9.2 AVL Trees

In the previous section, we discussed what should be an efficient dictionary data structure. However, the worst-case performance it achieves for the various operations is linear time, which is no better than the performance of sequence-based dictionary implementations (such as log files and look-up tables discussed in Chapter 8). In this section, we describe a simple way of correcting this problem so as to achieve logarithmic time for all the fundamental dictionary operations.

Definition

The simple correction is to add a rule to the binary search tree definition that will maintain a logarithmic height for the tree. The rule we consider in this section is the following ***height-balance property***, which characterizes the structure of a binary search tree T in terms of the heights of its internal nodes (recall from Section 6.2.2 that the height of a node v in a tree is the length of a longest path from v to an external node):

***Height-Balance Property*:** For every internal node v of T, the heights of the children of v differ by at most 1.

Any binary search tree T that satisfies the height-balance property is said to be an ***AVL tree***, named after the initials of its inventors, Adel'son-Vel'skii and Landis. An example of an AVL tree is shown in Figure 9.8.

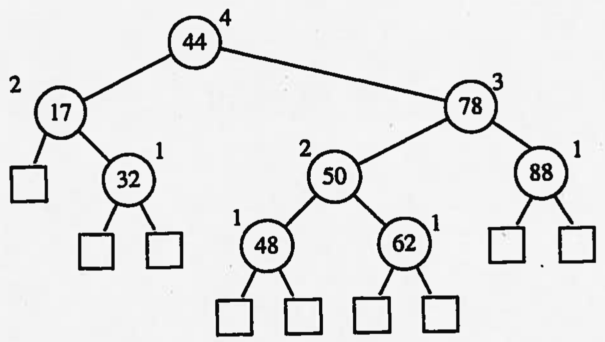

Figure 9.8: An example of an AVL tree. The keys are shown inside the nodes, and the heights are shown next to the nodes.

An immediate consequence of the height-balance property is that a subtree of an AVL tree is itself an AVL tree. The height-balance property has also the important consequence of keeping the height small, as shown in the following proposition.

Proposition 9.2: *The height of an AVL tree T storing n items is $O(\log n)$.*

Justification: Instead of trying to find an upper bound on the height of an AVL tree directly, it turns out to be easier to work on the "inverse problem" of finding a lower bound on the minimum number of internal nodes $n(h)$ of an AVL tree with height h. We will show that $n(h)$ grows at least exponentially, that is, $n(h)$ is $\Omega(c^h)$ for some constant $c > 1$. From this, it will be an easy step to derive that the height of an AVL tree storing n keys is $O(\log n)$.

To start with, notice that $n(1) = 1$ and $n(2) = 2$, because an AVL tree of height 1 must have at least one internal node and an AVL tree of height 2 must have at least two internal nodes. Now, for $h \geq 3$, an AVL tree with height h and the minimum number of nodes is such that both its subtrees are AVL trees with the minimum number of nodes: one with height $h-1$ and the other with height $h-2$. Taking the root into account, we obtain the following formula that relates $n(h)$ to $n(h-1)$ and $n(h-2)$, for $h \geq 3$:

$$n(h) = 1 + n(h-1) + n(h-2). \tag{9.1}$$

At this point, the reader familiar with the properties of Fibonacci progressions (Section 2.2.3 and Exercise C-3.12) will already see that $n(h)$ is a function exponential in h. For the rest of the readers, we will proceed with our reasoning.

