Lecture 8: Priority Queues and Heaps

Michael Dinitz

September 23, 2021
601.433/633 Introduction to Algorithms

Introduction

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

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

Extra Operations:

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

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

Introduction

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

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

Extra Operations:

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

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

Note: $\mathbf{x}$ is a *pointer* to an element. No way to lookup, so need a pointer to an element to change it.

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List				

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Sorted Array				

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$
Sorted Array	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$
Sorted Array	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$
Balanced Search Tree				

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$
Sorted Array	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$
Balanced Search Tree	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(n)}$

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$
Sorted Array	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$
Balanced Search Tree	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(n)}$

Goal: get as many of these to $\mathbf{O(1)}$ as possible

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$
Sorted Array	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$
Balanced Search Tree	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(n)}$

Goal: get as many of these to $\mathbf{O(1)}$ as possible

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

Obvious Approaches

	Insert	Extract-Min	Decrease-Key	Meld
Linked List	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(1)}$
Sorted Array	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$
Balanced Search Tree	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(n)}$

Goal: get as many of these to $\mathbf{O(1)}$ as possible

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

No! Sorting lower bound. But maybe can make one $\mathbf{O(1)}$, other $\mathbf{O(\log n)}$?

Today and State of the Art

State of the art: *strict Fibonacci Heaps*.

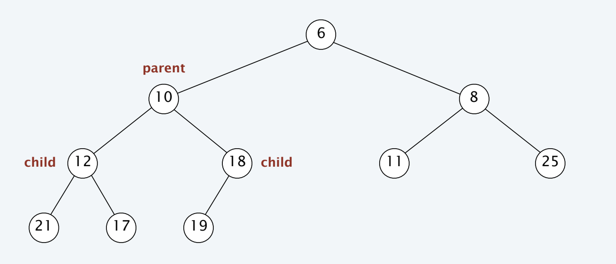
- Too complicated for this class, not practical. See CLRS 19 for Fibonacci Heaps.

Today: binary heaps (should be review), then binomial heaps

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

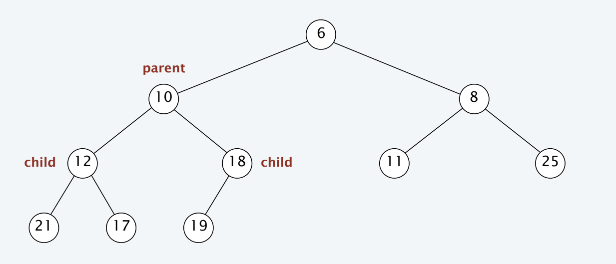
Binary Heaps

- 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.



Properties:

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

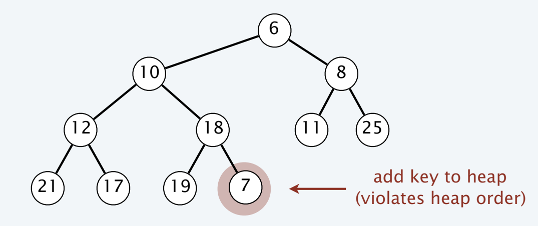
Representation:

- Pointers to root and rightmost leaf
- Every node has pointers to parent and children

Insert($\mathbf{H}, \mathbf{x}$)

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

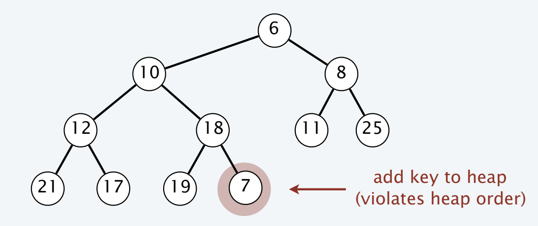
- Might violate heap *order*!



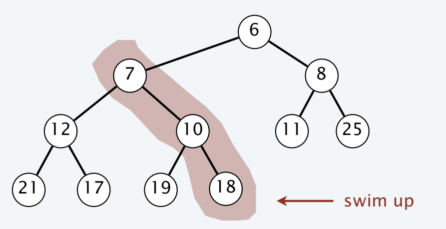
Insert($\mathbf{H}, \mathbf{x}$)

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

- Might violate heap *order*!



