ICCS312 Algorithms and Tractability (Term 1/2021-22)

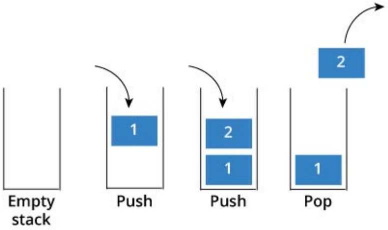
Lecture 5: Amortized Analysis

Sunsern Cheamanunkul

Mahidol University International College

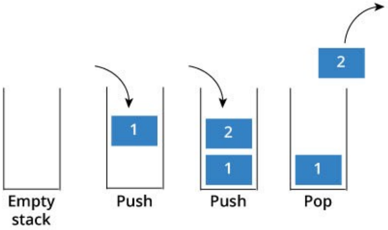
September 28, 2021

Stack



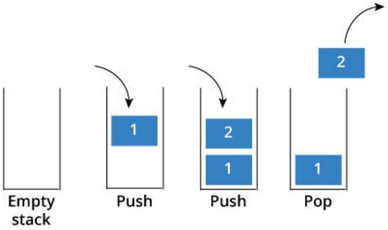
- Stack Operations:
  - Push(S,x) pushes object x onto stack S

Stack



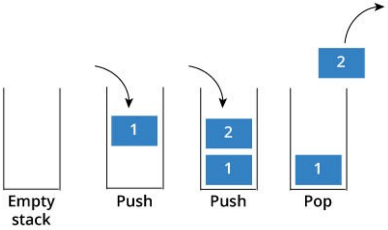
- Stack Operations:
  - Push(S,x) pushes object x onto stack S
  - Pop(S) pops the top of (non-empty) stack S

Stack



- Stack Operations:
  - Push(S,x) pushes object x onto stack S
  - Pop(S) pops the top of (non-empty) stack S
  - MultiPop(S,k) pops the k top objects from S

Stack



- Stack Operations:
  - Push(S,x) pushes object x onto stack S
  - Pop(S) pops the top of (non-empty) stack S
  - MultiPop(S,k) pops the k top objects from S
- What is the largest possible cost of a sequence of n stack operations?

Stack

- What is the largest possible cost of a sequence of n stack operations?

Stack

- What is the largest possible cost of a sequence of n stack operations?
- Simple worst-case bound analysis
  - The largest cost is $\Theta(n)$ from MultiPop
  - There are n stack operations
  - Total cost is $\Theta(n^2)$

Stack

- What is the largest possible cost of a sequence of n stack operations?
- Simple worst-case bound analysis
  - The largest cost is $\Theta(n)$ from MultiPop
  - There are n stack operations
  - Total cost is $\Theta(n^2)$
- Correct, but it's not tight (too pessimistic).

Stack

- What is the largest possible cost of a sequence of n stack operations?
- Simple worst-case bound analysis
  - The largest cost is $\Theta(n)$ from MultiPop
  - There are n stack operations
  - Total cost is $\Theta(n^2)$
- Correct, but it's not tight (too pessimistic).
- We can show that the **amortized** cost for each operation is just $\Theta(1)$.

Amortized Analysis

- Analyze a **sequence** of n operations
- Show that the *averaged* cost for each operation is small.

Amortized Analysis

- Analyze a **sequence** of n operations
- Show that the *averaged* cost for each operation is small.

Def.

The **amortized cost per operation** for a sequence of n operations is the total cost of the operations divided by n.

Amortized Analysis

- Analyze a **sequence** of n operations
- Show that the *averaged* cost for each operation is small.

Def.

The **amortized cost per operation** for a sequence of n operations is the total cost of the operations divided by n.

- Ex: If we have 100 operations at cost 1, followed by 1 operation at cost 100, the *amortized* cost per operation is $200/101 < 2$.

Amortized Analysis

- Three main techniques:
  1. The aggregate method – analyzes the total running time for a sequence of operations.
  2. The accounting method – charges extra and use to pay for future operations.
  3. The potential method – derives a *potential function* to describe the cost of the operations.

