

但是，我们和真正的实现还有一段距离。尽管我们可以用不同的数据类型表示 2- 结点和 3- 结点并写出变换所需的代码，但用这种直白的表示方法实现大多数的操作并不方便，因为需要处理的情况实在太多。我们需要维护两种不同类型的结点，将被查找的键和结点中的每个键进行比较，将链接和其他信息从一种结点复制到另一种结点，将结点从一种数据类型转换到另一种数据类型，等等。实现这些不仅需要大量的代码，而且它们所产生的额外开销可能会使算法比标准的二叉查找树更慢。平衡一棵树的初衷是为了消除最坏情况，但我们希望这种保障所需的代码能够越少越好。幸运的是你将看到，我们只需要一点点代价就能用一种统一的方式完成所有变换。

429
l
431

3.3.2 红黑二叉查找树

上文所述的 2-3 树的插入算法并不难理解，现在我们会看到它也不难实现。我们要学习一种名为红黑二叉查找树的简单数据结构来表达并实现它。最后的代码量并不大，但理解这些代码是如何工作的以及为什么能够工作却需要一番仔细的探究。

3.3.2.1 替换 3- 结点

红黑二叉查找树背后的基本思想是用标准的二叉查找树（完全由 2- 结点构成）和一些额外的信息（替换 3- 结点）来表示 2-3 树。我们将树中的链接分为两种类型：红链接将两个 2- 结点连接起来构成一个 3- 结点，黑链接则是 2-3 树中的普通链接。确切地说，我们将 3- 结点表示为由一条左斜的红色链接（两个 2- 结点其中之一是另一个的左子结点）相连的两个 2- 结点，如图 3.3.12 所示。这种表示法的一个优点是，我们无需修改就可以直接使用标准二叉查找树的 `get()` 方法。对于任意的 2-3 树，只要对结点进行转换，我们都可以立即派生出一棵对应的二叉查找树。我们将用这种方式表示 2-3 树的二叉查找树称为红黑二叉查找树（以下简称为红黑树）。

3.3.2.2 一种等价的定义

红黑树的另一种定义是含有红黑链接并满足下列条件的二叉查找树：

- 红链接均为左链接；
- 没有任何一个结点同时和两条红链接相连；
- 该树是完美黑色平衡的，即任意空链接到根结点的路径上的黑链接数量相同。

满足这样定义的红黑树和相应的 2-3 树是一一对应的。

3.3.2.3 一一对应

如果我们将一棵红黑树中的红链接画平，那么所有的空链接到根结点的距离都将是相同的（如图 3.3.13 所示）。如果我们将由红链接相连的结点合并，得到的就是一棵 2-3 树。相反，如果将一棵 2-3 树中的 3- 结点画作由红色左链接相连的两个 2- 结点，那么不存在能够和两条红链接相连的结点，且树必然是完美黑色平衡的，因为黑链接即 2-3 树中的普通链接，根据定义这些链接必然是完美平衡的。无论我们选择用何种方式去定义它们，红黑树都既是二叉查找树，也是 2-3 树，如图 3.3.14 所示。因此，如果我们能够在保持一一对应关系的基础上实现 2-3 树的插入算法，那么我们就能够将两个算法的优点结合起来：二叉查找树中简洁高效的查找方法和 2-3 树中高效的平衡插入算法。

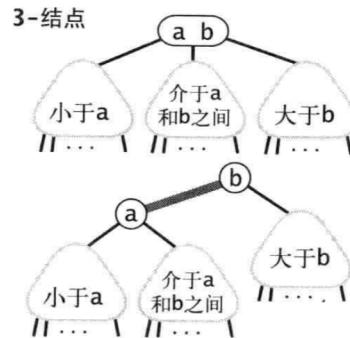
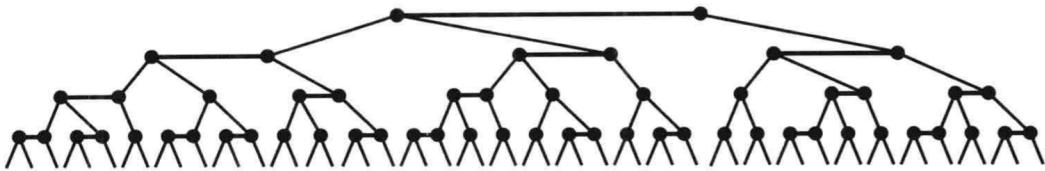


