Fibonacci heaps

Definition. A Fibonacci heap (Fredman & Tarjan 1987) is a data structure consisting of key-item pairs,

\[ \{ (k_1, x_1), (k_2, x_2), \ldots, (k_n, x_n) \} \]

where

- the keys \( k_i \) are drawn from a totally-ordered domain, and are not necessarily distinct, while
- the items \( x_i \) are unordered, and are all distinct.

A Fibonacci heap supports the following operations:

- **Create ()** Create and return an empty heap.

- **Insert \((k, x, H)\)** Insert pair \((k, x)\) into heap \( H \), and return a pointer to the pair.
  Assumes item \( x \) is not in \( H \).

- **Delete \((p, H)\)** Delete pair \( p \) (returned from a prior Insert) from \( H \).
Definition, cont'd

- **Union (A, B)**: Destructively merge the two heaps A and B into one heap, and return the merged heap. Assumes the items in A, B are disjoint.

- **Minimum (H)**: Return the item in H whose associated key is minimum.

- **Extract (H)**: Delete the pair in H with minimum key.

- **Decrease (p, k, H)**: Decrease the key of pair p in H to k. If k is not less than the current key of p, this has no effect.
### Remarks

- Fibonacci heaps support these operations in the following time bounds, using $\Theta(n)$ space:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst-case</th>
<th>Amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Union</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Minimum</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Decrease</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Extract</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>

Fibonacci heaps are efficient in an amortized sense, but not in the worst case. This is usually sufficient, as most applications of heaps perform a series of operations. (A heap is now known that achieves the amortized time of Fibonacci heaps in the worst-case (Brodal 1996), but it is very complicated.)
Remarks, cont'd

• Fibonacci heaps are most useful when a few Extract and many Decrease operations are performed. This occurs in many graph algorithms, such as for shortest paths and minimum spanning trees.

• Fibonacci heaps tend to be slow in practice. Another data structure, the pairing heap (Fredman, Sedgewick, Sleator, Tarjan 1986), tends to be fast in practice, but is theoretically less efficient (Fredman 1998).

• Note that in contrast to search trees, Fibonacci heaps do not support the Find operation on a key. (Heap elements are accessed through the pointer returned by an Insert.)
Representation

- A Fibonacci heap is represented as a forest of heap-ordered trees with arbitrary number of children:

```
  min
  ↓
one heap
```

- Along root-to-leaf paths, keys are nondecreasing.
- Roots and children are unordered.

- For the heap, we maintain:
  - a linked list of roots,
  - a pointer to the root with minimum key, and
  - a count of the total number of nodes.

- At each node, we maintain:
  - a linked list of children,
  - a pointer to its parent,
  - a count of the number of children (called its degree), and
  - a bit (called its mark).
Potential function

• Let

\[ R(H) := \text{number of roots in } H, \text{ and} \]
\[ M(H) := \text{number of marked nodes in } H. \]

In the analysis, we use the following potential function:

\[ \Phi(H) := R(H) + 2M(H). \]

The potential of a collection of heaps (when performing Union operations) is the sum of their potentials.

• Note that

\[ \Phi(H_0) = 0, \text{ and} \]
\[ \Phi(H_i) \geq 0 \text{ for all } i \geq 1, \]

so \( \Phi \) is a valid potential function.
Implementation of simple operations

Create

- Just create an empty heap $H$.
- This takes $\Theta(1)$ worst-case time, and since $\Delta M(H) = 0$, it also takes $\Theta(1)$ amortized time.

Insert

- Just create a new tree consisting of the single unmarked node $(k, x)$, add it to the forest for $H$, and update the minimum root pointer by comparing with $k$:

  $$
  \begin{align*}
  H & \quad \rightarrow \quad H' \\
  \bigtriangleup \quad \cdots \quad \bigtriangleup & \quad \rightarrow \quad \bigtriangleup \quad \cdots \quad \bigtriangleup (k, x)
  \end{align*}
  $$

- This takes amortized time:

  $$
  \Theta(1) + \frac{\Delta R(H) + 2 \Delta M(H)}{\Delta M(H)} = \Theta(1) + (1 + 2 \cdot 0) = \Theta(1).
  $$
Simple operations, cont.

