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

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
  $v \leftarrow T$.root()
  while $v$ is not an external node do
      if $k =$ key($v$) then
          return $v$
      else if $k <$ key($v$) then
          $v \leftarrow T$.leftChild($v$)
      else
          $v \leftarrow T$.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 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 $z$ 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.

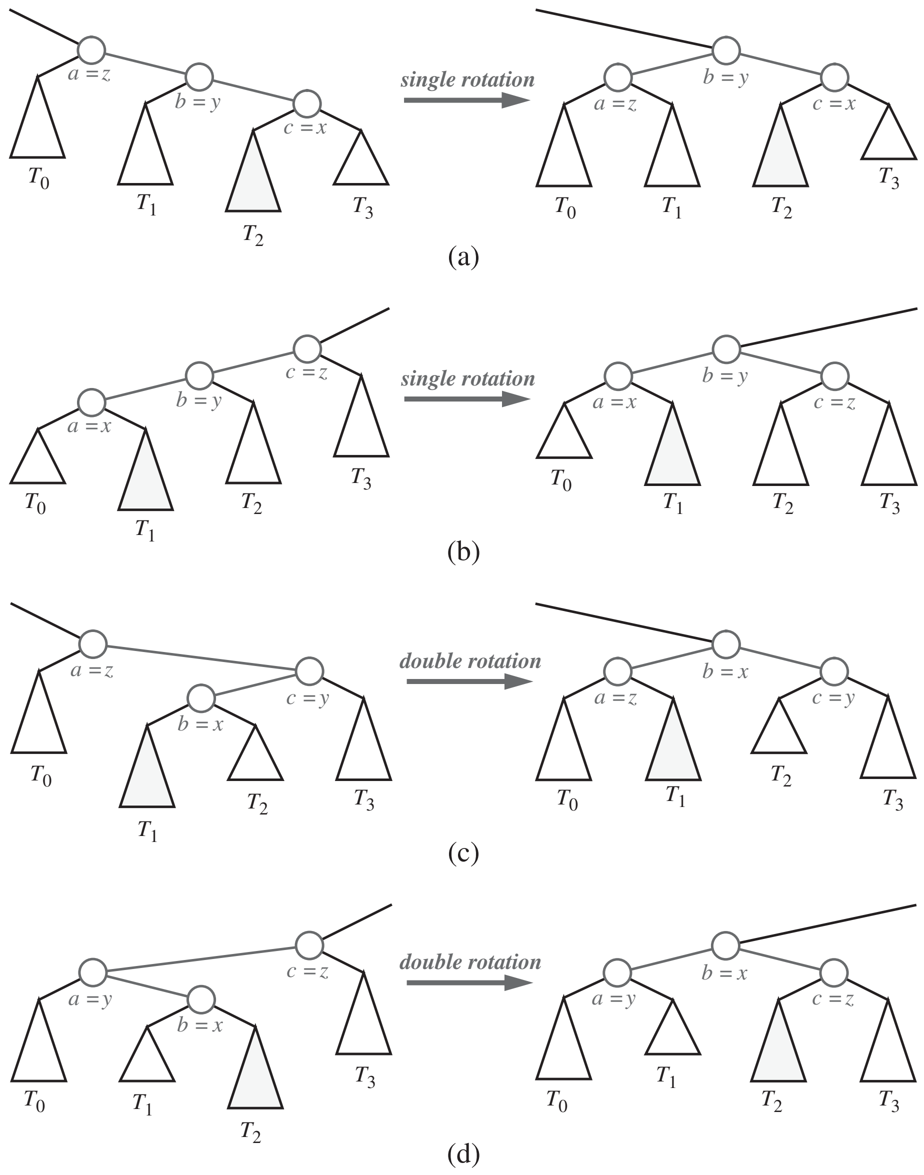


Figure 4.3: Schematic illustration of a trinode restructure operation (Algorithm 4.2). Parts (a) and (b) show a single rotation, and parts (c) and (d) show a double rotation.

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.)

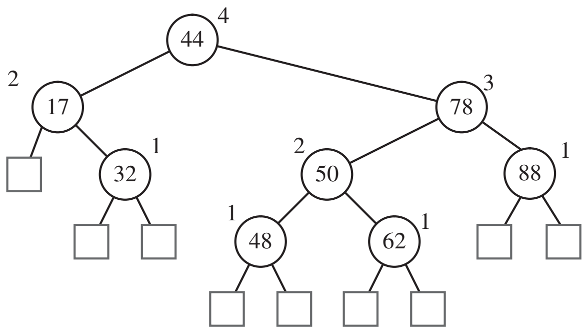


Figure 4.4: An AVL tree. Heights are shown next to the nodes.

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.

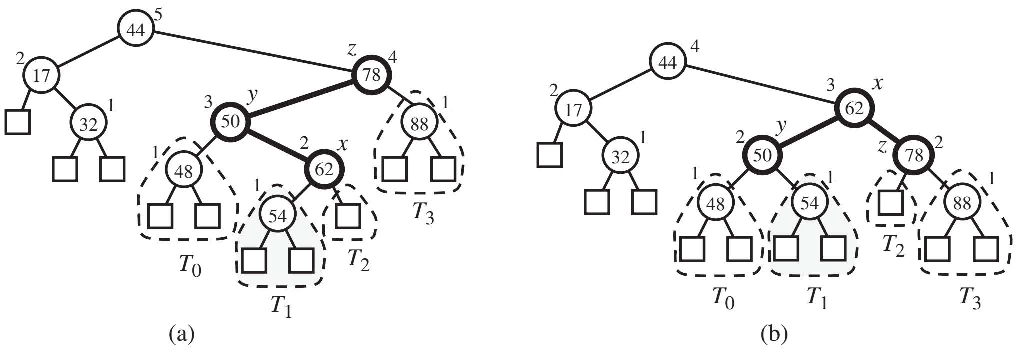


Figure 4.5: An example insertion of an element with key 54 in the AVL tree of Figure 4.4: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify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

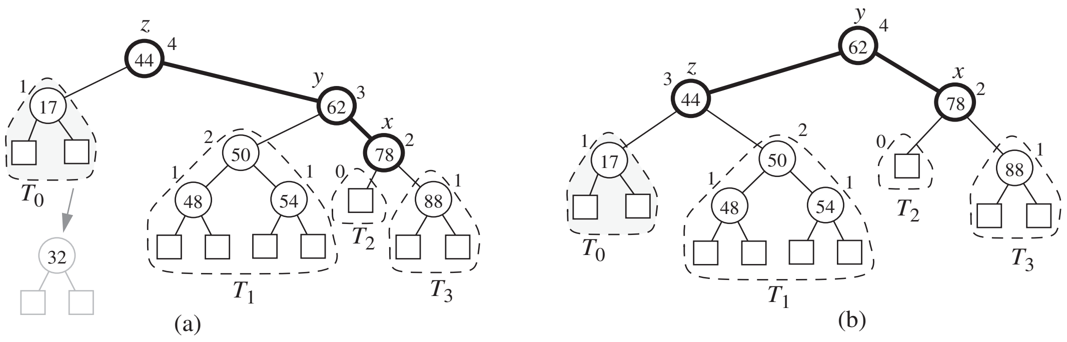


Figure 4.6: Removal of the element with key 32 from the AVL tree of Figure 4.4: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

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.

4.3 Red-Black Trees

The data structure we discuss in this section, the red-black tree, is a binary search tree that uses a kind of “pseudo-depth” to achieve balance using the approach of a depth-bounded search tree. In particular, a ***red-black tree*** is a binary search tree with nodes colored red and black in a way that satisfies the following properties:

***External-Node Property*:** Every external node is black.

***Internal-node Property*:** The children of a red node are black.

Black-depth Property*:** All the external nodes have the same ***black depth, that is, the same number of black nodes as proper ancestors.

Some definitions of red-black trees also require that the root of a red-black tree be black. An example of a red-black tree is shown in Figure 4.9.

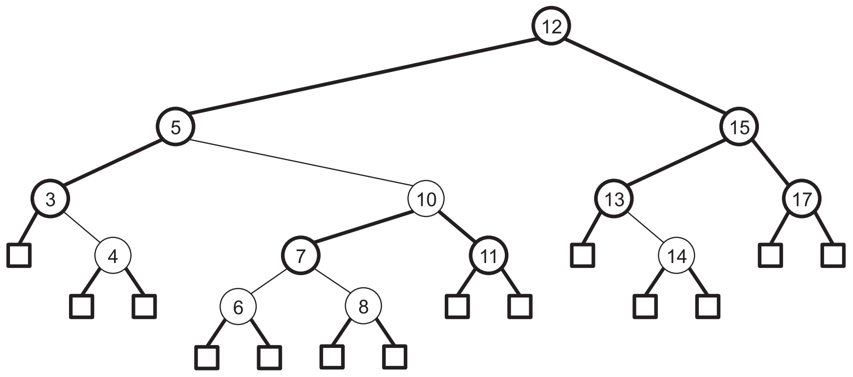


Figure 4.9: An example binary search tree that is a red-black tree. We used thin lines to denote red nodes and the edges from red nodes to their parents. Each external node of this red-black tree has three black proper ancestors; hence, each such node has black depth 3.

As is our convention in this book, we assume that items are stored in the internal nodes of a red-black tree, with the external nodes being empty placeholders. That is, we assume that red-black trees are proper binary search trees, so that every internal node has exactly two children. This allows us to describe search and update algorithms assuming external nodes are real, but we note in passing that at the expense of slightly more complicated search and update methods, external nodes could be **null** or references to a NULL_NODE object, as long as we can still imagine them as existing.

The reason behind all these properties that define red-black trees is that they lead to the following.

Theorem 4.3: *The height of a red-black tree storing n items is $O(\log n)$.*

Proof: Let n_d denote the minimum number of internal nodes in a red-black tree where all its external nodes have black depth d. Then

- $n_1 = 1$, since we could have a single internal node with two black external-node children.
- $n_d = 1 + 2n_{d-1}$, since we get the fewest number of internal nodes when we have a black root with two black children.

This implies that $n_d = 2^d - 1$, that is, $d = \log(n_d + 1)$. Now, let T be a red-black tree storing n items (in its internal nodes), and let d be the common black depth of each external node in T and let h be the height of T. By the definition of n_d, $d \le \log(n+1)$. Notice that the internal-node property of red-black trees implies that it is impossible to have two consecutive red nodes on any root-to-external-node path. Thus, if an external node, v, has black depth d, then the actual depth of v can be at most $2d + 1$ (since the root is allowed to be red). Therefore,

$$h \le 2d + 1 \le 2\log(n+1) + 1.$$

■

We assume that a red-black tree is realized with a linked structure for binary trees, in which we store an item and a color indicator at each node. Moreover, this color indicator only needs to be a single bit, since there are only two possible colors—red or black.

