17 Augmenting Data Structures

Some solutions 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. Rarely will you need to create an entirely new type of data structure, though. More often, you can augment a textbook data structure by storing additional information in it. You can then program new operations for the data structure to support your 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 based on red-black trees that are augmented with additional information. Section 17.1 describes a data structure that supports general order-statistic operations on a dynamic set: quickly finding the ith smallest number or the rank of a given element. Section 17.2 abstracts the process of augmenting a data structure and provides a theorem that you can use when augmenting red-black trees. Section 17.3 uses this theorem to help design a data structure for maintaining a dynamic set of intervals, such as time intervals. You can use this data structure to quickly find an interval that overlaps a given query interval.

17.1 Dynamic order statistics

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

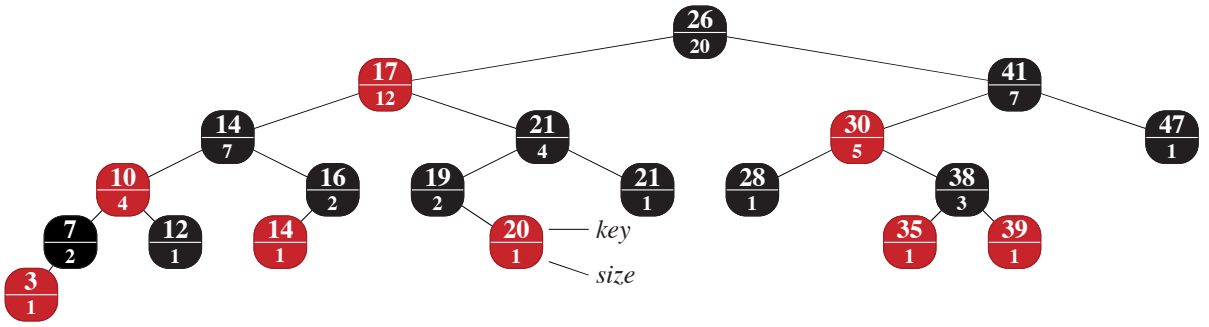

Figure 17.1 An order-statistic tree, which is an augmented red-black tree. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x.

Figure 17.1 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. Each node x contains the usual red-black tree attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$, along with a new attribute, $x.size$. This attribute contains the number of internal nodes in the subtree rooted at x (including x itself, but not including any sentinels), 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

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

Keys need not be distinct in an order-statistic tree. For example, the tree in Figure 17.1 has two keys with value 14 and two keys with value 21. When equal keys are present, 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 17.1, 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 the element with a given rank

Before we show how to maintain the size information during insertion and deletion, let's see how to implement two order-statistic queries that use this additional information. We begin with an operation that retrieves the element with a given rank. The procedure OS-SELECT(x, i) on the following page returns a pointer to the node containing the ith smallest key in the subtree rooted at x. To find the node with the ith smallest key in an order-statistic tree T, call OS-SELECT$(T.root, i)$.

Here is how OS-SELECT works. Line 1 computes r, the rank of node x within the subtree rooted at x. The value of $x.left.size$ is the number of nodes that come

```
OS-SELECT(x, i)
1  r = x.left.size + 1   // rank of x within the subtree rooted at x
2  if i == r
3       return x
4  elseif i < r
5       return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

before x in an inorder tree walk of the subtree rooted at x. Thus, $x.left.size + 1$ is the rank of x within the subtree rooted at x. If $i = r$, then node x is the ith smallest element, and so line 3 returns x. If $i < r$, then the ith smallest element resides in x's left subtree, and therefore, line 5 recurses on $x.left$. If $i > r$, then the ith 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 ith smallest element in the subtree rooted at x is the $(i - r)$th smallest element in the subtree rooted at $x.right$. Line 6 determines this element recursively.

As an example of how OS-SELECT operates, consider a search for the 17th smallest element in the order-statistic tree of Figure 17.1. The search starts 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, the node with rank 17 is the $17 - 13 = 4$th smallest element in 26's right subtree. In 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. Therefore, the node with rank 4 is the 4th smallest element in 41's left subtree. In the recursive call, x is the node with key 30, and its rank within its subtree is 2. The procedure recurses once again to find the $4 - 2 = 2$nd smallest element in the subtree rooted at the node with key 38. 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 on the facing page 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          // rank of x within the subtree rooted at x
2  y = x                        // root of subtree being examined
3  while y ≠ T.root
4      if y == y.p.right                  // if root of a right subtree ...
5          r = r + y.p.left.size + 1      // ... add in parent and its left subtree
6      y = y.p                            // move y toward the root
7  return r
```

The OS-RANK procedure works as follows. You 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, line 6 sets $y = y.p$. 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, consider the subtree rooted at $y.p$. The value of r already includes the number of nodes in the subtree rooted at node y that precede x in an inorder walk, and so the procedure 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 OS-RANK leaves 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. In this case, line 5 adds $y.p.left.size + 1$ to the current value of r.

