Lecture 8: Priority Queues and Heaps

Michael Dinitz

September 24, 2020
601.433/633 Introduction to Algorithms
Priority Queues / Heaps: Like a queue/stack, but instead of FIFO/LIFO, by priority

- Insert($H, x$): insert element $x$ into heap $H$.
- Extract-Min($H$): remove and return an element with smallest key
- Decrease-Key($H, x, k$): decrease the key of $x$ to $k$.
- Meld($H_1, H_2$): replace heaps $H_1$ and $H_2$ with their union

Extra Operations:
- Find-Min($H$): return the element with smallest key
- Delete($H, x$): delete element $x$ from heap $H$

Min-Heap, but can also do Max-Heap.
Introduction

Priority Queues / Heaps: Like a queue/stack, but instead of FIFO/LIFO, by priority

- **Insert**\((H, x)\): insert element \(x\) into heap \(H\).
- **Extract-Min**\((H)\): remove and return an element with smallest key
- **Decrease-Key**\((H, x, k)\): decrease the key of \(x\) to \(k\).
- **Meld**\((H_1, H_2)\): replace heaps \(H_1\) and \(H_2\) with their union

Extra Operations:

- **Find-Min**\((H)\): return the element with smallest key
- **Delete**\((H, x)\): delete element \(x\) from heap \(H\)

Min-Heap, but can also do Max-Heap.

Note: \(x\) is a *pointer* to an element. No way to lookup, so need a pointer to an element to change it.
Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
## Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linked List</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Sorted Array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal: get as many of these to $O(1)$ as possible.

Question: Can we make Insert and Extract-Min both $O(1)$, even mortized? No!

Sorting lower bound. But maybe can make one $O(1)$, other $O(\log n)$?
### Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced Search Tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal: get as many of these to $O(1)$ as possible

Question: Can we make Insert and Extract-Min both $O(1)$, even mortized?

No! Sorting lower bound. But maybe can make one $O(1)$, other $O(\log n)$?
### Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced Search Tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

**Goal:** get as many of these to $O(1)$ as possible

**Question:** Can we make Insert and Extract-Min both $O(1)$, even amortized? 

No! Sorting lower bound. But maybe can make one $O(1)$, other $O(\log n)$?
# Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced Search Tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Goal: get as many of these to $O(1)$ as possible
**Obvious Approaches**

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced Search Tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

**Goal:** get as many of these to $O(1)$ as possible

**Question:** Can we make Insert and Extract-Min both $O(1)$, even amortized?
### Obvious Approaches

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Extract-Min</th>
<th>Decrease-Key</th>
<th>Meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Balanced Search Tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Goal: get as many of these to **O(1)** as possible

**Question:** Can we make Insert and Extract-Min both **O(1)**, even amortized?

**No!** Sorting lower bound. But maybe can make one **O(1)**, other **O(log n)**?
State of the art: *strict Fibonacci Heaps*.

- Too complicated for this class, not practical. See CLRS 19 for Fibonacci Heaps.

Today: binary heaps, then binomial heaps

- Binomial heaps not quite as complicated as Fibonacci heaps, many of same core ideas
Binary Heaps

- Complete binary tree, except possibly at bottom level.
- Heap order: key of any node no larger than key of its children.
Binary Heaps

- Complete binary tree, except possibly at bottom level.
- Heap order: key of any node no larger than key of its children.

```
Binary heap
  ordered complete binary tree
```

```
<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>12 18</td>
</tr>
<tr>
<td>12 17</td>
<td>18 21</td>
</tr>
<tr>
<td>18 19</td>
<td>11 25</td>
</tr>
</tbody>
</table>
```

Properties:
- Since (almost) complete binary tree, depth $\Theta(\log n)$
- Min must be at root

Representation:
- Pointers to root and rightmost leaf
- Every node has pointers to parent and children
Insert($H, x$)

Preserve heap *structure*: insert $x$ into next open spot (bottom right, or left of new level if bottom level full)

- Might violate heap *order*!

