Lecture 1: Amortized Analysis

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

May 9, 2023

Overview

- Introduction
  - Why amortized analysis?
  - Types of amortized analyses
- Examples of Data Structures Using Amortized Analysis
  - Aggregate Analysis
  - Accounting Method
  - Potential Method
- Acknowledgements

Why Amortized Analysis?

In your first data structures course, you learned how to devise data structures that had good *worst-case* or *average-case* behaviour *per query*.

Why Amortized Analysis?

In your first data structures course, you learned how to devise data structures that had good *worst-case* or *average-case* behaviour *per query*.

Worst or average-case complexity of data structures

Data Structure	search	insertion	deletion
Doubly-Linked List	$O(n)$	$O(1)$	$O(n)$
Ordered Array	$O(\log n)$	$O(n)$	$O(n)$
Hash Tables[a]	$O(1)$	$O(1)$	$O(1)$
Balanced Binary Search Trees[b]	$O(\log n)$	$O(\log n)$	$O(\log n)$

[a] Average-case, although worst-case search time is $\Theta(n)$

[b] Also average-case. Worst-case complexity is $O(\mathit{height})$ of the tree, which can be $\Theta(n)$.

Why Amortized Analysis?

In **amortized analysis**, one averages the *total time* required to perform a sequence of data-structure operations over *all operations performed*.

Upshot of amortized analysis: worst-case cost *per query* may be high for one particular query, so long as overall average cost per query is small in the end!

Remark

Amortized analysis is a *worst-case* analysis. That is, it measures the average performance of each operation in the worst case.

Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.

Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.
2. **Accounting Method:** assign certain *charge* to each operation (independent of the actual cost of the operation). If operation is cheaper than the charge, then build up credit to use later.

Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.
2. **Accounting Method:** assign certain *charge* to each operation (independent of the actual cost of the operation). If operation is cheaper than the charge, then build up credit to use later.
3. **Potential Method:** one comes up with *potential energy* of a data structure, which maps each state of entire data-structure to a real number (its "potential"). Differs from accounting method because we assign credit to the data structure as a whole, instead of assigning credit to each operation.

One simple problem - several analyses

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)

One simple problem - Aggregate Analysis

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- *Question:* how many bit operations will it take to increment C from 0 to n?

One simple problem - Aggregate Analysis

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- *Question:* how many bit operations will it take to increment C from 0 to n?
- Notice that the *worst-case* time *per operation* is $\log(n)$. So an upper bound is $O(n \log n)$.

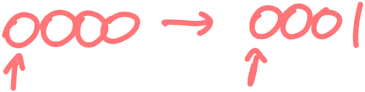
$n = 8$ $\log n = 3$

0111

1000

One simple problem - Aggregate Analysis

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- *Question:* how many bit operations will it take to increment C from 0 to n?
- Notice that the *worst-case* time *per operation* is $\log(n)$. So an upper bound is $O(n \log n)$.
- But overall, we see that the *most significant bits* get updated *very infrequently*.
- Is the above analysis tight?



One simple problem - Aggregate Analysis

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- *Question:* how many bit operations will it take to increment C from 0 to n?
- Notice that the *worst-case* time *per operation* is $\log(n)$. So an upper bound is $O(n \log n)$.
- But overall, we see that the *most significant bits* get updated *very infrequently*.
- Is the above analysis tight?
- How many times will we "flip" the k^{th} bit?

i.e. after 2^{k-1} operations/increments

flip k^{th} bit in intervals of 2^{k-1} increments

$\Rightarrow \left\lfloor n / 2^{k-1} \right\rfloor$ flips of k^{th} bit

One simple problem - Aggregate Analysis

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- *Question:* how many bit operations will it take to increment C from 0 to n?
- Notice that the *worst-case* time *per operation* is $\log(n)$. So an upper bound is $O(n \log n)$.
- But overall, we see that the *most significant bits* get updated *very infrequently*.
- Is the above analysis tight?
- How many times will we "flip" the k^{th} bit?
- Putting it all together, we get:

$$\sum_{k=0}^{\lceil \log n \rceil} \underbrace{\lfloor n/2^k \rfloor}_{\#\text{ times } k^{th} \text{ bit gets flipped}} < \sum_{k \geq 0} n/2^k = 2n$$

$\Rightarrow$ amortized cost per op. $\Theta(1)$ (total $\Theta(n)$)

Accounting method

- suppose that the *actual cost* of each operation of our algorithm is c_i (which may be hard to track)

$$\sum_{i=1}^{n} c_i$$

$c_i :=$ cost of i^{th} increment (# bit flips)

$c_1 = 1$ $\quad$ $0000 \mapsto 0001$

$c_2 = 2$ $\quad$ $0001 \mapsto 0010$

$c_3 = 1$ $\quad$ $0010 \mapsto 0011$

$c_4 = 3$

Accounting method

- suppose that the *actual cost* of each operation of an algorithm is c_i

  (which may be hard to track)