The algorithm for searching in a red-black tree T is the same as that for a standard binary search tree. That is, we traverse down T to either an internal node holding an item with key equal to the search key or to an external node (without finding an item with the desired key). Thus, searching in a red-black tree takes $O(\log n)$ time.

Performing the update operations in a red-black tree is similar to that of a binary search tree, except that we must additionally restore the color properties. Unfortunately, the methods for doing this restoration are a bit complicated when dealing directly with node colors; hence, as a first step toward simpler algorithms for performing updates in a red-black tree, we provide an alternative definition of red-black trees, which is equivalent to the standard definition given above.

A Rank-Based Definition of Red-Black Trees

One of the biggest challenges in designing algorithms for red-black trees is that any change to a red-black tree has to maintain the black-depth property, which is somewhat difficult to work with. Thus, it is useful to have an alternative definition of red-black trees that avoids an explicit black-depth property.

So, instead of assigning colors to the nodes of a binary search tree, T, let us instead assign an integer, $r(v)$, to each node, v, in T, which we refer to as the ***rank*** of v. In order for an assignment of ranks to be valid, the rank of any node can never be greater than the rank of its parent. For each node v in T other than the root, we define the ***rank difference*** of v as the difference between the rank of v and the rank of v's parent.

We say that an assignment of ranks to the nodes of a binary search tree is ***red-black-equivalent*** if it satisfies the following properties.

***External-Node Property*:** Every external node has rank 0 and its parent, if it exists, has rank 1.

***Parent Property*:** The rank difference of every node, other than the root, is 0 or 1.

***Grandparent Property*:** Any node with rank difference 0 is either a child of the root or its parent has rank difference 1.

We can show that this set of properties is equivalent to the standard properties for red-black trees. (See Figure 4.10.)

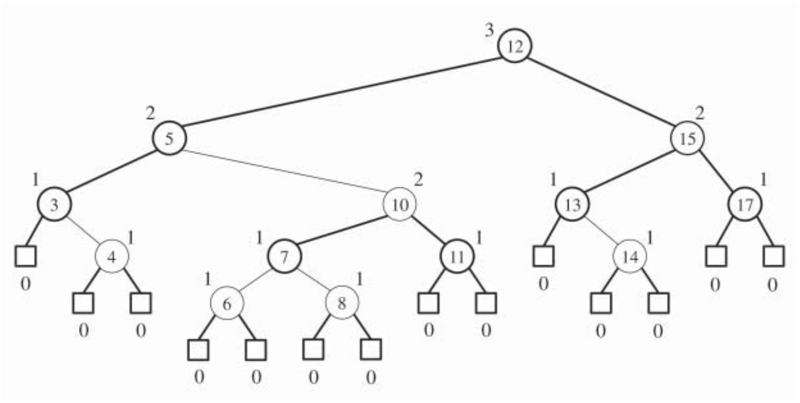


Figure 4.10: An example binary search tree that is a red-black tree, both by the color-based definition and the rank-based definition. We show the rank next to each node. Note that red nodes have rank difference equal to zero.

Theorem 4.4: *Every red-black tree has a red-black-equivalent rank assignment.*

Proof: Suppose T is a red-black tree. Without loss of generality, we can assume the root of T is black. Define the ***black height*** of each node v in T as the number of black descendants of v (not counting v) on a path from v to an external node. Note that the black-depth property implies that this notion is well-defined. Now, assign ranks to the nodes of T according to the following.

***Rank-Assignment Rule*:** Give each black node, v, in T a rank, $r(v)$, equal to its black height, and give each red node, v, a rank, $r(v)$, equal to the rank of its (black) parent.

Then this assignment of ranks satisfies all the properties for being red-black-equivalent. In particular, every external node will have rank 0 and a parent of rank 1, the rank differences of every node will be 0 or 1, and any node with rank difference 0 will have a parent with rank difference 1. ■

We also have the following result.

Theorem 4.5: *Every tree with a red-black-equivalent rank assignment can be colored as a red-black tree.*

Proof: Suppose T is a binary tree with a red-black-equivalent rank assignment. To come up with a coloring of T, use the following.

***Color-Assignment Rule*:** If a node has rank difference 0, then color it red; otherwise, color it black.

Thus, by the rules for a red-black-equivalent ranking, every external node will be colored black and every red node will have a black parent. To see that every external node has the same black depth, note that we can easily prove by induction that the black depth of every external node in a subtree of T rooted at v is equal to the black height of v, since every node has rank difference 0 or 1. Therefore, all the external nodes are at the same black depth from the root. ■

This rank-based definition of red-black trees is still not quite what we need to design simple update algorithms, but it is a good first step. In the next section, we provide an alternative rank-based balancing scheme, which results in balanced search trees that are structurally red-black trees but have improved performance and simpler update algorithms. Thus, we omit from this book a description of the complicated update algorithms for red-black trees. Let us, instead, take as a given that red-black trees can achieve $O(\log n)$-time performance for insertions and deletions.

Table 4.11 summarizes the running times of the main operations of a red-black tree. The main take-away message from this table is that a red-black tree achieves logarithmic worst-case running times for both searching and updating.

Operation	**Time**
find	$O(\log n)$
insert	$O(\log n)$
remove	$O(\log n)$

Table 4.11: Performance of an n-element red-black tree. The space usage is $O(n)$.

4.4 Weak AVL Trees

In this section, we discuss a type of rank-balanced trees, known as ***weak AVL trees*** or ***wavl trees***, that have features of both AVL trees and red-black trees. That is, as we will show, every AVL tree is a weak AVL tree, and every weak AVL tree can be colored as a red-black tree. Weak AVL trees achieve balance by enforcing rules about integer ranks assigned to nodes, which are a kind of "stand in" for height rather than having nodes have ranks exactly equal to their heights, as in AVL trees. Also, by using such ranks, wavl trees are able to achieve efficiency and have simple update methods.

In a ***weak AVL*** tree, T, we assign an integer rank, $r(v)$, to each node, v in T, such that the rank of any node is less than the rank of its parent. For each node v in T, the ***rank difference*** for v is the difference between the rank of v and the rank of v's parent. An internal node is a $1,1$***-node*** if its children each have rank difference 1. It is a $2,2$***-node*** if its children each have rank difference 2. And it is a $1,2$***-node*** if it has one child with rank difference 1 and one child with rank difference 2. The ranks assigned to the nodes in a wavl tree must satisfy the following properties.

***Rank-Difference Property*:** The rank difference of any non-root node is 1 or 2.

***External-Node Property*:** Every external node has rank 0.

***Internal-Node Property*:** An internal node with two external-node children cannot be a $2,2$-node.

Note that just by the rank-difference and external-node properties, the height of a wavl tree, T, is less than or equal to the rank of the root of T. (See Figure 4.12.)

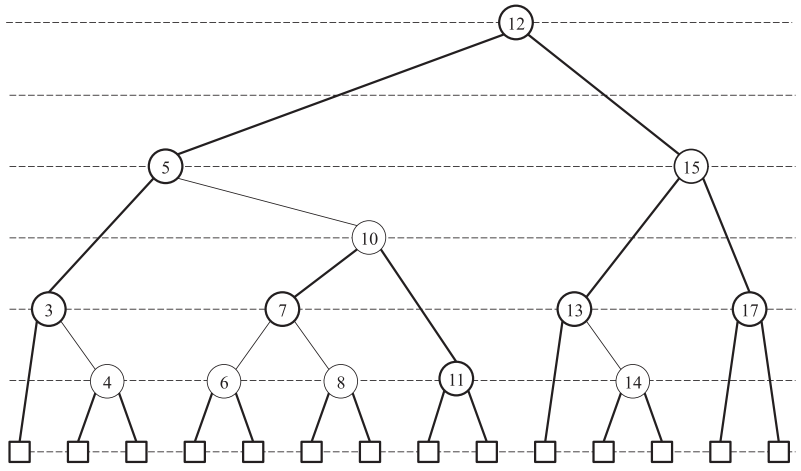


Figure 4.12: Example of a wavl tree. The nodes are placed at y-coordinates equal to their ranks.

Theorem 4.6: *The height of a wavl tree storing n items is at most $2\log(n+1)$.*

Proof: Let n_r denote the minimum number of internal nodes in a wavl tree whose root has rank r. Then, by the rules for ranks in a wavl tree,

$$\begin{aligned} n_0 &= 0 \\ n_1 &= 1 \\ n_2 &= 2 \\ n_r &= 1 + 2n_{r-2}, \text{ for } r \geq 3. \end{aligned}$$

This implies that $n_r \geq 2^{r/2} - 1$, that is, $r \leq 2\log(n_r + 1)$. Thus, by the definition of n_r, $r \leq 2\log(n+1)$. That is, the rank of the root is at most $2\log(n+1)$, which implies that the height of the tree is bounded by $2\log(n+1)$, since the height of a wavl tree is never more than the rank of its root. ■

Thus, wavl trees are balanced binary search trees. We also have the following.

Theorem 4.7: *Every AVL tree is a weak AVL tree.*

Proof: Suppose we are given an AVL tree, T, with a rank assignment, $r(v)$, for the nodes of T, as described in Section 4.2, so that $r(v)$ is equal to the height of v in T. Then, every external node in T has rank 0. In addition, by the height-balance property for AVL trees, every internal node is either a $1,1$-node or $1,2$-node; that is, there are no $2,2$-nodes. Thus, the standard rank assignment, $r(v)$, for an AVL tree implies T is a weak AVL tree. ■

Put another way, an AVL tree is just a weak AVL tree with no $2,2$-nodes, which motivates the name, "weak AVL tree." We also get a relationship that goes from wavl trees to red-black trees.

Theorem 4.8: *Every wavl tree can be colored as a red-black tree.*

Proof: Suppose we are given a wavl tree, T, with a rank assignment, $r(v)$. For each node v in T, assign a new rank, $r'(v)$, to each node v as follows:

$$r'(v) = \lfloor r(v)/2 \rfloor.$$

Then each external node still has rank 0 and the rank difference for any node is 0 or 1. In addition, note that the rank difference, in the $r(v)$ rank assignment, between a node and its grandparent must be at least 2; hence, in the $r'(v)$ rank assignment, the parent of any node with rank difference 0 must have rank difference 1. Thus, the $r'(v)$ rank assignment is red-black-equivalent; hence, by Theorem 4.5, T can be colored as a red-black tree. ■

Nevertheless, the relationship does not go the other way, as there are some red-black trees that cannot be given rank assignments to make them be wavl trees (e.g., see Exercise R-4.16). In particular, the internal-node property distinguishes wavl trees from red-black trees. Without this property, the two types of trees are equivalent, as Theorem 4.8 and the following theorem show.

