CSCI 104
Amortized Analysis

Aaron Cote
Mark Redekopp
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

```
push_back(21) =>
```

Old, full array

Double the size of the array

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
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 $\frac{1}{k} \sum k \ \theta(Operation_k) = \theta\left(\frac{m}{k}\right)$.
  - 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?
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:

\[
\sum_{i=1}^{\log n} 2^i + (n - \log n) = \Theta(n)
\]

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

\[\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c-1} = \theta(c^n)\]
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 $\frac{n}{2} + 1$ things leaving $\frac{n}{2} - 1$ free locations.
- So we can insert $\frac{n}{2} - 1$ 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 =

$$\frac{(\frac{n}{2} - 1) \cdot 1 + 1 \cdot n}{\frac{n}{2}} = \Theta(1)$$
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 $\frac{n}{2}$ 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 0111111 to 10000000) take a longer time.

What is the worst-case runtime of our increment function?

- $O(\log n)$, all bits may need to flip in the worst case.
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.
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!

<table>
<thead>
<tr>
<th>Cost</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>3</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>2</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>2</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>3</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>2</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>4</td>
</tr>
</tbody>
</table>
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)
- ... 

- \( E[X] = 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \cdots + \text{Last term} \leq 2 \)
- *or*

- \( E[X] = 1 \cdot 1 + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} + \cdots \leq \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2 \)