

3.3 平衡查找树

我们在前面几节中学习过的算法已经能够很好地用于许多应用程序中，但它们在最坏情况下的性能还是很糟糕。在本节中我们会介绍一种二分查找树并能保证无论如何构造它，它的运行时间都是对数级别的。理想情况下我们希望能够保持二分查找树的平衡性。在一棵含有 N 个结点的树中，我们希望树高为 $\sim \lg N$ ，这样我们就能保证所有查找都能在 $\sim \lg N$ 次比较内结束，就和二分查找一样（请见命题 B）。不幸的是，在动态插入中保证树的完美平衡的代价太高了。在本节中，我们稍稍放松完美平衡的要求并将学习一种能够保证符号表 API 中所有操作（范围查找除外）均能够在对数时间内完成的数据结构。

3.3.1 2-3 查找树

为了保证查找树的平衡性，我们需要一些灵活性，因此在这里我们允许树中的一个结点保存多个键。确切地说，我们将一棵标准的二叉查找树中的结点称为 2- 结点（含有一个键和两条链接），而现在我们引入 3- 结点，它含有两个键和三条链接。2- 结点和 3- 结点中的每条链接都对应着其中保存的键所分割产生的一个区间。

定义。一棵 2-3 查找树或为一棵空树，或由以下结点组成：

- 2- 结点，含有一个键（及其对应的值）和两条链接，左链接指向的 2-3 树中的键都小于该结点，右链接指向的 2-3 树中的键都大于该结点。
- 3- 结点，含有两个键（及其对应的值）和三条链接，左链接指向的 2-3 树中的键都小于该结点，中链接指向的 2-3 树中的键都位于该结点的两个键之间，右链接指向的 2-3 树中的键都大于该结点。

和以前一样，我们将指向一棵空树的链接称为空链接。2-3 查找树如图 3.3.1 所示。

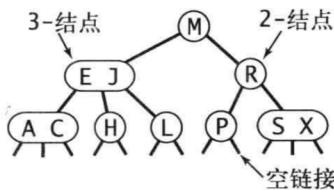


图 3.3.1 2-3 查找树示意图

一棵完美平衡的 2-3 查找树中的所有空链接到根结点的距离都应该是相同的。简洁起见，这里我们用 2-3 树指代一棵完美平衡的 2-3 查找树（在其他情况下这个词应该表示一种更一般的结构）。稍后我们将会学习定义并高效地实现 2- 结点、3- 结点和 2-3 树的基本操作。现在先假设我们已经能够自如地操作它们并来看看应该如何将它们用作查找树。

424

3.3.1.1 查找

将二叉查找树的查找算法一般化我们就能够直接得到 2-3 树的查找算法。要判断一个键是否在树中，我们先将它和根结点中的键比较。如果它和其中任意一个相等，查找命中；否则我们就根据比较的结果找到指向相应区间的链接，并在其指向的子树中递归地继续查找。如果这是个空链接，查找未命中。具体查找过程如图 3.3.2 所示。

