次对树的操作最多花费 $O(M \log N)$时间。虽然这种保证并不排除任意单次操作花费 $O(N)$时间的可能，而且这样的界也不如每次操作最坏情形的界为 $O(\log N)$时那么强，但是实际效果却是一样的：不存在坏的输入序列。一般说来，当 M 次操作的序列总的最坏情形运行时间为 $O(Mf(N))$时，我们就说它的摊还(amortized)运行时间为 $O(f(N))$。因此，一棵伸展树每次操作的摊还代价是 $O(\log N)$。经过一系列的操作，有的操作可能花费时间多一些，有的可能要少一些。

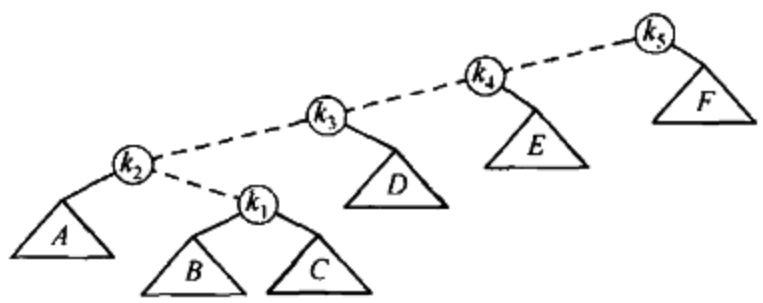
伸展树基于这样的事实：对于二叉查找树来说，每次操作最坏情形时间 $O(N)$并不坏，只要它相对不常发生就行。任何一次访问，即使花费 $O(N)$，仍然可能非常快。二叉查找树的问题在于，虽然一系列访问整体都是坏的操作有可能发生，但是很罕见。此时，累积的运行时间很重要。具有最坏情形运行时间 $O(N)$但保证对任意 M 次连续操作最多花费 $O(M \log N)$运行时间的查找树数据结构确实可以令人满意了，因为不存在坏的操作序列。

如果任意特定操作可以有最坏时间界 $O(N)$，而我们仍然要求一个 $O(\log N)$的摊还时间界，那么很清楚，只要一个节点被访问，它就必须被移动。否则，一旦发现一个深层的节点，我们就有可能不断对它进行访问。如果这个节点不改变位置，而每次访问又花费 $O(N)$，那么 M 次访问将花费 $O(M \cdot N)$的时间。

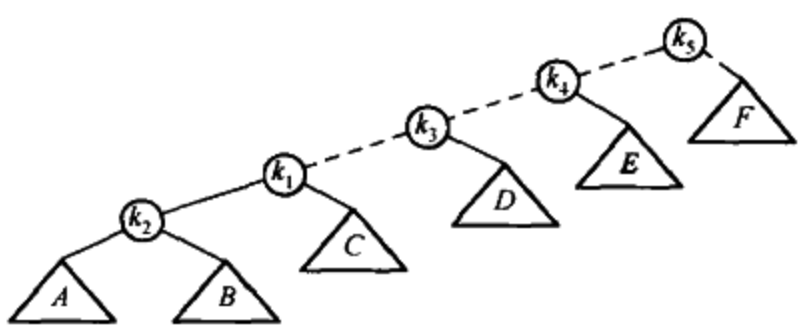
伸展树的基本想法是，当一个节点被访问后，它就要经过一系列 AVL 树的旋转被推到根上。注意，如果一个节点很深，那么在其路径上就存在许多也相对较深的节点，通过重新构造可以减少对所有这些节点的进一步访问所花费的时间。因此，如果节点过深，那么我们要求重新构造应具有平衡这棵树(到某种程度)的作用。除在理论上给出好的时间界外，这种方法还可能有实际的效用，因为在许多应用中当一个节点被访问时，它很可能不久再被访问。研究表明，这种情况的发生比人们预想的要频繁得多。另外，伸展树还不要求保留高度或平衡信息，因此它在某种程度上节省空间并简化代码(特别是当实现例程经过审慎考虑而被写出的时候)。

4.5.1 一个简单的想法(不能直接使用)

实施上面描述的重新构造的一种方法是执行单旋转，从底向上进行。这意味着我们将在访问路径上的每一个节点和它们的父节点实施旋转。作为例子，考虑在下面的树中对 k_1 进行一次访问 (一次 find) 之后所发生的情况。



虚线是访问的路径。首先，我们在 k_1 和它的父节点之间实施一次单旋转，得到下面的树



然后，我们在 k_1 和 k_3 之间旋转，得到下一棵树。