Minimum

- Just return the item at the root pointed at by the minimum pointer.
- This takes $\Theta(1)$ amortized time, as $\Delta \Xi = 0$.

Union

- Just concatenate the root lists of heaps $A$ and $B$ to form $H$, and compare the minimum pointers of $A$ and $B$ to determine $H$'s minimum.
- This takes $\Theta(1)$ amortized time, since

$$
\Delta \Xi = \Xi(H) - (\Xi(A) + \Xi(B)) \\
= 0.
$$
Extract

Idea

- Remove from \( H \) the node \( r \) pointed at by the minimum root pointer.

- Concatenate the children of \( r \) onto \( H \)'s root list.

- Scan the root list to determine the new minimum root.

- Since scanning is expensive, consolidate the root list by making some roots children of others. (This reduces the number of roots in the forest to speed up future Extracts.)

- Pictorially,
Implementation of Extract

procedure Consolidate(H) begin

\[ d := f(\text{Size}(H)) \]
\[ A := \text{Array}(0, d) \]  
\[ \Theta(d) \{ \text{for } i := 0 \text{ to } d \text{ do} \]
\[ A[i] := \text{Nil} \]
\[ \text{time} \]

\[ \text{for each root } r \text{ of } H \text{ do begin} \]
\[ s := A[\text{Degree}[r]] \]
\[ \text{while } s \neq \text{Nil} \text{ do begin} \]
\[ \text{if Key}[r] > \text{Key}[s] \text{ then} \]
\[ \text{Swap } r, s. \]
\[ \text{Remove } s \text{ from the root list of } H. \]
\[ \text{Make } s \text{ a child of } r. \]
\[ A[\text{Degree}[s]] := \text{Nil} \]
\[ \text{Degree}[r] := 1 \]
\[ \text{Mark}[s] := \text{False} \]
\[ s := A[\text{Degree}[r]] \]
\[ \text{end} \]
\[ A[\text{Degree}[r]] := r \]
\[ \text{end} \]

\[ \Theta(d) \{ \text{Scan } A[0:d] \text{ to collect the new root list of } H, \]
\[ \text{and update the minimum root pointer.} \]
\[ \text{time} \]
Analysis of Extract

- We measure the time for an Extract by counting the number of times roots are linked, and the number of times an array element is accessed. Thus the actual time is at most:

\[
\frac{f(n)}{ \text{concatenate the list of children of the extracted node onto the root list}} + \frac{f(n) + R(H) - 1}{\text{links the roots during consolidation}} + \frac{f(n) + 1}{\text{initialize the array}} + \frac{f(n) + 1}{\text{scan the array to collect the final root list}} = 4f(n) + R(H) + 1.
\]

- The change in potential \( \Delta \Phi \) is at most:

\[
\left( \frac{f(n) + 1 + 2M(H)}{\text{upper bound on final number of roots}} \right) - \left( \frac{R(H) + 2M(H)}{\text{no new nodes are marked}} \right) = \Phi(D') - \Phi(D) = f(n) - R(H) + 1.
\]
Analysis of Extract, cont'd

- So the amortized time for an Extract is at most:

\[
\frac{4f(n) + R(H) + 1}{\text{actual time}} + \frac{f(n) - R(H) + 1}{\text{change in potential}}
\]

\[
= 5f(n) + 2
\]

\[
= O(f(n))
\]

where \( f(n) \) is an upper bound on the maximum degree in an \( n \)-node Fibonacci heap.

- Intuitively, the time spent linking roots during an Extract is compensated by the reduction in the number of roots, as captured by the potential function.
Decrease \((p, k, H)\)

**Idea**

- Decrease the key of node \(p\).
  Let \(q\) be its parent.
  If \(p\)'s key is now less than \(q\)'s key (so heap order is violated), cut the link from \(p\) to \(q\) and make \(p\) a new root.

