Partition-based Simple Heaps

Sebastian Wild

www.wild-inter.net

based on joint work with
Casper Rysgaard, Gerth Stølting Brodal, John Iacono



Marburg University

LATIN 2026

Florianópolis

Our Data Structure



Our Result



teachable analysis

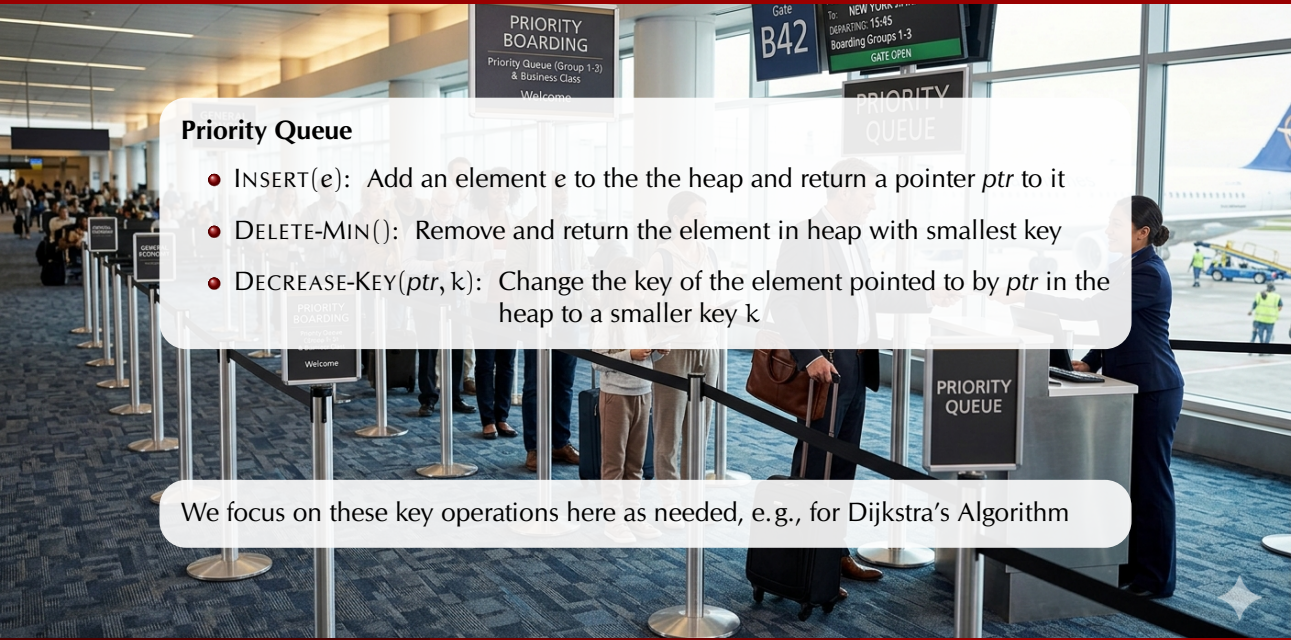
only lists and partition

Simple, entirely elementary efficient **priority queue** (*Lazy Partition Heaps*).

- INSERT and DECREASE-KEY in $\mathbf{O(\log \log n)}$ time
- DELETE-MIN in $O(\log n)$ amortized time

(Plus alternatives and extensions, see paper)

Priority Queues a.k.a. Heaps



Priority Queue

- INSERT(e): Add an element e to the the heap and return a pointer *ptr* to it
- DELETE-MIN(): Remove and return the element in heap with smallest key
- DECREASE-KEY(ptr, k): Change the key of the element pointed to by *ptr* in the heap to a smaller key k

We focus on these key operations here as needed, e.g., for Dijkstra's Algorithm

Priority Queue Implementations

- original **binary heap** by Williams 1964, as part of Heapsort
  - implicit heap-ordered binary tree
  - part of standard intro courses to algorithms and data structures
  - all operations in $O(\log n)$ time

  ⇝ same complexity as using a binary search tree 🤔

Priority Queue Implementations

- original **binary heap** by Williams 1964, as part of Heapsort
  - implicit heap-ordered binary tree
  - part of standard intro courses to algorithms and data structures
  - all operations in $O(\log n)$ time
  - ⇝ same complexity as using a binary search tree 🤔
- But we can do (much) better:
  (Strict) Fibonacci heaps achieve $O(1)$ time for all operations but DeleteMin
  (inevitable by sorting lower bound)
  - theoretical breakthroughs
  - but rather complicated to analyze
  - often slower in practice due to caching & constant-factor overhead

