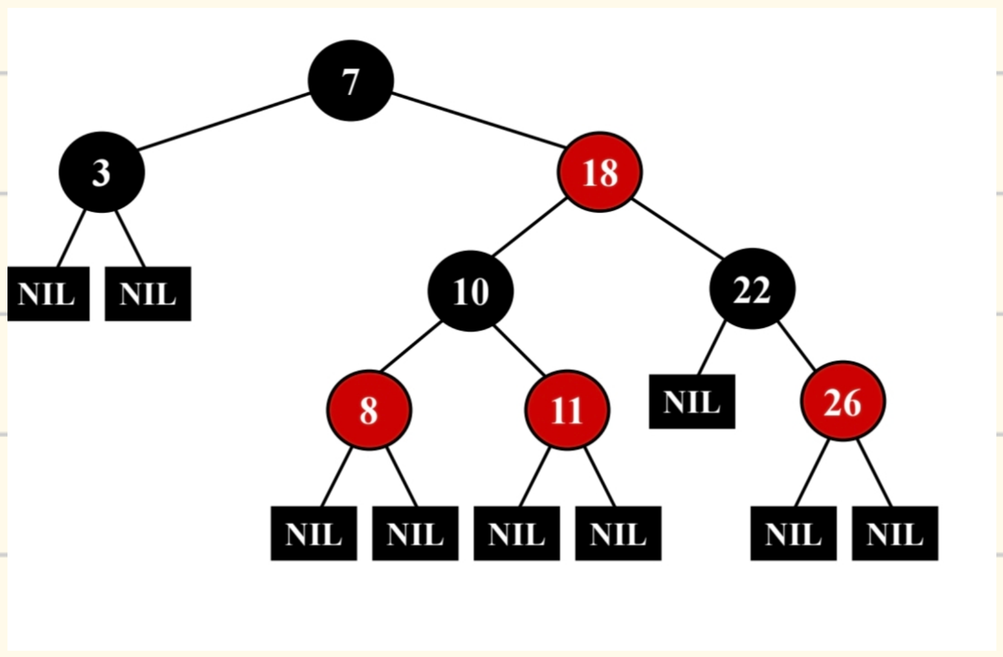
[1]. Red-Black Tree

一、红黑树的定义：

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)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



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。(黑高bh相等)



二、证明红黑树是平衡二叉树

下面的引理说明了为什么红黑树是一种好的搜索树。

引理 13.1 一棵有 n 个内部结点的红黑树的高度至多为 $2\lg(n+1)$。

证明 先证明以任一结点 x 为根的子树中至少包含 $2^{\mathrm{bh}(x)}-1$ 个内部结点。要证明这点，对 x 的高度进行归纳。如果 x 的高度为 0，则 x 必为叶结点($T.nil$)，且以 x 为根结点的子树至少包含 $2^{\mathrm{bh}(x)}-1=2^0-1=0$ 个内部结点。对于归纳步骤，考虑一个高度为正值且有两个子结点的内部结点 x。每个子结点有黑高 $\mathrm{bh}(x)$或 $\mathrm{bh}(x)-1$，其分别取决于自身的颜色是红还是黑。由于 x 子结点的高度比 x 本身的高度要低，可以利用归纳假设得出每个子结点至少有 $2^{\mathrm{bh}(x)-1}-1$ 个内部结点的结论。于是，以 x 为根的子树至少包含$(2^{\mathrm{bh}(x)-1}-1)+(2^{\mathrm{bh}(x)-1}-1)+1=2^{\mathrm{bh}(x)}-1$ 个内部结点，因此得证。

为完成引理的证明，设 h 为树的高度。根据性质 4，从根到叶结点(不包括根结点)的任何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 $h/2$；于是有

$$n \geqslant 2^{h/2}-1$$

把 1 移到不等式的左边，再对两边取对数，得到 $\lg(n+1)\geqslant h/2$，或者 $h\leqslant 2\lg(n+1)$。

由该引理可知，动态集合操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 可在红黑树上在 $O(\lg n)$时间内执行，因为这些操作在一棵高度为 h 的二叉搜索树上的运行时间为$O(h)$(参见第 12 章)，而任何包含 n 个结点的红黑树又都是高度为$O(\lg n)$的二叉搜索树。(当然，在第 12 章的算法中，NIL 的引用必须用 $T.nil$ 来代替。)虽然当给定一棵红黑树作为输入时，第 12 章的算法 TREE-INSERT 和 TREE-DELETE 的运行时间为 $O(\lg n)$，但是这两个算法并不直接支持动态集合操作 INSERT 和 DELETE，因为它们并不能保证被这些操作修改过的二叉搜索树仍是红黑树。那么如何在时间 $O(\lg n)$内支持这两个操作呢，我们将在 13.3 节和 13.4 节中介绍。

