Chapter 4 Balanced Binary Search Trees

U.S. Navy Blue Angels, performing their delta formation during the Blues on the Bay Air Show at Marine Corps Base Hawaii in 2007. U.S. government photo by Petty Officer 2nd Class Michael Hight, U.S. Navy.

Contents

4.1 Ranks and Rotations **117**
4.2 AVL Trees **120**
4.3 Red-Black Trees **126**
4.4 Weak AVL Trees **130**
4.5 Splay Trees **139**
4.6 Exercises **149**

Real-time systems are computational platforms that have real-time constraints, where computations must complete in a given amount of time. Examples include antilock braking systems on cars, video and audio processing systems, operating systems kernels, and web applications. In these real-time applications, if a software component takes too much time to finish, then the entire system can crash (sometimes literally).

Suppose, then, that you are designing a real-time web application for users to find A-list celebrities who are closest to them in age. In other words, your system should maintain a set of celebrities, sorted by their birth dates. In addition, it should allow for celebrities to be added to your set (namely, when they become popular enough to be added to the A-list) and removed from your set (say, when they fall off the A-list).

Most importantly, your system should support a ***nearest-neighbor query***, where a user specifies their birth date and your system then returns the ten A-list stars closest in age to the user. The real-time constraint for your system is that it has to respond in at most a few hundred milliseconds or users will notice the delay and go to your competitor. Of course, if users would simply be looking up celebrities with a specific birth date, you could use a lookup table, indexed by birth date, to implement your database. But such schemes don't support fast nearest neighbor queries.

Note that a binary search tree, T, provides almost everything you need in order to implement your system, since it can maintain a sorted set of items so as to perform insertions and removals based on their keys (which in this case are birth dates). It also supports nearest-neighbor queries, in that, for any key k, we can perform a search in T for the smallest key that is greater than or equal to k, or, alternatively, for the largest key that is less than or equal to k. Given either of the nodes in T storing such a key, we can then perform a forward or backward inorder traversal of T starting from that point to list neighboring smaller or larger keys.

The problem is that without some way of limiting the height of T, the worst-case running time for performing searches and updates in T can be linear in the number of items it stores. Indeed, this worst-case behavior occurs if we insert and delete keys in T in a somewhat sorted order, which is likely for your database, since celebrities are typically added to the A-list when they are in their mid-twenties and removed when they are in their mid-fifties. Without a way to restructure T while you are using it, this kind of updating will result in T becoming unbalanced, which will result in poor performance for searches and updates in your system.

Fortunately, there is a solution. Namely, as we discuss in this chapter, there are ways of restructuring a binary search tree while it is being used so that it can guarantee logarithmic-time performance for searches and updates. These restructuring methods result in a class of data structures known as ***balanced binary search trees***.

4.1 Ranks and Rotations

Recall that a binary search tree stores elements at the internal nodes of a proper binary tree so that the key at each left child is not greater than its parent's key and the key at each right child is not less than its parent's key. Searching in a binary search tree can be described as a recursive procedure, as we did in the previous chapter (in Algorithm 3.5), or as an iterative method, as we show in Algorithm 4.1.

```
Algorithm IterativeTreeSearch(k, T):
    Input: A search key k and a binary search tree, T
    Output: A node in T that is either an internal node storing key k or the external
        node where an item with key k would belong in T if it existed
    v ← T.root()
    while v is not an external node do
        if k = key(v) then
            return v
        else if k < key(v) then
            v ← T.leftChild(v)
        else
            v ← T.rightChild(v)
    return v
```

Algorithm 4.1: Searching a binary search tree iteratively.

As mentioned above, the worst-case performance of searching in a binary search tree can be as bad as linear time, since the time to perform a search is proportional to the height of the search tree. Such a performance is no better than that of looking through all the elements in a set to find an item of interest. In order to avoid this poor performance, we need ways of maintaining the height of a search tree to be logarithmic in the number of nodes it has.

Balanced Binary Search Trees

The primary way to achieve logarithmic running times for search and update operations in a binary search tree, T, is to perform restructuring actions on T based on specific rules that maintain some notion of "balance" between sibling subtrees in T. We refer to a binary search tree that can maintain a height of $O(\log n)$ through such balancing rules and actions as a ***balanced binary search tree***. Intuitively, the reason balance is so important is that when a binary search tree T is balanced, the number of nodes in the tree increases exponentially as one moves down the levels of T. Such an exponential increase in size implies that if T stores n items, then it will have height $O(\log n)$.

