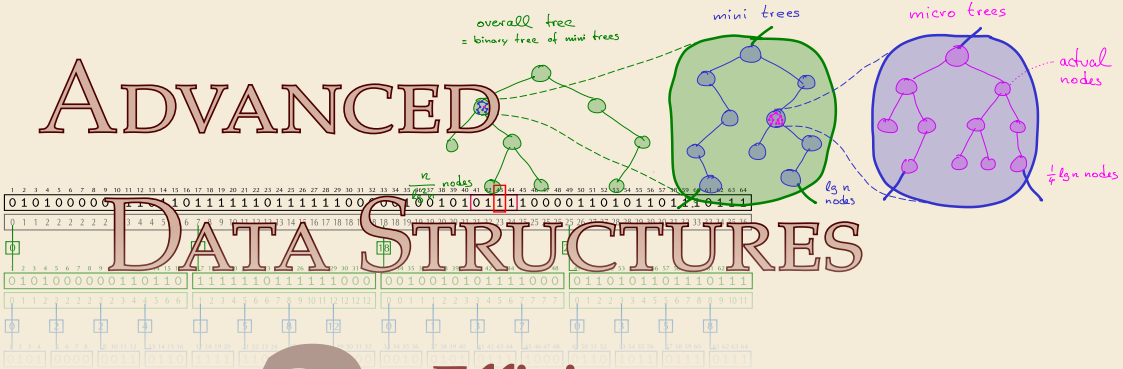


Advanced Data Structures

3 Efficient Priority Queues

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version **2026-06-01 16:01**

Outline

3 Efficient Priority Queues

Priority Queue ADT

(Min-oriented) Priority Queue (MinPQ/PQ):

- `construct`(A)
  Construct from from elements in array A.
- `insert`(x, p)
  Insert item x with priority p into PQ.
- `min`$()$
  Return item with smallest priority. (Does not modify the PQ.)
- `delMin`$()$
  Remove the item with smallest priority and return it.
- `decreaseKey`(x, p')
  Update x's priority to $p' \leq p$.
- `meld`(Q_1, Q_2)
  Build a PQ containing the union of Q_1 and Q_2

Fundamental building block
(Dijkstra, Prim, A*, event-driven simulation, . . .)

Elementary Solutions

Binary heaps can realize all operations efficiently (except meld).

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`decreaseKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`meld(`Q_1`,`Q_2`)`	$O(n)$

Elementary Solutions

Binary heaps can realize all operations efficiently (except meld).

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`decreaseKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`meld(`Q_1, Q_2`)`	$O(n)$

- apart from faster `construct`,
  BSTs always as good as binary heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`, `contains(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()`, `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`, `ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$
`split(`x`)`	$O(\log n)$
`join(`T_1, T_2`)` (for $T_1 \leq T_2$)	$O(\log n)$

Elementary Solutions

Binary heaps can realize all operations efficiently (except meld).

~~Binary heaps~~ *stay tuned*

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	~~$O(\log n)$~~ $O(1)$
`delMax()`	$O(\log n)$
`decreaseKey(`x`,`p'`)`	~~$O(\log n)$~~ $O(1)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`meld(`Q_1`,`Q_2`)`	~~$O(n)$~~ $O(1)$

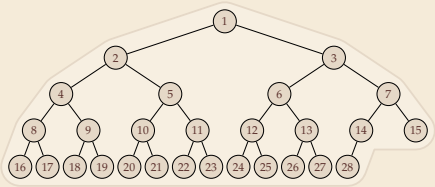
- apart from faster `construct`, BSTs always as good as binary heaps
- PQ abstraction still helpful
- faster heaps exist!

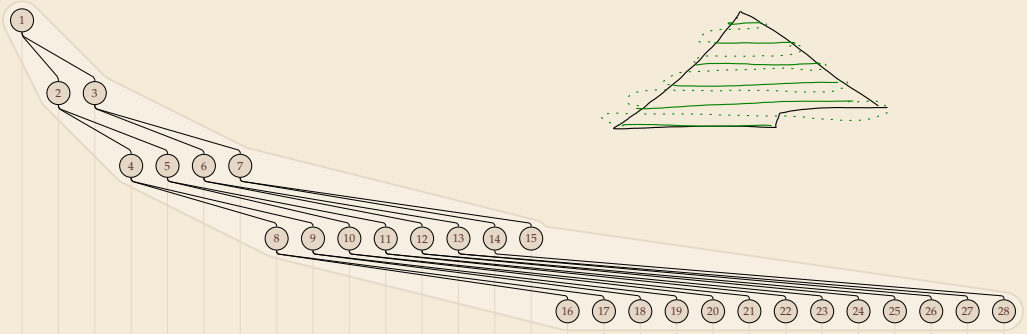
Balanced binary search tree

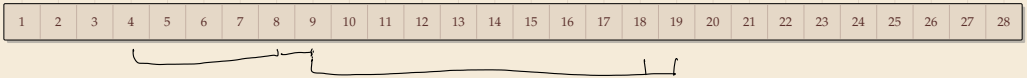
Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`),contains(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min(),max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`),ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$
`split(`x`)`	$O(\log n)$
`join(`T_1`,`T_2`)` (for $T_1 \leq T_2$)	$O(\log n)$

3.1 Tournament Trees

Implicit Complete Binary Trees







Del Min

remove root

swap last-root

sink down

$\leq \log n + 1$

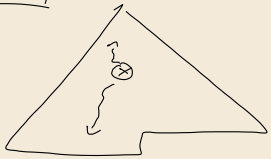
(Floyd's trick)

$\#\text{cmps} \leq 2 \log n \pm O(1)$

Insert

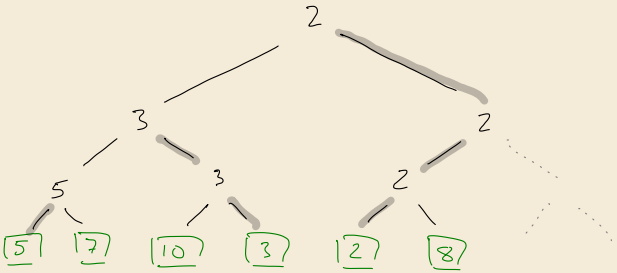
$\leq \log n \pm O(1)$

Change Key



$O(\log n)$

Tournament Trees = Leaf-Oriented Heaps



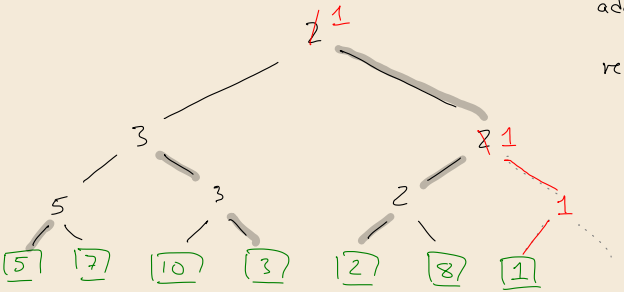
all elements in leaves

layers above promote min

⇒ heap ordered

can use similar "implicit" representation as for heap

Insert 1



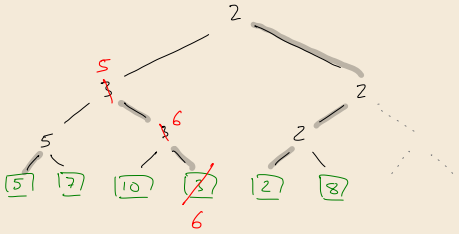
adds a new rightmost leaf

recompute path to root

$O(\log n)$

FindMin : in root

ChangeValue (ptr, p')



