A Nearly Quadratic Improvement for Memory Reallocation

Martin Farach-Colton [1], William Kuszmaul[2], Nathan Sterling Sheffield[3], Alek Westover[3]

NYU[1], Harvard[2], MIT[3]

SPAA 2024

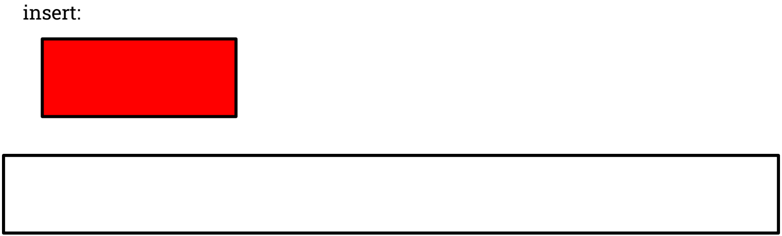
The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".

memory:

The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".

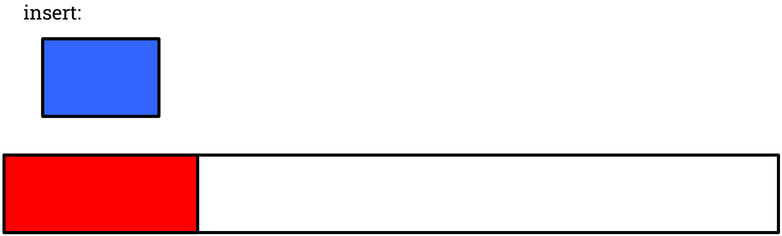


The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) “minimize the total size of items moved”.

The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) “minimize the total size of items moved”.

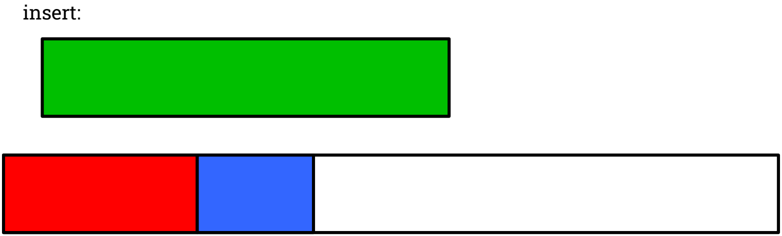


The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".

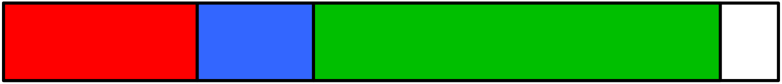
The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".



The Memory Reallocation Problem

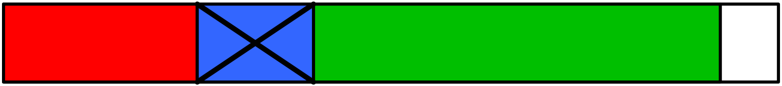
- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".



The Memory Reallocation Problem

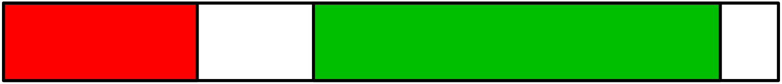
- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".

delete:



The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) “minimize the total size of items moved”.



The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".

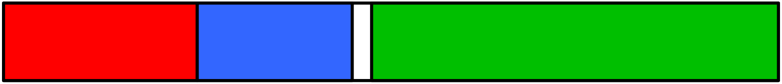
insert:

The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".

The Memory Reallocation Problem

- **Input**: a sequence of item inserts and deletes.
- **Output**: on each item update, arrange the items to occupy non-overlapping regions of memory.
- **Goal** (intuitively) "minimize the total size of items moved".



The Memory Reallocation Problem

Definition

Memory: $[0, 1]$.

The Memory Reallocation Problem

Definition

Memory: $[0, 1]$.

Update: insert or delete.

