Fibonacci Heaps

CLRS: Chapter 20

Last Revision: September 21, 2006
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Binomial heap (worst-case)</th>
<th>Fibonacci heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make-Heap</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Minimum</td>
<td>$\Theta(1)$</td>
<td>$O(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Extract-Min</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>Union</td>
<td>$\Theta(n)$</td>
<td>$O(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Decrease-Key</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
</tbody>
</table>

So far we have seen *Binomial heaps* and learnt some techniques for performing *amortized analysis*.

In this section we will design *Fibonacci heaps*, whose running times will be *amortized* and not worst case. Even with only amortized running times, Fibonacci heaps provide enough of an improvement to reduce the runtime of *Dijkstra’s* and *Prim’s* algorithms from $(|V| + |E|) \log |V|$ down to $|V| \log |V| + |E|$.

Our amortized analysis will use the *Potential method*. 
Fibonacci heaps, like Binomial heaps, are a collection of heap-ordered trees.

Some properties

- nodes in a F.H are not ordered (by degree) in the root list or as siblings.

- (root and sibling) lists kept as circularly-linked lists. Allows constant time deletion/insertion/concatenation.

- Each node stores its degree (number of children).

- \( \text{min}[H] \) is a pointer to minimum root in root list.

- \( N(H) \) keeps number of nodes currently in \( H \).
Marked Nodes:
Some nodes will be marked (indicated by the marked bit set to 1).
(i) A node $x$ will be marked if $x$ has lost a child since the last time that $x$ was made a child of another node.
(ii) Newly created nodes are unmarked
(iii) When node $x$ becomes child of another node it becomes unmarked.

Potential Function:
The potential of $H$ will be $t(H)$, the number of nodes in root list of $H$ plus two times $m(H)$, the number of marked nodes.

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

In example above $\Phi(H) = 5 + 2 \cdot 3 = 11$. 
**Assumption:** There is a maximum degree $D(n)$ on the degree of any node in an $n$-node Fibonacci heap.

We will prove later that $D(n) = O(\log n)$. 
**Make-Heap():**
This is a very easy $O(1)$ (both amortized and actual) operation.

**Minimum($H$):** Return the node pointed to by $min[H]$. This takes $O(1)$ actual time. The heap does not change before and after this operation so difference in potential is 0. Amortized cost is then also $O(1)$. 
**H’=Insert(H, x):**

Create new tree containing \( x \) & add it to root list. \( Min[H] = \min(Min[H], x) \).

Update pointers appropriately

**Do not** combine items in the root list.

*Clean up will be done during Extract-Min(\(H\)).*

If \( k \) nodes inserted into \( H \), then \( H \) becomes a linked list with \( k \) single nodes.

Actual cost \( c \) of operation is \( O(1) \).

\[
t(H') = t(H) + 1; \quad m(H') = m(H)
\]

so

\[
\Phi(H') - \Phi(H) = ((t(H') + 2m(H')) - (t(H) + 2m(H))) = 1
\]

and amortized cost satisfies

\[
\hat{c} = c + 1 = O(1).
\]
**H=Union**($H_1, H_2$):

Just concatenate the two root lists of $H_1$ and $H_2$.

**Do not** combine items in the root list.

Set $\text{min}[H] = \min(\text{min}[H_1], \text{min}[H_2])$.

Actual cost of this operation is $c = O(1)$.

Concatenating the root lists does not change the total number of items in the root lists or the total number of marked nodes so change in potential is

\[
\Phi(H) - (\Phi(H_1) + \Phi(H_2))
= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))
= 0
\]

and amortized cost is

\[
\tilde{c} = c + 0 = O(1).
\]
**Extract-Min(\(H\)):**
This is the most complicated operation. It is here where we *clean-up* large root lists. At the end of this operation, root list will contain at most one root of each possible degree. This implies that root list contains 
\[ \leq D(n - 1) + 1 \] nodes.

**Extract-Min(\(H\))** is quite similar to same operation in binomial heaps.
Let \(A\) be tree with root \(\text{min}[H]\).
Extract \(A\) from \(H\).
Remove the root of \(A\);
    reinsert remaining trees back into root list of heap.
    update \(\text{min}[H]\) during this procedure.
Link roots of equal degree until at most one root remains of each degree.