Theorem 4.9: *Every red-black tree can be given a rank assignment that satisfies the rank-difference and external-node properties.*

Proof: Let T be a red-black tree. For each black node in v, let $r(v)$ be 2 times the black height of v. If v is red, on the other hand, let $r(v)$ be 1 more than the (same) rank of its black children. Then the rank of every external node is 0 and the rank difference for any node is 1 or 2. ■

Insertion

The insertion algorithm for weak AVL trees is essentially the same as for AVL trees, except that we use node ranks for wavl trees instead of node ranks that are exactly equal to node heights. Let q denote the node in a wavl tree where we just performed an insertion, and note that q previously was an external node. Now q has two external-node children; hence, we increase the rank of q by 1, which in an action called a ***promotion*** at q. We then proceed according to the following rebalancing operation:

- If q has rank difference 1 after promotion, or if q is the root, then we are done with the rebalancing operation. Otherwise, if q now has rank-difference 0, with its parent, p, then we proceed according to the following two cases.
  1. q's sibling has rank-difference 1. In this case, we promote q's parent, p. See Figure 4.13. This fixes the rank-difference property for q, but it may cause a violation for p. So, we replace q by p and we repeat the rebalancing operation for this new version of q.
  2. q's sibling has rank-difference 2. Let t denote a child of q that has rank-difference 1. By induction, such a child always exists. We perform a trinode restructuring operation at t by calling the method, restructure(t), described in Algorithm 4.2. See Figure 4.14. We then set the rank of the node replacing p (i.e., the one temporarily labeled b) with the old rank p and we set the ranks of its children so that they both have rank-difference 1. This terminates the rebalancing operation, since it repairs all rule violations without creating any new ones.

As mentioned above, this insertion algorithm is essentially the same as that for AVL trees. Moreover, it doesn't create any $2,2$-nodes. Thus, if we construct a weak AVL tree, T, via a sequence of n insertions and don't perform any deletions, then the height of T is the same as that for a similarly constructed AVL tree, namely, at most $1.441 \log(n+1)$. (See Exercise C-4.4.)

In addition, note that we perform at most $O(1)$ restructuring operations for any insertion in a weak AVL tree. Interestingly, unlike AVL trees, this same efficiency also holds for deletion in a wavl tree.

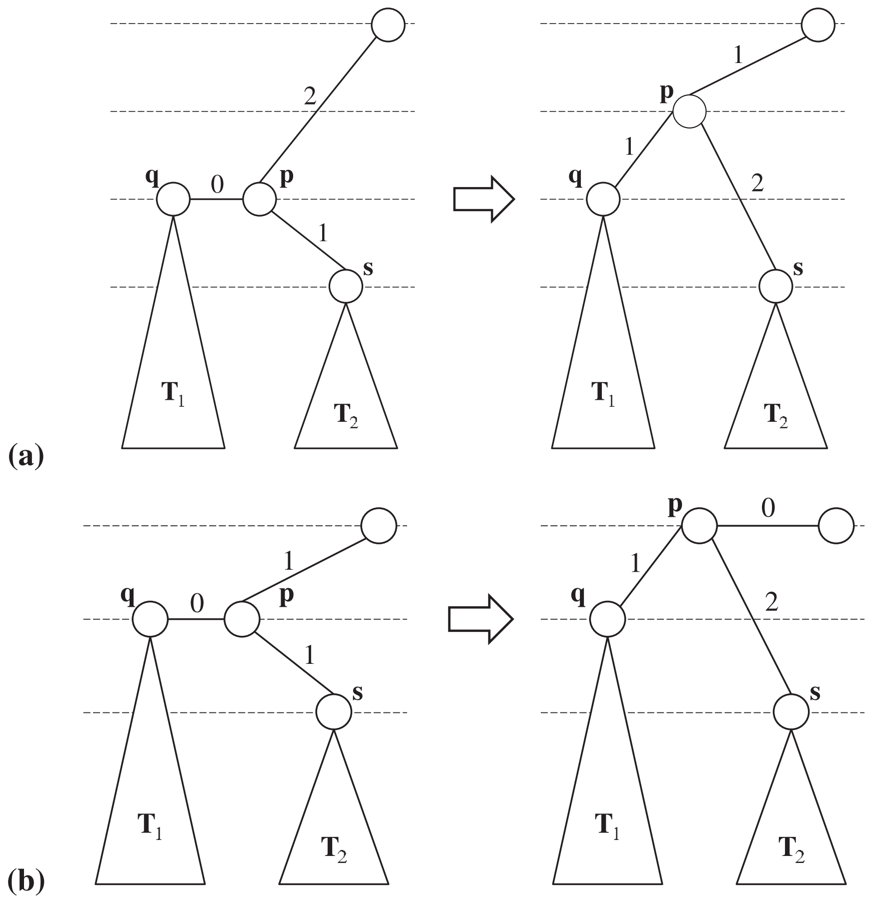
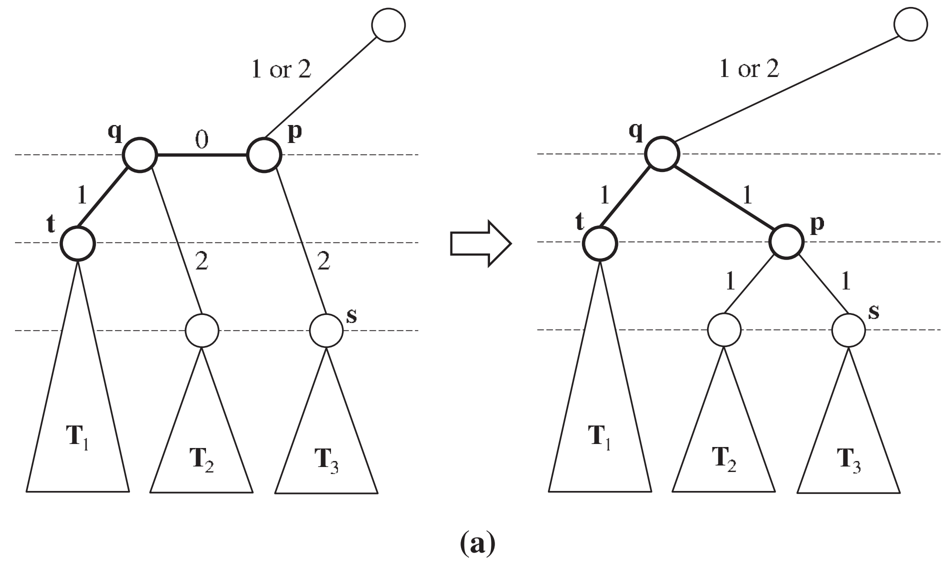


Figure 4.13: Case 1 of rebalancing in a wavl tree after an insertion: node q has rank-difference 0 (a violation) and its sibling, s, has rank-difference 1. To resolve the rank-difference violation for node q, we promote p, the parent of q. (a) If p had rank difference 2, we are done. (b) Else (p had rank-difference 1), we now have a violation at p.

Deletion

Let us next consider how to perform a deletion in a wavl tree T. Recall that a deletion in a binary search tree operates on a node v with an external-node child u and another child q (which may also be an external node). Note that if q is an internal node, q has rank 1 and v has rank 2. Else (q is an external node), q has rank 0 and v has rank 1.

To perform the deletion, nodes u and v are removed. If v was the root, then q becomes the new root. Otherwise, let p be the former parent of v. Node q now becomes a child of p. Note that p is either null or has rank 2, 3, or 4. Still, it



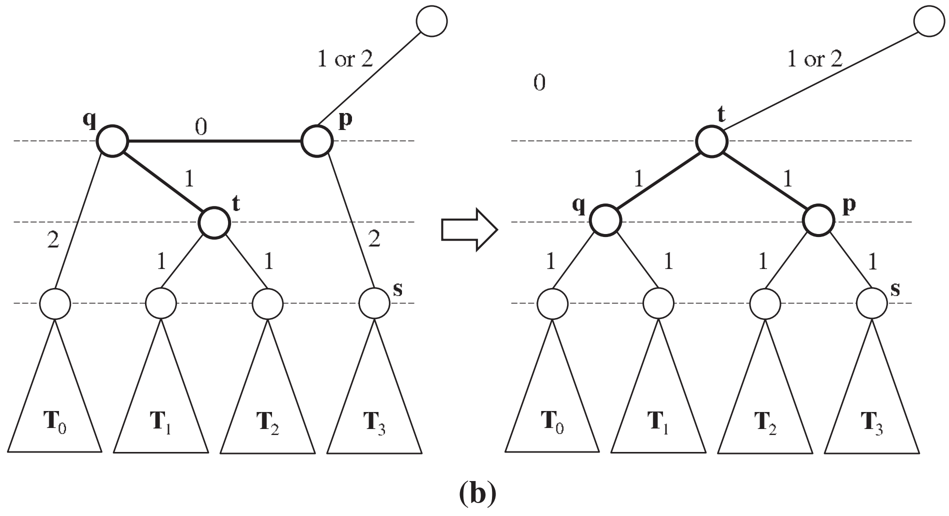


Figure 4.14: Case 2 of rebalancing in a wavl tree after an insertion: node q has rank-difference 0 (a violation) and its sibling, s, has rank-difference 2. To resolve the violation for node q, we perform a trinode restructuring operation at t.

cannot be the case that q has rank 0 and p has rank 4, since that would mean v had rank 2 with two rank-0 children, which is a violation of the internal-node property. Thus, after deletion, q has rank-difference 2 or 3 unless it is the root. If q has rank-difference 3, the deletion has caused a violation of the rank-difference property. To remedy this violation we perform a rebalancing operation. Let s be the sibling of q. We consider two cases:

1. If s has rank difference 2 with p, then we reduce the rank of p by 1, which is called a ***demotion***, as shown in Figure 4.15. The demotion of p repairs the violation of the rank-difference property at q. But it may case a violation of the rank-difference property at p. So, in this case, we relabel p as q, and we repeat the rebalancing operation.
2. Else (s has rank difference 1 with p), we consider two subcases:
   (a) Both children of s have rank-difference 2. In this case, we demote both p and s, as shown in Figure 4.16. These demotions repair the rank-difference violation at q (and avoid a violation at s). But they may create a rank-difference violation at p. So, in this case, we relabel p as q, and we repeat the rebalancing operation.
   (b) Node s has at least one child, t, with rank-difference 1. When both children of s have rank-difference 1, we pick t as follows: if s is a left child, then t is the left child of s, else t is the right child of s. In this case, we perform a trinode restructuring operation at t, by calling method, restructure(t), described in Algorithm 4.2. See Figure 4.17, which also shows how to set the ranks of p, s and t. These actions repair the rank-difference violation at q and do not create any new violations.

Thus, we can restore the balance after a deletion in a wavl tree after a sequence of at most $O(\log n)$ demotions followed by a single trinode restructuring operation.

Table 4.18 summarizes the performance of wavl trees, as compared to AVL trees and red-black trees (with black roots).

