Lecture 8: Priority Queues and Heaps

Michael Dinitz

September 19, 2024

601.433/633 Introduction to Algorithms

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.

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

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

Obvious Approaches

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

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	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$
Sorted Array	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$

Obvious Approaches

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

Obvious Approaches

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

Obvious Approaches

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

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

Obvious Approaches

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

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

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

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 $O(\log n)$?

Today and State of the Art

State of the art: *strict Fibonacci Heaps*.

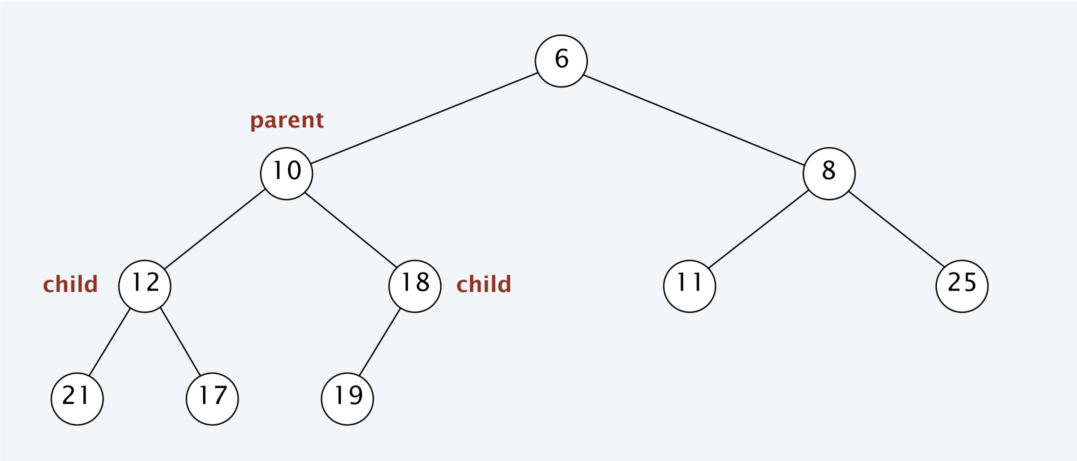
- Too complicated for this class, not practical. See CLRS 19 for Fibonacci Heaps.

