Fibonacci Heaps

Andres Mendez-Vazquez

November 7, 2018

Contents

1 Introduction

The Fibonacci heap data structure is used to support the operations "meargeable heap" operations:

1. MAKE -HEAP() creates and returns a new heap containing no elements.
2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
4. EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.
5. UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps are "destroyed" by this operation.
6. DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k.
7. DELETE(H, x) deletes element x from heap H.

The advantage of using Fibonacci heaps is in the fact that the amortized times are better than other implementations (Fig.)

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 1: Running Times Fibonacci Trees

2 Advantages of Fibonacci Heaps

From the theoretical point of view Fibonacci heaps are desirable for applications where Extract-Min and Delete operations are relative small with respect to the total number of operations performed. This situation arises in many applications as:

- Minimum Spanning Trees
- Single-Source Shortest paths
- Etc.

3 Before Fibonacci Heaps

It is necessary to review some definitions before we finally define what is Fibonacci Heap.

Definition 1. A free tree Is a connected acyclic undirected graph.

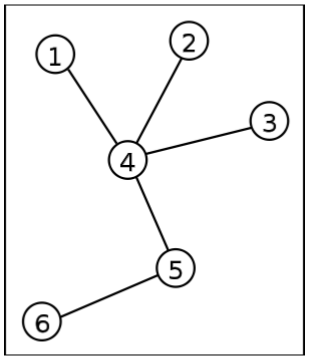

Figure 2: Free tree

Definition 2. A rooted tree is a free tree in which one of the nodes is a root.

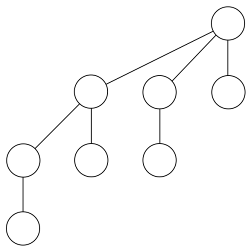

Figure 3: Binomial Tree

Definition 3. A ordered is a rooted tree where the children are ordered.

Finally we have that:

Definition 4. A Binomial Tree (Fig. 3) is a ordered tree defined recursively.

From this simple definition we can then prove the following lemma.

Lemma 1. *For the Binomial tree B_k,*

1. *There are 2^k nodes.*
2. *The Height of the tree is k.*
3. *There are exactly $\left(\begin{array}{c} k \\ i \end{array} \right)$ nodes at depth i for i=0,1,...,k.*
4. *The root has degree k, which is greater than that of any other node. It is more, if the children of the root are numbered from left to right by k-1, k-2,...,0, child i is the root of a subtree B_i*

   Proof. This is left to you. □

From here Fibonacci Heaps can be defined.

4 Fibonacci Heaps

The definition of a Fibonacci Heap is as follows.

Definition 5. 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.

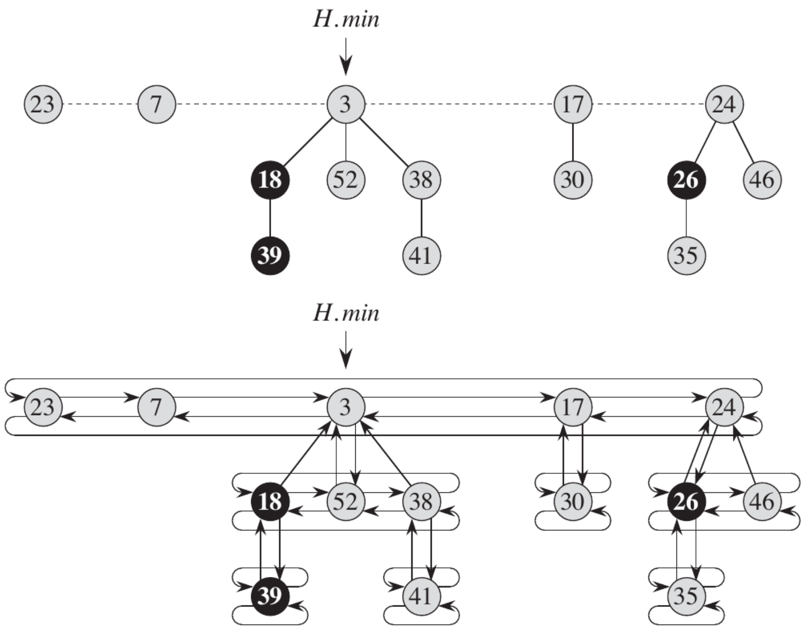

Figure 4: The Fibonacci Heap. The first figure represent the abstract construct, but the second contains the supporting underlying data structure. Here, the black nodes represent the marked nodes.