	AVL trees	**red-black trees**	**wavl trees**
$H(n)$	$1.441\log(n+1)$	$2\log(n+1)$	$2\log(n+1)$
$IH(n)$	$1.441\log(n+1)$	$2\log(n+1)$	$1.441\log(n+1)$
$IR(n)$	1	1	1
$DR(n)$	$O(\log n)$	2	1
search time	$O(\log n)$	$O(\log n)$	$O(\log n)$
insertion time	$O(\log n)$	$O(\log n)$	$O(\log n)$
deletion time	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 4.18: Comparing the performance of an n-element AVL tree, red-black tree, and wavl tree, respectively. Here, $H(n)$ denotes the worst-case height of the tree, $IH(n)$ denotes the worst-case height of the tree if it is built by doing n insertions (starting from an empty tree) and no deletions, $IR(n)$ denotes the worst-case number of trinode restructuring operations that are needed after an insertion, and $DR(n)$ denotes the worst-case number of trinode restructuring operations that are needed after performing a deletion in order to restore balance. The space usage is $O(n)$ for all of these balanced binary search trees.

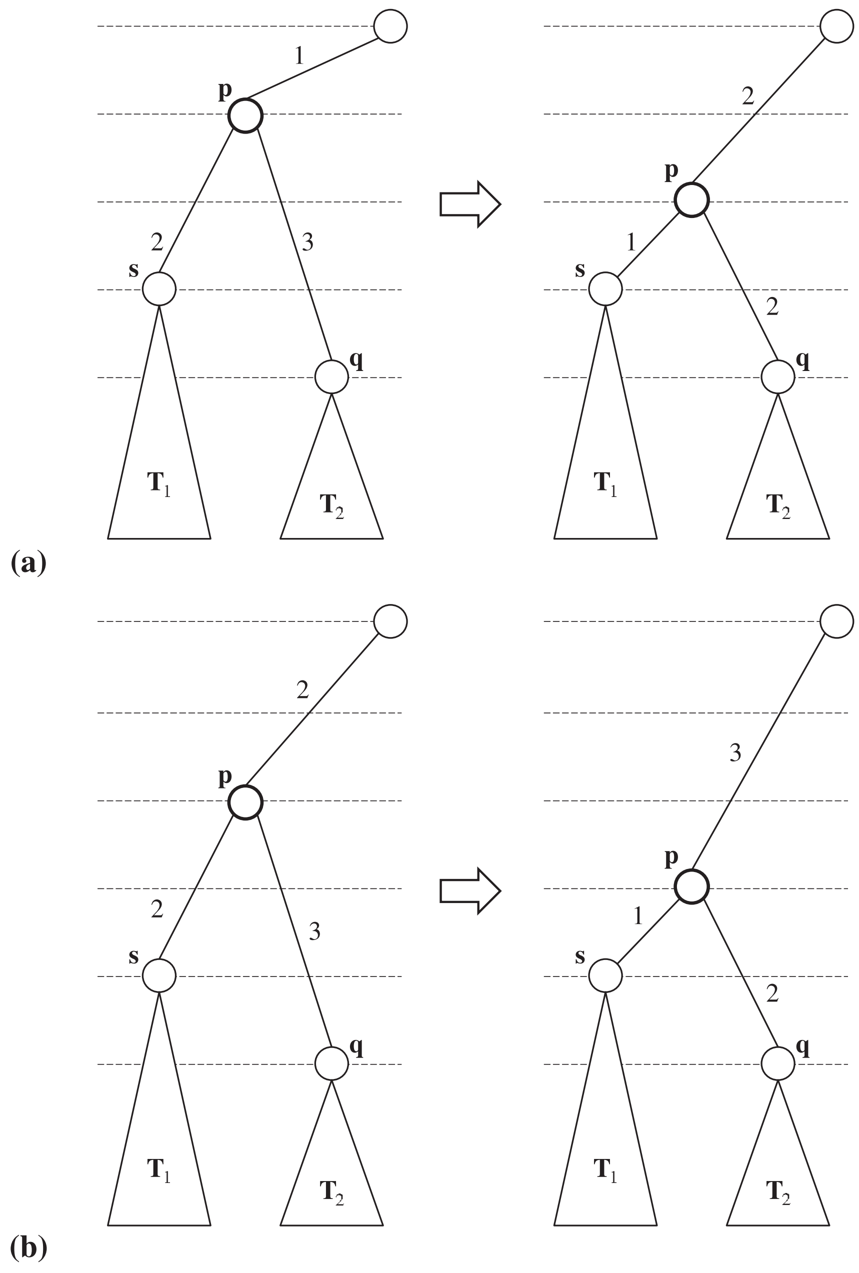


Figure 4.15: Case 1 of rebalancing in a wavl tree after a deletion: node q has rank-difference 3 (a violation) and its sibling, s, has rank-difference 2. To resolve the rank-difference violation for node q, we demote p, the parent of q. (a) If p had rank difference 1, we are done. (b) Else (p had rank-difference 2), we now have a violation at p.

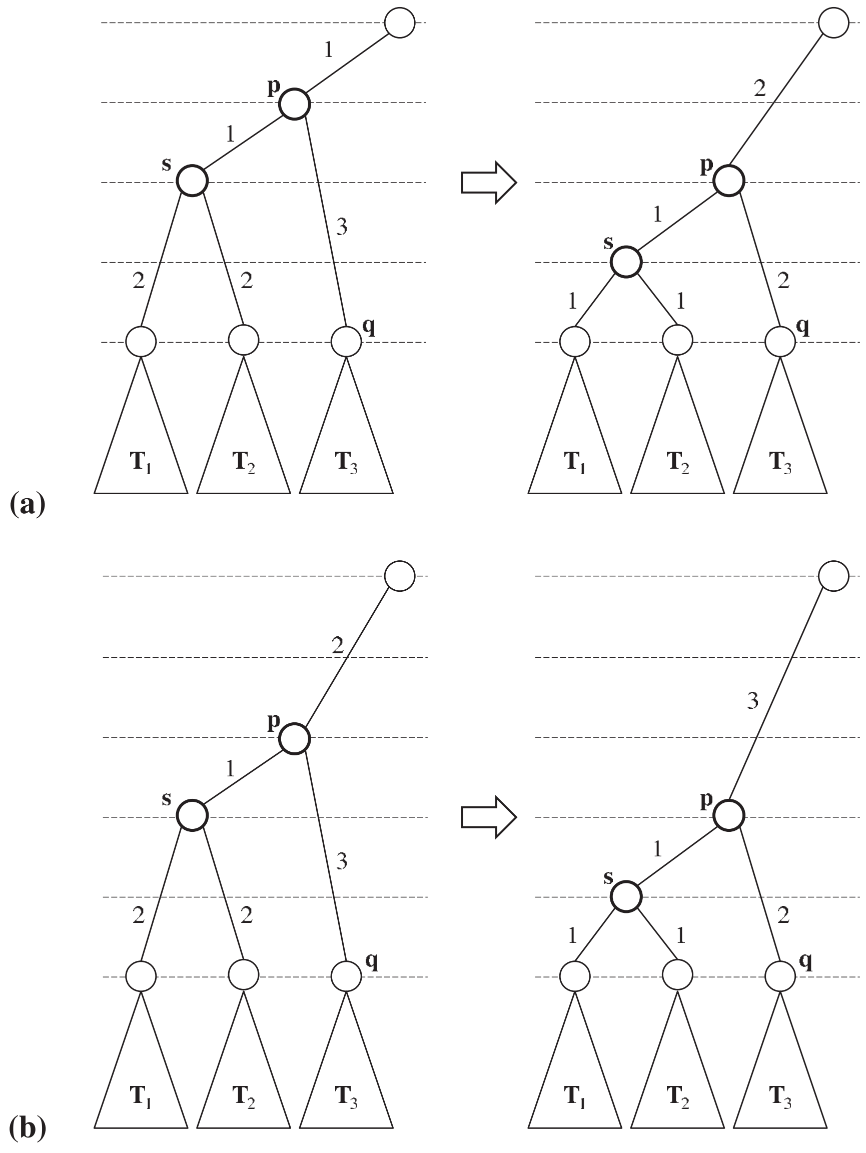
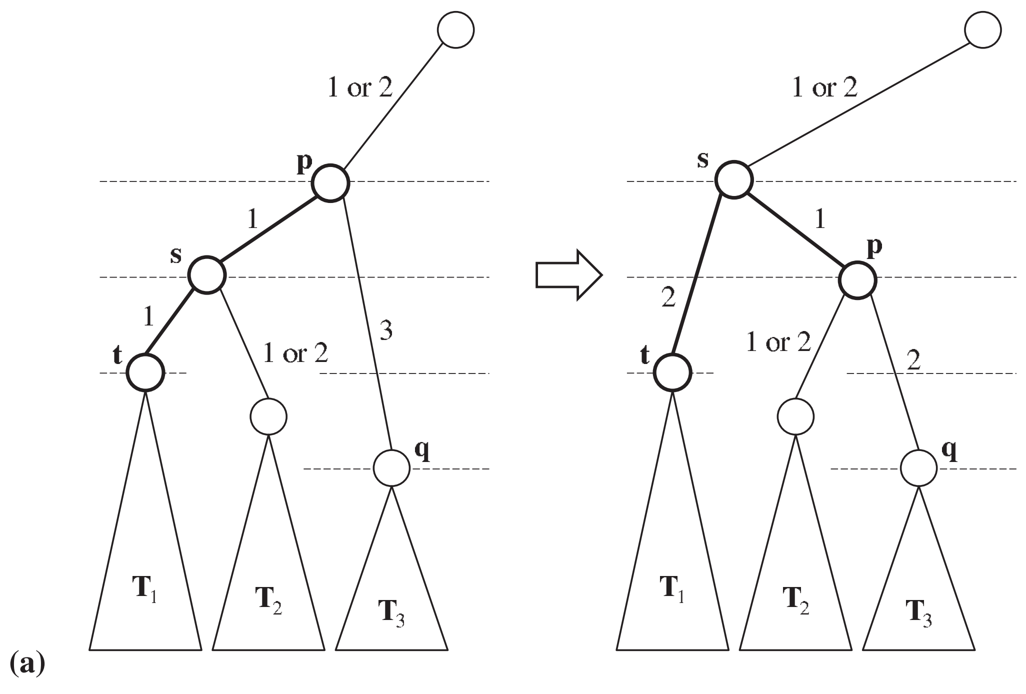


Figure 4.16: Case 2.a of rebalancing in a wavl tree after a deletion: node q has rank-difference 3 (a violation), its sibling, s, has rank-difference 1, and both children of s have rank-difference 2. To resolve the rank-difference violation for node q, we demote both s and p, the parent of q. (a) If p had rank difference 1, we are done. (b) Else (p had rank-difference 2), we now have a violation at p.



