V Advanced Data Structures

Introduction

This part returns to studying data structures that support operations on dynamic sets, but at a more advanced level than Part III. Two of the chapters, for example, make extensive use of the amortized analysis techniques we saw in Chapter 17.

Chapter 18 presents B-trees, which are balanced search trees specifically designed to be stored on disks. Because disks operate much more slowly than random-access memory, we measure the performance of B-trees not only by how much computing time the dynamic-set operations consume but also by how many disk accesses they perform. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, but B-tree operations keep the height low.

Chapter 19 gives an implementation of a mergeable heap, which supports the operations INSERT, MINIMUM, EXTRACT-MIN, and UNION.[1] The UNION operation unites, or merges, two heaps. Fibonacci heaps—the data structure in Chapter 19—also support the operations DELETE and DECREASE-KEY. We use amortized time bounds to measure the performance of Fibonacci heaps. The operations INSERT, MINIMUM, and UNION take only $O(1)$ actual and amortized time on Fibonacci heaps, and the operations EXTRACT-MIN and DELETE take $O(\lg n)$ amortized time. The most significant advantage of Fibonacci heaps, however, is that DECREASE-KEY takes only $O(1)$ amortized time. Because the DECREASE-

[1] As in Problem 10-2, we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, and so we can also refer to it as a ***mergeable min-heap***. Alternatively, if it supported MAXIMUM and EXTRACT-MAX, it would be a ***mergeable max-heap***. Unless we specify otherwise, mergeable heaps will be by default mergeable min-heaps.

KEY operation takes constant amortized time, Fibonacci heaps are key components of some of the asymptotically fastest algorithms to date for graph problems.

Noting that we can beat the $\Omega(n \lg n)$ lower bound for sorting when the keys are integers in a restricted range, Chapter 20 asks whether we can design a data structure that supports the dynamic-set operations SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $o(\lg n)$ time when the keys are integers in a restricted range. The answer turns out to be that we can, by using a recursive data structure known as a van Emde Boas tree. If the keys are unique integers drawn from the set $\{0, 1, 2, \ldots, u - 1\}$, where u is an exact power of 2, then van Emde Boas trees support each of the above operations in $O(\lg \lg u)$ time.

Finally, Chapter 21 presents data structures for disjoint sets. We have a universe of n elements that are partitioned into dynamic sets. Initially, each element belongs to its own singleton set. The operation UNION unites two sets, and the query FIND-SET identifies the unique set that contains a given element at the moment. By representing each set as a simple rooted tree, we obtain surprisingly fast operations: a sequence of m operations runs in $O(m\, \alpha(n))$ time, where $\alpha(n)$ is an incredibly slowly growing function—$\alpha(n)$ is at most 4 in any conceivable application. The amortized analysis that proves this time bound is as complex as the data structure is simple.

The topics covered in this part are by no means the only examples of "advanced" data structures. Other advanced data structures include the following:

- ***Dynamic trees***, introduced by Sleator and Tarjan [319] and discussed by Tarjan [330], maintain a forest of disjoint rooted trees. Each edge in each tree has a real-valued cost. Dynamic trees support queries to find parents, roots, edge costs, and the minimum edge cost on a simple path from a node up to a root. Trees may be manipulated by cutting edges, updating all edge costs on a simple path from a node up to a root, linking a root into another tree, and making a node the root of the tree it appears in. One implementation of dynamic trees gives an $O(\lg n)$ amortized time bound for each operation; a more complicated implementation yields $O(\lg n)$ worst-case time bounds. Dynamic trees are used in some of the asymptotically fastest network-flow algorithms.
- ***Splay trees***, developed by Sleator and Tarjan [320] and, again, discussed by Tarjan [330], are a form of binary search tree on which the standard search-tree operations run in $O(\lg n)$ amortized time. One application of splay trees simplifies dynamic trees.
- ***Persistent*** data structures allow queries, and sometimes updates as well, on past versions of a data structure. Driscoll, Sarnak, Sleator, and Tarjan [97] present techniques for making linked data structures persistent with only a small time

and space cost. Problem 13-1 gives a simple example of a persistent dynamic set.

- As in Chapter 20, several data structures allow a faster implementation of dictionary operations (INSERT, DELETE, and SEARCH) for a restricted universe of keys. By taking advantage of these restrictions, they are able to achieve better worst-case asymptotic running times than comparison-based data structures. Fredman and Willard introduced ***fusion trees*** [115], which were the first data structure to allow faster dictionary operations when the universe is restricted to integers. They showed how to implement these operations in $O(\lg n / \lg \lg n)$ time. Several subsequent data structures, including ***exponential search trees*** [16], have also given improved bounds on some or all of the dictionary operations and are mentioned in the chapter notes throughout this book.
- ***Dynamic graph data structures*** support various queries while allowing the structure of a graph to change through operations that insert or delete vertices or edges. Examples of the queries that they support include vertex connectivity [166], edge connectivity, minimum spanning trees [165], biconnectivity, and transitive closure [164].

Chapter notes throughout this book mention additional data structures.

18 B-Trees

B-trees are balanced search trees designed to work well on disks or other direct-access secondary storage devices. B-trees are similar to red-black trees (Chapter 13), but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands. That is, the "branching factor" of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used. B-trees are similar to red-black trees in that every n-node B-tree has height $O(\lg n)$. The exact height of a B-tree can be considerably less than that of a red-black tree, however, because its branching factor, and hence the base of the logarithm that expresses its height, can be much larger. Therefore, we can also use B-trees to implement many dynamic-set operations in time $O(\lg n)$.

B-trees generalize binary search trees in a natural manner. Figure 18.1 shows a simple B-tree. If an internal B-tree node x contains $x.n$ keys, then x has $x.n + 1$ children. The keys in node x serve as dividing points separating the range of keys handled by x into $x.n + 1$ subranges, each handled by one child of x. When searching for a key in a B-tree, we make an $(x.n + 1)$-way decision based on comparisons with the $x.n$ keys stored at node x. The structure of leaf nodes differs from that of internal nodes; we will examine these differences in Section 18.1.

Section 18.1 gives a precise definition of B-trees and proves that the height of a B-tree grows only logarithmically with the number of nodes it contains. Section 18.2 describes how to search for a key and insert a key into a B-tree, and Section 18.3 discusses deletion. Before proceeding, however, we need to ask why we evaluate data structures designed to work on a disk differently from data structures designed to work in main random-access memory.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory capacity. The ***primary memory*** (or ***main memory***) of a computer system normally

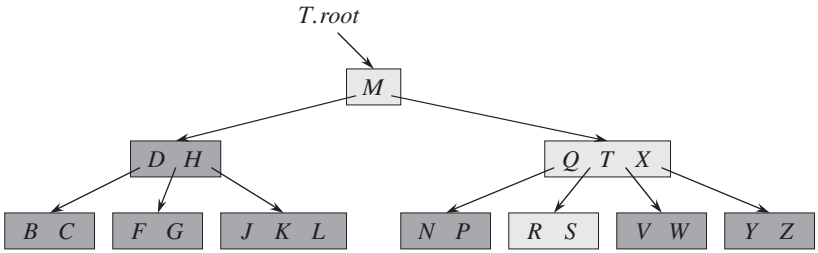


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

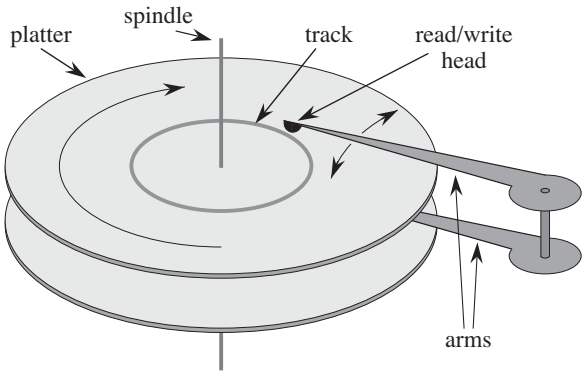


Figure 18.2 A typical disk drive. It comprises one or more platters (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head at the end of an arm. Arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when the head is stationary.

consists of silicon memory chips. This technology is typically more than an order of magnitude more expensive per bit stored than magnetic storage technology, such as tapes or disks. Most computer systems also have ***secondary storage*** based on magnetic disks; the amount of such secondary storage often exceeds the amount of primary memory by at least two orders of magnitude.

Figure 18.2 shows a typical disk drive. The drive consists of one or more ***platters***, which rotate at a constant speed around a common ***spindle***. A magnetizable material covers the surface of each platter. The drive reads and writes each platter by a ***head*** at the end of an ***arm***. The arms can move their heads toward or away

from the spindle. When a given head is stationary, the surface that passes underneath it is called a ***track***. Multiple platters increase only the disk drive's capacity and not its performance.

Although disks are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts.[1] The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disks rotate at speeds of 5400–15,000 revolutions per minute (RPM). We typically see 15,000 RPM speeds in server-grade drives, 7200 RPM speeds in drives for desktops, and 5400 RPM speeds in drives for laptops. Although 7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5 orders of magnitude longer than the 50 nanosecond access times (more or less) commonly found for silicon memory. In other words, if we have to wait a full rotation for a particular item to come under the read/write head, we could access main memory more than 100,000 times during that span. On average we have to wait for only half a rotation, but still, the difference in access times for silicon memory compared with disks is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disks are in the range of 8 to 11 milliseconds.

In order to amortize the time spent waiting for mechanical movements, disks access not just one item but several at a time. Information is divided into a number of equal-sized ***pages*** of bits that appear consecutively within tracks, and each disk read or write is of one or more entire pages. For a typical disk, a page might be 2^{11} to 2^{14} bytes in length. Once the read/write head is positioned correctly and the disk has rotated to the beginning of the desired page, reading or writing a magnetic disk is entirely electronic (aside from the rotation of the disk), and the disk can quickly read or write large amounts of data.

Often, accessing a page of information and reading it from a disk takes longer than examining all the information read. For this reason, in this chapter we shall look separately at the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

We measure the number of disk accesses in terms of the number of pages of information that need to be read from or written to the disk. We note that disk-access time is not constant—it depends on the distance between the current track and the desired track and also on the initial rotational position of the disk. We shall

[1] As of this writing, solid-state drives have recently come onto the consumer market. Although they are faster than mechanical disk drives, they cost more per gigabyte and have lower capacities than mechanical disk drives.

nonetheless use the number of pages read or written as a first-order approximation of the total time spent accessing the disk.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed. B-tree algorithms keep only a constant number of pages in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.

We model disk operations in our pseudocode as follows. Let x be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the attributes of the object as usual: $x.key$, for example. If the object referred to by x resides on disk, however, then we must perform the operation DISK-READ(x) to read object x into main memory before we can refer to its attributes. (We assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses; it is a "no-op.") Similarly, the operation DISK-WRITE(x) is used to save any changes that have been made to the attributes of object x. That is, the typical pattern for working with an object is as follows:

```
x = a pointer to some object
DISK-READ(x)
operations that access and/or modify the attributes of x
DISK-WRITE(x)          // omitted if no attributes of x were changed
other operations that access but do not modify attributes of x
```

The system can keep only a limited number of pages in main memory at any one time. We shall assume that the system flushes from main memory pages no longer in use; our B-tree algorithms will ignore this issue.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page, and this size limits the number of children a B-tree node can have.

For a large B-tree stored on a disk, we often see branching factors between 50 and 2000, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key. Figure 18.3 shows a B-tree with a branching factor of 1001 and height 2 that can store over one billion keys; nevertheless, since we can keep the root node permanently in main memory, we can find any key in this tree by making at most only two disk accesses.

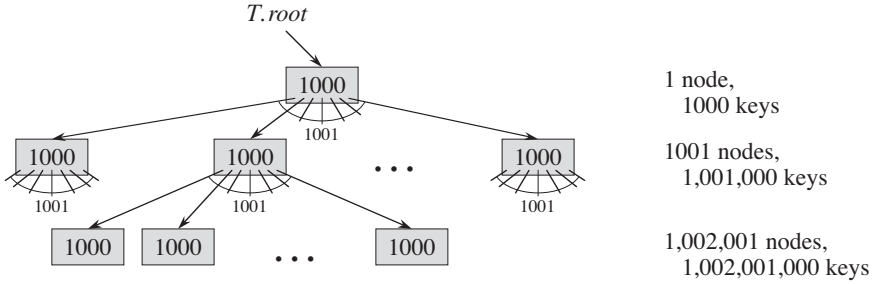


Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

18.1 Definition of B-trees

To keep things simple, we assume, as we have for binary search trees and red-black trees, that any "satellite information" associated with a key resides in the same node as the key. In practice, one might actually store with each key just a pointer to another disk page containing the satellite information for that key. The pseudocode in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. A common variant on a B-tree, known as a ***B*$^+$*-tree***, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A ***B-tree*** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:
   a. $x.n$, the number of keys currently stored in node x,
   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,
   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

   $$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$

4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:

   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

Theorem 18.1
If $n \ge 1$, then for any n-key B-tree T of height h and minimum degree $t \ge 2$,

$$h \le \log_t \frac{n+1}{2} .$$

Proof The root of a B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys. Thus, T, whose height is h, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. Thus, the

[2] Another common variant on a B-tree, known as a ***B*-tree***, requires each internal node to be at least $2/3$ full, rather than at least half full, as a B-tree requires.

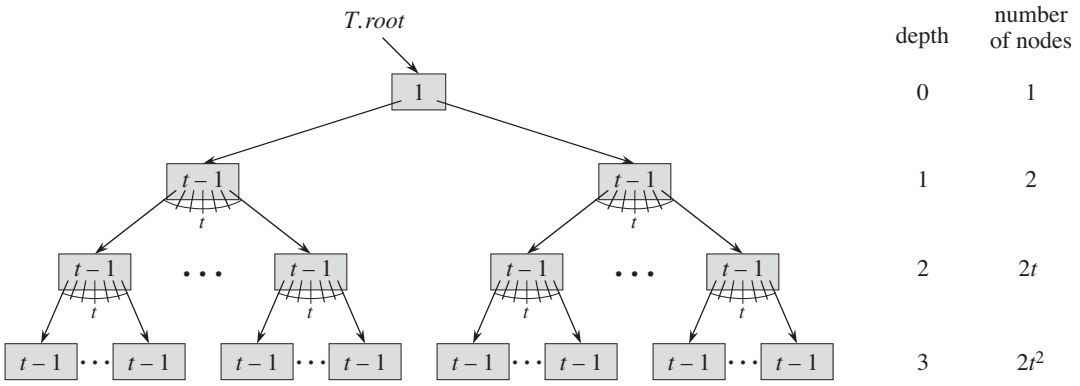


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

number n of keys satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \,. \end{aligned}$$

By simple algebra, we get $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ■

Here we see the power of B-trees, as compared with red-black trees. Although the height of the tree grows as $O(\lg n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because we usually have to access the disk to examine an arbitrary node in a tree, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1

Why don't we allow a minimum degree of $t = 1$?

18.1-2

For what values of t is the tree of Figure 18.1 a legal B-tree?

18.1-3
Show all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$.

18.1-4
As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

18.1-5
Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

In this section, we present the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two conventions:

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root; we do have to perform a DISK-WRITE of the root, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures we present are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

Searching a B-tree

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, we make a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, we make an $(x.n + 1)$-way branching decision.

B-TREE-SEARCH is a straightforward generalization of the TREE-SEARCH procedure defined for binary search trees. B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH$(T.root, k)$. If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that $y.key_i = k$. Otherwise, the procedure returns NIL.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

Using a linear-search procedure, lines 1–3 find the smallest index i such that $k \le x.key_i$, or else they set i to $x.n + 1$. Lines 4–5 check to see whether we have now discovered the key, returning if we have. Otherwise, lines 6–9 either terminate the search unsuccessfully (if x is a leaf) or recurse to search the appropriate subtree of x, after performing the necessary DISK-READ on that child.

Figure 18.1 illustrates the operation of B-TREE-SEARCH. The procedure examines the lightly shaded nodes during a search for the key R.

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. The B-TREE-SEARCH procedure therefore accesses $O(h) = O(\log_t n)$ disk pages, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $x.n < 2t$, the **while** loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

To build a B-tree T, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in $O(1)$ time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```
B-TREE-CREATE(T)
1  x = ALLOCATE-NODE()
2  x.leaf = TRUE
3  x.n = 0
4  DISK-WRITE(x)
5  T.root = x
```

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. As with binary search trees, we search for the leaf position at which to insert the new key. With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, we insert the new key into an existing leaf node. Since we cannot insert a key into a leaf node that is full, we introduce an operation that ***splits*** a full node y (having $2t-1$ keys) around its ***median key*** $y.key_t$ into two nodes having only $t-1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, we must split it before we can insert the new key, and thus we could end up splitting full nodes all the way up the tree.

As with a binary search tree, we can insert a key into a B-tree in a single pass down the tree from the root to a leaf. To do so, we do not wait to find out whether we will actually need to split a full node in order to do the insertion. Instead, as we travel down the tree searching for the position where the new key belongs, we split each full node we come to along the way (including the leaf itself). Thus whenever we want to split a full node y, we are assured that its parent is not full.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory) and an index i such that $x.c_i$ (also assumed to be in main memory) is a *full* child of x. The procedure then splits this child in two and adjusts x so that it has an additional child. To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD. The tree thus grows in height by one; splitting is the only means by which the tree grows.

Figure 18.5 illustrates this process. We split the full node $y = x.c_i$ about its median key S, which moves up into y's parent node x. Those keys in y that are greater than the median key move into a new node z, which becomes a new child of x.

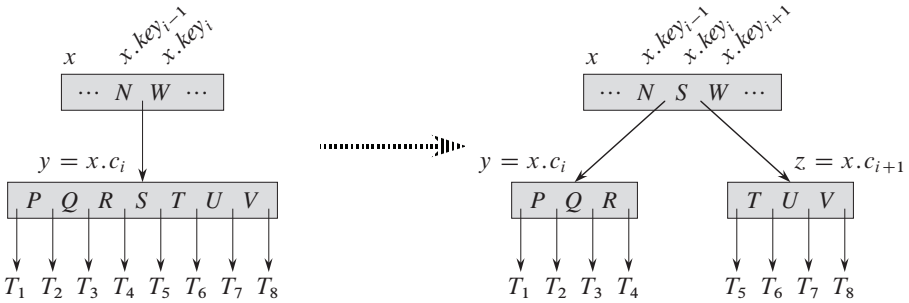


Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.

B-TREE-SPLIT-CHILD(x, i)

```
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD works by straightforward "cutting and pasting." Here, x is the node being split, and y is x's ith child (set in line 2). Node y originally has $2t$ children ($2t - 1$ keys) but is reduced to t children ($t - 1$ keys) by this operation. Node z takes the t largest children ($t - 1$ keys) from y, and z becomes a new child

of x, positioned just after y in x's table of children. The median key of y moves up to become the key in x that separates y and z.

