CSE373: Data Structures & Algorithms
Lecture Supplement: Amortized Analysis

Kevin Quinn

Fall 2015

Amortized

- 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 $O(\mathtt{1})$ time if resizing is $O(\mathtt{n})$ time?
  - We *can't*, but we *can* claim it's an $O(\mathtt{1})$ amortized operation

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

Will just do two simple examples

- Text has more sophisticated examples and proof techniques
- *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*($\mathbf{n}^3$), then amortized *O*($\mathbf{n}^2$)

- The worst case time per operation can be larger than **`f(n)`**
  - As long as the worst case is *always* "rare enough" in *any* sequence of operations

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

“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, which is also a sequence, would violate the bound

Example #1: Resizing stack

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`**)

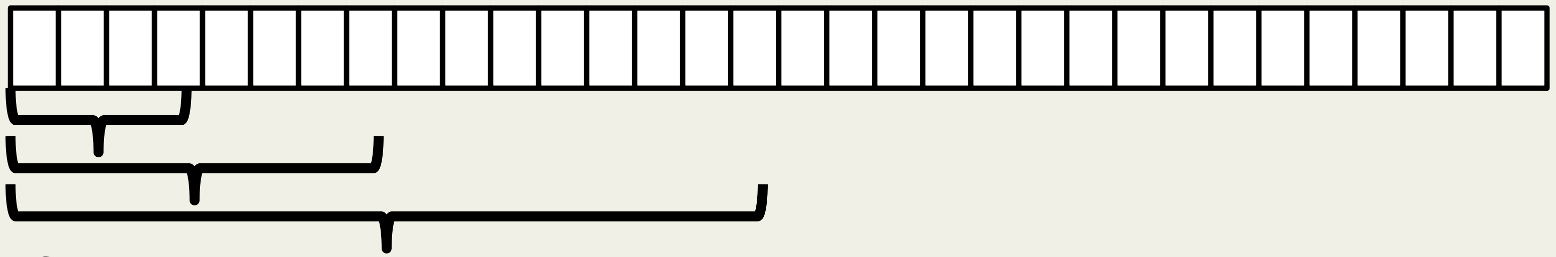
Need to show any sequence of **`M`** operations takes time *O*(**`M`**)

- Recall the non-resizing work is *O*(**`M`**) (i.e., **`M*`***O*(**`1`**))
- 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 average number of copies per operation is bounded by a constant)

Amount of copying



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

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

- Then we have done a total of:

  `n/2 + n/4 + n/8 + … INITIAL_SIZE < n`

  element copies
- Because we must have done at least enough **`push`** operations to cause resizing up to size **`n`**:

  $M \geq n/2$
- So

  $2M \geq n >$ *number of element copies*

Other approaches

- If array grows by a constant amount (say 1000), operations are not amortized *O*(`1`)
  - After *O*(`M`) operations, you may have done Θ(`M`^{2}) copies

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

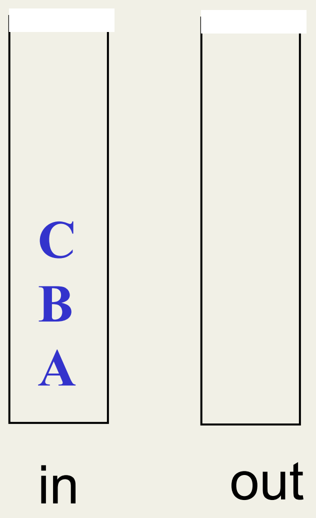
- If array shrinks when 3/4 empty, it is amortized *O*(`1`)
  - Proof is more complicated, but basic idea remains: by the time an expensive operation occurs, many cheap ones occurred

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();
  }
}
```

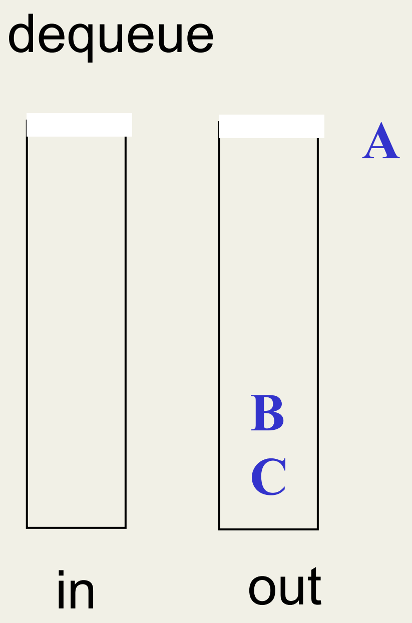
enqueue: A, B, C



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();
  }
}
```

