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.

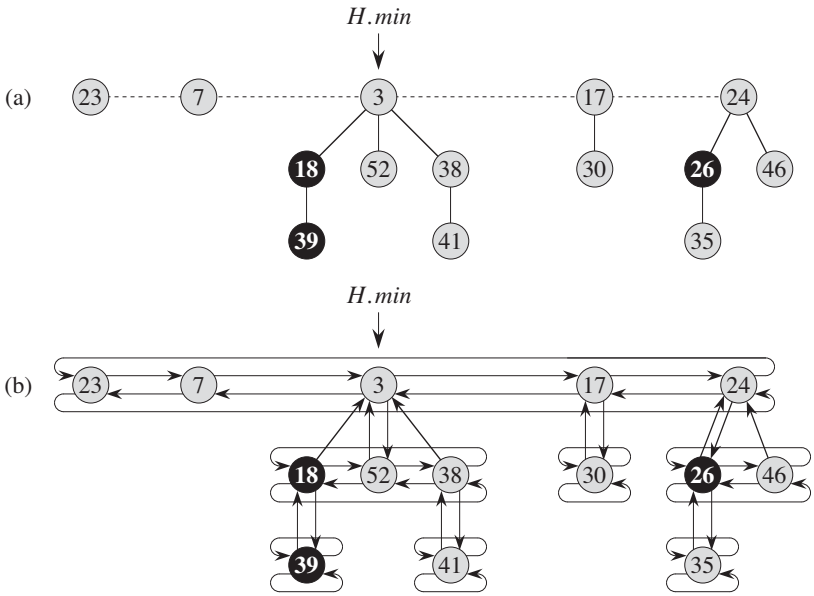

Figure 19.2 **(a)** A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. Black nodes are marked. The potential of this particular Fibonacci heap is $5 + 2 \cdot 3 = 11$. **(b)** 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

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

(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
```

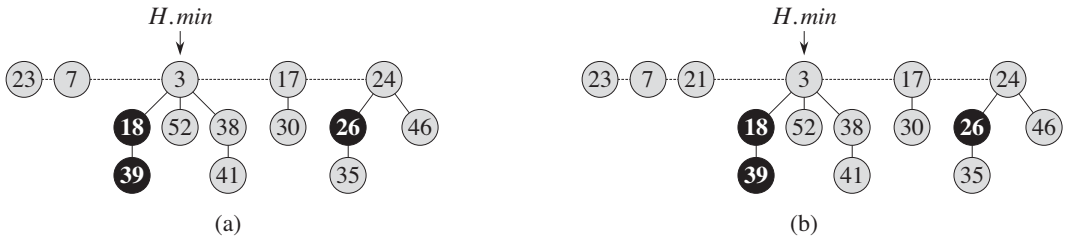

Figure 19.3 Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap H. **(b)** Fibonacci heap H after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

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

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

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 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 $x.key \le y.key$.
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.

