Insertion and Deletion in AVL Trees

Submitted in Partial Fulfillment of the Requirements for
Dr. Erich Kaltofen's 66621: Analysis of Algorithms
by Robert McCloskey
December 14, 1984[1]

Background

According to Knuth [Knuth73], the AVL version of the binary search tree represents a nice compromise between the "optimum" binary tree, whose height is minimal but for which maintenance is seemingly difficult, and the "arbitrary" binary tree, which requires no special maintenance but whose height could possibly grow to be much greater than minimal.

It accomplishes this by employing a "balanced" (or "admissible" [AVL62]) binary tree, which is defined as one in which each node meets the requirement that the respective heights of its left and right subtrees differ by no more than one.

The height of an admissible tree with $n > 0$ nodes is at least $\lfloor \lg n \rfloor$ (as it is for any binary tree) and, as Adel'son-Vel'skii and Landis [AVL62] show, less than $\log_\phi(n+1)$ (approximately $1.44 \cdot \lg(n+1)$), where ϕ is the "golden ratio" $(1+\sqrt{5})/2$. (For more on the golden ratio, see `http://en.wikipedia.org/wiki/Golden_ratio`.)

To obtain this result, first they computed the minimum number of nodes in an admissible tree of height h. For $h = 0$ and $h = 1$, these values are 1 and 2, respectively. For $h > 1$, an admissible tree having a minimal number of nodes consists of a root node and minimal subtrees of heights $h-1$ and $h-2$. Hence, the desired minimum is given by the recurrence

$$\begin{aligned} M(0) &= 1 \\ M(1) &= 2 \\ M(h) &= M(h-1) + M(h-2) + 1 \quad (h > 1) \end{aligned}$$

which closely resembles the Fibonacci recurrence.[2] The result then follows by induction.

The height of an AVL Tree, then, can be no worse than 50% greater than optimal[3], and so the number of steps required for a search is still proportional to $\lg n$, even in the worst case.

In order to easily maintain the tree's admissibility during the performance of a sequence of insertions and/or deletions, we associate with each node a *balance factor*, which is the difference between the heights of its left and right subtrees, respectively. In an admissible tree, then, each node has a balance factor of either -1, 0, or $+1$.

[1]revised January 2010 and again in January 2012

[2]Indeed, taking $F_0 = 0$ and $F_1 = 1$ as the initial values in the Fibonacci sequence and $F_k = F_{k-1} + F_{k-2}$ $(k > 1)$ as the recurrence, we get $M(h) = F_{h+3} - 1$ for all h.

[3]The largest ratio between actual height and optimum height is realized by a 7-node AVL tree of height three.

The beauty of the AVL method is that it provides a way to easily rebalance a subtree that has become inadmissible —as the result of an insertion or deletion of a node within it— in order to make it admissible again, without sacrificing more than a constant factor in run-time complexity. Consequently, both insertion and deletion require $O(\lg n)$ time.

Node Insertion

Insertion of a node into an AVL Tree proceeds in exactly the same manner as in an arbitrary binary search tree. Once the new node has been put in place, though, additional steps must be taken to update balance factors and to ensure the tree's admissibility. Specifically, the portion of the tree that is affected by an insertion is the subtree whose root A is the last node with a non-zero balance factor lying on what Adel'son-Vel'skii and Landis refer to as the *recorded chain*, which is the path from the root of the tree to the parent of the newly-inserted node. (If all nodes on the recorded chain have a balance factor of zero, A is taken to be the root of the tree.)

The newly-inserted node's balance factor should be set to zero, of course. Meanwhile, each node "below" A on the recorded chain has a subtree whose height has increased by one due to the insertion. (A proof is left to the reader.) Thus, each such node's balance factor should be changed from zero to either $+1$ or -1, respectively, according to whether the new node was put into its left subtree or right subtree.

As for A, it, too, has a subtree whose height has increased. If A's balance factor was zero —implying that it is the root of the tree— its new balance factor is determined in the same way as those of its proper descendants in the recorded chain (as described in the preceding paragraph). In this case, the tree is still admissible. If, on the other hand, A's balance factor was $+1$ or -1, there are two possibilities:

1. The recorded chain leads into what had been A's shorter subtree, in which case A's balance factor should be set to zero. The height of the subtree rooted at A has not been altered, and so the balance factors of A's proper ancestors need not be modified. The tree remains admissible.