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

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

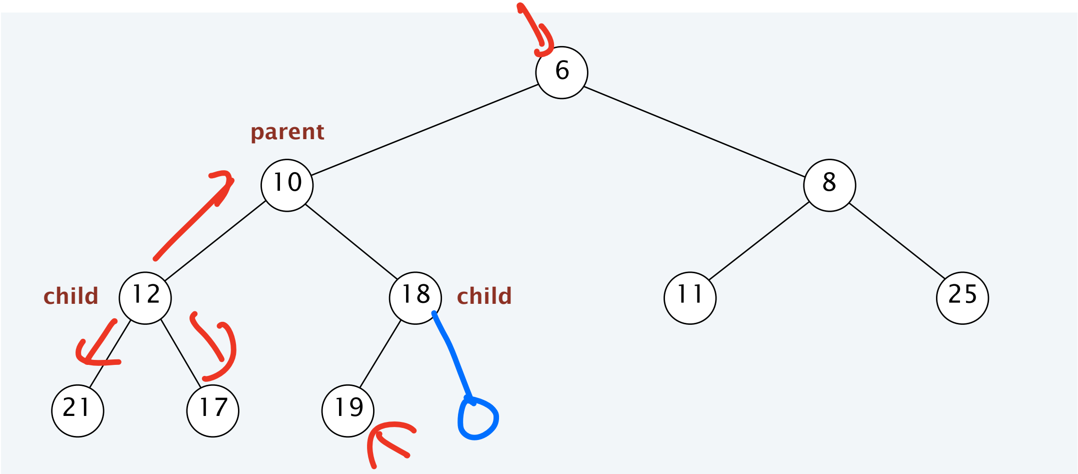
Binary Heaps

- 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

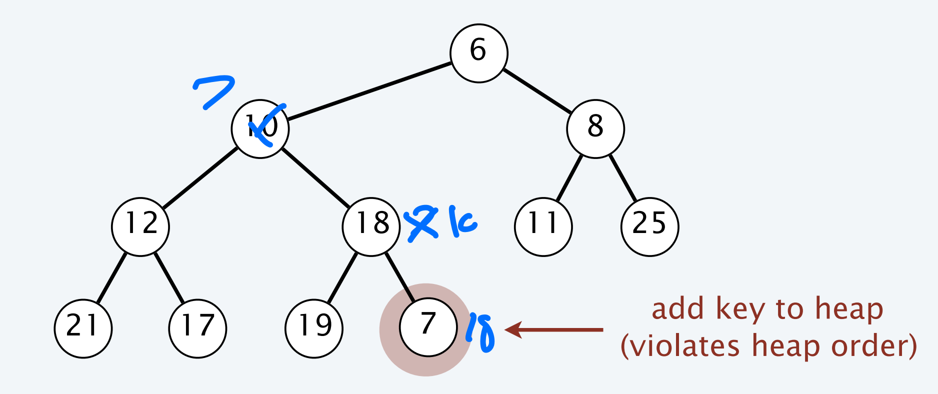
Representation:

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

Insert(H, x)

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

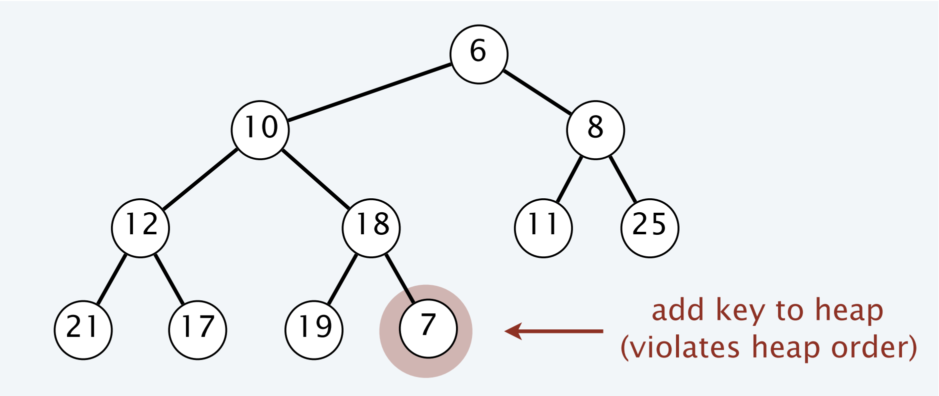
- Might violate heap *order*!



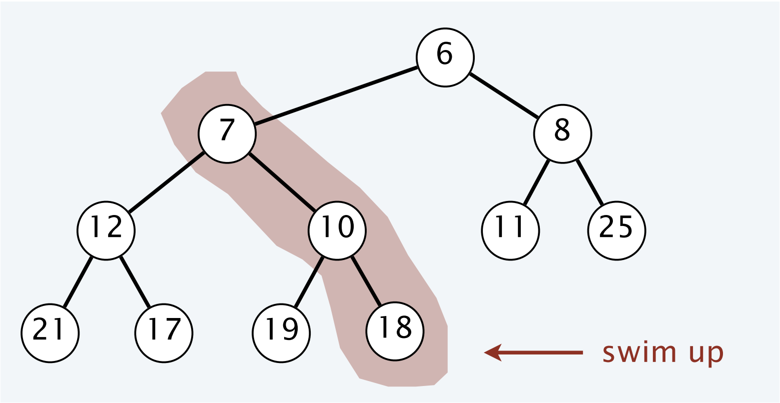
Insert($\boldsymbol{H}, x$)

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

- Might violate heap *order*!