The Memory Reallocation Problem

Definition

Memory: $[0, 1]$.
Update: insert or delete.

$$\text{Update cost} = \frac{\text{total size of items moved to handle update}}{\text{size of updated item}}.$$

The Memory Reallocation Problem

Definition

Memory: $[0, 1]$.
Update: insert or delete.

$$\text{Update cost} = \frac{\text{total size of items moved to handle update}}{\text{size of updated item}}.$$

$$\varepsilon = 1 - (\text{sum of item sizes}) = 1 - (\text{load factor}).$$

The Memory Reallocation Problem

Definition

Memory: $[0, 1]$.
Update: insert or delete.

$$\text{Update cost} = \frac{\text{total size of items moved to handle update}}{\text{size of updated item}}.$$

$$\varepsilon = 1 - (\text{sum of item sizes}) = 1 - (\text{load factor}).$$

Goal: Minimize update cost while handling load factor $1 - \varepsilon$.

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Proof.

$\square$

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Proof.

- Suppose an item of size k must be inserted.

$\square$

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Proof.

- Suppose an item of size k must be inserted.
- By averaging, there is a length $k\varepsilon^{-1}$ interval with k free space.

$\square$

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Proof.

- Suppose an item of size k must be inserted.
- By averaging, there is a length $k\varepsilon^{-1}$ interval with k free space.
- Re-arrange this interval and place the inserted item in it.

$\square$

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Theorem (Kuszmaul FOCS'23)

If all items have size at most $O(\varepsilon^4)$ then there is an allocator with expected update cost $O(\log \varepsilon^{-1})$.

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Theorem (Kuszmaul FOCS'23)

If all items have size at most $O(\varepsilon^4)$ then there is an allocator with expected update cost $O(\log \varepsilon^{-1})$.

Conjecture (Kuszmaul FOCS'23)

$\Omega(\varepsilon^{-1})$ expected update cost is required for items of size $\Theta(\varepsilon)$.

Background

Proposition (Folklore Algorithm)

There is an allocator with update cost $O(\varepsilon^{-1})$.

Theorem (Kuszmaul FOCS'23)

If all items have size at most $O(\varepsilon^4)$ then there is an allocator with expected update cost $O(\log \varepsilon^{-1})$.

Conjecture (Kuszmaul FOCS'23)

$\Omega(\varepsilon^{-1})$ expected update cost is required for items of size $\Theta(\varepsilon)$.

Main Result

Theorem

There is an allocator for arbitrary items with worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.

Main Result

Theorem

There is an allocator for arbitrary items with worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.

Up next:
Prove a simpler version of this theorem to illustrate some ideas.

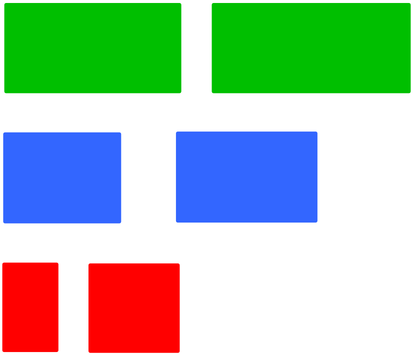
Theorem

There is an allocator for items with sizes in $[\varepsilon, 2\varepsilon]$ with average update cost $\widetilde{O}(\varepsilon^{-2/3})$.

Allocator Description

Partition $[\varepsilon, 2\varepsilon)$ into $\lceil \varepsilon^{-1/3} \rceil$ **size classes**.
i-th size class:

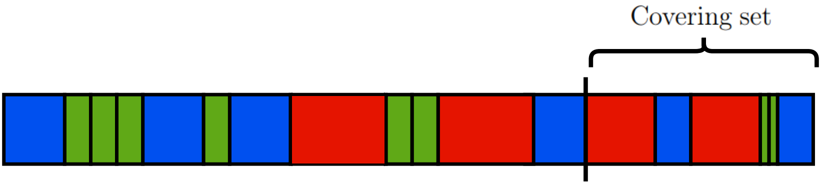
$$[\varepsilon + (i-1)\varepsilon^{4/3}, \varepsilon + i\varepsilon^{4/3}).$$



Allocator Description

Covering Set: Suffix of memory.
Let x_i = number of items of size class i.

Every $\lfloor \varepsilon^{-1/3} \rfloor$ updates the allocator performs an expensive **rebuild operation** where it rearranges all of memory to place the smallest $\min(x_i, \lfloor \varepsilon^{-1/3} \rfloor)$ items of size class i in the covering set.

