Computer Science 263/B63 University of Toronto
Design and Analysis of Data Structures

NOTES ON AVL TREES

by Vassos Hadzilacos

Binary search trees work well in the average case, but they still have the drawback of linear worst case time complexity for all three operations (Search, Insert and Delete).

Definition: A binary tree of height h is *ideally height-balanced* if every leaf has depth h or $h-1$, and every node of depth $< h-1$ has two children.

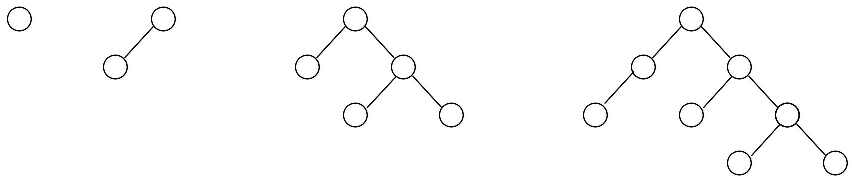
It would be nice if we could keep the binary search tree ideally height-balanced at all times. Then a tree of n nodes would be guaranteed to have height $h = \lfloor \log_2 n \rfloor$, so searches would always take time in $O(\log n)$. But insertions and deletions might destroy the ideally height-balanced property, and a reorganisation (to make the tree ideally height-balanced again, while maintaining the binary search tree property) might take as much as linear time.

AVL (or height-balanced) trees are a happy compromise between arbitrary binary search trees and ideally height-balanced binary search trees. The name "AVL" comes from the names of the two Soviet mathematicians, Adelson-Velski and Landis, who devised them.

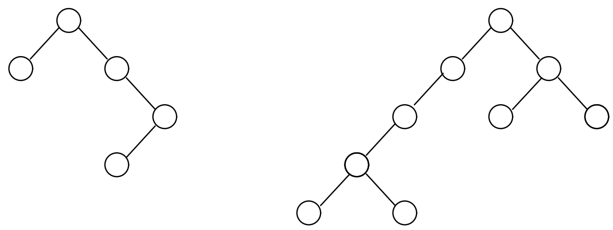
Definition: A binary tree is *height-balanced* if the heights of the left and right subtrees of *every* node differ by at most one. An *AVL tree* is a height-balanced binary search tree.

Note: By convention, the height of an *empty* binary tree (one with 0 nodes) is -1; the height of a tree consisting of a single node is 0.

Examples:



Non-examples:



Good news:

- The worst case height of an AVL tree with n nodes is $1.44 \log_2(n+2)$. Thus, the Search operation can be carried out in $O(\log n)$ time in the worst case.
- Insertions and deletions can also be done in $O(\log n)$ time, while preserving the "AVL-ness" of the tree.
- Empirical studies show that AVL trees work very well on the average case too.

Bad news: The algorithms for insertion and deletion are a bit complex.

Definition: Let h_R and h_L be the heights of the right and left subtrees of a node m in a binary tree respectively. The *balance factor* of m, $BF[m]$, is defined as $BF[m] = h_R - h_L$.

For an AVL tree, the balance factor of any node is -1, 0, or $+1$.

- If $BF[m] = +1$, m is *right heavy.*
- If $BF[m] = -1$, m is *left heavy.*
- If $BF[m] = 0$, m is *balanced.*

In AVL trees we will store $BF[m]$ in each node m. When we draw AVL trees we will put a "+", "−", or "0" next to each node to indicate, respectively, that the node's balance factor is $+1$, -1, or 0.

Next we consider algorithms for the SEARCH, INSERT and DELETE operations in AVL trees.

THE ALGORITHM FOR SEARCH

We simply treat T as an ordinary binary search tree — there is nothing new to say here.

THE ALGORITHM FOR INSERT

To insert a key x into an AVL tree T, let us first insert x in T as in ordinary binary search trees. That is, we trace a path from the root downward, and insert a new node with key x in it in the proper place, so as to preserve the binary search tree property. This may destroy the integrity of our AVL tree in that

- the addition of a new leaf may have destroyed the height-balance of some nodes, and,
- the balance factors of some nodes must be updated to take into account the new leaf.

We will address each of these points in turn.

Rebalancing an AVL tree after Insertion

The height-balance property of a node may have been destroyed as a result of the insertion of the new leaf in two ways:

(1) the new leaf increased the height of the right subtree of a node that was already right heavy (before the insertion); or,

(2) the new leaf increased the height of the left subtree of a node that was already left heavy (before the insertion).

These two cases are illustrated in Figures 1(a) and (b). Note that the insertion of the new leaf can affect the balance factors *only* of its ancestors. To see this, observe that the height of any node that is *not* an ancestor of the new leaf is the same as before the insertion; consequently the heights of the left and right children of such a node are the same as before the insertion. Node m in Figure 1 is assumed to be the *minimum height* ancestor of the new leaf which is no longer height balanced as a result of the insertion.

Since the two cases are symmetric (one is obtained from the other by changing every reference of "right" to "left", and of "+" to "−", and *vice versa*), we shall only consider case (1) in detail. There are two ways in which (1) could arise, illustrated in Figure 2(a) and (b) respectively. The balance factors indicated for A and B are *after* the insertion of the new node.

The subtree shown in Figure 2(a) can be rebalanced by a simple transformation called "single left rotation" on node m. This transformation is illustrated in Figure 3: In 3(a) we copied the subtree of Figure 2(a), and 3(b) shows the result of the single left rotation on that subtree.