Termination: Because each iteration of the loop moves y toward the root and the loop terminates when $y = T.root$, the loop eventually terminates. Moreover, 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 OS-RANK runs on the order-statistic tree of Figure 17.1 to find the rank of the node with key 38, the following sequence of values of $y.key$ and r occurs 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 if the basic modifying operations on red-black trees cannot efficiently maintain the *size* attribute, our work will have been for naught. Let's see how to maintain subtree sizes for both insertion and deletion without affecting the asymptotic running time of either operation.

Recall from 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, 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) on page 336, add the following lines:

```
13  y.size = x.size
14  x.size = x.left.size + x.right.size + 1
```

Figure 17.2 illustrates how the attributes are updated. The change to RIGHT-ROTATE is symmetric.

Since inserting into a red-black tree requires at most two rotations, updating the *size* attributes in the second phase costs only $O(1)$ additional time. 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.

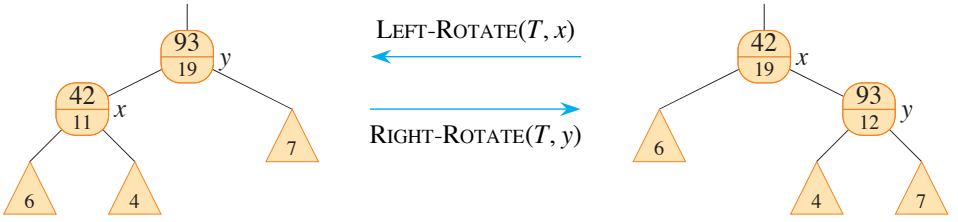

Figure 17.2 Updating subtree sizes during rotations. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

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 removes one node z from the tree and could move at most two other nodes within the tree (nodes y and x in Figure 12.4 on page 323). To update the subtree sizes, simply traverse a simple path from the lowest node that moves (starting from its original position within the tree) up to the root, decrementing the *size* 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 $O(\lg n)$. For the $O(1)$ rotations in the second phase of deletion, handle them 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

17.1-1

Show how OS-SELECT$(T.root, 10)$ operates on the red-black tree T shown in Figure 17.1.

17.1-2

Show how OS-RANK(T, x) operates on the red-black tree T shown in Figure 17.1 and the node x with $x.key = 35$.

17.1-3

Write a nonrecursive version of OS-SELECT.

17.1-4

Write a procedure OS-KEY-RANK(T, k) that takes 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.

17.1-5
Given an element x in an n-node order-statistic tree and a natural number i, show how to determine the ith successor of x in the linear order of the tree in $O(\lg n)$ time.

17.1-6
The procedures OS-SELECT and OS-RANK use the *size* attribute of a node only to compute a rank. Suppose that you store in each node its rank in the subtree of which it is the root instead of the *size* attribute. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

17.1-7
Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4 on page 47) in an array of n distinct elements in $O(n \lg n)$ time.

★ ***17.1-8***
Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

17.2 How to augment a data structure

The process of augmenting a basic data structure to support additional functionality occurs quite frequently in algorithm design. We'll use it again in the next section to design a data structure that supports operations on intervals. This section examines the steps involved in such augmentation. It includes a useful theorem that allows you to augment red-black trees easily in many cases.

You can break the process of augmenting a data structure into four steps:

1. Choose an underlying data structure.
2. Determine additional information to maintain in the underlying data structure.
3. Verify that you can maintain the additional information for the basic modifying operations on the underlying data structure.
4. Develop new operations.

As with any prescriptive design method, you'll rarely be able to follow the steps precisely in the order given. Most design work contains an element of trial and error, and progress on all steps usually proceeds in parallel. There is no point,

for example, in determining additional information and developing new operations (steps 2 and 4) if you cannot maintain the additional information efficiently. Nevertheless, this four-step method provides a good focus for your efforts in augmenting a data structure, and it is also a good framework for documenting an augmented data structure.