Formula 9.1 implies that $n(h)$ is a strictly increasing function of h. Thus, we know that $n(h-1) > n(h-2)$. Replacing $n(h-1)$ with $n(h-2)$ in Formula 9.1 and dropping the 1, we get, for $h \geq 3$,

$$n(h) > 2 \cdot n(h-2). \tag{9.2}$$

Formula 9.2 indicates that $n(h)$ at least doubles each time h increases by 2, which intuitively means that $n(h)$ grows exponentially. To show this fact in a formal way, we apply Formula 9.2 repeatedly, yielding the following series of inequalities:

$$\begin{aligned} n(h) &> 2 \cdot n(h-2) \\ &> 4 \cdot n(h-4) \\ &> 8 \cdot n(h-6) \\ &\ \ \vdots \\ &> 2^i \cdot n(h-2i). \end{aligned} \tag{9.3}$$

That is, $n(h) > 2^i \cdot n(h-2i)$, for any integer i, such that $h - 2i \geq 1$. Since we already know the values of $n(1)$ and $n(2)$, we pick i so that $h - 2i$ is equal to either 1 or 2. That is, we pick

$$i = \left\lceil \frac{h}{2} \right\rceil - 1.$$

By substituting the above value of i in formula 9.3, we obtain, for $h \geq 3$,

$$\begin{aligned} n(h) &> 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} n(1) \\ &\geq 2^{\frac{h}{2} - 1}. \end{aligned} \tag{9.4}$$

By taking logarithms of both sides of formula 9.4, we obtain

$$\log n(h) > \frac{h}{2} - 1,$$

from which we obtain

$$h < 2 \log n(h) + 2, \tag{9.5}$$

which implies that an AVL tree storing n keys has height at most $2 \log n + 2$. ■

By Proposition 9.2 and the analysis of binary search trees given in Section 9.1, the operations find() and findAll(), in a dictionary implemented with an AVL tree, run in time $O(\log n)$ and $O(\log n + s)$, respectively, where n is the number of items in the dictionary and s is the size of the iterator returned by findAll(). The important issue remaining is to show how to maintain the height-balance property of an AVL tree after an insertion or removal.

9.2.1 Update Operations

The insertion and removal operations for AVL trees are similar to those for binary search trees, but with AVL trees we must perform additional computations.

Insertion

An insertion in an AVL tree T begins as in an insertItem() operation described in Section 9.1.2 for a (simple) binary search tree. Recall that this operation always inserts the new item at a node w in T that was previously an external node, and it makes w become an internal node with operation expandExternal(). That is, it adds two external node children to w. This action may violate the height-balance property, however, for some nodes increase their heights by one. In particular, node w, and possibly some of its ancestors, increase their heights by one. Therefore, let us describe how to restructure T to restore its height balance.

Given a binary search tree T, we say that an internal node v of T is ***balanced*** if the absolute value of the difference between the heights of the children of v is at most 1, and we say that it is ***unbalanced*** otherwise. Thus, the height-balance property characterizing AVL trees is equivalent to saying that every internal node is balanced.

Suppose that T satisfies the height-balance property, and hence is an AVL tree, prior to our inserting the new item. As we have mentioned, after performing the operation expandExternal(w) on T, the heights of some nodes of T, including w, increase. All such nodes are on the path of T from w to the root of T, and these are the only nodes of T that may have just become unbalanced. (See Figure 9.9a.) Of course, if this happens, then T is no longer an AVL tree; hence, we need a mechanism to fix the "unbalance" that we have just caused.

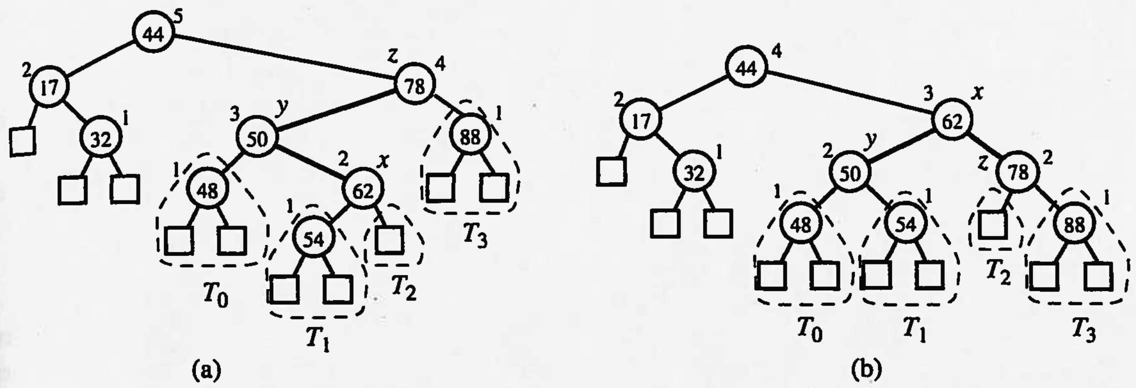

Figure 9.9: An example insertion of an element with key 54 in the AVL tree of Figure 9.8: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify the nodes x, y, and z participating in the trinode restructuring.

