5
AVL Trees

5.1 Bounding the Depth of Trees

We saw earlier that binary search trees are a good compromise between sequentially allocated arrays, which allow direct access, but are not updated easily, and linked lists, that support inserts and deletes in constant time, but cannot be searched efficiently. There is, however, still a major challenge left to be dealt with: the shape of the tree, and in particular its depth, on which ultimately all update complexities depend, is a function of the order in which the elements are processed. The same set of elements, when given in different orders, may produce completely different trees (see Figure 5.1).

Yet, we have generally no influence on the order the elements are presented, so we might get the worst case scenario, in which the depth of the tree is $\Omega(n)$. This is unacceptable, as it would also imply search and update times of the same order. Nothing has then be gained by passing from a linear linked list to a tree structure. What we would like is a tree structure for n elements in which the update times are guaranteed to be as close as possible to the optimal, which is $\log_2 n$. Imposing, however, that the tree should be completely balanced, which yields the $\log_2 n$ bound on the depth, seems to be too restrictive: it would require reshaping the tree constantly.

An elegant tradeoff has been suggested by Georgy Adelson-Velsky and Evgenii Landis and is known by the acronym of its inventors as AVL trees. The idea is to define a condition that on one hand is easily enforceable, but on the other hand yields trees that are *almost* balanced. Let us recall some definitions.

Background Concept: Depth of a Tree

The *depth* of a tree, sometimes also called its *height*, is the number of edges one has to traverse on the path from the root to a most distant

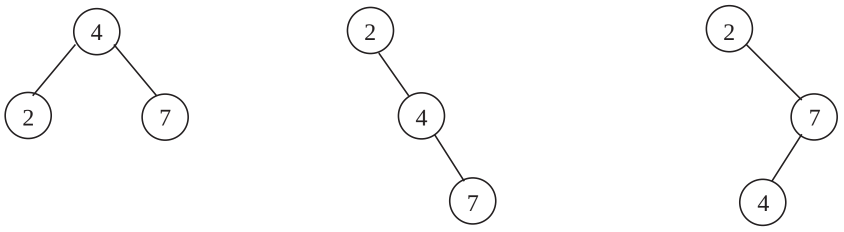

(a) $4-2-7$ (b) $2-4-7$ (c) $2-7-4$

Figure 5.1. Binary search trees for the set $\{2, 4, 7\}$. The different shapes correspond to different input orders.

leaf. The depths of the trees in Figure 5.1 are 1, 2 and 2, respectively. It is the number of levels minus 1, hence a tree consisting of a single node (the root is a leaf) has depth 0, and it is convenient to define the depth of an empty tree as -1. We shall denote the depth of a tree T by $d(T)$. For any node v of a tree T, denote by T_v the subtree of T rooted at v, and by $L(v)$ and $R(v)$ the left child, respectively right child, of v, if it exists.

Definition 5.1. A tree T is an AVL tree (also called *height-balanced tree*) if it is a binary search tree and for each node v of T, the difference of the depths of the subtrees of v is at most 1. Formally,

$$\forall v \in T \qquad |d(T_{L(v)}) - d(T_{R(v)})| \leq 1. \tag{5.1}$$

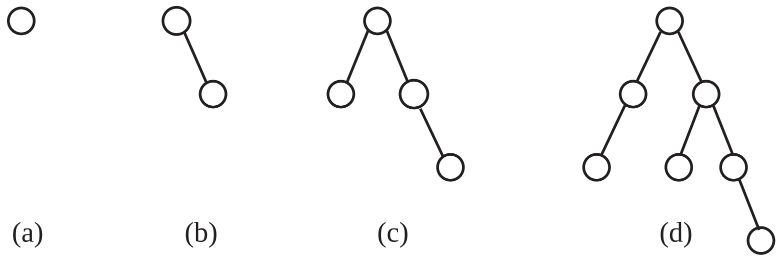
For example, the shape of the upper four trees of Figure 5.2 comply with the constraint of eq. (5.1), the tree in Figure 5.1(a) is also AVL, but the tree in Figure 5.2(e) is not, because of the black node: its left subtree is of depth 2 and its right one of depth 0.

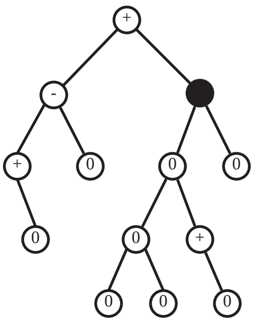
What remains to be done is the following:

- show that the constraint of eq. (5.1) implies a logarithmic depth;
- show how, at low cost, the tree can be maintained complying with the condition after insertions and deletions.

5.2 Depth of AVL Trees

One can get a surprisingly exact lower bound on the depth of the tree in our special case.





(e)

Figure 5.2. (a)–(d) Minimal AVL trees of depths 0–3, respectively. (e) A non-AVL tree.

