第 13 章
Introduction to Algorithms, Third Edition

红 黑 树

第 12 章介绍了一棵高度为 h 的二叉搜索树，它可以支持任何一种基本动态集合操作，如 SEARCH、PREDECESSOR、SUCCESSOR、MINIMUM、MAXIMUM、INSERT 和 DELETE 等，其时间复杂度均为 $O(h)$。因此，如果搜索树的高度较低时，这些集合操作会执行得较快。然而，如果树的高度较高时，这些集合操作可能并不比在链表上执行得快。**红黑树**(red-black tree)是许多“平衡”搜索树中的一种，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$O(\lg n)$。

13.1 红黑树的性质

红黑树是一棵二叉搜索树，它在每个结点上增加了一个存储位来表示结点的**颜色**，可以是 RED 或 BLACK。通过对任何一条从根到叶子的简单路径上各个结点的颜色进行约束，红黑树确保没有一条路径会比其他路径长出 2 倍，因而是近似于**平衡的**。

树中每个结点包含 5 个属性：*color*、*key*、*left*、*right* 和 p。如果一个结点没有子结点或父结点，则该结点相应指针属性的值为 NIL。我们可以把这些 NIL 视为指向二叉搜索树的叶结点(外部结点)的指针，而把带关键字的结点视为树的内部结点。

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。

308

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。

图 13-1(a)显示了一个红黑树的例子。

为了便于处理红黑树代码中的边界条件，使用一个哨兵来代表 NIL(参见 10.2 节)。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，而其他属性 p、*left*、*right* 和 *key* 可以设为任意值。如图 13-1(b)所示，所有指向 NIL 的指针都用指向哨兵 $T.nil$ 的指针替换。

使用哨兵后，就可以将结点 x 的 NIL 孩子视为一个普通结点，其父结点为 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 的父结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，使用一个哨兵 $T.nil$ 来代表所有的 NIL：所有的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，尽管为了方便起见可以在程序中设定它们。

我们通常将注意力放在红黑树的内部结点上，因为它们存储了关键字的值。在本章的后面部分，所画的红黑树都忽略了叶结点，如图 13-1(c)所示。

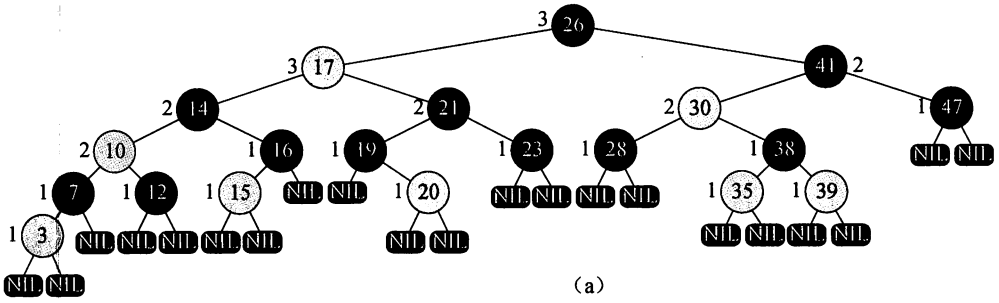
从某个结点 x 出发(不含该结点)到达一个叶结点的任意一条简单路径上的黑色结点个数称为该结点的**黑高**(black-height)，记为 $\mathrm{bh}(x)$。根据性质 5，黑高的概念是明确定义的，因为从该结点出发的所有下降到其叶结点的简单路径的黑结点个数都相同。于是定义红黑树的黑高为其根结点的黑高。

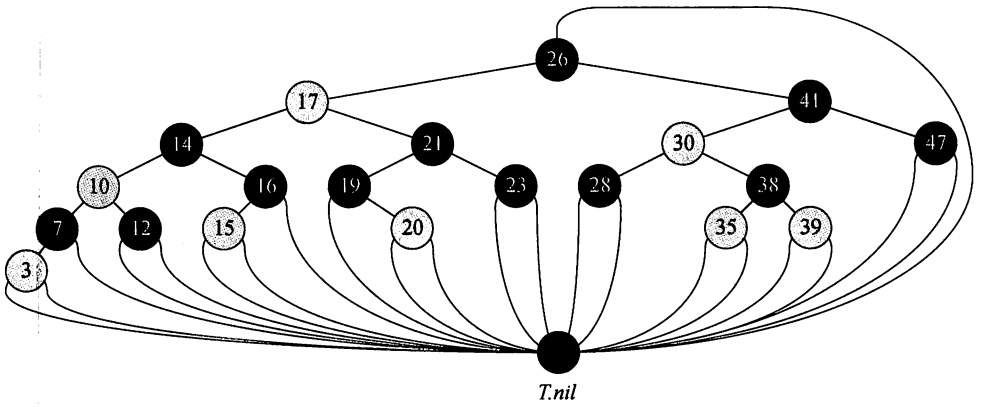
下面的引理说明了为什么红黑树是一种好的搜索树。

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

$2^{bh(x)}-1=2^0-1=0$ 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 $bh(x)$ 或 $bh(x)-1$，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 $2^{bh(x)-1}-1$ 个内部结点的结论。于是，以 x 为根的子树至少包含 $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ 个内部结点，因此得证。





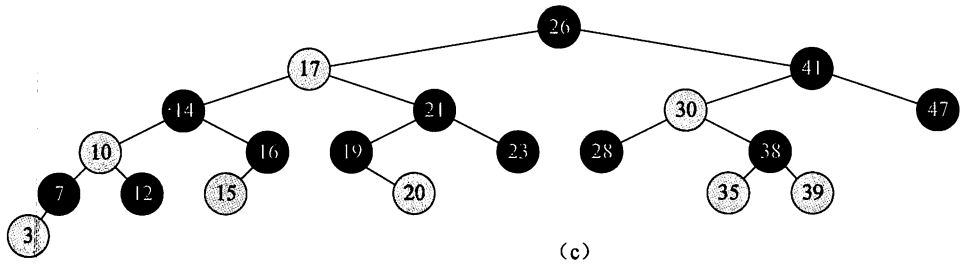


图 13-1　一棵红黑树，其中黑结点涂黑，红结点以浅阴影表示。在一棵红黑树内，每个结点或红或黑，红结点的两个子结点都是黑色，且从每个结点到其后代叶结点的每条简单路径上，都包含相同数目的黑结点。(a)每个标为 NIL 的叶结点都是黑的。每个非 NIL 结点都标上它的黑高；NIL 的黑高为 0。(b)同样的这棵红黑树，不是用一个个的 NIL 表示，而用一个总是黑色的哨兵 $T.nil$ 来代替，它的黑高也被省略。根结点的父结点也是这个哨兵。(c)同样的这棵红黑树，其叶结点与根结点的父结点全部被省略。本章的其余部分也采用这种画图方式

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

309
~
310

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。

练习

13.1-1 按照图 13-1(a)的方式，画出在关键字集合{1，2，…，15}上高度为 3 的完全二叉搜索树。以三种不同方式向图中加入 NIL 叶结点并对各结点着色，使所得的红黑树的黑高分别为 2、3 和 4。

13.1-2 对图 13-1 中的红黑树，画出对其调用 TREE-INSERT 操作插入关键字 36 后的结果。如果插入的结点被标为红色，所得的树是否还是一棵红黑树？如果该结点被标为黑色呢？

13.1-3 定义一棵**松弛红黑树**(relaxed red-black tree)为满足红黑性质 1、3、4 和 5 的二叉搜索树。换句话说，根结点可以是红色或是黑色。考虑一棵根结点为红色的松弛红黑树 T。如果将 T 的根结点标为黑色而其他都不变，那么所得到的是否还是一棵红黑树？

13.1-4 假设将一棵红黑树的每一个红结点"吸收"到它的黑色父结点中，使得红结点的子结点变成黑色父结点的子结点(忽略关键字的变化)。当一个黑结点的所有红色子结点都被吸收后，它可能的度为多少？所得的树的叶结点深度如何？

311

13.1-5 证明：在一棵红黑树中，从某结点 x 到其后代叶结点的所有简单路径中，最长的一条至多是最短一条的 2 倍。

13.1-6 在一棵黑高为 k 的红黑树中，内部结点最多可能有多少个？最少可能有多少个？

13.1-7 试描述一棵含有 n 个关键字的红黑树，使其红色内部结点个数与黑色内部结点个数的比值最大。这个比值是多少？该比值最小的树又是怎样呢？比值是多少？

13.2 旋转

搜索树操作 TREE-INSERT 和 TREE-DELETE 在含 n 个关键字的红黑树上，运行花费时间为 $O(\lg n)$。由于这两个操作对树做了修改，结果可能违反 13.1 节中列出的红黑性质。为了维护这些性质，必须要改变树中某些结点的颜色以及指针结构。