Let \(x\) be root of tree \(X\), \(y\) root of \(Y\).
Assume w.l.o.g. that \(\text{key}[x] \leq \text{key}[y]\).
When linking \(X\) and \(Y\) point \(y\) to \(x\) and increment \(\text{degree}(x)\) and set \(\text{mark}(y)=0\).
Actual Cost:
A has at most $D(n)$ children.
After concatenation there will be at most 
$$t(H) + D(n) - 1$$ nodes on root list.
Linking any two trees requires $O(1)$.
Total cost of concatenating, linking and updating $\min[H]$ is $O(t(H) + D(n))$.
Actual cost is then $c = O(t(H) + D(n))$.

Potential:
Original potential is $\Phi(H) = t(H) + 2m(H)$.
Marked nodes can not be created by operation; only cleared.
After all of the linking there will be at most $D(n - 1) + 1$ nodes on root list (Why?).

Final potential is then 
$$\Phi(H') = t(H') + 2m(H') \leq D(n) + 1 + 2m(H).$$
and
$$\Phi(H') - \Phi(H) \leq D(n) + 1 + 2m(H) - (t(H) + 2m(H)) = D(n) + 1 - t(H).$$
Actual Cost:
\[ c = O(t(H) + D(n)). \]

Potential:
\[ \Phi(H') - \Phi(H) \leq D(n) + 1 - t(H). \]

Amortized Cost:
If we scale the units of potential large enough then amortized cost is
\[
\hat{c} = c + \Phi(H') - \Phi(H) \\
\leq O(t(H) + D(n)) + D(n) + 1 - t(H) \\
= O(D(n))
\]
so
\[ \hat{c} = O(\log n). \]
**Decrease-Key**\((H, x, k)\):

Let \(p[x] = parent[x]\).

This operation is *very* different from any we’ve seen before.

We actually **Cut** subtree rooted at \(x\) out of the tree, move it to the root list and unmark \(x\).

We then look at \(p[x]\);
if it wasn’t marked, we mark it and stop.

If it was marked, we cut \(p[x]\), unmark it, move it to the root list, and then check \(p[p[x]]\), cascading this process up until either an unmarked ancestor or the root is found.
Decrease-Key($H$, $x$, $k$)

(i) First check if $k < key[p[x]]$. If not, do nothing.

(ii) Otherwise

\begin{align*}
& \text{Cut}(H, x, p[x]) \\
& \text{CascadingCut}(H, p[x])
\end{align*}

(iii) If $k < key[min[H]]$ then

\begin{align*}
& min[H] = x
\end{align*}

Cut($H$, $x$, $y$):  

(i) Move tree rooted at $x$ to root list; 
 decrement $degree[y]$. 

(ii) set $mark[x] = false$

CascadingCut($H$, $y$)

If $y$ not in root list

\begin{align*}
\text{then if } mark[y] == false \\
& \text{then } mark[y] == true \\
\text{else Cut}(H, y, p[y]) \\
& \text{CascadingCut}(H, p[y])
\end{align*}
Node containing 46, has key decreased to 15.

It is cut and moved to root list.

Its parent, 24, is then marked.
35 changed to 5, cut & moved to root
Its parent 26 was marked; so it’s cut as well, moved to root and mark cleared.
26’s parent 24 was marked; so it’s cut as well, moved to root and mark cleared.
24’s parent 7 is in root list so cascade terminates.
Decrease-Key\((H, x, k)\): Run Time

**Actual Cost:**
If CascadingCut is recursively called \(d\) times, 
\(c = O(d + 1)\).

**Potential:**
Original potential is \(\Phi = t(H) + 2m(H)\).

After operation there are 
\(t(H) + d\) trees in root list and at most 
\(m(H) - (d - 1) + 1 = m(H) - d + 2\) marked nodes. Then
\[\Phi(H') - \Phi(H) \leq 4 - d.\]

**Amortized Cost:**
If we scale the units of potential to be large enough then
\[
\tilde{c} = c + \Phi(H') - \Phi(H) \\
\leq O(d + 1) + 4 - d \\
= O(1)
\]
We can now begin to understand why the marked nodes contribute $2m(H)$ to potential.

The first unit of potential for each marked node was to pay for a step in the cascaded cut.