![Binary heap: insert](image)
**Insert**$(H, x)$

Preserve heap *structure*: insert $x$ into next open spot (bottom right, or left of new level if bottom level full)

- Might violate heap *order*!

“Swim up”: as long as $x$ smaller than its parent, swap with parent

---

**Running time:** $O(\log n)$ worst case (also amortized) via depth
Insert($H, x$)

Preserve heap *structure*: insert $x$ into next open spot (bottom right, or left of new level if bottom level full)

- Might violate heap *order*!

```
          6
         / \  
       10   8
      / \   /   \
    12  18 11  25
   / \ / \    / \ 
  21 17 19  7  
```

Running time: $O(\log n)$ worst case (also amortized) via depth

```
          6
         /   \  
        8     7
       /  \   /   \
      12  18 11  25
   / \ / \     / \    / \ 
  21 17 19  12 10  11
```

“Swim up”: as long as $x$ smaller than its parent, swap with parent

```
          6
         /   \  
        8     7
       /  \   /   \
      12  18 11  25
   / \ / \     / \    / \ 
  21 17 19  12 10  11
```

swim up
Extract-Min($H$)

Min is definitely at root. How to remove it while still have binary tree?
Extract-Min($H$)

Min is definitely at root. How to remove it while still have binary tree?

- Swap root with final heap element, remove former root.
- Sink down: swap root with smaller of its children until heap order restored

Running time: $O(\log n)$ worst case (via depth). Amortized: $O(1)$ (not obvious)
Extract-Min($H$)

Min is definitely at root. How to remove it while still have binary tree?

- Swap root with final heap element, remove former root.
- Sink down: swap root with smaller of its children until heap order restored

Running time: $O(\log n)$ worst case (via depth). Amortized: $O(1)$ (not obvious)
Decrease-Key($H, x, k$)

Decrease key of $x$ to $k$, “swim up” until heap order restored.

Running time: $O(\log n)$ (depth)
Meld($H_1, H_2$)

Assume both heaps have size $n$.

- Obvious approach: insert each element of $H_2$ into $H_1$. Time: $O(n \log n)$
Meld($H_1, H_2$)

Assume both heaps have size $n$.

- Obvious approach: insert each element of $H_2$ into $H_1$. Time: $O(n \log n)$

Better:

- Insert all elements of $H_2$ all at once (not fixing heap order)
- Instead of fixing by swimming up: iterate from bottom up and sink down to fix heap.
Meld($H_1, H_2$)

Assume both heaps have size $n$.

- Obvious approach: insert each element of $H_2$ into $H_1$. Time: $O(n \log n)$

Better:

- Insert all elements of $H_2$ all at once (not fixing heap order)
- Instead of fixing by swimming up: iterate from bottom up and sink down to fix heap.

Correctness: ends up in heap order (induction, or contradiction)
Meld($H_1, H_2$)

Assume both heaps have size $n$.

- Obvious approach: insert each element of $H_2$ into $H_1$. Time: $O(n \log n)$

Better:

- Insert all elements of $H_2$ all at once (not fixing heap order)
- Instead of fixing by swimming up: iterate from bottom up and sink down to fix heap.

**Correctness:** ends up in heap order (induction, or contradiction)

**Running Time:**

- Inserting: $O(n)$ total
**Meld**($H_1, H_2$)

Assume both heaps have size $n$.

- Obvious approach: insert each element of $H_2$ into $H_1$. Time: $O(n \log n)$

Better:

- Insert all elements of $H_2$ all at once (not fixing heap order)
- Instead of fixing by swimming up: iterate from bottom up and sink down to fix heap.

**Correctness:** ends up in heap order (induction, or contradiction)

**Running Time:**

- Inserting: $O(n)$ total
- Sinking down:
  - Nodes at height $h$ might have to sink down $h$.
  - At most $n/2^h$ nodes at height $h$
Meld($H_1, H_2$)

Assume both heaps have size $n$.

