Amortization

Last time:

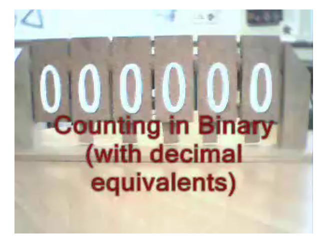
- We showed how we could get linear expected worst-case by sweeping things under the rug….
- Today, we will show how to improve analysis by carefully considering sequences of action.

Let's count!

- How hard is counting?
- [...in binary?]

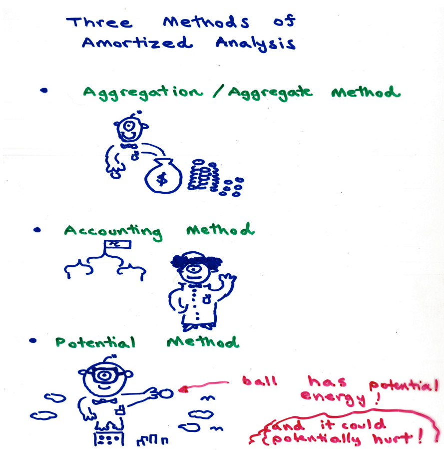
Amortization

- Idea:
  - Sometimes we don't care about the exact, potentially expensive cost for each operation
  - We do care about a *sequence* of operations
- Goal: show that, *on average*, cost per operation is small



Source: youtube.com

- Average
  - In the *worst-case*
  - Not over a distribution of inputs



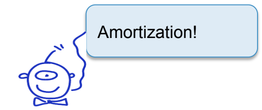
Amortization:

- Sometimes we don't care about the cost for one operation (but we do care about the cost of a sequence of operations!)

Examples:

- Multipop Stack
- Binary Counting
- Extendible ADT

Multipop Stack ADT



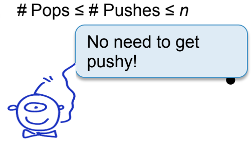
- PUSH(S, x): $O(1)$ each
- POP(S): $O(1)$ each
- MULTIPOP(S, k)
  while S is not empty and $k > 0$
  POP(S)
  $k = k - 1$

- Running time of multipop
  - Linear in number of pops
  - # of iterations of while loop is min(*s*,*k*) where *s*=|*S*|
  - Total cost:
- Total cost of sequence of n operations:

Sequencing Stack Operations

- Consider any sequence of *n* stack operations:
  - Push()
  - Pop()
  - Push()
  - Multipop()
  - Multipop()
  - Push()

Pops ≤ # Pushes ≤ *n*



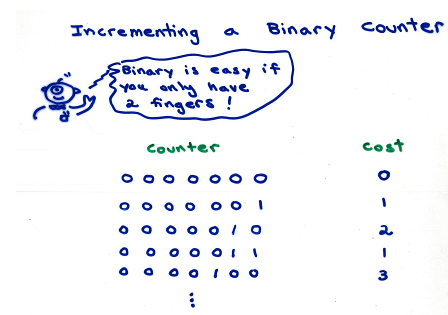
- But wait…
  - Each item will get popped from the stack at most once per push operation
  - Number of pops (including those in Multipop)
  - …is bound by number of pushes
  - …is bound by *n*

Total cost:

- Worst-case O(*n*) for sequence of size *n*
  - Average O(1) per operation
- Aggregate Analysis
  - No probability distribution needed!

Bit counter

INCREMENT(A, k)
$i = 0$
while $i < k$ and $A[i] == 1$
$A[i] = 0$
$i = i + 1$
if $i < k$
$A[i] = 1$



Cost of INCREMENT = Θ(# of bits flipped)

- Each call could flip *k* bits
- *n* increments → O(*nk*)

Analysis

- Not every bit flips every time

bit	flips how often	times in n INCREMENTS
0	every time	n
1	$1/2$ the time	$\lfloor n/2 \rfloor$
2	$1/4$ the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

counter value	A 2 1 0	cost
0	0 0 0	0
1	0 0 1	1
2	0 1 0	3
3	0 1 1	4
4	1 0 0	7
5	1 0 1	8
6	1 1 0	10
7	1 1 1	11
0	0 0 0	14
$\vdots$	$\vdots$	15

Analysis

- Total flips:

$=$

$<$

$=$

$=$

bit	flips how often	times in n INCREMENTS
0	every time	n
1	$1/2$ the time	$\lfloor n/2 \rfloor$
2	$1/4$ the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

- *n* increments costs:
- Average cost per operation:

Accounting Method

- Idea:
  - Some operations are charged an *amortized cost* that is more than actual cost
  - Store difference to specific item in data structure as **credit**
  - Use credit to pay for when actual cost > amortized cost
- Key points:
  - In accounting method, different operations have different costs
  - Credit must never go negative
    - Otherwise amortized cost is not an upper bound on actual cost
    - Amortized cost would tell us nothing

Accounting Method Overview

Let c_i = actual cost of ith operation ,
$\hat{c}_i$ = amortized cost of ith operation .

Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

$$\text{Total credit stored} = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0 .$$

Accounting Method: Multipop Stack

Let c_i = actual cost of ith operation ,
$\hat{c}_i$ = amortized cost of ith operation .

Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

Total credit stored $= \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0$.

Stack

operation	actual cost	amortized cost
PUSH	1	?
POP	1	?
MULTIPOP	$\min(k, s)$	?

- Total amortized cost:
- Upper bound on actual cost!

Accounting Method: Binary Counter

Let c_i = actual cost of ith operation ,
$\hat{c}_i$ = amortized cost of ith operation .

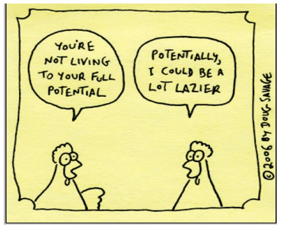
Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

Total credit stored $= \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0$.

	Actual cost	Amortized cost
A[*k*]→*1*	1	?
A[*k*]→0	1	?

- Total amortized cost:
- Upper bound on actual cost!

Potential Method



- Similar to accounting method, but we think of credit as **potential** stored with the entire data structure
- Key Ideas:
  - Accounting method stores *credit* with *specific objects*
  - Potential method stores *potential* in the *data structure* as a whole
  - Can release potential to pay for future operations
  - Most flexible of the amortized analysis

The Potential Function

Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i .

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .$$

D_i =

D_0 =

c_i =

$\hat{c}_i$ =

The (amortized) costs of having potential

Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .$$

$$\text{Total amortized cost} = \sum_{i=1}^{n} \hat{c}_i$$
$$=$$
$$=$$

The (amortized) costs of having potential

Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .$$

- If we require:

then amortized cost is always an upper bound on actual cost

- In practice:

Potential Method: Stack

$\Phi =$

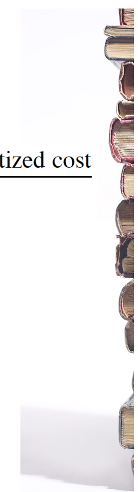
$(=$ $)$

$$\Phi(D_0) = 0\,?$$
$$\Phi(D_i) \geq 0 \,\forall i\,?$$

Potential Method: Stack

operation	actual cost	$\Delta\Phi$	amortized cost
PUSH	1		
POP	1		
MULTIPOP	$k' = \min(k, s)$		

Therefore, amortized cost of a sequence of n operations $=$



Potential Method: Binary Counting

$$\Phi = b_i = \#\text{ of 1's after } i\text{th INCREMENT}$$

- Suppose i^{th} operation resets t_i bits to 0
- $c_i \leq t_i+1$ (resets t_i bits, sets ≤ *1 bit to 1)*
- $b_i \leq b_{i-1} - t_i + 1$

$$\Delta\Phi(D_i) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i .$$

$$\hat{c}_i = c_i + \Delta\Phi(D_i) \leq (t_i + 1) + (1 - t_i) = 2 .$$

If counter starts at 0, $\Phi(D_0) =$

Therefore, amortized cost of n operations $=$

Dynamic Table ADT

Scenario

- Have a table – e.g., hash table.
- Don't know in advanced how many objects will be stored in it.
- When it fills, must reallocate with a larger size, copying all objects into the new larger table.
- Details of table organization not important

Goal: $O(1)$ amortized time per operation.

Dynamic Table Expansion

```
TABLE-INSERT(T, x)
  if T.size == 0
      allocate T.table with 1 slot
      T.size = 1
  if T.num == T.size                                   // expand?
      allocate new-table with 2 · T.size slots
      insert all items in T.table into new-table       // T.num elem insertions
      free T.table
      T.table = new-table
      T.size = 2 · T.size
  insert x into T.table                                // 1 elem insertion
  T.num = T.num + 1
```

Initially, $T.num = T.size = 0$.

Whenever table becomes full, double it in size

Count inserts

Dynamic Table Expansion: Aggregate Method

<This page intentionally left blank>

Dynamic Table Expansion: Accounting Method

<This page intentionally left blank>

Dynamic Table Expansion: Potential Method

<This page intentionally left blank>