
The Sync Heap

Delete First, Ask Questions Later

Benjamin Aram Berendsohn Egor Gorbachev László Kozma

Max Planck Institute for Informatics, ETH Zürich, Dresden University of Technology

Comparison-based lower bound for sorting

Problem. Given n distinct keys, how many comparisons are needed to sort them?

Comparison-based lower bound for sorting

Problem. Given n distinct keys, how many comparisons are needed to sort them?

There are $n!$ possible output orders. A comparison tree therefore needs at least $n!$ leaves.

Comparison-based lower bound for sorting

Problem. Given n distinct keys, how many comparisons are needed to sort them?

There are $n!$ possible output orders. A comparison tree therefore needs at least $n!$ leaves.

$$\text{height} \geq \log_2(n!) = \Omega(n \log n).$$

Comparison-based lower bound for sorting

Problem. Given n distinct keys, how many comparisons are needed to sort them?

There are $n!$ possible output orders. A comparison tree therefore needs at least $n!$ leaves.

$$\text{height} \geq \log_2(n!) = \Omega(n \log n).$$

If sorting reduces to problem X , then X inherits an $\Omega(n \log n)$ lower bound.

Heap lower bound

$\text{insert}(x_1), \dots, \text{insert}(x_n)$

Heap lower bound

`insert( $x_1$ ), \dots, \text{insert}(x_n),` `extract-min(), \dots, \text{extract-min}()`
 k extractions

Heap lower bound

`insert( $x_1$ ), \dots, \text{insert}(x_n),` $\underbrace{\text{extract-min}(), \dots, \text{extract-min}()}_k$
 k extractions

Find k smallest keys in sorted order. Requires $\Omega(n + k \log k) = \Omega(n + k \log n)$ comparisons.

Heap lower bound

`insert( $x_1$ ), \dots, \text{insert}(\mathbf{x}_n),` $\underbrace{\text{extract-min}(), \dots, \text{extract-min}()}_k$
 k extractions

Find k smallest keys in sorted order. Requires $\Omega(n + k \log k) = \Omega(n + k \log n)$ comparisons.
For $k = n$ we have: $2n$ heap operations perform sorting, so *some* operations cost $\Omega(\log n)$.

Heap lower bound

`insert( $x_1$ ), \dots, \text{insert}(x_n),` `extract-min(), \dots, \text{extract-min}()`
 k extractions

Find k smallest keys in sorted order. Requires $\Omega(n + k \log k) = \Omega(n + k \log n)$ comparisons.
For $k = n$ we have: $2n$ heap operations perform sorting, so *some* operations cost $\Omega(\log n)$.

	insert	extract-min
binary heap	$O(\log n)$	$O(\log n)$

Heap lower bound

$\text{insert}(x_1), \dots, \text{insert}(x_n), \underbrace{\text{extract-min}(), \dots, \text{extract-min}()}_k$
 k extractions

Find k smallest keys in sorted order. Requires $\Omega(n + k \log k) = \Omega(n + k \log n)$ comparisons.
For $k = n$ we have: $2n$ heap operations perform sorting, so *some* operations cost $\Omega(\log n)$.

	insert	extract-min
binary heap	$O(\log n)$	$O(\log n)$
Fibonacci heap	$O(1)$	$O(\log n)$

Sorting barrier for SSSP?

Classical view: Dijkstra's algorithm orders vertices by distance and thus its $O(m + n \log n)$ time complexity is optimal.

Sorting barrier for SSSP?

Classical view: Dijkstra's algorithm orders vertices by distance and thus its $O(m + n \log n)$ time complexity is optimal.

But single-source shortest paths problem (SSSP) asks for distances to each vertex, not for the vertices sorted by distance.

Sorting barrier for SSSP?

Classical view: Dijkstra's algorithm orders vertices by distance and thus its $O(m + n \log n)$ time complexity is optimal.

But single-source shortest paths problem (SSSP) asks for distances to each vertex, not for the vertices sorted by distance.

What is hard: finding the distances or sorting them?

Sorting barrier for SSSP?

Classical view: Dijkstra's algorithm orders vertices by distance and thus its $O(m + n \log n)$ time complexity is optimal.

But single-source shortest paths problem (SSSP) asks for distances to each vertex, not for the vertices sorted by distance.