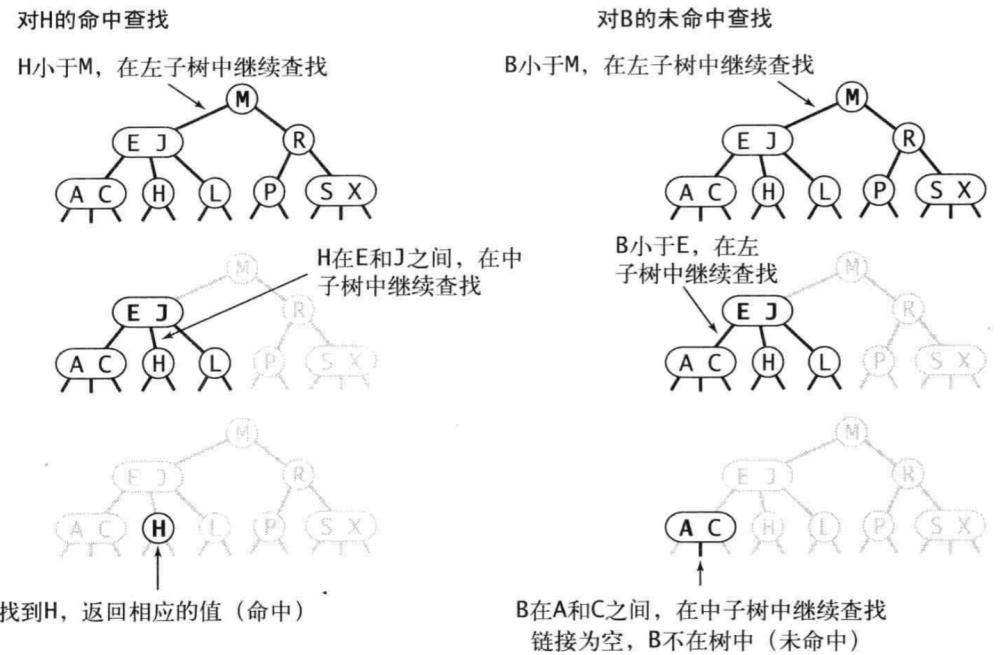


图 3.3.2 2-3 树中的查找命中（左）和未命中（右）

3.3.1.2 向 2- 结点中插入新键

要在 2-3 树中插入一个新结点，我们可以和二叉查找树一样先进行一次未命中的查找，然后把新结点挂在树的底部。但这样的话树无法保持完美平衡性。我们使用 2-3 树的主要原因就在于它能够在插入后继续保持平衡。如果未命中的查找结束于一个 2- 结点，事情就好办了：我们只要把这个 2- 结点替换为一个 3- 结点，将要插入的键保存在其中即可（如图 3.3.3 所示）。如果未命中的查找结束于一个 3- 结点，事情就要麻烦一些。

425

3.3.1.3 向一棵只含有一个 3- 结点的树中插入新键

在考虑一般情况之前，先假设我们需要向一棵只含有一个 3- 结点的树中插入一个新键。这棵树中有两个键，所以在它唯一的结点中已经没有可插入新键的空间了。为了将新键插入，我们先临时将新键存入该结点中，使之成为一个 4- 结点。它很自然地扩展了以前的结点并含有 3 个键和 4 条链接。创建一个 4- 结点很方便，因为很容易将它转换为一棵由 3 个 2- 结点组成的 2-3 树，其中一个结点（根）含有中键，一个结点含有 3 个键中的最小者（和根结点的左链接相连），一个结点含有 3 个键中的最大者（和根结点的右链接相连）。这棵树既是一棵含有 3 个结点的二叉查找树，同时也是一棵完美平衡的 2-3 树，因为其中所有的空链接到根结点的距离都相等。插入前树的高度为 0，插入后树的高度为 1。这个例子很简单但却值得学习，它说明了 2-3 树是如何生长的，如图 3.3.4 所示。

3.3.1.4 向一个父结点为 2- 结点的 3- 结点中插入新键

作为第二轮热身，假设未命中的查找结束于一个 3- 结点，而它的父结点是一个 2- 结点。在这种情况下我们需要在维持树的完美平衡的前提下为新键腾出空间。我们先像刚才一样构造一个临时

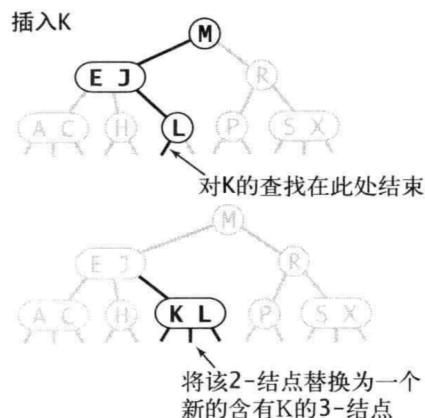


图 3.3.3 向 2- 结点中插入新的键

的 4- 结点并将其分解，但此时我们不会为中键创建一个新结点，而是将其移动至原来的父结点中。你可以将这次转换看成将指向原 3- 结点的一条链接替换为新父结点中的原中键左右两边的两条链接，并分别指向两个新的 2- 结点。根据我们的假设，父结点中是有空间的：父结点是一个 2- 结点（一个键两条链接），插入之后变成了一个 3- 结点（两个键 3 条链接）。另外，这次转换也并不影响（完美平衡的）2-3 树的主要性质。树仍然是有序的，因为中键被移动到父结点中去了；树仍然是完美平衡的，插入后所有的空链接到根结点的距离仍然相同。请确认你完全理解了这次转换——它是 2-3 树的动态变化的核心，其过程如图 3.3.5 所示。