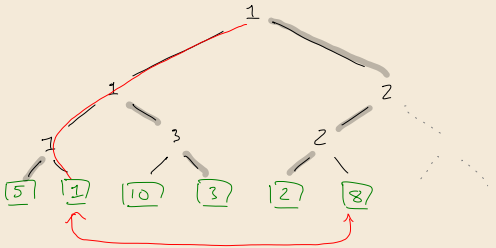
$O(\log n)$

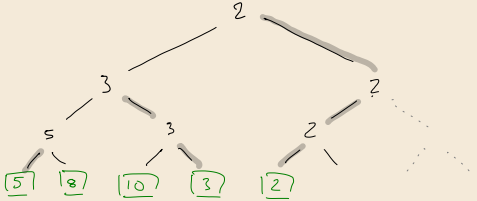
Delete Min : Lazy version : follow root-to-leaf path always $-\infty$ edge
→ min leaf
change value to ∞

$O(\log n)$

Eager version : Swap min-leaf (find as above) and rightmost
Recompute path from both changed leaves

Delete Min





Why tournament trees? • best multiway merging implementation!
• basis of quake heaps!

Who Needs Rigid Shapes?

For binary heaps & tournament trees (arrays do)

rigid shape

---

alternative : allow any binary-tree shape (but w/ heap order)

↳ & implicit representation in array

FindMin same

Insert same

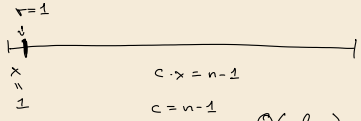
DeleteMin in binary heaps : remove root, promote smaller child up

tournament trees : simpler eager delete w/o swap

(not usually good, but possible ... and used in Quake Heaps)

3.2 Lazy-Partition Heaps

Our Solution: Lazy Search Trees



Sandlund, Wild: *Lazy Search Trees*, FOCS **2020**

Theorem 2.25 (Lazy search tree runtimes)

For n the current number of elements, $\{\Delta_i\}$ as above and q the total number of queries, we do

- `Insert(`x`)` in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|,\ \log q))$ worst-case time, where $x \in \Delta_i$. $|\Delta_i| = n$
- `RankBasedQuery(`r`)` in $O(x \log c + \log n)$ amortized time, resulting gaps have size cx and x, $c \geq 1$.
- **Sequence of insert and query w/o duplicate queries** in total $O(\mathcal{B} + \min(n \log\log n, n \log q))$.
- `Construction(`S`)` in $O(n)$ worst-case time, where $|S| = n$.
- `Delete(ptr)` in $O(\log n)$ worst-case time.
- `ChangeKey(ptr,` x'`)` in $O(\min(\log q, \log\log|\Delta_i|))$ worst-case time, where the element at `ptr`, is $x \in \Delta_i$ and x' lies to its closest query rank in Δ_i; otherwise, takes $O(\log n)$ worst-case time.
- `Split(`r`)` in time according to `RankBasedQuery(`r`)`.
- `Merge(`T_1`,` T_2`)` (where $T_1 \leq T_2$) in $O(\log n)$ worst-case time. ◀

In short: all operations amortized to lower bound, except $+O(\log\log n)$ per insert

$O(\log\log n)$ later removed via selectable heaps

Sandlund, Zhang: *Selectable Heaps and Optimal Lazy Search Trees*, SODA **2022**

72

Lazy Search Trees in PQ Mode

When using Lazy Search Trees only with Min-queries, we obtain

- `DeleteMin` in $O(\log n)$ amortized time
- `Insert` in $O(\log \log n)$ worst case time
- `DecreaseKey` in $O(\log \log n)$ worst case time

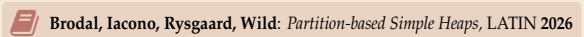
Lazy Search Trees in PQ Mode

When using Lazy Search Trees only with Min-queries, we obtain

- `DeleteMin` in $O(\log n)$ amortized time
- `Insert` in $O(\log \log n)$ worst case time
- `DecreaseKey` in $O(\log \log n)$ worst case time

It turns out, this special case allows some nontrivial simplification over general Lazy Search Trees.

- In particular: One ever have a single 1-sided gap $\rightsquigarrow$ no need to represent
- Present here as a self-contained data structure $\rightsquigarrow$ adapted notation



$I_{i,j} \rightsquigarrow S_j$

$o(I_{i,j}) \rightsquigarrow s(S_j)$

"

smaller

Lazy Partition Heaps

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

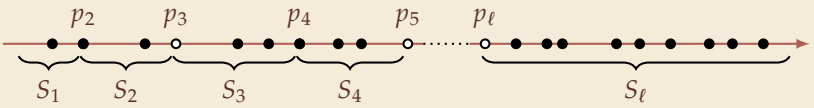
A **partition-based heap** is a *"partially completed Quicksort"*

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

A **partition-based heap** is a *"partially completed Quicksort"*

- It consists of *sets* S_i and *pivots* p_i:



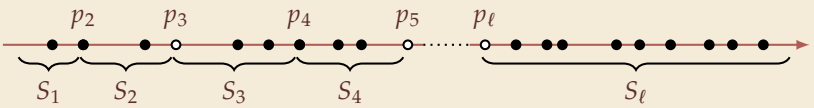
Pivots can be elements in the heap (black) or elements removed from the heap (white)

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

A **partition-based heap** is a *"partially completed Quicksort"*

- It consists of *sets* S_i and *pivots* p_i:



Pivots can be elements in the heap (black) or elements removed from the heap (white)

- $\boxed{S_i \subseteq [p_i, p_{i+1})}$

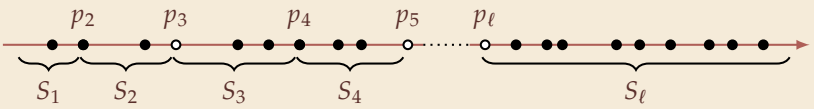
  Pivots $p_2 \ldots, p_\ell$ partition the n elements into ℓ sets $S_1, S_2, \ldots, S_\ell$