- Obvious approach: insert each element of $H_2$ into $H_1$. Time: $O(n \log n)$

Better:

- Insert all elements of $H_2$ all at once (not fixing heap order)
- Instead of fixing by swimming up: iterate from bottom up and sink down to fix heap.

Correctness: ends up in heap order (induction, or contradiction)

Running Time:

- Inserting: $O(n)$ total
- Sinking down:
  - Nodes at height $h$ might have to sink down $h$.
  - At most $n/2^h$ nodes at height $h$

$$
\sum_{h=0}^{\log n} h \left( \frac{n}{2^h} \right) = n \sum_{h=0}^{\log n} \frac{h}{2^h} \leq O(n)
$$
Amortized Extract-Min

Weights: $w(x) = \text{depth of } x$
  - Root has weight 0, its children have weight 1, etc.
Potential: $\Phi(H) = \sum_x w(x)$
Amortized Extract-Min

Weights: \( w(x) = \text{depth of } x \)
  - Root has weight 0, its children have weight 1, etc.
Potential: \( \Phi(H) = \sum_x w(x) \)

Insert: \( \Delta \Phi = O(\log n) \implies \text{amortized cost } \leq O(\log n) + O(\log n) = O(\log n) \)
Amortized Extract-Min

Weights: $w(x) = \text{depth of } x$
- Root has weight 0, its children have weight 1, etc.

Potential: $\Phi(H) = \sum_x w(x)$

Insert: $\Delta \Phi = O(\log n) \implies \text{amortized cost } \leq O(\log n) + O(\log n) = O(\log n)$

Extract-Min:
- True cost: height $h = \Theta(\log n)$ of tree, plus $O(1)$ (for initial swap).
- $\Delta \Phi$: one less node at depth $h \implies \Delta \Phi = -h$
- Amortized cost: $h + O(1) - h = O(1)$. 

Uses Inserts to "pay for" Extract-Mins.
Amortized Extract-Min

Weights: \( w(x) = \text{depth of } x \)
  - Root has weight 0, its children have weight 1, etc.
Potential: \( \Phi(H) = \sum_x w(x) \)

Insert: \( \Delta \Phi = O(\log n) \implies \text{amortized cost} \leq O(\log n) + O(\log n) = O(\log n) \)

Extract-Min:
  - True cost: height \( h = \Theta(\log n) \) of tree, plus \( O(1) \) (for initial swap).
  - \( \Delta \Phi \): one less node at depth \( h \implies \Delta \Phi = -h \)
  - Amortized cost: \( h + O(1) - h = O(1) \).

Uses Inserts to “pay for” Extract-Mins.
Improvements

Downsides of binary heaps:

- Do at least as many Inserts as Extract-Mins! Want $O(1)$ Insert, $O(\log n)$ Extract-Min
- Meld in $O(n)$ is better than trivial, but still not great.

Binomial Heaps:
- Get Insert down to $O(1)$ (amortized)
- Meld in $O(\log n)$ (worst-case and amortized)
- Downside: $O(\log n)$ Extract-Min, $O(\log n)$ Find-Min

Fibonacci Heaps:
- Everything $O(1)$ (amortized) except $O(\log n)$ Extract-Min (amortized)
Improvements

Downsides of binary heaps:

- Do at least as many Inserts as Extract-Mins! Want $O(1)$ Insert, $O(\log n)$ Extract-Min
- Meld in $O(n)$ is better than trivial, but still not great.

Binomial Heaps:

- Get Insert down to $O(1)$ (amortized)
- Meld in $O(\log n)$ (worst-case and amortized)
- Downside: $O(\log n)$ Extract-Min, $O(\log n)$ Find-Min
Improvements

Downsides of binary heaps:

- Do at least as many Inserts as Extract-Mins! Want $O(1)$ Insert, $O(\log n)$ Extract-Min
- Meld in $O(n)$ is better than trivial, but still not great.

