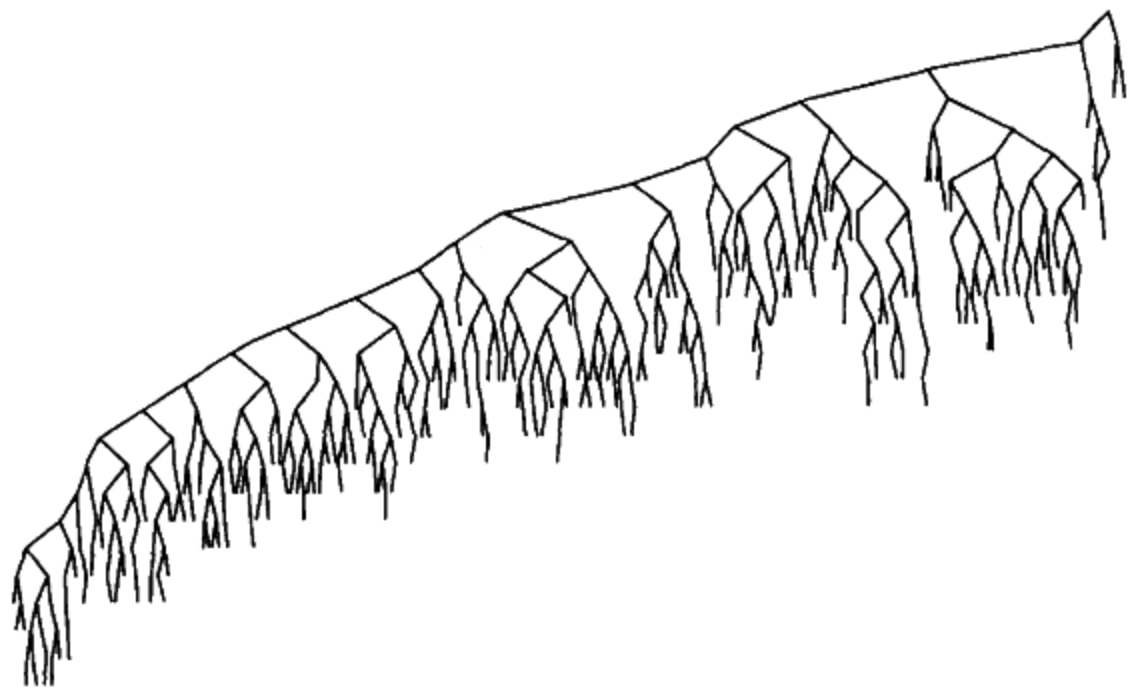

图 4-27 在 $\Theta(N^2)$次 insert/remove 对操作后的二叉查找树

如果向一棵树输入预先排好序的数据，那么一连串 insert 操作将花费二次的时间，而链表实现的代价会非常巨大，因为此时的树将只由那些没有左儿子的节点组成。一种解决办法就是要有一个称为平衡(balance)的附加的结构条件：任何节点的深度均不得过深。

许多一般的算法都能实现平衡树。但是，大部分算法都要比标准的二叉查找树复杂得多，而且更新要平均花费更长的时间。不过，它们确实防止了处理起来非常麻烦的一些简单情形。下面，我们将介绍最古老的一种平衡查找树，即 AVL 树。

另外，较新的方法是放弃平衡条件，允许树有任意的深度，但是在每次操作之后要使用一个调整规则进行调整，使得后面的操作效率要高。这种类型的数据结构一般属于自调整(self-adjusting)类结构。在二叉查找树的情况下，对于任意单个操作我们不再保证 $O(\log N)$的时间界，但是可以证明任意连续 M 次操作在最坏的情形下花费时间 $O(M \log N)$。一般这足以防止令人棘手的最坏情形。我们将要讨论的这种数据结构叫做伸展树(splay tree)；它的分析相当复杂，我们将在第 11 章讨论。

4.4 AVL 树

AVL(Adelson-Velskii 和 Landis)树是**带有平衡条件**(balance condition)的二叉查找树。这个平衡条件必须要容易保持，而且它保证树的深度须是 $O(\log N)$。最简单的想法是要求左右子树具有相同的高度。如图 4-28 所示，这种想法并不强求树的深度要浅。

另一种平衡条件是要求每个节点都必须有相同高度的左子树和右子树。如果空子树的高度定义为 -1(通常就是这么定义)，那么只有具有 $2^k - 1$ 个节点的理想平衡树(perfectly balanced tree)满足这个条件。因此，虽然这种平衡条件保证了树的深度小，但是它太严格而难以使用，需要放宽条件。

一棵 AVL 树是其每个节点的左子树和右子树的高度最多差 1 的二叉查找树(空树的高度定义为 -1)。在图 4-29 中，左边的树是 AVL 树，但是右边的树不是。每一个节点(在其节点结构中)保留高度信息。可以证明，粗略地说，一个 AVL 树的高度最多为 $1.44 \log(N+2) - 1.328$，但是实际上的高度只略大于 $\log N$。作为例子，图 4-30 显示了一棵具有最少节点(143)高度为 9 的 AVL 树。这棵树的左子树是高度为 7 且大小最小的 AVL 树，右子树是高度为 8 且大小最小的 AVL 树。它告诉我们，在高度为 h 的 AVL 树中，最少节点数 $S(h)$由 $S(h) = S(h-1) + S(h-$