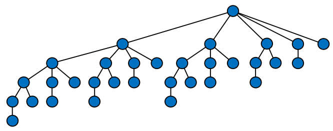
Lecture 3. Amortized Analysis (Continued)

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing
Clemson University
Spring, 2021

1

Amortized Analysis

- In general, an operation runs in O(f(n)) amortized time if any sequence of k such operations runs in O(k f(n)) time.
  - Example: Any sequence of k operations takes O(k) worst-case time, so we say that each operation takes O(1) **amortized** time.
- Amortized analysis is worst-case analysis, just averaged over a *sequence* of operations.

2

2

Amortization of Costs

Actual Cost	Amortized Cost
Jan: \$10	Jan: \$20
Feb: \$10	Feb: \$20
…	…
Nov: \$10	Nov: \$20
Dec: \$130	Dec: \$20

- So our cost is “\$20/month, amortized”.
- This is a simpler, more accurate description of our cost structure.
- Compare with actually paying \$20/month…

3

3

Amortized Analysis : Motivation

- It gives us a much clearer picture of the true performance of a data structure that more faithfully describes the true performance.
  - E.g., “Θ(N) worst case vs. O(1) amortized”.
- We are still doing the same amount of work; the data structure isn’t changing. We are just changing when we account for this work (earlier than it actually happens).

4

4

Amortized Analysis : Motivation

- Suppose we have 2 implementations of a data structure to choose from:
  - A: O(log n) worst-case time / operation.
  - B: O(log n) amortized time / operation.
- There is **<u>no difference</u>** if we use either A or B as part of a larger algorithm. For example, if our algorithm makes n calls to the data structure, the running time is O(n log n) in either case.
- The choice between A and B only matters in a "real-time" setting when the response time of an individual operation is important.

5

5

Generalizing to Multiple Operations

- We say an operation A requires O(f(n)) amortized time if *any* sequence of k invocations of A requires O(k f(n)) time in the worst case.
- We say operations A and B have amortized running times of $O(f_A(n))$ and $O(f_B(n))$ if *any* sequence containing k_A invocations of A and k_B invocations of B requires $O(k_A f_A(n) + k_B f_B(n))$ time in the worst case.
- And so on, for 3 or more operations…

6

6

Aggregate Analysis: A Simple, but Often Limited, Method for Amortized Analysis

- Compute the worst-case running time for an arbitrary sequence of k operations, then divide by k.
- Unfortunately, it is often hard to bound the running time of an arbitrary sequence of k operations (especially if the operations are of several types – for example “push” and “pop”)…

7

7

Recall our Initial Discussion: Think about an Algorithm from the Perspective of a Data Element…

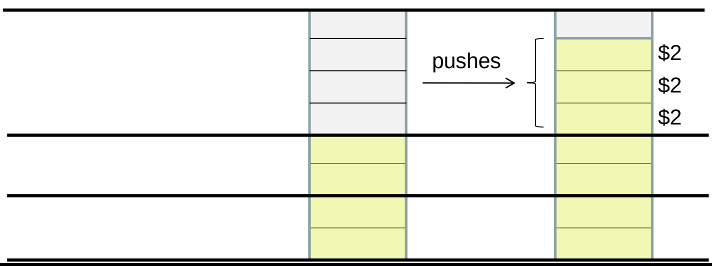
- Figure out how much work / running time is spent on a single generic element of data during the course of the algorithm.
- Add this up to get the total running time.
- In amortized analysis, this is often called the “accounting method”…

8

8

Accounting Method Analysis: Example Using Memory Re-Sizing

- Charge 3 units (i.e., O(1) amortized time) for each *push* operation.
  – 1 unit for the immediate *push*.
  – "$2" credit for future memory expansions.



Make the New Elements Pay!

- When it comes time to expand our buffer from size n to 2n (at a cost of n), exactly n/2 of the elements in our current buffer have been newly-added since the last memory expansion.
- All these elements have $2 credit on them.
- So we have $n worth of credit – enough to pay for the current memory expansion!
- After expansion, no credit remains (subsequently-added items will contribute toward next expansion).

What About Adding “Pop” – Will This Work Well?

- When the buffer fills up due to too many pushes, double its size.
- When the buffer becomes less than half full due to too many pops, halve its size.

11

What About Adding “Pop” – Will This Work Well?

- When the buffer fills up due to too many pushes, double its size.
- When the buffer becomes less than half full due to too many pops, halve its size.

  - NO! Every operation can end up taking Θ(n) time.
  - Amortization can’t save us when there aren’t any cheap operations to amortized over…

12

A Better Approach…

- When the buffer fills up due to too many pushes, double its size.
- When the buffer becomes less than <u>one quarter</u> full due to too many pops, halve its size.

