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.

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.

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>$O(\log n)$</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>$O(\log n)$</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

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.
**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.

- 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 $O(n)$ 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.

- Set of marked nodes.
  - Used to keep trees bushy (stay tuned).

---

**Fibonacci heap: structure**

- Set of heap-ordered trees.

- Set of marked nodes.
### 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: 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>

<table>
<thead>
<tr>
<th>( n = 14 )</th>
<th>( \text{rank}(H) = 3 )</th>
<th>( \text{trees}(H) = 5 )</th>
<th>( \text{marks}(H) = 3 )</th>
</tr>
</thead>
</table>

- \( \text{rank} = 1 \)
- \( \text{children are in a circular doubly-linked list} \)
Fibonacci heap: potential function

Potential function.

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

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

\[ \text{trees}(H) = 5 \quad \text{marks}(H) = 3 \]

Fibonacci heap: insert

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

Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).
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. \( \bar{c}_i = c_i + \Delta \Phi = O(1) \).

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

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.

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

link 23 to 17

Fibonacci heap: extract the minimum

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.

rank

- link 41 to 18

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.

![Diagram of Fibonacci heap](image-url)

**Fibonacci heap: extract the minimum analysis**

**Actual cost.** \( c_i = O(\text{rank}(H)) + O(\text{trees}(H)) \).
- \( O(\text{rank}(H)) \) to meld min’s children into root list.
- \( O(\text{rank}(H)) + O(\text{trees}(H)) \) to update min.
- \( O(\text{rank}(H)) + O(\text{trees}(H)) \) to consolidate trees.

**Change in potential.** \( \Delta \Phi \leq \text{rank}(H') + 1 - \text{trees}(H) \).
- No new nodes become marked.
- \( \text{trees}(H') \leq \text{rank}(H') + 1 \).

**Amortized cost.** \( O(\log n) \).
- \( \hat{c}_i = c_i + \Delta \Phi = O(\text{rank}(H)) + O(\text{rank}(H')) \).
- 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 binomial trees]

**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 heaps**

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

**SECTION 19.3**

**FIBONACCI HEAPS**

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

---

**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**

![Diagram of decrease-key operation from 30 to 7]

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

![Diagram of decrease-key operation 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

**Problem**
- Number of nodes not exponential in rank.

**Solution**
- As soon as a node has its second child cut, cut it off also and meld into root list (and unmark it).

**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).

**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).
**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).

**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).

---

**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).

**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).
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

\[
\begin{array}{c|c|c|c|c}
\text{x} & \text{min} & \text{p} & \text{26} \\
\hline
15 & 35 & 7 & 41 \\
24 & 17 & 52 & 39 \\
72 & 88 & 30 & 23 \\
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c}
\text{x} & \text{min} & \text{p} & \text{26} \\
\hline
15 & 35 & 7 & 41 \\
24 & 17 & 52 & 39 \\
72 & 88 & 30 & 23 \\
\end{array}
\]
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).

**Fibonacci heap: decrease key analysis**

**Actual cost.** $c_i = O(c)$, where $c$ is the number of cuts.
- $O(1)$ time for changing the key.
- $O(1)$ time for each of $c$ cuts, plus melding into root list.

**Change in potential.** $\Delta \Phi = O(1) - c$.
- $\text{trees}(H') = \text{trees}(H) + c$.
- $\text{marks}(H') \leq \text{marks}(H) - c + 2$.
- $\Delta \Phi \leq c + 2 \cdot (-c + 2) = 4 - c$.

**Amortized cost.** $c_i = c_i + \Delta \Phi = O(1)$.

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

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, $\text{rank}(H) = O(\log n)$.

Number of nodes is exponential in rank

---

**SECTION 19.4 FIBONACCI HEAPS**

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

**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.

**Lemma 2.** Let $s_k$ be minimum number of elements in any Fibonacci heap of rank $k$. Then $s_k \geq F_{k+2}$, where $F_k$ is the $k^{th}$ Fibonacci number.

