matically. Exercise 18.30 describes some alternate deletion strategies. All of them appear to exhibit degenerate behavior for sufficiently long random sequences.

The most important problem is not the potential imbalance caused by the `Remove` algorithm, but instead 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 bubble sort executed instead. The resulting running time is completely unacceptable. Moreover, it is not just sorted input that is problematic but 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.

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, and because they are so balanced they also 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.

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

18.4 AVL Trees

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. The AVL tree is a binary search tree that has an additional balance condition. This balance condition must be easy to maintain and 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, but 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.

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.

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.

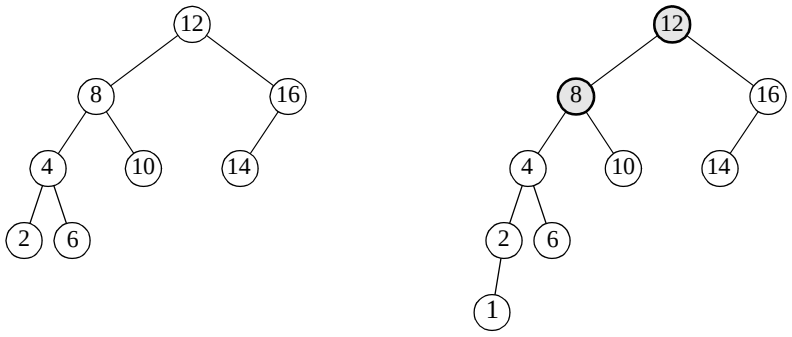

Figure 18.21 Two binary search trees: the left tree is an AVL tree, but the right tree is not (unbalanced nodes are darkened)

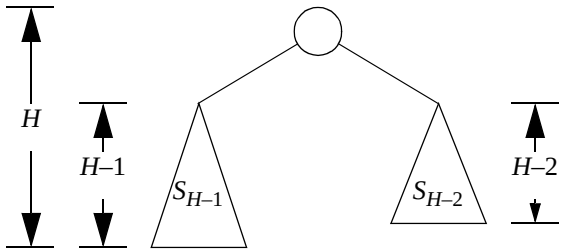

Figure 18.22 Minimum tree of height *H*

Every node in an AVL tree has subtrees whose heights differ by at most 1. An empty subtree has height –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 were inserted using the usual algorithm, then node 16 would also be in violation because the left subtree would have height 1 while the right subtree would have height –1.

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

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. If so, then the maximum depth of an N item tree is given by $\log_C N$.

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 shows that the smallest AVL tree of height H must have subtrees of height $H-1$ and $H-2$, because at least one subtree has height $H-1$ and the balance condition implies that subtree heights can differ by at most one. These subtrees must themselves have the fewest number of nodes for their height, 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 7.7 we know that $F_i \approx \phi^i/\sqrt{5}$, 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. This allows us to conclude that its depth is at most logarithmic. A precise calculation allows us to determine that 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.

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

As we will see, the depth of an average node in a randomly constructed AVL tree is 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, because as we can see in Figure 18.21, an insertion (or deletion) can destroy the balance of several nodes in the tree. The balance must then be restored before the operation can be considered complete. We will describe the insertion algorithm and leave deletion for Exercise 18.13.

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

The key observation, which also applies for almost all of the balanced search tree algorithms, 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. 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. We will show how to rebalance the tree at the first (that is, the deepest) such node, and we will prove that the 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.

Let us call the node that must be rebalanced *X*. Since any node has at most two children, and a height imbalance requires that *X*'s two subtrees' heights differ by two, it is easy to see that a violation might occur in 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, as a matter of theory, there are two basic cases. From a programming perspective, of course, there are still four cases (and numerous special cases).

Balance is restored by tree rotations. A *single rotation* switches the roles of the parent and child while maintaining 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 we will see 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, and the after is on the right. Let us analyze carefully 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, 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, and *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 would like 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 illustration on the right 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 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.

As a result of this work, which requires only the few pointer changes shown in Figure 18.24,[1] we have 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, but 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*.