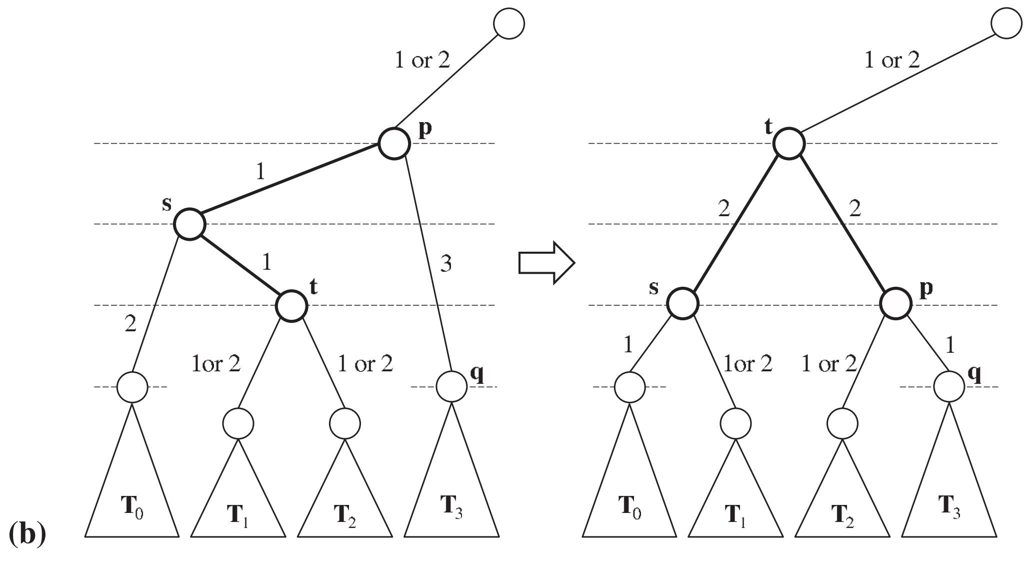


Figure 4.17: Case 2.b of rebalancing in a wavl tree after a deletion: node q has rank-difference 3 (a violation), its sibling, s, has rank-difference 1, and a child, t, of s has rank-difference 1. To resolve the violation for node q, we perform a trinode restructuring operation at t.

4.5 Splay Trees

The final balanced binary search tree data structure we discuss in this chapter is the ***splay tree***. This structure is conceptually quite different from the previously discussed balanced search trees (AVL, red-black, and wavl trees), for a splay tree does not use any explicit ranks or rules to enforce its balance. Instead, it applies a certain move-to-root operation, called ***splaying*** after every access, in order to keep the search tree balanced in an amortized sense. The splaying operation is performed at the bottommost node x reached during an insertion, deletion, or even a search. The surprising thing about splaying is that it allows us to guarantee amortized running times for insertions, deletions, and searches that are logarithmic. The structure of a ***splay tree*** is simply a binary search tree T. In fact, there are no additional height, balance, or color labels that we associate with the nodes of this tree.

Splaying

Given an internal node x of a binary search tree T, we ***splay*** x by moving x to the root of T through a sequence of restructurings. The particular restructurings we perform are important, for it is not sufficient to move x to the root of T by just any sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and (if it exists) x's grandparent z. There are three cases that we consider.

***zig-zig*:** The node x and its parent y are both left or right children. (See Figure 4.19.) We replace z by x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.

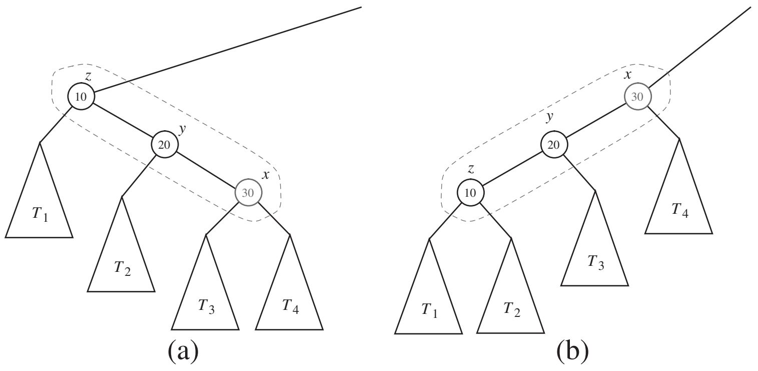


Figure 4.19: Zig-zig: (a) before; (b) after. There is another symmetric configuration where x and y are left children.

***zig-zag*:** One of x and y is a left child and the other is a right child. (See Figure 4.20.) In this case, we replace z by x and make x have as its children the nodes y and z, while maintaining the inorder relationships of the nodes in T.

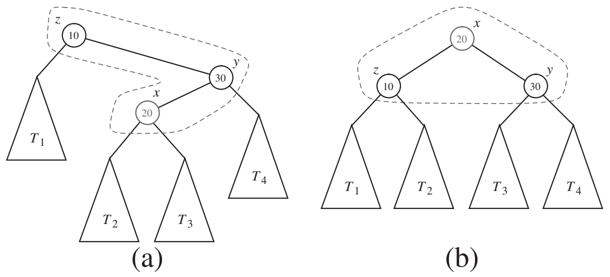


Figure 4.20: Zig-zag: (a) before; (b) after. There is another symmetric configuration where x is a right child and y is a left child.

***zig*:** x does not have a grandparent (or we are not considering x's grandparent for some reason). (See Figure 4.21.) In this case, we rotate x over y, making x's children be the node y and one of x's former children w, so as to maintain the relative inorder relationships of the nodes in T.

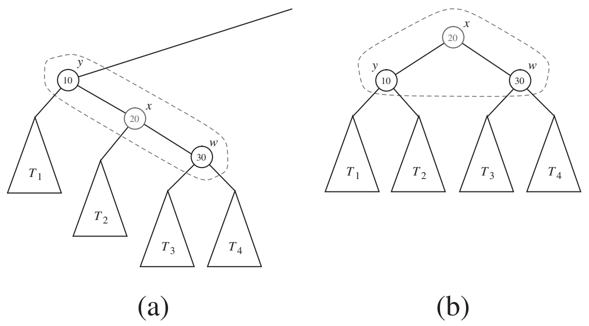


Figure 4.21: Zig: (a) before; (b) after. There is another symmetric configuration where x and w are left children.

We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent. A ***splaying*** step consists of repeating these restructurings at x until x becomes the root of T. Note that this is not the same as a sequence of simple rotations that brings x to the root. An example of the splaying of a node is shown in Figures 4.22 and 4.23.

After a zig-zig or zig-zag, the depth of x decreases by two, and after a zig the depth of x decreases by one. Thus, if x has depth d, splaying x consists of a sequence of $\lfloor d/2 \rfloor$ zig-zigs and/or zig-zags, plus one final zig if d is odd. Since a single zig-zig, zig-zag, or zig affects a constant number of nodes, it can be done in $O(1)$ time. Thus, splaying a node x in a binary search tree T takes time $O(d)$, where d is the depth of x in T. In other words, the time for performing a splaying step for a node x is asymptotically the same as the time needed just to reach that node in a top-down search from the root of T.

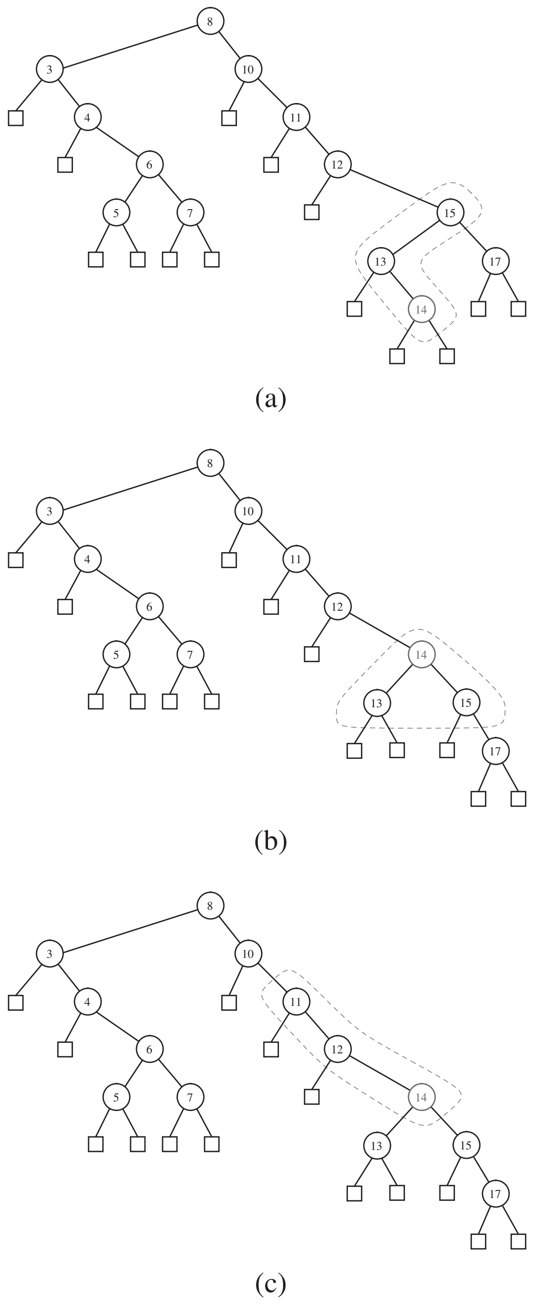


Figure 4.22: Example of splaying a node: (a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step is a zig-zig.

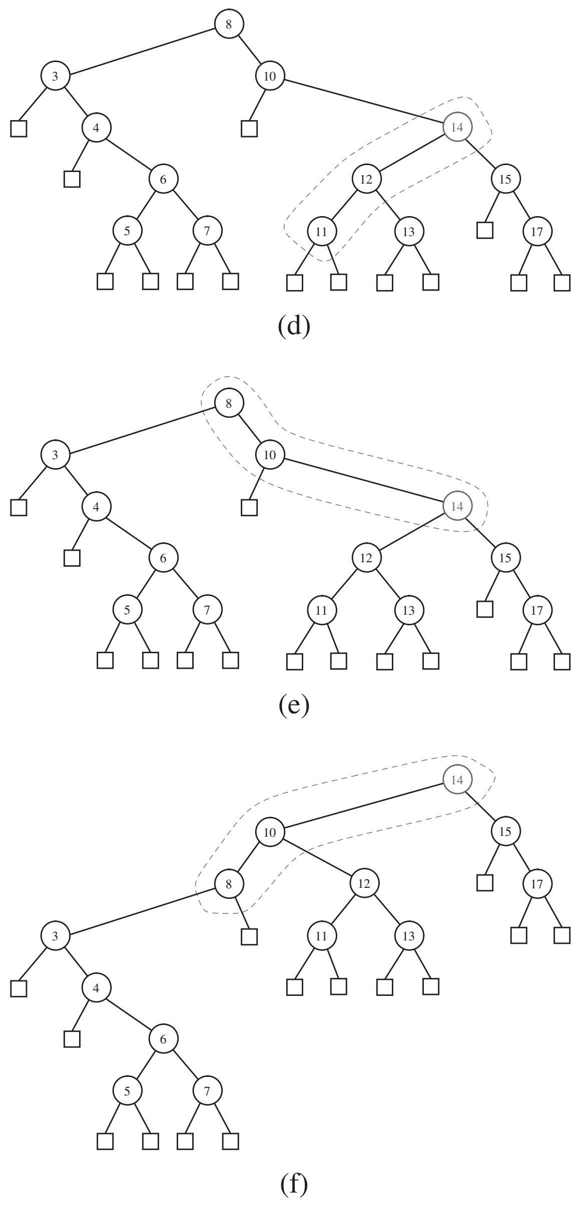


Figure 4.23: Example of splaying a node (continued from Figure 4.22): (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig.

