```
1     /**
2      * Internal method to remove from a subtree, adjusting
3      *    size fields as appropriate.
4      * @param x the item to remove.
5      * @param t the node that roots the tree.
6      * @return the new root.
7      * @exception ItemNotFound no item that
8      *    matches x is in the subtree rooted at t.
9      */
10    protected BinaryNode
11    remove( Comparable x, BinaryNode t ) throws ItemNotFound
12    {
13        if( t == null )
14            throw new ItemNotFound( "BSTWithRank remove" );
15        if( x.compares( t.element ) < 0 )
16            t.left = remove( x, t.left );
17        else if( x.compares( t.element ) > 0 )
18            t.right = remove( x, t.right );
19        else if( t.left != null && t.right != null )
20        {
21            t.element = findMin( t.right ).element;
22            t.right = removeMin( t.right );
23        }
24        else
25            return ( t.left != null ) ? t.left : t.right;
26        t.size--;
27        return t;
28    }
```

Figure 18.18 `remove` for the search tree with order statistics

18.3 Analysis of Binary Search Tree Operations

The cost of an operation is proportional to the depth of the last accessed node. This is logarithmic for a well-balanced tree, but it could be as bad as linear for a degenerate tree.

It is easy to see that the cost of each binary search tree operation (`insert`, `find`, and `remove`) is proportional to the number of nodes accessed during the operation. We can thus charge the access of any node in the tree a cost of 1, plus its depth (recall that the depth measures the number of edges on a path rather than the number of nodes). This gives the cost of a successful search.

Figure 18.19 shows two trees. On the left is a balanced tree of 15 nodes. The cost to access any node is at most 4 units, and some nodes require fewer accesses. This is exactly analogous to the situation that occurs in the binary search algorithm. If the tree is perfectly balanced, then the access cost is logarithmic.

Unfortunately, we have no guarantee that the tree is perfectly balanced. The second tree in Figure 18.19 is the classic example of an unbalanced tree. Here, all N nodes are on the path to the deepest node, so the worst-case search time is $O(N)$. Because the search tree has degenerated to a linked list, the average time required to search in *this particular instance* is half the cost of the worst case and

is also $O(N)$. So we have two extremes: In the best case, we have logarithmic access cost, and in the worst case, we have linear access cost. What, then, is the average? Do most binary search trees tend toward the balanced or unbalanced case, or is there some middle ground, such as $\sqrt{N}$? The answer is identical to that for quicksort: The average is 38 percent worse than the best case.

We prove in this section that the average depth over all nodes in a binary search tree is logarithmic, under the assumption that each tree is created as a result of random insertion sequences (with no `remove` operations). To see what this means, consider the result of inserting three items into an empty binary search tree. Since only their relative ordering is important, we can assume without loss of generality that the three items are 1, 2, and 3. Then there are six possible insertion orders: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3 ,2, 1). We will assume in our proof that each of these insertion orders is equally likely. The binary search trees that can result from these insertions are shown in Figure 18.20 (page 506). Notice that the tree with root 2 is formed from either the insertion sequence (2, 3, 1) or the sequence (2, 1, 3). Thus some trees are more likely than others, and as will be shown, balanced trees are more likely than unbalanced trees (although this is not evident from the three-element case).

On average, the depth is 38 percent worse than the best case. This result is identical to that obtained using quicksort.

We begin with the following definition.

DEFINITION: The *internal path length* of a binary tree is equal to the sum of the depths of its nodes.

When we divide the internal path length of a tree by the number of nodes in the tree, we obtain the average depth of a node in the tree. Adding 1 to this average gives the average cost of a successful search in the tree. So we want to compute the average internal path length for a binary search tree, where the average is taken over all (equally probable) input permutations. This is easily done by viewing the tree recursively and using techniques shown in the analysis of quicksort given in Section 8.6. The average internal path length is established in Theorem 18.1.

The *internal path length* is used to measure the cost of a successful search.

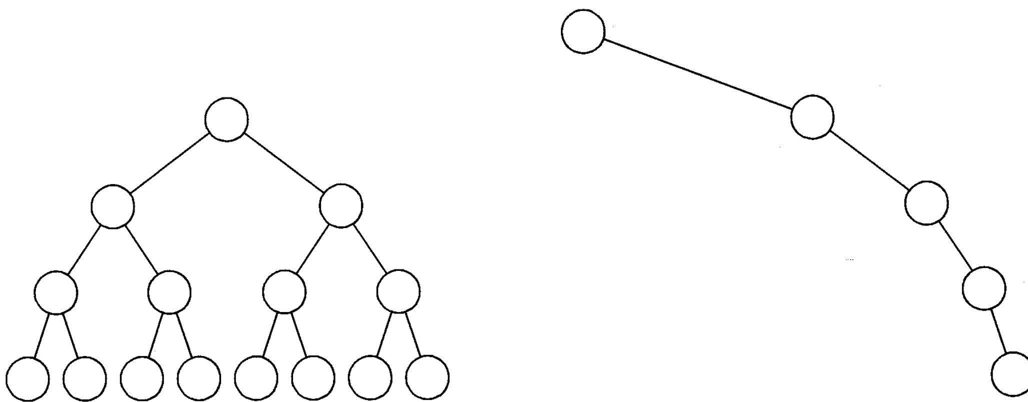

Figure 18.19 Balanced tree on the left has a depth of $\lfloor \log N \rfloor$; unbalanced tree on the right has a depth of $N - 1$

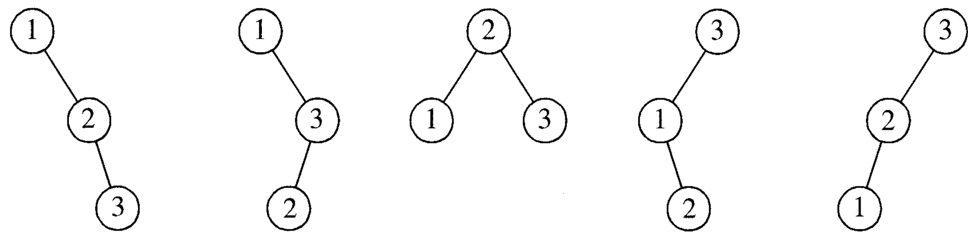

Figure 18.20 Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree in the middle is twice as likely to result from this as any others