2. The recorded chain leads into A's taller subtree, in which case A's balance factor should be set to either $+2$ or -2 (according to whether, respectively, the newly-inserted node is in A's left subtree or right subtree), implying inadmissibility. There are two possible cases, each having a mirror image. Let B be the child of A that is on the recorded chain. In the first case, A's and B's balance factors have the same sign. To make the subtree rooted at A admissible, a single rotation is performed, as illustrated in Figure 1.

   In the second case, A's and B's balance factors have opposite signs. To make the subtree rooted at A admissible, a double rotation is performed, as illustrated in Figure 2. (That figure depicts node C as having balance factor ± 1. However, the same remedy applies if C is itself the newly-inserted node (and hence has balance factor zero), in which case the subtrees labeled α, β, γ and δ in the figure are empty.)

   Note that, in both cases, the resulting subtree's height is the same as what it had been prior to the insertion. Thus, the balance factors of the proper ancestors of the subtree need not be modified.

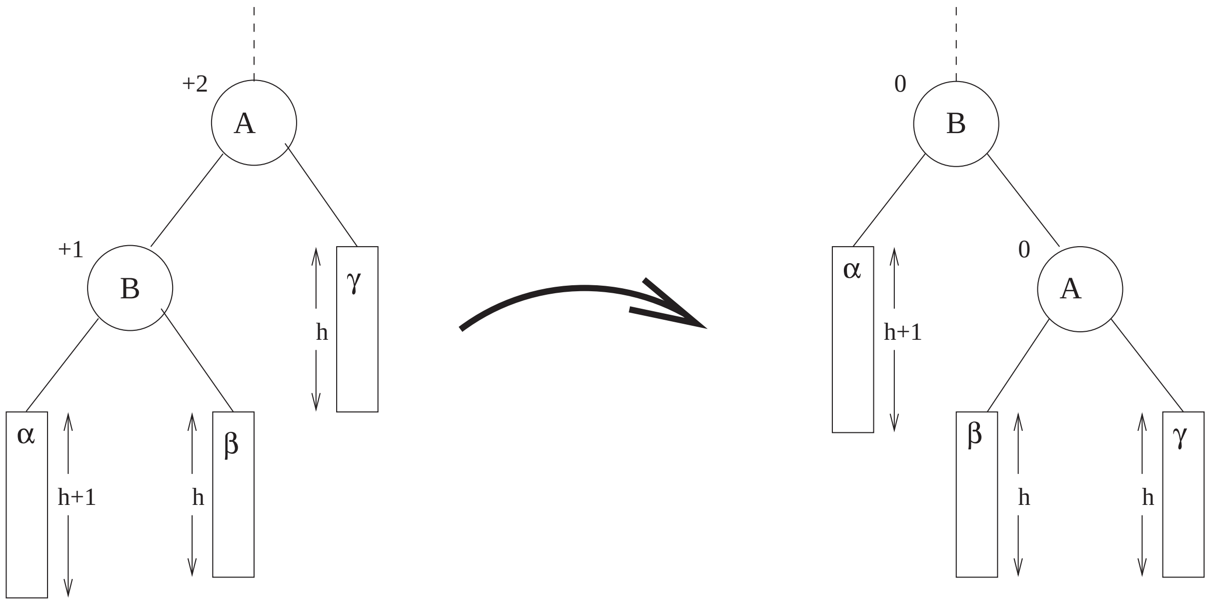

Figure 1: Case I: Rebalancing Using a Single Rotation

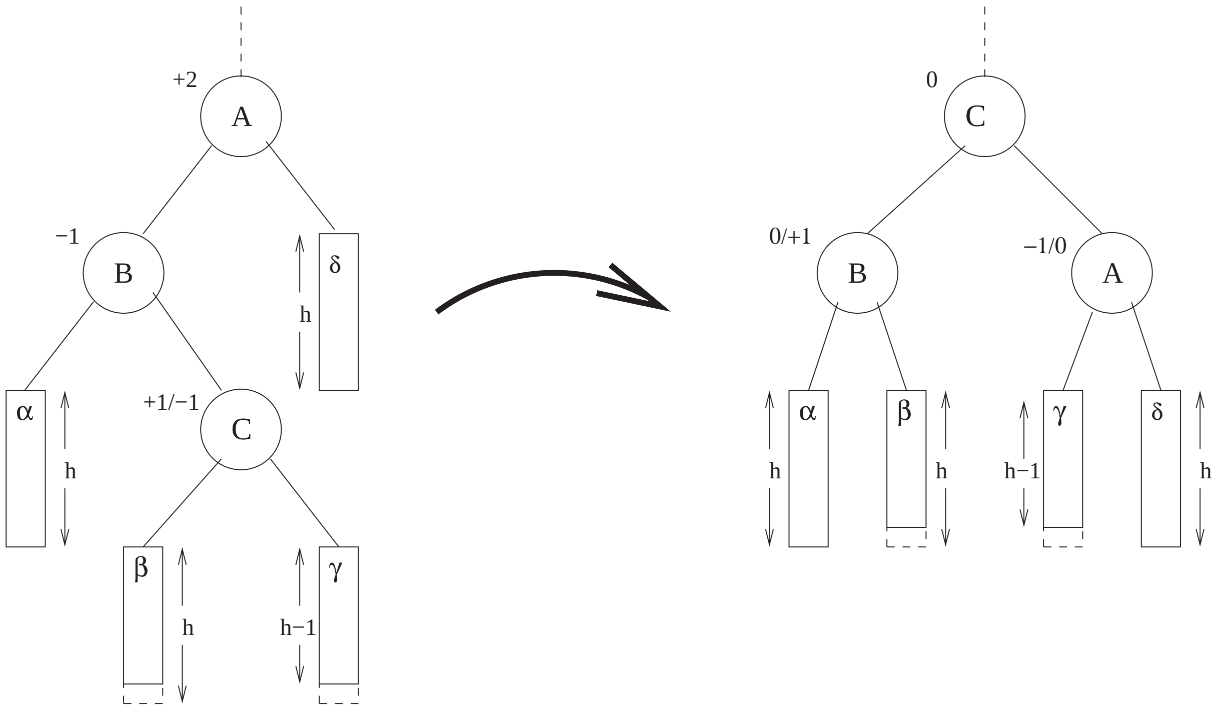

Figure 2: Case II: Rebalancing Using a Double Rotation

Node Deletion
Deletion of a node from an AVL Tree proceeds in exactly the same manner as in an arbitrary binary search tree. (The task of node deletion can always be reduced to that of deleting a node that has at most one child.) As with insertion, additional steps must be taken to maintain balance factors and tree admissibility. With respect to deletion, the