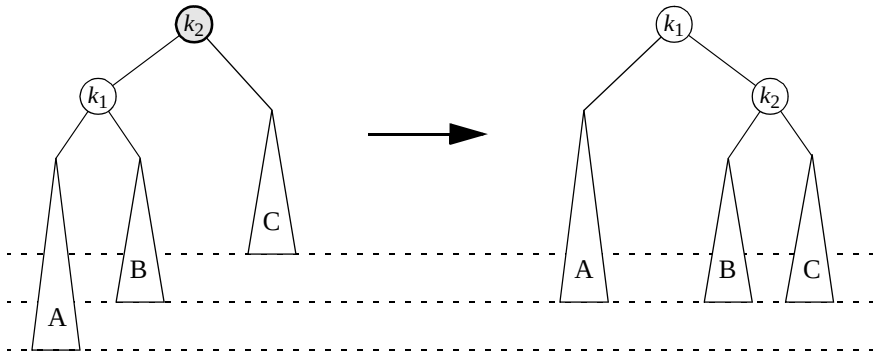

Figure 18.23 Single rotation to fix case 1

```
 1 // Rotate binary tree note with left child
 2 // For AVL trees, this is a single rotation for case 1
 3
 4 template <class Etype>
 5 BinaryNode<Etype> *
 6 RotateWithLeftChild( BinaryNode<Etype> *K2 )
 7 {
 8     BinaryNode<Etype> *K1 = K2->Left;
 9     K2->Left = K1->Right;
10     K1->Right = K2;
11     return K1;
12 }
```

Figure 18.24 Code for single rotation (case 1)

1. It would be better if the single and double rotation routines accepted a pointer passed by reference. Recall, however, that the template matching algorithm might not find this as a match. Since these routines are widely used in this chapter, we stick with safe C++ syntax. Note that we could use the more convenient syntax for member functions.

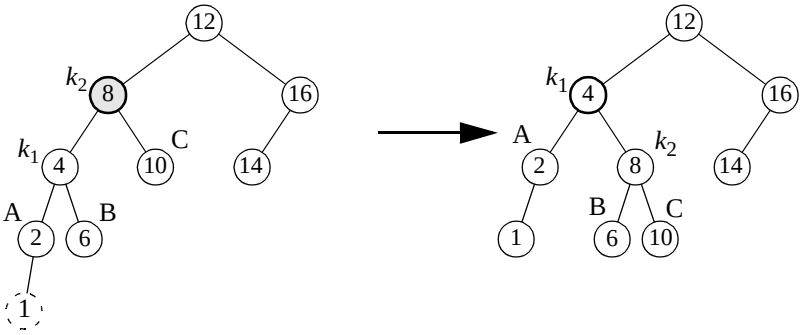

Figure 18.25 Single rotation fixes AVL tree after insertion of 1

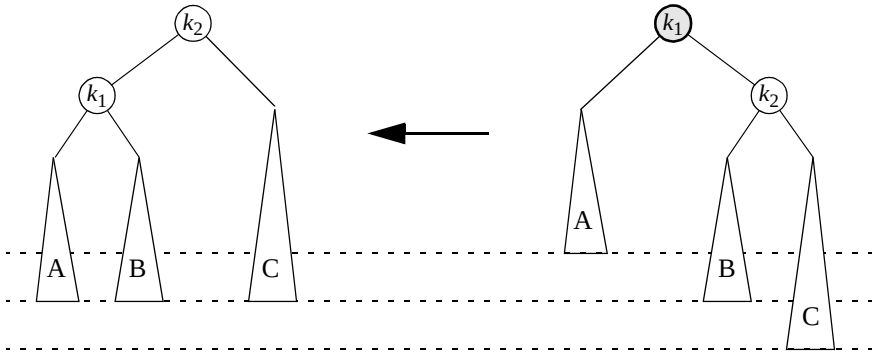

Figure 18.26 Symmetric single rotation to fix case 4

```
 1 // Rotate binary tree note with right child
 2 // For AVL trees, this is a single rotation for case 4
 3
 4 template <class Etype>
 5 BinaryNode<Etype> *
 6 RotateWithRightChild( BinaryNode<Etype> *K1 )
 7 {
 8     BinaryNode<Etype> *K2 = K1->Right;
 9     K1->Right = K2->Left;
10     K2->Left = K1;
11     return K2;
12 }
```