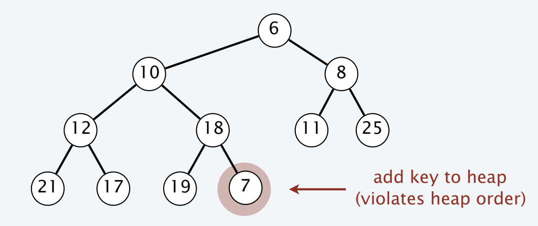
"Swim up": as long as **x** smaller than its parent, swap with parent



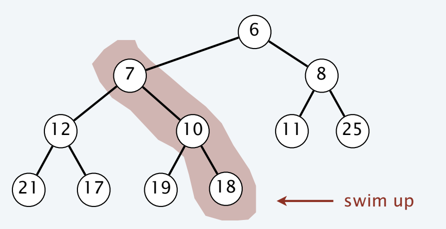
Insert($\mathbf{H}, \mathbf{x}$)

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: $\mathbf{O(\log n)}$ worst case (also amortized) via depth

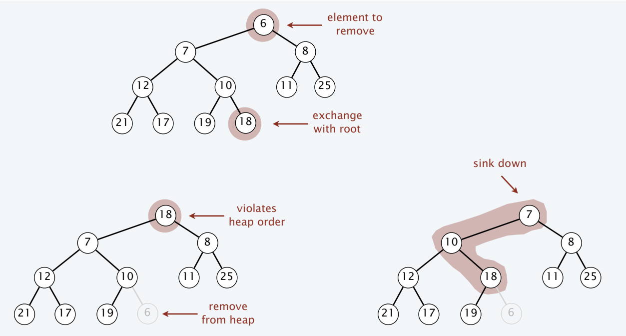
Extract-Min(**H**)

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?

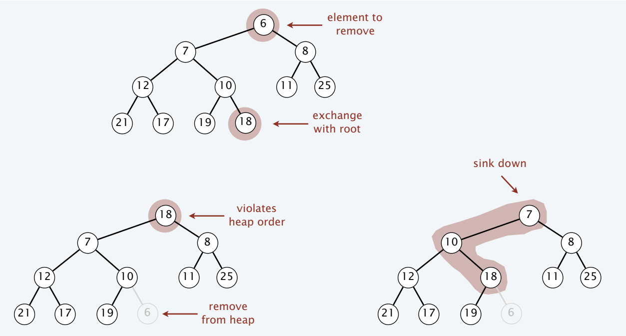
- Swap root with final heap element, remove former root.
- Sink down: swap root with smaller of its children until heap order restored



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: $\mathbf{O(\log n)}$ worst case (via depth). Amortized: $\mathbf{O(1)}$ (not obvious)

Decrease-Key($\mathbf{H}, \mathbf{x}, \mathbf{k}$)

Decrease key of $\mathbf{x}$ to $\mathbf{k}$, "swim up" until heap order restored.

Running time: $\mathbf{O(\log n)}$ (depth)

Meld($\mathbf{H_1}, \mathbf{H_2}$)

Assume both heaps have size $\mathbf{n}$.

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

Meld($\mathbf{H_1}, \mathbf{H_2}$)

Assume both heaps have size $\mathbf{n}$.

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

Better:

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

Meld($\mathbf{H_1}, \mathbf{H_2}$)

Assume both heaps have size $\mathbf{n}$.

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

Better:

- Insert all elements of $\mathbf{H_2}$ 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($\mathbf{H_1}, \mathbf{H_2}$)

Assume both heaps have size $\mathbf{n}$.

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

Better:

- Insert all elements of $\mathbf{H_2}$ 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: $\mathbf{O(n)}$ total

Meld($\mathbf{H_1}, \mathbf{H_2}$)

Assume both heaps have size $\mathbf{n}$.

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

Better:

- Insert all elements of $\mathbf{H_2}$ 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: $\mathbf{O(n)}$ total
- Sinking down:
  - Nodes at height $\mathbf{h}$ might have to sink down $\mathbf{h}$.
  - At most $\mathbf{n/2^h}$ nodes at height $\mathbf{h}$

Meld($\mathbf{H_1}, \mathbf{H_2}$)

Assume both heaps have size $\mathbf{n}$.

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

Better:

- Insert all elements of $\mathbf{H_2}$ 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: $\mathbf{O(n)}$ total
- Sinking down:
  - Nodes at height $\mathbf{h}$ might have to sink down $\mathbf{h}$.
  - At most $\mathbf{n/2^h}$ nodes at height $\mathbf{h}$

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

Amortized Extract-Min

Weights: $\mathbf{w(x)}$ = depth of $\mathbf{x}$

- Root has weight $\mathbf{0}$, its children have weight $\mathbf{1}$, etc.

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

Amortized Extract-Min

Weights: $\mathbf{w(x)}$ = depth of $\mathbf{x}$

- Root has weight $\mathbf{0}$, its children have weight $\mathbf{1}$, etc.

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

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

Amortized Extract-Min

Weights: $\mathbf{w(x)}$ = depth of $\mathbf{x}$

- Root has weight $\mathbf{0}$, its children have weight $\mathbf{1}$, etc.