We restore the balance of the nodes in the binary search tree T by a simple "search-and-repair" strategy. In particular, let z be the first node we encounter in going up from w toward the root of T, such that z is unbalanced. (See Figure 9.9a.) Also, let y denote the child of z with higher height (and note that y must be an ancestor of w). Finally, let x be the child of y with higher height (and if there is a tie, choose x to be an ancestor of w). Note that node x is a grandchild of z and could be equal to w. Since z becomes unbalanced because of an insertion in the subtree rooted at its child y, the height of y is 2 greater than its sibling.

We now rebalance the subtree rooted at z by calling the ***trinode restructuring*** function, restructure(x), described in Code Fragment 9.7 and illustrated in Figures 9.9 and 9.10. A trinode restructure temporarily renames the nodes x, y, and z as a, b, and c, so that a precedes b and b precedes c in an inorder traversal of T. There are four possible ways of mapping x, y, and z to a, b, and c, as shown in Figure 9.10, which are unified into one case by our relabeling. The trinode restructure then replaces z with the node called b, makes the children of this node be a and c, and makes the children of a and c be the four previous children of x, y, and z (other than x and y) while maintaining the inorder relationships of all the nodes in T.

```
Algorithm restructure($x$):
  Input: A node $x$ of a binary search tree $T$ that has both a parent $y$ and a grand-
      parent $z$
  Output: Tree $T$ after a trinode restructuring (which corresponds to a single or
      double rotation) involving nodes $x$, $y$, and $z$
1: Let $(a,b,c)$ be a left-to-right (inorder) listing of the nodes $x$, $y$, and $z$, and let
   $(T_0,T_1,T_2,T_3)$ be a left-to-right (inorder) listing of the four subtrees of $x$, $y$, and
   $z$ not rooted at $x$, $y$, or $z$.
2: Replace the subtree rooted at $z$ with a new subtree rooted at $b$.
3: Let $a$ be the left child of $b$ and let $T_0$ and $T_1$ be the left and right subtrees of $a$,
   respectively.
4: Let $c$ be the right child of $b$ and let $T_2$ and $T_3$ be the left and right subtrees of $c$,
   respectively.
```

Code Fragment 9.7: The trinode restructure operation in a binary search tree.

The modification of a tree T caused by a trinode restructure operation is often called a ***rotation***, because of the geometric way we can visualize the way it changes T. If $b = y$, the trinode restructure method is called a ***single rotation***, for it can be visualized as "rotating" y over z. (See Figure 9.10a and b.) Otherwise, if $b = x$, the trinode restructure operation is called a ***double rotation***, for it can be visualized as first "rotating" x over y and then over z. (See Figure 9.10c and d, and Figure 9.9.) Some computer researchers treat these two kinds of rotations as separate functions, each with two symmetric types. We have chosen, however, to unify these four types of rotations into a single trinode restructure operation. No matter how we view it, however, note that the trinode restructure function modifies parent-child relationships of $O(1)$ nodes in T, while preserving the inorder traversal ordering of all the nodes in T.

