第 12 章

Introduction to Algorithms, Third Edition

二叉搜索树

搜索树数据结构支持许多动态集合操作，包括 SEARCH、MINIMUM、MAXIMUM、PREDECESSOR、SUCCESSOR、INSERT 和 DELETE 等。因此，我们使用一棵搜索树既可以作为一个字典又可以作为一个优先队列。

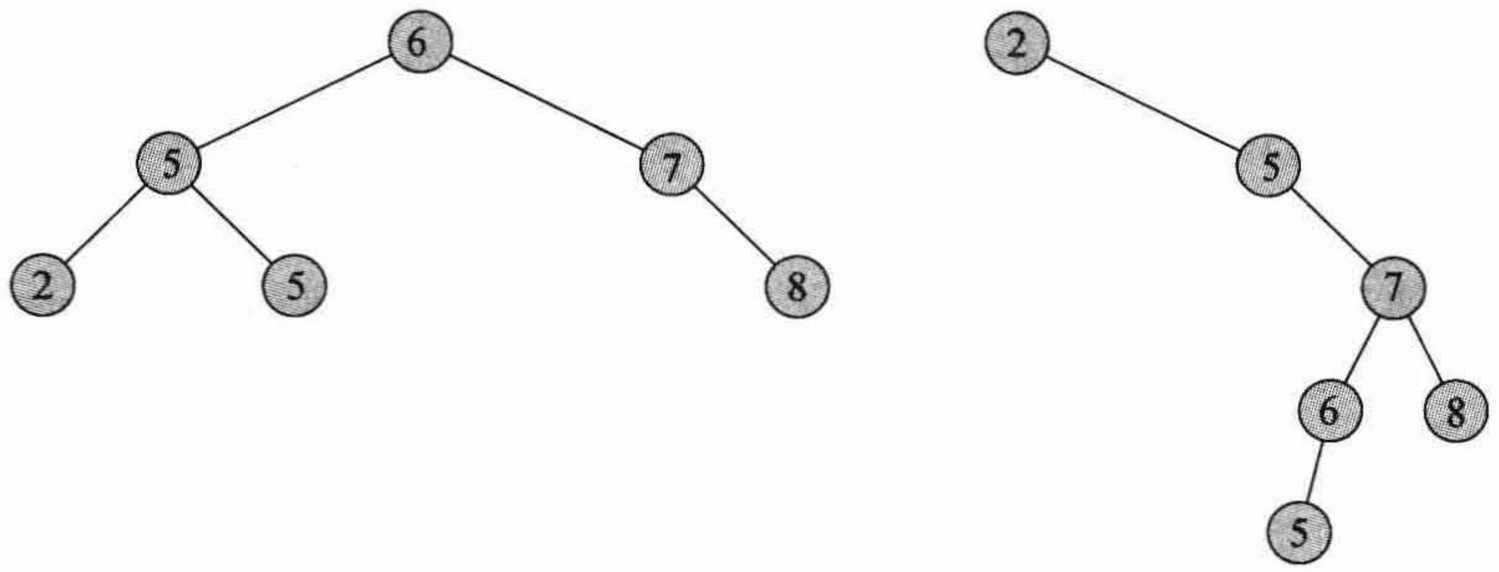
二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比。对于有 n 个结点的一棵完全二叉树来说，这些操作的最坏运行时间为 $\Theta(\lg n)$。然而，如果这棵树是一条 n 个结点组成的线性链，那么同样的操作就要花费 $\Theta(n)$的最坏运行时间。在 12.4 节中，我们将看到一棵随机构造的二叉搜索树的期望高度为 $O(\lg n)$，因此这样一棵树上的动态集合的基本操作的平均运行时间是 $\Theta(\lg n)$。

实际上，我们并不能总是保证随机地构造二叉搜索树，然而可以设计二叉搜索树的变体，来保证基本操作具有好的最坏情况性能。第 13 章给出了一个这样的变形，即红黑树，它的树高为 $O(\lg n)$。第 18 章将介绍 B 树，它特别适用于二级(磁盘)存储器上的数据库维护。

在给出二叉搜索树的基本性质之后，随后几节介绍如何遍历一棵二叉搜索树来按序输出各个值，如何在一棵二叉搜索树上查找一个值，如何查找最小或最大元素，如何查找一个元素的前驱和后继，以及如何对一棵二叉搜索树进行插入和删除。树的这些基本数学性质见附录 B。

12.1 什么是二叉搜索树

顾名思义，一棵二叉搜索树是以一棵二叉树来组织的，如图 12-1 所示。这样一棵树可以使用一个链表数据结构来表示，其中每个结点就是一个对象。除了 *key* 和卫星数据之外，每个结点还包含属性 *left*、*right* 和 p，它们分别指向结点的左孩子、右孩子和双亲。如果某个孩子结点和父结点不存在，则相应属性的值为 NIL。根结点是树中唯一父指针为 NIL 的结点。 286



(a) (b)

图 12-1 二叉搜索树。对任何结点 x，其左子树中的关键字最大不超过 $x.key$，其右子树中的关键字最小不低于 $x.key$。不同的二叉搜索树可以代表同一组值的集合。大部分搜索树操作的最坏运行时间与树的高度成正比。(a)一棵包含 6 个结点、高度为 2 的二叉搜索树。(b)一棵包含相同关键字、高度为 4 的低效二叉搜索树

二叉搜索树中的关键字总是以满足二叉搜索树性质的方式来存储：

设 x 是二叉搜索树中的一个结点。如果 y 是 x 左子树中的一个结点，那么 $y.key \leqslant x.key$。如果 y 是 x 右子树中的一个结点，那么 $y.key \geqslant x.key$。

因此，在图 12-1(a)中，树根的关键字为 6，在其左子树中有关键字 2、5 和 5，它们均不大于 6；而在其右子树中有关键字 7 和 8，它们均不小于 6。这个性质对树中的每个结点都成立。例如，