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

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

Extract-Min:

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

Amortized Extract-Min

Weights: $\mathbf{w(x)}$ = depth of $\mathbf{x}$

- Root has weight $\mathbf{0}$, its children have weight $\mathbf{1}$, etc.

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

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

Extract-Min:

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

Uses Inserts to "pay for" Extract-Mins.

Improvements

Downsides of binary heaps:

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

Improvements

Downsides of binary heaps:

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

Binomial Heaps:

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

Improvements

Downsides of binary heaps:

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

Binomial Heaps:

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

Fibonacci Heaps:

- Everything $\mathbf{O(1)}$ (amortized) except $\mathbf{O(\log n)}$ Extract-Min (amortized)

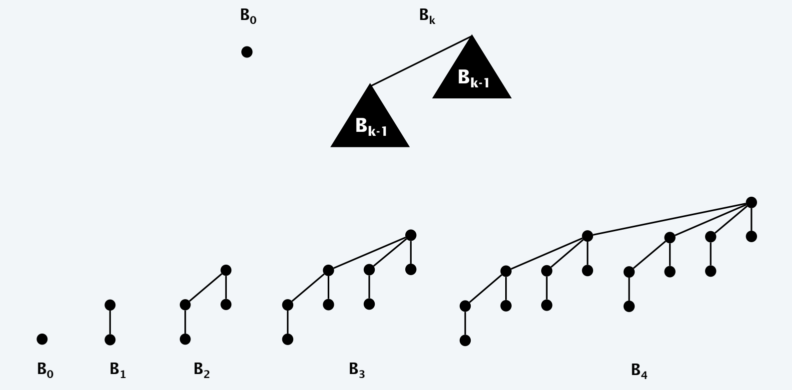
Binomial Heaps

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

Binomial Heaps

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

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

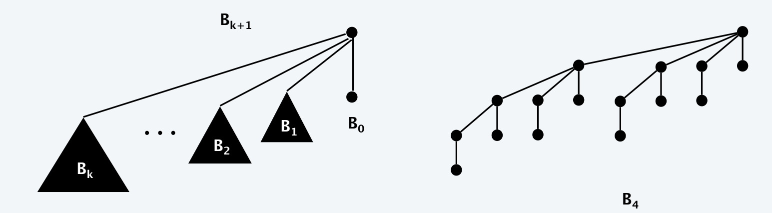


Structure Lemma

Lemma

The order $\mathbf{k}$ *binomial tree* $\mathbf{B_k}$ *has the following properties:*

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

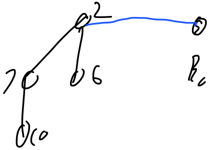


Binomial Heap

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 (not key!) to largest

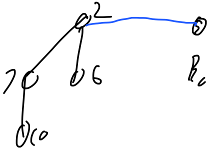


Binomial Heap

Definition

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

Keep roots of trees in linked list, from smallest order (not key!) to largest



With $\mathbf{n}$ items, no choices about which binomial trees exist in heap!

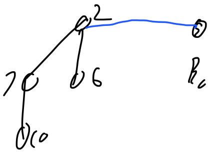
- Write $\mathbf{n}$ in binary: $\mathbf{b_a b_{a-1} \ldots b_1 b_0}$.
- Tree $\mathbf{B_k}$ exists if and only if $\mathbf{b_k = 1}$

Binomial Heap

Definition

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

Keep roots of trees in linked list, from smallest order (not key!) to largest



With $\mathbf{n}$ items, no choices about which binomial trees exist in heap!

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

$\Longrightarrow$ at most $\mathbf{\log n}$ trees, and by lemma each has height $\leq \mathbf{\log n}$

Analysis: Beginning

Analyze all operations both worst-case and amortized.

Analysis: Beginning

Analyze all operations both worst-case and amortized.

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

- Initially $\mathbf{0}$
- Never negative

Analysis: Beginning

Analyze all operations both worst-case and amortized.

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

- Initially $\mathbf{0}$
- Never negative

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

Analysis: Beginning

Analyze all operations both worst-case and amortized.

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

- Initially $\mathbf{0}$
- Never negative

Find-Min($\mathbf{H}$): Scan through roots of trees in $\mathbf{H}$, 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: $\mathbf{\Phi(H)}$ = # trees in $\mathbf{H}$

- Initially $\mathbf{0}$
- Never negative

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

- Correct: each tree heap-ordered, so global min one of the roots
- Worst-case: $\mathbf{O(\log n)}$
- Amortized: doesn't change potential, also $\mathbf{O(\log n)}$.

Meld($\mathbf{H_1}, \mathbf{H_2}$): Link

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

Meld($\mathbf{H_1}, \mathbf{H_2}$): Link

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

Warmup: $\mathbf{H_1}, \mathbf{H_2}$ both single trees of same order $\mathbf{k}$.

