

CSCI 104

Amortized Analysis

CSCI 104 Teaching Team

A different form of runtime analysis

- Recall that a vector (from the STL) is implemented using an array.
- What is the **worst-case** runtime for the pushback function?
 - Is it $O(1)$?
 - If the array is full, we'll need to double the size of the array, which takes $\Theta(n)$ time!
 - It is correct to say that pushback takes worst-case $\Theta(n)$ runtime.
 - But, this analysis seems rather unfair, given that the worst-case will happen rarely, and at **predictable** intervals.

Example

- You love going to Disneyland. You purchase an annual pass for \$240. You visit Disneyland once a month for a year. Each time you go you spend \$20 on food, etc.
 - What is the cost of a visit?
- Your annual pass cost is spread or "**amortized**" (or averaged) over the duration of its usefulness
- Often, an operation on a data structure will have similar "irregular" costs (i.e. if we can prove the worst case can't happen each call) that we can then amortize over future calls

Amortized Runtime

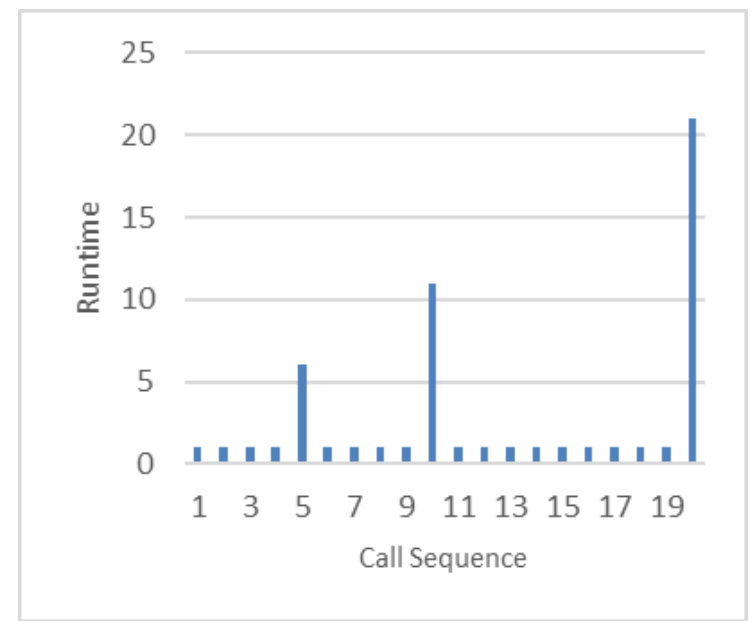
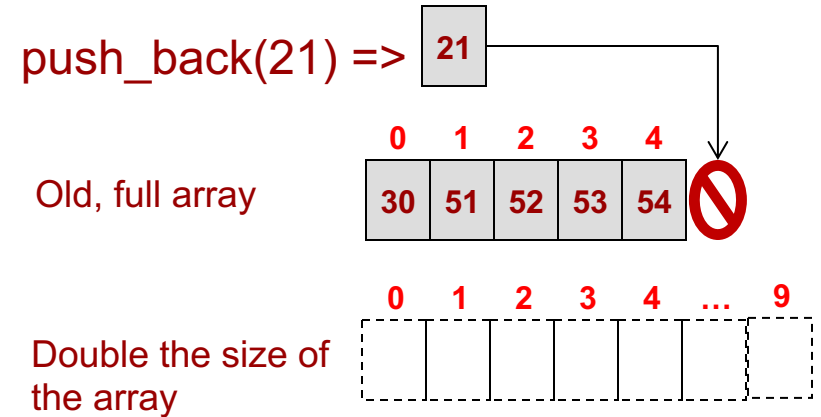
- We could accurately say that the **average** runtime for pushback is $O(1)$.
 - This still doesn't capture everything: that implies that if we get bad luck, the average will be worse than $O(1)$ [like a hash table]
 - There is no luck involved: we know exactly how many inputs will be required to produce the worst-case scenario, and it will always be the same effect.
- Amortized Runtime is a blend between average-case and worst-case. It is kind of the “**worst-case average-case**”.
 - Use when it is provable that the worst-case runtime CAN'T happen on each call

To use **amortized analysis**, usually some **state** must be maintained from one call to the next and that state will determine when the worst case happens.