When to Splay

The rules that dictate when splaying is performed are as follows:

- When searching for key k, if k is found at a node x, we splay x, else we splay the parent of the external node at which the search terminates unsuccessfully. For example, the splaying in Figures 4.22 and 4.23 would be performed after searching successfully for key 14 or unsuccessfully for key 14.5.
- When inserting key k, we splay the newly created internal node where k gets inserted. For example, the splaying in Figures 4.22 and 4.23 would be performed if 14 were the newly inserted key. We show a sequence of insertions in a splay tree in Figure 4.24.

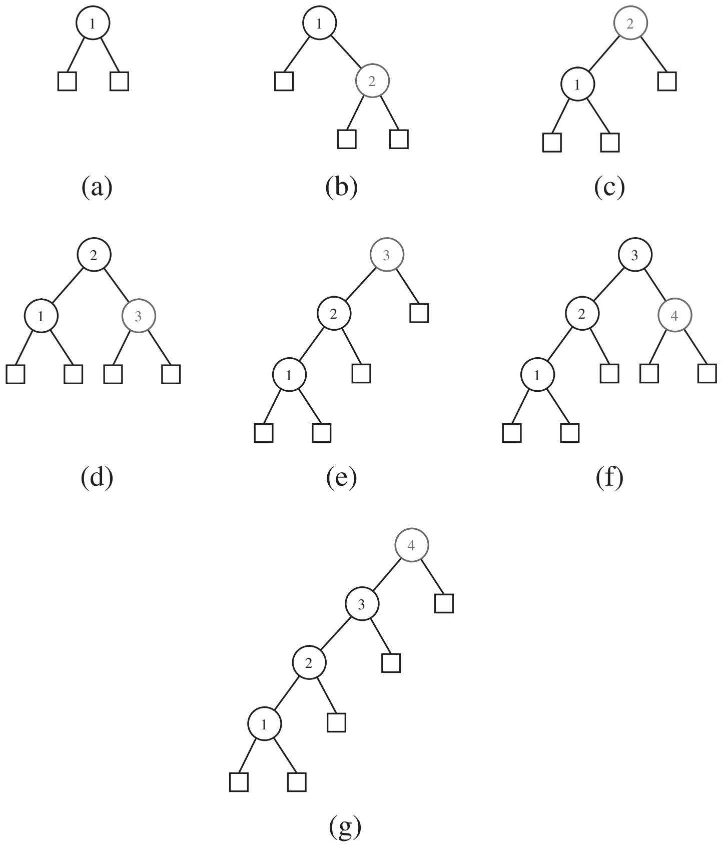


Figure 4.24: A sequence of insertions in a splay tree: (a) initial tree; (b) after inserting 2; (c) after splaying; (d) after inserting 3; (e) after splaying; (f) after inserting 4; (g) after splaying.

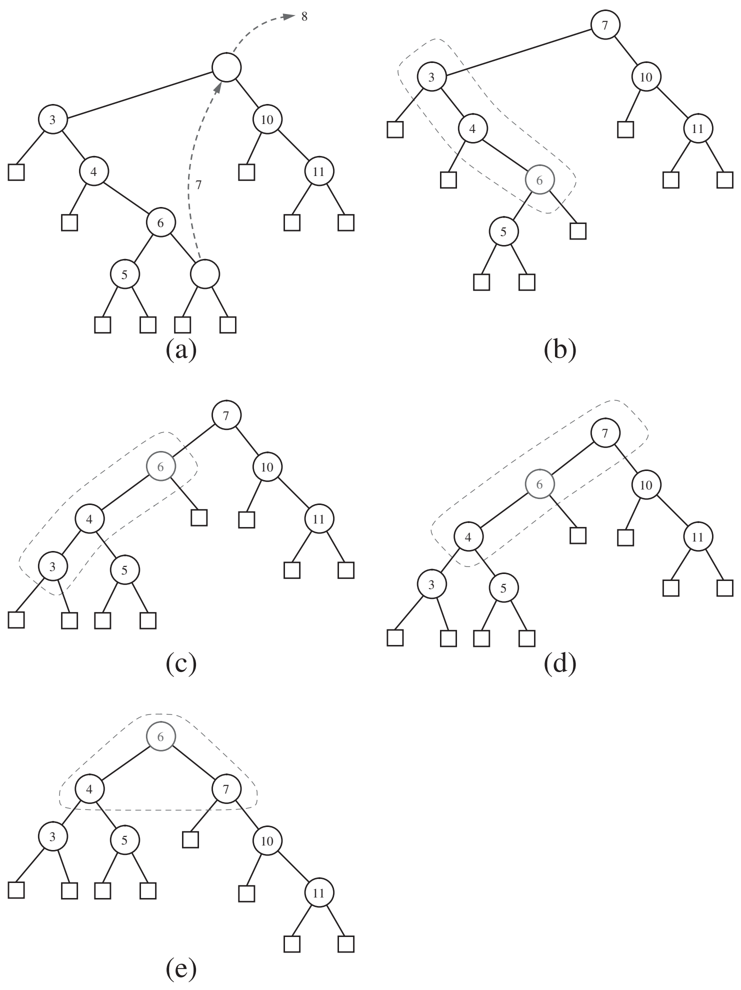


Figure 4.25: Deletion from a splay tree: (a) the deletion of 8 from node r is performed by moving to r the key of the right-most internal node v, in the left subtree of r, deleting v, and splaying the parent u of v; (b) splaying u starts with a zig-zig; (c) after the zig-zig; (d) the next step is a zig; (e) after the zig.

- When deleting a key k, we splay the parent of the node w that gets removed, that is, w is either the node storing k or one of its descendants. (Recall the deletion algorithm for binary search trees given in Section 3.1.4.) An example of splaying following a deletion is shown in Figure 4.25.

In the worst case, the overall running time of a search, insertion, or deletion in a splay tree of height h is $O(h)$, since the node we splay might be the deepest node in the tree. Moreover, it is possible for h to be $\Omega(n)$, as shown in Figure 4.24. Thus, from a worst-case point of view, a splay tree is not an attractive data structure.

Amortized Analysis of Splaying

In spite of its poor worst-case performance, a splay tree performs well in an amortized sense. That is, in a sequence of intermixed searches, insertions, and deletions, each operation takes on average logarithmic time. We note that the time for performing a search, insertion, or deletion is proportional to the time for the associated splaying; hence, in our analysis, we consider only the splaying time.

Let T be a splay tree with n keys, and let v be a node of T. We define the ***size*** $n(v)$ of v as the number of nodes in the subtree rooted at v. Note that the size of an internal node is one more than the sum of the sizes of its two children. We define the ***rank*** $r(v)$ of a node v as the logarithm in base 2 of the size of v, that is, $r(v) = \log n(v)$. Clearly, the root of T has the maximum size $2n+1$ and the maximum rank, $\log(2n+1)$, while each external node has size 1 and rank 0.

We use cyber-dollars to pay for the work we perform in splaying a node x in T, and we assume that one cyber-dollar pays for a zig, while two cyber-dollars pay for a zig-zig or a zig-zag. Hence, the cost of splaying a node at depth d is d cyber-dollars. We keep a virtual account, storing cyber-dollars, at each internal node of T. Note that this account exists only for the purpose of our amortized analysis, and does not need to be included in a data structure implementing the splay tree T. When we perform a splaying, we pay a certain number of cyber-dollars (the exact value will be determined later). We distinguish three cases:

- If the payment is equal to the splaying work, then we use it all to pay for the splaying.
- If the payment is greater than the splaying work, we deposit the excess in the accounts of several nodes.
- If the payment is less than the splaying work, we make withdrawals from the accounts of several nodes to cover the deficiency.

We show that a payment of $O(\log n)$ cyber-dollars per operation is sufficient to keep the system working, that is, to ensure that each node keeps a nonnegative account balance. We use a scheme in which transfers are made between the accounts of the nodes to ensure that there will always be enough cyber-dollars to withdraw for paying for splaying work when needed. We also maintain the following invariant:

Before and after a splaying, each node v ***of*** T ***has*** $r(v)$ ***cyber-dollars.***

Note that the invariant does not require us to endow an empty tree with any cyber-dollars. Let $r(T)$ be the sum of the ranks of all the nodes of T. To preserve the invariant after a splaying, we must make a payment equal to the splaying work plus the total change in $r(T)$. We refer to a single zig, zig-zig, or zig-zag operation in a splaying as a splaying ***substep***. Also, we denote the rank of a node v of T before and after a splaying substep with $r(v)$ and $r'(v)$, respectively. The following lemma gives an upper bound on the change of $r(T)$ caused by a single splaying substep.

Lemma 4.10: *Let δ be the variation of $r(T)$ caused by a single splaying substep (a zig, zig-zig, or zig-zag) for a node x in a splay tree T. We have the following:*

- $\delta \le 3(r'(x) - r(x)) - 2$ *if the substep is a zig-zig or zig-zag.*
- $\delta \le 3(r'(x) - r(x))$ *if the substep is a zig.*

Proof: We shall make use of the following mathematical fact (see Appendix A): If $a > 0$, $b > 0$, and $c > a + b$, then

$$\log a + \log b \le 2 \log c - 2. \tag{4.3}$$

Let us consider the change in $r(T)$ caused by each type of splaying substep.

