and finally, multiplying through by $2(n + 1)$, proves

> **Theorem 6.2:** The average complexity of a sequence of n insertions into an initially empty binary search tree, assuming that each of the $n!$ distinct permutations of $1, 2, \ldots, n$ is equally likely to be an instance, is
>
> $$A(n) = 2(n + 1)H_n - 4n$$
>
> where $H_n = \sum_{i=1}^{n} 1/i$ is the nth *Harmonic number.* ■

The formula $\sum_{i=1}^{n} 1/i$ cannot be simplified any further, as far as anyone knows, but there is an approximation $H_n \simeq \ln n + \gamma$, where $\gamma \simeq 0.5572$ is Euler's constant. So $A(n)$ is about $1.38n\log_2 n$ for the n insertions, which, loosely speaking, is $O(\log n)$ per insertion — considerably less than the worst case.

This analysis can be extended to include retrievals (Exercise 6.6) and updates (which are $O(1)$, since a pointer to the entry is given). But it tells us nothing about the binary search tree when deletions occur, and in fact almost nothing is known (Knuth, 1973b).

6.6 Splay Trees

In Section 6.3 several heuristics for adjusting a linked list to take advantage of locality of reference in the operation sequence were studied. The most successful one was move-to-front: after accessing entry x, move it to the front of the list. This suggests that the analogous heuristic for binary search trees should be tried: after accessing node x, move it to the root of the tree, where it will be found quickly by subsequent accesses.

A binary search tree is not as simple to adjust as a linked list, because the condition that the key in a node be greater than all the keys in its left subtree and smaller than all the keys in its right subtree must be preserved. (This condition was called the *binary search tree invariant* in Section 6.4.) Nevertheless, there is a way.

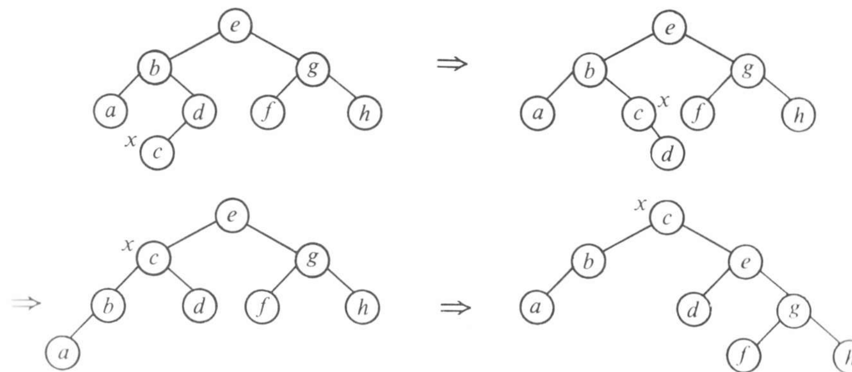
Consider any internal node y that has a left child x which is also internal. A *right rotation at* y adjusts the tree so that y becomes the right child of x:

A, B, and C are arbitrary subtrees, possibly empty. The adjusted tree has the same nodes as the original, and, most important, the binary search tree invariant is preserved. This is easily verified by traversing the two trees in inorder: both give the ordering A, x, B, y, C, so if the invariant holds in the first tree, it holds in the second.

A *left rotation* is similar, going the other way:

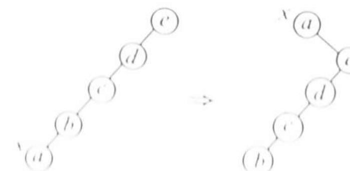
Again, the binary search tree invariant is preserved. These rotations provide a general way to adjust binary trees; they were first used by Adel'son-Vel'skii and Landis (1962).

