(L13) CS 25 - Algorithms 10/18/95

Last time (chap 17)
- Greedy

Today (chap 18)
- Amortized Analysis

(L14)

Last time (chap 18)
- Amortized Analysis

Today (chap 18)
- Amortized analyses

Handouts
- HW2 Sol.
- (HW5 - PUBLIC - Blitz)

Setting Algorithm which uses a data structure

Data Structure : table, stack, etc.

Operations : Insert, delete, search ; push, pop ; etc.

Goal

Determine worst case time to perform an arbitrary sequence of operations.

Naive analysis : For n operations, upper bound is $n \cdot$ (worst case time for single operation)

- Can be quite poor if worst case time for an operation occurs only rarely ...

Example

~~Simple~~ Example: Incrementing a binary counter

Data structure: $A =$ (array of bits) | ... | $A[\lfloor \log n \rfloor]$ | ... | $A[1]$ | $A[0]$ |

Operation: Increment

bit#	...5	4	3	2	1	0	cost
	0	0	0	0	0	0	
	0	0	0	0	0	<u>1</u>	1
	0	0	0	0	<u>1</u>	<u>0</u>	2
	0	0	0	0	1	<u>1</u>	1
	0	0	0	<u>1</u>	<u>0</u>	<u>0</u>	3
	0	0	0	1	0	<u>1</u>	1
	0	0	0	1	<u>1</u>	<u>0</u>	2
	0	0	0	1	1	<u>1</u>	1
	0	0	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	4
							⋮

underlined bits have flipped

How long to perform n increments?

Naive worst case analysis: $n \cdot O(\log n) = O(n \log n)$

But ~~an~~ $O(\log n)$ increments ~~happen~~ occur only rarely, and only after many "cheaper" increments.

→ Hallmark of amortized analysis

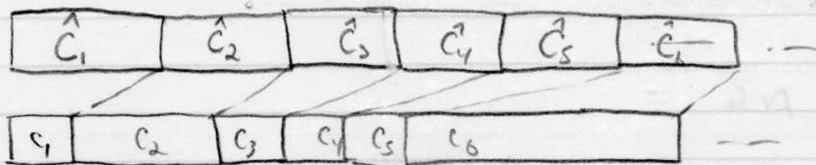
Amortized Analysis

- Each operation has a true cost which may vary over time. Let c_i be the true cost of the ith operation called. (just use c_i - true cost of ith oper)
- Each operation is assigned a fixed amortized cost. Let $\hat{c}_i$ be the amortized cost of the ith operation (just use $\hat{c}_i$ amortized cost of ith oper)

An amortized analysis guarantees that:

$$\forall n \quad \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \quad (\Rightarrow \text{more})$$

Graphically:



Amortized cost - Bound on the average cost per operation in the worst case.

(1) Aggregate Method ("brute force")

- Actually calculate a bound on the w.c. cost of performing n ops. – $T(n)$
- Amortized cost per op is $T(n)/n$

e.g. binary counter – n increments

bit 0 – flips every time — n work
bit 1 – flips every 2nd time – $n/2$ work
bit 2 – flips every 4th time – $n/4$ work

bit i – flips every 2^i time – $n/2^i$ work

bit $\lfloor \lg n \rfloor$ – flips once – 1 work

$$\text{Total work} = \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{n}{2^i} = n \sum_{i=0}^{\lfloor \lg n \rfloor} (1/2)^i$$

$$< n \sum_{i=0}^{\infty} (1/2)^i$$

$$= n \frac{1}{1-1/2}$$

$$= 2n$$

$$\text{a.c.} = 2n/n = 2 \checkmark$$

- usually difficult to do
- other methods better...

Two Other Methods for Amortized Analysis

Accounting method : (2)

- $1 = 1 unit time
- Assign amortized cost to each operation.
- If overcharge some operation, store excess as "prepaid credit" associated with some specific object in data structure.
- Use credit to help pay for expensive operations.

⇒ Maintain invariant on credit
⇒ Prove: for any sequence of ops $o_1, o_2, \ldots$, $\hat{c}_i$ is enough to maintain invariant & pay for c_i

(3)

Potential method :

- Assign amortized cost to each operation
- Maintain credit as "potential energy" of data structure

Claim: $\forall$ operation i, the amortized cost associated with operation i is sufficient to pay for the true cost of operation i and to maintain the invariant.