Stack with an extensible array

- Consider a stack implemented with an array.
- We have an array A with a variable `top` that points to the top of the stack.
- Operations:
  1. Push
     ```
     A[top] = x;
     top++;
     ```
  2. Pop
     ```
     top--;
     return A[top];
     ```

Stack with an extensible array

- When the array is full, we need to allocate a new larger array and copy the data over.
  - Sounds expensive!
- Some push operation will be more expensive than others.
- Luckily this doesn't happen too often.
- What can we say about the *amortized* cost for push?

Stack with an array

- Let's define our cost model first.

Cost model

- Write an item into the array costs 1.
- Read an item from the array costs 1.
- Cost for the growing array is the number of elements copied over.

Policy 1: Grow one by one

- When the array is full, increase its size by 1.

Policy 1: Grow one by one

- When the array is full, increase its size by 1.
- Consider n push operations. The total cost will be $1 + 2 + 3 + \ldots + n = n(n + 1)/2$.

Policy 1: Grow one by one

- When the array is full, increase its size by 1.
- Consider n push operations. The total cost will be $1 + 2 + 3 + \ldots + n = n(n+1)/2$.
- The *amortized* cost per operation is $(n+1)/2$ or $O(n)$.

Policy 2: Doubling

- When the array is full, double its size.

Policy 2: Doubling

- When the array is full, double its size.
- Consider n push operations.

Policy 2: Doubling

- When the array is full, double its size.
- Consider n push operations.
- Two types of costs:
  1. Total resize cost: $1 + 2 + 4 + 8 + \ldots + 2^i$ for some $2^i < n$.
     - The sum is at most $2n - 1$.
  2. Total push cost is n from n pushes.
- Therefore, the total cost is $3n - 1 < 3n$.

Policy 2: Doubling

- When the array is full, double its size.
- Consider n push operations.
- Two types of costs:
  1. Total resize cost: $1 + 2 + 4 + 8 + \ldots + 2^i$ for some $2^i < n$.
     - The sum is at most $2n - 1$.
  2. Total push cost is n from n pushes.
- Therefore, the total cost is $3n - 1 < 3n$.
- The *amortized* cost per operation is < 3 or $O(1)$.

Accounting method

- Idea: show that we have enough money pay for the total cost of n operations out of a bank account.
- Charge extra for *cheap* operations to pay for more *expensive* operations later.

Accounting method

- Idea: show that we have enough money pay for the total cost of n operations out of a bank account.
- Charge extra for *cheap* operations to pay for more *expensive* operations later.
- Let c_i be the true cost of the i-th operation.
- Show that by charging some cost c_i' for the i-th operation, we have

$$\sum_{1 \leqslant i \leqslant n} c_i \leqslant \sum_{1 \leqslant i \leqslant n} c_i'$$

- The *amortized* cost per operation is $(\sum_{1 \leqslant i \leqslant n} c_i')/n$.

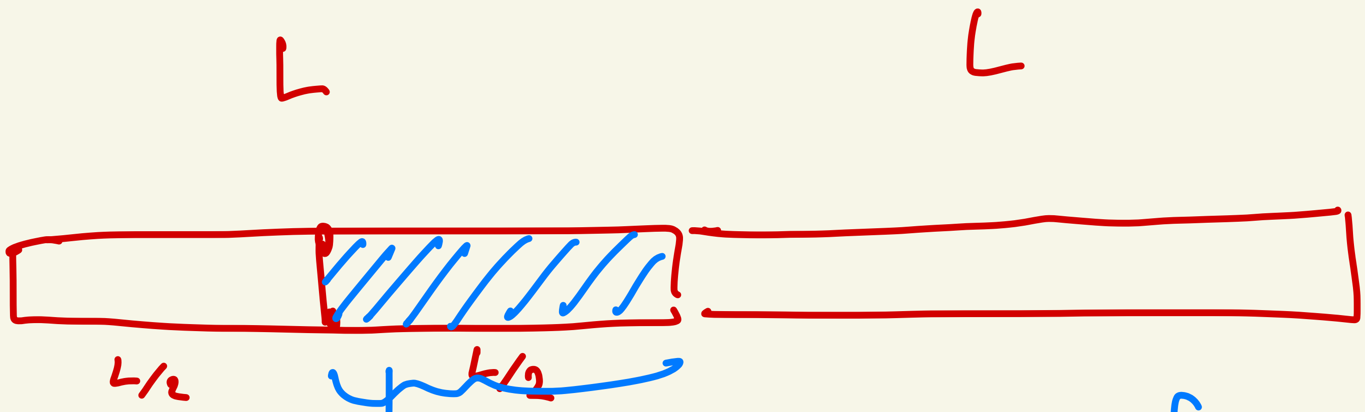
Accounting method

Claim.