In this chapter, we discuss several kinds of balanced binary search trees. Three of these search trees—AVL trees, red-black trees, and weak AVL trees—are ***rank-balanced trees***, where we define an integer ***rank***, $r(v)$, for each node, v, in a binary search tree, T, where $r(v)$ is either the height of v or a value related to the height of v. Balance in such a tree, T, is enforced by maintaining certain rules on the relative ranks of children and sibling nodes in T. These three rank-balanced trees have slightly different rules guaranteeing that the height of a binary search tree satisfying these rules has logarithmic height. The final type of balanced binary search tree we discuss in this chapter is the splay tree, which achieves its balance in an amortized way by blindly performing a certain kind of restructuring action, called splaying, after every access and update operation.

One restructuring operation, which is used in all of the balanced binary search trees we discuss is known as a ***rotation***, of which there are four types. Here, we describe a unified restructuring operation, called ***trinode restructuring***, which combines the four types of rotations into one action. The trinode restructuring operation involves a node, x, which has a parent, y, and a grandparent, z. This operation, restructure(x), is described in detail in Algorithm 4.2 and illustrated in Figure 4.3. At a high level, 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 4.3, 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 grandparent 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 that are 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.
5: Recalculate the heights of a, b, and c, (or a "standin" function for height), from the corresponding values stored at their children, and return b.

Algorithm 4.2: The trinode restructure operation for a binary search tree.

