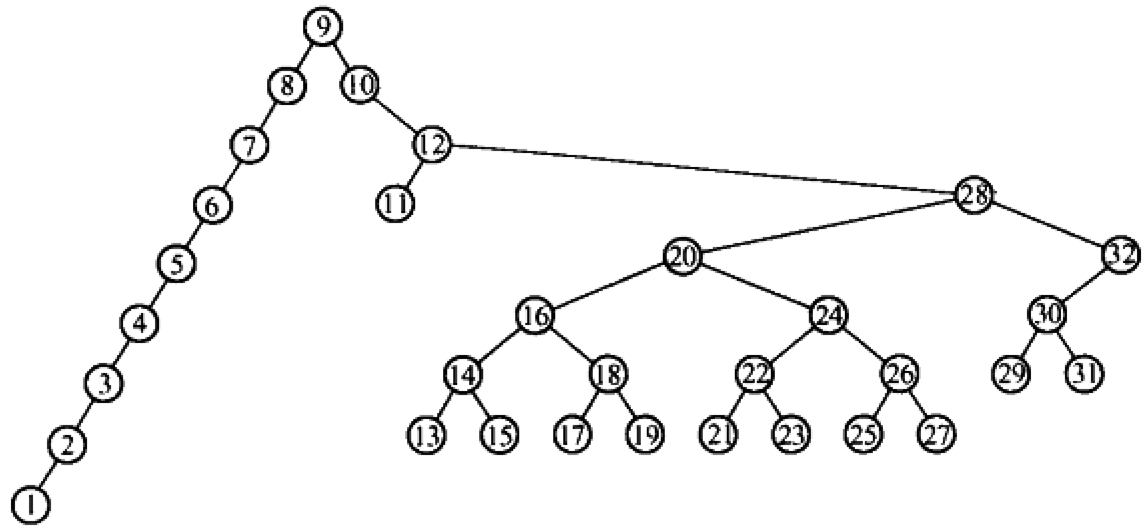

图 4-55 将前面的树在节点 9 处展开

点的树中访问项 1～9 的结果，这棵树最初只含有左儿子。我们从伸展树得不到在简单旋转策略中常见的那种低效率的坏现象(实际上，这个例子只是一种非常好的情况。有一个相当复杂的证明指出，对于这个例子，N 次访问共耗费 $O(N)$的时间)。

这些图着重强调了伸展树基本的和关键的性质。当访问路径长而导致超出正常查找时间的时候，这些旋转将对未来的操作有益。当访问耗时很少的时候，这些旋转则不那么有益甚至有害。极端的情形是经过若干插入而形成的初始树。所有的插入都是导致坏的初始树的花费常数时间的操作。此时，我们会得到一棵很差的树，但是运行却比预计的快，从而总的较少运行时间补偿了损失。这样，少数真正麻烦的访问却留给我们一棵几乎是平衡的树，其代价是必须返还某些已经省下的时间。在第 11 章我们将证明的主要定理指出，平均每个操作决不会落后 $O(\log N)$ 这个时间：我们总是遵守这个时间，即使偶尔有些坏的操作。

可以通过访问要被删除的节点来执行删除操作。这种操作将节点上推到根处。如果删除该节点，则得到两棵子树 T_L 和 T_R(左子树和右子树)。如果我们找到 T_L 中的最大的元素(这很容易)，那么这个元素就被旋转到 T_L 的根下，而此时 T_L 将有一个没有右儿子的根。我们可以使 T_R 为右儿子从而完成删除。

对伸展树的分析很困难，因为必须要考虑树的经常变化的结构。另一方面，伸展树的编程要比 AVL 树简单得多，这是因为要考虑的情形少并且不需要保留平衡信息。一些实际经验指出，在实践中它可以转化成更快的程序代码，不过这种状况离完善还很远。最后，我们指出，伸展树有几种变化，它们在实践中甚至运行得更好。有一种变化在第 12 章中已被完全编成程序。

4.6 树的遍历

由于二叉查找树中对信息进行的排序，因而按照排序的顺序列出所有的项很简单，图 4-56 中的递归方法进行的就是这项工作。

毫无疑问，该方法能够解决将项排序列出的问题。正如我们前面看到的，这类例程当用于树的时候则称为*中序遍历*(由于它依序列出了各项，因此是有意义的)。一个中序遍历的一般方法是首先处理左子树，然后是当前的节点，最后处理右子树。这个算法的有趣部分除它简单的特性外，还在于其总的运行时间是 $O(N)$。这是因为在树的每一个节点处进行的工作是常数时间的。每一个节点访问一次，而在每一个节点进行的工作是检测是否 `null`、建立两个方法调用、并执行 `println`。由于在每个节点的工作花费常数时间以及总共有 N 个节点，因此运行时间为 $O(N)$。

有时我们需要先处理两棵子树然后才能处理当前节点。例如，为了计算一个节点的高度，首先需要知道它的子树的高度。图 4-57 中的程序就是计算高度的。由于检查一些特殊的情况总是