Lines 1–9 create node z and give it the largest $t - 1$ keys and corresponding t children of y. Line 10 adjusts the key count for y. Finally, lines 11–17 insert z as a child of x, move the median key from y up to x in order to separate y from z, and adjust x's key count. Lines 18–20 write out all modified disk pages. The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the loops on lines 5–6 and 8–9. (The other loops run for $O(t)$ iterations.) The procedure performs $O(1)$ disk operations.

Inserting a key into a B-tree in a single pass down the tree

We insert a key k into a B-tree T of height h in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

Lines 3–9 handle the case in which the root node r is full: the root splits and a new node s (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree. Figure 18.6 illustrates this case. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. The procedure finishes by calling B-TREE-INSERT-NONFULL to insert key k into the tree rooted at the nonfull root node. B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD as necessary.

The auxiliary recursive procedure B-TREE-INSERT-NONFULL inserts key k into node x, which is assumed to be nonfull when the procedure is called. The operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.

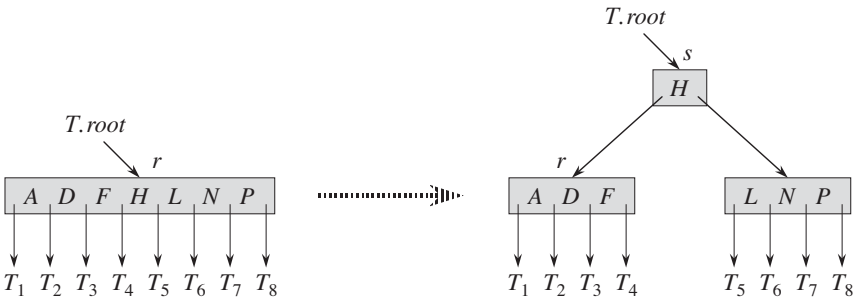


Figure 18.6 Splitting the root with $t = 4$. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

B-TREE-INSERT-NONFULL(x, k)

```
 1  i = x.n
 2  if x.leaf
 3       while i ≥ 1 and k < x.key_i
 4            x.key_{i+1} = x.key_i
 5            i = i − 1
 6       x.key_{i+1} = k
 7       x.n = x.n + 1
 8       DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10            i = i − 1
11       i = i + 1
12       DISK-READ(x.c_i)
13       if x.c_i.n == 2t − 1
14            B-TREE-SPLIT-CHILD(x, i)
15            if k > x.key_i
16                 i = i + 1
17       B-TREE-INSERT-NONFULL(x.c_i, k)
```

The B-TREE-INSERT-NONFULL procedure works as follows. Lines 3–8 handle the case in which x is a leaf node by inserting key k into x. If x is not a leaf node, then we must insert k into the appropriate leaf node in the subtree rooted at internal node x. In this case, lines 9–11 determine the child of x to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two nonfull children, and lines 15–16 determine which of the two children is now the

correct one to descend to. (Note that there is no need for a DISK-READ($x.c_i$) after line 16 increments i, since the recursion will descend in this case to a child that was just created by B-TREE-SPLIT-CHILD.) The net effect of lines 13–16 is thus to guarantee that the procedure never recurses to a full node. Line 17 then recurses to insert k into the appropriate subtree. Figure 18.7 illustrates the various cases of inserting into a B-tree.

For a B-tree of height h, B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(t \log_t n)$. Since B-TREE-INSERT-NONFULL is tail-recursive, we can alternatively implement it as a **while** loop, thereby demonstrating that the number of pages that need to be in main memory at any time is $O(1)$.

Exercises

18.2-1

Show the results of inserting the keys

$$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$$

in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

18.2-3

Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

18.2-4 ⋆

Suppose that we insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

18.2-5

Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

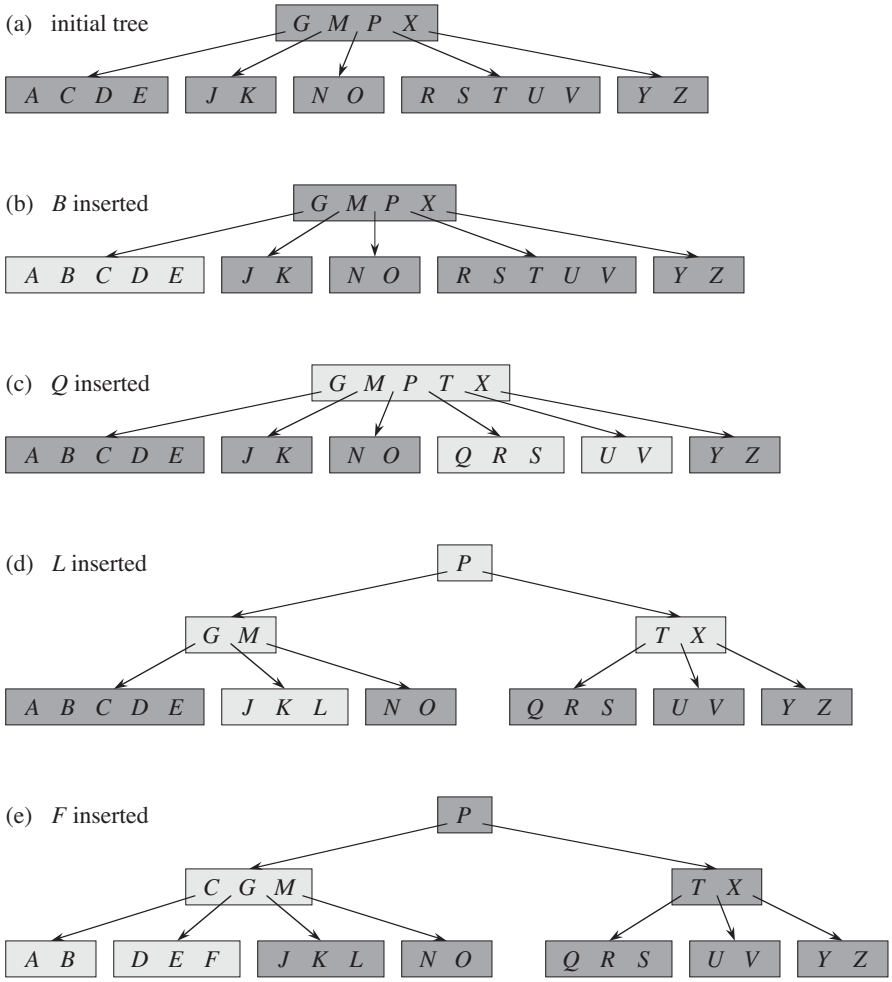


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. **(a)** The initial tree for this example. **(b)** The result of inserting B into the initial tree; this is a simple insertion into a leaf node. **(c)** The result of inserting Q into the previous tree. The node $RSTUV$ splits into two nodes containing RS and UV, the key T moves up to the root, and Q is inserted in the leftmost of the two halves (the RS node). **(d)** The result of inserting L into the previous tree. The root splits right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. **(e)** The result of inserting F into the previous tree. The node $ABCDE$ splits before F is inserted into the rightmost of the two halves (the DE node).

18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n.

18.2-7

Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

18.3 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children. As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x. We design this procedure to guarantee that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t. Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b on pages 501–502), then we delete x, and x's only child $x.c_1$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

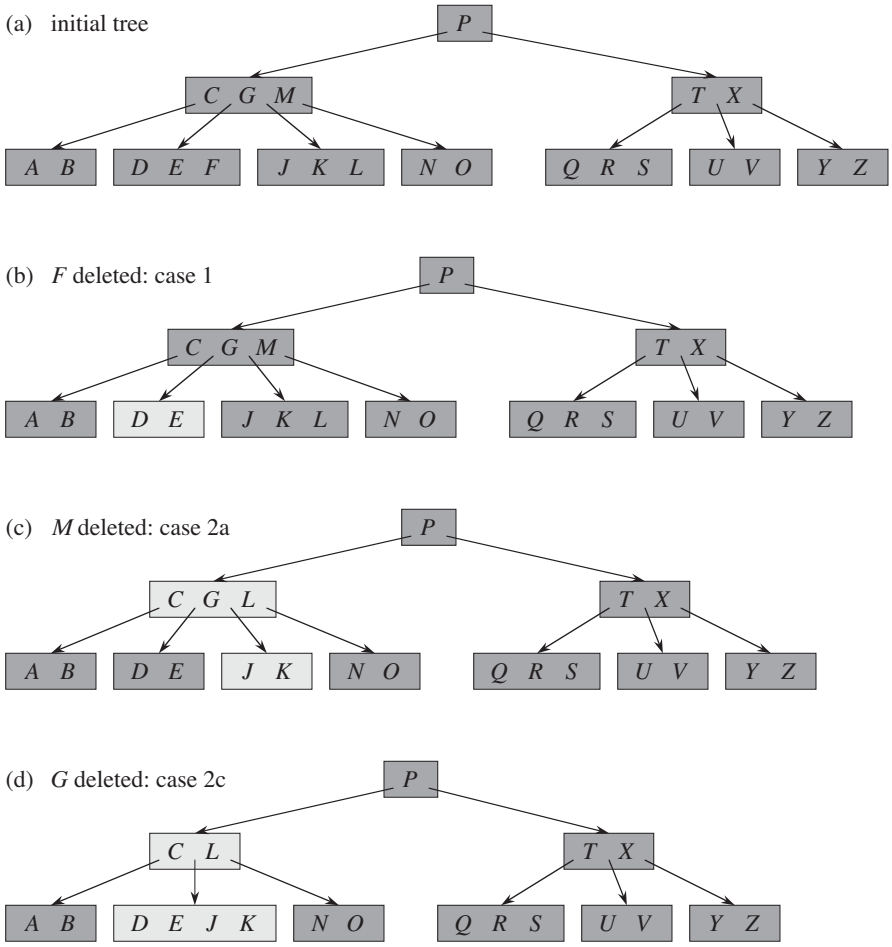


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M moves up to take M's position. **(d)** Deletion of G. This is case 2c: we push G down to make node $DEGJK$ and then delete G from this leaf (case 1).

We sketch how deletion works instead of presenting the pseudocode. Figure 18.8 illustrates the various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following:

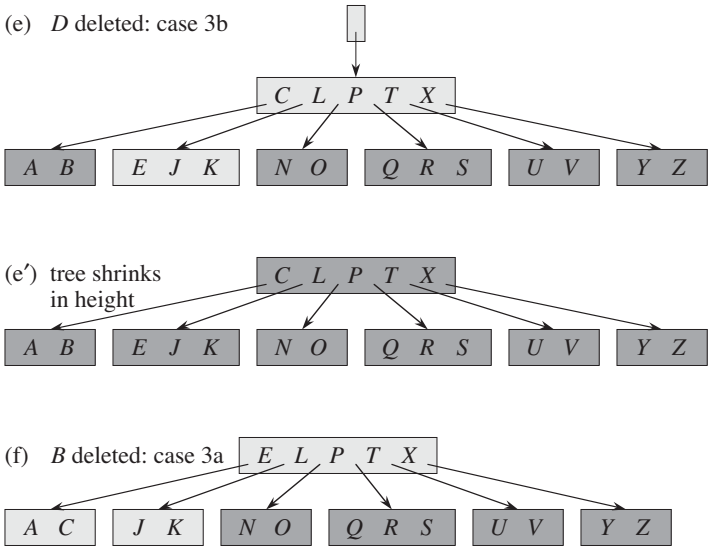


Figure 18.8, continued **(e)** Deletion of D. This is case 3b: the recursion cannot descend to node CL because it has only 2 keys, so we push P down and merge it with CL and TX to form $CLPTX$; then we delete D from a leaf (case 1). **(e′)** After (e), we delete the root and the tree shrinks in height by one. **(f)** Deletion of B. This is case 3a: C moves to fill B's position and E moves to fill C's position.

a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

c. Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t-1$ keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

a. If $x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.
b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, we may expect that in practice, deletion operations are most often used to delete keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h, since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log_t n)$.

Exercises

18.3-1
Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

18.3-2
Write pseudocode for B-TREE-DELETE.

Problems

18-1 Stacks on secondary storage
Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p, the top element is the $(p \bmod m)$th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.

To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)

Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

18-2 Joining and splitting 2-3-4 trees

The ***join*** operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.key < x.key < x''.key$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. In this problem, we investigate

how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$, where, for $i = 1, 2, \ldots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.

d. Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. (*Hint:* The costs for joining should telescope.)

Chapter notes

Knuth [211], Aho, Hopcroft, and Ullman [5], and Sedgewick [306] give further discussions of balanced-tree schemes and B-trees. Comer [74] provides a comprehensive survey of B-trees. Guibas and Sedgewick [155] discuss the relationships among various kinds of balanced-tree schemes, including red-black trees and 2-3-4 trees.

In 1970, J. E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4 trees, in which every internal node has either two or three children. Bayer and McCreight [35] introduced B-trees in 1972; they did not explain their choice of name.

Bender, Demaine, and Farach-Colton [40] studied how to make B-trees perform well in the presence of memory-hierarchy effects. Their ***cache-oblivious*** algorithms work efficiently without explicitly knowing the data transfer sizes within the memory hierarchy.

19 Fibonacci Heaps

The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a "mergeable heap." Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Mergeable heaps

A ***mergeable heap*** is any data structure that supports the following five operations, in which each element has a *key*:

MAKE-HEAP() creates and returns a new heap containing no elements.

INSERT(H, x) inserts element x, whose *key* has already been filled in, into heap H.

MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.

EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.[1]

DELETE(H, x) deletes element x from heap H.

---

[1] As mentioned in the introduction to Part V, our default mergeable heaps are mergeable min-heaps, and so the operations MINIMUM, EXTRACT-MIN, and DECREASE-KEY apply. Alternatively, we could define a ***mergeable max-heap*** with the operations MAXIMUM, EXTRACT-MAX, and INCREASE-KEY.

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Figure 19.1 Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n.

As the table in Figure 19.1 shows, if we don't need the UNION operation, ordinary binary heaps, as used in heapsort (Chapter 6), work fairly well. Operations other than UNION run in worst-case time $O(\lg n)$ on a binary heap. If we need to support the UNION operation, however, binary heaps perform poorly. By concatenating the two arrays that hold the binary heaps to be merged and then running BUILD-MIN-HEAP (see Section 6.3), the UNION operation takes $\Theta(n)$ time in the worst case.

Fibonacci heaps, on the other hand, have better asymptotic time bounds than binary heaps for the INSERT, UNION, and DECREASE-KEY operations, and they have the same asymptotic running times for the remaining operations. Note, however, that the running times for Fibonacci heaps in Figure 19.1 are amortized time bounds, not worst-case per-operation time bounds. The UNION operation takes only constant amortized time in a Fibonacci heap, which is significantly better than the linear worst-case time required in a binary heap (assuming, of course, that an amortized time bound suffices).

Fibonacci heaps in theory and practice

From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This situation arises in many applications. For example, some algorithms for graph problems may call DECREASE-KEY once per edge. For dense graphs, which have many edges, the $\Theta(1)$ amortized time of each call of DECREASE-KEY adds up to a big improvement over the $\Theta(\lg n)$ worst-case time of binary heaps. Fast algorithms for problems such as computing minimum spanning trees (Chapter 23) and finding single-source shortest paths (Chapter 24) make essential use of Fibonacci heaps.

From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications, except for certain applications that manage large amounts of data. Thus, Fibonacci heaps are predominantly of theoretical interest. If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.

Both binary heaps and Fibonacci heaps are inefficient in how they support the operation SEARCH; it can take a while to find an element with a given key. For this reason, operations such as DECREASE-KEY and DELETE that refer to a given element require a pointer to that element as part of their input. As in our discussion of priority queues in Section 6.5, when we use a mergeable heap in an application, we often store a handle to the corresponding application object in each mergeable-heap element, as well as a handle to the corresponding mergeable-heap element in each application object. The exact nature of these handles depends on the application and its implementation.

Like several other data structures that we have seen, Fibonacci heaps are based on rooted trees. We represent each element by a node within a tree, and each node has a *key* attribute. For the remainder of this chapter, we shall use the term "node" instead of "element." We shall also ignore issues of allocating nodes prior to insertion and freeing nodes following deletion, assuming instead that the code calling the heap procedures deals with these details.

Section 19.1 defines Fibonacci heaps, discusses how we represent them, and presents the potential function used for their amortized analysis. Section 19.2 shows how to implement the mergeable-heap operations and achieve the amortized time bounds shown in Figure 19.1. The remaining two operations, DECREASE-KEY and DELETE, form the focus of Section 19.3. Finally, Section 19.4 finishes a key part of the analysis and also explains the curious name of the data structure.

19.1 Structure of Fibonacci heaps

A ***Fibonacci heap*** is a collection of rooted trees that are ***min-heap ordered***. That is, each tree obeys the ***min-heap property***: the key of a node is greater than or equal to the key of its parent. Figure 19.2(a) shows an example of a Fibonacci heap.

