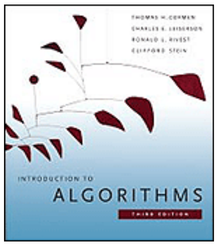

Chapter 19 - Fibonacci Heaps

Introduction to Algorithms, Third Edition
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
The MIT Press © 2009 *Citation*

Previous Next

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 an 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 $H.min$ = 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
```

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.

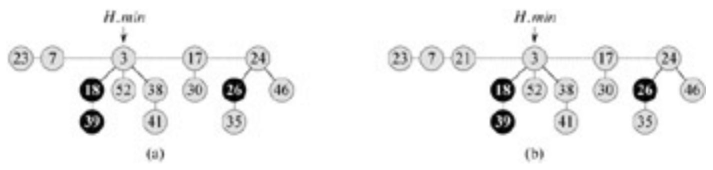

Figure 19.3: Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap *H*. **(b)** Fibonacci heap *H* after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

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 *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 Fibonzcci 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($H_1$, $H_2$)
1  $H$ = MAKE-FIB-HEAP()
2  $H.min = H_1.min$
3  concatenate the root list of $H_2$ with the root list of $H$
4  if ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5       $H.min = H_2.min$
6  $H.n = H_1.n + H_2.n$
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** $z \neq$ 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.

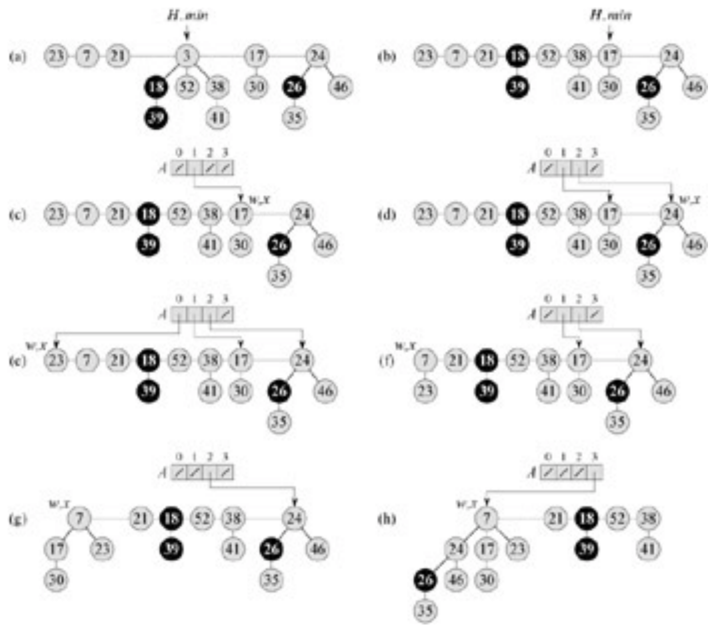

Figure 19.4: The action of FIB-HEAP-EXTRACT-MIN. **(a)** A Fibonacci heap *H*. **(b)** The situation after removing the minimum node *z* from the root list and adding its children to the root list. **(c)–(e)** The array *A* and the trees after each of the first three iterations of the **for** loop of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

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* ≤ *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*.

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 $t(H) + 2m(H)$, 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

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

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).

Previous Next