

CS473-Algorithms I

Lecture 12

Amortized Analysis

Amortized Analysis

Key point: The time required to perform a sequence of data structure operations is **averaged** over all operations performed

- Amortized analysis can be used to show that
 - The **average cost** of an operation **is small**
 - If one averages over a sequence of operations even though a single operation might be expensive

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**

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

Example: Stack Operations

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

POP(S): pops the top of the stack S and returns the popped object

MULTIPOP(S, k): removes the k top objects of the stack S or pops the entire stack if $|S| < k$

- **PUSH** and **POP** runs in $\Theta(1)$ time
 - The total cost of a sequence of n **PUSH** and **POP** operations is therefore $\Theta(n)$
- The running time of **MULTIPOP**(S, k) is
 - $\Theta(\min(s, k))$ where $s = |S|$

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 $s = |S|$

The Aggregate Method: Stack Operations

- Let us analyze a sequence of n POP, PUSH, and MULTIPOP operations on an initially empty stack
- The worst case of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n
- Hence, a sequence of n operations costs $O(n^2)$
 - we may have n MULTIPOP operations each costing $O(n)$
- The analysis is correct, however,
 - Considering worst-case cost of each operation, it is **not tight**
- We can obtain a better bound by using aggregate method of amortized analysis

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

- Implementing a k -bit binary counter that counts upward from 0
- Use array $A[0\dots k-1]$ of bits as the counter where $\text{length}[A]=k$;

$A[0]$ is the least significant bit;

$A[k-1]$ is the most significant bit;

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

Binary Counter: Increment

Initially $x = 0$, i.e., $A[i] = 0$ for $i = 0, 1, \dots, k-1$

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
```

Essentially same as the one implemented in hardware by a *ripple-carry counter*

A single execution of increment takes $\Theta(k)$ in the worst case in which array A contains all 1's

Thus, n increment operations on an initially zero counter takes $O(kn)$ time in the worst case.

≧**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

⋮

- 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

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

- The amortized cost of each operation is

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

The Accounting Method

We assign **different charges** to **different operations** with some operations **charged more or less** than they **actually cost**

The amount we **charge** 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 whose **amortized cost** is **less** than their **actual cost**

That is, amortized cost of an operation can be considered as being split between its actual cost and credit (either deposited or used)

The Accounting Method

Key points in the accounting method:

- The total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence
- This relationship must hold for all sequences of operations

Thus, the total credit associated with the data structure must be nonnegative at all times

Since it represents the amount by which the total amortized cost incurred so far **exceed** the total actual cost incurred so far

The Accounting Method: Stack Operations

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 $O(1)$, however, in general, amortized costs of different operations may differ asymptotically

Suppose we use \$1 bill to represent each unit of cost

The Accounting Method: Stack Operations

We start with an empty stack of plates

When we push a plate on the stack

- we use **\$1** to pay the actual cost of the push operation
- we put a **credit** of **\$1** on top of the pushed plate

At any time point, every plate on the stack has a **\$1** of **credit** on it

The **\$1** stored on the plate is a **prepayment** for the cost of **popping** it

In order to pop a plate from the stack

- we take **\$1 of credit** off the plate
- and use it to pay the **actual cost** of the **pop** operation

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 credits is always nonnegative**

- since each plate on the stack always has \$1 of credit
- and the stack always has a nonnegative number of plates

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: Stack Operations

Incrementing a binary counter:

Recall that, the running time of an **increment** operation is **proportional** to the **number of bits flipped**

Charge an amortized cost of \$2 to set a bit to 1

When a bit is set

- we use \$1 to pay for the **actual setting** of the bit and
- we place the other \$1 on the bit as **credit**

At any time point, every 1 in the counter has a \$1 of 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

The Accounting Method: Stack Operations

The amortized cost of increment can now be determined
the cost of resetting bits within the **while** loop
is paid by the dollars on the bits that are reset

At most one bit is set to 1, in an **increment** operation

Therefore, the amortized cost of an **increment** operation
is at most **2 dollars**

The number of 1's in the counter is never negative, thus
the amount of credit is always nonnegative

Thus, for n **increment operations**, the **total amortized cost**
is $O(n)$, which **bounds 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 on which we perform n operations

For each $i = 1, 2, \dots, n$, 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 $\phi(D_i) \geq \phi(D_0)$ 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 we **pay in advance** (as in the accounting method)

The Potential Method

If $\phi(D_i) - \phi(D_{i-1}) > 0$, then the amortized cost \hat{C}_i represents

- an **overcharge** to the i -th operation and
- the **potential** of the data structure **increases**

If $\phi(D_i) - \phi(D_{i-1}) < 0$, then the amortized cost \hat{C}_i represents

- an **undercharge** to the i -th operation and
- the **actual cost** of the operation is **paid**
by the **decrease in potential**

Different potential functions may yield different amortized costs
which are still upper bounds for the actual costs

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

The Potential Method: Stack Operations

- Define $\phi(S) = |S|$, the number of objects in the stack
- For the initial 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

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

- total amortized cost is an **upper bound** on **total actual cost**
- Let us compute the amortized costs of stack operations where i th operation is performed on a stack with s objects

The Potential Method: Stack Operations

$$\text{PUSH}(S): \phi(D_i) - \phi(D_{i-1}) = (s+1) - (s) = 1$$

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

$$\text{MULTIPOP}(S, k): \phi(D_i) - \phi(D_{i-1}) = -k' = -\min\{s, k\}$$

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

$$\text{POP}(S): \hat{C}_i = 0, \text{ similarly}$$

- The **amortized cost of each operation** is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$

The Potential Method: Incrementing a Binary Counter

- Define $\phi(D_i) = b_i$, number of 1s in the counter after the i th operation
- Compute the amortized cost of an **INCREMENT** operation wrt ϕ
- Suppose that i th **INCREMENT** resets t_i bits then,

$$t_i \leq C_i \leq t_i + 1$$

- The number of 1s in the counter after the i th operation is

$$b_{i-1} - t_i \leq b_i \leq b_{i-1} - t_i + 1 \Rightarrow b_i - b_{i-1} \leq 1 - t_i$$

- The amortized cost is therefore

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

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: Incrementing a Binary Counter

- Assume that the counter does not start at zero, i.e., $b_0 \neq 0$
- Then, after n **INCREMENT** operations the number of 1s is b_n , where $0 \leq b_0, b_n \leq k$

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

- Since $b_0 \leq k$, if we execute at least $n = \Omega(k)$ **INCREMENT** operations the total actual cost is $O(n)$
- No matter what initial value the counter contains