

# ADVANCED

overall tree  
= binary tree of mini trees

mini trees

micro trees

actual nodes

$\lg n$  nodes

$\frac{1}{7} \lg n$  nodes

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64

010100006 1110111110111111000 0010101011110000110101101 110111

0 1 1 2 2 2 2 2 2 2 3 4 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 18 19 18 19 20 20 20 20 21 21 21 21 22 22 23 23 24 24 25 25 26 26 27 27 28 28 29 29 30 30 31 31 32 32 33 33 34 34 35 35 36 36 37 37 38 38 39 39 40 40 41 41 42 42 43 43 44 44 45 45 46 46 47 47 48 48 49 49 50 50 51 51 52 52 53 53 54 54 55 55 56 56 57 57 58 58 59 59 60 60 61 61 62 62 63 63 64 64

# DATA STRUCTURES

0101000000110110 11111110111111000 0010010101111000 0110101101110111

0 1 1 2 2 2 2 2 2 3 4 4 5 6 6 6 7 7 8 9 10 11 12 12 12 0 0 1 1 1 2 2 3 3 4 5 6 7 7 7 7 0 1 2 2 3 3 4 5 6 7 8 9 10 11

0 1 1 2 2 2 2 2 3 4 4 5 6 6 6 7 7 8 9 10 11 12 12 12 0 0 1 1 1 2 2 3 3 4 5 6 7 7 7 7 0 1 2 2 3 3 4 5 6 7 8 9 10 11

0101 0000 0011 0110 1111 1111 0010 0101 0000 0110 1011 0113 0111

# 3

## Efficient Priority Queues

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

## 3 Efficient Priority Queues

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

# Priority Queue ADT

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

- ▶  $\text{construct}(A)$   
Construct from elements in array  $A$ .
- ▶  $\text{insert}(x, p)$   
Insert item  $x$  with priority  $p$  into PQ.
- ▶  $\text{min}()$   
Return item with smallest priority. (Does not modify the PQ.)
- ▶  $\text{delMin}()$   
Remove the item with smallest priority and return it.
- ▶  $\text{decreaseKey}(x, p')$   
Update  $x$ 's priority to  $p' \leq p$ .
- ▶  $\text{meld}(Q_1, Q_2)$   
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~~ stay tuned

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

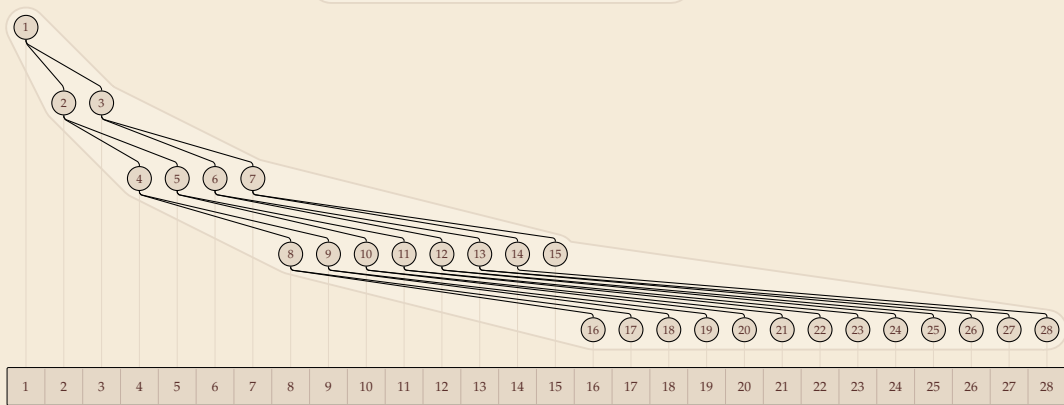
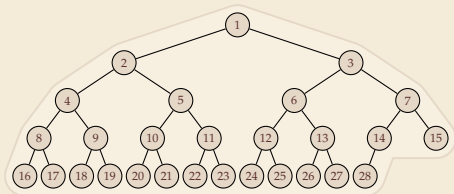
- ▶ apart from faster construct, BSTs always as good as binary heaps
- ▶ PQ abstraction still helpful
- ▶ faster heaps exist!

