calculations. The array *summary* contains the summary bits stored recursively in a proto-vEB structure, and the array *cluster* contains $\sqrt{u}$ pointers.

Figure 20.4 shows a fully expanded *proto-vEB*(16) structure representing the set $\{2, 3, 4, 5, 7, 14, 15\}$. If the value i is in the proto-vEB structure pointed to by *summary*, then the ith cluster contains some value in the set being represented. As in the tree of constant height, *cluster*$[i]$ represents the values $i\sqrt{u}$ through $(i+1)\sqrt{u}-1$, which form the ith cluster.

At the base level, the elements of the actual dynamic sets are stored in some of the *proto-vEB*(2) structures, and the remaining *proto-vEB*(2) structures store summary bits. Beneath each of the non-summary base structures, the figure indicates which bits it stores. For example, the *proto-vEB*(2) structure labeled "elements 6,7" stores bit 6 (0, since element 6 is not in the set) in its $A[0]$ and bit 7 (1, since element 7 is in the set) in its $A[1]$.

Like the clusters, each summary is just a dynamic set with universe size $\sqrt{u}$, and so we represent each summary as a *proto-vEB*$(\sqrt{u})$ structure. The four summary bits for the main *proto-vEB*(16) structure are in the leftmost *proto-vEB*(4) structure, and they ultimately appear in two *proto-vEB*(2) structures. For example, the *proto-vEB*(2) structure labeled "clusters 2,3" has $A[0] = 0$, indicating that cluster 2 of the *proto-vEB*(16) structure (containing elements $8, 9, 10, 11$) is all 0, and $A[1] = 1$, telling us that cluster 3 (containing elements $12, 13, 14, 15$) has at least one 1. Each *proto-vEB*(4) structure points to its own summary, which is itself stored as a *proto-vEB*(2) structure. For example, look at the *proto-vEB*(2) structure just to the left of the one labeled "elements 0,1." Because its $A[0]$ is 0, it tells us that the "elements 0,1" structure is all 0, and because its $A[1]$ is 1, we know that the "elements 2,3" structure contains at least one 1.

20.2.2 Operations on a proto van Emde Boas structure

We shall now describe how to perform operations on a proto-vEB structure. We first examine the query operations—MEMBER, MINIMUM, MAXIMUM, and SUCCESSOR—which do not change the proto-vEB structure. We then discuss INSERT and DELETE. We leave MAXIMUM and PREDECESSOR, which are symmetric to MINIMUM and SUCCESSOR, respectively, as Exercise 20.2-1.

Each of the MEMBER, SUCCESSOR, PREDECESSOR, INSERT, and DELETE operations takes a parameter x, along with a proto-vEB structure V. Each of these operations assumes that $0 \le x < V.u$.

Determining whether a value is in the set

To perform MEMBER(x), we need to find the bit corresponding to x within the appropriate *proto-vEB*(2) structure. We can do so in $O(\lg\lg u)$ time, bypassing the *summary* structures altogether. The following procedure takes a *proto-vEB* structure V and a value x, and it returns a bit indicating whether x is in the dynamic set held by V.

```
PROTO-vEB-MEMBER(V, x)
1  if V.u == 2
2      return V.A[x]
3  else return PROTO-vEB-MEMBER(V.cluster[high(x)], low(x))
```

The PROTO-vEB-MEMBER procedure works as follows. Line 1 tests whether we are in a base case, where V is a *proto-vEB*(2) structure. Line 2 handles the base case, simply returning the appropriate bit of array A. Line 3 deals with the recursive case, "drilling down" into the appropriate smaller proto-vEB structure. The value $\text{high}(x)$ says which *proto-vEB*($\sqrt{u}$) structure we visit, and $\text{low}(x)$ determines which element within that *proto-vEB*($\sqrt{u}$) structure we are querying.

Let's see what happens when we call PROTO-vEB-MEMBER$(V, 6)$ on the *proto-vEB*(16) structure in Figure 20.4. Since $\text{high}(6) = 1$ when $u = 16$, we recurse into the *proto-vEB*(4) structure in the upper right, and we ask about element $\text{low}(6) = 2$ of that structure. In this recursive call, $u = 4$, and so we recurse again. With $u = 4$, we have $\text{high}(2) = 1$ and $\text{low}(2) = 0$, and so we ask about element 0 of the *proto-vEB*(2) structure in the upper right. This recursive call turns out to be a base case, and so it returns $A[0] = 0$ back up through the chain of recursive calls. Thus, we get the result that PROTO-vEB-MEMBER$(V, 6)$ returns 0, indicating that 6 is not in the set.

To determine the running time of PROTO-vEB-MEMBER, let $T(u)$ denote its running time on a *proto-vEB*(u) structure. Each recursive call takes constant time, not including the time taken by the recursive calls that it makes. When PROTO-vEB-MEMBER makes a recursive call, it makes a call on a *proto-vEB*($\sqrt{u}$) structure. Thus, we can characterize the running time by the recurrence $T(u) = T(\sqrt{u}) + O(1)$, which we have already seen as recurrence (20.2). Its solution is $T(u) = O(\lg \lg u)$, and so we conclude that PROTO-vEB-MEMBER runs in time $O(\lg \lg u)$.

Finding the minimum element

Now we examine how to perform the MINIMUM operation. The procedure PROTO-vEB-MINIMUM(V) returns the minimum element in the proto-vEB structure V, or NIL if V represents an empty set.

```
PROTO-vEB-MINIMUM(V)
 1  if V.u == 2
 2      if V.A[0] == 1
 3          return 0
 4      elseif V.A[1] == 1
 5          return 1
 6      else return NIL
 7  else min-cluster = PROTO-vEB-MINIMUM(V.summary)
 8      if min-cluster == NIL
 9          return NIL
10      else offset = PROTO-vEB-MINIMUM(V.cluster[min-cluster])
11          return index(min-cluster, offset)
```

This procedure works as follows. Line 1 tests for the base case, which lines 2–6 handle by brute force. Lines 7–11 handle the recursive case. First, line 7 finds the number of the first cluster that contains an element of the set. It does so by recursively calling PROTO-vEB-MINIMUM on *V.summary*, which is a *proto-vEB*$(\sqrt{u})$ structure. Line 7 assigns this cluster number to the variable *min-cluster*. If the set is empty, then the recursive call returned NIL, and line 9 returns NIL. Otherwise, the minimum element of the set is somewhere in cluster number *min-cluster*. The recursive call in line 10 finds the offset within the cluster of the minimum element in this cluster. Finally, line 11 constructs the value of the minimum element from the cluster number and offset, and it returns this value.