We followed these four steps in Section 17.1 to design order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. Red-black trees seemed like a good starting point because they efficiently support other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

In Step 2, we added the *size* attribute, so that each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, it is possible to implement OS-SELECT and OS-RANK using just the keys stored in the tree, but then they would not run in $O(\lg n)$ time. Sometimes, the additional information is pointer information rather than data, as in Exercise 17.2-1.

For step 3, we ensured that insertion and deletion can maintain the *size* attributes while still running in $O(\lg n)$ time. Ideally, you would like to update only a few elements of the data structure in order to maintain the additional information. For example, if each node simply stores its rank in the tree, the OS-SELECT and OS-RANK procedures run quickly, but inserting a new minimum element might cause a change to this information in every node of the tree. Because we chose to store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

In Step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why anyone bothers to augment a data structure in the first place. Occasionally, rather than developing new operations, you can use the additional information to expedite existing ones, as in Exercise 17.2-1.

Augmenting red-black trees

When red-black trees underlie an augmented data structure, we can prove that insertion and deletion can always efficiently maintain certain kinds of additional information, thereby simplifying step 3. The proof of the following theorem is similar to the argument from Section 17.1 that we can maintain the *size* attribute for order-statistic trees.

Theorem 17.1 (Augmenting a red-black tree)
Let f be an attribute that augments a red-black tree T of n nodes, and suppose that the value of f for each node x depends only the information in nodes x, $x.left$, and $x.right$ (possibly including $x.left.f$ and $x.right.f$), and that the value of $x.f$ can

be computed from this information in $O(1)$ time. Then, the insertion and deletion operations can maintain the values of f in all nodes of T without asymptotically affecting the $O(\lg n)$ running times of these operations.

Proof The main idea of the proof is that a change to an f attribute in a node x propagates only to ancestors of x in the tree. That is, changing $x.f$ may require $x.p.f$ to be updated, but nothing else; updating $x.p.f$ may require $x.p.p.f$ to be updated, but nothing else; and so on up the tree. After updating $T.root.f$, no other node depends on the new value, and so the process terminates. Since the height of a red-black tree is $O(\lg n)$, changing an f attribute in a node costs $O(\lg n)$ time in updating all nodes that depend on the change.

As we saw in Section 13.3, insertion of a node x into red-black tree T consists of two phases. If the tree T is empty, then the first phase simply makes x be the root of T. If T is not empty, then the first phase inserts x as a child of an existing node. Because we assume that the value of $x.f$ depends only on information in the other attributes of x itself and the information in x's children, and because x's children are both the sentinel $T.nil$, it takes only $O(1)$ time to compute the value of $x.f$. Having computed $x.f$, the change propagates up the tree. Thus, the total time for the first phase of insertion is $O(\lg n)$. During the second phase, the only structural changes to the tree come from rotations. Since only two nodes change in a rotation, but a change to an attribute might need to propagate up to the root, the total time for updating the f attributes is $O(\lg n)$ per rotation. Since the number of rotations during insertion is at most two, the total time for insertion is $O(\lg n)$.

Like insertion, deletion has two phases, as Section 13.4 discusses. In the first phase, changes to the tree occur when a node is deleted, and at most two other nodes could move within the tree. Propagating the updates to f caused by these changes costs at most $O(\lg n)$, since the changes modify the tree locally along a simple path from the lowest changed node to the root. Fixing up the red-black tree during the second phase requires at most three rotations, and each rotation requires at most $O(\lg n)$ time to propagate the updates to f. Thus, like insertion, the total time for deletion is $O(\lg n)$. ■

In many cases, such as maintaining the *size* attributes in order-statistic trees, the cost of updating after a rotation is $O(1)$, rather than the $O(\lg n)$ derived in the proof of Theorem 17.1. Exercise 17.2-3 gives an example.

On the other hand, when an update after a rotation requires a traversal all the way up to the root, it is important that insertion into and deletion from a red-black tree require a constant number of rotations. The chapter notes for Chapter 13 list other schemes for balancing search trees that do not bound the number of rotations per insertion or deletion by a constant. If each operation might require $\Theta(\lg n)$ rota-

tions and each rotation traverses a path up to the root, then a single operation could require $\Theta(\lg^2 n)$ time, rather than the $O(\lg n)$ time bound given by Theorem 17.1.