三、插入

1. Insert：先完成 Naive BST 的插入后，再进行修复

```
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)
```

插入结点后，把该结点设为红色

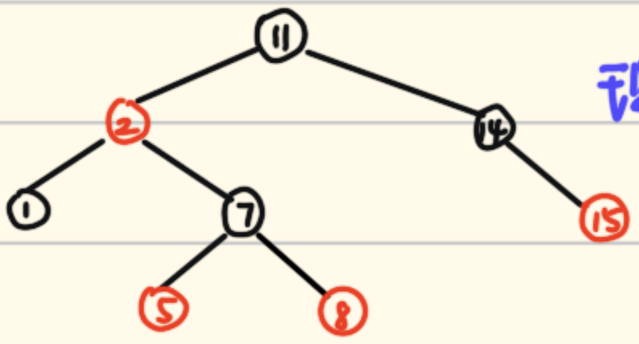
我们在Fix-up中处理红黑树定义2(根结点为黑)、定义4(红不相邻)的冲突

插入结点后，如果抬头看父亲是黑色，则无需 Fix-up；若为红，则要 Fix-up

2. Insert Fix-up：　　Insert-Fix-up(T, z)　(定义y为z的伯结点)

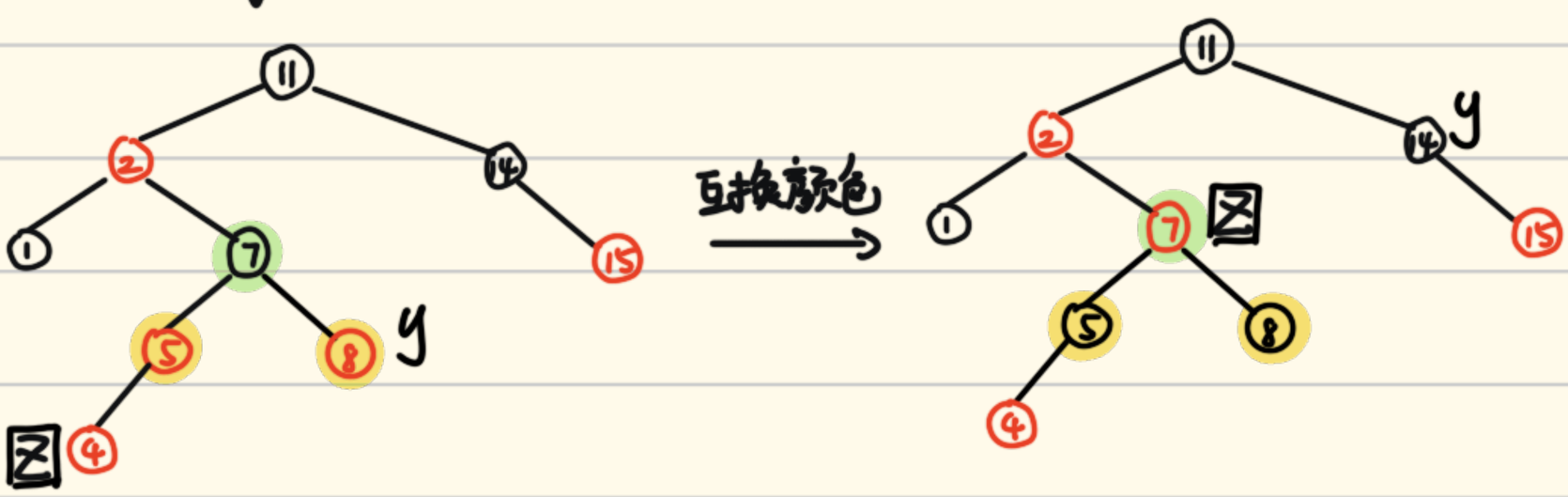
① 情况1：如果伯结点为红色：

(父结点、伯结点)与祖父节点互换颜色即可，并将z移至祖父处



理解：互换颜色代价最低，但只有伯和父均为红才可操作

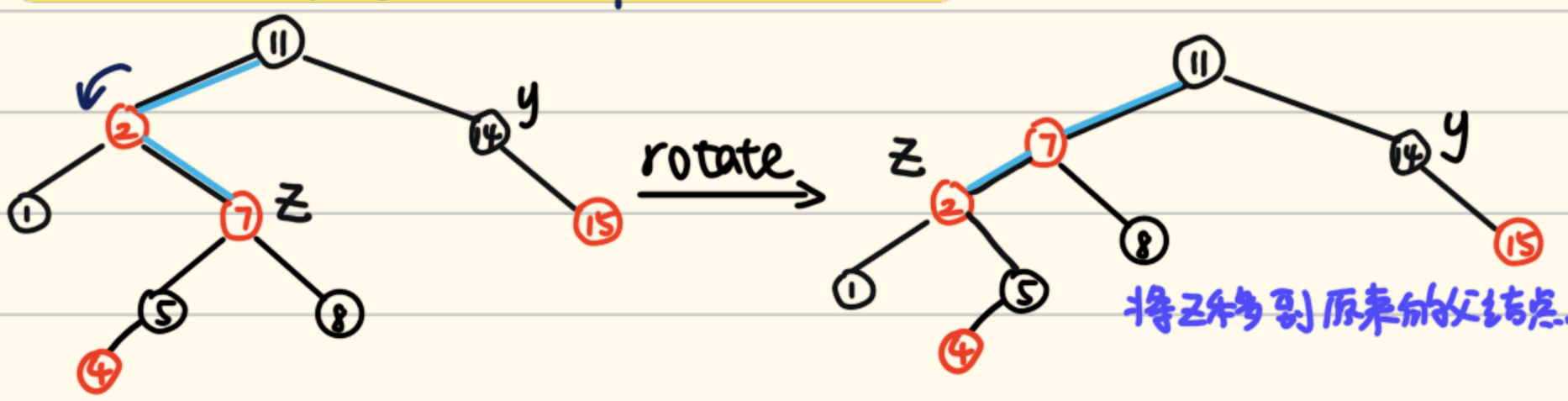
将z移到其祖父节点，若冲突继续操作



② 情况2：如果伯结点为黑色，且z和z的父结点是不同侧的孩子

即 (z==z.p.left && z.p==z.p.p.right) || (z==z.p.right && z.p == z.p.p.left)

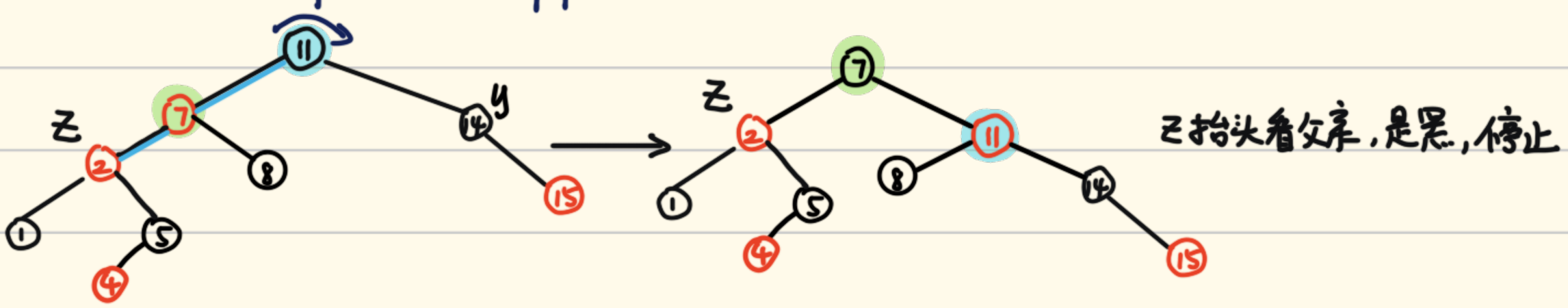
旋转转为情况3(z与z.p是同侧孩子)再处理



③情况3：如果伯结点为黑色，且z和z的父结点是同一侧的孩子

即 (z==z.p.left && z.p==z.p.p.right)||(z==z.p.right && z.p == z.p.p.left)

把原来的z.p置为黑，z.p.p置为红，以祖父结点为轴进行旋转



3. 红黑树的插入总结：

① 口诀：

看父：若父黑，则停。若父红，往下。

看伯：若伯红，互换颜色(case 1)。若伯黑，往下。

看父是否与我同侧：若异侧(case 2)，以父为轴旋转至同侧。往下。

父置黑，祖父置红，以祖父为轴旋转。(case 3)

②如遇case1：互换颜色后z更新为z.p.p，可能一直往上直至根部

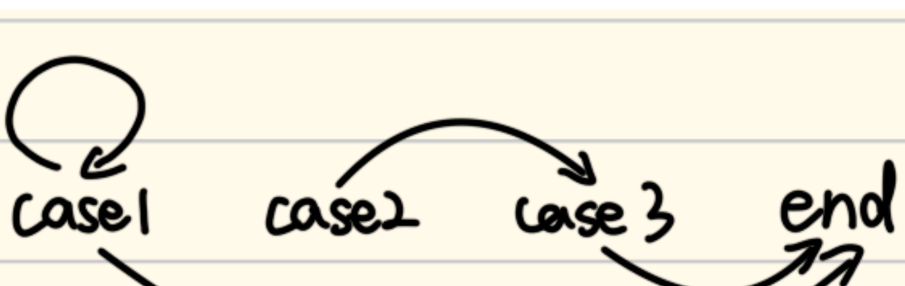
如遇 case 2：一次case 2操作后再进行一次 case 3操作即结束

如遇 case 3：一次case 3操作后即停止

⇒ 插入时间复杂度：O(h) = O(lgn)