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

A **partition-based heap** is a *"partially completed Quicksort"*

- It consists of *sets* S_i and *pivots* p_i:



Pivots can be elements in the heap (black) or elements removed from the heap (white)

- $\boxed{S_i \subseteq [p_i, p_{i+1})}$

  Pivots $p_2 \ldots, p_\ell$ partition the n elements into ℓ sets $S_1, S_2, \ldots, S_\ell$

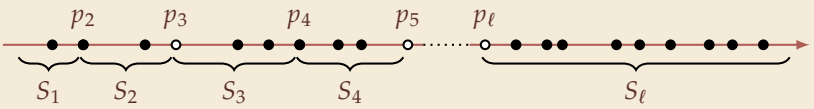
- Sets are linked lists $\rightsquigarrow$ pointers to list nodes are stable (*referential integrity*)

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

A **partition-based heap** is a *"partially completed Quicksort"*

- It consists of *sets* S_i and *pivots* p_i:



Pivots can be elements in the heap (black) or elements removed from the heap (white)

- $\boxed{S_i \subseteq [p_i, p_{i+1})}$

  Pivots $p_2 \ldots, p_\ell$ partition the n elements into ℓ sets $S_1, S_2, \ldots, S_\ell$

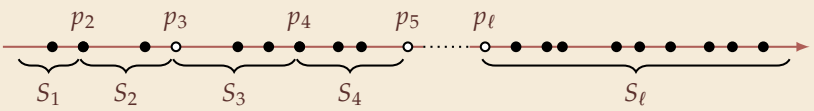
- Sets are linked lists $\rightsquigarrow$ pointers to list nodes are stable (*referential integrity*)
- Pivots are stored in a sorted array or BST $\rightsquigarrow$ can search sets in $O(\log \ell)$ time

Partition-Based Heaps Template

Generic template for priority queues based on partitioning; ***balance invariants left open****.*

A **partition-based heap** is a *"partially completed Quicksort"*

- It consists of *sets* S_i and *pivots* p_i:



Pivots can be elements in the heap (black) or elements removed from the heap (white)

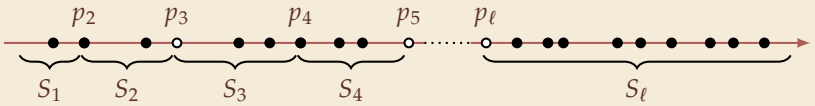
- $\boxed{S_i \subseteq [p_i, p_{i+1})}$

  Pivots $p_2 \ldots, p_\ell$ partition the n elements into ℓ sets $S_1, S_2, \ldots, S_\ell$

- Sets are linked lists $\rightsquigarrow$ pointers to list nodes are stable (*referential integrity*)
- Pivots are stored in a sorted array or BST $\rightsquigarrow$ can search sets in $O(\log \ell)$ time
- Keep number of sets $\ell = O(\log n)$ $\rightsquigarrow$ $O(\log \ell) = O(\log \log n)$

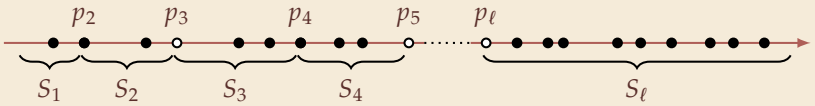
  How? Stay tuned!

Partition-Based Heaps – Operations



Core of operations fixed by template: may need to do maintenance on top

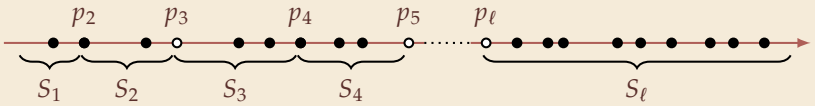
Partition-Based Heaps – Operations



Core of operations fixed by template: may need to do maintenance on top

- `Insert`(e):
  Search e among the ℓ pivots ($O(\log \ell)$ time)
  $\rightsquigarrow$ S_i with $p_i \leq e < p_{i+1}$
  Append e to S_i and return a pointer ptr to the new list node

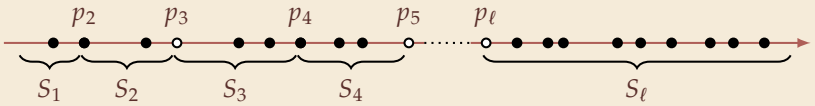
Partition-Based Heaps – Operations



Core of operations fixed by template: may need to do maintenance on top

- `Insert`(e):
  Search e among the ℓ pivots ($O(\log \ell)$ time)
  $\rightsquigarrow$ S_i with $p_i \leq e < p_{i+1}$
  Append e to S_i and return a pointer ptr to the new list node
- `DeleteMin`():
  Remove the smallest element from the first set S_1 ($O(|S_1|)$ time)

Partition-Based Heaps – Operations



Core of operations fixed by template: may need to do maintenance on top

- `Insert`(e):
  Search e among the ℓ pivots ($O(\log \ell)$ time)
  $\rightsquigarrow$ S_i with $p_i \leq e < p_{i+1}$
  Append e to S_i and return a pointer *ptr* to the new list node
- `DeleteMin`():
  Remove the smallest element from the first set S_1 ($O(|S_1|)$ time)
- `DecreaseKey`(ptr, key): Remove element via *ptr*, reinsert ($O(\log \ell)$ time)

Lazy Partition Heaps

Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

- `DeleteMin` always **partitions** smallest set S_1 (skipping empty sets) around **median**

Lazy Partition Heaps

Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

- `DeleteMin` always **partitions** smallest set S_1 (skipping empty sets) around **median**
- **Invariant (S):** Number of sets $\ell \leq 2 \lg n + 1$
  - only at risk upon `DeleteMin` $\rightsquigarrow$ there we apply rule (C) below
  - simple induction shows: (C) implies (S)

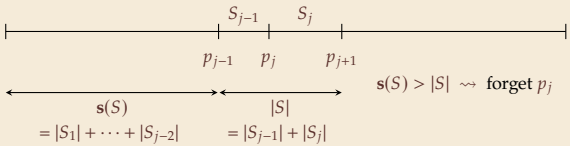
Lazy Partition Heaps

Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

skipping empty sets

- `DeleteMin` always **partitions** smallest set S_1 around **median**
- **Invariant (S):** Number of sets $\ell \leq 2 \lg n + 1$
  - only at risk upon `DeleteMin` $\rightsquigarrow$ there we apply rule (C) below
  - simple induction shows: (C) implies (S)
- **Lazy Concatenation rule (C):** **Forget** pivot p_j if $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$ (= rule (M))