Priority Queue Implementations

- original **binary heap** by Williams 1964, as part of Heapsort
  - implicit heap-ordered binary tree
  - part of standard intro courses to algorithms and data structures
  - all operations in $O(\log n)$ time
  - ⇝ same complexity as using a binary search tree 🤔
- But we can do (much) better:
  (Strict) Fibonacci heaps achieve $O(1)$ time for all operations but DeleteMin
  (inevitable by sorting lower bound)
  - theoretical breakthroughs
  - but rather complicated to analyze
  - often slower in practice due to caching & constant-factor overhead
- **Can't we have a simple efficient PQ to teach?**
  - various prior attempts with partial success (discussed in paper)
  - our verdict: *all retain somewhat complicated structures and invariants, are nontrivial to analyze, or require $\Omega(\log n)$ time for updates*

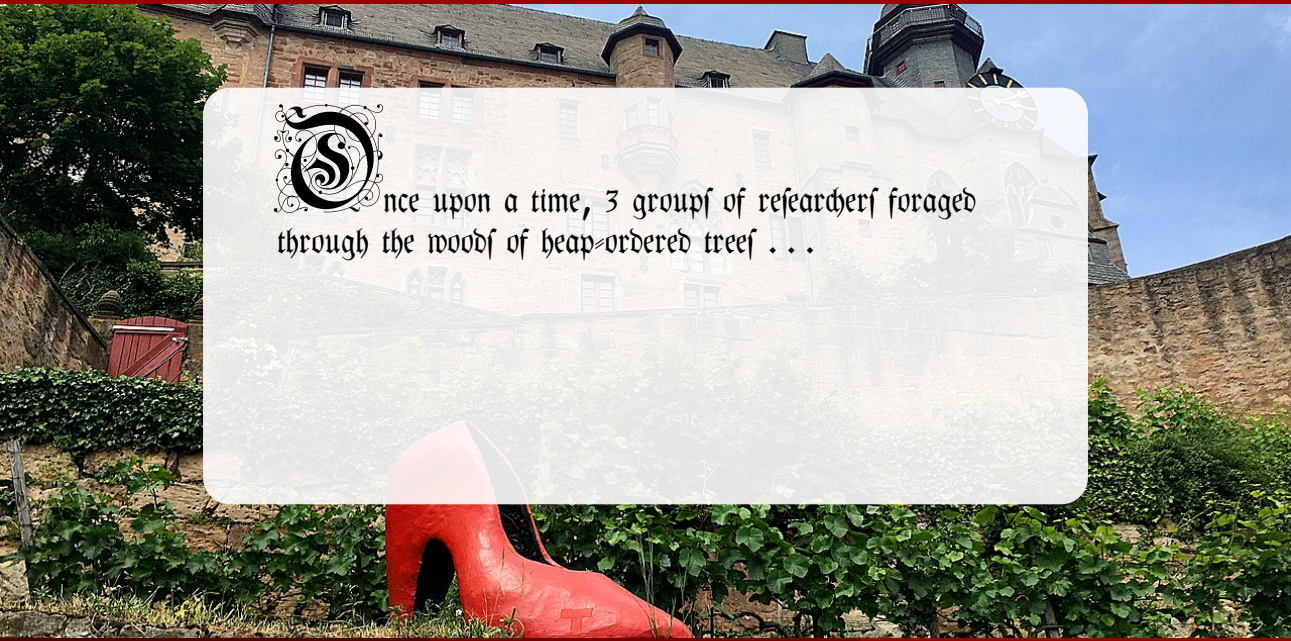
Priority Queue Implementations

- original **binary heap** by Williams 1964, as part of Heapsort
  - implicit heap-ordered binary tree
  - part of standard intro courses to algorithms and data structures
  - all operations in $O(\log n)$ time
  - ⇝ same complexity as using a binary search tree 🤔
- But we can do (much) better:
  (Strict) Fibonacci heaps achieve $O(1)$ time for all operations but DeleteMin
  (inevitable by sorting lower bound)
  - theoretical breakthroughs
  - but rather complicated to analyze
  - often slower in practice due to caching & constant-factor overhead
- **Can't we have a simple efficient PQ to teach?**
  - various prior attempts with partial success (discussed in paper)
  - our verdict: *all retain somewhat complicated structures and invariants, are nontrivial to analyze, or require $\Omega(\log n)$ time for updates* } Is that necessary??

The Time Was Ripe . . .

