FIBONACCI HEAPS

- preliminaries
- insert
- extract the minimum
- decrease key
- bounding the rank
- meld and delete

Fibonacci heaps

Theorem. [Fredman-Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of \(m\) INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving \(n\) INSERT operations takes \(O(m + n \log n)\) time.

Priority queues performance cost summary

<table>
<thead>
<tr>
<th>operation</th>
<th>linked list</th>
<th>binary heap</th>
<th>binomial heap</th>
<th>Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>IS-EMPTY</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>INSERT</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>(O(n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>DELETE</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>MELD</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(\log n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>FIND-MIN</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

† amortized

Ahead. \(O(1)\) INSERT and DECREASE-KEY, \(O(\log n)\) EXTRACT-MIN.

Fibonacci heaps

Theorem. [Fredman–Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of \(m\) INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving \(n\) INSERT operations takes \(O(m + n \log n)\) time.

History.

- Ingenious data structure and application of amortized analysis.
- Original motivation: improve Dijkstra’s shortest path algorithm from \(O(m \log n)\) to \(O(m + n \log n)\).
- Also improved best-known bounds for all-pairs shortest paths, assignment problem, minimum spanning trees.
**FIBONACCI HEAPS**

- structure
- insert
- extract the minimum
- decrease key
- bounding the rank
- meld and delete

**Fibonacci heaps**

**Basic idea.**
- Similar to binomial heaps, but less rigid structure.
- Binomial heap: *eagerly* consolidate trees after each INSERT; implement DECREASE-KEY by repeatedly exchanging node with its parent.

![Diagram]

- Fibonacci heap: *lazily* defer consolidation until next EXTRACT-MIN; implement DECREASE-KEY by cutting off node and splicing into root list.

**Remark.** Height of Fibonacci heap is $\Theta(n)$ in worst case, but it doesn’t use sink or swim operations.

**Fibonacci heap: structure**

- Set of heap-ordered trees.
  - each child no smaller than its parent

heap H

- Set of marked nodes.
  - used to keep trees bushy

heap H

- heap-ordered tree

![Example heap H]
Fibonacci heap: structure

Heap representation.

- Store a pointer to the minimum node.
- Maintain tree roots in a circular, doubly-linked list.

Fibonacci heap: representation

Node representation. Each node stores:

- A pointer to its parent.
- A pointer to any of its children.
- A pointer to its left and right siblings.
- Its rank = number of children.
- Whether it is marked.

Fibonacci heap: representation

Operations we can do in constant time:

- Determine rank of a node.
- Find the minimum element.
- Merge two root lists together.
- Add or remove a node from the root list.
- Remove a subtree and merge into root list.
- Link the root of a one tree to root of another tree.

Fibonacci heap: notation

<table>
<thead>
<tr>
<th>notation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>number of nodes</td>
</tr>
<tr>
<td>$\text{rank}(x)$</td>
<td>number of children of node $x$</td>
</tr>
<tr>
<td>$\text{rank}(H)$</td>
<td>max rank of any node in heap $H$</td>
</tr>
<tr>
<td>$\text{trees}(H)$</td>
<td>number of trees in heap $H$</td>
</tr>
<tr>
<td>$\text{marks}(H)$</td>
<td>number of marked nodes in heap $H$</td>
</tr>
</tbody>
</table>

$n = 14 \quad \text{rank}(H) = 3 \quad \text{trees}(H) = 5 \quad \text{marks}(H) = 3$
Fibonacci heap: potential function

Potential function.

\[ \Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H) \]

\[ \Phi(H) = 5 + 2 \cdot 3 = 11 \]

Trees(H) = 5

Marks(H) = 3

heap H

Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).

insert 21

heap H

Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).

insert 21

heap H
Fibonacci heap: insert analysis

Actual cost.  
\[ c_i = O(1). \]

Change in potential.  
\[ \Delta \Phi = \Phi(H_i) - \Phi(H_{i-1}) = +1. \]

Amortized cost.  
\[ \hat{c}_i = c_i + \Delta \Phi = O(1). \]

\[ \Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H) \]

Linking operation

Useful primitive. Combine two trees \( T_1 \) and \( T_2 \) of rank \( k \).
- Make larger root be a child of smaller root.
- Resulting tree \( T' \) has rank \( k + 1 \).

Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

link 23 to 17

link 17 to 7

link 24 to 7
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

link 41 to 18
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

Actual cost. $c_i = O(rank(H)) + O(trees(H))$.
- $O(rank(H))$ to meld min's children into root list.
- $O(rank(H)) + O(trees(H))$ to update min.
- $O(rank(H)) + O(trees(H))$ to consolidate trees.

