Fibonacci Heaps

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst
Fibonacci Heaps

- Collection of *meldable* heaps
  - *meld* operation combines two heaps
  - each heap is identified by one of its members (its *id*)
  - initially, all items form singleton heap
  - good *amortized running time*

- Heap operations
  - *findmin*(h): return an item of minimum key in (heap with id) h
  - *insert*(i, x, h): insert item i into heap h with key x
    - i must be a singleton heap
  - *delete*(i, h): delete item i from h and return resulting heap's id
  - *deletemin*(h): delete a min key item from h; return it and new id
  - *meld*(h₁, h₂): return id of heap formed by combining h₁ and h₂; operation destroys h₁ and h₂
  - *decreasekey*(Δ, i, h): decrease key of i in h by Δ; return new id
Structure of Fibonacci Heaps

- Each F-heap is represented by a collection of heap-ordered trees
  - each node has its item’s key, an integer rank and a mark bit
    - rank(i) equals the number of children of i
  - each node has pointers to its parent, its left and right siblings and one of its children
  - the tree roots are linked together on a circular list
  - heap is identified by a root node of minimum key

![Diagram of Fibonacci Heaps]
Implementing F-Heap Operations

- For *meld*, combine root lists; implement *insert* as *meld*
  - new heap identified by item of minimum key; takes $O(1)$ time
- For *delete*(i,h)
  - perform a *decreasekey* at i, to make i the item with smallest key
  - perform a *deletemin* to remove i from the heap
  - restore original key value of i
  - time is just sum of times for *deletemin* and *decreasekey*
- For *deletemin*
  - remove min key item from root list
  - combine its list of children with root list and clear mark bits of children
  - find new min key node
    - while doing this, combine trees with root nodes of equal rank until no two
      nodes in root list have same rank
**DeleteMin** combines trees with equal rank roots

- Insert tree roots into an array, at position determined by their rank
- Make one root a child of the other whenever there is a “collision”
  - Note that root of new tree increases its rank

**For decreasekey(Δ,i,h)**

- Subtract Δ from key(i) then cut edge joining i to its parent p
- Make detached subtree a separate tree in heap and clear its mark bit
- If key(i) < key(h), i becomes the min node of heap
- If p is not a tree root, and i is second child cut from p, since p became child of some other node, cut edge from p to its parent
  - Apply this rule recursively to parent of p, then its parent,...
  - Use mark bit to identify nodes that have lost a child
- Increases number of trees, decreases number of marked nodes
Amortized Analysis

- Objective is to bound total time for sequence of ops
  - some individual ops may take more time than others
  - expensive ops must be balanced by (earlier) inexpensive ops
- To facilitate analysis, imagine we’re given credits for each operation
  - one credit pays for one unit of computation
  - credits not used to pay for a current op can be saved for later
  - the credit allocation for each operation is its effective cost
- Central question: “How many new credits needed for each op to ensure there are always enough on hand?”
- Following credit invariant is key to analysis
  
  $\text{at all times, the number of credits on hand is at least the number of trees in all heaps, plus twice number of marked non-root nodes}$
- Determine number of new credits needed per op to pay for the op and maintain validity of invariant
  - \textit{findmin}, \textit{insert} and \textit{meld} each take constant time and don’t affect invariant, so just one new credit for each op
  - time for \textit{deletemin} bounded by number of steps in second part
    - so, need one new credit per step plus one for every net new tree
    - details to come
  - time for \textit{decreasekey} bounded by number of cuts performed and each cascading cut involves a marked node

- Detailed analysis of \textit{decreasekey}
  - let \( k \) = number of cuts made by \textit{decreasekey}
  - running time for \textit{decreasekey} is \( O(k) \)
  - number of trees increases by \( k \)
  - number of marked non-root nodes \textit{decreases} by \( k-2 \)
  - so, the number of new credits needed is \( k+k-2(k-2)=4 \)
  - so, cost of the \textit{decreasekey} is \( O(1) \)
Detailed Analysis of Deletemin

- Detailed analysis of *deletemin*
  - let $k$ = rank of node removed in *deletemin*
    - number of trees increases by $k$ during first part of the op
    - number of marked non-root nodes does not increase
  - in second part, trees with roots of equal rank are combined
  - let $p$ = # of times a tree root collides with another,
    - let $q$ = # of times a tree root is inserted with no collision
    - running time for *deletemin* is $O(p+q)$
    - number of trees decreases by $p$ during the second part
  - so, number of new credits needed to pay for the op and maintain credit invariant is $(p+q)+(k-p)=k+q$
  - note that both $k$ and $q$ are bounded by the max rank, which we will show is $O(\log n)$
- So, $O(s+t\log n)$ time for *s findmin, meld* or *decreasekey* ops plus $t$ *delete* or *deletemin* ops
Bound on Ranks

- **Lemma 1.** Let $x$ be any node and let $y_1, \ldots, y_r$ be children of $x$, in order of time in which they were linked to $x$ (earliest to latest); then, $\text{rank}(y_i) \geq i-2$ for all $i$

  Proof. Just before $y_i$ was linked to $x$, $x$ had at least $i-1$ children. So at that time, $\text{rank}(y_i)$ and $\text{rank}(x)$ were equal and $\geq i-1$. Since $y_i$ is still a child of $x$, its rank has been decremented at most once since it was linked, implying $\text{rank}(y_i) \geq i-2$.