Theorem 5.1. The depth of an AVL tree T with n nodes is at most logarithmic in n, and more precisely

$$d(T) \leq 1.4404 \log_2(n+1) - 1.33. \tag{5.2}$$

How can one prove such a statement? A first thought might be to try induction on the number of nodes n, since this is the parameter of Theorem 5.1. But the depth of a tree must be an integer, and $\log_2 n$ is not, for most values of n. For example, for $n = 1000$ and $n = 1001$, the theorem yields the upper bounds 12.155 and 12.157, which have both to be rounded to the same integer 12. There seems thus to be a technical problem with the inductive step. To overcome the difficulty in this and similar cases, in particular when stating properties of trees, we shall formulate an alternative, equivalent, theorem, that is easier to prove, by changing our point of view.

Theorem 5.1′. The number of nodes n of an AVL tree with depth k is at least exponential in k, that is $n \geq a^k$ for some constant $a > 1$.

Background Concept: Proving Properties of Trees

One may wonder how the inequality suddenly changed direction. Theorem 5.1 asserts a relationship between two properties of a tree, as can

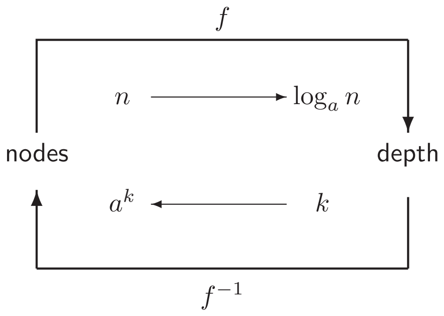


Figure 5.3. Schematic of the relation between the number of nodes of a tree and its depth.

be seen in the upper part of Figure 5.3. Given is a parameter n, the number of nodes in a tree, and the theorem reveals information about $f(n)$, the depth of the tree, stating that

$$f(n) \le \log_a n, \text{ for some constant } a > 1. \tag{5.3}$$

One gets to the alternative formulation of Theorem 5.1$'$ by inverting the roles of the properties: assume now that the depth of the tree is given, denote it by k, and let the new theorem convey information about the corresponding number of nodes, as depicted in the lower part of Figure 5.3. If the function from the nodes to the depth is f, its inverse is usually denoted by f^{-1}. But f is not injective, that is, trees with different numbers of nodes may have the same depth, and therefore f^{-1} is not necessarily a function. This mathematical difficulty can be overcome if we restrict our attention to *minimal* AVL trees, defined as those AVL trees having a minimal number of nodes for their depth. For example, the trees in Figure 5.2(a)–(d) are minimal, but the tree in Figure 5.1(a) is not.

To get to the alternative theorem, we apply the function f^{-1} to both sides of the inequality in eq. (5.3), getting

$$f^{-1}(f(n)) \le f^{-1}\left(\log_a n\right), \tag{5.4}$$

where the inequality did not change direction, because f, and thus also f^{-1}, are nondecreasing functions. Substituting k for $\log_a n$ and consequently a^k for n, we get

$$a^k \le f^{-1}(k). \tag{5.5}$$

The inequality did not change, but the main function of the theorem now appears on the right-hand side. It is customary to rewrite such an inequality as $f^{-1}(k) \geq a^k$, which yields Theorem 5.1′. This is similar to interpreting $x < y$ both as "x is smaller than y" and as "y is larger than x", two quite different statements, but obviously equivalent. Nonetheless, there is a different nuance: reading left to right gives information about x, namely that it is smaller than y, where y is assumed to be known in this context; reading from right to left assumes that x is known and that the statement adds information about y.

Theorem 5.1′ is easier to handle, because it calls for induction on the parameter k and all the values involved are integers. Note also that restricting the attention to minimal AVL trees did not do any harm, since if the theorem is true for minimal trees, it is *a fortiori* true for the nonminimal ones.

Proof by induction on the depth k. Define $N(k)$ as the number of nodes in a minimal AVL tree of depth k and let us try to evaluate it. For $k = 0$, there is only one tree with depth 0 (Figure 5.2(a)) and it contains a single node, thus $N(0) = 1$. To get a tree of depth 1, one needs at least one node in level 0 and in level 1, as in Figure 5.2(b). The tree in Figure 5.1(a) is also of depth 1, but it is not minimal. We conclude $N(1) = 2$. For $k = 2$, one could consider a tree as in Figure 5.1(b), but it is not AVL, because its root is unbalanced. Adding one more node yields a tree as in Figure 5.2(c), and we get $N(2) = 4$.

So far, the sequence is 1, 2, 4, which might suggest that the general term is of the form $N(k) = 2^k$, but this guess fails already at the next step. Indeed, Figure 5.2(d) shows that $N(3) = 7$, and not 8 as we could have expected.

For the general step, consider a minimal AVL tree T of depth $k > 2$. At least one of its subtrees has to be of depth $k - 1$, but must itself be an AVL tree, and minimal, thus with $N(k - 1)$ nodes. The depth of the other subtree cannot be larger than $k - 1$ (otherwise, T is of depth larger than k), nor smaller than $k - 2$ (otherwise the root wouldn't be balanced). It could also be of depth $k - 1$ from the AVL point of view, but then it would not be minimal. Therefore, it must have depth $k - 2$. Adding the root, we can conclude that

$$N(k) = N(k-1) + N(k-2) + 1. \tag{5.6}$$

As example, consider the tree in Figure 5.2(d); its subtrees appear in Figure 5.2(b) and (c).

