

Amortized Analysis

11/1/07

①

Ch. 17 of Cormen

- So far we have been analyzing the running time using worst-case analysis of individual operations
- It is easy to extend the worst-case analysis to a sequence of ops by determining the worst-case running time of each op, then summing them together to get a worst-case running time for the entire sequence.

Ex. Stack

- Operations: push, pop, empty
- Worst case running time (assume linked list implementation)

<u>Operation</u>	<u>Running Time</u>
Push()	$\Theta(1)$
Pop()	$\Theta(1)$
Empty()	$\Theta(n)$

When using a stack you are doing a sequence of ops.

Suppose you did m ops.

Worst-case scenario m Empty() calls, so a running time of $\Theta(mn)$ or $\Theta(n)$ time per operation. \leftarrow really over-counting.

• The simple worst-case analysis is overly-pessimistic.

• Does not capture correlation between ops.

\hookrightarrow Actual worst-case may be much lower than summing over individual ops worst case running times.

Amortized complexity analysis provides a means of arriving at these tighter bounds.

- Amortized analysis provides a way for arriving at these tighter bounds.
- Often applied to self-adjusting data structures.
- Amortizing = averaging the time for each operation in a worst possible sequence.
- Not the same as average-case analysis

Average-case analysis

- Requires probabilistic assumptions about ops in the sequence, as well as constitutes a "average" sequence.
- We sum the expected costs of the ops in an expected sequence to arrive at the expected running time for the sequence.
- Certain sequences may have running times that are much worse than the expected result. Analysis is only valid as long as the probabilistic assumptions are valid.

Amortized Analysis

- Does not make any probabilistic assumptions
- We can guarantee that the running time we arrive at will not be exceeded by any sequence of ops.

Key Idea:

Each op in a sequence has two values associated with it:

- Actual running time, denoted C
- Amortized running time, denoted C^1

- The actual running time of a given op may depend on the ops that preceded it.
- The amortized cost is going to be fixed and is independent of the order of ops.

Let c_i and \hat{c}_i denote the actual and amortized running time of the i -th operation in a sequence of n operations. Then for the sequence

Actual running time: $\sum_{i=1}^n c_i$

The amortized running time is: $\sum_{i=1}^n \hat{c}_i$

Central idea: Upperbound the actual running time of the sequence by the amortized running time of the sequence.

I.e. show that:

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$

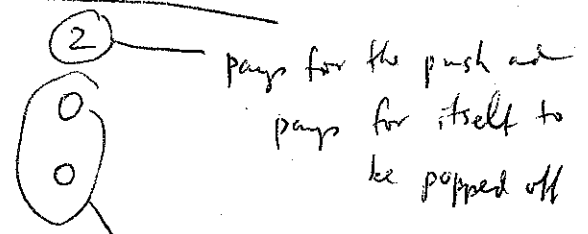
Accounting Method

Basic Idea: Banker's view by relating running time to cost.

- Individual ops in a sequence assign credits or debits to a balance sheet.
- Certain types of operations may "over pay" (i.e. assign more credits to the balance sheet than they actually cost).
- More expensive ops can use these credits to pay for themselves.

Ex: Stack: The actual costs are proportional to:

Operation	Actual cost (c)	Amortized Cost (c)
Push()	1	2
Pop()	1	0
Stack Empty()	n	0



doesn't cost anything to remove items from stack because they are already "pre-paid"

Analysis

Starting with an empty stack, any sequence of stack ops can be paid for using these amortized costs.

- If the stack is initially empty, the balance sheet can never become negative.
- All that remains is to determine the amortized cost of a worst-case sequence of n operations.

worst-case = n push ops \rightarrow amortized cost $2n$ \leftarrow upper bound to actual cost

\therefore Any sequence of n stack ops can be performed in $\Theta(n)$ time, and the average cost of each operation in these sequences is $\frac{2n}{n} = \Theta(1)$.

Potential Method

Basic idea: "Physicist's" point of view by relating running time to potential energy.

- Individual ops increase or decrease the potential energy in the system \rightarrow
- Some ops add energy, some remove



System is the data structure

Given data structure D

Denote the initial configuration D_0 , the i th config is D_i \leftarrow before applying any ops

Potential function $\Phi: D \rightarrow \mathbb{R}$

The amortized running time of an operation on D is defined

$$\hat{c} = \underbrace{c}_{\text{actual cost of operation}} + \underbrace{\Delta\Phi(D)}_{\text{change in potential that resulted from op } c}$$

Then the amortized running time of a sequence m

is

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

actual running time

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

Amortized running time will bound the actual running time of the sequence if $\Phi(D_m) \geq \Phi(D_0)$

- A sufficient (but not necessary) condition for satisfying this is that $\Phi(D_i) \geq \Phi(D_{i-1})$ for $i=1, 2, \dots, m$

Ex: Back to Stack

← choice of good potential function is tricky.

Let the potential function be the number of elements stored on the stack

- Since stack is empty at the beginning $\Phi(D_0) = 0$.

- Since stack cannot be negative then $\Phi(D_m) \geq \Phi(D_0)$.

∴ Amortized running time will represent an upper bound on the actual running time of any sequence of operations.

Derive amortized running time for each op. Φ after op Φ before op

• Push operation: $\Delta \Phi(D) = (n+1) - n = 1$

$\hat{c} = 1 + 1 = 2$ for push

• Pop operation: $\Delta \Phi(D) = -1$, $\hat{c} = 1 + (-1) = 0$

• Make empty: $\Delta \Phi(D) = -n$, $\hat{c} = n - n = 0$ ← same as before!

6

Ex: Implementing a stack with a dynamic list

Dynamic list: list elements are stored in a dynamically allocated array

Define: load factor of a non-empty dynamic list is the length of the list \leftarrow current # of elements divided by the current size of the list

$$\text{load factor is } \leq 1$$

Expansion: whenever the load factor reaches 1 through ops, we allocate a new array of twice the size.

Contraction: whenever the load factor reaches $\frac{1}{4}$, a new array that is $\frac{1}{2}$ the size is allocated.

In either case, the elements stored in the current array are then moved to the new one before deleting current array.

Accounting Method:

<u>Operation</u>	<u>Actual cost (i)</u>
Push()	1 or n
Pop()	1 or $(n-1)$
Empty()	1

Push & pop are the only ops that can be expensive.