Change in potential. $\Delta \Phi \leq rank(H') + 1 - (trees(H))$.
- No new nodes become marked.
- $trees(H') \leq rank(H') + 1$. no two trees have same rank after consolidation

Amortized cost. $O(\log n)$.
- $\hat{c}_i = c_i + \Delta \Phi = O(rank(H)) + O(rank(H'))$.
- The rank of a Fibonacci heap with $n$ elements is $O(\log n)$.
Fibonacci heap vs. binomial heaps

**Observation.** If only **INSERT** and **EXTRACT-MIN** operations, then all trees are binomial trees.

We link only trees of equal rank

![Diagram of Fibonacci heap vs. binomial heaps]

**Binomial heap property.** This implies \( \text{rank}(H) \leq \log_2 n \).

**Fibonacci heap property.** Our **DECREASE-KEY** implementation will not preserve this property, but we will implement it in such a way that \( \text{rank}(H) \leq \log_\phi n \).

Fibonacci heap: decrease key

**Intuition for decreasing the key of node** \( x \).
- If heap-order is not violated, decrease the key of \( x \).
- Otherwise, cut tree rooted at \( x \) and meld into root list.

**Decrease-key of** \( x \) **from 30 to 7**

![Decrease-key of x from 30 to 7]

**Decrease-key of** \( x \) **from 23 to 5**

![Decrease-key of x from 23 to 5]
Fibonacci heap: decrease key

Intuition for decreasing the key of node $x$.
- If heap-order is not violated, decrease the key of $x$.
- Otherwise, cut tree rooted at $x$ and meld into root list.

decrease-key of 22 to 4
decrease-key of 48 to 3
decrease-key of 31 to 2
decrease-key of 17 to 1

rank = 4, nodes = 5

Fibonacci heap: decrease key

Intuition for decreasing the key of node $x$.
- If heap-order is not violated, decrease the key of $x$.
- Otherwise, cut tree rooted at $x$ and meld into root list.
- Problem: number of nodes not exponential in rank.

Case 1. (heap order not violated)
- Decrease key of $x$.
- Change heap min pointer (if necessary).

decrease-key of $x$ from 46 to 29
Fibonacci heap: decrease key

**Case 1.** [heap order not violated]
- Decrease key of $x$.
- Change heap min pointer (if necessary).

decrease-key of $x$ from 46 to 29

decrease-key of $x$ from 29 to 15

**Case 2a.** [heap order violated]
- Decrease key of $x$.
- Cut tree rooted at $x$, meld into root list, and unmark.
- If parent $p$ of $x$ is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut $p$, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of $x$ from 29 to 15

decrease-key of $x$ from 29 to 15
Fibonacci heap: decrease key

**Case 2a.** [heap order violated]
- Decrease key of $x$.
- Cut tree rooted at $x$, meld into root list, and unmark.
- If parent $p$ of $x$ is unmarked (hasn’t yet lost a child), mark it;
  Otherwise, cut $p$, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

**decrease-key of $x$ from 29 to 15**

**Case 2b.** [heap order violated]
- Decrease key of $x$.
- Cut tree rooted at $x$, meld into root list, and unmark.
- If parent $p$ of $x$ is unmarked (hasn’t yet lost a child), mark it;
  Otherwise, cut $p$, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

**decrease-key of $x$ from 35 to 5**
**Fibonacci heap: decrease key**

**Case 2b.** [heap order violated]
- Decrease key of $x$.
- Cut tree rooted at $x$, meld into root list, and unmark.
  - If parent $p$ of $x$ is unmarked (hasn’t yet lost a child), mark it;
  - Otherwise, cut $p$, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

-decrease-key of $x$ from 35 to 5

**Fibonacci heap: decrease key**

**Case 2b.** [heap order violated]
- Decrease key of $x$.
- Cut tree rooted at $x$, meld into root list, and unmark.
  - If parent $p$ of $x$ is unmarked (hasn’t yet lost a child), mark it;
  - Otherwise, cut $p$, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of $x$ from 35 to 5
Fibonacci heap: decrease key

Case 2b. [heap order violated]
- Decrease key of $x$.
- Cut tree rooted at $x$, meld into root list, and unmark.
- If parent $p$ of $x$ is unmarked (hasn’t yet lost a child), mark it.
  Otherwise, cut $p$, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

**Analysis summary**

**Insert.** $O(1)$.
**Delete-min.** $O(\text{rank}(H))$ amortized.
**Decrease-key.** $O(1)$ amortized.

**Fibonacci lemma.** Let $H$ be a Fibonacci heap with $n$ elements.
Then, $\text{rank}(H) = O(\log n)$.
Lemma 1. Fix a point in time. Let $x$ be a node of rank $k$, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to $x$. Then:

$$\text{rank}(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$

Pf.