The recurrence relation of eq. (5.6) should remind us the definition of the famous *Fibonacci* sequence.

Background Concept: Fibonacci Sequence

The Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . is defined by

$$F(0) = 0, F(1) = 1 \quad F(i) = F(i-1) + F(i-2) \text{ for } i \geq 2, \tag{5.7}$$

and its properties have been investigated for centuries. The elements of the sequence can be represented as linear combinations with fixed coefficients of the powers of the roots of the polynomial $x^2 - x - 1$. These roots are

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618 \qquad \text{and} \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.618, \tag{5.8}$$

the first of which is known as the *golden ratio*. The Fibonacci sequence is given by

$$F(i) = \frac{1}{\sqrt{5}}\left(\phi^i - \hat{\phi}^i\right) \qquad \text{for} \quad i \geq 0. \tag{5.9}$$

Note that the absolute value of $\hat{\phi}$ is smaller than 1, so $\hat{\phi}^i$ quickly becomes negligible with increasing i, and one gets a good approximation for $F(i)$ by taking just the first term of the right-hand side of eq. (5.9): $F(i) \simeq \frac{1}{\sqrt{5}}\phi^i$. Actually, $F(i)$ is $\frac{1}{\sqrt{5}}\phi^i$, rounded to the nearest integer, for all elements of the sequence.

Comparing the sequences $N(k)$ and $F(k)$, one gets

k	0	1	2	3	4	5	6	7	8	9	10
$N(k)$	1	2	4	7	12	20	33	54	88	143	232
$F(k)$	0	1	1	2	3	5	8	13	21	34	55

suggesting that $N(k) = F(k+3) - 1$. Indeed, this is true for $k \in \{0, 1\}$, and for larger k, by induction,

$$\begin{aligned} N(k) &= N(k-1) + N(k-2) + 1 \\ &= (F(k+2) - 1) + (F(k+1) - 1) + 1 = F(k+3) - 1. \end{aligned}$$

Putting it all together, it follows that n, the number of nodes in an AVL tree of depth k satisfies

$$n \geq N(k) = F(k+3) - 1 \simeq \frac{1}{\sqrt{5}}\phi^{k+3} - 1,$$

giving the required exponential bound for Theorem 5.1′, from which the bound on the depth k for Theorem 5.1 can be derived:

$$\log_\phi \left(\sqrt{5}(n+1)\right) \geq k+3,$$

which is equivalent to eq. (5.2). ∎

We conclude that even in the worst possible case, the depth of an AVL tree is at most 44% larger than in the best possible scenario. The average depth can be shown to be only about $1.04 \log_2 n$, implying that accessing and searching AVL trees is very efficient.

5.3 Insertions into AVL Trees

Once it is known that the AVL constraints indeed imply logarithmic depth for the trees, we have to take care of how to maintain the validity of the constraints during possible tree updates. Recall that an AVL tree T is a search tree, so if we want to insert a new node with value x, the exact location of this new node is determined by T, and there is no freedom of choice. A new insert might thus well cause the tree to become unbalanced.

To avoid constantly recalculating the depths of subtrees, we add a new field to each node v, called its *Balance Factor*, defined by

$$BF(v) = d(T_{R(v)}) - d(T_{L(v)}). \tag{5.10}$$

By definition, each node v of an AVL tree satisfies $BF(v) \in \{-1, 0, 1\}$, which we shall indicate as $-$, 0, $+$ in the diagrams. Figure 5.4 depicts an example AVL

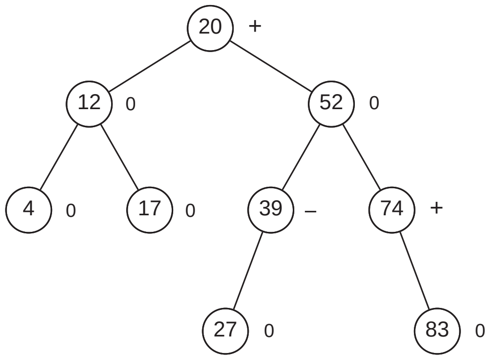


Figure 5.4. Example of AVL tree.

tree, with the BF fields appearing next to each node. BF can also be defined for non-AVL trees, as the one in Figure 5.2(e), where the nodes display their BF, except for the black node, which is unbalanced (its BF would be -2). A node with $BF = 0$ will be called *balanced*, otherwise unbalanced.

We shall assume that AVL trees are built incrementally from scratch, so there is no need to deal with how to turn an arbitrary tree into an AVL tree, and we may restrict our attention on how the balancing condition can be maintained during insertions and deletions.

To avoid a confusion between a node and the value it contains, we shall continue with the convention introduced earlier of denoting by v_x a node containing a value x.

To insert an element with value x into an existing AVL tree T, we proceed, in a first stage, according to the rules of binary search trees, ignoring for the moment the additional balancing constraints, and leaving the balance factors unchanged. After having inserted the new node (it forms a new leaf v_x), we check whether all the nodes of T still meet the conditions, take appropriate action if not, and finally update the BF fields.

