

CS473 - Algorithms I



Lecture 12 Amortized Analysis

View in slide-show mode

Amortized Analysis

- Consider a sequence of operations, where
 - some operations are expensive,
 - some others are cheap.
- Key point: The time required to perform a sequence of operations is averaged over all operations performed.
- Amortized analysis can be used to show that:
 - ▣ The **average cost** of an operation **is small**
even though a single operation might be expensive
(when we average over a sequence of operations).

Amortized Analysis vs Average Case Analysis

- Amortized analysis **does not** use any *probabilistic reasoning*
- Amortized analysis guarantees the **average performance** of each operation in **the worst case**

Example: Stack Operations

PUSH (S, x): push object x onto stack

POP(S): pop the top of the stack S and return the popped object

MULTIPOP(S, k):

pop and return the **k top objects** of the stack S if $|S| \geq k$
or pop and return the **entire stack** if $|S| < k$

Runtimes:

PUSH(S, x): $\Theta(1)$

POP(S): $\Theta(1)$

MULTIPOP(S, k): $\Theta(\min(|S|, k))$

Stack Operations: Multipop

MULTIPOP(S , k)

while not StackEmpty(S) **and** $k \neq 0$ **do**

$t \leftarrow \text{POP}(S)$

$k \leftarrow k - 1$

return

Running time:

$\Theta(\min(s, k))$ where $s = |S|$

Runtime Analysis of Stack Operations

- We want to analyze a sequence of n POP, PUSH, and MULTIPOP operations on an initially empty stack.
- What is the worst-case runtime of a MULTIPOP operation?
 $O(n)$ *because the stack size is at most n*
- What is the worst-case runtime of a sequence of n operations?
 $O(n^2)$ *because we may have $O(n)$ MULTIPOPS, each costing $O(n)$*
- The analysis is correct, but it is not tight!

We can obtain a tighter bound by using amortized analysis.

Amortized Analysis Techniques

The most common three techniques

- The **aggregate method**
- The **accounting method**
- The **potential method**

If there are several types of operations in a sequence

- The **aggregate method** assigns
 - The same amortized cost to each operation
- The **accounting method** and the **potential method** may assign
 - Different amortized costs to different types of operations

The Aggregate Method

- Show that sequence of n operations takes
 - Worst case time $T(n)$ in total for all n
- The amortized cost (average cost in the worst case) per operation is therefore $T(n)/n$
- This amortized cost applies to each operation
 - Even when there are several types of operations in the sequence

The Aggregate Method: Stack Operations

- Aggregate method considers the entire sequence of n operations
 - Although a single **MULTIPOP** can be expensive
 - Any sequence of n **POP**, **PUSH**, and **MULTIPOP** operations on an initially empty stack can cost at most $O(n)$

Proof: Each object can be popped once for each time it is pushed. Hence the number of times that **POP** can be called on a nonempty stack including the calls within **MULTIPOP** is at most the number of **PUSH** operations, which is at most n

⇒ The amortized cost of an operation is the average $O(n)/n = O(1)$

Example: Incrementing a Binary Counter

- Implement k-bit binary counter that counts upward from 0
- Store the bits of counter in array $A[0..k-1]$, where

$\text{length}(A) = k$

$A[0]$: the least significant bit

$A[k-1]$: the most significant bit

001

- The binary value stored is:

$$x = \sum_{i=0}^{k-1} A[i].2^i$$

Binary Counter Increment

To add 1 (mod 2^k) to the counter:

INCREMENT(A, k)

$i \leftarrow 0$

while $i < k$ **and** $A[i] = 1$ **do**

$A[i] \leftarrow 0$

$i \leftarrow i + 1$

if $i < k$ **then**

$A[i] \leftarrow 1$

return

Same idea as the hardware implementation of a ripple-carry counter.

e.g. 000010011111 \Rightarrow
000010100000

Binary Counter Increment

To add 1 (mod 2^k) to the counter:

INCREMENT(A, k)

$i \leftarrow 0$

while $i < k$ **and** $A[i] = 1$ **do**

$A[i] \leftarrow 0$

$i \leftarrow i + 1$

if $i < k$ **then**

$A[i] \leftarrow 1$

return

Initially, $x = 0$

i.e. $A[i] = 0$ for all $0 \leq i \leq k$

What is the worst case runtime for INCREMENT(A, k) ?