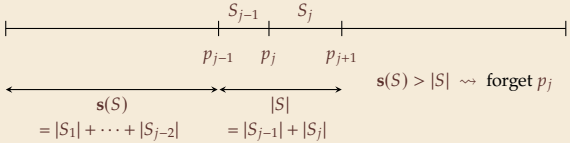
Lazy Partition Heaps

Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

skipping empty sets

- `DeleteMin` always **partitions** smallest set S_1 around **median**
- **Invariant (S):** Number of sets $\ell \leq 2 \lg n + 1$
  - only at risk upon `DeleteMin` $\rightsquigarrow$ there we apply rule (C) below
  - simple induction shows: (C) implies (S)
- **Lazy Concatenation rule (C):** **Forget** pivot p_j if $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$
  - **NOT** an invariant!
    we tolerate that (C) can be violated;
    only enforced right after `DeleteMin`
  - Forgetting a pivot means **concatenating** the linked lists of S_j and S_{j+1}



Lazy Partition Heaps

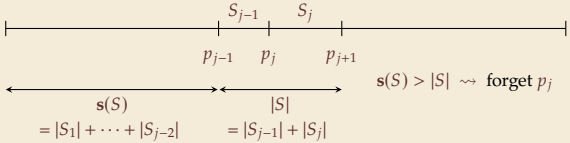
Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

skipping empty sets

- `DeleteMin` always **partitions** smallest set S_1 around **median**
- **Invariant (S):** Number of sets $\ell \leq 2 \lg n + 1$
  - only at risk upon `DeleteMin` $\rightsquigarrow$ there we apply rule (C) below
  - simple induction shows: (C) implies (S)
- **Lazy Concatenation rule (C):** **Forget** pivot p_j if $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$
  - **NOT** an invariant!
    we tolerate that (C) can be violated;
    only enforced right after `DeleteMin`
  - Forgetting a pivot means **concatenating** the linked lists of S_j and S_{j+1}

$\rightsquigarrow$ Enforcing (C) costs $O(\ell)$



LP Heaps – Analysis

- **Actual Costs**
  - `Insert` $O(\log \log n)$ — binary search over pivots, append
  - `DecreaseKey` $O(\log \log n)$ — remove from set, re-insert
  - `DeleteMin` $O(|S_1| + \log n)$ — split S_1 and apply rule (C); recompute sorted-pivot array

    could be n

LP Heaps – Analysis

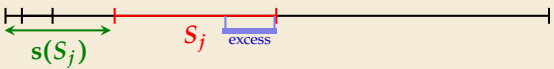
- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$

    could be n

$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - s(S_j)\}$$



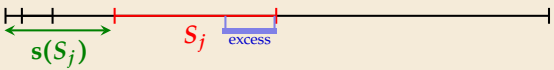
LP Heaps – Analysis

- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$
    could be n

$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - s(S_j)\}$$



- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$

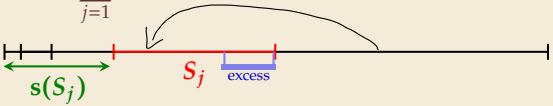
LP Heaps – Analysis

- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$
    could be n

$\rightsquigarrow$ Introduce **potential**

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - s(S_j)\}$$

excess size over distance from min



- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - `DecreaseKey` $\Delta\Phi \leq 0$ since we only increase "protection distance"

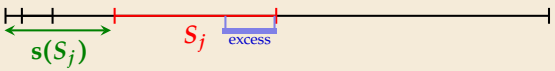
LP Heaps – Analysis

- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$
    could be n

$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

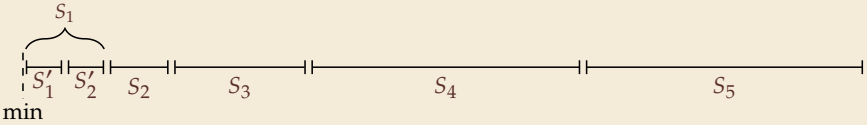
$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - s(S_j)\}$$



- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - `DecreaseKey` $\Delta\Phi \leq 0$ since we only increase "protection distance"
  - `DeleteMin` (1) Partitioning S_1 into S'_1 and S'_2 eliminates $\Phi_1 = |S_1|$ and adds $\Phi'_1 + \Phi'_2 = \frac{1}{2}|S_1|$

$\Phi'_1 + \Phi'_2 = 0$

$|S'_1|$



LP Heaps – Analysis

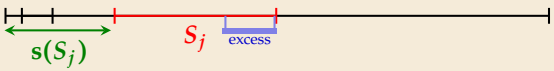
- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$

    could be n

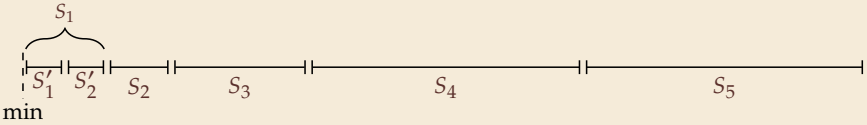
$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - \mathbf{s}(S_j)\}$$



- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - `DecreaseKey` $\Delta\Phi \leq 0$ since we only increase "protection distance"
  - `DeleteMin` (1) Partitioning S_1 into S_1' and S_2' eliminates $\Phi_1 = |S_1|$ and adds $\Phi_1' + \Phi_2' = \frac{1}{2}|S_1|$
    (2) Moreover, deleting the min removes 1 from $\mathbf{s}(S_j)$ of each set



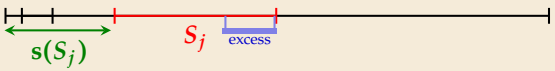
LP Heaps – Analysis

- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$
    could be n

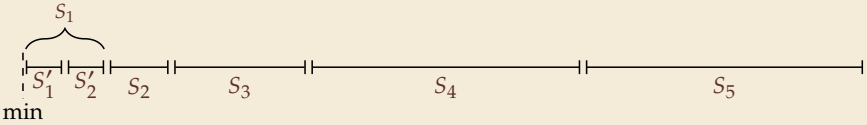
$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - \mathbf{s}(S_j)\}$$



- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - `DecreaseKey` $\Delta\Phi \leq 0$ since we only increase "protection distance"
  - `DeleteMin` (1) Partitioning S_1 into S'_1 and S'_2 eliminates $\Phi_1 = |S_1|$ and adds $\Phi'_1 + \Phi'_2 = \frac{1}{2}|S_1|$
    (2) Moreover, deleting the min removes 1 from $\mathbf{s}(S_j)$ of each set
    (3) Rule (C) precisely chosen, so that $\Phi_j = 0$ before and after $\rightsquigarrow$ no change!



