Algorithm Theory
16 Fibonacci Heaps

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08
Priority Queues: Operations

- Priority queue Q
  - Data structure for maintaining a set of elements, each having an associated priority

- Operations:
  - Q.initialize():
    - creates empty queue Q
  - Q.isEmpty():
    - returns true iff Q is empty
  - Q.insert(e):
    - inserts element e into Q and returns a pointer to the node containing e
  - Q.deletemin():
    - returns the element of Q with minimum key and deletes it
  - Q.min():
    - returns the element of Q with minimum key
  - Q.decreasekey(v,k):
    - decreases the value of v’s key to the new value
Priority Queues: Operations

- Additional Operations:
  - Q.delete(v):
    - deletes node v and its elements from Q
  - Q.meld(Q´):
    - unites Q and Q´ (concatenable queue)
  - Q.search(k):
    - searches for the element with key k in Q (searchable queue)

- possibly many more,
  - e.g. predecessor, successor, max, deletemax
## Priority Queues: Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>List</th>
<th>Heap</th>
<th>Binomial Queue</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insert</strong></td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>delete-min</strong></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)*</td>
</tr>
<tr>
<td><strong>meld (m≤n)</strong></td>
<td>O(1)</td>
<td>O(n) or O(m log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>decrease-key</strong></td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)*</td>
</tr>
</tbody>
</table>

* = amortized cost

\[ Q.delete(e) = Q.decreasekey(e, \infty) + Q.deletemin() \]
Fibonacci Heaps

- „Lazy meld“ version of binomial queues:
  - The melding of trees having the same order is delayed until the next `deletemin` operation

- Definition
  - A Fibonacci heap $Q$ is a collection of heap-ordered trees.

- Variables
  - $Q.min$
    - root of the tree containing the minimum key
  - $Q.rootlist$
    - circular, doubly linked, unordered list containing the roots of all trees
  - $Q.size$
    - number of nodes currently in $Q$
Structure of Fibonacci Heaps

- Let B be a heap-ordered tree in Q.rootlist
  - B.childlist: circular, doubly linked and unordered list of the children of B

- Advantages of circular, doubly linked lists
  - Deleting an element takes constant time
  - Concatenating two lists takes constant time

Structure of a node

<table>
<thead>
<tr>
<th></th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>entry</td>
</tr>
<tr>
<td></td>
<td>degree</td>
</tr>
<tr>
<td></td>
<td>child</td>
</tr>
<tr>
<td></td>
<td>mark</td>
</tr>
</tbody>
</table>

left

right
Trees in Fibonacci Heaps

Structure of a node

<table>
<thead>
<tr>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
</tr>
<tr>
<td>degree</td>
</tr>
<tr>
<td>child</td>
</tr>
<tr>
<td>mark</td>
</tr>
</tbody>
</table>

left

right

min

7

2

13 15

19 11

13 45 8

24

83 79

36

21

15
Fibonacci-Trees: Meld (Link)

Meld operation of trees $B$, $B'$ of same degree $k$

**Link-Operation:**

1. $\text{rank}(B) = \text{rank}(B') + 1$
2. $B'.\text{mark} = \text{false}$

Can be computed in constant time: $O(1)$

Resulting tree has degree $k+1$

Difference compared to Binomial Queues:

Trees do not need to have the binomial form.
Operations on Fibonacci Heaps

Q.initialize(): Q.rootlist= Q.min = null

Q.meld(F-Heap F):
/* concatenate root lists */
1 Q.min.right.left = F.min.left
2 F.min.left.right = Q.min.right
3 Q.min.right = F.min
4 F.min.left = Q.min
5 Q.min = min { F.min, Q.min }
/* no cleaning up - delayed until next deletemin */

Q.insert(e):
1. generate a new node with element e → Q′
2. Q.meld(Q′)

