

---

## 19 Binomial Heaps

This chapter and Chapter 20 present data structures known as *mergeable heaps*, which support the following five operations.

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

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

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

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

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

In addition, the data structures in these chapters also support the following two operations.

DECREASE-KEY( $H, x, k$ ) assigns to node  $x$  within heap  $H$  the new key value  $k$ , which is assumed to be no greater than its current key value.<sup>1</sup>

DELETE( $H, x$ ) deletes node  $x$  from heap  $H$ .

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 well. Operations other than UNION run in worst-case time  $O(\lg n)$  (or better) on a binary heap. If the UNION operation must be supported, however, binary heaps perform poorly. By concatenating the two arrays that hold the binary heaps to be merged and then running MIN-HEAPIFY (see Exercise 6.2-2), the UNION operation takes  $\Theta(n)$  time in the worst case.

---

<sup>1</sup>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)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

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

In this chapter, we examine “binomial heaps,” whose worst-case time bounds are also shown in Figure 19.1. In particular, the UNION operation takes only  $O(\lg n)$  time to merge two binomial heaps with a total of  $n$  elements.

In Chapter 20, we shall explore Fibonacci heaps, which have even better time bounds for some 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.

This chapter ignores issues of allocating nodes prior to insertion and freeing nodes following deletion. We assume that the code that calls the heap procedures deals with these details.

Binary heaps, binomial heaps, and Fibonacci heaps are all inefficient in their support of the operation SEARCH; it can take a while to find a node with a given key. For this reason, operations such as DECREASE-KEY and DELETE that refer to a given node require a pointer to that node 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 corresponding mergeable-heap element in each application object. The exact nature of these handles depends on the application and its implementation.

Section 19.1 defines binomial heaps after first defining their constituent binomial trees. It also introduces a particular representation of binomial heaps. Section 19.2 shows how we can implement operations on binomial heaps in the time bounds given in Figure 19.1.

---

## 19.1 Binomial trees and binomial heaps

A binomial heap is a collection of binomial trees, so this section starts by defining binomial trees and proving some key properties. We then define binomial heaps and show how they can be represented.

### 19.1.1 Binomial trees

The *binomial tree*  $B_k$  is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.2(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: the root of one is the leftmost child of the root of the other. Figure 19.2(b) shows the binomial trees  $B_0$  through  $B_4$ .

Some properties of binomial trees are given by the following lemma.

**Lemma 19.1 (Properties of binomial trees)**

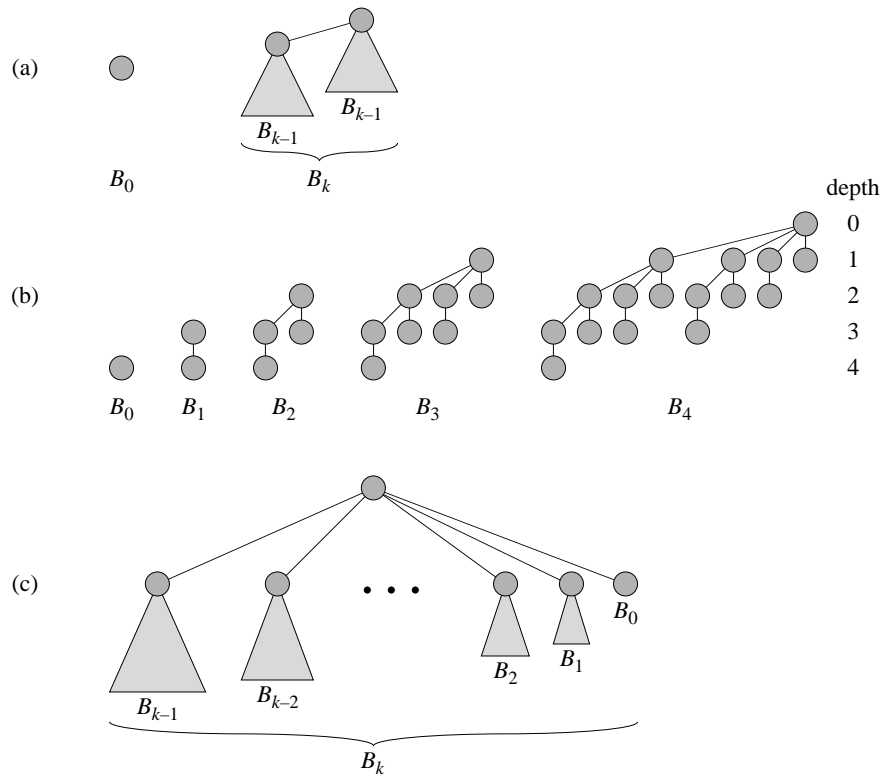
For the binomial tree  $B_k$ ,

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  $i = 0, 1, \dots, k$ , and
4. the root has degree  $k$ , which is greater than that of any other node; moreover if the children of the root are numbered from left to right by  $k - 1, k - 2, \dots, 0$ , child  $i$  is the root of a subtree  $B_i$ .

**Proof** The proof is by induction on  $k$ . For each property, the basis is the binomial tree  $B_0$ . Verifying that each property holds for  $B_0$  is trivial.

For the inductive step, we assume that the lemma holds for  $B_{k-1}$ .

1. Binomial tree  $B_k$  consists of two copies of  $B_{k-1}$ , and so  $B_k$  has  $2^{k-1} + 2^{k-1} = 2^k$  nodes.
2. Because of the way in which the two copies of  $B_{k-1}$  are linked to form  $B_k$ , the maximum depth of a node in  $B_k$  is one greater than the maximum depth in  $B_{k-1}$ . By the inductive hypothesis, this maximum depth is  $(k - 1) + 1 = k$ .
3. Let  $D(k, i)$  be the number of nodes at depth  $i$  of binomial tree  $B_k$ . Since  $B_k$  is composed of two copies of  $B_{k-1}$  linked together, a node at depth  $i$  in  $B_{k-1}$  appears in  $B_k$  once at depth  $i$  and once at depth  $i + 1$ . In other words, the number of nodes at depth  $i$  in  $B_k$  is the number of nodes at depth  $i$  in  $B_{k-1}$  plus



**Figure 19.2** (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  $B_k$ .

the number of nodes at depth  $i - 1$  in  $B_{k-1}$ . Thus,

$$\begin{aligned}
 D(k, i) &= D(k-1, i) + D(k-1, i-1) \quad (\text{by the inductive hypothesis}) \\
 &= \binom{k-1}{i} + \binom{k-1}{i-1} \quad (\text{by Exercise C.1-7}) \\
 &= \binom{k}{i}.
 \end{aligned}$$

4. The only node with greater degree in  $B_k$  than in  $B_{k-1}$  is the root, which has one more child than in  $B_{k-1}$ . Since the root of  $B_{k-1}$  has degree  $k - 1$ , the root of  $B_k$  has degree  $k$ . Now, by the inductive hypothesis, and as Figure 19.2(c) shows, from left to right, the children of the root of  $B_{k-1}$  are roots of  $B_{k-2}, B_{k-3}, \dots, B_0$ . When  $B_{k-1}$  is linked to  $B_{k-1}$ , therefore, the children of the resulting root are roots of  $B_{k-1}, B_{k-2}, \dots, B_0$ . ■

**Corollary 19.2**

The maximum degree of any node in an  $n$ -node binomial tree is  $\lg n$ .

**Proof** Immediate from properties 1 and 4 of Lemma 19.1. ■

The term “binomial tree” comes from property 3 of Lemma 19.1, since the terms  $\binom{k}{i}$  are the binomial coefficients. Exercise 19.1-3 gives further justification for the term.

**19.1.2 Binomial heaps**

A **binomial heap**  $H$  is a set of binomial trees that satisfies the following **binomial-heap properties**.

1. Each binomial tree in  $H$  obeys the **min-heap property**: the key of a node is greater than or equal to the key of its parent. We say that each such tree is **min-heap-ordered**.
2. For any nonnegative integer  $k$ , there is at most one binomial tree in  $H$  whose root has degree  $k$ .

The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree.

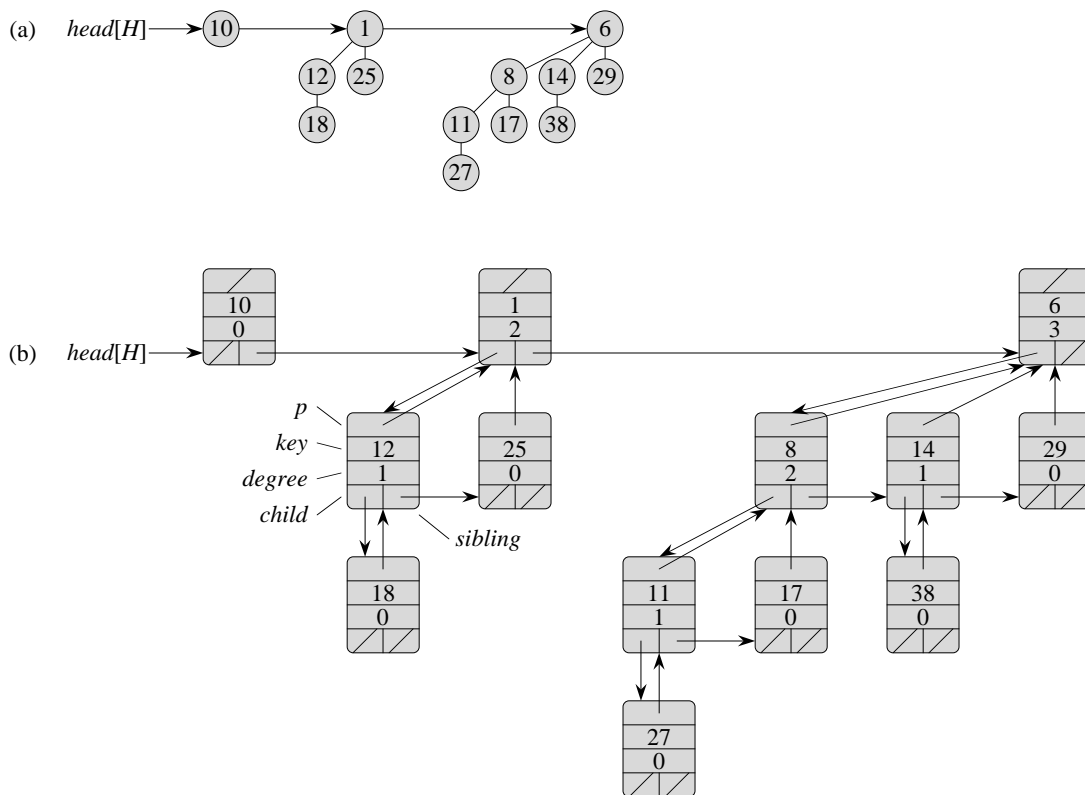
The second property implies that an  $n$ -node binomial heap  $H$  consists of at most  $\lfloor \lg n \rfloor + 1$  binomial trees. To see why, observe that the binary representation of  $n$  has  $\lfloor \lg n \rfloor + 1$  bits, say  $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \dots, b_0 \rangle$ , so that  $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$ . By property 1 of Lemma 19.1, therefore, binomial tree  $B_i$  appears in  $H$  if and only if bit  $b_i = 1$ . Thus, binomial heap  $H$  contains at most  $\lfloor \lg n \rfloor + 1$  binomial trees.