Theorem 18.1 *The internal path length of a binary search tree is approximately $1.38N\log N$, on average, under the assumption that all permutations are equally likely.*

Proof *Let $D(N)$ be the average internal path length for trees of N nodes. $D(1) = 0$. An N-node tree T consists of an i-node left subtree and an $(N-i-1)$-node right subtree, plus a root at depth 0 for $0 \le i < N$. By assumption, each value of i is equally likely. For a given i, $D(i)$ is the average internal path length of the left subtree with respect to its root. In T, all these nodes are one level deeper. Thus the average contribution of the nodes in the left subtree to the internal path length of T is $(1/N)\sum_{i=0}^{N-1} D(i)$, plus 1 for each node in the left subtree. The same holds for the right subtree. We thus obtain the recurrence formula $D(N) = (2/N)(\sum_{i=0}^{N-1} D(i)) + N - 1$, which is identical to the quicksort recurrence solved in Section 8.6. Thus we obtain an average internal path length of $O(N \log N)$.*

The *external path length* is used to measure the cost of an unsuccessful search.

The insertion algorithm implies that the cost of an insert is exactly equal to the cost of an unsuccessful search, which is measured by using the *external path length*. In an insertion or unsuccessful search, we eventually reach the test `t==null`. Recall that in a tree of N nodes, there are $N + 1$ `null` references. The external path length measures the total number of nodes that are accessed, including the `null` node for each of these $N + 1$ `null` references. (The `null` node is sometimes called an *external tree node*, which explains the term *external path*

length. As is shown later in the chapter, it is sometimes convenient to use a sentinel to replace the `null` node.)

DEFINITION: The *external path length* of a binary search tree is the sum of the depths of the $N+1$ `null` references. The terminating `null` node is considered a node for these purposes.

One plus the result of dividing the average external path length by $N+1$ yields the average cost of an unsuccessful search or insertion. As with the binary search algorithm, the average cost of an unsuccessful search is only slightly more than the cost of a successful search. This follows from Theorem 18.2.

Theorem 18.2

For any tree T, let IPL(T) be the internal path length of T and let EPL(T) be its external path length. Then, if T has N nodes, $EPL(T) = IPL(T) + 2N$.

Proof

This theorem is proved by induction and is left as Exercise 18.8.

Random `remove` operations do not preserve the randomness of a tree. The effects are not completely understood theoretically, but it appears that they are negligible in practice.

It is tempting to say immediately that these results imply that the average running time of all operations is $O(\log N)$. This is true in practice, but it has not been established analytically because the assumption used to prove the previous results do not take into account the deletion algorithm. In fact, close examination suggests that we might be in trouble with our deletion algorithm because the `remove` always replaces a two-child deleted node with a node from the right subtree. This would seem to have the effect of eventually unbalancing the tree and tending to make it left-heavy. It has been shown that if we build a random binary search tree and then perform roughly N^2 pairs of random `insert`/`remove` combinations, then the binary search trees will have an expected depth of $O(\sqrt{N})$. However, it has never been shown that a reasonable number of random `insert` and `remove` operations (in which the order of `insert` and `remove` is also random) unbalances the tree in any observable way. In fact, for small search trees, the `remove` algorithm seems to balance the tree. Consequently, it is reasonable to assume that for random input, all operations will behave in logarithmic average time, although this has not been proved mathematically. Exercise 18.26 describes some alternative deletion strategies.

The most important problem is not the potential imbalance caused by the `remove` algorithm. Rather, it is the fact that if the input sequence is sorted, then the worst-case tree occurs. When this happens, we are in deep trouble: We have linear time per operation (for a series of N operations) rather than logarithmic cost per operation. This is analogous to passing items to quicksort but having an insertion sort executed instead. The resulting running time is completely unacceptable.

Moreover, it is not just sorted input that is problematic, but also any input that contains long sequences of nonrandomness. One solution to this problem is to insist on an extra structural condition called *balance*: No node is allowed to get too deep.

A *balanced binary search tree* adds a structure property to guarantee logarithmic depth in the worst case. Updates are slower, but accesses are faster.

There are several algorithms to implement *balanced binary search trees*. Most are much more complicated than the standard binary search trees, and all take longer on average for insertion and deletion. They do, however, provide protection against the embarrassingly simple cases. Also, because they are so balanced, they tend to give faster access time. Typically, their internal path lengths are very close to the optimal $N \log N$ rather than $1.38N \log N$, so searching time is roughly 25 percent faster.

18.4 AVL Trees

The *AVL tree* was the first balanced binary search tree. It has historical significance and also illustrates most of the ideas that are used in other schemes.

The first balanced binary search tree was the *AVL tree* (named after its discoverers, Adelson-Velskii and Landis). The AVL tree illustrates the ideas that are thematic for a wide class of balanced binary search trees. It is a binary search tree that has an additional balance condition. This balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log N)$. The simplest idea is to require that the left and right subtrees have the same height. Recursion dictates that this idea applies to all nodes in the tree, since each node is itself a root of some subtree. This balance condition ensures that the depth of the tree is logarithmic. However, it is too restrictive because it is too difficult to insert new items while maintaining balance. Thus the AVL tree uses a notion of balance that is somewhat weaker but still strong enough to guarantee logarithmic depth.

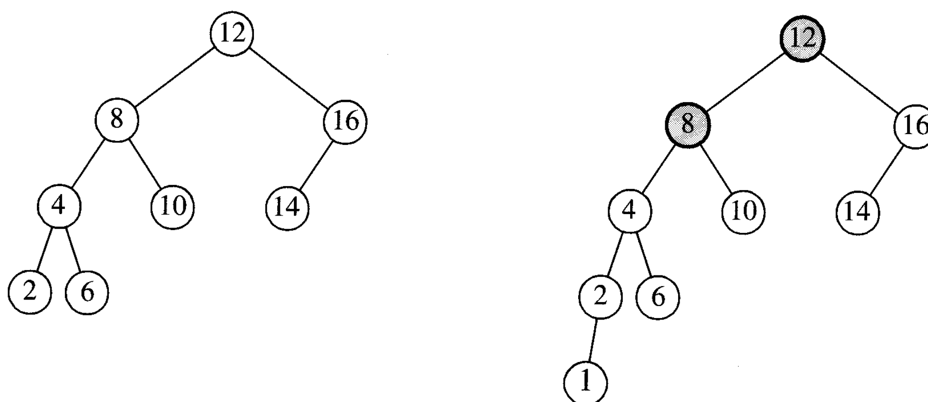

Figure 18.21 Two binary search trees: the left tree is an AVL tree; the right tree is not (unbalanced nodes are darkened)

18.4.1 Properties

DEFINITION: An *AVL tree* is a binary search tree with the additional balance property that, for any node in the tree, the height of the left and right subtrees can differ by at most 1. As usual, the height of an empty subtree is -1.