With the doubling policy, if we charge \$3 for each push, we have enough money to pay for all the costs incurred during the n operations.

Proof.

For each push, we use \$1 to pay for the write right away and put \$2 into our account. Anytime, we need to double the array, from L to $2L$, we pay for it using the money in our bank account. We know we will always have enough money is because, after the last resizing, there were $L/2$ elements in the arrays. So there must be at least $L/2 * 2$ dollars in the account. ∎



$1 for push

pay $3

$\underline{\underline{\$ 2}}$ saved in the account

since last resize,

$\$(L/2 \times 2)$ - to pay for resizing

$L \rightarrow 2L$

Accounting method

Claim.

With the doubling policy, if we charge \$3 for each push, we have enough money to pay for all the costs incurred during the n operations.

Proof.

For each push, we use \$1 to pay for the write right away and put \$2 into our account. Anytime, we need to double the array, from L to $2L$, we pay for it using the money in our bank account. We know we will always have enough money is because, after the last resizing, there were $L/2$ elements in the arrays. So there must be at least $L/2 * 2$ dollars in the account. ∎

- Hence, the *amortized* cost per operation is 3.

Potential method

- Similar to the accounting method, but this looks at states of the data structure.
- Let h_i be the state of our data structure after the i-th operation and h_0 be the initial state.

Definition

Suppose we define a potential function Φ on states of a data structure such that $\Phi(h_0) = 0$ and $\Phi(h_i) \geq 0$ for all states h_i.

The *amortized* cost $\tilde{c}$ of an operation is given by

$$\tilde{c} = c + (\Phi(h') - \Phi(h))$$

where c is the actual cost of the operation and h and h' is the states before and after the operation.

Stack with a fixed-size array

- Recall Push(S,x), Pop(), and MultiPop(S,x,k)
- Propose: $\Phi(h_i) =$ number of items in the stack after the i-th operation.
- Observation:
  - $\Phi(h_0) = 0$ as initial stack is empty.
  - $\Phi(h_i) \geqslant 0$ as the number of items in the stack cannot be negative.

Stack with a fixed-size array

$\Phi(h_i)$ = number of items in the stack after the i-th operation.

- Push
  - actual cost $c_i = 1$
  - potential change: $\Phi(h_i) - \Phi(h_{i-1}) = 1$
  - amortized cost: $\tilde{c} = c_i + (\Phi(h_i) - \Phi(h_{i-1})) = 1 + 1 = 2$

Stack with a fixed-size array

$\Phi(h_i)$ = number of items in the stack after the i-th operation.

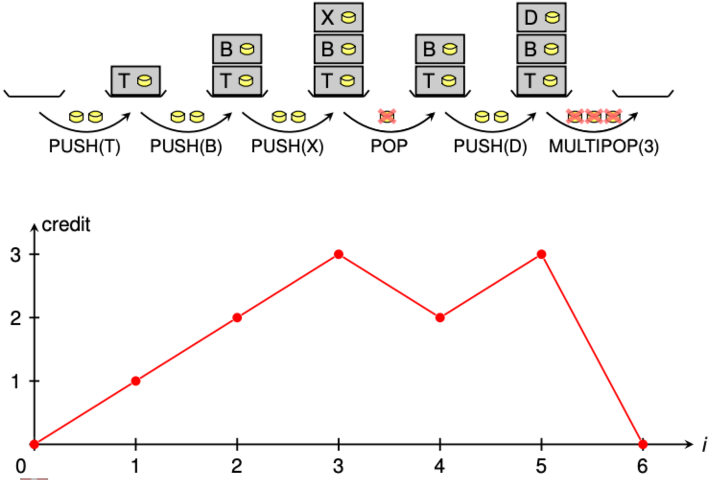
- Pop
  - actual cost $c_i = 1$
  - potential change: $\Phi(h_i) - \Phi(h_{i-1}) = -1$
  - amortized cost: $\tilde{c} = c_i + (\Phi(h_i) - \Phi(h_{i-1})) = 1 - 1 = 0$

Stack with a fixed-size array

$\Phi(h_i)$ = number of items in the stack after the i-th operation.