图 3.3.12 由一条红色左链接相连的两个 2- 结点表示一个 3- 结点（另见彩插）



432

图 3.3.13 将红链接画平时，一棵红黑树就是一棵 2-3 树（另见彩插）

3.3.2.4 颜色表示

方便起见，因为每个结点都只会有一条指向自己的链接（从它的父结点指向它），我们将链接的颜色保存在表示结点的 Node 数据类型的布尔变量 color 中。如果指向它的链接是红色的，那么该变量为 true，黑色则为 false。我们约定空链接为黑色。为了代码的清晰我们定义了两个常量 RED 和 BLACK 来设置和测试这个变量。我们使用私有方法 isRed() 来测试一个结点和它的父结点之间的链接的颜色。当我们提到一个结点的颜色时，我们指的是指向该结点的链接的颜色，反之亦然。颜色表示的代码实现如图 3.3.15 所示。

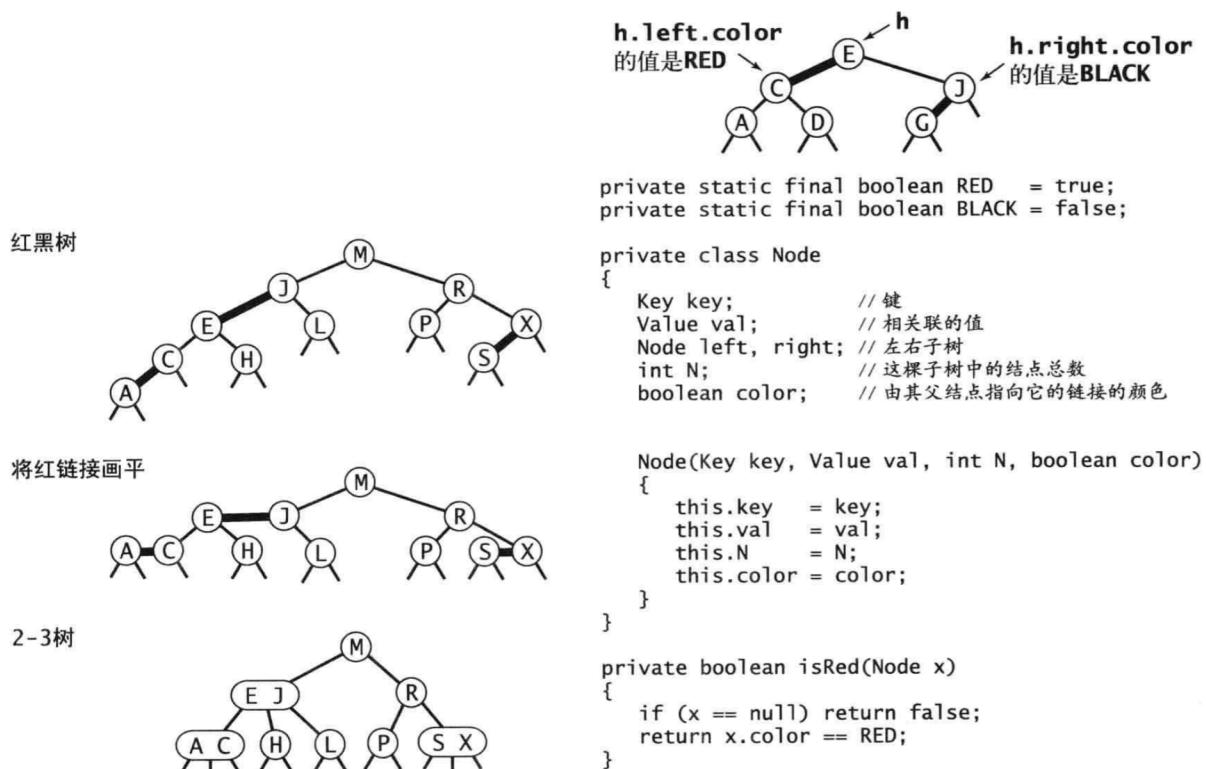


图 3.3.14 红黑树和 2-3 树的一一对应关系（另见彩插）

3.3.2.5 旋转

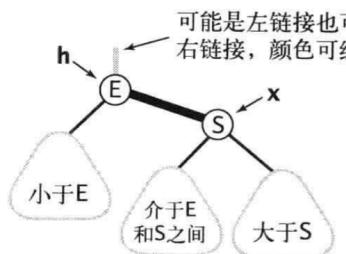
在我们实现的某些操作中可能会出现红色右链接或者两条连续的红链接，但在操作完成前这些情况都会被小心地旋转并修复。旋转操作会改变红链接的指向。首先，假设我们有一条红色的右链接需要被转化为左链接（请见图 3.3.16）。这个操作叫做左旋转，它对应的方法接受一条指向红黑树中的某个结点的链接作为参数。假设被指向的结点的右链接是红色的，这个方法会对树进行必要的调整并返回一个指向包含同一组键的子树且其左链接为红色的根结点的链接。如果你对照图示中调整前后的情况逐行阅读这段代码，你会发现这个操作很容易理解：我们只是将用两个键中的较小

433

者作为根结点变为将较大者作为根结点。实现将一个红色左链接转换为一个红色右链接的一个右旋转的代码完全相同，只需要将 `left` 换成 `right` 即可（如图 3.3.17 所示）。

3.3.2.6 在旋转后重置父结点的链接