树根的左孩子为关键字5，不小于其左子树中的关键字2并且不大于其右子树中的关键字5。

二叉搜索树性质允许我们通过一个简单的递归算法来按序输出二叉搜索树中的所有关键字，这种算法称为**中序遍历**(inorder tree walk)算法。这样命名的原因是输出的子树根的关键字位于其左子树的关键字值和右子树的关键字值之间。(类似地，**先序遍历**(preorder tree walk)中输出的根的关键字在其左右子树的关键字值之前，而**后序遍历**(postorder tree walk)输出的根的关键字在其左右子树的关键字值之后。)调用下面的过程 INORDER-TREE-WALK($T.root$)，就可以输
287 出一棵二叉搜索树 T 中的所有元素。

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x, left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

作为一个例子，对于图12-1中的两棵二叉搜索树，中序遍历输出的关键字次序均为2，5，5，6，7，8。根据二叉搜索树性质，可以直接应用归纳法证明该算法的正确性。

遍历一棵有 n 个结点的二叉搜索树需要耗费 $\Theta(n)$ 的时间，因为初次调用之后，对于树中的每个结点这个过程恰好要自己调用两次：一次是它的左孩子，另一次是它的右孩子。下面的定理给出了执行一次中序遍历耗费线性时间的一个证明。

定理12.1 如果 x 是一棵有 n 个结点子树的根，那么调用 INORDER-TREE-WALK(x)需要 $\Theta(n)$ 时间。

证明 当 INORDER-TREE-WALK 作用于一棵有 n 个结点子树的根时，用 $T(n)$ 表示需要的时间。由于 INORDER-TREE-WALK 要访问这棵子树的全部 n 个结点，所以有 $T(n)=\Omega(n)$。下面要证明 $T(n)=O(n)$。

由于对于一棵空树，INORDER-TREE-WALK 需要耗费一个小的常数时间(因为测试 $x\neq$ NIL)，因此对某个常数 $c>0$，有 $T(0)=c$。

对 $n>0$，假设调用 INORDER-TREE-WALK 作用在一个结点 x 上，x 结点左子树有 k 个结点且其右子树有 $n-k-1$ 个结点，则执行 INORDER-TREE-WALK(x)的时间由 $T(n)\leqslant T(k)+T(n-k-1)+d$ 限界，其中常数 $d>0$。此式反映了执行 INORDER-TREE-WALK(x)的一个时间上界，其中不包括递归调用所花费的时间。

使用替换法，通过证明 $T(n)\leqslant(c+d)n+c$，可以证得 $T(n)=O(n)$。对于 $n=0$，有 $(c+d)\cdot 0+c=c=T(0)$。对于 $n>0$，有

$$\begin{aligned}T(n) &\leqslant T(k)+T(n-k-1)+d=((c+d)k+c)+((c+d)(n-k-1)+c)+d\\ &=(c+d)n+c-(c+d)+c+d=(c+d)n+c\end{aligned}$$

288 于是，便完成了定理的证明。 ■

练习

12.1-1 对于关键字集合{1，4，5，10，16，17，21}，分别画出高度为2、3、4、5和6的二叉搜索树。

12.1-2 二叉搜索树性质与最小堆性质(见6.1节)之间有什么不同？能使用最小堆性质在 $O(n)$ 时间内按序输出一棵有 n 个结点树的关键字吗？可以的话，请说明如何做，否则解释理由。

12.1-3 设计一个执行中序遍历的非递归算法。(提示：一种容易的方法是使用栈作为辅助数据结构；另一种较复杂但比较简洁的做法是不使用栈，但要假设能测试两个指针是否相等。)

12.1-4　对于一棵有 n 个结点的树，请设计在 $\Theta(n)$ 时间内完成的先序遍历算法和后序遍历算法。

12.1-5　因为在基于比较的排序模型中，完成 n 个元素的排序，其最坏情况下需要 $\Omega(n \lg n)$ 时间。试证明：任何基于比较的算法从 n 个元素的任意序列中构造一棵二叉搜索树，其最坏情况下需要 $\Omega(n \lg n)$ 的时间。

12.2　查询二叉搜索树

我们经常需要查找一个存储在二叉搜索树中的关键字。除了 SEARCH 操作之外，二叉搜索树还能支持诸如 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 的查询操作。本节将讨论这些操作，并且说明在任何高度为 h 的二叉搜索树上，如何在 $O(h)$ 时间内执行完每个操作。

查找

我们使用下面的过程在一棵二叉搜索树中查找一个具有给定关键字的结点。输入一个指向树根的指针和一个关键字 k，如果这个结点存在，TREE-SEARCH 返回一个指向关键字为 k 的结点的指针；否则返回 NIL。

289

```
TREE-SEARCH(x,k)
1  if x == NIL or k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x,left,k)
5  else return TREE-SEARCH(x.right,k)
```

这个过程从树根开始查找，并沿着这棵树中的一条简单路径向下进行，如图 12-2 所示。对于遇到的每个结点 x，比较关键字 k 与 $x.key$。如果两个关键字相等，查找就终止。如果 k 小于 $x.key$，查找在 x 的左子树中继续，因为二叉搜索树性质蕴涵了 k 不可能被存储在右子树中。对称地，如果 k 大于 $x.key$，查找在右子树中继续。从树根开始递归期间遇到的结点就形成了一条向下的简单路径，所以 TREE-SEARCH 的运行时间为 $O(h)$，其中 h 是这棵树的高度。

