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 ith 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 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. 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.

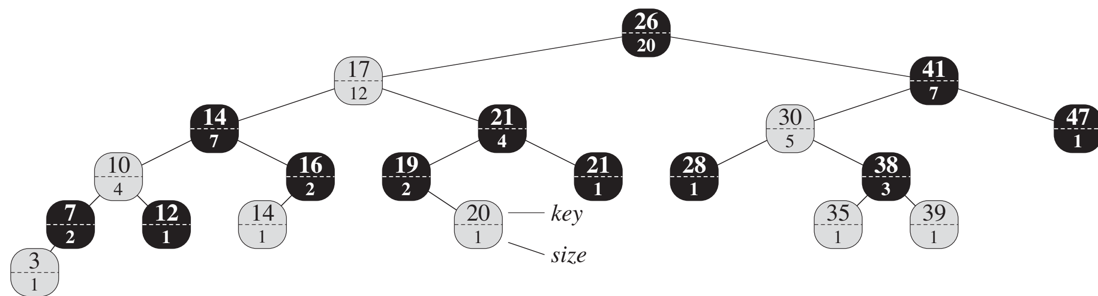

Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x.

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

```
OS-SELECT(x, i)
1  r = x.left.size + 1
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)
```

In line 1 of OS-SELECT, we compute 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 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 we return x in line 3. If $i < r$, then the ith smallest element resides in x's left subtree, and so we recurse on $x.left$ in line 5. 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.

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 ≠ 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*

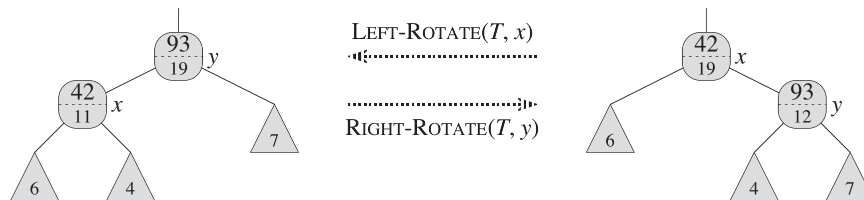

Figure 14.2 Updating subtree sizes during rotations. The link around which we rotate is incident on the two nodes whose *size* attributes need to be updated. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

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 OS-SELECT$(T.root, 10)$ operates on the red-black tree T of Figure 14.1.

14.1-2

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

14.1-3

Write a nonrecursive version of OS-SELECT.

14.1-4

Write a recursive procedure 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 ith successor of x in the linear order of the tree in $O(\lg n)$ time?

14.1-6
Observe that whenever we reference the *size* attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

14.1-7
Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4) in an array of size n in time $O(n \lg n)$.

14.1-8 ⋆
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 correct answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

14.2 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 shall use it again in the next section to design a data structure that supports operations on intervals. In this section, we examine the steps involved in such augmentation. We shall also prove a theorem that allows us to augment red-black trees easily in many cases.

We 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 we 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 should not blindly follow the steps 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 we will not be able to 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 way to organize the documentation of an augmented data structure.

We followed these steps in Section 14.1 to design our order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. A clue to the suitability of red-black trees comes from their efficient support of other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

For step 2, we added the *size* attribute, in which each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, we could have implemented OS-SELECT and OS-RANK using just the keys stored in the tree, but they would not have run in $O(\lg n)$ time. Sometimes, the additional information is pointer information rather than data, as in Exercise 14.2-1.

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

For step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why we bother to augment a data structure in the first place. Occasionally, rather than developing new operations, we use the additional information to expedite existing ones, as in Exercise 14.2-1.

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 making step 3 very easy. The proof of the following theorem is similar to the argument from Section 14.1 that we can maintain the *size* attribute for order-statistic trees.

Theorem 14.1 (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 on only the information in nodes x, $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$. Then, we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance 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 re-

quire $x.p.f$ 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. Once we have updated $T.root.f$, no other node will depend 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.

Insertion of a node x into T consists of two phases. (See Section 13.3.) The first phase inserts x as a child of an existing node $x.p$. We can compute the value of $x.f$ in $O(1)$ time since, by supposition, it depends only on information in the other attributes of x itself and the information in x's children, but x's children are both the sentinel $T.nil$. Once we have 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, 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. (See Section 13.4.) In the first phase, changes to the tree occur when the deleted node is removed from the tree. If the deleted node had two children at the time, then its successor moves into the position of the deleted node. Propagating the updates to f caused by these changes costs at most $O(\lg n)$, since the changes modify the tree locally. 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 14.1. Exercise 14.2-3 gives an example.

Exercises

14.2-1

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.

14.2-2

Can we 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?

14.2-3 ⋆

Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that we 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.

14.2-4 ⋆

We wish to augment red-black trees with an operation RB-ENUMERATE(x, a, b) that outputs all the keys k such that $a \leq k \leq b$ in a red-black tree rooted at x. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output and n is the number of internal nodes in the tree. (*Hint:* You do not need to add new attributes to the red-black tree.)