***zig-zig*:** (Recall Figure 4.19.) Since the size of each node is one more than the size of its two children, note that only the ranks of x, y, and z change in a zig-zig operation, where y is the parent of x and z is the parent of y. Also, $r'(x) = r(z)$, $r'(y) \le r'(x)$, and $r(y) \ge r(x)$. Thus,

$$\begin{aligned} \delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\le r'(y) + r'(z) - r(x) - r(y) \\ &\le r'(x) + r'(z) - 2r(x). \end{aligned} \tag{4.4}$$

Also, $n(x) + n'(z) \le n'(x)$. Thus, by 4.3, $r(x) + r'(z) \le 2r'(x) - 2$, and

$$r'(z) \le 2r'(x) - r(x) - 2.$$

This inequality and 4.4 imply

$$\begin{aligned} \delta &\le r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \\ &\le 3(r'(x) - r(x)) - 2. \end{aligned}$$

***zig-zag*:** (Recall Figure 4.20.) Again, by the definition of size and rank, only the ranks of x, y, and z change, where y denotes the parent of x and z denotes the parent of y. Also, $r'(x) = r(z)$ and $r(x) \le r(y)$. Thus,

$$\begin{aligned} \delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\le r'(y) + r'(z) - r(x) - r(y) \\ &\le r'(y) + r'(z) - 2r(x). \end{aligned} \tag{4.5}$$

Also, $n'(y) + n'(z) \le n'(x)$. Thus, by 4.3, $r'(y) + r'(z) \le 2r'(x) - 2$. This inequality and 4.5 imply

$$\begin{aligned} \delta &\le 2r'(x) - 2 - 2r(x) \\ &\le 3(r'(x) - r(x)) - 2. \end{aligned}$$

***zig*:** (Recall Figure 4.21.) In this case, only the ranks of x and y change, where y denotes the parent of x. Also, $r'(y) \le r(y)$ and $r'(x) \ge r(x)$. Thus,

$$\begin{aligned} \delta &= r'(y) + r'(x) - r(y) - r(x) \\ &\le r'(x) - r(x) \\ &\le 3(r'(x) - r(x)). \end{aligned}$$

■

We can now bound the total variation of $r(T)$ caused by splaying a node x.

Theorem 4.11: *Let T be a splay tree with root t, and let Δ be the total variation of $r(T)$ caused by splaying a node x at depth d. We have*

$$\Delta \leq 3(r(t) - r(x)) - d + 2.$$

Proof: Splaying node x consists of $p = \lceil d/2 \rceil$ splaying substeps, each of which is a zig-zig or a zig-zag, except possibly the last one, which is a zig if d is odd. Let $r_0(x) = r(x)$ be the initial rank of x, and for $i = 1, \ldots, p$, let $r_i(x)$ be the rank of x after the ith substep and δ_i be the variation of $r(T)$ caused by the ith substep. By Lemma 4.10, the total variation Δ of $r(T)$ caused by splaying node x is given by

$$\begin{aligned} \Delta &= \sum_{i=1}^{p} \delta_i \\ &\leq \sum_{i=1}^{p} (3(r_i(x) - r_{i-1}(x)) - 2) + 2 \\ &= 3(r_p(x) - r_0(x)) - 2p + 2 \\ &\leq 3(r(t) - r(x)) - d + 2. \end{aligned}$$

■

By Theorem 4.11, if we make a payment of $3(r(t) - r(x)) + 2$ cyber-dollars toward the splaying of node x, we have enough cyber-dollars to maintain the invariant, keeping $r(v)$ cyber-dollars at each node v in T, and pay for the entire splaying work, which costs d dollars. Since the size of the root t is $2n + 1$, its rank $r(t) = \log(2n + 1)$. In addition, we have $r(x) < r(t)$. Thus, the payment to be made for splaying is $O(\log n)$ cyber-dollars. To complete our analysis, we have to compute the cost for maintaining the invariant when a node is inserted or deleted.

When inserting a new node v into a splay tree with n keys, the ranks of all the ancestors of v are increased. Namely, let $v_0, v_i, \ldots, v_d$ be the ancestors of v, where $v_0 = v$, v_i is the parent of v_{i-1}, and v_d is the root. For $i = 1, \ldots, d$, let $n'(v_i)$ and $n(v_i)$ be the size of v_i before and after the insertion, respectively, and let $r'(v_i)$ and $r(v_i)$ be the rank of v_i before and after the insertion, respectively. We have

$$n'(v_i) = n(v_i) + 1.$$

Also, since $n(v_i) + 1 \leq n(v_{i+1})$, for $i = 0, 1, \ldots, d-1$, we have the following for each i in this range:

$$r'(v_i) = \log(n'(v_i)) = \log(n(v_i) + 1) \leq \log(n(v_{i+1})) = r(v_{i+1}).$$

Thus, the total variation of $r(T)$ caused by the insertion is

$$\begin{aligned} \sum_{i=1}^{d} \left(r'(v_i) - r(v_i)\right) &\leq r'(v_d) + \sum_{i=1}^{d-1} (r(v_{i+1}) - r(v_i)) \\ &= r'(v_d) - r(v_0) \\ &\leq \log(2n + 1). \end{aligned}$$

Thus, a payment of $O(\log n)$ cyber-dollars is sufficient to maintain the invariant when a new node is inserted.

When deleting a node v from a splay tree with n keys, the ranks of all the ancestors of v are decreased. Thus, the total variation of $r(T)$ caused by the deletion is negative, and we do not need to make any payment to maintain the invariant. Therefore, we may summarize our amortized analysis in the following theorem.

Theorem 4.12: *Consider a sequence of m operations on a splay tree, each a search, insertion, or deletion, starting from an empty splay tree with zero keys. Also, let n_i be the number of keys in the tree after operation i, and n be the total number of insertions. The total running time for performing the sequence of operations is*

$$O\left(m + \sum_{i=1}^{m} \log n_i\right),$$

which is $O(m \log n)$.

In other words, the amortized running time of performing a search, insertion, or deletion in a splay tree is $O(\log n)$, where n is the size of the splay tree at the time. Thus, a splay tree can achieve logarithmic time amortized performance for searching and updating. This amortized performance matches the worst-case performance of AVL trees, red-black trees, and wavl trees, but it does so using a simple binary tree that does not need any extra balance information stored at each of its nodes. In addition, splay trees have a number of other interesting properties that are not shared by these other balanced search trees. We explore one such additional property in the following theorem (which is sometimes called the "Static Optimality" theorem for splay trees).

Theorem 4.13: *Consider a sequence of m operations on a splay tree, each a search, insertion, or deletion, starting from a tree T with no keys. Also, let $f(i)$ denote the number of times the item i is accessed in the splay tree, that is, its* **frequency**, *and let n be total number of items. Assuming that each item is accessed at least once, then the total running time for performing the sequence of operations is*

$$O\left(m + \sum_{i=1}^{n} f(i) \log\left(m/f(i)\right)\right).$$

We leave the proof of this theorem as an exercise. The remarkable thing about this theorem is that it states that the amortized running time of accessing an item i is $O(\log(m/f(i)))$. For example, if a sequence of operations accesses some item i as many as $m/4$ times, then the amortized running time of each of these accesses is $O(1)$ when we use a splay tree. Contrast this to the $\Omega(\log n)$ time needed to access this item in an AVL tree, red-black tree, or wavl tree. Thus, an additional nice property of splay trees is that they can "adapt" to the ways in which items are being accessed so as to achieve faster running times for the frequently accessed items.

4.6 Exercises

Reinforcement

R-4.1 Consider the insertion of items with the following keys (in the given order) into an initially empty AVL tree: 30, 40, 24, 58, 48, 26, 11, 13. Draw the final tree that results.

R-4.2 Consider the insertion of items with the following keys (in the given order) into an initially empty wavl tree: 12, 44, 52, 58, 38, 27, 41, 11. Draw the final tree that results.

R-4.3 Consider the insertion of items with the following keys (in the given order) into an initially empty splay tree: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18. Draw the final tree that results.

R-4.4 A certain Professor Amongus claims that the order in which a fixed set of elements is inserted into an AVL tree does not matter—the same tree results every time. Give a small example that proves Professor Amongus wrong.

R-4.5 Professor Amongus claims he has a "patch" to his claim from the previous exercise, namely, that the order in which a fixed set of elements is inserted into a wavl tree does not matter—the same tree results every time. Give a small example that proves that Professor Amongus is still wrong.

R-4.6 What is the minimum number of nodes in an AVL tree of height 7?

R-4.7 What is the minimum number of nodes in a red-black tree of height 8?

R-4.8 What is the minimum number of nodes in a wavl tree of height 7?

R-4.9 What is the minimum number of nodes in a splay tree of height 9?

R-4.10 Is the rotation done in Figure 4.5 a single or a double rotation? What about the rotation in Figure 4.6?

R-4.11 Draw the AVL tree resulting from the insertion of an item with key 52 into the AVL tree of Figure 4.6b.

R-4.12 Draw the AVL tree resulting from the removal of the item with key 62 from the AVL tree of Figure 4.6b.

R-4.13 Draw the wavl tree resulting from the insertion of an item with key 52 into the tree of Figure 4.6b.

R-4.14 Draw the wavl tree resulting from the removal of the item with key 62 from the tree of Figure 4.6b.

R-4.15 Draw an example red-black tree that is not an AVL tree. Your tree should have at least 6 nodes, but no more than 16.

R-4.16 Draw an example of a red-black tree that is not structurally equivalent to a wavl tree.

R-4.17 For each of the following statements about wavl trees, determine whether it is true or false. If you think it is true, provide a justification. If you think it is false, give a counterexample.

a. A subtree of a wavl tree is itself a wavl tree.
b. The sibling of an external node is either external or it has rank 1.

R-4.18 What does a splay tree look like if its items are accessed in increasing order by their keys?

R-4.19 How many trinode restructuring operations are needed to perform the zig-zig, zig-zag, and zig updates in splay trees? Use figures to explain your counting.

R-4.20 Describe how to implement the methods, insert(k, v) and remove(k), as well methods, and min() and max(), which return the key-value pair with smallest and largest key, respectively, in $O(\log n)$ time each using a balanced binary search tree.

Creativity

C-4.1 Show that any n-node binary tree can be converted to any other n-node binary tree using $O(n)$ rotations.

Hint: Show that $O(n)$ rotations suffice to convert any binary tree into a ***left chain***, where each internal node has an external right child.

C-4.2 The ***Fibonacci sequence*** is the sequence of numbers,

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots,$$

defined by the base cases, $F_0 = 0$ and $F_1 = 1$, and the general-case recursive definition, $F_k = F_{k-1} + F_{k-2}$, for $k \geq 2$. Show, by induction, that, for $k \geq 3$,

$$F_k \geq \varphi^{k-2},$$

where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$, which is the well-known ***golden ratio*** that traces its history to the ancient Greeks.

Hint: Note that $\varphi^2 = \varphi + 1$; hence, $\varphi^k = \varphi^{k-1} + \varphi^{k-2}$, for $k \geq 3$.

C-4.3 Show, by induction, that the minimum number, n_h, of internal nodes in an AVL tree of height h, as defined in the proof of Theorem 4.1, satisfies the following identity, for $h \geq 1$:

$$n_h = F_{h+2} - 1,$$

where F_k denotes the ***Fibonacci number*** of order k, as defined in the previous exercise.

