数据结构的扩张

一些工程应用需要的只是一些“教科书”中的标准数据结构，比如双链表、散列表或二叉搜索树等，然而也有许多其他的应用需要对现有数据结构进行少许地创新和改造，但是只在很少情况下需要创造出一类全新类型的数据结构。更经常的是，通过存储额外信息的方法来扩张一种标准的数据结构，然后对这种数据结构，编写新的操作来支持所需要的应用。然而对数据结构的扩张并不总是简单直接的，因为添加的信息必须要能被该数据结构上的常规操作更新和维护。

本章讨论通过扩张红黑树构造出的两种数据结构。14.1 节介绍一种支持一般动态集合上顺序统计操作的数据结构。通过这种数据结构，我们可以快速地找到一个集合中的第 i 小的数，或给出一个指定元素在集合的全序中的位置。14.2 节抽象出数据结构的扩张过程，并给出一个简化红黑树扩张的定理。14.3 节使用这个定理来设计一种用于维护由区间(如时间区间)构成的动态集合的数据结构。给定一个要查询的区间，我们能快速地找到集合中一个能与其重叠的区间。

14.1 动态顺序统计

第 9 章中介绍了顺序统计的概念。n 个元素集合中的第 $i(i \in \{1, 2, \cdots, n\})$ 个顺序统计量就是简单地规定为该集合中的具有第 i 小关键字的元素。对于一个无序的集合，我们知道能够在 $O(n)$ 的时间内确定任何的顺序统计量。本节将介绍如何修改红黑树，使得可以在 $O(\lg n)$ 时间内确定任何的顺序统计量。我们还将看到如何在 $O(\lg n)$ 时间内计算一个元素的**秩**，即它在集合线性序中的位置。 339

图 14-1 显示了一种支持快速顺序统计操作的数据结构。**顺序统计树**(order-statistic tree) T 只是简单地在每个结点上存储附加信息的一棵红黑树。在红黑树的结点 x 中，除了通常属性 $x.key$、$x.color$、$x.p$、$x.left$ 和 $x.right$ 之外，还包括另一个属性 $x.size$。这个属性包含了以 x 为根的子树(包括 x 本身)的(内)结点数，即这棵子树的大小。如果定义哨兵的大小为 0，也就是设置 $T.nil.size$ 为 0，则有等式：

$$x.size = x.left.size + x.right.size + 1$$

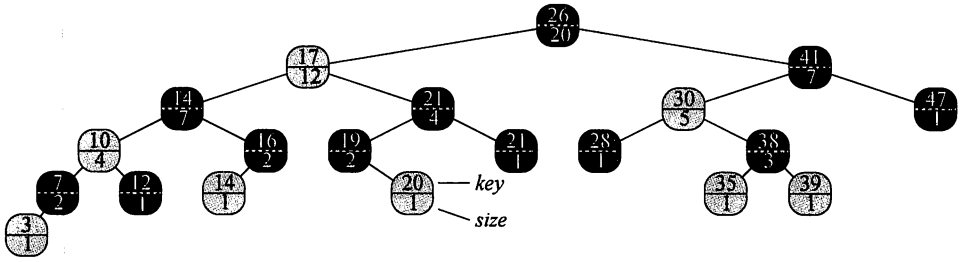

图 14-1 一棵顺序统计树，它是一棵扩张的红黑树。浅阴影结点为红色，深阴影结点为黑色。除了通常的红黑树所具有的属性外，每个结点 x 还具有属性 $x.size$，即以 x 为根的子树(除哨兵外)的结点个数

在一棵顺序统计树中，我们并不要求关键字各不相同。(例如，图 14-1 中的树就包含了两个值为 14 的关键字和两个值为 21 的关键字。)在有相等关键字的情况下，前面秩的定义便不再适合。为此，我们通过定义一个元素的秩为在中序遍历树时输出的位置，来消除原顺序统计树定义的不确定性。如图 14-1 所示，存储在黑色结点的关键字 14 的秩为 5，存储在红色结点的关键字 14 的秩为 6。

查找具有给定秩的元素

在说明插入和删除过程中如何维护 $size$ 信息之前，我们先来讨论利用这个附加信息来实现

的两个顺序统计查询。首先一个操作是对具有给定秩的元素的检索。过程 OS-SELECT(x, i)返回一个指针，其指向以 x 为根的子树中包含第 i 小关键字的结点。为找出顺序统计树 T 中的第 i
340 小关键字，我们调用过程 OS-SELECT($T.root$, i)。

```
OS-SELECT(x,i)
1  r = x.left.size + 1
2  if i == r
3       return x
4  elseif i < r
5       return OS-SELECT(x.left,i)
6  else return OS-SELECT(x.right,i−r)
```

