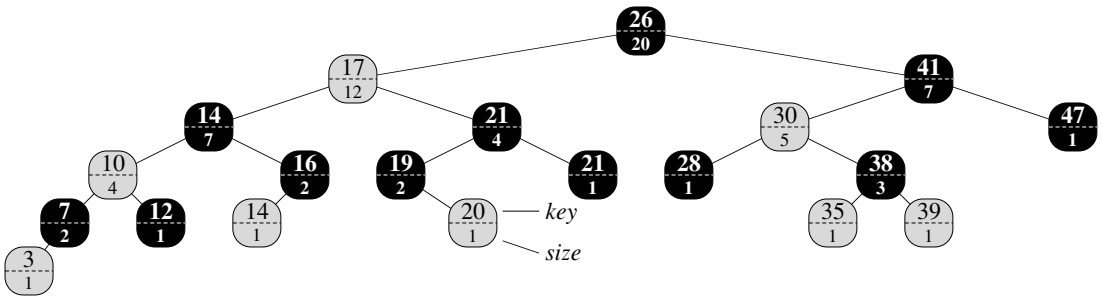

Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual fields, each node x has a field $size[x]$, which is the number of nodes in the subtree rooted at x.

A data structure that can support fast order-statistic operations is shown in Figure 14.1. An ***order-statistic tree*** T is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree fields $key[x]$, $color[x]$, $p[x]$, $left[x]$, and $right[x]$ in a node x, we have another field $size[x]$. This field contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is, the size of the subtree. If we define the sentinel's size to be 0, that is, we set $size[nil[T]]$ to be 0, then we have the identity

$$size[x] = size[left[x]] + size[right[x]] + 1 .$$

We do not require keys to be distinct in an order-statistic tree. (For example, the tree in Figure 14.1 has two keys with value 14 and two keys with value 21.) In the presence of equal keys, the above notion of rank is not well defined. We remove this ambiguity for an order-statistic tree by defining the rank of an element as the position at which it would be printed in an inorder walk of the tree. In Figure 14.1, for example, the key 14 stored in a black node has rank 5, and the key 14 stored in a red node has rank 6.

Retrieving an element with a given rank

Before we show how to maintain this size information during insertion and deletion, let us examine the implementation of two order-statistic queries that use this additional information. We begin with an operation that retrieves an element with a given rank. The procedure OS-SELECT(x, i) returns a pointer to the node containing the ith smallest key in the subtree rooted at x. To find the ith smallest key in an order-statistic tree T, we call OS-SELECT$(root[T], i)$.

```
OS-SELECT(x, i)
1  r ← size[left[x]]+1
2  if i = r
3     then return x
4  elseif i < r
5     then return OS-SELECT(left[x], i)
6  else return OS-SELECT(right[x], i − r)
```

The idea behind OS-SELECT is similar to that of the selection algorithms in Chapter 9. The value of $size[left[x]]$ is the number of nodes that come before x in an inorder tree walk of the subtree rooted at x. Thus, $size[left[x]] + 1$ is the rank of x within the subtree rooted at x.

In line 1 of OS-SELECT, we compute r, the rank of node x within the subtree rooted at x. If $i = r$, then node x is the ith smallest element, so we return x in line 3. If $i < r$, then the ith smallest element is in x's left subtree, so we recurse on $left[x]$ in line 5. If $i > r$, then the ith smallest element is in x's right subtree. Since there are r elements in the subtree rooted at x that come before x's right subtree in an inorder tree walk, the ith smallest element in the subtree rooted at x is the $(i - r)$th smallest element in the subtree rooted at $right[x]$. This element is determined recursively in line 6.

To see how OS-SELECT operates, consider a search for the 17th smallest element in the order-statistic tree of Figure 14.1. We begin with x as the root, whose key is 26, and with $i = 17$. Since the size of 26's left subtree is 12, its rank is 13. Thus, we know that the node with rank 17 is the $17 - 13 = 4$th smallest element in 26's right subtree. After the recursive call, x is the node with key 41, and $i = 4$. Since the size of 41's left subtree is 5, its rank within its subtree is 6. Thus, we know that the node with rank 4 is the 4th smallest element in 41's left subtree. After the recursive call, x is the node with key 30, and its rank within its subtree is 2. Thus, we recurse once again to find the $4 - 2 = 2$nd smallest element in the subtree rooted at the node with key 38. We now find that its left subtree has size 1, which means it is the second smallest element. Thus, a pointer to the node with key 38 is returned by the procedure.