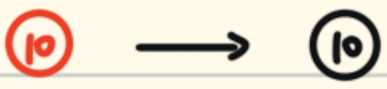
```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                  // case 1
 6              y.color = BLACK                    // case 1
 7              z.p.p.color = RED     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

Types		Operation
z's father is ***left*** *child*	***Case* 1L:** *z's uncle is red.*	*Change color.*
	Case* 2L:** *z's uncle is black and z is* ***right *child.*	***Left*** *rotation,* $p(z)$.
	Case* 3L:** *z's uncle is black and z is* ***left *child.*	***Right*** *rotation,* $p(p(z))$.
z's father is ***right*** *child*	***Case* 1R:** *z's uncle is red.*	*Change color.*
	Case* 2R:** *z's uncle is black and z is* ***left *child.*	***Right*** *rotation,* $p(z)$.
	Case* 3R:** *z's uncle is black and z is* ***right *child.*	***Left*** *rotation,* $p(p(z))$.



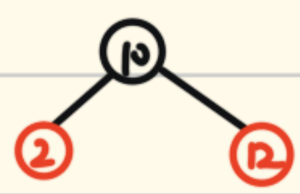
4. 红黑树的插入练习：10，2，12，4，6，8，1，9，7，3，11，5

10：

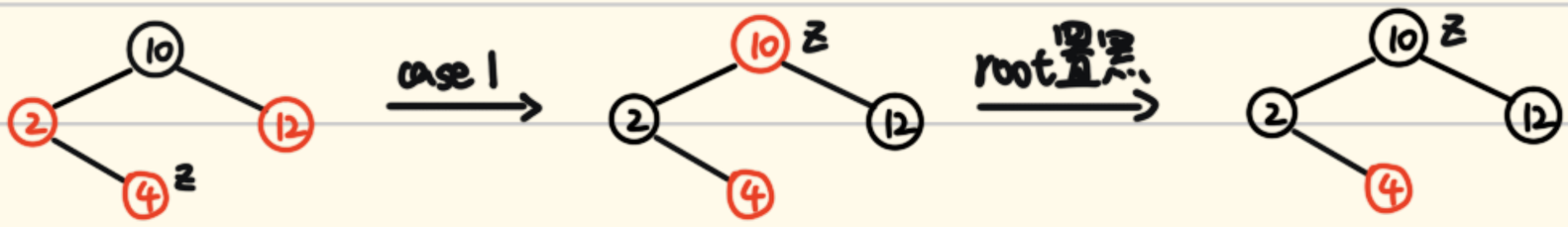


2：

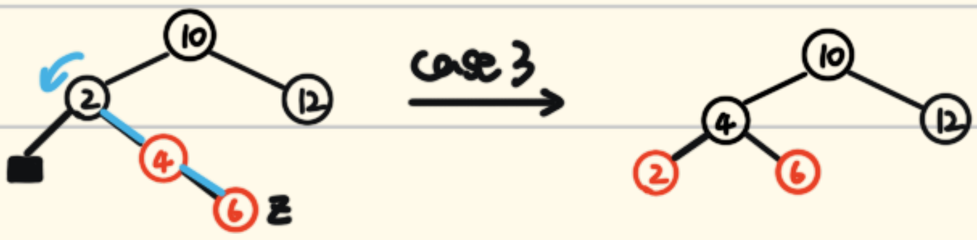
12：



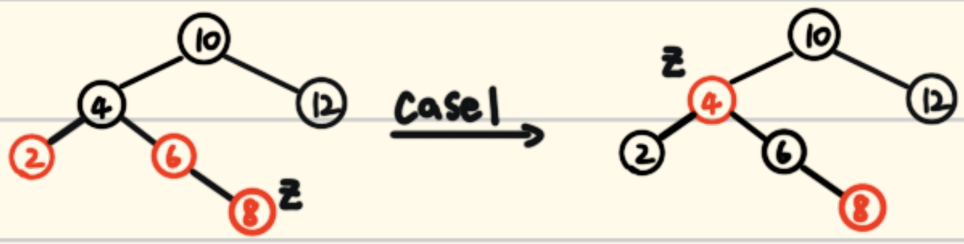
4：



6：

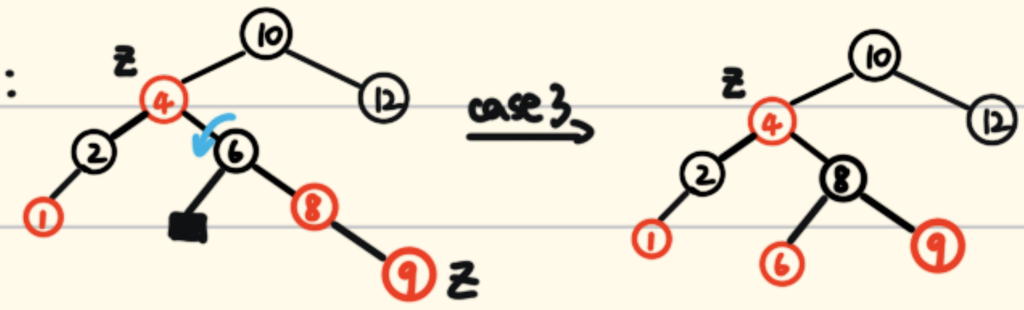


8：

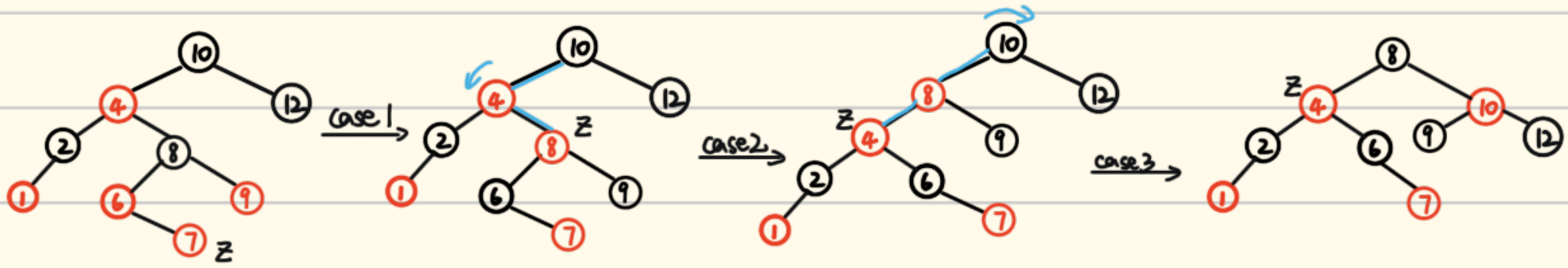


1：

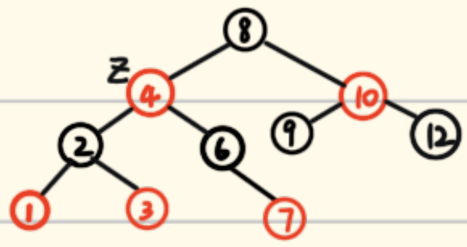
9：



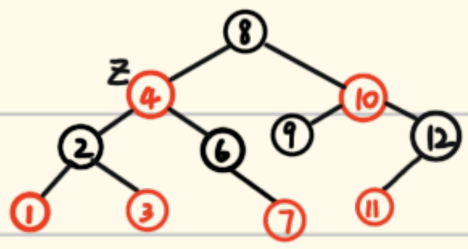
7：



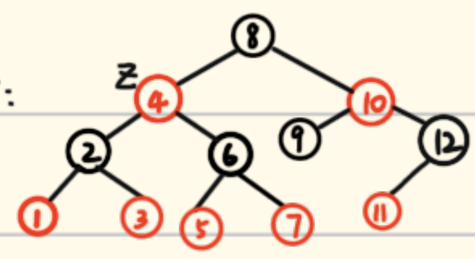
3：



11：



5：



四.删除

1. Delete：先完成Naive-BST的删除操作后，再对最后一次删除处进行Fix-up.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

y：最后一次要删的位置
（z或z的继承者）

x：删去y后补上来的结点

//如果删红，无需调整（两个黑在一起没关系）

对于x处调用Fix-up（仅有y为黑时才调用）

2. Delete Fix-up　　Delete-Fix-up(T, x)

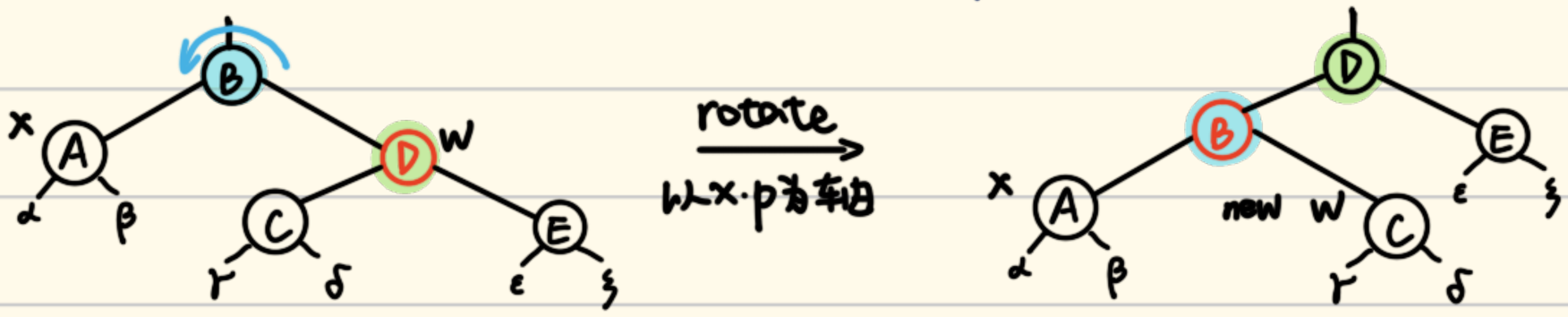
x是补位上来的结点，定义w为x的兄弟

若x为红，只要把x变黑就可弥补被删去的黑；否则若x为黑，它承担了"两份黑的责任"，要进一步调整

即如果x是红色，把x置为黑色的结束战斗，否则：

①情况1：x的兄弟w是红色的

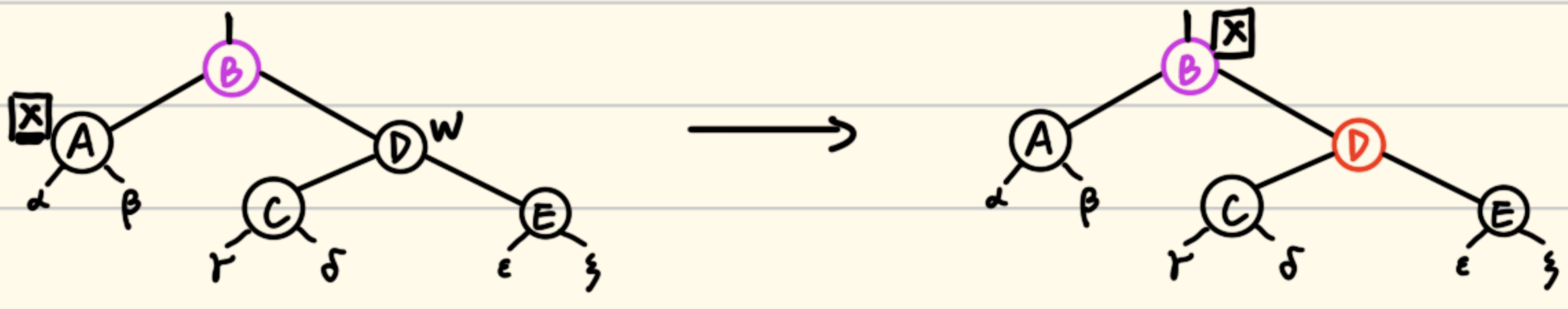
从x.p为轴旋转变，转化为情况2.3.4



互换x.p与老w的颜色

②情况2：x的兄弟结点w是黑色的，且w的两个儿子均为黑，

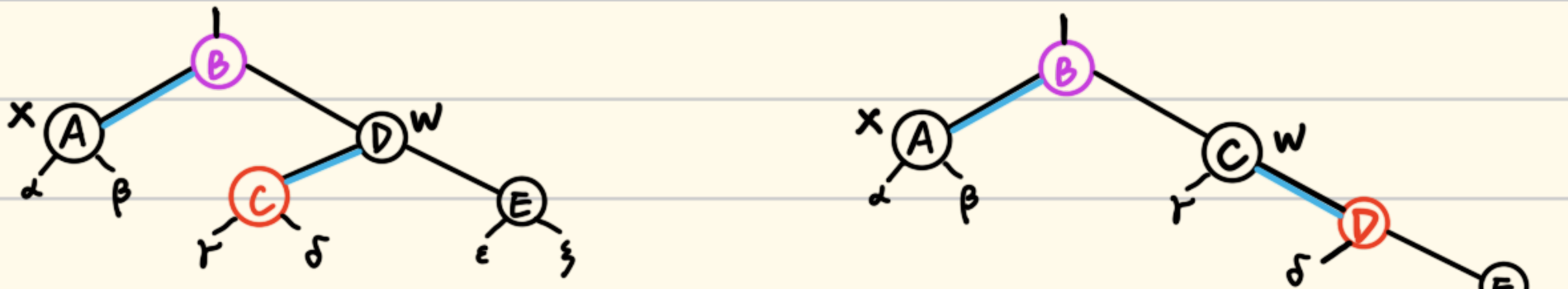
把w置为红色，并将x移至x.p，继续



理解：将D设为红色抵消了一份黑色，但同时导致B要"承担更多一份的黑色"

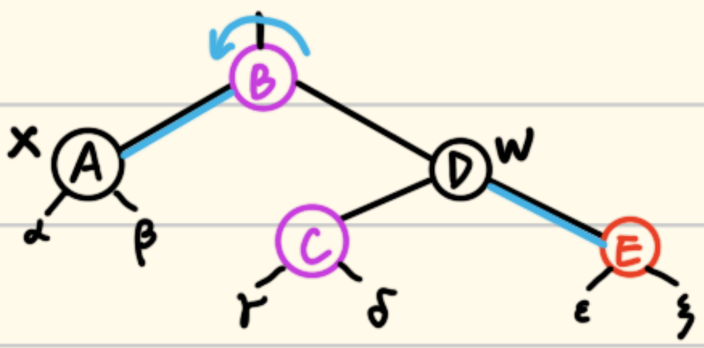
③情况3：X的兄弟结点是黑色的，W的与x同侧的孩子为红，W另一侧孩子为黑

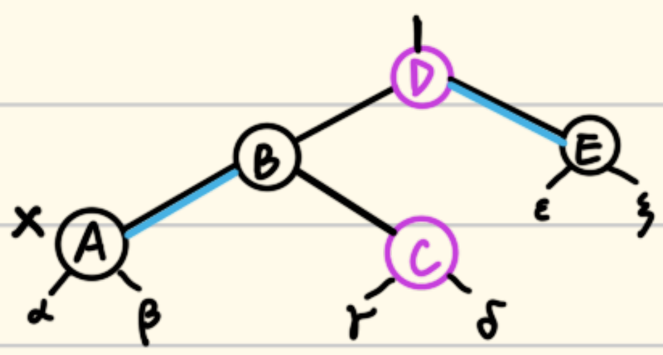
从W为中间进行右旋，转变化为情况4



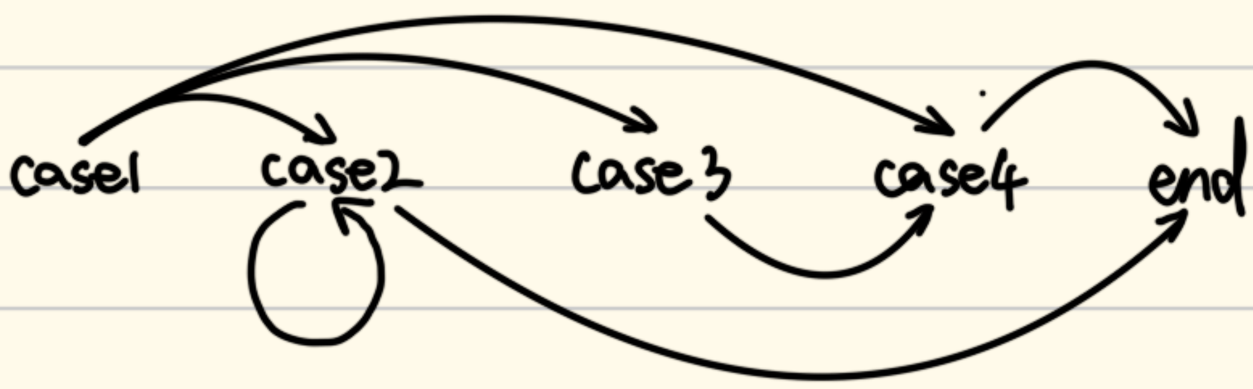
④情况4：X的兄弟结点是黑色的，W的与x异侧的孩子为红

从x.p为轴旋转，将异侧侄子置黑，对换B、D的颜色