4.1 Data Structure Support

Although the definition look great, in reality we require to be able to support this abstract idea for the computer implementation. For this, we use the following at each node:

1. Each node contains a $x.parent$ and $x.child$ field.
2. The children of each a node x are linked together in a circular double linked list (Child list of x). The advantages of using this double linked list is the deletion or insertion anywhere in the list can be done in $O(1)$ time.
   (a) Each child y of x has a $y.left$ and $y.right$ to do this.
3. Each child has a field $degree$ and field $mark$.
   (a) The field $mark$ indicates whether a node has lost a child since the last time was made the child of another node. Newly created nodes are unmarked (Boolean value $FALSE$), and a node becomes unmarked whenever it is made the child of another node.

In addition,

- The roots of all the trees in a Fibonacci heap H 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 $H.min$ of the Fibonacci data structure thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list.
- The Fibonacci data structure has the field $H.n$ =the number of nodes currently in the Fibonacci Heap H.

Note: The main idea of th Fibonacci heap is to delay housekeeping work as long as possible.

4.2 Preview of the Amortized Analysis

The following potential function is used for the amortized analysis of the data structure.

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

Here,

- $t(H)$ the number of tree at the root list.
- $m(H)$ the number of marked nodes.

At (Fig. 4) has a potential of $\Phi(H) = t(H) + 2m(H) = 5 + 2 * 3 = 11$.

4.3 Degree Observation

Here, we will use an upper bound $D(n)$ for the degrees at each node in the Fibonacci Heaps. Here, a proof will be given such that when we have

- Mergeable-Heap Operations
  - Make Fibonacci Heap
  - Insertion
  - Find the Minimum
  - Fib-Heap-Extract-Min
  - Fib-Heap-Union
  - Consolidate
- Decrease-Key
- Delete

operations, it is possible to say that $D(n) = O(\lg n)$.

4.4 Operations

4.4.1 Meargeable-Heap Operation

The mergeable-heap operations on Fibonacci heaps delay work as long as possible.

Crating a new Fibonacci Heap The Make-Fib-Heap procedure returns a heap object H, where $H.n = 0$ and $H.min = NIL$. Then, $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$, thus the amortized cost of the operation is $O(1)$.

Insertion Here is the Basic Code.

Algorithm 1 Insertion

```
FIB-HEAP-INSERT(H, x)
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
    create a root list for H containing just x
    H.min = x
else insert x into H's root list
    if x.key < H.min.key
        H.min = x
H.n = H.n + 1
```

The amortized analysis is as follow:

- H is the input Fibonacci Heap and H' be the resulting Fibonacci Heap after insertion.
- $t(H') = t(H) + 1$, $m(H') = m(H)$.
- $c_i = O(1)$ this is because the number of steps to insert the node is a constant.

Thus, $\widehat{c_i} = c_i + \Phi(H') - \Phi(H) = O(1) + [t(H') + 2 \cdot m(H') - t(H) - 2 \cdot m(H)] = O(1) + [t(H) + 1 - t(H)] = O(1) + 1 = O(1)$. Then, insertion has an amortized cost of $O(1)$.

Finding the Minimum The code is way simpler, simply return the key of $min(H)$. The amortized cost is simply $O(1)$.

Union of two Fibonacci Heaps The union code only merges the root list and determines the minimum.

Algorithm 2 Union

```
FIB-HEAP-UNION($H_1, H_2$)
H = MAKE-FIB-HEAP()
$H.min = H_1.min$
concatenate the root list of $H_2$ with the root list of $H$
if ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
    $H.min = H_2.min$
$H.n = H_1.n + H_2.n$
return $H$
```

The amortized cost is as follow:

- $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$.
- $c_i = O(1)$ this is because the number of steps to make the union operations is a constant.

Thus, $\widehat{c_i} = c_i + \Phi(H) - [\Phi(H_1) + \Phi(H_2)] = O(1) + 0 = O(1)$. Then, the amortized cost of the union operation is $O(1)$.

Extracting the Minimum The extracting code is as follow.

Algorithm 3 Extract Min

```
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 $z$
```

Here, the code in lines 3-6 remove the node z and adds the children of z to the root list of H. Next, if the Fibonacci Heap is not empty a consolidation code is triggered. An example can be seen at the slides. The consolidate code is used to **eliminate** subtrees that have the same root degree by linking them. It repeatedly executes the following steps

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.