Figure 18.27 Code for single rotation (case 4)

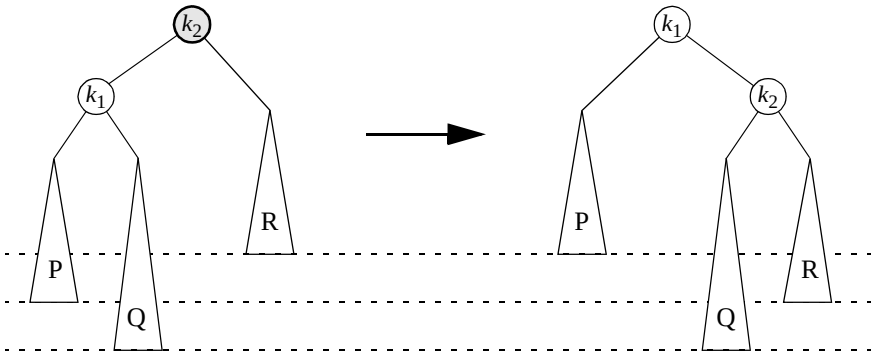

Figure 18.28 Single rotation does not fix case 2

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, obtaining the tree on the right. As we mentioned earlier, 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.

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

The fact that subtree *Q* in Figure 18.29 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 the diagram 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 Figure 18.29 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, and a rotation between k_3 and k_1 was shown in Figure 18.28 not to work, so 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, and it also completely determines the resulting locations of the four subtrees. It is easy to see that the resulting tree satisfies the AVL property, and 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.

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

A double rotation is equivalent to two single rotations.

Figure 18.31 shows that the symmetric case 3 can also be fixed by a double rotation. Finally, we remark that although a double rotation appears complex, it turns out that it is equivalent to the following:

- Rotate between *X*'s child and grandchild.
- Rotate between *X* and its new child.

