CHAPTER 21: FIBONACCI HEAPS

Previous Chapter Return to Table of Contents Next Chapter

In Chapter 20, 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. The asymptotically fastest algorithms to date for problems such as computing minimum spanning trees (Chapter 24) and finding single-source shortest paths (Chapter 25) 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 great 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 differ from binomial heaps, however, in that they have a more relaxed structure, 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 18.4, 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 18.3.

The exposition in this chapter assumes that you have read Chapter 20 on binomial heaps. The specifications for the operations appear in that chapter, as does the table in Figure 20.1, 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. You will also find that 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.

Section 21.1 defines Fibonacci heaps, discusses their representation, and presents the potential function used for their amortized analysis. Section 21.2 shows how to implement the mergeable-heap operations and achieve the amortized time bounds shown in Figure 20.1. The remaining two operations, DECREASE-KEY and DELETE, are presented in Section 21.3. Finally, Section 21.4 finishes off a key part of the analysis.

21.1 Structure of Fibonacci heaps

Like a binomial heap, a ***Fibonacci heap*** is a collection of heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees, however. Figure 21.1 (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 21.1(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 11.2) 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.

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 *h*as lost a child since the last time x was made the child of another node. We won't worry about the details of marking nodes until Section 21.3. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node.

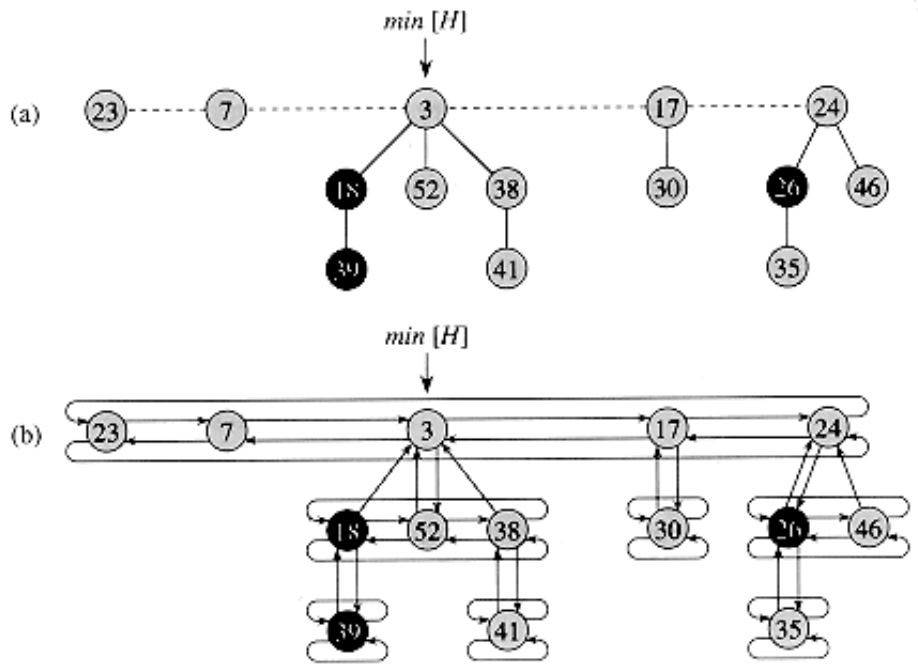

Figure 21.1 (a) A Fibonacci heap consisting of five heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened. The potential of this particular Fibonacci heap is $5 + 2 \cdot 3 = 11$. (b) 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).

A given Fibonacci heap H is accessed by a pointer $min[H]$ to the root of the tree containing a minimum key; this node is called the ***minimum node*** of the Fibonacci heap. If a Fibonacci heap H his empty, then $min[H]$ = `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 18.3 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

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

(21.1)

For example, the potential of the Fibonacci heap shown in Figure 21.1 is 5 + 2 • 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 (21.1), the potential is nonnegative at all subsequent times. From equation (18.2), 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 code in an *n*-node Fibonacci heap. Exercise 21.2-3 shows that when only the mergeable-heap operations are supported, $D(n) = \lfloor \lg n \rfloor$. In Section 21.3, we shall show that when we support `DECREASE-KEY` and `DELETE` as well, *D*(*n*) = *O*(lg *n*).

21.2 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 20.1, which gives properties of binomial trees, holds for unordered binomial trees as well, but with the following variation on property 4 (see Exercise 21.2-2):

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, . . . , U_k-1 in some order.

Thus, if an *n*-node Fibonacci in 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 we can quickly determine the new minimum node during an EXTRACT-MIN operation. However, as we saw with binomial heaps in Exercise 20.2-10, 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 $min[H] = \text{NIL}$; 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 of course that the node has already been allocated and that $key[x]$ has already been filled in.

```
FIB-HEAP-INSERT(H, x)

1  degree[x] ← 0

2  p[x] ← NIL

3  child[x] ← NIL

4  left[x] ← x

5  right[x] ← x

6  mark[x] ← FALSE

7  concatenate the root list containing x with root list H

8  if min[H] = NIL or key[x] < key[min[H]]

9     then min[H] ← x

10  n[H] ← n[H] + 1
```

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 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 21.2 shows a node with key 21 inserted into the Fibonacci heap of Figure 21.1.