2. Link y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on y.

For this, the procedure uses an array $A\left[0, 1, ..., D\left(H.n\right)\right]$ to keep track of roots according to their degrees. An example of it is at the slides.

Algorithm 4 Consolidate

```
CONSOLIDATE($H$)
 1  let $A[0 \,..\, D(H.n)]$ be a new array
 2  for $i = 0$ to $D(H.n)$
 3      $A[i] = \text{NIL}$
 4  for each node $w$ in the root list of $H$
 5      $x = w$
 6      $d = x.degree$
 7      while $A[d] \neq \text{NIL}$
 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          $A[d] = \text{NIL}$
13          $d = d + 1$
14      $A[d] = x$
15  $H.min = \text{NIL}$
16  for $i = 0$ to $D(H.n)$
17      if $A[i] \neq \text{NIL}$
18          if $H.min == \text{NIL}$
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  $y.mark = \text{FALSE}$
```

The Consolidate code has two main parts

1. Lines 4-14 are used to consolidate the numbers of subtrees with same degree into a single tree using the array $A\left[0, 1, ..., D\left(H.n\right)\right]$. This lines can be proved to maintain the invariance $d = x.degree$ by the induction (I am leaving this to you).
2. Lines 15-23 clean the original Fibonacci Heap, then using the pointers at the array A, each subtree is inserted into the root list of H.

Amortized Analysis's Observations In order to make the analysis, we have the following observations:

1. The cost of FIB-EXTRACT-MIN contributes at most $O(D(n))$ because
   (a) The for loop at lines 3 to 5 in the code FIB-EXTRACT-MIN.
   (b) for loop at lines 2-3 and 16-23 of CONSOLIDATE.
2. The size of the root list when calling Consolidate is at most $D(n)+t(H)-1$ because the min root was extracted an it has at most $D(n)$ children.
3. At lines 4 to 14 in the CONSOLIDATE code the amount of work done by the for and the while loop is proportional to $D(n)+t(H)$ because each time we go through an element in the root list (for loop), the while loop consolidate the tree pointed by the pointer to a tree x with same degree.
4. Then, the actual cost is $c_i = O(D(n)+t(H))$.

Thus, assuming that H' is the new heap and H is the old one, we have that

- $\Phi(H) = t(H) + 2 \cdot m(H)$.
- $\Phi(H') = D(n) + 1 + 2 \cdot m(H)$ because H' has at most $D(n)+1$ elements and no node is marked in the process.

Then,

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(H') - \Phi(H) \\
&= O(D(n)+t(H)) + D(n) + 1 + 2 \cdot m(H) - t(H) - 2 \cdot m(H) \\
&= O(D(n)+t(H)) - t(H) \\
&= O(D(n)),
\end{aligned}$$

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

4.4.2 The Rest Of the Operations

Decreasing a Key The examples of the codes are at the slides. The FIB-HEAP-DECREASE works as follow:

1. 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.
2. If x is not a root and if $x.key \leq y.key$, where y is x's parent, then CUT and CASCADING-CUT are triggered.

Then CUT simply removes x from the child-list of y. The CASCADING-CUT uses the mark attributes to obtain the desired time bounds. The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

As soon as the second child has been lost, we cut y from its parent, making it a new root. Then:

- The attribute $y.mark$ is TRUE if steps 1 and 2 have occurred and one child of y has been cut. The CUT procedure, therefore, clears $y.mark$ in line 4, since it performs step 1. We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$.

Now we have a new problem, x might be the second child cut from its parent y to another node. Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y. We have three cases:

1. If y is a root return.
2. if y is not a root and it is unmarked then y is marked.
3. If y is not a root and it is marked, then y is CUT and a cascading cut is performed in its parent z.