指针结构的修改是通过**旋转**(ratation)来完成的，这是一种能保持二叉搜索树性质的搜索树局部操作。图 13-2 中给出了两种旋转：左旋和右旋。当在某个结点 x 上做左旋时，假设它的右孩子为 y 而不是 $T.nil$；x 可以为其右孩子不是 $T.nil$ 结点的树内任意结点。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。

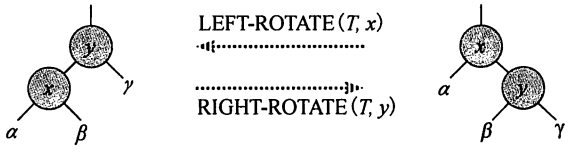


图 13-2 二叉搜索树上的旋转操作。操作 LEFT-ROTATE(T, x)通过改变常数数目的指针，可以将右边两个结点的结构转变成左边的结构。左边的结构可以使用相反的操作 RIGHT-ROTATE(T, y)来转变成右边的结构。字母 α、β 和 γ 代表任意的子树。旋转操作保持了二叉搜索树的性质：α 的关键字在 $x.key$ 之前，$x.key$ 在 β 的关键字之前，β 的关键字在 $y.key$ 之前，$y.key$ 在 γ 的关键字之前

在 LEFT-ROTATE 的伪代码中，假设 $x.right \neq T.nil$ 且根结点的父结点为 $T.nil$。

```
LEFT-ROTATE(T, x)
 1   y = x.right                  // set y
 2   x.right = y.left             // turn y's left subtree into x's right subtree
 3   if y.left ≠ T.nil
 4       y.left.p = x
 5   y.p = x.p                    // link x's parent to y
 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                   // put x on y's left
12   x.p = y
```

图 13-3 给出了一个 LEFT-ROTATE 操作修改二叉搜索树的例子。RIGHT-ROTATE 操作的代码是对称的。LEFT-ROTATE 和 RIGHT-ROTATE 都在 $O(1)$ 时间内运行完成。在旋转操作中只有指针改变，其他所有属性都保持不变。

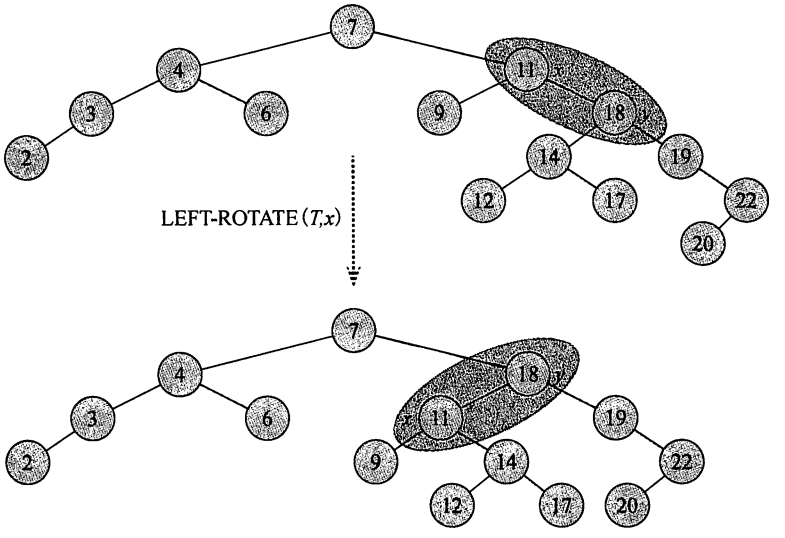


图 13-3　过程 LEFT-ROTATE(T, x)修改二叉搜索树的例子。输入的树和修改过的树进行中序遍历，产生相同的关键字值列表

练习

13.2-1　写出 RIGHT-ROTATE 的伪代码。

13.2-2　证明：在任何一棵有 n 个结点的二叉搜索树中，恰有 $n-1$ 种可能的旋转。

13.2-3　设在图 13-2 左边一棵树中，a、b 和 c 分别为子树 α、β 和 γ 中的任意结点。当结点 x 左旋之后，a、b 和 c 的深度会如何变化？

13.2-4　证明：任何一棵含 n 个结点的二叉搜索树可以通过 $O(n)$ 次旋转，转变为其他任何一棵含 n 个结点的二叉搜索树。（提示：先证明至多 $n-1$ 次右旋足以将树转变为一条右侧伸展的链。）

***13.2-5** 如果能够使用一系列的 RIGHT-ROTATE 调用把一个二叉搜索树 T_1 变为二叉搜索树 T_2，则称 T_1 可以**右转**(right-converted)成 T_2。试给出一个例子表示两棵树 T_1 和 T_2，其中 T_1 不能够右转成 T_2。然后，证明：如果 T_1 可以右转成 T_2，那么它可以通过 $O(n^2)$ 次 RIGHT-ROTATE 调用来实现右转。

312 ~ 314

13.3 插入

我们可以在 $O(\lg n)$ 时间内完成向一棵含 n 个结点的红黑树中插入一个新结点。为了做到这一点，利用 TREE-INSERT 过程(参见 12.3 节)的一个略作修改的版本来将结点 z 插入树 T 内，就好像 T 是一棵普通的二叉搜索树一样，然后将 z 着为红色。(练习 13.3-1 要求解释为什么选择将结点 z 着为红色，而不是黑色。)为保证红黑性质能继续保持，我们调用一个辅助程序 RB-INSERT-FIXUP 来对结点重新着色并旋转。调用 RB-INSERT(T, z)在红黑树 T 内插入结点 z，假设 z 的 key 属性已被事先赋值。

```
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)
```

过程 TREE-INSERT 和 RB-INSERT 之间有 4 处不同。第一，TREE-INSERT 内的所有 NIL 都被 $T.nil$ 代替。第二，RB-INSERT 的第 14～15 行置 $z.left$ 和 $z.right$ 为 $T.nil$，以保持合理的树结构。第三，在第 16 行将 z 着为红色。第四，因为将 z 着为红色可能违反其中的一条红黑性质，所以在 RB-INSERT 的第 17 行中调用 RB-INSERT-FIXUP(T, z)来保持红黑性质。

315

```
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
```

为了理解 RB-INSERT-FIXUP 过程如何工作，把代码分为三个主要的步骤。首先，要确定当结点 z 被插入并着为红色后，红黑性质中有哪些不能继续保持。其次，应分析第 1～15 行中 **while** 循环的总目标。最后，要分析 **while** 循环体中的三种情况[⊖]，看看它们是如何完成目标的。图 13-4 给出一个范例，显示在一棵红黑树上 RB-INSERT-FIXUP 如何操作。

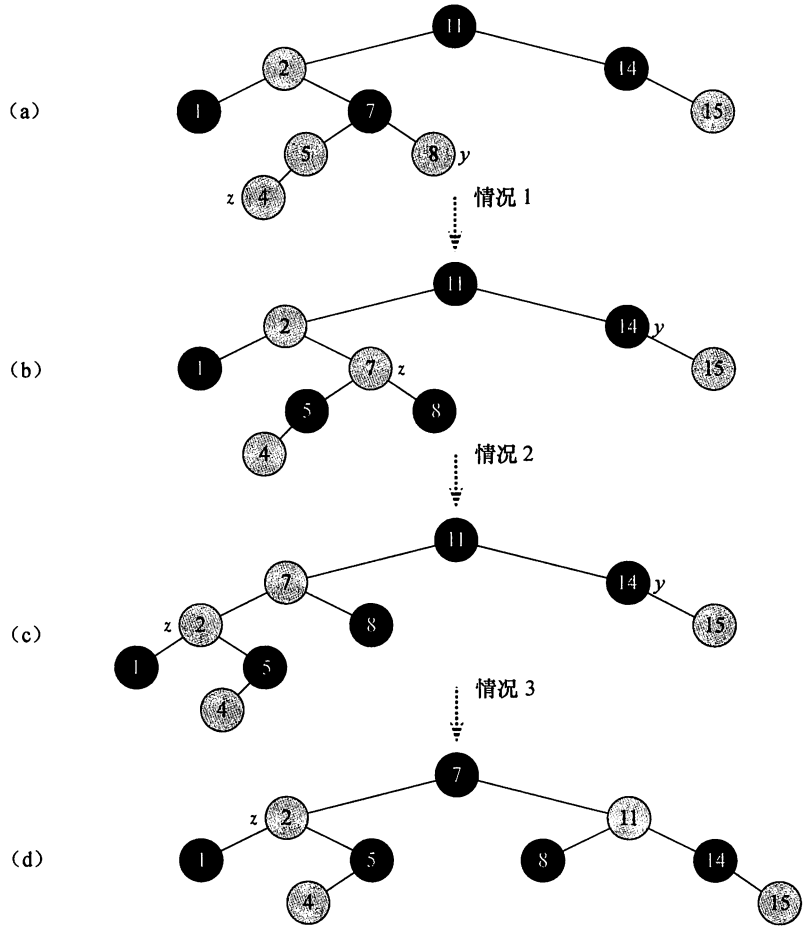


图 13-4　RB-INSERT-FIXUP 操作。(a)插入后的结点 z。由于 z 和它的父结点 $z.p$ 都是红色的，所以违反了性质 4。由于 z 的叔结点 y 是红色的，可以应用程序中的情况 1。结点被重新着色，并且指针 z 沿树上升，所得的树如(b)所示。再一次 z 及其父结点又都为红色，但 z 的叔结点 y 是黑色的。因为 z 是 $z.p$ 的右孩子，可以应用情况 2。在执行 1 次左旋之后，所得结果树见(c)。现在，z 是其父结点的左孩子，可以应用情况 3。重新着色并执行一次右旋后得(d)中的树，它是一棵合法的红黑树