$\Theta(k)$ when A contains all 1s

What is the worst case runtime of n INCREMENT operations starting from a zero counter?

$O(kn)$

NOT TIGHT!

The Aggregate Method: Incrementing a Binary Counter

Counter value	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Incre cost	Total cost
0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	0	2	3
3	0	0	0	0	0	0	1	1	1	4
4	0	0	0	0	0	1	0	0	3	7
5	0	0	0	0	0	1	0	1	1	8
6	0	0	0	0	0	1	1	0	2	10
7	0	0	0	0	0	1	1	1	1	11
8	0	0	0	0	1	0	0	0	4	15
9	0	0	0	0	1	0	0	1	1	16
10	0	0	0	0	1	0	1	0	2	18
11	0	0	0	0	1	0	1	1	1	19

Bits that
flip to
achieve the
next value
are shaded

The Aggregate Method: Incrementing a Binary Counter

- Note that, the running time of an increment operation is proportional to the number of bits flipped
- However, all bits are not flipped at each **INCREMENT**
 - $A[0]$ flips at each increment operation
 - $A[1]$ flips at alternate increment operations
 - $A[2]$ flips only once for 4 successive increment operations
 - \vdots
- In general, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n **INCREMENTS**

The Aggregate Method: Incrementing a Binary Counter

- Therefore, the total number of flips in the sequence is

$$\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = 2n$$

- The amortized cost of each operation is

$$O(n)/n = O(1)$$

The Accounting Method

- We assign **different charges** to different operations
 - Some operations are charged **more than their real cost**
 - Some are charged **less than their real cost**
- The amount charged for an operation is called its **amortized cost**.
- When the **amortized cost** of an operation **exceeds** its **actual cost**, the difference is assigned to specific objects in the data structure as **credit**.
- **Credit** can be used later to help pay for operations of which **amortized cost is less than their actual cost**.

Example: Accounting Method for Stack Operations

Suppose the unit cost of pushing or popping a stack element is \$1

Let's assign the following amortized costs:

PUSH: \$2

POP: \$0

MULTIPOP: \$0

Notes:

- Amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable
- All amortized costs are $O(1)$ in this example. In general, amortized costs of different operations **may differ asymptotically**.

Example: Accounting Method for Stack Operations

<u>Operation</u>	<u>Amortized Cost</u>	<u>Real Cost</u>	<u>Notes</u>
PUSH(47)	\$2	\$1	\$1 credit stored
PUSH(10)	\$2	\$1	\$1 credit stored
PUSH(39)	\$2	\$1	\$1 credit stored

39	\$1
10	\$1
47	\$1

STACK

Example: Accounting Method for Stack Operations

<u>Operation</u>	<u>Amortized Cost</u>	<u>Real Cost</u>	<u>Notes</u>
PUSH(47)	\$2	\$1	\$1 credit stored
PUSH(10)	\$2	\$1	\$1 credit stored
PUSH(39)	\$2	\$1	\$1 credit stored
POP()	\$0	\$1	\$1 credit used

39	\$1
10	\$1
47	\$1

STACK

Example: Accounting Method for Stack Operations

<u>Operation</u>	<u>Amortized Cost</u>	<u>Real Cost</u>	<u>Notes</u>
PUSH(47)	\$2	\$1	\$1 credit stored
PUSH(10)	\$2	\$1	\$1 credit stored
PUSH(39)	\$2	\$1	\$1 credit stored
POP()	\$0	\$1	\$1 credit used
PUSH(17)	\$2	\$1	\$1 credit stored
PUSH(23)	\$2	\$1	\$1 credit stored