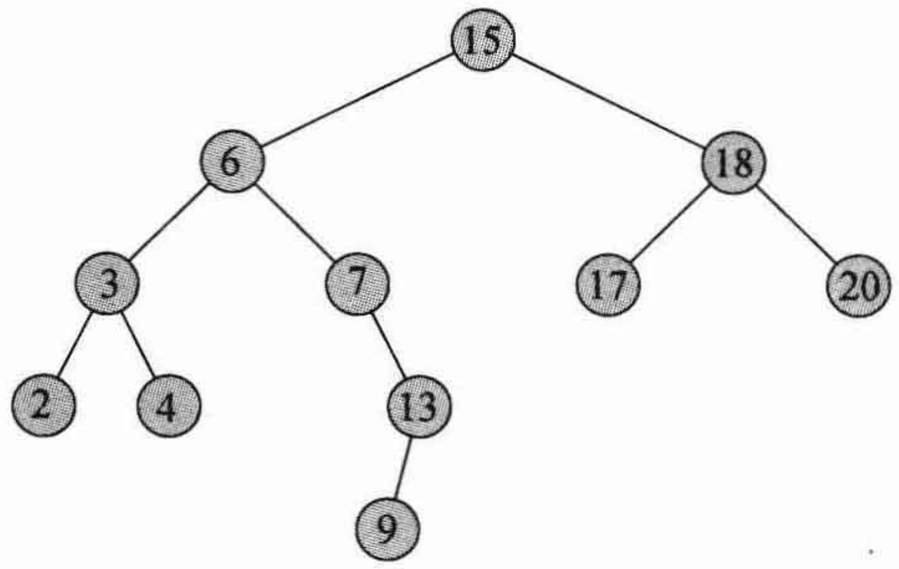


图 12-2　一棵二叉搜索树上的查询。为了查找这棵树中关键字为 13 的结点，从树根开始沿着 15→6→7→13 路径进行查找。这棵树中最小的关键字为 2，它是从树根开始一直沿着 *left* 指针被找到的。最大的关键字 20 是从树根开始一直沿着 *right* 指针被找到的。关键字为 15 的结点的后继是关键字为 17 的结点，因为它是 15 的右子树中的最小关键字。关键字为 13 的结点没有右子树，因此它的后继是最低的祖先并且其左孩子也是一个祖先。这种情况下，关键字为 15 的结点就是它的后继

我们可以采用 **while** 循环来展开递归，用一种迭代方式重写这个过程。对于大多数计算机，迭代版本的效率要高得多。

290

```
ITERATIVE-TREE-SEARCH(x,k)
1  while x ≠ NIL and k ≠ x.key
2       if k < x.key
3            x = x.left
4       else x = x.right
5  return x
```

最大关键字元素和最小关键字元素

通过从树根开始沿着 *left* 孩子指针直到遇到一个 NIL，我们总能在一棵二叉搜索树中找到一个元素，如图 12-2 所示。下面的过程返回了一个指向在以给定结点 x 为根的子树中的最小元素的指针，这里假设不

为 NIL：

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

二叉搜索树性质保证了 TREE-MINIMUM 是正确的。如果结点 x 没有左子树，那么由于 x 右子树中的每个关键字都至少大于或等于 $x.key$，则以 x 为根的子树中的最小关键字是 $x.key$。如果结点 x 有左子树，那么由于其右子树中没有关键字小于 $x.key$，且在左子树中的每个关键字不大于 $x.key$，则以 x 为根的子树中的最小关键字一定在以 $x.left$ 为根的子树中。

TREE-MAXIMUM 的伪代码是对称的，如下：

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

这两个过程在一棵高度为 h 的树上均能在 $O(h)$ 时间内执行完，因为与 TREE-SEARCH 一样，它们所遇到的结点均形成了一条从树根向下的简单路径。

后继和前驱

给定一棵二叉搜索树中的一个结点，有时候需要按中序遍历的次序查找它的后继。如果所
291 有的关键字互不相同，则一个结点 x 的后继是大于 $x.key$ 的最小关键字的结点。一棵二叉搜索树的结构允许我们通过没有任何关键字的比较来确定一个结点的后继。如果后继存在，下面的过程将返回一棵二叉搜索树中的结点 x 的后继；如果 x 是这棵树中的最大关键字，则返回 NIL。

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

把 TREE-SUCCESSOR 的伪代码分为两种情况。如果结点 x 的右子树非空，那么 x 的后继恰是 x 右子树中的最左结点，通过第 2 行中的 TREE-MINIMUM($x.right$)调用可以找到。例如，在图 12-2 中，关键字为 15 的结点的后继是关键字为 17 的结点。

另一方面，正如练习 12.2-6 所要做的，如果结点 x 的右子树非空并有一个后继 y，那么 y 就是 x 的有左孩子的最底层祖先，并且它也是 x 的一个祖先。在图 12-2 中，关键字为 13 的结点的后继是关键字为 15 的结点。为了找到 y，只需简单地从 x 开始沿树而上直到遇到一个其双亲有左孩子的结点。TREE-SUCCESSOR 中的第 3～7 行正是处理这种情况。

在一棵高度为 h 的树上，TREE-SUCCESSOR 的运行时间为 $O(h)$，因为该过程或者遵从一条简单路径沿树向上或者遵从简单路径沿树向下。过程 TREE-PREDECESSOR 与 TREE-SUCCESSOR 是对称的，其运行时间也为 $O(h)$。

即使关键字非全不相同，我们仍然定义任何结点 x 的后继和前驱为分别调用 TREE-SUCCESSOR(x)和 TREE-PREDECESSOR(x)所返回的结点。

总之，我们已经证明了下面的定理。

定理 12.2　*在一棵高度为 h 的二叉搜索树上，动态集合上的操作* SEARCH、MINIMUM、MAXIMUM、SUCCESSOR *和* PREDECESSOR *可以在 $O(h)$ 时间内完成。*