Define the *insertion path* of x as the sequence of nodes, starting with the root of T, whose values are compared with x during the insertion. For example, if we were to insert the value 31 into the tree of Figure 5.4, the insertion path would consist of the elements with values 20, 52, 39, and 27. Consider a node w not belonging to the insertion path. The subtrees of w are not affected by the insertion of v_x, so their depths, and consequently also $BF(w)$, do not change. In other words, the only elements that have to be checked are those on the insertion path. Since these elements have to be processed bottom up, and there is usually no pointer from a node to its parent in a tree, it is convenient to store the elements of the insertion path in a stack S. In addition, we also store with each element a flag indicating whether the next element on the path is a left or right child of the current one. For our example of inserting the value 31, the stack would consist of: $S = [(v_{20}, R), (v_{52}, L), (v_{39}, L), (v_{27}, R)]$.

The inserted node v_x is always a leaf. If its parent node z has $BF(z) = 0$, it must have been a leaf prior to the insertion, but now $BF(z)$ has to be changed to $+$ if v_x is a right child of z, and to $-$ if it is a left child. This, however, changes the depth of T_z, the subtree rooted at z, one therefore has to continue to the next higher level. For example, when inserting the value 15 into the tree of Figure 5.4, $BF(v_{17})$ is changed to $-$, and $BF(v_{12})$ is changed to $+$. The update procedure thus climbs up the tree along the insertion path, by means of the elements stored in the stack S, as long as the BF of the current node is 0, and changes this value to $+$ or $-$.

This loop is terminated in one of the following three cases:

(i) When the root is reached. In this case, the depth of the entire tree T increases by 1, but the tree itself still complies with the AVL definition.

(ii) When an unbalanced node is reached from the side which corrects the imbalance. That is, the update algorithm reaches a node w with $BF(w) = -$ and we come from a right child (in other words, the element in the stack is (w, R)), or $BF(w) = +$ and we come from a left child (in other words, the element in the stack is (w, L)). In both cases, $BF(w)$ is changed to 0, and the iteration can stop, since the tree rooted at w did not change its depth, hence the BF fields of the ancestors of w are not affected. To continue the example of inserting the value 15, after changing $BF(v_{12})$ to $+$, one reaches the root, v_{20}. Its BF is $+$ and we come from the left, so we set $BF(v_{20}) = 0$.

(iii) When an unbalanced node is reached, but from the side which aggravates the imbalance. That is, the update algorithm reaches a node w with $BF(w) = -$ and we come from a left child, or $BF(w) = +$ and we come from a right child. The BF of w is thus temporarily ± 2, which violates the AVL rules. Returning to the example of inserting 31, $BF(v_{27}) = 0$ and is changed to +, but the next node is v_{39}, its BF is $-$ and we come from the left: there is thus a violation at v_{39}.

Figure 5.5(a) displays a generic view of this last case schematically. The node v_a is the first node with $BF \neq 0$ on the path from the newly inserted node v_x to the root of the tree. In this figure, $BF(v_a) = +$, and the case for $BF(v_a) = -$ is symmetrical. The left subtree of v_a is α, of depth h, and the right subtree of v_a is δ, of depth $h + 1$, with $h \geq -1$. The new element v_x is inserted as a leaf in δ, increasing its depth to $h + 2$. The challenge is now to rebalance the tree rooted at v_a.

It turns out that the actions to be taken for the rebalancing are different according to which of the subtrees of δ contains the new node v_x. We therefore have to take a closer look, given in Figure 5.5(b), which zooms in on δ: the root of δ is v_b and its subtrees are denoted β and γ. Note that both are of depth h, as we know that $BF(v_b) = 0$, because v_a is the *first* node we extract from the stack S for which $BF \neq 0$. The tree in Figure 5.5(b) gives enough information for the case that v_x is inserted into γ, the right subtree of v_b.

In the other case, when v_x is inserted into β, the left subtree of v_b, one needs to zoom in even further. Different actions are then required, and the update procedure separates the subtrees of β. This leads to the tree in Figure 5.5(c), where the tree β is given in more detail: if β is not empty, i.e., $h \geq 0$, then its root is v_c, and the subtrees are β_1 and β_2, both of the same depth $h - 1$, as