14.3 Interval trees

In this section, we shall augment red-black trees to support operations on dynamic sets of intervals. A ***closed interval*** is an ordered pair of real numbers $[t_1, t_2]$, with $t_1 \leq t_2$. The interval $[t_1, t_2]$ represents the set $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$. ***Open*** and ***half-open*** intervals omit both or one of the endpoints from the set, respectively. In this section, we shall assume that intervals are closed; extending the results to open and half-open intervals is conceptually straightforward.

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

We can represent an interval $[t_1, t_2]$ 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 $i'.low \leq i.high$. As Figure 14.3 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:

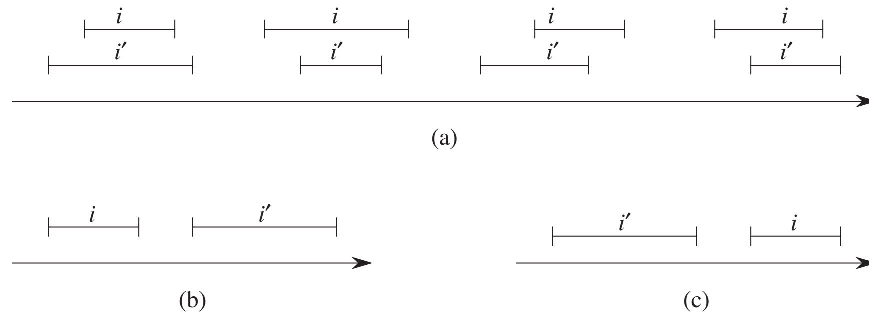

Figure 14.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations; in each, $i.low \le i'.high$ and $i'.low \le i.high$. **(b)** The intervals do not overlap, and $i.high < i'.low$. **(c)** The intervals do not overlap, and $i'.high < i.low$.

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 is in the set.

Figure 14.4 shows how an interval tree represents a set of intervals. We shall track the four-step method from Section 14.2 as we review the design of an interval tree and the operations that run on it.

Step 1: Underlying data structure

We choose a red-black tree in which each node x contains an interval $x.int$ and 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.

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. We can determine $x.max$ given interval $x.int$ and the *max* values of node x's children:

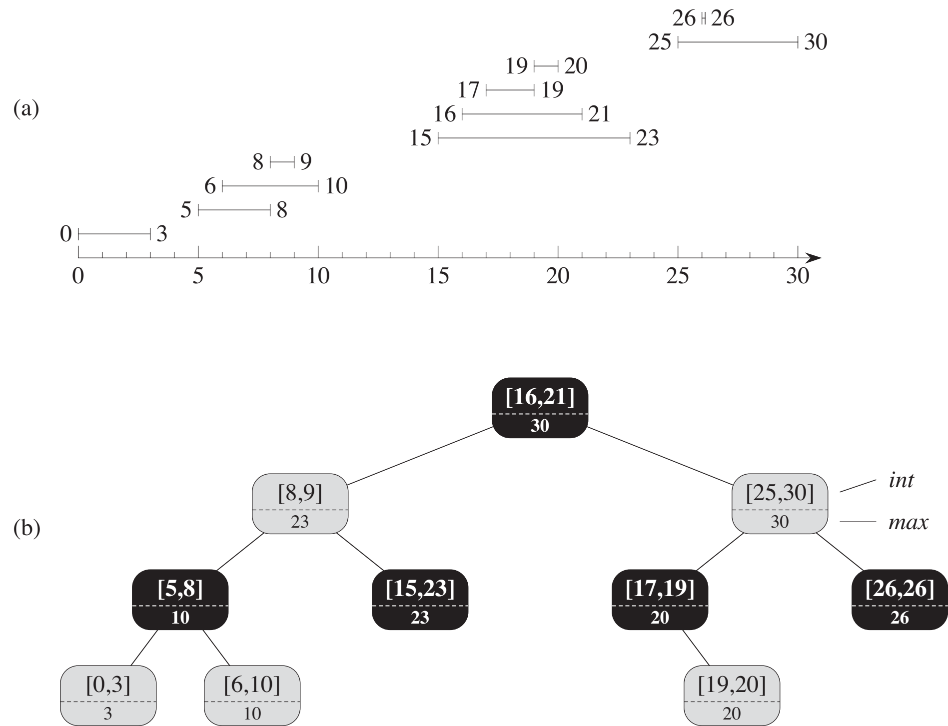

Figure 14.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. Each node x contains an interval, shown above the dashed line, and the maximum value of any interval endpoint in the subtree rooted at x, shown below the dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

$$x.max = \max(x.int.high, x.left.max, x.right.max) .$$

Thus, by Theorem 14.1, insertion and deletion run in $O(\lg n)$ time. In fact, we can update the *max* attributes after a rotation in $O(1)$ time, as Exercises 14.2-3 and 14.3-1 show.