Although querying the summary information allows us to quickly find the cluster containing the minimum element, because this procedure makes two recursive calls on *proto-vEB*$(\sqrt{u})$ structures, it does not run in $O(\lg\lg u)$ time in the worst case. Letting $T(u)$ denote the worst-case time for PROTO-vEB-MINIMUM on a *proto-vEB*(u) structure, we have the recurrence

$$T(u) = 2T(\sqrt{u}) + O(1) . \tag{20.3}$$

Again, we use a change of variables to solve this recurrence, letting $m = \lg u$, which gives

$$T(2^m) = 2T(2^{m/2}) + O(1) .$$

Renaming $S(m) = T(2^m)$ gives

$$S(m) = 2S(m/2) + O(1) ,$$

which, by case 1 of the master method, has the solution $S(m) = \Theta(m)$. By changing back from $S(m)$ to $T(u)$, we have that $T(u) = T(2^m) = S(m) = \Theta(m) = \Theta(\lg u)$. Thus, we see that because of the second recursive call, PROTO-vEB-MINIMUM runs in $\Theta(\lg u)$ time rather than the desired $O(\lg\lg u)$ time.

Finding the successor

The SUCCESSOR operation is even worse. In the worst case, it makes two recursive calls, along with a call to PROTO-vEB-MINIMUM. The procedure PROTO-vEB-SUCCESSOR(V, x) returns the smallest element in the proto-vEB structure V that is greater than x, or NIL if no element in V is greater than x. It does not require x to be a member of the set, but it does assume that $0 \le x < V.u$.

```
PROTO-vEB-SUCCESSOR(V, x)
 1  if $V.u == 2$
 2      if $x == 0$ and $V.A[1] == 1$
 3          return 1
 4      else return NIL
 5  else offset = PROTO-vEB-SUCCESSOR($V.cluster[\text{high}(x)], \text{low}(x)$)
 6      if offset ≠ NIL
 7          return index(high($x$), offset)
 8      else succ-cluster = PROTO-vEB-SUCCESSOR($V.summary, \text{high}(x)$)
 9          if succ-cluster == NIL
10              return NIL
11          else offset = PROTO-vEB-MINIMUM($V.cluster[succ\text{-}cluster]$)
12              return index(succ-cluster, offset)
```

The PROTO-vEB-SUCCESSOR procedure works as follows. As usual, line 1 tests for the base case, which lines 2–4 handle by brute force: the only way that x can have a successor within a *proto-vEB*(2) structure is when $x = 0$ and $A[1]$ is 1. Lines 5–12 handle the recursive case. Line 5 searches for a successor to x within x's cluster, assigning the result to *offset*. Line 6 determines whether x has a successor within its cluster; if it does, then line 7 computes and returns the value of this successor. Otherwise, we have to search in other clusters. Line 8 assigns to *succ-cluster* the number of the next nonempty cluster, using the summary information to find it. Line 9 tests whether *succ-cluster* is NIL, with line 10 returning NIL if all succeeding clusters are empty. If *succ-cluster* is non-NIL, line 11 assigns the first element within that cluster to *offset*, and line 12 computes and returns the minimum element in that cluster.

In the worst case, PROTO-vEB-SUCCESSOR calls itself recursively twice on *proto-vEB*$(\sqrt{u})$ structures, and it makes one call to PROTO-vEB-MINIMUM on a *proto-vEB*$(\sqrt{u})$ structure. Thus, the recurrence for the worst-case running time $T(u)$ of PROTO-vEB-SUCCESSOR is

$$\begin{aligned} T(u) &= 2T(\sqrt{u}) + \Theta(\lg \sqrt{u}) \\ &= 2T(\sqrt{u}) + \Theta(\lg u) \;. \end{aligned}$$

We can employ the same technique that we used for recurrence (20.1) to show that this recurrence has the solution $T(u) = \Theta(\lg u \lg\lg u)$. Thus, PROTO-vEB-SUCCESSOR is asymptotically slower than PROTO-vEB-MINIMUM.

Inserting an element

To insert an element, we need to insert it into the appropriate cluster and also set the summary bit for that cluster to 1. The procedure PROTO-vEB-INSERT(V, x) inserts the value x into the proto-vEB structure V.

```
PROTO-vEB-INSERT(V, x)
1  if $V.u == 2$
2      $V.A[x] = 1$
3  else PROTO-vEB-INSERT($V.cluster[\text{high}(x)], \text{low}(x)$)
4      PROTO-vEB-INSERT($V.summary, \text{high}(x)$)
```

In the base case, line 2 sets the appropriate bit in the array A to 1. In the recursive case, the recursive call in line 3 inserts x into the appropriate cluster, and line 4 sets the summary bit for that cluster to 1.

Because PROTO-vEB-INSERT makes two recursive calls in the worst case, recurrence (20.3) characterizes its running time. Hence, PROTO-vEB-INSERT runs in $\Theta(\lg u)$ time.

Deleting an element

The DELETE operation is more complicated than insertion. Whereas we can always set a summary bit to 1 when inserting, we cannot always reset the same summary bit to 0 when deleting. We need to determine whether any bit in the appropriate cluster is 1. As we have defined proto-vEB structures, we would have to examine all $\sqrt{u}$ bits within a cluster to determine whether any of them are 1. Alternatively, we could add an attribute n to the proto-vEB structure, counting how many elements it has. We leave implementation of PROTO-vEB-DELETE as Exercises 20.2-2 and 20.2-3.

Clearly, we need to modify the proto-vEB structure to get each operation down to making at most one recursive call. We will see in the next section how to do so.

Exercises

20.2-1
Write pseudocode for the procedures PROTO-vEB-MAXIMUM and PROTO-vEB-PREDECESSOR.

20.2-2
Write pseudocode for PROTO-VEB-DELETE. It should update the appropriate summary bit by scanning the related bits within the cluster. What is the worst-case running time of your procedure?

20.2-3
Add the attribute n to each proto-vEB structure, giving the number of elements currently in the set it represents, and write pseudocode for PROTO-VEB-DELETE that uses the attribute n to decide when to reset summary bits to 0. What is the worst-case running time of your procedure? What other procedures need to change because of the new attribute? Do these changes affect their running times?

20.2-4
Modify the proto-vEB structure to support duplicate keys.

20.2-5
Modify the proto-vEB structure to support keys that have associated satellite data.

20.2-6
Write pseudocode for a procedure that creates a *proto-vEB*(u) structure.