无论左旋转还是右旋转，旋转操作都会返回一条链接。我们总是会用 `rotateRight()` 或 `rotateLeft()` 的返回值重置父结点（或是根结点）中相应的链接。返回的链接可能是左链接也可能是右链接，但是我们总会将它赋予父结点中的链接。这个链接可能是红色也可能是黑色——`rotateLeft()` 和 `rotateRight()` 都通过将 `x.color` 设为 `h.color` 保留它原来的颜色。这可能会产生两条连续的红链接，但我们的算法会继续用旋转操作修正这种情况。例如，代码 `h = rotateLeft(h);` 将旋转结点 `h` 的红色右链接，使得 `h` 指向了旋转后的子树的根结点（组成该子树中的所有键和旋转前相同，只是根结点发生了变化）。这种简洁的代码是我们使用递归实现二叉查找树的各种方法的主要原因。你会看到，它使得旋转操作成为了普通插入操作的一个简单补充。



```
Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```

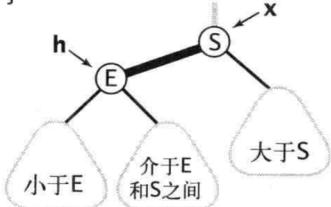
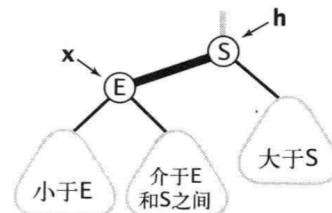


图 3.3.16 左旋转 `h` 的右链接（另见彩插）



```
Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```

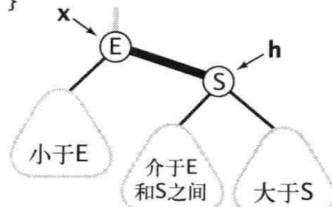


图 3.3.17 右旋转 `h` 的左链接（另见彩插）

434

在插入新的键时我们可以使用旋转操作帮助我们保证 2-3 树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：有序性和完美平衡性。也就是说，我们在红黑树中进行旋转时无需为树的有序性或者完美平衡性担心。下面我们来看看应该如何使用旋转操作来保持红黑树的另外两个重要性质（不存在两条连续的红链接和不存在红色的右链接）。我们先用一些简单的情况热热身。