- If this causes \(q\) to have lost two children since the time \(q\) was linked to its parent, cut the link from \(q\) to its parent and make \(q\) a new root.
  Continue this test at \(q\)'s parent.
  (This is called a cascading cut.)
Idea of Decrease, cont.

- Pictorially,

\[ H \rightarrow H' \]

Cascading cut

Lost 2 children: cut.
Lost 2 children: cut.
Heap order violated: cut.

Several new roots.

\[ p \ q \ q' \rightarrow \]
Idea of Decrease, cont'd

- We detect whether a node has lost 2 children by its mark:
  - When \( q \) is linked to its parent, we set \( \text{Mark}[q] := \text{False} \).
    (Examine procedure \text{Consolidate}.)
  - When \( q \) loses a child, and \( \text{Mark}[q] = \text{False} \), we set \( \text{Mark}[q] := \text{True} \).
  - When \( q \) loses a child, and \( \text{Mark}[q] = \text{True} \), we continue cascading to \( q \)'s parent.
Implementation of Decrease

procedure Decrease(p, k, H) begin
    Key[p] := min{Key[p], k}
    q := Parent[p]
    if q ≠ Nil and Key[p] < Key[q] then begin
        Cut(p, H)
        Cascade(q, H)
    end
    Update the minimum root pointer for H by comparing with Key[p].
end

procedure Cut(p, H) begin
    q := Parent[p]
    Remove p from q's child list.
    Degree[q] := 1
    Parent[p] := Nil
    Add p to the root list of H.
    Mark[p] := False
end
procedure Cascade (p, H) begin

  • Node p has just lost a child.
  Consider a cascading cut at p.

  while Parent[p] ≠ Nil and Mark[p] do begin
    q := Parent[p]
    Cut (p, H)
    p := q
  end

  if Parent[p] ≠ Nil then
    Mark[p] := True  • Note that
    Mark[p] = False.
end
Analysis of Decrease

• We measure the actual time for Decrease by:

\[
\frac{1}{\text{time outside Cut and Cascade}} + \frac{c}{\text{number of calls to Cut (including those in Cascade)}}
\]

since the time taken by Decrease is \(\Theta(c+1)\).

• We bound the change in potential as follows. Each call to Cut creates one new root. Moreover, each call to Cut with Cascade unmarks a marked node. In addition, the call to Cascade may mark a new node. Thus,

\[
\Delta \Phi = \Delta R + 2 \Delta M \\
\leq c + 2(-c+1) + 1 \\
\leq 4 - c.
\]
Analysis of Decrease, cont'd

• Thus the amortized time for Decrease is at most:

\[
\frac{(1 + c)}{\text{actual time}} + \frac{(4 - c)}{\text{change in potential}} = 5.
\]

So Decrease takes \(O(1)\) amortized time.

• Intuitively, the factor of two in the \(2M(H)\) term in \(\Phi(H)\) is needed so that, when unmarking a node,

  • one unit pays for the cut, and
  • the other unit pays for the addition of a new root to the potential.
Delete \((p, H)\)

- We simply perform:
  - \(\text{Decrease } (p, -\infty, H)\), followed by
  - \(\text{Extract } (H)\).

- By our analysis of \(\text{Decrease}\) and \(\text{Extract}\), this takes

\[
O(1) + O(f(n)) = O(f(n))
\]

amortized time, where \(f(n)\) is again our upper bound on the maximum degree in an \(n\)-node Fibonacci heap.
Bounding \( f(n) \), cont'd

- We now turn to examining Fibonacci heaps with Decrease and Delete.

**Definition 2** The Fibonacci tree \( T_k \), for \( k \geq 0 \), is defined inductively by:

\[
T_k \begin{cases} 
  0, & k = 0 \\
  \cup, & k = 1 \\
  T_{k-1} \cup T_{k-2}, & k \geq 2.
\end{cases}
\]

**Example**

<table>
<thead>
<tr>
<th>Degree of root</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of tree</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>...</td>
</tr>
</tbody>
</table>
Bounding $f(n)$, cont'd

Definition 3  The Fibonacci number $F_k$, for $k \geq 0$, is defined inductively by:

$$F_k := \begin{cases} 
0, & k = 0; \\
1, & k = 1; \\
F_{k-2} + F_{k-1}, & k \geq 2.
\end{cases}$$

Example

<table>
<thead>
<tr>
<th>$F_0$</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_4$</th>
<th>$F_5$</th>
<th>$F_6$</th>
<th>$F_7$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>
Lemma 3  The Fibonacci tree $T_k$ has

- maximum degree $k$, which is the degree of its root, and
- size $F_{k+2}$.