**Pf.** [by strong induction on $k$]
- Base cases: $s_0 = 1$ and $s_1 = 2$.
- Inductive hypothesis: assume $s_i \geq F_{i+2}$ for $i = 0, \ldots, k - 1$.
- As in Lemma 1, let let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to $x$.

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

$$\geq (1 + F_1) + F_2 + F_3 + \ldots + F_k \quad \text{(inductive hypothesis)}$$

$$= F_{k+2}. \quad \text{(Fibonacci fact 1)}$$
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 $H$ is a Fibonacci heap with $n$ elements and rank $k$.
  - Then $n \geq F_{k+2} \geq \phi^k$.
- 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 $k \geq 0$, $F_{k+2} = 1 + F_0 + F_1 + \ldots + F_k$.

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

$$F_{k+2} = F_k + F_{k+1} \quad \text{(definition)}$$
$$= F_k + (1 + F_0 + F_1 + \ldots + F_{k-1}) \quad \text{(inductive hypothesis)}$$
$$= 1 + F_0 + F_1 + \ldots + F_{k-1} + F_k. \quad \text{(algebra)}$$

---

**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: $F_2 = 1 \geq 1$, $F_3 = 2 \geq \phi$.
- Inductive hypotheses: assume $F_k \geq \phi^k$ and $F_{k+1} \geq \phi^{k+1}$

$$F_{k+2} = F_k + F_{k+1} \quad \text{(definition)}$$
$$\geq \phi^k + \phi^{k+1} \quad \text{(inductive hypothesis)}$$
$$= \phi^{k-1} + \phi^{k-2} \quad \text{(inductive hypothesis)}$$
$$= \phi^{k-2} (1 + \phi) \quad \text{(algebra)}$$
$$= \phi^{k-2} (\phi^2) \quad \text{(inductive hypothesis)}$$
$$= \phi^k. \quad \text{(algebra)}$$

---

**Fibonacci numbers and nature**

Fibonacci numbers arise both in nature and algorithms.

![Pinecone](pinecone.png)

![Cauliflower](cauliflower.png)
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. \( \hat{c}_i = c_i + \Delta \Phi = O(1) \).

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

Delete. Given a handle to an element \( x \), delete it from heap \( H \).
- \( \text{DECREASE-KEY}(H, x, -\infty) \).
- \( \text{EXTRACT-MIN}(H) \).

Amortized cost. \( \hat{c} = O(\text{rank}(H)) \).
- \( O(1) \) amortized for \( \text{DECREASE-KEY} \).
- \( O(\text{rank}(H)) \) amortized for \( \text{EXTRACT-MIN} \).

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

---

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 \( \text{DECREASE-KEY} \), \( O(\log n) \) 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)?


Worst-Case Efficient Priority Queues

Gerth Stølting Brodal

Abstract

A strict Fibonacci heap implementation with time bounds matching those of Fibonacci heaps in the worst case. The supported operations are insert, find-min, meld and decrease-key in worst-case $O(\log n)$ time, where $n$ is the size of the heap. The data structure uses linear space.

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]
- $2^{\sqrt{\log \log n}}$ amortized. [Pettie 2005]
Pairing heaps

**Pairing heap.** A self-adjusting heap-ordered general tree.

**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(1))</td>
<td>(O(1))</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>

\(\dagger\) 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).

**Representation.**
- Set of binomial trees (with some subtrees missing).
- Each node may store several elements.
- Each node stores a value that is an upper bound on the original keys.
- Binomial trees are heap-ordered with respect to these values.

**Theorem.** [Chazelle 2000] Starting from an empty soft heap, any sequence of \( n \) \textsc{insert}, \textsc{min}, \textsc{extract-min}, \textsc{meld}, and \textsc{delete} operations takes \( O(n) \) time and at most 10% of its elements are corrupted at any given time.

**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 \textit{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.
- \( T(n) \leq T(3n / 5) + O(n) \Rightarrow T(n) = O(n) \).
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 + ...