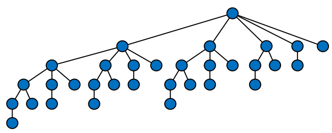
Lecture 2. Amortized Analysis

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Useful Analysis Technique: 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.

  (compared to adding up the time spent on each "operation", summed over each operation in chronological order)

Useful Analysis Technique: Think about an Algorithm from the Perspective of a Data Element…

Elements of Data

Steps Taken by Algorithm (in chronological order…)

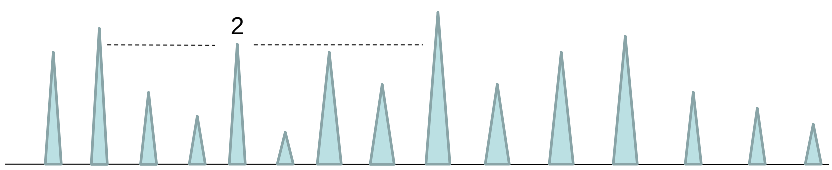
	Elem 1	Elem 2	Elem 3	Elem 4	Elem 5	
Step 1		1	1			*"Swap elements 2 and 3"*
Step 2						
Step 3				1		*"Modify element 4"*
Step 4						
Step 5	1	1	1	1	1	*"Copy elements into a new array"*

Time/work done to each element at each major "step" of our algorithm's execution

(that is, sum the columns first, not the rows…)

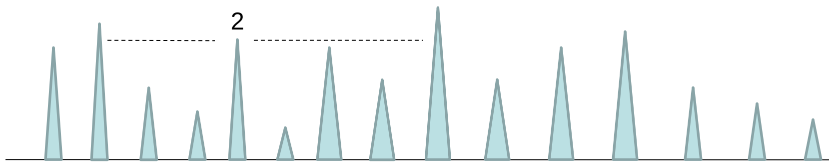
3

Example: Domination Radius



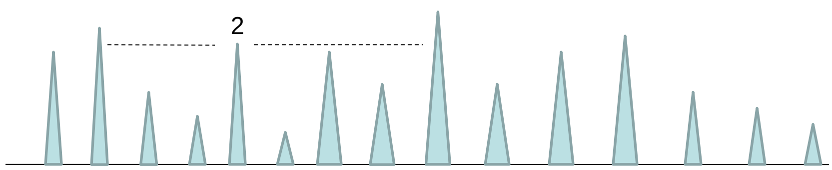
- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.

Example: Domination Radius



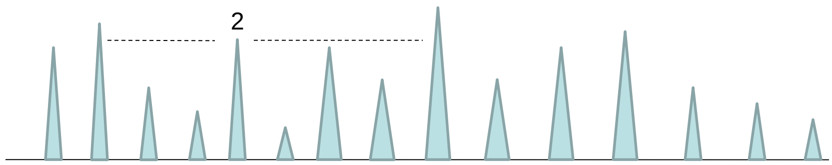
- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked.

Example: Domination Radius



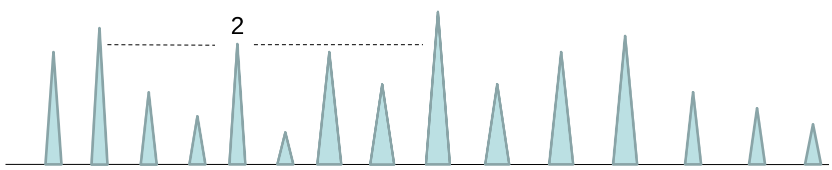
- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked. ($O(N^2)$ worst-case)

Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked. ($O(N^2)$ worst-case)
- Refinement: from each element, scan left and right simultaneously until blocked.

Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked. ($O(N^2)$ worst-case)
- Refinement: from each element, scan left and right simultaneously until blocked. ($O(N \log N)$ time!)
- $O(N)$ is even possible, using fancier data structures…

Re-Sizing Memory Blocks

- Since memory blocks often cannot expand after allocation, what do we do when a memory block fills up?
- For example, suppose we allocate 100 words of memory space for a stack (implemented as an array), but then realize we have more than 100 elements to push onto the stack!