Figure 19.3(a) shows a binomial heap  $H$  with 13 nodes. The binary representation of 13 is  $\langle 1101 \rangle$ , and  $H$  consists of min-heap-ordered binomial trees  $B_3$ ,  $B_2$ , and  $B_0$ , having 8, 4, and 1 nodes respectively, for a total of 13 nodes.

**Representing binomial heaps**

As shown in Figure 19.3(b), each binomial tree within a binomial heap is stored in the left-child, right-sibling representation of Section 10.4. Each node has a *key* field and any other satellite information required by the application. In addition, each node  $x$  contains pointers  $p[x]$  to its parent,  $child[x]$  to its leftmost child, and  $sibling[x]$  to the sibling of  $x$  immediately to its right. If node  $x$  is a root, then  $p[x] = \text{NIL}$ . If node  $x$  has no children, then  $child[x] = \text{NIL}$ , and if  $x$  is the rightmost child of its parent, then  $sibling[x] = \text{NIL}$ . Each node  $x$  also contains the field  $degree[x]$ , which is the number of children of  $x$ .

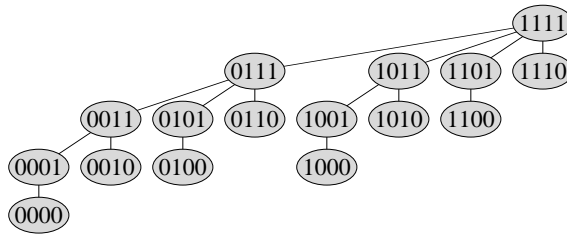
As Figure 19.3 also shows, the roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the **root list**. The degrees



**Figure 19.3** A binomial heap  $H$  with  $n = 13$  nodes. (a) The heap consists of binomial trees  $B_0$ ,  $B_2$ , and  $B_3$ , which have 1, 4, and 8 nodes respectively, totaling  $n = 13$  nodes. Since each binomial tree is min-heap-ordered, the key of any node is no less than the key of its parent. Also shown is the root list, which is a linked list of roots in order of increasing degree. (b) A more detailed representation of binomial heap  $H$ . Each binomial tree is stored in the left-child, right-sibling representation, and each node stores its degree.

of the roots strictly increase as we traverse the root list. By the second binomial-heap property, in an  $n$ -node binomial heap the degrees of the roots are a subset of  $\{0, 1, \dots, \lfloor \lg n \rfloor\}$ . The *sibling* field has a different meaning for roots than for nonroots. If  $x$  is a root, then *sibling* $[x]$  points to the next root in the root list. (As usual, *sibling* $[x] = \text{NIL}$  if  $x$  is the last root in the root list.)

A given binomial heap  $H$  is accessed by the field *head* $[H]$ , which is simply a pointer to the first root in the root list of  $H$ . If binomial heap  $H$  has no elements, then *head* $[H] = \text{NIL}$ .



**Figure 19.4** The binomial tree  $B_4$  with nodes labeled in binary by a postorder walk.

## Exercises

### 19.1-1

Suppose that  $x$  is a node in a binomial tree within a binomial heap, and assume that  $\text{sibling}[x] \neq \text{NIL}$ . If  $x$  is not a root, how does  $\text{degree}[\text{sibling}[x]]$  compare to  $\text{degree}[x]$ ? How about if  $x$  is a root?

### 19.1-2

If  $x$  is a nonroot node in a binomial tree within a binomial heap, how does  $\text{degree}[x]$  compare to  $\text{degree}[p[x]]$ ?

### 19.1-3

Suppose we label the nodes of binomial tree  $B_k$  in binary by a postorder walk, as in Figure 19.4. Consider a node  $x$  labeled  $l$  at depth  $i$ , and let  $j = k - i$ . Show that  $x$  has  $j$  1's in its binary representation. How many binary  $k$ -strings are there that contain exactly  $j$  1's? Show that the degree of  $x$  is equal to the number of 1's to the right of the rightmost 0 in the binary representation of  $l$ .

---

## 19.2 Operations on binomial heaps

In this section, we show how to perform operations on binomial heaps in the time bounds shown in Figure 19.1. We shall only show the upper bounds; the lower bounds are left as Exercise 19.2-10.

### Creating a new binomial heap

To make an empty binomial heap, the `MAKE-BINOMIAL-HEAP` procedure simply allocates and returns an object  $H$ , where  $\text{head}[H] = \text{NIL}$ . The running time is  $\Theta(1)$ .

**Finding the minimum key**

The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an  $n$ -node binomial heap  $H$ . This implementation assumes that there are no keys with value  $\infty$ . (See Exercise 19.2-5.)

BINOMIAL-HEAP-MINIMUM( $H$ )

```

1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{head}[H]$ 
3   $\text{min} \leftarrow \infty$ 
4  while  $x \neq \text{NIL}$ 
5      do if  $\text{key}[x] < \text{min}$ 
6          then  $\text{min} \leftarrow \text{key}[x]$ 
7               $y \leftarrow x$ 
8           $x \leftarrow \text{sibling}[x]$ 
9  return  $y$ 

```

Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number at most  $\lfloor \lg n \rfloor + 1$ , saving the current minimum in  $\text{min}$  and a pointer to the current minimum in  $y$ . When called on the binomial heap of Figure 19.3, BINOMIAL-HEAP-MINIMUM returns a pointer to the node with key 1.

Because there are at most  $\lfloor \lg n \rfloor + 1$  roots to check, the running time of BINOMIAL-HEAP-MINIMUM is  $O(\lg n)$ .

**Uniting two binomial heaps**

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the  $B_{k-1}$  tree rooted at node  $y$  to the  $B_{k-1}$  tree rooted at node  $z$ ; that is, it makes  $z$  the parent of  $y$ . Node  $z$  thus becomes the root of a  $B_k$  tree.

BINOMIAL-LINK( $y, z$ )

```

1   $p[y] \leftarrow z$ 
2   $\text{sibling}[y] \leftarrow \text{child}[z]$ 
3   $\text{child}[z] \leftarrow y$ 
4   $\text{degree}[z] \leftarrow \text{degree}[z] + 1$ 

```

The BINOMIAL-LINK procedure makes node  $y$  the new head of the linked list of node  $z$ 's children in  $O(1)$  time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a  $B_k$  tree, the leftmost child of the root is the root of a  $B_{k-1}$  tree.



The following procedure unites binomial heaps  $H_1$  and  $H_2$ , returning the resulting heap. It destroys the representations of  $H_1$  and  $H_2$  in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIAL-HEAP-MERGE that merges the root lists of  $H_1$  and  $H_2$  into a single linked list that is sorted by degree into monotonically increasing order. The BINOMIAL-HEAP-MERGE procedure, whose pseudocode we leave as Exercise 19.2-1, is similar to the MERGE procedure in Section 2.3.1.

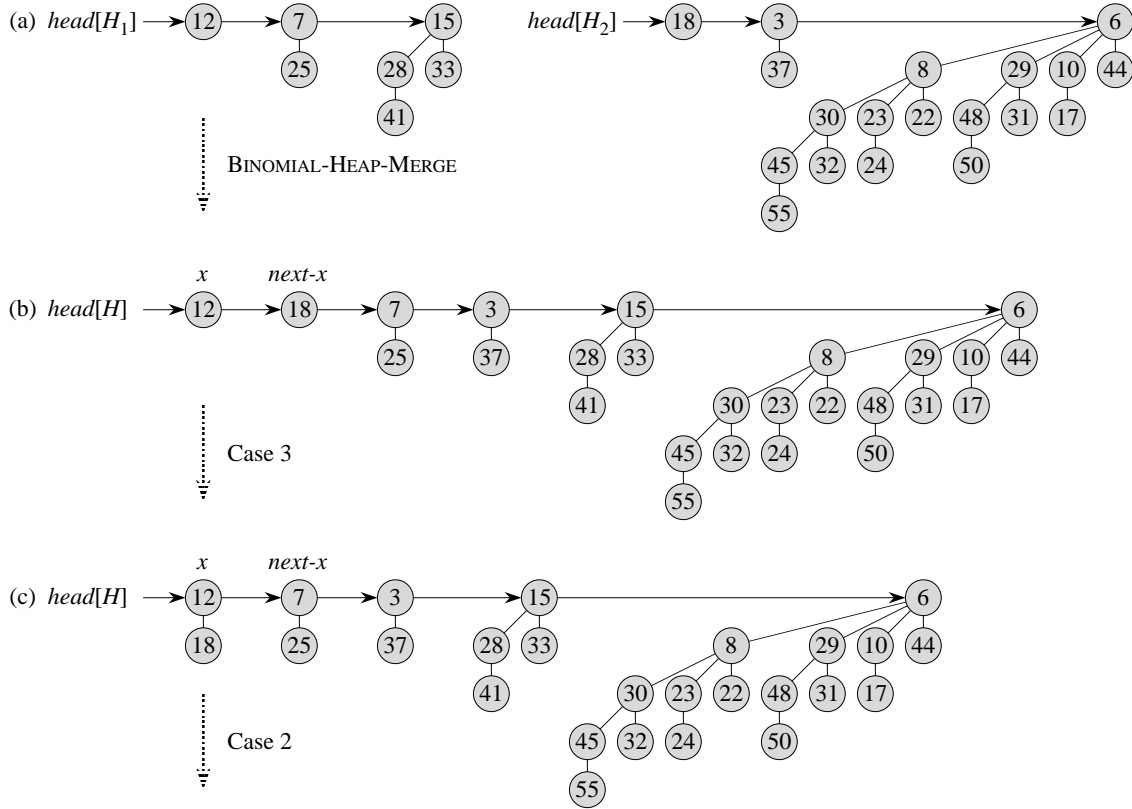
```

BINOMIAL-HEAP-UNION( $H_1, H_2$ )
1   $H \leftarrow$  MAKE-BINOMIAL-HEAP()
2   $head[H] \leftarrow$  BINOMIAL-HEAP-MERGE( $H_1, H_2$ )
3  free the objects  $H_1$  and  $H_2$  but not the lists they point to
4  if  $head[H] = \text{NIL}$ 
5      then return  $H$ 
6   $prev-x \leftarrow \text{NIL}$ 
7   $x \leftarrow head[H]$ 
8   $next-x \leftarrow sibling[x]$ 
9  while  $next-x \neq \text{NIL}$ 
10     do if ( $degree[x] \neq degree[next-x]$ ) or
           ( $sibling[next-x] \neq \text{NIL}$  and  $degree[sibling[next-x]] = degree[x]$ )
11         then  $prev-x \leftarrow x$                                 ▷ Cases 1 and 2
12              $x \leftarrow next-x$                                 ▷ Cases 1 and 2
13         else if  $key[x] \leq key[next-x]$ 
14             then  $sibling[x] \leftarrow sibling[next-x]$           ▷ Case 3
15                 BINOMIAL-LINK( $next-x, x$ )                    ▷ Case 3
16         else if  $prev-x = \text{NIL}$                                 ▷ Case 4
17             then  $head[H] \leftarrow next-x$                     ▷ Case 4
18                 else  $sibling[prev-x] \leftarrow next-x$         ▷ Case 4
19                     BINOMIAL-LINK( $x, next-x$ )                ▷ Case 4
20                      $x \leftarrow next-x$                         ▷ Case 4
21              $next-x \leftarrow sibling[x]$ 
22 return  $H$ 