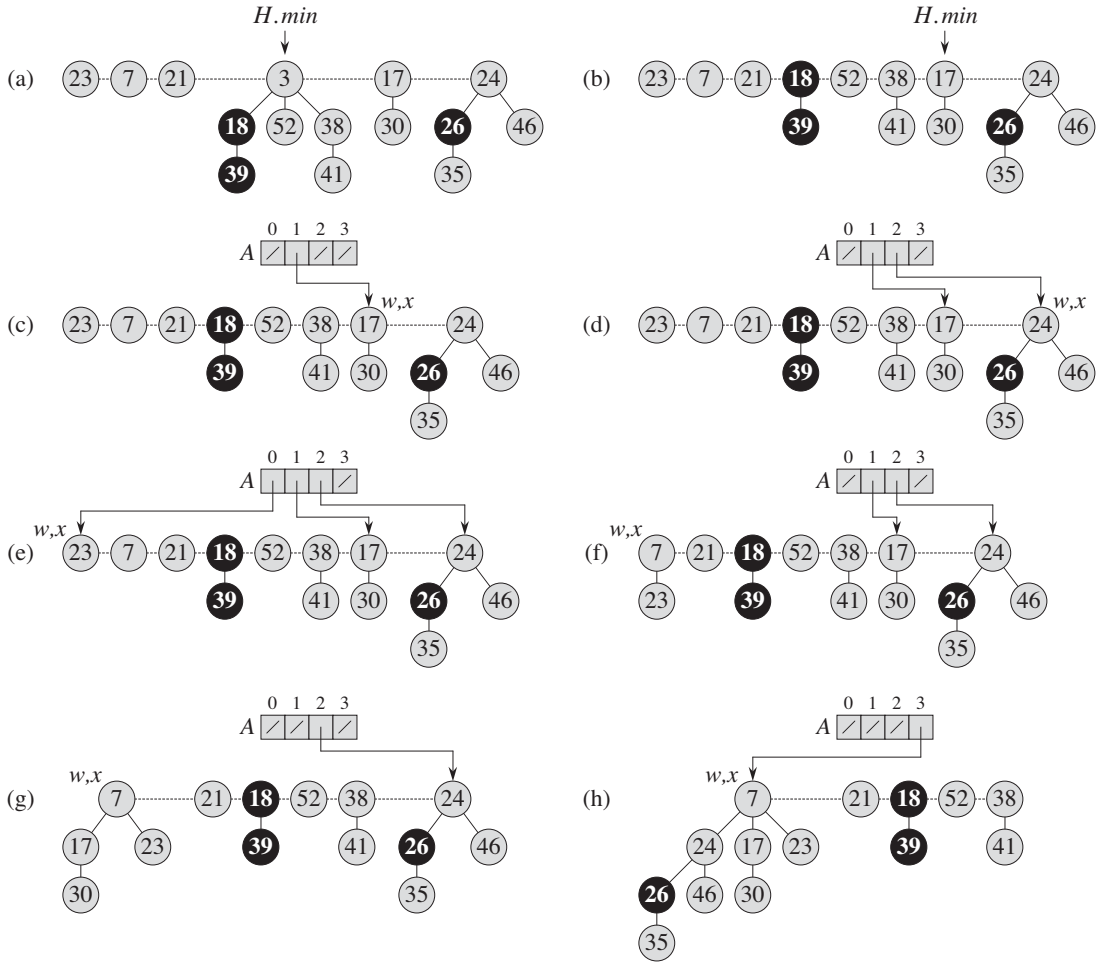

Figure 19.4 The action of FIB-HEAP-EXTRACT-MIN. **(a)** A Fibonacci heap H. **(b)** The situation after removing the minimum node z from the root list and adding its children to the root list. **(c)–(e)** The array A and the trees after each of the first three iterations of the **for** loop of lines 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.

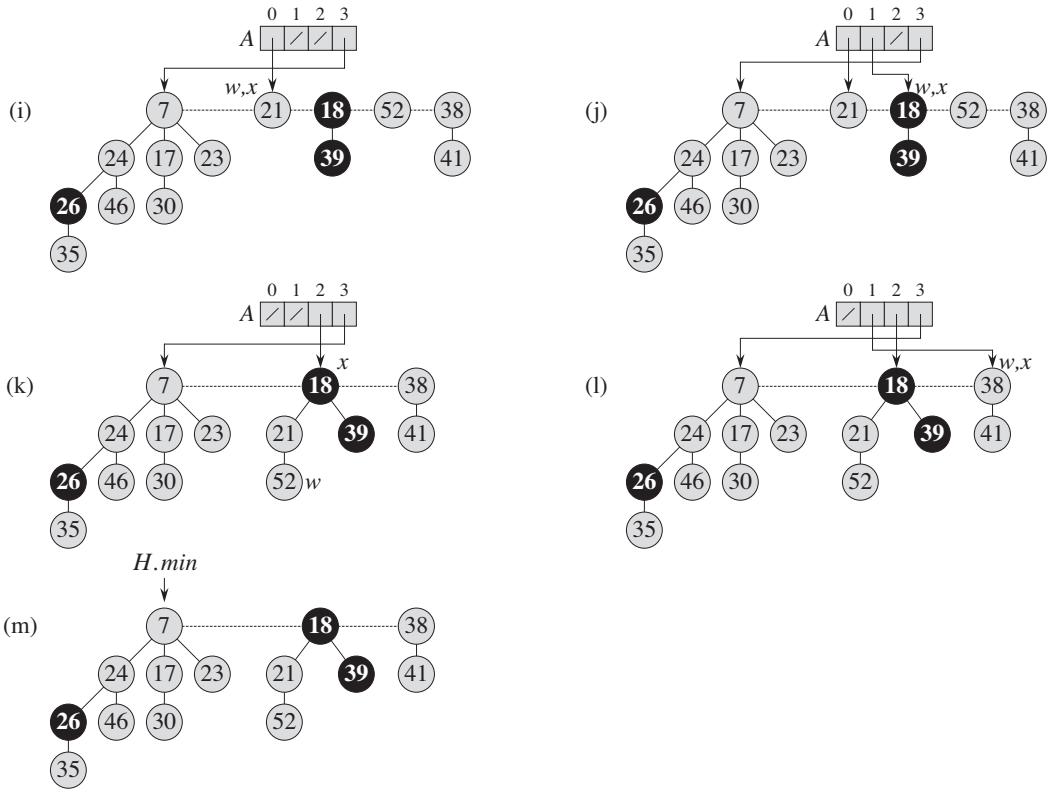