As Figure 19.2(b) shows, each node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the ***child list*** of x. Each child y in a child list has pointers $y.left$ and $y.right$ that point to y's left and right siblings, respectively. If node y is an only child, then $y.left = y.right = y$. Siblings may appear in a child list in any order.

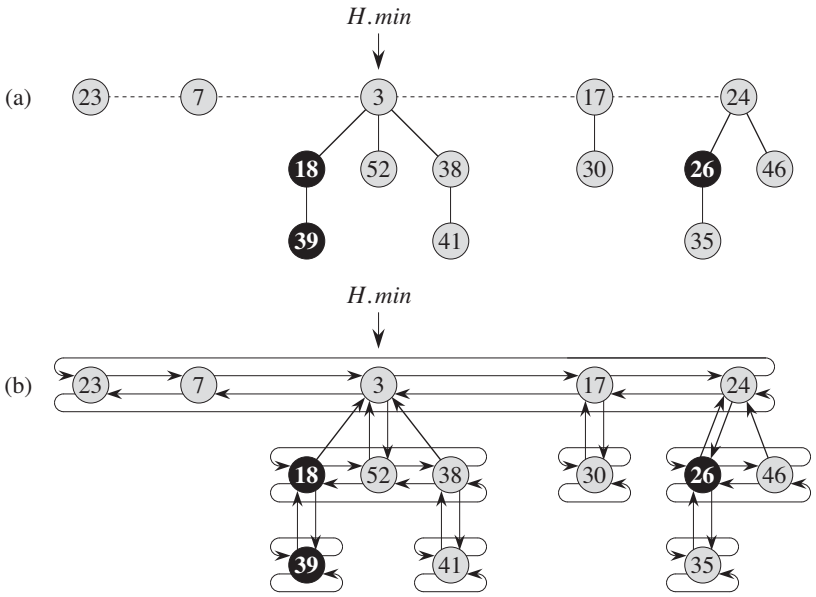


Figure 19.2 **(a)** A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. Black nodes are marked. The potential of this particular Fibonacci heap is $5 + 2 \cdot 3 = 11$. **(b)** A more complete representation showing pointers p (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows). The remaining figures in this chapter omit these details, since all the information shown here can be determined from what appears in part (a).

Circular, doubly linked lists (see Section 10.2) have two advantages for use in Fibonacci heaps. First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in $O(1)$ time. Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in $O(1)$ time. In the descriptions of Fibonacci heap operations, we shall refer to these operations informally, letting you fill in the details of their implementations if you wish.

Each node has two other attributes. We store the number of children in the child list of node x in $x.degree$. The boolean-valued attribute $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Until we look at the DECREASE-KEY operation in Section 19.3, we will just set all *mark* attributes to FALSE.

We access a given Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the ***minimum node*** of the Fibonacci

heap. If more than one root has a key with the minimum value, then any such root may serve as the minimum node. When a Fibonacci heap H is empty, $H.min$ is NIL.

The roots of all the trees in a Fibonacci heap are linked together using their *left* and *right* pointers into a circular, doubly linked list called the ***root list*** of the Fibonacci heap. The pointer $H.min$ thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list.

We rely on one other attribute for a Fibonacci heap H: $H.n$, the number of nodes currently in H.

Potential function

As mentioned, we shall use the potential method of Section 17.3 to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. We then define the potential $\Phi(H)$ of Fibonacci heap H by

$$\Phi(H) = t(H) + 2\,m(H)\,. \tag{19.1}$$

(We will gain some intuition for this potential function in Section 19.3.) For example, the potential of the Fibonacci heap shown in Figure 19.2 is $5 + 2 \cdot 3 = 11$. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter.

We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and by equation (19.1), the potential is nonnegative at all subsequent times. From equation (17.3), an upper bound on the total amortized cost provides an upper bound on the total actual cost for the sequence of operations.

Maximum degree

The amortized analyses we shall perform in the remaining sections of this chapter assume that we know an upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap. We won't prove it, but when only the mergeable-heap operations are supported, $D(n) \leq \lfloor \lg n \rfloor$. (Problem 19-2(d) asks you to prove this property.) In Sections 19.3 and 19.4, we shall show that when we support DECREASE-KEY and DELETE as well, $D(n) = O(\lg n)$.

19.2 Mergeable-heap operations

The mergeable-heap operations on Fibonacci heaps delay work as long as possible. The various operations have performance trade-offs. For example, we insert a node by adding it to the root list, which takes just constant time. If we were to start with an empty Fibonacci heap and then insert k nodes, the Fibonacci heap would consist of just a root list of k nodes. The trade-off is that if we then perform an EXTRACT-MIN operation on Fibonacci heap H, after removing the node that $H.min$ points to, we would have to look through each of the remaining $k-1$ nodes in the root list to find the new minimum node. As long as we have to go through the entire root list during the EXTRACT-MIN operation, we also consolidate nodes into min-heap-ordered trees to reduce the size of the root list. We shall see that, no matter what the root list looks like before a EXTRACT-MIN operation, afterward each node in the root list has a degree that is unique within the root list, which leads to a root list of size at most $D(n)+1$.

Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $H.n = 0$ and $H.min = \text{NIL}$; there are no trees in H. Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $x.key$ has already been filled in.

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```

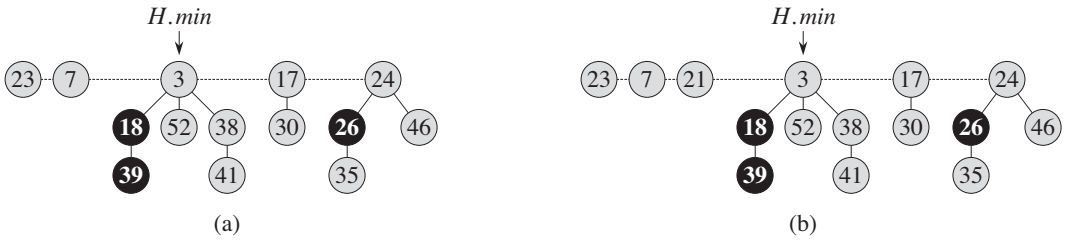


Figure 19.3 Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap H. **(b)** Fibonacci heap H after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

Lines 1–4 initialize some of the structural attributes of node x. Line 5 tests to see whether Fibonacci heap H is empty. If it is, then lines 6–7 make x be the only node in H's root list and set $H.min$ to point to x. Otherwise, lines 8–10 insert x into H's root list and update $H.min$ if necessary. Finally, line 11 increments $H.n$ to reflect the addition of the new node. Figure 19.3 shows a node with key 21 inserted into the Fibonacci heap of Figure 19.2.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is

$$((t(H) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) = 1\ .$$

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer $H.min$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process. It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node. Afterward, the objects representing H_1 and H_2 will never be used again.

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5       H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

Lines 1–3 concatenate the root lists of H_1 and H_2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets $H.n$ to the total number of nodes. Line 7 returns the resulting Fibonacci heap H. As in the FIB-HEAP-INSERT procedure, all roots remain roots.

The change in potential is

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) + (t(H_2) + 2\,m(H_2))) \\ &= 0\,,\end{aligned}$$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section. It is also where the delayed work of consolidating trees in the root list finally occurs. The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also calls the auxiliary procedure CONSOLIDATE, which we shall see shortly.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z == z.right
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

As Figure 19.4 illustrates, FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

We start in line 1 by saving a pointer z to the minimum node; the procedure returns this pointer at the end. If z is NIL, then Fibonacci heap H is already empty and we are done. Otherwise, we delete node z from H by making all of z's children roots of H in lines 3–5 (putting them into the root list) and removing z from the root list in line 6. If z is its own right sibling after line 6, then z was the only node on the root list and it had no children, so all that remains is to make the Fibonacci heap empty in line 8 before returning z. Otherwise, we set the pointer $H.min$ into the root list to point to a root other than z (in this case, z's right sibling), which is not necessarily going to be the new minimum node when FIB-HEAP-EXTRACT-MIN is done. Figure 19.4(b) shows the Fibonacci heap of Figure 19.4(a) after executing line 9.

The next step, in which we reduce the number of trees in the Fibonacci heap, is ***consolidating*** the root list of H, which the call CONSOLIDATE(H) accomplishes. Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value:

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
2. ***Link*** y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on y.

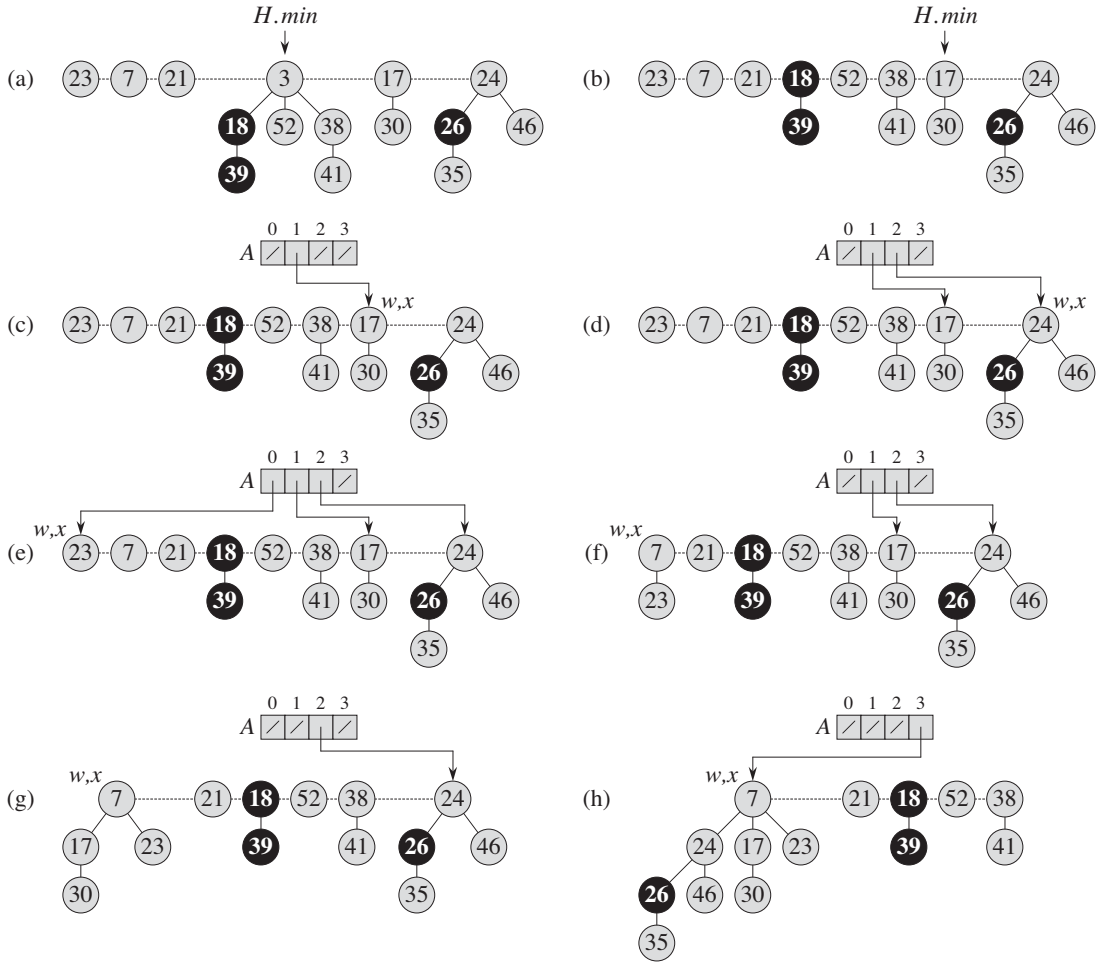


Figure 19.4 The action of FIB-HEAP-EXTRACT-MIN. **(a)** A Fibonacci heap H. **(b)** The situation after removing the minimum node z from the root list and adding its children to the root list. **(c)–(e)** The array A and the trees after each of the first three iterations of the **for** loop of lines 4–14 of the procedure CONSOLIDATE. The procedure processes the root list by starting at the node pointed to by $H.min$ and following *right* pointers. Each part shows the values of w and x at the end of an iteration. **(f)–(h)** The next iteration of the **for** loop, with the values of w and x shown at the end of each iteration of the **while** loop of lines 7–13. Part (f) shows the situation after the first time through the **while** loop. The node with key 23 has been linked to the node with key 7, which x now points to. In part (g), the node with key 17 has been linked to the node with key 7, which x still points to. In part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by $A[3]$, at the end of the **for** loop iteration, $A[3]$ is set to point to the root of the resulting tree.

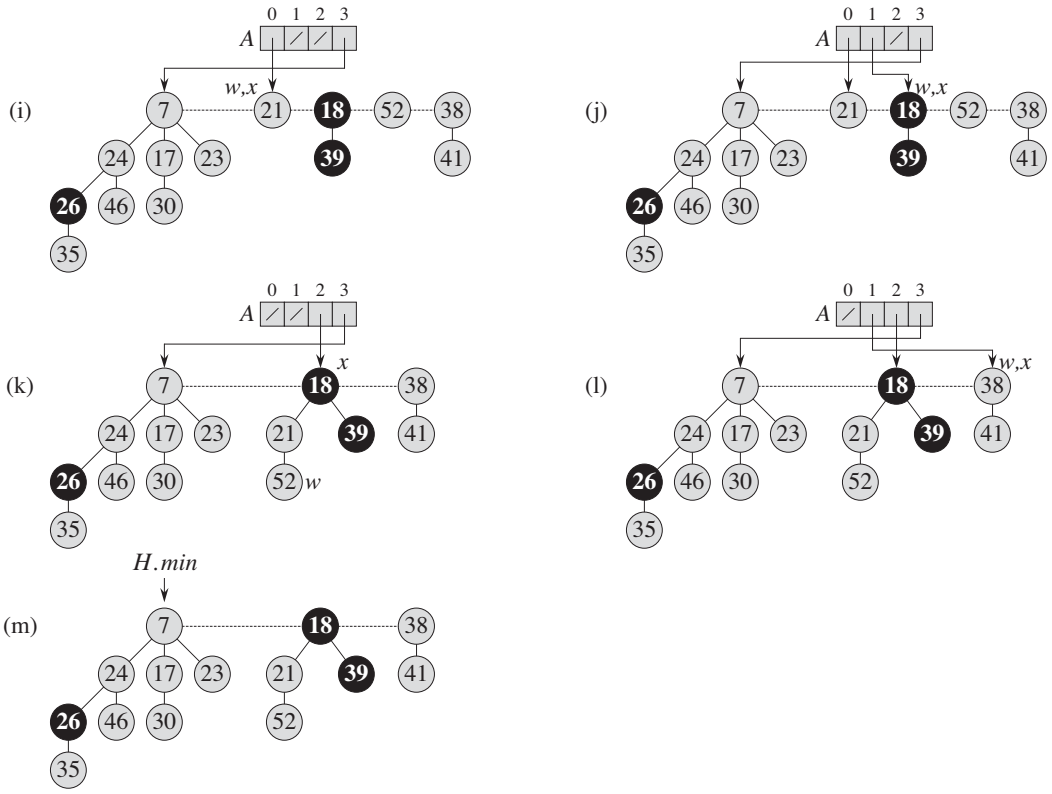


Figure 19.4, continued **(i)–(l)** The situation after each of the next four iterations of the **for** loop. **(m)** Fibonacci heap H after reconstructing the root list from the array A and determining the new $H.min$ pointer.

The procedure CONSOLIDATE uses an auxiliary array $A[0\,..\,D(H.n)]$ to keep track of roots according to their degrees. If $A[i] = y$, then y is currently a root with $y.degree = i$. Of course, in order to allocate the array we have to know how to calculate the upper bound $D(H.n)$ on the maximum degree, but we will see how to do so in Section 19.4.

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]          // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

In detail, the CONSOLIDATE procedure works as follows. Lines 1–3 allocate and initialize the array A by making each entry NIL. The **for** loop of lines 4–14 processes each root w in the root list. As we link roots together, w may be linked to some other node and no longer be a root. Nevertheless, w is always in a tree rooted at some node x, which may or may not be w itself. Because we want at most one root with each degree, we look in the array A to see whether it contains a root y with the same degree as x. If it does, then we link the roots x and y but guaranteeing that x remains a root after linking. That is, we link y to x after first exchanging the pointers to the two roots if y's key is smaller than x's key. After we link y to x, the degree of x has increased by 1, and so we continue this process, linking x and another root whose degree equals x's new degree, until no other root

that we have processed has the same degree as x. We then set the appropriate entry of A to point to x, so that as we process roots later on, we have recorded that x is the unique root of its degree that we have already processed. When this **for** loop terminates, at most one root of each degree will remain, and the array A will point to each remaining root.

The **while** loop of lines 7–13 repeatedly links the root x of the tree containing node w to another tree whose root has the same degree as x, until no other root has the same degree. This **while** loop maintains the following invariant:

At the start of each iteration of the **while** loop, $d = x.degree$.

We use this loop invariant as follows:

Initialization: Line 6 ensures that the loop invariant holds the first time we enter the loop.

Maintenance: In each iteration of the **while** loop, $A[d]$ points to some root y. Because $d = x.degree = y.degree$, we want to link x and y. Whichever of x and y has the smaller key becomes the parent of the other as a result of the link operation, and so lines 9–10 exchange the pointers to x and y if necessary. Next, we link y to x by the call FIB-HEAP-LINK(H, y, x) in line 11. This call increments $x.degree$ but leaves $y.degree$ as d. Node y is no longer a root, and so line 12 removes the pointer to it in array A. Because the call of FIB-HEAP-LINK increments the value of $x.degree$, line 13 restores the invariant that $d = x.degree$.

Termination: We repeat the **while** loop until $A[d] =$ NIL, in which case there is no other root with the same degree as x.