What is hard: finding the distances or sorting them?

Duan-Mao-Shu-Yin (2023): randomized, undirected, positive-real-weighted SSSP in $O(m\sqrt{\log n \log \log n})$ time. Faster for sparse graphs.

What does `extract-min` **do?**

The heap sorting reduction mixes two actions: modification + observation

What does `extract-min` do?

The heap sorting reduction mixes two actions: modification + observation

$$\text{extract-min} = \text{find-min} + (\text{silent}) \text{delete-min}$$

What does `extract-min` do?

The heap sorting reduction mixes two actions: modification + observation

$$\text{extract-min} = \text{find-min} + (\text{silent}) \text{delete-min}$$

n insertions, k extractions $\mapsto x_{(1)}, \dots, x_{(k)}$ in sorted order

n insertions, k silent deletions $\mapsto \perp, \dots, \perp$

What does `extract-min` do?

The heap sorting reduction mixes two actions: modification + observation

$$\text{extract-min} = \text{find-min} + (\text{silent}) \text{delete-min}$$

n insertions, k extractions $\mapsto x_{(1)}, \dots, x_{(k)}$ in sorted order

n insertions, k silent deletions $\mapsto \perp, \dots, \perp$

The first sequence reveals the deleted identities and their order. The second reports nothing.

What does `extract-min` do?

The heap sorting reduction mixes two actions: modification + observation

$$\text{extract-min} = \text{find-min} + (\text{silent}) \text{delete-min}$$

n insertions, k extractions $\mapsto x_{(1)}, \dots, x_{(k)}$ in sorted order

n insertions, k silent deletions $\mapsto \perp, \dots, \perp$

The first sequence reveals the deleted identities and their order. The second reports nothing.

Which part is expensive: updating the heap or observing it?

One observation at the end

Insert n keys $\rightarrow$ silently delete k minima $\rightarrow$ observe only at the end

One observation at the end

Insert n keys $\rightarrow$ silently delete k minima $\rightarrow$ observe only at the end

Computes the unordered set of the k smallest keys. This is *selection*, solvable in $O(n)$ time.

One observation at the end

Insert n keys $\rightarrow$ silently delete k minima $\rightarrow$ observe only at the end

Computes the unordered set of the k smallest keys. This is *selection*, solvable in $O(n)$ time.

Heap evaluation: Given interleaved insertions and silent deletions, report which elements remain and which are deleted after the entire sequence.

One observation at the end

Insert n keys $\rightarrow$ silently delete k minima $\rightarrow$ observe only at the end

Computes the unordered set of the k smallest keys. This is *selection*, solvable in $O(n)$ time.

Heap evaluation: Given interleaved insertions and silent deletions, report which elements remain and which are deleted after the entire sequence.

Example:

`insert, insert, delete-min, insert, insert, delete-min, insert, insert, delete-min, ...`

One observation at the end

Insert n keys $\rightarrow$ silently delete k minima $\rightarrow$ observe only at the end

Computes the unordered set of the k smallest keys. This is *selection*, solvable in $O(n)$ time.

Heap evaluation: Given interleaved insertions and silent deletions, report which elements remain and which are deleted after the entire sequence.

Example:

insert, insert, delete-min, insert, insert, delete-min, insert, insert, delete-min, ...

Generalization of selection.

One observation at the end

Insert n keys $\rightarrow$ silently delete k minima $\rightarrow$ observe only at the end

Computes the unordered set of the k smallest keys. This is *selection*, solvable in $O(n)$ time.

Heap evaluation: Given interleaved insertions and silent deletions, report which elements remain and which are deleted after the entire sequence.

Example:

insert, insert, delete-min, insert, insert, delete-min, insert, insert, delete-min, ...

Generalization of selection.

Can the final set still be found in linear time?