- When $y_i$ was linked into $x$, $x$ had at least $i - 1$ children $y_1, \ldots, y_{i-1}$.
- Since only trees of equal rank are linked, at that time $\text{rank}(y_i) = \text{rank}(x) \geq i - 1$.
- Since then, $y_i$ has lost at most one child (or $y_i$ would have been cut).
- Thus, right now $\text{rank}(y_i) \geq i - 2$.  

Def. Let $T_k$ be smallest possible tree of rank $k$ satisfying property.

 DEF. Let $T_k$ be smallest possible tree of rank $k$ satisfying property.

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \quad T_4 \quad T_5 \]

![Tree Diagram](image)

$s_k \geq 1 + 1 + (s_0 + s_1 + \ldots + s_{k-2})$ (Lemma 1)

$\geq (1 + F_1) + F_2 + F_3 + \ldots + F_k$ (inductive hypothesis)

$= F_{k+2}$ (Fibonacci fact 1)
Bounding the rank

Fibonacci lemma. Let $H$ be a Fibonacci heap with $n$ elements. Then, \( \text{rank}(H) \leq \log_\phi n \), where $\phi$ is the golden ratio $= (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf. 
- Let $H$ be a Fibonacci heap with $n$ elements and rank $k$. Then $n \geq F_{k+2} \geq \phi^k$.

Lemma 2 Fibonacci Fact 2

- Taking logs, we obtain $\text{rank}(H) = k \leq \log_\phi n$. ■

Fibonacci fact 1

Def. The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

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

Fibonacci fact 1. For all integers $k \geq 0$, $F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf. [by induction on $k$]
- Base case: $F_2 = 1 + F_0 = 2$.
- Inductive hypothesis: assume $F_{k+1} = 1 + F_0 + F_1 + \ldots + F_{k-1}$.

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

(definition)
\[
= F_k + (1 + F_0 + F_1 + \ldots + F_{k-1}) 
\]

(inductive hypothesis)
\[
= 1 + F_0 + F_1 + \ldots + F_{k-1} + F_k 
\]

(algebra)

Fibonacci fact 2

Def. The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

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

Fibonacci fact 2. $F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf. [by induction on $k$]
- Base cases: $F_3 = 1 + 1 = 2$, $F_4 = 2 + 2 \geq \phi$.
- Inductive hypotheses: assume $F_k \geq \phi^k$ and $F_{k+1} \geq \phi^{k+1}$.

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

(definition)
\[
\geq \phi^k + \phi^{k+1} 
\]

(inductive hypothesis)
\[
= \phi^k + \phi^{k+1} (1 + \phi) 
\]

(algebra)
\[
= \phi^k \phi + \phi^k 
\]

(\(\phi^0 = \phi + 1\))
\[
= \phi^k 
\]

(algebra)

Fibonacci numbers and nature

Fibonacci numbers arise both in nature and algorithms.
**Fibonacci Heaps**

- preliminaries
- insert
- extract the minimum
- decrease key
- bounding the rank
- meld and delete

---

**Fibonacci heap: meld**

**Meld.** Combine two Fibonacci heaps (destroying old heaps).

**Recall.** Root lists are circular, doubly-linked lists.

---

**Fibonacci heap: meld analysis**

**Actual cost.** $c_i = O(1)$.

**Change in potential.** $\Delta \Phi = 0$.

**Amortized cost.** $\check{c}_i = c_i + \Delta \Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$
Fibonacci heap: delete

Delete. Given a handle to an element \( x \), delete it from heap \( H \).
- Decrease-Key\((H,x,\infty)\).
- Extract-Min\((H)\).

Amortized cost. \( \delta_i = O(rank(H)) \).
- \( O(1) \) amortized for Decrease-Key.
- \( O(rank(H)) \) amortized for Extract-Min.

\[
\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)
\]

### Priority queues performance cost summary

<table>
<thead>
<tr>
<th>operation</th>
<th>linked list</th>
<th>binary heap</th>
<th>binomial heap</th>
<th>Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>IS-EMPTY</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>INSERT</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>( O(n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>DELETE</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>MELD</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>FIND-MIN</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

↑ amortized

Accomplished. \( O(1) \) INSERT and Decrease-Key, \( O(\log n) \) Extract-Min.

### Heaps of heaps

- b-heaps.
- Fat heaps.
- 2–3 heaps.
- Leaf heaps.
- Thin heaps.
- Skew heaps.
- Splay heaps.
- Weak heaps.
- Leftist heaps.
- Quake heaps.
- Pairing heaps.
- Violation heaps.
- Run-relaxed heaps.
- Rank-pairing heaps.
- Skew-pairing heaps.
- Rank-relaxed heaps.
- Lazy Fibonacci heaps.
Brodal queues

Q. Can we achieve same running time as for Fibonacci heap but with worst-case bounds per operation (instead of amortized)?


