Amortization

Karen Edwards

Tufts University

Amortization - Intro

What is amortization?

CONCEPT: From accounting. E.g. amortization in taxes (laptop = 3 years, roof = 20 years) or loans (house = 30 years)

CS CONTEXT: Sequence of operations which are mostly cheap but sometimes expensive

GOAL: Compute the **guaranteed** time to perform n operations

Example - Expanding via Linked List or Dynamic Array

Suppose we want to insert repeatedly into a data structure, either a linked list or a dynamic array. What is the cost of the following?

	Linked list	Dynamic array
One operation		
n operations, computed using (worst case)·(# ops)		
n operations, computed using amortization		

Methods of Proving Amortizations

- Aggregate - take total time T(n) as an upper bound, average to T(n)/n for each operation.
- Accounting - track credits/debits per operation.
- Potential - treat entire data structure as storing energy to be used for operations.

All hwk/exam problems will allow the accounting option

Proof by Accounting

- Accounting - track credits/debits per operation.

- You begin with a data structure and operations that have a **real** cost.
- You assign each operation an **amortized** cost, usually constant. (Our model: think of these costs as coins).
- Show that if you pay the amortized cost each time (saving coins in the bank as able, spending them when needed), you never run out of coins. Don't go into debt!!

Mathematically, given any sequence of n operations $\{1, 2, \ldots n\}$
If we let c_i = real cost of operation i,
and $\hat{c}_i$ = amortized cost of operation i,
We want to show that $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$.
In other words, (total amortized cost) - (total real cost) ≥ 0.

Example 1 - Multipop

Given: a stack S of size *s* and three operations

- Push
- Pop
- Multipop(k)

What is the real cost? What should we set for the amortized cost?

Why do we never run out of coins?

Conclusion:

Example 2 - Dynamic Array

Simple data structure

- Data pointer $+$ 2 counters (current size/max capacity)
- Insertion is often fast (first empty slot) $\Rightarrow \Theta(1)$
- If full, we **double** size of array. Copy everything and insert
  $\Rightarrow \Theta(n)$ runtime if this happens
- Let's prove that amortized insertion time is $O(1)$

Accountant approach

Two operations involved in n insertions

Insert	Insertion when array has space. User deposits 3 coins in each invocation
Resize	Grow array when full. Implicitly invoked. 0 coins deposited

Checking balance

- Let's look at INSERTION
  - 3 coins are deposited
  - Insertion needs $O(1)$ number of operations
    - Check if space, insert, increase counter
    - $O(1)$ operations $\Rightarrow$ 1 coin withdrawn
    - Net gain: 2 coins. We associate them to the inserted element
- What about EXPAND?
  - 0 coins are deposited. Must withdraw coins to pay for runtime
  - **Key property**: only when array full.
    - Items in second half have 2 coins saved each.
    - $\Rightarrow$ one coin per element in array!
    - Withdraw 1 coin for element we copy onto bigger array
    - $\Rightarrow$ Use all coins but never go negative
    - After expansion half of the array is full

Big picture

Theorem

n insertions in a dynamic array will need $\Theta(n)$ time in total.
(or insertion in a dynamic array uses $\Theta(1)$ amortized time).

User deposits 3 coins per INSERT (0 on EXPAND)

n operations invoked $\Rightarrow$ $3n$ coins deposited

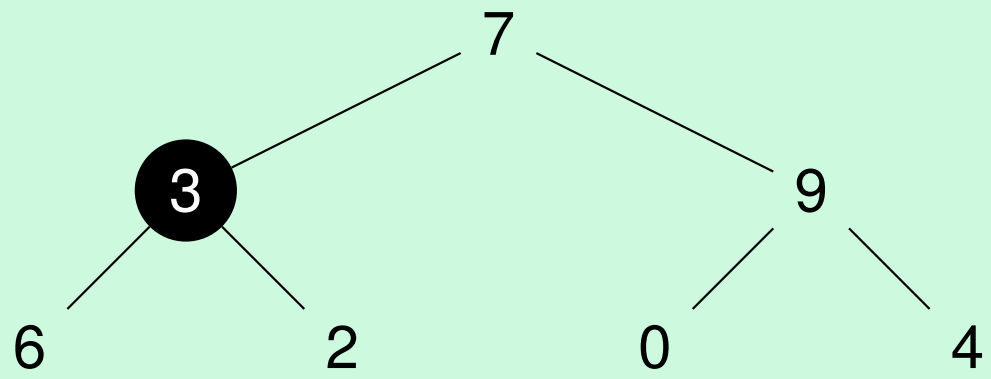
Coins withdrawn each time computer spends time

Never go negative balance $\Rightarrow$ at most $3n$ coins withdrawn

Each coin is $O(1)$ operations $\Rightarrow$ at most $3n \cdot O(1)$ time spent

Aggregate Method Example - Bottom-up Heap Build

- Aggregate - take total time T(n) as an upper bound, average to T(n)/n for each operation.



Insert all numbers at once. Fix from leaves upwards

At each node we have to **sink** root $\Rightarrow \Theta(\log n_i)$

$n_i =$ number of elements in the sub-heap

Sometimes n_i is big but often n_i is small

n sink operations need $\Theta(n)$ time in total

$\Rightarrow \Theta(1)$ amortized runtime

Potential Method Example - Bit counter

- Potential - treat entire data structure as storing energy to be used for operations.

How much time does it take to INCREMENT() a bit counter?

0	1	1	1	0	0	1	1	1	1

0	1	1	1	0	1	0	0	0	0