- In the accounting method, at each step of the algorithm, we assign *charges* γ_i to each operation such that

$$\sum_{i=1}^{\ell} \gamma_i \geq \sum_{i=1}^{\ell} c_i$$

  for any $\ell \geq 1$

- That is, the *total charged* up to step ℓ is greater than or equal to the *actual cost* of all operations up to that point

Accounting method

- suppose that the *actual cost* of each operation of an algorithm is c_i

  (which may be hard to track)
- In the accounting method, at each step of the algorithm, we assign *charges* γ_i to each operation such that

  $$\sum_{i=1}^{\ell} \gamma_i \geq \sum_{i=1}^{\ell} c_i$$

  for any $\ell \geq 1$
- That is, the *total charged* up to step ℓ is greater than or equal to the *actual cost* of all operations up to that point
- In other words, we charge certain operations *before they happen*

Accounting method

- suppose that the *actual cost* of each operation of an algorithm is c_i

  (which may be hard to track)

- In the accounting method, at each step of the algorithm, we assign *charges* γ_i to each operation such that

$$\sum_{i=1}^{\ell} \gamma_i \geq \sum_{i=1}^{\ell} c_i$$

  for any $\ell \geq 1$

- That is, the *total charged* up to step ℓ is greater than or equal to the *actual cost* of all operations up to that point
- In other words, we charge certain operations *before they happen*
- If we manage to do the above, then

$$\text{Total cost} \leq \text{Total charged}$$

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *charge earlier operations* for the *cost of subsequent operations*?

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *charge earlier operations* for the *cost of subsequent operations*?
- Suppose we charge the cost of "clearing a bit" (changing the bit from 1 to 0) to the operation that sets the bit to 1 in the first place.

$0000 \longmapsto 0001$

$\uparrow$

$\gamma_1 = 2$

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *charge earlier operations* for the *cost of subsequent operations*?
- Suppose we charge the cost of "clearing a bit" (changing the bit from 1 to 0) to the operation that sets the bit to 1 in the first place.
- If we flip k bits during an increment, we have already charged $k - 1$ of those bit flips to earlier bit flips.

Why?

$k = 3$

$011 \mapsto 100$

change k → clear $k-1$ bits
set 1 bit

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *charge earlier operations* for the *cost of subsequent operations*?
- Suppose we charge the cost of "clearing a bit" (changing the bit from 1 to 0) to the operation that sets the bit to 1 in the first place.
- If we flip k bits during an increment, we have already charged $k - 1$ of those bit flips to earlier bit flips.

Why?

- Note that if we flip k bits, we must set $k - 1$ of these bits to 0 (so that it carries over)

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *charge earlier operations* for the *cost of subsequent operations*?
- Suppose we charge the cost of "clearing a bit" (changing the bit from 1 to 0) to the operation that sets the bit to 1 in the first place.
- If we flip k bits during an increment, we have already charged $k-1$ of those bit flips to earlier bit flips.

Why?

- Note that if we flip k bits, we must set $k-1$ of these bits to 0 (so that it carries over)
- So, instead of *paying* for k bit flips in this increment, we *charge* at most 2:
  - one for setting a bit to 1, *actual cost*
  - and the other is the charge to "clear this bit" *clearing charge*

$\gamma_i = 2$

One simple problem - accounting method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *charge earlier operations* for the *cost of subsequent operations*?
- Suppose we charge the cost of "clearing a bit" (changing the bit from 1 to 0) to the operation that sets the bit to 1 in the first place.
- If we flip k bits during an increment, we have already charged $k-1$ of those bit flips to earlier bit flips.

Why?

- Note that if we flip k bits, we must set $k-1$ of these bits to 0 (so that it carries over)
- So, instead of *paying* for k bit flips in this increment, we *charge* at most 2:
  - one for setting a bit to 1, *actual cost*
  - and the other is the charge to "clear this bit" *clearing charge*
- Total cost $\leq$ Total Charged $= 2 \times n$

Example of the accounting method

$\gamma_1 = 2$

(1 ← setting one bit to 1)

(1 ← clearing this bit (maybe done later))

$\sum \gamma_i = 2 \times n = 2n$

Formal Analysis of the accounting method

Potential Method

- suppose that the *actual cost* of each operation of an algorithm is c_i
  (which may be hard to track)

Potential Method

- suppose that the *actual cost* of each operation of an algorithm is c_i

  (which may be hard to track)

- potential method: assign potential Φ_i to data structure at time i. Amortized cost of i^{th} operation is

$$\gamma_i = c_i + \Phi_i - \Phi_{i-1}$$

- That is, total amortized cost is the *actual cost* of the operation plus the *change in potential*

Potential Method

- suppose that the *actual cost* of each operation of an algorithm is c_i

  (which may be hard to track)