Figure 18.21 shows two binary search trees. The tree on the left satisfies the AVL balance condition and is thus an AVL tree. The tree on the right, which results from inserting 1 using the usual algorithm, is not an AVL tree because the darkened nodes have left subtrees whose heights are 2 larger than their right subtrees. If 13 was inserted using the usual binary search tree insertion algorithm, then node 16 would also be in violation. This is because the left subtree would have height 1, while the right subtree would have height -1.

Every node in an AVL tree has subtrees whose heights differ by at most 1. An empty subtree has height -1.

The AVL balance condition implies that the tree has only logarithmic depth. To prove this, we need to show that a tree of height H must have at least C^H nodes for some constant $C > 1$. In other words, the minimum number of nodes in a tree is exponential in its height. Then the maximum depth of an N-item tree is given by $\log_C N$. Theorem 18.3 shows that every AVL tree of height H has many nodes.

The AVL tree has height at most roughly 44 percent greater than the minimum.

Theorem 18.3

An AVL tree of height H has at least $F_{H+3} - 1$ nodes, where F_i is the ith Fibonacci number (see Section 7.3.4).

Proof

Let S_H be the size of the smallest AVL tree of height H. Clearly $S_0 = 1$ and $S_1 = 2$. Figure 18.22 (page 510) shows that the smallest AVL tree of height H must have subtrees of height $H - 1$ and $H - 2$. This is because at least one subtree has height $H - 1$ and the balance condition implies that subtree heights can differ by at most 1. These subtrees must themselves have the fewest number of nodes for their heights, so $S_H = S_{H-1} + S_{H-2} + 1$. It is then a simple matter to complete the proof by using an induction argument.

From Exercise 7.8, $F_i \approx \phi^i/\sqrt{5}$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$. Consequently, an AVL tree of height H has at least (roughly) $\phi^{H+3}/\sqrt{5}$ nodes. Hence, its depth is at most logarithmic. The height of an AVL tree satisfies

$$H < 1.44\log(N + 2) - 1.328, \qquad \textbf{(18.1)}$$

so the worst-case height is at most 44 percent more than the minimum possible for binary trees.

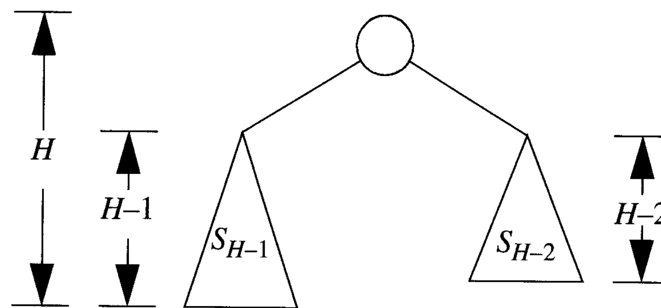

Figure 18.22 Minimum tree of height H

The depth of a typical node in an AVL tree is very close to the optimal log N.

The depth of an average node in a randomly constructed AVL tree tends to be very close to $\log N$. The exact answer has not yet been established analytically. It is not even known if the form is $\log N + C$ or $(1 + \varepsilon)\log N + C$, for some ε that would be approximately 0.01. Simulations have been unable to convincingly demonstrate that one form is more plausible than the other.

An update in an AVL tree could destroy the balance. We must then rebalance before the operation is complete.

A consequence of these arguments is that all searching operations in an AVL tree have logarithmic worst-case bounds. The difficulty is that operations that change the tree, such as `insert` and `remove`, are not quite as simple as before. This is because an insertion (or deletion) can destroy the balance of several nodes in the tree, as shown in Figure 18.21. The balance must then be restored before the operation can be considered complete. The insertion algorithm is described here and the deletion algorithm is left for Exercise 18.10.

Only nodes on the path from the root to the insertion point can have their balances altered.

A key observation is that after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered because only those nodes have their subtrees altered. This applies for almost all of the balanced search tree algorithms. As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition. This section shows how to rebalance the tree at the first (that is, the deepest) such node and proves that this rebalancing guarantees that the entire tree satisfies the AVL property.

If we fix the balance at the deepest unbalanced node, we will rebalance the entire tree. There are four cases that we might have to fix; two are mirror-images of the other two.

The node to be rebalanced is X. Since any node has at most two children and a height imbalance requires that the heights of X's two subtrees differ by 2, a violation might occur in any of four cases:

1. An insertion into the left subtree of the left child of X
2. An insertion into the right subtree of the left child of X
3. An insertion into the left subtree of the right child of X
4. An insertion into the right subtree of the right child of X

Cases 1 and 4 are mirror-image symmetries with respect to X, as are cases 2 and 3. Consequently, there theoretically are two basic cases. From a programming perspective, of course, there are still four cases (and numerous special cases).

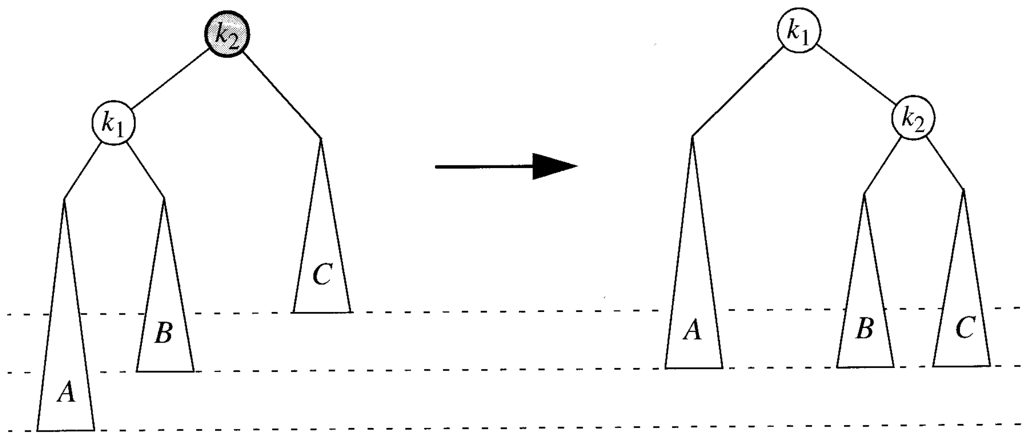

Figure 18.23 Single rotation to fix case 1

Balance is restored by tree rotations. A *single rotation* switches the roles of the parent and child while maintaining the search order.