```

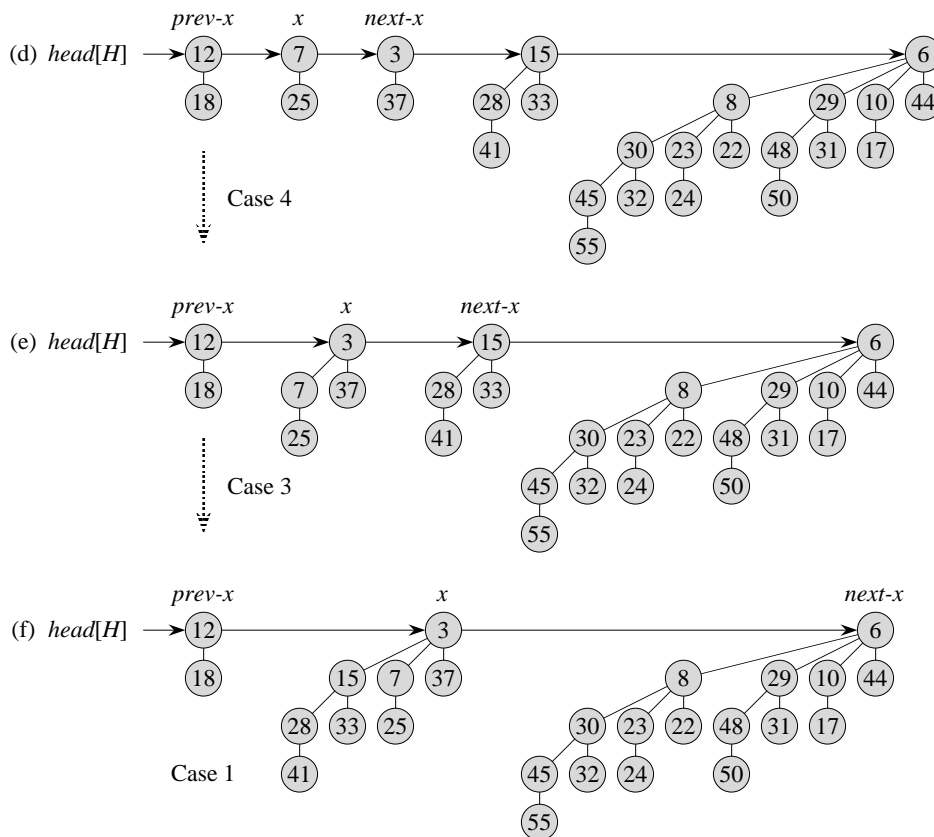
Figure 19.5 shows an example of BINOMIAL-HEAP-UNION in which all four cases given in the pseudocode occur.

The BINOMIAL-HEAP-UNION procedure has two phases. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps  $H_1$  and  $H_2$  into a single linked list  $H$  that is sorted by degree into monotonically increasing order. There might be as many as two roots (but no more) of each degree, however, so the second phase links roots of equal degree until at most one root remains of each degree. Because the linked list  $H$  is sorted by degree, we can perform all the link operations quickly.



**Figure 19.5** The execution of BINOMIAL-HEAP-UNION. **(a)** Binomial heaps  $H_1$  and  $H_2$ . **(b)** Binomial heap  $H$  is the output of BINOMIAL-HEAP-MERGE( $H_1, H_2$ ). Initially,  $x$  is the first root on the root list of  $H$ . Because both  $x$  and  $next-x$  have degree 0 and  $key[x] < key[next-x]$ , case 3 applies. **(c)** After the link occurs,  $x$  is the first of three roots with the same degree, so case 2 applies. **(d)** After all the pointers move down one position in the root list, case 4 applies, since  $x$  is the first of two roots of equal degree. **(e)** After the link occurs, case 3 applies. **(f)** After another link, case 1 applies, because  $x$  has degree 3 and  $next-x$  has degree 4. This iteration of the **while** loop is the last, because after the pointers move down one position in the root list,  $next-x = \text{NIL}$ .

In detail, the procedure works as follows. Lines 1–3 start by merging the root lists of binomial heaps  $H_1$  and  $H_2$  into a single root list  $H$ . The root lists of  $H_1$  and  $H_2$  are sorted by strictly increasing degree, and BINOMIAL-HEAP-MERGE returns a root list  $H$  that is sorted by monotonically increasing degree. If the root lists of  $H_1$  and  $H_2$  have  $m$  roots altogether, BINOMIAL-HEAP-MERGE runs in  $O(m)$  time by repeatedly examining the roots at the heads of the two root lists and appending the root with the lower degree to the output root list, removing it from its input root list in the process.



The BINOMIAL-HEAP-UNION procedure next initializes some pointers into the root list of  $H$ . First, it simply returns in lines 4–5 if it happens to be uniting two empty binomial heaps. From line 6 on, therefore, we know that  $H$  has at least one root. Throughout the procedure, we maintain three pointers into the root list:

- $x$  points to the root currently being examined,
- $prev-x$  points to the root preceding  $x$  on the root list:  $sibling[prev-x] = x$  (since initially  $x$  has no predecessor, we start with  $prev-x$  set to NIL), and
- $next-x$  points to the root following  $x$  on the root list:  $sibling[x] = next-x$ .

Initially, there are at most two roots on the root list  $H$  of a given degree: because  $H_1$  and  $H_2$  were binomial heaps, they each had at most one root of a given degree. Moreover, BINOMIAL-HEAP-MERGE guarantees us that if two roots in  $H$  have the same degree, they are adjacent in the root list.

In fact, during the execution of BINOMIAL-HEAP-UNION, there may be three roots of a given degree appearing on the root list  $H$  at some time. We shall see

in a moment how this situation could occur. At each iteration of the **while** loop of lines 9–21, therefore, we decide whether to link  $x$  and  $next-x$  based on their degrees and possibly the degree of  $sibling[next-x]$ . An invariant of the loop is that each time we start the body of the loop, both  $x$  and  $next-x$  are non-NIL. (See Exercise 19.2-4 for a precise loop invariant.)

Case 1, shown in Figure 19.6(a), occurs when  $degree[x] \neq degree[next-x]$ , that is, when  $x$  is the root of a  $B_k$ -tree and  $next-x$  is the root of a  $B_l$ -tree for some  $l > k$ . Lines 11–12 handle this case. We don't link  $x$  and  $next-x$ , so we simply march the pointers one position farther down the list. Updating  $next-x$  to point to the node following the new node  $x$  is handled in line 21, which is common to every case.

Case 2, shown in Figure 19.6(b), occurs when  $x$  is the first of three roots of equal degree, that is, when

$$degree[x] = degree[next-x] = degree[sibling[next-x]] .$$

We handle this case in the same manner as case 1: we just march the pointers one position farther down the list. The next iteration will execute either case 3 or case 4 to combine the second and third of the three equal-degree roots. Line 10 tests for both cases 1 and 2, and lines 11–12 handle both cases.

Cases 3 and 4 occur when  $x$  is the first of two roots of equal degree, that is, when

$$degree[x] = degree[next-x] \neq degree[sibling[next-x]] .$$

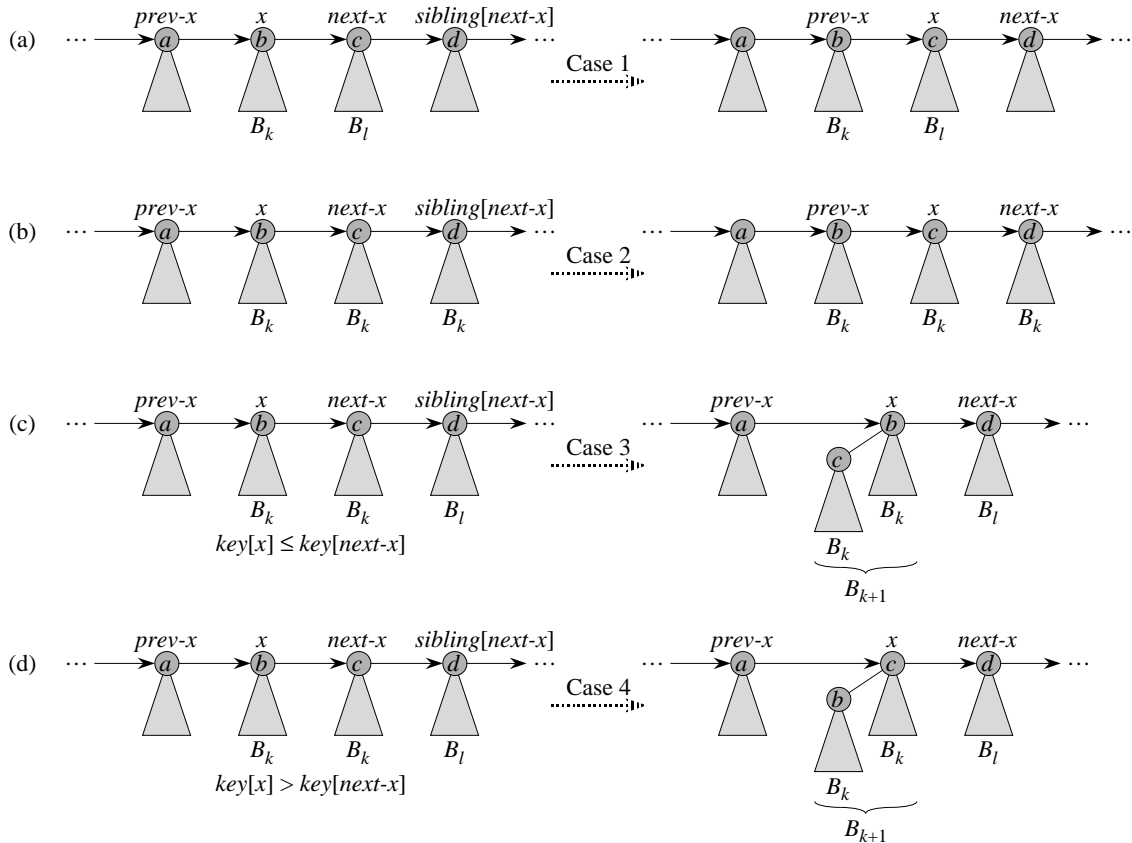
These cases may occur in any iteration, but one of them always occurs immediately following case 2. In cases 3 and 4, we link  $x$  and  $next-x$ . The two cases are distinguished by whether  $x$  or  $next-x$  has the smaller key, which determines the node that will be the root after the two are linked.

In case 3, shown in Figure 19.6(c),  $key[x] \leq key[next-x]$ , so  $next-x$  is linked to  $x$ . Line 14 removes  $next-x$  from the root list, and line 15 makes  $next-x$  the leftmost child of  $x$ .

In case 4, shown in Figure 19.6(d),  $next-x$  has the smaller key, so  $x$  is linked to  $next-x$ . Lines 16–18 remove  $x$  from the root list; there are two cases depending on whether  $x$  is the first root on the list (line 17) or is not (line 18). Line 19 then makes  $x$  the leftmost child of  $next-x$ , and line 20 updates  $x$  for the next iteration.