20.2-7
Argue that if line 9 of PROTO-VEB-MINIMUM is executed, then the proto-vEB structure is empty.

20.2-8
Suppose that we designed a proto-vEB structure in which each *cluster* array had only $u^{1/4}$ elements. What would the running times of each operation be?

20.3 The van Emde Boas tree

The proto-vEB structure of the previous section is close to what we need to achieve $O(\lg \lg u)$ running times. It falls short because we have to recurse too many times in most of the operations. In this section, we shall design a data structure that is similar to the proto-vEB structure but stores a little more information, thereby removing the need for some of the recursion.

In Section 20.2, we observed that the assumption that we made about the universe size—that $u = 2^{2^k}$ for some integer k—is unduly restrictive, confining the possible values of u an overly sparse set. From this point on, therefore, we will allow the universe size u to be any exact power of 2, and when $\sqrt{u}$ is not an inte-

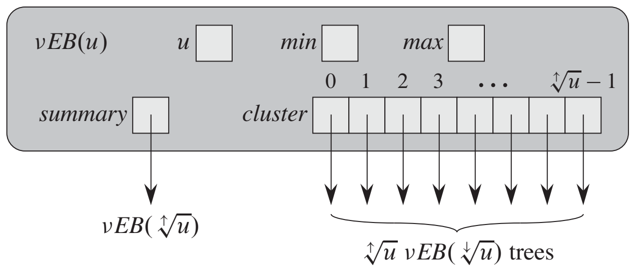

$vEB(u)$
u
min
max
0 1 2 3 … $\sqrt[\uparrow]{u}-1$
$summary$
$cluster$
$vEB(\sqrt[\uparrow]{u})$
$\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$ trees

Figure 20.5 The information in a $vEB(u)$ tree when $u > 2$. The structure contains the universe size u, elements *min* and *max*, a pointer *summary* to a $vEB(\sqrt[\uparrow]{u})$ tree, and an array $cluster[0\,.\,.\,\sqrt[\uparrow]{u}-1]$ of $\sqrt[\uparrow]{u}$ pointers to $vEB(\sqrt[\downarrow]{u})$ trees.

ger—that is, if u is an odd power of 2 ($u = 2^{2k+1}$ for some integer $k \geq 0$)—then we will divide the $\lg u$ bits of a number into the most significant $\lceil (\lg u)/2 \rceil$ bits and the least significant $\lfloor (\lg u)/2 \rfloor$ bits. For convenience, we denote $2^{\lceil (\lg u)/2 \rceil}$ (the "upper square root" of u) by $\sqrt[\uparrow]{u}$ and $2^{\lfloor (\lg u)/2 \rfloor}$ (the "lower square root" of u) by $\sqrt[\downarrow]{u}$, so that $u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$ and, when u is an even power of 2 ($u = 2^{2k}$ for some integer k), $\sqrt[\uparrow]{u} = \sqrt[\downarrow]{u} = \sqrt{u}$. Because we now allow u to be an odd power of 2, we must redefine our helpful functions from Section 20.2:

$$\begin{aligned}
\text{high}(x) &= \left\lfloor x / \sqrt[\downarrow]{u} \right\rfloor , \\
\text{low}(x) &= x \bmod \sqrt[\downarrow]{u} , \\
\text{index}(x, y) &= x \sqrt[\downarrow]{u} + y .
\end{aligned}$$

20.3.1 van Emde Boas trees

The ***van Emde Boas tree***, or ***vEB tree***, modifies the proto-vEB structure. We denote a vEB tree with a universe size of u as $vEB(u)$ and, unless u equals the base size of 2, the attribute *summary* points to a $vEB(\sqrt[\uparrow]{u})$ tree and the array $cluster[0\,.\,.\,\sqrt[\uparrow]{u}-1]$ points to $\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$ trees. As Figure 20.5 illustrates, a vEB tree contains two attributes not found in a proto-vEB structure:

- *min* stores the minimum element in the vEB tree, and
- *max* stores the maximum element in the vEB tree.

Furthermore, the element stored in *min* does not appear in any of the recursive $vEB(\sqrt[\downarrow]{u})$ trees that the *cluster* array points to. The elements stored in a $vEB(u)$ tree V, therefore, are $V.min$ plus all the elements recursively stored in the $vEB(\sqrt[\downarrow]{u})$ trees pointed to by $V.cluster[0\,.\,.\,\sqrt[\uparrow]{u}-1]$. Note that when a vEB tree contains two or more elements, we treat *min* and *max* differently: the element stored in *min* does not appear in any of the clusters, but the element stored in *max* does.

Since the base size is 2, a $vEB(2)$ tree does not need the array A that the corresponding *proto*-$vEB(2)$ structure has. Instead, we can determine its elements from its *min* and *max* attributes. In a vEB tree with no elements, regardless of its universe size u, both *min* and *max* are NIL.

Figure 20.6 shows a $vEB(16)$ tree V holding the set $\{2, 3, 4, 5, 7, 14, 15\}$. Because the smallest element is 2, $V.min$ equals 2, and even though $\text{high}(2) = 0$, the element 2 does not appear in the $vEB(4)$ tree pointed to by $V.cluster[0]$: notice that $V.cluster[0].min$ equals 3, and so 2 is not in this vEB tree. Similarly, since $V.cluster[0].min$ equals 3, and 2 and 3 are the only elements in $V.cluster[0]$, the $vEB(2)$ clusters within $V.cluster[0]$ are empty.

The *min* and *max* attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:

1. The MINIMUM and MAXIMUM operations do not even need to recurse, for they can just return the values of *min* or *max*.
2. The SUCCESSOR operation can avoid making a recursive call to determine whether the successor of a value x lies within $\text{high}(x)$. That is because x's successor lies within its cluster if and only if x is strictly less than the *max* attribute of its cluster. A symmetric argument holds for PREDECESSOR and *min*.
3. We can tell whether a vEB tree has no elements, exactly one element, or at least two elements in constant time from its *min* and *max* values. This ability will help in the INSERT and DELETE operations. If *min* and *max* are both NIL, then the vEB tree has no elements. If *min* and *max* are non-NIL but are equal to each other, then the vEB tree has exactly one element. Otherwise, both *min* and *max* are non-NIL but are unequal, and the vEB tree has two or more elements.
4. If we know that a vEB tree is empty, we can insert an element into it by updating only its *min* and *max* attributes. Hence, we can insert into an empty vEB tree in constant time. Similarly, if we know that a vEB tree has only one element, we can delete that element in constant time by updating only *min* and *max*. These properties will allow us to cut short the chain of recursive calls.