23	\$1
17	\$1
10	\$1
47	\$1

STACK

Example: Accounting Method for Stack Operations

		<u>Operation</u>	<u>Amortized Cost</u>	<u>Real Cost</u>	<u>Notes</u>
		PUSH(47)	\$2	\$1	\$1 credit stored
		PUSH(10)	\$2	\$1	\$1 credit stored
		PUSH(39)	\$2	\$1	\$1 credit stored
23	\$1	POP()	\$0	\$1	\$1 credit used
17	\$1	PUSH(17)	\$2	\$1	\$1 credit stored
10	\$1	PUSH(23)	\$2	\$1	\$1 credit stored
47	\$1	MULTIPOP(3)	\$0	\$3	\$3 credit used

STACK

Example: Accounting Method for Stack Operations

<u>Operation</u>	<u>Amortized Cost</u>	<u>Real Cost</u>	<u>Notes</u>
PUSH(47)	\$2	\$1	\$1 credit stored
PUSH(10)	\$2	\$1	\$1 credit stored
PUSH(39)	\$2	\$1	\$1 credit stored
POP()	\$0	\$1	\$1 credit used
PUSH(17)	\$2	\$1	\$1 credit stored
PUSH(23)	\$2	\$1	\$1 credit stored
MULTIPOP(3)	\$0	\$3	\$3 credit used

47 \$1

STACK

sum of amortized costs \geq sum of real costs

Accounting Method for Stack Operations - Notes

- Intuitively:
 - For every **PUSH** operation, we pay \$2:
 - \$1 for the **real cost** of **PUSH**
 - \$1 **pre-payment** for the **future POP** of this item
(stored as **credit**)
 - Each **POP** operation (stand-alone or within **MULTI-POP**):
 - pays for the real cost by **using the credit** stored for the corresponding item.
 - The total credit is always nonnegative in a sequence of n operations starting with an empty stack.

The Accounting Method: Stack Operations

Thus by charging the **push** operation a little bit more we don't need to charge anything from the **pop** & **multipop** operations

We have ensured that the **amount of credit is always nonnegative**

- since each item in the stack always has \$1 of credit
- and the stack always has a nonnegative number of items

Thus, for any sequence of n **push**, **pop**, **multipop** operations the **total amortized cost** is an **upper bound** on the **total actual cost**

The Accounting Method (cont'd)

- The amortized cost of an operation can be considered as:

$$\text{amortized cost} = \text{actual cost} + \text{credit}$$

where credit is either deposited (positive) or used (negative)

- Key point in accounting method:
 - The total amortized cost of a sequence of operations must be an upper bound on the total actual cost.
 - This relationship must hold for all sequences of operations

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

a_i : amortized cost of operation i
 c_i : real cost of operation i

The Accounting Method (cont'd)

For any sequence of n operations, we must have:

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

a_i : amortized cost of operation i
 c_i : real cost of operation i

The total credit stored after n operations is:

$$total_credit = \sum_{i=1}^n a_i - \sum_{i=1}^n c_i$$

For the above inequality to hold, the total credit must be nonnegative at all times.

The Accounting Method - Summary

- Assign an amortized cost for each operation.
- For operation i , let a_i and c_i be the amortized and the real cost of i .
 - If $a_i > c_i \Rightarrow$ store $(a_i - c_i)$ as credit
 - If $a_i < c_i \Rightarrow$ use $(c_i - a_i)$ stored credit
- If we never run out of credit in a sequence of n operations, we can say that:

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

a_i : amortized cost of operation i
 c_i : real cost of operation i

In other words, the sum of amortized costs for n operations is an upper bound for the sum of real costs.

Accounting Method Example: Binary Counter Increment

- *Reminder: The running time of an increment operation is proportional to the # of bits flipped.*
- Analyze using accounting method:
 - ▣ Charge an amortized cost of \$2 to set a bit from 0 to 1, and \$0 to set a bit from 1 to 0.
 - ▣ Intuition: When a bit is set to 1
 - We use \$1 to pay for the actual cost of setting the bit to 1
 - We place the other \$1 on the bit as credit.
 - At any point, every 1-bit in the counter has \$1 credit on it
 - Hence, we don't need to charge anything to reset a bit to 0
 - We just pay for the reset with the \$1 on it.