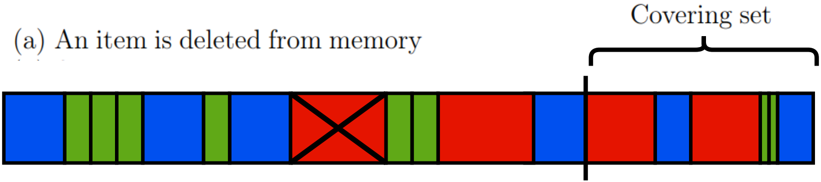


Allocator Description

Item Inserts: Add inserted items to the covering set. Place them after the final item in memory.

Allocator Description

Item Deletes:

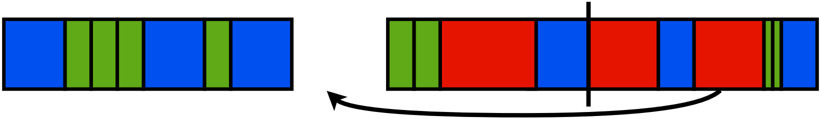


(a) An item is deleted from memory

Allocator Description

Item Deletes:

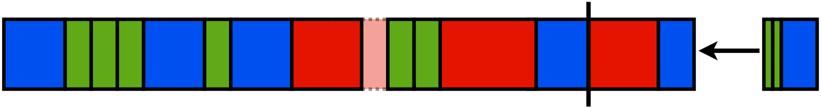
(b) SIMPLE replaces the deleted item with a smaller item of the same size class from the covering set



Allocator Description

Item Deletes:

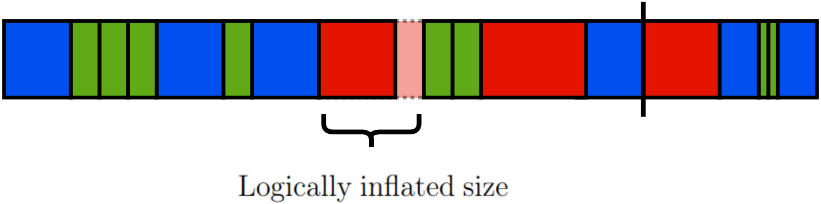
(c) SIMPLE compacts the covering set



Allocator Description

Item Deletes:

(d) The moved item is logically treated as being the size of the one it replaced until SIMPLE performs a full rebuild



Allocator Correctness

Lemma

The allocator is well defined and produces a valid allocation.

Allocator Correctness

Lemma

The allocator is well defined and produces a valid allocation.

Proof.

Periodic rebuilds prevent gaps from building up too much: we introduce up to $\varepsilon^{4/3}$ gap per delete, and rebuild after $\lfloor \varepsilon^{-1/3} \rfloor$ updates. $\square$

Allocator Performance

Lemma

The allocator achieves amortized update cost $O(\varepsilon^{-2/3})$.

Proof.

- The covering set consists of at most $O(\varepsilon^{-2/3})$ items, and so has total size at most $O(\varepsilon^{1/3})$.
- Compacting the covering set on each delete thus costs $O(\varepsilon^{1/3}/\varepsilon)$.
- The periodic rebuilds cost $O(\varepsilon^{-1})$ and happen every $\lfloor \varepsilon^{-1/3} \rfloor$ updates.

$\square$

Extending to Our Full Allocator

Limitations of the simple allocator:

Extending to Our Full Allocator

Limitations of the simple allocator:

1. Simple allocator has average update cost $O(\varepsilon^{-2/3})$; our full allocator achieves worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.

Extending to Our Full Allocator

Limitations of the simple allocator:

1. Simple allocator has average update cost $O(\varepsilon^{-2/3})$; our full allocator achieves worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.
2. Simple allocator requires item sizes to be in $[\varepsilon, 2\varepsilon]$.

Extending to Our Full Allocator

Limitations of the simple allocator:

1. Simple allocator has average update cost $O(\varepsilon^{-2/3})$; our full allocator achieves worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.
2. Simple allocator requires item sizes to be in $[\varepsilon, 2\varepsilon]$.

Solution ideas:

Extending to Our Full Allocator

Limitations of the simple allocator:

1. Simple allocator has average update cost $O(\varepsilon^{-2/3})$; our full allocator achieves worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.
2. Simple allocator requires item sizes to be in $[\varepsilon, 2\varepsilon]$.

Solution ideas:

- Use Kuszmaul's Allocator to handle items with size $< \varepsilon^4$.