⊖ 情况 2 可以转为情况 3，于是这两种情况就不是各自独立的了。

在调用 RB-INSERT-FIXUP 操作时，哪些红黑性质可能会被破坏呢？性质 1 和性质 3 继续成立，因为新插入的红结点的两个子结点都是哨兵 $T.nil$。性质 5，即从一个指定结点开始的每条简单路径上的黑结点的个数都是相等的，也会成立，因为结点 z 代替了(黑色)哨兵，并且结点 z 本身是有哨兵孩子的红结点。这样来看，仅可能被破坏的就是性质 2 和性质 4，即根结点需要为黑色以及一个红结点不能有红孩子。这两个性质可能被破坏是因为 z 被着为红色。如果 z 是根结点，则破坏了性质 2；如果 z 的父结点是红结点，则破坏了性质 4。图 13-4(a)显示在插入结点 z 之后性质 4 被破坏的情况。

316～317

第 1～15 行中的 **while** 循环在每次迭代的开头保持下列 3 个部分的不变式：

a. 结点 z 是红结点。

b. 如果 $z.p$ 是根结点，则 $z.p$ 是黑结点。

c. 如果有任何红黑性质被破坏，则至多只有一条被破坏，或是性质 2，或是性质 4。如果性质 2 被破坏，其原因为 z 是根结点且是红结点。如果性质 4 被破坏，其原因为 z 和 $z.p$ 都是红结点。

c 部分处理红黑性质的破坏，相比 a 部分和 b 部分来说，显得更是 RB-INSERT-FIXUP 保持红黑性质的中心内容，我们以此来理解代码中的各种情形。由于将注意力集中在结点 z 以及树中靠近它的结点上，所以有助于从 a 部分得知 z 为红结点。当在第 2、3、7、8、13 和 14 行中引用 $z.p.p$ 时，我们使用 b 部分来表明它的存在。

需要证明在循环的第一次迭代之前循环不变式为真，每次迭代都保持这个循环不变式成立，并且在循环终止时，这个循环不变式会给出一个有用的性质。

先从初始化和终止的不变式证明开始。然后，依据细致地考察循环体如何工作，来证明循环在每次迭代中都保持这个循环不变式。同时，还要说明循环的每次迭代会有两种可能的结果：或者指针 z 沿着树上移，或者执行某些旋转后循环终止。

初始化：在循环的第一次迭代之前，从一棵正常的红黑树开始，并新增一个红结点 z。

要证明当 RB-INSERT-FIXUP 被调用时，不变式的每个部分都成立。

a. 当调用 RB-INSERT-FIXUP 时，z 是新增的红结点。

b. 如果 $z.p$ 是根，那么 $z.p$ 开始是黑色的，且在调用 RB-INSERT-FIXUP 之前保持不变。

c. 注意到在调用 RB-INSERT-FIXUP 时，性质 1、性质 3 和性质 5 成立。

如果违反了性质 2，则红色根结点一定是新增结点 z，它是树中唯一的内部结点。因为 z 的父结点和两个子结点都是黑色的哨兵，没有违反性质 4。这样，对性质 2 的违反是整棵树中唯一违反红黑性质的地方。

318

如果违反了性质 4，则由于 z 的子结点是黑色哨兵，且该树在 z 加入之前没有其他性质的违反，所以违反必然是因为 z 和 $z.p$ 都是红色的。而且，没有其他红黑性质被违反。

终止：循环终止是因为 $z.p$ 是黑色的。(如果 z 是根结点，那么 $z.p$ 是黑色哨兵 $T.nil$。)这样，树在循环终止时没有违反性质 4。根据循环不变式，唯一可能不成立的是性质 2。第 16 行恢复这个性质，所以当 RB-INSERT-FIXUP 终止时，所有的红黑性质都成立。

保持：实际需要考虑 **while** 循环中的 6 种情况，而其中三种与另外三种是对称的。这取决于第 2 行中 z 的父结点 $z.p$ 是 z 的祖父结点 $z.p.p$ 的左孩子，还是右孩子。我们只给出 $z.p$ 是左孩子时的代码。根据循环不变式的 b 部分，如果 $z.p$ 是根结点，那么 $z.p$ 是黑色的，可知结点 $z.p.p$ 存在。因为只有在 $z.p$ 是红色时才进入一次循环迭代，所以 $z.p$ 不可能是根结点。因此，$z.p.p$ 存在。

情况 1 和情况 2、情况 3 的区别在于 z 父亲的兄弟结点(或称为“叔结点”)的颜色不同。第 3 行使 y 指向 z 的叔结点 $z.p.p.right$，在第 4 行测试 y 的颜色。如果 y 是红色的，那么执行情况 1。否则，控制转向情况 2 和情况 3 上。在所有三种情况中，z 的祖父结点 $z.p.p$ 是黑色的，因为它的

父结点 $z.p$ 是红色的，故性质 4 只在 z 和 $z.p$ 之间被破坏了。

情况 1：z 的叔结点 y 是红色的

图 13-5 显示了情况 1(第 5～8 行)的情形，这种情况在 $z.p$ 和 y 都是红色时发生。因为 $z.p.p$ 是黑色的，所以将 $z.p$ 和 y 都着为黑色，以此解决 z 和 $z.p$ 都是红色的问题，将 $z.p.p$ 着为红色以保持性质 5。然后，把 $z.p.p$ 作为新结点 z 来重复 **while** 循环。指针 z 在树中上移两层。

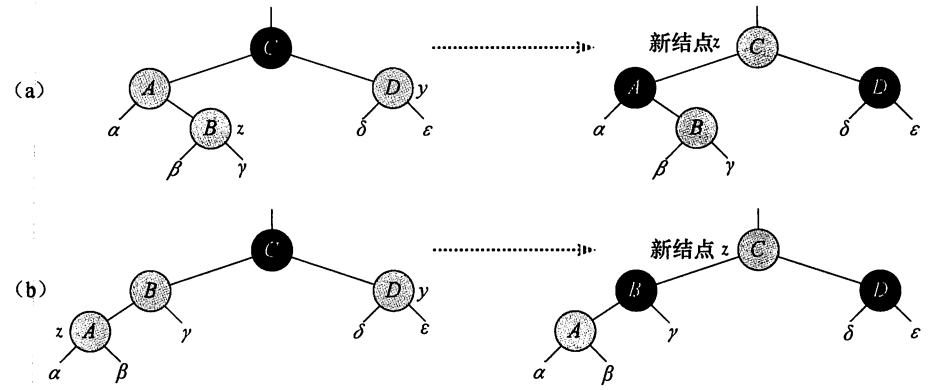


图 13-5　过程 RB-INSERT-FIXUP 中的情况 1。性质 4 被违反，因为 z 和它的父结点 $z.p$ 都是红色的。无论 z 是一个右孩子(图(a))还是一个左孩子(图(b))，都同样处理。每一棵子树 α、β、γ、δ 和 ε 都有一个黑色根结点，而且具有相同的黑高。情况 1 的代码改变了某些结点的颜色，但保持了性质 5：从一个结点向下到一个叶结点的所有简单路径都有相同数目的黑结点。**while** 循环将结点 z 的祖父 $z.p.p$ 作为新的 z 以继续迭代。现在性质 4 的破坏只可能发生在新的红色结点 z 和它的父结点之间，条件是如果父结点也为红色的

现在，证明情况 1 在下一次循环迭代的开头会保持这个循环不变式。用 z 表示当前迭代中的结点 z，用 $z'=z.p.p$ 表示在下一次迭代第 1 行测试时的结点 z。

a. 因为这次迭代把 $z.p.p$ 着为红色，结点 z' 在下次迭代的开始是红色的。

b. 在这次迭代中结点 $z'.p$ 是 $z.p.p.p$，且这个结点的颜色不会改变。如果它是根结点，则在此次迭代之前它是黑色的，且它在下次迭代的开头仍然是黑色的。

c. 我们已经证明情况 1 保持性质 5，而且它也不会引起性质 1 或性质 3 的破坏。

如果结点 z' 在下一次迭代开始时是根结点，则在这次迭代中情况 1 修正了唯一被破坏的性质 4。由于 z' 是红色的而且是根结点，所以性质 2 成为唯一被违反的性质，这是由 z' 导致的。