- Since buffer is half-full after expansion or contraction, this means we must do many intervening "cheap" pushes / pops before the next "expensive" operation…

13

13

Accounting Analysis of Push + Pop

- Charge 3 units (i.e., O(1) amortized time) for each *push* operation.
  - 1 unit for the immediate *push*.
  - "$2" credit for future memory expansions.
- Charge 2 units per pop (1 unit for the immediate operation, "$1" credit)

14

14

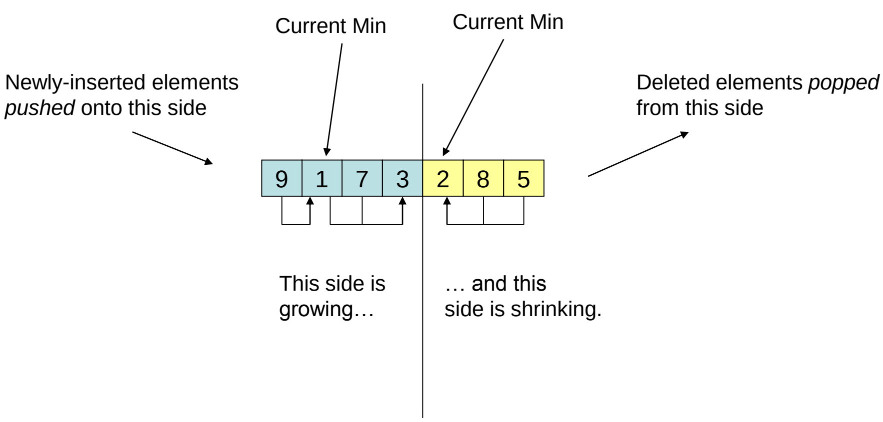
Example : The Min-Queue

- Using either a linked list or a (circular) array, it is easy to implement a FIFO queue supporting the *insert* and *delete* operations both in O(1) worst-case time.
- Suppose that we also want to support a *find-min* operation, which returns the value of the minimum element currently present in the queue.
- It is possible to implement a "min-queue" supporting *insert*, *delete,* and *find-min* all in O(1) worst-case time?
- Easier question: what about a "min-stack"?

15

15

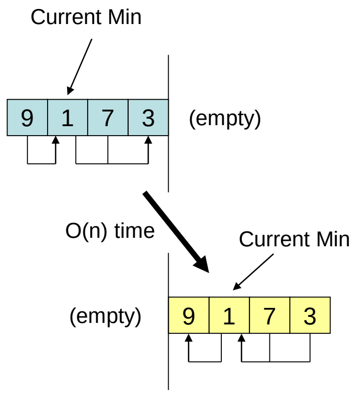
The Min-Queue as a Pair of "Back-to-Back" Min-Stacks



16

16

Expensive (but Rare) Operations

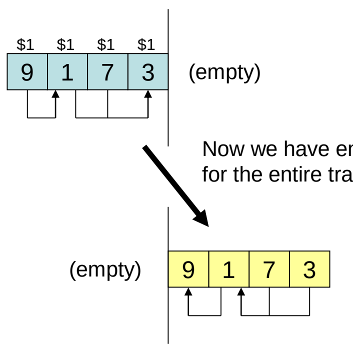


When yellow stack becomes empty, spend O(n) time and transfer the contents of blue stack into the yellow stack.

Worst-case running time for *delete*: O(n)

Amortized analysis

Charge insert <u>2</u> units of time: 1 for the push, and $1 in credit for each new element.



Now we have enough credit to pay for the entire transfer when it occurs!

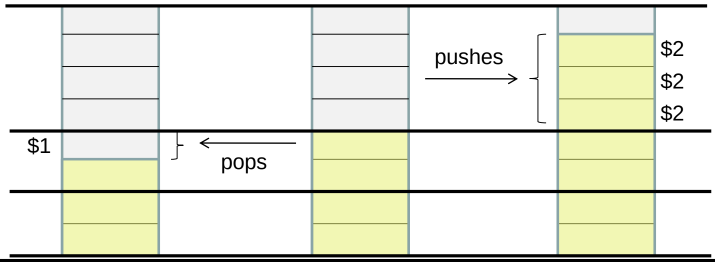
Final running times:
- *Insert* and *Delete*: O(1) amortized time
- *Find-Min*: O(1) worst-case time

Recap:
Block Expansion and Contraction

- The approach:
  – Expand buffer (to size 2m) if n = m.
  – Contract buffer (to size m/2) if n < m/4.
  – This ensures that m/4 ≤ n ≤ m.
  (recall n = # current elements, m = buffer size)
- The accounting method shows that *push* and *pop* run in only O(1) amortized time.
  – When we expand, we need m units of credit.
  – When we contract, we need m/4 units of credit.