The first case, in which the insertion occurs on the "outside" (that is, left-left or right-right), is fixed by a *single rotation* of the tree. A single rotation switches the roles of the parent and child while maintaining search order. The second case, in which the insertion occurs on the "inside" (that is, left-right or right-left), is handled by the slightly more complex *double rotation*. These are fundamental operations on the tree that are used several times in balanced tree algorithms. The remainder of this section describes these rotations and proves that they suffice to maintain the balance condition.

18.4.2 Single Rotation

A single rotation handles the outside cases (1 and 4). We rotate between a node and its child. The result is a binary search tree that satisfies the AVL property.

Figure 18.23 shows the single rotation that fixes case 1. The before picture is on the left; the after is on the right. Here is what is going on. Node k_2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree (the dashed lines are used to mark the levels in this section). The situation depicted is the only possible case 1 scenario that allows k_2 to satisfy the AVL property before the insertion but violate it afterward. Subtree A has grown to an extra level, thus causing it to be exactly two levels deeper than C. B cannot be at the same level as the new A because then k_2 would have been out of balance *before* the insertion. B cannot be at the same level as C because then k_1 would have been the first node on the path that was in violation of the AVL balancing condition (and we are claiming that k_2 is).

To ideally rebalance the tree, we want to move A up one level and C down one level. Note that this is more than the AVL property would require. To do this, we rearrange nodes into an equivalent search tree, as shown in the right-hand illustration of Figure 18.23. Here is an abstract scenario: Visualize the tree as being flexible, grab the child node k_1, close your eyes, and shake the tree, letting gravity take hold. The result is that k_1 will be the new root. The binary search tree

property tells us that in the original tree, $k_2 > k_1$, so k_2 becomes the right child of k_1 in the new tree. A and C remain as the left child of k_1 and the right child of k_2, respectively. Subtree B, which holds items that are between k_1 and k_2 in the original tree, can be placed as k_2's left child in the new tree and satisfy all the ordering requirements.

One rotation suffices to fix cases 1 and 4 in an AVL tree.

This work requires only the few child reference changes shown in Figure 18.24 and results in another binary tree that is an AVL tree. This happens because A moves up one level, B stays at the same level, and C moves down one level. k_1 and k_2 not only satisfy the AVL requirements; they also have subtrees that are exactly the same height. Furthermore, the new height of the entire subtree is *exactly the same* as the height of the original subtree prior to the insertion that caused A to grow. Thus no further updating of the heights on the path to the root is needed, and consequently, *no further rotations are needed.* This single rotation is widely used in other balanced tree algorithms in this chapter. As a result, we make it part of the class `Rotations`.

Figure 18.25 shows that after the insertion of 1 into an AVL tree, node 8 becomes unbalanced. This is clearly a case 1 problem because 1 is in 8's left-left subtree. Thus we do a single rotation between 8 and 4, thereby obtaining the tree on the right. As mentioned earlier in this section, case 4 represents a symmetric case. The required rotation is shown in Figure 18.26, and the code that implements it is shown in Figure 18.27. This method, too, is part of class `Rotations`.

18.4.3 Double Rotation

The single rotation does not fix the inside cases (2 and 3). These cases require a *double rotation,* involving three nodes and four subtrees.

The single rotation has one problem. As Figure 18.28 (page 514) shows, it does not work for case 2 (or, by symmetry, for case 3). The problem is that subtree Q is too deep; a single rotation does not make it any less deep. The *double rotation* that solves the problem is shown in Figure 18.29 (page 514).

```
1     /**
2      * Rotate binary tree node with left child.
3      * For AVL trees, this is a single rotation for case 1.
4      */
5     static BinaryNode withLeftChild( BinaryNode k2 )
6     {
7         BinaryNode k1 = k2.left;
8         k2.left = k1.right;
9         k1.right = k2;
10        return k1;
11    }
```

Figure 18.24 Code for a single rotation (case 1)