Q.min():
return Q.min.key
Fibonacci-Heaps: Deletemin

Q.deletemin()
/*Delete the node with minimum key from Q and return its
element.*/
1  m = Q.min()
2  if Q.size() > 0
3     then remove Q.min() from Q.rootlist
4      add Q.min.childlist to Q.rootlist
5  Q.consolidate()
/* Repeatedly meld nodes in the root list having the same
degree. Then determine the element with minimum key. */
6 return m
Fibonacci Heaps Maximum Degree of a Node

› **Definition**
  • \( \text{rank}(v) \) = degree of a node \( v \) in \( Q \)
  • \( \text{rank}(Q) \) = maximum degree of any node in \( Q \)

› **Assumption:**
  • \( \text{rank}(Q) \leq 2 \log n \)
    - where \( n = Q.\text{size} \)
Cost of \textit{deletemin}

- Deleting has constant time cost $O(1)$
- Time cost results from the consolidate operation
  - i.e. the length of the root list and the number of necessary link-operations
- How to efficiently perform the consolidate operation?
  - Observation:
    - Every root must be considered at least once
    - For every possible rank there is at most one node
**delete**\textit{emin}: Example

![Binary Tree Example](image-url)
deletemin: Example
deletemin: Example

Rank array: 0 1 2 3 4 5

7 13 45 8 19 11
15 36 21 24 83 52 79
consolidate: Example

Rank array:

0 1 2 3 4 5

7
13 45 8
15 36 21
19
24
83 52
79
11
**consolidate**: Example

Rank array:

```
0 1 2 3 4 5
```

```
  7
  13
   15
   19
    24
     83
     52
      79
  45
   8
   36
   21
  11
```
**consolidate**: Example

Rank array:

```
0 1 2 3 4 5
```

Diagram:

```
7 → 45 → 8
   → 36
   → 21
   → 13
      → 15
         → 19
            → 24
               → 83
                  → 52
                     → 79
```

consolidate: Example

Rank array:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
7 & 8 & 36 & 21 & 13 & 11 \\
\end{array}
\]

\[
\begin{array}{cccccc}
8 & 15 & 19 & 24 & 83 & 45 \\
\end{array}
\]

\[
\begin{array}{cccccc}
19 & 52 & 79 \\
\end{array}
\]
consolidate: Example
Analysis of *consolidate*

```java
rankArray = new FibNode[maxRank(n)+1];  // Create array

for „each FibNode N in rootlist“ {
    while (rankArray[N.rank] != null) {  // position occupied
        N = link(N, rankArray[N.rank]);  // link trees
        rankArray[N.rank-1] = null;      // delete old position
    }
    rankArray[N.rank] = N;            // insert into the array
}
```
Analysis

for „each FibNode N in rootlist“ {
    while (rankArray[N.rank] != null) {
        N = link(N, rankArray[N.rank]);
        rankArray[N.rank-1] = null;
    }
    rankArray[N.rank] = N;
}

Let $k = \#\text{root node}$ before the consolidation. These $k$ nodes can be classified as

$W = \{\text{Nodes which are in the root list in the end}\}$

$L = \{\text{Nodes which are appended to another node}\}$

We have: $\text{cost(for-loop)} = \text{cost}(W) + \text{cost}(L)$

$= |\text{rankArray}| + \#\text{links}$
Cost of `deletemin`

- **Worst case time:** \( O(\text{maxRank}(n)) + O(\# \text{links}) \)
  - where \( \text{maxRank}(n) \) is the largest possible array entry, i.e. the largest possible root degree

- **Worst case:** \( \# \text{links} = n-2 \)
  - happens usually once at the beginning

- **Amortized Analysis**
  - sums up all the running times
  - and averages (worst case!) over all single Operations
Fibonacci-Heaps: `decreasekey`

Q.`decreasekey`(FibNode N, int k):

- Decrease the key `N` to the value `k`
- If the heap condition does not hold (`k < N.parent.key`):
  - Disconnect `N` from its father (using `cut`) and append it (to the right of the minimal node) into the rootlist and unmark it
  - If the father is marked (`N.parent.mark == true`), disconnect it from his father and unmark it; if his father is also marked, disconnect it, etc. ("cascading cuts")
  - Mark the node whose son is disconnected at last (if it is node a root node).
  - Update the minimum pointer (if `k < min.key`).
Example for \textit{decreasekey}