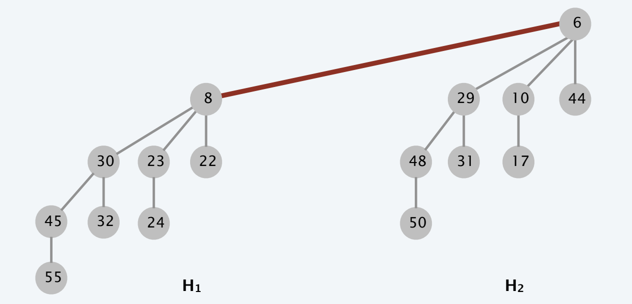
- Union has size $\mathbf{2^k + 2^k = 2^{k+1}}$: just a single $\mathbf{B_{k+1}}$
- Easy to make a $\mathbf{B_{k+1}}$ out of two $\mathbf{B_k}$'s!

Meld($\mathbf{H_1, H_2}$): Link

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

Warmup: $\mathbf{H_1, H_2}$ both single trees of same order $\mathbf{k}$.

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

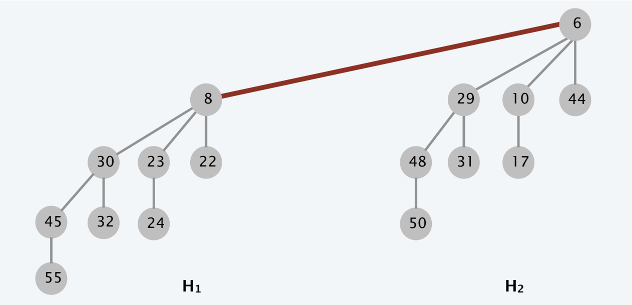


Meld($\mathbf{H_1}, \mathbf{H_2}$): Link

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

Warmup: $\mathbf{H_1}, \mathbf{H_2}$ both single trees of same order $\mathbf{k}$.

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

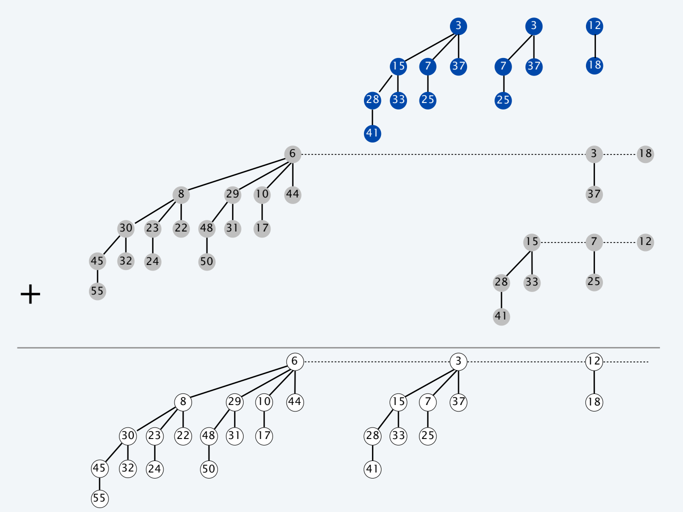


Link of two trees.

- Worst-case time: $\mathbf{O(1)}$ (create a single link). Normalize: call $\mathbf{1}$
- $\mathbf{\Delta\Phi}$: two trees to one: $\mathbf{-1}$
- Amortized cost: $\mathbf{1 - 1 = 0 = O(1)}$.

Meld($\mathbf{H_1}, \mathbf{H_2}$): General Case

(Almost) just like binary addition!



Meld($\mathbf{H_1}, \mathbf{H_2}$): Analysis

Easy to prove correct (exercise for home).

Running time:

- Worst case: $\mathbf{O(1)}$ per "order" $\mathbf{k} \implies \leq \mathbf{O(\log n)}$
- Amortized: Potential does not go up, but could stay the same $\implies$ $\mathbf{O(\log n)}$ amortized

Insert($\mathbf{H}, \mathbf{x}$)

Use Meld:

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

Correctness: Obvious

Insert($\mathbf{H}, \mathbf{x}$)

Use Meld:

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

Correctness: Obvious

Running Time:

- Worst case: $\mathbf{O(\log n)}$ (via Meld)

Insert($\mathbf{H}, \mathbf{x}$)

Use Meld:

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

Correctness: Obvious

Running Time:

- Worst case: $\mathbf{O(\log n)}$ (via Meld)
- Amortized:
  - Like incrementing a binary counter!

Insert($\mathbf{H}, \mathbf{x}$)

Use Meld:

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

Correctness: Obvious

Running Time:

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

Extract-Min($\mathbf{H}$)

Use Meld again!

- $\mathbf{O(\log n)}$ 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($\mathbf{H}$)

Use Meld again!

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