Accounting Analysis of Push + Pop

- Charge 3 units (i.e., O(1) amortized time) for each *push* operation.
  – 1 unit for the immediate *push*.
  – "$2" credit for future memory expansions.
- Charge 2 units per pop (1 unit for the immediate operation, "$1" credit)



Potential Functions

- A **potential function** provides a somewhat more formulaic way to perform amortized analysis.

  (although It's really just another way of looking at the accounting method)
- Express total amount of "credit" present in our data structure using a non-negative potential function of the state of our data structure.
  - Example: for the memory allocation problem, our

    potential function is: $\phi = \begin{cases} 2n - m & \text{if } n \geq m/2 \\ m/2 - n & \text{if } n < m/2 \end{cases}$
  - If n = m/2, then Φ = 0. No credit right after expansion or contraction.
  - If n = m, then Φ = m. Just enough credit to expand!
  - If n = m/4, then Φ = m/4. Just enough credit to contract!

Potential Functions

- Required properties of a potential function:
  - It should start out initially at zero (no credit initially).
  - It should be nonnegative (can't go into "debt").
- Some notation:
  - Let $c_1, c_2, \ldots, c_k$ denote the actual cost (running time) of each of k successive invocations of some operation.
  - Let φ_j denote the potential function value right after the jth invocation.
- The amortized cost a_j of the jth operation is now:

$$a_j = \underbrace{c_j}_{\text{Actual cost}} + \underbrace{(\varphi_j - \varphi_{j-1})}_{\text{Change in potential (i.e., total credit added or consumed)}}$$

Potential Functions : Example

$a_j = \underbrace{c_j}_{\text{Actual cost}} + \underbrace{(\varphi_j - \varphi_{j-1})}_{\text{Change in potential (i.e., total credit added or consumed)}}$

$$\phi = \begin{cases} 2n - m & \text{if } n \geq m/2 \\ m/2 - n & \text{if } n < m/2 \end{cases}$$

- Amortized cost of *push*:
  - Push by itself: $a_j = c_j + (\varphi_j - \varphi_{j-1}) \leq 1 + 2 = 3$. (contributes 2 units of potential)
  - Expansion by itself: $a_j = c_j + (\varphi_j - \varphi_{j-1}) = m + (-m) = 0$. (draws m units of potential to pay for expansion)
- Amortized cost of *pop*:
  - Pop by itself: $a_j = c_j + (\varphi_j - \varphi_{j-1}) \leq 1 + 1 = 2$. (contributes 1 unit of potential)
  - Contraction by itself: $a_j = c_j + (\varphi_j - \varphi_{j-1}) = m/4 + (-m/4) = 0$. (draws m/4 units of potential to pay for contraction).

23

23

Amortized Running Times as Upper Bounds

Recall: $a_j = \underbrace{c_j}_{\text{Actual cost}} + \underbrace{\varphi_j - \varphi_{j-1}}_{\text{Change in potential (i.e., total credit added or consumed)}}$

- Over a sequence of k operations:

$$\Sigma_j\, a_j = \Sigma_j\, (c_j + \varphi_j - \varphi_{j-1}) = (\Sigma_j\, c_j) + \underbrace{\varphi_k}_{\geq 0} - \cancel{\varphi_0}^{\,0} \geq \Sigma_j\, c_j$$

- Therefore, over any sequence of operations, the total amortized running time gives us an upper bound on the total actual running time (as we expected!)

24

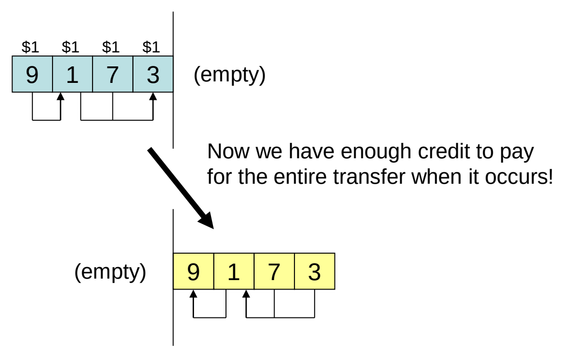
24

Designing and Using Potential Functions

- Some potential functions look substantially more complicated than the ones we've seen so far.
- Although it is more or less equivalent to the accounting method, potential functions give us a widely-accepted "formulaic" means of performing amortized analysis.
  - State potential function.
  - Show that it's zero initially and always nonnegative.
    (and should only depend on *current* state)
  - Then use $a_j = c_j + \varphi_j - \varphi_{j-1}$ to compute the amortized running time of each operation.

Example: The Min-Queue

Charge insert 2 units of time: 1 for the push, and $1 in credit for each new element.



Final running times:
- *Insert* and *Delete*: O(1) amortized time
- *Find-Min*: O(1) worst-case time