OS-SELECT 的第 1 行计算以 x 为根的子树中结点 x 的秩 r。$x.left.size$ 的值是对以 x 为根的子树进行中序遍历后排在 x 之前的结点个数。因此，$x.left.size+1$ 就是以 x 为根的子树中结点 x 的秩。如果 $i=r$，那么结点 x 就是第 i 小元素，这样第 3 行返回 x。如果 $i<r$，那么第 i 小元素在 x 的左子树中，因此在第 5 行中对 $x.left$ 进行递归调用。如果 $i>r$，那么第 i 小元素在 x 的右子树中。因为在对以 x 为根的子树进行中序遍历时，共有 r 个元素排在 x 的右子树之前，故在以 x 为根的子树中第 i 小元素就是以 $x.right$ 为根的子树中第($i-r$)小元素。第 6 行通过递归调用来确定这个元素。

为明白 OS-SELECT 是如何操作的，考察在图 14-1 所示的顺序统计树上查找第 17 小元素的查找过程。以 x 为根开始，其关键字为 26，$i=17$。因为 26 的左子树的大小为 12，故它的秩为 13。因此，秩为 17 的结点是 26 的右子树中第 17−13=4 小的元素。递归调用后，x 为关键字 41 的结点，$i=4$。因为 41 的左子树大小为 5，故它的秩为 6。这样，可以知道秩为 4 的结点是 41 的左子树中第 4 小元素。再次递归调用后，x 为关键字 30 的结点，在其子树中它的秩为 2。如此，再进行一次递归调用，就能找到以关键字 38 的结点为根的子树中第 4−2=2 小的元素。它的左子树大小为 1，这意味着它就是第 2 小元素。最终，该过程返回一个指向关键字为 38 的结点的指针。

因为每次递归调用都在顺序统计树中下降一层，OS-SELECT 的总时间最差与树的高度成正比。又因为该树是一棵红黑树，其高度为 $O(\lg n)$，其中 n 为数的结点数。所以，对于 n 个元素的动态集合，OS-SELECT 的运行时间为 $O(\lg n)$。

确定一个元素的秩

给定指向顺序统计树 T 中结点 x 的指针，过程 OS-RANK 返回对 T 中序遍历对应的线性序
341 中 x 的位置。

```
OS-RANK(T,x)
1  r = x.left.size+1
2  y = x
3  while y ≠ T.root
4       if y == y.p.right
5            r = r + y.p.left.size+1
6       y = y.p
7  return r
```

这个过程工作如下。我们可以认为 x 的秩是中序遍历次序排在 x 之前的结点数再加上 1(代表 x 自身)。OS-RANK 保持了以下的循环不变式：

第 3～6 行 **while** 循环的每次迭代开始，r 为以结点 y 为根的子树中 $x.key$ 的秩。

下面使用这个循环不变式来说明 OS-RANK 能正确地工作。

初始化：第一次迭代之前，第 1 行置 r 为以 x 为根的子树中 $x.key$ 的秩。第 2 行置 $y=x$，使得首次执行第 3 行中的测试时，循环不变式为真。

保持：在每一次 **while** 循环迭代的最后，都要置 $y=y.p$。这样，我们必须要证明：如果 r 是在循环体开始处以 y 为根的子树中 $x.key$ 的秩，那么 r 是在循环体结尾处以 $y.p$ 为根的子树中 $x.key$ 的秩。在 **while** 循环的每次迭代中，考虑以 $y.p$ 为根的子树。我们对以结点 y 为根的子树已经计数了以中序遍历次序先于 x 的结点个数，故要加上以 y 的兄弟结点为根的子树以中序遍历次序先于 x 的结点数，如果 $y.p$ 也先于 x，则该计数还要加 1。如果 y 是左孩子，$y.p$ 和 $y.p$ 的右子树中的所有结点都不会先于 x，r 保持不变；否则，y 是右孩子，并且 $y.p$ 和 $y.p$ 左子树中的所有结点都先于 x，于是在第 5 行中，将当前的 r 值再加上 $y.p.left.size+1$。

终止：当 $y=T.root$ 时，循环终止，此时以 y 为根的子树是一棵完整树。因此，r 的值就是这棵完整树中 $x.key$ 的秩。

作为一个例子，当我们在图 14-1 的顺序统计树上运行 OS-RANK，以确定关键字为 38 的结点的秩时，在 **while** 循环的开始处，$y.key$ 和 r 的一系列值如下：

迭　代	$y.key$	r
1	38	2
2	30	4
3	41	4
4	26	17

342

该过程返回的秩为 17。

因为 **while** 循环的每次迭代耗费 $O(1)$时间，且 y 在每次迭代中沿树上升一层，所以最坏情况下 OS-RANK 的运行时间与树的高度成正比：在 n 个结点的顺序统计树上为 $O(\lg n)$。

对子树规模的维护

给定每个结点的 *size* 属性后，OS-SELECT 和 OS-RANK 能迅速计算出所需的顺序统计信息。然而除非能用红黑树上经过修改的基本操作对 *size* 属性加以有效的维护，否则，我们的工作将变得没意义。下面就来说明在不影响插入和删除操作的渐近运行时间的前提下，如何维护子树规模。