Decrease key 64 to 14
Example for \textit{decreasekey}
Example for *decreasekey*
Example for \textit{decreasekey}
Example for *decreasekey*
Cost of \textit{decreasekey}

- Placement of key and comparison with father: $O(1)$
- Disconnect from father node and insertion into root list: $O(1)$
- Cascading cuts: $\#\text{cuts}$
- Mark of the last node: $O(1)$

$\Rightarrow$ Costs depends on the number of „cascading cuts“.

Worst case: $\#\text{cuts} = n-1$

Amortized Analysis gives smaller value
Amortized Analysis – Potential Method

- Assign every state \( i \) of the data structure a value \( \Phi_i \) (Potential)
  - \( \Phi_i \) = potential after the \( i \)-th operation
- The amortized costs \( a_i \) of the \( i \)-th operation is defined as
  - \( a_i = c_i + (\Phi_i - \Phi_{i-1}) \),
  - the actual cost plus the change of the potential by the \( i \)-th operation
- Define \( \Phi_i = w_i + 2m_i \)
  - where \( w_i \) = number of root nodes
    and \( m_i \) = number of marked nodes (no roots)
- Example: insert
  - real costs: \( c_i = O(1) \)
    Potential increases by 1, i.e. \( \Phi_i - \Phi_{i-1} = 1 \)
    \( a_i = c_i + 1 \)
Potential: Example

\[ w_i = \]
\[ m_i = \]
\[ \Phi_i = \]
Potential Method: Cost of \textit{insert}

- Real coast: \( c_i = O(1) \)
- Change of potential:
  \( \Phi_i - \Phi_{i-1} = 1 \)
- Amortized costs: \( a_i = c_i + 1 \)
Potential Method: Cost of \textit{decreasekey}

- Real cost: \( c_i = O(1) + \text{#cascading cuts} \)

- Change of potential:
  \[
  w_i \leq w_{i-1} + 1 + \text{#cascading cuts}
  \]
  \[
  m_i \leq m_{i-1} + 1 - \text{#cascading cuts}
  \]
  \[
  \Phi_i - \Phi_{i-1} = w_i + 2 \cdot m_i - (w_{i-1} + 2 \cdot m_{i-1})
  = w_i - w_{i-1} + 2 \cdot (m_i - m_{i-1})
  \leq 1 + \text{#cascading cuts} + 2 \cdot (1 - \text{#cascading cuts})
  = 3 - \text{#cascading cuts}
  \]

- Amortized costs: \( a_i = c_i + (\Phi_i - \Phi_{i-1}) \)
  \[
  \leq O(1) + \text{#cascading cuts} + 3 - \text{#cascading cuts} = O(1)
  \]
Potential Cost

Cost of *deletemin*

- Real cost: \( c_i = O(\text{maxRank}(n)) + \#\text{links} \)

- Change of potential:
  \[
  w_i = w_{i-1} - 1 + \text{rank}(\text{min}) - \#\text{links} \leq w_{i-1} + \text{maxRank}(n) - \#\text{links}
  \]
  \[
  m_i \leq m_{i-1}
  \]
  \[
  \Phi_i - \Phi_{i-1} = w_i + 2m_i - (w_{i-1} + 2m_{i-1})
  \]
  \[
  = w_i - w_{i-1} + 2(m_i - m_{i-1})
  \]
  \[
  \leq \text{maxRank}(n) - \#\text{links}
  \]

- Amortized costs:
  \[
  a_i = c_i + (\Phi_i - \Phi_{i-1})
  \]
  \[
  \leq O(\text{maxRank}(n)) + \#\text{links} + \text{maxRank}(n) - \#\text{links}
  \]
  \[
  = O(\text{maxRank}(n))
  \]
$maxRank$ is the maximal possible number of sons which a node can have in a Fibonacci-Heap with $n$ elements.