Exercises

17.2-1

Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(1)$ worst-case time on an augmented order-statistic tree. The asymptotic performance of other operations on order-statistic trees should not be affected.

17.2-2

Can you maintain the black-heights of nodes in a red-black tree as attributes in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not. How about maintaining the depths of nodes?

17.2-3

Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that you want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$, where $x_1, x_2, \ldots, x_m$ is the inorder listing of nodes in the subtree rooted at x. Show how to update the f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the *size* attributes in order-statistic trees.

17.3 Interval trees

This section shows how to augment red-black trees to support operations on dynamic sets of intervals. In this section, we'll assume that intervals are closed. Extending the results to open and half-open intervals is conceptually straightforward. (See page 1157 for definitions of closed, open, and half-open intervals.)

Intervals are convenient for representing events that each occupy a continuous period of time. For example, you could query a database of time intervals to find out which events occurred during a given interval. The data structure in this section provides an efficient means for maintaining such an interval database.

A simple way to represent an interval $[t_1, t_2]$ is as an object i with attributes $i.low = t_1$ (the ***low endpoint***) and $i.high = t_2$ (the ***high endpoint***). We say that intervals i and i' ***overlap*** if $i \cap i' \neq \emptyset$, that is, if $i.low \leq i'.high$ and $i'.low \leq i.high$.

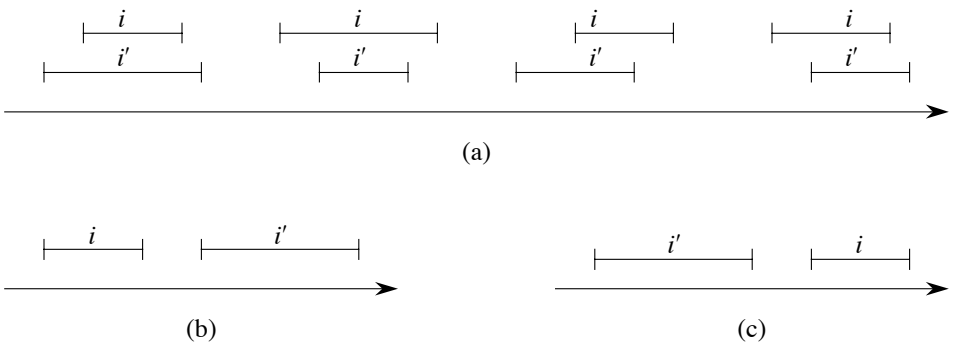

Figure 17.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations, and in each, $i.low \le i'.high$ and $i'.low \le i.high$. **(b)** The intervals do not overlap, and $i.high < i'.low$. **(c)** The intervals do not overlap, and $i'.high < i.low$.

As Figure 17.3 shows, any two intervals i and i' satisfy the ***interval trichotomy***, that is, exactly one of the following three properties holds:

a. i and i' overlap,

b. i is to the left of i' (i.e., $i.high < i'.low$),

c. i is to the right of i' (i.e., $i'.high < i.low$).

An ***interval tree*** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $x.int$. Interval trees support the following operations:

INTERVAL-INSERT(T, x) adds the element x, whose *int* attribute is assumed to contain an interval, to the interval tree T.

INTERVAL-DELETE(T, x) removes the element x from the interval tree T.

INTERVAL-SEARCH(T, i) returns a pointer to an element x in the interval tree T such that $x.int$ overlaps interval i, or a pointer to the sentinel $T.nil$ if no such element belongs to the set.

Figure 17.4 shows how an interval tree represents a set of intervals. The four-step method from Section 17.2 will guide our design of an interval tree and the operations that run on it.

Step 1: Underlying data structure

A red-black tree serves as the underlying data structure. Each node x contains an interval $x.int$. The key of x is the low endpoint, $x.int.low$, of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