recorded chain is the path from the root to the parent of the newly-deleted node. The portion of the tree that may be affected by a node deletion is the subtree whose root D is the last node on the recorded chain having a zero balance factor. (If all nodes on the recorded chain have a non-zero balance factor, D is taken to be the root of the tree.)

The nodes on the recorded chain are considered in order from the "bottom" (i.e., the parent of the deleted node) up to D. Referring to the currently considered node as A, there are three possibilities:

1. A has a balance factor of zero, which is to say that A is D. In this case, the subtree of A in which the deletion occurred has decreased in height by one; hence, A's balance factor should be set to either -1 or $+1$, respectively, according to whether the deletion occurred in the left subtree or the right subtree. Because the height of the subtree rooted at A (i.e., D) is unchanged, the balance factors of A's proper ancestors need not be modified.

2. The recorded chain leads into what had been A's taller subtree, the height of which decreased by one as a result of the deletion. Hence, A's balance factor should be set to zero. Because the height of the tree rooted at A has decreased by one, its parent should be the node to be considered next. (That is, A's parent will play the role of A during the next iteration.)

3. The recorded chain leads into A's shorter subtree, the height of which decreased by one as a result of the deletion. Hence, A's balance factor should be set to -2 or $+2$, respectively, according to whether the deletion occurred in its left subtree or right subtree. This makes the subtree rooted at A inadmissible.