如果结点 z' 在下一次迭代开始时不是根结点，则情况 1 不会导致性质 2 的破坏。情况 1 修正了在这次迭代的开始唯一违反的性质 4。然后它把 z' 着为红色而 $z'.p$ 不变。如果 $z'.p$ 是黑色的，则没有违反性质 4。如果 $z'.p$ 是红色的，则把 z' 着为红色会在 z' 与 $z'.p$ 之间造成性质 4 的违反。

情况 2：z 的叔结点 y 是黑色的且 z 是一个右孩子

情况 3：z 的叔结点 y 是黑色的且 z 是一个左孩子

在情况 2 和情况 3 中，z 的叔结点 y 是黑色的。通过 z 是 $z.p$ 的右孩子还是左孩子来区别这两种情况。第 10～11 行构成了情况 2，它和情况 3 一起显示在图 13-6 中。在情况 2 中，结点 z 是它的父结点的右孩子。可以立即使用一个左旋来将此情形转变为情况 3(第 12～14 行)，此时结点 z 为左孩子。因为 z 和 $z.p$ 都是红色的，所以该旋转对结点的黑高和性质 5 都无影响。无论是直接进入情况 2，还是通过情况 3 进入情况 2，z 的叔结点 y 总是黑色的，因为否则就要执行情况 1。此外，结点 $z.p.p$ 存在，因为已经推断在执行第 2 行和第 3 行时该结点存在，且在第 10 行将 z 往上移一层，然后在第 11 行将 z 往下移一层之后，$z.p.p$ 的身份保持不变。在情况 3 中，改变某些结点的颜色并做一次右旋，以保持性质 5。这样，由于在一行中不再有两个红色结点，

319
～
320

所有的处理到此完毕。因为此时 $z.p$ 是黑色的，所以无需再执行一次 **while** 循环。

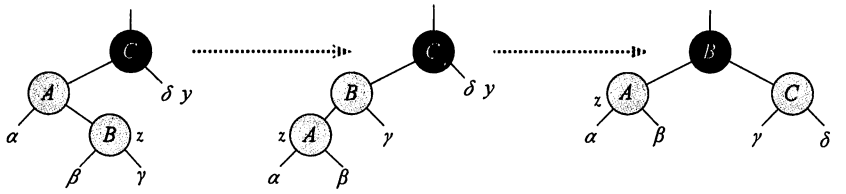


图 13-6 过程 RB-INSERT-FIXUP 中的情况 2 和情况 3。如同情况 1，由于 z 和它的父结点 $z.p$ 都是红色的，性质 4 在情况 2 或情况 3 中会被破坏。每一棵子树 α、β、γ 和 δ 都有一个黑色根结点(α、β 和 γ 是由性质 4 而来，δ 也有黑色根结点，因为否则将导致情况 1)，而且具有相同的黑高。通过左旋将情况 2 转变为情况 3，以保持性质 5：从一个结点向下到一个叶结点的所有简单路径都有相同数目的黑结点。情况 3 引起某些结点颜色的改变，以及一个同样为了保持性质 5 的右旋。然后 **while** 循环终止，因为性质 4 已经得到了满足：一行中不再有两个红色结点

现在来证明情况 2 和情况 3 保持了循环不变式。(正如已经讨论的，$z.p$ 在第 1 行中下一次测试会是黑色，循环体不会再次执行。)

a. 情况 2 让 z 指向红色的 $z.p$。在情况 2 和情况 3 中 z 或 z 的颜色都不再改变。

b. 情况 3 把 $z.p$ 着成黑色，使得如果 $z.p$ 在下一次迭代开始时是根结点，则它是黑色的。

c. 如同情况 1，性质 1、性质 3 和性质 5 在情况 2 与情况 3 中得以保持。

由于结点 z 在情况 2 和情况 3 中都不是根结点，所以性质 2 没有被破坏。情况 2 和情况 3 不会引起性质 2 的违反，因为唯一着为红色的结点在情况 3 中通过旋转成为一个黑色结点的子结点。

情况 2 和情况 3 修正了对性质 4 的违反，也不会引起对其他红黑性质的违反。

证明了循环的每一次迭代都会保持循环不变式之后，也就证明了 RB-INSERT-FIXUP 能够
321 正确地保持红黑性质。

分析

RB-INSERT 的运行时间怎样呢？由于一棵有 n 个结点的红黑树的高度为 $O(\lg n)$，因此 RB-INSERT 的第 1～16 行要花费 $O(\lg n)$时间。在 RB-INSERT-FIXUP 中，仅当情况 1 发生，然后指针 z 沿着树上升 2 层，**while** 循环才会重复执行。所以 **while** 循环可能被执行的总次数为 $O(\lg n)$。因此，RB-INSERT 总共花费 $O(\lg n)$时间。此外，该程序所做的旋转从不超过 2 次，因为只要执行了情况 2 或情况 3，**while** 循环就结束了。

练习

13.3-1 在 RB-INSERT 的第 16 行，将新插入的结点 z 着为红色。注意到，如果将 z 着为黑色，则红黑树的性质 4 就不会被破坏。那么为什么不选择将 z 着为黑色呢？

13.3-2 将关键字 41、38、31、12、19、8 连续地插入一棵初始为空的红黑树之后，试画出该结果树。

13.3-3 假设图 13-5 和图 13-6 中子树 α、β、γ、δ 和 ε 的黑高都是 k。给每张图中的每个结点标上黑高，以验证图中所示的转换能保持性质 5。

13.3-4 Teach 教授担心 RB-INSERT-FIXUP 可能将 $T.nil.color$ 设为 RED，这时，当 z 为根时，第 1 行的测试就不会让循环终止。通过讨论 RB-INSERT-FIXUP 永远不会将 $T.nil.color$ 设置为 RED，来说明这位教授的担心是没有必要的。

13.3-5 考虑一棵用 RB-INSERT 插入 n 个结点而成的红黑树。证明：如果 $n>1$，则该树至少有一个红结点。

13.3-6 说明如果红黑树的表示中不提供父指针，应当如何有效地实现 RB-INSERT。 322

13.4 删除

与 n 个结点的红黑树上的其他基本操作一样，删除一个结点要花费 $O(\lg n)$ 时间。与插入操作相比，删除操作要稍微复杂些。

从一棵红黑树中删除结点的过程是基于 TREE-DELETE 过程(见 12.3 节)而来的。首先，需要特别设计一个供 TREE-DELETE 调用的子过程 TRANSPLANT，并将其应用到红黑树上：

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

过程 RB-TRANSPLANT 与 TRANSPLANT 有两点不同。首先，第 1 行引用哨兵 $T.nil$ 而不是 NIL。其次，第 6 行对 $v.p$ 的赋值是无条件执行：即使 v 指向哨兵，也要对 $v.p$ 赋值。实际上，当 $v=T.nil$ 时，也能给 $v.p$ 赋值。

过程 RB-DELETE 与 TREE-DELETE 类似，只是多了几行伪代码。多出的几行代码记录结点 y 的踪迹，y 有可能导致红黑性质的破坏。当想要删除结点 z，且此时 z 的子结点少于 2 个时，z 从树中删除，并让 y 成为 z。当 z 有两个子结点时，y 应该是 z 的后继，并且 y 将移至树中的 z 位置。在结点被移除或者在树中移动之前，必须记住 y 的颜色，并且记录结点 x 的踪迹，将 x 移至树中 y 的原来位置，因为结点 x 也可能引起红黑性质的破坏。删除结点 z 之后，RB-DELETE 调用一个辅助过程 RB-DELETE-FIXUP，该过程通过改变颜色和执行旋转来恢复红黑性质。 323

```
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)
```

虽然 RB-DELETE 包含的伪代码行数几乎是 TREE-DELETE 的 2 倍，但这两个过程具有相同的基本结构。在 RB-DELETE 中能够找到 TREE-DELETE 的每一行语句(其中 NIL 被替换成了 $T.nil$，而调用 TRANSPLANT 换成了调用 RB-TRANSPLANT)，其执行的条件相同。