292

练习

12.2-1　假设一棵二叉搜索树中的结点在 1 到 1 000 之间，现在想要查找数值为 363 的结点。下面序列中哪个不是查找过的序列？

a. 2，252，401，398，330，344，397，363。

b. 924，220，911，244，898，258，362，363。

c. 925，202，911，240，912，245，363。

d. 2，399，387，219，266，382，381，278，363。

e. 935，278，347，621，299，392，358，363。

12.2-2　写出 TREE-MINIMUM 和 TREE-MAXIMUM 的递归版本。

12.2-3　写出过程 TREE-PREDECESSOR 的伪代码。

12.2-4　Bunyan 教授认为他发现了一个二叉搜索树的重要性质。假设在一棵二叉搜索树中查找一个关键字 k，查找结束于一个树叶。考虑三个集合：A 为查找路径左边的关键字集合；B 为查找路径上的关键字集合；C 为查找路径右边的关键字集合。Bunyan 教授声称：任何 $a\in A$，$b\in B$ 和 $c\in C$，一定满足 $a\leqslant b\leqslant c$。请给出该教授这个论断的一个最小可能的反例。

12.2-5　证明：如果一棵二叉搜索树中的一个结点有两个孩子，那么它的后继没有左孩子，它的前驱没有右孩子。

12.2-6　考虑一棵二叉搜索树 T，其关键字互不相同。证明：如果 T 中一个结点 x 的右子树为空，且 x 有一个后继 y，那么 y 一定是 x 的最底层祖先，并且其左孩子也是 x 的祖先。（注意到，每个结点都是它自己的祖先。）

12.2-7　对于一棵有 n 个结点的二叉搜索树，有另一种方法来实现中序遍历，先调用 TREE-MINIMUM 找到这棵树中的最小元素，然后再调用 $n-1$ 次的 TREE-SUCCESSOR。证明：该算法的运行时间为 $\Theta(n)$。

293

12.2-8　证明：在一棵高度为 h 的二叉搜索树中，不论从哪个结点开始，k 次连续的 TREE-SUCCESSOR 调用所需时间为 $O(k+h)$。

12.2-9　设 T 是一棵二叉搜索树，其关键字互不相同；设 x 是一个叶结点，y 为其父结点。证明：$y.key$ 或者是 T 树中大于 $x.key$ 的最小关键字，或者是 T 树中小于 $x.key$ 的最大关键字。

12.3　插入和删除

插入和删除操作会引起由二叉搜索树表示的动态集合的变化。一定要修改数据结构来反映这个变化，但修改要保持二叉搜索树性质的成立。正如下面将看到的，插入一个新结点带来的树修改要相对简单些，而删除的处理有些复杂。

插入

要将一个新值 v 插入到一棵二叉搜索树 T 中，需要调用过程 TREE-INSERT。该过程以结点 z 作为输入，其中 $z.key=v$，$z.left=$NIL，$z.right=$NIL。这个过程要修改 T 和 z 的某些属性，来把 z 插入到树中的相应位置上。

```
TREE-INSERT(T,z)
 1  y = NIL
```

```
2  x = T.root
3  while x ≠ 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 == NIL
10     T.root = z            // tree T was empty
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
```

294

图 12-3 显示了 TREE-INSERT 是如何工作的。正如过程 TREE-SEARCH 和 ITERATIVE-TREE-SEARCH 一样，TREE-INSERT 从树根开始，指针 x 记录了一条向下的简单路径，并查找要替换的输入项 z 的 NIL。该过程保持**遍历指针**(trailing pointer) y 作为 x 的双亲。初始化后，第 3～7 行的 **while** 循环使得这两个指针沿树向下移动，向左或向右移动取决于 $z.key$ 和 $x.key$ 的比较，直到 x 变为 NIL。这个 NIL 占据的位置就是输入项 z 要放置的地方。我们需要遍历指针 y，这是因为找到 NIL 时要知道 z 属于哪个结点。第 8～13 行设置相应的指针，使得 z 插入其中。

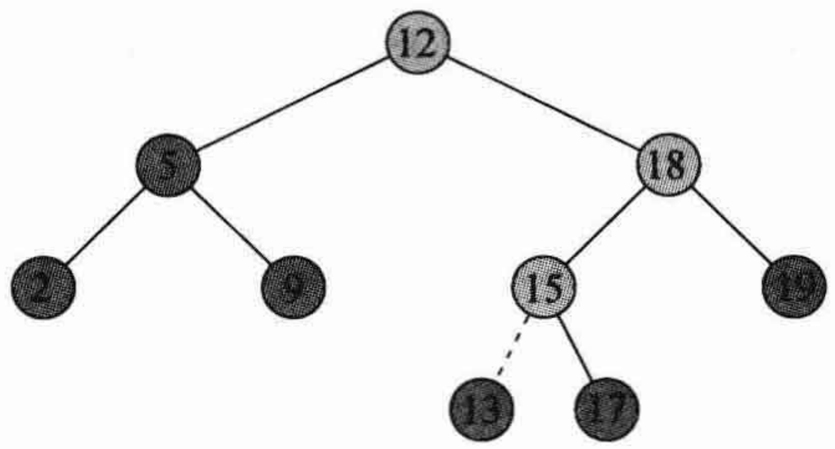


图 12-3 将关键字为 13 的数据项插入到一棵二叉搜索树中。浅阴影结点指示了一条从树根向下到要插入数据项位置处的简单路径。虚线表示了为插入数据项而加入的树中的一条链