A LOOK BACK: AMORTIZED RUNTIME WITH VECTORS

Amortized Run-time

- Used when it is impossible for the worst case of an operation to happen on each call (i.e. we can prove after paying a high cost that we will not have to pay that cost again for some number of future operations)
 - Example: Resizing a vector
- We will see 3 methods of performing amortized analysis

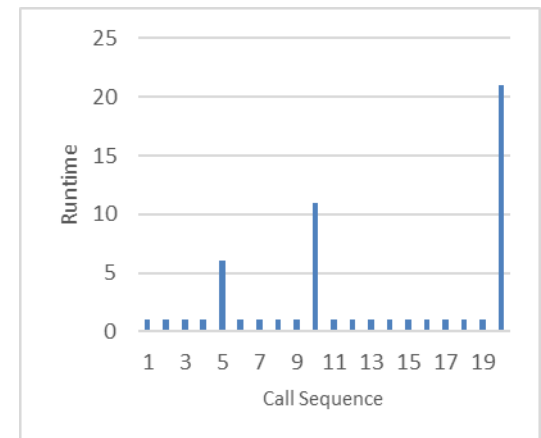


Amortized Runtime

- **Method 1:** Analyze all k operations
- If in the **worst case**, the first k operations take a total (sum) of $\Theta(m)$ time, then the average time per operation is $\frac{1}{k} \sum_k \theta(\text{Operation}_k) = \theta\left(\frac{m}{k}\right)$.
 - The amortized runtime chooses the number and sequence of operations that produces the worst-possible average runtime.
 - It is like the “worst-case average-case”.
 - Assume that the array starts at size 1, and you do n inserts. What is the amortized runtime for pushback?

Pushback analysis, method 1

- Suppose we start at size 1 and double the array size when it becomes full
- There will be a few expensive pushbacks when we have to resize the array.
- When we have to resize and the array is of size, i , how costly is the pushback?
 - _____, where i is the current size of the array.
- If we started with an array size of 1, what values of i would cause us to resize: _____
- How many expensive pushbacks will there be over n pushbacks?
 - _____

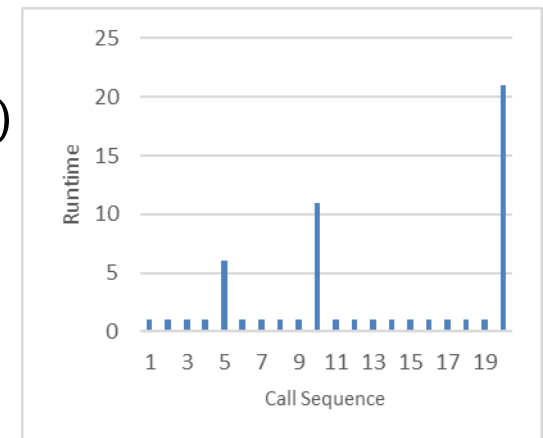


Pushback analysis, method 1

- Suppose we start at size 1 and double the array size when it becomes full
- There will be a few expensive pushbacks, when we have to resize the array.
- When we have to resize and the array is of size, i , how costly is the pushback?
 - $\Theta(i)$, where i is the current size of the array.
- If we started with an array size of 1, what values of i would cause us to resize: 1, 2, 4, 8, 16, ...
- How many expensive pushbacks will there be over n pushbacks?
 - $\log n$
- The total runtime/cost is:

$$\begin{array}{c}
 \text{Pushbacks} \\
 \text{w/ resize.}
 \end{array}
 \left(\sum_{i=1}^{\log n} 2^i \right) + \begin{array}{c} \text{Pushbacks} \\ \text{w/o resize.} \end{array} (n - \log n) = \Theta(n)$$

$$\sum_{i=0}^n c^i = \frac{c^{n+1}-1}{c-1} = \theta(c^n)$$



- The average is then $\frac{\Theta(n) \text{ total cost}}{n \text{ calls}} = \Theta(1)$
- So, the average time per operation is $O(1)$.
Guaranteed!

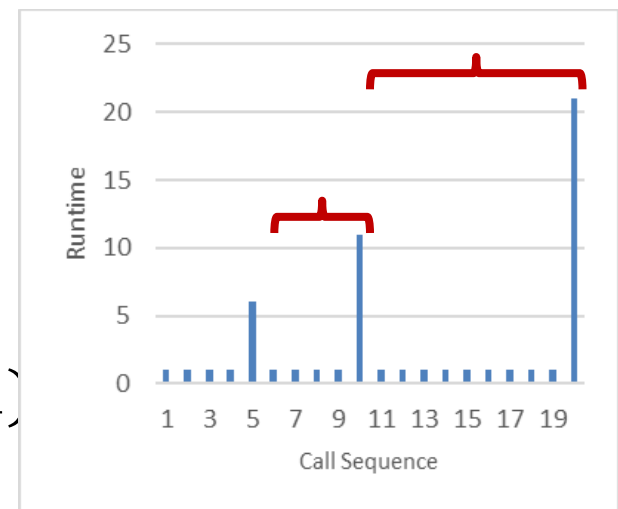
Pushback analysis, Method 2



- **Method 2:** Analyze one "period/phase"
 - Let a new "phase" start just after the array has resized from $n/2$ to n .
- Analyze the amortized runtime for an arbitrary phase:
 - The array has just grown to size n , because we inserted $n/2 + 1$ things leaving $n/2 - 1$ free locations.
 - So we can insert $n/2 - 1$ more things in $\Theta(1)$ time.
 - On the next push back, we have to copy all n items to a new array (of size $2n$), which takes $\Theta(n)$ time.