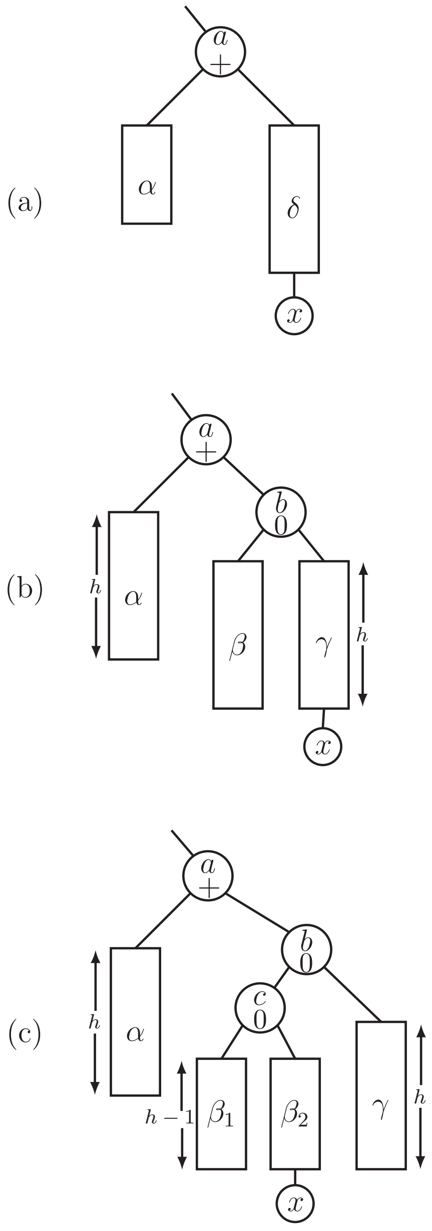


Figure 5.5. AVL insertion: the need for rebalancing.

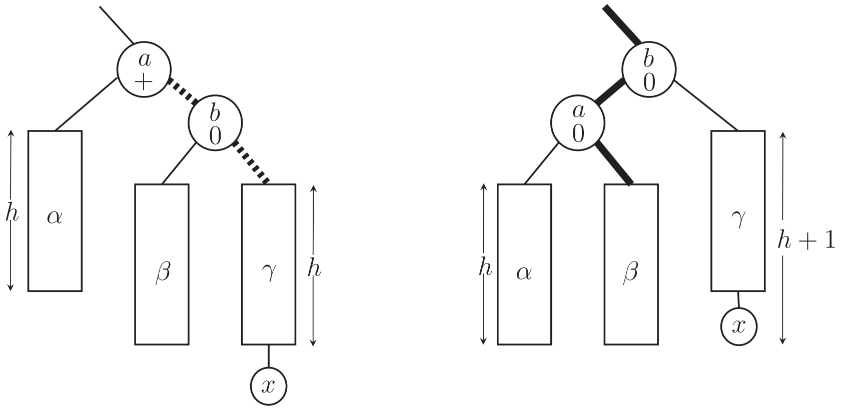


Figure 5.6. AVL insertion: single rotation.

explained earlier for β and γ. The inserted node v_x can now be either in β_2, as in the figure, or in β_1. This will not change the update commands, only the balance factors. If β is empty, the inserted node v_x is identified with the node v_c in Figure 5.5(c). The good news is that no further zooming in is required, and these are all the cases to be dealt with.

The rebalancing is achieved by performing what has become known as *rotations*. The subtree rooted at the node v_a, where the imbalance has occurred, is rearranged by changing some of the connections between its components. Figure 5.6(a) redraws the first case presented in Figure 5.5(b) (v_x inserted in γ), showing the structure before the rotation, whereas Figure 5.6(b) is the resulting subtree consisting of the same nodes, but after the rotation. The root of the subtree is now v_b, and v_a is its new left child. This does not violate the rules of search trees, since v_b was previously the right child of v_a, implying that $b > a$; the new connection means that $a < b$, and we mentioned already earlier that these are of course equivalent. The left subtree of v_a remained α and the right subtree of v_b remained γ, and only β changed its position from being previously the left subtree of v_b to become now the right subtree of v_a. Again, both layouts define the range of all the values y stored in β to be $a < y < b$. We conclude that also after the rotation, the tree remains a search tree.

The depth of the entire subtree after the rotation is $h + 2$, just as was the subtree prior to the insertion of v_x, and the *BF* fields are indicated in the figure. We have thus succeeded in adding a new node without changing the depth of this subtree, meaning that the insertion process can stop after the rotation.

Note that though the extent of the necessary update steps seems frightening at first sight, possibly encompassing a substantial part or even the entire tree, the actually needed programming commands are very simple: only the three pointers indicated as boldfaced edges in Figure 5.6(b) have to be changed. It

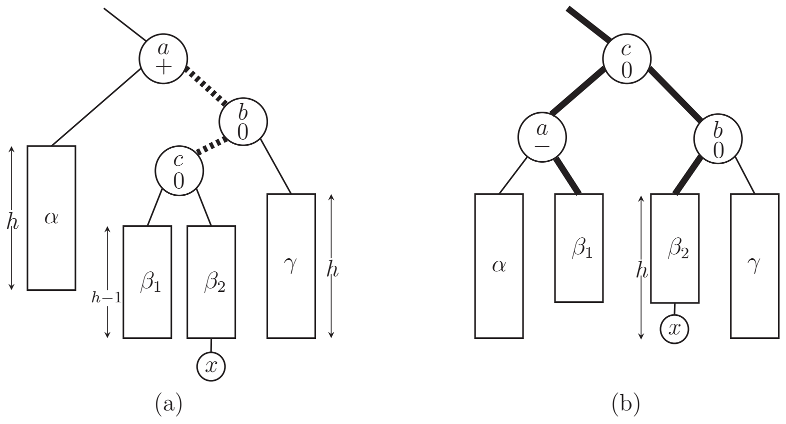


Figure 5.7. AVL insertion: double rotation.

is particularly noticeable that this small number of steps is independent of h, and might be performed quite far from the node v_x which triggered the whole update process. Nevertheless, after having located the exact position to perform the correcting steps, this can be done in $O(1)$.