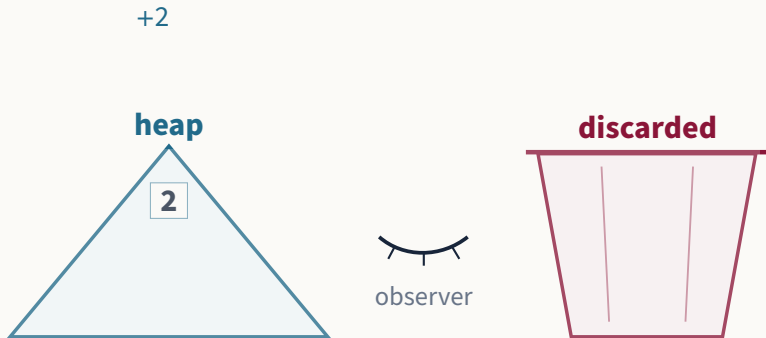
Heap evaluation example

Input: a sequence of operations on an initially empty heap.



Heap evaluation example

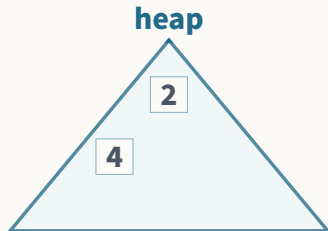
Input: a sequence of operations on an initially empty heap.



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

+2 +4



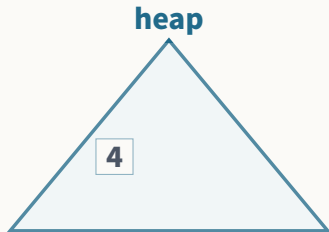

observer



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

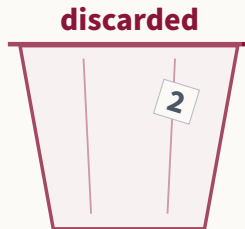
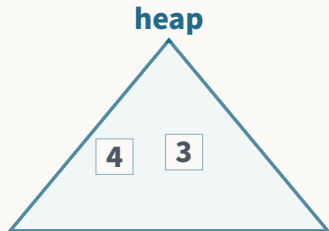
+2 +4 -



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

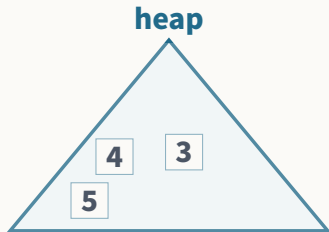
+2 +4 - +3



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

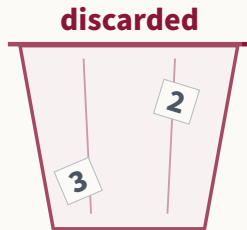
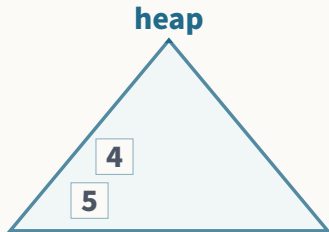
+2 +4 - +3 +5



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

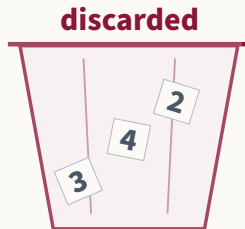
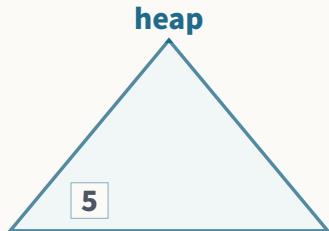
+2 +4 - +3 +5 -



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

+2 +4 - +3 +5 - -



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

+2 +4 - +3 +5 - - +1



Heap evaluation example

Input: a sequence of operations on an initially empty heap.

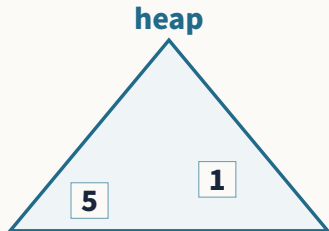
+2 +4 - +3 +5 - - +1



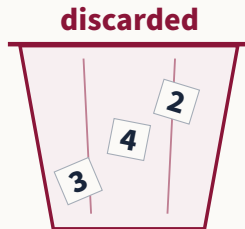
Heap evaluation example

Input: a sequence of operations on an initially empty heap.

+2 +4 - +3 +5 - - +1



observer



Soft heap: partial information is enough

Chazelle's **soft heap** (2000), introduced for their MST algorithm, buys fast heap operations by allowing controlled errors.

Soft heap: partial information is enough

Chazelle's **soft heap** (2000), introduced for their MST algorithm, buys fast heap operations by allowing controlled errors.