In addition to its order-preserving property, a trinode restructuring changes the heights of several nodes in T, in order to restore balance. Recall that we execute the function restructure(x) because z, the grandparent of x, is unbalanced. Moreover, this unbalance is due to one of the children of x now having too large a height relative to the height of z's other child. As a result of a rotation, we move the "tall" child of x up while pushing the "short" child of z down. Thus, after performing restructure(x), all the nodes in the subtree now rooted at the node we called b are balanced. (See Figure 9.10.) Thus, we restore the height-balance property ***locally*** at the nodes x, y, and z. In addition, since after performing the new item insertion the subtree rooted at b replaces the one formerly rooted at z, which was taller by one unit, all the ancestors of z that were formerly unbalanced become balanced. (See Figure 9.9.) (The justification of this fact is left as Exercise C-9.13.) Therefore, this one restructuring also restores the height-balance property ***globally***.

nd a grand-

a single or

nd z, and let
s of x, y, and

subtrees of a,

subtrees of c,

search tree.

ration is often
way it changes
n, for it can be
se, if $b = x$, the
e visualized as
nd Figure 9.9.)
arate functions,
nify these four
matter how we
ies parent-child
ersal ordering of

ring changes the
at we execute the
nced. Moreover,
o large a height
e move the "tall"
after performing
le we called b are
e property ***locally***
ew item insertion
was taller by one
ne balanced. (See
-9.13.) Therefore,
lobally.