Even if the universe size u is an odd power of 2, the difference in the sizes of the summary vEB tree and the clusters will not turn out to affect the asymptotic running times of the vEB-tree operations. The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence

$$T(u) \le T(\sqrt[\uparrow]{u}) + O(1) . \tag{20.4}$$

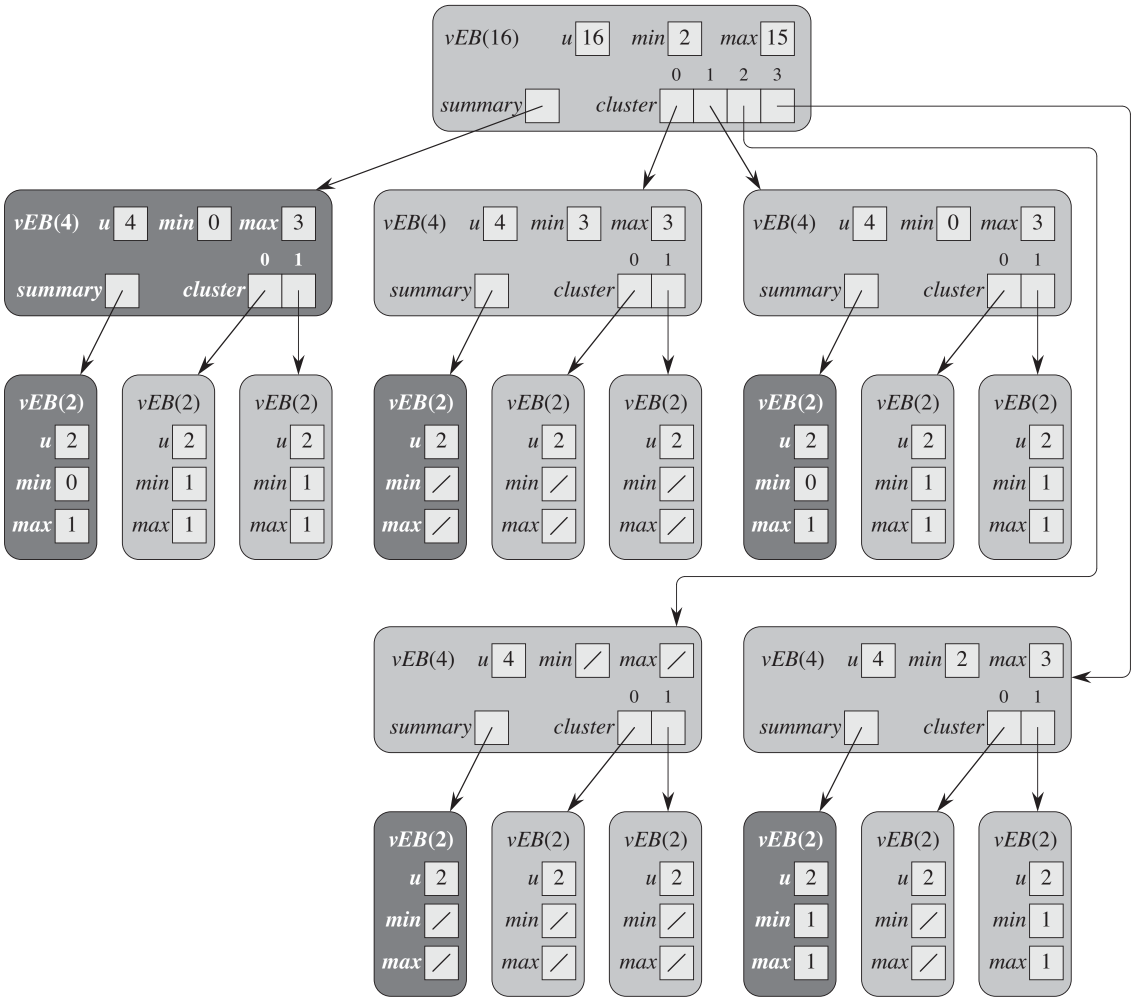


Figure 20.6 A *vEB*(16) tree corresponding to the proto-vEB tree in Figure 20.4. It stores the set $\{2, 3, 4, 5, 7, 14, 15\}$. Slashes indicate NIL values. The value stored in the *min* attribute of a vEB tree does not appear in any of its clusters. Heavy shading serves the same purpose here as in Figure 20.4.

This recurrence looks similar to recurrence (20.2), and we will solve it in a similar fashion. Letting $m = \lg u$, we rewrite it as

$$T(2^m) \le T(2^{\lceil m/2 \rceil}) + O(1) .$$

Noting that $\lceil m/2 \rceil \le 2m/3$ for all $m \ge 2$, we have

$$T(2^m) \le T(2^{2m/3}) + O(1) .$$

Letting $S(m) = T(2^m)$, we rewrite this last recurrence as

$$S(m) \le S(2m/3) + O(1) ,$$

which, by case 2 of the master method, has the solution $S(m) = O(\lg m)$. (In terms of the asymptotic solution, the fraction $2/3$ does not make any difference compared with the fraction $1/2$, because when we apply the master method, we find that $\log_{3/2} 1 = \log_2 1 = 0$.) Thus, we have $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

Before using a van Emde Boas tree, we must know the universe size u, so that we can create a van Emde Boas tree of the appropriate size that initially represents an empty set. As Problem 20-1 asks you to show, the total space requirement of a van Emde Boas tree is $O(u)$, and it is straightforward to create an empty tree in $O(u)$ time. In contrast, we can create an empty red-black tree in constant time. Therefore, we might not want to use a van Emde Boas tree when we perform only a small number of operations, since the time to create the data structure would exceed the time saved in the individual operations. This drawback is usually not significant, since we typically use a simple data structure, such as an array or linked list, to represent a set with only a few elements.

20.3.2 Operations on a van Emde Boas tree

We are now ready to see how to perform operations on a van Emde Boas tree. As we did for the proto van Emde Boas structure, we will consider the querying operations first, and then INSERT and DELETE. Due to the slight asymmetry between the minimum and maximum elements in a vEB tree—when a vEB tree contains at least two elements, the minumum element does not appear within a cluster but the maximum element does—we will provide pseudocode for all five querying operations. As in the operations on proto van Emde Boas structures, the operations here that take parameters V and x, where V is a van Emde Boas tree and x is an element, assume that $0 \le x < V.u$.