Some elements in the heap are **corrupted**. A corrupted element gets its key *increased*. At any time, at most εn elements in the heap are corrupted, where n is the number of insertions so far.

Soft heap: partial information is enough

Chazelle's **soft heap** (2000), introduced for their MST algorithm, buys fast heap operations by allowing controlled errors.

Some elements in the heap are **corrupted**. A corrupted element gets its key *increased*. At any time, at most εn elements in the heap are corrupted, where n is the number of insertions so far.

Insertions and deletions cost $O(\log(1/\varepsilon))$ amortized per operation.

Soft heap: partial information is enough

Chazelle's **soft heap** (2000), introduced for their MST algorithm, buys fast heap operations by allowing controlled errors.

Some elements in the heap are **corrupted**. A corrupted element gets its key *increased*. At any time, at most εn elements in the heap are corrupted, where n is the number of insertions so far.

Insertions and deletions cost $O(\log(1/\varepsilon))$ amortized per operation.

Certificate: an uncorrupted element remaining after a soft execution also remains after the exact execution.

Soft heap: partial information is enough

Chazelle's **soft heap** (2000), introduced for their MST algorithm, buys fast heap operations by allowing controlled errors.

Some elements in the heap are **corrupted**. A corrupted element gets its key *increased*. At any time, at most εn elements in the heap are corrupted, where n is the number of insertions so far.

Insertions and deletions cost $O(\log(1/\varepsilon))$ amortized per operation.

Certificate: an uncorrupted element remaining after a soft execution also remains after the exact execution.

Set $\varepsilon = 1/4$. If at most $n/2$ elements are deleted, at least $n/4$ uncorrupted survivors can be certified and removed from the instance.

Soft heap: partial information is enough

Chazelle's **soft heap** (2000), introduced for their MST algorithm, buys fast heap operations by allowing controlled errors.

Some elements in the heap are **corrupted**. A corrupted element gets its key *increased*. At any time, at most εn elements in the heap are corrupted, where n is the number of insertions so far.

Insertions and deletions cost $O(\log(1/\varepsilon))$ amortized per operation.

Certificate: an uncorrupted element remaining after a soft execution also remains after the exact execution.

Set $\varepsilon = 1/4$. If at most $n/2$ elements are deleted, at least $n/4$ uncorrupted survivors can be certified and removed from the instance.

What if very few elements survive?

The two sides of heap evaluation

Soft min-heap: operations take constant time; corrupted keys increase; after n insertions, at most $n/4$ elements in the heap are corrupted.

Main property: An uncorrupted final survivor also survives the exact execution.

Let d be the number of deletions.

The two sides of heap evaluation

Soft min-heap: operations take constant time; corrupted keys increase; after n insertions, at most $n/4$ elements in the heap are corrupted.

Main property: An uncorrupted final survivor also survives the exact execution.

Let d be the number of deletions.

Few deletions: $d \leq n/2$

At least $n/2$ survive.

Soft min-heap certifies $\Omega(n)$ survivors.

The two sides of heap evaluation

Soft min-heap: operations take constant time; corrupted keys increase; after n insertions, at most $n/4$ elements in the heap are corrupted.

Main property: An uncorrupted final survivor also survives the exact execution.

Let d be the number of deletions.

Few deletions: $d \leq n/2$

At least $n/2$ survive.

Soft min-heap certifies $\Omega(n)$ survivors.

Many deletions: $d > n/2$

The two sides of heap evaluation

Soft min-heap: operations take constant time; corrupted keys increase; after n insertions, at most $n/4$ elements in the heap are corrupted.

Main property: An uncorrupted final survivor also survives the exact execution.

Let d be the number of deletions.

Few deletions: $d \leq n/2$

At least $n/2$ survive.

Soft min-heap certifies $\Omega(n)$ survivors.

Many deletions: $d > n/2$

At least $n/2$ are deleted.

The two sides of heap evaluation

Soft min-heap: operations take constant time; corrupted keys increase; after n insertions, at most $n/4$ elements in the heap are corrupted.

Main property: An uncorrupted final survivor also survives the exact execution.

Let d be the number of deletions.

Few deletions: $d \leq n/2$

At least $n/2$ survive.