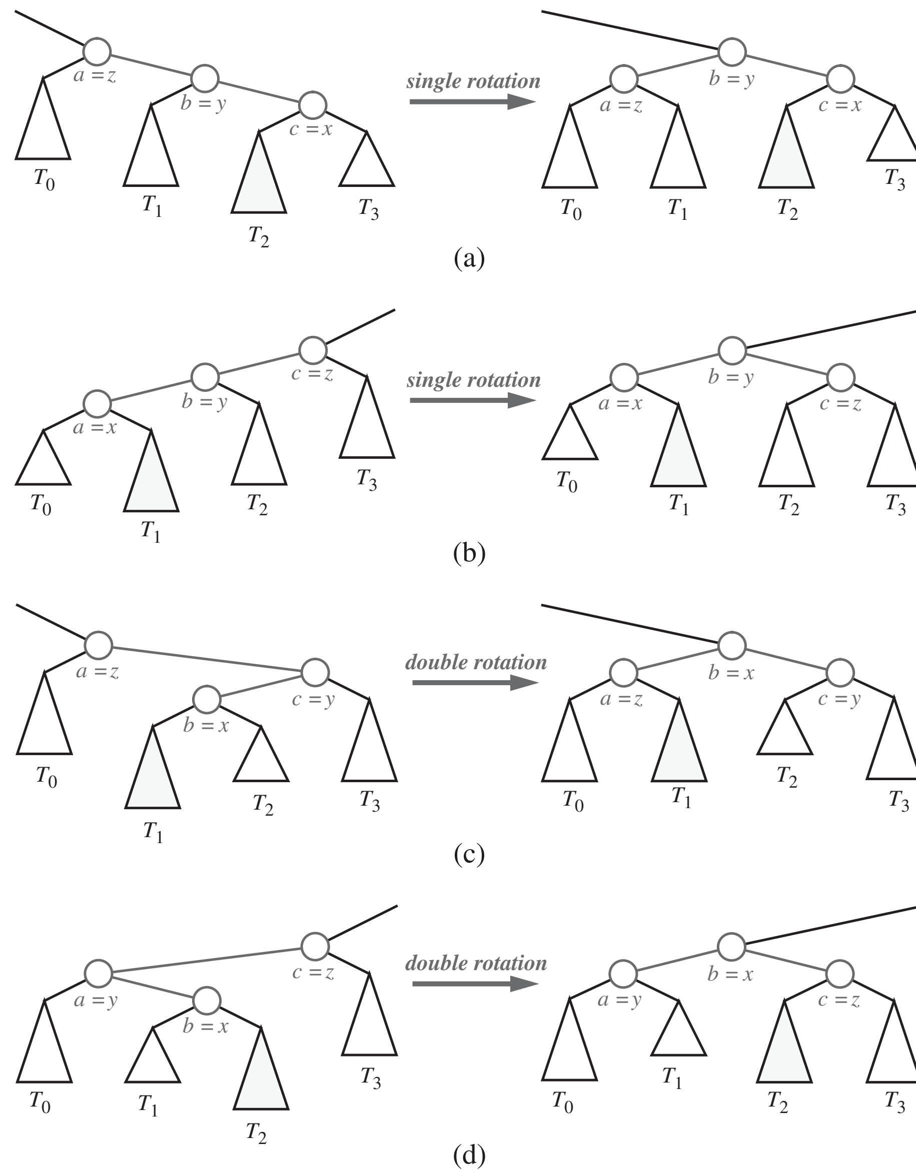

Figure 4.3: Schematic illustration of a trinode restructure operation (Algorithm 4.2). Parts (a) and (b) show a single rotation, and parts (c) and (d) show a double rotation.

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 $b = y$ (see Algorithm 4.2), the trinode restructure method is called a ***single rotation***, for it can be visualized as "rotating" y over z. (See Figure 4.3a and b.) Otherwise, if $b = x$, the trinode restructure operation is a ***double rotation***, for it can be visualized as first "rotating" x over y and then over z. (See Figure 4.3c and d.)

4.2 AVL Trees

The first rank-balanced search tree we discuss is the ***AVL tree***, which is named after its inventors, Adel'son-Vel'skii and Landis, and is also the oldest known balanced search tree, having been invented in 1962. In this case, we define the rank, $r(v)$, of a node, v, in a binary tree, T, simply to be the height of v in T. The rank-balancing rule for AVL trees is then defined as follows:

***Height-balance Property*:** For every internal node, v, in T, the heights of the children of v may differ by at most 1. That is, if a node, v, in T has children, x and y, then $|r(x) - r(y)| \leq 1$.

(See Figure 4.4.)

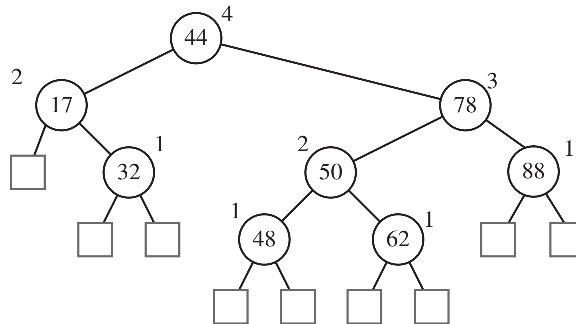

Figure 4.4: An AVL tree. Heights are shown next to the nodes.

An immediate consequence of the height-balance property is that any 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.

Theorem 4.1: *The height of an AVL tree, T, storing n items is $O(\log n)$.*

Proof: Instead of trying to find an upper bound for the height of an AVL tree directly, let us instead concentrate on the "inverse problem" of characterizing the minimum number of internal nodes, n_h, of an AVL tree with height h. As base cases for a recursive definition, notice that $n_1 = 1$, because an AVL tree of height 1 must have at least one internal node, and $n_2 = 2$, because an AVL tree of height 2 must have at least two internal nodes. Now, for the general case of an AVL tree, T, with the minimum number of nodes for height, h, note that the root of such a tree will have as its children's subtrees an AVL tree with the minimum number of nodes for height $h - 1$ and an AVL tree with the minimum number of nodes for height $h - 2$. Taking the root itself into account, we obtain the following formula

for the general case, $h \geq 3$:

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

This formula implies that the n_h values are strictly increasing as h increases, in a way that corresponds to the Fibonacci sequence (e.g., see Exercise C-4.3). In other words, $n_{h-1} > n_{h-2}$, for $h \geq 3$, which allows us to simplify the above formula as

$$n_h > 2n_{h-2}.$$

This simplified formula shows that n_h at least doubles each time h increases by 2, which intuitively means that n_h grows exponentially. Formally, this simplified formula implies that

$$n_h \;>\; 2^{\frac{h}{2}-1}. \tag{4.1}$$

By taking logarithms of both sides of Equation (4.1), we obtain

$$\log n_h \;>\; \frac{h}{2} - 1,$$

from which we get

$$h \;<\; 2\log n_h + 2, \tag{4.2}$$

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

In fact, the bound of $2\log n + 2$, from Equation (4.2), for the height of an AVL tree is an overestimate. It is possible, for instance, to show that the height of an AVL tree storing n items is at most $1.441\log(n+1)$, as is explored, for instance, in Exercise C-4.4. In any case, by Theorem 4.1 and the analysis of binary search trees given in Section 3.1.1, searching in an AVL tree runs in $O(\log n)$ time. The important issue remaining is to show how to maintain the height-balance property of an AVL tree after an insertion or removal.

Insertion

An insertion in an AVL tree T begins as in an insert operation described in Section 3.1.3 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.

In the AVL tree approach to achieving balance, given a binary search tree, T, we say that a 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 4.5a.) 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.

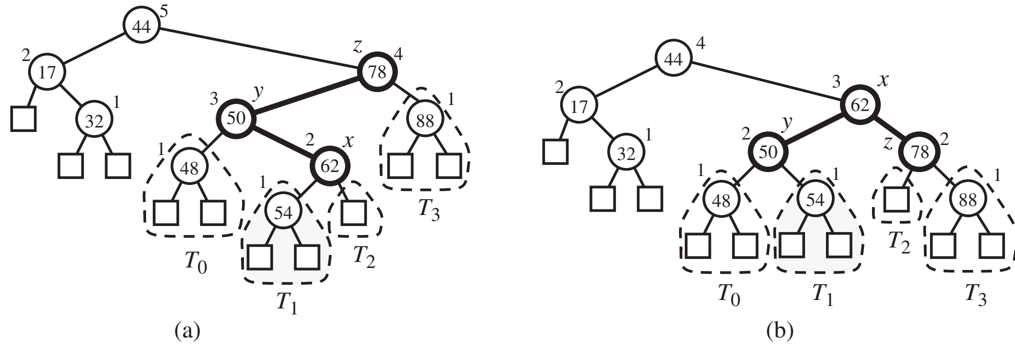

Figure 4.5: An example insertion of an element with key 54 in the AVL tree of Figure 4.4: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify the nodes x, y, and z.

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 4.5a.) 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 could be equal to w and x is a grandchild of z. Since z became 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*** method, restructure(x), described in Algorithm 4.2. (See Figure 4.5b.)

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. (The justification of this fact is left as Exercise C-4.9.) Therefore, this one restructuring also restores the height-balance property ***globally***. That is, one rotation (single or double) is sufficient to restore the height-balance in an AVL tree after an insertion. Of course, we may have to update the height values (ranks) of $O(\log n)$ nodes after an insertion, but the amount of structural changes after an insertion in an AVL tree is $O(1)$.

Removal

We begin the implementation of the remove operation on an AVL tree T as in a regular binary search tree. We may have some additional work, however, if this update violates 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 4.6a.) In fact, there can be one such unbalanced node at most. (The justification of this fact is left as Exercise C-4.8.)

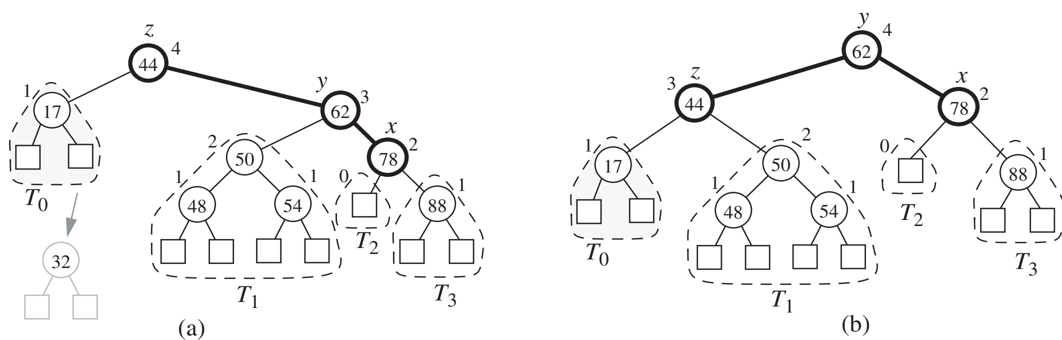

Figure 4.6: Removal of the element with key 32 from the AVL tree of Figure 4.4: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

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). Finally, let x be the child of y defined as follows:

- if one of the children of y is taller than the other, let x be the taller child of y;
- else (both children of y have the same height), let x be the child of y on the same side as y (that is, if y is a left child, let x be the left child of y, else let x be the right child of y).