A closer look also reveals why this is not a solution in the case v_x is inserted in β. In that case, the tree in Figure 5.6(b) would not be AVL, because the right and left subtrees of v_b would be of depths h and $h+2$, respectively. This explains the need for a different rebalancing procedure.

If v_x is inserted in β, the corresponding rotations are shown in the two parts of Figure 5.7. The new root of the subtree is v_c and the three nodes v_a, v_b, and v_c are rearranged; they satisfy before and after the rotation $a < c < b$. The subtrees α and γ remain in place, but the subtrees β_1 and β_2 of v_c are separated and moved to become the left subtree of v_a and the right subtree of v_b, respectively. The defining conditions for the values y of the elements of β_1 are, before and after the update: $a < y < c$ and for β_2: $c < y < b$. If v_x would have been inserted into β_1, the rotation would be the same, but we would have $BF(v_a) = 0$ and $BF(v_b) = +$.

In both cases, the depth of the entire subtree after the rotation is $h+2$, just as was the subtree prior to the insertion of v_x. The number of necessary pointer updates is now five, and they are indicated, as earlier, by boldfaced edges. Referring to the nodes v_a and v_b in Figure 5.6 and to the nodes v_a, v_b, and v_c in Figure 5.7, the first update is called a *single* and the second a *double* rotation.

One question still remains: how do we know which of the rotations to apply? The answer is to refer to the last two steps of the iteration using the stack to process the elements of the insertion path bottom up. Each node on this path has been stored together with a direction indicator, so the four possibilities for the directions of the last two steps are *LL*, *LR*, *RL*, and *RR*. If these steps are in the same direction, *LL* or *RR*, a single rotation is needed; if they are of opposite directions, *LR* or *RL*, we perform a double rotation. The edges corresponding to these last two steps appear in dashed boldface in Figures 5.6(a) and 5.7(a).

Summarizing the insertion, we ascend the tree, starting at the newly inserted leaf and passing from one node to its parent node, as long as we see $BF = 0$, changing it to $+$ or $-$ as appropriate. If a node w is reached with $BF(w) \neq 0$, then if we come from the "good" side, $BF(w)$ is set to 0 and the process stops; if we come from the "bad" side, the subtree rooted at w is rebalanced by means of a single or double rotation, and the process stops. In any case, there is only $O(1)$ of work per treated node, thus the entire insertion does not require more than $O(\log n)$.

5.4 Deletions from AVL Trees

As already mentioned in the previous chapter dealing with trees in general, insertions and deletions in search trees are not always symmetrical tasks. An insertion always adds the new element as a leaf, but we might want to delete any element, not necessarily a leaf.

When dealing with the deletion of a node v_x from an AVL tree, we may nevertheless restrict the attention to deleting a leaf. Indeed, if v_x has one child, this child must be a leaf, otherwise v_x would be unbalanced; and deleting a parent of a leaf can be considered as replacing its value with that of its child and then deleting the leaf. For example, in Figure 5.4, from the point of view of the layout of the nodes in the tree, deleting the node with value 74 results in exactly the same operations as if we were deleting the leaf with value 83. And if v_x has two children, we actually delete the successor of v_x, which is either a leaf or has only one child.

Assume then that we wish to delete a leaf v_x. Just as for the insertion, we start by storing in a stack the elements compared with x in its insertion path (which, formally, should rather be called now deletion path). Proceed then along this path from v_x toward the root, as long as the encountered nodes have $BF \neq 0$, and coming from the good side. More precisely, if the current node w has $BF(w) = +$ and we come from the right subtree, or if $BF(w) = -$ and we come from the left subtree, we set $BF(w) = 0$ and continue: the subtree rooted at w, T_w, was not balanced and now it is, but its depth is reduced by 1, hence one has to check also above w.

If a node w with $BF(w) = 0$ is encountered, this is changed to + (resp., –) if we come from the left (resp., right). This is the case in which T_w was balanced, and the deletion of v_x made it unbalanced, but without violating the AVL constraint. The depth of T_w, though, did not change, thus the update process stops.

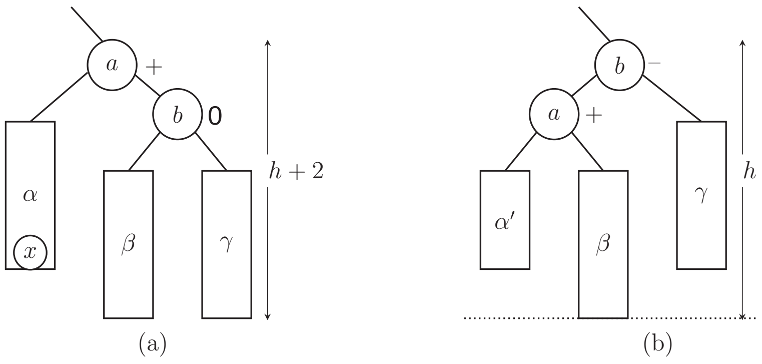


Figure 5.8. AVL deletion: $BF(b) = 0$.