Soft min-heap certifies $\Omega(n)$ survivors.

Many deletions: $d > n/2$

At least $n/2$ are deleted.

Soft max-heap certifies $\Omega(n)$ deletions?

The two sides of heap evaluation

Soft min-heap: operations take constant time; corrupted keys increase; after n insertions, at most $n/4$ elements in the heap are corrupted.

Main property: An uncorrupted final survivor also survives the exact execution.

Let d be the number of deletions.

Few deletions: $d \leq n/2$

At least $n/2$ survive.

Soft min-heap certifies $\Omega(n)$ survivors.

Many deletions: $d > n/2$

At least $n/2$ are deleted.

Soft max-heap certifies $\Omega(n)$ deletions?

One linear-time pass always certifies a constant fraction.

Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 0$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 0$

1 has no slot

Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 1$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 2$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 **+5** - - +1

deletion candidate slots

$d = 2$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - **+3** +5 - - +1

deletion candidate slots

$d = 2$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 3$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 3$



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1

deletion candidate slots

$d = 3$

5

3

4

2

Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

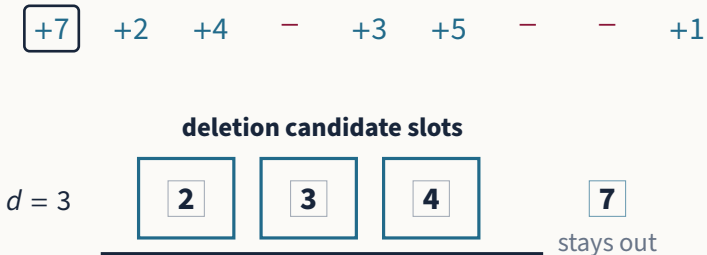
+7 +2 +4 - +3 +5 - - +1

deletion candidate slots



Reverse evaluation

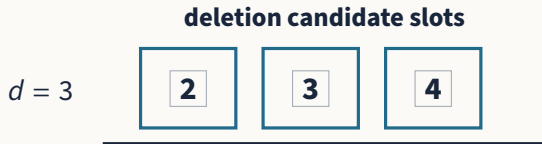
Scan the operation sequence from right to left. Each deletion opens a slot.



Reverse evaluation

Scan the operation sequence from right to left. Each deletion opens a slot.

+7 +2 +4 - +3 +5 - - +1



With a soft *max*-heap, uncorrupted candidates certify deletions.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

For certified deleted elements, need to remove matching deletion operations.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

For certified deleted elements, need to remove matching deletion operations.

Matching exactly would reintroduce the sorting lower bound.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

For certified deleted elements, need to remove matching deletion operations.

Matching exactly would reintroduce the sorting lower bound.

Sufficient to pick corresponding deletion operations greedily.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

For certified deleted elements, need to remove matching deletion operations.

Matching exactly would reintroduce the sorting lower bound.

Sufficient to pick corresponding deletion operations greedily.

$$T(n) \leq T(3n/4) + O(n) = O(n).$$

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

For certified deleted elements, need to remove matching deletion operations.

Matching exactly would reintroduce the sorting lower bound.

Sufficient to pick corresponding deletion operations greedily.

$$T(n) \leq T(3n/4) + O(n) = O(n).$$

Theorem. Heap evaluation has a deterministic comparison-based $O(n)$ -time algorithm.

Linear-time heap evaluation

In either case, a linear-time pass certifies at least $n/4$ elements.

Remove certified survivors directly.

For certified deleted elements, need to remove matching deletion operations.

Matching exactly would reintroduce the sorting lower bound.

Sufficient to pick corresponding deletion operations greedily.

$$T(n) \leq T(3n/4) + O(n) = O(n).$$

Theorem. Heap evaluation has a deterministic comparison-based $O(n)$ -time algorithm.

When would an algorithm delete minima without needing their identities right away?

Application: unit-time scheduling

Job j is released at time 0, has unit processing time, integer deadline d_j , and profit w_j . We seek a feasible set of jobs of maximum total profit.

Application: unit-time scheduling

Job j is released at time 0, has unit processing time, integer deadline d_j , and profit w_j . We seek a feasible set of jobs of maximum total profit.