After the **while** loop terminates, we set $A[d]$ to x in line 14 and perform the next iteration of the **for** loop.

Figures 19.4(c)–(e) show the array A and the resulting trees after the first three iterations of the **for** loop of lines 4–14. In the next iteration of the **for** loop, three links occur; their results are shown in Figures 19.4(f)–(h). Figures 19.4(i)–(l) show the result of the next four iterations of the **for** loop.

All that remains is to clean up. Once the **for** loop of lines 4–14 completes, line 15 empties the root list, and lines 16–23 reconstruct it from the array A. The resulting Fibonacci heap appears in Figure 19.4(m). After consolidating the root list, FIB-HEAP-EXTRACT-MIN finishes up by decrementing $H.n$ in line 11 and returning a pointer to the deleted node z in line 12.

We are now ready to show that the amortized cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$. Let H denote the Fibonacci heap just prior to the FIB-HEAP-EXTRACT-MIN operation.

We start by accounting for the actual cost of extracting the minimum node. An $O(D(n))$ contribution comes from FIB-HEAP-EXTRACT-MIN processing at

most $D(n)$ children of the minimum node and from the work in lines 2–3 and 16–23 of CONSOLIDATE. It remains to analyze the contribution from the **for** loop of lines 4–14 in CONSOLIDATE, for which we use an aggregate analysis. The size of the root list upon calling CONSOLIDATE is at most $D(n) + t(H) - 1$, since it consists of the original $t(H)$ root-list nodes, minus the extracted root node, plus the children of the extracted node, which number at most $D(n)$. Within a given iteration of the **for** loop of lines 4–14, the number of iterations of the **while** loop of lines 7–13 depends on the root list. But we know that every time through the **while** loop, one of the roots is linked to another, and thus the total number of iterations of the **while** loop over all iterations of the **for** loop is at most the number of roots in the root list. Hence, the total amount of work performed in the **for** loop is at most proportional to $D(n) + t(H)$. Thus, the total actual work in extracting the minimum node is $O(D(n) + t(H))$.

The potential before extracting the minimum node is $t(H) + 2\,m(H)$, and the potential afterward is at most $(D(n) + 1) + 2\,m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation. The amortized cost is thus at most

$$\begin{aligned} & O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) \\ & \quad = \quad O(D(n)) + O(t(H)) - t(H) \\ & \quad = \quad O(D(n)) \,, \end{aligned}$$

since we can scale up the units of potential to dominate the constant hidden in $O(t(H))$. Intuitively, the cost of performing each link is paid for by the reduction in potential due to the link's reducing the number of roots by one. We shall see in Section 19.4 that $D(n) = O(\lg n)$, so that the amortized cost of extracting the minimum node is $O(\lg n)$.

Exercises

19.2-1

Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in Figure 19.4(m).

19.3 Decreasing a key and deleting a node

In this section, we show how to decrease the key of a node in a Fibonacci heap in $O(1)$ amortized time and how to delete any node from an n-node Fibonacci heap in $O(D(n))$ amortized time. In Section 19.4, we will show that the maxi-

mum degree $D(n)$ is $O(\lg n)$, which will imply that FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE run in $O(\lg n)$ amortized time.

Decreasing a key

In the following pseudocode for the operation FIB-HEAP-DECREASE-KEY, we assume as before that removing a node from a linked list does not change any of the structural attributes in the removed node.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2       error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6       CUT(H, x, y)
7       CASCADING-CUT(H, y)
8  if x.key < H.min.key
9       H.min = x

CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3       if y.mark == FALSE
4            y.mark = TRUE
5       else CUT(H, y, z)
6            CASCADING-CUT(H, z)
```

The FIB-HEAP-DECREASE-KEY procedure works as follows. Lines 1–3 ensure that the new key is no greater than the current key of x and then assign the new key to x. If x is a root or if $x.key \geq y.key$, where y is x's parent, then no structural changes need occur, since min-heap order has not been violated. Lines 4–5 test for this condition.

If min-heap order has been violated, many changes may occur. We start by ***cutting*** x in line 6. The CUT procedure "cuts" the link between x and its parent y, making x a root.

We use the *mark* attributes to obtain the desired time bounds. They record a little piece of the history of each node. Suppose that the following events have happened to node x:

1. at some time, x was a root,
2. then x was linked to (made the child of) another node,
3. then two children of x were removed by cuts.

As soon as the second child has been lost, we cut x from its parent, making it a new root. The attribute $x.mark$ is TRUE if steps 1 and 2 have occurred and one child of x has been cut. The CUT procedure, therefore, clears $x.mark$ in line 4, since it performs step 1. (We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$: node y is being linked to another node, and so step 2 is being performed. The next time a child of y is cut, $y.mark$ will be set to TRUE.)

We are not yet done, because x might be the second child cut from its parent y since the time that y was linked to another node. Therefore, line 7 of FIB-HEAP-DECREASE-KEY attempts to perform a ***cascading-cut*** operation on y. If y is a root, then the test in line 2 of CASCADING-CUT causes the procedure to just return. If y is unmarked, the procedure marks it in line 4, since its first child has just been cut, and returns. If y is marked, however, it has just lost its second child; y is cut in line 5, and CASCADING-CUT calls itself recursively in line 6 on y's parent z. The CASCADING-CUT procedure recurses its way up the tree until it finds either a root or an unmarked node.

Once all the cascading cuts have occurred, lines 8–9 of FIB-HEAP-DECREASE-KEY finish up by updating $H.min$ if necessary. The only node whose key changed was the node x whose key decreased. Thus, the new minimum node is either the original minimum node or node x.

Figure 19.5 shows the execution of two calls of FIB-HEAP-DECREASE-KEY, starting with the Fibonacci heap shown in Figure 19.5(a). The first call, shown in Figure 19.5(b), involves no cascading cuts. The second call, shown in Figures 19.5(c)–(e), invokes two cascading cuts.

We shall now show that the amortized cost of FIB-HEAP-DECREASE-KEY is only $O(1)$. We start by determining its actual cost. The FIB-HEAP-DECREASE-KEY procedure takes $O(1)$ time, plus the time to perform the cascading cuts. Suppose that a given invocation of FIB-HEAP-DECREASE-KEY results in c calls of CASCADING-CUT (the call made from line 7 of FIB-HEAP-DECREASE-KEY followed by $c - 1$ recursive calls of CASCADING-CUT). Each call of CASCADING-CUT takes $O(1)$ time exclusive of recursive calls. Thus, the actual cost of FIB-HEAP-DECREASE-KEY, including all recursive calls, is $O(c)$.

We next compute the change in potential. Let H denote the Fibonacci heap just prior to the FIB-HEAP-DECREASE-KEY operation. The call to CUT in line 6 of

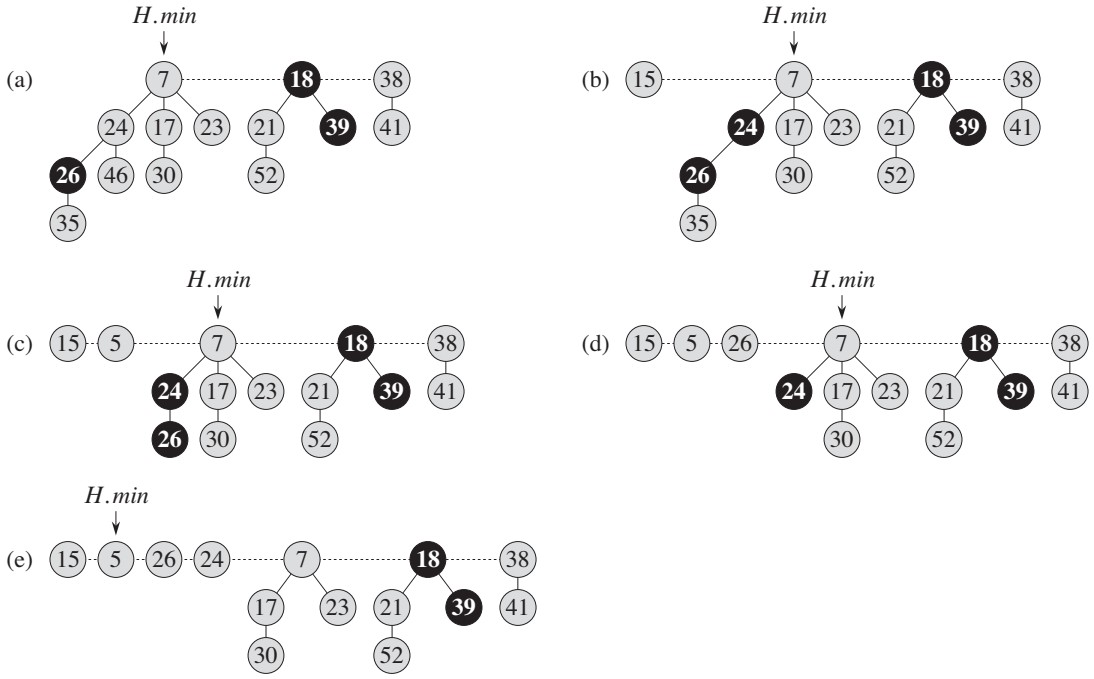


Figure 19.5 Two calls of FIB-HEAP-DECREASE-KEY. **(a)** The initial Fibonacci heap. **(b)** The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. **(c)–(e)** The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) Part (e) shows the result of the FIB-HEAP-DECREASE-KEY operation, with $H.min$ pointing to the new minimum node.

FIB-HEAP-DECREASE-KEY creates a new tree rooted at node x and clears x's mark bit (which may have already been FALSE). Each call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. Afterward, the Fibonacci heap contains $t(H)+c$ trees (the original $t(H)$ trees, $c-1$ trees produced by cascading cuts, and the tree rooted at x) and at most $m(H)-c+2$ marked nodes ($c-1$ were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node). The change in potential is therefore at most

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2\,m(H))=4-c\ .$$

Thus, the amortized cost of FIB-HEAP-DECREASE-KEY is at most

$$O(c) + 4 - c = O(1) ,$$

since we can scale up the units of potential to dominate the constant hidden in $O(c)$.

You can now see why we defined the potential function to include a term that is twice the number of marked nodes. When a marked node y is cut by a cascading cut, its mark bit is cleared, which reduces the potential by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y becoming a root.

Deleting a node

The following pseudocode deletes a node from an n-node Fibonacci heap in $O(D(n))$ amortized time. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE makes x become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-\infty$. The FIB-HEAP-EXTRACT-MIN procedure then removes node x from the Fibonacci heap. The amortized time of FIB-HEAP-DELETE is the sum of the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY and the $O(D(n))$ amortized time of FIB-HEAP-EXTRACT-MIN. Since we shall see in Section 19.4 that $D(n) = O(\lg n)$, the amortized time of FIB-HEAP-DELETE is $O(\lg n)$.

Exercises

19.3-1

Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a marked root. Argue that it doesn't matter to the analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.

19.3-2

Justify the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY as an average cost per operation by using aggregate analysis.

19.4 Bounding the maximum degree

To prove that the amortized time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE is $O(\lg n)$, we must show that the upper bound $D(n)$ on the degree of any node of an n-node Fibonacci heap is $O(\lg n)$. In particular, we shall show that $D(n) \leq \lfloor \log_\phi n \rfloor$, where ϕ is the golden ratio, defined in equation (3.24) as

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots\ .$$

The key to the analysis is as follows. For each node x within a Fibonacci heap, define size(x) to be the number of nodes, including x itself, in the subtree rooted at x. (Note that x need not be in the root list—it can be any node at all.) We shall show that size(x) is exponential in $x.degree$. Bear in mind that $x.degree$ is always maintained as an accurate count of the degree of x.

Lemma 19.1
Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, \ldots, k$.

Proof Obviously, $y_1.degree \geq 0$.

For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, and so we must have had $x.degree \geq i - 1$. Because node y_i is linked to x (by CONSOLIDATE) only if $x.degree = y_i.degree$, we must have also had $y_i.degree \geq i - 1$ at that time. Since then, node y_i has lost at most one child, since it would have been cut from x (by CASCADING-CUT) if it had lost two children. We conclude that $y_i.degree \geq i - 2$. ∎

We finally come to the part of the analysis that explains the name "Fibonacci heaps." Recall from Section 3.2 that for $k = 0, 1, 2, \ldots$, the kth Fibonacci number is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k = 0\,, \\ 1 & \text{if } k = 1\,, \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2\,. \end{cases}$$

The following lemma gives another way to express F_k.

Lemma 19.2
For all integers $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \ .$$

Proof The proof is by induction on k. When $k = 0$,

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= F_2 \ . \end{aligned}$$

We now assume the inductive hypothesis that $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, and we have

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i \ . \end{aligned}$$

■

Lemma 19.3
For all integers $k \geq 0$, the $(k+2)$nd Fibonacci number satisfies $F_{k+2} \geq \phi^k$.

Proof The proof is by induction on k. The base cases are for $k = 0$ and $k = 1$. When $k = 0$ we have $F_2 = 1 = \phi^0$, and when $k = 1$ we have $F_3 = 2 > 1.619 > \phi^1$. The inductive step is for $k \geq 2$, and we assume that $F_{i+2} > \phi^i$ for $i = 0, 1, \ldots, k-1$. Recall that ϕ is the positive root of equation (3.23), $x^2 = x+1$. Thus, we have

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geq \phi^{k-1} + \phi^{k-2} && \text{(by the inductive hypothesis)} \\ &= \phi^{k-2}(\phi + 1) \\ &= \phi^{k-2} \cdot \phi^2 && \text{(by equation (3.23))} \\ &= \phi^k \ . \end{aligned}$$

■

The following lemma and its corollary complete the analysis.

Lemma 19.4
Let x be any node in a Fibonacci heap, and let $k = x.degree$. Then $\text{size}(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2$.

Proof Let s_k denote the minimum possible size of any node of degree k in any Fibonacci heap. Trivially, $s_0 = 1$ and $s_1 = 2$. The number s_k is at most $\text{size}(x)$ and, because adding children to a node cannot decrease the node's size, the value of s_k increases monotonically with k. Consider some node z, in any Fibonacci heap, such that $z.degree = k$ and $\text{size}(z) = s_k$. Because $s_k \leq \text{size}(x)$, we compute a lower bound on $\text{size}(x)$ by computing a lower bound on s_k. As in Lemma 19.1, let $y_1, y_2, \ldots, y_k$ denote the children of z in the order in which they were linked to z. To bound s_k, we count one for z itself and one for the first child y_1 (for which $\text{size}(y_1) \geq 1$), giving

$$\begin{aligned}
\text{size}(x) &\geq s_k \\
&\geq 2 + \sum_{i=2}^{k} s_{y_i.degree} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2} ,
\end{aligned}$$

where the last line follows from Lemma 19.1 (so that $y_i.degree \geq i - 2$) and the monotonicity of s_k (so that $s_{y_i.degree} \geq s_{i-2}$).

We now show by induction on k that $s_k \geq F_{k+2}$ for all nonnegative integers k. The bases, for $k = 0$ and $k = 1$, are trivial. For the inductive step, we assume that $k \geq 2$ and that $s_i \geq F_{i+2}$ for $i = 0, 1, \ldots, k - 1$. We have

$$\begin{aligned}
s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} F_i \\
&= 1 + \sum_{i=0}^{k} F_i \\
&= F_{k+2} && \text{(by Lemma 19.2)} \\
&\geq \phi^k && \text{(by Lemma 19.3) .}
\end{aligned}$$

Thus, we have shown that $\text{size}(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

Corollary 19.5
The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\lg n)$.

Proof Let x be any node in an n-node Fibonacci heap, and let $k = x.degree$. By Lemma 19.4, we have $n \geq \text{size}(x) \geq \phi^k$. Taking base-$\phi$ logarithms gives us $k \leq \log_\phi n$. (In fact, because k is an integer, $k \leq \lfloor \log_\phi n \rfloor$.) The maximum degree $D(n)$ of any node is thus $O(\lg n)$. ■

Exercises

19.4-1
Professor Pinocchio claims that the height of an n-node Fibonacci heap is $O(\lg n)$. Show that the professor is mistaken by exhibiting, for any positive integer n, a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

19.4-2
Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. (The rule in Section 19.3 uses $k = 2$.) For what values of k is $D(n) = O(\lg n)$?

Problems

19-1 Alternative implementation of deletion
Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE procedure, claiming that it runs faster when the node being deleted is not the node pointed to by $H.min$.