Rotations are useful here because in each case the depth of node x decreases by 1. Thus, a sequence of rotations at the parent of x will move x to the root. For example,



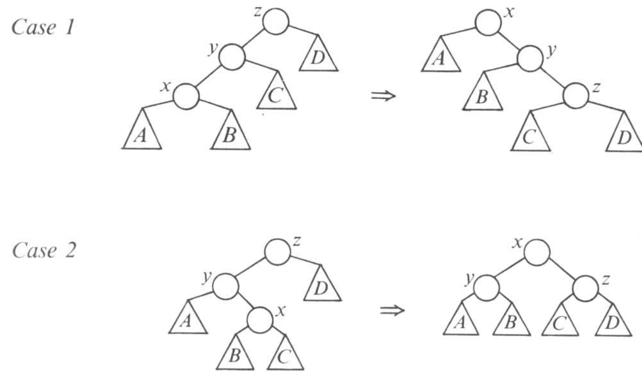
This gives a first heuristic for binary search trees: after accessing (that is, inserting or retrieving) a node x, move it to the root by a sequence of left and right rotations. This heuristic, which is called *move-to-root*, has been studied by Allen and Munro (1978) and by Bitner (1979).

Move-to-root should improve the performance of the binary search tree when there is locality of reference in the operation sequence, but it is not ideal. The final tree in the example above is marginally less balanced than the starting tree, and the example



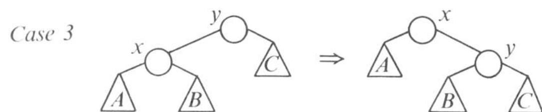
makes it clear that move-to-root will not turn an unbalanced tree into a balanced one.

Sleator and Tarjan (1985b) have found a way to move x to the root and simultaneously clean up an unbalanced tree. They perform the equivalent of two rotations at each *splaying step*:



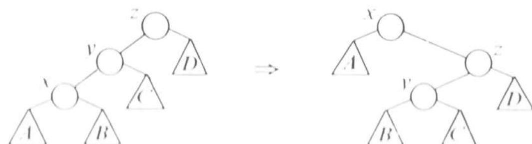
Each case has a symmetric variant, not shown here. Altogether these four cases account for the four possible places that x could occupy as a grandchild of z. It is easy to verify that these transformations preserve the binary search tree invariant.

After these splaying steps have been performed as many times as possible, x will be the root or a child of the root. If x is a child of the root, we perform a final rotation to bring x to the root:

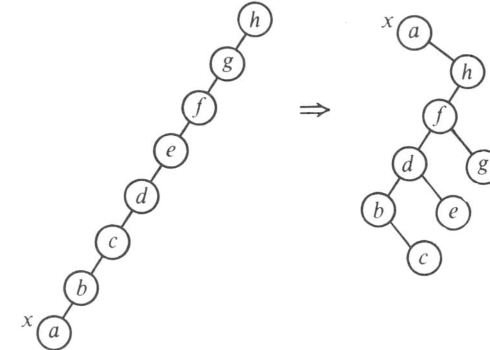


The whole process is called a *splay at x*, and a binary search tree with splaying is called a *splay tree*.

It happens that Cases 2 and 3 do exactly what move-to-root would do in the same situation. However, applying move-to-root to Case 1 would yield



which is not the same. The crucial difference is that, while move-to-root leaves B at its original depth, the splaying step moves both subtrees of x up at least one level. Although subtrees C and D appear to lose out in splaying's Case 1 transformation, they become descendants of x and so move upwards in later splaying steps. Here is a larger example which shows clearly how splaying balances an unbalanced tree:



When implemented carefully, splay trees are very fast in practice (Jones, 1986).

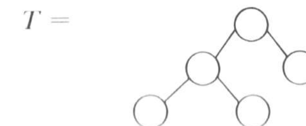
Amortized analysis of splay trees. An insertion or retrieval has two stages: the search down some path, followed by splaying back up the path. The search can be ignored, since its cost is of the same order as the splay, and the number of rotations performed while splaying (or equivalently, the depth of the accessed node) can be taken as the measure of complexity.

The potential function used in this analysis is quite remarkable. Let $s(x)$ be the number of nodes in the subtree rooted at x, and let $r(x)$, the *rank* of x, be defined by $r(x) = \log_2 s(x)$. For any splay tree T, define

$$\Phi(T) = \sum_{x \in T} r(x) = \sum_{x \in T} \log_2 s(x)$$

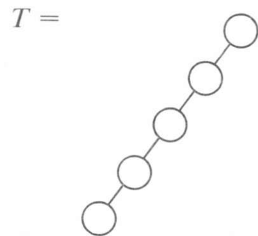
If T is complete, Φ is small. For example,

$T =$



yields $\Phi(T) = \log_2 5 + \log_2 3 + 3\log_2 1 \simeq 3.9$. If T is skew, $\Phi(T)$ is large:

$T =$



yielding $\Phi(T) = \log_2 5 + \log_2 4 + \log_2 3 + \log_2 2 + \log_2 1 \simeq 5.9$. So $\Phi(T)$ is reminiscent of the internal path length $i(T)$, modified to have an $O(n\log n)$ maximum rather than $O(n^2)$. This connection with internal path length is clarified by Exercise 5.9.