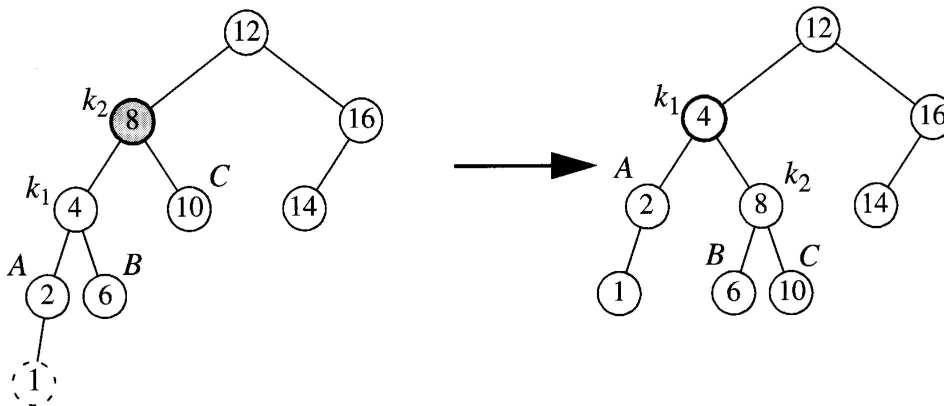

Figure 18.25 Single rotation fixes AVL tree after insertion of 1

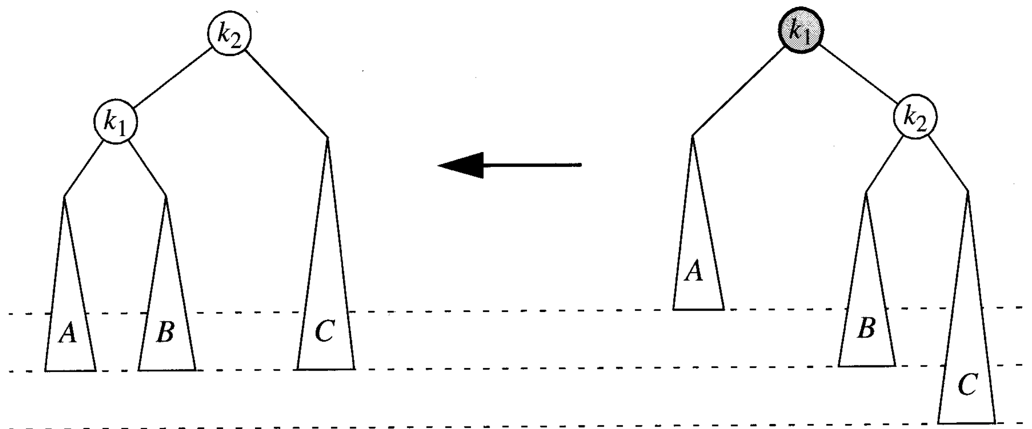

Figure 18.26 Symmetric single rotation to fix case 4

```
1    /**
2     * Rotate binary tree node with right child.
3     * For AVL trees, this is a single rotation for case 4.
4     */
5    static BinaryNode withRightChild( BinaryNode k1 )
6    {
7        BinaryNode k2 = k1.right;
8        k1.right = k2.left;
9        k2.left = k1;
10       return k2;
11   }
```

Figure 18.27 Code for a single rotation (case 4)

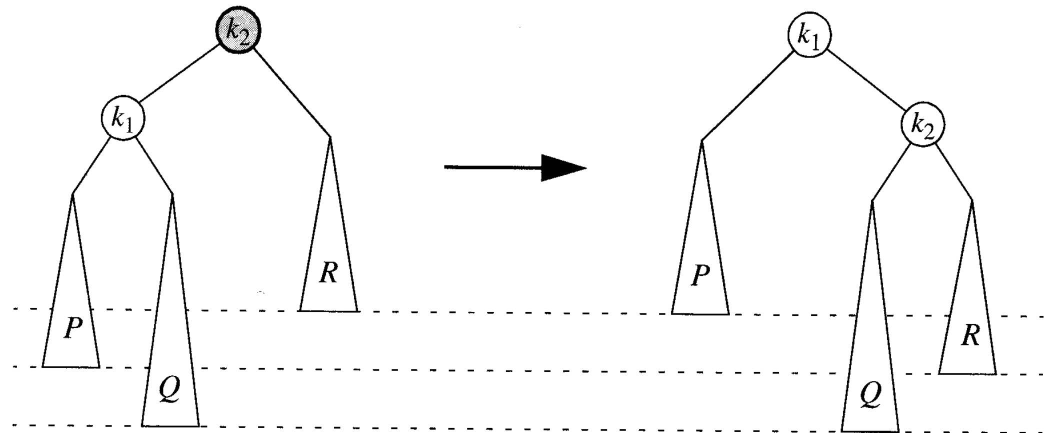

Figure 18.28 Single rotation does not fix case 2

The fact that subtree Q in Figure 18.28 has had an item inserted into it guarantees that it is not empty. We may assume that it has a root and two (possibly empty) subtrees, so we may view the tree as four subtrees connected by three nodes. We therefore rename the four trees A, B, C, and D. As Figure 18.29 suggests, exactly one of tree B or C is two levels deeper than D, but we cannot be sure which one. It turns out not to matter; in the figure, both B and C are drawn at 1.5 levels below D.

To rebalance, we see that we cannot leave k_3 as the root, while a rotation between k_3 and k_1 was shown in Figure 18.28 not to work. Hence, the only alternative is to place k_2 as the new root. This forces k_1 to be k_2's left child and k_3 to be k_2's right child. It also completely determines the resulting locations of the four subtrees. It is easy to see that the resulting tree satisfies the AVL property. Also, as was the case with the single rotation, it restores the height to what it was before the insertion, thus guaranteeing that all rebalancing and height updating is complete.