Finding the minimum and maximum elements

Because we store the minimum and maximum in the attributes *min* and *max*, two of the operations are one-liners, taking constant time:

```
vEB-TREE-MINIMUM(V)
1  return V.min
```

```
vEB-TREE-MAXIMUM(V)
1  return V.max
```

Determining whether a value is in the set

The procedure vEB-TREE-MEMBER(V, x) has a recursive case like that of PROTO-vEB-MEMBER, but the base case is a little different. We also check directly whether x equals the minimum or maximum element. Since a vEB tree doesn't store bits as a proto-vEB structure does, we design vEB-TREE-MEMBER to return TRUE or FALSE rather than 1 or 0.

```
vEB-TREE-MEMBER(V, x)
1  if x == V.min or x == V.max
2      return TRUE
3  elseif V.u == 2
4      return FALSE
5  else return vEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

Line 1 checks to see whether x equals either the minimum or maximum element. If it does, line 2 returns TRUE. Otherwise, line 3 tests for the base case. Since a $vEB(2)$ tree has no elements other than those in *min* and *max*, if it is the base case, line 4 returns FALSE. The other possibility—it is not a base case and x equals neither *min* nor *max*—is handled by the recursive call in line 5.

Recurrence (20.4) characterizes the running time of the vEB-TREE-MEMBER procedure, and so this procedure takes $O(\lg \lg u)$ time.

Finding the successor and predecessor

Next we see how to implement the SUCCESSOR operation. Recall that the procedure PROTO-vEB-SUCCESSOR(V, x) could make two recursive calls: one to determine whether x's successor resides in the same cluster as x and, if it does not, one to find the cluster containing x's successor. Because we can access the maximum value in a vEB tree quickly, we can avoid making two recursive calls, and instead make one recursive call on either a cluster or on the summary, but not on both.

```
vEB-TREE-SUCCESSOR$(V, x)$
 1  if $V.u == 2$
 2      if $x == 0$ and $V.max == 1$
 3          return 1
 4      else return NIL
 5  elseif $V.min \neq$ NIL and $x < V.min$
 6      return $V.min$
 7  else $max\text{-}low =$ vEB-TREE-MAXIMUM$(V.cluster[\text{high}(x)])$
 8      if $max\text{-}low \neq$ NIL and $\text{low}(x) < max\text{-}low$
 9          $offset =$ vEB-TREE-SUCCESSOR$(V.cluster[\text{high}(x)], \text{low}(x))$
10          return $\text{index}(\text{high}(x), offset)$
11      else $succ\text{-}cluster =$ vEB-TREE-SUCCESSOR$(V.summary, \text{high}(x))$
12          if $succ\text{-}cluster ==$ NIL
13              return NIL
14          else $offset =$ vEB-TREE-MINIMUM$(V.cluster[succ\text{-}cluster])$
15              return $\text{index}(succ\text{-}cluster, offset)$
```

This procedure has six **return** statements and several cases. We start with the base case in lines 2–4, which returns 1 in line 3 if we are trying to find the successor of 0 and 1 is in the 2-element set; otherwise, the base case returns NIL in line 4.

If we are not in the base case, we next check in line 5 whether x is strictly less than the minimum element. If so, then we simply return the minimum element in line 6.

If we get to line 7, then we know that we are not in a base case and that x is greater than or equal to the minimum value in the vEB tree V. Line 7 assigns to *max-low* the maximum element in x's cluster. If x's cluster contains some element that is greater than x, then we know that x's successor lies somewhere within x's cluster. Line 8 tests for this condition. If x's successor is within x's cluster, then line 9 determines where in the cluster it is, and line 10 returns the successor in the same way as line 7 of PROTO-vEB-SUCCESSOR.

We get to line 11 if x is greater than or equal to the greatest element in its cluster. In this case, lines 11–15 find x's successor in the same way as lines 8–12 of PROTO-vEB-SUCCESSOR.

It is easy to see how recurrence (20.4) characterizes the running time of vEB-TREE-SUCCESSOR. Depending on the result of the test in line 7, the procedure calls itself recursively in either line 9 (on a vEB tree with universe size $\sqrt[\downarrow]{u}$) or line 11 (on a vEB tree with universe size $\sqrt[\uparrow]{u}$). In either case, the one recursive call is on a vEB tree with universe size at most $\sqrt[\uparrow]{u}$. The remainder of the procedure, including the calls to vEB-TREE-MINIMUM and vEB-TREE-MAXIMUM, takes $O(1)$ time. Hence, vEB-TREE-SUCCESSOR runs in $O(\lg\lg u)$ worst-case time.

The vEB-TREE-PREDECESSOR procedure is symmetric to the vEB-TREE-SUCCESSOR procedure, but with one additional case:

```
vEB-TREE-PREDECESSOR$(V, x)$
 1  if $V.u == 2$
 2      if $x == 1$ and $V.min == 0$
 3          return 0
 4      else return NIL
 5  elseif $V.max \neq$ NIL and $x > V.max$
 6      return $V.max$
 7  else $min\text{-}low = $ vEB-TREE-MINIMUM$(V.cluster[\text{high}(x)])$
 8      if $min\text{-}low \neq$ NIL and $\text{low}(x) > min\text{-}low$
 9          $offset = $ vEB-TREE-PREDECESSOR$(V.cluster[\text{high}(x)], \text{low}(x))$
10          return $\text{index}(\text{high}(x), offset)$
11      else $pred\text{-}cluster = $ vEB-TREE-PREDECESSOR$(V.summary, \text{high}(x))$
12          if $pred\text{-}cluster ==$ NIL
13              if $V.min \neq$ NIL and $x > V.min$
14                  return $V.min$
15              else return NIL
16          else $offset = $ vEB-TREE-MAXIMUM$(V.cluster[pred\text{-}cluster])$
17              return $\text{index}(pred\text{-}cluster, offset)$
```

Lines 13–14 form the additional case. This case occurs when x's predecessor, if it exists, does not reside in x's cluster. In vEB-TREE-SUCCESSOR, we were assured that if x's successor resides outside of x's cluster, then it must reside in a higher-numbered cluster. But if x's predecessor is the minimum value in vEB tree V, then the successor resides in no cluster at all. Line 13 checks for this condition, and line 14 returns the minimum value as appropriate.

This extra case does not affect the asymptotic running time of vEB-TREE-PREDECESSOR when compared with vEB-TREE-SUCCESSOR, and so vEB-TREE-PREDECESSOR runs in $O(\lg \lg u)$ worst-case time.

Inserting an element

Now we examine how to insert an element into a vEB tree. Recall that PROTO-vEB-INSERT made two recursive calls: one to insert the element and one to insert the element's cluster number into the summary. The vEB-TREE-INSERT procedure will make only one recursive call. How can we get away with just one? When we insert an element, either the cluster that it goes into already has another element or it does not. If the cluster already has another element, then the cluster number is already in the summary, and so we do not need to make that recursive call. If the cluster does not already have another element, then the element being inserted becomes the only element in the cluster, and we do not need to recurse to insert an element into an empty vEB tree:

```
vEB-EMPTY-TREE-INSERT($V, x$)
1  $V.min = x$
2  $V.max = x$
```

With this procedure in hand, here is the pseudocode for vEB-TREE-INSERT(V, x), which assumes that x is not already an element in the set represented by vEB tree V:

```
vEB-TREE-INSERT($V, x$)
 1  if $V.min$ == NIL
 2      vEB-EMPTY-TREE-INSERT($V, x$)
 3  else if $x < V.min$
 4          exchange $x$ with $V.min$
 5      if $V.u > 2$
 6          if vEB-TREE-MINIMUM($V.cluster[\text{high}(x)]$) == NIL
 7              vEB-TREE-INSERT($V.summary, \text{high}(x)$)
 8              vEB-EMPTY-TREE-INSERT($V.cluster[\text{high}(x)], \text{low}(x)$)
 9          else vEB-TREE-INSERT($V.cluster[\text{high}(x)], \text{low}(x)$)
10      if $x > V.max$
11          $V.max = x$
```

This procedure works as follows. Line 1 tests whether V is an empty vEB tree and, if it is, then line 2 handles this easy case. Lines 3–11 assume that V is not empty, and therefore some element will be inserted into one of V's clusters. But that element might not necessarily be the element x passed to vEB-TREE-INSERT. If $x < min$, as tested in line 3, then x needs to become the new *min*. We don't want to lose the original *min*, however, and so we need to insert it into one of V's clusters. In this case, line 4 exchanges x with *min*, so that we insert the original *min* into one of V's clusters.

We execute lines 6–9 only if V is not a base-case vEB tree. Line 6 determines whether the cluster that x will go into is currently empty. If so, then line 7 inserts x's cluster number into the summary and line 8 handles the easy case of inserting x into an empty cluster. If x's cluster is not currently empty, then line 9 inserts x into its cluster. In this case, we do not need to update the summary, since x's cluster number is already a member of the summary.

Finally, lines 10–11 take care of updating *max* if $x > max$. Note that if V is a base-case vEB tree that is not empty, then lines 3–4 and 10–11 update *min* and *max* properly.

Once again, we can easily see how recurrence (20.4) characterizes the running time. Depending on the result of the test in line 6, either the recursive call in line 7 (run on a vEB tree with universe size $\sqrt[\uparrow]{u}$) or the recursive call in line 9 (run on a vEB with universe size $\sqrt[\downarrow]{u}$) executes. In either case, the one recursive call is on a vEB tree with universe size at most $\sqrt[\uparrow]{u}$. Because the remainder of vEB-TREE-INSERT takes $O(1)$ time, recurrence (20.4) applies, and so the running time is $O(\lg \lg u)$.

Deleting an element

Finally, we look at how to delete an element from a vEB tree. The procedure vEB-TREE-DELETE(V, x) assumes that x is currently an element in the set represented by the vEB tree V.