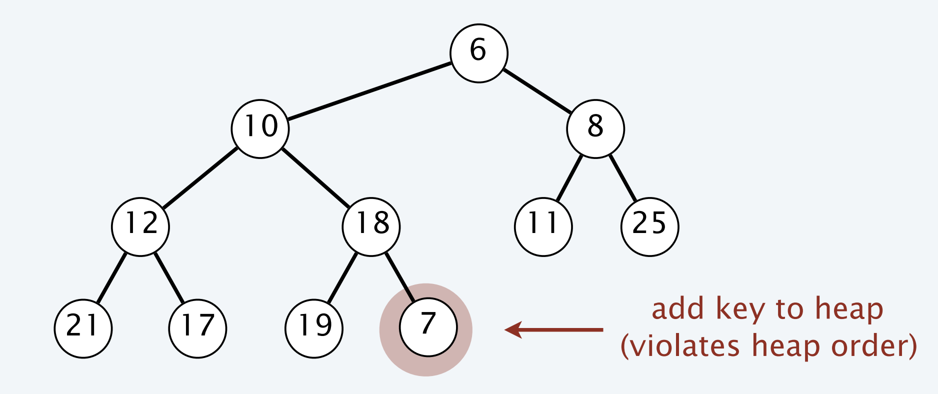
"Swim up": as long as $\boldsymbol{x}$ smaller than its parent, swap with parent



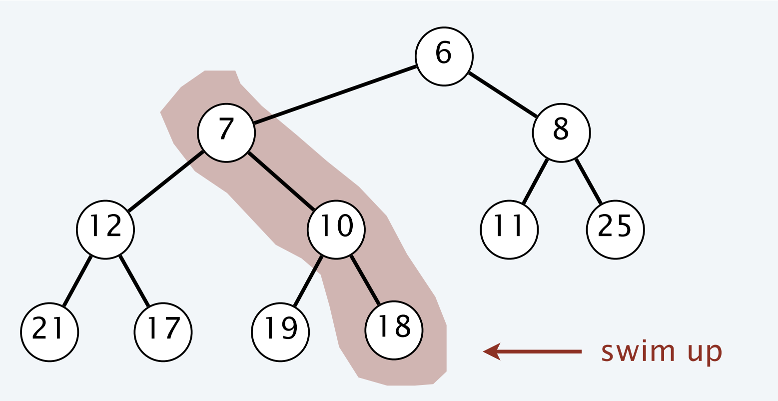
Insert($\boldsymbol{H}, \boldsymbol{x}$)

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

- Might violate heap *order*!



"Swim up": as long as $\boldsymbol{x}$ smaller than its parent, swap with parent



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

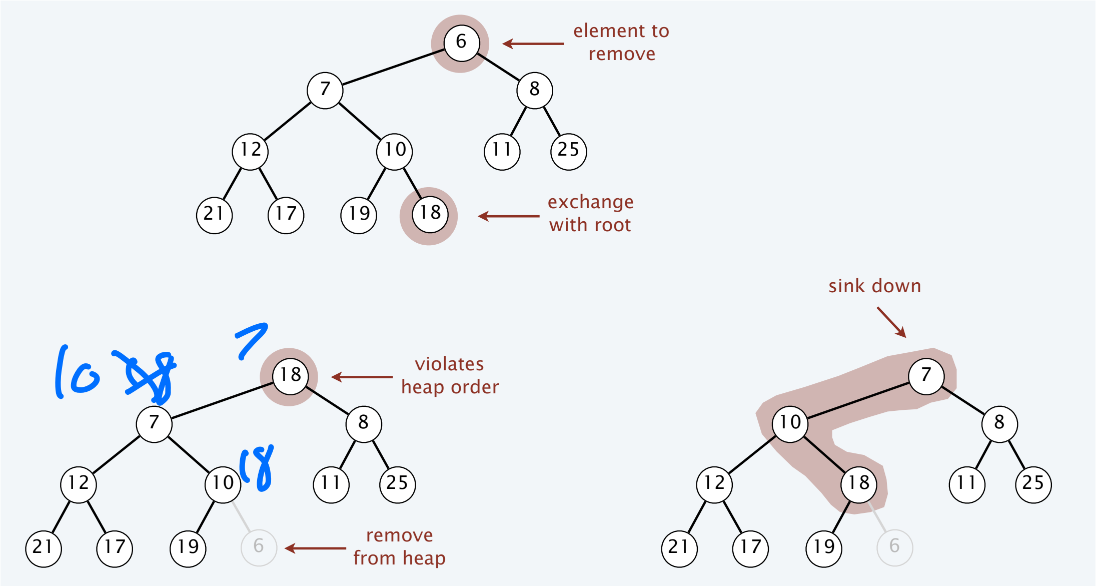
Extract-Min($\boldsymbol{H}$)