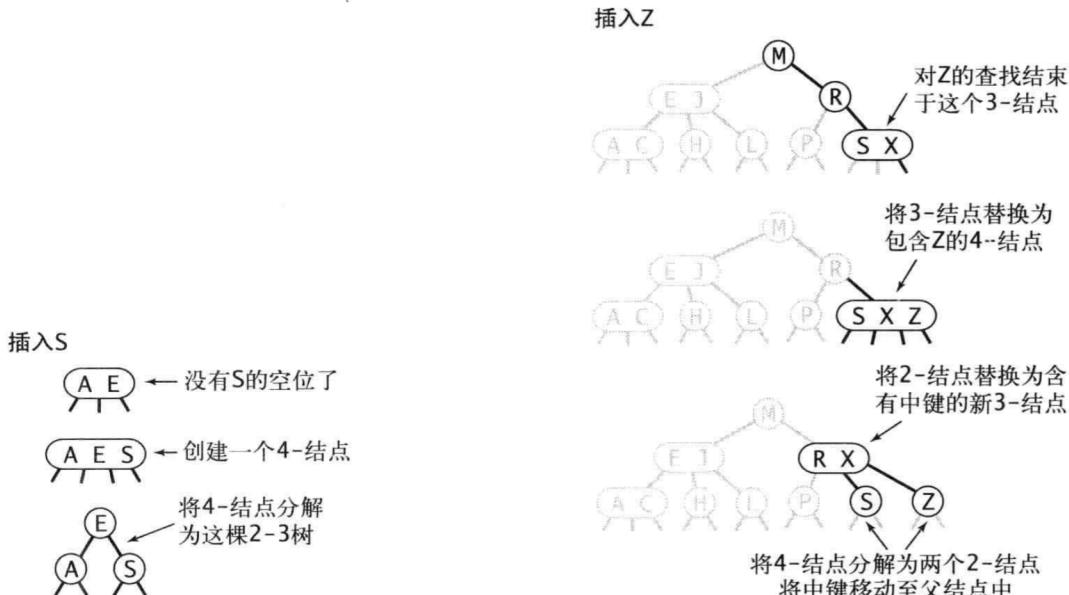


图 3.3.4 向一棵只含有一个 3- 结点的树中插入新键

图 3.3.5 向一个父结点为 2- 结点的 3- 结点中插入新键

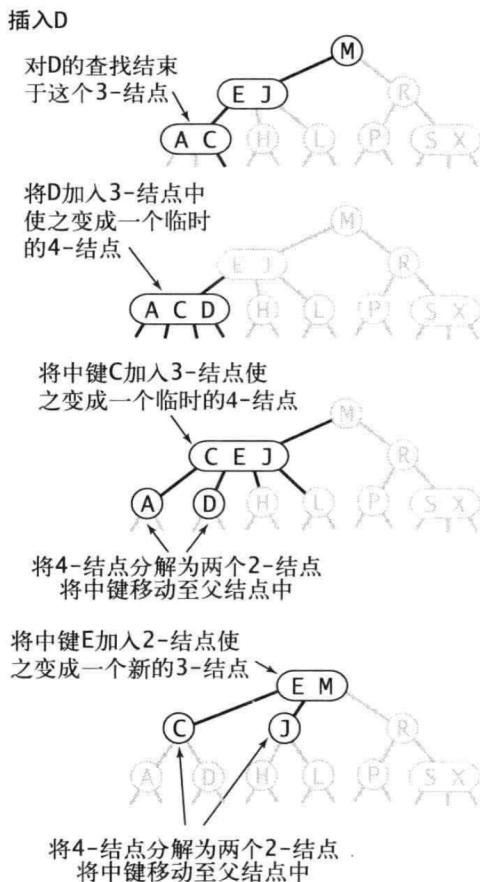
426

3.3.1.5 向一个父结点为 3- 结点的 3- 结点中插入新键

现在假设未命中的查找结束于一个父结点为 3- 结点的结点。我们再次和刚才一样构造一个临时的 4- 结点并分解它，然后将它的中键插入它的父结点中。但父结点也是一个 3- 结点，因此我们再用这个中键构造一个新的临时 4- 结点，然后在这个结点上进行相同的变换，即分解这个父结点并将它的中键插入到它的父结点中去。推广到一般情况，我们就这样一直向上不断分解临时的 4- 结点并将中键插入更高层的父结点，直至遇到一个 2- 结点并将它替换为一个不需要继续分解的 3- 结点，或者是到达 3- 结点的根。该过程如图 3.3.6 所示。