下面是两个过程之间的其他区别：

- 始终维持结点 y 为从树中删除的结点或者移至树内的结点。当 z 的子结点少于 2 个时，第 1 行将 y 指向 z，并因此要移除。当 z 有两个子结点时，第 9 行将 y 指向 z 的后继，这与 TREE-DELETE 相同，y 将移至树中 z 的位置。
- 由于结点 y 的颜色可能改变，变量 *y-original-color* 存储了发生改变前的 y 颜色。第 2 行和第 10 行在给 y 赋值之后，立即设置该变量。当 z 有两个子结点时，则 $y \neq z$ 且结点 y 移至红黑树中结点 z 的原始位置；第 20 行给 y 赋予和 z 一样的颜色。我们需要保存 y 的
324 原始颜色，以在 RB-DELETE 结束时测试它；如果它是黑色的，那么删除或移动 y 会引起红黑性质的破坏。
- 正如前面讨论过的，我们保存结点 x 的踪迹，使它移至结点 y 的原始位置上。第 4、7 和 11 行的赋值语句令 x 或指向 y 的唯一子结点或指向哨兵 $T.nil$(如果 y 没有子结点)。(回忆一下 12.3 节 y 没有左孩子的情形。)
- 因为结点 x 移动到结点 y 的原始位置，属性 $x.p$ 总是被设置指向树中 y 父结点的原始位置，甚至当 x 是哨兵 $T.nil$ 时也是这样。除非 z 是 y 的原始父结点(该情况只在 z 有两个孩子且它的后继 y 是 z 的右孩子时发生)，否则对 $x.p$ 的赋值在 RB-TRANSPLANT 的第 6 行。(注意到，在第 5、8 或 14 行调用 RB-TRANSPLANT 时，传递的第 2 个参数与 x 相同。)然而，当 y 的原父结点是 z 时，我们并不想让 $x.p$ 指向 y 的原始父结点，因为要在树中删除该结点。由于结点 y 将在树中向上移动占据 z 的位置，第 13 行将 $x.p$ 设置为 y，使得 $x.p$ 指向 y 父结点的原始位置，甚至当 $x = T.nil$ 时也是这样。
- 最后，如果结点 y 是黑色，就有可能已经引入了一个或多个红黑性质被破坏的情况，所以在第 22 行调用 RB-DELETE-FIXUP 来恢复红黑性质。如果 y 是红色，当 y 被删除或移动时，红黑性质仍然保持，原因如下：
  1. 树中的黑高没有变化。
  2. 不存在两个相邻的红结点。因为 y 在树中占据了 z 的位置，再考虑到 z 的颜色，树中 y 的新位置不可能有两个相邻的红结点。另外，如果 y 不是 z 的右孩子，则 y 的原右孩子 x 代替 y。如果 y 是红色，则 x 一定是黑色，因此用 x 替代 y 不可能使两个红结点相邻。
  3. 如果 y 是红色，就不可能是根结点，所以根结点仍旧是黑色。

如果结点 y 是黑色的，则会产生三个问题，可以通过调用 RB-DELETE-FIXUP 进行补救。第一，如果 y 是原来的根结点，而 y 的一个红色的孩子成为新的根结点，这就违反了性质 2。第二，如果 x 和 $x.p$ 是红色的，则违反了性质 4。第三，在树中移动 y 将导致先前包含 y 的任何简单路径上黑结点个数少 1。因此，y 的任何祖先都不满足性质 5。改正这一问题的办法是将现在占有 y 原来位置的结点 x 视为还有一重额外的黑色。也就是说，如果将任意包含结点 x 的简单路径上黑结点个数加 1，则在这种假设下，性质 5 成立。当将黑结点 y 删除或移动时，将其黑色
25 “下推”给结点 x。现在问题变为结点 x 可能既不是红色，又不是黑色，从而违反了性质 1。现在的结点 x 是双重黑色或者红黑色，这就分别给包含 x 的简单路径上黑结点数贡献了 2 或 1。x 的 *color* 属性仍然是 RED(如果 x 是红黑色的)或者 BLACK(如果 x 是双重黑色的)。换句话说，结点额外的黑色是针对 x 结点的，而不是反映在它的 *color* 属性上的。

现在我们来看看过程 RB-DELETE-FIXUP 是如何恢复搜索树的红黑性质的。

```
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
```

过程 RB-DELETE-FIXUP 恢复性质 1、性质 2 和性质 4。练习 13.4-1 和 13.4-2 要求读者说明这个过程是如何恢复性性质 2 和性质 4 的，因此，本节的其余部分将专注于性质 1。第 1～22 行中 **while** 循环的目标是将额外的黑色沿树上移，直到：

1. x 指向红黑结点，此时在第 23 行中，将 x 着为(单个)黑色。

2. x 指向根结点，此时可以简单地“移除”额外的黑色。

3. 执行适当的旋转和重新着色，退出循环。

在 **while** 循环中，x 总是指向一个具有双重黑色的非根结点。在第 2 行中要判断 x 是其父结点 $x.p$ 的左孩子还是右孩子。(已经给出了 x 为左孩子时的代码；x 为右孩子的第 22 行的代码是对称的。)保持指针 w 指向 x 的兄弟。由于结点 x 是双重黑色的，故 w 不可能是 $T.nil$，因为否则，从 $x.p$ 至(单黑色)叶子 w 的简单路径上的黑结点个数就会小于从 $x.p$ 到 x 的简单路径上的黑结点数。

图 13-7 给出了代码中的 4 种情况[⊖]。在具体研究每一种情况之前，先看看如何证实每种情况中的变换保持性质 5。关键思想是在每种情况中，从子树的根(包括根)到每棵子树 α，β，…，ζ 之间的黑结点个数(包括 x 的额外黑色)并不被变换改变。因此，如果性质 5 在变换之前成立，那么变换之后也仍然成立。举例说明，图 13-7(a)说明了情况 1，在变换前后，根结点至子树 α 或 β 之间的黑结点数都是 3。(再次记住，结点 x 增加了额外一重黑色。)类似地，在变换前后根结点至子树 γ、δ、ε 和 ζ 中的任何一个之间的黑结点数都是 2。在图 13-7(b)中，计数时还要包括所示子树的根结点的 *color* 属性的值 c，它或是 RED 或是 BLACK。如果定义 count(RED)＝0 以及 count(BLACK)＝1，那么变换前后根结点至 α 的黑结点数都为 2＋count(c)。在此情况下，变换

⊖ 参见过程 RB-INSERT-FIXUP，RB-DELETE-FIXUP 中的 4 种情况并不是完全独立的。

之后新结点 x 具有 *color* 属性值 c，但是这个结点的颜色是红黑(如果 c=RED)或者双重黑色的(如果 c=BLACK)。其他情况可以类似地加以验证(见练习 13.4-5)。