Binomial Heaps:

- Get Insert down to $O(1)$ (amortized)
- Meld in $O(\log n)$ (worst-case and amortized)
- Downside: $O(\log n)$ Extract-Min, $O(\log n)$ Find-Min

Fibonacci Heaps:

- Everything $O(1)$ (amortized) except $O(\log n)$ Extract-Min (amortized)
Binomial Heaps

Not based on binary tree anymore! Based on *binomial tree*. 
Binomial Heaps

Not based on binary tree anymore! Based on *binomial tree*.

- $B_0$ = single node.
- $B_k$ = one $B_{k-1}$ linked to another $B_{k-1}$.
### Lemma

The order \( k \) binomial tree \( B_k \) has the following properties:

1. Its height is \( k \).
2. It has \( 2^k \) nodes
3. The degree of the root is \( k \)
4. If we delete the root, we get \( k \) binomial trees \( B_{k-1}, \ldots, B_0 \).

<table>
<thead>
<tr>
<th>Properties</th>
<th>YH</th>
<th>HgivenH</th>
<th>HorderH</th>
<th>HkH</th>
<th>HbinomialHtreeH</th>
<th>B_kH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>ooo</td>
<td>HkH</td>
<td>H2^kH</td>
<td></td>
<td>B_k+1</td>
<td>B_{k+1}</td>
</tr>
<tr>
<td>Nodes</td>
<td></td>
<td>H2^k</td>
<td></td>
<td></td>
<td>B_1</td>
<td>B_1</td>
</tr>
<tr>
<td>Root</td>
<td></td>
<td>Hk</td>
<td></td>
<td></td>
<td>B_0</td>
<td>B_0</td>
</tr>
<tr>
<td>Trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B_{k-1}, \ldots, B_0</td>
<td>B_{k-1}, \ldots, B_0</td>
</tr>
</tbody>
</table>

**Diagram:**
- \( B_k \)
- \( B_{k+1} \)
- \( B_1 \)
- \( B_0 \)
- More binomial trees indicated with circles.
Binomial Heap

Definition

A binomial heap is a collection of binomial trees so that each tree is heap ordered, and there is exactly 0 or 1 tree of order $k$ for each integer $k$.

Keep roots of trees in linked list, from smallest order to largest

![Diagram of binomial heap with two trees: one of order 1 and one of order 2.](attachment://binomial_heap_diagram.png)
Binomial Heap

Definition

A binomial heap is a collection of binomial trees so that each tree is heap ordered, and there is exactly 0 or 1 tree of order $k$ for each integer $k$.

Keep roots of trees in linked list, from smallest order to largest

With $n$ items, no choices about which binomial trees exist in heap!

- Write $n$ in binary: $b_a b_{a-1} \ldots b_1 b_0$.
- Tree $B_k$ exists if and only if $b_k = 1$.
Binomial Heap

Definition

A binomial heap is a collection of binomial trees so that each tree is heap ordered, and there is exactly 0 or 1 tree of order $k$ for each integer $k$.

Keep roots of trees in linked list, from smallest order to largest

With $n$ items, no choices about which binomial trees exist in heap!

- Write $n$ in binary: $b_a b_{a-1} \ldots b_1 b_0$.
- Tree $B_k$ exists if and only if $b_k = 1$

$\implies$ at most $\log n$ trees, and by lemma each has height $\leq \log n$
Analysis: Beginning

Analyze all operations both worst-case and amortized.
Analysis: Beginning

Analyze all operations both worst-case and amortized.

Potential function: $\Phi(H) = \# \text{ trees in } H$

- Initially 0
- Never negative
Analysis: Beginning

Analyze all operations both worst-case and amortized.

Potential function: $\Phi(H) = \# \text{ trees in } H$
  
  - Initially 0
  - Never negative

Find-Min($H$): Scan through roots of trees in $H$, return min
Analysis: Beginning

Analyze all operations both worst-case and amortized.

Potential function: \( \Phi(H) = \# \) trees in \( H \)

- Initially 0
- Never negative

Find-Min(\( H \)): Scan through roots of trees in \( H \), return min

- Correct: each tree heap-ordered, so global min one of the roots
Analysis: Beginning

Analyze all operations both worst-case and amortized.