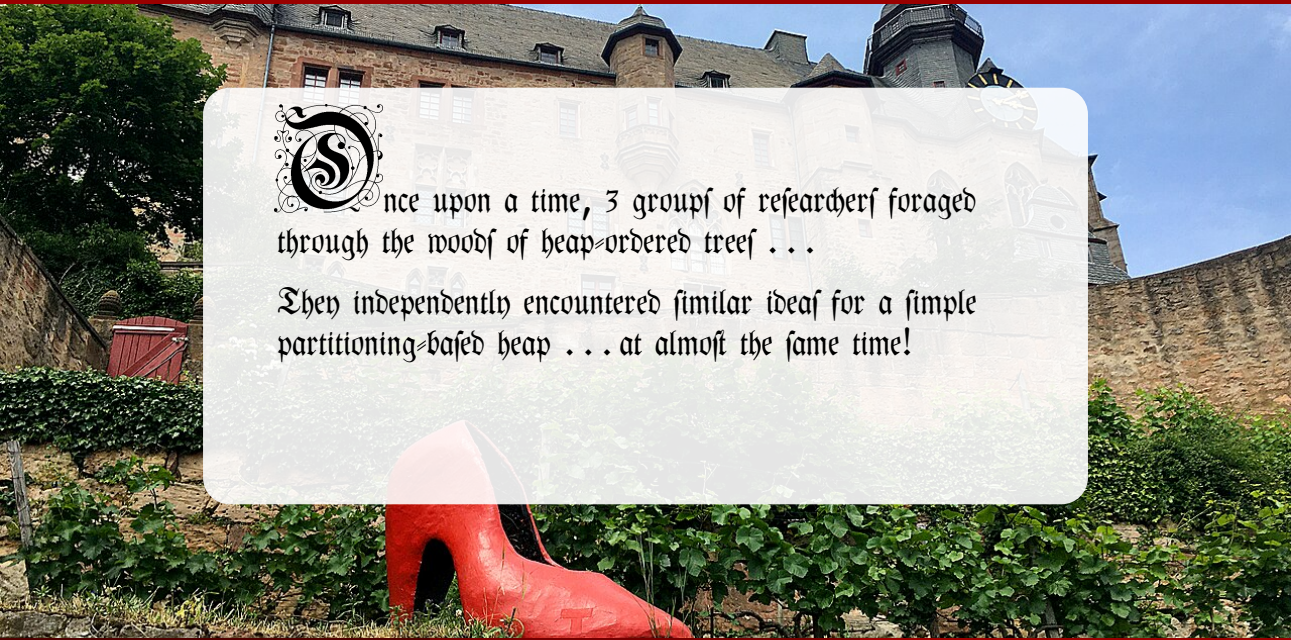


The Time Was Ripe . . .



Once upon a time, 3 groups of researchers foraged through the woods of heap-ordered trees . . .

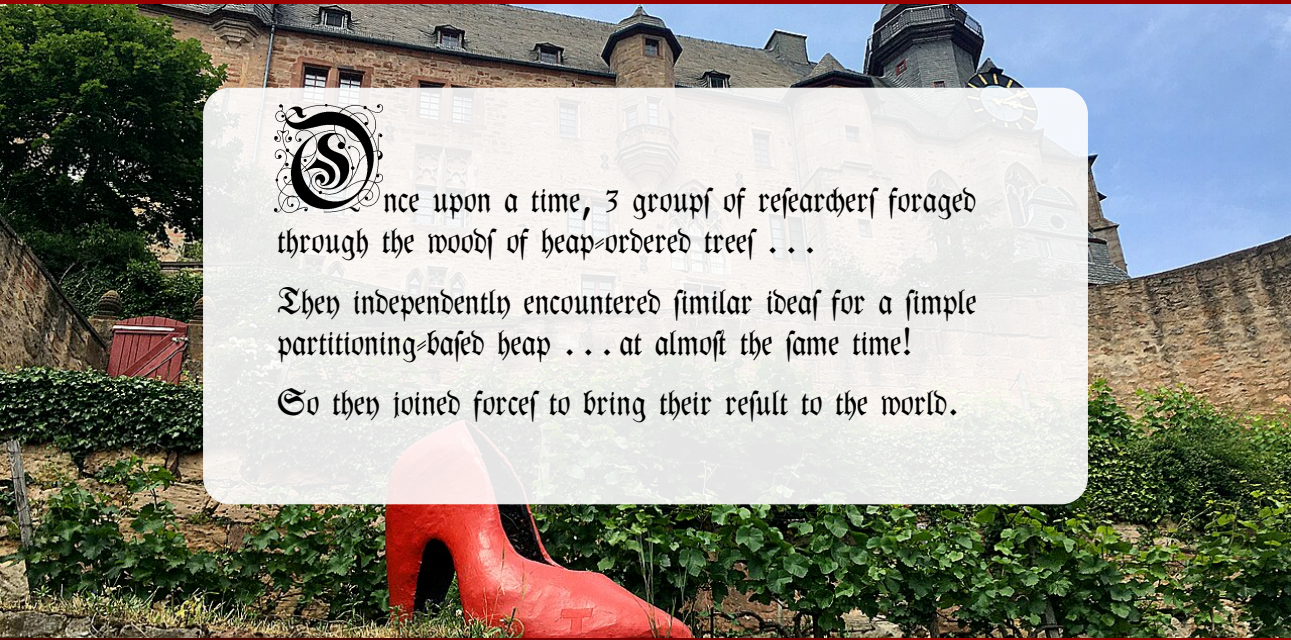
The Time Was Ripe . . .