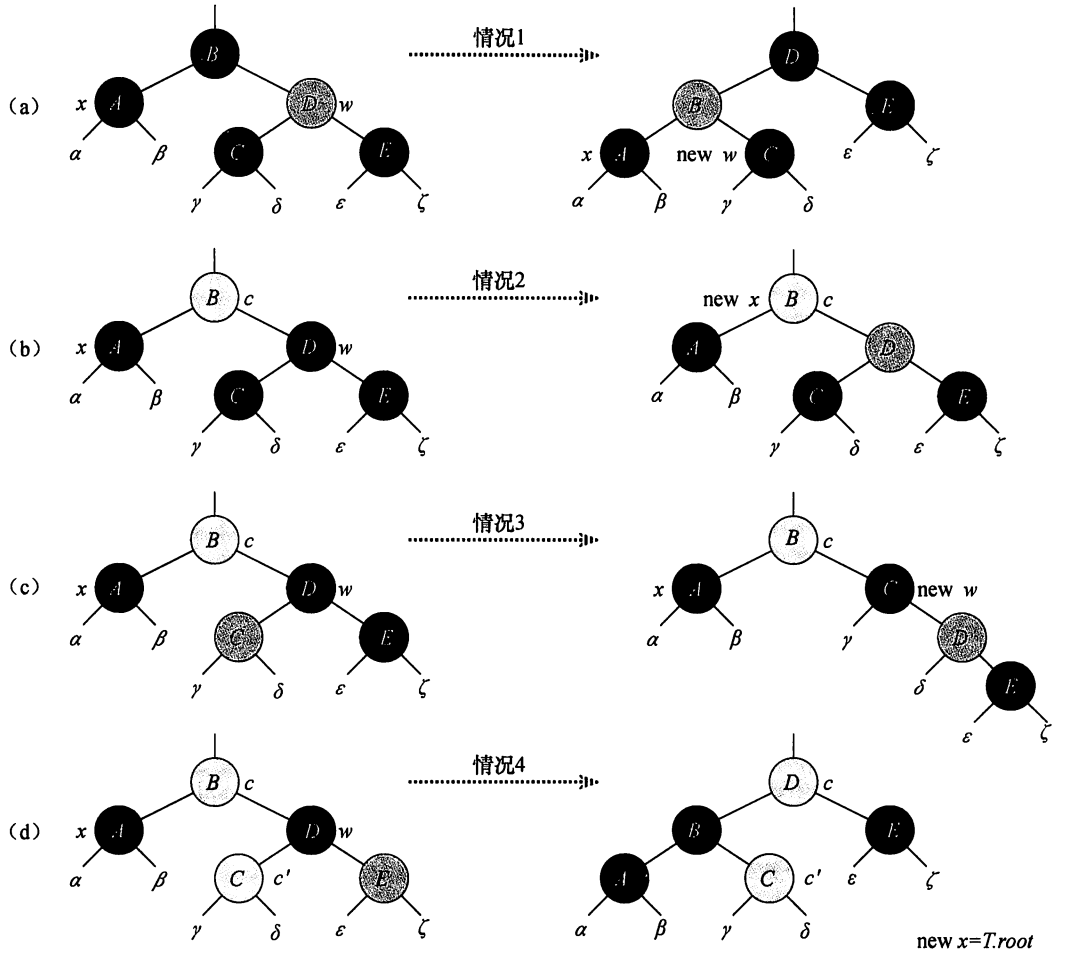


图 13-7 过程 RB-DELETE-FIXUP 中 **while** 循环的各种情况。加黑的结点 *color* 属性为 BLACK，深阴影的结点 *color* 属性为 RED，浅阴影的结点 *color* 属性用 c 和 c' 表示，它既可为 RED 也可为 BLACK。字母 α，β，…，ζ 代表任意的子树。在每种情况中，通过改变某些结点的颜色及/或进行一次旋转，可以将左边的结构转化为右边的结构。x 指向的任何结点都具有额外的一重黑色而成为双重黑色或红黑色。只有情况 2 引起循环重复。(a)通过交换结点 B 和 D 的颜色以及执行一次左旋，可将情况 1 转化为情况 2、3 或 4。(b)在情况 2 中，在将结点 D 着为红色，并将 x 设为指向结点 B 后，由指针 x 所表示的额外黑色沿树上升。如果通过情况 1 进入情况 2，则 **while** 循环结束，因为新的结点 x 是红黑的，因此其 *color* 属性 c 是 RED。(c)通过交换结点 C 和 D 的颜色并执行一次右旋，可以将情况 3 转换成情况 4。(d)在情况 4 中，通过改变某些结点的颜色并执行一次左旋(不违反红黑性质)，可以将由 x 表示的额外黑色去掉，然后循环终止

情况 1：x 的兄弟结点 w 是红色的

情况 1(见 RB-DELETE-FIXUP 的第 5～8 行和图 13-7(a))发生在结点 x 的兄弟结点 w 为红色时。因为 w 必须有黑色子结点，所以可以改变 w 和 $x.p$ 的颜色，然后对 $x.p$ 做一次左旋而不违反红黑树的任何性质。现在，x 的新兄弟结点是旋转之前 w 的某个子结点，其颜色为黑色。这样，就将情况 1 转换为情况 2、3 或 4 处理。

当结点 w 为黑色时，属于情况 2、3 和 4；这些情况是由 w 的子结点的颜色来区分的。

情况 2：x 的兄弟结点 w 是黑色的，而且 w 的两个子结点都是黑色的

在情况 2(见 RB-DELETE-FIXUP 的第 10～11 行和图 13-7(b))中，w 的两个子结点都是黑色的。因为 w 也是黑色的，所以从 x 和 w 上去掉一重黑色，使得 x 只有一重黑色而 w 为红色。为了补偿从 x 和 w 中去掉的一重黑色，在原来是红色或黑色的 $x.p$ 上新增一重额外的黑色。通过将 $x.p$ 作为新结点 x 来重复 **while** 循环。注意到，如果通过情况 1 进入到情况 2，则新结点 x 是红黑色的，因为原来的 $x.p$ 是红色的。因此，新结点 x 的 *color* 属性值 c 为 RED，并且在测试循环条件后循环终止。然后，在第 23 行中将新结点 x 着为(单一)黑色。

情况 3：x 的兄弟结点 w 是黑色的，w 的左孩子是红色的，w 的右孩子是黑色的

情况 3(见第 13～16 行和图 13-7(c))发生在 w 为黑色且其左孩子为红色、右孩子为黑色时。可以交换 w 和其左孩子 $w.left$ 的颜色，然后对 w 进行右旋而不违反红黑树的任何性质。现在 x 的新兄弟结点 w 是一个有红色右孩子的黑色结点，这样我们就将情况 3 转换成了情况 4。

情况 4：x 的兄弟结点 w 是黑色的，且 w 的右孩子是红色的

情况 4(见第 17～21 行和图 13-7(d))发生在结点 x 的兄弟结点 w 为黑色且 w 的右孩子为红色时。通过进行某些颜色修改并对 $x.p$ 做一次左旋，可以去掉 x 的额外黑色，从而使它变为单重黑色，而且不破坏红黑树的任何性质。将 x 设置为根后，当 **while** 循环测试其循环条件时，循环终止。

分析

RB-DELETE 的运行时间怎样呢？因为含 n 个结点的红黑树的高度为 $O(\lg n)$，不调用 RB-DELETE-FIXUP 时该过程的总时间代价为 $O(\lg n)$。在 RB-DELETE-FIXUP 中，情况 1、3 和 4 在各执行常数次数的颜色改变和至多 3 次旋转后便终止。情况 2 是 **while** 循环可以重复执行的唯一情况，然后指针 x 沿树上升至多 $O(\lg n)$ 次，且不执行任何旋转。所以，过程 RB-DELETE-FIXUP 要花费 $O(\lg n)$时间，做至多 3 次旋转，因此 RB-DELETE 运行的总时间为 $O(\lg n)$。

练习

13.4-1 在执行 RB-DELETE-FIXUP 之后，证明：树根一定是黑色的。

13.4-2 在 RB-DELETE 中，如果 x 和 $x.p$ 都是红色的，证明：可以通过调用 RB-DELETE-FIXUP(T，x)来恢复性质 4。

13.4-3 在练习 13.3-2 中，将关键字 41、38、31、12、19、8 连续插入一棵初始的空树中，从而得到一棵红黑树。请给出从该树中连续删除关键字 8、12、19、31、38、41 后的红黑树。

13.4-4 在 RB-DELETE-FIXUP 代码的哪些行中，可能会检查或修改哨兵 $T.nil$?

13.4-5 在图 13-7 的每种情况中，给出所示子树的根结点至每棵子树 α，β，…，ζ之间的黑结点个数，并验证它们在转换之后保持不变。当一个结点的 *color* 属性为 c 或 c'时，在计数中用记号 count(c)或 count(c')来表示。

13.4-6 Skelton 和 Baron 教授担心在 RB-DELETE-FIXUP 的情况 1 开始时，结点 $x.p$ 可能不是黑色的。如果这两位教授是对的，则第 5～6 行就是错的。证明：$x.p$ 在情况 1 开始时必是黑色的，从而说明这两位教授没有担心的必要。

13.4-7 假设用 RB-INSERT 将一个结点 x 插入一棵红黑树，紧接着又用 RB-DELETE 将它从树中删除。结果的红黑树与初始的红黑树是否一样？证明你的答案。

327～330

思考题

13-1 (持久动态集合) 有时在算法的执行过程中，我们会发现在更新一个动态集合时，需要维护其过去的版本。我们称这样的集合为**持久的**(persistent)。实现持久集合的一种方法是每

当该集合被修改时，就将其完整地复制下来，但是这种方法会降低一个程序的执行速度，而且占用过多的空间。有时候，我们可以做得更好些。

考虑一个有 INSERT、DELETE 和 SEARCH 操作的持久集合 S，我们使用如图 13-8(a)所示的二叉搜索树来实现。对集合的每一个版本都维护一个不同的根。为了将关键字 5 插入到集合中，创建一个具有关键字 5 的新结点。该结点成为具有关键字 7 的新结点的左孩子，因为我们不能更改具有关键字 7 的已存在结点。类似地，具有关键字 7 的新结点成为具有关键字 8 的新结点的左孩子，后者的右孩子为具有关键字 10 的已存在结点。关键字为 8 的新结点又成为关键字为 4 的新根结点 r' 的右孩子，而 r' 的左孩子是关键字为 3 的已存在结点。这样，我们只是复制了树的一部分，新树共享了原树的一些结点，如图 13-8(b)所示。