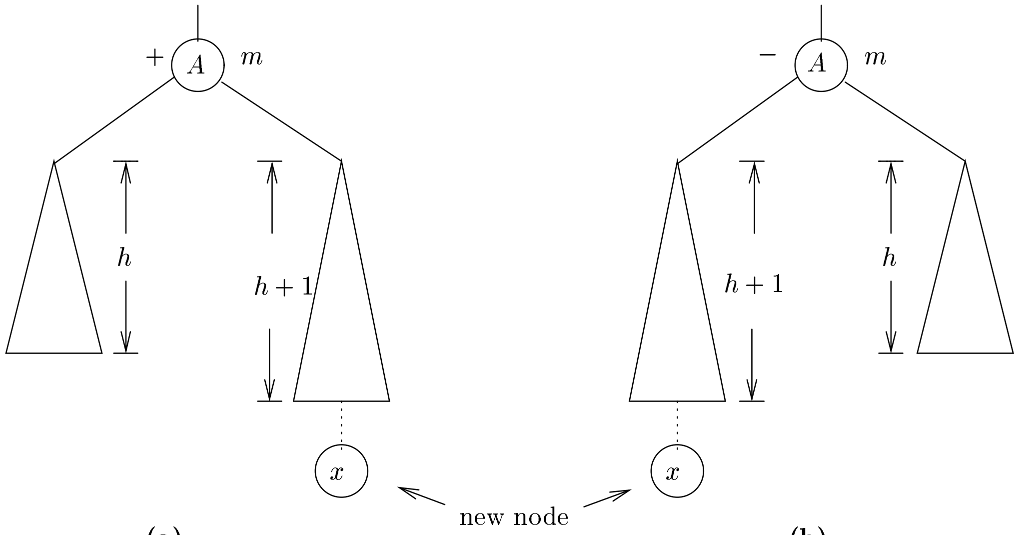
Note that this transformation has the following properties.

S.1 It rebalances the subtree rooted at node m (so that subtree becomes height-balanced again).

S.2 It maintains the binary search tree property.

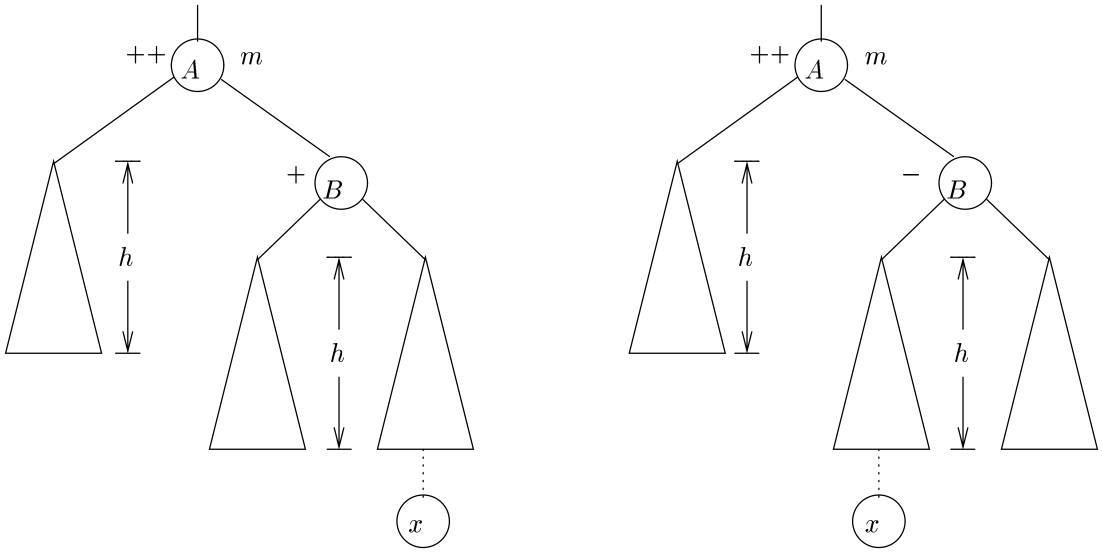
S.3 It can be done in constant time: only a few pointers have to be switched around. As an exercise, write a program that implements this rotation, given a pointer to node m, assuming the standard representation for binary trees.

S.4 It keeps the height of m equal to its height *before* the insertion of the new node, namely, height $h + 2$.