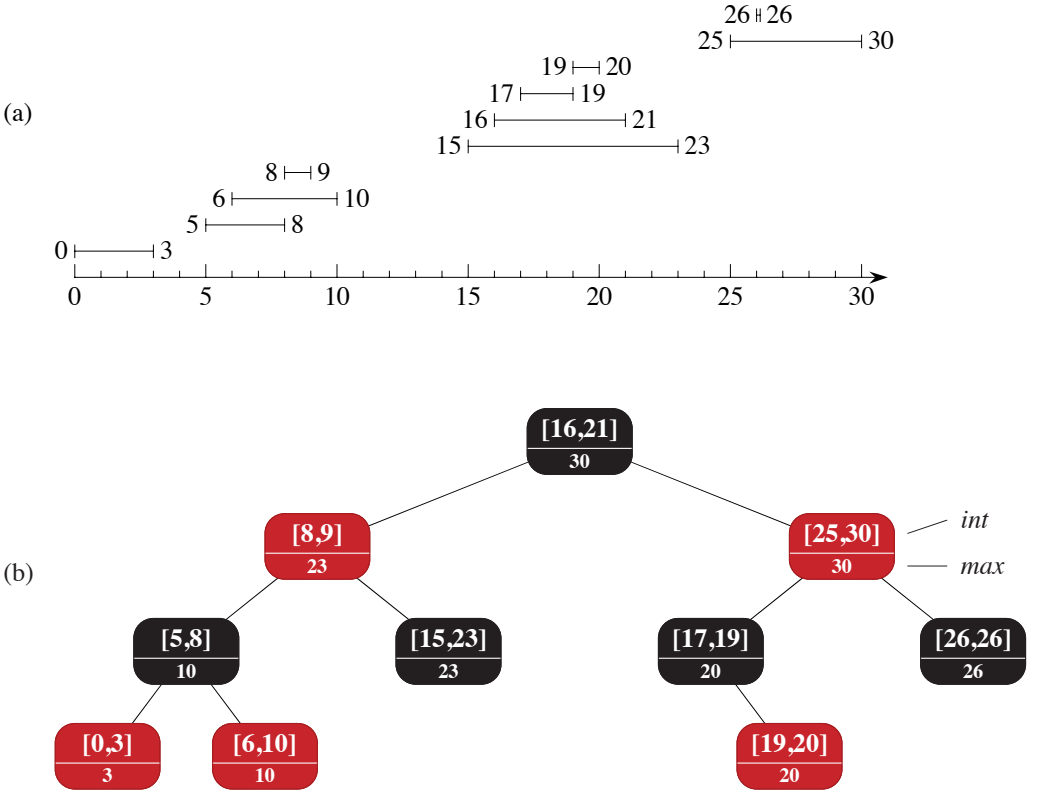

Figure 17.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. Each node x contains an interval, shown above the dashed line, and the maximum value of any interval endpoint in the subtree rooted at x, shown below the dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

Step 2: Additional information

In addition to the intervals themselves, each node x contains a value $x.max$, which is the maximum value of any interval endpoint stored in the subtree rooted at x.

Step 3: Maintaining the information

We must verify that insertion and deletion take $O(\lg n)$ time on an interval tree of n nodes. It is simple enough to determine $x.max$ in $O(1)$ time, given interval $x.int$ and the max values of node x's children:

$$x.max = \max\{x.int.high, x.left.max, x.right.max\} \ .$$

Thus, by Theorem 17.1, insertion and deletion run in $O(\lg n)$ time. In fact, you can use either Exercise 17.2-3 or 17.3-1 to show how to update all the *max* attributes that change after a rotation in just $O(1)$ time.

Step 4: Developing new operations

The only new operation is INTERVAL-SEARCH(T, i), which finds a node in tree T whose interval overlaps interval i. If there is no interval in the tree that overlaps i, the procedure returns a pointer to the sentinel $T.nil$.