Following either case 3 or case 4, the setup for the next iteration of the **while** loop is the same. We have just linked two  $B_k$ -trees to form a  $B_{k+1}$ -tree, which  $x$  now points to. There were already zero, one, or two other  $B_{k+1}$ -trees on the root list resulting from BINOMIAL-HEAP-MERGE, so  $x$  is now the first of either one, two, or three  $B_{k+1}$ -trees on the root list. If  $x$  is the only one, then we enter case 1 in the next iteration:  $degree[x] \neq degree[next-x]$ . If  $x$  is the first of two, then we enter either case 3 or case 4 in the next iteration. It is when  $x$  is the first of three that we enter case 2 in the next iteration.

The running time of BINOMIAL-HEAP-UNION is  $O(\lg n)$ , where  $n$  is the total number of nodes in binomial heaps  $H_1$  and  $H_2$ . We can see this as follows. Let  $H_1$



**Figure 19.6** The four cases that occur in BINOMIAL-HEAP-UNION. Labels  $a, b, c,$  and  $d$  serve only to identify the roots involved; they do not indicate the degrees or keys of these roots. In each case,  $x$  is the root of a  $B_k$ -tree and  $l > k$ . **(a)** Case 1:  $degree[x] \neq degree[next-x]$ . The pointers move one position farther down the root list. **(b)** Case 2:  $degree[x] = degree[next-x] = degree[sibling[next-x]]$ . Again, the pointers move one position farther down the list, and the next iteration executes either case 3 or case 4. **(c)** Case 3:  $degree[x] = degree[next-x] \neq degree[sibling[next-x]]$  and  $key[x] \leq key[next-x]$ . We remove  $next-x$  from the root list and link it to  $x$ , creating a  $B_{k+1}$ -tree. **(d)** Case 4:  $degree[x] = degree[next-x] \neq degree[sibling[next-x]]$  and  $key[next-x] \leq key[x]$ . We remove  $x$  from the root list and link it to  $next-x$ , again creating a  $B_{k+1}$ -tree.

contain  $n_1$  nodes and  $H_2$  contain  $n_2$  nodes, so that  $n = n_1 + n_2$ . Then  $H_1$  contains at most  $\lceil \lg n_1 \rceil + 1$  roots and  $H_2$  contains at most  $\lceil \lg n_2 \rceil + 1$  roots, and so  $H$  contains at most  $\lceil \lg n_1 \rceil + \lceil \lg n_2 \rceil + 2 \leq 2 \lceil \lg n \rceil + 2 = O(\lg n)$  roots immediately after the call of BINOMIAL-HEAP-MERGE. The time to perform BINOMIAL-HEAP-MERGE is thus  $O(\lg n)$ . Each iteration of the **while** loop takes  $O(1)$  time, and there are at most  $\lceil \lg n_1 \rceil + \lceil \lg n_2 \rceil + 2$  iterations because each iteration either advances the

pointers one position down the root list of  $H$  or removes a root from the root list. The total time is thus  $O(\lg n)$ .

### Inserting a node

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

```

BINOMIAL-HEAP-INSERT( $H, x$ )
1   $H' \leftarrow$  MAKE-BINOMIAL-HEAP()
2   $p[x] \leftarrow$  NIL
3   $child[x] \leftarrow$  NIL
4   $sibling[x] \leftarrow$  NIL
5   $degree[x] \leftarrow$  0
6   $head[H'] \leftarrow x$ 
7   $H \leftarrow$  BINOMIAL-HEAP-UNION( $H, H'$ )

```

The procedure simply makes a one-node binomial heap  $H'$  in  $O(1)$  time and unites it with the  $n$ -node binomial heap  $H$  in  $O(\lg n)$  time. The call to BINOMIAL-HEAP-UNION takes care of freeing the temporary binomial heap  $H'$ . (A direct implementation that does not call BINOMIAL-HEAP-UNION is given as Exercise 19.2-8.)

### Extracting the node with minimum key

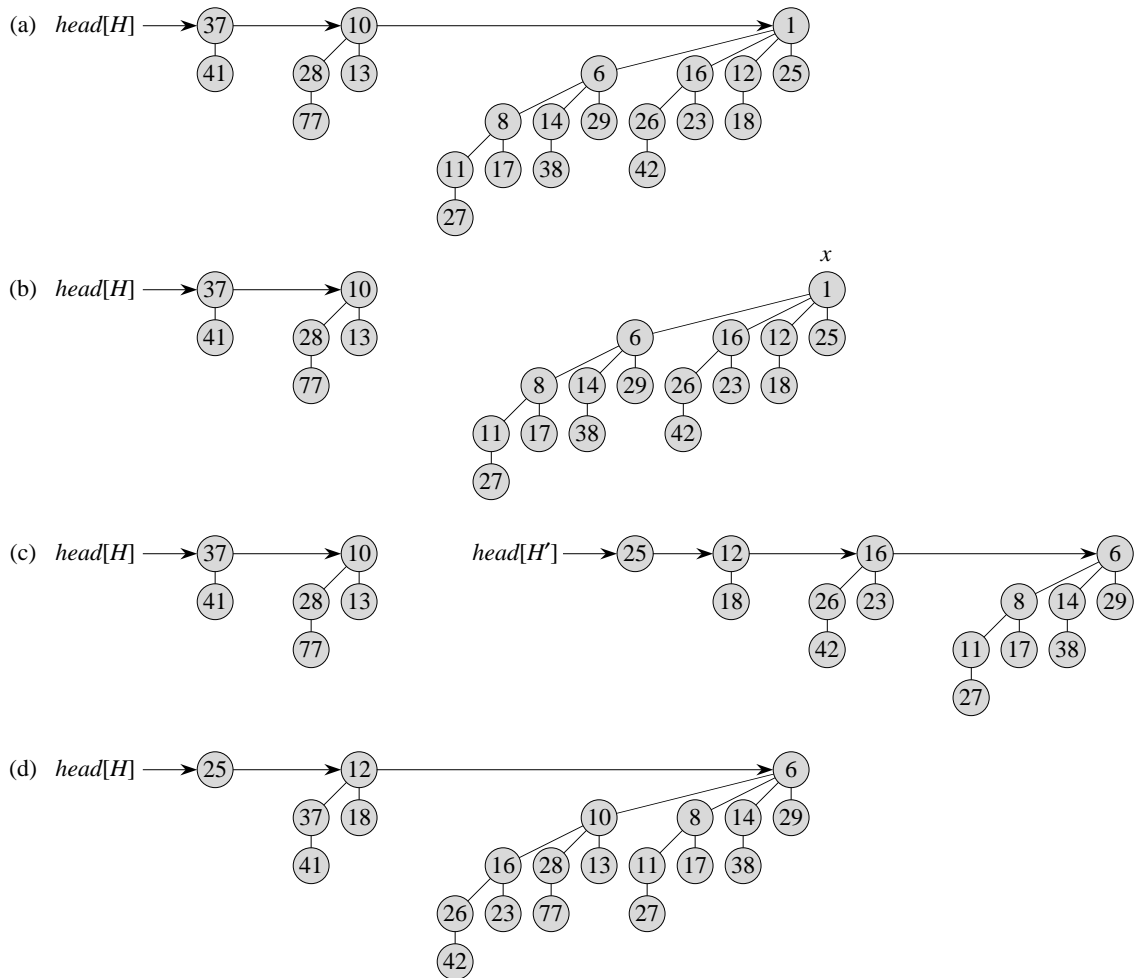
The following procedure extracts the node with the minimum key from binomial heap  $H$  and returns a pointer to the extracted node.

```

BINOMIAL-HEAP-EXTRACT-MIN( $H$ )
1  find the root  $x$  with the minimum key in the root list of  $H$ ,
   and remove  $x$  from the root list of  $H$ 
2   $H' \leftarrow$  MAKE-BINOMIAL-HEAP()
3  reverse the order of the linked list of  $x$ 's children,
   and set  $head[H']$  to point to the head of the resulting list
4   $H \leftarrow$  BINOMIAL-HEAP-UNION( $H, H'$ )
5  return  $x$ 

```

This procedure works as shown in Figure 19.7. The input binomial heap  $H$  is shown in Figure 19.7(a). Figure 19.7(b) shows the situation after line 1: the root  $x$  with the minimum key has been removed from the root list of  $H$ . If  $x$  is the root of a  $B_k$ -tree, then by property 4 of Lemma 19.1,  $x$ 's children, from left to right, are roots of  $B_{k-1}$ -,  $B_{k-2}$ -,  $\dots$ ,  $B_0$ -trees. Figure 19.7(c) shows that by reversing the list of  $x$ 's children in line 3, we have a binomial heap  $H'$  that contains every node



**Figure 19.7** The action of BINOMIAL-HEAP-EXTRACT-MIN. (a) A binomial heap  $H$ . (b) The root  $x$  with minimum key is removed from the root list of  $H$ . (c) The linked list of  $x$ 's children is reversed, giving another binomial heap  $H'$ . (d) The result of uniting  $H$  and  $H'$ .

in  $x$ 's tree except for  $x$  itself. Because  $x$ 's tree was removed from  $H$  in line 1, the binomial heap that results from uniting  $H$  and  $H'$  in line 4, shown in Figure 19.7(d), contains all the nodes originally in  $H$  except for  $x$ . Finally, line 5 returns  $x$ .

Since each of lines 1–4 takes  $O(\lg n)$  time if  $H$  has  $n$  nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in  $O(\lg n)$  time.

**Decreasing a key**

The following procedure decreases the key of a node  $x$  in a binomial heap  $H$  to a new value  $k$ . It signals an error if  $k$  is greater than  $x$ 's current key.

BINOMIAL-HEAP-DECREASE-KEY( $H, x, k$ )

```

1  if  $k > \text{key}[x]$ 
2     then error "new key is greater than current key"
3   $\text{key}[x] \leftarrow k$ 
4   $y \leftarrow x$ 
5   $z \leftarrow p[y]$ 
6  while  $z \neq \text{NIL}$  and  $\text{key}[y] < \text{key}[z]$ 
7     do exchange  $\text{key}[y] \leftrightarrow \text{key}[z]$ 
8          $\triangleright$  If  $y$  and  $z$  have satellite fields, exchange them, too.
9          $y \leftarrow z$ 
10         $z \leftarrow p[y]$ 

```

As shown in Figure 19.8, this procedure decreases a key in the same manner as in a binary min-heap: by "bubbling up" the key in the heap. After ensuring that the new key is in fact no greater than the current key and then assigning the new key to  $x$ , the procedure goes up the tree, with  $y$  initially pointing to node  $x$ . In each iteration of the **while** loop of lines 6–10,  $\text{key}[y]$  is checked against the key of  $y$ 's parent  $z$ . If  $y$  is the root or  $\text{key}[y] \geq \text{key}[z]$ , the binomial tree is now min-heap-ordered. Otherwise, node  $y$  violates min-heap ordering, and so its key is exchanged with the key of its parent  $z$ , along with any other satellite information. The procedure then sets  $y$  to  $z$ , going up one level in the tree, and continues with the next iteration.

The BINOMIAL-HEAP-DECREASE-KEY procedure takes  $O(\lg n)$  time. By property 2 of Lemma 19.1, the maximum depth of  $x$  is  $\lfloor \lg n \rfloor$ , so the **while** loop of lines 6–10 iterates at most  $\lfloor \lg n \rfloor$  times.

**Deleting a key**

It is easy to delete a node  $x$ 's key and satellite information from binomial heap  $H$  in  $O(\lg n)$  time. The following implementation assumes that no node currently in the binomial heap has a key of  $-\infty$ .