(yes, use of a linked list would have solved this problem, but suppose we really want to use arrays instead…)

9

Memory Allocation : Successive Doubling

- A common technique for block expansion: whenever our current block fills up, allocate a new block of twice its size and transfer the contents to the new block.
- Unfortunately, now some of our push operations will be quite slow!
  - Most push operations take only $O(1)$ time.
  - However, a push operation resulting in an expansion (and a copy of the n elements currently in the stack) will take $\Theta(n)$ time.
- This motivates the importance of **amortized analysis**…

10

How to Describe the Running Time of Push…?

- Push has a somewhat non-uniform running time profile:
  - O(1) almost always
  - Except Θ(N) every now and then.
- But just saying the running time is "Θ(N) in the worst case" doesn't tell the whole story…
  - Doesn't do the structure justice.
  - People might be scared to use it for large input sizes…

11

How Expensive is Your Car to Maintain…?

Jan: $10

Feb: $10

…

Nov: $10

Dec: $130 = $10 + yearly $120 tune-up

- Same problem: saying it's "$130/month in the worst case" doesn't tell the complete story…

12

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…

13

Back to Push…

- How much does each push actually cost?

Op#:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

14

Back to Push…

- How much does each push actually cost?

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

"True" cumulative cost after any sequence of k operations is upper bounded by "fictitious" cumulative cost of 3k…

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Total insert cost = k

Total copy cost ≤ 2k

17

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8 +2	+2	+2	+2	+2	+2	+2	+2	~~16~~		…
Total:	1	2	3	1	5	1	1	1		1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Total insert cost = k

Total copy cost ≤ 2k

18

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4	+2	+2	+2	~~8~~	+2	+2	+2	+2	+2	+2	+2	~~16~~		…
Total:	1	2	3	1	+2	1	1	1	+2	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Total insert cost = k

Total copy cost ≤ 2k

Back to Push…

- How much does each push actually cost?

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

- So how different is our version of push from a version that takes 3 units in the worst case?

Amortized Analysis

- Any sequence of k pushes takes O(k) worst-case time, so we say that push takes O(1) **amortized** time.
- "On average", each individual push therefore takes O(1) time.
- 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.

21

Amortized Analysis : Motivation

- Amortized analysis is an ideal way to characterize the worst-case running time of operations with highly non-uniform performance.
- **It is still <u>worst-case</u> analysis, just averaged over an arbitrary sequence of operations.**
- It gives us a much clearer picture of the true performance of a data structure that more faithfully describes the true performance.
  - E.g., "O(N) worst case vs. O(1) amortized".

22

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 **no difference** 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.

23

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…

24

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)…

25

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

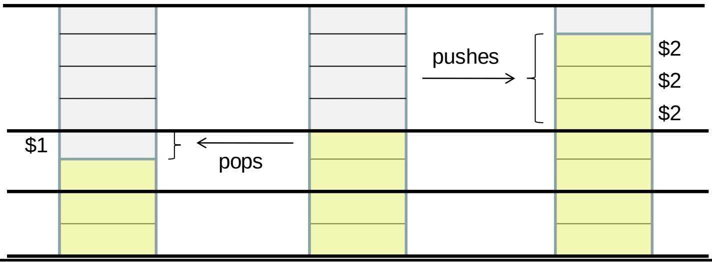
- Figure out how much work / running time is spent on a single generic element of data during the course of the algorithm.
- Add this up to get the total running time.

- In amortized analysis, this is often called the "accounting method"…

26

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.
- Charge 2 units per pop (1 unit for the immediate operation, "$1" credit)



27

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).