3.3.2.7 向 2- 结点中插入新键

一棵只含有一个键的红黑树只含有一个 2- 结点。插入另一个键之后，我们马上就需要将它们旋转。如果新键小于老键，我们只需要新增一个红色的结点即可，新的红黑树和单个 3- 结点完全等价。如果新键大于老键，那么新增的红色结点将会产生一条红色的右链接。我们需要使用 `root = rotateLeft(root);` 来将其旋转为红色左链接并修正根结点的链接，插入操作才算完成。两种情

况的结果均为一棵和单个3-结点等价的红黑树，其中含有两个键，一条红链接，树的黑链接高度为1，如图3.3.18所示。

3.3.2.8 向树底部的2-结点插入新键

用和二叉查找树相同的方式向一棵红黑树中插入一个新键会在树的底部新增一个结点（为了保证有序性），但总是用红链接将新结点和它的父结点相连。如果它的父结点是一个2-结点，那么刚才讨论的两种处理方法仍然适用。如果指向新结点的是父结点的左链接，那么父结点就直接变成了一个3-结点；如果指向新结点的是父结点的右链接，这就是一个错误的3-结点，但一次左旋转就能够修正它，如图3.3.19所示。



图3.3.18 向单个2-结点中插入一个新键
(另见彩插)

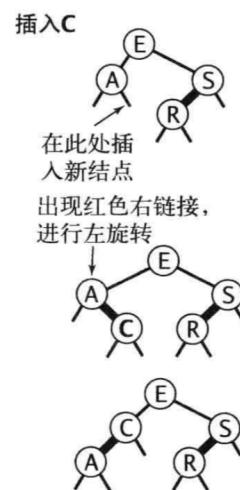


图3.3.19 向树底部的2-结点插入一个新键
(另见彩插)

3.3.2.9 向一棵双键树(即一个3-结点)中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。每种情况下都会产生一个同时连接到两条红链接的结点，而我们的目标就是修正这一点。

- 三者中最简单的情况是新键大于原树中的两个键，因此它被连接到3-结点的右链接。此时树是平衡的，根结点为中间大小的键，它有两条红链接分别和较小和较大的结点相连。如果我们将两条链接的颜色都由红变黑，那么我们就得到了一棵由三个结点组成、高为2的平衡树。它正好能够对应一棵2-3树，如图3.3.20(左)。其他两种情况最终也会转化为这种情况。
- 如果新键小于原树中的两个键，它会被连接到最左边的空链接，这样就产生了两条连续的红链接，如图3.3.20(中)。此时我们只需要将上层的红链接右旋转即可得到第一种情况(中值键为根结点并和其他两个结点用红链接相连)。
- 如果新键介于原树中的两个键之间，这又会产生两条连续的红链接，一条红色左链接接一条红色右链接，如图3.3.20(右)。此时我们只需要将下层的红链接左旋转即可得到第二种情况(两条连续的红色左链接)。