BINOMIAL-HEAP-DELETE( $H, x$ )

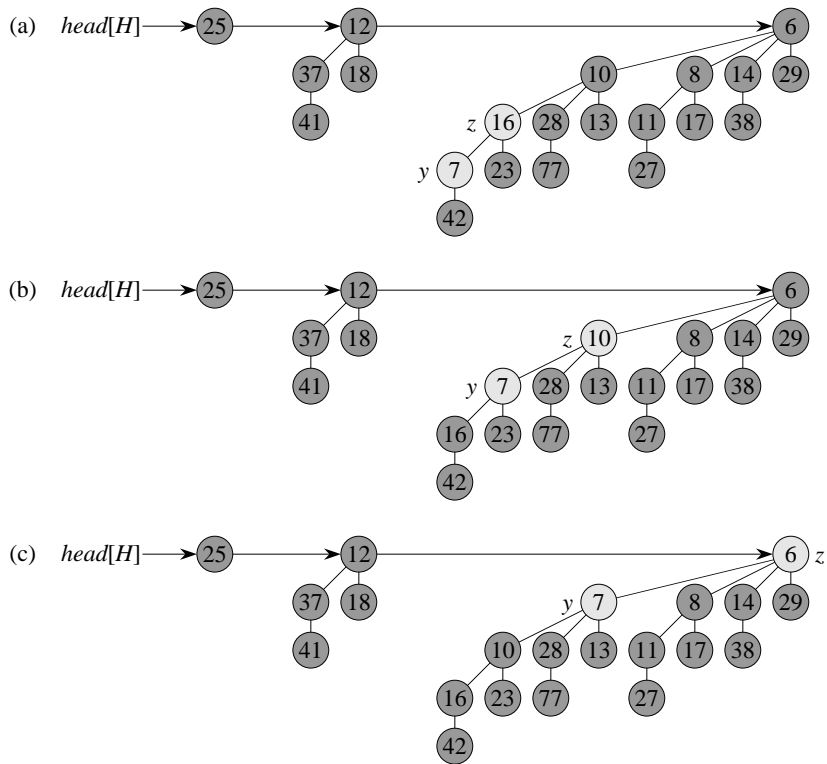
```

1  BINOMIAL-HEAP-DECREASE-KEY( $H, x, -\infty$ )
2  BINOMIAL-HEAP-EXTRACT-MIN( $H$ )

```

The BINOMIAL-HEAP-DELETE procedure makes node  $x$  have the unique minimum key in the entire binomial heap by giving it a key of  $-\infty$ . (Exercise 19.2-6)





**Figure 19.8** The action of BINOMIAL-HEAP-DECREASE-KEY. **(a)** The situation just before line 6 of the first iteration of the **while** loop. Node  $y$  has had its key decreased to 7, which is less than the key of  $y$ 's parent  $z$ . **(b)** The keys of the two nodes are exchanged, and the situation just before line 6 of the second iteration is shown. Pointers  $y$  and  $z$  have moved up one level in the tree, but min-heap order is still violated. **(c)** After another exchange and moving pointers  $y$  and  $z$  up one more level, we find that min-heap order is satisfied, so the **while** loop terminates.

deals with the situation in which  $-\infty$  cannot appear as a key, even temporarily.) It then bubbles this key and the associated satellite information up to a root by calling BINOMIAL-HEAP-DECREASE-KEY. This root is then removed from  $H$  by a call of BINOMIAL-HEAP-EXTRACT-MIN.

The BINOMIAL-HEAP-DELETE procedure takes  $O(\lg n)$  time.

## Exercises

### 19.2-1

Write pseudocode for BINOMIAL-HEAP-MERGE.

**19.2-2**

Show the binomial heap that results when a node with key 24 is inserted into the binomial heap shown in Figure 19.7(d).

**19.2-3**

Show the binomial heap that results when the node with key 28 is deleted from the binomial heap shown in Figure 19.8(c).

**19.2-4**

Argue the correctness of BINOMIAL-HEAP-UNION using the following loop invariant:

At the start of each iteration of the **while** loop of lines 9–21,  $x$  points to a root that is one of the following:

- the only root of its degree,
- the first of the only two roots of its degree, or
- the first or second of the only three roots of its degree.

Moreover, all roots preceding  $x$ 's predecessor on the root list have unique degrees on the root list, and if  $x$ 's predecessor has a degree different from that of  $x$ , its degree on the root list is unique, too. Finally, node degrees monotonically increase as we traverse the root list.

**19.2-5**

Explain why the BINOMIAL-HEAP-MINIMUM procedure might not work correctly if keys can have the value  $\infty$ . Rewrite the pseudocode to make it work correctly in such cases.

**19.2-6**

Suppose there is no way to represent the key  $-\infty$ . Rewrite the BINOMIAL-HEAP-DELETE procedure to work correctly in this situation. It should still take  $O(\lg n)$  time.

**19.2-7**

Discuss the relationship between inserting into a binomial heap and incrementing a binary number and the relationship between uniting two binomial heaps and adding two binary numbers.

**19.2-8**

In light of Exercise 19.2-7, rewrite BINOMIAL-HEAP-INSERT to insert a node directly into a binomial heap without calling BINOMIAL-HEAP-UNION.

**19.2-9**

Show that if root lists are kept in strictly decreasing order by degree (instead of strictly increasing order), each of the binomial heap operations can be implemented without changing its asymptotic running time.

**19.2-10**

Find inputs that cause BINOMIAL-HEAP-EXTRACT-MIN, BINOMIAL-HEAP-DECREASE-KEY, and BINOMIAL-HEAP-DELETE to run in  $\Omega(\lg n)$  time. Explain why the worst-case running times of BINOMIAL-HEAP-INSERT, BINOMIAL-HEAP-MINIMUM, and BINOMIAL-HEAP-UNION are  $\tilde{\Omega}(\lg n)$  but not  $\Omega(\lg n)$ . (See Problem 3-5.)

**Problems****19-1 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 field  $key[x]$ . There is no particular ordering of the keys in the leaves; that is, from left to right, the keys may be in any order. Each internal node  $x$  contains a value  $small[x]$  that is equal to the smallest key stored in any leaf in the subtree rooted at  $x$ . The root  $r$  contains a field  $height[r]$  that is the height of the tree. Finally, 2-3-4 heaps are intended to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. Each of the operations in parts (a)–(e) 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  $k \leq key[x]$ .
- 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.

### 19-2 Minimum-spanning-tree algorithm using binomial heaps

Chapter 23 presents two algorithms to solve the problem of finding a minimum spanning tree of an undirected graph. Here, we shall see how binomial heaps can be used to devise a different minimum-spanning-tree algorithm.

We are given a connected, undirected graph  $G = (V, E)$  with a weight function  $w : E \rightarrow \mathbf{R}$ . We call  $w(u, v)$  the weight of edge  $(u, v)$ . We wish to find a minimum spanning tree for  $G$ : an acyclic subset  $T \subseteq E$  that connects all the vertices in  $V$  and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized.

The following pseudocode, which can be proven correct using techniques from Section 23.1, constructs a minimum spanning tree  $T$ . It maintains a partition  $\{V_i\}$  of the vertices of  $V$  and, with each set  $V_i$ , a set

$$E_i \subseteq \{(u, v) : u \in V_i \text{ or } v \in V_i\}$$

of edges incident on vertices in  $V_i$ .

MST( $G$ )

```

1   $T \leftarrow \emptyset$ 
2  for each vertex  $v_i \in V[G]$ 
3      do  $V_i \leftarrow \{v_i\}$ 
4       $E_i \leftarrow \{(v_i, v) \in E[G]\}$ 
5  while there is more than one set  $V_i$ 
6      do choose any set  $V_i$ 
7          extract the minimum-weight edge  $(u, v)$  from  $E_i$ 
8          assume without loss of generality that  $u \in V_i$  and  $v \in V_j$ 
9          if  $i \neq j$ 
10             then  $T \leftarrow T \cup \{(u, v)\}$ 
11                  $V_i \leftarrow V_i \cup V_j$ , destroying  $V_j$ 
12                  $E_i \leftarrow E_i \cup E_j$ 
```

Describe how to implement this algorithm using binomial heaps to manage the vertex and edge sets. Do you need to change the representation of a binomial heap? Do you need to add operations beyond the mergeable-heap operations given in Figure 19.1? Give the running time of your implementation.

---

**Chapter notes**

Binomial heaps were introduced in 1978 by Vuillemin [307]. Brown [49, 50] studied their properties in detail.

In Chapter 19, we saw how binomial heaps support in  $O(\lg n)$  worst-case time the mergeable-heap operations INSERT, MINIMUM, EXTRACT-MIN, and UNION, plus the operations DECREASE-KEY and DELETE. In this chapter, we shall examine Fibonacci heaps, which support the same operations but have the advantage that operations that do not involve deleting an element run in  $O(1)$  amortized time.

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  $O(1)$  amortized time of each call of DECREASE-KEY adds up to a big improvement over the  $\Theta(\lg n)$  worst-case time of binary or binomial 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. 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.

Like a binomial heap, a Fibonacci heap is a collection of trees. Fibonacci heaps, in fact, are loosely based on binomial heaps. If neither DECREASE-KEY nor DELETE is ever invoked on a Fibonacci heap, each tree in the heap is like a binomial tree. Fibonacci heaps have a more relaxed structure than binomial heaps, however, allowing for improved asymptotic time bounds. Work that maintains the structure can be delayed until it is convenient to perform.

Like the dynamic tables of Section 17.4, Fibonacci heaps offer a good example of a data structure designed with amortized analysis in mind. The intuition and analyses of Fibonacci heap operations in the remainder of this chapter rely heavily on the potential method of Section 17.3.

The exposition in this chapter assumes that you have read Chapter 19 on binomial heaps. The specifications for the operations appear in that chapter, as does the table in Figure 19.1, which summarizes the time bounds for operations on binary heaps, binomial heaps, and Fibonacci heaps. Our presentation of the structure of Fibonacci heaps relies on that of binomial-heap structure, and some of the operations performed on Fibonacci heaps are similar to those performed on binomial heaps.

Like binomial heaps, Fibonacci heaps are not designed to give efficient support to the operation SEARCH; operations that refer to a given node therefore require a pointer to that node as part of their input. When we use a Fibonacci heap in an application, we often store a handle to the corresponding application object in each Fibonacci-heap element, as well as a handle to corresponding Fibonacci-heap element in each application object.

Section 20.1 defines Fibonacci heaps, discusses their representation, and presents the potential function used for their amortized analysis. Section 20.2 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, are presented in Section 20.3. Finally, Section 20.4 finishes off a key part of the analysis and also explains the curious name of the data structure.