Once upon a time, 3 groups of researchers foraged through the woods of heap-ordered trees . . .

They independently encountered similar ideas for a simple partitioning-based heap . . . at almost the same time!

The Time Was Ripe . . .

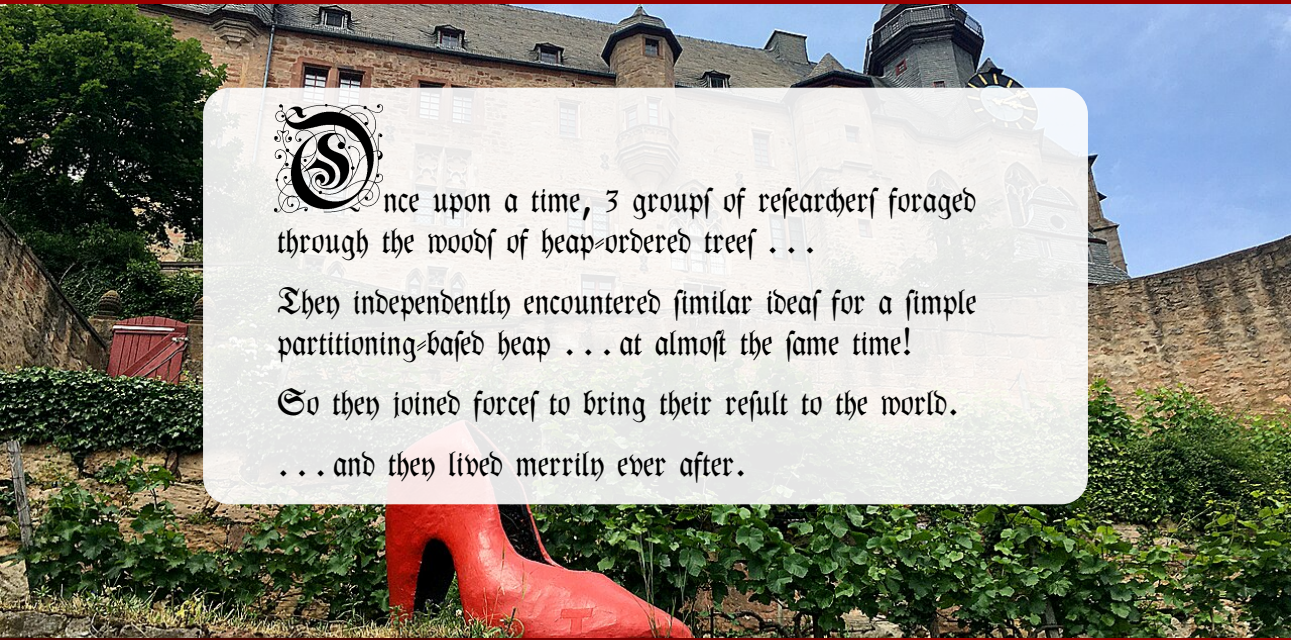


Once upon a time, 3 groups of researchers foraged through the woods of heap-ordered trees . . .

They independently encountered similar ideas for a simple partitioning-based heap . . . at almost the same time!

So they joined forces to bring their result to the world.

The Time Was Ripe . . .



Once upon a time, 3 groups of researchers foraged through the woods of heap-ordered trees . . .

They independently encountered similar ideas for a simple partitioning-based heap . . . at almost the same time!

So they joined forces to bring their result to the world.

. . . and they lived merrily ever after.

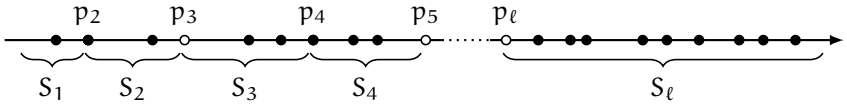
Partition-Based Heaps

A **partition-based heap** is a *“partially completed Quicksort”*

Partition-Based Heaps

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

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

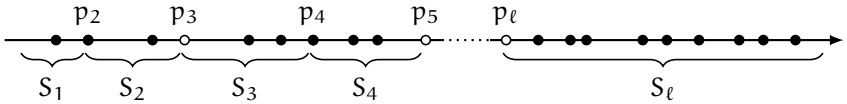


Pivots can be elements in the heap (black) or elements removed from the heap (white)