Once all the cascading cuts are done, the $H.min$ us updated if necessary. The amortized analysis is then:

- Let H the Fibonacci Heap before the FIB-HEAP-DECREASE-KEY with $\Phi(H) = t(H) - 2 \cdot m(H)$
- $H'$is the final heap.
- Then, $c_i = O(1) + cascading - cuts$. Assume that you requiere c calls to cascade CASCADING-CUT (One for the line 7 at the code FIB-HEAP-DECREASE-KEY followed by $c-1$ others). Thus, $c_i = O(c)$.
- Finally, $\Phi(H') = t(H) + c$(The original trees + the ones created by the c calls)+ $2(m(H) - (c-1) + 1)$(The original marks - $(c-1)$ cleared marks by CUT + the branch to $y.mark$==FALSE true making $y.mark$=TRUE). Thus $\Phi(H') = t(H) + c + 2(m(H) - c + 2) = t(H) + 2m(H) - c + 4$

In this way, we have the amortized cost of decrease key is

$$\begin{aligned} \widehat{c}_i &= c_i + t(H) + 2m(H) - c + 4 - t(H) - 2m(H) \\ &= c_i + 4 - c = O(c) + 4 - c = O(1) \end{aligned}$$

Delete a key Code and Analysis can be seen at the slides.

4.5 Proving the $D(n)$ bound!!!

For this, we define a quantity $size(x)$= the number of nodes at subtree rooted at x, x itself. We will prove that $size(x)$ is exponential in $x.degree$.

Lemma 19.1

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

Proof: We know that $y_1.degree \geq 0$. Now, for $i \geq 2$, y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$ were children of x, and so we must have had $x.degree \geq i-1$. Then, node y_i is linked to x (by CONSOLIDATE) only if $x.degree = y_i.degree$, we must have also had $y_i.degree \geq i-1$ at that time. Since then, node y_i has lost at most one child, since it would have been cut from x (by CASCADING -CUT) if it had lost two children. We conclude that $y_i.degree \geq i-2$.

Then, using the Fibonacci sequence definition:

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

We can prove the following lemmas.

Lemma 19.2

For all integers $k \geq 0$, $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.

Proof: For $k = 0$, we have that $1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = F_2$.

Assume the inductive hypothesis that $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, then:

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

Lemma 19.3

For all integers $k \geq 0$, $(k+2)\,nd$ Fibonacci number satisfies $F_{k+1} \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$.

Proof: For $k = 0$, we have that $F_2 = 1 = \Phi^0$, and when $k = 1 \Rightarrow F_3 = 2 > 1.619 > \Phi$. Now, we assume that $F_{i+2} > \Phi^i$ for $i = 0, 1, 2, ..., k-1$. Now, we know that Φ is a positive root of $x^2 = x + 1$. Thus,

$$\begin{aligned}
F_{k+2} &= F_{k+1} + F_k \\
&\geq \Phi^{k-1} + \Phi^{k-2} \\
&= \Phi^{k-2}\,(\Phi + 1) \\
&= \Phi^{k-2} \cdot \Phi^2 \\
&= \Phi^k.
\end{aligned}$$

Here, we prove the relation between $size(x)$ and the degree of any node of a Fibonacci heap.

Lemma 19.4

Let x be any node in a Fibonacci Heap, $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \Phi^k$.

Proof: Let s_k denote the minimum possible size of any node with degree k. It is trivial that $s_0 = 1$ and $s_1 = 2$. In addition, $s_k \leq size(x)$ and for $k \leq k' \Rightarrow s_k \leq s_{k'}$. Now, consider some node x, in any Fibonacci heap, such that $x.degree = k$ and $size(x) = s_k$. Therefore, if bound from below s_k, we bound from below $size(x)$. Using Lemma 19.1, let $y_1, ..., y_k$ denote the children of z in the order in which they were linked to z. Thus

$$\begin{aligned}
size(x) &\geq s_k \\
&\geq 1(x \text{ itself}) + 1(\text{ For the first child } y_1) + \sum_{i=2}^{k} s_{y_i.degree} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2},
\end{aligned}$$

Now by using induction on k, we can prove that $s_k \geq F_{k+2}$. This is trivial for $k = 0$ and $k = 1$. Assume that is true for $k \geq 2$ then $s_i \geq F_{i+2}$ for $i = 0, 1, ..., k-1$.

$$\begin{aligned}
s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} F_i \\
&= 1 + \sum_{i=1}^{k} F_i \\
&= F_{k+2} \geq \Phi^k.
\end{aligned}$$

Then, $size(x) \geq \Phi^k$.

Corollary 19.5

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

Proof: Let x be any node in a n-node Fibonacci heap, and $k = x.degree$. By Lemma 19.4 $n \geq size(x) \geq \Phi^k \Rightarrow k \leq log_\Phi n \leq \log_2 n \times log_\Phi n$ for all $n > n_0$. Then, the maximum degree of any node is thus bounded by $O(\log_2 n)$.