3.3.1.6 分解根结点

如果从插入结点到根结点的路径上全都是 3- 结点，我们的根结点最终变成一个临时的 4- 结点。此时我们可以按照向一棵只含有一个 3- 结点的树中插入新键的方法处理这个问题。我们将临时的 4- 结点分解为 3 个 2- 结点，使得树高加 1，如图 3.3.7 所示。请注意，这次最后的变换仍然保持了树的完美平衡性，因为它变换的是根结点。



427 图 3.3.6 向一个父结点为 3- 结点的 3- 结点中插入新键

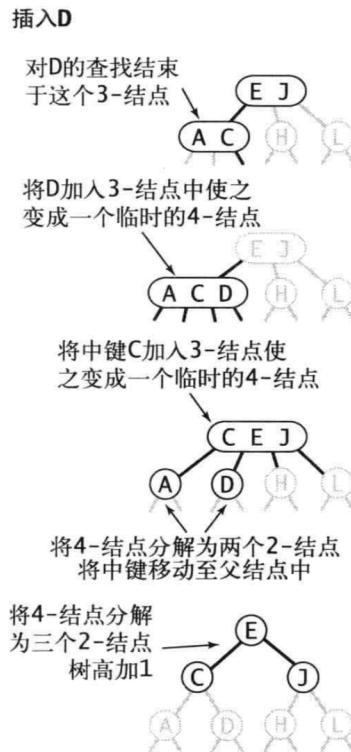


图 3.3.7 分解根结点

3.3.1.7 局部变换

将一个 4- 结点分解为一棵 2-3 树可能有 6 种情况，都总结在了图 3.3.8 中。这个 4- 结点可能是根结点，可能是一个 2- 结点的左子结点或者右子结点，也可能是一个 3- 结点的左子结点、中子结点或者右子结点。2-3 树插入算法的根本在于这些变换都是局部的：除了相关的结点和链接之外不必修改或者检查树的其他部分。每次变换中，变更的链接数量不会超过一个很小的常数。需要特别指出的是，不光是在树的底部，树中的任何地方只要符合相应的模式，变换都可以进行。每个变换都会将 4- 结点中的一个键送入它的父结点中，并重构相应的链接而不必涉及树的其他部分。

3.3.1.8 全局性质

这些局部变换不会影响树的全局有序性和平衡性：任意空链接到根结点的路径长度都是相等的。作为参考，图 3.3.9 所示的是当一个 4- 结点是一个 3- 结点的中子结点时的完整变换情况。如果在变换之前根结点到所有空链接的路径长度为 h ，那么变换之后该长度仍然为 h 。所有的变换都具有这个性质，即使是将一个 4- 结点分解为两个 2- 结点并将其父结点由 2- 结点变为 3- 结点，或是由 3- 结点变为一个临时的 4- 结点时也是如此。当根结点被分解为 3 个 2- 结点时，所有空链接到根结点的路径长度才会加 1。如果你还没有完全理解，请完成练习 3.3.7。它要求你为其他的 5 种情况画出图 3.3.8 的扩展图来证明这一点。理解所有局部变换都不会影响整棵树的有序性和平衡性是理解这个算法的关键。