与其他搜索树上的原始操作一样，过程 TREE-INSERT 在一棵高度为 h 的树上的运行时间为 $O(h)$。

删除

从一棵二叉搜索树 T 中删除一个结点 z 的整个策略分为三种基本情况(如下所述)，但只有一种情况有点棘手。

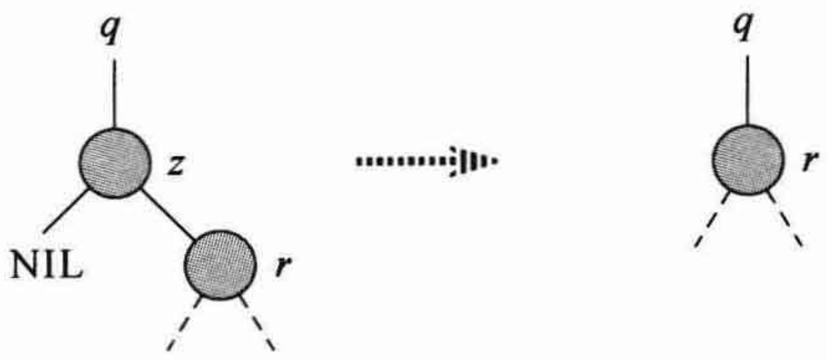
- 如果 z 没有孩子结点，那么只是简单地将它删除，并修改它的父结点，用 NIL 作为孩子来替换 z。
- 如果 z 只有一个孩子，那么将这个孩子提升到树中 z 的位置上，并修改 z 的父结点，用 z 的孩子来替换 z。
- 如果 z 有两个孩子，那么找 z 的后继 y(一定在 z 的右子树中)，并让 y 占据树中 z 的位置。z 的原来右子树部分成为 y 的新的右子树，并且 z 的左子树成为 y 的新的左子树。这种情况稍显麻烦(如下所述)，因为还与 y 是否为 z 的右孩子相关。

295

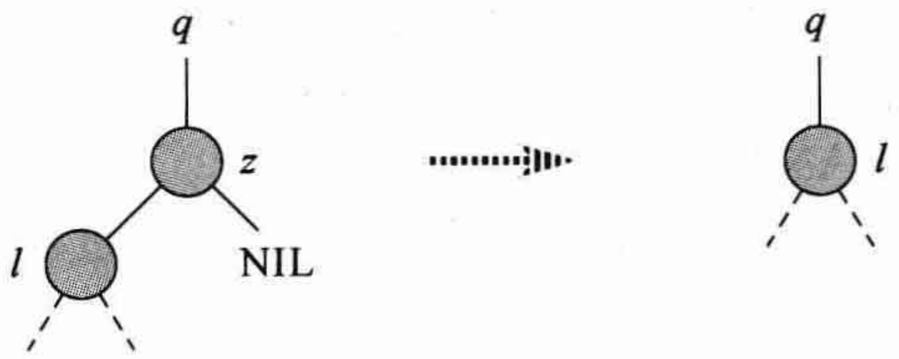
从一棵二叉搜索树 T 中删除一个给定的结点 z，这个过程取指向 T 和 z 的指针作为输入参数。考虑在图 12-4 中显示的 4 种情况，它与前面概括出的三种情况有些不同。

- 如果 z 没有左孩子(图 12-4(a))，那么用其右孩子来替换 z，这个右孩子可以是 NIL，也可以不是。当 z 的右孩子是 NIL 时，此时这种情况归为 z 没有孩子结点的情形。当 z 的右孩子非 NIL 时，这种情况就是 z 仅有一个孩子结点的情形，该孩子是其右孩子。
- 如果 z 仅有一个孩子且为其左孩子(图 12-4(b))，那么用其左孩子来替换 z。
- 否则，z 既有一个左孩子又有一个右孩子。我们要查找 z 的后继 y，这个后继位于 z 的右子树中并且没有左孩子(见练习 12.2-5)。现在需要将 y 移出原来的位置进行拼接，并替换树中的 z。
- 如果 y 是 z 的右孩子(图 12-4(c))，那么用 y 替换 z，并仅留下 y 的右孩子。

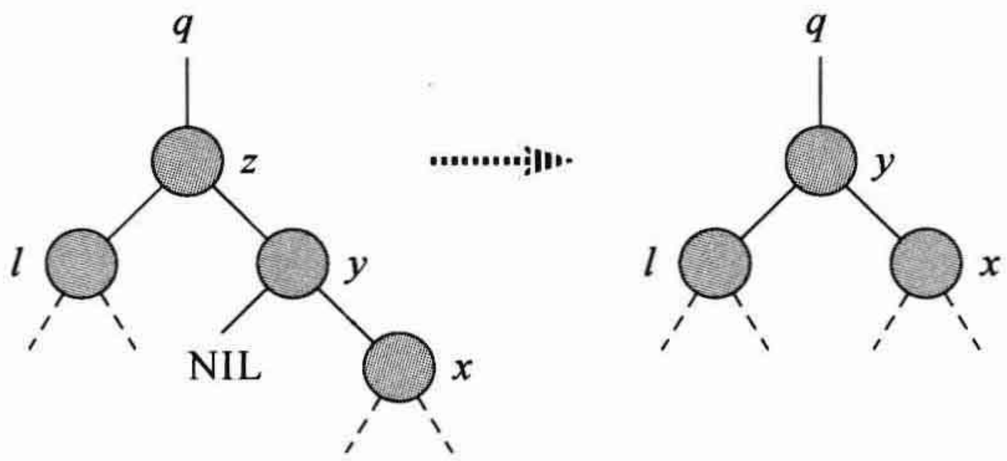
- 否则，y 位于 z 的右子树中但并不是 z 的右孩子(图 12-4(d))。在这种情况下，先用 y 的右孩子替换 y，然后再用 y 替换 z。