As has been seen, splaying at x consists of a number of steps:

1 splaying step at x (Case 1 or 2)

⋮ ⋮

i splaying step at x (Case 1 or 2)

⋮ ⋮

m splaying step at x (Case 1, 2 or 3)

The amortized complexity $a_i = t_i + \Phi(T_i) - \Phi(T_{i-1})$ will be calculated for each step. Then the total amortized complexity of all the steps will equal the amortized complexity of splaying at x, which is taken to be the cost of the insertion or retrieval.

Before starting the main analysis, the following technical lemma is required.

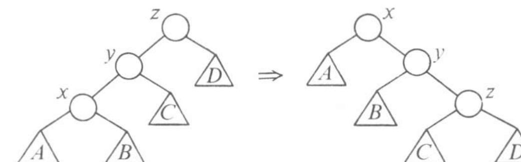
Lemma: For all α and β such that $\alpha > 0$, $\beta > 0$, and $\alpha + \beta \le 1$, $\log_2\alpha + \log_2\beta \le -2$.

Proof: Since $\log_2\alpha + \log_2\beta = \log_2\alpha\beta$, and the logarithm is a monotone increasing function, its value will be maximum when $\alpha\beta$ is maximum. In the given region, it is clear that this occurs when $\alpha = \beta = 1/2$, when $\log_2\alpha + \log_2\beta = -2$. ■

As will be seen, this -2 is used to cancel out the actual complexity of 2 per splaying step. The main theorem, which was named the 'Access Lemma' by its inventors, can now be covered.

Theorem 6.3 (Access Lemma for Splay Trees): Suppose that node x has size $s_{i-1}(x)$ and rank $r_{i-1}(x)$ just before the ith splaying step, and that after the step its size and rank are $s_i(x)$ and $r_i(x)$. Then the amortized complexity of the ith splaying step is at most $3r_i(x) - 3r_{i-1}(x)$, unless it is the final step, in which case the amortized complexity is at most $1 + 3r_i(x) - 3r_{i-1}(x)$.

Proof: Consider first the amortized complexity of a Case 1 splaying step:



The actual complexity is two rotations, and only x, y, and z change in size and rank. So

$$
\begin{aligned}
a_i &= t_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 2 + r_i(x) + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) - r_{i-1}(z) \\
&= 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y)
\end{aligned}
$$

This last line follows because $s_{i-1}(z)$, the size of the subtree rooted at z before the step, equals $s_i(x)$, the size of the subtree rooted at x after the step.

Now before the step, y is an ancestor of x, so $r_{i-1}(y) \ge r_{i-1}(x)$. Similarly, after the step, y is a descendant of x, so $r_i(y) \le r_i(x)$. Substituting these values gives

$$a_i \le 2 + r_i(x) + r_i(z) - 2r_{i-1}(x)$$

The lemma is now used to cancel out the actual complexity of 2. Let $\alpha = s_{i-1}(x)/s_i(x)$, and let $\beta = s_i(z)/s_i(x)$. Clearly, $\alpha > 0$ and $\beta > 0$, but we also have

$$\alpha + \beta = (s_{i-1}(x) + s_i(z))/s_i(x) \le 1$$

This follows because, as the diagram above shows, $s_{i-1}(x)$ encompasses A, x, and B; $s_i(z)$ encompasses C, z, and D; and together these contain exactly one node less than $s_i(x)$. Therefore, by the lemma,

$$\log_2(s_{i-1}(x)/s_i(x)) + \log_2(s_i(z)/s_i(x)) \le -2$$

so that

$$r_{i-1}(x) + r_i(z) - 2r_i(x) \le -2$$

and then

$$2r_i(x) - r_{i-1}(x) - r_i(z) - 2 \geq 0.$$

This non-negative quantity can now be added to the expression for a_i above, giving

$$\begin{aligned} a_i &\leq [2 + r_i(x) + r_i(z) - 2r_{i-1}(x)] + [2r_i(x) - r_{i-1}(x) - r_i(z) - 2] \\ &= 3r_i(x) - 3r_{i-1}(x) \end{aligned}$$

and the theorem is proved for Case 1. The other two cases are left as an exercise; Case 2 is similar, and Case 3 is quite simple. ∎