Potential function: \( \Phi(H) = \# \text{ trees in } H \)

- Initially 0
- Never negative

Find-Min\((H)\): Scan through roots of trees in \(H\), return min

- Correct: each tree heap-ordered, so global min one of the roots
- Worst-case: \(O(\log n)\)
- Amortized: doesn’t change potential, also \(O(\log n)\).
Meld($H_1, H_2$): Link

Key operation: we’ll use Meld to do Insert and Extract-Min
Meld($H_1, H_2$): Link

Key operation: we’ll use Meld to do Insert and Extract-Min

Warmup: $H_1, H_2$ both single trees of same order $k$.

- Union has size $2^k + 2^k = 2^{k+1}$: just a single $B_{k+1}$
- Easy to make a $B_{k+1}$ out of two $B_k$’s!
**Meld**($H_1, H_2$): Link

Key operation: we’ll use Meld to do Insert and Extract-Min

Warmup: $H_1, H_2$ both single trees of same order $k$.
- Union has size $2^k + 2^k = 2^{k+1}$: just a single $B_{k+1}$
- Easy to make a $B_{k+1}$ out of two $B_k$’s!
Meld($H_1, H_2$): Link

Key operation: we’ll use Meld to do Insert and Extract-Min

Warmup: $H_1, H_2$ both single trees of same order $k$.

- Union has size $2^k + 2^k = 2^{k+1}$: just a single $B_{k+1}$
- Easy to make a $B_{k+1}$ out of two $B_k$’s!

Link of two trees.

- Worst-case time: $O(1)$ (create a single link).
  Normalize: call 1
- $\Delta \Phi$: two trees to one: $-1$
- Amortized cost:
  $1 - 1 = 0 = O(1)$. 
Meld($H_1, H_2$): General Case

(Almost) just like binary addition!
Meld($H_1, H_2$): Analysis

Easy to prove correct (exercise for home).

Running time:
- Worst case: $O(1)$ per “order” $k \leq O(\log n)$
- Amortized: Potential does not go up, but could stay the same
  $\Rightarrow O(\log n)$ amortized
Insert($H, x$)

Use Meld:
- Create new heap $H'$ with one $B_0$ consisting of just $x$
- $\text{Meld}(H, H')$

Correctness: Obvious
Insert($H, x$)

Use Meld:
  - Create new heap $H'$ with one $B_0$ consisting of just $x$
  - $\text{Meld}(H, H')$

Correctness: Obvious

Running Time:
  - Worst case: $O(\log n)$ (via Meld)
Insert($H, x$)

Use Meld:
  - Create new heap $H'$ with one $B_0$ consisting of just $x$
  - Meld($H, H'$)

Correctness: Obvious

Running Time:
  - Worst case: $O(\log n)$ (via Meld)
  - Amortized:
    - Like incrementing a binary counter!
Insert($H, x$)

Use Meld:
- Create new heap $H'$ with one $B_0$ consisting of just $x$
- Meld($H, H'$)

Correctness: Obvious

Running Time:
- Worst case: $O(\log n)$ (via Meld)
- Amortized:
  - Like incrementing a binary counter!
  - If we link $k$ trees, potential goes down by $k - 1$
  - Cost = # links plus 1 (for making new heap)
  - Amortized cost = $k + 1 + \Delta \Phi = k + 1 - (k - 1) = 2 = O(1)$
**Extract-Min**($H$)

Use Meld again!

- $O(\log n)$ to Find-Min: one of the roots.
- Delete and return this root: tree turns into a new heap!
- Meld with original heap (minus the tree)

Correctness: Obvious
Extract-Min($H$)

Use Meld again!

- $O(\log n)$ to Find-Min: one of the roots.
- Delete and return this root: tree turns into a new heap!
- Meld with original heap (minus the tree)

Correctness: Obvious

Running Time:

- Worst-Case: $O(\log n)$ from creating new heap, Meld
- Amortized:
  - Potential can go up! But by at most $\log n$
  - Amortized time at most $O(\log n) + \log n = O(\log n)$