Extending to Our Full Allocator

Limitations of the simple allocator:

1. Simple allocator has average update cost $O(\varepsilon^{-2/3})$; our full allocator achieves worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.
2. Simple allocator requires item sizes to be in $[\varepsilon, 2\varepsilon]$.

Solution ideas:

- Use Kuszmaul's Allocator to handle items with size $< \varepsilon^4$.
- Main difficulty is extending simple allocator to handle sizes $[\varepsilon^4, 1]$.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1 + \varepsilon)]$.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1 + \varepsilon)]$.
- With geometric size classes, larger size classes waste more space than smaller size classes.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1 + \varepsilon)]$.
- With geometric size classes, larger size classes waste more space than smaller size classes.
- Further complication: rearranging items costs more when performed on small item updates.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1 + \varepsilon)]$.
- With geometric size classes, larger size classes waste more space than smaller size classes.
- Further complication: rearranging items costs more when performed on small item updates.

Solutions:

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1 + \varepsilon)]$.
- With geometric size classes, larger size classes waste more space than smaller size classes.
- Further complication: rearranging items costs more when performed on small item updates.

Solutions:

- Create nested covering sets.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1+\varepsilon)]$.
- With geometric size classes, larger size classes waste more space than smaller size classes.
- Further complication: rearranging items costs more when performed on small item updates.

Solutions:

- Create nested covering sets.
- Number of items allowed in each size class is inversely proportional to the item size.

Extending to Our Full Allocator

Extending our simple allocator to handle sizes $[\varepsilon^4, 1]$:

- Too broad a size range to use uniformly-sized size classes.
- Use *geometric* size classes $[s, s \cdot (1 + \varepsilon)]$.
- With geometric size classes, larger size classes waste more space than smaller size classes.
- Further complication: rearranging items costs more when performed on small item updates.

Solutions:

- Create nested covering sets.
- Number of items allowed in each size class is inversely proportional to the item size.
- Randomized rebuilds.

Can we Outperform $\widetilde{O}(\varepsilon^{-1/2})$?

Seems challenging to extend current techniques.

Can we Outperform $\widetilde{O}(\varepsilon^{-1/2})$?

Seems challenging to extend current techniques.

However, we can improve substantially in interesting special cases.

Can we Outperform $\widetilde{O}(\varepsilon^{-1/2})$?

Seems challenging to extend current techniques.

However, we can improve substantially in interesting special cases.

Definition

Stochastic Items: Alternating inserts of items with random sizes and deletes of random items.

Can we Outperform $\widetilde{O}(\varepsilon^{-1/2})$?

Seems challenging to extend current techniques.

However, we can improve substantially in interesting special cases.

Definition

Stochastic Items: Alternating inserts of items with random sizes and deletes of random items.

Theorem

There is an allocator for stochastic items of sizes in $[\varepsilon, 2\varepsilon)$ *with worst-case expected update cost* $O(\log \varepsilon^{-1})$.

Stochastic Items — Proof Ideas

- When an item is deleted, group it together with a set of $\Theta(\log \varepsilon^{-1})$ surrounding items, and call the size of this group y.
- A random set of $\Theta(\log \varepsilon^{-1})$ items has good probability of having a subset sum which is close to y.
- Replace the deleted item and its group with a subset of a block near the end of memory.
- Compact the end of memory.

Main Open Question

Many cases where $O(\log \varepsilon^{-1})$ expected update cost is possible:

Main Open Question

Many cases where $O(\log \varepsilon^{-1})$ expected update cost is possible:

- Small items

Main Open Question

Many cases where $O(\log \varepsilon^{-1})$ expected update cost is possible:

- Small items
- Stochastic items

Main Open Question

Many cases where $O(\log \varepsilon^{-1})$ expected update cost is possible:

- Small items
- Stochastic items
- Few distinct types of items

Main Open Question

Many cases where $O(\log \varepsilon^{-1})$ expected update cost is possible:

- Small items
- Stochastic items
- Few distinct types of items

Lower Bounds?

Theorem

There is no "resizable" allocator with expected update cost $o(\log \varepsilon^{-1})$.

Main Open Question

Many cases where $O(\log \varepsilon^{-1})$ expected update cost is possible:

- Small items
- Stochastic items
- Few distinct types of items

Question

Is there an allocator with expected update cost $o(\varepsilon^{-1/2})$?