Accounting Method Example: Binary Counter Increment

<u>Binary Counter</u>	<u>Amortized Cost</u>	<u>Real Cost</u>	<u>Notes</u>
0 0 0 0 0	\$2	\$1	\$1 credit stored for bit 0
0 0 0 0 1	\$0+\$2	\$1 + \$1	\$1 credit used for bit 0 \$1 credit stored for bit 1
0 0 0 1 0	\$2	\$1	\$1 credit stored for bit 0
0 0 0 1 1	\$0+\$0+\$2	\$1+\$1+\$1	\$1 credit used for bit 0 \$1 credit used for bit 1 \$1 credit stored for bit 2
0 0 1 0 0			

Accounting Method Example: Binary Counter Increment

INCREMENT(A, k)

$i \leftarrow 0$

while $i < k$ **and** $A[i] = 1$ **do**

$A[i] \leftarrow 0$

$i \leftarrow i + 1$

if $i < k$ **then**

$A[i] \leftarrow 1$

return

- The amortized cost of **setting bits to 0** in the first while loop:

$\$0$

(the real cost is paid by the credits)

- The amortized cost of **setting a single bit to 1** at the end:

$\$2$

(\$1 is stored as credit for the bit)

- Total amortized cost for an **INCREMENT** operation?

$\$2$

Accounting Method Example: Binary Counter Increment

- For any sequence of n **INCREMENT** operations starting from counter value 0:

- ▣ The credit never goes negative

We have \$1 stored as credit for each bit-1.

We can use the stored credit to flip each bit to 0.

- ▣ So, we have:

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

a_i : amortized cost of operation i
 c_i : real cost of operation i

In other words, the sum of amortized costs for n operations is an upper bound for the sum of real costs.

Accounting Method Example: Binary Counter Increment

□ So, we have showed that:

For n increment operations:

the total amortized cost is $O(n)$.

This amortized cost is an upper bound for the actual cost

The Potential Method

Accounting method represents **prepaid work** as **credit** stored with **specific objects in the data structure**

Potential method represents the **prepaid work** as **potential energy or just potential** that can be released to pay for the future operations

The **potential** is **associated with the data structure as a whole** rather than with specific objects within the data structure

The Potential Method

We start with an initial data structure D_0 and perform n operations.

For $1 \leq i \leq n$, let:

C_i : the actual cost of the i^{th} operation

D_i : data structure that results after applying i^{th} operation to D_{i-1}

ϕ : potential function that maps each data structure D_i to a real number $\phi(D_i)$

$\phi(D_i)$: the potential associated with data structure D_i

\hat{C}_i : amortized cost of the i^{th} operation w.r.t. function ϕ

The Potential Method

$$\hat{C}_i = \underbrace{C_i}_{\text{actual cost}} + \underbrace{\phi(D_i) - \phi(D_{i-1})}_{\text{increase in potential due to the operation}}$$

The total amortized cost of n operations is

$$\begin{aligned} \sum_{i=1}^n \hat{C}_i &= \sum_{i=1}^n (C_i + \phi(D_i) - \phi(D_{i-1})) \\ &= \sum_{i=1}^n C_i + \phi(D_n) - \phi(D_0) \end{aligned}$$

The Potential Method

If we can ensure that $\phi(D_n) \geq \phi(D_0)$ then
the **total amortized cost** $\sum_{i=1}^n \hat{C}_i$ is an **upper bound** on the
total actual cost

However, $\phi(D_n) \geq \phi(D_0)$ should hold for all possible n
since, in practice, we do not always know n in advance

Hence, if we require that $\phi(D_i) \geq \phi(D_0)$, for all i , **then** we ensure
that the total amortized cost is an upper bound for the total cost

The Potential Method: Stack Operations

- Define $\phi(S) = |S|$, the number of objects in the stack
- For the initially empty stack, we have $\phi(D_0) = 0$
- Since $|S| \geq 0$, stack D_i that results after i th operation has nonnegative potential for all i , that is

$$\phi(D_i) \geq 0 = \phi(D_0) \text{ for all } i$$

- Hence, the total amortized cost is an **upper bound** on the **total actual cost**