The second unit was to pay for the increase in potential caused by a cut node becoming a root, which in turn pays for a later linking of that root to another root during an Extract-Min.
Delete($H, x$):

This is equivalent to

an amortized $O(1)$ time $\text{Decrease-Key}(H, x, -\infty)$ and

an amortized $O(\log n)$ time $\text{Extract-Min}(H)$
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Binomial heap (worst-case)</th>
<th>Fibonacci heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make-Heap</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Insert</td>
<td>Θ(lg (n))</td>
<td>O(lg (n))</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Minimum</td>
<td>Θ(1)</td>
<td>O(lg (n))</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Extract-Min</td>
<td>Θ(lg (n))</td>
<td>Θ(lg (n))</td>
<td>O(lg (n))</td>
</tr>
<tr>
<td>Union</td>
<td>Θ((n))</td>
<td>O(lg (n))</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Decrease-Key</td>
<td>Θ(lg (n))</td>
<td>Θ(lg (n))</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Delete</td>
<td>Θ(lg (n))</td>
<td>Θ(lg (n))</td>
<td>O(lg (n))</td>
</tr>
</tbody>
</table>

We have demonstrated that Fibonacci Heaps satisfy the stated amortized running times under the assumption that \(D(n) = O(\log n)\) where \(D(n)\) is the maximum degree of a tree in a Fibonacci Heap containing \(n\) nodes.

We still have to

- Prove that \(D(n) = O(\log n)\)
- Explain why the data structure is called a Fibonacci heap.
Define trees $T_i$ as follows: $T_0$ is a single node, $T_1$ is a node with one child and, for $i > 1$, $T_i$ is constructed by pointing the root of a $T_{i-2}$ to the root of a $T_{i-1}$.

It is easy to see that $|T_i| = F_{i+2} > \phi^i$ where $F_i$ is the $i$th Fibonacci number and $\phi = (1 + \sqrt{5})/2$.

Our analysis will essentially will show that if a node in a Fibonacci heap has degree $k$, then the tree rooted at that node must include $T_k$. This means that no node can have degree greater than $\log_\phi n$ so $D(n) = O(\log n)$. 

22
Recall that

\[ F_k = \begin{cases} 
0 & \text{if } k = 0, \\
1 & \text{if } k = 1, \\
F_{k-1} + F_{k-2} & \text{if } k \geq 2. 
\end{cases} \]

We will need the following fact

**Lemma:** \( \forall k \geq 0, \ F_{k+2} = 1 + \sum_{i=0}^{k} F_i. \)

**Proof:** By induction. Proof is obviously true for \( k = 0. \) For \( k > 0, \)

\[
F_{k+2} = F_k + F_{k+1} \\
= F_k + \left( 1 + \sum_{i=0}^{k-1} F_i \right) \\
= 1 + \sum_{i=0}^{k} F_i
\]
Lemma:
Let $x$ be any node in a F. heap.
Let $y_1, y_2, \ldots, y_k$ be current children of $x$ in order in which they were linked to $x$.
Then $\text{degree}[y_1] \geq 0$ and
$\forall i > 1, \text{degree}[y_i] \geq i - 2$.

Proof: $\text{degree}[y_1] \geq 0$ trivially.
In other cases, when $y_i$ linked to $x$, all of $y_1, y_2, \ldots, y_{i-1}$ were already children so $\text{degree}[x] \geq i - 1$.

Node $y_i$ is only linked to $x$ when $\text{degree}[y_i] = \text{degree}[x]$ so at that time $\text{degree}[y_i] \geq i - 1$.

Since then $y_i$ could have lost at most one more child (otherwise it would have been cut and put in root list).
So, as long as $y_i$ is linked to $x$, $\text{degree}[y_i] \geq i - 2$. 
Lemma:
Let $x$ be any node in a F. heap and $k = \text{degree}[x]$. Then $size(x) =$ number of nodes in tree rooted at $x$ is $\geq F_{k+2} \geq \phi^k$.

Proof: Let $s_k$ be the minimum value of $size(x)$ for a node $x$ with degree $k$. It is easy to see that $s_0 = 1$, $s_1 = 2$ and $s_2 = 3$. Also easy to see that $s_i \geq s_{i-1}$.

As before, let $y_1, y_2, \ldots, y_k$ be current children of $x$ in order in which they were linked to $x$.

Then, counting 1 for $x$ and another 1 for $size(y_1)$,

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} s_{\text{degree}[y_i]}$$

$$\geq 2 + \sum_{i=2}^{k} s_{i-2}$$

We now prove lemma by induction on $s_k \geq F_{k-2}$.

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i$$

$$= 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$
We have just proven that if $x$ is any node in a Fibonacci heap and $k = \text{degree}[x]$, then $\text{size}(x) \geq F_{k+2} \geq \phi^k$.

The intuition behind the proof is that the tree rooted at $x$ must contain the Fibonacci tree $T_i$ defined a few slides back.

An immediate corollary is that

$$\text{size}(x) \geq F_{\text{degree}(x)+2} \geq \phi^{\text{degree}(x)}$$

so no node in an $n$-node Fibonacci heap can have

$$\text{degree} \geq \log_\phi n$$

so

$$D(n) = O(\log n).$$