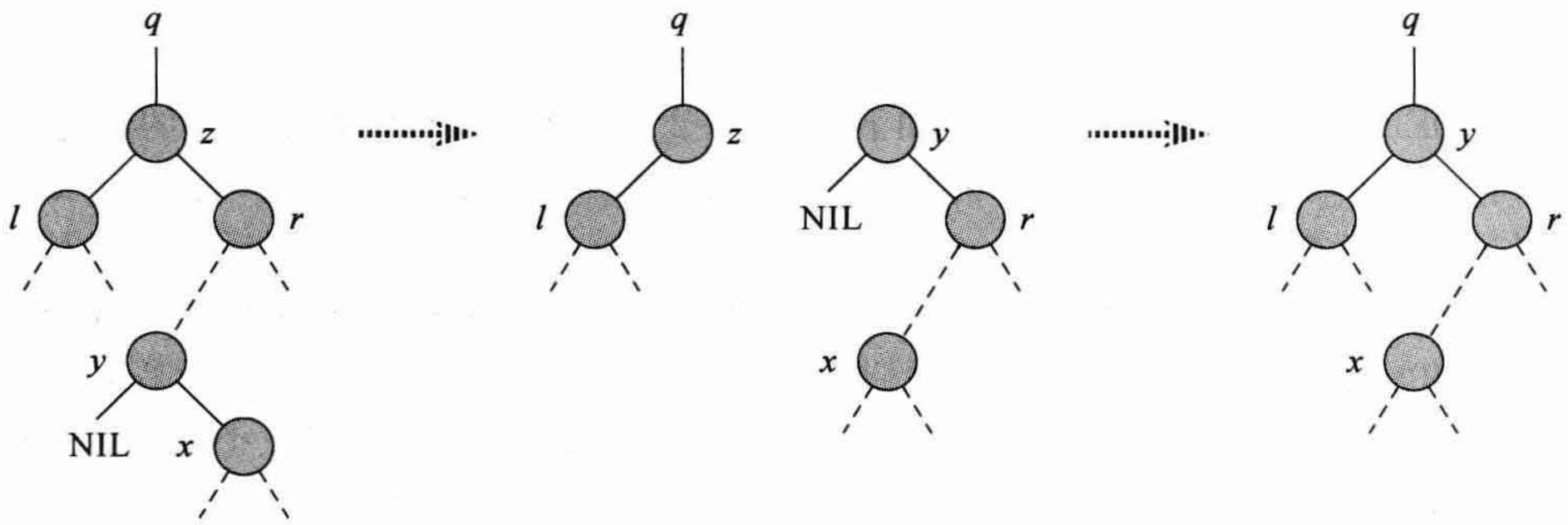
(a)



(b)



(c)



(d)

图 12-4　从一棵二叉搜索树中删除结点 z。结点 z 可以是树根，可以是结点 q 的一个左孩子，也可以是 q 的一个右孩子。(a)结点 z 没有左孩子。用其右孩子 r 来替换 z，其中 r 可以是 NIL，也可以不是。(b)结点 z 有一个左孩子 l 但没有右孩子。用 l 来替换 z。(c)结点 z 有两个孩子，其左孩子是结点 l，其右孩子 y 还是其后继，y 的右孩子是结点 x。用 y 替换 z，修改使 l 成为 y 的左孩子，但保留 x 仍为 y 的右孩子。(d)结点 z 有两个孩子(左孩子 l 和右孩子 r)，并且 z 的后继 $y \neq r$ 位于以 r 为根的子树中。用 y 自己的右孩子 x 来代替 y，并且置 y 为 r 的双亲。然后，再置 y 为 q 的孩子和 l 的双亲

为了在二叉搜索树内移动子树，定义一个子过程 TRANSPLANT，它是用另一棵子树替换一棵子树并成为其双亲的孩子结点。当 TRANSPLANT 用一棵以 v 为根的子树来替换一棵以 u 为根的子树时，结点 u 的双亲就变为结点 v 的双亲，并且最后 v 成为 u 的双亲的相应孩子。

```
TRANSPLANT(T,u,v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p,left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

第1～2行处理u是T的树根的情况。否则，u是其双亲的左孩子或右孩子。如果u是一个左孩子，第3～4行负责$u.p.left$的更新；如果u是一个右孩子，第5行更新$u.p.right$。我们允许v为NIL，如果v为非NIL时，第6～7行更新$v.p$。注意到，TRANSPLANT并没有处理$v.left$和$v.right$的更新；这些更新都由TRANSPLANT的调用者来负责。

296
～
297

利用现成的TRANSPLANT过程，下面是从二叉搜索树T中删除结点z的删除过程：