```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                        // case 1
 6              x.p.color = RED                        // case 1
 7              LEFT-ROTATE(T, x.p)                    // case 1
 8              w = x.p.right                          // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                          // case 2
11              x = x.p                                // case 2
12          else if w.right.color == BLACK
13              w.left.color = BLACK                   // case 3
14              w.color = RED                          // case 3
15              RIGHT-ROTATE(T, w)                     // case 3
16              w = x.p.right                          // case 3
17          w.color = x.p.color                        // case 4
18          x.p.color = BLACK                          // case 4
19          w.right.color = BLACK                      // case 4
20          LEFT-ROTATE(T, x.p)                        // case 4
21          x = T.root                                 // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```



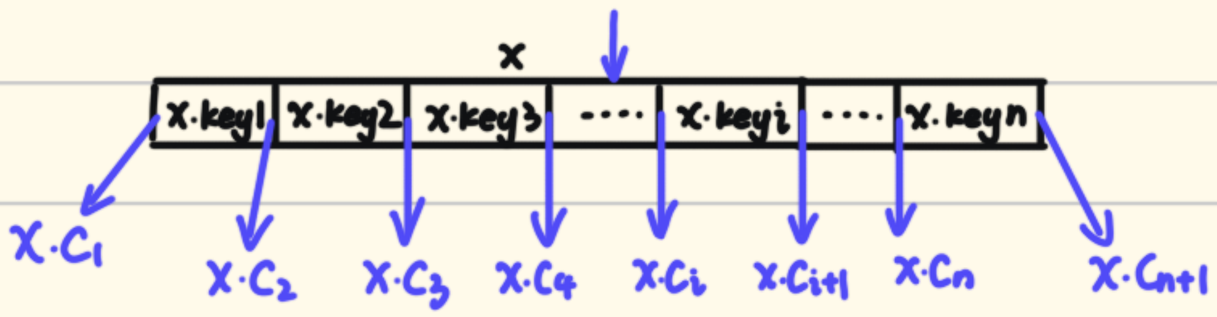
删除

1. 看实际删除结点。红：停止；黑：往下。
2. 看继承结点。红：改色、停止；黑：往下。
3. 看兄弟。红：旋转（按我侧方向，以父亲为轴。重新分析）；黑：往下。
4. 看侄子。都黑：换色（产生新的待调整结点）；红：往下。
5. 看不同侧侄子。红：旋转（按我侧方向，以父亲为轴，不同侧侄子改黑）。黑：往下。
6. 预旋转（按我侧相反方向，以兄弟为轴）；旋转（按我侧方向，以父亲为轴，不同侧侄子改黑）。

(IV). B-tree

一、B树的定义：

1. B树的节点：



2. B树的定义

① 每个叶结点具有相同的深度，即树的高度h

② 每个结点所包含的关键字(key)的个数有上、下界：

设B树的最小度数为t (t≥2)：

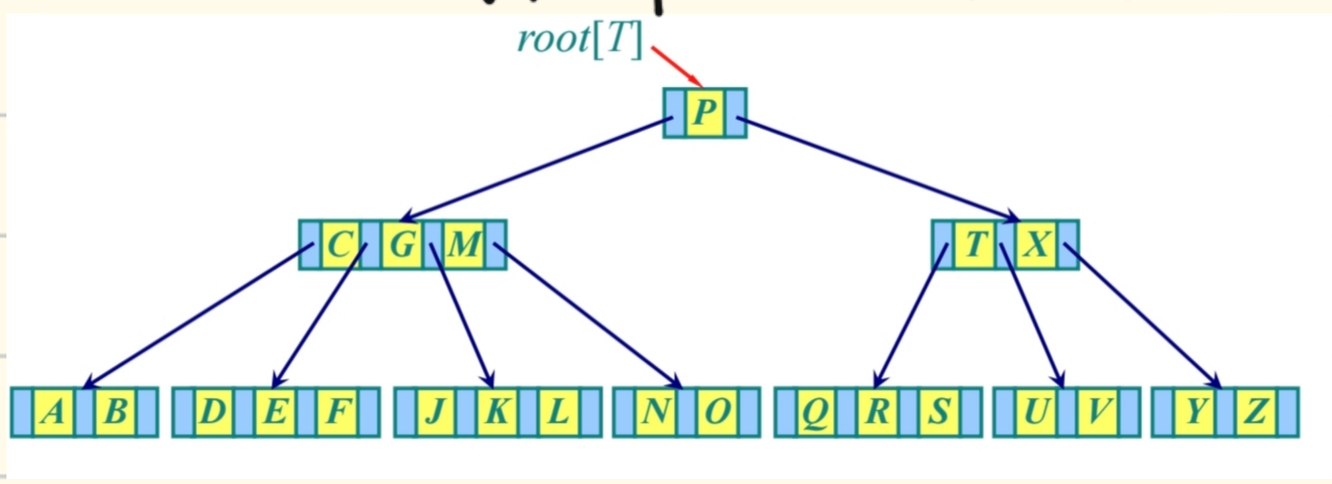
(1) 除根结点外每个结点至少有 t-1个关键字 (每个内部结点至少t个子孩子)