Proof  By induction on $k$.

Basis ($k = 0, 1$)

\[
\begin{align*}
T_0 &= 0 & \text{max. deg.} & 0 & \text{size} & 1 = F_2 \\
T_1 &= 1 & 1 & 2 = F_3
\end{align*}
\]

Thus the basis holds.

Induction ($k \geq 2$)

By induction, the maximum degree of $T_k$ is the degree of the root of $T_k$, which is $1 + (k-1) = k$.

By induction, the size of $T_k$ is

\[
F_{(k-2)+2} + F_{(k-1)+2} = F_k + F_{k+1} = F_{k+2}.
\]

\[\square\]
Lemma 4: For all \( k \geq 0 \),

\[
F_k = \frac{1}{\sqrt{5}} \left( \phi^k - \hat{\phi}^k \right),
\]

where

\[
\phi = \frac{1}{2} \left( 1 + \sqrt{5} \right) > 1.618,
\]

\[
\hat{\phi} = \frac{1}{2} \left( 1 - \sqrt{5} \right) < -0.618.
\]

Proof: By induction on \( k \).

Basis \((k = 0, 1)\)

\[
\frac{1}{\sqrt{5}} \left( \phi^0 - \hat{\phi}^0 \right) = 0 = F_0.
\]

\[
\frac{1}{\sqrt{5}} \left( \phi^1 - \hat{\phi}^1 \right) = \frac{1}{\sqrt{5}} \left( \frac{2 \sqrt{5}}{2} \right) = 1 = F_1.
\]

Induction \((k > 2)\)

First notice that \( \phi \) and \( \hat{\phi} \) satisfy

\[
1 + \phi = \phi^2, \tag{\text{*}}
\]

\[
1 + \hat{\phi} = \hat{\phi}^2.
\]

So,

\[
F_k = F_{k-2} + F_{k-1}
\]

\[
= \frac{1}{\sqrt{5}} \left( \phi^{k-2} - \hat{\phi}^{k-2} \right) + \frac{1}{\sqrt{5}} \left( \phi^{k-1} - \hat{\phi}^{k-1} \right)
\]

\[
= \frac{1}{\sqrt{5}} \left( \phi^{k-2} \left( 1 + \phi \right) - \hat{\phi}^{k-2} \left( 1 + \hat{\phi} \right) \right)
\]

\[
= \frac{1}{\sqrt{5}} \left( \phi^k - \hat{\phi}^k \right), \quad \text{by (\text{*})}
\]
Bounding $f(n)$, cont'd

Corollary 3  A Fibonacci tree on $n$ nodes has maximum degree $\Theta(\log n)$.

Proof  By Lemma 4, for all $k \geq 0$,

$$\frac{1}{\sqrt{5}} (\varphi^k - |\hat{\varphi}|^k) \leq F_k \leq \frac{1}{\sqrt{5}} (\varphi^k + |\hat{\varphi}|^k).$$

Since $|\hat{\varphi}|^k \leq 1$ for all $k \geq 0$,

$$\frac{1}{\sqrt{5}} (\varphi^k - 1) \leq F_k \leq \frac{1}{\sqrt{5}} (\varphi^k + 1).$$

Since $\frac{2}{\sqrt{5}} \varphi^k \geq 1$ for all $k \geq 2$,

$$\frac{3}{5 \sqrt{5}} \varphi^k \leq F_k \leq \frac{7}{5 \sqrt{5}} \varphi^k.$$

Hence

$$F_k = \Theta(\varphi^k).$$

Taking logarithms and noting $\varphi^k = \omega(1)$,

$$\log \varphi F_k = \Theta(k). \quad (*)$$

Let $d$ be the maximum degree of a Fibonacci tree $T$. By Lemma 3, $T$ has $n = F_{d+2}$ nodes. Thus by ($*$),

$$\log n = \Theta(d+2) = \Theta(d).$$

Hence

$$d = \Theta(\log n).$$
Bounding $f(n)$, cont'd

- We need one more fact about Fibonacci trees.