由 13.3 节可知，红黑树上的插入操作包括两个阶段。第一阶段从根开始沿树下降，将新结点插入作为某个已有结点的孩子。第二阶段沿树上升，做一些变色和旋转操作来保持红黑树性质。

在第一阶段中为了维护子树的规模，对由根至叶子的路径上遍历的每一个结点 x，都增加 $x.size$ 属性。新增加结点的 *size* 为 1。由于一条遍历的路径上共有 $O(\lg n)$个结点，故维护 *size* 属性的额外代价为 $O(\lg n)$。

在第二阶段，对红黑树结构上的改变仅仅是由旋转所致，旋转次数至多为 2。此外，旋转是一种局部操作：它仅会使两个结点的 *size* 属性失效，而围绕旋转操作的链就是与这两个结点关联。参照 13.2 节的 LEFT-ROTATE(T，x)代码，增加下面两行：

```
13  y.size = x.size
14  x.size = x.left.size+x.right.size+1
```

图 14-2 说明了 *size* 属性是如何被更新的。对 RIGHT-ROTATE 做相应的改动。

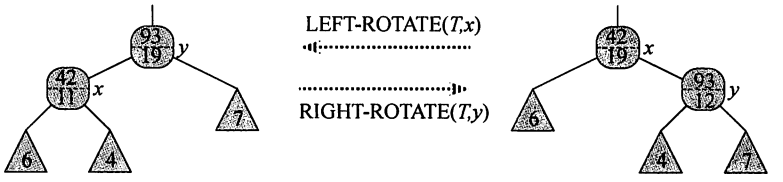

图 14-2　在旋转过程中修改子树的大小。与围绕旋转的链相关联的两个结点，它们的 *size* 属性要更新。这些更新是局部的，仅需要存储在 x 和 y 中的 *size* 信息，以及图中三角形子树的根中的 *size* 信息

因为在红黑树的插入过程中至多进行两次旋转，所以在第二阶段更新 *size* 属性只需要 $O(1)$ 的额外时间。因此，对一棵有 n 个结点的顺序统计树插入元素所需要的总时间为 $O(\lg n)$，从渐近意义上看，这与一般的红黑树是一样的。

红黑树上的删除操作也包括两个阶段：第一阶段对搜索树进行操作，第二阶段做至多三次旋转，其他对结构没有任何影响(见 13.4 节)。第一阶段中，要么将结点 y 从树中删除，要么将它在树中上移。为了更新子树的规模，我们只需要遍历一条由结点 y(从它在树中的原始位置开始)至根的简单路径，并减少路径上每个结点的 *size* 属性的值。因为在 n 个结点的红黑树中，这样一条路径的长度为 $O(\lg n)$，所以第一阶段维护 *size* 属性所耗费的额外时间为 $O(\lg n)$。第二阶段采用与插入相同的方式来处理删除操作中的 $O(1)$ 次旋转。所以对有 n 个结点的顺序统计树进行插入与删除操作，包括维护 *size* 属性，都只需要 $O(\lg n)$ 的时间。

练习

14.1-1 对于图 14-1 中的红黑树 T，说明执行 OS-SELECT($T.root$, 10)的过程。

14.1-2 对于图 14-1 中的红黑树 T 和关键字 $x.key$ 为 35 的结点 x，说明执行 OS-RANK(T, x)的过程。

14.1-3 写出 OS-SELECT 的非递归版本。

14.1-4 写出一个递归过程 OS-KEY-RANK(T, k)，以一棵顺序统计树 T 和一个关键字 k 作为输入，要求返回 k 在由 T 表示的动态集合中的秩。假设 T 的所有关键字都不相同。

343 ~ 344

14.1-5 给定 n 个元素的顺序统计树中的一个元素 x 和一个自然数 i，如何在 $O(\lg n)$ 的时间内确定 x 在该树线性序中的第 i 个后继？

14.1-6 在 OS-SELECT 或 OS-RANK 中，注意到无论什么时候引用结点的 *size* 属性都是为了计算一个秩。相应地，假设每个结点都存储它在以自己为根的子树中的秩。试说明在插入和删除时，如何维护这个信息。(注意，这两种操作都可能引起旋转。)

14.1-7 说明如何在 $O(n \lg n)$ 时间内，利用顺序统计树对大小为 n 的数组中的逆序对(见思考题 2-4)进行计数。

***14.1-8** 现有一个圆上的 n 条弦，每条弦都由其端点来定义。请给出一个能在 $O(n \lg n)$ 时间内确定圆内相交弦对数的算法。(例如，如果 n 条弦都为直径，它们相交于圆心，则正确的答案为 $\binom{n}{2}$。)假设任意两条弦都不会共享端点。

14.2 如何扩张数据结构

对基本的数据结构进行扩张以支持一些附加功能，在算法设计过程中是相当常见的。在下一节中，我们将再次通过对数据结构进行扩张，来设计一种支持区间操作的数据结构。本节先来介绍这种扩张过程的步骤，同时证明一个定理，在许多情况下，该定理使得我们可以很容易地扩张红黑树。