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?

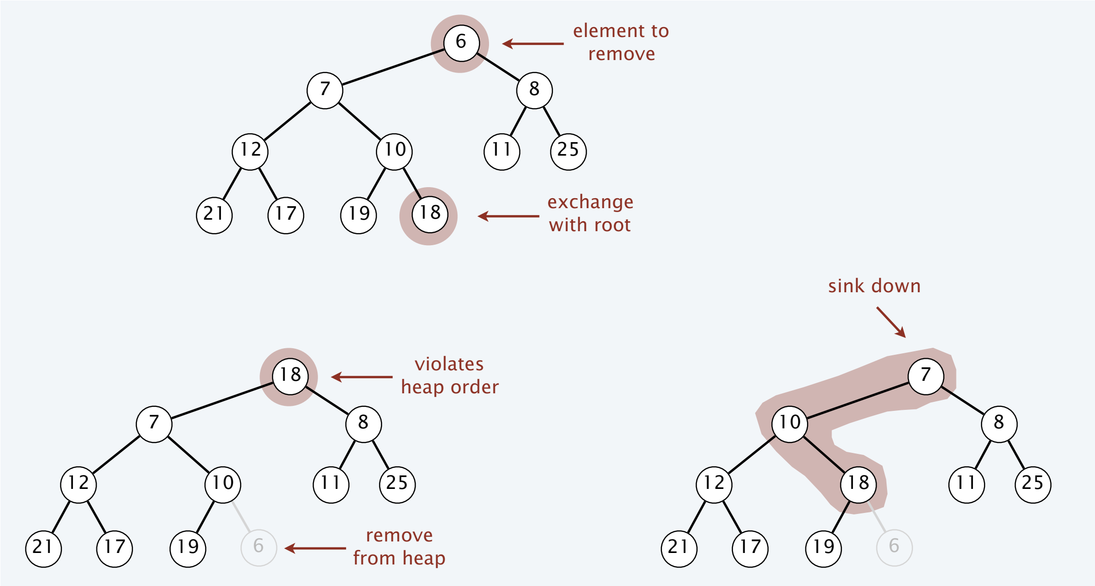
- Swap root with final heap element, remove former root.
- Sink down: swap root with smaller of its children until heap order restored



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: $O(\log n)$ 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: $O(n \log n)$

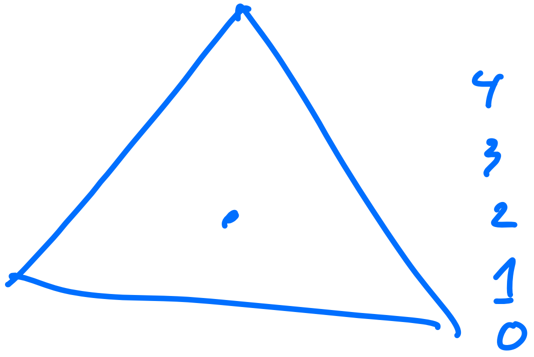
Better:

- 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

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

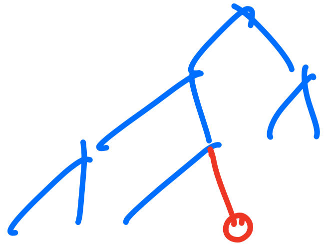
Amortized Extract-Min

Weights: $w(x)$ = depth of x

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

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

Amortized Extract-Min



Weights: $w(x)$ = depth of x

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

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

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

Amortized Extract-Min

Weights: $w(x)$ = depth of x

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

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

Insert: $\Delta\Phi = O(\log n) \implies$ 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: $h + O(1) - h = O(1)$.