**Lemma 5** For $k \geq 2$, Fibonacci tree $T_k$ has the structure,

\[
T_k = \begin{array}{c}
\uparrow \\
T_{k-2} \quad T_1 \quad T_0 \quad T_0
\end{array}
\]

**Proof** By induction on $k$.

**Basis** ($k = 2$)

\[
T_2 = \begin{array}{c}
\uparrow \\
T_0 \quad T_1
\end{array} = \begin{array}{c}
\uparrow \\
T_0 \quad T_0
\end{array}
\]

**Induction** ($k > 2$)

\[
T_k = \begin{array}{c}
\uparrow \\
T_{k-2} \quad T_{k-1}
\end{array} = \begin{array}{c}
\uparrow \\
T_{k-2} \quad T_{k-3} \quad T_0 \quad T_0 \quad T_0
\end{array}
\]

by def'n \hspace{1cm} \text{by ind. hyp.}
Bounding $f(n)$, cont’d

- We are now ready to relate Fibonacci heaps to Fibonacci trees.

Lemma 6

In a Fibonacci heap, the smallest possible subtree rooted at a node of degree $k$ is the Fibonacci tree $T_k$.

Proof. By induction on $k$.

Basis ($k = 0, 1$)

Note that $T_0$ and $T_1$ are the smallest possible trees with roots of degree 0 and 1, and that they can be formed by Fibonacci heap operations.

Induction ($k > 2$)

Let $v$ be a node of degree $k$ in a Fibonacci heap, and number its children $w_1, w_2, \ldots, w_k$ in the order they were linked to $v$:

```
  v
   / \  \
  w_1 \  / \  /  /  /  \
  w_2  w_i  w_k
```
We claim that the degree of $w_i$, for $1 \leq i \leq k$, is at least $\max \{ i-2, 0 \}$.

To see this, note that when $w_i$ was linked to $v$, they had the same degree. At that point, $v$ had degree at least $i-1$. Hence, when linked to $v$, $w_i$ had degree at least $i-1$. Since the link, $w_i$ can have lost at most one child. (Otherwise, $w_i$ could not be a child of $v$.) Thus $w_i$ has degree at least $i-2$.

The smallest possible subtree at $v$ must consist of smallest possible subtrees at $w_1, w_2, \ldots, w_k$.

From the claim, these are $T_0, T_0, T_1, \ldots, T_{k-2}$ by induction:

![Diagram](attachment:image.png)

By Lemma 5, this tree is $T_k$.

Finally, observe that $T_k$ can be built by a series of Fibonacci heap operations. (Exercise.)
**Theorem**  In a Fibonacci heap on \( n \) nodes, the maximum degree, \( f(n) \), is \( O(\log n) \).

**Proof** Given a Fibonacci heap \( H \) with \( n \) nodes, let

- \( k \) be its maximum degree, and
- \( m \) be the size of the smallest subtree of \( H \) rooted at a node of degree \( k \).

We have,

\[
\begin{align*}
    k &= O(\log |T_k|) & \text{by Lemma 3, Corollary 3,} \\
        &= O(\log m) & \text{by Lemma 6,} \\
        &= O(\log n) & \text{since } m \leq n.
\end{align*}
\]

**Remark** Tracing the details of the proof, we get

\[
f(n) \leq \left\lfloor \log \frac{5n^{5/3}}{3} \right\rfloor - 2.
\]
Corollary On a Fibonacci heap of \( n \) nodes,

- Delete, Extract take \( O(\log n) \) amortized time, and
- Create, Insert, Union, Minimum, Decrease take \( O(1) \) amortized time.