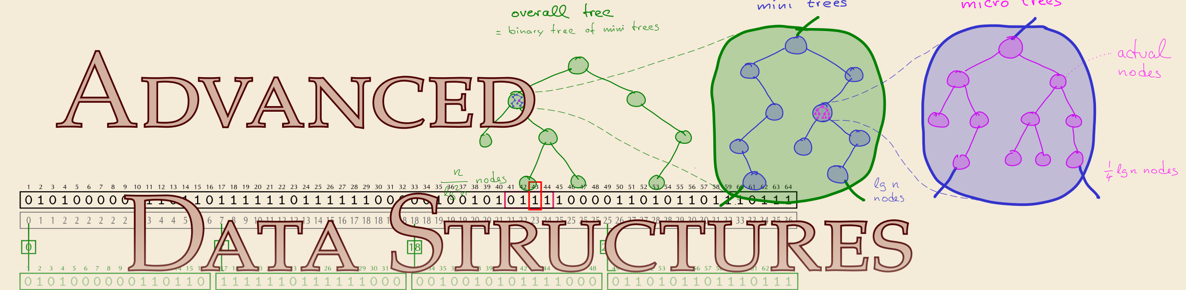


Advanced Data Structures

3 Efficient Priority Queues

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg

version `2026-05-25 15:17` *H*

Outline

3 Efficient Priority Queues

3.1 Tournament Trees
3.2 Lazy Partition Heaps
3.3 Fibonacci Heaps – Informal
3.4 Quake Heaps

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 $p' \le p$.
- `meld`(Q_1, Q_2)
  Build a PQ containing the union of Q_1 and Q_2

Fundamental building block in many applications.

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`)`	~~$O(n)$~~ $O(1)$

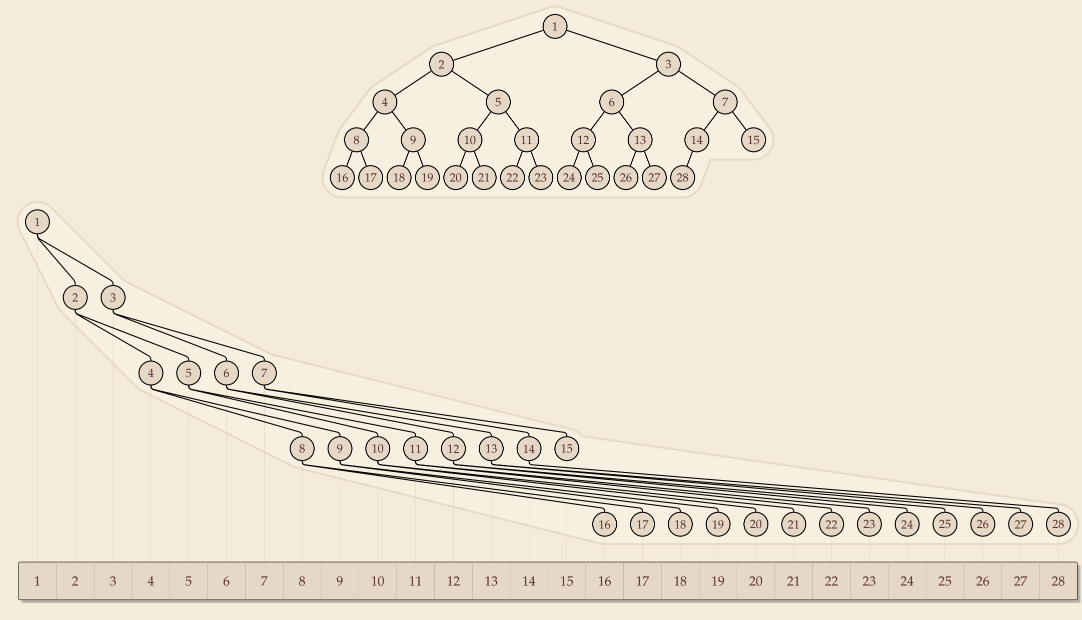
- apart from faster `construct`,
  BSTs always as good as binary heaps
- PQ abstraction still helpful
- faster heaps exist!

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 $T_1 \le T_2$)	$O(\log n)$

3.1 Tournament Trees

Implicit Complete Binary Trees



Tournament Trees = Leaf-Oriented Heaps

Who Needs Rigid Shapes?

(arrays do)

3.2 Lazy Partition Heaps

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.

- Present here as a self-contained data structure

Brodal, Iacono, Rysgaard, Wild: *Partition-based Simple Heaps*, LATIN **2026**

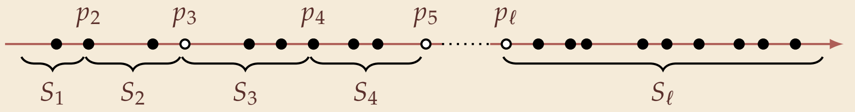
Lazy Partition Heaps

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)

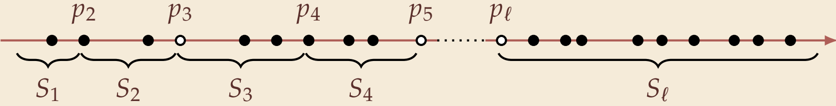
- $S_i \subseteq [p_i, p_{i+1})$

  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

- `Insert`(e):
  Search e among the ℓ pivots ($O(\log \ell)$ time)
  $\rightsquigarrow$ S_i with $p_i \le e < p_{i+1}$
  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

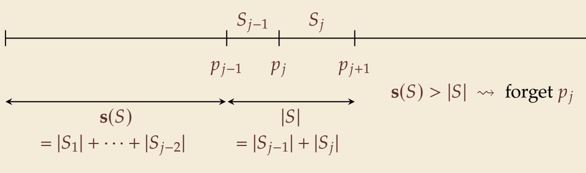
Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

skipping empty sets

- `DeleteMin` always **partitions** smallest set S_1 around **median**
- **Invariant (S):** Number of sets $\ell \le 2 \lg n + 1$
  - 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

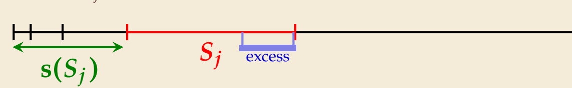
- **Actual Costs**
  - `Insert` $O(\log\log n)$
  - `DecreaseKey` $O(\log\log n)$
  - `DeleteMin` $O(|S_1| + \log n)$

    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!

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

$\rightsquigarrow$ **amortized cost $O(\log n)$**

S_1

S_1' S_2' S_2 S_3 S_4 S_5

min

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

👍 probably among fastest options with true stable pointers

👎 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**

partitioning substantially faster inplace in array

overall very low overhead and fast in practice

$O(\log n)$ time insert

- we can quick 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

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

Fibonacci Heaps

- 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 $\le$ 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
- Invariant: If c is the ith child **from the right**, $r(c) \ge i - 1$

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
- `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 using 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, requirement of 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*

- 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

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

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

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

To prove the theorem, we need one lemma

Lemma 3.3 (Unary node lemma)

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

Amortized Analysis [2]

Quake Heap – Space Usage

Lemma 3.4

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

Discussion

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