Pf. induction

$\hat{c}_i = 2$

$c_i = t = \begin{cases} 1 & \text{to flip } 0 \to 1 \\ t-1 & \text{to flip } 1 \to 0 \end{cases}$

⋮

Accounting Method Applied to Binary Counter

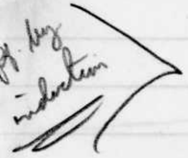
	t.c.	a.c	Σt.c.	≤ Σa.c
0 0 0 0 0 0				
0 0 0 0 0 1	1	2	1	2
0 0 0 0 1 0	2	2	3	4
0 0 0 0 1 1	1	2	4	6
0 0 0 1 0 0	3	2	7	8
0 0 0 1 0 1	1	2	8	10
0 0 0 1 1 0	2	2	10	12
0 0 0 1 1 1	1	2	11	14
0 0 1 0 0 0	4	2	15	16 ✓

- Each increment turns one bit on and same number off.
- Idea: Prepay to turn bit off.

Amortized cost - $2

- $1 to turn bit on
- $1 associated with bit to turn it off at a later time

Invariant : Every 1-bit has $1 associated with it. (carefully prove w/ induction on op #)



⟹ Always have sufficient funds to perform operation!

Total amortized cost for n increments = 2n

∴ Total time cost for n increments ≤ 2n = O(n) !

Potential Method — more rigorous analysis, more powerful technique

- Associate a potential function Φ with data structure D.
- $\Phi(D_i)$ = "potential" associated with data structure after i^{th} operation
  - some measure of how "complex" the data structure has become (e.g. its size)
- Require that:

$$\boxed{\hat{c}_i \geq c_i + \Phi(D_i) - \Phi(D_{i-1})} \qquad (*)$$

- $$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_{n-1}) + \Phi(D_{n-1}) - \Phi(D_{n-2}) + \Phi(D_{n-2}) - \cdots$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\Rightarrow \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i - \underbrace{\left[\Phi(D_n) - \Phi(D_0) \right]}_{\text{need this to be non-negative}}$$

- Usually require that $\boxed{\Phi(D_0) = 0}$ and $\boxed{\Phi(D_i) \geq 0 \ \forall i}$

$$\Rightarrow \boxed{\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i}$$

Potential Method Applied to binary counter

	t.c.	a.c.	$\Phi(D_i)$
0 0 0 0 0 0			0
0 0 0 0 0 1	1	2	1
0 0 0 0 1 0	2	2	1
0 0 0 0 1 1	1	2	2
0 0 0 1 0 0	3	2	1
0 0 0 1 0 1	1	2	2
0 0 0 1 1 0	2	2	2
0 0 0 1 1 1	1	2	3
0 0 1 0 0 0	4	2	1

- Complexity of our "data structure" – # 1's

  $\Phi(D_i)$ = # 1's in D_i

- Must verify three properties
  - $\Phi(D_0) = 0$ ✓
  - $\Phi(D_i) \geq 0 \ \forall i$ ✓
  - $\hat{c}_i \geq c_i + \Phi(D_i) - \Phi(D_{i-1})$
    - Consider $\hat{c}_i = 2$ as before
    - Suppose $c_i = t$
      - 1 unit time to turn 0-bit "on"
      - $t-1$ time to turn $t-1$ 1-bits "off"

$$\Rightarrow \Phi(D_i) = \Phi(D_{i-1}) + 1 - (t-1)$$

$$\Leftrightarrow \Phi(D_i) - \Phi(D_{i-1}) = 2 - t$$

$$2 \geq t + (2-t) = 2 \quad ✓$$

Amortized Analysis of Stack Operations

Stack S :

Operations : Empty (s) — returns 1 if stack empty, 0 otherwise

Push (s, x) — push x on stack S

Pop (s) — If Empty(s), return nil
else, pop element and return value

Multipop (S, k) — Pop and return $\min(k, |S|)$ elements from S

How long to perform n operations ?

Naive worst case analysis : $n \cdot O(n) = O(n^2)$

But Multipop can't be expensive too often, and only if many Push's had preceded it.

Accounting Method

- Idea: Prepay to Pop or Multipop elements

- Amortized cost for Push – $3

  - $1 to push element
  - $1 associated with element to check if stack is empty
  - $1 associated with element to pop element.