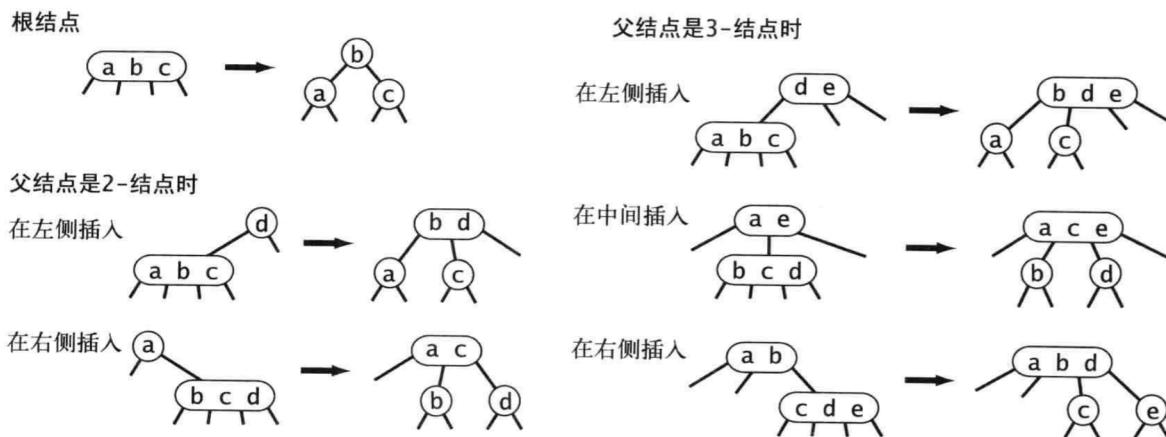


图 3.3.8 在一棵 2-3 树中分解一个 4- 结点的情况汇总

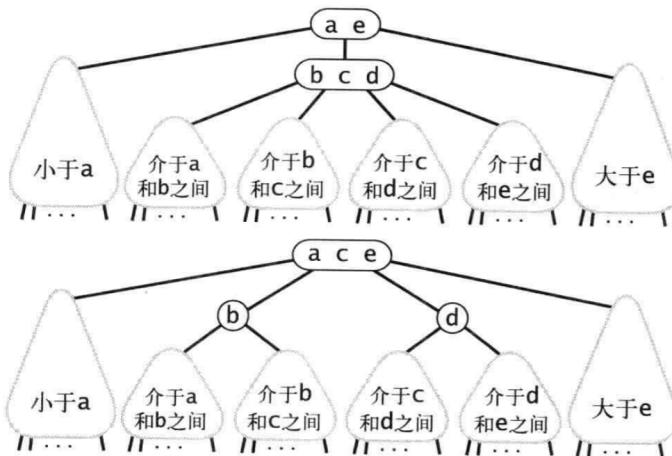


图 3.3.9 4- 结点的分解是一次局部变换，不会影响树的有序性和平衡性

428

和标准的二叉查找树由上向下生长不同，2-3 树的生长是由下向上的。如果你花点时间仔细研究一下图 3.3.10，就能很好地理解 2-3 树的构造方式。它给出了我们的标准索引测试用例中产生的一系列 2-3 树，以及一系列由同一组键按照升序依次插入到树中时所产生的所有 2-3 树。还记得在二叉查找树中，按照升序插入 10 个键会得到高度为 9 的一棵最差查找树吗？如果使用 2-3 树，树的高度是 2。

以上的文字已经足够为我们定义一个使用 2-3 树作为数据结构的符号表的实现了。2-3 树的分析和二叉查找树的分析大不相同，因为我们主要感兴趣的是最坏情况下的性能，而非一般情况（这种情况下我们会用随机键模型分析预期的性能）。在符号表的实现中，一般我们无法控制用例会按照什么顺序向表中插入键，因此对最坏情况的分析是唯一能够提供性能保证的办法。

命题 F。在一棵大小为 N 的 2-3 树中，查找和插入操作访问的结点必然不超过 $\lg N$ 个。

证明。一棵含有 N 个结点的 2-3 树的高度在 $\lceil \log_3 N \rceil = \lceil (\lg N) / (\lg 3) \rceil$ （如果树中全是 3- 结点）和 $\lfloor \lg N \rfloor$ （如果树中全是 2- 结点）之间（请见练习 3.3.4）。