Step 4: Developing new operations

The only new operation we need is INTERVAL-SEARCH(T, i), which finds a node in tree T whose interval overlaps interval i. If there is no interval that overlaps i in the tree, 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
5      else x = x.right
6  return x
```

The search for an interval that overlaps i starts with x at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or x points to 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 14.4. Suppose we wish to find an interval that overlaps the interval $i = [22, 25]$. We 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.

As an example of an unsuccessful search, suppose we wish to find an interval that overlaps $i = [11, 14]$ in the interval tree of Figure 14.4. We once 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 $i.low = 11$, we 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 we go right. (Note that no interval in the left subtree overlaps i.) Interval $[15, 23]$ does not overlap i, and its left child is $T.nil$, so again we go right, the loop terminates, and we return 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 14.2

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.

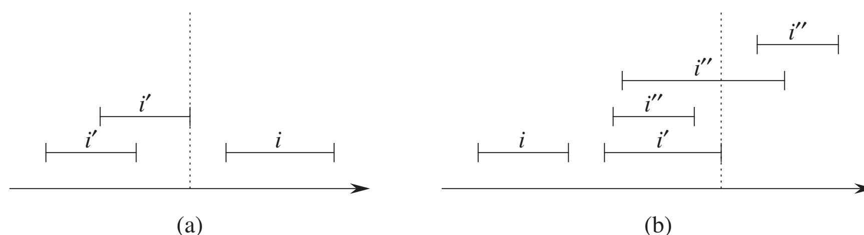

Figure 14.5 Intervals in the proof of Theorem 14.2. The value of $x.left.max$ is shown in each case as a dashed line. **(a)** The search goes right. No interval i' in x's left subtree can overlap i. **(b)** The search goes left. The left subtree of x contains an interval that overlaps i (situation not 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 $i'.low \leq i''.low$.

Proof The **while** loop of lines 2–5 terminates either when $x = T.nil$ 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 $x = T.nil$.

We use the following invariant for the **while** loop of lines 2–5:

> If tree T contains an interval that overlaps i, then the subtree rooted at x contains such an interval.

We use this loop invariant as follows:

Initialization: Prior to the first iteration, line 1 sets x to be the root of T, so that the invariant holds.

Maintenance: Each iteration of the **while** loop executes either line 4 or line 5. We shall show that both cases maintain the loop invariant.

If line 5 is executed, then because of the branch condition in line 3, we have $x.left = T.nil$, or $x.left.max < i.low$. If $x.left = T.nil$, the subtree rooted at $x.left$ clearly contains no interval that overlaps i, and so setting x to $x.right$ maintains the invariant. Suppose, therefore, that $x.left \neq T.nil$ and $x.left.max < i.low$. As Figure 14.5(a) shows, for each interval i' in x's left subtree, we have

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

By the interval trichotomy, therefore, i' and i do not overlap. Thus, the left subtree of x contains no intervals that overlap i, so that setting x to $x.right$ maintains the invariant.

If, on the other hand, line 4 is executed, then we will show that the contrapositive of the loop invariant holds. That is, if the subtree rooted at $x.left$ contains no interval overlapping i, then no interval anywhere in the tree overlaps i. Since line 4 is executed, then because of the branch condition in line 3, we have $x.left.max \geq i.low$. Moreover, by definition of the *max* attribute, x's left subtree must contain some interval i' such that

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

(Figure 14.5(b) illustrates the situation.) Since i and i' do not overlap, and since it is not true that $i'.high < i.low$, it follows by the interval trichotomy that $i.high < i'.low$. Interval trees are keyed on the low endpoints of intervals, and thus the search-tree property implies that for any interval i'' in x's right subtree,

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

By the interval trichotomy, i and i'' do not overlap. We conclude that whether or not any interval in x's left subtree overlaps i, setting x to $x.left$ maintains the invariant.

Termination: If the loop terminates when $x = T.nil$, then the subtree rooted at x contains no interval overlapping i. The contrapositive of the loop invariant implies that T contains no interval that overlaps i. Hence it is correct to return $x = T.nil$. ■

Thus, the INTERVAL-SEARCH procedure works correctly.

Exercises

14.3-1

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

14.3-2

Rewrite the code for INTERVAL-SEARCH so that it works properly when all intervals are open.

14.3-3

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.

14.3-4

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.)

14.3-5

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.

14.3-6

Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the magnitude of the difference of the two closest numbers in Q. For example, if $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP$(Q)$ returns $18 - 15 = 3$, 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.

14.3-7 ⋆

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 we represent a rectangle by its minimum and maximum x- and y-coordinates. Give an $O(n \lg n)$-time algorithm to decide whether or not 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

14-1 Point of maximum overlap

Suppose that we 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 will always be a point of maximum overlap that is an endpoint of one of the segments.

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.)

14-2 Josephus permutation

We define the ***Josephus problem*** as follows. Suppose that n people form a circle and that we are given a positive integer $m \le n$. Beginning with a designated first person, we proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until we have removed all n people. 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 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 [282] 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.