```
TREE-DELETE(T,z)
 1  if z.left == NIL
 2      TRANSPLANT(T,z,z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T,z,z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T,y,y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T,z,y)
11      y.left = z.left
12      y.left.p = y
```

TREE-DELETE过程处理4种情况如下。第1～2行处理结点z没有左孩子的情况，第3～4行处理z有一个左孩子但没有右孩子情况。第5～12行处理剩下的两种情况，也就是z有两个孩子的情形。第5行查找结点y，它是z的后继。因为z的右子树非空，这样后继一定是这个子树中具有最小关键字的结点，因此就调用TREE-MINIMUM($z.right$)。如前所述，y没有左孩子。将y移出它的原来位置进行拼接，并替换树中的z。如果y是z的右孩子，那么第10～12行用y替换z并成为z的双亲的一个孩子，用z的左孩子替换y的左孩子。如果y不是z的左孩子，第7～9行用y的右孩子替换y并成为y的双亲的一个孩子，然后将z的右孩子转变为y的右孩子，最后第10～12行用y替换z并成为z的双亲的一个孩子，再用z的左孩子替换为y的左孩子。

除了第5行调用TREE-MINIMUM之外，TREE-DELETE的每一行，包括调用TRANSPLANT，都只花费常数时间。因此，在一棵高度为h的树上，TREE-DELETE的运行时间为$O(h)$。

总之，我们证明了下面的定理。

定理12.3 *在一棵高度为h的二叉搜索树上，实现动态集合操作INSERT和DELETE的运行时间均为$O(h)$。* ■

298

练习

12.3-1 给出TREE-INSERT过程的一个递归版本。

12.3-2 假设通过反复向一棵树中插入互不相同的关键字来构造一棵二叉搜索树。证明：在这棵

树中查找关键字所检查过的结点数目等于先前插入这个关键字所检查的结点数目加 1。

12.3-3　对于给定的 n 个数的集合，可以通过先构造包含这些数据的一棵二叉搜索树(反复使用 TREE-INSERT 逐个插入这些数)，然后按中序遍历输出这些数的方法，来对它们排序。这个排序算法的最坏情况运行时间和最好情况运行时间各是多少?

12.3-4　删除操作可交换吗？可交换的含义是，先删除 x 再删除 y 留下的结果树与先删除 y 再删除 x 留下的结果树完全一样。如果是，说明为什么？否则，给出一个反例。

12.3-5　假设为每个结点换一种设计，属性 $x.p$ 指向 x 的双亲，属性 $x.succ$ 指向 x 的后继。试给出使用这种表示法的二叉搜索树 T 上 SEARCH、INSERT 和 DELETE 操作的伪代码。这些伪代码应在 $O(h)$时间内执行完，其中 h 为树 T 的高度。(提示：应该设计一个返回某个结点的双亲的子过程。)

12.3-6　当 TREE-DELETE 中的结点 z 有两个孩子时，应该选择结点 y 作为它的前驱，而不是作为它的后继。如果这样做，对 TREE-DELETE 应该做些什么必要的修改？一些人提出了一个公平策略，为前驱和后继赋予相等的优先级，这样得到了较好的实验性能。如何对 TREE-DELETE 进行修改来实现这样一种公平策略?

12.4　随机构建二叉搜索树

我们已经证明了二叉搜索树上的每个基本操作都能在 $O(h)$时间内完成，其中 h 为这棵树的高度。然而，随着元素的插入和删除，二叉搜索树的高度是变化的。例如，如果 n 个关键字 299
按严格递增的次序被插入，则这棵树一定是高度为 $n-1$ 的一条链。另外，练习 B.5-4 说明了 $h \geqslant \lfloor \lg n \rfloor$。和快速排序一样，我们可以证明其平均情形性能更接近于最好情形，而不是最坏情形时的性能。

遗憾的是，当一棵二叉搜索树同时由插入和删除操作生成时，我们对这棵树的平均高度了解的甚少。当树是由插入操作单独生成时，分析就会变得容易得多。因此，我们定义 n 个关键字的一棵**随机构建二叉搜索树**(randomly built binary search tree)为按随机次序插入这些关键字到一棵初始的空树中而生成的树，这里输入关键字的 $n!$ 个排列中的每个都是等可能地出现。(练习 12.4-3 要求读者证明，这个概念与假定每棵含有 n 个关键字的二叉搜索树为等可能的概念不同。)接下来，这里要证明下面的定理。

定理 12.4　一棵有 n 个不同关键字的随机构建二叉搜索树的期望高度为 $O(\lg n)$。

证明　从定义三个随机变量开始，这些随机变量有助于度量一棵随机构建二叉搜索树。用 X_n 表示一棵有 n 个不同关键字的随机构建二叉搜索树的高度，并定义**指数高度**(exponential height)$Y_n = 2^{X_n}$。当构造一棵有 n 个关键字的二叉搜索树时，选择一个关键字作为树根，并设 R_n 为一个随机变量，表示这个关键字在 n 个关键字集合中的**秩**(rank)，即 R_n 代表的是这些关键字排好序后这个关键字应占据的位置。R_n 的值对于集合$\{1, 2, \cdots, n\}$中的任何元素都是等可能的。如果 $R_n = i$，那么根的左子树是一棵有 $i-1$ 个关键字的随机构建二叉搜索树，并且右子树是一棵有 $n-i$ 个关键字的随机构建二叉搜索树。因为二叉树的高度比根的两棵子树较高的那棵子树大 1，因此二叉树的指数高度是根的两棵子树较高的那棵子树的 2 倍。如果 $R_n = i$，则有

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$$

作为基础情况，设 $Y_1 = 1$，因为 1 个结点的树的指数高度是 $2^0 = 1$，为了方便起见，我们定义 $Y_0 = 0$。

接下来，定义指示器随机变量 $Z_{n,1}$，$Z_{n,2}$，$\cdots$，$Z_{n,n}$，其中 $Z_{n,i} = \mathrm{I}\{R_n = i\}$。因为 R_n 对于集合$\{1, 2, \cdots, n\}$中的任何元素都是等可能的，即有 $\Pr\{R_n = i\} = 1/n$，其中 $i = 1, 2, \cdots, n$，所以由引理 5.1，有

$$\mathrm{E}[Z_{n,i}] = 1/n \qquad (12.1)$$ 300

其中 $i=1, 2, \cdots, n$。由于 $Z_{n,i}$ 恰有一个值为 1，其余所有的值为 0，因此有

$$Y_n = \sum_{i=1}^{n} Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))$$

下面将证明 $\mathrm{E}[Y_n]$ 是 n 的一个多项式，由此最终推出 $\mathrm{E}[X_n]=O(\lg n)$。

指示器随机变量 $Z_{n,i}=\mathrm{I}\{R_n=i\}$ 独立于 Y_{i-1} 和 Y_{n-i} 的值。已经选择了 $R_n=i$，左子树(其指数高度是 Y_{i-1})是由 $i-1$ 个关键字随机构建的，其每个元素的秩都小于 i。这棵子树就像任何其他由 $i-1$ 个关键字随机构建的二叉搜索树一样。除了所包含的关键字数目之外，这棵子树的结构完全不受 $R_n=i$ 选择的影响，因此随机变量 Y_{i-1} 和 $Z_{n,i}$ 是独立的。同样，右子树(其指数高度是 Y_{n-i})是由 $n-i$ 个关键字随机构建的，其每个元素的秩都大于 i。它的结构独立于 R_n 的值，因此随机变量 Y_{n-i} 和 $Z_{n,i}$ 是独立的。所以，有