Amortized Extract-Min

Weights: $w(x)$ = depth of x

- Root has weight 0, its children have weight 1, etc.

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

Insert: $\Delta\Phi = O(\log n) \implies$ 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.

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)

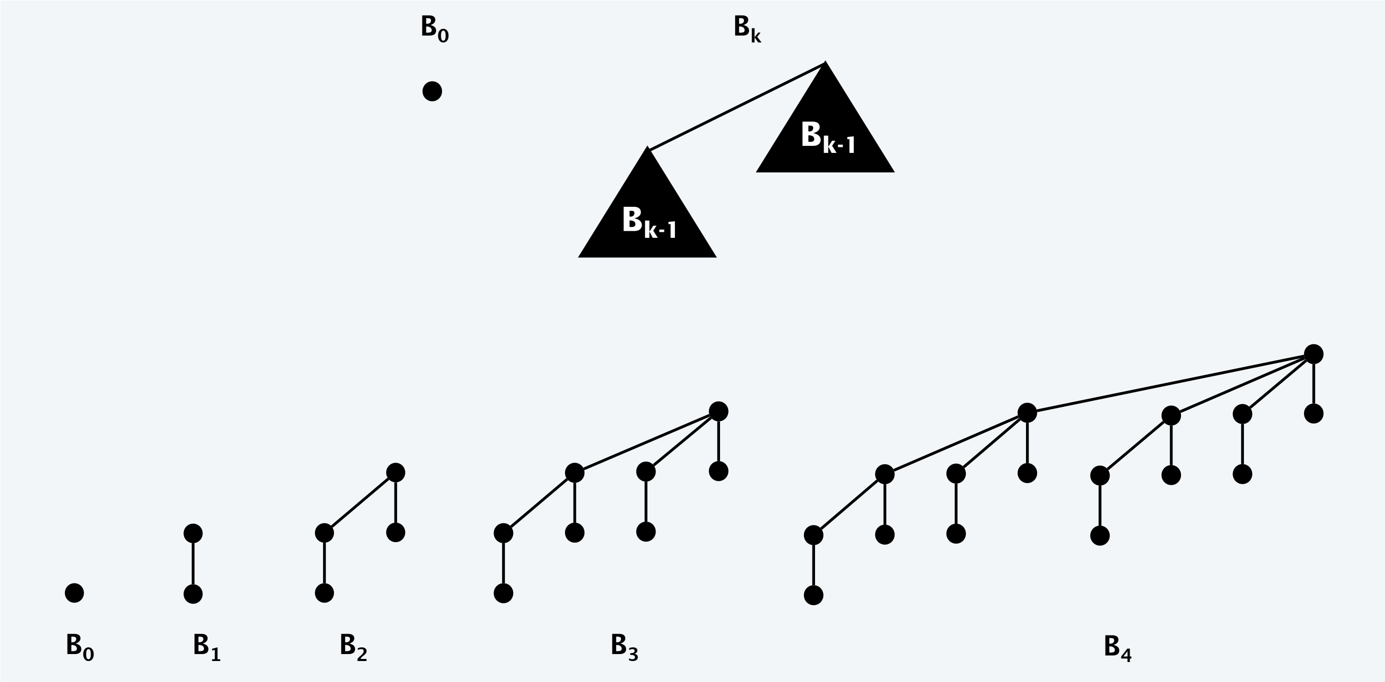
Binomial Heaps

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}.

