

图 3.3.24 红黑树的构造轨迹（另见彩插）

3.3.4 删除操作

算法 3.4 中的 `put()` 方法是本书中最复杂的实现之一，而红黑树的 `deleteMin()`、`deleteMax()` 和 `delete()` 的实现更麻烦，我们将它们的完整实现留做练习，但这里仍然需要学习它们的基本原理。要描述删除算法，首先我们要回到 2-3 树。和插入操作一样，我们也可以定义一系列局部变换来在删除一个结点的同时保持树的完美平衡性。这个过程比插入一个结点更加复杂，因为我们不仅要在（为了删除一个结点而）构造临时 4- 结点时沿着查找路径向下进行变换，还要在分解

遗留的 4- 结点时沿着查找路径向上进行变换（同插入操作）。

3.3.4.1 自顶向下的 2-3-4 树

作为第一轮热身，我们先学习一个沿查找路径既能向上也能向下进行变换的稍简单的算法：2-3-4 树的插入算法，2-3-4 树中允许存在我们以前见过的 4- 结点。它的插入算法沿查找路径向下进行变换是为了保证当前结点不是 4- 结点（这样树底才有空间来插入新的键），沿查找路径向上进行变换是为了将之前创建的 4- 结点配平，如图 3.3.25 所示。向下的变换和我们在 2-3 树中分解 4- 结点所进行的变换完全相同。如果根结点是 4- 结点，我们就将它分解成三个 2- 结点，使得树高加 1。在向下查找的过程中，如果遇到一个父结点为 2- 结点的 4- 结点，我们将 4- 结点分解为两个 2- 结点并将中间键传递给它的父结点，使得父结点变为一个 3- 结点；如果遇到一个父结点为 3- 结点的 4- 结点，我们将 4- 结点分解为两个 2- 结点并将中间键传递给它的父结点，使得父结点变为一个 4- 结点；我们不必担心会遇到父结点为 4- 结点的 4- 结点，因为插入算法本身就保证了这种情况不会出现。到达树的底部之后，我们也只会遇到 2- 结点或者 3- 结点，所以我们可以插入新的键。要用红黑树实现这个算法，我们需要：

- 将 4- 结点表示为由三个 2- 结点组成的一棵平衡的子树，根结点和两个子结点都用红链接相连；
- 在向下的过程中分解所有 4- 结点并进行颜色转换；
- 和插入操作一样，在向上的过程中用旋转将 4- 结点配平^①。

令人惊讶的是，你只需要移动算法 3.4 的 `put()` 方法中的一行代码就能实现 2-3-4 树中的插入操作：将 `colorFlip()` 语句（及其 `if` 语句）移动到递归调用之前（`null` 测试和比较操作之间）。在多个进程可以同时访问同一棵树的应用中这个算法优于 2-3 树，因为它操作的总是当前结点的一个或两个链接。我们下面要讲的删除算法和它的插入算法类似，而且也适用于 2-3 树。

3.3.4.2 删除最小键

在第二轮热身中我们要学习 2-3 树中删除最小键的操作。我们注意到从树底部的 3- 结点中删除键是很简单的，但 2- 结点则不然。从 2- 结点中删除一个键会留下一个空结点，一般我们会将它替换为一个空链接，但这样会破坏树的完美平衡性。所以我们需要这样做：为了保证我们不会删除一个 2- 结点，我们沿着左链接向下进行变换，确保当前结点不是 2- 结点（可能是 3- 结点，也可能是临时的 4- 结点）。首先，根结点可能有两种情况。如果根是 2- 结点且它的两个子结点都是 2- 结点，我们可以直接将这三个结点变成一个 4- 结点；否则我们需要保证根结点的左子结点不是 2- 结点，如有必要可以从它右侧的兄弟结点“借”一个键来。以上情况如图 3.3.26 所示。在沿着左链接向下过程中，保证以下情况之一成立：

- 如果当前结点的左子结点不是 2- 结点，完成；
- 如果当前结点的左子结点是 2- 结点而它的亲兄弟结点不是 2- 结点，将左子结点的兄弟结点中的一个键移动到左子结点中；