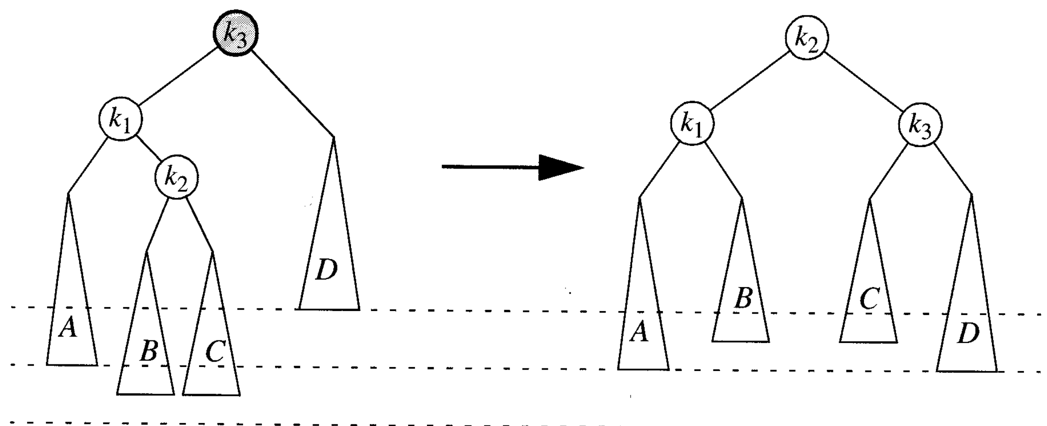

Figure 18.29 Left-right double rotation to fix case 2

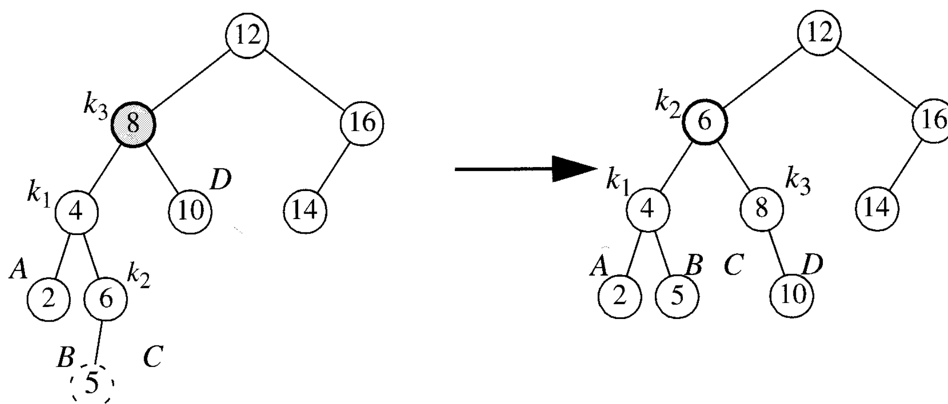

Figure 18.30 Double rotation fixes AVL tree after the insertion of 5

As an example, Figure 18.30 shows the result of inserting 5 into an AVL tree. A height imbalance is caused at node 8, thus resulting in a case 2 problem. We perform a double rotation at that node, thereby producing the tree on the right.

Figure 18.31 (page 516) shows that the symmetric case 3 can also be fixed by a double rotation. Finally, notice that although a double rotation appears complex, it turns out to be equivalent to the following:

A double rotation is equivalent to two single rotations.

- A rotation between *X*'s child and grandchild
- A rotation between *X* and its new child

The code to implement the case 2 double rotation is compact and is shown in Figure 18.32 (page 516). The mirror-image code for case 3 is shown in Figure 18.33 (page 516).

18.4.4 Summary of AVL Insertion

Here is a brief summary how an AVL insertion is implemented. A recursive algorithm turns out to be the simplest method. To insert a new node with key *X* into an AVL tree *T*, we recursively insert it into the appropriate subtree of *T* (called T_{LR}). If the height of T_{LR} does not change, then we are done. Otherwise, if a height imbalance appears in *T*, we do the appropriate single or double rotation, depending on *X* and the keys in *T* and T_{LR}, and then we are done (because the old height is the same as the postrotation height). This recursive description is best described as a casual implementation. For instance, at each node we compare the subtree's heights. In general, it is more efficient to store the result of the comparison in the node rather than maintain the height information. This avoids the repetitive calculation of balance factors. Furthermore, recursion incurs substantial overhead over an iterative version. This is because, in effect, we go down the tree and completely back up instead of stopping as soon as a rotation is performed. Consequently, in practice, other balanced search tree schemes are used.

A casual AVL implementation is relatively painless. However, it is not efficient. Better balanced search trees have since been discovered, so it is not worthwhile to implement an AVL tree.

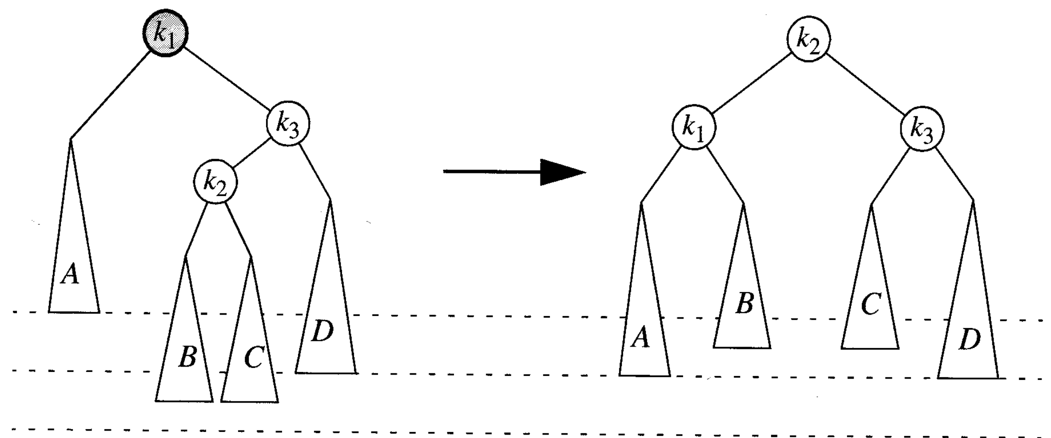

Figure 18.31 Left-right double rotation to fix case 3

```
1     /**
2      * Double rotate binary tree node: first left child
3      * with its right child; then node k3 with new left child.
4      * For AVL trees, this is a double rotation for case 2.
5      */
6     static BinaryNode doubleWithLeftChild( BinaryNode k3 )
7     {
8         k3.left = withRightChild( k3.left );
9         return withLeftChild( k3 );
10    }
```

Figure 18.32 Code for a double rotation (case 2)

```
1     /**
2      * Double rotate binary tree node: first right child
3      * with its left child; then node k1 with new right child.
4      * For AVL trees, this is a double rotation for case 3.
5      */
6     static BinaryNode doubleWithRightChild( BinaryNode k1 )
7     {
8         k1.right = withLeftChild( k1.right );
9         return withRightChild( k1 );
10    }
```

Figure 18.33 Code for a double rotation (case 3)

The *partition* step places every element except the pivot in one of two groups.

In the *partition* step, every element in S, except for the pivot, is placed in either L (which stands for the left-hand part of the array) or R (which stands for the right-hand part of the array). The intent is that elements that are smaller than the pivot go to L, while elements that are larger than the pivot go to R. The description in the algorithm, however, ambiguously describes what to do with elements equal to the pivot. It allows each instance of a duplicate to go into either subset, specifying only that it must go to one or the other. Part of a good Java implementation is handling this case as efficiently as possible. Once again, the analysis will allow us to make an informed decision.

Figure 8.8 shows the action of quicksort on a set of numbers. The pivot is chosen (by chance) to be 65. The remaining elements in the set are partitioned into two smaller subsets. Each of these groups is then sorted recursively. By the third rule of recursion, we can assume that this step works. The sorted arrangement of the entire group is then trivially obtained. In a Java implementation, the items would be stored in a part of an array delimited by `low` and `high`. After the partitioning step, the pivot would wind up in some array cell `p`. The recursive calls would then be on the parts from `low` to `p-1` and then `p+1` to `high`.