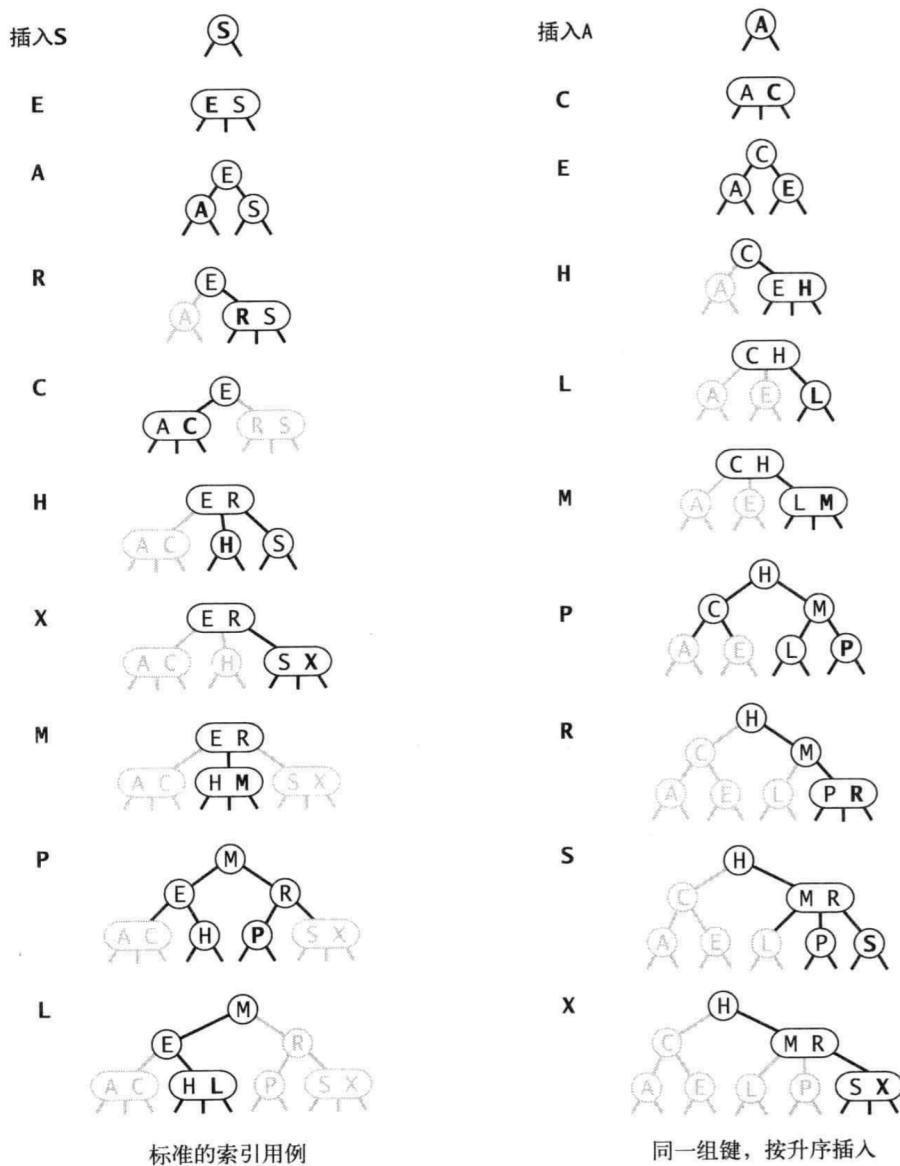


图 3.3.10 2-3 树的构造轨迹

因此我们可以确定 2-3 树在最坏情况下仍有较好的性能。每个操作中处理每个结点的时间都不会超过一个很小的常数，且这两个操作都只会访问一条路径上的结点，所以任何查找或者插入的成本都肯定不会超过对数级别。通过对图 3.3.11 中的 2-3 树和表 3.2.1 中由相同的键构造的二叉查找树你也可以看到，完美平衡的 2-3 树要平展得多。例如，含有 10 亿个结点的一棵 2-3 树的高度仅在 19 到 30 之间。我们最多只需要访问 30 个结点就能够在 10 亿个键中进行任意查找和插入操作，这是相当惊人的。

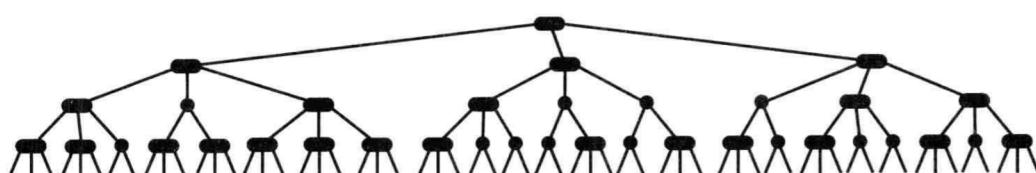


图 3.3.11 由随机键构造的一棵典型的 2-3 树

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。

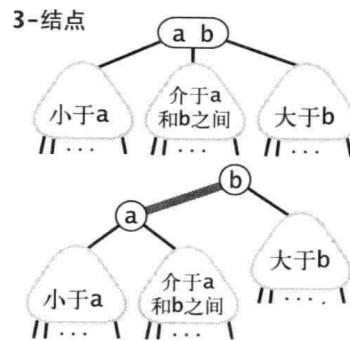


图 3.3.12 由一条红色左链接相连的两个 2- 结点表示一个 3- 结点（另见彩插）