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left    // overlap in left subtree or no overlap in right subtree
5      else x = x.right  // no overlap in left subtree
6  return x
```

The search for an interval that overlaps i starts at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or it reaches the sentinel $T.nil$. Since each iteration of the basic loop takes $O(1)$ time, and since the height of an n-node red-black tree is $O(\lg n)$, the INTERVAL-SEARCH procedure takes $O(\lg n)$ time.

Before we see why INTERVAL-SEARCH is correct, let's examine how it works on the interval tree in Figure 17.4. Let's look for an interval that overlaps the interval $i = [22, 25]$. Begin with x as the root, which contains $[16, 21]$ and does not overlap i. Since $x.left.max = 23$ is greater than $i.low = 22$, the loop continues with x as the left child of the root—the node containing $[8, 9]$, which also does not overlap i. This time, $x.left.max = 10$ is less than $i.low = 22$, and so the loop continues with the right child of x as the new x. Because the interval $[15, 23]$ stored in this node overlaps i, the procedure returns this node.

Now let's try an unsuccessful search, for an interval that overlaps $i = [11, 14]$ in the interval tree of Figure 17.4. Again, begin with x as the root. Since the root's interval $[16, 21]$ does not overlap i, and since $x.left.max = 23$ is greater than $i.low = 11$, go left to the node containing $[8, 9]$. Interval $[8, 9]$ does not overlap i, and $x.left.max = 10$ is less than $i.low = 11$, and so the search goes right. (No interval in the left subtree overlaps i.) Interval $[15, 23]$ does not overlap i, and its left child is $T.nil$, so again the search goes right, the loop terminates, and INTERVAL-SEARCH returns the sentinel $T.nil$.

To see why INTERVAL-SEARCH is correct, we must understand why it suffices to examine a single path from the root. The basic idea is that at any node x, if $x.int$ does not overlap i, the search always proceeds in a safe direction: the search will definitely find an overlapping interval if the tree contains one. The following theorem states this property more precisely.

Theorem 17.2
Any execution of INTERVAL-SEARCH(T, i) either returns a node whose interval overlaps i, or it returns $T.nil$ and the tree T contains no node whose interval overlaps i.

Proof The **while** loop of lines 2–5 terminates when either $x = T.nil$ or i overlaps $x.int$. In the latter case, it is certainly correct to return x. Therefore, we focus on the former case, in which the **while** loop terminates because $x = T.nil$, which is the node that INTERVAL-SEARCH returns.

We'll prove that if the procedure returns $T.nil$, then it did not miss any intervals in T that overlap i. The idea is to show that whether the search goes left in line 4 or right in line 5, it always heads toward a node containing an interval overlapping i, if any such interval exists. In particular, we'll prove that

1. If the search goes left in line 4, then the left subtree of node x contains an interval that overlaps i or the right subtree of x contains no interval that overlaps i. Therefore, even if x's left subtree contains no interval that overlaps i but the search goes left, it does not make a mistake, because x's right subtree does not contain an interval overlapping i, either.
2. If the search goes right in line 5, then the left subtree of x contains no interval that overlaps i. Thus, if the search goes right, it does not make a mistake.

For both cases, we rely on the interval trichotomy. Let's start with the case where the search goes right, whose proof is simpler. By the tests in line 3, we know that $x.left = T.nil$ or $x.left.max < i.low$. If $x.left = T.nil$, then x's left subtree contains no interval that overlaps i, since it contains no intervals at all. Now suppose that $x.left \neq T.nil$, so that we must have $x.left.max < i.low$. Consider any interval i' in x's left subtree. Because $x.left.max$ is the maximum endpoint in x's left subtree, we have $i'.high \leq x.left.max$. Thus, as Figure 17.5(a) shows,

$$\begin{aligned} i'.high &\leq x.left.max \\ &< i.low . \end{aligned}$$

By the interval trichotomy, therefore, intervals i and i' do not overlap, and so x's left subtree contains no interval that overlaps i.

Now we examine the case in which the search goes left. If the left subtree of node x contains an interval that overlaps i, we're done, so let's assume that no node

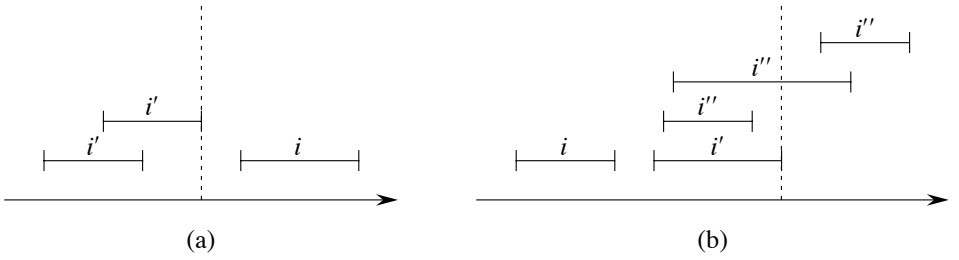

Figure 17.5 Intervals in the proof of Theorem 17.2. The value of $x.left.max$ is shown in each case as a dashed line. **(a)** The search goes right. No interval i' in x's left subtree can overlap i. **(b)** The search goes left. The left subtree of x contains an interval that overlaps i (situation not shown), or x's left subtree contains an interval i' such that $i'.high = x.left.max$. Since i does not overlap i', neither does it overlap any interval i'' in x's right subtree, since $i'.low \leq i''.low$.