Because recursion allows us to take the giant leap of faith, the correctness of the algorithm is guaranteed as follows:

- The group of small elements is sorted, by virtue of the recursion.
- The largest element in the group of small elements is not larger than the pivot, by virtue of the partition.
- The pivot is not larger than the smallest element in the group of large elements, by virtue of the partition.
- The group of large elements is sorted, by virtue of the recursion.

Quicksort is fast because the partitioning step can be performed quickly and in place.

Although the correctness of the algorithm is easily established, it is not clear why it is any faster than mergesort. Like mergesort, it recursively solves two subproblems and requires linear additional work (in the form of the partitioning step). Unlike with mergesort, however, the subproblems are not guaranteed to be of equal size. This is bad for performance. Quicksort is faster than mergesort because the partitioning step can be performed significantly faster than the merging step can. In particular, the partitioning step can be performed without using an extra array and the code to implement it is very compact and efficient. This makes up for the lack of equal-sized subproblems.

8.6.2 Analysis of Quicksort

The algorithm description leaves several questions unanswered. How do we choose the pivot? How do we perform the partition? What do we do if we see an element that is equal to the pivot? All these questions can dramatically affect the running time of the algorithm. We will perform an analysis to help us decide how we should implement the unspecified steps in quicksort.

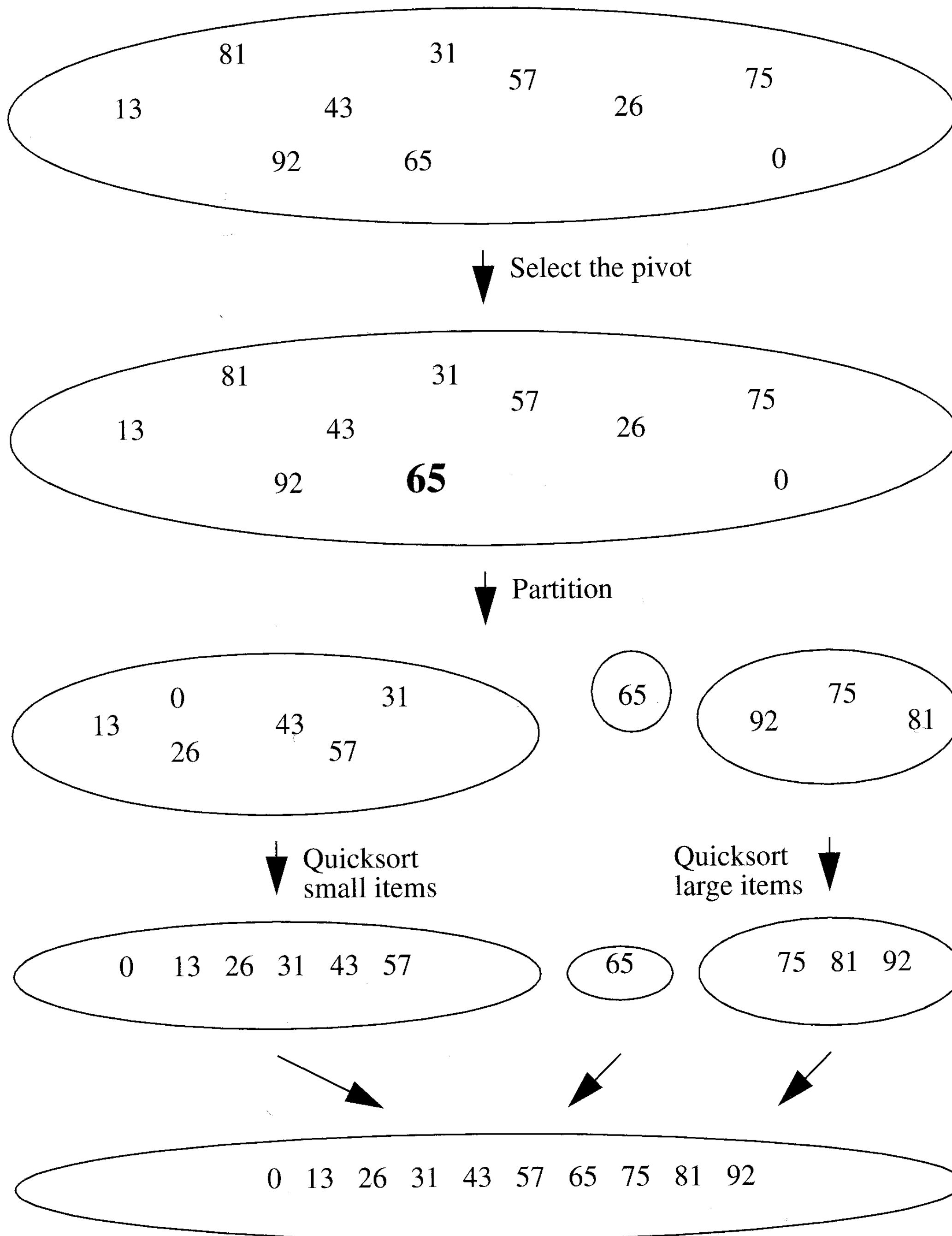

Figure 8.8 The steps of quicksort

Best Case

The best case for quicksort is that the pivot partitions the set into two equal-sized subsets and that this happens at each stage of the recursion. We then have two half-sized recursive calls plus linear overhead, which matches the performance of mergesort. The running time, for this case, is $O(N \log N)$. (We have not actually proved that this is the best case. Such a proof is possible; however, we omit the details.)

The best case occurs when the partition always splits into equal subsets. The running time is $O(N \log N)$.

Worst Case

The worst case occurs when the partition repeatedly generates an empty subset. The running time is $O(N^2)$.