   There are three cases (each having a mirror image), two of which are identical to those that can be produced by an insertion (and therefore are remedied by the methods illustrated in Figures 1 and 2).[4] The third case, shown in Figure 3, is the same as that shown in Figure 1, except that the balance factor of node B is initially zero instead of ± 1.[5]

   What is most interesting is that, in each of the first two cases, the performance of the appropriate rotation(s) results in a decrease in the height of the rebalanced subtree, whereas, in the third case, that subtree's height is unchanged. As a consequence, after doing the appropriate rotation to remedy the third case, nothing else needs to be done. On the other hand, in each of the first two cases, the parent node of the root of the rebalanced

[4]Regarding Case II (Figure 2): In the context of a deletion there is the additional possibility for both of subtrees β and γ to have height h (and thus for node C to have a zero balance factor). In that case, both nodes A and B end up with a zero balance factor after the double rotation.

[5]In the context of a deletion, node B (in all three cases) is either the child of A *opposite* to that on the recorded chain or else the sibling of the deleted node. That is, in Figures 1 and 3 (resp., Figure 2) the deleted node had been in subtree γ (resp., δ).

subtree must be considered next, as its balance factor needs adjustment. Indeed, it is possible, in the worst case, for the deletion of a node to result in a rotation (single or double) occurring at each node on the recoded chain, of which there are $O(\lg n)$.

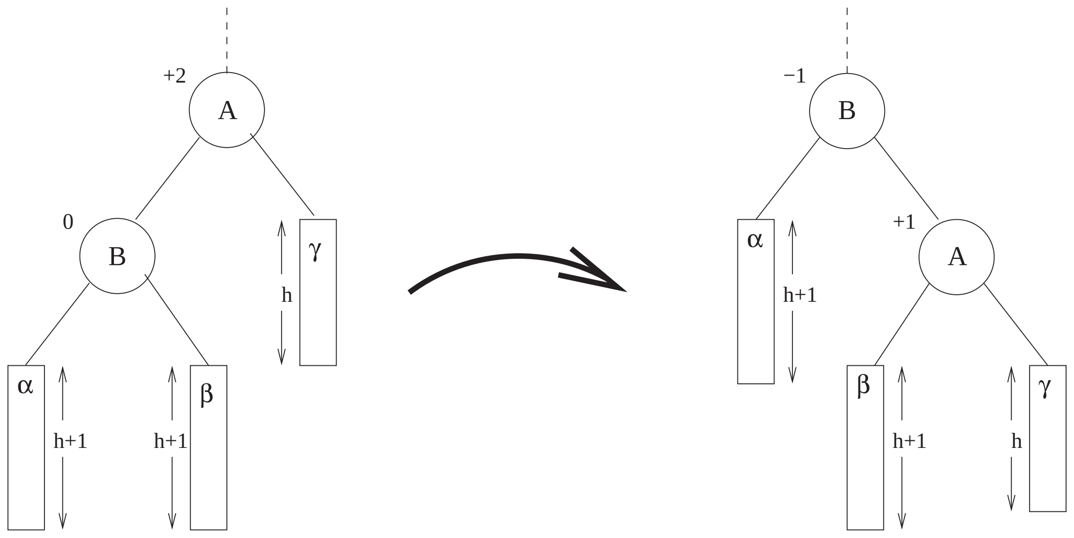

Figure 3: Case III: Rebalancing After a Deletion Using a Single Rotation

Run-time complexity
The major difference between insertion and deletion is that deletion can require up to $O(\lg n)$ rotations, whereas insertion requires at most one. Both operations entail an initial search ($O(\lg n)$ time), a "physical" insertion or deletion of a node ($O(1)$ time), and a modification of the balance factors of at most all the nodes on the recorded chain ($O(\lg n)$ time). Because a rotation can be done in constant time (via a few pointer assignments), the time required by rotations is $O(1)$ during an insertion and $O(\lg n)$ during a deletion. In summary, then, both insertion and deletion take $O(\lg n)$ time.

Original Paper
Interestingly, (the English translation of) the paper in which AVL Trees were first introduced [AVL62] said nothing about node deletion. Moreover, it erroneously omitted the rebalancing case illustrated here in Figure 1 and instead included the case illustrated in Figure 3, which cannot possibly occur during an insertion!

References

[AVL62] Adel'son-Vel'skii, G.M. and Y.M. Landis, *An Algorithm for the Organization of Information*, Soviet Math. Dokl 3 (1962) (English translation), pp. 1259-1262.

[Knuth73] Knuth, Donald E., *Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973.