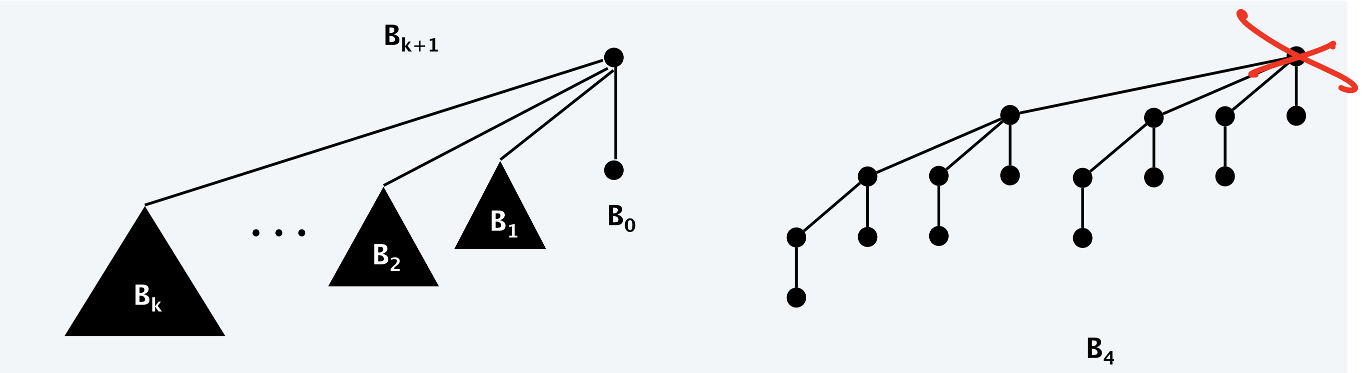


Structure Lemma

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 $B_{k-1}, \ldots, B_0$.*

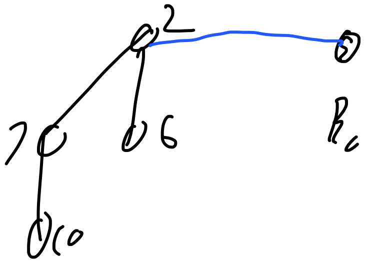


Binomial Heap

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

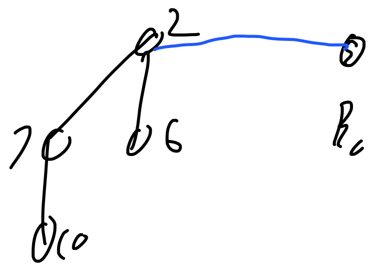


Binomial Heap

Definition

A *binomial heap* is a collection of binomial trees so that each tree is heap ordered, and there is exactly **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



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

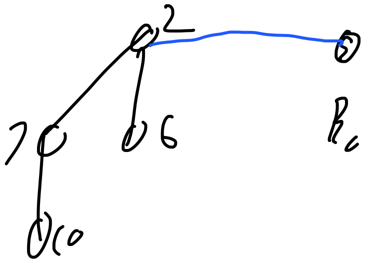
- Write n in binary: $b_a b_{a-1} \ldots b_1 b_0$.
- Tree B_k exists if and only if $b_k = 1$

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 (not key!) to largest



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

- Write n in binary: $b_a b_{a-1} \dots b_1 b_0$.
- 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: $\Phi(H) = \#$ trees in H

- Initially 0
- Never negative

Analysis: Beginning

Analyze all operations both worst-case and amortized.

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

- Initially 0
- Never negative

Find-Min(H):

Analysis: Beginning

Analyze all operations both worst-case and amortized.

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

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

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

Analysis: Beginning

Analyze all operations both worst-case and amortized.

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

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

Find-Min($\boldsymbol{H}$): Scan through roots of trees in $\boldsymbol{H}$, 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: $\Phi(H) = \#$ 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($\boldsymbol{H_1}, \boldsymbol{H_2}$): 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.