树非空时，根结点至少有一个关键字

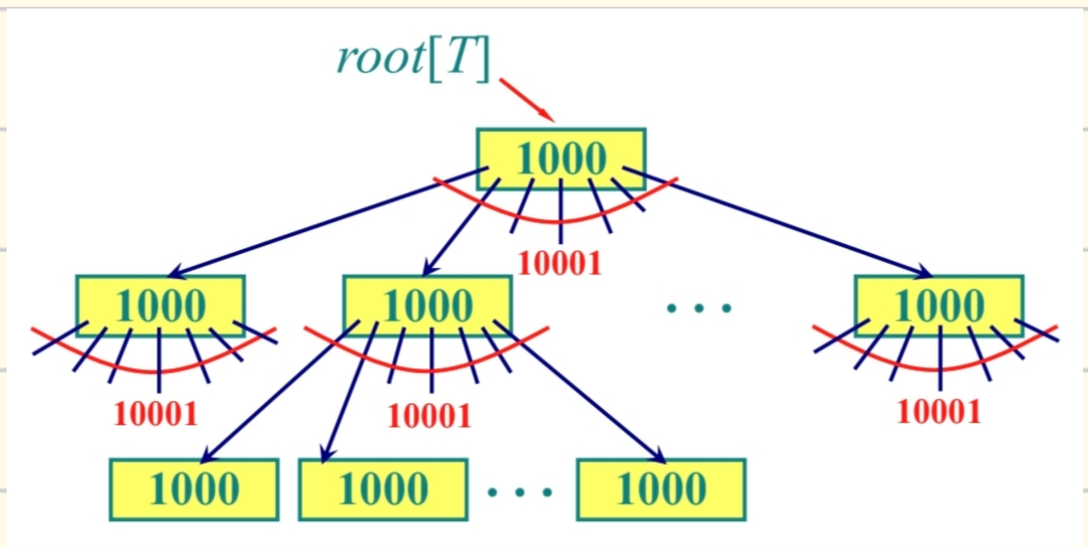
(2) 每个结点最多有 2t-1 个关键字 (每个内部结点最多有2t个子孩子)

当一个结点恰有 2t-1个关键字时，称该结点是满(full)的

t=2时，为 2-3-4树 (每个内部结点有2或3或4个孩子)



t=1000



3. B树的搜索(search)：两叉分支选择 → 多叉分支选择

二、B树的插入

1. B树结点的分裂

过程 B-TREE-SPLIT-CHILD 的输入是一个非满的内部结点 x（假定在主存中）和一个使 $x.c_i$（也假定在主存中）为 x 的满子结点的下标 i。该过程把这个子结点分裂成两个，并调整 x，使之包含多出来的孩子。要分裂一个满的根，首先要让根成为一个新的空根结点的孩子，这样才能使用 B-TREE-SPLIT-CHILD。树的高度因此增加 1；分裂是树长高的唯一途径。

图 18-5 显示了这个过程。满结点 $y=x.c_i$ 按照其中间关键字 S 进行分裂，S 被提升到 y 的父结点 x。y 中的那些大于中间关键字的关键字都置于一个新的结点 z 中，它成为 x 的一个新的孩子。

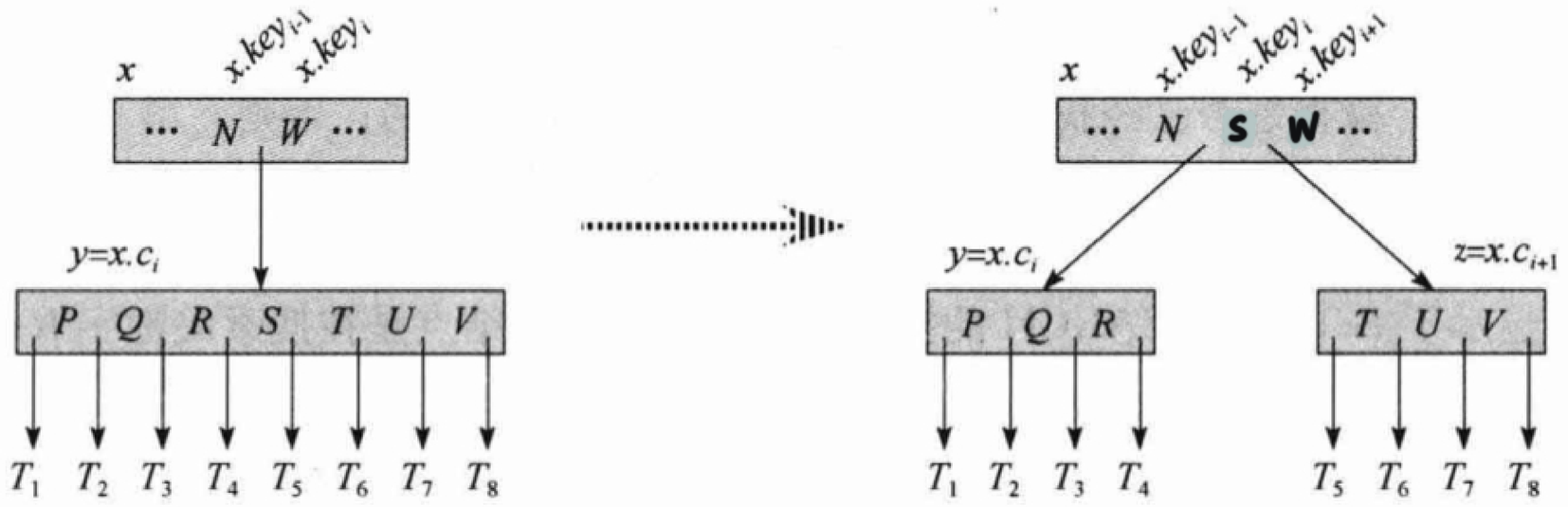


图 18-5　分裂一个 $t=4$ 的结点。结点 $y=x.c_i$ 分为两个结点 y 和 z，y 的中间关键字 S 被提升到 y 的父结点中

2t-1个key { k-1个key / 1个key / k-1个key

B-TREE-SPLIT-CHILD 以直接的“剪贴”方式工作。这里 x 是被分裂的结点，y 是 x 的第 i 个孩子（见第 2 行）。开始时，结点 y 有 $2t$ 个孩子（$2t-1$ 个关键字），在分裂后减少至 t 个孩子（$t-1$ 个关键字）。结点 z 取走 y 的 t 个最大的孩子（$t-1$ 个关键字），并且 z 成为 x 的新孩子，它在 x 的孩子表中仅位于 y 之后。y 的中间关键字上升到 x 中，成为分隔 y 和 z 的关键字。

2. 插入：

在一棵高度为 h 的 B 树 T 中，以沿树单程下行方式插入一个关键字 k 的操作需要 $O(h)$ 次磁盘存取。所需要的 CPU 时间为 $O(th)=O(t\log_t n)$。过程 B-TREE-INSERT 利用 B-TREE-SPLIT-CHILD 来保证递归始终不会降至一个满结点上。

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t - 1
 3       s = ALLOCATE-NODE()
 4       T.root = s
 5       s.leaf = FALSE
 6       s.n = 0
 7       s.c_1 = r
 8       B-TREE-SPLIT-CHILD(s, 1)
 9       B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

第 3～9 行处理了根结点 r 为满的情况：原来的根结点被分裂，一个新的结点 s（有两个孩子）成为根。对根进行分裂是增加 B 树高度的唯一途径。图 18-6 说明了这种情况。与二叉搜索树不同，B 树高度的增加发生在顶部而不是底部。过程通过调用 B-TREE-INSERT-NONFULL 完成将关键字 k 插入以非满的根结点为根的树中。B-TREE-INSERT-NONFULL 在需要时沿树向下递

归，在必要时通过调用 B-TREE-SPLIT-CHILD 来保证任何时刻它所递归处理的结点都是非满的。

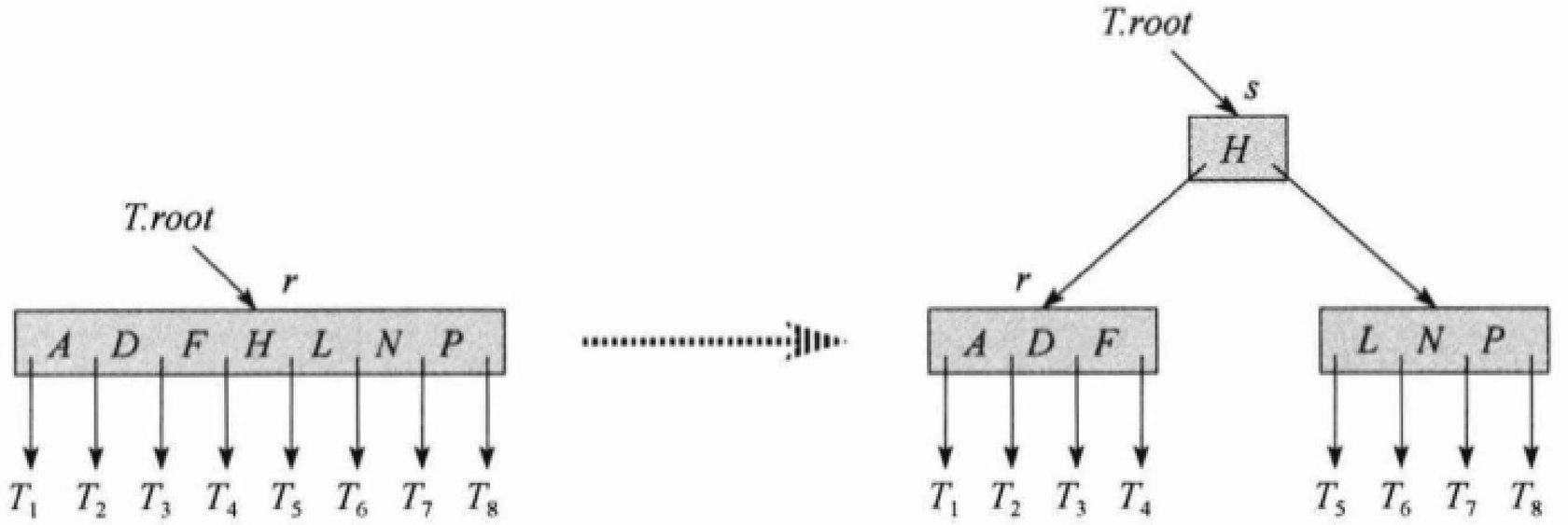


图 18-6 分裂 $t=4$ 的根。根结点 r 一分为二，并创建了一个新结点 s。新的根包含了 r 的中间关键字，且以 r 的两半作为孩子。当根被分裂时，B 树的高度增加 1