Partition-Based Heaps

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)

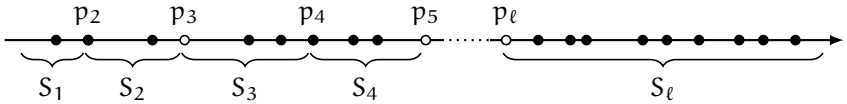
- $\boxed{S_i \subseteq [p_i, p_{i+1})}$

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

Partition-Based Heaps

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)

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

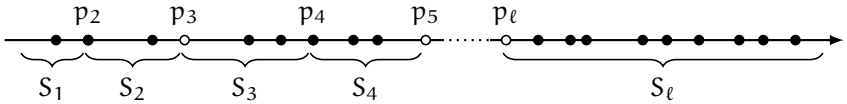
  Pivots $p_2 \ldots, p_\ell$ partition the n elements into ℓ sets $S_1, S_2, \ldots, S_\ell$

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

Partition-Based Heaps

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)

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

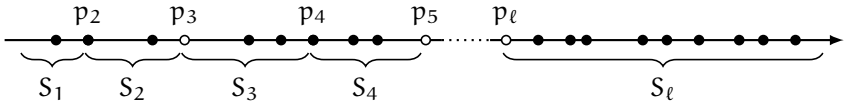
  Pivots $p_2 \ldots, p_\ell$ partition the n elements into ℓ sets $S_1, S_2, \ldots, S_\ell$

- 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

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)

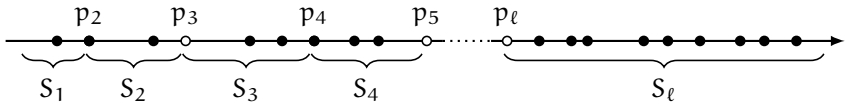
- $\boxed{S_i \subseteq [p_i, p_{i+1})}$

  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!

Operations



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

Lazy Partition Heaps

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

Lazy Partition (LP) Heaps

Lazy Partition Heaps

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

Lazy Partition (LP) Heaps

- DELETE-MIN always **partitions** smallest set S_1 around **median**

skipping empty sets

Lazy Partition Heaps

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

Lazy Partition (LP) Heaps

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

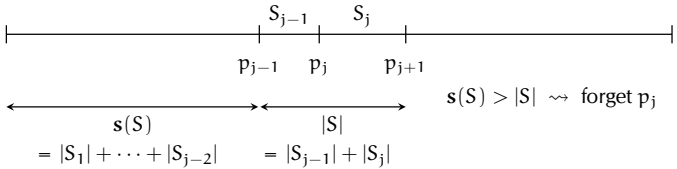
Lazy Partition Heaps

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

Lazy Partition (LP) Heaps

skipping empty sets

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



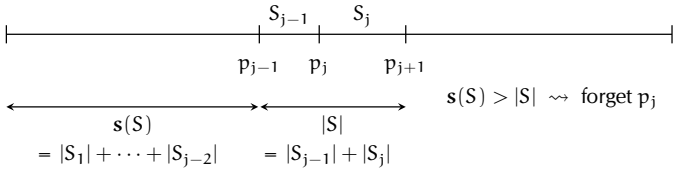
Lazy Partition Heaps

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

Lazy Partition (LP) Heaps

skipping empty sets

- DELETE-MIN always **partitions** smallest set S_1 around **median**
- **Invariant (S):** Number of sets $\ell \leqslant 2 \lg n + 1$
  - only at risk upon DELETE-MIN $\rightsquigarrow$ 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 DELETE-MIN
  - Forgetting a pivot means **concatenating** the linked lists of S_j and S_{j+1}



Lazy Partition Heaps

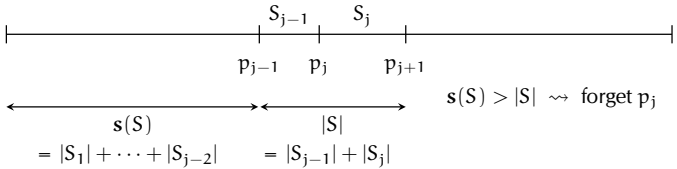
Focus on our arguably simplest instantiation of partition-based Heaps:

Lazy Partition (LP) Heaps

skipping empty sets

- DELETE-MIN always **partitions** smallest set S_1 around **median**
- **Invariant (S):** Number of sets $\ell \leqslant 2 \lg n + 1$
  - only at risk upon DELETE-MIN $\rightsquigarrow$ 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 DELETE-MIN
  - Forgetting a pivot means **concatenating** the linked lists of S_j and S_{j+1}