Now the total amortized complexity of an insertion or retrieval is

$$\begin{aligned} \sum_{i=1}^{m} a_i &= \sum_{i=1}^{m-1} a_i + a_m \\ &\leq \sum_{i=1}^{m-1} (3r_i(x) - 3r_{i-1}(x)) + 1 - 3r_m(x) - 3r_{m-1}(x) \\ &= 1 + 3r_m(x) - 3r_0(x) \\ &\leq 1 + 3r_m(x) \\ &= 1 + 3\log_2 n \end{aligned}$$

since x is the root after the final rotation, so that $s_m(x) = n$. Thus the amortized complexity of an insertion or retrieval is $O(\log n)$. Any sequence of m of these splay tree operations will have $O(m\log n)$ worst-case complexity, which is much better than the $O(mn)$ worst-case complexity of the binary search tree.

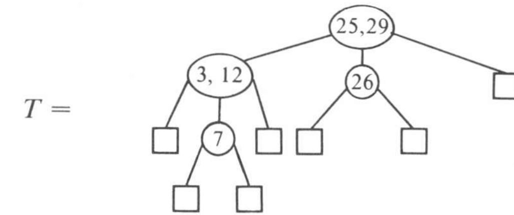
6.7 B-trees

Having studied the binary search treee, which has $O(\log n)$ average complexity per operation, and the splay tree, which has $O(\log n)$ amortized complexity, a data structure of $O(\log n)$ complexity in the worst case, the *B-tree* of Bayer and McCreight (1972), will now be considered.

The B-tree is only one of a large class of tree structures which achieve this performance; the first was the AVL tree of Adel'son-Vel'skii and Landis (1962). The B-tree has been chosen because it is a popular method of implementing ordered symbol tables on disk units – that is, databases – and so is the most widely used of all the methods.

First, the binary search tree is generalized. A *multiway search tree* may have more than one entry in each node. For example,

$T =$



is a multiway search tree. The entries within each node are stored in sorted order. For each gap between two entries, there is a subtree containing all the entries whose keys lie between those two entries. To the left of the entry with the smallest key in the node is a subtree whose entries are all smaller than that entry; similarly there is a subtree containing large entries at the right end. Thus, the number of subtrees of any internal node is one greater than the number of keys in the node.

Searching and traversal in a multiway search tree are simple generalizations of the corresponding algorithms for binary search trees. For example, consider searching for a key x in the tree shown above. If $x < 25$ the search goes left; if $25 < x < 29$ the search goes down; and if $29 < x$ the search goes right. Linear search within the node may be used if there are only a few entries in it, or binary search if there are many. The process is repeated recursively. Traversal in inorder is also quite simple: to traverse a tree T, traverse its first subtree, then visit the first entry of its root, then traverse the second subtree, etc., finishing with a traverse of the last subtree.

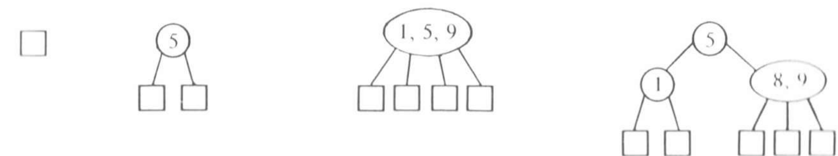
A *B-tree of order m* is a multiway search tree that obeys the following invariant:

(1) The root is either an external node, or it has between 2 and m children inclusive;

(2) Every internal node (except possibly the root) has between $\lceil m/2 \rceil$ and m children inclusive;

(3) The external nodes all have equal depth.

For example, here are some B-trees of order $m = 4$:



The first two conditions make it possible to allocate a fixed amount of memory to each node (space for m links, $m - 1$ entries, and a count field), yet be sure of wasting less than half of it, except possibly in the root.