Pop – ~~0~~ ~~$0~~ $1 (to check if stack empty)

Multipop – ~~0~~ ~~$0~~ $1 (to check if stack empty)

Empty – $1

$\Rightarrow$ Always have sufficient funds to perform operations!

← Total amortized cost for n operations $\leq 3n$

$\therefore$ Total true cost for n operations $\leq 3n = O(n)$!

Potential Method

- Complexity of our data structure – twice size of stack

$$\Phi(S_i) = 2 \cdot |S_i|$$

- Must verify three properties
  - $\Phi(S_0) = 0$ ✓
  - $\Phi(S_i) \geq 0 \ \forall i$ ✓
  - $\hat{c}_i \geq c_i + \Phi(D_i) - \Phi(D_{i-1})$
    - Push $\hat{c}_i = 3$

      $3 \geq 1 + 2$ ✓
    - Pop $\hat{c}_0 = 0$

      $0 \geq 2 - 2$ ✓
    - Multipop $\hat{c}_0 = 0$

      $0 \geq 2 \cdot \min(k, |S_i|) - 2 \min(k, |S_i|)$ ✓
    - Empty $\hat{c}_i = 1$

      $1 \geq 1 + 0$ ✓

fix

Dynamic Table

- insert, ~~delete~~, search

array of slots

# of elements ≤ m

consider insertion & searches:

- initially $m = 1$

insert elements until $> m$,
then generate table of size $\underline{2m}$
reinsert old elements to new table

n insertions

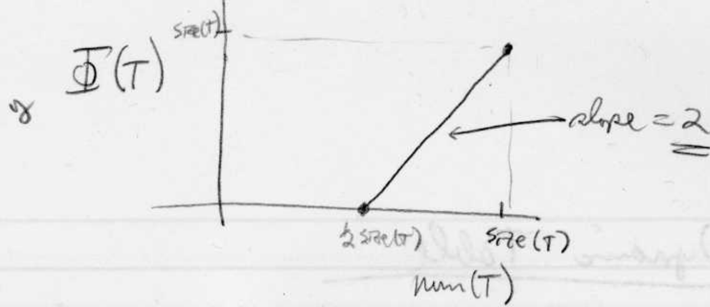
c_1 : 1
c_2 : 1+1 (new table)
c_3 : 1+2 (new table)
c_4 : 1
c_5 : 1+4 (new table)
c_6 : 1
⋮ : ⋮
c_8 : 1
c_9 : 1+8 (new table)

aggregate — n insert cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=1}^{\lg n} 2^j$$

$$\le n + 2n = 3n$$

$$a.c. = \frac{3n}{n} = 3$$



<u>potential</u>

$c_i =$ (see table)

$\hat{c}_i = 3$

let $size(T) = c$; let $\Phi(T) = y$; let $num(T) = x$

$y = mx + b$

2 points: (c, c) & $(\frac{1}{2}c, 0)$

$C = mC + b$

$- \; 0 = m(\frac{1}{2}c) + b$

$C = m(\frac{1}{2}c)$

$\Rightarrow m = 2$

$\Rightarrow b = -c$

get:

$y = 2x - c$

i.e.

$\Phi(T) = 2$(# elements in table) $- size(T)$

$\Phi(T) = 2$(# elements in table) $-$ size(table)

non neg. ✓ (table always at least half full)

~~$c_i \le \hat{c}_i + \Phi(T_{i-1}) - \Phi(T_i)$~~ $\hat{c}_i \ge c_i + \Phi(T_i) - \Phi(T_{i-1})$