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.

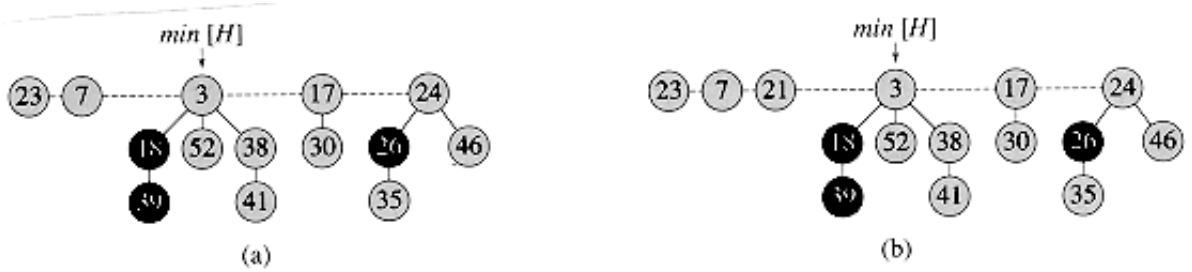

Figure 21.2 Inserting a node into a Fibonacci heap. (a) A Fibonacci heap H. (b) Fibonacci heap H after the node with key 21 has been inserted. The node becomes its own heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

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 $min[H]$, 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.

```
FIB-HEAP-UNION(H1, H2)

1   H ← MAKE-FIB-HEAP()
2   min[H] ← min[H1]
3   concatenate the root list of H2 with the root list of H
4   if (min[H1] = NIL) or (min[H2]  NIL and min[H2] < min[H1])
5   then min[H] ← min[H2]
6   n[H] ← n[H1] + n[H2]
7   free the objects H1 and H2
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

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2))$$

$$= (t(H) + 2m(H)) - ((t(H_1) + 2\ m(H_1)) + (t(H_2) + 2m(H_2)))$$

$$= 0,$$

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 ← min[H]
2  if z  NIL
3      then for each child x of z
4               do add x to the root list of H
5                  p[x] ← NIL
6           remove z from the root list of H
7           if z = right[z]
8              then min[H] ← NIL
9              else min[H] ← right[z]
10                   CONSOLIDATE(H)
11           n[H] ← n[H] - 1
12 return z
```

As shown in Figure 21.3, 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 = 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 $z = right[z]$ 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 $min[H]$ into the root list to point to a node other than z (in this case, $right[z]$). Figure 21.3(b) shows the Fibonacci heap of Figure 21.3(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 $key[x] \le key[y]$.

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 $degree[x]$ 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 $degree[y] = i$.

```
CONSOLIDATE(H)

1 for i ← 0 to D(n[H])
2      do A[i] ← NIL
3 for each node w in the root list of H
4      do x ← w
5         d ← degree[x]
6         while A[d]  NIL
7            do y ← A[d]
8               if key[x] > key[y]
9                  then exchange x ↔ y
10                FIB-HEAP-LINK(H,y,x)
11                A[d] ← NIL
12                d ← d + 1
13         A[d] ← x
14 min[H] ← NIL
15 for i ← 0 to D(n[H])
16      do if A[i]  NIL
17            then add A[i] to the root list of H
18                 if min[H] = NIL or key[A[i]] < key[min[H]]
19                    then min[H] ← 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 degree[x]
3  mark[y] ← FALSE
```

In detail, the CONSOLIDATE procedure works as follows. In lines 1-2, we initialize A by making each

entry NIL. When we are done processing each root w, it ends up in a tree rooted at some node x, which may or may not be identical to w. Array entry $A[degree[x]]$ will then be set to point to x. In the **for** loop of lines 3-13, we examine each root w in the root list. The invariant maintained during each iteration of the **for** loop is that node x is the root of the tree containing node w. The **while** loop of lines 6-12 maintains the invariant that $d = degree[x]$ (except in line 11, as we shall see in a moment). In each iteration of the **while** loop, $A[d]$ points to some root y. Because $d = degree[x] = degree[y]$, 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 FIB-HEAP-LINK(H,y,x) in line 10. This call increments $degree[x]$ but leaves $degree[y]$ as d. Because node y is no longer a root, the pointer to it in array A is removed in line 11. Because the value of $degree[x]$ is incremented by the call of FIB-HEAP-LINK, line 12 restores the invariant that $d = degree[x]$. We repeat the **while** loop until $A[d]$ = NIL, in which case there is no other root with the same degree as x. We set $A[d]$ to x in line 13 and perform the next iteration of the **for** loop. Figures 21.3(c)-(e) show the array A and the resul