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.
  – This analysis seems rather unfair, given that the worst-case will happen rarely, and at predictable intervals.
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 $O(1)$.
  – 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”.
A LOOK BACK: AMORTIZED RUNTIME WITH VECTORS
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 times an operation on a data structure will have similar "irregular" (i.e. if we can prove the worst case can't happen each call) costs that we can then amortize over future calls
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)

- Amortized Runtime = (Total runtime over k calls) / k
  - Average runtime over k calls
  - Use a "period" of calls from when the large cost is incurred until the next time the large cost will be incurred
Amortized Runtime

• If the first $x$ operations take a total of $\Theta(y)$ time, then the average time per operation is $\Theta(y/x)$.
  – 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

• There will be a few expensive pushbacks, when we have to resize the array.
• How costly is an expensive pushback?
  – $\Theta(i)$, where $i$ is the current size of the array.
• How many expensive pushbacks will there be?
  – $\log n$
• The total runtime is $\sum_{i=1}^{\log n} 2^i + (n - \log n) = \Theta(n)$
• So the average time per operation is $O(1)$. Guaranteed!
Pushback analysis, Method 2

• Let a new “phase” start just after the array has resized.

• Analyze the amortized runtime for an arbitrary phase:
  – The array has just grown to size $n$, because we inserted $1 + \frac{n}{2}$ things.
  – We insert $\frac{n}{2}$ things this phase, all but one of them take $O(1)$ time.
  – The last thing takes $\Theta(n)$ time.

• Amortized runtime $= \frac{1 \cdot \frac{n}{2} + (\frac{n}{2} - 1) \cdot 1}{\frac{n}{2}} = \Theta(1)$
Amortized Array Resize Run-time

- What if we resize by adding 5 new locations each time
- Start analyzing when the list is full...
  - 1 call to insert will cost: \(n+1\)
  - What can I guarantee about the next 4 calls to insert?
    - They will cost 1 each because I have room
  - After those 4 calls the next insert will cost: \((n+5)\)
  - Then 4 more at cost=1
- If the list is size \(n\) and full
  - Next insert cost = \(n+1\)
  - 4 inserts after than = 1 each = 4 total
  - Thus total cost for 5 inserts = \(n+5\)
  - Runtime = cost / inserts = \((n+5)/5 = O(n)\)
Pushback analysis, Method 3

• Every time we call pushback, we pay 5 dollars.
  – Cheap operations only require 1 dollar, so we place the excess in a piggy bank.
  – When we get to an expensive operation, the last \( \frac{n}{2} \) things have each paid 4 extra dollars.
  – We need to make an array of size 2n, so we have one dollar for each index we need to make: we always have enough money saved up!
  – 5 = \( \Theta(1) \), so the amortized runtime is constant.
Practice

• We are using a Boolean array as a binary counter.
  – Each index starts at 0 (false), and the counter counts up in binary.
  – Some increments (from 1010 to 1011, for example) require only constant time.
  – Other increments (from 01111111 to 10000000) take a long time.
  – What is the worst-case runtime of our increment function?
  – $\Theta(\log n)$, since if we insert $n$ times, we require $\log n$ bits.
Amortized analysis of the Binary counter

• Starting at the least significant 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.
  – 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.
• When we call the increment function, we pay 2 dollars. Every bit takes a single dollar to flip, from either 0 to 1 or 1 to 0.

• All of the bits start at 0.
  – Whenever we flip a bit from 0 to 1, we spend 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.
  – Since only a single bit flips from 0 to 1 every increment, we always have enough money saved up for the 1s that flip to 0s.
  – Since $2 = \Theta(1)$, this takes amortized constant time!