扩张一种数据结构可以分为 4 个步骤：

1. 选择一种基础数据结构。
2. 确定基础数据结构中要维护的附加信息。
3. 检验基础数据结构上的基本修改操作能否维护附加信息。
4. 设计一些新操作。

以上仅作为一个一般模式，读者不应盲目地按照上面给定的次序来执行这些步骤。大多数的设计工作都包含试探和纠错的成分，过程中的所有步骤通常都可以并行进行。例如，如果我们

不能有效地维护附加信息，那么确定附加信息以及设计新的操作(步骤 2 和步骤 4)就没有任何意义。然而，这个 4 步法可以使读者在扩张数据结构时，目标明确且有条不紊。 345

在 14.1 节设计顺序统计树时，我们就依照了这 4 个步骤。对于步骤 1，选择红黑树作为基础数据结构。红黑树是一种合适的选择，这源于它能有效地支持一些基于全序的动态集合操作，如 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR。

对于步骤 2，添加了 *size* 属性，在每个结点 x 中的 *size* 属性存储了以 x 为根的子树的大小。一般地，附加信息可使得各种操作更加有效。例如，我们本可以仅用树中存储的关键字来实现 OS-SELECT 和 OS-RANK，但它们却不能在 $O(\lg n)$运行时间内完成。有时候，附加信息是指针类信息，而不是具体的数据，如练习 14.2-1。

对于步骤 3，我们保证了插入和删除操作仍能在 $O(\lg n)$时间内维护 *size* 属性。比较理想的是，只需要更新该数据结构中的几个元素就可以维护附加信息。例如，如果把每个结点的秩存储在树中，那么 OS-SELECT 和 OS-RANK 能够较快运行，但是当插入一个新的最小元素时，会导致树中每个结点的秩发生变化。如果我们存储的是子树的大小，则插入一个新的元素时仅会使 $O(\lg n)$个结点的信息发生改变。

对于步骤 4，我们设计了新操作 OS-SELECT 和 OS-RANK。归根结底，一开始考虑去扩张一个数据结构的原因就是为了满足新操作的需要。然而有时并不是为了设计一些新操作，而是利用附加信息来加速已有的操作，如练习 14.2-1。

对红黑树的扩张

当红黑树作为基础数据结构时，可以证明，某些类型的附加信息总是可以用插入和删除操作来进行有效的维护，从而使步骤 3 非常容易做到。下面定理的证明与 14.1 节用顺序统计树来维护 *size* 属性的论证类似。

定理 14.1(红黑树的扩张)　*设 f 是 n 个结点的红黑树 T 扩张的属性，且假设对任一结点 x，f 的值仅依赖于结点 x、$x.left$ 和 $x.right$ 的信息，还可能包括 $x.left.f$ 和 $x.right.f$。那么，我们可以在插入和删除操作期间对 T 的所有结点的 f 值进行维护，并且不影响这两个操作的 $O(\lg n)$渐近时间性能。*

证明　证明的主要思想是，对树中某结点 x 的 f 属性的变动只会影响到 x 的祖先。也就是说，修改 $x.f$ 只需要更新 $x.p.f$，改变 $x.p.f$ 的值只需要更新 $x.p.p.f$，如此沿树向上。一旦 346 更新到 $T.root.f$，就不再有其他任何结点依赖于新值，于是过程结束。因为红黑树的高度为 $O(\lg n)$，所以改变某结点的 f 属性要耗费 $O(\lg n)$时间，来更新被该修改所影响的所有结点。

一个结点 x 插入到树 T 由两个阶段构成(见 13.3 节)。第一阶段是将 x 作为一个已有结点 $x.p$ 的孩子被插入。$x.f$ 的值可以在 $O(1)$时间内计算出。因为根据假设，$x.f$ 仅依赖于 x 本身的其他属性信息和 x 的子结点中的信息，而此时 x 的子结点都是哨兵 $T.nil$。当 $x.f$ 被计算出时，这个变化就沿树向上传播。这样，插入第一阶段的全部时间为 $O(\lg n)$。在第二阶段期间，树结构的仅有变动来源于旋转操作。由于在一次旋转过程中仅有两个结点发生变化，所以每次旋转更新 f 属性的全部时间为 $O(\lg n)$。又因为插入操作中的旋转次数至多为 2，所以插入的总时间为 $O(\lg n)$。

与插入操作类似，删除操作也由两个阶段构成(见 13.4 节)。在第一阶段中，当被删除的结点从树中移除时，树发生变化。如果被删除的结点当时有两个孩子，那么它的后继移入被删除结点的位置。这些变化引起 f 的更新传播的代价至多为 $O(\lg n)$，因为这些变化对树的修改是局部的。第二阶段对红黑树的修复至多需要三次旋转，且每次旋转至多需要 $O(\lg n)$的时间就可完成 f 的更新传播。因此，和插入一样，删除的总时间也是 $O(\lg n)$。 ■