Lawler's algorithm [Lawler 1976] processes jobs in the increasing order of deadlines, inserts each into a min-heap keyed by w_j , and deletes the minimum whenever the heap size exceeds d_j .

Application: unit-time scheduling

Job j is released at time 0, has unit processing time, integer deadline d_j , and profit w_j . We seek a feasible set of jobs of maximum total profit.

Lawler's algorithm [Lawler 1976] processes jobs in the increasing order of deadlines, inserts each into a min-heap keyed by w_j , and deletes the minimum whenever the heap size exceeds d_j .

Only the final heap state matters.

Application: unit-time scheduling

Job j is released at time 0, has unit processing time, integer deadline d_j , and profit w_j . We seek a feasible set of jobs of maximum total profit.

Lawler's algorithm [Lawler 1976] processes jobs in the increasing order of deadlines, inserts each into a min-heap keyed by w_j , and deletes the minimum whenever the heap size exceeds d_j .

Only the final heap state matters.

$$O(n \log n) \longrightarrow O(n)$$

Between one observation and regular heap

`extract-min = find-min + (silent) delete-min`

Between one observation and regular heap

`extract-min = find-min + (silent) delete-min`

heap evaluation

observe only

at the end

$O(n)$

Between one observation and regular heap

`extract-min = find-min + (silent) delete-min`

heap evaluation

observe only
at the end
 $O(n)$

regular heap

find-min before
every deletion
 $\Omega(n + d \log n)$

Between one observation and regular heap

`extract-min = find-min + (silent) delete-min`

heap evaluation

observe only
at the end
 $O(n)$

?

regular heap

find-min before
every deletion
 $\Omega(n + d \log n)$

Between one observation and regular heap

`extract-min = find-min + (silent) delete-min`

heap evaluation

observe only
at the end
 $O(n)$

?

regular heap

find-min before
every deletion
 $\Omega(n + d \log n)$



Between one observation and regular heap

`extract-min = find-min + (silent) delete-min`

heap evaluation

observe only
at the end
 $O(n)$

?

regular heap

find-min before
every deletion
 $\Omega(n + d \log n)$



At an observation, **synchronize**: report the current minimum and the *unordered* list of elements deleted since the last observation.

Synchronizing one block

Goal. Process an interleaved batch of insertions and deletions between observations.

Synchronizing one block

Goal. Process an interleaved batch of insertions and deletions between observations.



Synchronizing one block

Goal. Process an interleaved batch of insertions and deletions between observations.



Selectable heap [Sandlund–Zhang 2022]: d deletions cost $O(d \log(n/d))$.

Synchronizing one block

Goal. Process an interleaved batch of insertions and deletions between observations.



Selectable heap [Sandlund–Zhang 2022]: d deletions cost $O(d \log(n/d))$.

Consider a batch S_2 of operations. Using heap evaluation, we can reorder it to group all deletions together.



Synchronizing one block

Goal. Process an interleaved batch of insertions and deletions between observations.



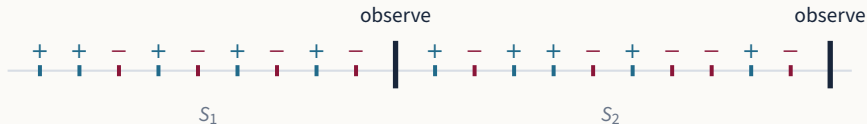
Selectable heap [Sandlund–Zhang 2022]: d deletions cost $O(d \log(n/d))$.

Consider a batch S_2 of operations. Using heap evaluation, we can reorder it to group all deletions together.



Synchronizing one block

Goal. Process an interleaved batch of insertions and deletions between observations.



Selectable heap [Sandlund–Zhang 2022]: d deletions cost $O(d \log(n/d))$.

Consider a batch S_2 of operations. Using heap evaluation, we can reorder it to group all deletions together.



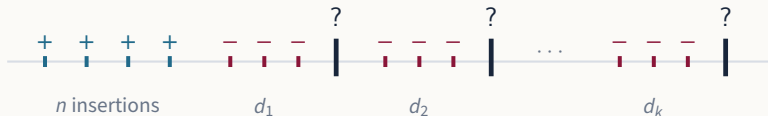
Processing a block costs $O(|S_i| + d_i \log(n/d_i))$.