```
vEB-TREE-DELETE(V, x)
 1  if V.min == V.max
 2      V.min = NIL
 3      V.max = NIL
 4  elseif V.u == 2
 5      if x == 0
 6          V.min = 1
 7      else V.min = 0
 8      V.max = V.min
 9  else if x == V.min
10          first-cluster = vEB-TREE-MINIMUM(V.summary)
11          x = index(first-cluster,
                vEB-TREE-MINIMUM(V.cluster[first-cluster]))
12          V.min = x
13      vEB-TREE-DELETE(V.cluster[high(x)], low(x))
14      if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
15          vEB-TREE-DELETE(V.summary, high(x))
16          if x == V.max
17              summary-max = vEB-TREE-MAXIMUM(V.summary)
18              if summary-max == NIL
19                  V.max = V.min
20              else V.max = index(summary-max,
                        vEB-TREE-MAXIMUM(V.cluster[summary-max]))
21      elseif x == V.max
22          V.max = index(high(x),
                vEB-TREE-MAXIMUM(V.cluster[high(x)]))
```

The vEB-TREE-DELETE procedure works as follows. If the vEB tree V contains only one element, then it's just as easy to delete it as it was to insert an element into an empty vEB tree: just set *min* and *max* to NIL. Lines 1–3 handle this case. Otherwise, V has at least two elements. Line 4 tests whether V is a base-case vEB tree and, if so, lines 5–8 set *min* and *max* to the one remaining element.

Lines 9–22 assume that V has two or more elements and that $u \geq 4$. In this case, we will have to delete an element from a cluster. The element we delete from a cluster might not be x, however, because if x equals *min*, then once we have deleted x, some other element within one of V's clusters becomes the new *min*, and we have to delete that other element from its cluster. If the test in line 9 reveals that we are in this case, then line 10 sets *first-cluster* to the number of the cluster that contains the lowest element other than *min*, and line 11 sets x to the value of the lowest element in that cluster. This element becomes the new *min* in line 12 and, because we set x to its value, it is the element that will be deleted from its cluster.

When we reach line 13, we know that we need to delete element x from its cluster, whether x was the value originally passed to vEB-TREE-DELETE or x is the element becoming the new minimum. Line 13 deletes x from its cluster. That cluster might now become empty, which line 14 tests, and if it does, then we need to remove x's cluster number from the summary, which line 15 handles. After updating the summary, we might need to update *max*. Line 16 checks to see whether we are deleting the maximum element in V and, if we are, then line 17 sets *summary-max* to the number of the highest-numbered nonempty cluster. (The call vEB-TREE-MAXIMUM($V.summary$) works because we have already recursively called vEB-TREE-DELETE on $V.summary$, and therefore $V.summary.max$ has already been updated as necessary.) If all of V's clusters are empty, then the only remaining element in V is *min*; line 18 checks for this case, and line 19 updates *max* appropriately. Otherwise, line 20 sets *max* to the maximum element in the highest-numbered cluster. (If this cluster is where the element has been deleted, we again rely on the recursive call in line 13 having already corrected that cluster's *max* attribute.)

Finally, we have to handle the case in which x's cluster did not become empty due to x being deleted. Although we do not have to update the summary in this case, we might have to update *max*. Line 21 tests for this case, and if we have to update *max*, line 22 does so (again relying on the recursive call to have corrected *max* in the cluster).

Now we show that vEB-TREE-DELETE runs in $O(\lg \lg u)$ time in the worst case. At first glance, you might think that recurrence (20.4) does not always apply, because a single call of vEB-TREE-DELETE can make two recursive calls: one on line 13 and one on line 15. Although the procedure can make both recursive calls, let's think about what happens when it does. In order for the recursive call on line 15 to occur, the test on line 14 must show that x's cluster is empty. The only way that x's cluster can be empty is if x was the only element in its cluster when we made the recursive call on line 13. But if x was the only element in its cluster, then that recursive call took $O(1)$ time, because it executed only lines 1–3. Thus, we have two mutually exclusive possibilities:

- The recursive call on line 13 took constant time.
- The recursive call on line 15 did not occur.

In either case, recurrence (20.4) characterizes the running time of vEB-TREE-DELETE, and hence its worst-case running time is $O(\lg \lg u)$.

Exercises

20.3-1
Modify vEB trees to support duplicate keys.

20.3-2
Modify vEB trees to support keys that have associated satellite data.

20.3-3
Write pseudocode for a procedure that creates an empty van Emde Boas tree.

20.3-4
What happens if you call vEB-TREE-INSERT with an element that is already in the vEB tree? What happens if you call vEB-TREE-DELETE with an element that is not in the vEB tree? Explain why the procedures exhibit the behavior that they do. Show how to modify vEB trees and their operations so that we can check in constant time whether an element is present.

20.3-5
Suppose that instead of $\sqrt[\uparrow]{u}$ clusters, each with universe size $\sqrt[\downarrow]{u}$, we constructed vEB trees to have $u^{1/k}$ clusters, each with universe size $u^{1-1/k}$, where $k > 1$ is a constant. If we were to modify the operations appropriately, what would be their running times? For the purpose of analysis, assume that $u^{1/k}$ and $u^{1-1/k}$ are always integers.

20.3-6
Creating a vEB tree with universe size u requires $O(u)$ time. Suppose we wish to explicitly account for that time. What is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\lg \lg u)$?

Problems

20-1 Space requirements for van Emde Boas trees

This problem explores the space requirements for van Emde Boas trees and suggests a way to modify the data structure to make its space requirement depend on the number n of elements actually stored in the tree, rather than on the universe size u. For simplicity, assume that $\sqrt{u}$ is always an integer.

a. Explain why the following recurrence characterizes the space requirement $P(u)$ of a van Emde Boas tree with universe size u:

$$P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u}) . \tag{20.5}$$

b. Prove that recurrence (20.5) has the solution $P(u) = O(u)$.

In order to reduce the space requirements, let us define a ***reduced-space van Emde Boas tree***, or ***RS-vEB tree***, as a vEB tree V but with the following changes:

- The attribute $V.cluster$, rather than being stored as a simple array of pointers to vEB trees with universe size $\sqrt{u}$, is a hash table (see Chapter 11) stored as a dynamic table (see Section 17.4). Corresponding to the array version of $V.cluster$, the hash table stores pointers to RS-vEB trees with universe size $\sqrt{u}$. To find the ith cluster, we look up the key i in the hash table, so that we can find the ith cluster by a single search in the hash table.
- The hash table stores only pointers to nonempty clusters. A search in the hash table for an empty cluster returns NIL, indicating that the cluster is empty.
- The attribute $V.summary$ is NIL if all clusters are empty. Otherwise, $V.summary$ points to an RS-vEB tree with universe size $\sqrt{u}$.

Because the hash table is implemented with a dynamic table, the space it requires is proportional to the number of nonempty clusters.

When we need to insert an element into an empty RS-vEB tree, we create the RS-vEB tree by calling the following procedure, where the parameter u is the universe size of the RS-vEB tree:

```
CREATE-NEW-RS-VEB-TREE($u$)
1  allocate a new vEB tree $V$
2  $V.u = u$
3  $V.min =$ NIL
4  $V.max =$ NIL
5  $V.summary =$ NIL
6  create $V.cluster$ as an empty dynamic hash table
7  return $V$
```

c. Modify the VEB-TREE-INSERT procedure to produce pseudocode for the procedure RS-VEB-TREE-INSERT(V, x), which inserts x into the RS-vEB tree V, calling CREATE-NEW-RS-VEB-TREE as appropriate.

d. Modify the VEB-TREE-SUCCESSOR procedure to produce pseudocode for the procedure RS-VEB-TREE-SUCCESSOR(V, x), which returns the successor of x in RS-vEB tree V, or NIL if x has no successor in V.

e. Prove that, under the assumption of simple uniform hashing, your RS-VEB-TREE-INSERT and RS-VEB-TREE-SUCCESSOR procedures run in $O(\lg \lg u)$ expected time.

f. Assuming that elements are never deleted from a vEB tree, prove that the space requirement for the RS-vEB tree structure is $O(n)$, where n is the number of elements actually stored in the RS-vEB tree.

g. RS-vEB trees have another advantage over vEB trees: they require less time to create. How long does it take to create an empty RS-vEB tree?

20-2 *y*-fast tries

This problem investigates D. Willard's "y-fast tries" which, like van Emde Boas trees, perform each of the operations MEMBER, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on elements drawn from a universe with size u in $O(\lg \lg u)$ worst-case time. The INSERT and DELETE operations take $O(\lg \lg u)$ amortized time. Like reduced-space van Emde Boas trees (see Problem 20-1), y-fast tries use only $O(n)$ space to store n elements. The design of y-fast tries relies on perfect hashing (see Section 11.5).

As a preliminary structure, suppose that we create a perfect hash table containing not only every element in the dynamic set, but every prefix of the binary representation of every element in the set. For example, if $u = 16$, so that $\lg u = 4$, and $x = 13$ is in the set, then because the binary representation of 13 is 1101, the perfect hash table would contain the strings 1, 11, 110, and 1101. In addition to the hash table, we create a doubly linked list of the elements currently in the set, in increasing order.

a. How much space does this structure require?

b. Show how to perform the MINIMUM and MAXIMUM operations in $O(1)$ time; the MEMBER, PREDECESSOR, and SUCCESSOR operations in $O(\lg \lg u)$ time; and the INSERT and DELETE operations in $O(\lg u)$ time.

To reduce the space requirement to $O(n)$, we make the following changes to the data structure:

- We cluster the n elements into $n/\lg u$ groups of size $\lg u$. (Assume for now that $\lg u$ divides n.) The first group consists of the $\lg u$ smallest elements in the set, the second group consists of the next $\lg u$ smallest elements, and so on.
- We designate a "representative" value for each group. The representative of the ith group is at least as large as the largest element in the ith group, and it is smaller than every element of the $(i+1)$st group. (The representative of the last group can be the maximum possible element $u-1$.) Note that a representative might be a value not currently in the set.
- We store the $\lg u$ elements of each group in a balanced binary search tree, such as a red-black tree. Each representative points to the balanced binary search tree for its group, and each balanced binary search tree points to its group's representative.
- The perfect hash table stores only the representatives, which are also stored in a doubly linked list in increasing order.

We call this structure a ***y*-fast trie**.

c. Show that a y-fast trie requires only $O(n)$ space to store n elements.

d. Show how to perform the MINIMUM and MAXIMUM operations in $O(\lg\lg u)$ time with a y-fast trie.

e. Show how to perform the MEMBER operation in $O(\lg\lg u)$ time.

f. Show how to perform the PREDECESSOR and SUCCESSOR operations in $O(\lg\lg u)$ time.

g. Explain why the INSERT and DELETE operations take $\Omega(\lg\lg u)$ time.

h. Show how to relax the requirement that each group in a y-fast trie has exactly $\lg u$ elements to allow INSERT and DELETE to run in $O(\lg\lg u)$ amortized time without affecting the asymptotic running times of the other operations.

Chapter notes

The data structure in this chapter is named after P. van Emde Boas, who described an early form of the idea in 1975 [339]. Later papers by van Emde Boas [340] and van Emde Boas, Kaas, and Zijlstra [341] refined the idea and the exposition. Mehlhorn and Näher [252] subsequently extended the ideas to apply to universe sizes that are prime. Mehlhorn's book [249] contains a slightly different treatment of van Emde Boas trees than the one in this chapter.

Using the ideas behind van Emde Boas trees, Dementiev et al. [83] developed a nonrecursive, three-level search tree that ran faster than van Emde Boas trees in their own experiments.

Wang and Lin [347] designed a hardware-pipelined version of van Emde Boas trees, which achieves constant amortized time per operation and uses $O(\lg \lg u)$ stages in the pipeline.

A lower bound by Pătraşcu and Thorup [273, 274] for finding the predecessor shows that van Emde Boas trees are optimal for this operation, even if randomization is allowed.

21 Data Structures for Disjoint Sets

Some applications involve grouping n distinct elements into a collection of disjoint sets. These applications often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets. This chapter explores methods for maintaining a data structure that supports these operations.

Section 21.1 describes the operations supported by a disjoint-set data structure and presents a simple application. In Section 21.2, we look at a simple linked-list implementation for disjoint sets. Section 21.3 presents a more efficient representation using rooted trees. The running time using the tree representation is theoretically superlinear, but for all practical purposes it is linear. Section 21.4 defines and discusses a very quickly growing function and its very slowly growing inverse, which appears in the running time of operations on the tree-based implementation, and then, by a complex amortized analysis, proves an upper bound on the running time that is just barely superlinear.

21.1 Disjoint-set operations

A ***disjoint-set data structure*** maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. We identify each set by a ***representative***, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times. Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered).

As in the other dynamic-set implementations we have studied, we represent each element of a set by an object. Letting x denote an object, we wish to support the following operations:

MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y, removing them from the collection $\mathcal{S}$. In practice, we often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

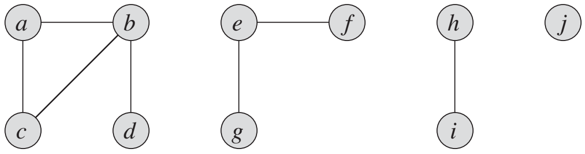
Throughout this chapter, we shall analyze the running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m, the total number of MAKE-SET, UNION, and FIND-SET operations. Since the sets are disjoint, each UNION operation reduces the number of sets by one. After $n - 1$ UNION operations, therefore, only one set remains. The number of UNION operations is thus at most $n - 1$. Note also that since the MAKE-SET operations are included in the total number of operations m, we have $m \geq n$. We assume that the n MAKE-SET operations are the first n operations performed.

An application of disjoint-set data structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph (see Section B.4). Figure 21.1(a), for example, shows a graph with four connected components.

The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has preprocessed the graph, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component.[1] (In pseudocode, we denote the set of vertices of a graph G by $G.V$ and the set of edges by $G.E$.)

[1]When the edges of the graph are static—not changing over time—we can compute the connected components faster by using depth-first search (Exercise 22.3-12). Sometimes, however, the edges are added dynamically and we need to maintain the connected components as each edge is added. In this case, the implementation given here can be more efficient than running a new depth-first search for each new edge.



(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

Figure 21.1 **(a)** A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. **(b)** The collection of disjoint sets after processing each edge.

```
CONNECTED-COMPONENTS(G)
1  for each vertex $v \in G.V$
2      MAKE-SET($v$)
3  for each edge $(u, v) \in G.E$
4      if FIND-SET($u$) $\neq$ FIND-SET($v$)
5          UNION($u, v$)
```

```
SAME-COMPONENT(u, v)
1  if FIND-SET($u$) == FIND-SET($v$)
2      return TRUE
3  else return FALSE
```

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u, v), it unites the sets containing u and v. By Exercise 21.1-2, after processing all the edges, two vertices are in the same connected component if and only if the corresponding objects are in the same set. Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component. Figure 21.1(b) illustrates how CONNECTED-COMPONENTS computes the disjoint sets.

In an actual implementation of this connected-components algorithm, the representations of the graph and the disjoint-set data structure would need to reference each other. That is, an object representing a vertex would contain a pointer to the corresponding disjoint-set object, and vice versa. These programming details depend on the implementation language, and we do not address them further here.

Exercises

21.1-1

Suppose that CONNECTED-COMPONENTS is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ and the edges of E are processed in the order $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f)$, $(g, j), (a, e)$. List the vertices in each connected component after each iteration of lines 3–5.

21.1-2

Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices are in the same connected component if and only if they are in the same set.

21.1-3

During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k.

21.2 Linked-list representation of disjoint sets

Figure 21.2(a) shows a simple way to implement a disjoint-set data structure: each set is represented by its own linked list. The object for each set has attributes *head*, pointing to the first object in the list, and *tail*, pointing to the last object. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Within each linked list, the objects may appear in any order. The representative is the set member in the first object in the list.

With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring $O(1)$ time. To carry out MAKE-SET(x), we create a new linked list whose only object is x. For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that *head* points to. For example, in Figure 21.2(a), the call FIND-SET(g) would return f.