The Potential Method for Stack Operations

<u>Operation</u>	<u>Real Cost</u>	<u>Potential</u>	<u>Amortized Cost</u>
		0	
PUSH(47)	1	1	2
PUSH(10)	1	2	2
PUSH(39)	1	3	2

39
10
47

STACK

The Potential Method for Stack Operations

<u>Operation</u>	<u>Real Cost</u>	<u>Potential</u>	<u>Amortized Cost</u>
		0	
PUSH(47)	1	1	2
PUSH(10)	1	2	2
PUSH(39)	1	3	2
POP()	1	2	0

39

10

47

STACK

The Potential Method for Stack Operations

<u>Operation</u>	<u>Real Cost</u>	<u>Potential</u>	<u>Amortized Cost</u>
		0	
PUSH(47)	1	1	2
PUSH(10)	1	2	2
PUSH(39)	1	3	2
POP()	1	2	0
PUSH(17)	1	3	2
PUSH(23)	1	4	2

23
17
10
47

STACK

The Potential Method for Stack Operations

23
 17
 10
 47
 STACK

<u>Operation</u>	<u>Real Cost</u>	<u>Potential</u>	<u>Amortized Cost</u>
		0	
PUSH(47)	1	1	2
PUSH(10)	1	2	2
PUSH(39)	1	3	2
POP()	1	2	0
PUSH(17)	1	3	2
PUSH(23)	1	4	2
MULTIPOP(3)	3	1	0

The Potential Method for Stack Operations

<u>Operation</u>	<u>Real Cost</u>	<u>Potential</u>	<u>Amortized Cost</u>
		0	
PUSH(47)	1	1	2
PUSH(10)	1	2	2
PUSH(39)	1	3	2
POP()	1	2	0
PUSH(17)	1	3	2
PUSH(23)	1	4	2
MULTIPOP(3)	3	1	0

47

STACK

sum of amortized costs \geq sum of real costs

The Potential Method for Stack Operations

Reminder: $\phi(D_i)$: The number of objects in stack after operation i

PUSH(S, x):

$$\phi(D_i) - \phi(D_{i-1}) = 1 \quad (\text{because the stack size increases by 1})$$

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$$

Amortized cost of PUSH operation is 2

POP(S):

$$\phi(D_i) - \phi(D_{i-1}) = -1 \quad (\text{because the stack size decreases by 1})$$

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1}) = 1 - 1 = 0$$

Amortized cost of POP operation is 0

The Potential Method for Stack Operations

Reminder: $\phi(D_i)$: The number of objects in stack after operation i

MULTIPOP(S, k):

$$\phi(D_i) - \phi(D_{i-1}) = -k', \text{ where } k' = \min\{|S|, k\}$$

because the stack size decreases by k'

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1}) = k' - k' = 0$$

Amortized cost of MULTIPOP operation is 0

The amortized cost of each operation is $O(1)$.

Thus, the amortized cost of a sequence of n operations is $O(n)$

The Potential Method - Intuition

If $\phi(D_i) - \phi(D_{i-1}) > 0$, then:

Amortized cost \hat{C}_i is an **overcharge** for the i^{th} operation.

The **potential** of the data structure **increases**.

If $\phi(D_i) - \phi(D_{i-1}) < 0$, then:

Amortized cost \hat{C}_i is an **undercharge** for the i^{th} operation.

The **actual cost** of the operation is **paid** by the **decrease in potential**.

The Potential Method - Intuition

Different potential functions may yield different amortized costs.

The best potential function to use depends on the desired time bounds.

Choose a potential function such that $\phi(D_i) - \phi(D_0) \geq 0$ for all i values. This ensures that the amortized cost of any i operations is an upper bound for the actual cost.

Practical guideline:

Choose a potential function that increases a little after every cheap operation, and decreases a lot after an expensive operation.