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



LP Heap – Analysis

- **Actual Costs**
  - INSERT $O(\log \log n)$
  - DECREASE-KEY $O(\log \log n)$
  - DELETE-MIN $O(|S_1| + \log n)$



LP Heap – Analysis

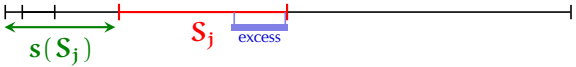
- **Actual Costs**
  - INSERT $O(\log \log n)$
  - DECREASE-KEY $O(\log \log n)$
  - DELETE-MIN $O(|S_1| + \log n)$

    could be n

$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

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



LP Heap – Analysis

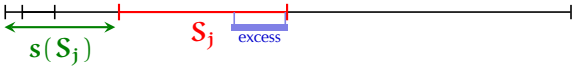
- **Actual Costs**
  - INSERT $O(\log \log n)$
  - DECREASE-KEY $O(\log \log n)$
  - DELETE-MIN $O(|S_1| + \log n)$

    could be n

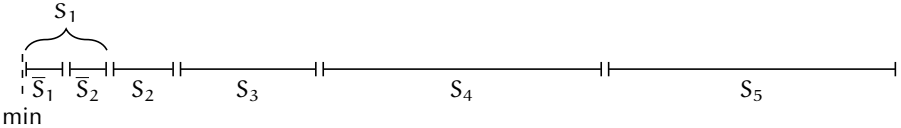
$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

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



- **Change in Potential**
  - INSERT Only φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$



LP Heap – Analysis

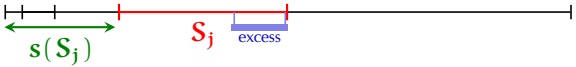
- **Actual Costs**
  - Insert $O(\log \log n)$
  - Decrease-Key $O(\log \log n)$
  - Delete-Min $O(|S_1| + \log n)$

  could be n

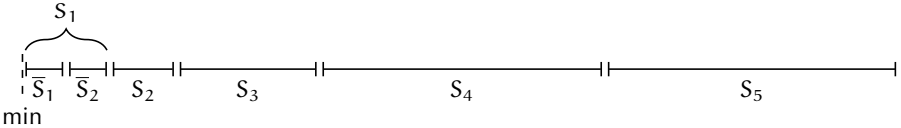
$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

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



- **Change in Potential**
  - Insert Only φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - Decrease-Key $\Delta\Phi \leqslant 0$ since we only increase "protection distance"



LP Heap – Analysis

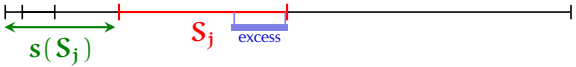
- **Actual Costs**
  - Insert $O(\log \log n)$
  - Decrease-Key $O(\log \log n)$
  - Delete-Min $O(|S_1| + \log n)$

  could be n

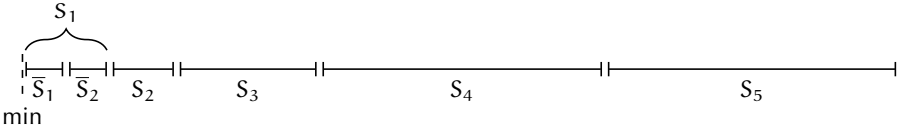
$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

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



- **Change in Potential**
  - Insert Only φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - Decrease-Key $\Delta\Phi \leqslant 0$ since we only increase "protection distance"
  - Delete-Min (1) Partitioning S_1 into $\overline{S}_1$ and $\overline{S}_2$ eliminates $\varphi_1 = |S_1|$ and adds $\overline{\varphi_1} + \overline{\varphi_2} = \frac{1}{2}|S_1|$



LP Heap – Analysis

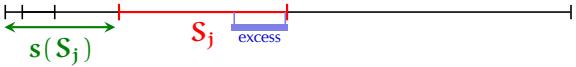
- **Actual Costs**
  - INSERT $O(\log \log n)$
  - DECREASE-KEY $O(\log \log n)$
  - DELETE-MIN $O(|S_1| + \log n)$

  could be n

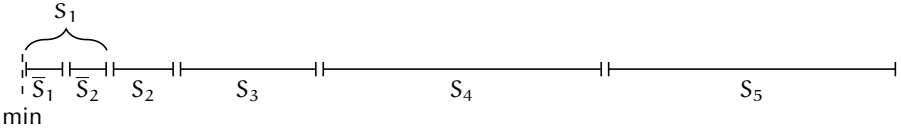
$\rightsquigarrow$ Introduce **potential**

excess size over distance from min

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



- **Change in Potential**
  - INSERT Only φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - DECREASE-KEY $\Delta\Phi \leqslant 0$ since we only increase "protection distance"
  - DELETE-MIN (1) Partitioning S_1 into $\overline{S}_1$ and $\overline{S}_2$ eliminates $\varphi_1 = |S_1|$ and adds $\overline{\varphi_1} + \overline{\varphi_2} = \frac{1}{2}|S_1|$
    (2) Moreover, deleting the min removes 1 from $s(S_j)$ of each set