- potential method: assign potential Φ_i to data structure at time i. Amortized cost of i^{th} operation is

$$\gamma_i = c_i + \Phi_i - \Phi_{i-1}$$

- That is, total amortized cost is the *actual cost* of the operation plus the *change in potential*
- We have:

$$\sum_{i=1}^{n} \gamma_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \Phi_n - \Phi_0 + \sum_{i=1}^{n} c_i$$

total amortized cost

telescoping

actual cost

Potential Method

- suppose that the *actual cost* of each operation of an algorithm is c_i

  (which may be hard to track)

- potential method: assign potential Φ_i to data structure at time i. Amortized cost of i^{th} operation is

$$\gamma_i = c_i + \Phi_i - \Phi_{i-1}$$

- That is, total amortized cost is the *actual cost* of the operation plus the *change in potential*
- We have:

$$\sum_{i=1}^{n} \gamma_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \Phi_n - \Phi_0 + \sum_{i=1}^{n} c_i$$

≥ 0

- So if $\Phi_k - \Phi_0 \geq 0$ for all $k \geq 0$ (*valid potential function*) the total amortized cost is an *upper bound* on total cost.

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?
- Potential:

$$\Phi_i = \text{ number of bits with value 1 at step } i$$

$\phi_0 = 0$ 0000

$\phi_1 = 1$ 0001

$\phi_2 = 1$ 0010

$\phi_3 = 2$ 0011

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?
- Potential:

$$\Phi_i = \text{ number of bits with value 1 at step } i$$

- $\Phi_0 = 0$ and $\Phi_i = \underbrace{\# \text{ of 1 bits of } i}_{\geq 0} \geq 0$ (valid potential function)

$\forall k \quad \phi_k \geq \phi_0 = 0 \quad$ valid

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?
- Potential:

$$\Phi_i = \text{ number of bits with value 1 at step } i$$

- $\Phi_0 = 0$ and $\Phi_i = \# \text{ of 1 bits of } i \geq 0$ (valid potential function)
- What is the amortized cost of the i^{th} operation:
  - $c_i = (\# \text{ bits } 0 \to 1) + (\# \text{ bits } 1 \to 0)$ *cost*

set clearing

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?
- Potential:

$$\Phi_i = \text{ number of bits with value 1 at step } i$$

- $\Phi_0 = 0$ and $\Phi_i = \# \text{ of 1 bits of } i \geq 0$ (valid potential function)
- What is the amortized cost of the i^{th} operation:
  - $c_i = (\# \text{ bits } 0 \to 1) + (\# \text{ bits } 1 \to 0)$ *cost*
  - $\Phi_i - \Phi_{i-1} = (\# \text{ bits } 0 \to 1) - (\# \text{ bits } 1 \to 0)$ *potential*

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?
- Potential:

$$\Phi_i = \text{ number of bits with value 1 at step } i$$

- $\Phi_0 = 0$ and $\Phi_i = \#$ of 1 bits of $i \geq 0$ (valid potential function)
- What is the amortized cost of the i^{th} operation:
  - $c_i = (\#\text{ bits } 0 \to 1) + (\#\text{ bits } 1 \to 0)$ *cost*
  - $\Phi_i - \Phi_{i-1} = (\#\text{ bits } 0 \to 1) - (\#\text{ bits } 1 \to 0)$ *potential*
- Amortized cost:

$$\gamma_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times (\#\text{ bits } 0 \to 1)$$

= 1

One simple problem - potential method

- **Input:** A binary counter C initially set to zero
- **Output:** increment this counter up to n (a given integer)
- Is there a way to *assign potential function* to the *entire data structure* (i.e. the bits that we are incrementing)?
- Potential:

$$\Phi_i = \text{ number of bits with value 1 at step } i$$

- $\Phi_0 = 0$ and $\Phi_i = \#$ of 1 bits of $i \geq 0$ (valid potential function)
- What is the amortized cost of the i^{th} operation:
  - $c_i = (\#\text{ bits } 0 \to 1) + (\#\text{ bits } 1 \to 0)$ *cost*
  - $\Phi_i - \Phi_{i-1} = (\#\text{ bits } 0 \to 1) - (\#\text{ bits } 1 \to 0)$ *potential*
- Amortized cost:

$$\gamma_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times (\#\text{ bits } 0 \to 1)$$

- Since each increment *only changes 1 bit from 0 to 1* each amortized cost is 2.

Example of the potential method

$$\sum c_i \le \sum c_i + \underbrace{\Phi_n - \Phi_0}_{\text{valid}}$$

$$= \sum (c_i + \Phi_i - \Phi_{i-1})$$

$$= \sum \gamma_i = 2 \cdot n$$

Discussion of the potential method

Acknowledgements

- Lecture largely based on Jeff Erickson's notes (with exercises!)

  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/09-amortize.pdf

- More exercises and another example using all methods can also be found at the [CLRS] book, chapter 17. (see useful resources page)