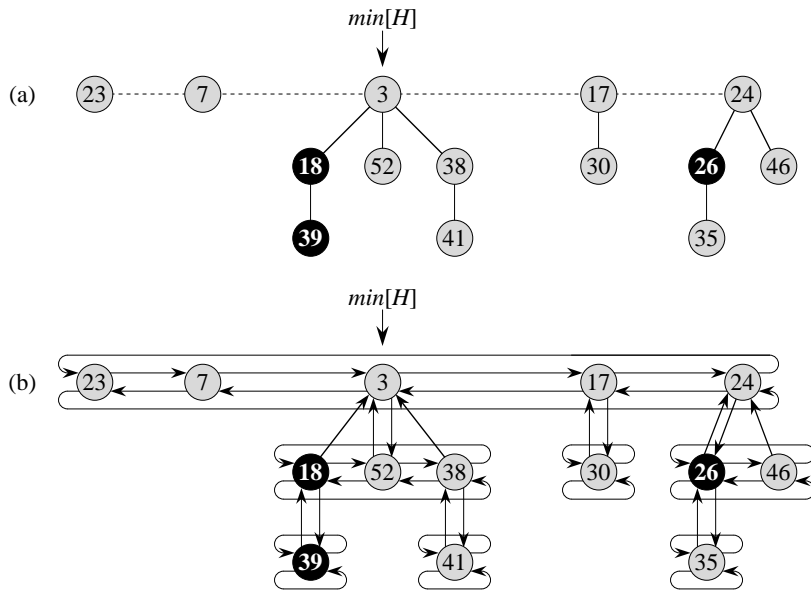
---

## 20.1 Structure of Fibonacci heaps

Like a binomial heap, a *Fibonacci heap* is a collection of min-heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees, however. Figure 20.1(a) shows an example of a Fibonacci heap.

Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. As Figure 20.1(b) shows, each node  $x$  contains a pointer  $p[x]$  to its parent and a pointer  $child[x]$  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  $left[y]$  and  $right[y]$  that point to  $y$ 's left and right siblings, respectively. If node  $y$  is an only child, then  $left[y] = right[y] = y$ . The order in which siblings appear in a child list is arbitrary.

Circular, doubly linked lists (see Section 10.2) have two advantages for use in Fibonacci heaps. First, we can remove a node from 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 the reader fill in the details of their implementations.



**Figure 20.1** (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. The three marked nodes are blackened. 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). These details are omitted in the remaining figures in this chapter, since all the information shown here can be determined from what appears in part (a).

Two other fields in each node will be of use. The number of children in the child list of node  $x$  is stored in  $degree[x]$ . The boolean-valued field  $mark[x]$  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 20.3, we will just set all  $mark$  fields to FALSE.

A given Fibonacci heap  $H$  is accessed by a pointer  $min[H]$  to the root of a tree containing a minimum key; this node is called the **minimum node** of the Fibonacci heap. If a Fibonacci heap  $H$  is empty, then  $min[H] = \text{NIL}$ .

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  $min[H]$  thus points to the node in the root list whose key is minimum. The order of the trees within a root list is arbitrary.

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



### 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  $H$ . The potential of Fibonacci heap  $H$  is then defined by

$$\Phi(H) = t(H) + 2m(H). \quad (20.1)$$

(We will gain some intuition for this potential function in Section 20.3.) For example, the potential of the Fibonacci heap shown in Figure 20.1 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 (20.1), the potential is nonnegative at all subsequent times. From equation (17.3), an upper bound on the total amortized cost is thus 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 there is a known upper bound  $D(n)$  on the maximum degree of any node in an  $n$ -node Fibonacci heap. Exercise 20.2-3 shows that when only the mergeable-heap operations are supported,  $D(n) \leq \lfloor \lg n \rfloor$ . In Section 20.3, we shall show that when we support DECREASE-KEY and DELETE as well,  $D(n) = O(\lg n)$ .

---

## 20.2 Mergeable-heap operations

In this section, we describe and analyze the mergeable-heap operations as implemented for Fibonacci heaps. If only these operations—MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, and UNION—are to be supported, each Fibonacci heap is simply a collection of “unordered” binomial trees. An *unordered binomial tree* is like a binomial tree, and it, too, is defined recursively. The unordered binomial tree  $U_0$  consists of a single node, and an unordered binomial tree  $U_k$  consists of two unordered binomial trees  $U_{k-1}$  for which the root of one is made into *any* child of the root of the other. Lemma 19.1, which gives properties of binomial trees, holds for unordered binomial trees as well, but with the following variation on property 4 (see Exercise 20.2-2):

- 4'. For the unordered binomial tree  $U_k$ , the root has degree  $k$ , which is greater than that of any other node. The children of the root are roots of subtrees  $U_0, U_1, \dots, U_{k-1}$  in some order.

Thus, if an  $n$ -node Fibonacci heap is a collection of unordered binomial trees, then  $D(n) = \lg n$ .

The key idea in the mergeable-heap operations on Fibonacci heaps is to delay work as long as possible. There is a performance trade-off among implementations of the various operations. If the number of trees in a Fibonacci heap is small, then during an EXTRACT-MIN operation we can quickly determine which of the remaining nodes becomes the new minimum node. However, as we saw with binomial heaps in Exercise 19.2-10, we pay a price for ensuring that the number of trees is small: it can take up to  $\Omega(\lg n)$  time to insert a node into a binomial heap or to unite two binomial heaps. As we shall see, we do not attempt to consolidate trees in a Fibonacci heap when we insert a new node or unite two heaps. We save the consolidation for the EXTRACT-MIN operation, which is when we really need to find the new minimum node.

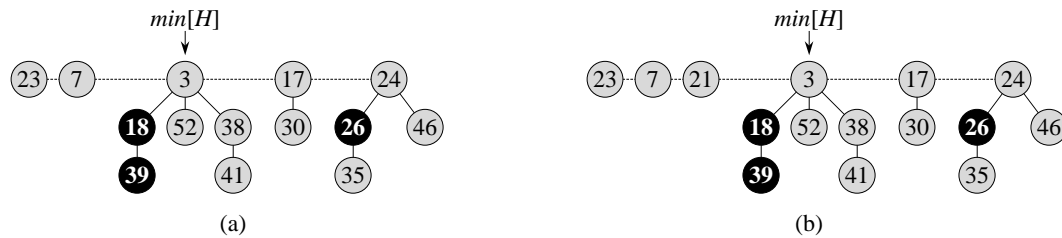
### Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object  $H$ , where  $n[H] = 0$  and  $\text{min}[H] = \text{NIL}$ ; 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  $\text{key}[x]$  has already been filled in.

```
FIB-HEAP-INSERT( $H, x$ )
1   $\text{degree}[x] \leftarrow 0$ 
2   $p[x] \leftarrow \text{NIL}$ 
3   $\text{child}[x] \leftarrow \text{NIL}$ 
4   $\text{left}[x] \leftarrow x$ 
5   $\text{right}[x] \leftarrow x$ 
6   $\text{mark}[x] \leftarrow \text{FALSE}$ 
7  concatenate the root list containing  $x$  with root list  $H$ 
8  if  $\text{min}[H] = \text{NIL}$  or  $\text{key}[x] < \text{key}[\text{min}[H]]$ 
9     then  $\text{min}[H] \leftarrow x$ 
10  $n[H] \leftarrow n[H] + 1$ 
```



**Figure 20.2** Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap  $H$ . **(b)** Fibonacci heap  $H$  after the node with key 21 has been inserted. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

After lines 1–6 initialize the structural fields of node  $x$ , making it its own circular, doubly linked list, line 7 adds  $x$  to the root list of  $H$  in  $O(1)$  actual time. Thus, node  $x$  becomes a single-node min-heap-ordered tree, and thus an unordered binomial tree, in the Fibonacci heap. It has no children and is unmarked. Lines 8–9 then update the pointer to the minimum node of Fibonacci heap  $H$  if necessary. Finally, line 10 increments  $n[H]$  to reflect the addition of the new node. Figure 20.2 shows a node with key 21 inserted into the Fibonacci heap of Figure 20.1.

Unlike the BINOMIAL-HEAP-INSERT procedure, FIB-HEAP-INSERT makes no attempt to consolidate the trees within the Fibonacci heap. If  $k$  consecutive FIB-HEAP-INSERT operations occur, then  $k$  single-node trees are added to the root list.

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

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

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  $\text{min}[H]$ , 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.

```

FIB-HEAP-UNION( $H_1, H_2$ )
1   $H \leftarrow \text{MAKE-FIB-HEAP}()$ 
2   $\text{min}[H] \leftarrow \text{min}[H_1]$ 
3  concatenate the root list of  $H_2$  with the root list of  $H$ 
4  if ( $\text{min}[H_1] = \text{NIL}$ ) or ( $\text{min}[H_2] \neq \text{NIL}$  and  $\text{min}[H_2] < \text{min}[H_1]$ )
5     then  $\text{min}[H] \leftarrow \text{min}[H_2]$ 
6   $n[H] \leftarrow n[H_1] + n[H_2]$ 
7  free the objects  $H_1$  and  $H_2$ 
8  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  $n[H]$  to the total number of nodes. The Fibonacci heap objects  $H_1$  and  $H_2$  are freed in line 7, and line 8 returns the resulting Fibonacci heap  $H$ . As in the FIB-HEAP-INSERT procedure, no consolidation of trees occurs.

The change in potential is

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

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 uses the auxiliary procedure CONSOLIDATE, which will be presented shortly.