The Potential Method for Binary Counter Increment

- Define $\phi(D_i) = b_i$

where b_i : *number of 1s in the counter after the i^{th} operation*

- The actual cost of **INCREMENT** operation:

$$C_i = (\# \text{ bits changed } 0 \Rightarrow 1) + (\# \text{ of bits changed } 1 \Rightarrow 0)$$

- The potential change after the i^{th} **INCREMENT** operation:

$$\phi(D_i) - \phi(D_{i-1}) = (\# \text{ of bits changed } 0 \Rightarrow 1) - (\# \text{ of bits changed } 1 \Rightarrow 0)$$

- Amortized cost of the i^{th} **INCREMENT** operation:

$$\begin{aligned}\hat{C}_i &= C_i + \phi(D_i) - \phi(D_{i-1}) \\ &= 2 \cdot (\# \text{ of bits changed from } 0 \Rightarrow 1)\end{aligned}$$

The Potential Method for Binary Counter Increment

- Amortized cost of the i^{th} **INCREMENT** operation:

$$\hat{C}_i = 2 \cdot (\# \text{ of bits changed from } 0 \Rightarrow 1)$$

- In one **INCREMENT** operation, we change at most 1 bit $0 \Rightarrow 1$

- Hence, the amortized cost of an **INCREMENT** operation:

$$\hat{C}_i \leq 2$$

The Potential Method for Binary Counter Increment

<u>Binary Counter</u>	<u>Real Cost</u>	<u>Potential</u>	<u>Amortized Cost</u>
0 0 0 0 0		0	
0 0 0 0 1	1	1	2
0 0 0 1 0	2	1	2
0 0 0 1 1	1	2	2
0 0 1 0 0	3	1	2

The Potential Method: Incrementing a Binary Counter

- If the counter starts at zero, then $\phi(D_0) = 0$, the number of 1s in the counter after the i th operation
- Since $\phi(D_i) \geq 0$ for all i the total amortized cost is an upper bound on the total actual cost
- Hence, the worst-case cost of n operations is $O(n)$

The Potential Method for Binary Counter Increment

- What if the counter does not start from zero (*i.e.* $b_0 \neq 0$)?
- For a sequence of n **INCREMENT** operations, can we say that the sum of the amortized costs is an upper bound for the sum of the actual costs?

No, because:

$$\sum_{i=1}^n \hat{C}_i = \sum_{i=1}^n C_i + f(D_n) - f(D_0)$$

and $\phi(D_0) = b_0 \neq 0$.

So, $\phi(D_n) - \phi(D_0)$ is not necessarily ≥ 0

Reminder: $\phi(D_i) = b_i$

where b_i : number of 1s in the counter after the i^{th} operation

The Potential Method for Binary Counter Increment

- What if the counter does not start from zero (*i.e.* $b_0 \neq 0$)?
- For a sequence of n INCREMENT operations we can write:

$$\begin{aligned} \sum_{i=1}^n C_i &= \sum_{i=1}^n \hat{C}_i - f(D_n) + f(D_0) \\ &\leq 2n - b_n + b_0 \quad (\text{because } \hat{C}_i \leq 2 \text{ for all } i) \end{aligned}$$

Note: $b_0 \leq k$, where k is the number of bits of the counter.

If we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost will be $O(n)$, no matter what the initial counter value is.

Amortized Analysis - Summary

- With amortized analysis, we show that the average cost of an operation is small if we average over a sequence of operations (even though some single operations may be expensive).
- We studied 3 techniques for amortized analysis:
 - ▣ Aggregate Method
 - ▣ Accounting Method
 - ▣ Potential Method

Amortized Analysis - Summary

□ Aggregate Method:

- ▣ Directly compute the sum of n operations.
- ▣ Then, compute the average.

□ Accounting Method:

- ▣ Pay a little extra for the cheap operations and store the difference as credit on certain items
- ▣ Pay for the expensive operations using the stored credit.
- ▣ As long as we never run out of credits:
 - The total amortized cost is guaranteed to be an upper bound for the total actual cost.

Amortized Analysis - Summary

□ Potential Method:

- ▣ Similar to the accounting method, but a potential function is defined for the whole data structure instead of individual items.
- ▣ The potential is 0 initially.
- ▣ It increases slowly with every cheap operation.
- ▣ Expensive operations are paid using the potential stored.
- ▣ Amortized cost is the actual cost plus the change in potential.
- ▣ As long as potential is always nonnegative:

The total amortized cost is guaranteed to be an upper bound for the total actual cost.