```
PISANO-DELETE(H, x)
1  if x == H.min
2      FIB-HEAP-EXTRACT-MIN(H)
3  else y = x.p
4      if y ≠ NIL
5          CUT(H, x, y)
6          CASCADING-CUT(H, y)
7      add x's child list to the root list of H
8      remove x from the root list of H
```

a. The professor's claim that this procedure runs faster is based partly on the assumption that line 7 can be performed in $O(1)$ actual time. What is wrong with this assumption?

b. Give a good upper bound on the actual time of PISANO-DELETE when x is not $H.min$. Your bound should be in terms of $x.degree$ and the number c of calls to the CASCADING-CUT procedure.

c. Suppose that we call PISANO-DELETE(H, x), and let H' be the Fibonacci heap that results. Assuming that node x is not a root, bound the potential of H' in terms of $x.degree$, c, $t(H)$, and $m(H)$.

d. Conclude that the amortized time for PISANO-DELETE is asymptotically no better than for FIB-HEAP-DELETE, even when $x \neq H.min$.

19-2 Binomial trees and binomial heaps

The ***binomial tree*** B_k is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.6(a), the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together so that the root of one is the leftmost child of the root of the other. Figure 19.6(b) shows the binomial trees B_0 through B_4.

a. Show that for the binomial tree B_k,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and
4. the root has degree k, which is greater than that of any other node; moreover, as Figure 19.6(c) shows, if we number the children of the root from left to right by $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i.

A ***binomial heap*** H is a set of binomial trees that satisfies the following properties:

1. Each node has a *key* (like a Fibonacci heap).
2. Each binomial tree in H obeys the min-heap property.
3. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

b. Suppose that a binomial heap H has a total of n nodes. Discuss the relationship between the binomial trees that H contains and the binary representation of n. Conclude that H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

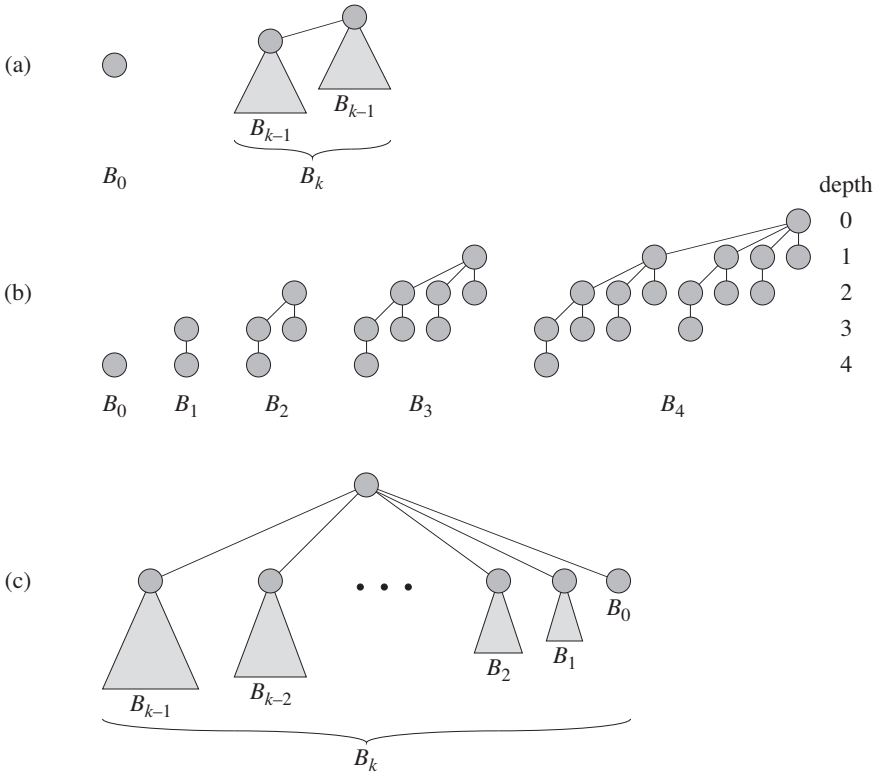


Figure 19.6 **(a)** The recursive definition of the binomial tree B_k. Triangles represent rooted subtrees. **(b)** The binomial trees B_0 through B_4. Node depths in B_4 are shown. **(c)** Another way of looking at the binomial tree B_k.

Suppose that we represent a binomial heap as follows. The left-child, right-sibling scheme of Section 10.4 represents each binomial tree within a binomial heap. Each node contains its key; pointers to its parent, to its leftmost child, and to the sibling immediately to its right (these pointers are NIL when appropriate); and its degree (as in Fibonacci heaps, how many children it has). The roots form a singly linked root list, ordered by the degrees of the roots (from low to high), and we access the binomial heap by a pointer to the first node on the root list.

c. Complete the description of how to represent a binomial heap (i.e., name the attributes, describe when attributes have the value NIL, and define how the root list is organized), and show how to implement the same seven operations on binomial heaps as this chapter implemented on Fibonacci heaps. Each operation should run in $O(\lg n)$ worst-case time, where n is the number of nodes in

the binomial heap (or in the case of the UNION operation, in the two binomial heaps that are being united). The MAKE-HEAP operation should take constant time.

d. Suppose that we were to implement only the mergeable-heap operations on a Fibonacci heap (i.e., we do not implement the DECREASE-KEY or DELETE operations). How would the trees in a Fibonacci heap resemble those in a binomial heap? How would they differ? Show that the maximum degree in an n-node Fibonacci heap would be at most $\lfloor \lg n \rfloor$.

e. Professor McGee has devised a new data structure based on Fibonacci heaps. A McGee heap has the same structure as a Fibonacci heap and supports just the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union consolidate the root list as their last step. What are the worst-case running times of operations on McGee heaps?

19-3 More Fibonacci-heap operations

We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

a. The operation FIB-HEAP-CHANGE-KEY(H, x, k) changes the key of node x to the value k. Give an efficient implementation of FIB-HEAP-CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is greater than, less than, or equal to $x.key$.

b. Give an efficient implementation of FIB-HEAP-PRUNE(H, r), which deletes $q = \min(r, H.n)$ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (*Hint:* You may need to modify the data structure and potential function.)

19-4 2-3-4 heaps

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement ***2-3-4 heaps***, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf x stores exactly one key in the attribute $x.key$. The keys in the leaves may appear in any order. Each internal node x contains a value $x.small$ that is equal to the smallest key stored in any leaf in the subtree rooted at x. The root r contains an attribute $r.height$ that gives the height of the

tree. Finally, 2-3-4 heaps are designed to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. In parts (a)–(e), each operation should run in $O(\lg n)$ time on a 2-3-4 heap with n elements. The UNION operation in part (f) should run in $O(\lg n)$ time, where n is the number of elements in the two input heaps.

a. MINIMUM, which returns a pointer to the leaf with the smallest key.

b. DECREASE-KEY, which decreases the key of a given leaf x to a given value $k \leq x.key$.

c. INSERT, which inserts leaf x with key k.

d. DELETE, which deletes a given leaf x.

e. EXTRACT-MIN, which extracts the leaf with the smallest key.

f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

Chapter notes

Fredman and Tarjan [114] introduced Fibonacci heaps. Their paper also describes the application of Fibonacci heaps to the problems of single-source shortest paths, all-pairs shortest paths, weighted bipartite matching, and the minimum-spanning-tree problem.

Subsequently, Driscoll, Gabow, Shrairman, and Tarjan [96] developed "relaxed heaps" as an alternative to Fibonacci heaps. They devised two varieties of relaxed heaps. One gives the same amortized time bounds as Fibonacci heaps. The other allows DECREASE-KEY to run in $O(1)$ worst-case (not amortized) time and EXTRACT-MIN and DELETE to run in $O(\lg n)$ worst-case time. Relaxed heaps also have some advantages over Fibonacci heaps in parallel algorithms.

See also the chapter notes for Chapter 6 for other data structures that support fast DECREASE-KEY operations when the sequence of values returned by EXTRACT-MIN calls are monotonically increasing over time and the data are integers in a specific range.

20 van Emde Boas Trees

In previous chapters, we saw data structures that support the operations of a priority queue—binary heaps in Chapter 6, red-black trees in Chapter 13,[1] and Fibonacci heaps in Chapter 19. In each of these data structures, at least one important operation took $O(\lg n)$ time, either worst case or amortized. In fact, because each of these data structures bases its decisions on comparing keys, the $\Omega(n \lg n)$ lower bound for sorting in Section 8.1 tells us that at least one operation will have to take $\Omega(\lg n)$ time. Why? If we could perform both the INSERT and EXTRACT-MIN operations in $o(\lg n)$ time, then we could sort n keys in $o(n \lg n)$ time by first performing n INSERT operations, followed by n EXTRACT-MIN operations.

We saw in Chapter 8, however, that sometimes we can exploit additional information about the keys to sort in $o(n \lg n)$ time. In particular, with counting sort we can sort n keys, each an integer in the range 0 to k, in time $\Theta(n + k)$, which is $\Theta(n)$ when $k = O(n)$.

Since we can circumvent the $\Omega(n \lg n)$ lower bound for sorting when the keys are integers in a bounded range, you might wonder whether we can perform each of the priority-queue operations in $o(\lg n)$ time in a similar scenario. In this chapter, we shall see that we can: van Emde Boas trees support the priority-queue operations, and a few others, each in $O(\lg \lg n)$ worst-case time. The hitch is that the keys must be integers in the range 0 to $n - 1$, with no duplicates allowed.

Specifically, van Emde Boas trees support each of the dynamic set operations listed on page 230—SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR—in $O(\lg \lg n)$ time. In this chapter, we will omit discussion of satellite data and focus only on storing keys. Because we concentrate on keys and disallow duplicate keys to be stored, instead of describing the SEARCH

[1] Chapter 13 does not explicitly discuss how to implement EXTRACT-MIN and DECREASE-KEY, but we can easily build these operations for any data structure that supports MINIMUM, DELETE, and INSERT.

operation, we will implement the simpler operation MEMBER(S, x), which returns a boolean indicating whether the value x is currently in dynamic set S.

So far, we have used the parameter n for two distinct purposes: the number of elements in the dynamic set, and the range of the possible values. To avoid any further confusion, from here on we will use n to denote the number of elements currently in the set and u as the range of possible values, so that each van Emde Boas tree operation runs in $O(\lg \lg u)$ time. We call the set $\{0, 1, 2, \ldots, u-1\}$ the ***universe*** of values that can be stored and u the ***universe size***. We assume throughout this chapter that u is an exact power of 2, i.e., $u = 2^k$ for some integer $k \geq 1$.

Section 20.1 starts us out by examining some simple approaches that will get us going in the right direction. We enhance these approaches in Section 20.2, introducing proto van Emde Boas structures, which are recursive but do not achieve our goal of $O(\lg \lg u)$-time operations. Section 20.3 modifies proto van Emde Boas structures to develop van Emde Boas trees, and it shows how to implement each operation in $O(\lg \lg u)$ time.

20.1 Preliminary approaches

In this section, we shall examine various approaches for storing a dynamic set. Although none will achieve the $O(\lg \lg u)$ time bounds that we desire, we will gain insights that will help us understand van Emde Boas trees when we see them later in this chapter.

Direct addressing

Direct addressing, as we saw in Section 11.1, provides the simplest approach to storing a dynamic set. Since in this chapter we are concerned only with storing keys, we can simplify the direct-addressing approach to store the dynamic set as a bit vector, as discussed in Exercise 11.1-2. To store a dynamic set of values from the universe $\{0, 1, 2, \ldots, u-1\}$, we maintain an array $A[0\,..\,u-1]$ of u bits. The entry $A[x]$ holds a 1 if the value x is in the dynamic set, and it holds a 0 otherwise. Although we can perform each of the INSERT, DELETE, and MEMBER operations in $O(1)$ time with a bit vector, the remaining operations—MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR—each take $\Theta(u)$ time in the worst case because

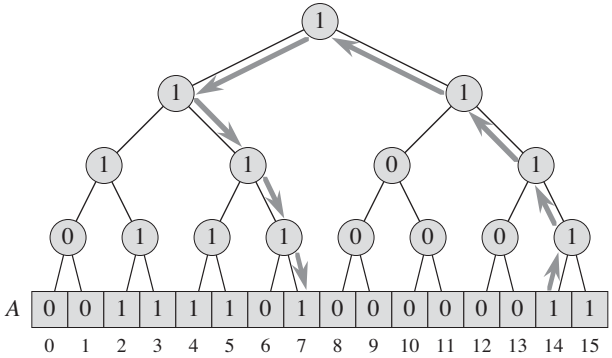


Figure 20.1 A binary tree of bits superimposed on top of a bit vector representing the set $\{2, 3, 4, 5, 7, 14, 15\}$ when $u = 16$. Each internal node contains a 1 if and only if some leaf in its subtree contains a 1. The arrows show the path followed to determine the predecessor of 14 in the set.

we might have to scan through $\Theta(u)$ elements.[2] For example, if a set contains only the values 0 and $u - 1$, then to find the successor of 0, we would have to scan entries 1 through $u - 2$ before finding a 1 in $A[u - 1]$.

Superimposing a binary tree structure

We can short-cut long scans in the bit vector by superimposing a binary tree of bits on top of it. Figure 20.1 shows an example. The entries of the bit vector form the leaves of the binary tree, and each internal node contains a 1 if and only if any leaf in its subtree contains a 1. In other words, the bit stored in an internal node is the logical-or of its two children.

The operations that took $\Theta(u)$ worst-case time with an unadorned bit vector now use the tree structure:

- To find the minimum value in the set, start at the root and head down toward the leaves, always taking the leftmost node containing a 1.
- To find the maximum value in the set, start at the root and head down toward the leaves, always taking the rightmost node containing a 1.

---

[2]We assume throughout this chapter that MINIMUM and MAXIMUM return NIL if the dynamic set is empty and that SUCCESSOR and PREDECESSOR return NIL if the element they are given has no successor or predecessor, respectively.

- To find the successor of x, start at the leaf indexed by x, and head up toward the root until we enter a node from the left and this node has a 1 in its right child z. Then head down through node z, always taking the leftmost node containing a 1 (i.e., find the minimum value in the subtree rooted at the right child z).
- To find the predecessor of x, start at the leaf indexed by x, and head up toward the root until we enter a node from the right and this node has a 1 in its left child z. Then head down through node z, always taking the rightmost node containing a 1 (i.e., find the maximum value in the subtree rooted at the left child z).

Figure 20.1 shows the path taken to find the predecessor, 7, of the value 14.

We also augment the INSERT and DELETE operations appropriately. When inserting a value, we store a 1 in each node on the simple path from the appropriate leaf up to the root. When deleting a value, we go from the appropriate leaf up to the root, recomputing the bit in each internal node on the path as the logical-or of its two children.

Since the height of the tree is $\lg u$ and each of the above operations makes at most one pass up the tree and at most one pass down, each operation takes $O(\lg u)$ time in the worst case.

This approach is only marginally better than just using a red-black tree. We can still perform the MEMBER operation in $O(1)$ time, whereas searching a red-black tree takes $O(\lg n)$ time. Then again, if the number n of elements stored is much smaller than the size u of the universe, a red-black tree would be faster for all the other operations.

Superimposing a tree of constant height

What happens if we superimpose a tree with greater degree? Let us assume that the size of the universe is $u = 2^{2k}$ for some integer k, so that $\sqrt{u}$ is an integer. Instead of superimposing a binary tree on top of the bit vector, we superimpose a tree of degree $\sqrt{u}$. Figure 20.2(a) shows such a tree for the same bit vector as in Figure 20.1. The height of the resulting tree is always 2.

As before, each internal node stores the logical-or of the bits within its subtree, so that the $\sqrt{u}$ internal nodes at depth 1 summarize each group of $\sqrt{u}$ values. As Figure 20.2(b) demonstrates, we can think of these nodes as an array $summary[0\,..\,\sqrt{u}-1]$, where $summary[i]$ contains a 1 if and only if the subarray $A[i\sqrt{u}\,..\,(i+1)\sqrt{u}-1]$ contains a 1. We call this $\sqrt{u}$-bit subarray of A the ith ***cluster***. For a given value of x, the bit $A[x]$ appears in cluster number $\lfloor x/\sqrt{u} \rfloor$. Now INSERT becomes an $O(1)$-time operation: to insert x, set both $A[x]$ and $summary[\lfloor x/\sqrt{u} \rfloor]$ to 1. We can use the *summary* array to perform

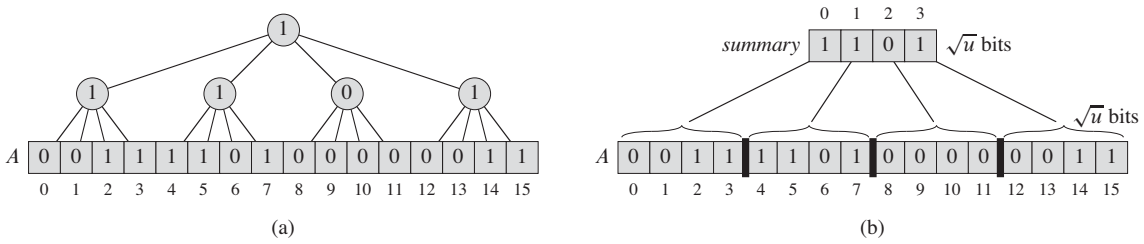


Figure 20.2 **(a)** A tree of degree $\sqrt{u}$ superimposed on top of the same bit vector as in Figure 20.1. Each internal node stores the logical-or of the bits in its subtree. **(b)** A view of the same structure, but with the internal nodes at depth 1 treated as an array $summary[0\,..\,\sqrt{u}-1]$, where $summary[i]$ is the logical-or of the subarray $A[i\sqrt{u}\,..\,(i+1)\sqrt{u}-1]$.

each of the operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and DELETE in $O(\sqrt{u})$ time:

- To find the minimum (maximum) value, find the leftmost (rightmost) entry in *summary* that contains a 1, say $summary[i]$, and then do a linear search within the ith cluster for the leftmost (rightmost) 1.
- To find the successor (predecessor) of x, first search to the right (left) within its cluster. If we find a 1, that position gives the result. Otherwise, let $i = \lfloor x/\sqrt{u} \rfloor$ and search to the right (left) within the *summary* array from index i. The first position that holds a 1 gives the index of a cluster. Search within that cluster for the leftmost (rightmost) 1. That position holds the successor (predecessor).
- To delete the value x, let $i = \lfloor x/\sqrt{u} \rfloor$. Set $A[x]$ to 0 and then set $summary[i]$ to the logical-or of the bits in the ith cluster.