```

FIB-HEAP-EXTRACT-MIN( $H$ )
1   $z \leftarrow \text{min}[H]$ 
2  if  $z \neq \text{NIL}$ 
3      then for each child  $x$  of  $z$ 
4          do add  $x$  to the root list of  $H$ 
5           $p[x] \leftarrow \text{NIL}$ 
6      remove  $z$  from the root list of  $H$ 
7      if  $z = \text{right}[z]$ 
8          then  $\text{min}[H] \leftarrow \text{NIL}$ 
9          else  $\text{min}[H] \leftarrow \text{right}[z]$ 
10         CONSOLIDATE( $H$ )
11      $n[H] \leftarrow n[H] - 1$ 
12 return  $z$ 

```

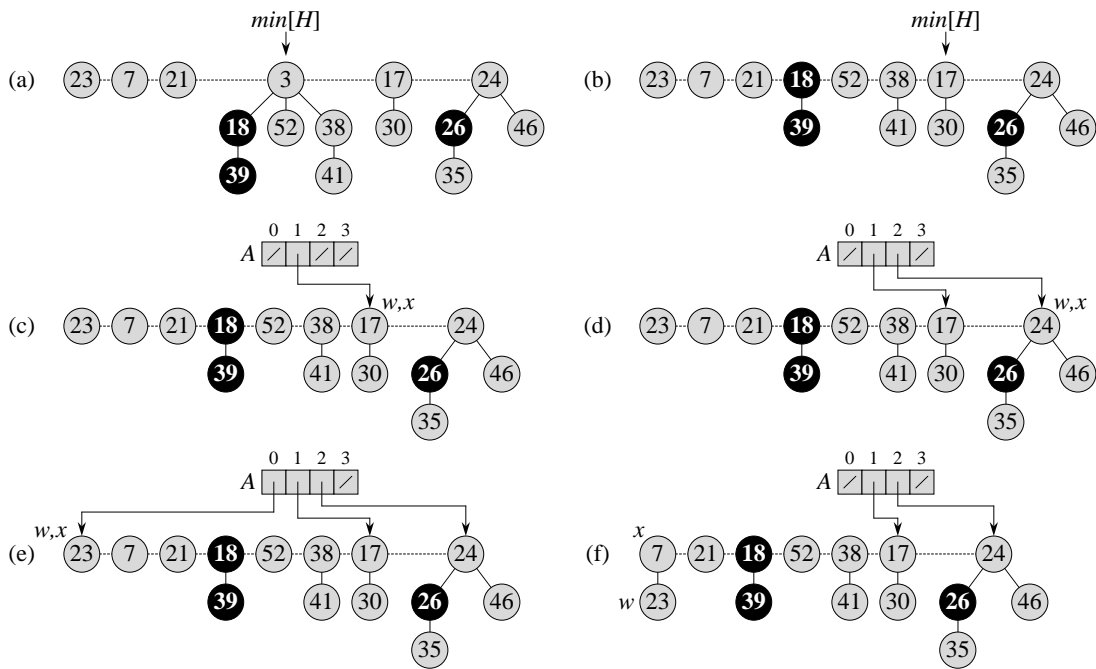
As shown in Figure 20.3, FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

We start in line 1 by saving a pointer  $z$  to the minimum node; this pointer is returned at the end. If  $z = \text{NIL}$ , then Fibonacci heap  $H$  is already empty and we are done. Otherwise, as in the BINOMIAL-HEAP-EXTRACT-MIN procedure, 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  $z = \text{right}[z]$  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  $\text{min}[H]$  into the root list to point to a node other than  $z$  (in this case,  $\text{right}[z]$ ), which is not necessarily going to be the new minimum node when FIB-HEAP-EXTRACT-MIN is done. Figure 20.3(b) shows the Fibonacci heap of Figure 20.3(a) after line 9 has been performed.

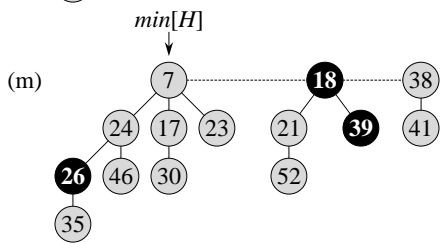
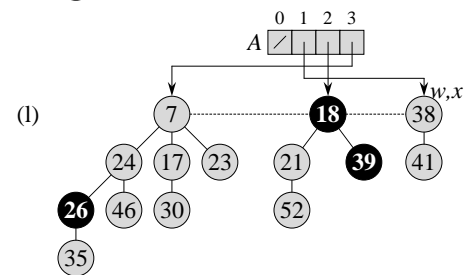
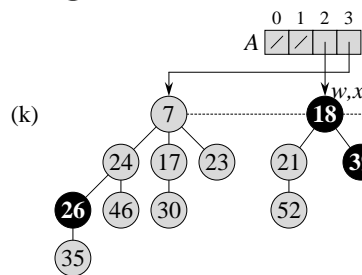
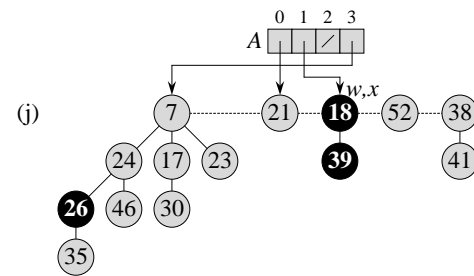
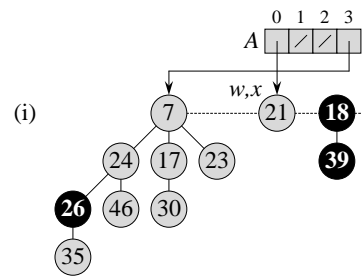
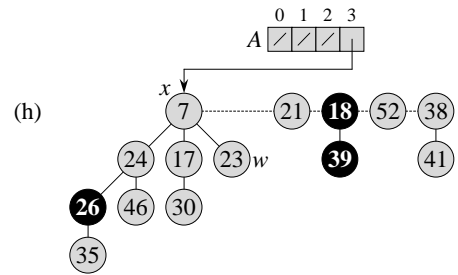
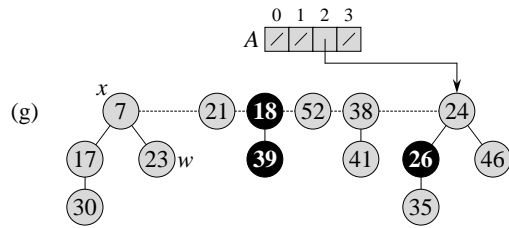
The next step, in which we reduce the number of trees in the Fibonacci heap, is **consolidating** the root list of  $H$ ; this is performed by the call CONSOLIDATE( $H$ ). Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct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

1. Find two roots  $x$  and  $y$  in the root list with the same degree, where  $\text{key}[x] \leq \text{key}[y]$ .
2. **Link**  $y$  to  $x$ : remove  $y$  from the root list, and make  $y$  a child of  $x$ . This operation is performed by the FIB-HEAP-LINK procedure. The field  $\text{degree}[x]$  is incremented, and the mark on  $y$ , if any, is cleared.

The procedure CONSOLIDATE uses an auxiliary array  $A[0..D(n[H])]$ ; if  $A[i] = y$ , then  $y$  is currently a root with  $\text{degree}[y] = i$ .



**Figure 20.3** The action of FIB-HEAP-EXTRACT-MIN. (a) A Fibonacci heap  $H$ . (b) The situation after the minimum node  $z$  is removed from the root list and its children are added 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 3–13 of the procedure CONSOLIDATE. The root list is processed by starting at the node pointed to by  $\text{min}[H]$  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 6–12. 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 is now pointed to by  $x$ . In part (g), the node with key 17 has been linked to the node with key 7, which is still pointed to by  $x$ . 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. (i)–(l) The situation after each of the next four iterations of the **for** loop. (m) Fibonacci heap  $H$  after reconstruction of the root list from the array  $A$  and determination of the new  $\text{min}[H]$  pointer.



```

CONSOLIDATE( $H$ )
1  for  $i \leftarrow 0$  to  $D(n[H])$ 
2      do  $A[i] \leftarrow \text{NIL}$ 
3  for each node  $w$  in the root list of  $H$ 
4      do  $x \leftarrow w$ 
5           $d \leftarrow \text{degree}[x]$ 
6          while  $A[d] \neq \text{NIL}$ 
7              do  $y \leftarrow A[d]$   $\triangleright$  Another node with the same degree as  $x$ .
8                  if  $\text{key}[x] > \text{key}[y]$ 
9                      then exchange  $x \leftrightarrow y$ 
10                     FIB-HEAP-LINK( $H, y, x$ )
11                      $A[d] \leftarrow \text{NIL}$ 
12                      $d \leftarrow d + 1$ 
13              $A[d] \leftarrow x$ 
14   $\text{min}[H] \leftarrow \text{NIL}$ 
15  for  $i \leftarrow 0$  to  $D(n[H])$ 
16      do if  $A[i] \neq \text{NIL}$ 
17          then add  $A[i]$  to the root list of  $H$ 
18              if  $\text{min}[H] = \text{NIL}$  or  $\text{key}[A[i]] < \text{key}[\text{min}[H]]$ 
19                  then  $\text{min}[H] \leftarrow A[i]$ 

```

FIB-HEAP-LINK( $H, y, x$ )

```

1  remove  $y$  from the root list of  $H$ 
2  make  $y$  a child of  $x$ , incrementing  $\text{degree}[x]$ 
3   $\text{mark}[y] \leftarrow \text{FALSE}$ 

```

In detail, the CONSOLIDATE procedure works as follows. Lines 1–2 initialize  $A$  by making each entry NIL. The **for** loop of lines 3–13 processes each root  $w$  in the root list. After processing each root  $w$ , it ends up in a tree rooted at some node  $x$ , which may or may not be identical to  $w$ . Of the processed roots, no others will have the same degree as  $x$ , and so we will set array entry  $A[\text{degree}[x]]$  to point to  $x$ . 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 6–12 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,  $d = \text{degree}[x]$ .

We use this loop invariant as follows:

**Initialization:** Line 5 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  $d = \text{degree}[x] = \text{degree}[y]$ , 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 8–9 exchange the pointers to  $x$  and  $y$  if necessary. Next, we link  $y$  to  $x$  by the call  $\text{FIB-HEAP-LINK}(H, y, x)$  in line 10. This call increments  $\text{degree}[x]$  but leaves  $\text{degree}[y]$  as  $d$ . Because node  $y$  is no longer a root, the pointer to it in array  $A$  is removed in line 11. Because the call of  $\text{FIB-HEAP-LINK}$  increments the value of  $\text{degree}[x]$ , line 12 restores the invariant that  $d = \text{degree}[x]$ .

**Termination:** We repeat the **while** loop until  $A[d] = \text{NIL}$ , 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 13 and perform the next iteration of the **for** loop.

Figures 20.3(c)–(e) show the array  $A$  and the resulting trees after the first three iterations of the **for** loop of lines 3–13. In the next iteration of the **for** loop, three links occur; their results are shown in Figures 20.3(f)–(h). Figures 20.3(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 3–13 completes, line 14 empties the root list, and lines 15–19 reconstruct it from the array  $A$ . The resulting Fibonacci heap is shown in Figure 20.3(m). After consolidating the root list,  $\text{FIB-HEAP-EXTRACT-MIN}$  finishes up by decrementing  $n[H]$  in line 11 and returning a pointer to the deleted node  $z$  in line 12.

Observe that if all trees in the Fibonacci heap are unordered binomial trees before  $\text{FIB-HEAP-EXTRACT-MIN}$  is executed, then they are all unordered binomial trees afterward. There are two ways in which trees are changed. First, in lines 3–5 of  $\text{FIB-HEAP-EXTRACT-MIN}$ , each child  $x$  of root  $z$  becomes a root. By Exercise 20.2-2, each new tree is itself an unordered binomial tree. Second, trees are linked by  $\text{FIB-HEAP-LINK}$  only if they have the same degree. Since all trees are unordered binomial trees before the link occurs, two trees whose roots each have  $k$  children must have the structure of  $U_k$ . The resulting tree therefore has the structure of  $U_{k+1}$ .

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  $\text{FIB-HEAP-EXTRACT-MIN}$  operation.

The actual cost of extracting the minimum node can be accounted for as follows. An  $O(D(n))$  contribution comes from there being at most  $D(n)$  children of the minimum node that are processed in  $\text{FIB-HEAP-EXTRACT-MIN}$  and from the work in lines 1–2 and 14–19 of  $\text{CONSOLIDATE}$ . It remains to analyze the contribution from the **for** loop of lines 3–13. The size of the root list upon calling  $\text{CONSOLIDATE}$  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  $D(n)$ . Every time through the **while** loop of lines 6–12, one of the roots is linked to another, and thus 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  $t(H) + 2m(H)$ , and the potential afterward is at most  $(D(n) + 1) + 2m(H)$ , since at most  $D(n) + 1$  roots remain and no nodes become marked during the operation. The amortized cost is thus at most

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

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 20.4 that  $D(n) = O(\lg n)$ , so that the amortized cost of extracting the minimum node is  $O(\lg n)$ .

## Exercises

### 20.2-1

Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in Figure 20.3(m).

### 20.2-2

Prove that Lemma 19.1 holds for unordered binomial trees, but with property 4' in place of property 4.

### 20.2-3

Show that if only the mergeable-heap operations are supported, the maximum degree  $D(n)$  in an  $n$ -node Fibonacci heap is at most  $\lfloor \lg n \rfloor$ .

### 20.2-4

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 the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union perform consolidation as their last step. What are the worst-case running times of operations on McGee heaps? How novel is the professor's data structure?

**20.2-5**

Argue that when the only operations on keys are comparing two keys (as is the case for all the implementations in this chapter), not all of the mergeable-heap operations can run in  $O(1)$  amortized time.

---

**20.3 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. These operations do not preserve the property that all trees in the Fibonacci heap are unordered binomial trees. They are close enough, however, that we can bound the maximum degree  $D(n)$  by  $O(\lg n)$ . Proving this bound, which we shall do in Section 20.4, 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 fields in the removed node.

FIB-HEAP-DECREASE-KEY( $H, x, k$ )

```

1  if  $k > \text{key}[x]$ 
2    then error "new key is greater than current key"
3   $\text{key}[x] \leftarrow k$ 
4   $y \leftarrow p[x]$ 
5  if  $y \neq \text{NIL}$  and  $\text{key}[x] < \text{key}[y]$ 
6    then CUT( $H, x, y$ )
7      CASCADING-CUT( $H, y$ )
8  if  $\text{key}[x] < \text{key}[\text{min}[H]]$ 
9    then  $\text{min}[H] \leftarrow x$ 

```

CUT( $H, x, y$ )

```

1  remove  $x$  from the child list of  $y$ , decrementing  $\text{degree}[y]$ 
2  add  $x$  to the root list of  $H$ 
3   $p[x] \leftarrow \text{NIL}$ 
4   $\text{mark}[x] \leftarrow \text{FALSE}$ 

```

```

CASCADING-CUT( $H, y$ )
1   $z \leftarrow p[y]$ 
2  if  $z \neq \text{NIL}$ 
3      then if  $\text{mark}[y] = \text{FALSE}$ 
4          then  $\text{mark}[y] \leftarrow \text{TRUE}$ 
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  $\text{key}[x] \geq \text{key}[y]$ , 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.

