CSE 332: Data Abstractions

Lecture 24: Amortized Analysis

Ruth Anderson

Winter 2015

Amortized Analysis

- Recall our plain-old stack implemented as an array that doubles its size if it runs out of room
  - How can we claim **`push`** is *O*(**1**)?
    - Sure, most calls will take O(1), but *some* (the ones when we have to resize to stack) will take O(n)
  - We *can't* claim O(1) as guaranteed worst-case run-time, but we *can* claim it's an *O*(**1**) ***amortized bound***
    - (Very) rough idea: Resizing won't happen 'very often'
  - **Amortized bounds** are *not* a handy-wave concept though
    - It's a provable bound for the running time, not necessarily for every operation, but over some sequence of operations
    - Don't use amortized to mean 'we don't really expect it to happen much'

Amortized Analysis

- This lecture:
  - What does amortized mean?
  - When are amortized bounds good enough?
  - How can we prove an amortized bound?

- Will do a couple simple examples
  - The text has more complicated examples and proof techniques
  - The *idea* of how amortized describes average cost is essential

Amortized Complexity

If a sequence of **`M`** operations takes $O($**`M f(n)`**$)$ time,
we say the amortized runtime is $O($**`f(n)`**$)$

Amortized bound: **worst-case** guarantee over **sequences** of operations

- Example: If any **`n`** operations take $O($**`n`**$)$, then amortized $O($**`1`**$)$
- Example: If any **`n`** operations take $O($**`n`**$^3)$, then amortized $O($**`n`**$^2)$

- The **worst case** time *per operation* can be larger than **`f(n)`**
  - For example, maybe **`f(n)=1`**, but the worst-case is **`n`**
- But the worst-case for *any* sequence of **`M`** operations is $O($**`M f(n)`**$)$

Amortized guarantee ensures the average time per operation for any sequence is $O($**`f(n)`**$)$

Amortized Complexity

If a sequence of `M` operations takes *O*(`M f(n)`) time,
we say the amortized runtime is *O*(`f(n)`)

- Another way to think about it: If *every possible sequence* of M operations runs in O(M*f(n)) time, we have an amortized bound of O(f(n))
- A good way to convince yourself that an amortized bound does or does not hold: Try to come up with a worst-possible sequence of operations
  - Ex: Do BST inserts have an amortized bound of O(logM)?
  - We can come up with a sequence of M inserts that takes $O(M^2)$ time (M in-order inserts); so, no.

Amortized != Average Case

- The ‘average case’ is generally **probabilistic**
  - What data/series of operations are we *most likely* to see?
  - Ex: Average case for insertion in BST is O(logn); worst case O(n)
  - For ‘expected’ data, operations take about O(logn) each
  - However, we **could** come up with a huge # of operations performed in a row that **each** have O(n) time
  - Thus O(logn) is *not* an amortized bound for BST operations
- Amortized bounds are **not probabilistic**
  - It’s not ‘we expect …’; it’s ‘we are guaranteed ...’
  - If the amortized bound is O(logn), then there ***does not exist*** a long series of operations whose average cost is greater than O(logn)

Amortized != Average Case Example

- Consider a hashtable using separate chaining
- We generally expect O(1) time lookups; why not O(1) amortized?
  - Imagine a worst-case where all n elements are in one bucket
  - Lookup one will likely take n/2 time
  - Because we can concoct this worst-case scenario, it can't be an amortized bound
- But the expected case is O(1) because of the expected distribution keys to different buckets (given a decent hash function, prime table size, etc.)

"Building Up Credit"

- Can think of preceding "cheap" operations as building up "credit" that can be used to "pay for" later "expensive" operations

- Because any sequence of operations must be under the bound, enough "cheap" operations must come *first*
  - Else a prefix of the sequence would violate the bound

Example #1: Resizing stack

From lecture 1: A stack implemented with an array where we double the size of the array if it becomes full

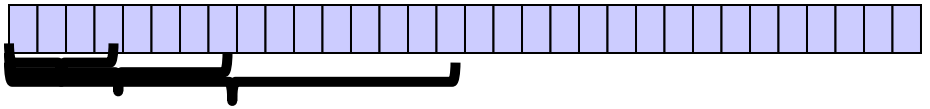
Claim: Any sequence of **`push`**/**`pop`**/**`isEmpty`** is amortized $O(1)$

Need to show any sequence of **`M`** operations takes time $O(M)$

- Recall the non-resizing work is $O(M)$ (i.e., $M*O(1)$)
- Need to show that resizing work is also $O(M)$ (or less)
- The resizing work is proportional to the total number of element copies we do for the resizing
- So it suffices to show that:

  After **`M`** operations, we have done **`< 2M`** total element copies

  (So number of copies per operation is bounded by a **constant**)



Amount of copying

Example: How much copying gets done?