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\Big[\sum_{i=1}^{n} Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))\Big] \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))] && \text{(由期望的线性性质)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(由独立性)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(由式(12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}[\max(Y_{i-1}, Y_{n-i})] && \text{(由式(C.22))} \\
&\leqslant \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(由练习 C.3-4)}
\end{aligned}$$

在上面最后一个和式中，因为 $\mathrm{E}[Y_0]$，$\mathrm{E}[Y_1]$，…，$\mathrm{E}[Y_{n-1}]$ 中每一项均出现两次，一次作为 $\mathrm{E}[Y_{i-1}]$，一次作为 $\mathrm{E}[Y_{n-i}]$，所以有下面的递归式：

301

$$\mathrm{E}[Y_n] \leqslant \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i] \tag{12.2}$$

使用替换法，下面证明对于所有的正整数 n，递归式(12.2)有解

$$\mathrm{E}[Y_n] \leqslant \frac{1}{4}\binom{n+3}{3}$$

在求解过程中，将使用下面的等式：

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4} \tag{12.3}$$

(练习 12.4-1 要求读者去证明这个等式。)

对于基础情况，注意到两个界 $0=Y_0=\mathrm{E}[Y_0] \leqslant (1/4)\binom{3}{3}=1/4$ 和 $1=Y_1=\mathrm{E}[Y_1] \leqslant (1/4)\binom{1+3}{3}=1$ 成立。对于归纳情况，有

$$\begin{aligned}
\mathrm{E}[Y_n] &\leqslant \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i] \leqslant \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4}\binom{i+3}{3} && \text{(由归纳假设)} \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} = \frac{1}{n}\binom{n+3}{4} && \text{(由式(12.3))} \\
&= \frac{1}{n} \cdot \frac{(n+3)!}{4!(n-1)!} = \frac{1}{4} \cdot \frac{(n+3)!}{3!n!} = \frac{1}{4}\binom{n+3}{3}
\end{aligned}$$

虽然我们有了 $\mathrm{E}[Y_n]$界，但最终的目标是要得到 $\mathrm{E}[X_n]$的界。正如练习 12.4-4 要求读者证明的函数 $f(x)=2^x$ 是凸的。因此，应用 Jensen 不等式(C.26)，也就有

$$2^{\mathrm{E}[X_n]} \leqslant \mathrm{E}[2^{X_n}] = \mathrm{E}[Y_n]$$

如下可得：

$$2^{\mathrm{E}[X_n]} \leqslant \frac{1}{4}\binom{n+3}{3} = \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} = \frac{n^3+6n^2+11n+6}{24}$$

302

两边取对数，得到 $\mathrm{E}[X_n]=O(\lg n)$。 ■

练习

12.4-1　证明等式(12.3)。

12.4-2　请描述这样一棵有 n 个结点的二叉搜索树，其树中结点的平均深度为 $\Theta(\lg n)$，但这棵树的高度是 $\omega(\lg n)$。一棵有 n 个结点的二叉搜索树中结点的平均深度为 $\Theta(\lg n)$，给出这棵树高度的一个渐近上界。

12.4-3　说明含有 n 个关键字的随机选择二叉搜索树的概念，这里每一棵 n 个结点的二叉搜索树是等可能地被选择，不同于本节中给出的随机构建二叉搜索树的概念。(提示：当 $n=3$ 时，列出所有的可能。)

12.4-4　证明：函数 $f(x)=2^x$ 是凸的。

***12.4-5**　考虑 RANDOMIZED-QUICKSORT 操作在 n 个互不相同的输入数据的序列上。证明：对于任何常数 $k>0$，$n!$ 种输入排列除了其中的 $O(1/n^k)$ 种之外，运行时间都为 $O(n\lg n)$。

思考题

12-1　(带有相同关键字的二叉搜索树)　相同关键字给二叉搜索树的实现带来了问题。

a. 当用 TREE-INSERT 将 n 个其中带有相同关键字的数据插入到一棵初始为空的二叉搜索树中时，其渐近性能是多少?

建议通过在第 5 行之前测试 $z.key=x.key$ 和在第 11 行之前测试 $z.key=y.key$ 的方法，来对 TREE-INSERT 进行改进。如果相等，根据下面的策略之一来实现。对于每个策略，得到将 n 个其中带有相同关键字的数据插入到一棵初始为空的二叉搜索树中的渐近性能。(对第 5 行描述的策略是比较 z 和 x 的关键字，用于第 11 行的策略是用 y 代替 x。)

303

b. 在结点 x 设置一个布尔标志 $x.b$，并根据 $x.b$ 的值，置 x 为 $x.left$ 或 $x.right$。当插入一个与 x 关键字相同的结点时，每次访问 x 时交替地置 $x.b$ 为 FALSE 或 TRUE。

c. 在 x 处设置一个与 x 关键字相同的结点列表，并将 z 插入到该列表中。

d. 随机地置 x 为 $x.left$ 或 $x.right$。(给出最坏情况性能，并非形式地导出期望运行时间。)

12-2　(基数树)　给定两个串 $a=a_0a_1\cdots a_p$ 和 $b=b_0b_1\cdots b_q$，这里每个 a_i 和 b_j 是以字符集的某种次序出现的，如果下面两种规则之一成立，就称串 a **按字典序小于**(lexicographically less than)串 b：

1. 存在一个整数 j，其中 $0\leqslant j\leqslant \min(p, q)$，使得对所有的 $i=0, 1, \cdots, j-1$，$a_i=b_i$ 成立，且 $a_j<b_j$。
2. $p<q$，且对所有的 $i=0, 1, \cdots, p$，$a_i=b_i$。

例如，如果 a 和 b 是位串，那么 10 100<10 110(由规则 1，取 $j=3$)，10 100<101 000(由规则 2)。这种次序类似于英语字典中使用的排序。

基数树(radix tree)数据结构如图 12-5 所示，这个树存储了位串 1011、10、011、100 和 0。