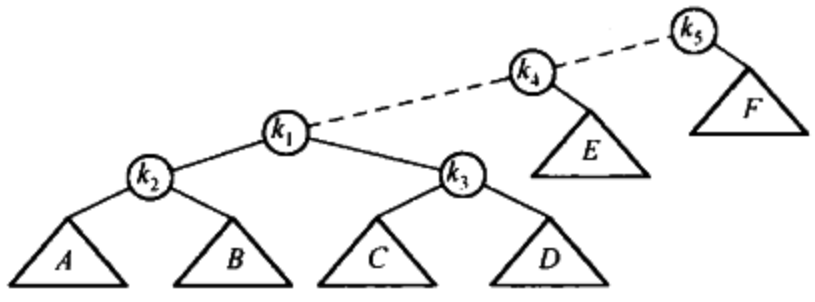
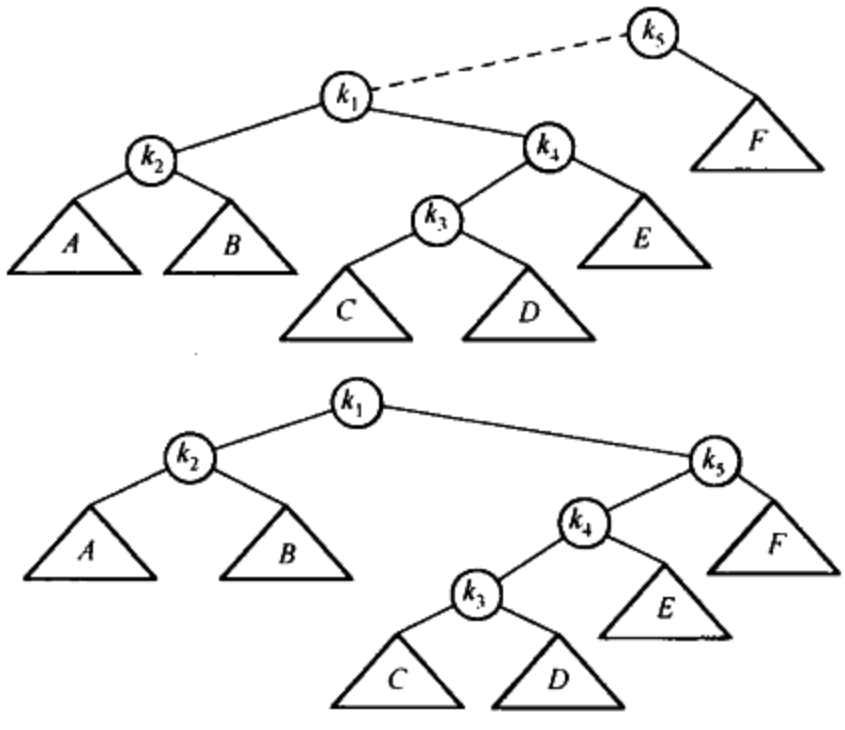


此后，再实行两次旋转直到 k_1 到达树根。



这些旋转的效果是将 k_1 一直推向树根，使得对 k_1 的进一步访问很容易(暂时的)。不足的是它把另外一个节点(k_3)几乎推向和 k_1 以前那么深。而对那个节点的访问又将把另外的节点向深处推进，如此等等。虽然这个策略使得对 k_1 的访问花费时间减少，但是它并没有明显改善(原先)访问路径上其他节点的状况。事实上可以证明，使用这种策略将会存在一系列 M 个操作共需要 $\Omega(M \cdot N)$的时间，因此这个想法还不够好。说明这个问题最简单的方法是考虑向初始的空树插入关键字 $1,2,3,\cdots,N$ 所形成的树(请将这个例子算出)。由此得到一棵树，这棵树只由一些左儿子构成。由于建立这棵树总共花费时间为 $O(N)$，因此这未必就有多坏。问题在于访问关键字为 1 的节点花费 $N-1$ 个单元的时间。在一些旋转完成以后，对关键字为 2 的节点的一次访问花费 $N-2$ 个单元的时间。依序访问所有关键字的总时间是 $\sum_{i=1}^{N-1} i = \Omega(N^2)$。在它们都被访问以后，该树转变回原始状态，而且我们可能重复这个访问顺序。

4.5.2 展开

展开(splaying)的思路类似于上面介绍的旋转的想法，不过在旋转如何实施上我们稍微有些选择的余地。我们仍然从底部向上沿着访问路径旋转。令 X 是在访问路径上的一个(非根)节点，我们将在这个路径上实施旋转操作。如果 X 的父节点是树根，那么只要旋转 X 和树根。这就是沿着访问路径上的最后的旋转。否则，X 就有父亲(P)和祖父(G)，存在两种情况以及对称的情形要考虑。第一种情况是之字形(zig-zag)情形(见图 4-44)。这里，X 是右儿子的形式，P 是左儿子的形式(反之亦然)。如果是这种情况，那么我们执行一次就像 AVL 双旋转那样的双旋转。否则，出现另一种一字形(zig-zig)情形：X 和 P 或者都是左儿子，或者其对称的情形，X 和 P 都是右儿子。在这种情况下，我们把图 4-45 左边的树变换成右边的树。