总的来说，我们通过 0 次、1 次和 2 次旋转以及颜色的变化得到了期望的结果。在 2-3 树中，请确认你完全理解了这些转换，它们是红黑树的动态变化的关键。

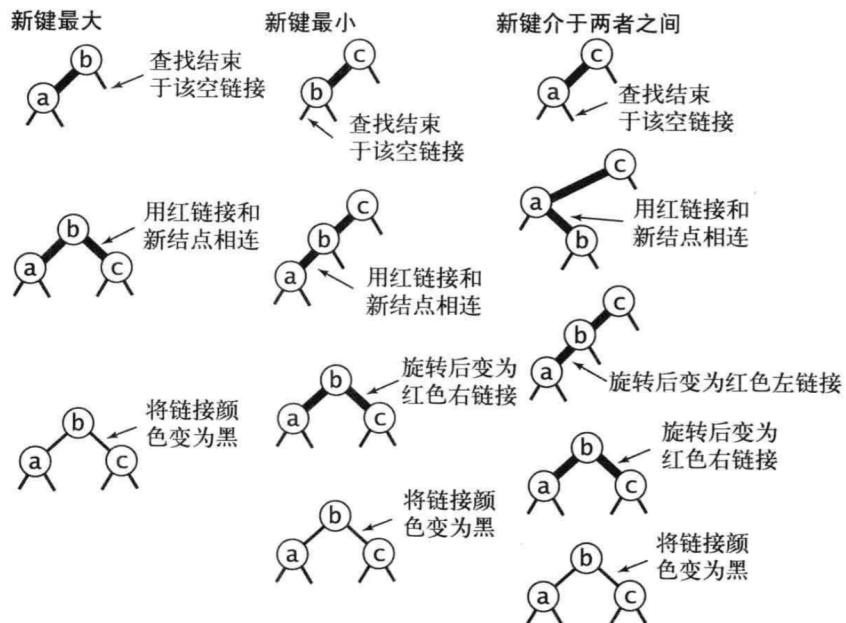


图 3.3.20 向一棵双键树（即一个 3- 结点）中插入一个新键的三种情况（另见彩插）

3.3.2.10 颜色转换

如图 3.3.21 所示，我们专门用一个方法 `flipColors()` 来转换一个结点的两个红色子结点的颜色。除了将子结点的颜色由红变黑之外，我们同时还要将父结点的颜色由黑变红。这项操作最重要的性质在于它和旋转操作一样是局部变换，不会影响整棵树的黑色平衡性。根据这一点，我们马上能够在下面完整地实现红黑树。

3.3.2.11 根结点总是黑色

在 3.3.29 所述的情况下，颜色转换会使根结点变为红色。这也可能出现在很大的红黑树中。严格地说，红色的根结点说明根结点是一个 3- 结点的一部分，但实际情况并不是这样。因此我们在每次插入后都会将根结点设为黑色。注意，每当根结点由红变黑时树的黑链接高度就会加 1。

3.3.2.12 向树底部的 3- 结点插入新键

现在假设我们需要在树的底部的一个 3- 结点下加入一个新结点。前面讨论过的三种情况都会出现，如图 3.3.22 所示。指向新结点的链接可能是 3- 结点的右链接（此时我们只需要转换颜色即可），或是左链接（此时我们需要进行右旋转然后再转换颜色），或是中链接（此时我们需要先左旋转下层链接然后右旋转上层链接，最后再转换颜色）。颜色转换

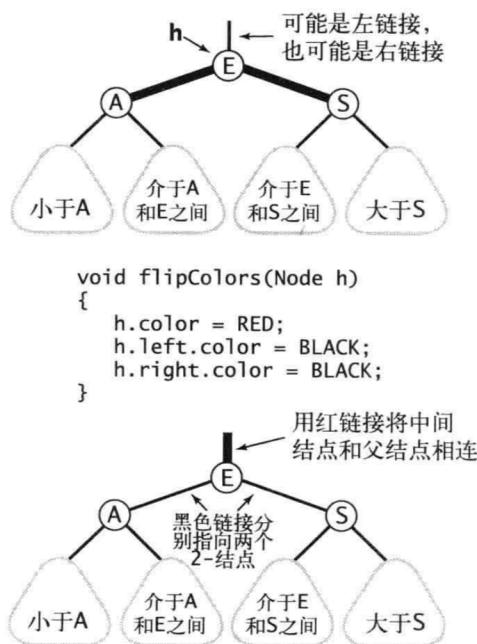


图 3.3.21 分解 4- 结点的同时转换链接的颜色（另见彩插）

会使到中结点的链接变红，相当于将它送入了父结点。这意味着在父结点中继续插入一个新键，我们也会继续用相同的方法解决这个问题。

3.3.2.13 将红链接在树中向上传递

2-3 树中的插入算法需要我们分解 3- 结点，将中间键插入父结点，如此这般直到遇到一个 2- 结点或是根结点。我们所考虑过的所有情况都正是为了达成这个目标：每次必要的旋转之后我们都会进行颜色转换，这使得中结点变红。在父结点看来，处理这样一个红色结点的方式和处理一个新插入的红色结点完全相同，即继续把红链接转移到中结点上去。图 3.3.23 中总结的三种情况显示了在红黑树中实现 2-3 树的插入算法的关键操作所需的步骤：要在 3- 结点下插入新键，先创建一个临时的 4- 结点，将其分解并将红链接由中间键传递给它的父结点。重复这个过程，我们就能将红链接在树中向上传递，直至遇到一个 2- 结点或者根结点。