In each of the above operations, we search through at most two clusters of $\sqrt{u}$ bits plus the *summary* array, and so each operation takes $O(\sqrt{u})$ time.

At first glance, it seems as though we have made negative progress. Superimposing a binary tree gave us $O(\lg u)$-time operations, which are asymptotically faster than $O(\sqrt{u})$ time. Using a tree of degree $\sqrt{u}$ will turn out to be a key idea of van Emde Boas trees, however. We continue down this path in the next section.

Exercises

20.1-1

Modify the data structures in this section to support duplicate keys.

20.1-2
Modify the data structures in this section to support keys that have associated satellite data.

20.1-3
Observe that, using the structures in this section, the way we find the successor and predecessor of a value x does not depend on whether x is in the set at the time. Show how to find the successor of x in a binary search tree when x is not stored in the tree.

20.1-4
Suppose that instead of superimposing a tree of degree $\sqrt{u}$, we were to superimpose a tree of degree $u^{1/k}$, where $k > 1$ is a constant. What would be the height of such a tree, and how long would each of the operations take?

20.2 A recursive structure

In this section, we modify the idea of superimposing a tree of degree $\sqrt{u}$ on top of a bit vector. In the previous section, we used a summary structure of size $\sqrt{u}$, with each entry pointing to another stucture of size $\sqrt{u}$. Now, we make the structure recursive, shrinking the universe size by the square root at each level of recursion. Starting with a universe of size u, we make structures holding $\sqrt{u} = u^{1/2}$ items, which themselves hold structures of $u^{1/4}$ items, which hold structures of $u^{1/8}$ items, and so on, down to a base size of 2.

For simplicity, in this section, we assume that $u = 2^{2^k}$ for some integer k, so that $u, u^{1/2}, u^{1/4}, \ldots$ are integers. This restriction would be quite severe in practice, allowing only values of u in the sequence $2, 4, 16, 256, 65536, \ldots$. We shall see in the next section how to relax this assumption and assume only that $u = 2^k$ for some integer k. Since the structure we examine in this section is only a precursor to the true van Emde Boas tree structure, we tolerate this restriction in favor of aiding our understanding.

Recalling that our goal is to achieve running times of $O(\lg \lg u)$ for the operations, let's think about how we might obtain such running times. At the end of Section 4.3, we saw that by changing variables, we could show that the recurrence

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n \tag{20.1}$$

has the solution $T(n) = O(\lg n \lg \lg n)$. Let's consider a similar, but simpler, recurrence:

$$T(u) = T(\sqrt{u}) + O(1) \, . \tag{20.2}$$

If we use the same technique, changing variables, we can show that recurrence (20.2) has the solution $T(u) = O(\lg \lg u)$. Let $m = \lg u$, so that $u = 2^m$ and we have

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

Now we rename $S(m) = T(2^m)$, giving the new recurrence

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

By case 2 of the master method, this recurrence has the solution $S(m) = O(\lg m)$. We change back from $S(m)$ to $T(u)$, giving $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

Recurrence (20.2) will guide our search for a data structure. We will design a recursive data structure that shrinks by a factor of $\sqrt{u}$ in each level of its recursion. When an operation traverses this data structure, it will spend a constant amount of time at each level before recursing to the level below. Recurrence (20.2) will then characterize the running time of the operation.

Here is another way to think of how the term $\lg \lg u$ ends up in the solution to recurrence (20.2). As we look at the universe size in each level of the recursive data structure, we see the sequence $u, u^{1/2}, u^{1/4}, u^{1/8}, \ldots$. If we consider how many bits we need to store the universe size at each level, we need $\lg u$ at the top level, and each level needs half the bits of the previous level. In general, if we start with b bits and halve the number of bits at each level, then after $\lg b$ levels, we get down to just one bit. Since $b = \lg u$, we see that after $\lg \lg u$ levels, we have a universe size of 2.

Looking back at the data structure in Figure 20.2, a given value x resides in cluster number $\lfloor x/\sqrt{u} \rfloor$. If we view x as a $\lg u$-bit binary integer, that cluster number, $\lfloor x/\sqrt{u} \rfloor$, is given by the most significant $(\lg u)/2$ bits of x. Within its cluster, x appears in position $x \bmod \sqrt{u}$, which is given by the least significant $(\lg u)/2$ bits of x. We will need to index in this way, and so let us define some functions that will help us do so:

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

The function $\text{high}(x)$ gives the most significant $(\lg u)/2$ bits of x, producing the number of x's cluster. The function $\text{low}(x)$ gives the least significant $(\lg u)/2$ bits of x and provides x's position within its cluster. The function $\text{index}(x, y)$ builds an element number from x and y, treating x as the most significant $(\lg u)/2$ bits of the element number and y as the least significant $(\lg u)/2$ bits. We have the identity $x = \text{index}(\text{high}(x), \text{low}(x))$. The value of u used by each of these functions will

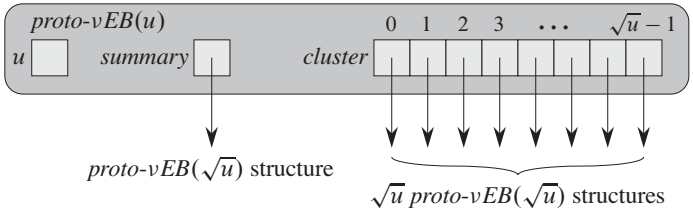


Figure 20.3 The information in a *proto-vEB*(u) structure when $u \geq 4$. The structure contains the universe size u, a pointer *summary* to a *proto-vEB*$(\sqrt{u})$ structure, and an array *cluster*$[0\,..\,\sqrt{u}-1]$ of $\sqrt{u}$ pointers to *proto-vEB*$(\sqrt{u})$ structures.

always be the universe size of the data structure in which we call the function, which changes as we descend into the recursive structure.

20.2.1 Proto van Emde Boas structures

Taking our cue from recurrence (20.2), let us design a recursive data structure to support the operations. Although this data structure will fail to achieve our goal of $O(\lg \lg u)$ time for some operations, it serves as a basis for the van Emde Boas tree structure that we will see in Section 20.3.

For the universe $\{0, 1, 2, \ldots, u-1\}$, we define a ***proto van Emde Boas structure***, or ***proto-vEB structure***, which we denote as *proto-vEB*(u), recursively as follows. Each *proto-vEB*(u) structure contains an attribute u giving its universe size. In addition, it contains the following:

- If $u = 2$, then it is the base size, and it contains an array $A[0\,..\,1]$ of two bits.
- Otherwise, $u = 2^{2^k}$ for some integer $k \geq 1$, so that $u \geq 4$. In addition to the universe size u, the data structure *proto-vEB*(u) contains the following attributes, illustrated in Figure 20.3:
  - a pointer named *summary* to a *proto-vEB*$(\sqrt{u})$ structure and
  - an array *cluster*$[0\,..\,\sqrt{u}-1]$ of $\sqrt{u}$ pointers, each to a *proto-vEB*$(\sqrt{u})$ structure.

The element x, where $0 \leq x < u$, is recursively stored in the cluster numbered high(x) as element low(x) within that cluster.

In the two-level structure of the previous section, each node stores a summary array of size $\sqrt{u}$, in which each entry contains a bit. From the index of each entry, we can compute the starting index of the subarray of size $\sqrt{u}$ that the bit summarizes. In the proto-vEB structure, we use explicit pointers rather than index

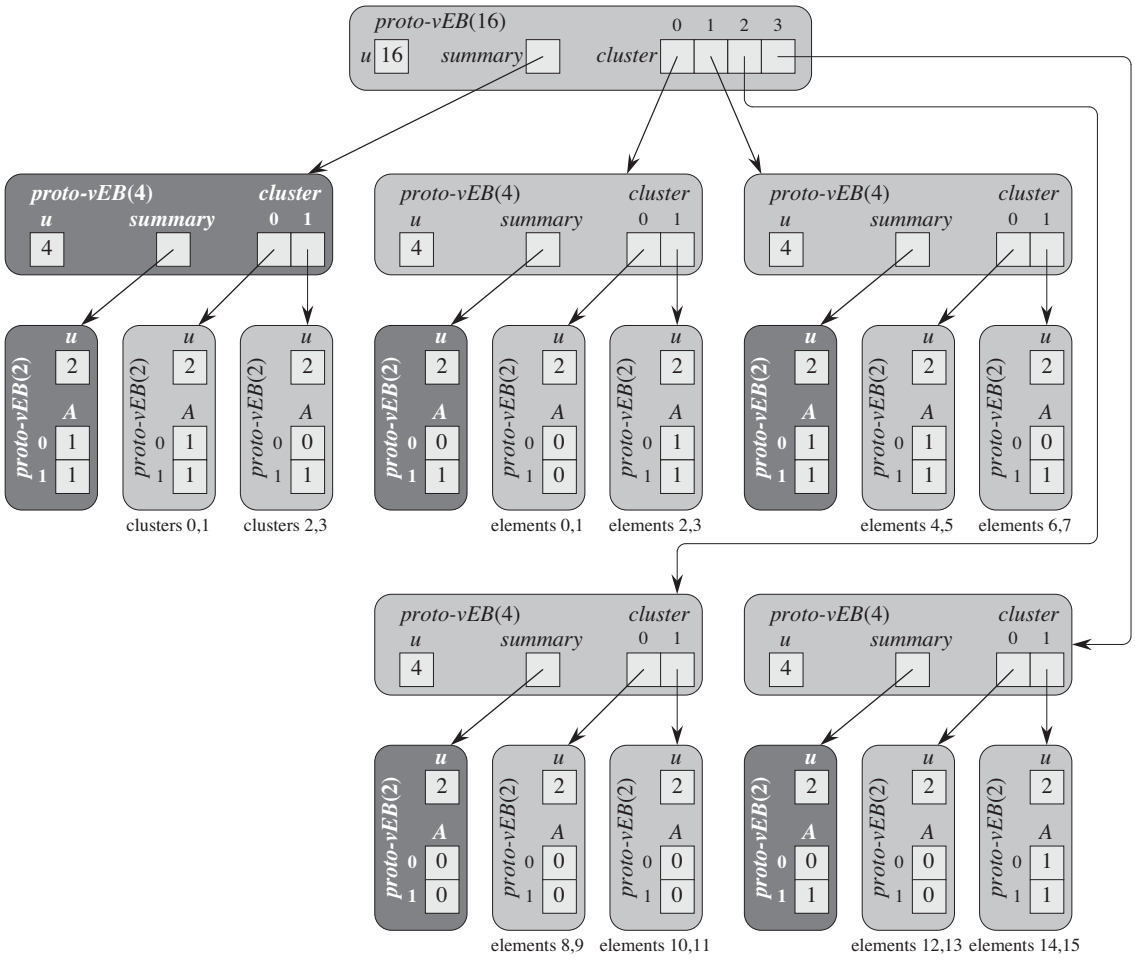


Figure 20.4 A *proto-vEB*(16) structure representing the set $\{2, 3, 4, 5, 7, 14, 15\}$. It points to four *proto-vEB*(4) structures in *cluster*$[0\,..\,3]$, and to a summary structure, which is also a *proto-vEB*(4). Each *proto-vEB*(4) structure points to two *proto-vEB*(2) structures in *cluster*$[0\,..\,1]$, and to a *proto-vEB*(2) summary. Each *proto-vEB*(2) structure contains just an array $A[0\,..\,1]$ of two bits. The *proto-vEB*(2) structures above "elements i, j" store bits i and j of the actual dynamic set, and the *proto-vEB*(2) structures above "clusters i, j" store the summary bits for clusters i and j in the top-level *proto-vEB*(16) structure. For clarity, heavy shading indicates the top level of a proto-vEB structure that stores summary information for its parent structure; such a proto-vEB structure is otherwise identical to any other proto-vEB structure with the same universe size.

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 high(x) says which *proto-vEB*($\sqrt{u}$) structure we visit, and 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 high(6) $= 1$ when $u = 16$, we recurse into the *proto-vEB*(4) structure in the upper right, and we ask about element low(6) $= 2$ of that structure. In this recursive call, $u = 4$, and so we recurse again. With $u = 4$, we have high(2) $= 1$ and 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 $proto\text{-}vEB(\sqrt{u})$ 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 $proto\text{-}vEB(\sqrt{u})$ 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 $proto\text{-}vEB(u)$ 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[high(x)], low(x))
 6      if offset ≠ NIL
 7          return index(high(x), offset)
 8      else succ-cluster = PROTO-VEB-SUCCESSOR(V.summary, high(x))
 9          if succ-cluster == NIL
10              return NIL
11          else offset = PROTO-VEB-MINIMUM(V.cluster[succ-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[high(x)], low(x))
4       PROTO-VEB-INSERT(V.summary, 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-

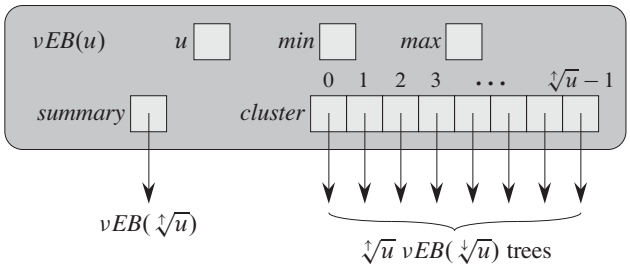


Figure 20.5 The information in a $vEB(u)$ tree when $u > 2$. The structure contains the universe size u, elements *min* and *max*, a pointer *summary* to a $vEB(\sqrt[\uparrow]{u})$ tree, and an array $cluster[0\,..\,\sqrt[\uparrow]{u} - 1]$ of $\sqrt[\uparrow]{u}$ pointers to $vEB(\sqrt[\downarrow]{u})$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

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

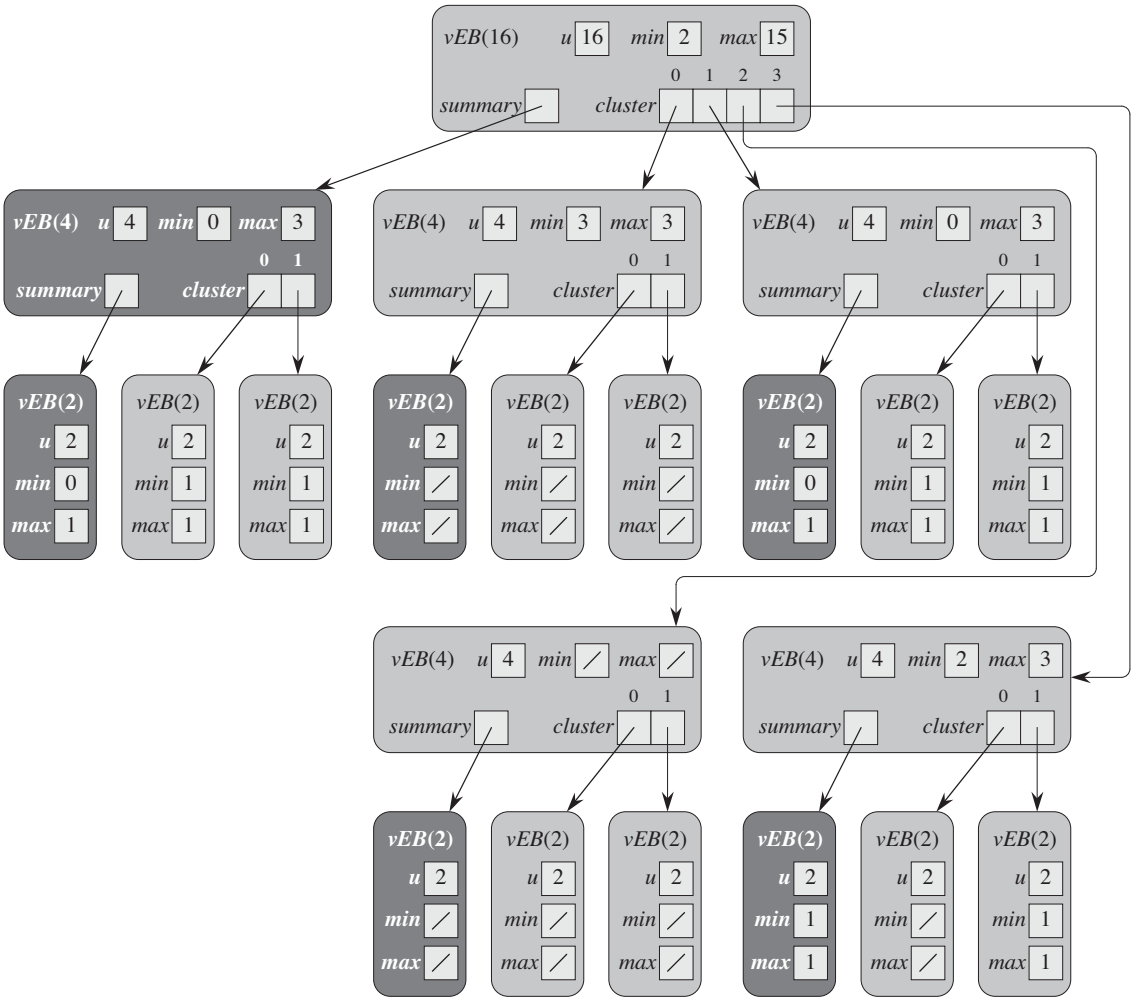


Figure 20.6 A $vEB(16)$ tree corresponding to the proto-vEB tree in Figure 20.4. It stores the set $\{2, 3, 4, 5, 7, 14, 15\}$. Slashes indicate NIL values. The value stored in the *min* attribute of a vEB tree does not appear in any of its clusters. Heavy shading serves the same purpose here as in Figure 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 ≠ NIL and x < V.min
 6      return V.min
 7  else max-low = VEB-TREE-MAXIMUM(V.cluster[high(x)])
 8      if max-low ≠ NIL and low(x) < max-low
 9          offset = VEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10          return index(high(x), offset)
11      else succ-cluster = VEB-TREE-SUCCESSOR(V.summary, high(x))
12          if succ-cluster == NIL
13              return NIL
14          else offset = VEB-TREE-MINIMUM(V.cluster[succ-cluster])
15              return index(succ-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 $\sqrt[\updownarrow]{u}$). In either case, the one recursive call is on a vEB tree with universe size at most $\sqrt[\updownarrow]{u}$. 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 ≠ NIL and x > V.max
 6      return V.max
 7  else min-low = VEB-TREE-MINIMUM(V.cluster[high(x)])
 8      if min-low ≠ NIL and low(x) > min-low
 9          offset = VEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
10          return index(high(x), offset)
11      else pred-cluster = VEB-TREE-PREDECESSOR(V.summary, high(x))
12          if pred-cluster == NIL
13              if V.min ≠ NIL and x > V.min
14                  return V.min
15              else return NIL
16          else offset = VEB-TREE-MAXIMUM(V.cluster[pred-cluster])
17              return index(pred-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[high(x)]) == NIL
 7              vEB-TREE-INSERT(V.summary, high(x))
 8              vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
 9          else vEB-TREE-INSERT(V.cluster[high(x)], 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 $\uparrow\sqrt{u}$) or the recursive call in line 9 (run on a vEB with universe size $\downarrow\sqrt{u}$) executes. In either case, the one recursive call is on a vEB tree with universe size at most $\uparrow\sqrt{u}$. 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 $\uparrow\sqrt{u}$ clusters, each with universe size $\downarrow\sqrt{u}$, 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.

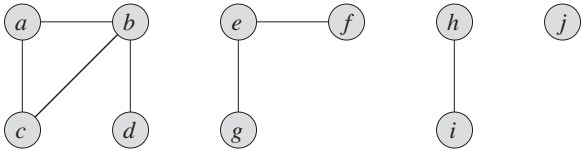
Throughout this chapter, we shall analyze the running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m, the total number of MAKE-SET, UNION, and FIND-SET operations. Since the sets are disjoint, each UNION operation reduces the number of sets 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 ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ 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 con-

nected 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.

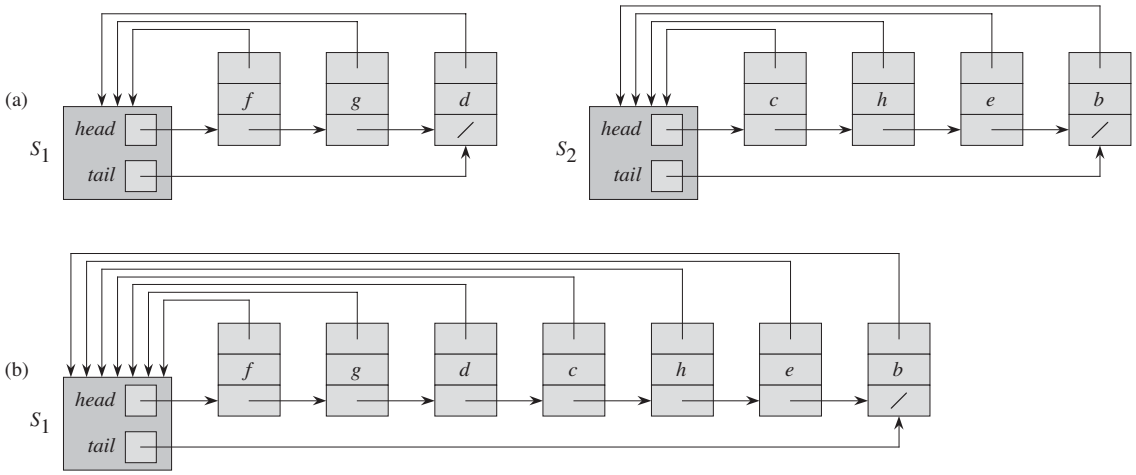


Figure 21.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

A simple implementation of union

The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET. As Figure 21.2(b) shows, we perform UNION(x, y) by appending y's list onto the end of x's list. The representative of x's list becomes the representative of the resulting set. We use the *tail* pointer for x's list to quickly find where to append y's list. Because all members of y's list join x's list, we can destroy the set object for y's list. Unfortunately, we must update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list. In Figure 21.2, for example, the operation UNION(g, e) causes pointers to be updated in the objects for b, c, e, and h.

In fact, we can easily construct a sequence of m operations on n objects that requires $\Theta(n^2)$ time. Suppose that we have objects $x_1, x_2, \ldots, x_n$. We execute the sequence of n MAKE-SET operations followed by $n - 1$ UNION operations shown in Figure 21.3, so that $m = 2n - 1$. We spend $\Theta(n)$ time performing the n MAKE-SET operations. Because the ith UNION operation updates i objects, the total number of objects updated by all $n - 1$ UNION operations is

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

Figure 21.3 A sequence of $2n-1$ operations on n objects that takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

The total number of operations is $2n-1$, and so each operation on average requires $\Theta(n)$ time. That is, the amortized time of an operation is $\Theta(n)$.

A weighted-union heuristic

In the worst case, the above implementation of the UNION procedure requires an average of $\Theta(n)$ time per call because we may be appending a longer list onto a shorter list; we must update the pointer to the set object for each member of the longer list. Suppose instead that each list also includes the length of the list (which we can easily maintain) and that we always append the shorter list onto the longer, breaking ties arbitrarily. With this simple ***weighted-union heuristic***, a single UNION operation can still take $\Omega(n)$ time if both sets have $\Omega(n)$ members. As the following theorem shows, however, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Theorem 21.1
Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Proof Because each UNION operation unites two disjoint sets, we perform at most $n-1$ UNION operations over all. We now bound the total time taken by these UNION operations. We start by determining, for each object, an upper bound on the number of times the object's pointer back to its set object is updated. Consider a particular object x. We know that each time x's pointer was updated, x must have started in the smaller set. The first time x's pointer was updated, therefore, the resulting set must have had at least 2 members. Similarly, the next time x's pointer was updated, the resulting set must have had at least 4 members. Continuing on, we observe that for any $k \leq n$, after x's pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members. Since the largest set has at most n members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations. Thus the total time spent updating object pointers over all UNION operations is $O(n \lg n)$. We must also account for updating the *tail* pointers and the list lengths, which take only $\Theta(1)$ time per UNION operation. The total time spent in all UNION operations is thus $O(n \lg n)$.

The time for the entire sequence of m operations follows easily. Each MAKE-SET and FIND-SET operation takes $O(1)$ time, and there are $O(m)$ of them. The total time for the entire sequence is thus $O(m + n \lg n)$. ■

Exercises

21.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

21.2-2

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic.