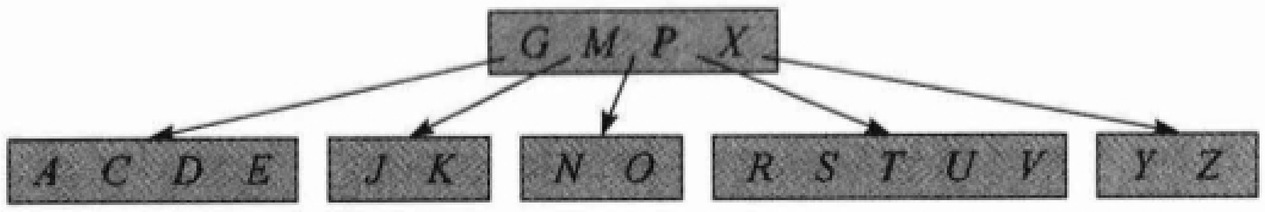
辅助的递归过程 B-TREE-INSERT-NONFULL 将关键字插入结点 x，要求假定在调用该过程时 x 是非满的。操作 B-TREE-INSERT 和递归操作 B-TREE-INSERT-NONFULL 保证了这个假
495 设成立。

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3       while i ⩾ 1 and k < x.key_i
 4            x.key_{i+1} = x.key_i
 5            i = i − 1
 6       x.key_{i+1} = k
 7       x.n = x.n + 1
 8       DISK-WRITE(x)
 9  else while i ⩾ 1 and k < x.key_i
10            i = i − 1
11       i = i + 1
12       DISK-READ(x.c_i)
13       if x.c_i.n == 2t − 1
14            B-TREE-SPLIT-CHILD(x, i)
15            if k > x.key_i
16                 i = i + 1
17       B-TREE-INSERT-NONFULL(x.c_i, k)
```

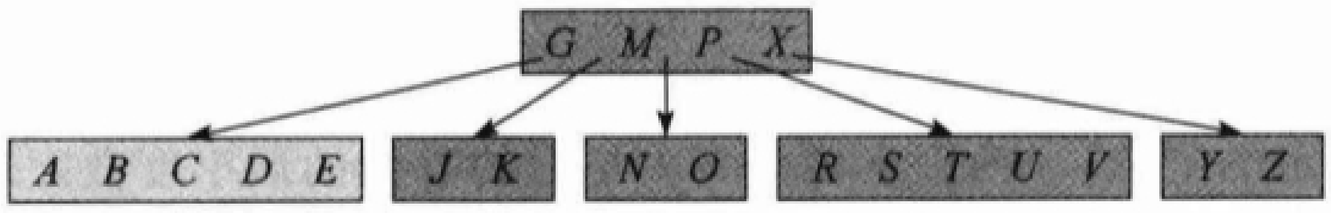
过程 B-TREE-INSERT-NONFULL 的工作方式如下。第 3～8 行处理 x 是叶结点的情况，将关键字 k 插入 x。如果 x 不是叶结点，则必须将 k 插入以内部结点 x 为根的子树中适当的叶结点中去。这种情况下，第 9～11 行决定向 x 的哪个子结点递归下降。第 13 行检查是否是递归降至了一个满子结点上，若是，则第 14 行用 B-TREE-SPLIT-CHILD 将该子结点分裂为两个非满的
496 孩子，第 15～16 行确定向两个孩子中的哪一个下降是正确的。（注意，在第 16 行中 i 增加 1 后无需做 DISK-READ($x.c_i$)，因为这种情况下递归会降至一个刚刚由 B-TREE-SPLIT-CHILD 创建的子结点上。）第 13～16 行的真正作用就是保证该程序始终不会降至一个满结点上。然后第 17 行递归地将 k 插入合适的子树中。图 18-7 说明了向 B 树中插入关键字的各种情况。

对一棵高度为 h 的 B 树来说，B-TREE-INSERT 要做 $O(h)$ 次磁盘存取，因为在每次调用 B-TREE-INSERT-NONFULL 之间，只做了 $O(1)$ 次 DISK-READ 和 DISK-WRITE 操作。所占用的总 CPU 时间为 $O(th) = O(t\log_t n)$。因为 B-TREE-INSERT-NONFULL 是尾递归的，所以它也

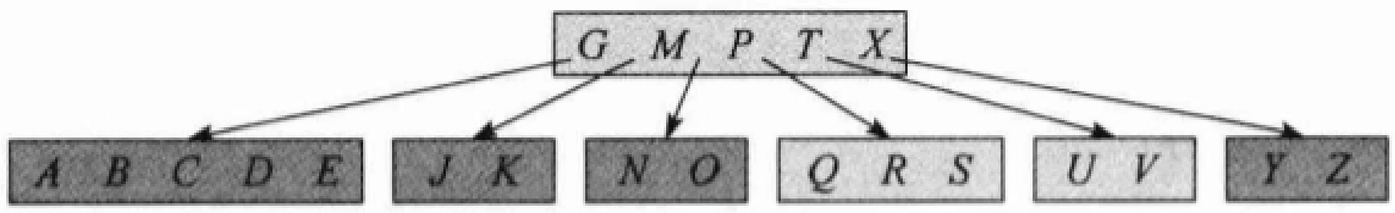
可以用一个 **while** 循环来实现，从而说明了在任何时刻，需要留在主存中的页面数为 $O(1)$。



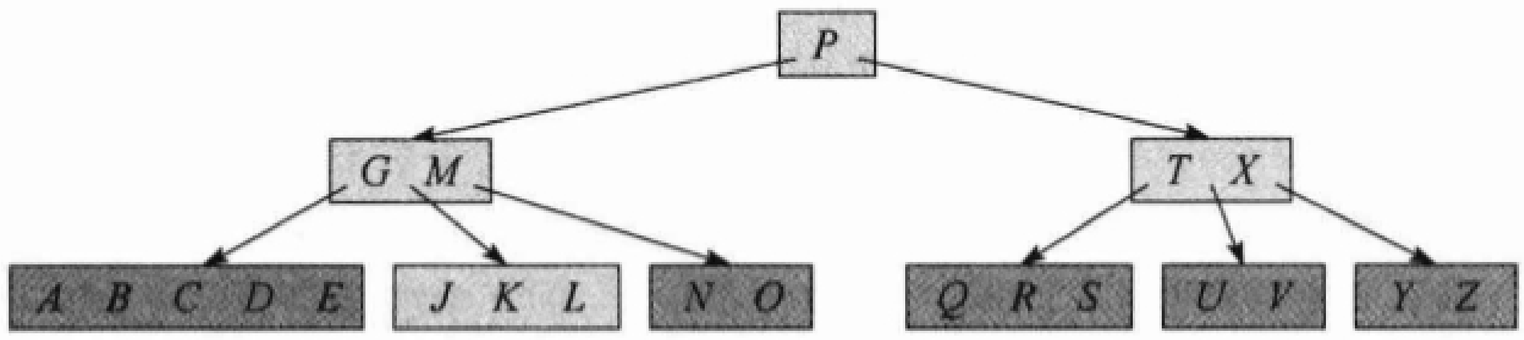
（a）初始树



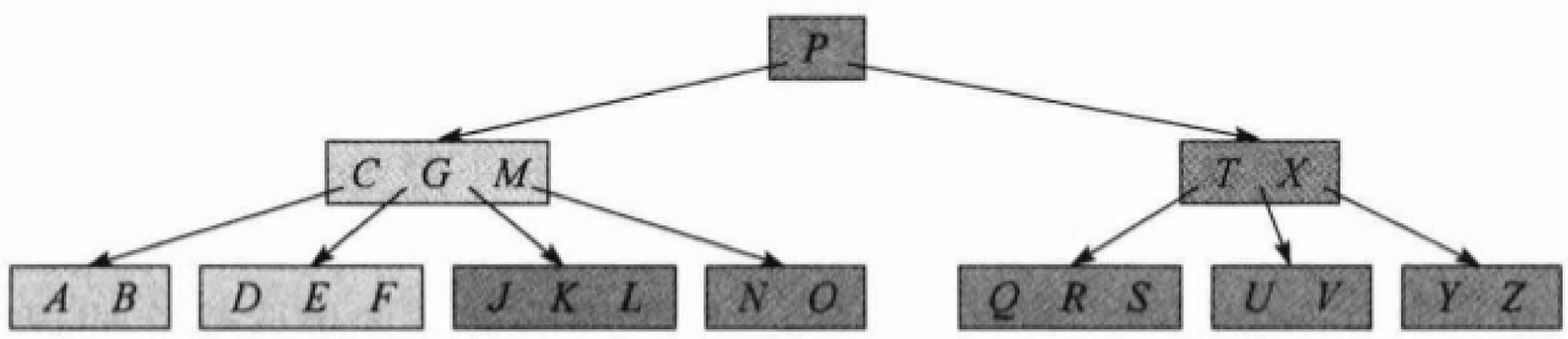
（b）插入 B



（c）插入 Q



（d）插入 L



（e）插入 F