总之，只要谨慎地使用左旋转、右旋转和颜色转换这三种简单的操作，我们就能够保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根结点的路径向上移动时在所经过的每个结点中顺序完成以下操作，我们就能完成插入操作：

- 如果右子结点是红色的而左子结点是黑色的，进行左旋转；
- 如果左子结点是红色的且它的左子结点也是红色的，进行右旋转；
- 如果左右子结点均为红色，进行颜色转换。

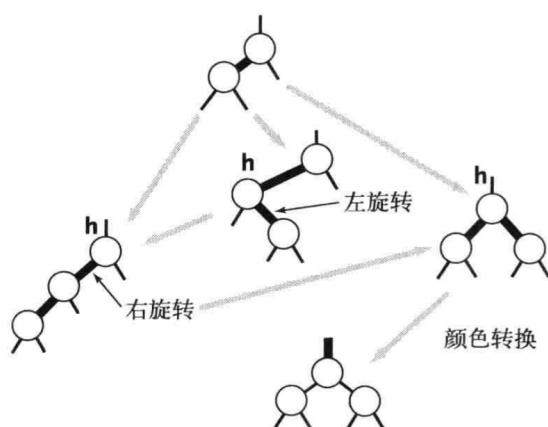


图 3.3.23 红黑树中红链接向上传递（另见彩插）

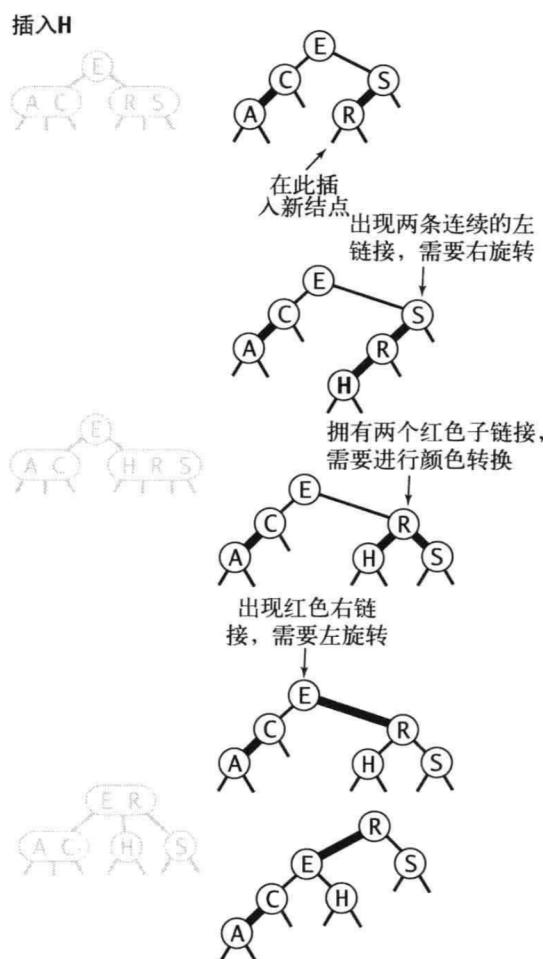


图 3.3.22 向树底部的 3- 结点插入一个新键（另见彩插）

你应该花点时间确认以上步骤处理了前文描述的所有情况。请注意，第一个操作表示将一个 2- 结点变为一个 3- 结点和插入的新结点与树底部的 3- 结点通过它的中链接相连的两种情况。

3.3.3 实现

因为保持树的平衡性所需的操作是由下向上在每个所经过的结点中进行的，将它们植入我们已有的实现中十分简单：只需要在递归调用之后完成这些操作即可，如算法 3.4 所示。上一段中列出的三种操作都可以通过一个检测两个结点的颜色的 if 语句完成。尽管实现所需的代码量很小，