在很多情况下，比如维护顺序统计树的 *size* 属性，一次旋转后更新的代价为 $O(1)$，而并不是定理 14.1 中所给出的 $O(\lg n)$。练习 14.2-3 就给出这样的一个例子。

练习

14.2-1 通过为结点增加指针的方式，试说明如何在扩张的顺序统计树上，支持每一动态集合查询操作 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 在最坏时间 $O(1)$ 内完成。顺序统计树上的其他操作的渐近性能不应受影响。

14.2-2 能否在不影响红黑树任何操作的渐近性能的前提下，将结点的黑高作为树中结点的一个属性来维护？说明如何做，如果不能，请说明理由。如何维护结点的深度？

347

***14.2-3** 设$\otimes$为一个满足结合律的二元运算符，a 为红黑树中每个结点上的一个要维护的属性。假设在每个结点 x 上增加一个属性 f，使 $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$，其中 x_1，x_2，…，x_m 是以 x 为根的子树中按中序次序排列的所有结点。说明在一次旋转后，如何在 $O(1)$ 时间内更新 f 属性。对你的扩张稍做修改，使得它能够应用到顺序统计树的 *size* 属性中。

***14.2-4** 希望设计一个操作 RB-ENUMERATE(x，a，b)，来对红黑树进行扩张。该操作输出所有的关键字 k，使得在以 x 为根的红黑树中有 $a \leqslant k \leqslant b$。描述如何在 $\Theta(m + \lg n)$ 时间内实现 RB-ENUMERATE，其中 m 为输出的关键字数目，n 为树中的内部结点数。（提示：不需要向红黑树中增加新的属性。）

14.3 区间树

在这一节里，我们将扩张红黑树来支持由区间构成的动态集合上的一些操作。**闭区间**(closed interval)是一个实数的有序对$[t_1, t_2]$，其中 $t_1 \leqslant t_2$。区间$[t_1, t_2]$表示了集合$\{t \in \mathbf{R}: t_1 \leqslant t \leqslant t_2\}$。**开**(open)区间和**半开**(half-open)区间分别略去了集合的两个或一个端点。在本节中，我们假设区间都是闭的，将结果推广至开和半开区间上是自然和直接的。

区间便于表示占用一连续时间段的一些事件。例如，查询一个由时间区间数据构成的数据库，去找出给定时间区间内发生了什么事件。本节中介绍的数据结构可用来有效地维护这样一个区间数据库。

我们可以把一个区间$[t_1, t_2]$表示成一个对象 i，其中属性 $i.low = t_1$ 为**低端点**(low endpoint)，属性 $i.high = t_2$ 为**高端点**(high endpoint)。我们称区间 i 和 i' **重叠**(overlap)，如果 $i \cap i' \neq \varnothing$，即如果 $i.low \leqslant i'.high$ 且 $i'.low \leqslant i.high$。如图 14-3 所示，任何两个区间 i 和 i' 满足**区间三分律**(interval trichotomy)，即下面三条性质之一成立：

a. i 和 i' 重叠。

b. i 在 i' 的左边(也就是 $i.high < i'.low$)。

c. i 在 i' 的右边(也就是 $i'.high < i.low$)。

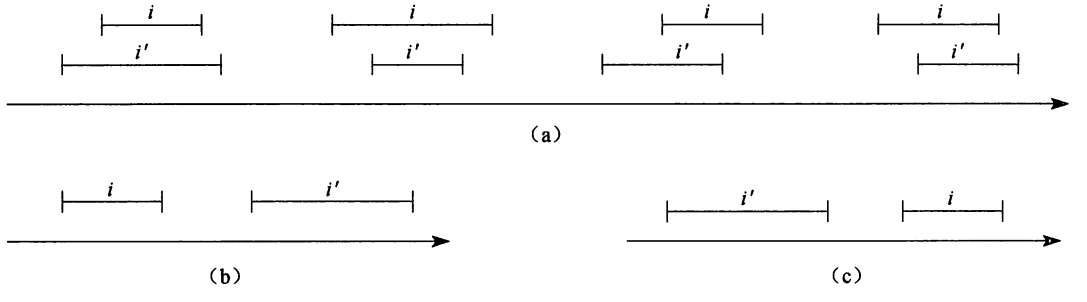

图 14-3 两个闭区间 i 和 i' 的区间三分律。(a)如果 i 和 i' 重叠，又分为 4 种情况；每种情况都有 $i.low \leqslant i'.high$ 且 $i'.low \leqslant i.high$。(b)区间没有重叠且 $i.high < i'.low$。(c)区间没有重叠且 $i'.high < i.low$