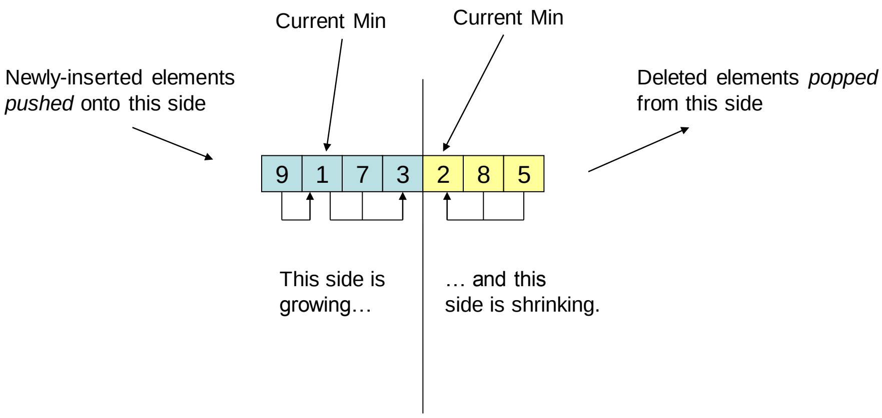
28

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?

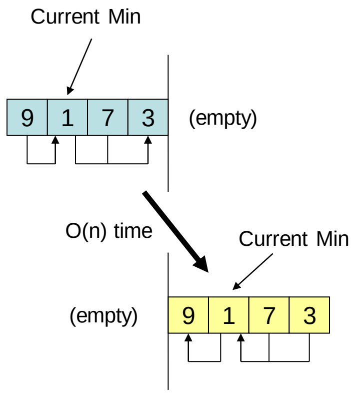
29

The Min-Queue as a Pair of “Back-to-Back” Min-Stacks



30

Expensive (but Rare) Operations

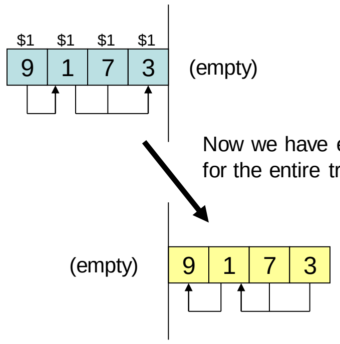


When yellow stack becomes empty, spend O(n) time and transfer the contents of blue stack into the yellow stack.

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

Amortized analysis

Charge insert 2 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.
- 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.

33

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!

34

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*:
  - Without expansion: $a_j = c_j + (\varphi_j - \varphi_{j-1}) \leq 1 + 2 = 3$. (contributes 2 units of potential)
  - With expansion: $a_j = c_j + (\varphi_j - \varphi_{j-1}) = 1 + m + (-m) = 1$. (draws m units of potential to pay for expansion)
- Amortized cost of *pop*:
  - Without contraction: $a_j = c_j + (\varphi_j - \varphi_{j-1}) \leq 1 + 1 = 2$. (contributes 1 unit of potential)
  - With contraction: $a_j = c_j + (\varphi_j - \varphi_{j-1}) = 1 + m/4 + (-m/4) = 1$. (draws m/4 units of potential to pay for contraction).

Amortized Running Times as Upper Bounds

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

- Over a sequence of k operations:

$$\Sigma_j a_j = \Sigma_j (c_j + \varphi_j - \varphi_{j-1}) = (\Sigma_j c_j) + \overbrace{\varphi_k}^{\geq 0} - \cancelto{0}{\varphi_0} \geq \Sigma_j c_j$$

- 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!)

37

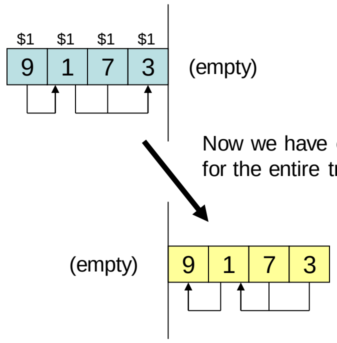
Designing and Using Potential Functions

- 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.

38

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