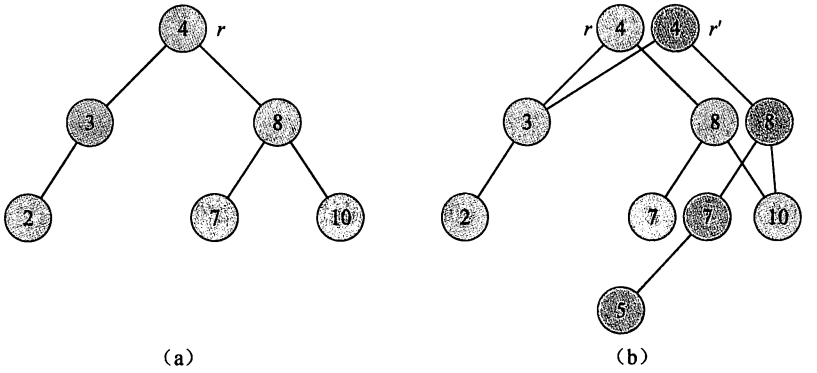


图 13-8 (a)包含关键字 2、3、4、7、8、10 的一棵二叉搜索树。(b)插入关键字 5 后得到的持久二叉搜索树。该集合的最新版本包括由根 r' 出发可到达的结点，而前一个版本包括由根 r 可到达的结点。深阴影的结点是插入关键字 5 时增加的

假设树中每个结点都有属性 *key*、*left* 和 *right*，但没有属性 *parent*。(参见练习 13.3-6。)

331

a. 对于一棵一般的持久二叉搜索树，为插入一个关键字 k 或删除一个结点 y，需要改变哪些结点。

b. 请写出一个过程 PERSISTENT-TREE-INSERT，使得在给定一棵持久树 T 和一个要插入的关键字 k 时，它返回将 k 插入 T 后得到的新的持久树 T'。

c. 如果持久二叉搜索树 T 的高度为 h，实现 PERSISTENT-TREE-INSERT 过程的时间和空间需求分别是多少？(空间需求与新分配的结点数成正比。)

d. 假设在每个结点中增加一个父结点属性。这种情况下，PERSISTENT-TREE-INSERT 需要做一些额外的复制工作。证明：PERSISTENT-TREE-INSERT 的时间需求和空间需求为 $\Omega(n)$，其中 n 为树中的结点个数。

e. 说明如何利用红黑树来保证每次插入或删除的最坏情况运行时间和空间为 $O(\lg n)$。

13-2 (**红黑树上的连接操作**) **连接**(join)操作取两个动态集合 S_1、S_2 和一个元素 x，使得对任何 $x_1 \in S_1$ 和 $x_2 \in S_2$，有 $x_1.key \leqslant x.key \leqslant x_2.key$。该操作返回一个集合 $S = S_1 \cup \{x\} \cup S_2$。在这个问题中，讨论如何在红黑树上实现连接操作。

a. 给定一棵红黑树 T，其黑高被存放在新属性 $T.bh$ 中。证明：在不需要树中结点的额外存储空间和不增加渐近运行时间的前提下，RB-INSERT 和 RB-DELETE 可以维护这个属性。并证明：当沿 T 下降时，可以对每个被访问的结点在 $O(1)$ 时间内确定其黑高。

要求实现操作 RB-JOIN(T_1，x，T_2)，它销毁 T_1 和 T_2 并返回一棵红黑树 $T = T_1 \cup \{x\} \cup T_2$。设 n 为 T_1 和 T_2 中的结点总数。

b. 假设 $T_1.bh \geqslant T_2.bh$。试描述一个 $O(\lg n)$时间的算法，使之能从黑高为 $T_2.bh$ 的结点中选出具有最大关键字的 T_1 中的黑结点 y。

c. 设 T_y 是以 y 为根结点的子树。试说明如何在不破坏二叉搜索树性质的前提下，在 $O(1)$时间内用 $T_y \cup \{x\} \cup T_2$ 来取代 T_y。

d. 要保持红黑性质 1、3 和 5，应将 x 着成什么颜色？试说明如何在 $O(\lg n)$时间内维护性质 2 和性质 4。

332

e. 论证使用(b)部分的假设是不失一般性的，并描述当 $T_1.bh \leqslant T_2.bh$ 时所出现的对称情况。

f. 证明：RB-JOIN 的运行时间是 $O(\lg n)$。

13-3 (AVL 树)　AVL 树是一种**高度平衡的**(height balanced)二叉搜索树：对每一个结点 x，x 的左子树与右子树的高度差至多为 1。要实现一棵 AVL 树，需要在每个结点内维护一个额外的属性：$x.h$ 为结点 x 的高度。与任何其他的二叉搜索树 T 一样，假设 $T.root$ 指向根结点。

a. 证明：一棵有 n 个结点的 AVL 树高度为 $O(\lg n)$。(提示：证明高度为 h 的 AVL 树至少有 F_h 个结点，其中 F_h 是斐波那契数列的第 h 个数。)

b. 要在一棵 AVL 树中插入一个结点，首先以二叉搜索树的顺序把该结点放在适当的位置上。此时，这棵树可能就不再是高度平衡的。具体来说，某些结点的左子树与右子树的高度差可能会到 2。请描述一个过程 BALANCE(x)，输入一棵以 x 为根的子树，其左子树与右子树都是高度平衡的，而且它们的高度差至多是 2，即 $|x.right.h - x.left.h| \leqslant 2$，并将这棵以 x 为根的子树转变为高度平衡的。(提示：使用旋转。)

c. 利用(b)来描述一个递归过程 AVL-INSERT(x，z)，该操作输入一个 AVL 树中的结点 x 以及一个新创建的结点 z(其关键字已经填入)，然后将 z 添加到以 x 为根的子树中，并保持 x 是一棵 AVL 树的根结点。和 12.3 节中的 TREE-INSERT 一样，假设 $z.key$ 已经被填入，且 $z.left = \text{NIL}$，$z.right = \text{NIL}$；再假设 $z.h = 0$。因此要把结点 z 插入到 AVL 树 T 中，需要调用 AVL-INSERT($T.root$，z)。

d. 证明：在一棵 n 个结点的 AVL 树上 AVL-INSERT 操作需花费 $O(\lg n)$时间，且执行 $O(1)$次旋转。

13-4 (treap 树)　如果将一个含 n 个元素的集合插入到一棵二叉搜索树中，所得到的树可能会相当不平衡，从而导致查找时间很长。然而从 12.4 节可知，随机构造二叉搜索树是趋向于平衡的。因此，一般来说，要为一组固定的元素建立一棵平衡树，可以采用的一种策略就是先随机排列这些元素，然后按照排列的顺序将它们插入到树中。

如果没法同时得到所有的元素，应该怎样处理呢？如果一次收到一个元素，是否仍然能用它们来随机建立一棵二叉搜索树？

我们将通过考察一个数据结构来正面回答这个问题。一棵 treap 树是一棵更改了结点排序方式的二叉搜索树。图 13-9 显示了一个例子。通常，树内的每个结点 x 都有一个关键字值 $x.key$。另外，还要为每个结点指定 $x.priority$，它是一个独立选取的随机数。假设所有的优先级都是不同的，而且所有的关键字也是不同的。treap 树的结点被排列成让关键字遵循二叉搜索树的性质，且优先级遵循最小堆顺序性质：

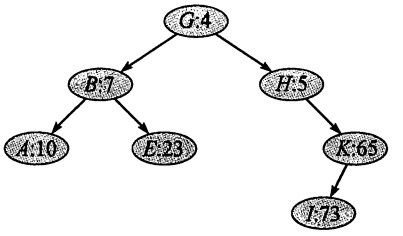


图 13-9　一棵 treap 树。每个结点 x 都用 $x.key$：$x.priority$ 来标记。例如，根结点的关键字是 G，优先级为 4

- 如果 v 是 u 的左孩子，则 $v.key < u.key$。
- 如果 v 是 u 的右孩子，则 $v.key > u.key$。

- 如果 v 是 u 的孩子，则 $v.\ priority > u.\ priority$。

(这两个性质的结合就是这种树被称为“treap”树的原因：它同时具有二叉搜索树和堆的特征。)

用以下方式考虑 treap 树是会有帮助的。假设将已有相应关键字的结点 x_1，x_2，…，x_n 插入到一棵 treap 树内。得到的 treap 树是通过将这些结点以它们的优先级(随机选取的)顺序插入一棵正常的二叉搜索树形成的，即 $x_i.\ priority < x_j.\ priority$ 表示 x_i 在 x_j 之前被插入。

a. 证明：给定一个已有相应关键字和优先级(互异)的结点 x_1，x_2，…，x_n 组成的集合，存在唯一的一棵 treap 树与这些结点相关联。

b. 证明：treap 树的期望高度是 $\Theta(\lg n)$，因此在 treap 内查找一个值所花的时间为 $\Theta(\lg n)$。让我们看看如何将一个新的结点插入到一个已存在的 treap 树中。要做的第一件事就是将一个随机的优先级赋予这个新结点。然后调用称为 TREAT-INSERT 的插入算法，其操作如图 13-10所示。