(a) **(b)**

Figure 1



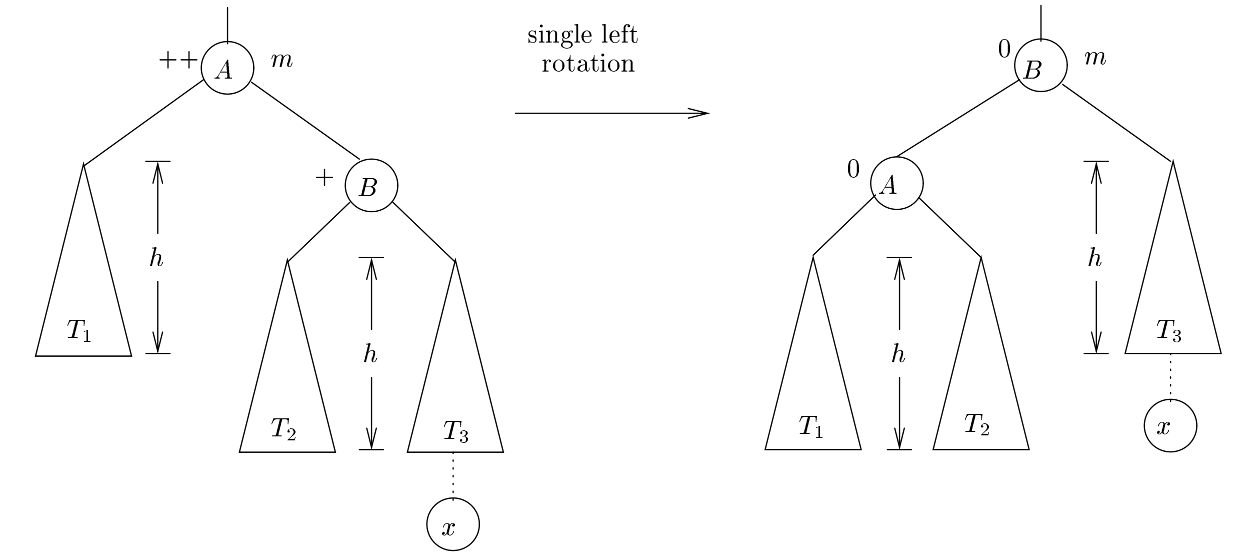
(a) **(b)**

Figure 2

Unfortunately the subtree in Figure 2(b) cannot be rebalanced by a single left rotation. You should check that the subtree resulting from such a transformation is not height-balanced.

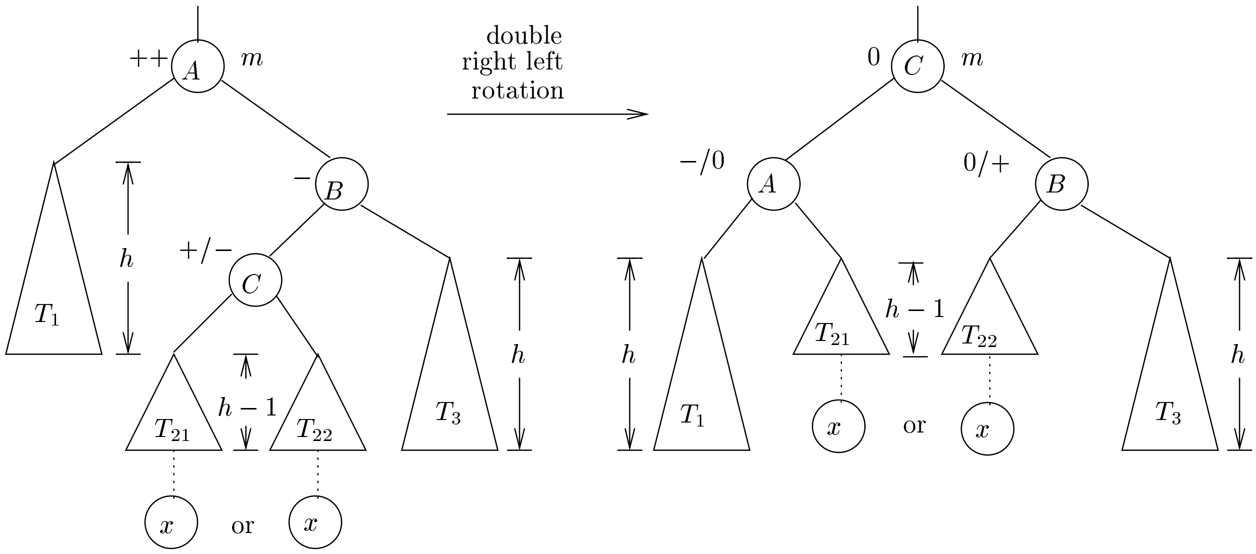
Assume for now that the subtrees of node B in Figure 2(b) are nonempty (i.e., $h \neq -1$). Figure 4(a) shows these subtrees in more detail. This more detailed picture leads to a different way of transforming the subtree into a height-balanced one. This transformation is called a "double right left rotation" and is illustrated in Figure 4(b). The name comes from the fact that this transformation can be obtained if we rotate first B to the right and then, in the resulting subtree, rotate C left. The balance factors labeled as "$*/*$" in Figure 4 depend on whether the new node was actually inserted under T_{22} (the first entry of the label) or under T_{21} (the second entry of the label).

If the subtrees of node B in Figure 2(b) are empty (i.e., $h = -1$) then A has only a right child, B, and B has only a left child, the new node x. The subtree rooted at A is not height-balanced and the situation can be rectified again with a double right-left rotation that makes x the root of the subtree, and A and B its



(a) **(b)**

Figure 3



(a) **(b)**

Figure 4

left and right children, respectively. This case can also be thought of as a degenerate instance of Figures 4(a) and (b), with $C = x$, and subtrees T_1, T_{21}, T_{22} and T_3 all empty.

The double right left rotation has the following properties.