```
 1  for i = 1 to 16
 2      MAKE-SET(x_i)
 3  for i = 1 to 15 by 2
 4      UNION(x_i, x_{i+1})
 5  for i = 1 to 13 by 4
 6      UNION(x_i, x_{i+2})
 7  UNION(x_1, x_5)
 8  UNION(x_11, x_13)
 9  UNION(x_1, x_10)
10  FIND-SET(x_2)
11  FIND-SET(x_9)
```

Assume that if the sets containing x_i and x_j have the same size, then the operation UNION(x_i, x_j) appends x_j's list onto x_i's list.

21.2-3
Adapt the aggregate proof of Theorem 21.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

21.2-4
Give a tight asymptotic bound on the running time of the sequence of operations in Figure 21.3 assuming the linked-list representation and the weighted-union heuristic.

21.2-5
Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (*head* and *tail*), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (*Hint:* Use the tail of a linked list as its set's representative.)

21.2-6
Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the *tail* pointer to the last object in each list. Whether or not the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (*Hint:* Rather than appending one list to another, splice them together.)

21.3 Disjoint-set forests

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, illustrated in Figure 21.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent. As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics—"union by rank" and "path compression"—we can achieve an asymptotically optimal disjoint-set data structure.

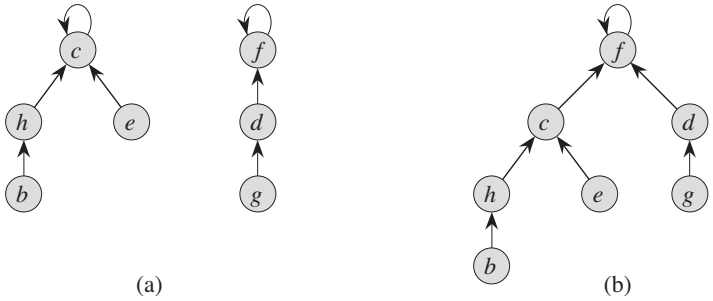


Figure 21.4 A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

We perform the three disjoint-set operations as follows. A MAKE-SET operation simply creates a tree with just one node. We perform a FIND-SET operation by following parent pointers until we find the root of the tree. The nodes visited on this simple path toward the root constitute the ***find path***. A UNION operation, shown in Figure 21.4(b), causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

So far, we have not improved on the linked-list implementation. A sequence of $n - 1$ UNION operations may create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number of operations m.

The first heuristic, ***union by rank***, is similar to the weighted-union heuristic we used with the linked-list representation. The obvious approach would be to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis. For each node, we maintain a ***rank***, which is an upper bound on the height of the node. In union by rank, we make the root with smaller rank point to the root with larger rank during a UNION operation.

The second heuristic, ***path compression***, is also quite simple and highly effective. As shown in Figure 21.5, we use it during FIND-SET operations to make each node on the find path point directly to the root. Path compression does not change any ranks.

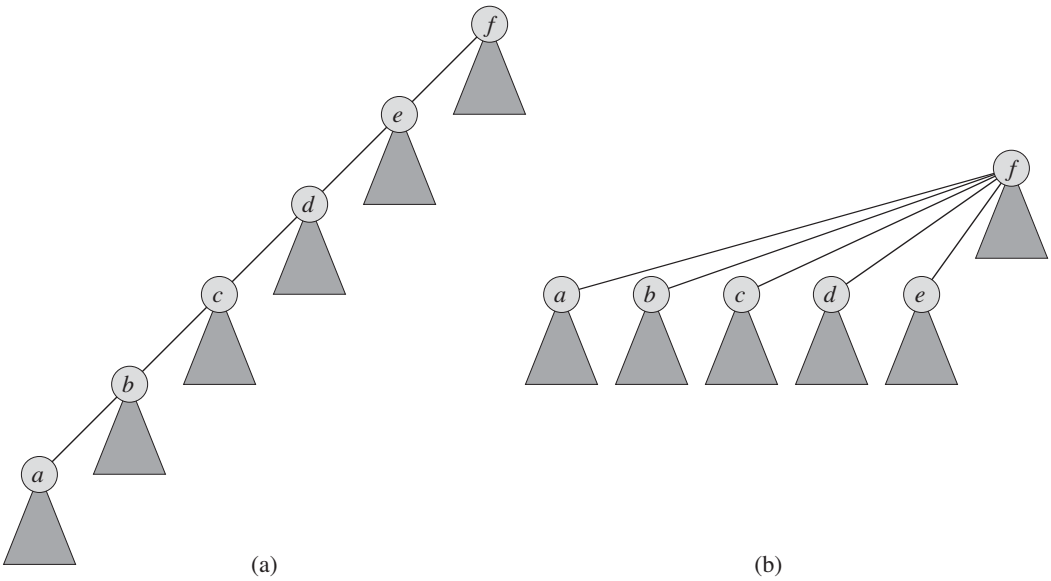


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Pseudocode for disjoint-set forests

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. With each node x, we maintain the integer value $x.rank$, which is an upper bound on the height of x (the number of edges in the longest simple path between x and a descendant leaf). When MAKE-SET creates a singleton set, the single node in the corresponding tree has an initial rank of 0. Each FIND-SET operation leaves all ranks unchanged. The UNION operation has two cases, depending on whether the roots of the trees have equal rank. If the roots have unequal rank, we make the root with higher rank the parent of the root with lower rank, but the ranks themselves remain unchanged. If, instead, the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

Let us put this method into pseudocode. We designate the parent of node x by $x.p$. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs.

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

The FIND-SET procedure with path compression is quite simple:

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

The FIND-SET procedure is a ***two-pass method***: as it recurses, it makes one pass up the find path to find the root, and as the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root. Each call of FIND-SET(x) returns $x.p$ in line 3. If x is the root, then FIND-SET skips line 2 and instead returns $x.p$, which is x; this is the case in which the recursion bottoms out. Otherwise, line 2 executes, and the recursive call with parameter $x.p$ returns a pointer to the root. Line 2 updates node x to point directly to the root, and line 3 returns this pointer.

Effect of the heuristics on the running time

Separately, either union by rank or path compression improves the running time of the operations on disjoint-set forests, and the improvement is even greater when we use the two heuristics together. Alone, union by rank yields a running time of $O(m \lg n)$ (see Exercise 21.4-4), and this bound is tight (see Exercise 21.3-3). Although we shall not prove it here, for a sequence of n MAKE-SET operations (and hence at most $n - 1$ UNION operations) and f FIND-SET operations, the path-compression heuristic alone gives a worst-case running time of $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

When we use both union by rank and path compression, the worst-case running time is $O(m\,\alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function, which we define in Section 21.4. In any conceivable application of a disjoint-set data structure, $\alpha(n) \leq 4$; thus, we can view the running time as linear in m in all practical situations. Strictly speaking, however, it is superlinear. In Section 21.4, we prove this upper bound.

Exercises

21.3-1
Redo Exercise 21.2-2 using a disjoint-set forest with union by rank and path compression.

21.3-2
Write a nonrecursive version of FIND-SET with path compression.

21.3-3
Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.

21.3-4
Suppose that we wish to add the operation PRINT-SET(x), which is given a node x and prints all the members of x's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that PRINT-SET(x) takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time.

21.3-5 ⋆
Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations, takes only $O(m)$ time if we use both path compression and union by rank. What happens in the same situation if we use only the path-compression heuristic?

⋆ 21.4 Analysis of union by rank with path compression

As noted in Section 21.3, the combined union-by-rank and path-compression heuristic runs in time $O(m\,\alpha(n))$ for m disjoint-set operations on n elements. In this section, we shall examine the function α to see just how slowly it grows. Then we prove this running time using the potential method of amortized analysis.

A very quickly growing function and its very slowly growing inverse

For integers $k \geq 0$ and $j \geq 1$, we define the function $A_k(j)$ as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases}$$

where the expression $A_{k-1}^{(j+1)}(j)$ uses the functional-iteration notation given in Section 3.2. Specifically, $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ for $i \geq 1$. We will refer to the parameter k as the ***level*** of the function A.

The function $A_k(j)$ strictly increases with both j and k. To see just how quickly this function grows, we first obtain closed-form expressions for $A_1(j)$ and $A_2(j)$.

Lemma 21.2
For any integer $j \geq 1$, we have $A_1(j) = 2j + 1$.

Proof We first use induction on i to show that $A_0^{(i)}(j) = j + i$. For the base case, we have $A_0^{(0)}(j) = j = j + 0$. For the inductive step, assume that $A_0^{(i-1)}(j) = j + (i-1)$. Then $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i-1)) + 1 = j + i$. Finally, we note that $A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$. ∎

Lemma 21.3
For any integer $j \geq 1$, we have $A_2(j) = 2^{j+1}(j+1) - 1$.

Proof We first use induction on i to show that $A_1^{(i)}(j) = 2^i(j+1) - 1$. For the base case, we have $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. For the inductive step, assume that $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$. Then $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2 \cdot (2^{i-1}(j+1) - 1) + 1 = 2^i(j+1) - 2 + 1 = 2^i(j+1) - 1$. Finally, we note that $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$. ∎

Now we can see how quickly $A_k(j)$ grows by simply examining $A_k(1)$ for levels $k = 0, 1, 2, 3, 4$. From the definition of $A_0(k)$ and the above lemmas, we have $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$, and $A_2(1) = 2^{1+1} \cdot (1+1) - 1 = 7$.

We also have

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

and

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \\ &\gg A_2(2047) \\ &= 2^{2048} \cdot 2048 - 1 \\ &> 2^{2048} \\ &= (2^4)^{512} \\ &= 16^{512} \\ &\gg 10^{80} , \end{aligned}$$

which is the estimated number of atoms in the observable universe. (The symbol "$\gg$" denotes the "much-greater-than" relation.)

We define the inverse of the function $A_k(n)$, for integer $n \geq 0$, by

$$\alpha(n) = \min \{k : A_k(1) \geq n\} .$$

In words, $\alpha(n)$ is the lowest level k for which $A_k(1)$ is at least n. From the above values of $A_k(1)$, we see that

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 , \\ 1 & \text{for } n = 3 , \\ 2 & \text{for } 4 \leq n \leq 7 , \\ 3 & \text{for } 8 \leq n \leq 2047 , \\ 4 & \text{for } 2048 \leq n \leq A_4(1) . \end{cases}$$

It is only for values of n so large that the term "astronomical" understates them (greater than $A_4(1)$, a huge number) that $\alpha(n) > 4$, and so $\alpha(n) \leq 4$ for all practical purposes.

Properties of ranks

In the remainder of this section, we prove an $O(m\,\alpha(n))$ bound on the running time of the disjoint-set operations with union by rank and path compression. In order to prove this bound, we first prove some simple properties of ranks.

Lemma 21.4
For all nodes x, we have $x.rank \leq x.p.rank$, with strict inequality if $x \neq x.p$. The value of $x.rank$ is initially 0 and increases through time until $x \neq x.p$; from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

Proof The proof is a straightforward induction on the number of operations, using the implementations of MAKE-SET, UNION, and FIND-SET that appear in Section 21.3. We leave it as Exercise 21.4-1. ■

Corollary 21.5
As we follow the simple path from any node toward a root, the node ranks strictly increase. ■

Lemma 21.6
Every node has rank at most $n - 1$.

Proof Each node's rank starts at 0, and it increases only upon LINK operations. Because there are at most $n - 1$ UNION operations, there are also at most $n - 1$ LINK operations. Because each LINK operation either leaves all ranks alone or increases some node's rank by 1, all ranks are at most $n - 1$. ■

Lemma 21.6 provides a weak bound on ranks. In fact, every node has rank at most $\lfloor \lg n \rfloor$ (see Exercise 21.4-2). The looser bound of Lemma 21.6 will suffice for our purposes, however.

Proving the time bound

We shall use the potential method of amortized analysis (see Section 17.3) to prove the $O(m\,\alpha(n))$ time bound. In performing the amortized analysis, we will find it convenient to assume that we invoke the LINK operation rather than the UNION operation. That is, since the parameters of the LINK procedure are pointers to two roots, we act as though we perform the appropriate FIND-SET operations separately. The following lemma shows that even if we count the extra FIND-SET operations induced by UNION calls, the asymptotic running time remains unchanged.

Lemma 21.7
Suppose we convert a sequence S' of m' MAKE-SET, UNION, and FIND-SET operations into a sequence S of m MAKE-SET, LINK, and FIND-SET operations by turning each UNION into two FIND-SET operations followed by a LINK. Then, if sequence S runs in $O(m\,\alpha(n))$ time, sequence S' runs in $O(m'\,\alpha(n))$ time.

Proof Since each UNION operation in sequence S' is converted into three operations in S, we have $m' \le m \le 3m'$. Since $m = O(m')$, an $O(m\,\alpha(n))$ time bound for the converted sequence S implies an $O(m'\,\alpha(n))$ time bound for the original sequence S'. ■

In the remainder of this section, we shall assume that the initial sequence of m' MAKE-SET, UNION, and FIND-SET operations has been converted to a sequence of m MAKE-SET, LINK, and FIND-SET operations. We now prove an $O(m\,\alpha(n))$ time bound for the converted sequence and appeal to Lemma 21.7 to prove the $O(m'\,\alpha(n))$ running time of the original sequence of m' operations.

Potential function

The potential function we use assigns a potential $\phi_q(x)$ to each node x in the disjoint-set forest after q operations. We sum the node potentials for the potential of the entire forest: $\Phi_q = \sum_x \phi_q(x)$, where Φ_q denotes the potential of the forest after q operations. The forest is empty prior to the first operation, and we arbitrarily set $\Phi_0 = 0$. No potential Φ_q will ever be negative.

The value of $\phi_q(x)$ depends on whether x is a tree root after the qth operation. If it is, or if $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$.

Now suppose that after the qth operation, x is not a root and that $x.rank \ge 1$. We need to define two auxiliary functions on x before we can define $\phi_q(x)$. First we define

$$\text{level}(x) = \max\{k : x.p.rank \ge A_k(x.rank)\} \ .$$

That is, level(x) is the greatest level k for which A_k, applied to x's rank, is no greater than x's parent's rank.

We claim that

$$0 \le \text{level}(x) < \alpha(n) \ , \tag{21.1}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\ge x.rank + 1 \quad \text{(by Lemma 21.4)} \\ &= A_0(x.rank) \quad \text{(by definition of } A_0(j)) \ , \end{aligned}$$

which implies that level$(x) \ge 0$, and we have

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(1) && \text{(because } A_k(j) \text{ is strictly increasing)} \\ &\geq n && \text{(by the definition of } \alpha(n)\text{)} \\ &> x.p.rank && \text{(by Lemma 21.6)} , \end{aligned}$$

which implies that $\text{level}(x) < \alpha(n)$. Note that because $x.p.rank$ monotonically increases over time, so does $\text{level}(x)$.

The second auxiliary function applies when $x.rank \geq 1$:

$$\text{iter}(x) = \max\left\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\right\} .$$

That is, $\text{iter}(x)$ is the largest number of times we can iteratively apply $A_{\text{level}(x)}$, applied initially to x's rank, before we get a value greater than x's parent's rank.

We claim that when $x.rank \geq 1$, we have

$$1 \leq \text{iter}(x) \leq x.rank , \tag{21.2}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq A_{\text{level}(x)}(x.rank) && \text{(by definition of level}(x)\text{)} \\ &= A^{(1)}_{\text{level}(x)}(x.rank) && \text{(by definition of functional iteration)} , \end{aligned}$$

which implies that $\text{iter}(x) \geq 1$, and we have

$$\begin{aligned} A^{(x.rank+1)}_{\text{level}(x)}(x.rank) &= A_{\text{level}(x)+1}(x.rank) && \text{(by definition of } A_k(j)\text{)} \\ &> x.p.rank && \text{(by definition of level}(x)\text{)} , \end{aligned}$$

which implies that $\text{iter}(x) \leq x.rank$. Note that because $x.p.rank$ monotonically increases over time, in order for $\text{iter}(x)$ to decrease, $\text{level}(x)$ must increase. As long as $\text{level}(x)$ remains unchanged, $\text{iter}(x)$ must either increase or remain unchanged.

With these auxiliary functions in place, we are ready to define the potential of node x after q operations:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0 , \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) & \text{if } x \text{ is not a root and } x.rank \geq 1 . \end{cases}$$

We next investigate some useful properties of node potentials.

Lemma 21.8

For every node x, and for all operation counts q, we have

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot x.rank .$$

Proof If x is a root or $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$ by definition. Now suppose that x is not a root and that $x.rank \geq 1$. We obtain a lower bound on $\phi_q(x)$ by maximizing $\text{level}(x)$ and $\text{iter}(x)$. By the bound (21.1), $\text{level}(x) \leq \alpha(n) - 1$, and by the bound (21.2), $\text{iter}(x) \leq x.rank$. Thus,

$$\begin{aligned}
\phi_q(x) &= (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) \\
&\geq (\alpha(n) - (\alpha(n) - 1)) \cdot x.rank - x.rank \\
&= x.rank - x.rank \\
&= 0 .
\end{aligned}$$

Similarly, we obtain an upper bound on $\phi_q(x)$ by minimizing $\text{level}(x)$ and $\text{iter}(x)$. By the bound (21.1), $\text{level}(x) \geq 0$, and by the bound (21.2), $\text{iter}(x) \geq 1$. Thus,

$$\begin{aligned}
\phi_q(x) &\leq (\alpha(n) - 0) \cdot x.rank - 1 \\
&= \alpha(n) \cdot x.rank - 1 \\
&< \alpha(n) \cdot x.rank .
\end{aligned}$$

■

Corollary 21.9
If node x is not a root and $x.rank > 0$, then $\phi_q(x) < \alpha(n) \cdot x.rank$. ■

Potential changes and amortized costs of operations

We are now ready to examine how the disjoint-set operations affect node potentials. With an understanding of the change in potential due to each operation, we can determine each operation's amortized cost.

Lemma 21.10
Let x be a node that is not a root, and suppose that the qth operation is either a LINK or FIND-SET. Then after the qth operation, $\phi_q(x) \leq \phi_{q-1}(x)$. Moreover, if $x.rank \geq 1$ and either $\text{level}(x)$ or $\text{iter}(x)$ changes due to the qth operation, then $\phi_q(x) \leq \phi_{q-1}(x) - 1$. That is, x's potential cannot increase, and if it has positive rank and either $\text{level}(x)$ or $\text{iter}(x)$ changes, then x's potential drops by at least 1.

Proof Because x is not a root, the qth operation does not change $x.rank$, and because n does not change after the initial n MAKE-SET operations, $\alpha(n)$ remains unchanged as well. Hence, these components of the formula for x's potential remain the same after the qth operation. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Now assume that $x.rank \geq 1$.

Recall that $\text{level}(x)$ monotonically increases over time. If the qth operation leaves $\text{level}(x)$ unchanged, then $\text{iter}(x)$ either increases or remains unchanged. If both $\text{level}(x)$ and $\text{iter}(x)$ are unchanged, then $\phi_q(x) = \phi_{q-1}(x)$. If $\text{level}(x)$

is unchanged and iter(x) increases, then it increases by at least 1, and so $\phi_q(x) \le \phi_{q-1}(x) - 1$.

Finally, if the qth operation increases level(x), it increases by at least 1, so that the value of the term $(\alpha(n) - \text{level}(x)) \cdot x.rank$ drops by at least $x.rank$. Because level(x) increased, the value of iter(x) might drop, but according to the bound (21.2), the drop is by at most $x.rank - 1$. Thus, the increase in potential due to the change in iter(x) is less than the decrease in potential due to the change in level(x), and we conclude that $\phi_q(x) \le \phi_{q-1}(x) - 1$. ■

Our final three lemmas show that the amortized cost of each MAKE-SET, LINK, and FIND-SET operation is $O(\alpha(n))$. Recall from equation (17.2) that the amortized cost of each operation is its actual cost plus the increase in potential due to the operation.

Lemma 21.11
The amortized cost of each MAKE-SET operation is $O(1)$.

Proof Suppose that the qth operation is MAKE-SET(x). This operation creates node x with rank 0, so that $\phi_q(x) = 0$. No other ranks or potentials change, and so $\Phi_q = \Phi_{q-1}$. Noting that the actual cost of the MAKE-SET operation is $O(1)$ completes the proof. ■

Lemma 21.12
The amortized cost of each LINK operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is LINK(x, y). The actual cost of the LINK operation is $O(1)$. Without loss of generality, suppose that the LINK makes y the parent of x.

To determine the change in potential due to the LINK, we note that the only nodes whose potentials may change are x, y, and the children of y just prior to the operation. We shall show that the only node whose potential can increase due to the LINK is y, and that its increase is at most $\alpha(n)$:

- By Lemma 21.10, any node that is y's child just before the LINK cannot have its potential increase due to the LINK.
- From the definition of $\phi_q(x)$, we see that, since x was a root just before the qth operation, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Otherwise,

  $$\begin{aligned} \phi_q(x) &< \alpha(n) \cdot x.rank \quad \text{(by Corollary 21.9)} \\ &= \phi_{q-1}(x) , \end{aligned}$$

  and so x's potential decreases.

- Because y is a root prior to the LINK, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. The LINK operation leaves y as a root, and it either leaves y's rank alone or it increases y's rank by 1. Therefore, either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

The increase in potential due to the LINK operation, therefore, is at most $\alpha(n)$. The amortized cost of the LINK operation is $O(1) + \alpha(n) = O(\alpha(n))$. ■

Lemma 21.13
The amortized cost of each FIND-SET operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is a FIND-SET and that the find path contains s nodes. The actual cost of the FIND-SET operation is $O(s)$. We shall show that no node's potential increases due to the FIND-SET and that at least $\max(0, s - (\alpha(n) + 2))$ nodes on the find path have their potential decrease by at least 1.

To see that no node's potential increases, we first appeal to Lemma 21.10 for all nodes other than the root. If x is the root, then its potential is $\alpha(n) \cdot x.rank$, which does not change.

Now we show that at least $\max(0, s - (\alpha(n) + 2))$ nodes have their potential decrease by at least 1. Let x be a node on the find path such that $x.rank > 0$ and x is followed somewhere on the find path by another node y that is not a root, where $\text{level}(y) = \text{level}(x)$ just before the FIND-SET operation. (Node y need not *immediately* follow x on the find path.) All but at most $\alpha(n) + 2$ nodes on the find path satisfy these constraints on x. Those that do not satisfy them are the first node on the find path (if it has rank 0), the last node on the path (i.e., the root), and the last node w on the path for which $\text{level}(w) = k$, for each $k = 0, 1, 2, \ldots, \alpha(n) - 1$.

Let us fix such a node x, and we shall show that x's potential decreases by at least 1. Let $k = \text{level}(x) = \text{level}(y)$. Just prior to the path compression caused by the FIND-SET, we have

$$\begin{aligned}
x.p.rank &\geq A_k^{(\text{iter}(x))}(x.rank) && \text{(by definition of iter}(x)) \ , \\
y.p.rank &\geq A_k(y.rank) && \text{(by definition of level}(y)) \ , \\
y.rank &\geq x.p.rank && \text{(by Corollary 21.5 and because} \\
& && \quad y \text{ follows } x \text{ on the find path)} \ .
\end{aligned}$$

Putting these inequalities together and letting i be the value of $\text{iter}(x)$ before path compression, we have

$$\begin{aligned}
y.p.rank &\geq A_k(y.rank) \\
&\geq A_k(x.p.rank) && \text{(because } A_k(j) \text{ is strictly increasing)} \\
&\geq A_k(A_k^{(\text{iter}(x))}(x.rank)) \\
&= A_k^{(i+1)}(x.rank) \ .
\end{aligned}$$

Because path compression will make x and y have the same parent, we know that after path compression, $x.p.rank = y.p.rank$ and that the path compression does not decrease $y.p.rank$. Since $x.rank$ does not change, after path compression we have that $x.p.rank \geq A_k^{(i+1)}(x.rank)$. Thus, path compression will cause either $\text{iter}(x)$ to increase (to at least $i + 1$) or $\text{level}(x)$ to increase (which occurs if $\text{iter}(x)$ increases to at least $x.rank + 1$). In either case, by Lemma 21.10, we have $\phi_q(x) \leq \phi_{q-1}(x) - 1$. Hence, x's potential decreases by at least 1.

The amortized cost of the FIND-SET operation is the actual cost plus the change in potential. The actual cost is $O(s)$, and we have shown that the total potential decreases by at least $\max(0, s - (\alpha(n) + 2))$. The amortized cost, therefore, is at most $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, since we can scale up the units of potential to dominate the constant hidden in $O(s)$. ■

Putting the preceding lemmas together yields the following theorem.

Theorem 21.14
A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\,\alpha(n))$.

Proof Immediate from Lemmas 21.7, 21.11, 21.12, and 21.13. ■

Exercises

21.4-1
Prove Lemma 21.4.

21.4-2
Prove that every node has rank at most $\lfloor \lg n \rfloor$.

21.4-3
In light of Exercise 21.4-2, how many bits are necessary to store $x.rank$ for each node x?

21.4-4
Using Exercise 21.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

21.4-5
Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other

words, if $x.rank > 0$ and $x.p$ is not a root, then $\text{level}(x) \leq \text{level}(x.p)$. Is the professor correct?

21.4-6 ⋆

Consider the function $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n+1)\}$. Show that $\alpha'(n) \leq 3$ for all practical values of n and, using Exercise 21.4-2, show how to modify the potential-function argument to prove that we can perform a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\,\alpha'(n))$.

Problems

21-1 Off-line minimum

The ***off-line minimum problem*** asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array $extracted[1..m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ is the key returned by the ith EXTRACT-MIN call. The problem is "off-line" in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.

a. In the following instance of the off-line minimum problem, each operation INSERT(i) is represented by the value of i and each EXTRACT-MIN is represented by the letter E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5 .

Fill in the correct values in the *extracted* array.

To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by

$$\mathrm{I}_1, \mathrm{E}, \mathrm{I}_2, \mathrm{E}, \mathrm{I}_3, \ldots, \mathrm{I}_m, \mathrm{E}, \mathrm{I}_{m+1} ,$$

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j, we initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. We then do the following:

```
OFF-LINE-MINIMUM(m, n)
1  for i = 1 to n
2      determine j such that i ∈ K_j
3      if j ≠ m + 1
4          extracted[j] = i
5          let l be the smallest value greater than j
               for which set K_l exists
6          K_l = K_j ∪ K_l, destroying K_j
7  return extracted
```

b. Argue that the array *extracted* returned by OFF-LINE-MINIMUM is correct.

c. Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

21-2 Depth determination

In the ***depth-determination problem***, we maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE(v) creates a tree whose only node is v.

FIND-DEPTH(v) returns the depth of node v within its tree.

GRAFT(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree than r but may or may not itself be a root.

a. Suppose that we use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that we implement GRAFT(r, v) by setting $r.p = v$ and FIND-DEPTH(v) by following the find path up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, we can reduce the worst-case running time. We use the disjoint-set forest $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{F}$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows us to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the

root of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

d. Show how to implement GRAFT(r, v), which combines the sets containing r and v, by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

e. Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

21-3 Tarjan's off-line least-common-ancestors algorithm

The ***least common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***off-line least-common-ancestors problem***, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and we wish to determine the least common ancestor of each pair in P.

To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call LCA($T.root$). We assume that each node is colored WHITE prior to the walk.

```
LCA(u)
 1  MAKE-SET(u)
 2  FIND-SET(u).ancestor = u
 3  for each child v of u in T
 4      LCA(v)
 5      UNION(u, v)
 6      FIND-SET(u).ancestor = u
 7  u.color = BLACK
 8  for each node v such that {u, v} ∈ P
 9      if v.color == BLACK
10          print "The least common ancestor of"
                u "and" v "is" FIND-SET(v).ancestor
```

a. Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.

b. Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equals the depth of u in T.

c. Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.

d. Analyze the running time of LCA, assuming that we use the implementation of the disjoint-set data structure in Section 21.3.

Chapter notes

Many of the important results for disjoint-set data structures are due at least in part to R. E. Tarjan. Using aggregate analysis, Tarjan [328, 330] gave the first tight upper bound in terms of the very slowly growing inverse $\hat{\alpha}(m, n)$ of Ackermann's function. (The function $A_k(j)$ given in Section 21.4 is similar to Ackermann's function, and the function $\alpha(n)$ is similar to the inverse. Both $\alpha(n)$ and $\hat{\alpha}(m, n)$ are at most 4 for all conceivable values of m and n.) An $O(m \lg^* n)$ upper bound was proven earlier by Hopcroft and Ullman [5, 179]. The treatment in Section 21.4 is adapted from a later analysis by Tarjan [332], which is in turn based on an analysis by Kozen [220]. Harfst and Reingold [161] give a potential-based version of Tarjan's earlier bound.

Tarjan and van Leeuwen [333] discuss variants on the path-compression heuristic, including "one-pass methods," which sometimes offer better constant factors in their performance than do two-pass methods. As with Tarjan's earlier analyses of the basic path-compression heuristic, the analyses by Tarjan and van Leeuwen are aggregate. Harfst and Reingold [161] later showed how to make a small change to the potential function to adapt their path-compression analysis to these one-pass variants. Gabow and Tarjan [121] show that in certain applications, the disjoint-set operations can be made to run in $O(m)$ time.

Tarjan [329] showed that a lower bound of $\Omega(m\, \hat{\alpha}(m, n))$ time is required for operations on any disjoint-set data structure satisfying certain technical conditions. This lower bound was later generalized by Fredman and Saks [113], who showed that in the worst case, $\Omega(m\, \hat{\alpha}(m, n))$ $(\lg n)$-bit words of memory must be accessed.