Because each recursive call goes down one level in the order-statistic tree, the total time for OS-SELECT is at worst proportional to the height of the tree. Since the tree is a red-black tree, its height is $O(\lg n)$, where n is the number of nodes. Thus, the running time of OS-SELECT is $O(\lg n)$ for a dynamic set of n elements.

Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T, the procedure OS-RANK returns the position of x in the linear order determined by an inorder tree walk of T.

```
OS-RANK(T, x)
1  r ← size[left[x]] + 1
2  y ← x
3  while y ≠ root[T]
4        do if y = right[p[y]]
5              then r ← r + size[left[p[y]]] + 1
6             y ← p[y]
7  return r
```

The procedure works as follows. The rank of x can be viewed as the number of nodes preceding x in an inorder tree walk, plus 1 for x itself. OS-RANK maintains the following loop invariant:

> At the start of each iteration of the **while** loop of lines 3–6, r is the rank of $key[x]$ in the subtree rooted at node y.

We use this loop invariant to show that OS-RANK works correctly as follows:

Initialization: Prior to the first iteration, line 1 sets r to be the rank of $key[x]$ within the subtree rooted at x. Setting $y \leftarrow x$ in line 2 makes the invariant true the first time the test in line 3 executes.

Maintenance: At the end of each iteration of the **while** loop, we set $y \leftarrow p[y]$. Thus we must show that if r is the rank of $key[x]$ in the subtree rooted at y at the start of the loop body, then r is the rank of $key[x]$ in the subtree rooted at $p[y]$ at the end of the loop body. In each iteration of the **while** loop, we consider the subtree rooted at $p[y]$. We have already counted the number of nodes in the subtree rooted at node y that precede x in an inorder walk, so we must add the nodes in the subtree rooted at y's sibling that precede x in an inorder walk, plus 1 for $p[y]$ if it, too, precedes x. If y is a left child, then neither $p[y]$ nor any node in $p[y]$'s right subtree precedes x, so we leave r alone. Otherwise, y is a right child and all the nodes in $p[y]$'s left subtree precede x, as does $p[y]$ itself. Thus, in line 5, we add $size[left[p[y]]] + 1$ to the current value of r.

Termination: The loop terminates when $y = root[T]$, so that the subtree rooted at y is the entire tree. Thus, the value of r is the rank of $key[x]$ in the entire tree.

As an example, when we run OS-RANK on the order-statistic tree of Figure 14.1 to find the rank of the node with key 38, we get the following sequence of values of $key[y]$ and r at the top of the **while** loop:

iteration	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

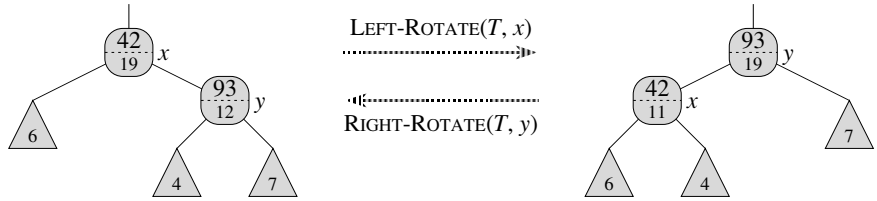

Figure 14.2 Updating subtree sizes during rotations. The link around which the rotation is performed is incident on the two nodes whose *size* fields need to be updated. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

The rank 17 is returned.

Since each iteration of the **while** loop takes $O(1)$ time, and y goes up one level in the tree with each iteration, the running time of OS-RANK is at worst proportional to the height of the tree: $O(\lg n)$ on an n-node order-statistic tree.

Maintaining subtree sizes

Given the *size* field in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information. But unless these fields can be efficiently maintained by the basic modifying operations on red-black trees, our work will have been for naught. We shall now show that subtree sizes can be maintained for both insertion and deletion without affecting the asymptotic running time of either operation.

We noted in Section 13.3 that insertion into a red-black tree consists of two phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and ultimately performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, we simply increment $size[x]$ for each node x on the path traversed from the root down toward the leaves. The new node added gets a *size* of 1. Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* fields is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: only two nodes have their *size* fields invalidated. The link around which the rotation is performed is incident on these two nodes. Referring to the code for LEFT-ROTATE(T, x) in Section 13.2, we add the following lines:

13 $size[y] \leftarrow size[x]$
14 $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$

Figure 14.2 illustrates how the fields are updated. The change to RIGHT-ROTATE is symmetric.