<u>Case 1</u>: no table expansion

$c_i = 1$

$\hat{c}_i = 3$

~~$\Phi(T_{i-1}) - \Phi(T_i) = 2\, num_{i-1} - size_{i-1}$~~

~~$-(2\, num_i - size_i)$~~

~~$= 2(num_{i-1} - num_i)$~~

~~$= -2$~~

$\Delta \Phi(T) = 2$

$\hat{c}_i \overset{?}{\ge} c_i + \Phi(D_i) - \Phi(D_{i-1})$

$3 \ge 1 + 2$ ✓

$1 \le 3 - 2$ ✓

<u>Case 2</u>: table expansion

$c_i = i = num_i$

$\hat{c}_i = 3$

$size_i = 2\, size_{i-1}$

$size_{i-1} = num_{i-1}$

$c_i = size(T) + 1$

$\hat{c}_i = 3$

$\Delta(\Phi) = \Phi(D_i) - \Phi(D_{i-1})$

$= 2 - size(T)$

$\hat{c}_i \overset{?}{\ge} c_i + \Phi(D_i) - \Phi(D_{i-1})$ ✓

$3 \overset{?}{\ge} (size(T)+1) + 2 - size(T)$

$\Phi(T_{i-1}) - \Phi(T_i) =$

$2(num_{i-1} - size_{i-1}) \cdot$

$- 2(num_i - size_i)$

$= (2\, num_{i-1} - num_{i-1}) - (2\, num_i - 2\, num_{i-1})$

$= num_{i-1} - 2$

$c_i \leq \hat{c}_i + \Phi(T_{i-1}) - \Phi(T_i)$

$num_i \leq 3 + num_{i-1} - 2$

$num_i \leq num_{i-1} + 1$ ✓

Expansion: when full, double size

Contraction: when < 1/4 full, halve size

<u>c_i:</u> insert - as before 1+ (old num, if expansion)

deletion - 1+ (new num, if contraction)

<u>$\hat{c}_i$:</u> 3 for insertion
2 for deletion

<u>Potential</u> $\Phi(T) = \begin{cases} 2\, num(T) - size(T) & \text{if } 2\, num(T) \geq size(T) \\ \frac{1}{2}(size(T) - 2\, num(T)) & \text{else} \end{cases}$

Javed Aslam

Today: - Finish amortized analysis Chap 18
- Greedy algorithms Chap. 17

Next time: - B-trees Chap 19

Dynamic tables with expansion and contraction

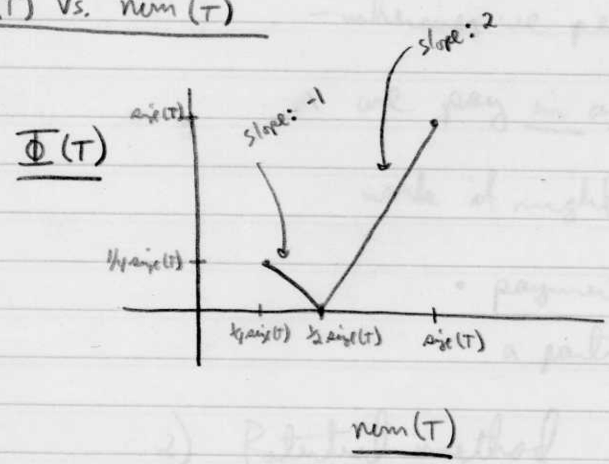
Expansion: when table full, double its size
Contraction: when table < 1/4 full, halve its size

true costs c_i : insert - 1 + (old # elements, if expansion)
deletion - 1 + (new # elements, if contraction)

amortized costs $\hat{c}_i$: insertion - 3
deletion - 2

potential function
$$\Phi(T) = \begin{cases} 2\,num(T) - size(T) & \text{if } 2\,num(T) \geq size(T) \quad [\geq 1/2 \text{ full}] \\ 1/2\,[size(T) - 2\,num(T)] & \text{otherwise} \quad [< 1/2 \text{ full}] \end{cases}$$

Plot $\Phi(T)$ vs. $num(T)$



- explain why this should work intuitively
- keep graph on board

Now, we must do two things:
1) Show $\Phi(T)$ always ≥ 0 ✓ (see graph)
2) Show ~~$c_i \leq \hat{c}_i + \Phi(T_{i-1}) - \Phi(T_i)$~~
$\hat{c}_i \geq c_i + \Phi(T_i) - \Phi(T_{i-1})$

Review

- Amortized analysis is a technique for determining the worst case running time of a sequence of operations
- Works by assigning amortized costs to the operations and showing that the sum of the true costs is at most the sum of the amortized costs.
  - Show this property in one of two ways
    1) Accounting method
       - whenever we perform an operation, we pay in advance for any future work it might cause us
         - payment usually associated with a particular object
    2) Potential method
       - whenever we perform an operation, we pay present cost + change in potential
         - ↳ future cost

(Over)