Multiple selection

Insert n keys first. Then alternate blocks of silent deletions and `find-min`:

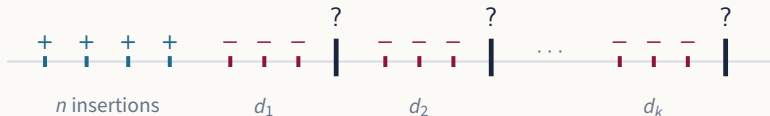
Multiple selection

Insert n keys first. Then alternate blocks of silent deletions and `find-min`:



Multiple selection

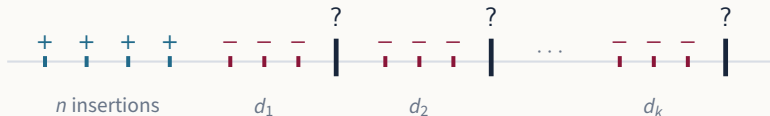
Insert n keys first. Then alternate blocks of silent deletions and `find-min`:



Each `find-min` returns an element of rank $p_i = d_1 + \dots + d_i$: *multiple selection*.

Multiple selection

Insert n keys first. Then alternate blocks of silent deletions and `find-min`:



Each `find-min` returns an element of rank $p_i = d_1 + \dots + d_i$: *multiple selection*.

Sorting gives the lower bound $\Omega\left(n + \sum_{i=1}^k d_i \log \frac{n}{d_i}\right)$.

Optimal observation-sensitive time complexity

Let n be the total number of insertions, k be the number of observations, and d_i be the number of deletions in block i .

Optimal observation-sensitive time complexity

Let n be the total number of insertions, k be the number of observations, and d_i be the number of deletions in block i .

$$\text{sync heap} \quad O\left(n + \sum_i d_i \log \frac{n}{d_i}\right)$$

$$\text{multiple selection} \quad \Omega\left(n + \sum_i d_i \log \frac{n}{d_i}\right)$$

Optimal observation-sensitive time complexity

Let n be the total number of insertions, k be the number of observations, and d_i be the number of deletions in block i .

$$\text{sync heap} \quad O\left(n + \sum_i d_i \log \frac{n}{d_i}\right)$$

$$\text{multiple selection} \quad \Omega\left(n + \sum_i d_i \log \frac{n}{d_i}\right)$$

Worst-case time complexity if observations are evenly spaced: $\Theta(n + d \log k)$.

Optimal observation-sensitive time complexity

Let n be the total number of insertions, k be the number of observations, and d_i be the number of deletions in block i .

$$\text{sync heap} \quad O\left(n + \sum_i d_i \log \frac{n}{d_i}\right)$$

$$\text{multiple selection} \quad \Omega\left(n + \sum_i d_i \log \frac{n}{d_i}\right)$$

Worst-case time complexity if observations are evenly spaced: $\Theta(n + d \log k)$.

Theorem (sync heap). There is a heap that processes an online sequence of N operations, of which d are deletions and k observations in

$O(N + d \log k)$ total time.

Don't look ~~up~~ at the heap!

Don't look ~~up~~ at the heap!

The logarithmic barrier of sorting is not inherently the cost of deleting the minimum but rather the information cost of immediately learning which element was deleted.

Don't look up at the heap!

The logarithmic barrier of sorting is not inherently the cost of deleting the minimum but rather the information cost of immediately learning which element was deleted.

One observation at the end: $O(N)$. k observations: $O(N + d \log k)$.

Don't look ~~up~~ at the heap!

The logarithmic barrier of sorting is not inherently the cost of deleting the minimum but rather the information cost of immediately learning which element was deleted.

One observation at the end: $O(N)$. k observations: $O(N + d \log k)$.

Presentation layout and figures produced by an LLM based on my detailed description of the expected presentation contents.

Don't look ~~up~~ at the heap!

The logarithmic barrier of sorting is not inherently the cost of deleting the minimum but rather the information cost of immediately learning which element was deleted.

One observation at the end: $O(N)$. k observations: $O(N + d \log k)$.

Presentation layout and figures produced by an LLM based on my detailed description of the expected presentation contents.

*Independent work: very similar unit-time scheduling algorithm by Matthias Görgens:
cs.stackexchange.com/a/171695/50292*