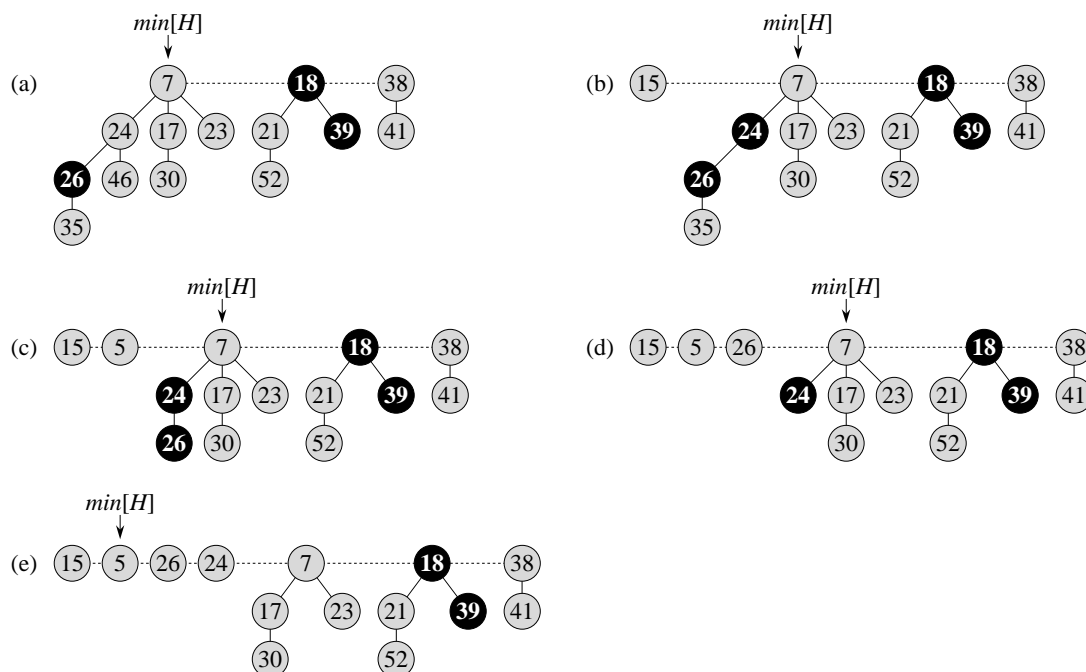
We use the *mark* fields to obtain the desired time bounds. They record a little piece of the history of each node. Suppose that the following events have happened to node  $x$ :

1. at some time,  $x$  was a root,
2. then  $x$  was linked to 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 field  $\text{mark}[x]$  is TRUE if steps 1 and 2 have occurred and one child of  $x$  has been cut. The CUT procedure, therefore, clears  $\text{mark}[x]$  in line 4, since it performs step 1. (We can now see why line 3 of FIB-HEAP-LINK clears  $\text{mark}[y]$ : node  $y$  is being linked to another node, and so step 2 is being performed. The next time a child of  $y$  is cut,  $\text{mark}[y]$  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 performs 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 either a root or an unmarked node is found.

Once all the cascading cuts have occurred, lines 8–9 of FIB-HEAP-DECREASE-KEY finish up by updating  $\text{min}[H]$  if necessary. The only node whose key changed



**Figure 20.4** 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.) The result of the FIB-HEAP-DECREASE-KEY operation is shown in part (e), with  $\text{min}[H]$  pointing to the new minimum node.

was the node  $x$  whose key decreased. Thus, the new minimum node is either the original minimum node or node  $x$ .

Figure 20.4 shows the execution of two calls of FIB-HEAP-DECREASE-KEY, starting with the Fibonacci heap shown in Figure 20.4(a). The first call, shown in Figure 20.4(b), involves no cascading cuts. The second call, shown in Figures 20.4(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 CASCADING-CUT is recursively called  $c$  times from a given invocation

of FIB-HEAP-DECREASE-KEY. 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. Each recursive call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. Afterward, there are  $t(H) + c$  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

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

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 the potential function was defined 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, so the potential is reduced 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

It is easy to delete a node from an  $n$ -node Fibonacci heap in  $O(D(n))$  amortized time, as is done by the following pseudocode. 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( $H, x, -\infty$ )
2  FIB-HEAP-EXTRACT-MIN( $H$ )
```

FIB-HEAP-DELETE is analogous to BINOMIAL-HEAP-DELETE. It makes  $x$  become the minimum node in the Fibonacci heap by giving it a uniquely small key of  $-\infty$ . Node  $x$  is then removed from the Fibonacci heap by the FIB-HEAP-EXTRACT-MIN procedure. 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 20.4 that  $D(n) = O(\lg n)$ , the amortized time of FIB-HEAP-DELETE is  $O(\lg n)$ .

**Exercises****20.3-1**

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.

**20.3-2**

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

**20.4 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  $O(\lg n)$ . By Exercise 20.2-3, when all trees in the Fibonacci heap are unordered binomial trees,  $D(n) = \lceil \lg n \rceil$ . The cuts that occur in FIB-HEAP-DECREASE-KEY, however, may cause trees within the Fibonacci heap to violate the unordered binomial tree properties. In this section, we shall show that because we cut a node from its parent as soon as it loses two children,  $D(n)$  is  $O(\lg n)$ . In particular, we shall show that  $D(n) \leq \lfloor \log_\phi n \rfloor$ , where  $\phi = (1 + \sqrt{5})/2$ .

The key to the analysis is as follows. For each node  $x$  within a Fibonacci heap, define  $\text{size}(x)$  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  $\text{size}(x)$  is exponential in  $\text{degree}[x]$ . Bear in mind that  $\text{degree}[x]$  is always maintained as an accurate count of the degree of  $x$ .

**Lemma 20.1**

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

**Proof** Obviously,  $\text{degree}[y_1] \geq 0$ .

For  $i \geq 2$ , we note that when  $y_i$  was linked to  $x$ , all of  $y_1, y_2, \dots, y_{i-1}$  were children of  $x$ , so we must have had  $\text{degree}[x] = i - 1$ . Node  $y_i$  is linked to  $x$  only if  $\text{degree}[x] = \text{degree}[y_i]$ , so we must have also had  $\text{degree}[y_i] = i - 1$  at that time. Since then, node  $y_i$  has lost at most one child, since it would have been cut from  $x$  if it had lost two children. We conclude that  $\text{degree}[y_i] \geq i - 2$ . ■

We finally come to the part of the analysis that explains the name “Fibonacci heaps.” Recall from Section 3.2 that for  $k = 0, 1, 2, \dots$ , the  $k$ th 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 20.2**

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  $k = 0$ ,

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

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

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

The following lemma and its corollary complete the analysis. They use the inequality (proved in Exercise 3.2-7)

$$F_{k+2} \geq \phi^k,$$

where  $\phi$  is the golden ratio, defined in equation (3.22) as  $\phi = (1 + \sqrt{5})/2 = 1.61803\dots$



**Lemma 20.3**

Let  $x$  be any node in a Fibonacci heap, and let  $k = \text{degree}[x]$ . 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 value of  $\text{size}(z)$  over all nodes  $z$  such that  $\text{degree}[z] = k$ . Trivially,  $s_0 = 1$ ,  $s_1 = 2$ , and  $s_2 = 3$ . The number  $s_k$  is at most  $\text{size}(x)$ , and clearly, the value of  $s_k$  increases monotonically with  $k$ . As in Lemma 20.1, let  $y_1, y_2, \dots, y_k$  denote the children of  $x$  in the order in which they were linked to  $x$ . To compute a lower bound on  $\text{size}(x)$ , we count one for  $x$  itself and one for the first child  $y_1$  (for which  $\text{size}(y_1) \geq 1$ ), giving

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

where the last line follows from Lemma 20.1 (so that  $\text{degree}[y_i] \geq i - 2$ ) and the monotonicity of  $s_k$  (so that  $s_{\text{degree}[y_i]} \geq s_{i-2}$ ).

We now show by induction on  $k$  that  $s_k \geq F_{k+2}$  for all nonnegative integer  $k$ . 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  $i = 0, 1, \dots, k - 1$ . We have

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^k s_{i-2} \\ &\geq 2 + \sum_{i=2}^k F_i \\ &= 1 + \sum_{i=0}^k F_i \\ &= F_{k+2} \quad (\text{by Lemma 20.2}) . \end{aligned}$$

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

**Corollary 20.4**

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  $k = \text{degree}[x]$ . By Lemma 20.3, 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****20.4-1**

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.

**20.4-2**

Suppose we generalize the cascading-cut rule to cut a node  $x$  from its parent as soon as it loses its  $k$ th child, for some integer constant  $k$ . (The rule in Section 20.3 uses  $k = 2$ .) For what values of  $k$  is  $D(n) = O(\lg n)$ ?

**Problems****20-1 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  $\text{min}[H]$ .

```

PISANO-DELETE( $H, x$ )
1  if  $x = \text{min}[H]$ 
2    then FIB-HEAP-EXTRACT-MIN( $H$ )
3    else  $y \leftarrow p[x]$ 
4          if  $y \neq \text{NIL}$ 
5            then 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  $\text{min}[H]$ . Your bound should be in terms of  $\text{degree}[x]$  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  $\text{degree}[x]$ ,  $c$ ,  $t(H)$ , and  $m(H)$ .

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

### 20-2 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  $key[x]$ .
- b.* Give an efficient implementation of FIB-HEAP-PRUNE( $H, r$ ), which deletes  $\min(r, n[H])$  nodes from  $H$ . Which nodes are deleted should be arbitrary. Analyze the amortized running time of your implementation. (*Hint:* You may need to modify the data structure and potential function.)

---

## Chapter notes

Fibonacci heaps were introduced by Fredman and Tarjan [98]. 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 [81] developed “relaxed heaps” as an alternative to Fibonacci heaps. There are 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.