333
~
334

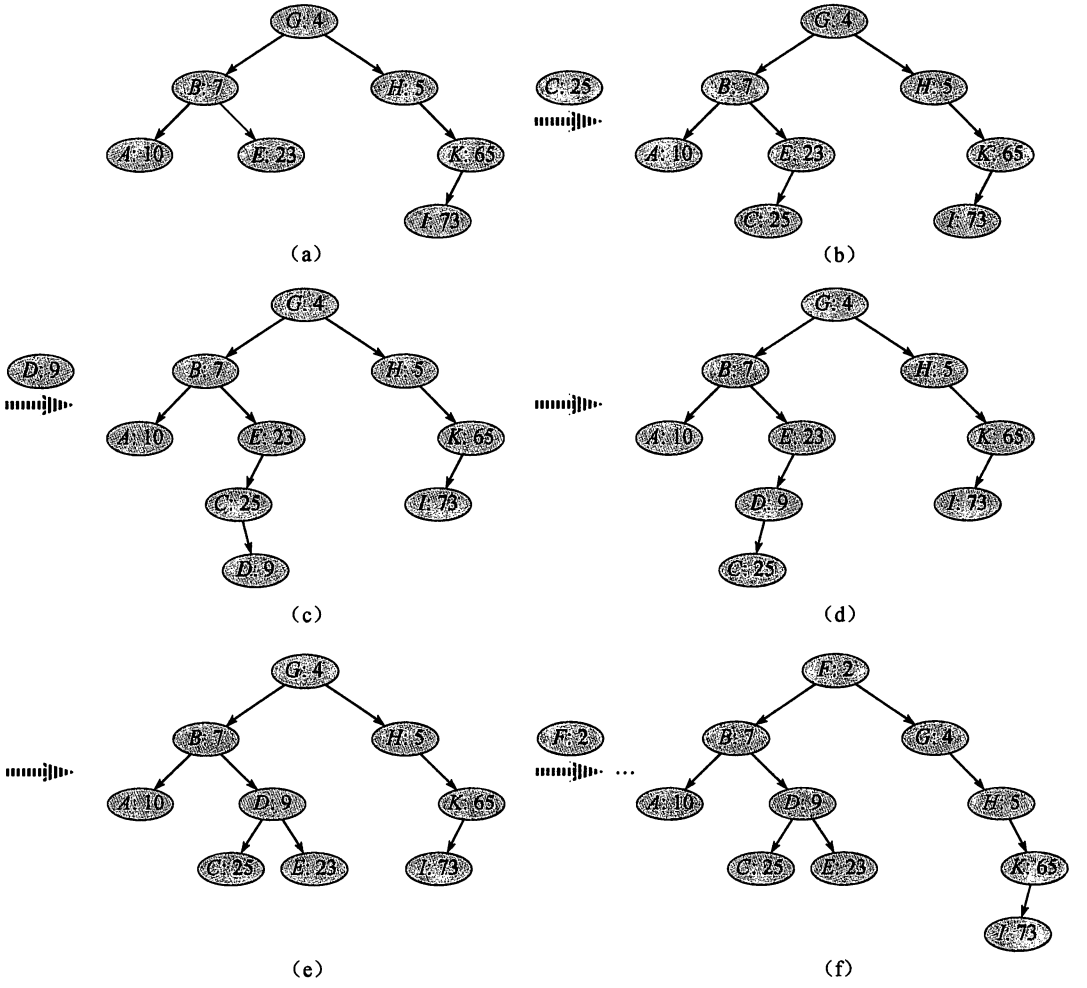


图 13-10 TREAP-INSERT 操作。(a)在插入之前的原 treap 树。(b)插入一个关键字为 C、优先级为 25 的结点之后的 treap 树。(c)～(d)插入一个关键字为 D、优先级为 9 的结点时的中间阶段。(e)在(c)和(d)的插入完成后的 treap 树。(f)在插入一个关键字为 F、优先级为 2 的结点后的 treap 树

335

c. 解释 TREAP-INSERT 是如何工作的。说明其思想并给出伪代码。(提示：执行通常的

二叉搜索树插入过程，然后做旋转来恢复最小堆顺序的性质。）

d. 证明：TREAP-INSERT 的期望运行时间是 $\Theta(\lg n)$。

TREAP-INSERT 先执行一个查找，然后做一系列旋转。虽然这两种操作的期望运行时间相同，但它们的实际代价不同。查找操作从 treap 树中读取信息而不做修改。相反，旋转操作会改变 treap 树内的父结点和子结点的指针。在大部分的计算机上，读取操作要比写入操作快很多。所以我们希望 TREAP-INSERT 执行少量的旋转。后面将说明所执行旋转的期望次数有一个常数界。

为此，需要做一些定义，如图 13-11 所示。一棵二叉搜索树 T 的**左脊柱**(left spine)是从根结点到有最小关键字的结点的简单路径。换句话说，左脊柱是从根结点开始只包含左边缘的简单路径。对称地，T 的**右脊柱**(right spine)是从根结点开始只包含右边缘的简单路径。一条脊柱的长度是它包含的结点数目。

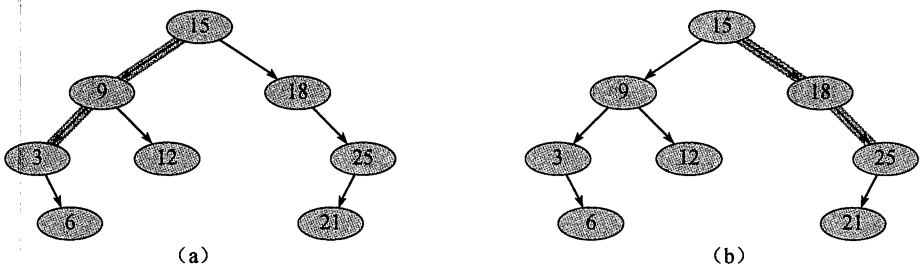


图 13-11　一棵二叉搜索树的脊柱。左脊柱在(a)中用阴影表示，右脊柱在(b)中用阴影表示

e. 考虑利用 TREAP-INSERT 插入结点 x 后的 treap T。设 C 为 x 左子树的右脊柱的长度，D 为 x 右子树的左脊柱的长度。证明：在插入 x 期间所执行的旋转的总次数等于 $C+D$。

现在来计算 C 和 D 的期望值。不失一般性，假设关键字为 1，2，…，n，因为只是将它们两两比较。

对 treap T 中的结点 x 和 y，其中 $y \neq x$，设 $k = x.key$ 以及 $i = y.key$。定义指示器随机变量

$$X_{ik} = \mathrm{I}\{y \text{ 在 } x \text{ 的左子树的右脊柱中}\}$$

f. 证明：$X_{ik}=1$ 当且仅当 $y.priority > x.priority$，$y.key < x.key$ 成立，且对于每个满足 $y.key < z.key < x.key$ 的 z，有 $y.priority < z.priority$。

g. 证明：

$$\Pr\{X_{ik}=1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}$$

h. 证明：

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}$$

i. 利用对称性证明：

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1}$$

j. 得出如下结论：当在一棵 treap 树中插入一个结点时，执行旋转的期望次数小于 2。

本章注记

使搜索树平衡的想法源自 Adel'son-Vel'skiĭ 和 Landis[2]，他们在 1962 年提出了一类称为“AVL 树”的平衡搜索树，如思考题 13-3 所述。另外一类称为“2-3 树”的搜索树是由

J. E. Hopcroft 在 1970 年提出的(未发表)。2-3 树是通过操纵结点的度数来维持平衡的。Bayer 和 McCreight[35]提出了一种 2-3 树的推广，称为 B 树，有关内容将在第 18 章中介绍。

红黑树是由 Bayer[34]以“对称的二叉 B 树”的名字发明的。Guibas 和 Sedgewick[155]仔细研究了它们的性质，并引入了红/黑着色的有关约定。Andersson[15]提出了一种代码更简单些的红
337 黑树变种。Weiss[351]把这种变种称为 AA 树。AA 树和红黑树类似，只是左边的孩子永远不能为红色。

思考题 13-4 中的 treap 树是由 Seidel 和 Aragon[309]提出的。它们是 LEDA[253]内字典的默认实现，LEDA 是一组精心实现的数据结构和算法。

平衡二叉树还有很多其他的变种，包括带权平衡树[264]、k 近邻树[245]，以及替罪羊树[127]。或许其中最有趣的要数 Sleator 和 Tarjan[320]提出的“伸展树”，它可以“自我调整”。(参见 Tarjan[330]，该文给出了有关伸展树的详细描述。)伸展树不需要明确的平衡条件(如颜色)来维持平衡。替代的是，每次存取时“伸展操作”(涉及旋转)在树内执行。在一棵有 n 个结点的树上每个操作的摊还代价是 $O(\lg n)$(参见第 17 章)。

跳表[286]是另一种平衡的二叉树。跳表是扩充了一些额外指针的链表。在一个包含 n 个元
338 素的跳表上，每一种字典操作都在 $O(\lg n)$ 期望时间内执行。