If we get to an unbalanced node w, but from the bad side, that is, $BF(w) = +$ and we come from the left, or $BF(w) = -$ and we come from the right, the subtree T_w has to be rebalanced. Fortunately, the single and double rotations we saw earlier are all that is needed also in this case, just the conditions when to apply which kind of rotation differ. We shall treat, as before, only the case in which v_x is deleted from a left subtree α of a node v_a with $BF(v_a) = +$, see Figure 5.8(a), as the complementing case, with $BF(v_a) = -$, is symmetric.

The depth of the subtrees α, β, and γ is h, and the constraint on h is now $h \geq 0$, because α contains at least the node v_x and thus cannot be empty. Let v_b be the root of the right subtree of v_a. We need to know the balance factor of v_b to correctly draw its subtrees, but how can we know it? In the case of insertion, the node v_b was a part of the insertion path of x, so we knew from the algorithm that $BF(v_b)$ had to be 0. But for deletion, we assume that v_x is deleted from α, whereas v_b belongs to the *other* subtree of v_a. We thus have no information about $BF(v_b)$ and indeed, all three possibilities, +, – or 0, are plausible. These three possibilities give rise to the three cases to be treated for the deletion of an element from an AVL tree; the balance factor $BF(v_b)$ appears in boldface in the corresponding diagrams in Figures 5.8(a), 5.9(a), and 5.10(a).

Figure 5.8 assumes $BF(v_b) = 0$. The subtrees β and γ are then both of the same depth h as α, and when v_x is removed, the node v_a violates the AVL constraint. The solution, given in part (b) of the figure, is a single rotation. We denote by α', of depth $h - 1$, the tree obtained by deleting v_x from α. After the rotation, the new subtree T_{v_b} is of the same depth $h + 2$ as was T_{v_a} prior to the rotation; the update process may thus stop.

For the second case, given in Figure 5.9, assume $BF(v_b) = +$. The only difference with the previous case is that β is now shorter, but the solution seems at first sight to be the same: a single rotation. However, the tree in part (b) of

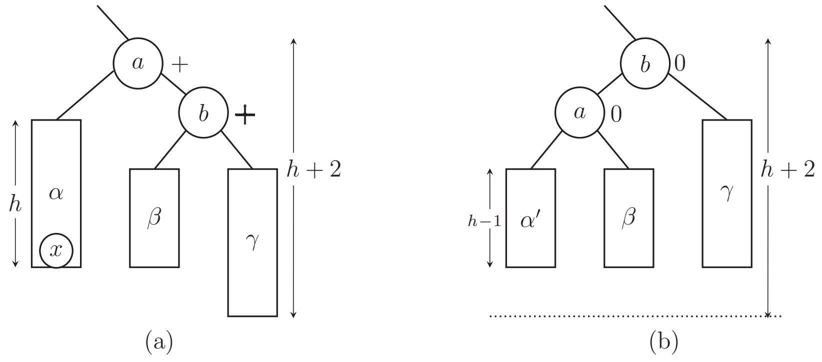


Figure 5.9. AVL deletion: $BF(b) = +$.

the figure has now depth only $h + 1$, which means that there might be more updates needed. This is the first example we see, in which the depth of the tree is changed during a rotation. It may thus happen that during our continuing ascent in the tree, more nodes are encountered that violate the AVL rules, triggering more rotations. Still, since only $O(1)$ operations are needed for rotating, the overall update time stays bounded by $O(\log n)$.

The third case is depicted in Figure 5.10, with $BF(v_b) = -$, implying that the left subtree of v_b is now deeper. The solution here is a double rotation, but as before, we ignore the balance factor of v_c. The figure assumes $BF(v_c) = 0$, with the subtrees β_1 and β_2 both of depth $h - 1$, but the same solution is also valid if $BF(v_c) = +$ and the depth of β_1 is only $h - 2$ (as indicated by the dotted line in β_1), or if $BF(v_c) = -$ and it is β_2 that has depth $h - 2$. As in the second

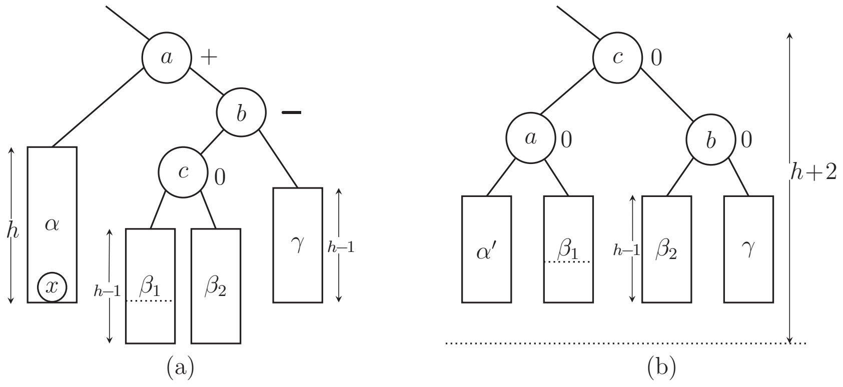


Figure 5.10. AVL deletion: $BF(b) = -$.

case, the double rotation results in a tree of depth $h + 1$, and one has to continue upward with the update process.

