
14 Augmenting Data Structures

Some engineering situations require no more than a “textbook” data structure—such as a doubly linked list, a hash table, or a binary search tree—but many others require a dash of creativity. Only in rare situations will you need to create an entirely new type of data structure, though. More often, it will suffice to augment a textbook data structure by storing additional information in it. You can then program new operations for the data structure to support the desired application. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

This chapter discusses two data structures that we construct by augmenting red-black trees. Section 14.1 describes a data structure that supports general order-statistic operations on a dynamic set. We can then quickly find the i th smallest number in a set or the rank of a given element in the total ordering of the set. Section 14.2 abstracts the process of augmenting a data structure and provides a theorem that can simplify the process of augmenting red-black trees. Section 14.3 uses this theorem to help design a data structure for maintaining a dynamic set of intervals, such as time intervals. Given a query interval, we can then quickly find an interval in the set that overlaps it.

14.1 Dynamic order statistics

Chapter 9 introduced the notion of an order statistic. Specifically, the i th order statistic of a set of n elements, where $i \in \{1, 2, \dots, n\}$, is simply the element in the set with the i th smallest key. We saw how to determine any order statistic in $O(n)$ time from an unordered set. In this section, we shall see how to modify red-black trees so that we can determine any order statistic for a dynamic set in $O(\lg n)$ time. We shall also see how to compute the **rank** of an element—its position in the linear order of the set—in $O(\lg n)$ time.

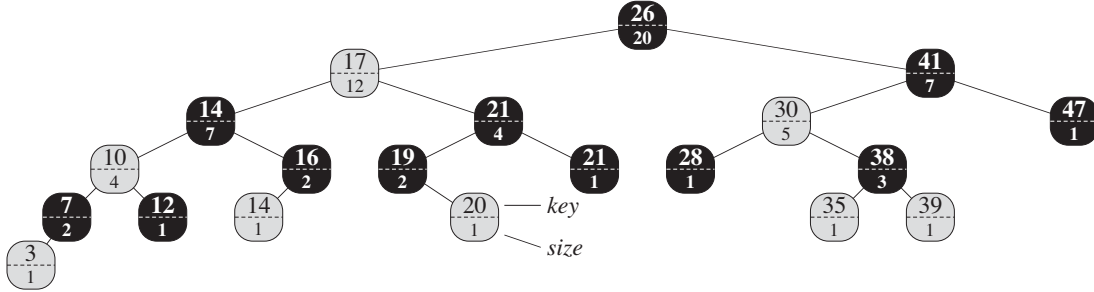


Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x .

Figure 14.1 shows a data structure that can support fast order-statistic operations. An **order-statistic tree** T is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$ in a node x , we have another attribute, $x.size$. This attribute 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 $T.nil.size$ to be 0—then we have the identity

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

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 i th smallest key in the subtree rooted at x . To find the node with the i th smallest key in an order-statistic tree T , we call $OS-SELECT(T.root, i)$.

```

OS-SELECT( $x, i$ )
1   $r = x.\text{left.size} + 1$ 
2  if  $i == r$ 
3      return  $x$ 
4  elseif  $i < r$ 
5      return OS-SELECT( $x.\text{left}, i$ )
6  else return OS-SELECT( $x.\text{right}, i - r$ )

```

In line 1 of OS-SELECT, we compute r , the rank of node x within the subtree rooted at x . The value of $x.\text{left.size}$ is the number of nodes that come before x in an inorder tree walk of the subtree rooted at x . Thus, $x.\text{left.size} + 1$ is the rank of x within the subtree rooted at x . If $i = r$, then node x is the i th smallest element, and so we return x in line 3. If $i < r$, then the i th smallest element resides in x 's left subtree, and so we recurse on $x.\text{left}$ in line 5. If $i > r$, then the i th smallest element resides in x 's right subtree. Since the subtree rooted at x contains r elements that come before x 's right subtree in an inorder tree walk, the i th smallest element in the subtree rooted at x is the $(i - r)$ th smallest element in the subtree rooted at $x.\text{right}$. Line 6 determines this element recursively.

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, the procedure returns a pointer to the node with key 38.

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   $r = x.left.size + 1$ 
2   $y = x$ 
3  while  $y \neq T.root$ 
4      if  $y == y.p.right$ 
5           $r = r + y.p.left.size + 1$ 
6       $y = y.p$ 
7  return  $r$ 

```

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

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

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

The procedure returns the rank 17.

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* attribute in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information. But unless we can efficiently maintain these attributes within the basic modifying operations on red-black trees, our work will have been for naught. We shall now show how to maintain subtree sizes 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 performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, we simply increment $x.size$ for each node x on the simple 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* attributes 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* attributes 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   $y.size = x.size$ 
14   $x.size = x.left.size + x.right.size + 1$ 
```

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

Since at most two rotations are performed during insertion into a red-black tree, we spend only $O(1)$ additional time updating *size* attributes in the second phase. Thus, the total time for insertion into an n -node order-statistic tree is $O(\lg n)$, which is asymptotically the same as for an ordinary red-black tree.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. (See Section 13.4.) The first phase either removes one node y from the tree or moves upward it within the tree. To update the subtree sizes, we simply traverse a simple path from node y (starting from its original position within the tree) up to the root, decrementing the *size*

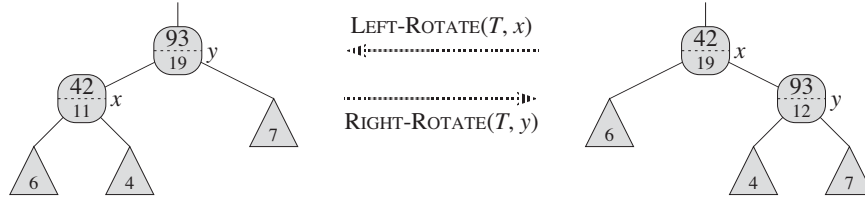


Figure 14.2 Updating subtree sizes during rotations. The link around which we rotate is incident on the two nodes whose *size* attributes need to be updated. The updates are local, requiring only the *size* information stored in x , y , and the roots of the subtrees shown as triangles.

attribute of each node on the path. Since this path has length $O(\lg n)$ in an n -node red-black tree, the additional time spent maintaining *size* attributes in the first phase is $O(\lg n)$. We handle the $O(1)$ rotations in the second phase of deletion in the same manner as for insertion. Thus, both insertion and deletion, including maintaining the *size* attributes, take $O(\lg n)$ time for an n -node order-statistic tree.

Exercises

14.1-1

Show how $\text{OS-SELECT}(T.\text{root}, 10)$ operates on the red-black tree T of Figure 14.1.

14.1-2

Show how $\text{OS-RANK}(T, x)$ operates on the red-black tree T of Figure 14.1 and the node x with $x.\text{key} = 35$.

14.1-3

Write a nonrecursive version of OS-SELECT .

14.1-4

Write a recursive procedure $\text{OS-KEY-RANK}(T, k)$ that takes as input an order-statistic tree T and a key k and returns the rank of k in the dynamic set represented by T . Assume that the keys of T are distinct.

14.1-5

Given an element x in an n -node order-statistic tree and a natural number i , how can we determine the i th successor of x in the linear order of the tree in $O(\lg n)$ time?