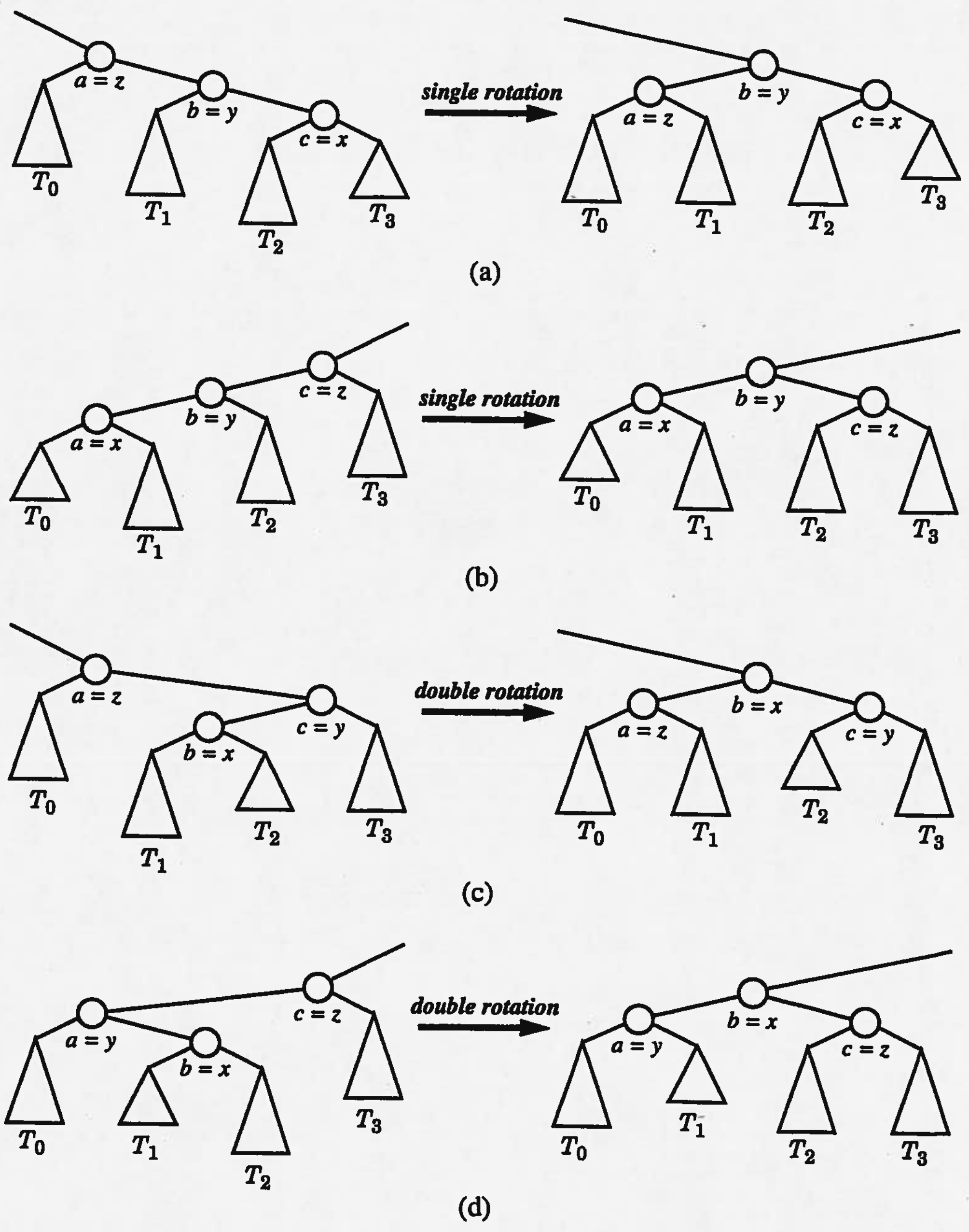

Figure 9.10: Schematic illustration of a trinode restructure operation (Code Fragment 9.7). Parts (a) and (b) show a single rotation, and parts (c) and (d) show a double rotation.

Removal

As was the case for insertItem(), we begin the implementation of the dictionary operation removeElement(k) on an AVL tree T by using the algorithm for performing this operation on a regular binary search tree (Section 9.1.2). The added difficulty in using this approach with an AVL tree is that it may violate the height-balance property. In particular, after removing an internal node with operation removeAboveExternal() and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent w of the previously removed node to the root of T. (See Figure 9.11a.) In fact, there can be at most one such unbalanced node. (The justification of this fact is left as Exercise C-9.10.)

As with insertion, we use trinode restructuring to restore balance in the tree T. In particular, let z be the first unbalanced node encountered going up from w toward the root of T. Also, let y be the child of z with larger height (note that node y is the child of z that is not an ancestor of w), and let x be a child of y with largest height. The choice of x may not be unique, since the subtrees of y may have the same height. If there is such a tie, x should be chosen to be on the same side relative to y as y is relative to z. In other words, both x and y are right children or both x and y are left children. (We leave the justification of this tie-breaking rule as Exercise C-9.11.) In any case, we then perform a restructure(x) operation, which restores the height-balance property ***locally***, at the subtree that was formerly rooted at z and is now rooted at the node we temporarily called b. (See Figure 9.11b.)

Unfortunately, this trinode restructuring may reduce the height of the subtree rooted at b by 1, which may cause an ancestor of b to become unbalanced. Thus, a single trinode restructuring does not necessarily restore the height-balance property globally after a removal. So, after rebalancing z, we continue walking up T looking for unbalanced nodes. If we find another, we perform a restructure operation to restore its balance, and continue up T looking for more, all the way to the root. Since the height of T is $O(\log n)$, where n is the number of items, by Proposition 9.2, $O(\log n)$ trinode restructurings are sufficient to restore the height-balance property.

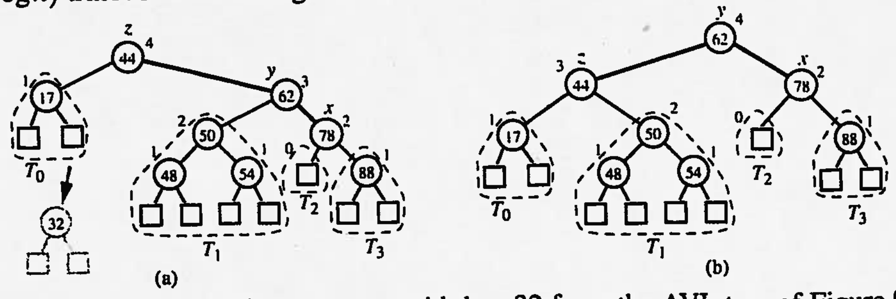

Figure 9.11: Removal of the element with key 32 from the AVL tree of Figure 9.8: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