Figure 19.4, continued **(i)–(l)** The situation after each of the next four iterations of the **for** loop. **(m)** Fibonacci heap H after reconstructing the root list from the array A and determining the new $H.min$ pointer.

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

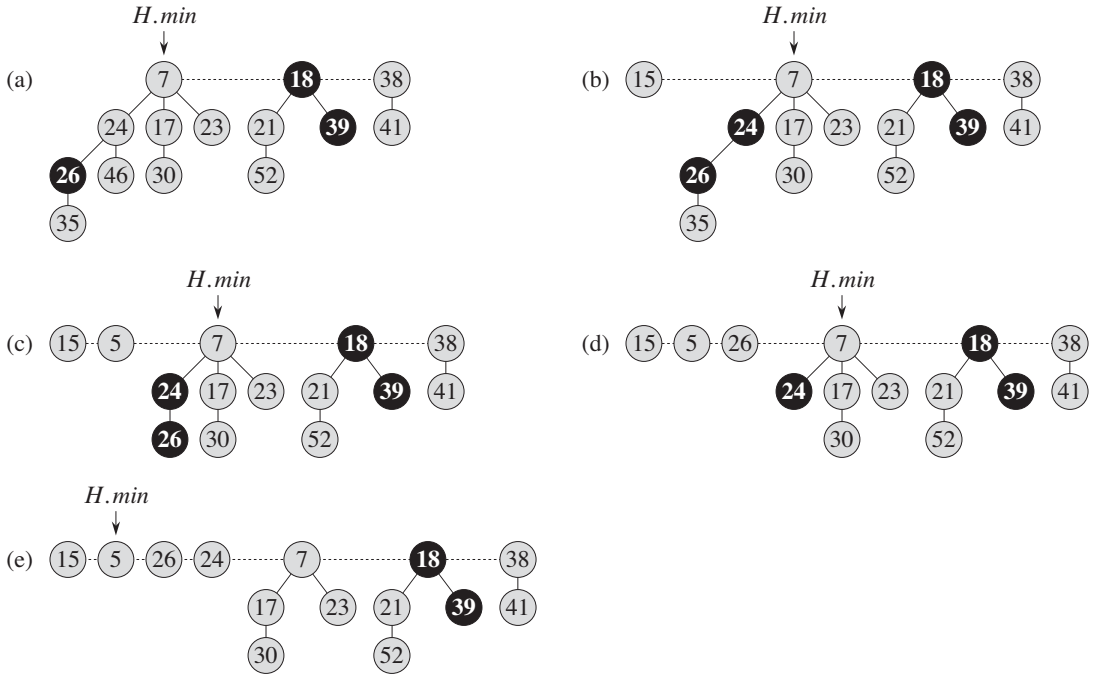

Figure 19.5 Two calls of FIB-HEAP-DECREASE-KEY. **(a)** The initial Fibonacci heap. **(b)** The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. **(c)–(e)** The node with key 35 has its key decreased 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.