- **Corollary 1.** A node of rank $k$ has $\geq F_{k+2} \geq \phi^k$ descendants (including itself), where $F_k$ is the $k$-th Fibonacci number, defined by $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$ and $\phi = (1 + 5^{1/2})/2$.

  Proof. Let $S_k$ be min possible number of descendants of a node of rank $k$; clearly, $S_0 = 1$, $S_1 = 2$ and by Lemma 1, $S_k \geq 2 + \sum_{i=1}^{k-2} S_i$ for $k \geq 2$; the Fibonacci numbers satisfy $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$, from which $S_k \geq F_{k+2}$ follows by induction on $k$.

Corollary implies that $\text{rank}(x)$ is $O(\log n)$.
Exercises

1. Assume that items a through m with keys 3, 5, 2, 7, 4, 10, 8, 6, 3, 6, 1, 2, 9 are inserted in alphabetical order into a
Fibonacci heap. Show the heap following the insertions. Then do a deleteMin and
show the resulting heap state.

Data structure after insertions (single
node trees linked in circular list)

Data structure after deleteMin (including
linking process).

2. Let $P_d(n)$ denote the running time of Prim’s
algorithm using $d$-heaps, where the value of $d$ is
chosen dynamically to give the best overall
running time. Let $P_f(n)$ denote the running time of
Prim’s algorithm, using Fibonacci heaps. Which of
the following statements is true? Justify your
answers.

- $P_d$ is $O(P_f)$ when $m = 3n$.
  
  This is true, since $P_d = O(m \log n)$ and $P_f = O(n \log n)$.

- $P_d$ is $O(P_f)$ when $m = n^2/4$.
  
  This is true, since $P_d = O(m \log n)$ and $P_f = O(n \log n)$.

- $P_d$ is $O(n^2)$ when $m = n$.
  
  This is false, since $P_d = O(m \log n)$ and $P_f = O(n \log n)$.

- $P_d$ is $O(n^{3/2})$ when $m = 3n$.
  
  This is true, since $P_d = O(m \log n)$ and $P_f = O(n \log n)$.
3. In the Fibonacci heaps data structure, a cut between a vertex \( u \) and its parent \( v \) causes a cascading cut at \( v \) if \( v \) has already lost a child since it last became a child of some other vertex. Suppose we change this, so that a cascading cut is done at \( v \) only if \( v \) has already lost two children. How does this change alter the lemma shown below (this lemma is from the analysis of the running time of Fibonacci heaps)? Explain your answer.

**Lemma.** Let \( x \) be any node in an F-heap. Let \( y_1, \ldots, y_r \) be the children of \( x \), in order of time in which they were linked to \( x \) (earliest to latest). Then, \( \text{rank}(y_i) \geq i-2 \) for all \( i \).

The inequality in the lemma becomes \( \text{rank}(y_i) \geq i-3 \). Since \( y_i \) had the same rank as \( x \) when it became a child of \( x \) and \( x \) must have had at least \( i-1 \) children at that time, \( y_i \) must have had rank of at least \( i-1 \) when it became a child of \( x \). Since it still is a child of \( x \), it can have lost at most two children since that time, so its rank must be at least \( i-3 \).

Let \( S_i \) be the smallest possible number of descendants that a node of rank \( k \) has, in our modified version of Fibonacci heaps. Give a recursive lower bound on \( S_i \). That is, give an inequality of the form \( S_i \geq f(S_0, S_1, \ldots, S_{i-1}) \) where \( f \) is some function of the \( S_i \)'s for \( i \leq k \).

Clearly \( S_0 = 1, S_1 = 2 \) and \( S_2 = 3 \). For \( k > 2 \), we can use the modified lemma to conclude that \( S_k \geq 3 + S_0 + S_1 + \ldots + S_{k-1} \). Note that the difference between the bounds for \( S_k \) and for \( S_{k-1} \) is \( S_{k-2} \).

Use this to give a lower bound on the smallest number of descendants that a node with rank 7 can have.

*From the above, we have* \( S_3 = 3 + S_0 = 4, S_4 = 4 + S_1 = 6, S_5 = 6 + S_2 = 9, S_6 = 9 + S_3 = 13, S_7 = 13 + S_4 = 19 \).