LP Heaps – Analysis

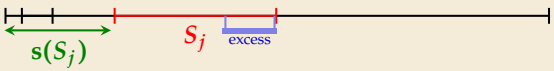
- **Actual Costs**
  - `Insert` $O(\log\log n)$
  - `DecreaseKey` $O(\log\log n)$
  - `DeleteMin` $O(|S_1| + \log n)$

    could be n

$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - \mathbf{s}(S_j)\}$$

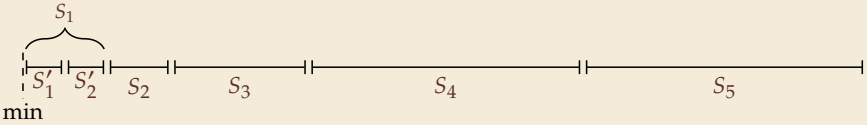


- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - `DecreaseKey` $\Delta\Phi \le 0$ since we only increase "protection distance"
  - `DeleteMin` (1) Partitioning S_1 into S_1' and S_2' eliminates $\Phi_1 = |S_1|$ and adds $\Phi_1' + \Phi_2' = \frac{1}{2}|S_1|$

    (2) Moreover, deleting the min removes 1 from $\mathbf{s}(S_j)$ of each set

    (3) Rule (C) precisely chosen, so that $\Phi_j = 0$ before and after $\rightsquigarrow$ no change!

$\rightsquigarrow$ $\Delta\Phi \le -\frac{1}{2}|S_1| + \ell = \Theta(-|S_1| + \log n)$



LP Heaps – Analysis

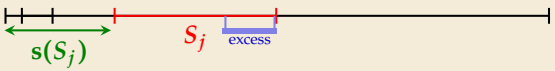
- **Actual Costs**
  - `Insert` $O(\log \log n)$
  - `DecreaseKey` $O(\log \log n)$
  - `DeleteMin` $O(|S_1| + \log n)$

    could be n

⇝ Introduce **potential**

excess size over distance from min

$$\Phi := \sum_{j=1}^{\ell} \Phi_j \quad \text{with} \quad \Phi_j := \max\{0, |S_j| - \mathbf{s}(S_j)\}$$



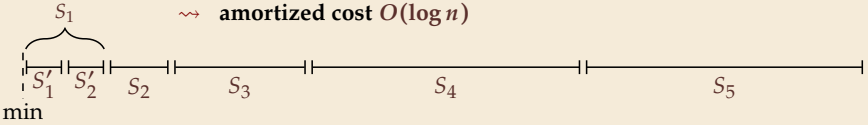
- **Change in Potential**
  - `Insert` Only Φ_j increases (by 1) ⇝ $\Delta\Phi = O(1)$
  - `DecreaseKey` $\Delta\Phi \le 0$ since we only increase "protection distance"
  - `DeleteMin` (1) Partitioning S_1 into S_1' and S_2' eliminates $\Phi_1 = |S_1|$ and adds $\Phi_1' + \Phi_2' = \frac{1}{2}|S_1|$

    (2) Moreover, deleting the min removes 1 from $\mathbf{s}(S_j)$ of each set

    (3) Rule (C) precisely chosen, so that $\Phi_j = 0$ before and after ⇝ no change!

⇝ $\Delta\Phi \le -\frac{1}{2}|S_1| + \ell = \Theta(-|S_1| + \log n)$

⇝ **amortized cost $O(\log n)$**



Differences to General Lazy Search Trees

Several things became easier

- No gaps! . . . or really, just one $\rightsquigarrow$ no biased search trees needed
- One 1-sided gap $\rightsquigarrow$ simpler potential
- Query only at left boundary
  - $\rightsquigarrow$ can allow even more laziness: **single partitioning** call upon query

Differences to General Lazy Search Trees

Several things became easier

- No gaps! . . . or really, just one ⇝ no biased search trees needed
- One 1-sided gap ⇝ simpler potential
- Query only at left boundary
  - ⇝ can allow even more laziness: **single partitioning** call upon query

probably among fastest options with true stable pointers

in practice

cannot support fast `meld`

QuickHeaps

The pointer chasing in linked lists is in practice comparatively slow

If we're doing Quicksortus interruptus, why not in one big array?

- Idea predates LP Heaps (but without clear rule for forgetting pivots)

Navarro, Paredes: *On Sorting, Heaps, and Minimum Spanning Trees*, Algorithmica **2010**

QuickHeaps

The pointer chasing in linked lists is in practice comparatively slow

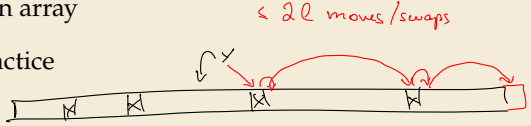
If we're doing Quicksortus interruptus, why not in one big array?

- Idea predates LP Heaps (but without clear rule for forgetting pivots)

> **Navarro, Paredes**: *On Sorting, Heaps, and Minimum Spanning Trees*, Algorithmica **2010**

👍 partitioning substantially faster inplace in array

👍 overall very low overhead and fast in practice



👎 $O(\log n)$ time insert

- we can quickly find *where* to insert, but then need to swap elements to make room

👎 With elements moved, cannot keep stable pointers

3.3 Fibonacci Heaps – Informal

Optimal Heaps

Goal: Theoretically *optimal* PQ

- $O(1)$ time `meld` (LP Heaps and binary heaps don't properly support this at all)
- $O(1)$ time `decreaseKey` instead of $O(\log \log n)$
- *focus on theoretical / asymptotically optimal running time*

Fibonacci Heaps – Informal Overview

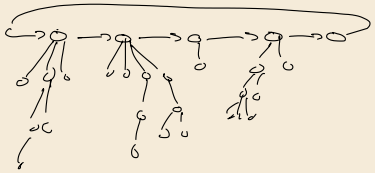
Cormen et al. Intro. to Alg. do all details

The historically first optimal PQ and widely taught . . . good to know key facts

Fibonacci Heaps

Fredman, Tarjan: *Fibonacci Heaps and Their Use in Improved Network Algorithms*, JACM **1987**

- Fibonacci Heap is a list (forest) of "F-trees"
- **F-tree:** general heap-ordered ordinal tree
  - unbounded degree
  - children ordered (first to last)



Fibonacci Heaps – Informal Overview

The historically first optimal PQ and widely taught . . . good to know key facts

Fibonacci Heaps

Fredman, Tarjan: *Fibonacci Heaps and Their Use in Improved Network Algorithms*, JACM **1987**