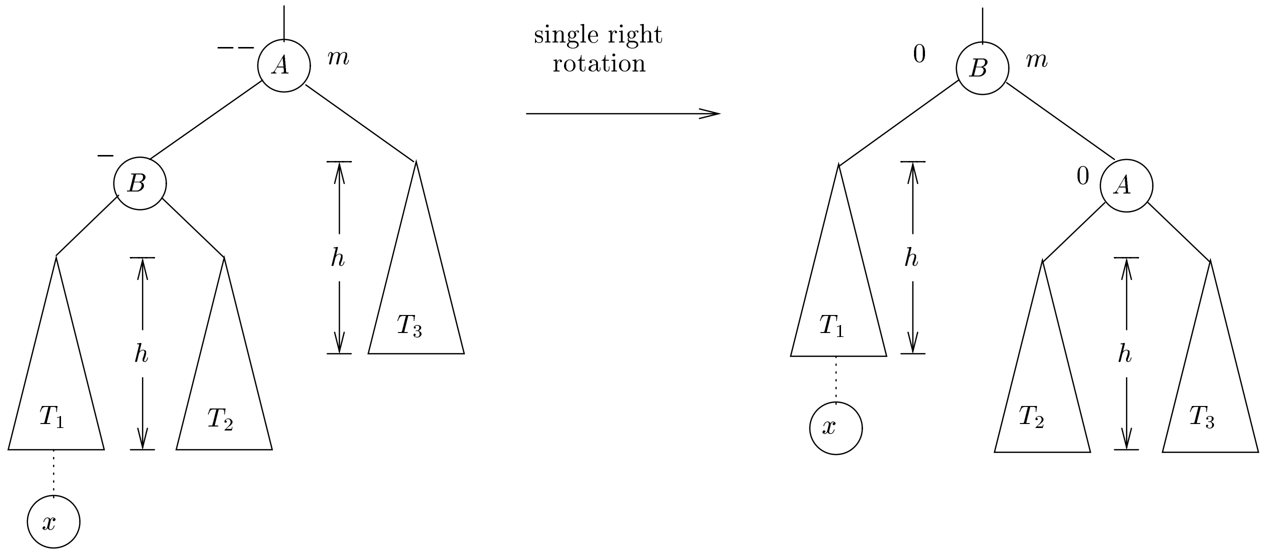
D.1 It rebalances the subtree rooted at m, (so that subtree becomes height-balanced again).

D.2 It maintains the binary search tree property.

D.3 It can be done in constant time: we only have to change a few pointers. As an exercise, write a program that implements this rotation, given a pointer to node m.

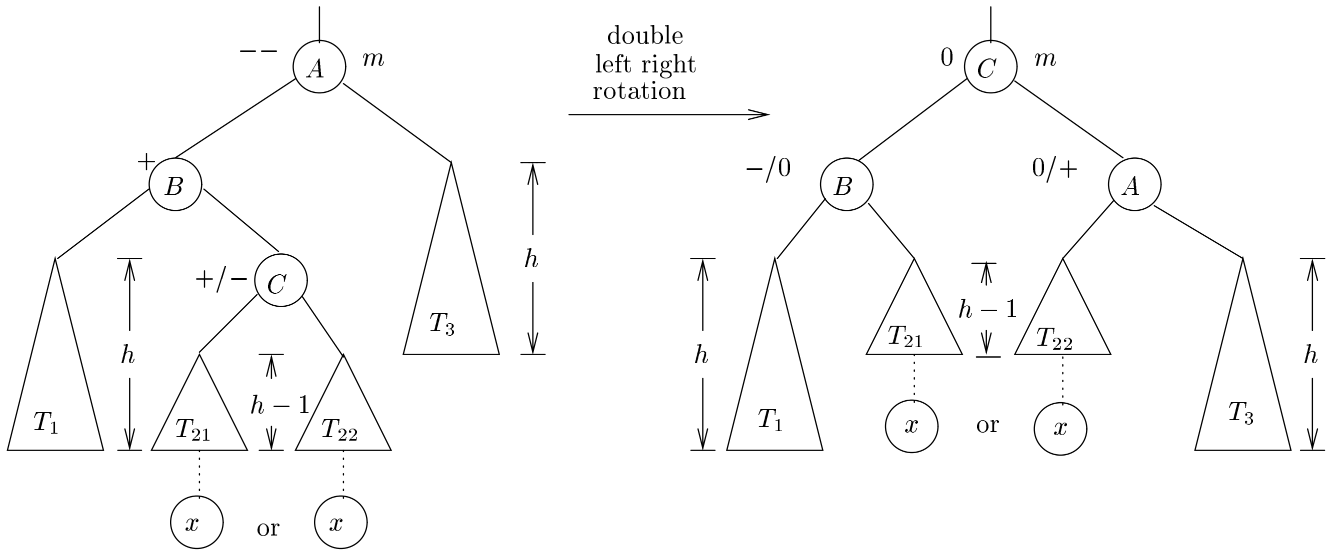
D.4 It keeps the height of m equal to that node's height *before* the insertion of the new node, namely, height $h + 2$.

As we already remarked, the imbalance shown in Figure 1(b) can be fixed in a symmetric way. The two subcases, and the transformations that rebalance the subtrees, called “single right rotation” and “double left right rotation”, are illustrated in Figures 5 and 6 respectively. Remarks analogous to S.1–S.4 and D.1–D.4 apply in these transformations as well.



(a) **(b)**

Figure 5



(a) **(b)**

Figure 6

Updating the Balance Factors after Insertion

The balance factors of some nodes may change as a result of inserting a new node. First of all, observe that only the balance factors of the new node's ancestors may need updating: For any other node i, i's left and right subtrees (and, in particular, their heights) have not changed and thus neither has the balance factor of i. But not *all* of the new node's ancestors' balance factors may need updating. Figure 7 illustrates the issue. Insertion of key 8 to the AVL tree in 7(a) results in the AVL tree in 7(b). Note that only the balance of 9, 8's parent, has changed. On the other hand, insertion of key 8 to the AVL tree in 7(c) results in the AVL tree in 7(d), where the balance factors of all of 9's ancestors have changed.