• Amortized runtime =

$$\frac{\overbrace{(n/2 - 1) \cdot 1}^{\text{Pushbacks w/o resize.}} + \overbrace{1 \cdot n}^{\text{1 Pushback w/ resize.}}}{n/2} = \Theta(1)$$



Pushback analysis, Method 3

- **Method 3: Credit/Debit (Piggy Bank Method)**
- Again, let a new "phase" start just after the array has resized from $n/2$ to n .
- Every time we call pushback, we pay **3 dollars**.
 - Cheap operations only truly cost 1 dollar (to write the new value), so each of the cheap operations saves us a net of \$2 which we place in a piggy bank.
 - When we get to an expensive operation, the last $n/2$ cheap operations have each paid 2 extra dollars for a total of n dollars saved up
 - We need to copy over the n elements to a new larger array, so we have one dollar for each item we need to copy.
 - We always have enough money saved up!
 - 3 dollars per pushback = $\Theta(1)$, so the amortized runtime is constant.



Practice

- Let an integer, n , be represented as a Boolean array (requiring $\log(n)$ bits). You are given an **increment()** function
- What is the cost of incrementing the binary value?
- Each call to increment must visit the bits from right to left until we flip a bit from 0 to 1
- The runtime depends on how many bits we must visit
 - Some increments (from 1010 to 1011, for example) require only constant time.
 - Other increments (from 01111111 to 10000000) take a longer time.
- What is the worst-case runtime of our increment function?
 - $\Theta(\log n)$, all bits may need to flip in the worst case

n	
Bin.	Dec.
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15
0000	0

Amortized analysis of the Binary Increment

- Starting at the least significant (rightmost) bit
 - If the current bit is a 0, we flip it and **stop!**
 - Otherwise, we flip the 1 to a 0 and continue to the next bit and **repeat**.
- Costs:
 - Define our "cost" as 1 unit for each bit we flip (i.e. every bit takes a single dollar to flip, from either 0 to 1 or 1 to 0)
 - We will always flip a single 0 to a 1.
 - We will flip a variable number of 1s to 0s.
- We will use the piggy bank method (method 3) to solve this.

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Practice

- Recall: As stated, each time we call increment a single bit will flip from 0 to 1
- Each time we call the increment function, we will pay a constant \$2
 - \$1 for the bit that will flip from 0 to 1, and
 - \$1 more in advance for when that bit eventually flips back to 0)
 - All of the bits start at 0.
 - Whenever we flip a bit from 0 to 1, we give both of our 2 dollars towards that bit. 1 dollar to cover the immediate costs, and the other dollar to be stored for when it eventually flips from 1 to 0.
 - A bit cannot flip from 1 to 0 if hadn't first flipped 0 to 1...so we'll never be in debt.
 - Since $\$2 = \Theta(1)$, this takes amortized constant time!

Cost Balance

0000	-	0
0001	1	1
0010	2	1
0011	1	2
0100	3	1
0101	1	2
0110	2	2
0111	1	3
1000	4	1
1001	1	2
1010	2	2
1011	1	3
1100	3	2
1101	1	3
1110	2	3
1111	1	4
0000	4	2

An Alternate Approach – Expected Value

- We might also let X be a random variable defined to be *the number of bits that flip (i.e. cost of) on a call to increment* and compute $E[X]$

(recall $E[X] = \sum_x x \cdot p(X = x)$)

- $X=1, P(X=1) = 1/2$ or All calls cost ≥ 1
- $X=2, P(X=2) = 1/4$ or $1/2$ calls cost ≥ 2 (at least 1 more)
- $X=3, P(X=3) = 1/8$ or $1/4$ calls cost ≥ 3 (at least 1 more)
- ...

– $E[X] = 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \dots + \text{Last term} \leq 2$

– **or**

– $E[X] = 1 \cdot 1 + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} + \dots \leq \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2$

0000	-
0001	1
0010	2
0011	1
0100	3
0101	1
0110	2
0111	1
1000	4
1001	1
1010	2
1011	1
1100	3
1101	1
1110	2
1111	1
0000	4