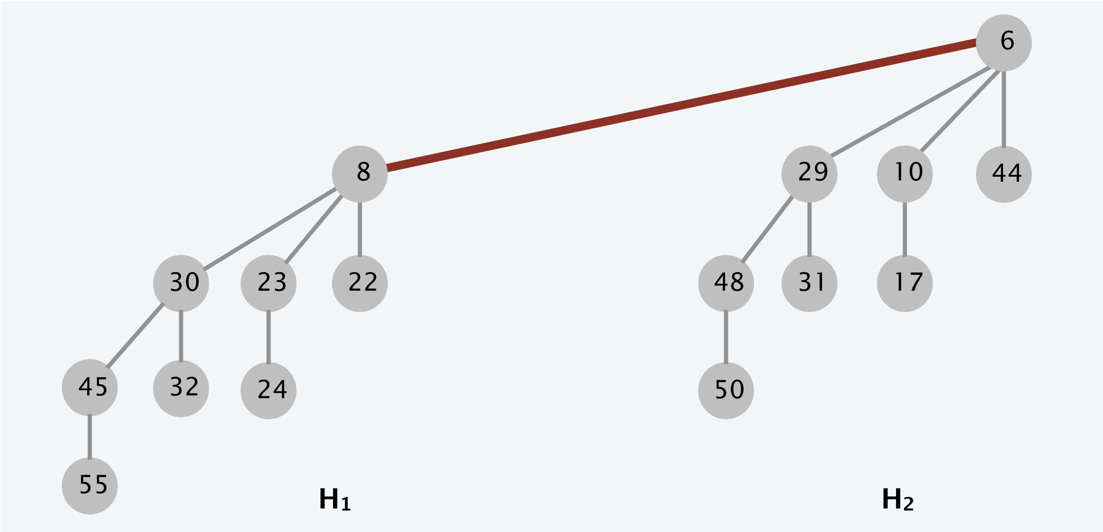
- Union has size $2^k + 2^k = 2^{k+1}$: just a single B_{k+1}
- Easy to make a B_{k+1} out of two B_k's!

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!

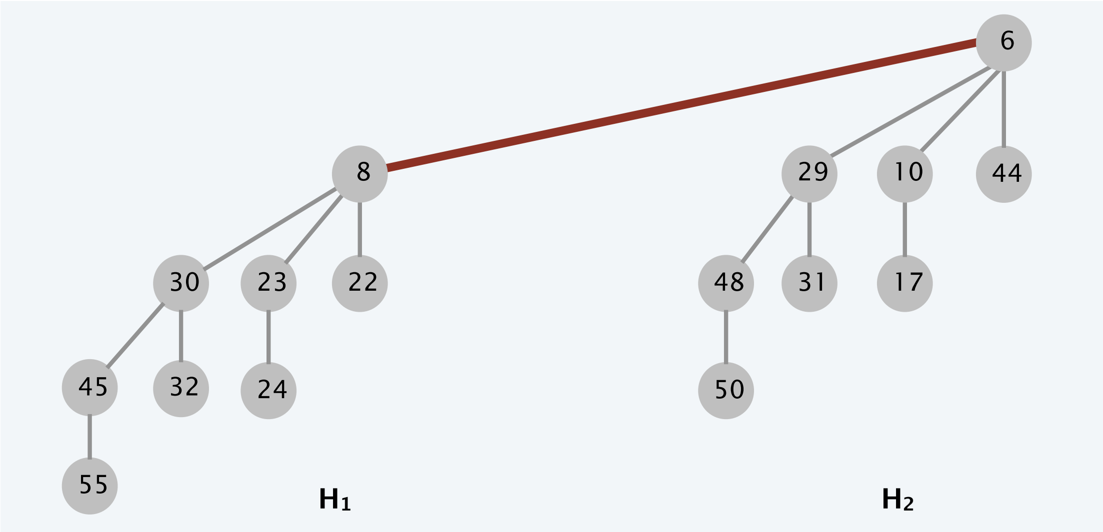


Meld(H_1, H_2): Link

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!