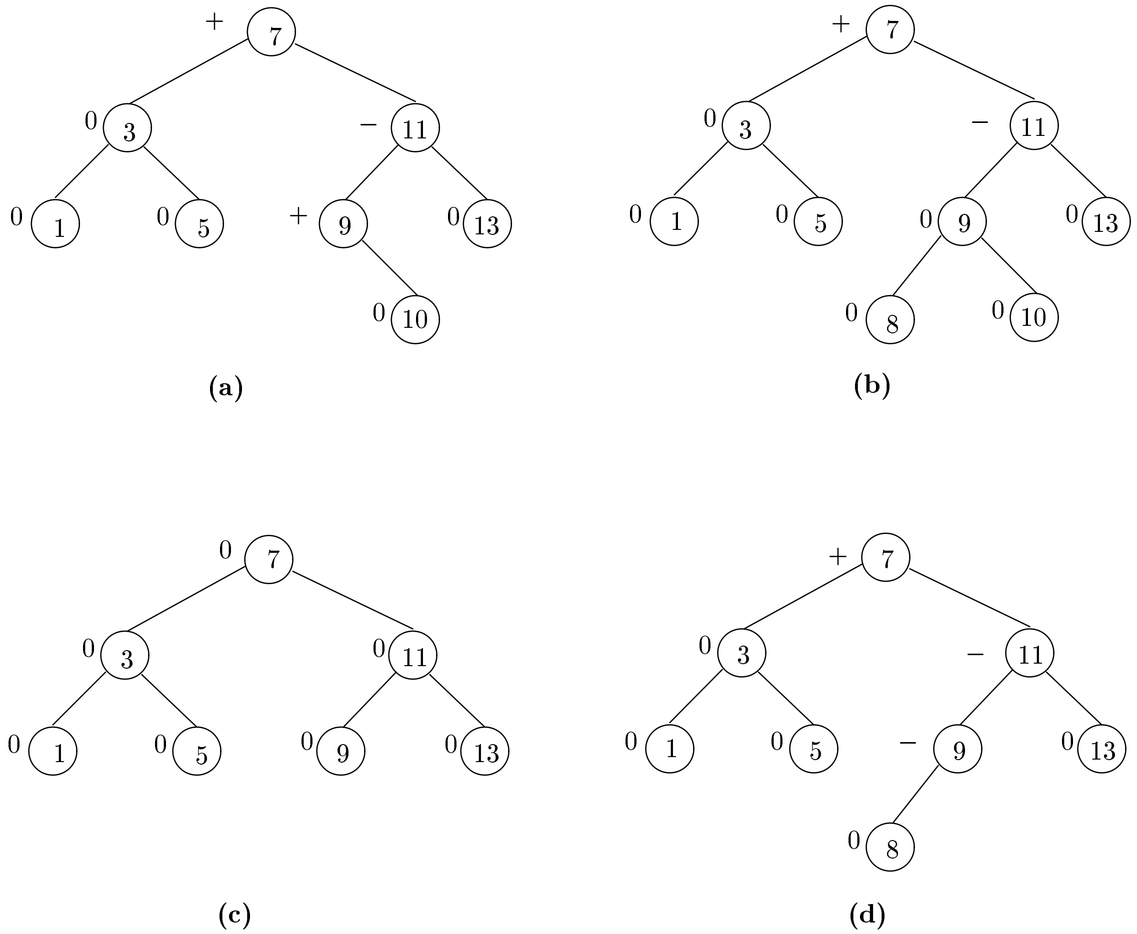


Figure 7

In general, let n be the new node just inserted into the tree and let p be n's parent. Further, let m be the closest ancestor of p that was *not* balanced (that is, that had balance factor $\pm$ 1) before the insertion of n; if no such ancestor of p exists, let m be the root of the tree. (Note that m could be p, if $BF[p] \neq 0$ before the insertion.)

Claim. *Only the balance factors of the nodes between p and m (included) need to be changed as a result of the insertion of n.*

Justification: Consider the (0 or more) nodes that are ancestors of p and proper descendents of m. By choice of m, all these nodes were balanced before the insertion of n. Thus their two subtrees had the same height and the insertion of n has increased the height of one of the subtrees; hence for each such node, its balance factor must be set to -1 or $+1$, depending on whether n was inserted to the left or right subtree, respectively. Next consider node m. If m is the root and was balanced before the insertion, similar remarks as above apply to m: in this case the insertion of n has the effect of increasing the height of the entire tree. If m was *not* balanced before the insertion, we have two possibilities:

- If m was left heavy and n was inserted to m's right subtree, or if m was right heavy and n was inserted to m's left subtree, the subtree rooted at m becomes balanced as a result of the insertion (so we must set $BF[m] = 0$), but its height does not change. Therefore, neither do the heights of m's ancestors' subtrees; so the balance factors of m's proper ancestors do not change, and we can stop the process of balance factor updating here.
- If, on the other hand, m was right heavy and n was inserted to m's right subtree, or if m was left heavy and n was inserted to m's left subtree, the subtree rooted at m becomes unbalanced (these are the two cases illustrated in Figures 1(a) and 1(b) respectively). We can rebalance the subtree as we discussed previously (by the appropriate type of rotation). After the rebalancing, however, the subtree rooted at m will have the same height as it did before the insertion of n (recall Remarks S.4 and D.4). Thus, as argued before, the balance factors of m's ancestors do not change. Note, however, that when we rotate, the balance factors of the rotated nodes need updating, so we must do that before stopping.† $\square$

The discussion on rebalancing and updating the balance factors after an insertion leads us to the following outline for the AVL tree insertion algorithm.