in x's left subtree overlaps i. We need to show that in this case, no node in x's right subtree overlaps i, so that going left will not miss any overlaps in x's right subtree. By the tests in line 3, the left subtree of x is not empty and $x.left.max \geq i.low$. By the definition of the *max* attribute, x's left subtree contains some interval i' such that

$$\begin{aligned} i'.high &= x.left.max \\ &\geq i.low , \end{aligned}$$

as illustrated in Figure 17.5(b). Since i' is in x's left subtree, it does not overlap i, and since $i'.high \geq i.low$, the interval trichotomy tells us that $i.high < i'.low$. Now we bring in the property that interval trees are keyed on the low endpoints of intervals. Because i' is in x's left subtree, we have $i'.low \leq x.int.low$. Now consider any interval i'' in x's right subtree, so that $x.int.low \leq i''.low$. Putting inequalities together, we get

$$\begin{aligned} i.high &< i'.low \\ &\leq x.int.low \\ &\leq i''.low . \end{aligned}$$

Because $i.high < i''.low$, the interval trichotomy tells us that i and i'' do not overlap. Since we chose i'' as any interval in x's right subtree, no node in x's right subtree overlaps i. ■

Thus, the INTERVAL-SEARCH procedure works correctly.

Exercises

17.3-1

Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates all the *max* attributes that change in $O(1)$ time.

17.3-2

Describe an efficient algorithm that, given an interval i, returns an interval overlapping i that has the minimum low endpoint, or $T.nil$ if no such interval exists.

17.3-3

Given an interval tree T and an interval i, describe how to list all intervals in T that overlap i in $O(\min\{n, k \lg n\})$ time, where k is the number of intervals in the output list. (*Hint:* One simple method makes several queries, modifying the tree between queries. A slightly more complicated method does not modify the tree.)

17.3-4

Suggest modifications to the interval-tree procedures to support the new operation INTERVAL-SEARCH-EXACTLY(T, i), where T is an interval tree and i is an interval. The operation should return a pointer to a node x in T such that $x.int.low = i.low$ and $x.int.high = i.high$, or $T.nil$ if T contains no such node. All operations, including INTERVAL-SEARCH-EXACTLY, should run in $O(\lg n)$ time on an n-node interval tree.

17.3-5

Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the absolute value of the difference of the two closest numbers in Q. For example, if we have $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP$(Q)$ returns 3, since 15 and 18 are the two closest numbers in Q. Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

★ ### 17.3-6

VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x- and y-axes), so that each rectangle is represented by four values: its minimum and maximum x- and y-coordinates. Give an $O(n \lg n)$-time algorithm to decide whether a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (*Hint:* Move a "sweep" line across the set of rectangles.)

Problems

17-1 Point of maximum overlap

You wish to keep track of a ***point of maximum overlap*** in a set of intervals—a point with the largest number of intervals in the set that overlap it.

a. Show that there is always a point of maximum overlap that is an endpoint of one of the intervals.

b. Design a data structure that efficiently supports the operations INTERVAL-INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of maximum overlap. (*Hint:* Keep a red-black tree of all the endpoints. Associate a value of $+1$ with each left endpoint, and associate a value of -1 with each right endpoint. Augment each node of the tree with some extra information to maintain the point of maximum overlap.)

17-2 Josephus permutation

We define the ***Josephus problem*** as follows. A group of n people form a circle, and we are given a positive integer $m \leq n$. Beginning with a designated first person, proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until nobody remains in the circle. The order in which the people are removed from the circle defines the ***(n, m)-Josephus permutation*** of the integers $1, 2, \ldots, n$. For example, the $(7, 3)$-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Suppose that m is a constant. Describe an $O(n)$-time algorithm that, given an integer n, outputs the (n, m)-Josephus permutation.

b. Suppose that m is not necessarily a constant. Describe an $O(n \lg n)$-time algorithm that, given integers n and m, outputs the (n, m)-Josephus permutation.

Chapter notes

In their book, Preparata and Shamos [364] describe several of the interval trees that appear in the literature, citing work by H. Edelsbrunner (1980) and E. M. McCreight (1981). The book details an interval tree that, given a static database of n intervals, allows us to enumerate all k intervals that overlap a given query interval in $O(k + \lg n)$ time.