- We want to show that this number is at most $O(\log n)$.
- Every node has a minimum number of successors which is exponential in the number of sons.
Lemma 1:
Let $N$ be a node in a Fibonacci-Heap and let $k = N.\text{rank}$. Consider the sons $C_1, ..., C_k$ of $N$ in the order how they have been inserted (via $\text{link}$) to $N$. We have:

1. $C_1.\text{rank} \geq 0$
2. $C_i.\text{rank} \geq i - 2$ für $i = 2, ..., k$

Proof: (1) straight forward

(2) When $C_i$ became a son of $N$ nodes $C_1, ..., C_{i-1}$ were already sons of $N$, i.e. $N.\text{rank} \geq i-1$. Since $\text{link}$ always connects nodes with the same rank we had at the insertion also $C_i.\text{rank} \geq i-1$.
Since then $C_i$ might have lost only one son (because of $\text{cascading cuts}$).
Hence we have: $C_i.\text{rank} \geq i - 2$
Lemma 2:
Let $N$ be a node in a Fibonacci Heap and let $k = N$.rank.
Let $size(N) = \#$ be the number of nodes in a sub-tree with root $N$.
Then: $size(N) \geq F_{k+2} \geq 1.618^k$

i.e. a node with $k$ children has at least $F_{k+2}$ successors
(including itself).
Proof: Let $S_k = \min \{ \text{size}(N) \mid N \text{ with } N.\text{rank} = k \}$, i.e. the smallest possible size of tree with root rank $k$.
(clearly $S_0 = 1$ and $S_1 = 2$)

Let $C_1, \ldots, C_k$ be the children $N$ in the order how they were inserted into $N$.

We have

$$\text{size}(N) \geq S_k = \sum_{i=1}^{k} S_{C_{i}.\text{rank}}$$

$$= 1 + 1 + \sum_{i=2}^{k} S_{i-2}$$

$$= 2 + \sum_{i=2}^{k} S_{i-2}$$
maxRank(n)

Remember: Fibonacci-Numbers

\[
F_0 = 0 \\
F_1 = 1 \\
F_{k+2} = F_{k+1} + F_k \quad \text{für } k \geq 0
\]

Fibonacci numbers grow exponentially where \( F_{k+2} \geq 1.618^k \)

Furthermore:

\[
F_{k+2} = 2 + \sum_{i=2}^{k} F_i
\]

(Follows by straight-forward induction over \( k \).)
Summary

\[ F_{k+2} = 2 + \sum_{i=2}^{k} F_i \quad \quad S_k = 2 + \sum_{i=2}^{k} S_{i-2} \]

- Furthermore \( S_0 = 1 = F_2 \) and \( S_1 = 2 = F_3 \)
- So, it follows: \( S_k = F_{k+2} \) \hspace{1cm} (Proof by induction)
- For a node \( N \) with rank \( k \) we observe

\[ \text{size}(N) \geq S_k = F_{k+2} \geq 1.618^k \]
Theorem: The maximal value of $\text{maxRank}(n)$ of any node in a Fibonacci-Heap with $n$ nodes is bound by $O(\log n)$.

Proof: Let $N$ be an arbitrary node of a Fibonacci-Heaps with $n$ nodes and let $k = N.\text{rank}$.

Lemma 2 implies $n \geq \text{size}(N) \geq 1.618^k$

Therefore $k \leq \log_{1.618}(n) = O(\log n)$
## Summary

<table>
<thead>
<tr>
<th></th>
<th>List</th>
<th>Heap</th>
<th>Binomial Queue</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insert</strong></td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>delete-min</strong></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)*</td>
</tr>
<tr>
<td><strong>decrease key</strong></td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)*</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)*</td>
</tr>
</tbody>
</table>

* = amortized cost

\[ Q.delete(e) = Q.decreasekey(e, \infty) + Q.deletemin() \]
Algorithm Theory
16 Fibonacci Heaps

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08