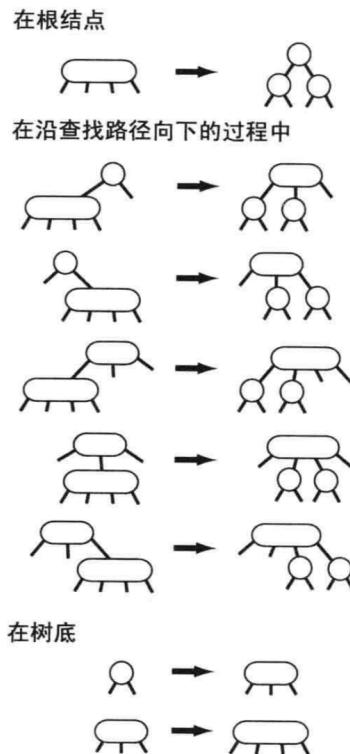


图 3.3.25 自顶向下的 2-3-4 树的插入算法中的变换

441

^① 因为 4- 结点可能存在，所以可以允许一个结点同时连接到两条链接。——译者注

□ 如果当前结点的左子结点和它的亲兄弟结点都是 2- 结点，将左子结点、父结点中的最小键和左子结点最近的兄弟结点合并为一个 4- 结点，使父结点由 3- 结点变为 2- 结点或者由 4- 结点变为 3- 结点。

在遍历的过程中执行这个过程，最后能够得到一个含有最小键的 3- 结点或者 4- 结点，然后我们就可以直接从中将其删除，将 3- 结点变为 2- 结点，或者将 4- 结点变为 3- 结点。然后我们再回头向上分解所有临时的 4- 结点。

3.3.4.3 删除操作

在查找路径上进行和删除最小键相同的变换同样可以保证在查找过程中任意当前结点均不是 2- 结点。如果被查找的键在树的底部，我们可以直接删除它。如果不呢，我们需要将它和它的后继结点交换，就和二叉查找树一样。因为当前结点必然不是 2- 结点，问题已经转化为在一棵根结点不是 2- 结点的子树中删除最小的键，我们可以在这棵子树中使用前文所述的算法。和以前一样，删除之后我们需要向上回溯并分解余下的 4- 结点。

本节末尾的练习中有几道是关于这些删除算法的例子和实现的。有兴趣理解或实现删除算法的读者应该掌握这些练习中的细节。对算法研究感兴趣的读者应该认识到这些方法的重要性，因为这是我们见过的第一种能够同时实现高效的查找、插入和删除操作的符号表实现。下面我们将验证这一点。

442
443

3.3.5 红黑树的性质

研究红黑树的性质就是要检查对应的 2-3 树并对相应的 2-3 树进行分析的过程。我们的最终结论是所有基于红黑树的符号表实现都能保证操作的运行时间为对数级别（范围查找除外，它所需的额外时间和返回的键的数量成正比）。我们重复并强调这一点是因为它十分重要。

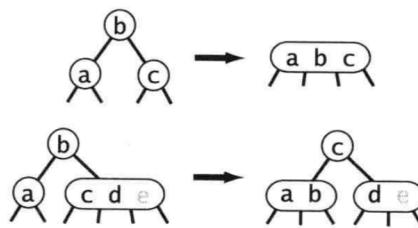
3.3.5.1 性能分析

首先，无论键的插入顺序如何，红黑树都几乎是完美平衡的（请见图 3.3.27）。这从它和 2-3 树的一一对应关系以及 2-3 树的重要性质可以得到。

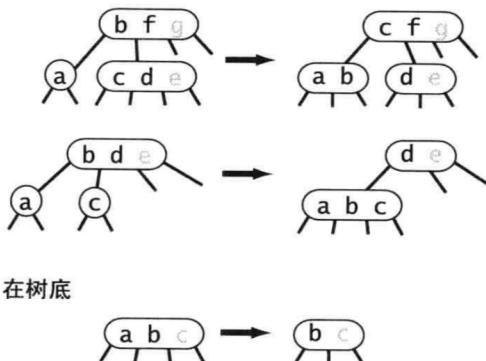
命题 G。 一棵大小为 N 的红黑树的高度不会超过 $2\lg N$ 。

简略的证明。 红黑树的最坏情况是它所对应的 2-3 树中构成最左边的路径结点全部都是 3- 结点而其余均为 2- 结点。最左边的路径长度是只包含 2- 结点的路径长度 ($\sim \lg N$) 的两倍。要按照某种顺序构造一棵平均路径长度为 $2\lg N$ 的最差红黑树虽然可能，但不容易。如果你喜欢数学，你也许会喜欢在练习 3.3.24 中探究这个问题的答案。

在根结点



在沿左链接向下的过程中



在树底



图 3.3.26 删除最小键操作中的变换