INSERT(x, T)

1. Trace a path from the root down, as in binary search trees, and insert x into a new leaf at the end of that path (the new leaf must be in the proper position, so as to maintain the binary search tree property).
2. Set the balance factor of the new leaf to 0. Retrace the path from the leaf up towards the root and process each node i encountered as follows:
   (a) If the new node was inserted in i's right subtree, then increase $BF[i]$ by 1 (because i's right subtree got taller); otherwise, decrease $BF[i]$ by 1 (because i's left subtree got taller).
   (b) If $BF[i] = 0$ (so the subtree rooted at i became balanced as a result of the insertion, and its height did not change) then stop.
   (c) If $BF[i] = +2$ and $BF[rchild(i)] = +1$ then do a single left rotation on i, adjust the balance factors of the rotated nodes (A and B in Figure 3(b)), and stop.
   (d) If $BF[i] = +2$ and $BF[rchild(i)] = -1$ then do a double right left rotation on i, adjust the balance factors of the rotated nodes (A, B and C in Figure 4(b)), and stop.
   (e) If $BF[i] = -2$ and $BF[lchild(i)] = -1$ then do a single right rotation on i, adjust the balance factors of the rotated nodes (A and B in Figure 5(b)), and stop.
   (f) If $BF[i] = -2$ and $BF[lchild(i)] = +1$ then do a double left right rotation on i, adjust the balance factors of the rotated nodes (A, B and C in Figure 6(b)), and stop.
   (g) If $i = root$ then stop.

† After a rotation, some of the rotated nodes are no longer ancestors of the inserted node; however, they may still need to have their balance factors updated.

THE ALGORITHM FOR DELETE

To delete a key x from an AVL tree T, we first locate the node n where x is stored. (This can be done by using the algorithm for SEARCH.) If no such node exists, we're done (there's nothing to delete). Otherwise we have three cases (as with ordinary binary search trees).

(1) *n is a leaf:* Then we simply remove it. This may cause the tree to cease being height-balanced. So we may need to rebalance it. We also have to update the balance factors of some nodes. These issues will be dealt with shortly.

(2) *n is a node with only one child:* Let n' be n's only child. Note that n' must be a leaf; otherwise the subtree rooted at n would not have been height-balanced before the deletion. In this case we copy the key stored at n' into n and we remove n' as in case (1) (since, as we just argued, it must be a leaf).

(3) *n has two children:* Then we find the smallest key in n's right subtree which, by the binary search tree property, is the smallest key in T larger than the key stored in n. To find this key, we go to n's right child (which exists), and we follow the longest chain of left child pointers until we get to a node n' that has no left child. We copy the key stored in n' into n and remove n' from the tree, as in (1), if n' does not have a right child either, or as in (2), if n' has only a right child.

To complete the algorithm we must discuss the conditions under which rebalancing is required and how the rebalancing can be performed. Since cases (2) and (3) ultimately reduce to deleting a leaf, case (1) is the only one we need to consider.

Rebalancing an AVL Tree after Deleting a Leaf

The deletion of a leaf n will cause the tree to become unbalanced in one of two cases:

(a) It reduces the height of the right subtree of a left heavy node; or,

(b) It reduces the height of the left subtree of a right heavy node.

These two cases, illustrated in Figure 8, are symmetric (as are the analogous cases in insertion), so we will only consider the first. As an exercise, you should treat the other.

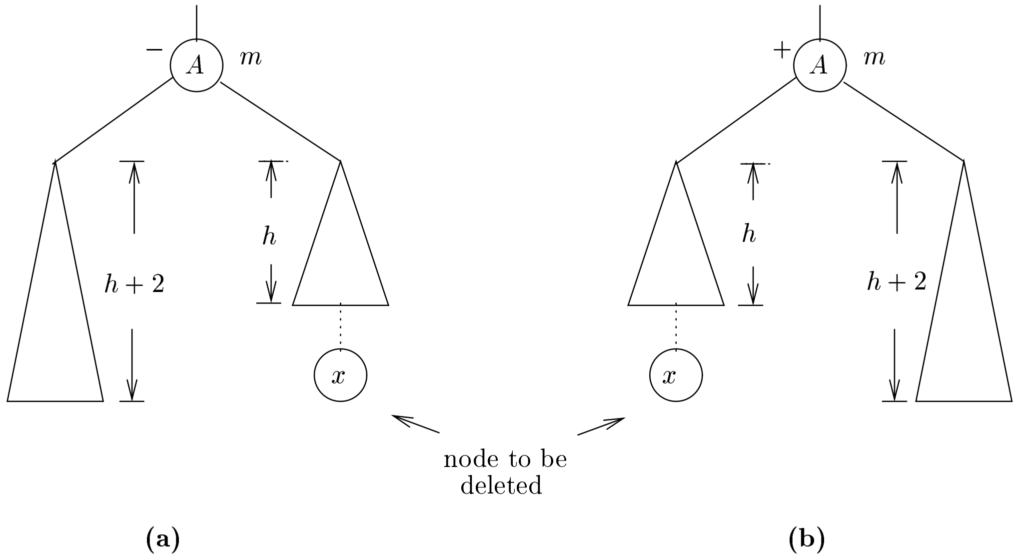


Figure 8

We consider case (a). As with insertion there are two ways this case could arise, shown in Figures 9(a) and 10(a). The subtree in 9(a) can be rebalanced by means of a single right rotation, and the result of this transformation in shown in 9(b). The "0/−" next to B in 9(a) means that this case will arise if the balance