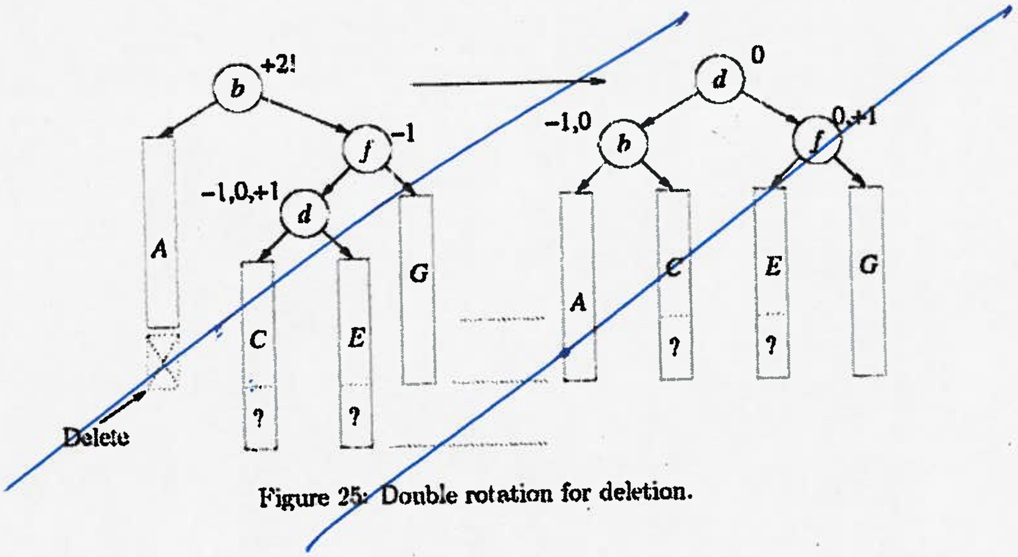

Figure 25: Double rotation for deletion.

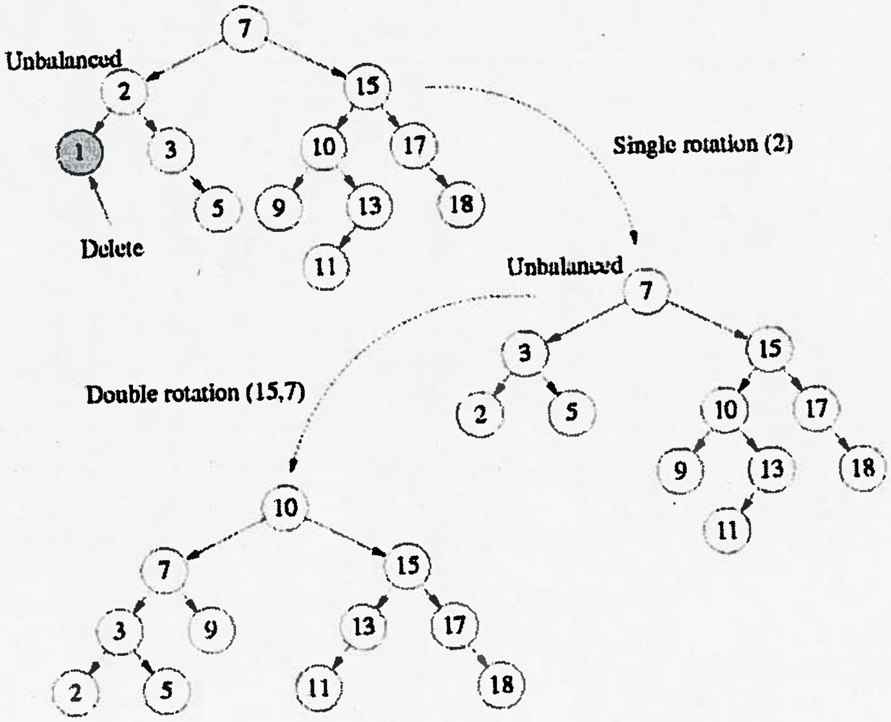

Figure 26: AVL Deletion example.