图 18-7　向 B 树中插入关键字。这棵 B 树的最小度数 t 为 3，所以一个结点至多可包含 5 个关键字。在插入过程中被修改的结点由浅阴影标记。(a)这个例子初始时的树。(b)向初始树中插入 B 后的结果；这是一个对叶结点的简单插入。(c)将 Q 插入前一棵树中的结果。结点 $RSTUV$ 被分裂为两个分别包含 RS 和 UV 的结点，关键字 T 被提升到根中，Q 被插入两半的最左边(RS 结点)。(d)将 L 插入前一棵树中的结果。由于根结点是满的，所以它立即被分裂，同时 B 树的高度增加 1。然后 L 被插入包含 JK 的叶结点中。(e)将 F 插入前一棵树中的结果。在将 F 插入两半的最右边(DE 结点)之前，结点 $ABCDE$ 会进行分裂

练习

18.2-1　请给出关键字 F、S、Q、K、C、L、H、T、V、W、M、R、N、P、A、B、X、Y、D、Z、E 依序插入一棵最小度数为 2 的空 B 树的结果。只要画出在某些结点分裂之前的结构以及最终的结构。

18.2-2　请解释在什么情况下(如果有的话)，在调用 B-TREE-INSERT 的过程中，会执行冗余的 DISK-READ 或 DISK-WRITE 操作。(所谓冗余的 DISK-READ，是指对已经在主存中的某页做 DISK-READ。冗余的 DISK-WRITE 是指将已经存在于磁盘上的某页又完全相同地重写一遍。)

18.2-3　请说明如何在一棵 B 树中找出最小关键字，以及如何找出某一给定关键字的前驱。

三、B树的删除

B树上的删除操作与插入操作类似，只是略微复杂一点，因为可以从任意一个结点中删除一个关键字，而不仅仅是叶结点，而且当从一个内部结点删除一个关键字时，还要重新安排这个结点的孩子。与插入操作一样，必须防止因删除操作而导致树的结构违反B树性质。就像必须保证一个结点不会因为插入而变得太大一样，必须保证一个结点不会在删除期间变得太小(根结点除外，因为它允许有比最少关键字数 $t-1$ 还少的关键字个数)。一个简单插入算法，如果插入关键字的路径上结点满，可能需要向上回溯；与此类似，一个简单删除算法，当要删除关键字的路径上结点(非根)有最少的关键字个数时，也可能需要向上回溯。

过程B-TREE-DELETE从以 x 为根的子树中删除关键字 k。我们设计的这个过程必须保证无论何时，结点 x 递归调用自身时，x 中关键字个数至少为最小度数 t。注意到，这个条件要求比通常B树中的最少关键字个数多一个以上，使得有时在递归下降至子结点之前，需要把一个关键字移到子结点中。这个加强的条件允许在一趟下降过程中，就可以将一个关键字从树中删除，无需任何“向上回溯”(有一个例外，后面会解释)。对下面的B树上删除操作的规定应当这样理解，如果根结点 x 成为一个不含任何关键字的内部结点(这种情况可能出现在图18-8中的情况2c和情况3b中)，那么 x 就要被删除，x 的唯一孩子 $x.c_1$ 成为树的新根，从而树的高度降低1，同时也维持树根必须包含至少一个关键字的性质(除非树是空的)。

现在我们来简要地介绍删除操作是如何工作的，而不是给出其伪代码。图18-8描绘了从B树中删除关键字的各种情况。

1. 如果关键字 k 在结点 x 中，并且 x 是叶结点，则从 x 中删除 k。

2. 如果关键字 k 在结点 x 中，并且 x 是内部结点，则做以下操作：

a. 如果结点 x 中前于 k 的子结点 y 至少包含 t 个关键字，则找出 k 在以 y 为根的子树中的前驱 k'。递归地删除 k'，并在 x 中用 k' 代替 k。(找到 k' 并删除它可在沿树下降的单过程中完成。)

b. 对称地，如果 y 有少于 t 个关键字，则检查结点 x 中后于 k 的子结点 z。如果 z 至少有 t 个关键字，则找出 k 在以 z 为根的子树中的后继 k'。递归地删除 k'，并在 x 中用 k' 代替 k。(找到 k' 并删除它可在沿树下降的单过程中完成。)