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 4.6b.)

This trinode restructuring may reduce the height of the subtree rooted at b by 1, which may cause in turn an ancestor of b to become unbalanced. Thus, a single trinode restructuring may not 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 unbalanced node, we perform a restructure operation to restore its balance, and continue marching 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 Theorem 4.1, $O(\log n)$ trinode restructurings are sufficient to restore the height-balance property.

Pseudo-code for AVL Trees

Pseudo-code descriptions of the metods for performing the insertion and removal operations in an AVL tree are given in Algorithm 4.7.

We also include a common rebalancing method, rebalanceAVL, which restores the balance to an AVL tree after performing either an insertion or a removal.

This method, in turn, makes use of the trinode restructure operation to restore local balance to a node after an update. The rebalanceAVL method continues testing for unbalance up the tree, and restoring balance to any unbalanced nodes it finds, until it reaches the root.

Summarizing the Analysis of AVL Trees

We summarize the analysis of the performance of AVL trees as follows. (See Table 4.8.) Operations find, insert, and remove visit the nodes along a root-to-leaf path of T, plus, possibly their siblings, and spend $O(1)$ time per node. The insertion and removal methods perform this path traversal twice, actually, once down this path to locate the node containing the update key and once up this path after the update has occurred, to update height values (ranks) and do any necessary rotations to restore balance. Thus, since the height of T is $O(\log n)$ by Theorem 4.1, each of the above operations takes $O(\log n)$ time. That is, we have the following theorem.

Theorem 4.2: *An AVL tree for n key-element items uses $O(n)$ space and executes the operations* find, insert *and* remove *to each take $O(\log n)$ time.*

Operation	Time	Structural Changes
find	$O(\log n)$	none
insert	$O(\log n)$	$O(1)$
remove	$O(\log n)$	$O(\log n)$

Table 4.8: Performance of an n-element AVL tree. The space usage is $O(n)$.

Algorithm insertAVL(k, e, T):
Input: A key-element pair, (k, e), and an AVL tree, T
Output: An update of T to now contain the item (k, e)

$v \leftarrow$ IterativeTreeSearch(k, T)
if v is not an external node **then**
  return "An item with key k is already in T"
Expand v into an internal node with two external-node children
v.key $\leftarrow k$
v.element $\leftarrow e$
v.height $\leftarrow 1$
rebalanceAVL(v, T)

Algorithm removeAVL(k, T):
Input: A key, k, and an AVL tree, T
Output: An update of T to now have an item (k, e) removed

$v \leftarrow$ IterativeTreeSearch(k, T)
if v is an external node **then**
  return "There is no item with key k in T"
if v has no external-node child **then**
  Let u be the node in T with key nearest to k
  Move u's key-value pair to v
  $v \leftarrow u$
Let w be v's smallest-height child
Remove w and v from T, replacing v with w's sibling, z
rebalanceAVL(z, T)

Algorithm rebalanceAVL(v, T):
Input: A node, v, where an imbalance may have occurred in an AVL tree, T
Output: An update of T to now be balanced

v.height $\leftarrow 1 + \max\{v$.leftChild().height, v.rightChild().height$\}$
while v is not the root of T **do**
  $v \leftarrow v$.parent()
  if $|v$.leftChild().height $-$ v.rightChild().height$| > 1$ **then**
    Let y be the tallest child of v and let x be the tallest child of y
    $v \leftarrow$ restructure(x) // trinode restructure operation
  v.height $\leftarrow 1 + \max\{v$.leftChild().height, v.rightChild().height$\}$

Algorithm 4.7: Methods for item insertion and removal in an AVL tree, as well as the method for rebalancing an AVL tree. This version of the rebalance method does not include the heuristic of stopping as soon as balance is restored, and instead always performs any needed rebalancing operations all the way to the root.