区间树(interval tree)是一种对动态集合进行维护的红黑树，其中每个元素 x 都包含一个区

间 $x.int$。区间树支持下列操作：

INTERVAL-INSERT(T，x)：将包含区间属性 int 的元素 x 插入到区间树 T 中。

INTERVAL-DELETE(T，x)：从区间树 T 中删除元素 x。

INTERVAL-SEARCH(T，i)：返回一个指向区间树 T 中元素 x 的指针，使 $x.int$ 与 i 重叠；若此元素不存在，则返回 $T.nil$。

图 14-4 说明了区间树是如何表达一个区间集合的。我们将按照 14.2 节中的 4 步法，来分析区间树以及区间树上各种操作的设计。

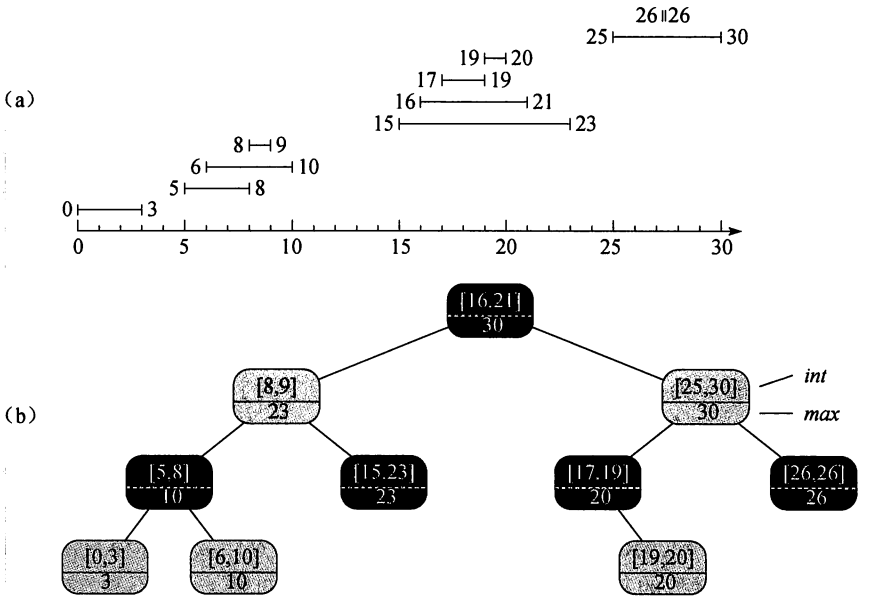

图 14-4　一棵区间树。(a)10 个区间的集合，它们按左端点自底向上顺序示出。(b)表示它们的区间树。每个结点 x 包含一个区间，显示在虚线的上方；一个以 x 为根的子树中所包含的区间端点的最大值，显示在虚线的下方。这棵树的中序遍历列出按左端点顺序排列的各个结点

步骤 1：基础数据结构

我们选择这样一棵红黑树，其每个结点 x 包含一个区间属性 $x.int$，且 x 的关键字为区间的低端点 $x.int.low$。因此，该数据结构按中序遍历列出的就是按低端点的次序排列的各区间。

步骤 2：附加信息

每个结点 x 中除了自身区间信息之外，还包含一个值 $x.max$，它是以 x 为根的子树中所有区间的端点的最大值。

步骤 3：对信息的维护

我们必须验证 n 个结点的区间树上的插入和删除操作能否在 $O(\lg n)$ 时间内完成。通过给定区间 $x.int$ 和结点 x 的子结点的 max 值，可以确定 $x.max$ 值：

$$x.max = \max(x.int.high, x.left.max, x.right.max)$$

这样，根据定理 14.1 可知，插入和删除操作的运行时间为 $O(\lg n)$。事实上，在一次旋转后，更新 max 属性只需 $O(1)$ 的时间，如练习 14.2-3 和练习 14.3-1 所示。

步骤 4：设计新的操作

这里我们仅需要唯一的一个新操作 INTERVAL-SEARCH(T，i)，它是用来找出树 T 中与区间 i 重叠的那个结点。若树中与 i 重叠的结点不存在，则下面过程返回指向哨兵 $T.nil$ 的指针。

348
~
350

```
INTERVAL-SEARCH(T,i)
1  x = T.root
2  while x≠T.nil and i does not overlap x.int
3      if x.left≠T.nil and x.left.max≥i.low
4          x = x.left
5      else x = x.right
6  return x
```

查找与 i 重叠的区间 x 的过程从以 x 为根的树根开始，逐步向下搜索。当找到一个重叠区间或者 x 指向 $T.nil$ 时过程结束。由于基本循环的每次迭代耗费 $O(1)$ 的时间，又因为 n 个结点的红黑树的高度为 $O(\lg n)$，所以 INTERVAL-SEARCH 过程耗费 $O(\lg n)$ 的时间。

在说明 INTERVAL-SEARCH 的正确性之前，先来看一下这个过程在图 14-4 所示的区间树上是如何查找的。假设要找一个与区间 $i=[22, 25]$ 重叠的区间。开始时 x 为根结点，它包含区间[16，21]，与 i 不重叠。由于 $x.left.max=23$ 大于 $i.low=22$，所以这时以这棵树根的左孩子作为 x 继续循环。现在结点 x 包含区间[8，9]，仍不与 i 重叠。此时，$x.left.max=10$ 小于 $i.low=22$，因此以 x 的右孩子作为新的 x 继续循环。现在，由于结点 x 所包含的区间[15，23]与 i 重叠，过程结束并返回这个结点。