- Take M=100: Say we start with an empty array of length 32 and finish with 100 elements
- Will resize to 64, then resize to 128
- Each resize has half that many copies (32 the first time, 64 the second)
- In this case, 96 total element copies; 96< 2M

Show: after **M** operations: Total element copies **< 2M**

Let **n** be the size of the array after **M** operations

- Every time we're too full to insert, we resize via doubling:

Total element copies = **INITIAL_SIZE + 2*INITIAL_SIZE + … + n/8 + n/4 + n/2 < n**

= n * (1/2 + 1/4 + 1/8 + ..)

= n * (< 1 [Sum of Geometric Series - 1, see p. 4 Weiss])

< n

- To get it to resize to size n, we need at least half that many pushes:

n/2 ≤ M

n ≤ 2M

- So:

Total number of element copies < **n ≤ 2M**

Why doubling? Other approaches

- If array grows by a constant amount (say 1000), operations are not amortized $O(\mathbf{1})$
  - Every 1000 inserts, we do additional O(n) work copying
  - So work per insert is roughly O(1)+O(n/1000) = O(n)
  - After $O(\mathbf{M})$ operations, you may have done $\Theta(\mathbf{M}^2)$ copies

- If array shrinks when 1/2 empty, operations are not amortized $O(\mathbf{1})$
  - Terrible case: **`pop`** once and shrink, **`push`** once and grow, **`pop`** once and shrink, …

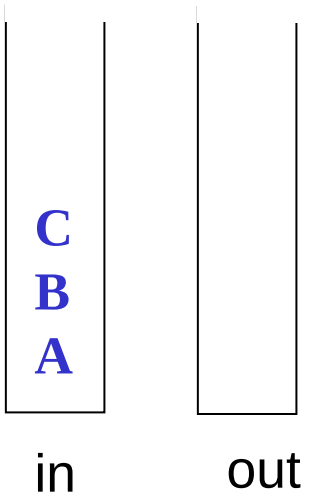
- If array shrinks when 3/4 empty, it is amortized $O(\mathbf{1})$
  - Proof is more complicated, but basic idea remains: by the time an expensive operation occurs, many cheap ones occurred

Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

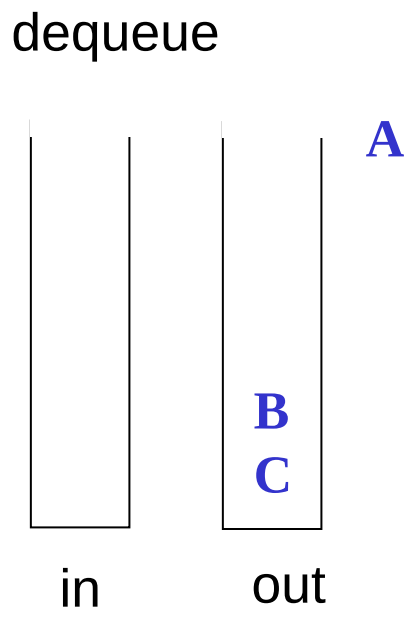
enqueue: A, B, C



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

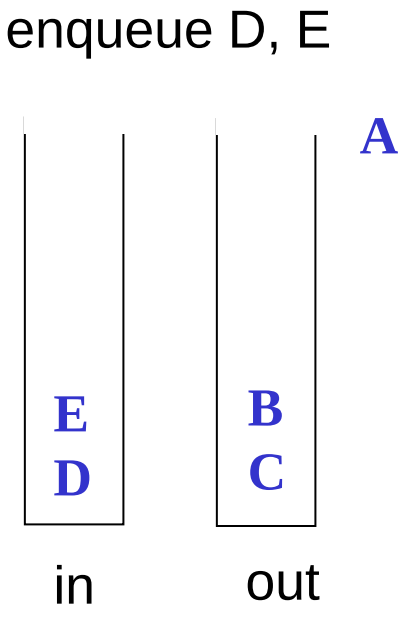
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