## 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



# Tournament Trees = Leaf-Oriented Heaps


# Who Needs Rigid Shapes?

(arrays do)

## 3.2 Lazy-Partition Heaps

# Recall Unit 2

## 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

- ▶  $\text{Insert}(x)$  in  $O(\min(\log(n/|\Delta_i|) + \log \log |\Delta_i|, \log q))$  worst-case time, where  $x \in \Delta_i$ .
- ▶  $\text{RankBasedQuery}(r)$  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  $O(\mathcal{B} + \min(n \log \log n, n \log q))$ .
- ▶  $\text{Construction}(S)$  in  $O(n)$  worst-case time, where  $|S| = n$ .
- ▶  $\text{Delete}(\text{ptr})$  in  $O(\log n)$  worst-case time.
- ▶  $\text{ChangeKey}(\text{ptr}, x')$  in  $O(\min(\log q, \log \log |\Delta_i|))$  worst-case time, where the element at  $\text{ptr}$ , is  $x \in \Delta_i$  and  $x'$  lies to its closest query rank in  $\Delta_i$ ; otherwise, takes  $O(\log n)$  worst-case time.
- ▶  $\text{Split}(r)$  in time according to  $\text{RankBasedQuery}(r)$ .
- ▶  $\text{Merge}(T_1, T_2)$  (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

# 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



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

# Lazy Partition Heaps

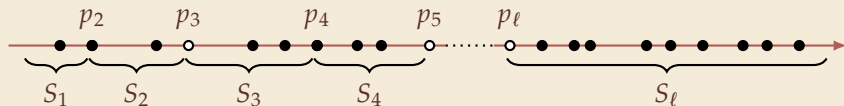


# Partition-Based Heaps Template

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)

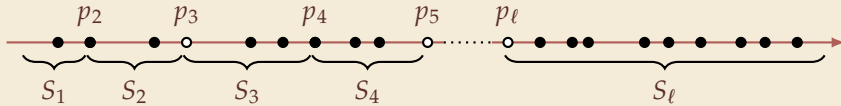
- ▶  $S_i \subseteq [p_i, p_{i+1})$

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

- ▶ 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  $\ell$  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  $S_1$  around **median**

skipping empty sets

▶ **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)

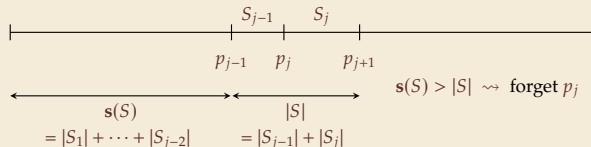
▶ **Lazy Concatenation rule (C):** Forget pivot  $p_j$  if  $|S_j| + |S_{j+1}| < s(S_j)$

▶ **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

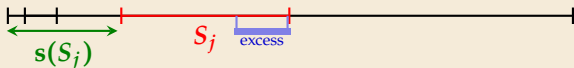
## ▶ Actual Costs

- ▶ Insert  $O(\log \log n)$
- ▶ DecreaseKey  $O(\log \log n)$
- ▶ DeleteMin  $O(|S_1| + \log n)$   
↖ could be  $n$

↪ 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

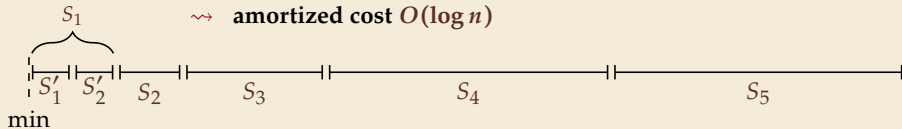


## ▶ Change in Potential

- ▶ Insert Only  $\Phi_j$  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  $s(S_j)$  of each set
  - (3) Rule (C) precisely chosen, so that  $\Phi_j = 0$  before and after  $\rightsquigarrow$  no change!

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


$\rightsquigarrow$  **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

 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 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

*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
- ▶ 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  $i$ th 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
- ▶ `delMin`:
  - ▶ Remove min root, add its children to the forest
  - ▶ **Bucket sort roots by rank** and repeatedly **link** equal-rank roots
  - ↪ 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  $\geq 2$  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*

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

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

## 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. ◀

**Quake Potential:**  $\Phi = N + 2T + 6U$

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