- MultiPop
  - actual cost $c_i = min(k, |S|)$
  - potential change: $\Phi(h_i) - \Phi(h_{i-1}) = -min(k, |S|)$
  - amortized cost: $\tilde{c} = c_i + (\Phi(h_i) - \Phi(h_{i-1})) = 0$

Stack with a fixed-size array



Example - Binary Counter

- Imagine we want to store a big binary counter in an array A
- Operation: Inc() – increment the counter by 1.
- Cost for flipping each bit is 1

count val	A[m]	A[m-1]	...	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	...	0	0	0	0	
								$1
1	0	0	...	0	0	0	1	
								$2
2	0	0	...	0	0	1	0	
								$1
3	0	0	...	0	0	1	1	
								$3
4	0	0	...	0	1	0	0	
								$1
5	0	0	...	0	1	0	1	
								$2

Example - Binary Counter

- Simple wort-case bound:
    - $O(\log n)$ as each increment changes at most $\log n$
- Let's do amortized analysis of each increment.

Example - Binary Counter

Aggregate method

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- How often do we flip each bit?

Example - Binary Counter

- A[0] is flipped every time.
- A[1] is flipped every other time.
- A[2] is flipped every 4th time, and so on.

Example - Binary Counter

- A[0] is flipped every time.
- A[1] is flipped every other time.
- A[2] is flipped every 4th time, and so on.
- So, after n increments,
  - The total number of A[0] flips is n
  - The total number of A[1] flips is $n/2$
  - The total number of A[2] flips is $n/4$, and so on.

Example - Binary Counter

- A[0] is flipped every time.
- A[1] is flipped every other time.
- A[2] is flipped every 4th time, and so on.
- So, after n increments,
  - The total number of A[0] flips is n
  - The total number of A[1] flips is $n/2$
  - The total number of A[2] flips is $n/4$, and so on.
- Total cost is $n + n/2 + n/4 + \ldots \leqslant 2n$.
- Thus, the *amortized* cost for each increment is $\leqslant 2$.

Example - Binary Counter

Accounting method

- Imagine each bit having its own bank account.
- Pay \$2 every time we flip $0 \to 1$.
  - \$1 for the actual cost and put \$1 into the account.
- When we flip $1 \to 0$, use the money from the account.
- We will always have enough money to pay for the increments.
- Hence, the amortized cost per increment is \$2.

* Each increment has exactly one $0 \to 1$ flip.

Example - Binary Counter

Potential method

$\Phi(h_i)$ = the number of 1-bits in the current count.

- Easy to check that:
  - $\Phi(h_0) = 0$
  - $\Phi(h_i) \geqslant 0$ for all h_i
- Proof: Leave this as exercise to you!

Discussion

- Suppose now it costs 2^k to flip the k-th bit
- In a sequence of n increments,
  - Simple worst-case bound gives $O(n)$ per increment.
  - What is the *amortized* cost per increment?

$2^2=4$ $2^1=2$ $2^0=1$

0 0 0
$1
0 0 1
$3
0 1 0
$1
0 1 1
$7
1 0 0

total cost $\le 2^0 \cdot n + 2^1 \dfrac{n}{2} + 2^2 \cdot \dfrac{n}{2^2} + \ldots + 2^k \dfrac{n}{2^k}$

where $k \le \lceil \log n \rceil$ ← we perform n increments only $\lceil \log n \rceil$ bits are affected.

$\le \underbrace{n + n + \ldots + n}_{(k+1) \text{ term}}$

$= n \cdot (k+1) \le n \cdot (\lceil \log n \rceil + 1)$

So, amortized cost per increment is

$$\frac{n \lceil \log n \rceil}{n} = O(\log n)$$

∎

		bits affected
0	000	0
1	001	1
2	010	2
3	011	2
4	100	3
5	101	3