Since equal-sized subsets are good for quicksort, one might expect that unequal-sized subsets are bad. This is indeed the case. Suppose that in each step of the recursion, the pivot happens to be the smallest element. Then the set of small elements L will be empty and the set of large elements R will have all the elements except for the pivot. We will then have to recursively call quicksort on subset R. Suppose $T(N)$ is the running time to quicksort N elements and we assume that the time to sort 0 or 1 element is just 1 time unit. Suppose also that we charge N units to partition a set that contains N elements. Then for $N > 1$, we obtain a running time that satisfies

$$T(N) = T(N-1) + N. \quad \textbf{(8.1)}$$

In other words, Equation 8.1 states that the time to quicksort N items is equal to the time to recursively sort the $N-1$ items in the subset of larger elements plus the N units of cost to perform the partition. This assumes that in each step of the iteration, we are unfortunate enough to pick the smallest element as the pivot. To simplify the analysis, we normalize by throwing out constant factors. We can solve this recurrence by telescoping Equation 8.1 repeatedly:

$$\begin{aligned} T(N) &= T(N-1) + N \\ T(N-1) &= T(N-2) + (N-1) \\ T(N-2) &= T(N-3) + (N-2) \\ &\dots \\ T(2) &= T(1) + 2 \end{aligned} \quad \textbf{(8.2)}$$

When we add up everything in Equation 8.2, we obtain massive cancellations, yielding

$$T(N) = T(1) + 2 + 3 + \dots + N = N(N+1)/2 = O(N^2). \quad \textbf{(8.3)}$$

This analysis verifies the intuition that an uneven split is bad. We spend N units of time to partition and then have to make a recursive call for $N-1$. Then we spend $N-1$ units to partition that group, only to have to make a recursive call for $N-2$ elements. In that call, we spend $N-2$ units performing the partition, and so on. The total time to perform all the partitions throughout the recursive calls exactly matches what is obtained in Equation 8.3. This tells us that when implementing the selection of the pivot and the partitioning step, we do not want to do anything that might encourage the subsets to be unbalanced in size.

Average Case

The average case is $O(N \log N)$. Although this seems intuitive, a formal proof is required.

The first two analyses tell us that the best and worst cases are wildly different. Naturally, we want to know what happens in the average case. We would expect that since each subproblem is half of the original on average, the $O(N \log N)$ would now become an average-case bound. Such expectation, while correct for the particular quicksort application we examine here, does not constitute a formal

proof. Averages cannot be thrown around lightly. For example, suppose we have a pivot algorithm that is guaranteed to select only the smallest or largest element, each with probability 1/2. Then the average size of the small group of elements is roughly $N/2$, as is the average size of the large group of elements (because each is equally likely to have 0 or $N-1$ elements). But the running time of quicksort with that pivot selection will always be quadratic because we always get a poor split of elements. We must be careful how we assign the label "average." We can argue that the group of small elements is as likely to contain 0, 1, 2, ..., or $N-1$ elements. This is also true for the group of large elements. Under this assumption, we can establish that the average-case running time is indeed $O(N \log N)$.

Since the cost of quicksort of N items is equal to N units for the partitioning step plus the cost of the two recursive calls, we need to determine the average cost of each of the recursive calls. If $T(N)$ represents the average cost to quicksort N elements, the average cost of each recursive call is equal to the average, over all possible subproblem sizes, of the average cost of a recursive call on the subproblem:

The average cost of a recursive call is obtained by averaging the costs of all possible subproblem sizes.

$$T(L) = T(R) = \frac{T(0) + T(1) + T(2) + \ldots + T(N-1)}{N} \qquad \textbf{(8.4)}$$

Equation 8.4 states that we are looking at the costs for each possible subset size and averaging them. Since we have two recursive calls plus linear time to perform the partition, we obtain

$$T(N) = 2\left(\frac{T(0) + T(1) + T(2) + \ldots + T(N-1)}{N}\right) + N. \qquad \textbf{(8.5)}$$

To solve Equation 8.5, we begin by multiplying both sides by N, obtaining

The average running time is given by $T(N)$. We solve Equation 8.5 by removing all but the most recent recursive value of T.

$$NT(N) = 2(T(0) + T(1) + T(2) + \ldots + T(N-1)) + N^2. \qquad \textbf{(8.6)}$$

We then write Equation 8.6 for the case $N-1$, with the idea being that we can perform a subtraction and greatly simplify the equation. If we do this, we obtain

$$(N-1)T(N-1) = 2(T(0) + T(1) + T(2) + \ldots + T(N-2)) + (N-1)^2. \qquad \textbf{(8.7)}$$

Now if we subtract Equation 8.7 from Equation 8.6, we obtain

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2N - 1.$$

We rearrange terms and drop the insignificant –1 on the right to obtain

$$NT(N) = (N+1)T(N-1) + 2N. \qquad \textbf{(8.8)}$$

We now have a formula for $T(N)$ in terms of $T(N-1)$ only. Again, the idea is to telescope, but Equation 8.8 is in the wrong form. If we divide Equation 8.8 by $N(N+1)$, we obtain

Once we have $T(N)$ in terms of $T(N-1)$ only, we attempt to telescope.

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1}.$$

Now we can telescope:

$$\begin{aligned}\frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2}{N+1}\\ \frac{T(N-1)}{N} &= \frac{T(N-2)}{N-1} + \frac{2}{N}\\ \frac{T(N-2)}{N-1} &= \frac{T(N-3)}{N-2} + \frac{2}{N-1}\\ &\dots\\ \frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2}{3}\end{aligned} \tag{8.9}$$

If we add all the equations in Equation 8.9, we obtain

$$\begin{aligned}\frac{T(N)}{N+1} &= \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N} + \frac{1}{N+1}\right)\\ &= 2\left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N+1}\right) - \frac{5}{2}\\ &= O(\log N)\end{aligned} \tag{8.10}$$

We use the fact that the Nth harmonic number is $O(\log N)$.

The last line in Equation 8.10 follows from Theorem 5.5. When we multiply both sides by $N+1$, we obtain the final result of

$$T(N) = O(N \log N). \tag{8.11}$$

8.6.3 Picking the Pivot

Now that we have established that quicksort will run in $O(N \log N)$ time on average, our primary concern is to ensure that the worst case does not occur. By performing a complex analysis, we can compute the standard deviation of quicksort's running time. The result is that if a single random permutation is presented, it is almost certain that the running time used to sort it will be close to the average. Thus we must be careful that degenerate inputs do not result in bad running times. Degenerate inputs include data that is already sorted and data that contains only N completely identical elements. As we will see below, sometimes it is the easy cases that give algorithms trouble.

A Wrong Way

Picking the pivot is crucial to good performance. Never choose the first element as the pivot.

The popular, uninformed choice is to use the first element (that is, the element that is in position `low`) as the pivot. This is acceptable if the input is random, but if the input is presorted or in reverse order, then the pivot provides a poor partition because it will be an extreme element. Furthermore, this behavior will continue recursively. As we saw earlier in the chapter, we would get quadratic