现在来看一个查找不成功的例子。假设要在图 14-4 所示的区间树中找出与 $i=[11, 14]$ 重叠的区间。再一次，开始时 x 为根。因为根包含的区间[16，21]不与 i 重叠，且 $x.left.max=23$ 大于 $i.low=11$，则转向左边包含区间[8，9]的结点。区间[8，9]仍不与 i 重叠，且 $x.left.max=10$ 小于 $i.low=11$，因此我们转向右子树。(注意，其左子树中没有一个区间与 i 重叠。)这时区间[15，23]仍不与 i 重叠，且它的左孩子为 $T.nil$，故向右转，循环结束，返回 $T.nil$。

要明白 INTERVAL-SEARCH 的正确性，我们必须理解为什么该过程只需检查一条由根开始的简单路径即可。该过程的基本思想是在任意结点 x 上，如果 $x.int$ 不与 i 重叠，则查找总是沿着一个安全的方向进行：如果树中包含一个与 i 重叠的区间，则该区间必定会被找到。下面的定理更精确地叙述了这个性质。

定理 14.2 *INTERVAL-SEARCH(T, i)的任意一次执行，或者返回一个其区间与 i 重叠的结点，或者返回 $T.nil$，此时树 T 中没有任何结点的区间与 i 重叠。*

351

证明 当 $x=T.nil$ 或 i 与 $x.int$ 重叠时，第 2～5 行的 **while** 循环终止。后一种情况，过程返回 x，显然是正确的。因此，主要考虑前一种情况，也就是当 $x=T.nil$ 时 **while** 循环终止的情况。

对第 2～5 行的 **while** 循环使用如下的循环不变式：

如果树 T 包含与 i 重叠的区间，那么以 x 为根的子树必包含此区间。

循环不变式使用如下：

初始化：在第一次迭代之前，第 1 行置 x 为 T 的根，循环不变式成立。

保持：在 **while** 循环的每次迭代中，第 4 行或第 5 行被执行。下面将证明循环不变式在这两种情况下都能成立。

如果执行第 5 行，则由于第 3 行的分支条件，有 $x.left=T.nil$ 或 $x.left.max<i.low$。如果 $x.left=T.nil$，则以 $x.left$ 为根的子树显然不包含与 i 重叠的区间，所以置 x 为 $x.right$ 以保持这个不变式。因此，假设 $x.left\neq T.nil$ 且 $x.left.max<i.low$。如图 14-5(a)所示，对 x 左子树的任一区间 i'，都有

$$i'.high \leqslant x.left.max < i.low$$

根据区间三分律，i' 和 i 不重叠。因此，x 的左子树不包含与 i 重叠的任何区间，置 x 为 $x.right$ 使循环不变式保持成立。

另外，如果是第 4 行被执行，我们将证明循环不变式的对等情况。也就是说，如果在以 $x.left$ 为根的子树中没有与 i 重叠的区间，则树的其他部分也不会包含与 i 重叠的区间。因为第 4 行被执行，是由于第 3 行的分支条件导致的，所以有 $x.left.max \geqslant i.low$。根据 max 属性的定义，在 x 的左子树中必定存在某区间 i'，满足：

$$i'.high = x.left.max \geqslant i.low$$

(图 14-5(b)显示了这种情况。)因为 i 和 i' 不重叠，又因为 $i'.high < i.low$ 不成立，所以根据区间三分律有 $i.high < i'.low$。区间树是以区间的低端点为关键字的，所以搜索树性质隐含了对 x 右子树中的任意区间 i''，有

$$i.high < i'.low \leqslant i''.low$$

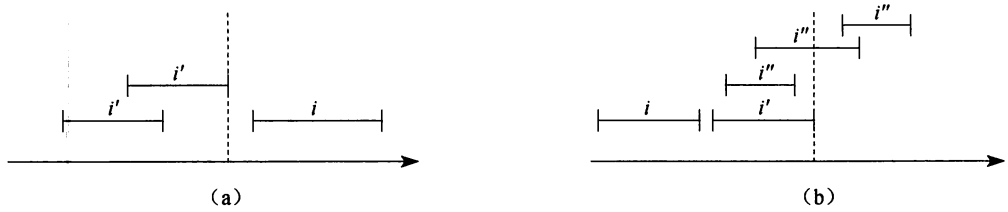

图 14-5　在定理 14.2 的证明中用到的各个区间。在每种情况下，$x.left.max$ 的值用虚线表示。(a)向右查找。在 x 的左子树中没有与之重叠的区间 i'。(b)向左查找。x 的左子树中包含与 i 重叠的区间(此状态未显示)，或者 x 左子树中有一个区间 i'，满足 $i'.high = x.left.max$。既然 i 与 i' 不重叠，则与 x 右子树任意区间 i'' 都不重叠，因为 $i'.\text{low} \leqslant i''.\text{low}$