C-4.4 Use the facts given in the previous two exercises (and a scientific calculator) to show that an AVL tree storing n items has height at most $1.441 \log (n+1)$, where, as is the custom in this book, we take the base of the "log" function to be 2.

C-4.5 Describe how to perform the operation findAllElements(k), which returns all the items with keys equal to k in a balanced search tree, and show that it runs in time $O(\log n + s)$, where n is the number of elements stored in the tree and s is the number of items returned.

C-4.6 Describe how to perform the operation removeAllElements(k), which removes all elements with keys equal to k, in a balanced search tree T, and show that this method runs in time $O(s \log n)$, where n is the number of elements stored in the tree and s is the number of elements with key k.

C-4.7 Draw an example of an AVL tree such that a single remove operation could require $\Theta(\log n)$ trinode restructurings (or rotations) from a leaf to the root in order to restore the height-balance property. (Use triangles to represent subtrees that are not affected by this operation.)

C-4.8 Show that at most one node in an AVL tree becomes unbalanced after operation removeAboveExternal is performed within the execution of a remove operation.

C-4.9 Show that at most one trinode restructure operation (which corresponds to one single or double rotation) is needed to restore balance after any insertion in an AVL tree.

C-4.10 Let T and U be wavl trees storing n and m items, respectively, such that all the items in T have keys less than the keys of all the items in U. Describe an $O(\log n + \log m)$ time method for ***joining*** T and U into a single tree that stores all the items in T and U (destroying the old versions of T and U).

C-4.11 Justify Theorem 20.1.

C-4.12 The Boolean used to mark nodes in a red-black tree as being "red" or "black" is not strictly needed when storing a set of distinct keys. Describe a scheme for implementing a red-black tree without adding any extra space to binary search tree nodes in this case.

C-4.13 Let T be a wavl tree storing n items, and let k be the key of an item in T. Show how to construct from T, in $O(\log n)$ time, two wavl trees T' and T'', such that T' contains all the keys of T less than k, and T'' contains all the keys of T greater than k. This operation destroys T.

C-4.14 Show that the nodes of any AVL tree T can be colored "red" and "black" so that T becomes a red-black tree.

C-4.15 A ***mergeable heap*** supports operations insert(k, x), remove(k), unionWith(h), and min$()$, where the unionWith(h) operation performs a union of the mergeable heap h with the present one, destroying the old versions of both, and min$()$ returns the element with minimum key. Describe a implementation of a mergeable heap that achieves $O(\log n)$ performance for all its operations. For simplicity, you may assume that all keys in existing mergeable heaps are distinct, although this is not strictly necessary.

C-4.16 Consider a variation of splay trees, called ***half-splay trees***, where splaying a node at depth d stops as soon as the node reaches depth $\lfloor d/2 \rfloor$. Perform an amortized analysis of half-splay trees.

C-4.17 The standard splaying step requires two passes, one downward pass to find the node x to splay, followed by an upward pass to splay the node x. Describe a method for splaying and searching for x in one downward pass. Each substep now requires that you consider the next two nodes in the path down to x, with a possible zig substep performed at the end. Describe the details for performing each of the zig-zig, zig-zag, and zig substeps.

C-4.18 Describe a sequence of accesses to an n-node splay tree T, where n is odd, that results in T consisting of a single chain of internal nodes with external node children, such that the internal-node path down T alternates between left children and right children.

C-4.19 Justify Theorem 4.13. A way to establish this justification is to note that we can redefine the "size" of a node as the sum of the access frequencies of its children and show that the entire justification of Theorem 4.11 still goes through.

C-4.20 Suppose we are given a sorted sequence S of items $(x_0, x_1, \ldots, x_{n-1})$ such that each item x_i in S is given a positive integer weight a_i. Let A denote the total weight of all elements in S. Construct an $O(n \log n)$-time algorithm that builds a search tree T for S such that the depth of each item a_i is $O(\log A/a_i)$.

Hint: Find the item x_j with largest j such that $\sum_{i=0}^{j-1} a_i < A/2$. Consider putting this item at the root and recursing on the two subsequences that this induces.

C-4.21 Design a linear-time algorithm for the previous problem.

Applications

A-4.1 Suppose you are working for a large dog adoption organization and are asked to build a website showing all the dogs that your organization currently knows of that are waiting to be placed in loving homes. You are being asked to create a separate web page for each such dog, with these pages ordered by age, so that the web page for a waiting dog, d, has a link to the dog just younger than d and a link to the dog just older than d (with ties broken arbitrarily). Describe an efficient scheme for designing an automated way to add and remove dogs from this website (as they are respectively put up for adoption and placed in loving homes). Sketch your methods for adding and removing dogs, and characterize the running times of these methods in terms of n, the number of dogs currently displayed on the website.

A-4.2 Suppose you are working for a fast-growing startup company, which we will call "FastCo," and it is your job to write a software package that can maintain the set, E, of all the employees working for FastCo. In particular, your software has to maintain, for each employee, x in E, the vital information about x, such as his or her name and address, as well as the number of shares of stock in FastCo that the CEO has promised to x. When an employee first joins FastCo they start out with 0 shares of stock. Every Friday afternoon, the CEO hosts a party for all the FastCo employees and kicks things off by promising every employee that they are getting y more shares of stock, where the value of y tends to be different every Friday. Describe how to implement this software so that it can simultaneously achieve the following goals:

- The time to insert or remove an employee in E should be $O(\log n)$, where n is the number of employees in E.
- Your system must be able to list all the employees in E in alphabetical order in $O(n)$ time, showing, for each x in E, the number of shares of FastCo the CEO has promised to x.
- Your software must be able to process each Friday promise from the CEO in $O(1)$ time, to reflect the fact that everyone working for FastCo on that day is being promised y more shares of stock. (Your system needs to be this fast so that processing this update doesn't make you miss too much of the party.)

A-4.3 Suppose a used car dealer, Jalopy Joe, has asked you to build a website for his car dealership. He wants this website to allow users to be able to search for a set of cars on his lot that are in their price range. That is, viewed abstractly, he would like you to build a system that can maintain an ordered collection, D, of key-value pairs, such that, in addition to the standard insert and removal methods, the implementation for D can support the following operation:

findAllInRange(k_1, k_2): Return all the elements in D with key k such that $k_1 \leq k \leq k_2$.

In this case, of course, the keys are car prices. Describe an algorithm to implement this method in $O(\log n + s)$ time, where n is the number of cars in D and s is the number of cars returned by an instance of this range query.

A-4.4 Suppose you are working for a victim-support group to build a website for maintaining a set, S, containing the names of all the registered sex offenders in a given area. The system should be able to list out the names of the people in S ordered by their Zip codes, and, within each Zip code, ordered alphabetically. It should also be able to list out the names of the people in S just for a given Zip code. The running time for a full listing should be $O(n)$, where n is the number of people in S, and the running time for a listing for a given Zip code should be $O(\log n + s)$, where s is the number of names returned. Insertions and removals from S should run in $O(\log n)$ time. Describe a scheme for achieving these bounds.

A-4.5 Suppose you are hired as a consultant to a professor, Dr. Bob Loblaw, from the Sociology department. He is asking that you build him a software system that can maintain a set, P, of people from a country, Phishnonia, that he is studying. People in Phishnonia are free to come and go as they please, so Dr. Bob Loblaw is asking that your system support fast insertions and deletions. In addition, he is also interested in doing queries that respectively return the mean and median ages of the people in P. His thesis is that if the median age is much smaller than the mean, then the Phishnonia is ripe for revolution, for it implies that there are a disproportionate number of young people. Describe a scheme for maintaining P that can support median and mean age queries, subject to insertions and removals, with each of these operations running in at most $O(\log n)$ time, where n is the number of people in P.

A-4.6 Suppose your neighbor, sweet Mrs. McGregor, has invited you to her house to help her with a computer problem. She has a huge collection of JPEG images of bunny rabbits stored on her computer and a shoebox full of 1 gigabyte USB drives. She is asking that you help her copy her images onto the drives in a way that minimizes the number of drives used. It is easy to determine the size of each image, but finding the optimal way of storing images on the fewest number of drives is an instance of the ***bin packing*** problem, which is a difficult problem to solve in general. Nevertheless, Mrs. McGregor has suggested that you use the ***first fit*** heuristic to solve this problem, which she recalls from her days as a young computer scientist. In applying this heuristic here, you would consider the images one at a time and, for each image, I, you would store it on the first USB drive where it would fit, considering the drives in order by their remaining storage capacity. Unfortunately, Mrs. McGregor's way of doing this results in an algorithm with a running time of $O(mn)$, where m is the number of images and $n < m$ is the number of USB drives. Describe how to implement the first fit algorithm here in $O(m \log n)$ time instead.

A-4.7 Suppose there is a computer game, Land-of-Candy (LoC), where a player moves through a three-dimensional world defined by the cells in an $n \times n \times n$ array, C. Each cell, $C[i, j, k]$, specifies the number of points that a player in LoC gets when they are at position (i, j, k). At the start of a game in LoC, there are only $O(n)$ nonzero cells in C; all the other $O(n^3)$ cells in C are equal to 0. During the course of the game, if a player moves to a position (i, j, k) such that $C[i, j, k]$ is nonzero, then all the points in $C[i, j, k]$ are awarded to the player and the value of $C[i, j, k]$ is reduced to 0. Then the game picks another position, (i', j', k'), at random and adds 100 points to $C[i', j', k']$. The problem is that this game was designed for playing on a large computer and now must be adapted to run on a smartphone, which has much less memory. So you cannot afford to use $O(n^3)$ space to represent C, as in the original version. Describe an efficient way to represent C, which uses only $O(n)$ space. Also describe how to lookup the value of any cell, $C[i, j, k]$, and how to add 100 points to any cell, $C[i, j, k]$, efficiently so that looking up the value of any cell, $C[i, j, k]$, can be done in $O(\log n)$ worst-case time or better.

Chapter Notes

AVL trees are due to Adel'son-Vel'skii and Landis [2]. Average-height analyses for binary search trees can be found in the books by Aho, Hopcroft, and Ullman [9]. The handbook by Gonnet and Baeza-Yates [85] contains a number of theoretical and experimental comparisons among search-tree implementations. Red-black trees were defined by Bayer [22], and are discussed further by Guibas and Sedgewick [94]. Weak AVL trees were introduced by Haeupler, Sen, and Tarjan [96]. Splay trees were invented by Sleator and Tarjan [197] (see also [207]). Additional reading can be found in the books by Mehlhorn [157] and Tarjan [207], and the chapter by Mehlhorn and Tsakalidis [160]. Knuth [131] provides excellent additional reading that includes early approaches to balancing trees. Exercise C-4.21 is inspired by a paper by Mehlhorn [156]. We are thankful to Siddhartha Sen and Bob Tarjan for several helpful discussions regarding the topics of this chapter.