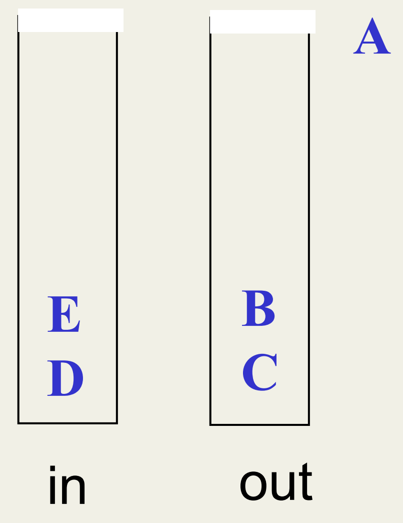


Example #2: Queue with two stacks

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();
  }
}
```

enqueue D, E

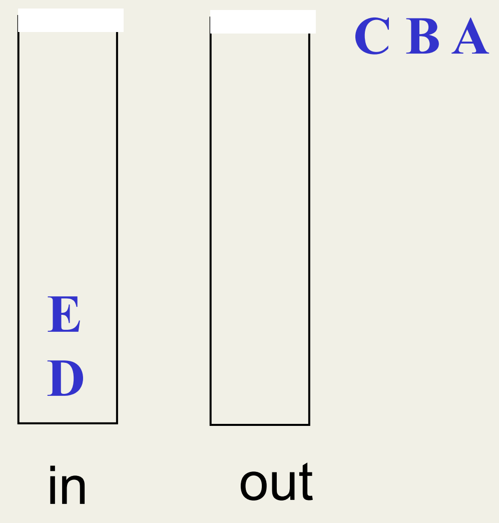


Example #2: Queue with two stacks

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

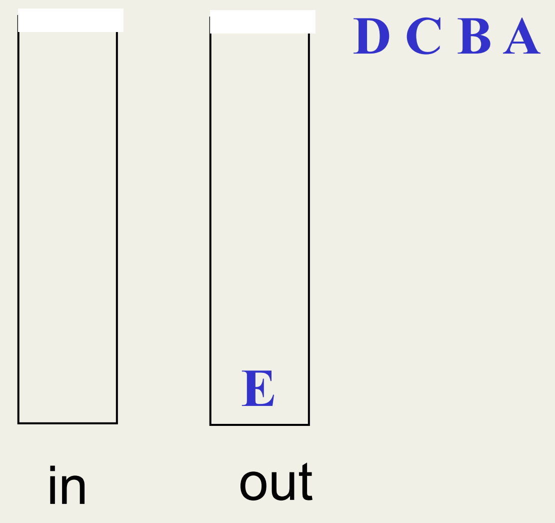


Example #2: Queue with two stacks

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*

Analysis

- **`dequeue`** is not *O*(**`1`**) worst-case because **out** might be empty and **in** may have lots of items

- But if the stack operations are (amortized) *O*(**`1`**), then any sequence of queue operations is amortized *O*(**`1`**)
  - The total amount of work done per element is 1 **`push`** onto **in**, 1 **`pop`** off of **in**, 1 **`push`** onto **out**, 1 **`pop`** off of **out**
  - When you reverse **`n`** elements, there were **`n`** earlier *O*(**`1`**) **`enqueue`** operations to average with

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 (e.g., in a web server), amortized bounds may be too weak

- While amortized analysis is about averages, we are averaging cost-per-operation on worst-case input
  - **Contrast**: Average-case analysis is about averages across 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
- Example: Splay trees are dictionaries with amortized *O*(`log n`) operations
  - No extra height field like AVL trees
  - See Chapter 4.5 if curious
- 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 if curious
- But complicated *proofs* have nothing to do with the code!