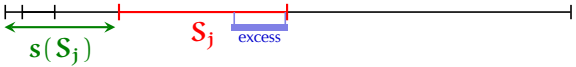
LP Heap – Analysis

- **Actual Costs**
  - Insert $O(\log \log n)$
  - Decrease-Key $O(\log \log n)$
  - Delete-Min $O(|S_1| + \log n)$ — could be n

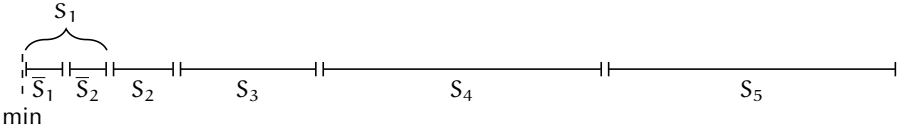
$\rightsquigarrow$ Introduce **potential**

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

excess size over distance from min



- **Change in Potential**
  - Insert Only φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - Decrease-Key $\Delta\Phi \leqslant 0$ since we only increase "protection distance"
  - Delete-Min (1) Partitioning S_1 into $\overline{S}_1$ and $\overline{S}_2$ eliminates $\varphi_1 = |S_1|$ and adds $\overline{\varphi_1} + \overline{\varphi_2} = \frac{1}{2}|S_1|$
    (2) Moreover, deleting the min removes 1 from $s(S_j)$ of each set
    (3) Rule (C) is precisely chosen, so that $\varphi_j = 0$ before and after $\rightsquigarrow$ no change!



LP Heap – Analysis

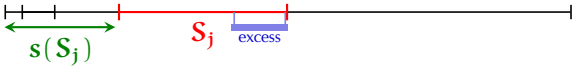
- **Actual Costs**
  - Insert $O(\log \log n)$
  - Decrease-Key $O(\log \log n)$
  - Delete-Min $O(|S_1| + \log n)$

  could be n

$\rightsquigarrow$ Introduce **potential**

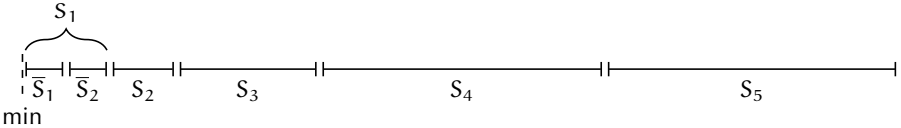
excess size over distance from min

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



- **Change in Potential**
  - Insert Only φ_j increases (by 1) $\rightsquigarrow$ $\Delta\Phi = O(1)$
  - Decrease-Key $\Delta\Phi \leqslant 0$ since we only increase "protection distance"
  - Delete-Min (1) Partitioning S_1 into $\overline{S}_1$ and $\overline{S}_2$ eliminates $\varphi_1 = |S_1|$ and adds $\overline{\varphi_1} + \overline{\varphi_2} = \frac{1}{2}|S_1|$
    (2) Moreover, deleting the min removes 1 from $s(S_j)$ of each set
    (3) Rule (C) is precisely chosen, so that $\varphi_j = 0$ before and after $\rightsquigarrow$ no change!

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



LP Heap – Analysis

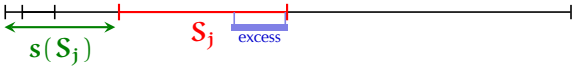
- **Actual Costs**
  - INSERT $O(\log\log n)$
  - DECREASE-KEY $O(\log\log n)$
  - DELETE-MIN $O(|S_1| + \log n)$

  could be n

⇝ Introduce **potential**

excess size over distance from min

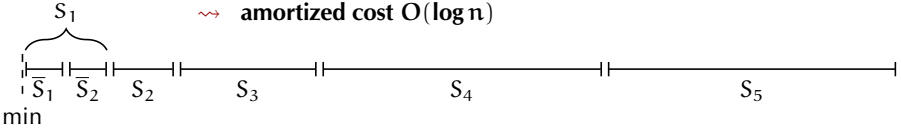
$$\Phi := \sum_{j=1}^{\ell} \varphi_j \quad \text{with} \quad \varphi_j := \max\{0, |S_j| - s(S_j)\}$$



- **Change in Potential**
  - INSERT Only φ_j increases (by 1) ⇝ $\Delta\Phi = O(1)$
  - DECREASE-KEY $\Delta\Phi \leqslant 0$ since we only increase "protection distance"
  - DELETE-MIN (1) Partitioning S_1 into $\overline{S}_1$ and $\overline{S}_2$ eliminates $\varphi_1 = |S_1|$ and adds $\overline{\varphi_1} + \overline{\varphi_2} = \frac{1}{2}|S_1|$
    (2) Moreover, deleting the min removes 1 from $s(S_j)$ of each set
    (3) Rule (C) is precisely chosen, so that $\varphi_j = 0$ before and after ⇝ no change!