Summarizing the deletion, we ascend the tree, starting at the deleted leaf v_x and passing from one node to its parent node, as long as we see $BF = +$ or – and we come from the good side, changing the BF to 0. If a node w is reached with $BF(w) = 0$, it is changed to + or – as appropriate and the process stops. If $BF(w) = +$ or – and we come from the bad side, the subtree rooted at w is rebalanced by means of a single or double rotation, depending on the BF of the child u of w belonging to the *other* side, not that of v_x. Only in case $BF(u) = 0$ does the process stop, otherwise it continues to the next higher level. The entire deletion thus requires at most $O(\log n)$.

5.5 Alternatives

AVL trees are not the only possibility to enforce logarithmic update times. Many other balanced structures have been suggested, and we mention only trees that have become known as *red-black* trees, and *B-trees* that are treated in the next chapter, though the latter have different application areas.

The general mechanism, however, remains the same. One first defines a set of rules defining the trees. Then one has to show that these rules imply a bound on the depth. Finally, algorithms have to be devised that enable the constraints to be maintained, even when elements are adjoined or removed.

Exercises

5.1 Insert the elements of the following sequence, in order, into an initially empty AVL tree, taking care of rebalancing the tree after every insertion, if needed:

$$3, 14, 15, 9, 2, 6, 5, 35.$$

Now delete these elements, one by one, in the same order as they have been inserted.

5.2 We saw that the number of nodes $N(k)$ in an AVL tree of depth k is at least $N(k-1) + N(k-2) + 1$, which lead to a tight bound connected to the Fibonacci sequence. One can get a logarithmic bound for the depth more easily, by using the fact that $N(k-1) > N(k-2)$. What is the resulting bound on $N(k)$?

5.3 Define AVL2 trees as binary search trees, in which for each node v,

$$|d(T_{L(v)}) - d(T_{R(v)})| \leq 2.$$

(a) Derive a bound on the depth on AVL2 trees.
(b) What are the advantages and disadvantages of AVL2 trees relative to AVL trees?

5.4 We defined four kinds of *rotations* in the rebalancing of AVL trees: simple and double, to the left or right.

(a) Show that when given two binary search trees T_1 and T_2, not necessarily AVL, with the same set of values, one can transform the one into the other by a sequence of simple rotations. **Hint:** Show first how to deal with the root, then with the subtrees.
(b) Show that the number of rotations is $O(n^2)$.

5.5 Given is a connected, undirected, unweighed graph $G = (V, E)$ and a pair of vertices $u, v \in V$. Let $d(u, v)$ denote the length of a shortest path from u to v (in terms of number of edges). The *diameter* of the graph is defined as

$$D(G) = \max_{u,v \in V} d(u, v).$$

We wish to study the diameter of a binary tree, considering it as a special case of a graph rather than as a data structure.

(a) Show why the following claim is wrong for a binary tree T. **Claim:** *If the diameter of the tree is D, then there must be a path of length D passing through its root; therefore the diameter of a tree is simply the sum of the depths of its left and right subtrees,* $+\,2$.
(b) Is the claim true for AVL trees?
(c) Write an algorithm getting an AVL tree as input and returning its diameter.

5.6 Denote the Fibonacci numbers by F_i for $i \geq 0$. Beside the direct way to compute F_i by eq. (5.9), one can obviously do it in time $O(n)$ using eq. (5.7). Let us show that one can do better. We assume that any multiplication, also of matrices 2×2, requires time $O(1)$.

(a) Define the matrix $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. We then have $\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} = A \begin{pmatrix} F_{i-1} \\ F_{i-2} \end{pmatrix}$. Derive from it a formula for $\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$, as a function of A and $\begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

(b) We thus would like to calculate A^k in time less than $O(k)$. Suppose A^8 is already known. Do we really need 8 more multiplications to calculate A^{16}? Show how to do it in a single operation.

(c) Using the same principle, how many operations are needed for the evaluation of A^8? Generalize to A^k when k is a power of 2.

(d) How do we calculate A^5 (note that 5 is not a power of 2)? And A^{12}? Generalize to any power A^k, where k is any integer.

(e) Summarizing: given any k, how many matrix multiplications are needed to calculate A^k? Deduce from it a bound for the evaluation of the nth Fibonacci number.

5.7 Build an example of an AVL tree with a minimal number of nodes, such that the deletion of one of the nodes requires two rotations. **Hint:** The number of nodes in such a tree is 12.

Generalize to find an AVL tree with a minimal number of nodes, for which the deletion of one of the nodes requires k rotations, for $k \geq 1$.