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

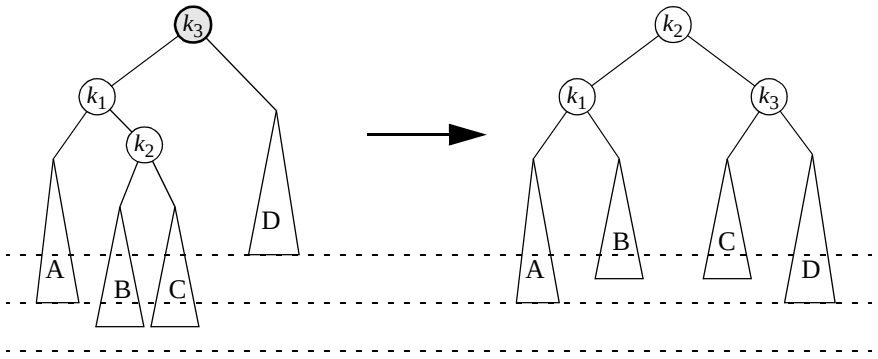

Figure 18.29 Left-right double rotation to fix case 2

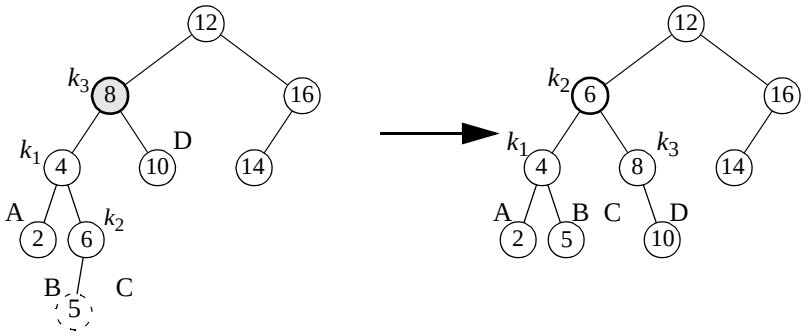

Figure 18.30 Double rotation fixes AVL tree after insertion of 5

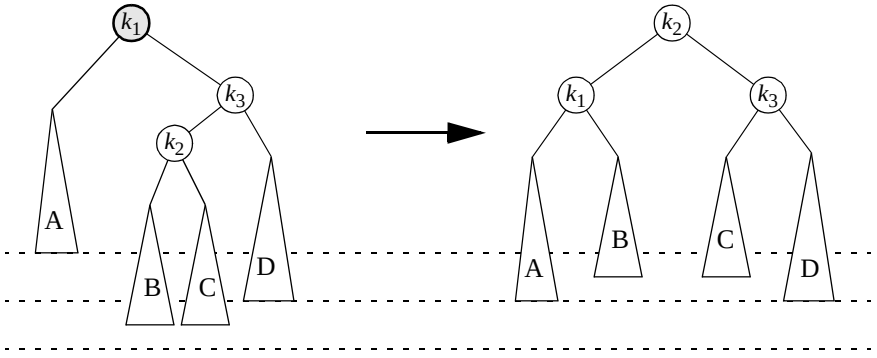

Figure 18.31 Left-right double rotation to fix case 3

```
// Double rotate binary tree node: first left child
// with its right child; then node K3 with new left child
// For AVL trees, this is a double rotation for case 2

template <class Etype>
BinaryNode<Etype> *
DoubleRotateWithLeftChild( BinaryNode<Etype> *K3 )
{
    K3->Left = RotateWithRightChild( K3->Left );
    return RotateWithLeftChild( K3 );
}
```

Figure 18.32 Code for double rotation (case 2)

```
// Double rotate binary tree node: first right child
// with its left child; then node K1 with new right child
// For AVL trees, this is a double rotation for case 3

template <class Etype>
BinaryNode<Etype> *
DoubleRotateWithRightChild( BinaryNode<Etype> *K1 )
{
    K1->Right = RotateWithLeftChild( K1->Right );
    return RotateWithRightChild( K1 );
}
```

Figure 18.33 Code for double rotation (case 3)

18.4.4 Summary of AVL Insertion

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.

Let us briefly summarize 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 (let us call this 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 are done (because the old height is the same as the post-rotation 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 maintaining the height information. This avoids repetitive calculation of balance factors. Furthermore, recursion incurs substantial overhead over an iterative version, because in effect we go down the tree and completely back up instead of stopping as soon as a rotation is performed. Consequently, other balanced search tree schemes are used.

18.5 Red Black Trees

A *red black tree* is a good alternative to the AVL tree. The coding details tend to give a faster implementation because a single top-down pass can be used during the insertion and deletion routines.

A historically popular alternative to the AVL tree is the *red black tree*. As on AVL trees, operations on red black trees take logarithmic worst-case time. The main advantage of red black trees is that a single top-down pass can be used during the insertion and deletion routines. This contrasts with an AVL tree in which a pass down the tree is used to establish the insertion point, and a second pass up the tree is used to update heights and possibly rebalance. As a result, a careful nonrecursive implementation of the red black tree is simpler and faster than an AVL tree implementation.

A red black tree is a binary search tree with the following ordering properties:

1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a `NULL` pointer must contain the same number of black nodes.

Consecutive red nodes are disallowed, and all paths have the same number of black nodes.

Red nodes are shaded throughout this chapter.

In our discussion of red black trees, we will draw red nodes by shading them. Figure 18.34 shows a red black tree. Every path from the root to a `NULL` node contains three black nodes.

The depth of a red black tree is guaranteed to be logarithmic. Typically the depth is the same as an AVL tree.

We can show by induction that if every path from the root to a `NULL` node contains B black nodes, then there must be at least $2^B - 1$ black nodes in the tree. Furthermore, since the root is black and we may not have two consecutive red nodes on a path, we find that the height of a red black tree is at most $2\log(N + 1)$. Consequently, searching is guaranteed to be a logarithmic operation.