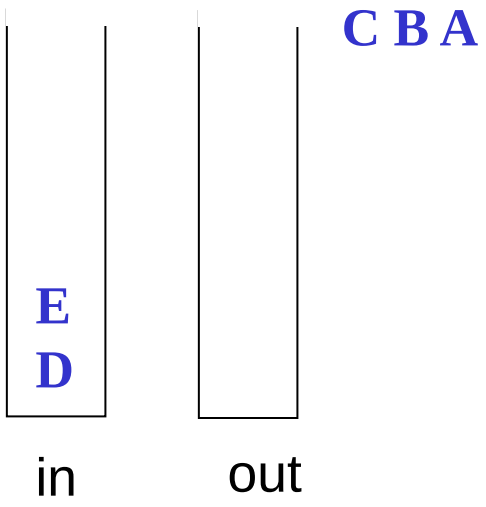


Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

dequeue twice

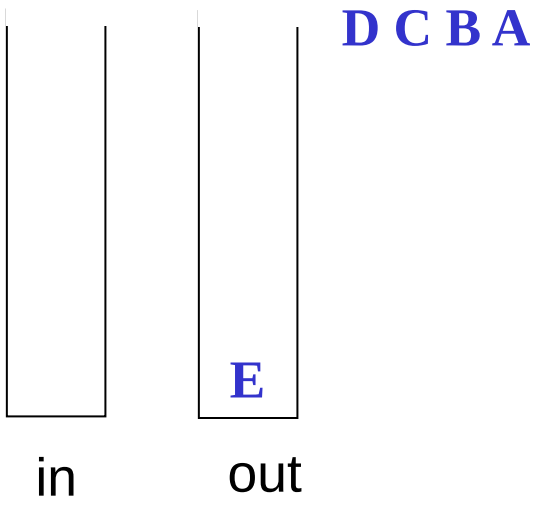


Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

dequeue again



Correctness and usefulness

- If **`x`** is enqueued before **`y`**, then **`x`** will be popped from **in** later than **`y`** and therefore popped from **out** sooner than **`y`**
  - So it is a queue

- Example:
  - Wouldn't it be nice to have a queue of t-shirts to wear instead of a stack (like in your dresser)?
  - So have two stacks
    - *in*: stack of t-shirts go after you wash them
    - *out*: stack of t-shirts to wear
    - if *out* is empty, reverse *in* into *out*

Queue Analysis

- **`dequeue`** is not *O*(**`1`**) worst-case because **out** might be empty and **in** may have lots of items that need to be copied over (taking O(n) time)
- But if the stack operations are (amortized) *O*(**`1`**), then any sequence of queue operations is amortized *O*(**`1`**)
  - How much total work is done *per element?*
    - 1 **`push`** onto **in**
    - 1 **`pop`** off of **in**
    - 1 **`push`** onto **out**
    - 1 **`pop`** off of **out**
  - So the total work should be 4n; just shifted around
  - When you reverse **`n`** elements by popping them off **in** and pushing them onto **out**, there were **`n`** earlier *O*(**`1`**) **`enqueue`** operations you got 'for cheap'

*Amortized bounds are **not** limited to array-based data structures*

- **Splay Tree**: Another balanced BST data structure
  - Comparable in many ways to AVL tree
  - Like BST & AVL trees, operations are a function of the height
  - Height for splay tree can be O(n)
  - Nonetheless, we have O(logn) amortized bound on operations
  - A single operation may need to work through a O(n) height tree
    - *But* splay tree operations (even lookups) shift around the tree – a worst-case tree of height O(n) is guaranteed to be 'fixed' over the course of sufficiently many operations

Amortized useful?

- When the average per operation is all we care about (i.e., sum over all operations), amortized is perfectly fine
  - This is the usual situation
- If we need **every** operation to finish quickly, amortized bounds are too weak
  - In a concurrent setting
  - Time-critical real-time applications
- While amortized analysis is about averages, we are averaging cost-per-operation on worst-case input
  - Contrast: Average-case analysis is about averages across all possible inputs. Example: if all initial permutations of an array are equally likely, then quicksort is $O(\mathbf{n \log n})$ on average even though on some inputs it is $O(\mathbf{n^2})$)

Not always so simple

- Proofs for amortized bounds can be much more complicated

- For more complicated examples, the proofs need much more sophisticated invariants and “potential functions” to describe how earlier cheap operations build up “energy” or “money” to “pay for” later expensive operations
  - See Chapter 11

- But complicated *proofs* have nothing to do with the code! (The code might not be that complicated.)