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

⇝ **amortized cost $O(\log n)$**



Conclusion

You've seen (if you didn't nap)

- fully elementary efficient priority queue (*Lazy Partition Heaps*)
- amortized analysis suitable for undergrad teaching

Conclusion

You've seen (if you didn't nap)

- fully elementary efficient priority queue (*Lazy Partition Heaps*)
- amortized analysis suitable for undergrad teaching

There's more ($\rightarrow$ paper!)

- alternative variants *Fibonacci Heaps: The Next Generation* and *Exponential Bounded Sizes*
- LP Heaps can be implemented in pointer machine with bounded indegree $\rightsquigarrow$ *persistence!*
- partitioning around exact median can be replaced by single random pivot
- instead of linked lists, can use one big array $\rightsquigarrow$ *Quickheaps*
  LP Heaps yield new simple optimal pivot-forgetting rule for Quickheaps

Conclusion

You've seen (if you didn't nap)

- fully elementary efficient priority queue (*Lazy Partition Heaps*)
- amortized analysis suitable for undergrad teaching

There's more ($\rightarrow$ paper!)

- alternative variants *Fibonacci Heaps: The Next Generation* and *Exponential Bounded Sizes*
- LP Heaps can be implemented in pointer machine with bounded indegree $\rightsquigarrow$ *persistence!*
- partitioning around exact median can be replaced by single random pivot
- instead of linked lists, can use one big array $\rightsquigarrow$ *Quickheaps*
  LP Heaps yield new simple optimal pivot-forgetting rule for Quickheaps

Open Problems & Future Work

- (ongoing) empirical evaluation
- design space is huge; other/better partition-based heaps possible?
- Open: efficient support for MELD?



Icons made by *Freepik*, *Gregor Cresnar*, *Those Icons*, *Smashicons*, *Good Ware*, *Pause08*, and *Madebyoliver* from `www.flaticon.com`.
Other photos from `www.pixabay.com` or generated by *Gemini*.

Alternative Partition-Based Heaps

We give two further options for invariants of increasing “eagerness” / strictness

Alternative Partition-Based Heaps

We give two further options for invariants of increasing “eagerness” / strictness

1. ***Exponential Upper-Bounded Sets***
   - $|S_j| < 3 \cdot 2^j$
   - potential punishes sets for being close to upper bound and collectively for begin too small ($|S_1| + \cdots + |S_j|$ smaller than 2^{j-1})

Alternative Partition-Based Heaps

We give two further options for invariants of increasing “eagerness” / strictness

1. ***Exponential Upper-Bounded Sets***
   - $|S_j| < 3 \cdot 2^j$
   - potential punishes sets for being close to upper bound and collectively for begin too small ($|S_1| + \cdots + |S_j|$ smaller than 2^{j-1})
2. ***Fibonacci Heaps – The Next Generation (FH:TNG)***
   - sets can be absent / empty
   - $F_j \leqslant |S_j| \leqslant F_{j+3}$ (if present) $\quad F_k$ = kth *Fibonacci number*
   - $\leqslant 2$ consecutive nonempty sets
   - $\leqslant 8$ consecutive empty sets
   - potential similar: increases when close to upper/lower bound and when prefix total small

Alternative Partition-Based Heaps

We give two further options for invariants of increasing "eagerness" / strictness

1. ***Exponential Upper-Bounded Sets***
   - $|S_j| < 3 \cdot 2^j$
   - potential punishes sets for being close to upper bound and collectively for begin too small ($|S_1| + \cdots + |S_j|$ smaller than 2^{j-1})
2. ***Fibonacci Heaps – The Next Generation (FH:TNG)***
   - sets can be absent / empty
   - $F_j \leqslant |S_j| \leqslant F_{j+3}$ (if present) $\quad F_k$ = kth *Fibonacci number*
   - $\leqslant 2$ consecutive nonempty sets
   - $\leqslant 8$ consecutive empty sets
   - potential similar: increases when close to upper/lower bound and when prefix total small

- upon violation both concatenate sets down if underfull resp. partition up when overfull
  - may partition sets other than smallest (unlike LP Heaps)
  - may need to partition sets upon INSERT/DECREASE-KEY if invariants violated
  - $\rightsquigarrow$ amortized via potential