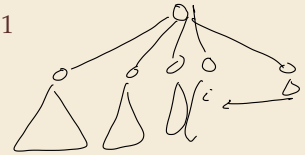
- Fibonacci Heap is a list (forest) of "F-trees"
- **F-tree:** general heap-ordered ordinal tree
  - unbounded degree
  - children ordered (first to last)
  - heap-ordered by priorities (parent $\leq$ children)
  - represented by first-child-next-sibling representation

Fibonacci Heaps – Informal Overview

The historically first optimal PQ and widely taught . . . good to know key facts

Fibonacci Heaps — **Fredman, Tarjan**: *Fibonacci Heaps and Their Use in Improved Network Algorithms*, JACM **1987**

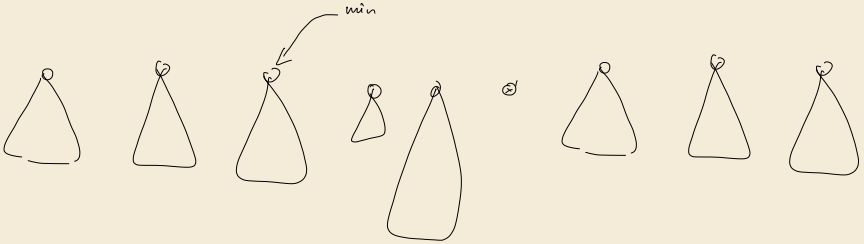
- Fibonacci Heap is a list (forest) of "F-trees"
- **F-tree:** general heap-ordered ordinal tree
  - unbounded degree
  - children ordered (first to last)
  - heap-ordered by priorities (parent ≤ children)
  - represented by first-child-next-sibling representation
- all nodes have a rank $r(v) \in \{\deg(v), \deg(v)+1\}$
  $r(v) = \deg(v)+1$ means "marked" as having lost a child $\rightsquigarrow$ stored in node
- **Invariant:** If c is the ith child **from the right**, $r(c) \geq i-1$



Fibonacci Heaps – Informal Overview [2]

Fibonacci Heap Operations

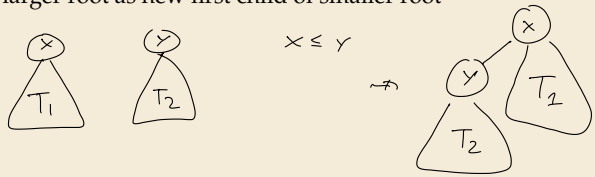
- `findMin`: maintain pointer to min-priority root
- `insert`(x): add a new tree of rank 0 storing x
- `meld`: concatenate forests



Fibonacci Heaps – Informal Overview [2]

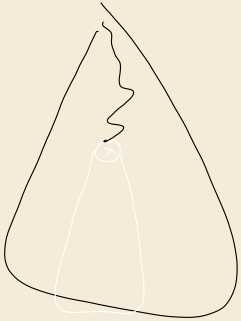
Fibonacci Heap Operations

- `findMin`: maintain pointer to min-priority root
- `insert`(x): add a new tree of rank 0 storing x
- `meld`: concatenate forests
- `delMin`:
  - Remove min root, add its children to the forest
  - **Bucket sort roots by rank** and repeatedly **link** equal-rank roots
  - $\rightsquigarrow$ At most as many roots as max rank
- `link`(T_1, T_2): attach larger root as new first child of smaller root



Fibonacci Heaps – Informal Overview [2]

Fibonacci Heap Operations



- `findMin`: maintain pointer to min-priority root
- `insert`(x): add a new tree of rank 0 storing x
- `meld`: concatenate forests
- `delMin`:
  - Remove min root, add its children to the forest
  - **Bucket sort roots by rank** and repeatedly **link** equal-rank roots
  - $\rightsquigarrow$ At most as many roots as max rank
- `link`(T_1, T_2): attach larger root as new first child of smaller root
- `decreaseKey`(v):
  - Cut v from parent, update priority, add as new tree in forest
  - If parent p now has $r(p) - 2$ children, **recursively cut parent**

Fibonacci Heaps – Informal Overview [3]

Vital tool for analysis:

Rank Lemma: The largest rank of a node in an F-tree is $O(\log n)$.

- Inductive proof
- use that v has ≥ 2 children of rank $\geq r(v) - 3$

$\rightsquigarrow$ subtree size $\geq 2^{\lfloor r/3 \rfloor}$

- (precise counts involve Fibonacci numbers, hence the name)

Fibonacci Heaps – Informal Overview [3]

Vital tool for analysis:

Rank Lemma: The largest rank of a node in an F-tree is $O(\log n)$.

- Inductive proof
- use that v has ≥ 2 children of rank $\geq r(v) - 3$

$\rightsquigarrow$ subtree size $\geq 2^{\lfloor r/3 \rfloor}$

- (precise counts involve Fibonacci numbers, hence the name)

Amortized analysis

- potential $\Phi = 2 \cdot \#\text{marks} + \#\text{roots}$
- —*details skipped*—

Fibonacci Heaps – Informal Overview [3]

Vital tool for analysis:

Rank Lemma: The largest rank of a node in an F-tree is $O(\log n)$.

- Inductive proof
- use that v has ≥ 2 children of rank $\geq r(v) - 3$

$\rightsquigarrow$ subtree size $\geq 2^{\lfloor r/3 \rfloor}$

- (precise counts involve Fibonacci numbers, hence the name)

Amortized analysis

- potential $\Phi = 2 \cdot \#\text{marks} + \#\text{roots}$
- —*details skipped*—

Considerable complications (recursive cut, marking of nodes, rank analysis)

3.4 Quake Heaps

Quake Heaps

Goal: **Simple** theoretically optimal PQ

- Same order of growth for running time as Fibonacci Heaps
- but ideally simpler operations and analysis

⇝ *focus on conceptual simplicity, OK to compromise on practical efficiency*

Quake Heaps

Goal: **Simple** theoretically optimal PQ

- Same order of growth for running time as Fibonacci Heaps
- but ideally simpler operations and analysis

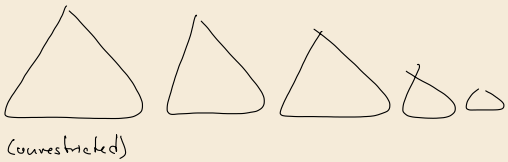
⇝ *focus on conceptual simplicity, OK to compromise on practical efficiency*