Practice. Ever implemented? Constants are high (and requires RAM model).

Fibonacci heaps: practice

Q. Are Fibonacci heaps useful in practice?
A. They are part of LEDA and Boost C++ libraries.
(but other heaps seem to perform better in practice)

Strict Fibonacci heaps

Q. Can we achieve same running time as for Fibonacci heap but with worst-case bounds per operation (instead of amortized) in pointer model?


Pairing heaps


Theory. Same amortized running times as Fibonacci heaps for all operations except DECREASE-KEY.

- $O(\log n)$ amortized. [Fredman et al. 1986]
- $\Omega(\log \log n)$ lower bound on amortized cost. [Fredman 1999]
- $2\sqrt{\Theta(\log \log n)}$ amortized. [Pettie 2005]
Pairing heaps


Practice. As fast as (or faster than) the binary heap on some problems. Included in GNU C++ library and LEDA.

Priority queues with integer priorities

Assumption. Keys are integers between 0 and C.

Theorem. [Thorup 2004] There exists a priority queue that supports INSERT, FIND-MIN, and DECREASE-KEY in constant time and EXTRACT-MIN and DELETE-KEY in either $O(\log \log n)$ or $O(\log \log C)$ time.

Priority queues performance cost summary

<table>
<thead>
<tr>
<th>operation</th>
<th>linked list</th>
<th>binary heap</th>
<th>binomial heap</th>
<th>pairing heap</th>
<th>Fibonacci heap</th>
<th>Brodal queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IS-EMPTY</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$2\sqrt{\log \log n}$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MELD</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>FIND-MIN</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

† amortized

Priority queues with integer priorities

Assumption. Keys are integers between 0 and C.

Theorem. [Thorup 2004] There exists a priority queue that supports INSERT, FIND-MIN, and DECREASE-KEY in constant time and EXTRACT-MIN and DELETE-KEY in either $O(\log \log n)$ or $O(\log \log C)$ time.

Corollary 1. Can implement Dijkstra’s algorithm in either $O(m \log \log n)$ or $O(m \log \log C)$ time.

Corollary 2. Can sort n integers in $O(n \log \log n)$ time.

Computational model. Word RAM.
Soft heaps

**Goal.** Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

---

**The Soft Heap: An Approximate Priority Queue with Optimal Error Rate**

BERNARD CHAZELLE

Princeton University, Princeton, New Jersey, and NEC Research Institute

Abstract. A simple variant of a priority queue, called a soft heap, is introduced. The data structure supports the usual operations: insert, delete, meld, and findmin. Its novelty is to beat the logarithmic bound on the complexity of a heap in a comparison-based model. To break this information-theoretic bound, the entropy of the data structure is reduced by artificially raising the values of certain keys. Three and scaled versions of the operations, a soft heap with error rate 4 − 1/2 δ 12 constraints. The size of the soft heap is constant, except for insert, which takes (log 1/δ) time. The soft heap is optimal for any value of δ, in a comparison-based model. The data structure is purely pointer-based. No secret are used and no insecure assumptions are made on the keys. The main idea behind the soft heap is to move values across the data structure, not individually, as in customary, but in groups, as in a data-structures equivalent of “car pooling.” Keys must be raised as a result, in order to preserve the heap ordering of the data structure. The soft heap can be used to compute exact or approximate median and percentiles optimally. It is also useful for approximate sorting and for computing minimum spanning trees of general graphs.

---

**Q.** Brilliant. But how could it possibly be useful?

**Ex.** Linear-time deterministic selection. To find \( k \)th smallest element:

- Insert the \( n \) elements into soft heap.
- Extract the minimum element \( n / 2 \) times.
- The largest element deleted \( \geq 4n / 10 \) elements and \( \leq 6n / 10 \) elements.
- Can remove \( \geq 4n / 10 \) of elements and recur.
- \( T(n) \leq T(3n/5) + O(n) \implies T(n) = O(n). \)
Soft heaps

Theorem. [Chazelle 2000] There exists an $O(m \alpha(m, n))$ time deterministic algorithm to compute an MST in a graph with $n$ nodes and $m$ edges.

Algorithm. Borůvka + nongreedy + divide-and-conquer + soft heap + ...

A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity

BERNARD CHAZELLE

Princeton University, Princeton, New Jersey, and NEC Research Institute

Abstract. A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m \alpha(m, n))$, where $\alpha$ is the classical functional inverse of Ackermann's function and $m$ (respectively, $n$) is the number of vertices (respectively, edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.