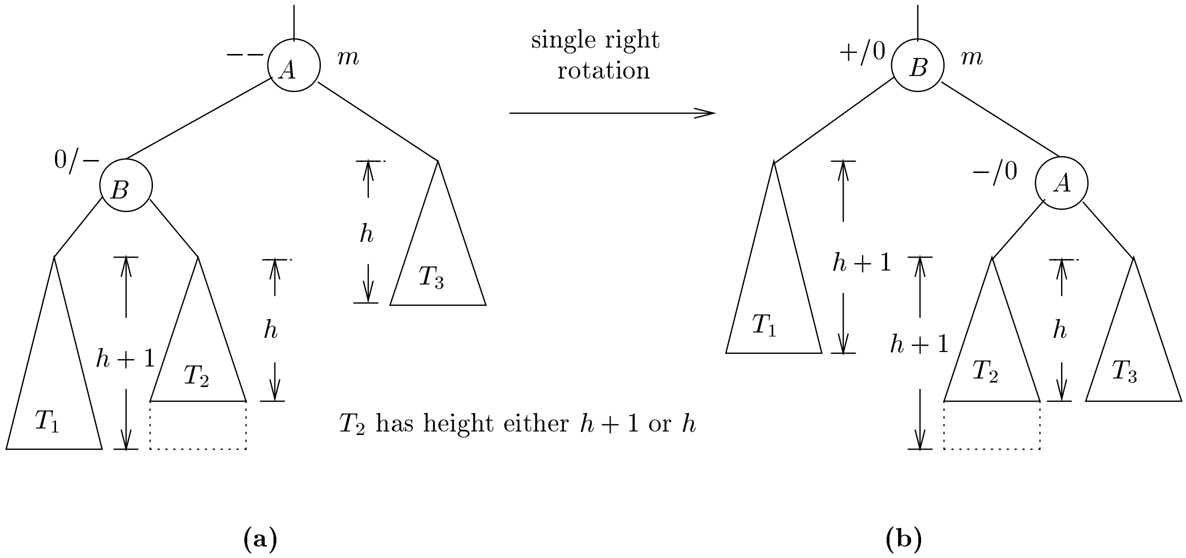


Figure 9

factor of B is 0 or -1 (that is, the height of T_2 is $h+1$ or h). Accordingly, the balance factors of B and A will be $+1$ or 0, and -1 or 0, after the rotation (see 9(b)).

The unbalanced subtree of Figure 10(a) can be rebalanced by means of a double left right rotation, the result of which is shown in 10(b). The balance factors of the nodes that have a label of the form "$*/*/*$" in Figure 10 depend on the heights of T_{21} and T_{22}. Note that *at least one* of these subtrees must have height h (the other could have height $h-1$ or h). The first entry of the label indicates the balance factor in the event T_{21} has height $h-1$ and T_{22} has height h; the second entry of the label indicates the balance factor in the event both trees have height h; and the third entry indicates the balance factor when T_{21} has height h and T_{22} has height $h-1$.

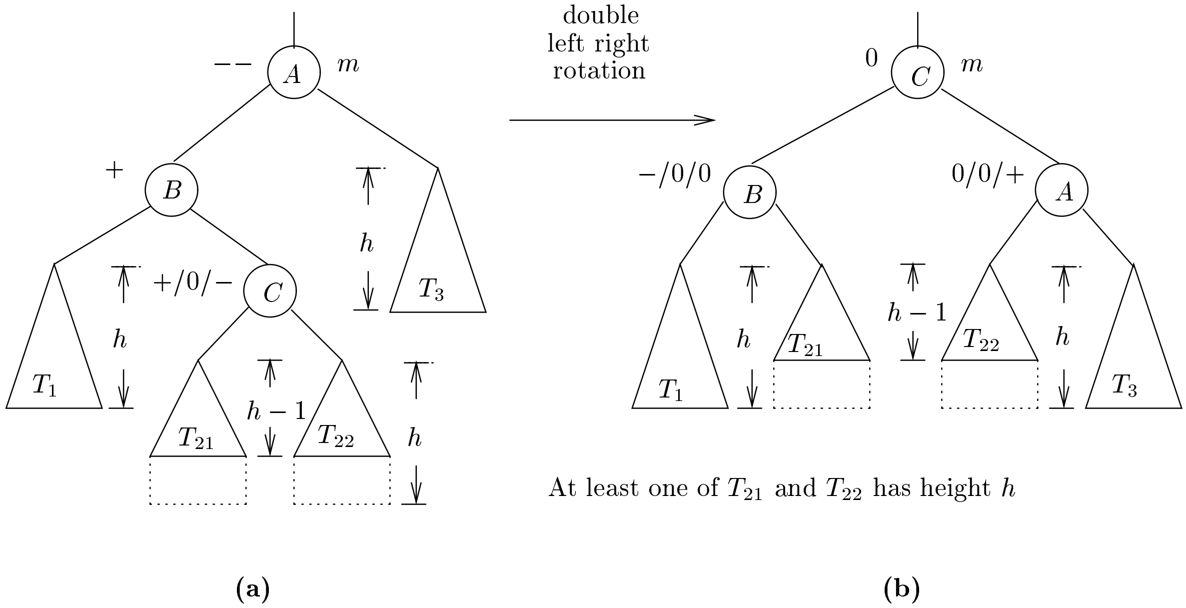


Figure 10

The above two transformations have the following properties.