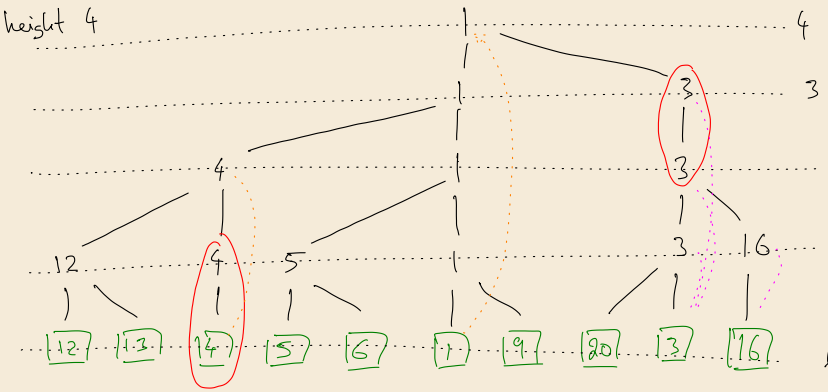
- subtle assumption (shared with Fibonacci heaps):
  - Need arrays (for bucket sort by rank)
  - ⇝ *word-RAM* data structure
    (not a more restrictive model of computation such as *pointer machines*)

Quake Heaps – Structure

A ***Quake Heap*** consists of

- collection of (unrestricted) **tournament trees**
  - all data in leaves
  - internal nodes allowed to *binary* or *unary*
  - no fixed shape constraint





every internal node has
- 1 equal-value child
- ≤ 1 other child
- level
- pointer to its leaf

leaves store highest internal node of that value

level 0 $n_0 = 10$

Quake heap stores

linked list of roots of tournament trees by height / level

store n_i for entire QH

O(1)-time meld:

We store the n_i as linked list of arrays (index=level),
where the true values are obtained by summing entries with the same index.

Upon meld, simply concatenate these lists. (O(1))

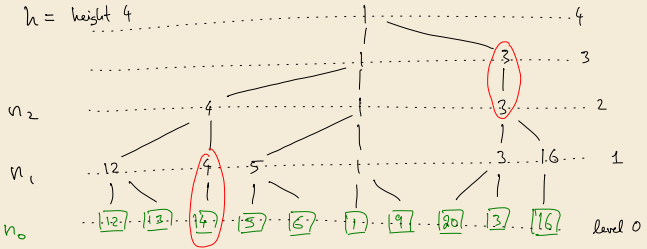
Upon deleteMin, after Consolidate, combine these counters into single array of counts.
Actual cost O(∂C) where ∂C is the number of deleted counters.

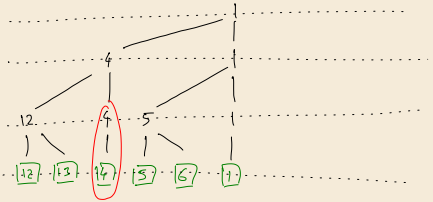
We add to Phi a term +C, which is unchanged in all operations, grows up O(log n) in Consolidate
and drops by ∂C when combining the counters, paying for the sums.

Invariant

Notation:

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$





Invariant

Notation:

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$
- u_i = **#unary** nodes on level i $\quad (u_0 = 0)$
- $U = u_1 + \cdots + u_h$

Invariant

Notation:

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $\widehat{N} = n_0 + n_1 + \cdots + n_h$
- u_i = **#unary** nodes on level i $\quad (u_0 = 0)$
- $\widehat{U} = u_1 + \cdots + u_h$
- $\widehat{T}$ = **#trees** in Quake Heap

Invariant

Notation:

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$
- u_i = **#unary** nodes on level i $\quad (u_0 = 0)$
- $U = u_1 + \cdots + u_h$
- T = **#trees** in Quake Heap

Quake Heap Invariant: $\forall i \geq 1 \; : \; n_i \leq \frac{2}{3} n_{i-1} \qquad (Q)$

(at most half of nodes on each level are unary)

Invariant

Notation:

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$ $\qquad n = n_0 = \#\text{keys}$
- u_i = **#unary** nodes on level i $\quad (u_0 = 0)$
- $U = u_1 + \cdots + u_h$
- T = **#trees** in Quake Heap

Quake Heap Invariant: $\forall i \geq 1 \; : \; n_i \leq \frac{2}{3} n_{i-1} \qquad (Q)$

Corollary 3.1 (Height Bound)

In any Quake Heap, we have $h \leq 2 \lg n$.

Proof:

We have $n_i \leq \left(\frac{2}{3}\right)^i n \leq \left(\frac{4}{9}\right)^{i/2} n < \left(\frac{1}{2}\right)^{i/2} n$.

Quake Heaps – Operations

actual cost

Insert(x): add x (new leaf) as new tournament tree — $O(1)$

(Q)✓

Meld(Q_1, Q_2): just concatenate lists of tournament trees — $O(1)$

update root lists and n_i: $O(h) = O(\log n)$ (Q)✓

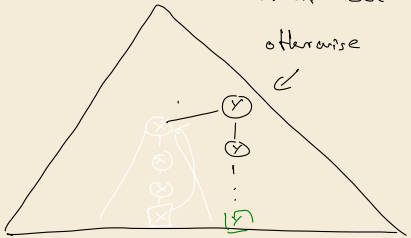
DecreaseKey(ptr, x'): ptr points to leaf and thus get to highest internal node of same value

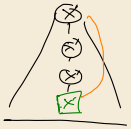
if at root: update x to x' (internal nodes only point to leaf; don't store x)

$O(1)$

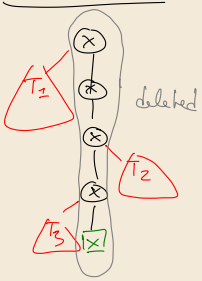
otherwise

(Q)✓





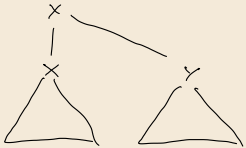
DeleteMin()



(1) Find root w/ smallest priority x $O(T)$

(2) Remove x and its internal nodes
add other children $T_1, \ldots$ as
new trees to QH
$O(h) = O(\log n)$

(3) CONSOLIDATE
Bucket-sort roots by height $O(T)$
While there 2 equal-height trees, join them
// ≤ 1 tree per height



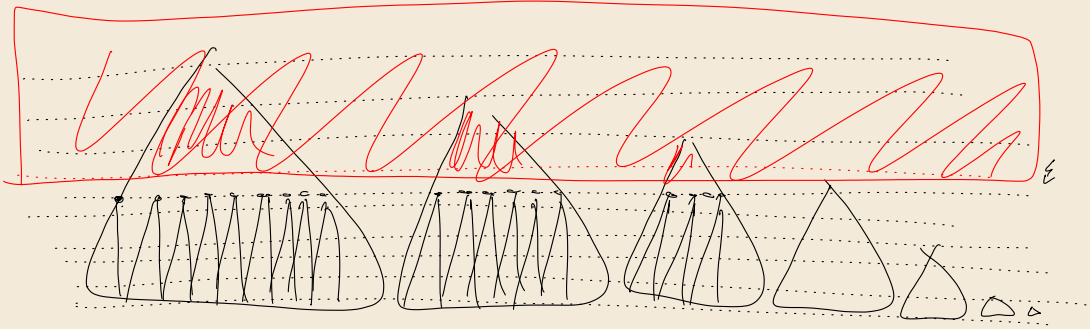
(4) Check invariant (Q) $O(h) = O(\log n)$
Pick lowest level i where $n_i > \frac{2}{3} n_{i-1}$
QUAKE : Remove all nodes at levels $\geq i$ $O(n_{\geq i})$
(⇒ insert nodes on level $i-1$ into root list)