c. 否则，如果 y 和 z 都只含有 $t-1$ 个关键字，则将 k 和 z 的全部合并进 y，这样 x 就失去了 k 和指向 z 的指针，并且 y 现在包含 $2t-1$ 个关键字。然后释放 z 并递归地从 y 中删除 k。

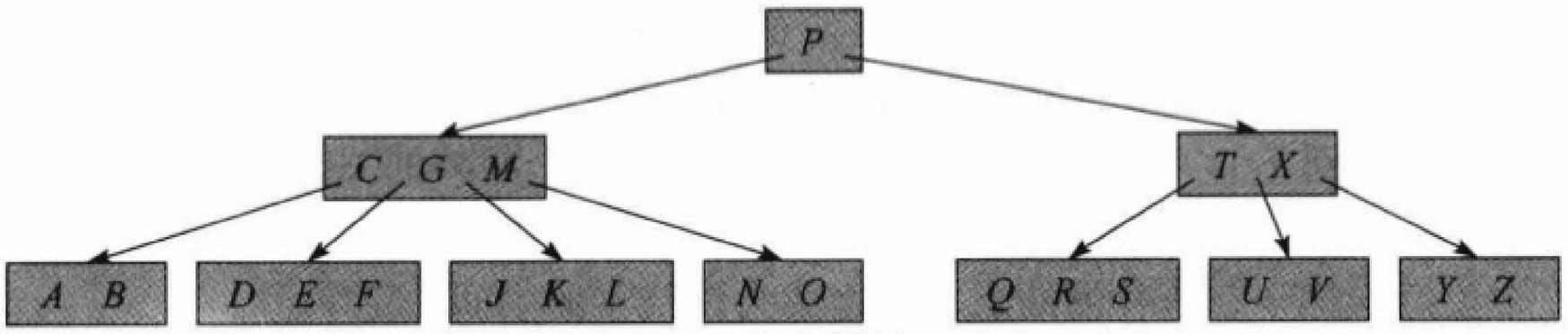
3. 如果关键字 k 当前不在内部结点 x 中，则确定必包含 k 的子树的根 $x.c_i$(如果 k 确实在树中)。如果 $x.c_i$ 只有 $t-1$ 个关键字，必须执行步骤3a或3b来保证降至一个至少包含 t 个关键字的结点。然后，通过对 x 的某个合适的子结点进行递归而结束。

a. 如果 $x.c_i$ 只含有 $t-1$ 个关键字，但是它的一个相邻的兄弟至少包含 t 个关键字，则将 x 中的某一个关键字降至 $x.c_i$ 中，将 $x.c_i$ 的相邻左兄弟或右兄弟的一个关键字升至 x，将该兄弟中相应的孩子指针移到 $x.c_i$ 中，这样就使得 $x.c_i$ 增加了一个额外的关键字。

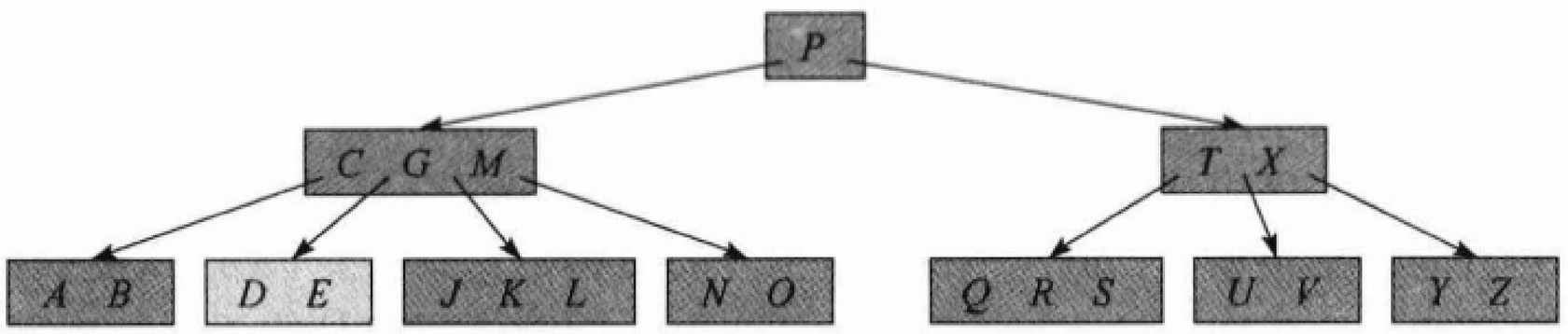
b. 如果 $x.c_i$ 以及 $x.c_i$ 的所有相邻兄弟都只包含 $t-1$ 个关键字，则将 $x.c_i$ 与一个兄弟合并，即将 x 的一个关键字移至新合并的结点，使之成为该结点的中间关键字。

由于一棵B树中的大部分关键字都在叶结点中，我们可以预期在实际中，删除操作最经常用于从叶结点中删除关键字。这样B-TREE-DELETE过程只要沿树下降一趟即可，不需要向上回溯。然而，当要删除某个内部结点的关键字时，该过程也要沿树下降一趟，但可能还要返回删除了关键字的那个结点，以用其前驱或后继来取代被删除的关键字(情况2a和情况2b)。

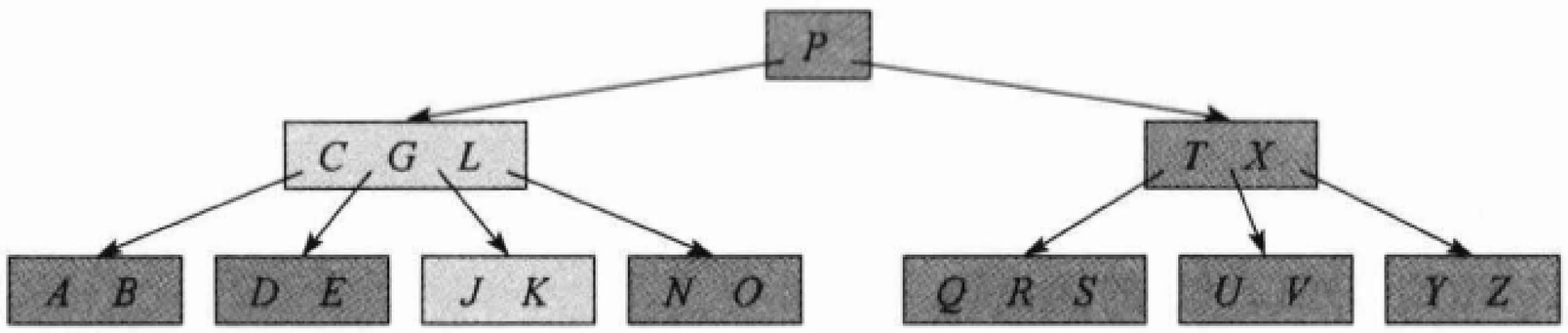
尽管这个过程看起来很复杂，但对一棵高度为 h 的B树，它只需要 $O(h)$ 次磁盘操作，因为在递归调用该过程之间，仅需 $O(1)$ 次对DISK-READ和DISK-WRITE的调用。所需CPU时间为 $O(th)=O(t\log_t n)$。



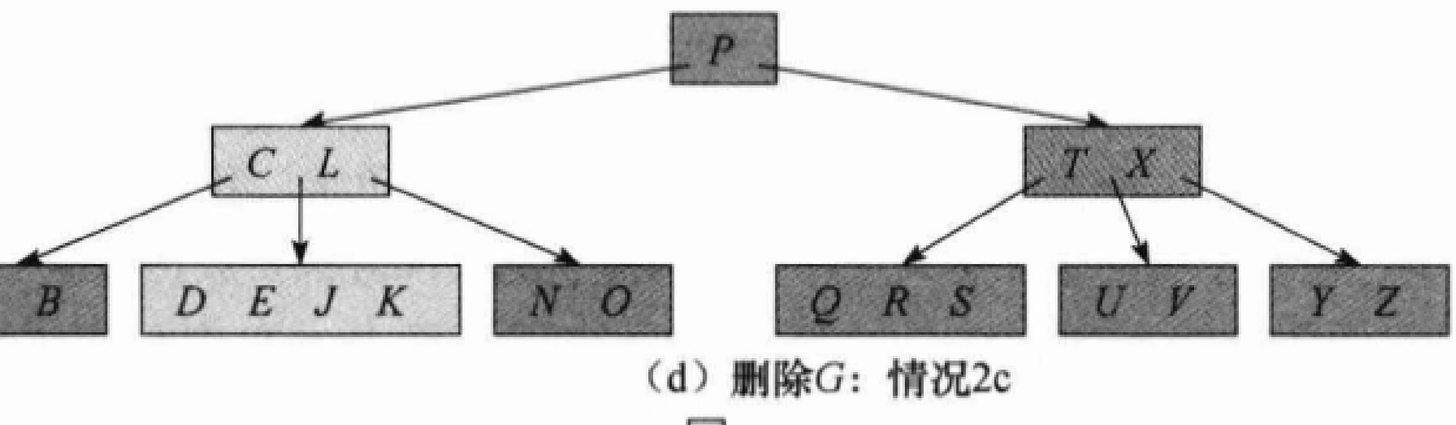
（a）初始树



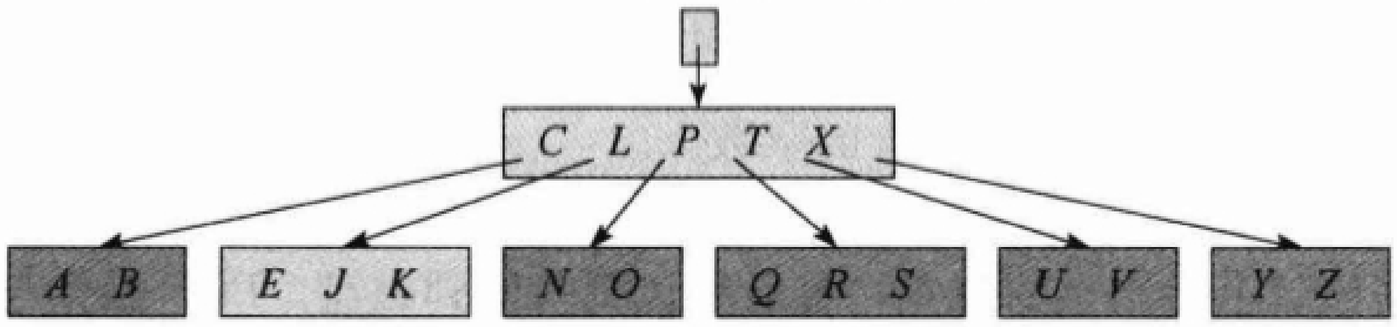
（b）删除F：情况1



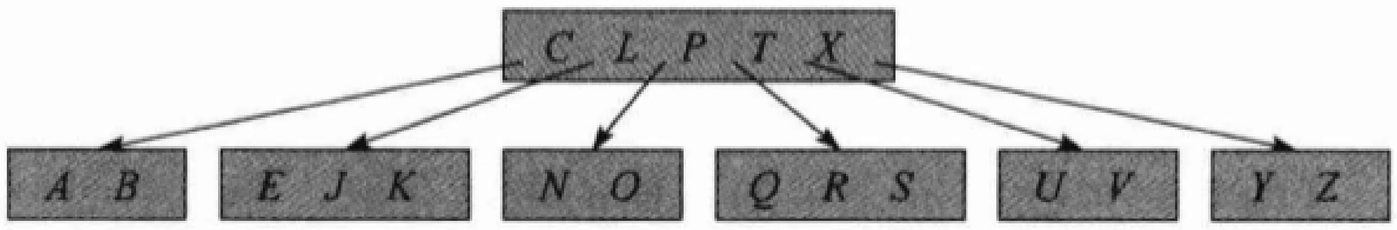
（c）删除M：情况2a



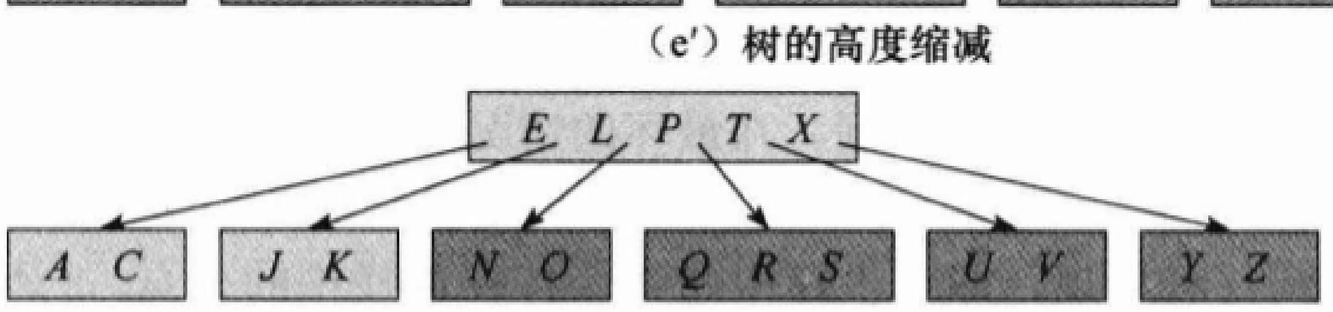
（d）删除G：情况2c



（e）删除D：情况3b



（e′）树的高度缩减



（f）删除B：情况3a

图 18-8　从一棵 B 树中删除关键字。这棵 B 树的最小度数 $t=3$，因此一个结点(非根)包含的关键字个数不能少于两个。被修改了的结点都以浅阴影标记。(a)图 18-7e 中的 B 树。(b)删除 F。这是情况 1：从一个叶结点中进行的简单删除。(c)删除 M。这是情况 2a：M 的前驱 L 提升并占据 M 的位置。(d)删除 G。这是情况 2c：G 下降以构成结点 $DEGJK$，然后从这个叶结点中删除 G(情况 1)。(e)删除 D。这是情况 3b：递归不能降至结点 CL，因为它仅有两个关键字，所以将 P 下降并与 CL 和 TX 合并以构成 $CLPTX$；然后将 D 从这个叶结点中删除(情况 1)。(e′)在(e)之后，根结点被删除，树的高度减小 1。(f)删除 B。这是情况 3a：移动 C 以填补 B 的位置，移动 E 以填补 C 的位置

500
~
501