1. They rebalance the subtree rooted at m (so the subtree becomes height-balanced again).

2. They maintain the binary search tree property.
3. They can be done in constant time by simply manipulating a few pointers. As an exercise, write programs that implement the rotations of Figures 9 and 10, given a pointer to m.
4. *They may decrease the height of the subtree rooted at* m, compared to the height of the subtree before the deletion.

Compare 4 with remarks S.4 and D.4 about rotations to restore balance in insertions. The difference is important: In the insertion algorithm just one rotation always rebalances the subtree, and, by maintaining the height of that subtree, it rebalances the entire tree. In the deletion algorithm the rotation balances the subtree, but since the height is decreased, the balance factor of nodes higher up (closer to the root) may change as a result — so we may have to go on rotating subtrees all the way up to the root in order to rebalance the entire tree. Thus in deletion we may have to do as many as $O(\log n)$ rotations. (That's acceptable though, because each one takes only constant time! We will say more about the complexity of operations shortly.)

Updating the Balance Factors after Deleting a Leaf

We must also address the question of how the deletion of a leaf affects the balance factors of its ancestors (clearly, it doesn't affect the balance factors of other nodes).

Let n be the deleted leaf and let p be its parent. We trace the path from p back to the root and we process each node i we encounter on the way as follows:

- If i was balanced before the deletion (so $BF[i] = 0$) then the left and right subtrees of i had the same height. The removal of n shortened one of them (so i's balance factor must be updated), but the height of the subtree rooted at i after the deletion remains the same as before it. This means that the deletion of n does not affect the balance factors of i's proper ancestors. So, in this case, all we have to do is increase $BF[i]$ by one if n was in i's left subtree (because then the deletion made the right subtree of i taller than the left), or decrease $BF[i]$ by one if n was in i's right subtree (because then the deletion made the left subtree of i taller than the right). After this, we can stop the process of updating balance factors.
- If i was right or left heavy before the deletion ($BF[i] = \pm 1$), we again update $BF[i]$ as above. If this balances node i, the deletion of n shortened one of the two subtrees of i, so we go up the path to consider the next node. Otherwise, the increase or decrease of $BF[i]$ by one causes the subtree rooted at i to become (height) unbalanced ($BF[i]$ becomes ± 2). In this case we need to rebalance the subtree by the appropriate rotation, as discussed previously. If the rotation causes the height of i to decrease compared to its height before the delection (see Remark 4 above), the process of updating balance factors and, possibly, rotating, must continue with i's parent. Otherwise, the rotation leaves the height of i the same as it was before the deletion, and therefore the process stops at i.
- Finally, if the process propagated all the way to the root ($i = \text{root}$) we can stop.

From this discussion you should be able to distill the outline of an algorithm for AVL tree deletion.

WORST CASE TIME COMPLEXITY FOR Search, Insert, Delete

Theorem. (Adelson-Velski and Landis) *The height of an AVL tree with n nodes is at most* $1.44\log_2(n+2)$.

Proof: Let T_h be a height-balanced tree of height h with the *minimum possible number of nodes*, and let n_h be that number of nodes. Since T_h is height-balanced, one of its left subtrees must have height $h-1$ and the other height $h-1$ or $h-2$. Since we want T_h to have the minimum number of nodes, we may assume that one of its subtrees is T_{h-1} and the other is T_{h-2}. Thus, the number of nodes in T_h is equal to the number of nodes in T_{h-1} plus the number of nodes in T_{h-2} plus one (for the root); that is,

$$n_h = n_{h-1} + n_{h-2} + 1.$$

Thus $n_0 = 1$, $n_1 = 2$, $n_2 = 4$, $n_3 = 7$, $n_4 = 12$, and so on. Comparing this with the sequence of Fibonacci numbers we see that, in general, $n_h = F_{h+3} - 1$ (where F_h is the h^{th} Fibonacci number).† From the theory of Fibonacci numbers we know that $F_h > (\phi^h/\sqrt{5}) - 1$, where $\phi = (1+\sqrt{5})/2$‡ (if interested in this and other results on Fibonacci numbers, see Knuth, *The Art of Computer Programming*, Vol. 1 (Fundamental Algorithms), pp. 78–83.)

Thus for the number n of nodes in any AVL tree of height h we must have:

$$n \geq n_h = F_{h+3} - 1 > \left(\frac{\phi^{h+3}}{\sqrt{5}}\right) - 2.$$

Therefore,

$$h < \log_\phi((n+2)\sqrt{5}) - 3,$$

so

$$h < \left(\frac{1}{\log_2 \phi}\right) \cdot (\log_2 \sqrt{5} + \log_2(n+2)) - 3,$$

from which the theorem follows by arithmetic. □

In the worst case, the algorithms for SEARCH, INSERT, and DELETE have to process all nodes in a path from the root to a leaf. The above theorem says that this path must involve at most $O(\log n)$ nodes. Processing a node (be it just comparing the key stored in it to a key we are searching for, updating the balance factor, or performing a rotation on that node) takes constant time. Thus all these algorithms take $O(\log n)$ time in the worst case.

† The i^{th} Fibonacci number is defined inductively as follows: $F_1 = F_2 = 1$, and for $i > 2$, $F_i = F_{i-1} + F_{i-2}$.
‡ ϕ is known as the "golden ratio".