$n_{\geq i} = n_i + n_{i+1} + \cdots + n_h$

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$
- u_i = **#unary** nodes on level i $(u_0 = 0)$
- $U = u_1 + \cdots + u_h$
- T = **#trees** in Quake Heap

Quake Heap Invariant: $\forall i \geq 1 : n_i \leq \frac{2}{3} n_{i-1}$ (Q)

total cost
$O(T + \log n + n_{\geq i})$



3.5 Quake Heaps Analysis

Amortized Analysis

Theorem 3.2 (Amortized Cost of Quake Heaps)

In a Quake Heap, `insert`, `decreaseKey`, and `meld` have $O(1)$ worst-case cost and `deleteMin` has $O(\log n)$ amortized cost. ◀

Amortized Analysis

Theorem 3.2 (Amortized Cost of Quake Heaps)

In a Quake Heap, `insert`, `decreaseKey`, and `meld` have $O(1)$ worst-case cost and `deleteMin` has $O(\log n)$ amortized cost. ◀

Quake Potential: $\boxed{\Phi = N + 2T + 6U}$

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$
- u_i = **#unary** nodes on level i $\quad (u_0 = 0)$
- $U = u_1 + \cdots + u_h$
- T = **#trees** in Quake Heap

Quake Heap Invariant: $\forall i \geq 1 \; : \; n_i \leq \frac{3}{5} n_{i-1} \qquad (Q)$

Amortized Analysis

Theorem 3.2 (Amortized Cost of Quake Heaps)

In a Quake Heap, `insert`, `decreaseKey`, and `meld` have $O(1)$ worst-case cost and `deleteMin` has $O(\log n)$ amortized cost.

Quake Potential: $\boxed{\Phi = N + 2T + 6U}$

r_i = # roots at level i

- Denote **levels** $0, 1, \ldots, h$, from bottom to top
- n_i = **#nodes** on level i (across all tournament trees)
- **Height** $h = \max\{i : n_i > 0\}$
- $N = n_0 + n_1 + \cdots + n_h$
- u_i = **#unary** nodes on level i $(u_0 = 0)$
- $U = u_1 + \cdots + u_h$
- T = **#trees** in Quake Heap

Quake Heap Invariant: $\forall i \geq 1 : n_i \leq \frac{2}{3} n_{i-1}$ (Q)

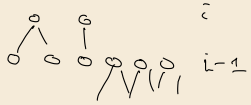
To prove the theorem, we need one lemma

Lemma 3.3 (Unary node lemma)

If $n_i > \frac{2}{3} n_{i-1}$, then $u_i > \frac{1}{3} n_{i-1}$. $i \geq 1$

invariant violated then remove many unary nodes

Proof: $n_i = b_i + u_i$ $\quad n_{i-1} = 2b_i + u_i + r_{i-1}$ (*)



By assumption $n_i > \frac{2}{3} n_{i-1} \overset{(*)}{=} \frac{4}{3} b_i + \frac{2}{3} u_i + \frac{2}{3} r_{i-1} \; | - \frac{2}{3} u_i - b_i$

$b_i + u_i$

$\frac{1}{3} u_i > \frac{1}{3} b_i + \frac{2}{3} r_{i-1}$

$$\Rightarrow \quad u_i > b_i + 2r_{i-1} = \frac{1}{2}n_{i-1} - \frac{1}{2}u_i + \underbrace{\frac{3}{2}r_{i-1}}_{\geq 0} \geq \frac{1}{2}n_{i-1} - \frac{1}{2}u_i$$

$$\text{(def)} \quad \frac{1}{2}n_{i-1} - \frac{1}{2}u_i - \frac{1}{2}r_{i-1}$$

$$\Rightarrow \quad u_i > \frac{1}{3}n_{i-1} \qquad \square$$

Amortized Analysis [2]

actual cost

insert $O(1)$

decreaseKey $O(1)$

meld $O(\log n)$

delMin w/o Quake $O(T + \log n)$

Quake $O(n_{\geq c})$

Change in potential $\Phi = N + 2T + 6U$

Insert: $\Delta\Phi = 1 + 2 = O(1)$

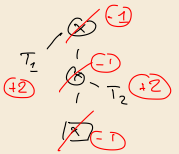
Meld: $\Delta\Phi = 0$

DecreaseKey $\Delta\Phi \leq 2 + 6 = O(1)$

DeleteMin (1) $\Delta\Phi = 0$

(2) $\Delta\Phi \leq h + 1 = O(\log n)$

(3) $\Delta U = 0$



(1) Find root w/ smallest priority x

(2) Remove x and its internal nodes

add other children $T_1, \ldots$ as new trees to QH

$O(h) = O(\log n)$

(3) CONSOLIDATE

Bucket-sort roots by height $O(T)$

While there 2 equal-height trees, join them

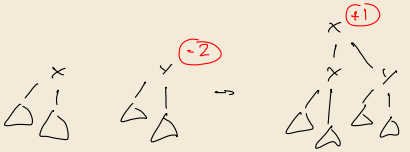
$\#$ ≤ 1 tree per height

(4) Check invariant (Q) $O(h) = O(\log n)$

Pick lowest level i where $n_i > \frac{2}{3} n_{i-1}$

QUAKE: Remove all nodes at levels $\geq i$

(⇒ insert nodes on level $i-1$ into root list)



$\Delta\Phi \leq T' - T \leq O(\log n) - T$

T': $\leq$ one per height

$O(\log n)$: pays for consolidate

(4) QUAKE at level i

$\Delta N = -n_{\geq i}$

$\Delta T \leq n_{i-1}$

$\Delta U \leq -u_i < -\frac{1}{3} n_{i-1}$ (Lem 3.3)

$\Delta\Phi \leq -n_{\geq i} + 2n_{i-1} - 2n_{i-1}$

Total amortized cost $O(\log n)$

Quake Heap – Space Usage

Lemma 3.4

A Quake Heap storing n keys uses $O(n)$ total space.

◀

$N = O(n)$ in particular $U = O(n)$

Discussion

- Quake Heaps support all operations (meld) in the best possible amortized time
- Reasonable easy invariants and structure
- tournament tree representation uses a lot of pointers