根据区间三分律，i 和 i'' 不重叠。我们得出这样的结论，即不管 x 的左子树中是否存在与 i 重叠的区间，置 x 为 $x.left$ 保持循环不变式成立。

终止：如果循环在 $x = T.nil$ 时终止，则表明在以 x 为根的子树中，没有与 i 重叠的区间。循环不变式的对等情况说明了 T 中不包含与 i 重叠的区间，故返回 $x = T.nil$ 是正确的。 ■

因此，过程 INTERVAL-SEARCH 是正确的。

练习

14.3-1　写出作用于区间树的结点且在 $O(1)$ 时间内更新 max 属性的过程 LEFT-ROTATE 的伪代码。

14.3-2　改写 INTERVAL-SEARCH 的代码，使得当所有区间都是开区间时，它也能正确地工作。

14.3-3　请给出一个有效的算法，对一个给定的区间 i，返回一个与 i 重叠且具有最小低端点的区间；或者当这样的区间不存在时返回 $T.nil$。

352 ~ 353

14.3-4　给定一棵区间树 T 和一个区间 i，请描述如何在 $O(min(n, k\lg n))$ 时间内列出 T 中所有与 i 重叠的区间，其中 k 为输出的区间数。(提示：一种简单的方法是做若干次查询，并且在这些查询操作中修改树，另一种略微复杂点的方法是不对树进行修改。)

14.3-5　对区间树 T 和一个区间 i，请修改有关区间树的过程来支持新的操作 INTERVAL-SEARCH-EXACTLY(T, i)，它返回一个指向 T 中结点 x 的指针，使得 $x.int.low = i.low$ 且 $x.int.high = i.high$；或者，如果 T 不包含这样的区间时返回 $T.nil$。所有的操作(包括 INTERVAL-SEARCH-EXACTLY)对于包含 n 个结点的区间树的运行时间都应为 $O(\lg n)$。

14.3-6　说明如何来维护一个支持操作 MIN-GAP 的一些数的动态集 Q，使得该操作能给出 Q 中两个最接近的数之间的差值。例如，$Q = \{1, 5, 9, 15, 18, 22\}$，则 MIN-GAP 返回 $18 - 15 = 3$，因为 15 和 18 是 Q 中两个最接近的数。要使得操作 INSERT、DELETE、

SEARCH 和 MIN-GAP 尽可能高效，并分析它们的运行时间。

*14.3-7 VLSI 数据库通常将一块集成电路表示成一组矩形，假设每个矩形的边都平行于 x 轴或者 y 轴，这样可以用矩形的最小和最大的 x 轴与 y 轴坐标来表示一个矩形。请给出一个 $O(n \lg n)$ 时间的算法，来确定 n 个这种表示的矩形集合中是否存在两个重叠的矩形。你的算法不一定要输出所有重叠的矩形，但对于一个矩形完全覆盖另一个（即使边界线不相交），一定能给出正确的判断。（提示：移动一条“扫描”线，穿过所有的矩形。）

思考题

14-1 （最大重叠点） 假设我们希望记录一个区间集合的**最大重叠点**（a point of maximum overlap），即被最多数目区间所覆盖的那个点。

354

a. 证明：最大重叠点一定是其中一个区间的端点。

b. 设计一个数据结构，使得它能够有效地支持 INTERVAL-INSERT、INTERVAL-DELETE，以及返回最大重叠点的 FIND-POM 操作。（提示：使红黑树记录所有的端点。左端点关联 +1 值，右端点关联 −1 值，并且给树中的每个结点扩张一个额外信息来维护最大重叠点。）

14-2 （Josephus 排列） 定义 Josephus 问题如下：假设 n 个人围成一个圆圈，给定一个正整数 m 且 $m \leqslant n$。从某个指定的人开始，沿环将遇到的每第 m 个人移出队伍。每个人移出之后，继续沿环数剩下来的人。这个过程直到所有的 n 个人都被移出后结束。每个人移出的次序定义了一个来自整数 1，2，…，n 的 (n, m)-Josephus 排列。例如，(7, 3)-Josephus 排列为 $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$。

a. 假设 m 是常数，描述一个 $O(n)$ 时间的算法，使得对于给定的 n，能够输出 (n, m)-Josephus 排列。

b. 假设 m 不是常数，描述一个 $O(n \lg n)$ 时间的算法，使得对于给定的 n，能够输出 (n, m)-Josephus 排列。

本章注记

在 Preparata 和 Shamos [282] 的书中，描述了出现在 H. Edelsbrunner (1980) 和 E. M. McCreight(1981)所引用文献内的一些区间树。该书详细介绍了一种区间树，给定包含 n 个

355

区间的静态数据库，它能够在 $O(k + \lg n)$ 时间内，列出所有与指定查询区间重叠的 k 个区间。