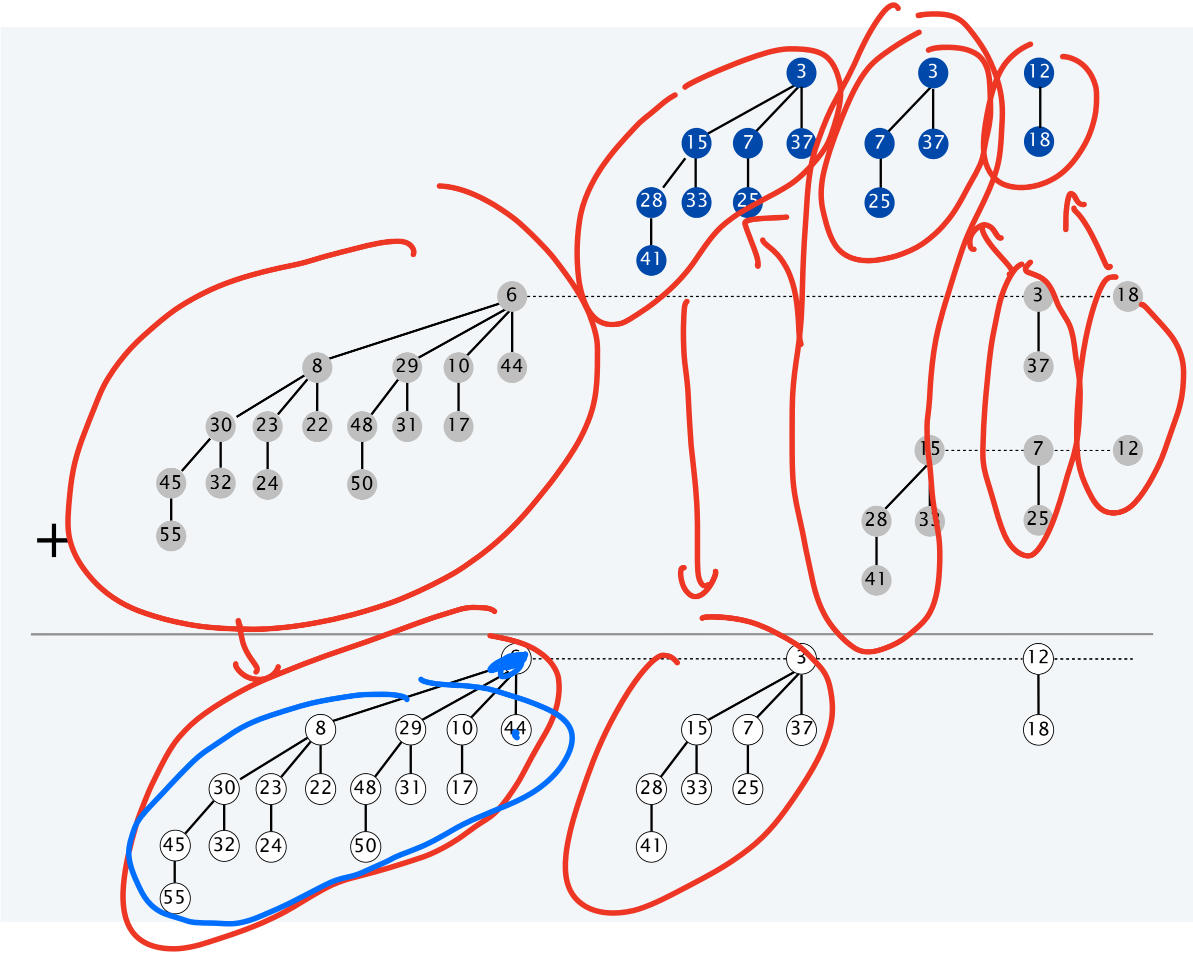


Link of two trees.

- 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" $k \implies \leq O(\log n)$
- Amortized: Potential does not go up, but could stay the same $\implies O(\log n)$ amortized

Insert(H, x)

Use Meld:

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

Correctness: Obvious

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)

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($\boldsymbol{H}$)

Use Meld again!

- $\boldsymbol{O(\log n)}$ 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 $O(\log n) + \log n = O(\log n)$